Глава 8. Дополнительные возможности Ассемблера

1. *Да не будешь ты смешивать в одной команде операнды разной длины.*
2. *Да не будешь ты путать команды для знаковых и беззнаковых чисел.*
3. *Да не будешь ты использовать в одной команде два явных операнда из памяти.*
4. *Да будет у тебя каждая команда знать длину своих операндов.*
5. *Да будешь ты различать адрес ячейки памяти и её содержимое.*

Заповеди программиста
на Ассемблере

8.1. Строковые команды

Строка – это застывшая структура данных, и повсюду, куда она передается, происходит значительное дублирование процесса.[1]

Алан Перлис

Сейчас будет изучен весьма полезный для понимания архитектуры данного компьютера формат команд память-память (формат `SS`). До сих пор для работы с переменными в оперативной памяти использовался формат команд регистр-память (или память-регистр), при этом только один из аргументов находился в оперативной памяти. Например, для присваивания переменной Y значения переменной X вместо команды пересылки формата память-память

```
mov Y,X;  такого формата нет!
```

приходилось использовать две команды пересылки или запись в стек и чтение из стека, например

```
mov eax,X          push X
mov Y,eax          pop  Y
```

Это не всегда практично, если не предполагается больше никаких операций с выбранным на регистр операндом, тогда его пересылка из памяти на регистр оказывается лишним действием.[2] При использовании же стека нельзя работать с байтовыми переменными. Именно для таких случаев и предназначены команды формата память-память, в некоторых случаях они позволяют получать более компактные и быстрые программы. В архитектуре компьютера такие команды относятся к так называемым строковым или цепочечным командам (скоро Вы поймете, почему они получили такие названия).

Строковые команды нашего компьютера хорошо использовать для обработки элементов массивов. Массивы коротких беззнаковых целых чисел могут трактоваться как строки (цепочки) символов, отсюда и название – *строковые* или *цепочечные* команды. Можно считать, что каждая строковая команда обрабатывает *один* элемент такого массива. При выполнении строковых команд в цикле получается удобный способ обработки массивов, элементами которых являются целые числа длиной в 1, 2 или 4 байта.

Знакомство со строковыми командами начнём с команд пересылки байта (**`movsb`**), слова (**`movsw`**) или двойного слова (**`movsd`**) с одного места оперативной памяти в другое (в старших моделях добавляется команда **`movsq`** для пересылки восьми байт). Эти команды различаются только битом размера операнда `w` в коде операции. С битом `w` Вы уже познакомились при изучении форматов

---

[1] Для продвинутых читателей рекомендуется изучить, как приходится обрабатывать параметры-строки в консольных макрокомандах из файла `io.inc`.

[2] Точнее, лишним будет только использование *адресуемого* регистра центрального процессора, их в распоряжении программиста и так немного (`EAX`, `EBX` и т.д.). Однако, двухадресные команды формата регистр-память `КОП r1, op2` обязательно требуют для своего выполнения использования *служебного* регистра центрального процессора (напомним, что в учебных машинах эти регистры обозначались как `R1`, `R2` и `S`), и выполняются, например, для команды сложения по схеме: `R2:=op2; S:=r1+R2; r1:=S`.

команд регистр-регистр и регистр-память, w=0 для операндов-байтов и w=1 для операндов другой длины. Как уже говорилось ранее, для w=1 выбор длины операнда (16, 32 или 64 бита) определяется режимом работы процессора. Команды **movsb**, **movsw** и **movsd** не имеют *явных* операндов, оба их операнда op1 и op2 формата m8, m16 или m32 заданы неявно (по умолчанию).

Алгоритм выполнения этих команд существенно зависит от так называемого флага направления DF (Direction Flag) из регистра флагов EFLAGS. Для изменения значения этого флага можно использовать команды **cld** (CLear Direction, DF:=0), и **std** (SeT Direction, DF:=1). Чтобы описания правил выполнения команд **movs** были более компактными и строгими, введём следующие условные обозначения:

$\varepsilon = w*(-1)^{DF}$; $\varphi(r32)=\{ r32 := (r32 + \varepsilon) \bmod 2^{32}\}$

Здесь w определяется длиной операнда в байтах (1, 2 или 4). Как видим, **ε** может принимать только значения ±1, ±2 и ±4, а оператор **φ** меняет величину заданного регистра r32 на значение **ε**. В этих обозначениях команды **movsb**, **movsw** и **movsd** выполняются по правилу:

[edi] := [esi]; **φ**(edi); **φ**(esi)

Таким образом, неявный операнд op1 находится в памяти по адресу, заданному на регистре EDI, а неявный операнд op2 – на регистре ESI.[1] Индексный регистр EDI называется индексом получателем (Destination Index), а регистр ESI индексом источником (Source Index).

Для того, чтобы лучше понять логику работы описанной выше команды, рассмотрим широко распространённую задачу пересылки массива символов с одного места оперативной памяти в другое. На языке Паскаль такая задача решается совсем просто, например:

```
Const N=50000; Var A,B: array[1..N] of char;
        . . .
      B := A;
```

А теперь рассмотрим реализацию похожей задачи пересылки массива из одного места памяти в другое на языке Ассемблер. Сначала опишем наши массивы A и B:

```
N  equ 50000
.data
A  db  N dup (?)
B  db  N dup (?)
```

Длину массива обозначим N, в частном случае длина массива может быть и нулевой (вспомним, какую важную роль в программировании играют *пустые* строки символов). Теперь фрагмент программы для реализации оператора присваивания B:=A может, например, иметь на Ассемблере такой вид:

```
    mov   esi,offset A
    mov   edi,offset B
    mov   ecx,N
    jecxz KOH; при N=0
L:  mov   al,[esi]
    mov   [edi],al
    inc   esi
    inc   edi
    loop  L
KOH:
```

Оценим вычислительную сложность этого алгоритма пересылки массива. За единицу измерения сложности примем операции обмена данными и командами между процессором и оперативной памятью. В нашем случае сложность алгоритма пересылки массива равна 7*N, где N – это длина массива. Действительно, необходимо N чтений элементов массива из памяти на регистр AL, N записей из этого регистра в память, и ещё нужно 5*N раз считать команды тела цикла из памяти на регистр команд в устройстве управления.

Как видим, для пересылки целого числа из одного места памяти в другое нам понадобились две команды

```
mov al,[esi]
```

---

[1] Вообще говоря, процессор также считает, что операнд op1 находится в сегменте ES, а операнд op2 в сегменте DS, но в плоской модели памяти эти сегменты полностью совпадают.

```
mov [edi],al
```
так как написать одну команду
```
mov byte ptr[esi],[edi]
```
нельзя – она требует несуществующего формата команды пересылки **mov** m8,m8. Здесь, однако, хорошо подходит новая команда пересылки короткого целого числа **movsb**, с её помощью заключённый в рамку фрагмент предыдущей программы можно более компактно записать в виде:

```
      jecxz KOH; при N=0
      cld
   L: movsb
      loop  L
```

Теперь в цикле пересылки массива осталось всего две команды, следовательно, сложность алгоритма снизилась до 4*N операций обмена с оперативной памятью. Для дальнейшего ускорения выполнения таких циклов в язык машины была включена специальная команда цикла с кодом операции **rep** (0F3h), которая называется *префиксом повторения*. Она похожа на команду цикла **loop**, но не имеет явного операнда – метки перехода на начало тела цикла. Эта метка не нужна, так как в теле цикла **rep** может находиться только *одна*, непосредственно следующая за ней команда, другими словами, пара команд

rep <строковая команда>

выполняется по схеме

```
while ecx<>0 do begin
      dec(ecx); <строковая команда>
end;
```

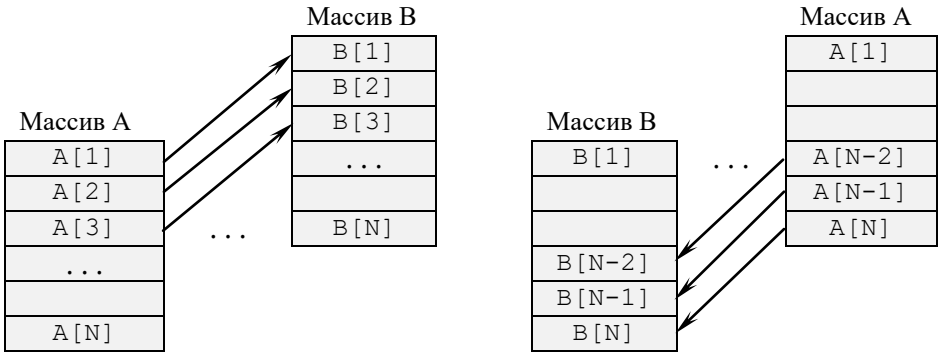
Заметим, что команда цикла **rep**, в отличие от команды **loop**, задаёт цикл *с предусловием*, поэтому команда **jecxz** L1 уже не нужна. С использованием этой новой команды цикла заключенный в рамку фрагмент нашей программы пересылки массива можно записать уже совсем кратко в виде:

```
    cld
rep movsb
```

Если за командой **rep** следует не строковая команда, а какая-нибудь другая, то команда **rep** просто игнорируется, т.е. выполняется так же, как и пустая команда с кодом операции **nop** (No OPeration) которая эквивалентна пустому оператору языка Паскаль.[i] [см. сноску в конце главы] Заметим, что хотя **rep** и является служебным словом (кодом операции), но в программах его часто пишут на месте метки (в качестве первого поля предложения Ассемблера), так как служебное слово нельзя спутать с именем метки, заданным пользователем. Пара команд **rep** **movsb** может быть *вместе* выбрана на регистр команд процессора, так что теперь в цикле пересылки массива вообще нет необходимости обращаться в память *за командами*. Теперь сложность нашего алгоритма снизилась до теоретического минимума в 2*N операций, т.е. это позволило значительно поднять эффективность пересылки массива.[ii] [см. сноску в конце главы] Теперь Вам должно быть понятно, для чего цепочечные команды введены в язык этого компьютера, хотя они и являются избыточными (пересылку массива можно делать и без использования этих команд в обычном цикле).

Разберёмся теперь с назначением флага направления DF. Отметим сначала, что этот флаг позволяет производить пересылку массива в *прямом* направлении (от первого элемента к последнему) при значении DF=0 и в *обратном* направлении (от последнего элемента массива к его первому элементу) при DF=1, отсюда и название флага – Direction Flag (флаг направления пересылки массива).

Пересылка массива в обратном направлении – не прихоть программиста, а часто единственный способ *правильного* присвоения значения одного массива другому. Правда следует сразу сказать, что флаг направления влияет на правильность присваивания одному массиву значения другого массива только в том случае, если эти массивы *перекрываются* в памяти (т.е. один массив полностью или частично занимает то же место в памяти, что и второй массив). В качестве примера на рис. 8.1 показано два случая перекрытия массивов A и B в памяти при выполнении пересылок. Из этого рисунка видно, что для случая 8.1 а) необходима пересылка в *прямом* направлении с флагом DF=0, а для случая 8.1 б) правильный результат присваивания массивов получается только при *обратном* направлении пересылки элементов массива с флагом DF=1. Обязательно поймите, что пересылка перекрывающихся массивов не в том направлении приведёт к ошибке.



