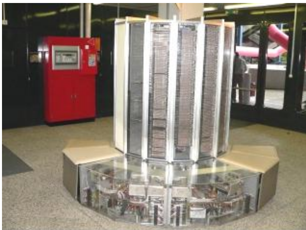
17. Работа с векторными регистрами

Человек, который почувствовал ветер перемен, должен строить не щит от ветра, а ветряную мельницу.

Мао Цзэдун

Вектора́, матрицы и строки символов широко используются в задачах обработки больших объёмов данных. При этом в математике широко используются соответствующие операции, с точки зрения программиста соответствующие им машинные команды имеют формат SIMD, т.е. одна команды обрабатывает сразу много данных. Таким образом, использование векторных регистров является естественным механизмом обработки таких данных,[1] и многие алгоритмы могут быть, как говорится, "векторизованы".

Разумеется, современные оптимизирующие компиляторы с языков высокого уровня сами векторизуют Ваш алгоритм (если им это разрешено), причём сделают это, скорее всего лучше, чем программист на Ассемблере. Однако, если программист хотя бы примерно понимает, как это происходит, он может написать эквивалентный алгоритм, который будет векторизоваться лучше! Отсюда следует практическая необходимость понимания выполнения векторных операций на уровне языка машины, даже для прикладного программиста на языках высокого уровня.



Первая ЭВМ Cray-1 с векторными регистрами

Впервые векторные регистры появились на ЭВМ Cray-1 в 1975 году, потом эту машину стали называть первой супер-ЭВМ, у неё была производительность 133 Мегафлопа. В этой ЭВМ было 8 особых векторных регистров, на каждом из которых мог располагаться вектор из 64-х вещественных чисел длиной по 64 бита. Правда, эти векторные регистры располагались не в самом процессоре, а в "соседнем" блоке быстродействующей статической памяти, но в языке машины была реализована адресация регистр-регистр. Для повышения эффективности в каждой команде можно было задавать длину обрабатываемого вектора (от 1 до 64 элементов).

Итак, мы уже говорили, что, кроме регистров общего назначения и стека регистров для работы с вещественными числами, в нашей машине есть ещё и векторные регистры. Они, как и регистры общего назначения, строятся по иерархическому принципу. Так, 128-разрядные регистры `XMM` являются младшими частями 256-разрядных регистров `YMM` (`YMM0-YMM15`) [2] которые, в свою очередь, можно рассматривать как младшие части 512-разрядных регистров `ZMM`. [3]

Использовать младшие части векторных регистров в качестве "самостоятельных" переменных программисту надо с осторожностью. Дело в том, что при любом изменении, например, какого-либо регистра `XMM` автоматически меняется (чаще всего просто обнуляется) и старшая часть соответствующего регистра `YMM` и `ZMM` [i] [см. сноску в конце главы].

Это вообще характерно для регистров современных ЭВМ. Скажем, при изменении регистра `EAX` автоматически меняется и старшая часть соответствующего регистра `RAX`. Отметим, что для младших моделей нашего семейства это не так, например, при изменении регистра `AX` старшая часть регистра `EAX` не меняется.

Интересно, что, как и вещественные регистры `st(0)-st(7)`, векторные регистры могут быть "пустыми", не содержащими чисел, читать из таких регистров нельзя. Это обеспечивается тем, что, как и для вещесвенных регистров, с каждым векторным регистров связан тэг, определяющий содержимое этого регистра.

---

[1] Другим способом параллельной обработки больших объёмов данных является так называемая массово-параллелльная обработка, когда на каждый элемент выделяется своё процессорное ядро. Это типичный способ работы графических процессоров, когда, скажем, каждый пиксель (или группа пикселей) изображения обрабатывается своим процессорным ядром графической карты.

[2] В 32-битном режиме доступны только первые 8 из них: `YMM0-YMM7`, а в старших (AVX-512) моделях уже 32 `YMM0-YMM31`.

[3] Длинные `ZMM` регистры реализуются не во всех ЭВМ семейства, в основном, в высокопроизводительных серверах, в наших примерах эти регистры мы использовать не будем.

Регистры YMM6-YMM15 считаются *постоянными*, по соглашению о связях, если они меняются в подпрограммах, то их надо запомнить, а потом восстановить.

За одно обращение эти регистры обмениваются с памятью 128, 256 или 512 байтами, соответственно. При обмене данными между памятью и векторными регистрами эти данные практически всегда должны быть выровнены (boundary) на границу 16 (для XMM регистров), 32 (для YMM регистров) и 64 (для ZMM регистров) байтов. Выравнивание позволяет более эффективно использовать кэш-память, строки (линии) которой выровнены на границу 32 байтов.[1]

Каждая секция программы (команд, данных, стека) загружается в память с новой страницы, т.е. выровнена на границу 4096 байт. Для выравнивания внутри секций может использоваться директива **align** n, где n∈{1,2,4,8,16}, но нам нужно выравнивание на 32 или даже 64 (для ZMM регистров) байт. К сожалению, **align** 32 у нас работать не будет, так как сокращённая директива сегментации **.data** имеет 16-байтное выравнивание, и по правилам, параметр директивы **align**. не может превышать "глобального" выравнивания всей секции.

Так что для правильного выравнивания данных нам придётся пойти на хитрость. Сначала надо учесть, что в начале секции данных располагаются служебные переменные, которые помещаются туда макрокомандами (**ClrScr**, **outint** и т.д.). Исходя из этого, наши данные мы будем располагать, отступив от начала секции **.data**, например, на 1000h байт с помощью директивы **org**. Эта директива устанавливает счётчик размещения, который обозначается как $ (позицию относительно начала секции) на величину, равную параметру этой директивы. В качестве таких параметров допускаются адресные выражения, значения которых можно вычислить в этом месте программы, например:

```
.data
  org 1000h; $=начало секции+1000h
; ↓↓ адрес A выровнен на 32 байта
A db 16 dup (?); A[0..15] of char;
; теперь выравнивание только на 16 байт
; ↓ ↓ $=A+32=начало секции+1020h
  org 1020h
; ↓↓ адрес B выровнен на 32 байта
B dd 8 dup (?); B[0..7] of longword;
C db  ?; какая-то однобайтная переменная
  org 1060h; $=начало секции+B+64
; ↓↓ адрес D выровнен на 64 байта
D real8 8 dup (?); D[0..7] of double;
```

Векторные регистры могут хранить привычные для нас логические данные в виде 128-, 256- или 512-битовых векторов. Кроме того, на каждом векторном регистре можно хранить и *упакованные векторы* целых или вещественных чисел. Так, на YMM регистре помещаются 4 вещественных числа типа double или 8 чисел типа single. При работе с целыми числами (знаковыми и беззнаковыми) на этот регистры помещается сразу по 4 числа типа int64/qword, по 8 чисел типа longint/longword, 16 чисел типа smallint/word или 32 числа типа shortint/byte. Операции (сложение, вычитание, умножения и т.д.) могут выполняться сразу над всеми элементами двух векторов параллельно. Таким образом, только сам программист, задавая соответствующие команды, *приказывает* трактовать данные на векторном регистре как логические или же вектора целых или вещественных чисел.

Векторные регистры создавались как универсальные. На регистр общего назначения можно записать любые данные, но выполнять на нём можно только целочисленные и логические операции. Аналогично, на регистры st(0)-st(7) можно загрузить что угодно, но выполнять операции только над вещественными числами (правда, можно загружать целые числа с их автоматическим преобразованием в вещественные и наоборот). В противовес этому над содержимым векторных регистров можно выполнять как логические операции, так и операции целочисленной и вещественной арифметики, причём с разными длинами упакованных операндов. На современных ЭВМ вычисления на "обычных" вещест-

[1] Для обмена между векторными регистрами и невыровненными данными в памяти есть аналогичные команды, (мы их приведём), но они примерно на 80% медленнее.

венных регистрах `st(0)-st(7)` создают слишком много зависимостей по данным и плохо выполняются на конвейерах (см. разд. 14.2), поэтому предпочтительнее использовать векторные регистры. В то же время без вещественных регистров не обойтись, если нужна повышенная (80-битная) точность вычислений.

При работе векторные регистры потребляют много энергии, нагревая ядро процессора, поэтому обычно они выключены (на них не подаётся электрическое питание, и не приходят тактовые импульсы). Современные процессоры просматривают программу вперёд (по предполагаемому пути её выполнения) примерно на 4 Кб, это около 1000 команд, 300-500 тактов работы или 0.1-0.3 мкс (см. разд. 14.2). Ядро "включает" векторные регистры, когда при таком просмотре встречает векторные команды. И, соответственно, когда в программе долго не встречаются векторные команды, ядро "выключает" свои векторные регистры.

К сожалению, "включение" векторных регистров занимает около 10000 тактов. Сначала ядро обращается к расположенному на кристалле микроконтроллеру PCU (Package Control Unit) с просьбой выдать "лицензию" на дополнительное электрическое питание. Для простых векторных команд (загрузка, сложение) ядро запрашивает минимальную лицензию LVL0, для более сложных лицензию LVL1, а для 512-битных операций вообще лицензию LVL2 (Turbo License). При этом ядро "понимает", что после получения такой лицензии его максимальная тактовая частота, скорее всего, будет снижена на 15-30% 🐷.

Микроконтроллер PCU является общим "начальником", который распоряжается распределением электрического питания и следит за перегревом различных устройств на кристалле (где разбросано несколько десятков термометров). PCU может зафиксировать опасное повышение температуры (обычно это около 100 градусов) на некотором устройстве (например, на блоке векторных регистров ядра). Когда такое происходит, PCU включает для этого устройства так называемый режим троттлинга (throttling), заставляя его "отдыхать" и пропускать процессорные такты. Например, устройство может пропускать каждый третий или четвёртый такт. При сбое охлаждения (сломался вентилятор) производится немедленное отключение всего процессора.

Пока питание блока векторных регистров не включилось (а программу-то считать надо ☹), ядро будет *эмулировать* векторные операции по *микропрограммам* с помощью "обычных" 64-битных регистров (что значительно медленнее). Отсюда понятно, что использовать в программе "одиночные" векторные команды бессмысленно, они будут эмулироваться на "обычных" регистрах. Опасно также сильно загружать векторными операциями много ядер процессора. Программист (а, скорее, компилятор с языка высокого уровня) может "помочь" процессору, вставив в программу фиктивные векторные команды за некоторое время до интенсивного использования векторных вычислений.

Большинство современных компиляторов с языков высокого уровня широко используют векторные регистры для обработки данных в программах пользователей. Сам программист обычно описывает свои данные в виде векторов и матриц и записывает их обработку в виде циклов по элементам, на основании чего компилятор производит, как говорят, авто векторизацию алгоритма. Обычно можно попросить компилятор выдать отчёт о проведённой авто векторизации программы, чтобы выявить проблемы с повышением эффективности.

На 32-битных Ассемблерах MASM (до версии 6.15 включительно) доступны только векторные 128-разрядные регистры `XMM`, для их использования в начало программы надо вставить дополнительную директиву

```
.XMM
```

17.1. Команды пересылок

Сутью искусства программирования обычно считается умение составлять операции. Но не менее важно умение составлять данные.

Никлаус Вирт

Как всегда, начнём с команд пересылок, в их код операции, как обычно, входит слово **mov**. Сначала рассмотрим пересылку просто битовых последовательностей соответствующей длины. При этом нам, вообще говоря, не важна структура этих данных (длинные или короткие, целые или вещественные числа и т.д.), существенен только размер данных, который определяется длиной регистра:

```
vmovdqa op1,op2; op1:=op2
```

Первая буква **v** означает, что операнды являются векторными регистрами.

Последняя буква кода операции пересылки **a** (**a**lign), означает, что операнды *в памяти* должны быть выровнены на "естественную" границу (8, 16 или 32 байта). Именно для этого при описании переменных в секции **.data** мы применяли директиву **org**. Заметим, что аналогичные команды с кодами операций **vmovdqu** и **vmovup**{**s**,**d**} (**u**nalign) выравнивания не требуют. При не выровненном обмене, однако, часто приходится "беспокоить" две соседние линейки кеш памяти, что снижает производительность. Рекомендуется привыкать выравнивать векторные данные в секции данных и использовать команду **vmovdqa**, так как при обращении в стек (это тоже секция памяти!) за не выровненными данными обе эти команды могут давать ошибку.

Когда операндами векторных команд являются регистры XMM букву **v** можно опускать, для многих команд это приводит к тому, что старшая часть соответствующего YMM (и ZMM) регистра не обнуляется.