а). Должно быть `DF=0` б). Должно быть `DF=1`

Рис. 8.1. Два случая перекрытия массивов в памяти при пересылке.

Перекрытие массивов при пересылке одного массива в другой часто встречается в практике программирования. Типичным примером является работа текстового редактора, когда при выполнении операций вставки и удаления фрагментов текста приходится раздвигать или сжимать редактируемый текст.[1]

Продолжим изучение строковых команд. Команды сравнения двух операндов **cmpsb**, **cmpsw** и **cmpsd** имеют тот же формат память-память, что и строковые команды пересылки. В старших моделях добавляется команда **cmpsq** для сравнения чисел длиной восемь байт. Команды **cmpsb**, **cmpsw** и **cmpsd** выполняются по схеме:

```
cmp [esi],[edi]; φ(edi); φ(esi)
```

т.е. производится сравнение между собой двух целых чисел длиной 1, 2 или 4 байта, в результате чего соответствующим образом устанавливаются флаги (так же, как при выполнении обычной команды сравнения **cmp**).

Как известно, команды сравнения обычно необходимы для правильной работы команд условного перехода. Со строковыми командами сравнения удобно использовать команды-префиксы повторения **repe/repz** (выполнять цикл, пока сравниваемые аргументы равны, т.е. флаг `ZF=1`) и **repne/repnz** (выполнять цикл, пока сравниваемые аргументы *не* равны, т.е. `ZF=0`). Эти команды похожи на рассмотренную ранее команду **rep**, но обеспечивают возможность *досрочного* выхода из цикла по значению флага `ZF=0` (для команд **repe/repz**) и `ZF=1` (для команд **repne/repnz**).

В качестве примера рассмотрим следующую задачу. Как известно, строки текста во многих языках программирования высокого уровня можно сравнивать между собой не только на равенство и неравенство, но и с помощью других операций отношения (больше, меньше и т.д.). При этом строки считаются упорядоченными в так называемом *лексикографическом* порядке (а по-простому – как в словаре). С помощью строковых команд и префиксов повторения операцию сравнения двух строк можно реализовать на Ассемблере таким образом (правда, здесь строки должны иметь одинаковую длину, сравнение строк разной длины реализуется несколько более сложно):

```
N  equ 256
.data
X  db  N dup (?)
Y  db  N dup (?)
.code
   mov ecx,N
   jecxz EQ; Пустые строки
   cld
   lea esi,X
   lea edi,Y
```

---

[1] Для продвинутых читателей. Оперативная память современных ЭВМ оптимизирована на чтение и запись в прямом направлении (по возрастанию адресов), при чтении и записи в обратном направлении скорость работы может упасть примерно в полтора раза. Можно модифицировать алгоритм копирования массива так, что чтение и запись будет всегда проводится в прямом направлении, попробуйте сами понять, как это сделать, или читайте книгу [Касперски К. Техника оптимизации программ. Эффективное использование памяти. – Санкт-Петербург, 2003, 456 с.].

```
repe cmpsb
    je  EQ; Строки X и Y равны
    jb  LT; Строка X меньше Y
    ja  GT; Срока X больше Y
```

Как видно, основная часть работы – последовательное сравнение в цикле символов двух строк до первых несовпадающих символов или до конца строк – выполняется двумя тесно связанными командами **repe** **cmpsb** (команда **repe** является *префиксом* команды **cmpsb**).

Остальные строковые команды имеют формат регистр-память, но они тоже ориентированы на задачи обработки элементов строк (массивов) целых чисел длиной 1, 2 или 4 байта. Команды *сканирования строки* являются командами *сравнения* и, при использовании в цикле, хорошо подходят для поиска в строке заданного целого значения длиной 1 (**scasb**), 2 (**scasw**) или 4 (**scasd**) байта. В старших моделях добавляется команда **scasq** для сравнения чисел восемь байт. Эти команды отличаются только битом размера аргументов `w`, и имеют два *неявных* операнда `op1` и `op2`, где `op1=AL` для `w=0`, и `op1=AX` или `EAX` для `w=1`, а второй неявный операнд `op2=<EDI>` является целым числом длины 1, 2 или 4 байта в оперативной памяти. Если для краткости обозначить буквой `r` регистр `AL`, `AX` или `EAX`, то схему выполнения команд сканирования строки можно записать так:

```
    cmp r,<edi>; φ(edi)
```

Для иллюстрации использования команды сканирования напишем фрагмент программы для реализации следующей задачи: в массиве длинных целых чисел (**dw**) найти и вывести номер *последнего* нулевого элемента (пусть элементы в массиве нумеруются с единицы). Если в массиве нет нулевых элементов, то будем в качестве ответа выдавать номер ноль, т.к. элемента с таким номером в нашем массиве нет.

```
N     equ  200000
.data
   X  dw  N dup (?)
.code
      . . .
    mov ecx,N;            число элементов
    sub ax,ax;            образец для поиска=0
    lea esi,X[2*ecx-2]; адрес последнего элемента
    std;                  обратный просмотр эдементов
repne scasw
    jnz NoZero;           нет нулевых элементов
    inc ecx;              номер последнего нулевого
NoZero:
    outwordln ecx,,"Номер последнего нулевого="
```

Заметим, что выход из нашего цикла возможен при попадании на нулевой элемент массива, при исчерпании счётчика цикла, а также при совпадении обоих этих условий. Следовательно, после команд **repne** **scasw** необходимо проверить, имел ли место случай просто выхода из цикла *без* нахождения нулевого элемента, что и сделано командой условного перехода **jnz** `NoZero`.

Следующими рассмотрим команды *загрузки* элемента строки, которые являются командами *пересылки* и, при использовании в цикле, хорошо подходят для эффективной последовательной загрузки на регистр целых чисел длиной 1 (**lodsb**), 2 (**lodsw**) или 4 байта (**lodsd**). В старших моделях добавляется команды **scasq** и **lodsq** для сравнения и загрузки чисел длиной восемь байт Эти команды отличаются только битом размера аргументов `w`, и имеют два *неявных* операнда `op1` и `op2`, где `op1=AL` для `w=0`, и `op1=AX` или `EAX` для `w=1`, а второй неявный операнд `op2=<ESI>` является целым числом длиной 1, 2 или 4 байта в оперативной памяти. Если обозначить буквой `r` регистр `AL`, `AX` или `EAX`, то схему выполнения этих команд можно записать так:

```
    mov r,<esi>; φ(esi)
```

Ясно, что эту команду имеет смысл использовать только совместно с командами сравнения и условного перехода. В качестве примера использования команды загрузки напишем фрагмент программы для реализации следующей задачи: найти и вывести сумму тех элементов *беззнакового* массива коротких целых чисел, значения которых больше 100. Если в массиве нет таких элементов, то будем в качестве ответа выдавать число ноль.

```
N     equ  1000000
```

```
.data
   X  db   N dup (?)
.code
      . . .
   mov  ecx,N;   число элементов
   sub  edx,edx; сумма=0
   sub  eax,eax
   lea  esi,X;   адрес первого элемента
   cld;          прямое направление
L: lodsb
   cmp  al,100
   jbe  NoSum
   add  edx,eax; edx:=edx+Longword(X[i])
NoSum:
   loop L
   outwordln edx
```

Нужная сумма коротких целых чисел получается на регистре EDX. Для упрощения алгоритма переполнение самого регистра EDX не проверяется.

Последними в этом формате SS рассмотрим команды *сохранения* элемента строки, которые являются командами *пересылки* и, при использовании в цикле, хорошо подходят для эффективного присваивания всем элементам массива целого значения длиной 1 (**stosb**), 2 (**stosw**) или 4 (**stosd**) байта. В старших моделях добавляется команда **stosq** для запись в память чисел длиной восемь байт. Эти команды отличаются только битом размера аргументов w, и имеют два *неявных* операнда op1 и op2, где *второй* неявный операнд op2=AL для w=0, и op2=AX или EAX для w=1, а *первый* неявный операнд op1=<EDI> является целым числом соответствующей длины в оперативной памяти. Если обозначить буквой r регистр AL, AX или EAX, то схему выполнения команд можно записать так:

```
      mov <edi>,r; φ(edi)
```

В качестве примера использования команды сохранения напишем фрагмент программы для присваивания всем элементам массива целых чисел в формате **dd** значения единицы.

```
N     equ 300000
.data
   X  dd  N dup (?)
.code
      . . .
   mov  ecx,N; число элементов
   mov  eax,1; присваиваемое значение
   lea  edi,X; адрес первого элемента
   cld;        прямой просмотр массива
rep stosd
```

Рассмотрим ещё один пример. Напишем фрагмент программы для решения задачи присваивания всем элементам знакового массива целых чисел двойной точности Y абсолютных значений соответствующих им элементов массива X, т.е. Y:=abs(X) (такой оператор присваивания – это только иллюстрация условия задачи, так в Паскале для массивов написать нельзя).

```
N     equ 5000
.data
   X  dd  N dup (?)
   Y  dd  N dup (?)
.code
      . . .
   mov  ecx,N
   cld;          Прямой просмотр
   lea  esi,X
   lea  edi,Y
L: lodsd
   cmp  eax,0
   jge  L1
   neg  eax
   jo   Err;     Не существует abs(X[i])
L1: stosd
```

```
    loop L
       . . .
Err:
    outstrln 'В массиве X есть MinLongint !'
       . . .
```

Так как не у каждого отрицательного числа есть соответствующее ему абсолютное значение, то при обнаружении такой ситуации выдается аварийная диагностика. Если контролировать правильность взятия абсолютной величины не надо, то конец этого фрагмента можно записать короче:

```
      L: lodsd
      L2:neg  eax
         js   L2; if eax<0 then eax:=-eax
      ; К сожалению, при eax=MinLongint зациклимся
         stosw
         loop L
```

А можно и вот так, без команды условного перехода, но с использованием лишнего регистра:

```
      L: lodsd
         mov   ebx,eax
         neg   ebx
         cmp   eax,0
         cmovl eax,ebx; if eax<0 then eax:=ebx
         stosw
         loop  L
```

На этом закончим наше краткое изучение строковых команд, напомним, что для хорошего освоения этой темы необходимо использовать рекомендованные Вам учебник и задачник по языку Ассемблера. А теперь перейдём к следующему классу команд – логическим командам.

8.2. Логические команды

Всё объяснимо, если дать себе труд овладеть некоторыми навыками логических построений.