Буквы **dq** задают размер операнда (**d**ouble **q**uadword, в нашем Ассемблере это тип **oword** – **o**cta **w**ord). Подчеркнём, что для регистра-получателя XMM всегда изменяется 256 бит, так как при этом обнуляется старшая часть соответствующего регистра YMM. Допустимые форматы операндов:

op1	op2
r128	r128, m128
m128	r128
r256	r256, m256
m256	r256

Заметим, что по сравнению с регистрами общего назначения, при работе с векторными регистрами отсутствует формат данных в виде непосредственного операнда i128 и i256. Это легко понять, так как для таких операндов длиной 32 байта команды станут очень длинными. Напомним, что в нашей архитектуре длина команды не может превышать 15-ти байт.

У данной команды есть синонимы с мнемокодом **vmovap**{**s**,**d**}, используя их, программист считает, что он загружает на регистр вектор *вещественных* чисел типов **s** (**s**ingle) или **d** (**d**ouble). В то же время, используя команду **vmovdqa**, программист подчёркивает, что загружаются логические, целочисленные или строковые данные. На логическом уровне (как говорят, на уровне макроархитектуры), это, разумеется, не влияет на выполнение команды пересылки. А вот ответ на вопрос, зачем сделаны такие команды-синонимы, достаточно сложен [ii] [см. сноску в конце главы].

Важно отметить, что, как и целые числа, вектора при чтении на векторные регисты "переворачиваются", поэтому элементы вектора на векторных регистрах индексируются от нуля и справа налево. Пусть, например, массив в языке Free Pascal описан как **var** X[0..3] **of double**; или в Ассемблере как X **dq** 4 **dup** (?). После загрузки вектора X, скажем, на регистр YMM0, этот регистр будем выглядеть как

ymm0[3..0]=X[3],X[2],X[1],X[0]

Отсюда видно, что ymm0[i]=X[i], поэтому мы не будем обращать внимание на "перевёрнутую" нумерацию в регистре и писать ymm0[0..3]:=X[0..3].

Пример команд пересылок:

```
.data
  org 1000h
A oword 100 dup (0)
B oword ?,?
C dq  1,2,3,4; C[0..3] of int64
.code
; m256=32*db,16*dw,8*dd,4*dq
  vmovdqa ymm0,B; ymm0:=B; формат r256,m256
; m256=32*db,16*dw,8*dd,4*dq
; 8*dd=8*Single=8*Longint=8*Longword
; 4*dq=4*Double=4*int64=4*qword и т.д.
  vmovdqa ymm0,ymmword ptr C
; ↑ ↑ ymm0[0..3]:=C[0..3] of dq={4,3,2,1}
```

```
; вектор C переворачивается при загрузке на регистр!
    vmovdqa ymm0,ymm1
; напоминаем, что при загрузке в младшую часть регистра
; ymm его старшая часть обнуляется
    vmovdqa xmm0,xmmword ptr C; сначала ymm0:=0!
```

В языке машины есть ещё специфическая команда пересылки с кодом операции **vmovdntqa**. Она идентична команде **vmovdqa**, однако обмен с память делается, минуя кеш. Например, при чтении данные передаются из оперативной памяти прямо на регистр, и копия этих данных *не оставляется* в кеш памяти. Впрочем, если эти данные в кеше уже были, то они там модифицируются. При записи данные с регистра передаются прямо в оперативную память, и копия этих данных *не оставляется* в кеш памяти (конечно, если они там были, то эти данные объявляются в кеше недействительными).

Буквы **nt** в коде операции означают Non-Temporal, т.е. это данные "не временны́е" (не обладающие временно́й локальностью), они в ближайшее время процессору не понадобятся, и пусть они лучше не "засоряют" кеш. Кроме того, запись новых данных в кеш требует сложной проверки, что эти данные не находятся в кеш памяти других процессорных ядер. Типичным примером является заполнение буфера в оперативной памяти для вывода данных на внешнее запоминающее устройство.

Стоит отметить и "широковещательную" операцию, которая пересылает *одно* значение во *все* позиции векторного регистра, сначала рассмотрим рассылку целого значения:

```
    vpbroadcast{b,w,d,q} op1,op2; op1[N-1..0]:=op2
```

Буква **p** (**p**acked) в коде операции означает *упакованный* вектор, **broadcast** переводится как "передавать" или "распространять". Буквы {**b**,**w**,**d**,**q**} задают длину вектора и тип его элемента, который совпадает с типом второго параметра: **b** → **byte** (N=32), **w** → **word** (N=16), **d** → **dword** (N=8) и **q** → **qword** (N=4). Допустимые форматы операндов:

op1	op2
r128, r256	m8, m16, m32, m64, r128

Например:

```
.data
  org 1000h
X db  ?
  org X+32
 Y dw  ?
.code
  vpbroadcastb ymm0,X; ymm0[0..31]:=X
  vpbroadcastw ymm0,Y; ymm0[0..15]:=Y
  vpbroadcastd ymm0,xmm1; ymm0[0..7]:=xmm1[0]
```

Аналогичная команда есть и для рассылки вещественного значения во все позиции векторного регистра:

```
    vbroadcast{ss,sd,f128} op1,op2; op1[0..N-1]:=op2
```

Буквы в конце кода операции задают длину вектора и тип его элемента, который совпадает с типом второго параметра: **ss** (**s**calar **s**ingle, N=8), **sd** (**s**calar **d**ouble, N=4), **f128** → **m128** (**m**emory 128 бит, N=2). Допустимые форматы операндов:

op1	op2
r128, r256	m32, m64, m128, r128

Далее следуют команды пересылки, *расширяющие* свои операнды. Например, мы уже знаем команды **movsx** и **movzx**. Для векторных регистров тоже предусмотрены аналогичные команды:

```
    vpmov{s,z}x{bw,bd,bq,wd,wq,dq} op1,op2
```

Буквы **s** и **z** задают соответственно знаковое и беззнаковое расширение элементов операнда op2 в элементы операнда op1. В конце кода операции можно задавать буквы для разных типов исходного и расширенного операндов: **bw** это **byte** → **word**, **wd** это **word** → **dword**, **bq** это **byte** → **qword** и т.д. Допустимые форматы операндов:

op1	op2
r128	m16, m32, m64, m128, r128
r256	m32, m64, m128, r128

Скажем, для команды с операндами форматов `r256,m64` элементы вектора `op2` будут расширены в `256/64=4` раза, следовательно, в коде операции будут буквы **bd** или **wq**. Например:

```
.data
  org 1000h
X dw 8 dup (?); X[0..7] of smallint;
  org X+32
Y db 4 dup (?); Y[0..3] of byte;
.code
; ↓↓ 8 слов из X расширяются в 8 двойных слов в ymm0
; ↓↓ ymm0[0..7]:=longint(X[0..7])
  vpmovsxwd ymm0,xmmword ptr X
; ↓↓ 4 байта из Y расширяются в 4 двойных слов в ymm0
; ↓↓ ymm0[0..3]:=qword(Y[0..3])
  vpmovzxbq ymm0,dword ptr Y
; ↓↓ 16 младших байт из ymm1 расширяются в 16 слов в ymm0
  vpmovzxbw ymm0,ymm1; ymm0[7..0]:=dword(ymm1[7..0])
```

А теперь команды <u>пересылки с усечением</u> (упаковкой) *целых* чисел:

vpack{**s**,**u**}**s**{**wb**,**dw**,**qd**} $op1,op2_1,op2_2$

Производится только усечение со знаковым (**s** – sign) или беззнаковым (**u** – unsign) насыщением, обозначим эту операцию как `truncs` [1]. Исходные операнды усекаются всегда наполовину: **wb** это **word**→**byte**, **dw** это **dword**→**word**, **qd** это **qword**→**dword**, поэтому в исходных данных надо задать в два раза больше байт, чем в результате, т.е. приходится задавать <u>два</u> исходных операнда. Допустимые форматы операндов:

op1	$op2_1$	$op2_2$
r128	r128	r128, m128
r256	r256	r256, m256

Алгоритм выполнения команды:

```
{N=4,8(qd), 8,16 (dw), 16,32 (wb); j:=N div 2;}
for i:=0 to j-1 do op1[i]:=truncs(op2_1[i]);
for i:=j to N-1 do op1[i]:=truncs(op2_2[i])
```

Например:

```
.data
  org 1000h
X dq 4 dup (?)
  dq 4 dup (?); X[0..7] of dq;
.code
; ↓↓ ymm1:=X[0..3]
  vmovdqa ymm1,ymmword ptr X
; ↓↓ 8 dq из X → 8 dd ymm0
  vpackssqd ymm0,ymm1,ymmword ptr X+32
```

А вот команда <u>излечения младшей или старшей части</u> YMM регистра и записи результата в XMM регистр (а значит в младшую часть соответствующего YMM регистра) или в память:

vextract{**i**,**f**}**128** op1,op2,op3

Здесь **extract** это извлечь, а **i128** или **f128** это 128-битное целое или вещественное число, `i8[0]` определяет индекс `(0..1)` извлекаемого элемента. Допустимые форматы операндов:

---

[1] Операции над целыми числами с насыщением описываются в разделе 17.2.1. Усекать числа без насыщения смысла не имеет, так как для "хороших" чисел обрезание старшей части с насыщением и без насыщения дадут одинаковый результат, а для "плохих" чисел ответ всё равно будет неправильный.

op1	op2	op3
r128, m128	r256	i8

Алгоритм выполнения команды:

```
{op2:array[0..1] of i128/f128}
j:=i8[0]; {j=0 или 1}
op1:=op2[j]
```

Далее следуют команды, производящие извлечение одного элемента векторного регистра на регистр общего назначения или в память. Эти команды трёхадресные, третий операнд формата i8 задаёт индекс нужного элемента на векторном регистре, т.е. op3=i8∈[0..N-1]:

vpextr{**b**,**w**,**d**,**q**} op1,op2,op3; op1:=op2[op3]

Здесь **pextr** это извлечь из упакованного (**p**) вектора. Размер извлекаемого операнда задаётся буквой в коде операции: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Извлечение производится на регистр общего назначения r32/r64 (в младшую часть регистра с обнуление оставшейся части регистра) или в память **m8**, **m16**, **m32** или **m64**. Допустимые форматы операндов:

op1	op2	op3
r32, r64, m8, m16, m32, m64	r128	i8

Например:

```
.data
  org 1000h
; ↓↓ A=16*db, 8*dw, 4*dd, 2*dq
A oword ?
B db ?
.code
  movdqa xmm0,A; xmm0:=A
; ↓↓ xmm0=16*db, 8*dw, 4*dd, 2*dq
  vpextrb ecx,xmm0,9; cl:=xmm0[9]
  vpextrw edx,xmm0,7; dx:=xmm0[7]
  vpextrd ebx,xmm0,0; ebx:=xmm0[0]
  vpextrq rax,xmm0,1; rax:=xmm0[1]
  vpextrb B,xmm0,14;  B:=xmm0[14]
```

А вот, наоборот, запись (вставка) в один выбранный элемент массива на векторном регистре величины с регистра общего назначения или из памяти:

pinsr{**b**,**w**,**d**,**q**} op1,op2,op3; op1[op3]:=op2 [1]

Здесь **pinsr** это **p**acked **inser**t (вставка в упакованный вектор). Размер вставляемого операнда задаётся буквой в коде операции: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Вставкапроизводится из регистра общего назначения r32/r64 (из младшей части регистра) или из памяти m8, m16, m32 или m64. Допустимые форматы операндов:

op1	op2	op3
r128	r32, r64, m8, m16, m32, m64	i8

Например:

```
.data
  org 1000h
; ↓↓ A=16*db, 8*dw, 4*dd, 2*dq
A oword ?
B db ?
.code
  movdqa xmm0,A; xmm0:=A
; ↓↓ xmm0=16*db, 8*dw, 4*dd, 2*dq
  pinsrb xmm0,ecx,10; xmm0[10]:=cl
  pinsrw xmm0,edx,7;  xmm0[7]:=dx
```

[1] Аналогичные 4-адресные команды с несколько бо́льшими возможностями мы рассматривать не будем.

```
pinsrd xmm0,ebx,0;   xmm0[0]:=ebx
pinsrq xmm0,rax,1;   xmm0[1]:=rax
pinsrb xmm0,B,13;    xmm0[13]:=B
```

Вспомним теперь, что в языке нашей машины есть и (скалярные) команды *условных пересылок*, например:

```
cmovz eax,ebx; if ZF=1 then eax:=ebx
```

Для векторных регистров есть очень ограниченный аналог, это 3-адресные команды условного чтения на регистр или записи в память элемента векторного регистра по маске:

```
vpmaskmov{d,q} op1,op2,op3
```

Буквы **d** и **q** в конце кода операции задают тип элементов векторных регистров: **d** → **dd**, **q** → **dq**. Есть синоним с мнемокодом **vmaskmovp**{**s**,**d**}. Буквы {**d**,**q**} "намекают" на целые типы **dd** и **dq**, а буквы {**s**,**d**} – на вещественные типы **s**ingle и **d**ouble. Вам надо понять, что это именно команды-синонимы, так как при пересылке имеет значение только длина переменных. Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256
m128	r128	r128
m256	r256	r256

Второй операнд здесь задаёт так называемую векторную маску. Когда в элементе маски в крайней левой позиции (т.е. это SF) находится бит со значением "1" это **true**, а со значением "0" это **false**. Алгоритм выполнения команды:

```
for i:=0 to N-1 do {N=32,16,8,4}
  if SF(op2[i]) then op1[i]:=op3[i] else
  if {op1=r128,r256 – регистр}
     then op1[i]:=0
     else {op1=m128,m226 – память,
            тогда op3[i] не меняется}
```

Например:

```
.data
T equ 80000000h; Маска[i]=1 (true)
F equ 00000000h; Маска[i]=0 (false)
  org 1000h
M dd  T,F,F,F,T,T,F,T; Маска[0..7]
A dd 8 dup (?); A[0..7] of dword;
.code
; ↓↓ ymm1:=Маска[0..7]
  vmovdqa ymm1,ymmword ptr M
; if SF(ymm1[i]) then ymm0[i]:=A[i]
;       ↓↓         else ymm0[i]:=0
  vpmaskmovd ymm0,ymm1,ymmword ptr A
; if SF(ymm1[i]) then A[i]:=ymm2[i]
;       ↓↓         else {A[i] не меняется}
  vpmaskmovd ymmword ptr A,ymm1,ymm2
```

А сейчас рассмотрим команды, производящие "расстановку" верхних или нижних частей двух регистров в третий регистр:

```
vunpck{l,h}p{s,d} op1,op2,op3
```

Буквы **l** (low) и **h** (high) задают нижние или верхние половины операндов op2 и op3, а буквы **s** (**s**ingle) и **d** (**d**ouble) задают тип элемента вектора. Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

Например:

```
.data
  org 1000h
A real8 10.0,11.0,12.0,13.0; A[0..3]
B real8 20.0,21.0,22.0,23.0; B[0..3]
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vmovdqa ymm2,ymmword ptr B; ymm2:=B
```

> Считается, что каждый регистр YMM состоит из двух "линий" (lane) по 128 бит, эти линии обрабатываются независимо, поэтому получается такая картинка:

vunpckld ymm0,ymm1,ymm2

ymm1	13.0	12.0	11.0	10.0
ymm2	23.0	22.0	21.0	20.0
ymm0	22.0	12.0	20.0	10.0

vunpckhd ymm0,ymm1,ymm2

ymm1	13.0	12.0	11.0	10.0
ymm2	23.0	22.0	21.0	20.0
ymm0	23.0	13.0	21.0	11.0

А теперь пример с элементами типа `single`. Важно помнить, что регистр YMM состоит из двух "линий" по 128 бит, которые обрабатываются независимо, поэтому получается такая картинка:

```
.data
  org 1000h
A real4 10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0; A[0..7]
B real4 20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0; B[0..7]
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vmovdqa ymm2,ymmword ptr B; ymm2:=B
  vunpckls ymm0,ymm1,ymm2
```

ymm1	17.0	16.0	15.0	14.0	13.0	12.0	11.0	10.0
ymm2	27.0	26.0	25.0	24.0	23.0	22.0	21.0	20.0
ymm0	25.0	15.0	24.0	14.0	21.0	11.0	20.0	10.0

Далее рассмотрим аналогичные команды расстановки, операнды которых рассматриваются как целые числа длиной в 2, 4 или 8 байт:

vpunpck{**l**,**h**}{**bw**,**wd**,**dq**} op1,op2,op3

Здесь, например, буквы **bw** "намекают", что байт из op2 и байт из op3 образуют слово в op1 и т.д. Таким образом, например, команда **vunpckls** (для вещественных чисел) работает так же, как и команда **vpunpckldq** (для целых чисел), т.е. это синонимы.

Пример команды **vpunpcklwd**:

```
.data
  org 1000h
A dw 10,11,12,13,14,15,16,17,18,19,  \
     20,21,22,23,44,25;  A[0..15] of dw
B dw 30,31,32,33,34,35,36,37,38,39,  \
     40,41,42,43,44,45;  B[0..15] of dw
.code
  vmovdqa ymm1,ymmword ptr A
  vmovdqa ymm2,ymmword ptr B
  vpunpcklwd ymm0,ymm1,ymm2
```

ymm1	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
ymm2	35	34	33	32	31	30	29	28	27	26	26	24	23	22	21	20
ymm0	31	11	30	10	29	09	28	08	23	03	22	02	21	01	20	00

Следующие команды "перемешивают" (shuffling) пары стоящих рядом элементов вектора, выбирая один из элементов пары в соответствии с маской, заданной в op4:

```
vshufp{s,d} op1,op2,op3,op4
```

Буквы **s** (single) и **d** (double) задают тип элемента вектора. Допустимые форматы операндов:

op1	op2	op3	op4
r128	r128	r128, m128	i8
r256	r256	r256, m256	i8

Команда "перемешивает" внутри каждой 128-битной линейки независимо. При этом младшая половина результата заполняется из элементов op2, а старшая – из элементов op3. Выбор элементов производится по набору индексов, заданному в операнде i8 (каждый индекс один или два бита).

Например:

```
.data
  org 1000h
A real8 10.0,11.0,12.0,13.0; A[0..3]
B real8 20.0,21.0,22.0,23.0; B[0..3]
; ↓↓ C,D:array[0..7] of single;
C real4 30.0,31.0,32.0,33.0,34.0,35.0,36.0,37.0
D real4 40.0,41.0,42.0,43.0,44.0,45.0,46.0,47.0
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vmovdqa ymm2,ymmword ptr B; ymm2:=B
  vshufpd ymm0,ymm1,ymm2,1101b
```

op2=ymm1	13.0	12.0	11.0	10.0
op3=ymm2	23.0	22.0	21.0	20.0
op1=ymm0	23.0	13.0	20.0	11.0

```
  vmovdqa ymm1,ymmword ptr C; ymm1:=C
  vmovdqa ymm2,ymmword ptr D; ymm2:=D
  vshufps ymm0,ymm1,ymm2,11000101b
```

op2=ymm1	37.0	36.0	35.0	34.0	33.0	32.0	31.0	30.0
op3=ymm2	47.0	46.0	45.0	44.0	43.0	42.0	41.0	40.0
op1=ymm0	47.0	44.0	35.0	35.0	43.0	40.0	31.0	31.0

Долго приходится разбираться 🙄.

Есть и похожие команды "расстановки" целых чисел формата **dd** из одного вектора op2 в позиции другого вектора op1 в соответствии с двух битовыми индексами, заданными в битовой маске op3:

```
vpshufd op1,op2,op3
```

Допустимые форматы операндов:

op1	op2	op3
r128	r128, m128	i8
r256	r256, m256	i8

К сожалению, для регистров YMM такая "расстановка" производится в границах каждой 128-битной линейки независимо. Команда выполняется по такому алгоритму: [1]

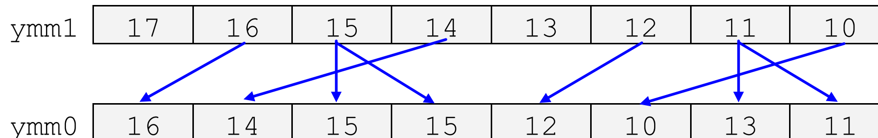
[1] Команда "расстановки" считает свои операнды целочисленными векторами, поэтому будет работать менее эффективно, если использовать её после команд загрузки на регистр вещественного вектора (см. первую сноску в конце этой главы). Похожие команды

vperm{**w**,**d**,**q**} op1,op2,op3 и **vpshuf**{**l**,**h**}**w** op1,op2,op3=i8

```
for i:=0 to 3 do begin j:=i8[2*i+1..2*i]
    op1[i]:=op2[j]; op1[i+4]:=op1[4+j]
end;
```

Например:

```
.data
Z  equ 80000000h
  org 1000h
A dd 10,11,12,13,14,15,16,17; A[0..7]
B dd 0,2,0,Z,5,7,Z,3; B[0..7]; Маска
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vpshufd ymm0,ymm1,10000101b
```



Обратите внимание, что каждый элемент из `op2` может вставляться более, чем в одну позицию `op1`. Аналогичная команда для "расстановки" байт из `op2` в позиции `op1` в соответствии с индексами, заданными в векторной маске `op3`:

vpshufb `op1,op2,op3`

Допустимые форматы операндов:

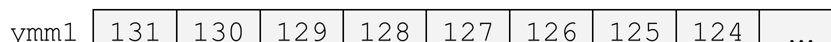
op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

Для каждой 128-битной линейки байт в маске `op3` задаёт позицию, в которой соответствующий байт из `op2` размещается в `op1`. Команда выполняется по такому алгоритму:

```
for i:=0 to 15 do begin
  if SF(op3[i]) then op1[i]=0 else begin
     j:=op3[i] and 1111b; op1[i]:=op2[j]
  end;
  if SF(op3[15+i]) then op1[15+i]:=0 else begin
     j:=op3[15+i] and 1111b; op1[15+i]:=op2[15+j]
  end
```

Например:

```
.data
Z  equ 80h
  org 1000h
; ↓↓  A[0..31] of byte
A db 100,101,102,103,104,105,106,107, \
     108,109,110,111,112,113,114,115, \
     116,117,118,119,120,121,122,123, \
     124,125,126,127,128,129,130,131
; ↓↓ Маска B[0..31] of byte
B db 1,12,17,Z,5,9,Z,11,5,0,0,6,Z,Z,1,4 \
     Z,3,3,0,11,0,Z,1,5,5,Z,0,11,0,Z,10
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vmovdqa ymm2,ymmword ptr B; ymm2:=B
  vpshufb ymm0,ymm1,ymm2
```



мы рассматривать не будем.

ymm2	10	Z	0	11	0	Z	5	5	...
ymm0	110	0	100	111	100	0	105	105	...

Следующая команда "смешивает" элементы двух векторов, выбирая один из элементов пары в соответствии с битовой или векторной маской, заданной в последнем операнде. Сначала команды с битовыми масками:

vpblend{**w**,**d**} op1,op2,op3,op4

рассматривают свои операнды как целые числа форматов **dw** и **dd**, а аналогичные команды

vblendp{**s**,**d**} op1,op2,op3,op4

рассматривают свои операнды как вещественные числа форматов **s**ingle и **d**ouble. Допустимые форматы операндов:

op1	op2	op3	op4
r128	r128	r128, m128	i8
r256	r256	r256, m256	i8

Команды **vpblendd** и **vblendp**{**s**,**d**} выполняется по такому алгоритму:

```
{N=8,4,2}
for i:=0 to N-1 do
  if op4[i] then op1[i]:=op3[i]
             else op1[i]:=op2[i]
```

Команда **vpblendw** выполняется по такому алгоритму:

```
{N=16,8}
for i:=0 to 7 do
  if op4[i] then begin
     op1[i]:=op3[i] else op1[i]:=op2[i];
     if (8+i<N) then op1[8+i]:=op3[8+i]
                else op1[8+i]:=op2[8+i]
  end
```

Ясно, что для этих команд важен только размер элемента вектора, а совсем не его тип (целый или вещественный). Например:

```
.data
  org 1000h
A dd 10,11,12,13,14,15,16,17; A[0..7]
B dd 20,21,22,23,24,25,26,27; B[0..7]
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vmovdqa ymm2,ymmword ptr B; ymm2:=B
  vpblendd ymm0,ymm1,ymm2,11010010b
```

ymm1	17	16	15	14	13	12	11	10
ymm2	27	26	25	24	23	22	21	20
ymm0	27	26	15	24	13	12	21	10