Макс Фрай. «Тихий город»

Все логические команды рассматривают свои операнды как упорядоченные наборы (вектора) битовых значений. В таком наборе может содержаться 8, 16 или 32 бит, в зависимости от размера операнда. Для двух команд из этого класса (для двух команд сдвига) в этом наборе на один бит больше, далее это будет отмечено особо. Бит со значением 1 может трактоваться как логическое значение **`true`**, а нулевой бит – как логическое значение **`false`**, поэтому такие команды и называются логическими. Сначала рассмотрим двухадресные логические команды, имеющие такой общий вид:

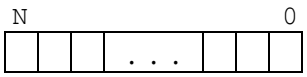
```
КОП op1,op2
```

Ниже приведены допустимые операнды таких команд:

op1	op2
r8	r8, m8, i8
r16	r16, m16, i16
r32	r32, m32, i32
m8	r8, i8
m16	r16, i16
m32	r32, i32

Как видно, у этих команд допускаются точно такие же операнды, как и у команд сложения, вычитания и сравнения.[iii] [см. сноску в конце главы]

Биты в операндах логических команд нумеруются так же, как и в регистрах, справа налево от нуля до числа N, которое равно 7, 15 или 31 соответственно:[1]



[1] Далее следуют N=63 для 64-битных ЭВМ и N=127, 255 и 511 для так тазываемых векторных регистров нашей ЭВМ длиной 128, 256 и 512 бит.

Таким образом, операнды логической команды можно рассматривать как массивы (вектора) логических элементов, проиндексированных от 0 до N. На языке Free Pascal такие массивы можно, например, описать в следующем виде:

```
var op1,op2: bitpacked array[0..N] of Boolean;
```

Здесь не принимается во внимание, что номера битов в логическом операнде, в отличие от индексов элементов массива в Паскале, идут в обратном порядке. Служебное слово **bitpacked** есть *рекомендация* для компилятора Free Pascal по возможности более компактно хранить данные (вплоть до битовой упаковки), даже если это повлечёт за собой увеличения времени доступа к этим данным по чтению и записи. Обычно компилятор "не прислушивается" к такой рекомендации.

Рассмотрим теперь коды операций логических команд. Схему выполнения команды *логического умножения*

```
and op1,op2
```

на языке Паскаль можно записать в виде

```
for i:=0 to N do op1[i]:=op1[i] and op2[i]
```

Схему выполнения команды *логического сложения*

```
or op1,op2
```

на языке Паскаль можно записать в виде

```
for i:=0 to N do op1[i]:=op1[i] or op2[i]
```

Схему выполнения команды *неэквивалентности* (её часто называют также командой побитового сложения по модулю 2, что соответствует алгоритму выполнения этой команды)

```
xor op1,op2
```

на языке Паскаль можно записать в виде

```
for i:=0 to N do op1[i]:=op1[i] <> op2[i]
```

Как уже упоминалось, если допускается портить флаги, то команду **xor**, вместе с командой **sub** часто используют для обнуления регистра, например `xor eax,eax`.

Команда логического тестирования

```
test op1,op2
```

выполняется точно так же, как и команда логического умножения, но *без записи* результата на место первого операнда, т.е. как и у команды сравнения с кодом **cmp** её единственным результатом является установка флагов.

Все перечисленные выше логические команды устанавливают флаги так же, как команды сложения и вычитания, например, флаги CF и OF будут всегда равны нулю (никакого переноса и переполнения, конечно же, нет), во флаг знака SF переносится левый бит результата op1[N] и т.д. Но, как Вы вскоре увидите, чаще всего интерес для программиста здесь представляет только флаг нулевого результата ZF, который, как обычно, равен единице (поднят) для полностью нулевого результата, и равен нулю (опущен), если в результате есть хотя бы один ненулевой бит.

И, наконец, рассмотрим одноадресную команду логического отрицания

```
not op1
```

где op1 имеет такие же допустимые форматы, как и первый операнд всех рассмотренных выше логических команд. Схему выполнения этой команды на Паскале можно записать как

```
for i:=0 to N do op1[i]:=not op1[i]
```

Заметим, что команда логического отрицания, в отличие от остальных логических команд, не меняет флаги.

Следует заметить, что цикл **for** языка Паскаль использовался только для иллюстрации и более строгого описания работы логических команд. Не следует понимать это слишком буквально: на самом деле логические команды выполняют операции над всеми битами своих операндов одновременно и быстро – за один такт работы процессора, а совсем не в последовательном цикле.[1]

---

[1] Для любознательных читателей отметим, что для описания работы логических команд более подходит оператор цикла, скажем, какого-нибудь, например, диалекта параллельного Фортрана, скажем, для команды логического умножения двух векторов:

```
for all i in [0..N] do op1[i]:=op1[i] and op2[i]
```

В языках высокого уровня тоже реализованы логические операции для параллельной обработки всех битов целочисленных операндов, например:

```
var x,y: Longword;
   . . .
    x:=x and y; y:=not x;
```

это будет на Ассемблере реализовано как

```
.data
   x dd ?
   y dd ?
   . . .
   mov eax,y
   and x,eax; x:=x and y
   not eax;   not x
   mov y,eax; y:=not x
```

Теперь рассмотрим использование логических команд для работы с *логическими* переменными языков высокого уровня. Пусть на Паскале заданы описания

```
var X,Y: boolean; A: integer;
```

и оператор присваивания

```
X:=(not X or Y) and (A>1)
```

В качестве логических констант будем использовать байтовые значения **false**=0 и **true**=0FFh,[1] тогда на Ассемблере этот оператор присваивания можно записать так:

```
X  db ?
Y  db ?
A  dw ?
    . . .
   mov al,X
   not al
   or  al,Y
   jz  @F;   al=(x or y)=false
   cmp A,1;  сейчас al=true
   jg  @F
   not al;   al:=false
@@: mov X,al
```

В этом примере мы использовали две команды условного перехода, на которые очень "болезненно" реагируют конвейеры современных ЭВМ (об этом мы будем подробно говорить в другой главе). От второй из этих команд (**jg**) можно избавиться, заменив две команды

```
jg  @F
not al;     al:=false
```

на одну команду условного присваивания:

```
movle al,0; if A<=1 then al:=false
```

Команды **and** и **test** часто используются для так называемой операции "выделения по маске". Имеется в виду, что в результате этих команд неизменными остаются только те биты первого операнда, для которых соответствующие биты второго операнда установлены в "1". Например, рассмотрим команды:

```
A  dw ?
      . . .
   test A,111b;
   jz   L
```

Этот оператор предписывает выполнить оператор в теле цикла *одновременно* (параллельно) сразу для *всех* значений параметра цикла i. На языке Free Pascal можно использовать аналогичный цикл:

```
for i in [0..N] do op1[i]:=op1[i] and op2[i]
```

но параллельного исполнения операторов в теле цикла здесь не будет. Впрочем, современные компиляторы могут использовать особые векторные регистры для распараллеливания таких циклов. Один такой пример мы вскоре приведём.

[1] В некоторых языках высокого уровня (например, в языке C), ненулевое значение любого типа считается константой **true**, а нулевое – **false**. Это не очень удобно, так как мы бы хотели, чтобы машинная команда **not false** давала **true** и, наоборот, **not true** давала **false.**

Эти две команды реализуют оператор Паскаля

```
if A mod 8 = 0 then goto L
```

Первая команда по маске 111b оставила в результате только три последних бита переменной A, что и является остатком от деления этой переменной на 8. Простейшая маска – однобитовая, она позволяет проверить в первом операнде один конкретный бит, например

```
test A,1b;
jnz  Odd; Нечётное A
```

8.3. Команды сдвига

Чего не понимают, тем не владеют.
Иоганн Вольфганг Гёте

Команды логического сдвига предназначены для сдвига (изменения индексов) битов в своём операнде. Операнд при этом можно рассматривать как битовый вектор, элементы которого пронумерованы от 0 до N так же, как и для рассмотренных до этого логических команд. Команды сдвига имеют формат

```
КОП op1,op2
```

Здесь первый операнд op1 задаёт что сдвигать, а второй – на сколько бит. Операнд op1 может быть r8, r16, r32, m8, m16 или m32, а op2 – иметь формат i8 или быть коротким регистром cl.[1] Сначала рассмотрим частный случай команды сдвига вида **КОП** op1,1, а затем обобщим их на общий случай.

Команда сдвига операнда на один бит *влево* имеет показанный ниже вид

```
shl op1,1
```

и её выполнение можно так описать в виде операторов на Паскале:

```
CF:=op1[N];
for i:=N downto 1 do op1[i]:=op1[i-1];
op1[0]:=0
```

Электронные схемы, реализующие такие операции, называются *сдвиговыми регистрами*, они выполняют эту работу за один такт (все биты "дружно" перебегают на нужное число позиций влево или вправо). На рис. 8.2 показан результат выполнения этой команды.

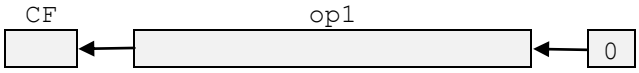


Рис. 8.2. Схема выполнения команды **shl** op1,1.

Как видим, при сдвиге битового вектора на одну позицию влево самый левый бит не теряется, а попадает во флаг переноса CF, а на освободившееся справа место записывается бит со значением ноль. Все команды сдвига устанавливают флаги по тем же правилам, что и команды сложения и вычитания. Например, флагу-признаку отрицательного результата SF присваивается самый левый бит результата, т.е. SF:=op1[N]. Однако среди флагов, изменяемых командами сдвигов, в практическом программировании имеет смысл рассматривать только флаг переноса CF и флаг нуля ZF.

Необходимо также заметить, что сдвиг операндов возможен и в оперативной памяти. При этом, однако, так как *на самом деле* сдвиг производится на некотором служебном сдвиговом регистре АЛУ, то при нумерации битов в операнде уже не учитывается *перевёрнутое* представление байтов в оперативной памяти, так как при вызове на этот служебный регистр байты снова переворачиваются.

У названия кода операции сдвига влево **shl** (она называется *логическим* сдвигом влево), есть синоним **sal**, который называется *арифметическим* сдвигом влево. Логический и арифметический сдвиги выполняются одинаково, а различие в их названиях будет понятно из дальнейшего изложения.

В том случае, если операнд команды сдвига влево на один бит трактуется как *целое число*, то результатом сдвига является умножение этого числа на два. При этом результат умножения получается правильным, если во флаг переноса CF попадает *незначащий* бит результата. Таким образом, для *беззнакового* числа при правильном умножении на 2 должно быть CF=0. А вот для *знакового* опе-

---

[1] В старших моделях семейства у команд сдвига первый параметр может также иметь форматы r64 и m64.

ранда результат получается правильным только тогда, когда значение флага переноса совпадает со знаковым (крайним слева) битом результата, т.е. после выполнения команды сдвига справедливо равенство CF=op1[N]=SF (в этом случае флаг переполнения OF=0). Таким образом, правильность умножения на два с помощью команды сдвига влево можно контролировать, как обычно, анализируя флаги переноса CF и переполнения OF для беззнаковых и знаковых чисел соответственно.