Далее рассмотрим аналогичные команды **blendv**, но с векторными масками:

vblendvp{**s**,**d**} op1,op2,op3,op4

для вещественных, и

vpblendvb op1,op2,op3,op4

для однобайтных целых чисел. Допустимые форматы операндов:

op1	op2	op3	op4
r128	r128	r128, m128	r128
r256	r256	r256, m256	r256

Команды выполняется по такому алгоритму:

```
{N=32,8,4,2}
```

```
for i:=0 to N-1 do
  if SF(op4[i]) then op1[i]:=op3[i]
                 else op1[i]:=op2[i]
```

Специфической является команда, загружающая в первый операнд попарные копии (дубликаты) вещественных элементов из чётных или нечётных позиций второго операнда:

vmov{**sl**,**sh**,**d**}**dup** op1,op2

Буквы {**sl**,**sh**} задают дублирование вещественных чисел одинарной точности (**s**) с чётных (**l** – low) или нечётных (**h** – high) позиций вектора op2 в вектор op1. Для op1 формата XMM, предусмотрен особый случай op2 формата m64. Буква {**d**} определяет дублирование вещественного числа типа double (только с нечётных позиций вектора op2). Допустимые форматы операндов:

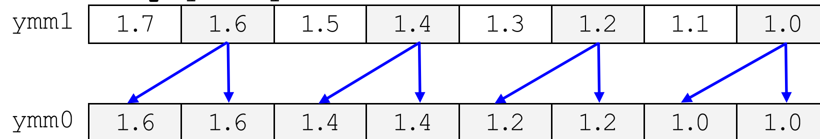
op1	op2
r128	r128, m64
r256	r256, m256

Команды выполняется по такому алгоритму:

```
{N=4,2}
case
  sl:   j:=0;  {чётные элементы}
  sh,d: j:=1   {нечётные элементы}
end;
for i:=0 to N-1 do begin
    op1[2*i]:=op2[2*i+j];
    op1[2*i+1]:=op2[2*i+j]
end
```

Например, для команды

vmovsldup ymm0,ymm1



Далее рассмотрим команды, производящие запись в op1 "перестановки" из элементов op2 и op3 по правилу, заданному в op4. Начнём с команды, переставляющей 128-битные поля:

vperm2{**f**,**i**}**128** op1,op2,op3,op4

Буквы {**f**,**i**} задают синонимы, они просто "намекают" ЭВМ, что 128-битовые поля нужно рассматривать как целые или как вещественные значения. Ценность этой команды в том, что она, в числе немногих, может переставлять данные из одной 128-битной линейки регистра YMM в другую. Допустимые форматы операндов:

op1	op2	op3	op3
r256	r256	r256, m256	i8

Алгоритм выполнения команды:

```
{op1,op2,op3:array[1..0] of f128;
 opi=| opi[1]=f128 | opi[0]=f128 |
 i8[1..0]=000000xxb=0xh}
case i8[1..0] of {левая линейка}
  0: op1[0]:=op2[0];
  1: op1[0]:=op2[1];
  2: op1[0]:=op3[0];
  3: op1[0]:=op3[1]
end;
{i8[5..4]=00xx0000b=x0h}
case i8[5..4] of {правая линейка}
  0: op1[1]:=op2[0];
```

```
  1: op1[1]:=op2[1];
```

```
  2: op1[1]:=op3[0];
  3: op1[1]:=op3[1]
end;
if i8[3]=1
   then op1[0]:=0;
if i8[7]=1 {i8[7]=0x000000b}
   then op1[1]:=0
```

Пример:

```
.data
  org 1000h
; A,B,C[3..0] of double
A real8 4.0,3.0,2.0,1.0
B real8 8.0,7.0,6.0,5.0
C real8 4 dup (?)
.code
  vmovdqa ymm0,ymmword ptr A;              ymm0=1.0,2.0,3.0,4.0
  vmovdqa ymm1,ymmword ptr B;              ymm1=5.0,6.0,7.0,8.0
  vperm2f128 ymm2,ymm0,ymm1,1001 0011b;  ymm2=0.0,0.0,5.0,6.0
```

Отдельно стоит отметить безадресную команду **vzeroall**, обнуляющую сразу все YMM регистры (за один такт!). Это очень полезная команда, она позволяет при работе с XMM регистрами не заботиться о старших половинках соответствующих YMM регистров (они всё равно нулевые). Рекомендуется, если подпрограмма работает с XMM регистрами, ставить эту команду в прологе и эпилоге каждой подпрограммы. Почти так же работает немного более быстрая команда **vzeroapper**.

17.1.1. Задача транспонирования матрицы

Следует помнить, что при описании языка главная цель состоит в том, чтобы объяснить, как писать на нём программы и что такие программы означают.

Марвин Ли Минский

Рассмотрим применение команд расстановки и перестановки для задачи транспонирования матрицы 4x4 из вещественных чисел двойной точности B:=A^T:

$$\begin{vmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{vmatrix} \rightarrow \begin{vmatrix} A_{00} & A_{10} & A_{20} & A_{30} \\ A_{01} & A_{11} & A_{21} & A_{31} \\ A_{02} & A_{12} & A_{22} & A_{32} \\ A_{03} & A_{13} & A_{23} & A_{33} \end{vmatrix}$$

```
.data
  org 1000h
; ↓↓ A[0..3,0..3] of double
A real8 4.0,3.0,2.0,1.0
  real8 8.0,7.0,6.0,5.0
  real8 12.0,11.0,10.0,9.0
  real8 16.0,15.0,14.0,13.0
; ↓↓ B[0..3,0..3] of double
B real8 4 dup (4 dup (?))
.code
; загрузка строк матрицы A в ymm0..ymm3
for i,<0,1,2,3>
  vmovdqa ymm&i,ymmword ptr A+32*i
endif
;                                ymm0=01,02,03,04
;                                ymm1=05,06,07,08
;                                ymm2=09,10,11,12
```

```
;                                   ymm3=13,14,15,16
; расстановка
  vunpckld    ymm4,ymm0,ymm1;      ymm4=01,05,03,07
  vunpckhd    ymm5,ymm0,ymm1;      ymm5=02,06,04,08
  vunpckld    ymm6,ymm2,ymm3;      ymm6=09,13,11,15
  vunpckhd    ymm7,ymm2,ymm3;      ymm7=10,14,12,16
; перестановка
  vperm2f128 ymm0,ymm4,ymm6,13h;  ymm0=01,05,09,13
  vperm2f128 ymm1,ymm5,ymm7,13h;  ymm1=02,06,10,14
  vperm2f128 ymm2,ymm4,ymm6,02h;  ymm2=03,07,11,15
  vperm2f128 ymm3,ymm5,ymm7,02h;  ymm7=04,08,12,16
```

17.1.2. Загрузка регистра из несмежных областей памяти

История показывает, что во всём новом
обычно кроется какой-то подвох.
Айзек Азимов. «Сами боги»

В наборе векторных команд AVX2 (процессор Haswell, 2013 год) появилась возможность загружать в векторный регистр элементы, расположенные в несмежных областях памяти. Как Вы знаете, для регистров общего назначения есть доступ к памяти SIB (Scale Index Base) для регистров, когда исполнительный адрес памяти вычисляется как `Disp[B1+Scale*I2]`. Аналогичный режим для векторных регистров называется VSIB (Vector Scale Index Base), в этом случае адрес операнда выглядит как `Disp[B1+Scale*VI2]`, где на месте индексного регистра `VI2` можно задавать любой векторный регистр `xmm` или `ymm`, `B1` базовый регистр, а `Disp` – смещение. Этот *индексный* векторный регистр задаёт вектор индексов для обращения в память. Сначала команды "сборки" (gather – собирать), т.е. загрузки вещественных чисел из памяти на векторный регистр:

vgather{**d**,**q**}**p**{**s**,**d**} op1,op2,op3

Буквы {**d** → **d**word, **q** → **q**word} определяют размер набора индексов в индексном векторном регистре из op2, т.е. 32-х или 64-хбитные адреса. Буквы {**s** → **s**ingle, **d** → **d**ouble} определяют тип элементов вектора, собираемого в op1. Третий операнд op3 содержит векторную маску из целочисленных элементов (индексов) типа longint (для **s**ingle) или int64 (для **d**ouble). Обратите внимание, что индексы знаковые, т.е. могут быть меньше нуля. У команды есть синоним

vpgather{**d**,**q**}{**d**,**q**} op1,op2,op3

Здесь последние буквы (как и первые) {**d** → **d**word, **q** → **q**word} "говорят", что программист рассматривает операнды как целые числа форматов longword и qword соответственно. Ясно, что для команд пересылки имеет значение только длина операнда (4 или 8 байт). Допустимые форматы операндов: [1]

op1	op2	op3
r128	m32, m64	r128
r256	m32, m64	r256

Алгоритм выполнения команды:

```
{ op2=Disp[B1+Scale*ymm] }
for i:=0 to N-1 do begin
  temp:=B1+Disp+Scale*int64(imm[i]);
  Ref:=qword(temp) mod 2^64;
  if SF[op3[i] then op1[i]:=Ref↑
end;
op3:=0
```

Рассмотрим пример со "сборкой" из памяти целых чисел формата **dd**:

[1] Есть *двухадресная* модификация этой команды, в которой вместо векторной маски в третьем операнде используется регистр маски, указанный у первого операнда (см. разд. 17.7):

vgather{**d**,**q**}**p**{**s**,**d**} op1{k1},op2

```
.data
; нет org, т.к. vmovdqu
A dd -8,-7,-6,-5,-4,-3,-2,-1
B dd 0,1,2,3,4,5,6,7
T dd 20,21,22,23,24,25,26,27
; ↓↓ индексы I[0..7] of longint
I dd 4,-6,1,5,5,-2,1,3
; ↓↓ маска M[0..7] of [-1..0]
M dd -1,-1,0,-1,0,-1,-1,-1
.code
  vmovdqu ymm1,ymmword ptr I; индексы выборки
  vmovdqu ymm2,ymmword ptr M; маска
  vmovdqu ymm0,ymmword ptr T; начальные значения ymm0
```

ymm0	27	26	25	24	23	22	21	20

```
  mov rbx,offset B; база на начало B
  vpgatherdd ymm0,[rbx+4*ymm1],ymm2; выборка single
```

индекс	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
A→	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6	7
									↑ rbx							
ymm1	3	1	-2	5	5	1	-6	4								
ymm2	-1	-1	0	-1	0	-1	-1	-1								
ymm0	3	1	25	5	23	1	-6	4								

Никаких проверок на допустимость индексов не делается, при выходе на чужую память будет исключение. Обратные команды, "разбрасывающие" элементы векторного регистра по памяти:

```
vscatter{d,q}p{s,d} op1,op2,op3        и
vpscatter{d,q}{d,q} op1,op2,op3
```

Допустимые форматы операндов:

op1	op2	op3
m32, m64	r128	r128
m32, m64	r256	r256

В качестве примера рассмотрим задачу загрузки на векторный регистр главной диагонали квадратной матрицы вещественных чисел типа single:

```
.data?
  org 1000h
X dd 8 dup(8 dup (?))
.data
  org 1000h
; ↓↓ индексы I[0..7] главной диагонали
I dd 63,54,45,36,27,18,9,0
; ↓↓ маска M[0..7]
M dd 8 dup (-1)
.code
  vmovdqu ymm1,ymmword ptr I; индексы выборки
  vmovdqu ymm2,ymmword ptr M; маска
; ↓↓ ymm0[7..0]:=[X77, X66, … ,X11, X00]
  vgatherdps ymm0,X[4*ymm1],ymm2
```

На регистре ymm0 располагается главная диагональ матрицы в обратном порядке. Можно загрузить диагональ и в прямом порядке, если индексы в векторе индексов I задать в обратном порядке:

```
I dd 0,9,18,27,36,45,54,63
```

Говорят, что в первых реализациях этого механизма (лет 10 назад) эффективность его была не намного лучше, чем "ручная" загрузка элементов из памяти на регистр, однако в последних моделях ЭВМ, эта ситуация, судя по всему, исправляется.

17.2. Арифметические команды для целых чисел

В мире есть много трудных вещей, но нет ничего труднее, чем четыре действия арифметики.

Бе́да Достопочтенный (673-735)

(достала римская система счисления ☺)

Кроме "традиционных" операций над *векторами* целых и вещественных чисел, на векторных регистрах можно производить и скалярные операции над вещественными числами. Вещественное число при этом располагается в младших разрядах регистра. Векторные регистры сейчас надо рассматривать как основные при обработке вещественных чисел, они работают более быстро, хотя и имеют меньшую разрядность (64 бита вместо 80 бит на старых вещественных регистрах `st(0)-st(7)`). Скалярные операции на векторных регистрах будут рассмотрены в разд. 17.3.1.