Заметим, что беззнаковое и знаковое умножение в нашем компьютере делается *разными* командами **mul** и **imul**, однако умножение на 2 с помощью команды сдвига влево делается по одному алгоритму, как для знаковых, так и для беззнаковых чисел. Именно поэтому операция логического сдвига влево имеет два имени-синонима: логический (т.е. беззнаковый) сдвиг **shl** и арифметический (т.е. знаковый) сдвиг **sal**.[1]

Рассмотрим теперь команды сдвига на один разряд *вправо*. По аналогии со сдвигом влево, сдвиг на один разряд вправо можно трактовать как *деление* целого числа на два. Однако так как деление на два должно выполняться по разным правилам для знаковых и беззнаковых целых чисел (вспомним *различные* команды **div** и **idiv**), то существуют две *разные* команды сдвига операнда на один бит вправо. Команда *логического* сдвига на один разряд вправо

```
shr op1,1
```

выполняется по правилу, которое можно так описать на Паскале:

```
CF:=op1[0];
for i:=0 to N-1 do op1[i]:=op1[i+1];
op1[N]:=0
```

На рис. 8.3 показана схема выполнения этой команды.

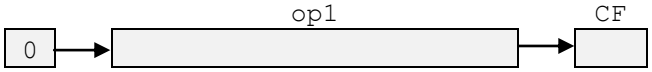


Рис. 8.3. Схема выполнения команды **shr** op1,1.

Команда *логического* сдвига на один разряд вправо эквивалентна делению на два *беззнакового* целого числа, результат при этом всегда получается правильным. Делению на два *знакового* целого числа в определённом смысле эквивалентна команда *арифметического* сдвига операнда на один бит вправо

```
sar op1,1
```

она выполняется по правилу:

```
CF:=op1[0];
for i:=0 to N-1 do op1[i]:=op1[i+1]
```

На рис. 8.4 показана схема выполнения этой команды.

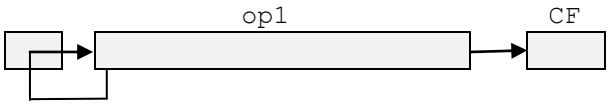


Рис. 8.4. Схема выполнения команды **sar** op1,1.

Как видно, крайний левый бит аргумента при арифметическом сдвиге вправо *не меняется*. При трактовке операнда сдвига как знакового числа простой анализ показывает, что для *неотрицательных* и *отрицательных* чётных значений арифметический сдвиг вправо всегда эквивалентен операции деления этого аргумента на 2. В то же время для отрицательных нечётных значений это неверно, например, (-5) **div** 2 = (-2), а **sar** (-5), 1 = (-3), т.е. на единицу меньше [iv] [см. сноску в конце главы]. Таким образом, правильное деление на 2 сдвигом любой знаковой величины X реализует, например, оператор на языке Free Pascal

```
X:=(X+ord(X<0)) shr 1; {X+1 для X<0}
```

На Ассемблере можно предложить такие фрагменты на Ассемблере:

---

[1] Команды **shl** и **sal**, хотя и выполняются одинаково, но имеют разные коды операций (D0Exh и D0Fxh). Обычно Ассемблеры транслируют эти команды в код D0Exh (x определяет операнд так же, как поле mem в формате команд RX).

```
   sar X,1                  mov eax,X                 mov eax,X
   jns @F;  X >= 0          shl X,1;   CF=1 для X<0   cdq
   adc X,0                  adc eax,0; +1 для X<0     sub eax,edx; 1 для X<0
; +1 для нечётного X<0      sar eax,1                 sar eax,1
@@:; X:=X div 2             mov X,eax; X:=X div 2     mov X,eax; X:=X div 2
```

Последние колонки длиннее на две команды и использует регистры, однако там нет команды условного перехода, которую очень "не любят" конвейеры современных процессоров. Третья колонка использует два регистра (EAX и EDX), но содержит всего два обращения в память.[1]

Заметим далее, что, как и "настоящие" команды деления, сдвиг вправо даёт *два* результата: частное на месте своего операнда и остаток от деления на 2 во флаге CF. Действительно, легко видеть, что

```
CF:=op1 mod 2 =  0 или 1  для беззнакового операнда и
              = -1 или 1  для знакового операнда
```

Следующая группа команд сдвига – так называемые *циклические* сдвиги. Эти команды рассматривают свой операнд как замкнутый в кольцо: после бита с номером N располагается бит с номером 0, а перед битом с номером 0 – бит с номером N. Ясно, что при циклическом сдвиге операнд сохраняет все свои биты, меняются только номера этих битов внутри операнда. Команда циклического сдвига *влево*

```
rol op1,1
```

выполняется по правилу

```
shl op1,1; op1[0]:=CF
```

Другими словами, сначала выполняется команда логического сдвига влево, а потом в результате корректируется значение нулевого бита. На рис. 8.5 показана схема выполнения этой команды.

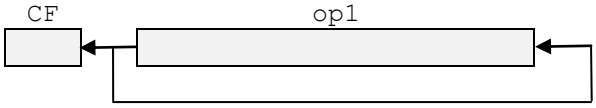


Рис. 8.5. Схема выполнения команды **rol** op1,1.

Команда циклического сдвига *вправо*

```
ror op1,1
```

выполняется по правилу

```
shr op1,1; op1[N]:=CF
```

На рис 8.6 показана схема выполнения этой команды.

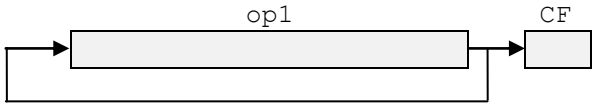


Рис. 8.6. Схема выполнения команды **ror** op1,1.

Команды циклического сдвига *через флаг переноса* включают в кольцо сдвигаемых битов дополнительный бит – флаг переноса CF, который включается между битами с номерами 0 и N. Таким образом, в сдвиге участвуют N+1 бит. Команда циклического сдвига *влево через флаг переноса*

```
rcl op1,1
```

выполняется по правилу (см. рис. 8.7)

```
temp:=CF; shl op1,1; op1[0]:=temp
```

Здесь temp – битовая вспомогательная (временная) переменная процессора.

---

[1] Для продвинутых читателей. Как видно, при реализации деления *знаковых* чисел посредством сдвигов, например, x:=x **shr** 1 (на Free Pascal) или **int** x; x>>=3; (на языке C), возникают определённые трудности. Исходя из этого, в стандарте языков C и C++ сказано, что сдвиги знаковых значений в программах приводят к так называемому *неопределенному поведению* UB (undefined behavior). К такому же эффекту приводит и *переполнение* знаковых целых чисел. Это означает, что "как это будет работать толком неизвестно, но всё будет плохо"  . На 2017 год в стандарте C и C++ описано около 200 видов UB. Заметим также, что для языков высокого уровня у "жуткого демона" UB есть и "бес-младший брат" *не специфицированного* поведения программы (unspecified behaviour).

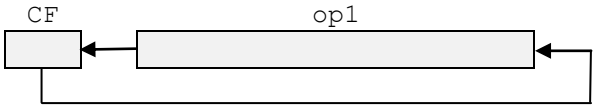


Рис. 8.7. Схема выполнения команды **rcl** op1,1.

Команда циклического сдвига *вправо через флаг переноса*

```
rcr op1,1
```

выполняется по правилу

```
temp:=CF; shr op1,1; op1[N]:=temp
```

На рис. 8.8 показана схема выполнения этой команды.

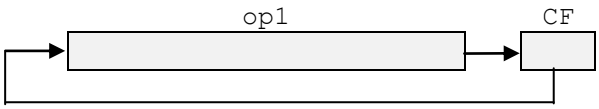


Рис. 8.8. Схема выполнения команды **rcr** op1,1.

Теперь осталось описать общий случай команды сдвига. Каждая такая команда (**КОП** – любой из кодов операций команд сдвигов)

```
КОП op1,op2; op2=i8,CL
```

выполняется по правилу

```
for i:=1 to (op2 mod 32) do КОП op1,1
```

Таким образом, значение операнда op2 задаёт число разрядов, на которые происходит сдвиг операнда, при этом допускается и op2=0. Ясно, что задавать сдвиги операнда, которые больше или равны его длине бессмысленно, поэтому для величины сдвига берутся младшие 3, 4 или 5 бит (6 бит в 64-битном режиме). При сдвигах на несколько бит флаг OF не определён, остальные флаги устанавливаются по *последнему* сдвигу, то есть проверить правильность операций умножения и деления на степень двойки не получится 😕.

Команды сдвига в практике программирования иногда очень эффективны, вот в качестве примера вычисление среднего арифметического двух *беззнаковых* чисел:

```
add op1,op2
rcr op1,1; op1:=(op1+op2) div 2
```

Вот проверка регистра EAX на принадлежность его значения типу integer ($-2^{15}..2^{15}-1$):

```
shr eax,15
inc eax
shr eax,1
jnz Out_of_integer
```

Пример деления регистра AX на 512, предложенный Марком Збиковским, автором предшественницы Windows, древней ОС MS-DOS:

```
shr  ax,1
xchg ah,al
cbw
```

А вот обмен двух переменных формата **dw** *в памяти*:

```
X  dw ?
Y  dw ?
   ...
   rol dword ptr X,16; xchg(X,Y)
```
[1]

В современных процессорах Intel существует и много других логических команд.[v] [Смотри сноску в конце главы]. Главное назначение логических команд – обрабатывать отдельные биты и группы битов. Как известно, минимальной порцией данных, отрабатываемой в командах, является в нашей архитектуре байт, а не бит, поэтому в языке машины и предусмотрены логические команды.

---

[1] Не надо обольщаться, на уровне микроархитектуры команда **rol dword ptr** X,16 выполняется по схеме: temp:=**dword ptr** X; **rol** temp,16; **dword ptr** X:=temp, где temp – служебный, так называемый сдвиговый регистр процессора, т.е. без вспомогательной переменной, как и в классическом алгоритме обмена, тоже не обойтись.

Разберём несколько примеров использования логических команд в программировании на Ассемблере. Сначала составим фрагмент программы, в котором подсчитывается и выводится число битов со значением "1" во внутреннем машинном представлении переменной X размером в слово. Без использования логических команд это можно сделать, например, с помощью такого фрагмента программы на Ассемблере:

```
    mov ax,X
    sub cx,cx; число "1"
    mov bx,2
@@:test ax,ax
    jz   @F
    xor  dx,dx
    div  bx;     dx:=<dx:ax> mod 2
    add  cx,dx
    jmp  @B
@@:outint cx
```

А теперь решим эту же задачу с использованием логических команд:

```
    mov  ax,X
    sub  cx,cx; число "1"
@@:shl  ax,1
    adc  cx,0; cx:=cx+0+CF
    test ax,ax
    jnz  @B
    outint cx,,"Число '1' в X="
```

Этот алгоритм будет работать быстрее за счёт того, что медленная команда деления заменяется на быструю команду сдвига, кроме того, теперь использовано на два регистра меньше.

Заметим, что операция подсчёта числа битов машинного слова со значением "1" является весьма важной в архитектуре ЭВМ. В качестве примера можно привести отечественную ЭВМ третьего поколения БЭСМ-6, которая производилась в конце 60-х годов прошлого века [3]. В этой ЭВМ сигналы прерывания устанавливали в "1" биты в специальном 48-разрядном *регистре прерываний* (каждый бит соответствовал своему номеру сигнала прерывания). В этой архитектуре существовала специальная машинная команда для подсчёта количества "1" в своем аргументе, что позволяло быстро определить число ещё необработанных сигналов прерывания. Такая же команда **popcnt** (POPulation CouNT) введена с 2008 года в процессоры Intel с поддержкой нового набора инструкций SSE4.2 (только в Ассемблере MASM 10.xx и выше):

```
popcnt op1,op2; op1:=Число "1" в op2
```

с такими допустимыми операндами

op1	op2
r16	r16, m16
r32	r32, m32

В нашей версии MASM 6.14 приходится кодировать эту команду через константы, например, команда `popcnt eax,ecx` будет выглядеть как

```
db 0F3h,0Fh,0B8h,11000001b; (11=RR)(eax=000)(ecx=001).
```

Рассмотрим теперь использование логических команд при обработке *упакованных* структур данных. Пусть, например, на языке Free Pascal дано описание упакованного массива с элементами ограниченного целого типа:

```
const N=1000000; type Int4=0..15;
var   A: bitpacked array[0..N-1] of Int4; X: byte;
```

Рассмотрим фрагмент программы на Ассемблере, который работает с описанным выше упакованным массивом A. Реализуем, например, оператор присваивания X:=A[i]. Каждый элемент массива требует для своего хранения 4 бита памяти, так что будем в одном байте хранить два последовательных элемента массива A:

```
  N    equ 1000000
.data
  A  db  N/2 dup (?); N/2 ≡ N div 2
  X  db  ?
.code
```

```
;  реализация оператора X:=A[i]
   mov  ebx,i
   shr  ebx,1;     CF:=i mod 2; ebx:=i div 2
   mov  al,A[ebx]; в al два элемента из A
   jc   L1;        i-ый элемент - правый
   shr  al,4;      i-ый элемент был левым
L1: and  al,1111b; выделение A[i]
   mov  X,al;      X:=A[i]
```

Здесь сначала командой сдвига вправо значение индекса `i` (это беззнаковое число!) было разделено на 2, так определён тот байт массива `A`, в котором находится пара элементов, один из которых нам нужен. На положение нужного элемента в байте указывает остаток от деления индекса `i` на 2 во флаге `CF`: если остаток равен нулю (т.е. `i` чётное), то элемент расположен в *левых* четырёх битах байта, иначе – в *правых*. Для выделения нужного элемента, который занимает в байте только 4 бита из 8-ми, использовалась команда логического умножения **`and al,1111b`**, где второй операнд для удобства понимания смысла команды задан в виде двоичного числа, на что указывает буква `b` в конце (вместо `b` в Ассемблере можно использовать и букву `y` – **b**inar**y**).

Использование команды логического умножения для выделения нужной нам части байта или слова, и обнуление остальной части является типичной для программирования на Ассемблере. При этом константа, используемая для выделения нужной части (у нас это `00001111b`), называется *маской выделения* или просто маской. Таким образом, элемент массива `A`, который занимает 4 бита, записан в регистр `AL` и дополнен до 8 бит нулями слева. Заметим, что это не изменило значения элемента, так как он беззнаковый (`0..15`).

Этот простой пример показывает, что при работе с упакованными данными скорость доступа к ним уменьшается в несколько раз, и программист должен решить, стоит ли это достигнутой экономии памяти (в нашем примере сэкономлено 500000 байт оперативной памяти). Обычно это типичная ситуация в программировании: выигрывая в скорости обработки данных, проигрываем в объёме памяти для хранения этих данных, и наоборот. Иногда, по аналогии с физикой, это называют "законом рычага", который гласит, что, выигрывая в силе, проигрываем в длине перемещения конца рычага, и наоборот. Заметим также, что самый большой выигрыш по памяти получается при обработке упакованных массивов логических величин, например:

```
var Z: bitpacked array[1..1000000] of boolean;
```

В качестве ещё одного примера рассмотрим реализацию на Ассемблере некоторых операций языка Паскаль над *множествами*. Пусть на Паскале есть описания двух символьных множеств `X` и `Y`, а также символьной переменной `Sym`:

```
var X,Y: set of char; Sym: char;
```

Сначала напишем на Ассемблере фрагмент программы для реализации операции объединения двух множеств:

```
X := X + Y
```

Каждое такое множество будем хранить в памяти в виде 256 последовательных битов, т.е. 32 байт, 16 слов или 8 двойных слов. Бит, расположенный в позиции `i` принимает значение 1, если символ с кодом (номером) `i` присутствует во множестве, иначе этот бит имеет значение 0 (заметим, что язык Free Pascal реализует именно такое представление для переменных типа **`set of char`**). Множества `X` и `Y` можно так описать на Ассемблере:

```
.data
   X  dd  8 dup (?); 256 бит
   Y  dd  8 dup (?)
```

Тогда операцию объединения двух множеств можно реализовать, например, таким фрагментом на Ассемблере:

```
   mov  ecx,8
@@: mov  eax,Y[4*ecx-4]
   or   X[4*ecx-4],eax
   loop @B
```

Обратите внимание на такое обстоятельство. Несмотря на то, что множества `X` и `Y` из последнего примера являются *упакованными* структурами данных (упакованными логическими векторами из 256 элементов), при работе с ними происходит не замедление, а *увеличение* (в нашем случае в 32

раза) скорости работы, по сравнению с неупакованными данными. Ясно, что такое увеличение скорости происходит из-за параллельной обработки логической командой всех битов своих операндов.[1]

В заключение рассмотрим условный оператор, включающий знакомую нам по языку Паскаль операцию отношения **in** (входит ли символ Sym во множество X), например:

```
if Sym in X then goto L
```

На Ассемблере этот оператор можно реализовать в виде такого фрагмента программы:

```
X   db 32 dup (?); 32*8=256 бит
Sym db ?
      . . .
    movzx ebx,Sym;    ebx:=Longword(ord(Sym))
    shr   ebx,3;      индекс нужного байта в X
    mov   al,X[ebx]; байт с битом символа Sym
    mov   cl,Sym
    and   cl,111b;    позиция символа Sym в байте
    shl   al,cl;      наш бит в al – крайний слева
    js    L;          и он же в SF
```

Разберём, как работает эта программа. Сначала, используя команду логического сдвига на 3 (это аналогично делению беззнакового числа на 8), на регистр EBX заносим индекс того байта в массиве X, в котором находится бит, соответствующий символу Sym во множестве X. Затем этот байт выбирается на регистр AL, а на регистр CL помещаем три последних бита номера символа Sym в алфавите – это и будет номер нужного нам бита в выбранном байте (биты в байте для переменной-множества нумеруются, как нам и нужно, *слева направо*, начиная с нуля). После этого первой командой сдвига перемещаем нужный нам бит в крайнюю левую позицию в байте, и он же попадает в SF. Теперь осталось только сделать условный переход **js** по значению этого флага.

Как видим, логические команды очень удобны для работы с битовыми (логическими векторами). Поэтому не удивительно, что во многие реализации языков высокого уровня включена возможность использовать эти машинные операции. В языке Free Pascal над целочисленными данными, которые, естественно, в этом случае трактуются как битовые вектора, определены бинарные операции **and**, **or**, **xor** и унарная операция **not**, эквивалентные соответствующим машинным командам. Кроме того, предусмотрены также сдвиги влево и вправо над такими целочисленными данными. Например, в языке Free Pascal команду Ассемблера `shl x, 3` можно записать в виде оператора присваивания `x:=x shl 3`.

На этом заканчивается рассмотрение логических команд.

8.4. Упакованные битовые поля

Раз мы взялись за новое дело, мы должны иначе думать и действовать.

Авраам Линкольн

Одним из применений логических команд и команд сдвигов является работа с полями битов, являющихся частями переменных. Для автоматизации работы с такими данными Ассемблер предоставляет в распоряжение программиста определённые языковые средства. К таким средствам, в частности, относятся так называемые упакованные битовые поля или *записи* (**record**). На примере упакованных битовых полей Вы познакомитесь со способами описания *типов* в языке Ассемблера.

До сих пор для работы с переменными в Ассемблере использовались только предложения резервирования памяти (**db**, **dw**, и т.д.), которые являются аналогами описания переменных стандартных типов в языках высокого уровня. Теперь познакомимся с аналогами в языке Ассемблера описания пользовательских *типов* данных. Использование таких типов данных в Ассемблере, как и во всяком

---

[1] Для продвинутых читателей. Ещё быстрее объединение двух множеств реализуется с использованием 256-разрядных YMM регистров, например:

```
vmovdqa ymm0,ymmptr X;       ymm0:=8*dd=X
vpor    ymm0,ymm0,ymmptr Y; ymm0:=X or Y
vmovdqa ymmptr Y,ymm0;       Y:=ymm0=X or Y
```

алгоритмическом языке, повышает надёжность программирования и делает программу лучше для понимания и модификации.

Битовым полем в Ассемблере называется последовательный набор битов в байте, слове или двойном слове, причём каждому такому битовому полю пользователь присваивает имя. В одном байте, слове или двойном слове можно определить несколько битовых полей, если их общая длина не превышает, соответственно, размера байта, слова или двойного слова. Так организованные данные и называются *упакованными битовыми полями* (имеется в виду, что эти поля плотно упакованы, т.е. примыкают друг к другу, в памяти).