17.2.1. Команды сложения и вычитания векторов целых чисел

Конечной целью образования должно быть искусство конструктивного мышления.

Никлаус Вирт

Мы уже знаем, что неразличимые в памяти целые числа могут *трактоваться* программистом как знаковые, или как беззнаковые, в то время как операции сложения и вычитания над ними выполняются по *одинаковым* алгоритмам. Операции сложения и вычитания могут давать результат, не помещающийся в отводимое для него место, говорят, что это операции с *переполнением* (wraparound), о переполнении программиста предупреждают установленные флаги `CF` и `OF`.

На векторных регистрах, однако, возможны и такие операции над целыми числами, которые никогда не дают переполнения результата, они называются операциями с *насыщением* (with saturation, saturated arithmetic). При выходе значения такой операции за верхнюю или нижнюю допустимые границы своего типа, в качестве результата берётся соответственно эта верхняя или нижняя граница. При этом есть, например, две команды сложения с насыщением (одна для знаковых и одна для беззнаковых чисел), они, естественно, выполняются по *разным* алгоритмам. Обозначим такие операции сложения и вычитания, как ⊕ и ⊖.

Например, пусть для `X db 200` выполняется беззнаковое сложение с насыщением `X:=X ⊕ 100`, тогда получится ответ `X=255`, а для операции `X:=X⊖210` получится ответ `X=0`. Аналогично, если для `X db 120` выполняются знаковое сложение с насыщением `X:=X ⊕ 10`, тогда получится ответ `X=127`, а для `X db -20` операция `X:=X⊖110` получится ответ `X=-128`.

Никакие флаги при этом не устанавливаются. Такая "хитрая" арифметика широко применяется при обработке мультимедийных данных (изображения и звука). Действительно, например, при сложении или вычитании двух "звуков" их громкость не может выйти за некоторый максимальный и минимальный пределы.

Как мы уже говорили, при чтении данных на векторный регистр эти данные можно трактовать просто, как битовую строку длиной 128 или 256 бит. Вся структура такой строки (количество и тип элементов в векторе) указывается в коде операции команды, которая обрабатывает такие данные.

Сначала команды сложения и вычитания, эти команды *трёхадресные*:

```
vp{add,sub}{ ,s,us}{b,w,d,q} op1,op2,op3
```

Буква **p** задаёт упакованный (**p**acked) вектор, обозначения **s** и **us** задают знаковые и беззнаковые операции с насыщением, а отсутствие этих букв задают "обычные" операции с переполнением (эти знаковые и беззнаковые операции не различаются и никакие флаги не устанавливаются). Буква в конце кода операции задаёт размер элемента упакованного вектора: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128

r256	r256	r256, m256

Пример:

```
.data
  org 1000h
X db 32 dup (?); X[0..31]
Y dd 8  dup (?); Y[0..7]
.code
  vmovdqa ymm1,ymmword ptr X
; ↓↓ "обычное" сложение знаковое или беззнаковое
; ↓↓ ymm0[0..31]:=ymm1[0..31]+X[0..31]
  vpaddb ymm0,ymm1,ymmword ptr X
; ↓↓ ymm0[0..7]:=ymm0[0..7]-Y[0..7]
  vpsubd ymm0,ymm0,ymmword ptr Y
```

Не удивляйтесь, что сначала мы работаем с регистрами, считая, что на них вектор чисел формата **db**, а следующей командой "приказываем" считать, что этот же регистр содержит вектор чисел формата **dd**. Вряд ли в этом есть какой-то смысл, но, по большому счёту, "машине всё равно" ☹, [это не совсем так, см. первую сноску в конце этой главы].

А теперь операции сложения и вычитания с насыщением:

```
.data
  org 1000h
X db 32 dup (?); X[0..31]
Y dq 4  dup (?); Y[0..3]
.code
  vmovdqa ymm1,ymmword ptr X
  vpbroadcastq ymm2,Y+8; ymm2[0..7]:=Y[2]
; ↓↓ ymm0[0..31]:=ymm1[0..31]⊕ymm1[0..31]
  vpaddsb ymm0,ymm1,ymm1; знаковые байты
; ↓↓ ymm0[0..3]:=ymm2[0..3]⊖ymm2[0..3]
  vpsubusq ymm0,ymm2,ymm2; беззнаковые qword
```

Ясно, что никакие флаги эти команды не устанавливают, да и нельзя придумать (разумные) правила, как их устанавливать (флаг один, а результатов много). Там, где без флагов не обойтись (в командах сравнения) приходится вырабатывать векторную маску результатов (см. разд. 17.2.4 и 17.3.5). Следовательно, о правильности операций должен позаботиться сам программист (ну, или компилятор ☺).

Далее следует упомянуть команды для вычисления максимума и минимума целых чисел:

```
;  ∀i op1[i]:={max,min}(op2[i],op3[i])
   vp{max,min}{s,u}{b,w,d,q} op1,op2,op3
```

Обозначения **s** и **u** задают знаковые и беззнаковые операнды, а последняя буква определяет размер элемента вектора: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

Команды умножения элементов вектора op2 на знак числа соответствующего элемента вектора op3 (sign adjustment):

```
   vpsign{b,w,d} op1,op2,op3
```

Последняя буква определяет размер элемента вектора: **b** → **byte**, **w** → **word**, **d** → **dword**. Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

Алгоритм выполнения команды:

```
for i:=0 to N-1 do
```

```
  case sign(op3[i]) of
     0: op1[i]:=0;
     1: {op1[i] не меняется};
    -1: op1[i]:=-op2[i]
  end;
```

Отметим, что при совпадении всех трёх операндов это просто вычисление абсолютной величины. Здесь же нужно упомянуть и команды взятия абсолютной величины:

vpabs{**b**,**w**,**d**,**q**} op1,op2

Последняя буква определяет размер элемента вектора: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Допустимые форматы операндов:

op1	op2
r128	r128, m128
r256	r256, m256

Алгоритм выполнения команды:

```
for i:=0 to N-1 do
  op1([i]):=abs(op2([i])
```

Далее рассмотрим команды *горизонтального* сложения и вычитания векторов целых чисел:

vph{**add**,**sub**}{**w**,**sw**,**d**} op1,op2,op3; op1:=op2⊗op3

Конец кода операции определяет размер элемента вектора: **w** → **dw** (**sw** → **dw** с насыщением), **d** → **dd**. Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

Алгоритм выполнения команды **vph**{**add**,**sub**}**d**:

```
i:=0; {N=8}
repeat
  op1[i]:=   op2[i]⊗op2[i+1];
  op1[i+1]:=op2[i+2]⊗op2[i+3];
  op1[i+2]:=op3[i]⊗op3[i+1];
  op1[i+3]:=op3[i+2]⊗op3[i+3];
  i:=i+4
until i>=8
```

А теперь алгоритм выполнения команд **vph**{**add**,**sub**}{**w**,**sw**}:

```
i:=0; {N=16}
repeat
  op1[i]:=   op2[i]   ⊗op2[i+1];
  op1[i+1]:=op2[i+2]⊗op3[i+3];
  op1[i+2]:=op2[i+4]⊗op2[i+5];
  op1[i+3]:=op2[i+6]⊗op3[i+7];
  op1[i+4]:=op3[i]   ⊗op3[i+1];
  op1[i+5]:=op2[i+2]⊗op3[i+3];
  op1[i+6]:=op3[i+4]⊗op3[i+5];
  op1[i+7]:=op3[i+6]⊗op3[i+7];
  i:=i+8
until i>=8
```

Например:

```
.data
  org 1000h
A dd 10,11,12,13,14,15,16,17; A[0..7]
B dd 30,31,32,33,34,35,36,37; B[0..7]
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
```

```
vmovdqa ymm2,ymmword ptr B; ymm2:=B
vphaddd ymm0,ymm1,ymm2
```

ymm1	17	16	15	14	13	12	11	10
ymm2	37	36	35	34	33	32	31	30
ymm0	73	69	33	29	65	61	25	21

17.2.2. Задача суммирования элементов вектора целых чисел

Программирование – это ремесло, и каждый программист должен достичь нужного профессионального уровня.
Чарльз Уэзерелл
«Этюды для программистов»

Рассмотрим такую задачу. Пусть надо просуммировать массив из 4*N *беззнаковых* чисел типа longword, тогда, вообще говоря, для хранения суммы необходима переменная типа qword:

```
.data
  org 1000h
N equ 1000000
X dd  4*N dup (?); X[0..4*N-1] of longword;
S dq  0; var S: qword;
.code
; правильное суммирование массива  S:=Сумма(X[i])
; ↓↓ ymm0=4 частичных суммы qword
  pxor xmm0,xmm0; Это xmm0:=xmm0 xor xmm0 [1]
  mov ecx,N; для цикла
  xor ebx,ebx; индекс массива i:=0
; ↓↓ ymm1[0..3]:=qworb(X[i..i+3])
L:vpmovzxdq ymm1,xmmword ptr X[ebx]
; ↓↓ +очередные 4 элемента X в ymm0
  vpaddq ymm0,ymm0,ymm1
  add ebx,16; i:=i+4
  loop L
; ↓↓ S:=S+4 частичные суммы из ymm0
for i,<0,1,2,3>
  pextrq rax,ymm0,i; rax:=ymm0[i]
  add S,rax
endm
comment &
  вместо извлечения 4-х частичных сумм можно делать
  перестановку элементов и сложение
  и два извлечения, будет 5 команд вместо 8-ми ⚠
&
; ↓↓ ymm1:=?,?,ymm0[3],ymm0[2]
; vextracea128 xmm1,ymm0,1; ↓ ? – не важно
; ↓↓ ymm0:=?,?,ymm0[3]+ymm0[1],ymm0[2]+ymm0[0]
; vpaddq ymm0,ymm1,ymm0
```

---

[1] Тонкий момент. Так как регистр ymm состоит из двух линий по 128 бит, то для его обнуления можно использовать логическую команду **vpxor** ymm0,ymm0,ymm0 (логические команды см. раздел 7.5). Для выполнения этой команды, однако, нужно 2 такта, поэтому выгоднее использовать команду **pxor** xmm0,xmm0, которая короче, и выполняется за 1 такт, она сама обнуляет старшую часть регистра ymm0. По аналогичным причинам для обнуления регистра rax вместо команды **xor** rax,rax лучше использовать более короткую команду **xor** eax,eax.

```
; pextrq S,ymm0,0; S:=ymm0[2]+ymm0[0]
; pextrq rax,ymm0,1; rax:=ymm0[3]+ymm0[1]
; add    S,rax
  outwordln S,,"Сумма массива="
```

Ясно, что проблемы возникнут только при суммировании массива с элементами формата **dq**, придётся делать обычный цикл, а под сумму отвести 128 бит на регистре XMM.

Похожая проблема правильности результата операций существует и в языках высокого уровня, там тоже надо стараться при необходимости отводить под результат операции больше места, чем под операнды. Например:

```
var A,B: word; C: longword;
    X,Y: longint; Z: int64;
```

При этом оператор `C:=A+B` будет *всегда* давать правильный результат, так как понимается компилятором как

```
C:=longword(A)+longword(B)
```

Т.е. компилятор считает тип `longword` стандартным при проведении промежуточных вычислений. Здесь, однако, надо быть внимательнее, так как на первый взгляд "хороший" оператор `Z:=X+Y` будет давать правильный ответ не всегда, так как понимается компилятором как

```
Z:=int64(A+B)
```
☹

Для получения правильного результата программисту придётся использовать *явное* преобразование типов и использовать оператор `Z:=int64(X)+Y`, он будет пониматься компилятором как

```
Z:=int64(A)+int64(B)
```

17.2.3. Умножение векторов целых чисел (деления нет 🐷)

Всё действительное – разумно, всё разумное – действительно.

Георг Гегель

Итак, команды умножения. Как Вы уже знаете, для регистров общего назначения существуют операции знакового и беззнакового умножения. *Одноадресные* операции **mul** op2 и **imul** op2 *всегда* дают правильный ответ, так как под произведение отводится место, в два раза больше, чем под каждый сомножитель. В то же время есть двух и трёхадресные команды умножения

```
imul op1,op2;     op1:=op1*op2          и
imul op1,op2,op3; op1:=op2*op2
```

Эти команды выдают в качестве результата только младшую часть произведения, и, таким образом, могут давать *неправильный* ответ. Предусмотрен контроль правильности такого произведения: когда оно не помещается в свою младшую часть, то устанавливаются флаги `CF=OF=1`. Заметим также, что реализовывать такие же команды для умножения *беззнаковых* чисел бессмысленно, так как младшие части произведения знаковых и беззнаковых чисел *одинаковы*.