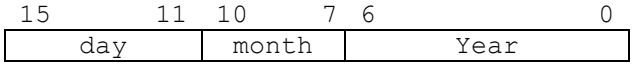
В Ассемблере для упакованных битовых полей предусмотрено описание типа. В качестве примера рассмотрим описание трёх упакованных битовых полей. Присвоим этому типу данных имя `date`, что будет отражать суть трёх входящих в этот тип битовых полей, предназначенных для хранения некоторой даты (числа, месяца и двух последних цифр года):

```
date record day:5,month:4,year:7=20
```

Из этого описания следует, что переменная с именем типа `date` для своего хранения потребует одно машинное слово (два байта=5+4+7=16 бит), в котором будут содержаться битовые поля с именами `day` (длиной 5 бит), `month` (4 бита) и `year` (7 битов). Заметим, что некоторым аналогом этого описания в языке Free Pascal будет, например, такое описание типа (при условии, что Паскаль-машина "прислушается" к нашей рекомендации использовать именно упакованную структуру данных):

```
type date = bitpacked record
       day: 1..31; month: 1..12; year: 0..99
     end;
```

По сравнению с Паскалем, однако, имена полей являются глобальными (а не локализованы внутри записи). Упакованные битовые поля располагаются в памяти в порядке их описания, но выравниваются по *правому* краю байта, слова или двойного слова, в зависимости от суммы длин всех полей записи. Ниже показан вид переменной типа `date` с указанием номеров битов машинного слова, занимаемыми битовыми полями:



Подчеркнём, что имя `date` является именем *типа,* и показанное выше предложение Ассемблера не резервирует в памяти никакой области данных (никакой переменной), то есть, является с точки зрения языка Ассемблер *директивой*. Само же резервирование областей памяти для хранения переменных типа `date` производится в некоторой секции по предложениям резервирования памяти. Ниже показаны примеры таких предложений резервирования памяти для хранения переменных типа `date`, в комментариях показаны начальные значения полей.

```
dt1  date <,8>
dt2  date <21,4,96>
dt3  date <>
dt4  date {13,5,?}
```

	day	**month**	**year**
dt1	0	8	18
dt2	21	4	96
dt3	0	0	18
dt4	13	5	0

Обратите внимание, что переменные типа упакованных битовых полей *всегда* порождаются с начальными значениями, которые либо задаются программистом в параметрах предложения резервирования памяти, либо берутся из описания данного типа, в остальных случаях они по умолчанию считаются равными нулю.[1] В нашем примере описания типа `date` указано, что битовое поле с именем `year` должно иметь начальное значение по умолчанию 18, если при порождении такой переменной этому полю *явно* не будет задано другое значение. В директивах резервирования памяти начальные значения битовых полей перечисляются через запятые в угловых или фигурных скобках. Заметьте также, что символ `?` здесь при резервировании памяти задаёт *не неопределённое*, как в других случаях, а *нулевое* начальное значение соответствующего упакованного битового поля.

---

[1] Как следствие для переменных типа **`record`** в секции **`.data?`** будет выдаваться предупредительная диагностика, а сами переменные будут иметь *неопределённые* начальные значения.

Языковые средства, которые Ассемблер предоставляет для работы с упакованными битовыми полями, разберём на следующем примере. Напишем фрагмент программы, в котором берётся значение переменной с именем X типа date, и хранящаяся в этой переменной дата выводится в привычном виде, например, как день/месяц/год. Так как для значения года хранятся только две последние десятичные цифры, то здесь у нас тоже возникает своя "проблема 2000 года". Сделаем следующую спецификацию нашей задачи: если значение года больше 50, то будем считать дату принадлежащей *прошлому* веку, иначе – *текущему* веку.[1] Ниже приведён фрагмент программы, решающий эту задачу.

```
date record day:5,month:4,year:7=20
.data
   X  date <>
.code
      . .
   mov     ax,X
   mov     bx,ax
;  имена полей имеют значения day=11,month=7,year=0
;  это № первой слева позиции поля в переменной
   shr     ax,day;         ax:=<0,0,day>
   outword ax
   mov     ax,bx
   and     ax,mask month; mask month=0000011110000000b
   shr     ax,month;       ax:=<0,0,month>
   outword ax,,"/"
   mov     ax,bx
   and     ax,mask year;  mask year=0000000001111111b
   mov     bx,2000;        нынешний век
   cmp     ax,50
   jbe     @F
   mov     bx,1900;        прошлый век
@@: add     ax,bx;          ax:=Год из 4-х цифр
   outword ax,,"/"
```

Прокомментируем эту программу. Почти всем именам, которые программист записывает в предложениях Ассемблера, приписываются определённые *числовые* значения, которые Ассемблер и *подставляет* на место этого имени в программу. Так, значением имён переменных и имён меток являются их адреса в памяти,[2] значением имён целочисленных констант, определяемых директивами эквивалентности – сами эти константы и т.д. Отметим, что некоторые имена могут вообще не иметь *числовых* значений, это, например, вот такое уже знакомое Вам имя переменной S в стековом кадре, заданное директивой

```
S equ dword ptr [ebp-8]
```

Имя упакованного битового поля тоже имеет значение, это номер, начальной позиции этого поля в переменной. Как следствие, это количество разрядов, на которое надо сдвинуть это поле, чтобы оно попало в самую правую позицию той области памяти, в которой хранится данное поле. Например, имя month имеет значение 7. Таким образом, имена упакованных битовых полей в Ассемблере предназначены в программе для задания числа сдвигов, что и использовалось в этом примере.

Другое языковое средство, предоставляемое Ассемблером для работы с упакованными битовыми полями – это одноместный оператор, который задаётся служебным именем **mask**. Операндом этого оператора должно являться имя упакованного битового поля, а значением данного оператора будет байт, слово или двойное слово, содержащее биты "1" только в позициях, занимаемых указанным полем, в остальных позициях будут находиться нулевые биты. Как видим, оператор **mask** удобно ис-

---

[1] Такая спецификация сделана исключительно для учебных целей, она проста для понимания и использования, но допускает хранение некоторых дат в двух форматах. Например, даты, записанные как 01/01/110 и 01/01/10, будет соответствовать одной "настоящей" дате 01/01/2010. Это происходит из-за того, что из всего возможного диапазона хранимых в таком виде 128 лет (0–127) использовался только диапазон в 100 лет.

[2] Как исключение, метки, указанные как операнды в командах *относительного* перехода имеют значения констант, задающих знаковое расстояние до точки перехода в байтах.

пользовать для задания констант (масок) в командах логического умножения для выделения нужного битового поля, это и было продемонстрировано в приведённом выше примере.

И, наконец, рассмотрим ещё один оператор Ассемблера, предназначенный для работы с именами упакованных битовых полей. Значением оператора

width <имя битового поля>

является ширина этого поля, т.е. количество составляющих его бит. Например, для нашего описания типа `date` выражение **`width`** `month` равно 4. Заметим, что если применить этот оператор к имени *типа*, задающего упакованные битовые поля, то значением оператора **width** будет сумма длин всех полей в этом типе. Например, **`width`** `date` равно 16. Оператор **width** достаточно редко встречается в практике программирования, и примеры его использования здесь приводиться не будут.

Итак, из рассмотренного выше примера видно, что упакованные битовые поля, как и другие упакованные структуры данных, которые рассматривались ранее, позволяют экономить место в памяти для размещения данных, но требуют значительно больше команд для манипулирования такими упакованными данными. На этом закончим наше краткое знакомство с упакованными битовыми полями в языке Ассемблера.

8.5. Структуры на Ассемблере

Умные структуры данных и тупой код работают куда лучше, чем наоборот.
Эрик Стивен Рэймонд

Другим примером описания *типов* в языке Ассемблера являются структуры (structure). В языке Паскаль аналогом структур Ассемблера являются записи, которые позволяют описать совокупность именованных переменных (вообще говоря, разных типов) как новый самостоятельный тип данных. Описания типа структуры задаются в языке Ассемблера в виде

```
<имя типа структуры> struc
    Предложения резервирования
    памяти под поля структуры
<имя типа структуры> ends
```

Например, в динамической библиотеке Windows `kernal32.dll` используется такая структура для хранения времени:

```
SystemTime struc
   wYear         dw ?
   wMonth        dw ?
   wDayOfWeek    dw ?
   wDay          dw ?
   wHour         dw ?
   wMinute       dw ?
   wSecond       dw ?
   wMilliseconds dw ?
SystemTime ends
```
[1]

Полями структуры могут быть другие структуры, например:

```
Koord struc
  X   dw ?
  Y   dw ?
Koord ends
ColorPount struc
  Color db    ?
  Point Koord <>
ColorPount ends
```

В качестве ещё одного примера рассмотрим описание простой структуры, предназначенной для хранения информации о студенте: текстовой строки, в которой будет храниться фамилия студента (не более 20 символов), а также трёх полей для хранения даты рождения (дня, месяца и года).

---

[1] Как это принято в языке C, имена полей начинаются с буквы `w`, чтобы напоминать программисту о типе этих переменных (**dw**). Такой выбор имён переменных называется венгерской нотацией, о ней будет немного говориться в следующей главе.

```
Stud  struc
Fio   db '********************'; 20 '*'
      db 0; признак конца строки для вывода
Day   db ?
Month db ?
Year  dw 2001
Stud  ends
```

В этой структуре описано 5 полей. Первое поле с именем `Fio` длиной 20 байт предназначено для хранения фамилии студента, у этого поля есть значение по умолчанию – это строка из 20 звёздочек. Обратите внимание на следующую тонкость: это предложение нельзя записать в виде

```
Fio db 20 dup ('*'); 20 '*'
```

так как в этом случае будет определено 20 полей длиной по одному байту каждое. Соответственно, если в первом случае имя `Fio` является именем всей *строки символов*, то во втором – будет только именем *первого* символа в массиве из 20 символов. Размер памяти в байтах, занимаемой переменной соответствующего типа, можно определить с помощью оператора **`sizeof`**, например:

```
X dw ?
Y dd 10 dup (?)
  type X = 2          sizeof X = 2
  type Y = 4          sizeof Y = 40
  type Stud.Fio = 1   sizeof Stud.Fio = 20
```

Следующее *безымянное* поле структуры `Stud` со значением нуля предназначено для удобного вывода фамилии студента с использованием макрокоманды **`outstr`**.

Три последних поля нашей структуры определяют переменные для хранения числовых значений дня, месяца и года рождения студента, причем у года рождения предусмотрено значение по умолчанию 2001. Необходимо подчеркнуть, что в отличие от имён упакованных битовых полей, имена полей структуры *локализованы* в ней, и вне структуры *не видны*, здесь полная аналогия с именами полей записей Паскаля.

После описания структуры можно резервировать в некоторой секции переменные описанного типа с помощью предложений резервирования памяти, например:

```
St1  Stud <'Иванов И.И.',,21,4>
St2  Stud <'Петров П.П.',,31,5,1998>
Grup Stud 24 dup (<>); Студенческая группа
```

При резервировании памяти под структуры вместо угловых скобок можно, если так больше нравится, использовать и фигурные скобки,[1] например:

```
St3  Stud {'Сидоров С.С.',,1,12}
```

Строки, задающие начальные значения фамилий, дополняются *пробелами справа*, если их длина менее длины поля `Fio` (в нашем примере менее 20 символов). Обратите внимание, что двумя запятыми подряд выделено второе *безымянное* поле, которому при размещении переменной в памяти, таким образом, не присваивается *нового* значения, и это поле сохраняет начальное значение по умолчанию 0, определённое при описании структуры `Stud`. Для студента с фамилией `Иванов` не задано начального значения поля, в котором хранится год рождения, таким образом, это поле будет иметь начальное значение 2001, заданное в описании типа `Stud`.