Для векторных регистров тоже предусмотрены *трёхадресные* команды умножения, дающие как усечённый или полный результаты:

```
vpmul{l,h,hu}{b,w,d,q} op1,op2,op3; усечённое
```

Обозначения **l** (**l**ow), **h** (**h**igh) и **hu** (**u**nsign **h**igh) задают сохранение в результате только младшей или старшей половины произведения соответственно, причём старшая половина произведения вычисляется по-разному для знаковых (**h**) и беззнаковых (**hu**) чисел. Буква в конце кода операции задаёт размер элемента упакованного вектора: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**.

Теперь полное произведение:

```
vpmul{ ,u}{bw, wd, dq} op1,op2,op3
```

Суффикс **u** задаёт *беззнаковое* полное произведение, а отсутствие этого суффикса – *знаковое*. Две буквы в конце кода операции задают размеры сомножителей и произведения: **bw**: **byte** → **word**, **wd**: **w** → **dword**, **dq**: **dword** → **qword**. Допустимые форматы операндов:

op1	op2	op3

r128	r128	r128, m128
r256	r256	r256, m256

Сначала примеры команд, дающих *усечённое* (знаковое) произведение:

```
.data
  org 1000h
A dq 10,11,12,13; A[0..3]
B dq 20,21,22,23; B[0..3]
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vmovdqa ymm2,ymmword ptr B; ymm2:=B
; ↓↓ ymm0[0..3]:=low(ymm1[0..3]*ymm2[0..3])
  vpmullq ymm0,ymm1,ymm2
```

ymm1	13	12	11	10
ymm2	23	22	21	20
ymm0	13*23	12*22	11*21	10*20

```
; ↓↓ ymm0[0..7]:=high(ymm1[0..7]*B[0..7])
  vpmulhd ymm0,ymm1,ymm2
```

А теперь рассмотрим знаковые и беззнаковые команды, дающие полное произведение, под которое отводится в два раза больше места, чем под каждый из сомножителей:

```
.data
  org 1000h
A dd 10,?,11,?,12,?,13,?; A[0..7]
B dd 20,?,21,?,22,?,23,?; B[0..7]
.code
  vmovdqa ymm1,ymmword ptr A
  vmovdqa ymm2,ymmword ptr B
; ↓↓ ymm0[0..3]:=low(ymm1[0..3])*low(ymm2[0..3])
  vpmuldq ymm0,ymm1,ymm2; знаковое произведение
```

ymm1	?	13	?	12	?	11	?	10
ymm2	?	23	?	22	?	21	?	20
ymm0	13*23		12*22		11*21		10*20	

```
  vpmuludq ymm0,ymm1,ymmword ptr A; беззнаковое произведение
```

В математике есть "странное" умножение без переноса (carry-less multiplication), математическое название "умножение над конечным полем". Например, рассмотрим произведение "в столбик" 3-значных двоичных чисел, обычное и без переноса:

Обычное	Без переноса
110	110
101	101
110	110
000	000
101	101
011010	010010

Как видно, при сложении двоичных чисел без переноса "единица в уме" игнорируется, и не прибавляется к старшему разряду. Такая операция оказалась очень полезной в различных алгоритмах шифрования и вычисления контрольной суммы массива CRC (Cyclic Redundancy Checks). В нашей машине такое произведение реализовано на XXM регистрах, при этом два 64-битных целых числа дают 128-битное произведение без переноса (обозначим это произведение как ⍟):

```
vpclmulqdq op1,op2,op3,op4
```

Буквы в конце кода операции задают размеры сомножителей и произведения: первое **q** → **dq**, далее две буквы **dq** → **d**ouble **q**uadword (**xmm**). Допустимые форматы операндов:

op1	**op2**	**op3**	**op4**
r128	r128	r128, m128	i8

Алгоритм выполнения команды:

```
var op1,op1,op3: xmm; op4: byte; t1,t2: qword;
case op4[0]
  0: t1:=op2[63..0];
  1: t1:=op2[127..64]
end;
case op4[4]
  0: t2:=op3[63..0];
  1: t2:=op3[127..64]
end;
op1:=t1⊛t2
```

Отдельно стоит упомянуть трёхадресные команды, совмещающие операции умножения и сложения векторов целых чисел.

vpmadd{ ,**u**}{**bw**,**wd**,**dq**} op1,op2,op3

Суффикс **u** задаёт беззнаковые операции, а его отсутствие – знаковые. Две буквы в конце кода операции задают размеры сомножителей и произведения-суммы: **bw**: **byte** → **word**, **wd**: **word** → **dword**, **dq**: **dword** → **qword**. Допустимые форматы операндов:

op1	**op2**	**op3**
r128	r128	r128, m128
r256	r256	r256, m256

Вот пример такой команды:

```
.data
  org 1000h
A dd 10,11,12,13,14,15,15,17; A[0..7] of dd
B dd 20,21,22,23,24,25,26,27; B[0..7] of dd
.code
  vmovdqa ymm1,ymmword ptr A
  vmovdqa ymm2,ymmword ptr B
; ↓↓ ymm0[i]:=A[2*i]*B[2*i]+A[2*i+1]*B[2*i+1]
  vpmaddwd ymm0,ymm1,ymm2
```

ymm1	17	16	15	14	13	12	11	10
ymm2	27	26	25	24	23	22	21	20
ymm0	17*27+16*26		15*25+14*24		13*23+12*22		11*21+10*20	

К сожалению, команда целочисленного деления на векторных регистрах не реализована, уж очень сложной и медленной она получается. Предлагается вычислять частное по такой формуле (X,Y: longint):

X **div** Y=longint(single(X)*(1.0/single(Y)))

Для вычисления 1.0/Z предлагается использовать команду **vrcpps**. Необходимо, однако, учитывать, что ответ получается приближённым. А вот с вычислением остатка от деления дело обстоит ещё сложнее...

Интересны команды вычисления среднего арифметического:

vpavg{**b**,**w**} op1,op2,op3

Буква в конце кода операции задаёт размеры операндов: **b** → **byte**, **w** → **word**. Допустимые форматы операндов:

op1	**op2**	**op3**
r128	r128	r128, m128

r256	r256	r256, m256

Алгоритм выполнения команды:

```
{op1,2,3:array[31/15/7..0] of db/dw}
for i:=0 to 31/15/1 do
    op1[i]:=(op1[i]+op2[i]) rcr 1
```

Здесь для вычисления среднего арифметического вместо `(op1[i]+op2[i]) div 2`, что не всегда даёт правильный ответ, использовано эквивалентное выражение с операцией **rcr** (поворот через флаг переноса CF, т.е. в 9-ти или 17-ти битах). Флаг переноса CF предварительно сформирован командой сложения `op1[i]+op2[i]`.

17.2.4. Сравнение векторов целых чисел

Программирование – это неестественный процесс.

Алан Перлис,
первый лауреат премии Тьюринга

Как мы знаем, при сравнении двух целых чисел (например, командой **cmp**), устанавливаются флаги, которые фиксируют результат сравнения. При сравнении двух векторов целых чисел приходится делать отдельные команды сравнения на равенство, на больше и т.д., получая маску результатов. Маска позволяет соотнести каждому элементу вектора значение **false** или **true**.

Различают битовые маски, в которых каждый элемент вектора представлен одним битом, и векторные маски. Векторные маски по структуре совпадают со сравниваемыми векторами, но каждый элемент маски равен константам **false** (обычно это все ноли) и **true** (обычно все биты этого значения равны "1", или первый бит равен "1", а остальные нули). Так построенные маски можно использовать в некоторых командах, обрабатывающих данные на векторных регистрах. В современных моделях ЭВМ появились также и специальные *регистры масок*, о которых будет рассказано в разделе 17.6.

В простейших случаях битовая маска строится из знаковых бит элементов вектора целых или вещественных чисел. Вот построение битовой маски для байтового вектора:

```
vpmovmskb op1,op2
```

Допустимые форматы операндов:

op1	op2
r32, r64	r128, r256

В регистре общего назначения op1 строится маска из знаковых бит элементов вектора op2. Алгоритм выполнения команды:

```
op1:=0; {N=16(r128),32(r256)}
for i:=0 to N-1 do
    op1[i]:=SF(op2[i])
```

Для построения битовой маски по векторной с элементами типов **dd** или **dq** можно воспользоваться построением битовой маски для вектора вещественных чисел:

```
vmovmskp{s,d} op1,op2
```

Буквы **s** (single) и **d** (double) задают тип элемента вектора. Допустимые форматы операндов те же, что и в предыдущем примере. Алгоритм выполнения команды:

```
op1:=0; {N=2,4,8}
for i:=0 to N-1 do op1[i]:=SF(op2[i])
```

К сожалению, аналогичной команды для векторов с элементами типов **dw** нет, приходится получать байтовую маску, а потом с помощью маленькой программы с командами сдвига оставлять в этой маске только каждый второй бит. Например:

```
;   EAXbyte → EBXword
    mov ecx,16
    xor ebx,ebx
@@:shl eax,1
```

```
    rcr eax,1
    rcl ebx,1
    loop @B
```

В более сложных случаях векторная маска получается как результат сравнения двух векторов. Вот получение векторной маски как результат сравнения векторов знаковых целых чисел:

```
vpcmp{eq,gt}{b,w,d,q} op1,op2,op3
```

Обозначения операций отношения **eq** → равно, **gt** → больше. Буква в конце кода операции задаёт длину элемента вектора: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Допустимые форматы операндов

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

Эта команда вычисляет векторную маску и выполняется по такому алгоритму:

```
for i:=0 to N-1 do {N=32,16,8,4,2}
  if op2[i] {eq,gt} op1[i]
    then op1[i]:=-1 else op1[i]:=0
```

Например:

```
.data
  org 1000h
X db 32 dup (?); X[0..31] of db;
Y dd 8  dup (?); Y[0..7] of dd;
.code
; ↓↓ ymm0[0..31]:=ymm1[0..31]=X[0..31]
; ↓↓ eq → 0FFh, ne → 00h
  vpcmpeqb ymm0,ymm1,ymmword ptr X
; ↓↓ eq → 0FFFFFFFFh, ne → 0
  vpcmpeqd ymm2,ymm0,ymm1
; ↓↓ gt → 0FFh, le → 00h
  vpcmpgtb ymm0,ymm1,ymmword ptr X
; ↓↓ присвоить всем битам регистра 1
  vpcmpeq ymm0,ymm1,ymm1; ymm0:=-1
```

В качестве примера рассмотрим реализацию цикла

```
const N=100000000;
var X,Y,Z: array[1..8*N] of longint;
    i: longint;
begin {ввод массивов X и Y}
  for i:=1 to 8*N do
    if X[i]>Y[i] then Z[i]:=X[i]
                 else Z[i]:=Y[i]
```

После векторизации получается примерно такая программа:

```
.data?
  org 1000h
N equ 100000000
X oword 2*N dup (?)
Y oword 2*N dup (?)
Z oword 2*N dup (?)
.code
; ввод векторов X и Y
  mov ecx,N
  xor ebx,ebx; индекс массивов
L:vmovdqa ymm0,ymmword ptr X[ebx]; 8 очередных X[i]
  vmovdqa ymm1,ymmword ptr Y[ebx]; 8 очередных Y[i]
```

```
; ↓↓ ymm2[0..7]:=X[0..7]>Y[0..7]; это -1 и 0
  vpcmpGTd ymm2,ymm0,ymm1
; ↓↓ ymm0[0..7]:=ymm2[0..7] ? X[0..7] : Y[0..7]
  vpblendd ymm0,ymm0,ymm1,ymm2
  vmovdqa ymmword ptr Z[ebx],ymm0; 8 очередных Z[i]
  add ebx,32
  loop L
```

17.3. Арифметические команды для вещественных чисел

Когда вам покажется, что цель недостижима, не изменяйте цель – изменяйте свой план действий.

Конфуций, V век до н.э.

Как уже говорилось, на векторных регистрах можно выполнять как скалярные, так и векторные операции с вещественными числами.

17.3.1. Скалярные операции на векторных регистрах

Программы предназначены для чтения людьми и только случайно для выполнения компьютерами.