Так как имя поля структуры локализовано внутри этой структуры, то для доступа к этому полю, как и в Паскале, необходимо указывать *селектор* поля в виде точки, например:

```
mov ax,St2.Year;          ax:=1998
mov ax,type St2.Year;     ax:=2
mov al,St1.Day;           al:=21
mov ax,type St2.Day;      ax:=1
mov ebx,sizeof Stud;      ebx:=25
mov ax,Grup[2*ebx].Year; ax:=Grup[3].Year=2001
```

---

[1] В некоторых языках программирования высокого уровня (например, в языке Free Pascal) начальные значения для переменных сложных (составных) типов тоже задаются в фигурных скобках.

Каждое *имя поля* в структуре имеет для Ассемблера целочисленное значение, которое равно смещению в байтах начала этого поля от начала структуры. Так, например, имя поля `Month` в описанной выше структуре `Stud` имеет значение 22. Таким образом, команда

```
mov al,St1.Month;        al:=4
```

эквивалентно команде

```
mov al,byte ptr St1+22; al:=4
```

Труднее организовать доступ к именованным полям *безымянных* структур, например

```
mov ebx,offset St2
mov al,[ebx].Day; ОШИБКА, имя Day не виднó !
```

Для обращения к полям безымянной структуры нужно дополнительно задавать имя типа структуры:

```
mov al,[ebx].Stud.Day; имя Day из структуры Stud
```

В качестве альтернативы имя структуры можно *приписать* базовому регистру, для этого в Ассемблере предусмотрена директива **assume**:

```
mov ebx,offset St2
assume ebx:ptr Stud
mov al,[ebx].Day;     al=31
```

Область действия директивы **assume** распространяется вниз до следующей директивы **assume** с таким же регистром, или до директивы отмены, которая записывается в виде

```
assume ebx:nothing
```
[1]

Ещё один способ доступа к полю безымянной структуры заключается в явном указании типа операнда:

```
mov al,(Stud ptr [ebx]).Day; al=31
```

Круглые скобки здесь необходимы, так как оператор **.** (точка) имеет для компилятора Ассемблера более высокий приоритет, чем оператор **ptr** (приоритет операций в адресных выражениях Ассемблера приводится в главе, посвящённой макросредствам).

В качестве небольшого примера рассмотрим фрагмент программы на Ассемблере, который выводит (в одну строку) информацию о студенте `St1`:

```
outword St1.Day,3,offset St1.Fio
outword St1.Month,,"/"
outword St1.Year,,"/"; Иванов И.И.     21/4/2001
```

В качестве ещё одного примера напишем на Ассемблере функцию со стандартным соглашением о связях, которая получает в качестве параметров массив `KURS` структур с данными о студентах, длину этого массива `N` и беззнаковое целое число `Y`. Функция будет вырабатывать в качестве своего значения число студентов, родившихся в году `Y`. Соответствующие данные в Ассемблере можно, например, описать так (предполагаем, что структура `Stud` уже описана):

```
.data
N    equ  500
KURS Stud N dup (<>)
Y    dw   ?
```

Тогда вызов нашей функции (дадим ей, например, имя `F`) будет производиться командами:

```
push dword ptr Y; 3-й параметр
push N;           2-й параметр
push offset KURS; 1-й параметр
call F
```

На рис. 8.9 приведён вид полного стекового кадра этой функции, видно, что число `Y` формата **dw** занимает *первую* (из-за перевёрнутого представления чисел в памяти) половину двойного слова стека, вторая половина имеет неопределённое значение (для приведённого выше вызова это значение из секции данных с адресом `Y+2`).

---

[1] Директива **assume** может и вообще *запретить* использование некоторого регистра далее по тексту программы. Например, `assume ebx:error` запрещает использование регистра `EBX` далее до конца программы или до другой директивы **assume** с этим же регистром.

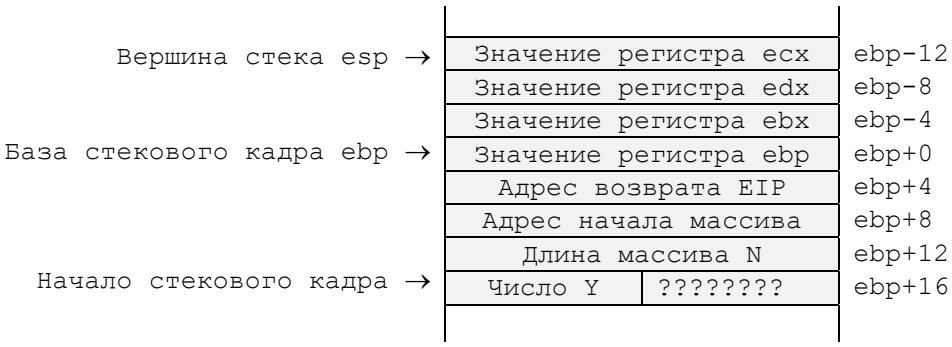


Рис. 8.9. Вид стекового кадра функции F (справа показаны смещения слов кадра относительно значения регистра EBP).

Предполагается, что не каждый элемент массива представляет "настоящего" студента, у отсутствующих фамилия начинается со звёздочки. Далее текст этой процедуры.

```
F  proc
   push  ebp
   mov   ebp,esp
   push  ebx
   push  edx
   push  ecx
   mov   dx,[ebp+16];   Год Y
   mov   ecx,[ebp+12];  Длина массива N
   mov   ebx,[ebp+8];   Адрес начала массива
   mov   eax,-1;        Если на курсе нет студентов
   jecxz Vozv;          Всех студентов отчислили? 🙁
   xor   eax,eax;       Число студентов с г.рожд. Y
   assume ebx:ptr Stud
L: cmp   byte ptr [ebx].Fio,'*'
   je    L1
   cmp   [ebx].Year,dx
   jne   L1
   inc   eax
L1:add   ebx,type Stud; На след. студента в массиве
   loop  L
   assume ebx:nothing
Vozv:
   pop   ecx
   pop   edx
   pop   ebx
   pop   ebp
   ret   12
P  endp
```

Из этого примера видно, что значением одноместного оператора **type**, применённого к имени типа структуры (как, впрочем, и к имени самой переменной этого типа) является длина структуры в байтах. Для нашего примера **type** Stud = **type** KURS = **type** St1 = 25. Здесь следует подчеркнуть, что наша функция, конечно, не знает, что переданный ей массив программист назвал именем KURS, для неё это *безымянный* массив. В то же время для правильной работы функция обязана знать имя типа Stud, который имеют (безымянные) элементы массива, а также должна знать имена всех полей структуры, с которыми она работает. Стоит отметить, что это же будет справедливо и для аналогичной функции, написанной на Паскале.

8.6. Объединения на Ассемблере

Если объединяются слабые, то они становятся сильными. Если объединяются сильные, то они становятся непобедимыми.

Георг Вильгельм Фридрих Гегель

Прежде, чем объединяться, и для того, чтобы объединиться, мы должны сначала решительно и определённо размежеваться.

В.И. Ленин

Объединение (union) является типом данных, состоящим из нескольких переменных, которые хранятся, начиная с одного и того же адреса памяти (перекрывая друг друга). Этот тип данных есть во многих языках программирования, например, в языке C. Некоторым аналогом объединения в Паскале являются записи с вариантами. Пример на Ассемблере:

```
pdate record d:5,m:4,y:7=4
udate struc
  d   db ?;  локальное имя!
  m   db ?
  y   dw 4
udate ends
date union
  Dp  pdate <>;      2 байта
  Du  udate <>;      4 байта
  Z   dw 3 dup (?); 6 байт
date ends
.data
MyDate date <>
.code
   mov ax,MyDate.Dp; 2 байта
   and ax,d; d - глобальное имя из pdate
   mov al,MyDate.Du.d
   mov ax,MyDate.Z; ax:=Z[0]
   outword sizeof MyDate; sizeof MyDate=6
```

Имена полей в объединении, как и имена полей в структурах, локализованы внутри объединения. Объединения используются в основном для экономии места в памяти для хранения данных. На рис. 8.10. показано расположение полей объединения с именем `MyDate` в памяти.

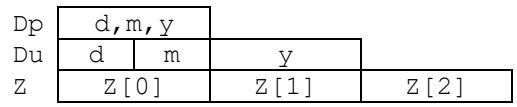


Рис. 8.10. Наложение полей объединения друг на друга.

На этом завершается знакомство с дополнительными возможностями языка Ассемблер, более подробно эту тему необходимо изучить по учебникам [5-7].

Вопросы и упражнения

1. Что такое флаг направления ?
2. Напишите фрагмент программы на Ассемблере, который выводит на печать (по макрокоманде **outword**) текущее значение флага направления (знать номер этого флага в регистре флагов для этого не нужно).
3. Почему необходимо уметь копировать массивы как в прямом, так и в обратном направлении ?
4. Определите, какие значения должен иметь флаг направления `DF` при операции вставки и при операции удаления участка редактируемого текста в некотором текстовом редакторе, который использует для этих целей строковые команды.
5. Что является операндами логических команд ?
6. Когда можно, а когда нельзя использовать команды сдвига для умножения и деления целых чисел на степень двойки ?
7. В каком случае следует использовать в программе упакованные структуры данных ?
8. Каковы области видимости имён полей в упакованных битовых полях, структурах и объединениях ?
9. Для описания упакованного массива данных

```
N     equ 1000000
.data
; var A: bitpacked array[1..N] of 0..15;
```

```
A  db  N/2 dup (?); N/2 ≡ N div 2
X  db  ?
```

Реализуйте оператор присваивания `A[i]:=X`.

---

[i] На самом деле **nop** – это однобайтная команда **`xchg eax,eax`** с кодом операции `90h`, которая "ничего не делает" и не меняет флагов. Эта команда может использоваться при резервирования "пустого" места в секции кода. Например, это можно делать для выравнивания адресов меток и процедур на границу 16 байт (только если метка находится далеко от команды перехода, при этом лучше работает предсказание ветвлений, о чём говорится в другой главе). Команды **nop** можно использовать (в "хитрых" самомодифицирующихся программах) для "затирания" ненужных команд во время счёта (если, конечно, секция кода открыта на запись). Современные процессоры выполняют до 4-х подряд расположенных **nop** за один такт. Для увеличения времени выполнения, перед командой **nop** можно поставить несколько префиксов `66h`, на каждый префикс процессор будет тратить один лишний такт.