Дональд Эрвин Кнут

«Программирование как искусство»

Младшие части векторных регистров XMM могут использоваться для хранения скалярных вещественных чисел (форматов single и double) и выполнения над ними арифметических операций. Как уже говорилось, при этом чаще всего "портятся" и старшие части как регистров XMM, так и старшие части регистров YMM (и ZMM), это надо учитывать при программировании.

Сначала обмен между регистром и памятью:

```
vmov{d,q} op1,op2; op1:=op2
```

Буквы в окончании задают длину скалярного операнда: **d** → **dd**, **q** → **dq**. При записи скалярного операнда в младшую часть регистра его старшая часть обнуляется. Допустимые форматы операндов:

op1	op2
r128	r128, m32(**d**), m64(**q**)
m32(**d**), m64(**q**)	r128, r256
r256	r256, m32(**d**), m64(**q**)

Теперь трёхадресные скалярные операции *вещественной* арифметики (вспомним добрым словом учебную машину УМ-3 🙂). Обозначим знаком ⊗ двуместную арифметическую операцию:

```
v{⊗}s{s,d} op1,op2,op3; op1:=op2⊗op3
```

Буква **v** означает векторный регистр, а буква **s** – скалярную операцию, окончание задаёт тип скалярного операнда: **s** (**s**ingle), **d** (**d**ouble). ⊗ определяет операцию (**add**, **sub**, **mul**, **div**). Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m32(**s**), m64(**d**)

Примеры скалярных вещественных операций:

```
.data
  org 1000h
A real4 1.23
  org A+32
B real8 3.13
```

```
C real4 1.[1]
.code
  vmovd xmm1,A
  vmovq xmm2,B
; ↓↓ xmm0[0..31]:=xmm1[0..31]+A
  vaddss xmm0,xmm1,A
; ↓↓ xmm0[0..63]:=xmm2[0..63]-B
  vsubsd xmm0,xmm2,B
; ↓↓ xmm0[0..31]:=low(xmm1[0..31]*B)
  vmulss xmm0,xmm1,xmm3; (неполное) умножение
; ↓↓ xmm0[0..63]:=xmm1[0..63]/B
  vdivsd xmm0,xmm1,A; деление
; ↓↓ xmm0[0..63]:=sqrt(xmm2[0..63])
; ↓↓ xmm0[127..64]:=xmm1[127..64] ⚠
  vsqrtsd xmm0,xmm1,B; странный квадратный корень
```

Программирование со скалярными величинами похоже на работу в учебной машине УМ-3. Например, вычислим объём сферы V радиуса R по формуле $V=4\pi R^3/3$:

```
.data
  org 1000h
Pi real4 3.1415926535
T  real4 3.0
R  real4 ?
V  real4 ?
.code
; Здесь ввод R
  vmovd  xmm1,R
  vmulss xmm0,xmm1,xmm1; R^2
  vmulss xmm1,xmm0,xmm1; R^3
  vmulss xmm1,xmm1,Pi;   π*R^3
  vaddss xmm1,xmm1,xmm1; 2*π*R^3
  vaddss xmm1,xmm1,xmm1; 4*π*R^3
  vdivss xmm1,xmm1,T;    4*π*R^3/3
  vmovd  V,xmm1
```

Работа с вещественными числами на векторных регистрах управляется специальным регистром MXCSR. В нём есть битовые маски, разрешающие ("0") или блокирующие ("1") возникновение исключений для всех специальных случаев (в квадратных скобках номер бита в регистре):

- IM[7] – недействительное значением (NaN, 0/0 и т.д.);
- DM[8] – денормализованный операнд (denormalised);
- ZM[9] – деление на ноль; (divide by zero, ±∞, NaN);
- OM[10] – переполнение, (overflow, ±∞);
- UM[11] – антипереполнение (underflow, денормализованный результат);
- PM[12] – потеря точности (precision, 1.0/3.0);

> Обычно языки высокого уровня (и язык Free Pascal) считает стандартным режим, в котором все исключения <u>заблокированы</u>, т.е. управляющие биты установлены в единицы.

Вместе с тем, в регистре MXCSR есть и флаги, <u>фиксирующие</u> наступление исключительных ситуаций (они при этом устанавливаются в "1").

- IE[0] – недействительное значением (NaN, 0/0 и т.д.);
- DE[1] – денормализованный операнд (denormalised);

---

[1] Наш Ассемблер позволяет, следуя дурному примеру некоторых других языков, записывать вещественные числа без нулевой дробной части, т.е. вместо `1.0` писать просто `1.`, однако вместо `0.1` писать просто `.1` не разрешается ☹. Мы так делать не будем, это выглядит некрасиво.

ZE [2] – деление на ноль; (divide by zero, ±∞, NaN);
OE [3] – переполнение, (overflow, ±∞);
UE [4] – антипереполнение (underflow, денормализованный результат);
PE [5] – потеря точности (precision, 1.0/3.0);

Отметим и другие интересные флаги в этом регистре:

DAZ [6] – округлять денормализованные число до нуля;
FTZ [15] – при потере значимости округлять число до нуля;
RC[14..13] – режимы округления:
00 – к ближайшему чётному;
01 – в меньшую сторону (к –∞);
10 – в бóльшую сторону (к +∞);
11 – к нулю.

Чтение из регистра MXCSR в память производит команда

stmxcsr m32; <m32>:=MXCSR

Запись из памяти в регистр MXCSR производит команда

ldmxcsr m32; MXCSR:=<m32>

Команда **ldmxcsr** в основном используется, чтобы сбросить в ноль биты обнаружения аварийных ситуаций, так как после установки этих битов сам компьютер их никогда не сбрасывает. Заметим также, что биты 31..16 регистра MXCSR зарезервированы и должны всегда быть нулевыми.

Кроме того, прикладная программа может "попросить" операционную систему при возникновении определённого исключения вызвать свою собственную функцию обработки такого исключения (так называемую функцию обратного вызова).

Далее рассмотрим *скалярную* операцию сравнения двух вещественных чисел, хотя бы одно из которых располагается в младших разрядах векторного регистра xmm. Как и для целых чисел, команда сравнения двух таких вещественных чисел фиксирует результаты сравнения путём установки флагов в регистре EFLAGS. Рассмотрим команду:

[**u**]**comis**{**s**,**d**} op1,op2

В отличие от целых чисел, два вещественных числа могут быть и *несравнимы* между собой. Так будет, если хотя бы одно из этих чисел имеет неверный формат или равно NaN (Not a Number – не число, см. разд. 5.4 и дополнительный материал в конце этого раздела). Буква **u** (**u**nordered) говорит о том, что эта команда вызывает исключительную ситуацию (выставляя флаг IM в регистре MXCSR), когда хотя бы один операнд SNaN. Отсутствие буквы **u** делает команду "более строгой", вызывая исключительную ситуацию как по значению SNaN ("громкое" NaN), так и по QNaN ("тихое" NaN).

Первая из букв **s** означает скалярную операцию, окончания задают тип скалярных операндов: **s** (**s**ingle), **d** (**d**ouble). Допустимые форматы операндов:

op1	**op2**
r128	r128, m32(**s**), m64(**d**)

По результату операции сравнения устанавливаются флаги, выбор и трактовка которых существенно отличается от случая целочисленного сравнения. По результатам сравнения устанавливаются три флага: CF (флаг переполнения), ZF (флаг нуля) и PF (флаг чётности) [1]:

ZF	PF	CF	отношение
0	0	0	op1 > op2
0	0	1	op1 < op2
1	0	0	op1 = op2
1	1	1	op1 несравним с op2

[1] Флаг чётности PF:=1 по результатам этой операции установится только тогда, когда маска IM=1, т.е. заблокировано возникновение исключительной ситуации по недействительному операнду. Напомним, что при сравнении *целых* чисел PF:=1, когда в последнем байте результата содержится *чётное* число бит со значением "1".

Кроме того, команды вещественного сравнения всегда обнуляют флаги OF, AF и SF. Получается, что переход по не сравнимым операндам имеет код операции **jp**, а для остальных случаев надо использовать команды условных переходов, как для беззнаковых целых чисел:

ja, jnbe	op1 > op2
jae, jnb	op1 >= op2
jb, jnae	op1 < op2
jbe, jna	op1 <= op2
jz, je	op1 = op2
jnz, jne	op1 <> op2

Понятно, что использование флагов можно применять, когда сравниваются два числа, а когда сравниваются два вектора чисел, этот метод непригоден. Для векторов нам придётся не просто сравнивать величины между собой, а проверять конкретную операцию отношения (равно, больше и т.д.), а результат сравнения будет вектором логических констант. Скалярная величина на векторном регистре рассматривается как частный случай вектора из одного элемента, поэтому команда скалярного сравнения (как и векторного) формирует результат в первом операнде. Например, рассмотрим *четырёхадресную* команду:

```
vcmps{s,d} op1,op2,op3,op4
```

Буква **s** означает скалярную операцию, окончания задают тип скалярных операндов: **s** (**s**ingle), **d** (**d**ouble). Допустимые форматы операндов:

op1	op2	op3	op4
r128	r128	r128, m32(**s**), m64(**d**)	i8

В младшую часть первого операнда записывается результат сравнения **true** и **false** в виде значений 80000000h и 0h (для типа **s**ingle) и 8000000000000000h и 0h (для типа **d**ouble). В четвёртом операнде i8[2..0] задаётся проверяемое отношение между операндами op2 и op3:

i8	<op2> отношение <op3>	Псевдокод
0	**eq** – равно	**vcmpEQs**{**s**,**d**}
1	**lt** – меньше	**vcmpLTs**{**s**,**d**}
2	**le** – меньше или равно	**vcmpLEs**{**s**,**d**}
3	**unord** – несравнимы	**vcmpUNORDs**{**s**,**d**}
4	**ne** – неравно	**vcmpNEQs**{**s**,**d**}
5	**nlt**=**ge** – больше или равно	**vcmpNLT**t**s**{**s**,**d**}
6	**nle**=**gt** – больше	**vcmpNLEs**{**s**,**d**}
7	**ord** – сравнимы	**vcmpORDs**{**s**,**d**}

Последняя колонка содержит псевдокод команды, внутрь кода операции включена мнемоника проверяемого отношения, таким образом, вместо команды:

```
vcmpss ymm1,ymm2,ymm3,4
```

можно писать

```
vcmpNEss ymm1,ymm2,ymm3
```

Вообще говоря, операция сравнения выполняется компьютером как вычитание и сравнение результата с нулём. Здесь необходимо отметить, что выполнение всех операций с вещественными числами имеет свою, порой трудно понимаемую, специфику. Сравнение выработает результат "несравнимы", если хотя бы один операнд имеет ошибочный формат, либо равен специальному значению NaN (Not a Number – "не число"). В нашей машине, однако, есть два вида таких "не чисел": QNaN (quite – "тихое" не число) и SNaN (signaling – "громкое" не число).

Любая операция с использованием SNaN приводит к исключительной ситуации (если она разрешена, т.е. в регистре MXCSR бит IM=0) и к выдаче ответа QNaN (если исключение запрещено, а обычно так и делается, так как его всё равно никто не обрабатывает 🐷). Таким образом, достаточно в программе появиться одному NaN, как они начинают быстро размножаться (так называемое "отравление" программы NaN). Далее следующая засада: любое сравнение, где участвует NaN, возвращает **false**, даже сравнение NaN с NaN (любого вида) ⚠. Любители алгебры могут только сожалеть, что у нас вещественные числа не образуют ни поля, ни кольца, ни даже группы... ☹

Исходя из сказанного, относитесь с осторожностью к описанию команд вещественной арифметики. Когда говорится, что операция возвращает минимум из двух чисел, знайте, что никакого "минимума" эта команда может и не вернуть (далее будет такой пример, см. разд. 17.3.3). В языках высокого уровня предусмотрены специальные константы и функции, проверяющие значение на "не число", например, в языке Free Pascal это константа `const Nan=0.0/0.0` и стандартная логическая функция `IsNan(x)`.

17.3.2. Команды сложения и вычитания векторов вещественных чисел

Как только вы поняли, как писать программу, заставьте сделать это кого-нибудь другого.