Кроме того, в языке машины есть и другие команды (длиной от 2 до 15 байт), которые тоже "ничего не делают" и выполняются за один такт, их использование позволяет сократить время выполнения цепочки "пустых" команд, например:

```
xchg eax,eax;                (1 байт)  90h
mov  eax,eax;                (2 байта) 89C0h
lea  eax,[eax+00h];          (3 байта) 8d4000h
lea  eax,[eax+1];            (4 байта) 8d742600h
lea  eax,[eax+00000000h];    (6 байт)  и т.д.
```

Но здесь надо учитывать, что только однобайтный **nop** не создаёт так называемую зависимость по данным и не тормозит конвейер ЭВМ, о чём будет говориться в другой главе. Заметим, однако, что команда с префиксом **`rep nop`** (`F390h`, приходится задавать в MASM 6.14 как **db** `0F3h,90h`, иначе Ассемблер "не понимает") трактуется как команда **pause**. Эта команда может использоваться в так называемых циклах ожидания (spin wait loop) некоторого события, она оптимизирует (менее загружает) конвейер в таком цикле.

Любопытно, что существует и "вещественная" пустая команда **fnop** (`D9D0h`), которая "на самом деле" команда **`fld st(0)`**, выполняющая операцию `st(0):=st(0)`, т.е. копирование вершины стека на себя.

К сожалению, в 64-битном режиме команда **nop** перестала быть однобайтной `90h` и стала "настоящей" пустой командой `86C0h`. Так произошло из-та специфики выполнения команды **`xchg eax,eax`**, которая в 64-битном режиме дополнительно обнуляет старшую часть регистра `RAX`.

[ii] Такое значительное уменьшение сложности алгоритма пересылки массива при использовании цепочечных команд верно только для *младших* моделей нашего семейства. В старших моделях появилась специальная быстродействующая область памяти – кэш, в которую *автоматически* помещаются, в частности, последние выполняемые команды. Таким образом, весь наш первоначальный цикл пересылки из 5-ти команд будет находиться в этой кэш-памяти, скорость чтения из которой в несколько раз больше скорости чтения из оперативной памяти. Другими словами, обращения в относительно медленную оперативную память *за командами* при выполнении цикла пересылки массива в старших моделях тоже производиться не будет (команды цикла будут читаться из кэш-памяти), и сложность алгоритма также приближается к `2*N` обменов. Более того, современные компьютеры за одно обращение к памяти читают не один байт, а несколько (обычно 8 или 16) подряд расположенных байт. Таким образом, получается, что сложность нашего алгоритма падает уже до `N/4` обменов. Ещё быстрее копирование памяти производится с помощью регистров `YMM` (длиной 256 бит) и `ZMM` (длиной 512 бит)

[iii] В качестве "бонуса", у логических команд допускается сокращённая форма *непосредственного* второго операнда. Например, команда **`or ebx,-1`** будет иметь не формат `r32,i32`, а формат `r32,i8` (`83 CB FF`), т.е. операнд `-1` будет занимать в команде не 4, а только 1 байт. При выполнении таких команд над вторым операндом предварительно выполняется знаковое расширение `i8` (sign extended imm8), например, команда из нашего примера выполнится по правилу `ebx:=ebx` **or** `Longint(-1)`. Это одно из немногих исключений из общего правила, что операнды команды должны иметь одинаковую длину.

[iv] В математике такое округление при делении нечётных чисел называется округлением в меньшую сторону (к *отрицательной бесконечности* – round half down), в Паскале это `Round`, а машинная команда деления **idiv** производит округление *к нулю* (round half toward zero), в Паскале это `Trunc`. Возможны также округление в большую сторону (к *положительной бесконечности* – round half from zero) и округление к *ближайшему чётному* (odd–even rounding – младший бит сбрасывается).

Любопытно отметить, что для работы с *вещественными* числами для мантиссы обычно используется округление `Round`, но в двухбитовом поле `RC` (Round Control field) регистра управления сопроцессором `CR` (Control Register) можно задать (с помощью команды **`fldcw`**) любое из этих правил преобразования вещественного числа в целое. Вообще говоря, самым точным считается округление к ближайшему чётному, его часто называют банковским округлением (banker's rounding), этот режим используется по умолчанию. Например, при округлении вещественных двоичных чисел `X=+1.0111` и `X=-1.0111` до трёх цифр в дробной части получаются такие результаты:

Метод округления	**+1.0111**	**-1.0111**
К ближайшему чётному	+1.100	-1.100
В меньшую сторону	+1.011	-1.100
В бóльшую сторону	+1.100	-1.011
К нулю	+1.011	-1.011

При работе с вещественными числами на векторных регистрах режим округления устанавливается командой **`ldmxcsr`**.

---

[v] В процессоре Intel 386 появились также команды сдвига **`shld`** (**`sh`**ift **`l`**eft **`d`**ouble – сдвиг влево удвоенный) и **`shrd`** (**`sh`**ift **`r`**ight **`d`**ouble – сдвиг вправо удвоенный), у которых три операнда, эти команды имеют вид:

`shld` `op1,op2,op3`
`shrd` `op1,op2,op3`

Ниже приведены допустимые операнды этих команд:

op1	**op2**	**op3**
`r16, m16`	`r16`	`cl, i8`
`r32, m32`	`r32`	`cl, i8`

У этих команд величина сдвига задается в третьем операнде, а вправо или влево на нужное количество разрядов логически сдвигается первый операнд. При сдвиге влево освобождающиеся биты первого операнда заполняются *старшими* битами второго операнда, а при сдвиге вправо освобождающиеся биты первого операнда заполняются *младшими* битами второго операнда. Таким образом, в сдвиге участвуют значения *двух* операндов, но второй операнд при этом *не меняется*. Обычно эти команды используются при обработке растровых изображений, при шифровании и др.

У команд сдвига **`shl`**, **`shr`** и **`sar`** появились трёхоперандные модификации с кодами **`shlx`**, **`shrx`** и **`sarx`** сответственно:

`КОП` `op1,op2,op3; op1:=op2` **`shift`** `op3[4..0]`

Эта команда удобна тем, что второй операнд *не меняется*. В отличие от обычных сдвигов, никакие флаги не меняются. Ниже приведены допустимые операнды этих команд:

op1	**op2**	**op3**
`r32`	`r32,m32`	`r32`

В процессорах, начиная с Intel 386 появились новые логические команды для работы с отдельными битами. Команды

`bsf` `op1,op2` и **`bsr`** `op1,op2`

Ниже приведены допустимые операнды таких команд:

op1	**op2**
`r16`	`r16, m16`
`r32`	`r32, m32`

Эти команды производят поиск первого ненулевого бита второго операнда справа налево, от 0-го бита к 15-му или 31-му биту (команда **`bsf`**) или слева направо (команда **`bsr`**), и помещают номер этого бита в первый операнд, при этом флаг `ZF:=0`, остальные флаги не определены. Если же единичный бит не найден, то `ZF:=1`, а значение первого операнда не определено. Например, после команды **`bsf`** `eax,1000b` в регистр `EAX` запишется число 3 и будет `ZF=0`. Как и для сдвигов, операнды в памяти рассматриваются в *не перевёрнутом* виде.

Команда

`bt` `op1,op2`

(**`b`**it **`t`**est – тестирование бита) с такими допустимыми операндами

op1	**op2**
`r16, m16`	`r16, i8`

r32, m32	r32, i8

Эта команда копирует в CF бит из первого операнда, номер которого задан во втором операнде. Когда первый операнд – регистр, то биты в нём, как обычно, нумеруются с нуля (справа налево), а из второго операнда для номера берутся только 4 или 5 последних бит (в зависимости от длины первого операнда). Для первого операнда в памяти отсчёт позиции бит идёт от нулевого бита в байте по заданному адресу (m16 или m32), а номер позиции бита является знаковым числом в максимальном диапазоне от -2^{31} до $+2^{31}-1$ (для op2=r32). Таким образом, несмотря на "кажущийся" формат первого операнда m16 или m32, на самом деле он является битовой строкой длиной до $2^{32}:8=2^{24}$ байт (16 Мб), но из этой строки в процессор читаются только 2 или 4 байта с нужным битом. После выполнения команды все флаги, кроме CF, не определены.

Модификации этой команды с кодами операций **btc** (**b**it **t**est and **c**omplement – тестирование бита с инверсией), **btr** (**b**it **t**est and **r**eset – тестирование бита со сбросом) и **bts** (**b**it **t**est and **s**et – тестирование бита с установкой) дополнительно соответственно инвертируют, очищают или устанавливают перенесённый в CF бит первого операнда. Наряду с командой **xchg** это тоже *атомарные* команды, они читают старое значение некоторого бита во флаг CF и *сразу* (т.е. без освобождения шины обмена с памятью) записывают в этот же бит новое значение. Например, приведённый ниже фрагмент сбрасывает в ноль первый слева ненулевой бит в регистре EAX:

```
      bsr eax,ecx; ecx:=№ первой "1"
      jz  @F;      нет "1"
      btr eax,ecx; сброс "1"
@@:
```

В новых процессорах есть команды для работы с *битовыми масками*. Например, команда

blsmsk op1,op2

(**b**it **l**owest **s**et **m**a**sk** – получение маски до старшего единичного бита) с такими допустимыми операндами

op1	op2
r32	r32, m32

Эта команда устанавливает op1[n..0]:="1", где n – младший бит со значением "1" в op2. Если op2=0, то op1=-1 (все"1"), при этом CF:=1, иначе CF:=0. Естественно, всегда ZF:=0 и OF:=0. Например:

```
mov     ebx,100h
blsmsk eax,ebx; eax:=1FFh
```

Видно, что описывать работу команды на естественном языке сложно, поэтому в документации часто используется формальное описание выполнения команды, например для **blsmsk**:

```
temp := (op2-1) xor op2;  во как !
SF := temp[OperandSize -1]
ZF := 0; OF := 0
if op2=0
   CF := 1
else
   CF = 0
endif
op1 := temp
```

Команды

blsi op1,op2 и **blsr** op1,op2

(**b**it **l**owest **s**et **i**solated – извлечь последний единичный бит) и (**b**it **l**owest **s**et **r**eset – обнулить последний единичный бит) с такими допустимыми операндами

op1	op2
r32	r32, m32

Эти команды реализуют операции op1:=(-op2) **and** op2 и op1:=(op2-1) **and** op2 соответственно. Первая команда оставляет в op1 только один единичный бит, соответствующий последнему единичному биту op2, а вторая команда устанавливает в ноль последний единичный бит op2. Если op2=0, то op1:=0, при этом CF:=1, иначе CF:=0. Флаги ZF и SF устанавливаются так же, как и для операции сложения, а OF:=0. Например:

```
mov  ebx,0FFFF1000h
blsi eax,ebx; eax:=1000h
blsr eax,ebx; eax:=0FFFF0000h
```

Команда

lzcnt op1,op2 и **tzcnt** op1,op2

(**l**eading/**t**railing **z**ero **cou**n**t** – подсчитать число лидирующих/концевых нулевых бит) с такими допустимыми операндами

op1	op2
r16	r16, m16
r32	r32, m32

op1:=число лидирующих/концевых нулевых бит (op2). Для op2=0 op1:=длина (op2) , при этом CF:=1, иначе CF:=0, остальные флаги не определены.