Алан Перлис,
первый лауреат премии Тьюринга

Команды сложения и вычитания *трёхадресные*:

```
v{add,sub}p{s,d} op1,op2,op3; op1:=op2⊗op3
```

Буква **p** (**p**acked) обозначает упакованный вектор, буква в конце кода операции задаёт тип элемента упакованного вектора: **s** (`single`) или **d** (`double`). Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

```
.data
  org 1000h
A real8 10.0,11.1,12.2,13.3; A[0..3] of double;
B real8 20.0,21.1,22.2,23.3; B[0..3] of double;
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vmovdqa ymm2,xmmword ptr B; ymm2:=B
; ↓↓ ymm0[0..3]:=ymm1[0..3]+ymm2[0..3]
  vaddpd ymm0,ymm1,ymm2
```

ymm1	13.3	12.2	11.1	10.0
ymm2	23.3	22.2	21.1	20.0
ymm0	36.6	33.4	33.2	30.0

Представляет интерес команда

```
vaddsubp{s,d} op1,op2,op3
```

Для элементов на чётных позициях вектора (0, 2, 4...) выполняются операции вычитания, а для нечётных позиций – сложения:

```
; ↓↓ ymm0[0..3]:=ymm1[0..3]⊗ymm2[0..3]
  vaddsubpd ymm0,ymm1,ymm2
```

ymm1	13.3	12.2	11.1	10.0
ymm2	23.3	22.2	21.1	20.0
ymm0	13.3+23.3	12.2-22.2	11.1+21.1	10.0-20.0

Полезны команды вычисления попарного максимума и минимума элементов векторов:

```
v{max,min}p{s,d} op1,op2,op3; op1:=op2⊗op3
```

Как уже говорилось в разделе, описывающем скалярные операции с вещественными числами, существование значений NaN, ±0.0 и ±∞ приводит к существенному усложнению семантики таких операций. Например, рассмотрим команду поиска минимума

```
vminpd op1,op2,op3; op1:=min(op2,op3)
```

Алгоритм выполнения этой команды:

```
if (op2=0.0) and (op3=0.0) then op1:=op3 else
if op2=NaN then op1:=op3 🙄 else
```

```
if op3=NaN then op1:=op3 else
if op2<op3 then op1:=op2 else op1:=op3
```

Видно, что это немного "странный" минимум, например:

min(-0.0,+0.0)=+0.0

{ ↓↓ Операция min некоммутативная! }

min(1.0,NaN)=NaN; min(NaN,1.0)=1.0 ⚠

{ Если надо, чтобы min(NaN,1.0)=NaN, то надо вызывать стандартную функцию, какую-нибудь MINPS }

min(0.0,-∞)=-∞; min(-∞,SNaN)=SNaN

Далее, как и для целых чисел, доступны команды *горизонтального* сложения и вычитания:

vh{**add**,**sub**}**p**{**s**,**d**} op1,op2,op3; op1:=op2⊗op3

Например:

```
; ↓↓ ymm0[0]:=ymm1[0]+ymm1[1]; ymm0[2]:=ymm1[2]+ymm1[3]
; ↓↓ ymm0[1]:=ymm2[0]+ymm2[1]; ymm0[3]:=ymm2[2]+ymm2[3]
  vhaddpd ymm0,ymm1,ymm2
```

ymm1	13.3	12.2	11.1	10.0
ymm2	23.3	22.2	21.1	20.0
ymm0	22.2+23.3	12.2+13.3	20.0+21.1	10.0+11.1

Современные языки высокого уровня позволяют практически "напрямую" вставлять в программу машинные команды для выполнения векторных операций. Для этого используются так называемые интрисики (intrinsic function), мнемонические обозначения ассемблерных команд, по сути, это маленькие встраиваемые (inline) функции. Например, пусть в языке Free Pascal есть описания

```
type V=array[0..3] of double;
var x,y,z: V;
```

и надо выполнить операцию сложения векторов вещественных чисел

```
vaddpd z,x,y
```

Тогда можно написать встраиваемую функцию

```
funcnion _vaddpd(var x,y: V)z: V; inline;
begin asm
  vmovdqa ymm0,ymmword ptr x
  vaddpd  ymm0,ymm0,ymmword ptr y
  vmovdqa ymmword ptr z,ymm0
end;
```

Или/и встраиваемую процедуру

```
procedure _vaddpd(var x,y,x: V); inline;
begin asm
  vmovdqa ymm0,ymmword ptr x
  vaddpd  ymm0,ymm0,ymmword ptr y
  vmovdqa ymmword ptr z,ymm0
end;
```

После этого можно использовать в программе оператор присваивания z:=**_vaddpd**(x,y) или вызывать процедуру **_vaddpd**(x,y,z); Многие языки включают целые пакеты таких функций и используемых в них типов данных, покрывающие практически все векторные команды ЭВМ. Например, в языке C/C++ в векторной библиотеке определены типы

__m256 эквивалентно **array**[0..7] **of** single;
__m256d эквивалентно **array**[0..3] **of** double;
__m256i эквивалентно на нашем Ассемблере

```
__m256i union
   i8  db 32 dup (?)
   i16 dw 16 dup (?)
```

```
        i32 dd   8 dup (?)
        i64 dq   4 dup (?)
    __m256i ends
и т.д.
```

17.3.3. Команды умножения и деления векторов вещественных чисел

Чтобы понять программу, необходимо отождествить себя и с машиной, и с программой.

Алан Перлис
первый лауреат премии Тьюринга

Команды умножения и деления *трёхадресные*:

v{**mul**,**div**}**p**{**s**,**d**} op1,op2,op3; op1:=op2⊗op3

Буква **p** (**p**acked) обозначает упакованный вектор**,** буква в конце кода операции задаёт тип элемента упакованного вектора: **s** (single) или **d** (double). Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

```
.data
  org 1000h
A real8 10.0,11.1,12.2,13.3; A[0..3] of double;
B real8 20.0,21.1,22.2,23.3; B[0..3] of double;
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vmovdqa ymm2,ymmword ptr B; ymm2:=B
; ↓↓ ymm0[0..3]:=ymm1[0..3]*ymm2[0..3]
  vmulpd ymm0,ymm1,ymm2
```

ymm1	13.3	12.2	11.1	10.0
ymm2	23.3	22.2	21.1	20.0
ymm0	13.3*23.3	12.2*22.2	11.1*21.1	10.0*20.0

Отдельно стоит описать *трёх-* и *четырехадресные* команды FMA (FusedMultiply-Add), совмещающие операции умножения и сложения вещественных чисел. По некоторым оценкам, использование таких команд вместо соответствующих пар команд умножения и сложения может повысить скорость счёта примерно на 30%, так как операция сложения, по сравнению с умножением, выполняется "почти бесплатно". Кроме того, повышается и точность вычислений.

Рассмотрим трёхадресную команду:

vfm{**add**,**sub**}{xxx}**p**{**s**,**d**} op1,op2,op3

Обозначим для краткости op1=X, op2=Y, op3=Z. Тогда для xxx=123 → X:=Z±X*Y; 213 → X:=Z±Y*X; 231 → X:=X±Y*Z; 321 → X:=X±Z*Y и т.д. Буквы **s** (**s**ingle) и **d** (**d**ouble) задают размер элемента упакованного вещественного вектора.

Допустимые форматы операндов

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

Модификация этих команд:

vfnm{**add**,**sub**}{xxx}**p**{**s**,**d**} op1,op2,op3

при вычислении результата берут произведение со знаком "минус": X:=-X*Y±Z и т.д. Аналогичные команды:

vfm{**addsub**,**subadd**}{xxx}**p**{**s**,**d**} op1,op2,op3

при вычислении результата чередует команды сложения на нечётных позициях вектора с командами вычитания на чётных позициях (**addsub**), или, наоборот, команды сложения на чётных позициях вектора с командами вычитания на нечётных позициях (**subadd**).

Такие же *четырёхадресные* команды, относящиеся к так называемому набору команд FMA4 (поддерживается далеко не всеми компьютерами):

vfm{**add,sub,addsub,subadd**}{xxx}**p**{**s,d**} op1,op2,op3,op4

Допустимые форматы операндов

op1	op2	op3	op4
r128, m128	r128	r128	r128, m128
r256, m256	r256	r256	r256, m256

В этих командах результат может не портить операнды. Снова обозначим для краткости op1=A, op2=X, op3=Y, op4=Z. Тогда для xxx=123 → A:=X*Y+Z и т.д.

Двухадресная команда:

vrcpps op1,op2

Вычисляет обратные величины элементов вещественного вектора op2 и записывает ответ в op1. Допустимые форматы операндов:

op1	op2
r128	r128, m128
r256	r256, m256

Примерная точность вычислений $1.5*2^{-12}$, режимы округления из поля RC регистра MXCSR не используются. Алгоритм выполнения команды немного необычен:

```
for i:=0 to 7 do
  if op2[i]=±0.0 then op1[i]:=±∞ else
  if {op2[i] денормализован}
     then op1[i]:=±0.0
     else op1[i]:=1.0/op2[i]
```

А вот команда вычисления обратной величины квадратного корня от вещественных чисел одинарной точности:

vrsqrtps op1,op2; op1[i]

Примерная точность вычислений $1.5*2^{-12}$, режимы округления из поля RC регистра MXCSR не используются. Допустимые форматы операндов

op1	op2
r128	r128, m128
r256	r256, m256

По аналогии со скалярными вещественными операциями, определены и соответствующие векторные операции. Например, вот команда вычисления квадратного корня:

vsqrtp{**s,d**} op1,op2,op3

> Команд для вычисления тригонометрических функций, экспоненты, логарифма и т.д. не предусмотрена, но есть стандартная библиотека SVML (Short Vector Math Library), содержащая inline-функции для вычисления этих величин.

7.3.4. Задача вычисления произведения матриц

Все усилия тщетны, если студент не будет практиковаться в написании программ, поскольку навык программирования (как, впрочем, и всякий навык) даётся только практикой.

Чарльз Уэзерелл
«Этюды для программистов»

В качестве примера рассмотрим программу вычисления произведения матриц. На векторных регистрах YMM можно эффективно вычислять произведения матриц 4x4 с элементами типа double и 8x8 с элементами типа single. Итак, надо вычислить C:=A×B:

$$\begin{vmatrix} C_{00} & C_{01} & C_{02} & C_{03} \\ C_{10} & C_{11} & C_{12} & C_{13} \\ C_{20} & C_{21} & C_{22} & C_{23} \\ C_{30} & C_{31} & C_{32} & C_{33} \end{vmatrix} = \begin{vmatrix} A_{00} & A_{01} & A_{02} & A_{03} \\ A_{10} & A_{11} & A_{12} & A_{13} \\ A_{20} & A_{21} & A_{22} & A_{23} \\ A_{30} & A_{31} & A_{32} & A_{33} \end{vmatrix} \times \begin{vmatrix} B_{00} & B_{01} & B_{02} & B_{03} \\ B_{10} & B_{11} & B_{12} & B_{13} \\ B_{20} & B_{21} & B_{22} & B_{23} \\ B_{30} & B_{31} & B_{32} & B_{33} \end{vmatrix}$$

Обозначим строки этих матриц как $\overline{C}_0..\overline{C}_3$, $\overline{A}_0..\overline{A}_3$ и $\overline{B}_0..\overline{B}_3$ соответственно. Как Вы знаете, классическая формула вычисления произведения матриц предписывает производить скалярное произведение строки матрицы A на столбец матрицы B. Элементы столбца матрицы, однако, располагаются в памяти ЭВМ не подряд, что вызовет трудности в умножении на векторных регистрах. Изменим классическую формулу вычисления произведения матриц:

$$\overline{C}_0 = A_{00}*\overline{B}_0 + A_{01}*\overline{B}_1 + A_{02}*\overline{B}_2 + A_{03}*\overline{B}_3$$
$$\overline{C}_1 = A_{10}*\overline{B}_0 + A_{11}*\overline{B}_1 + A_{12}*\overline{B}_2 + A_{13}*\overline{B}_3$$
$$\overline{C}_2 = A_{20}*\overline{B}_0 + A_{21}*\overline{B}_1 + A_{22}*\overline{B}_2 + A_{23}*\overline{B}_3$$
$$\overline{C}_3 = A_{30}*\overline{B}_0 + A_{31}*\overline{B}_1 + A_{32}*\overline{B}_2 + A_{33}*\overline{B}_3$$

Как видим, теперь надо умножать вектор из одинаковых элементов из матрицы A на строки матрицы B, например (A_{00}, A_{00}, A_{00}, A_{00})*(B_{00}, B_{01}, B_{02}, B_{03}) и т.д. Наша программу во время своей работы будет хранить векторы $\overline{B}_0..\overline{B}_3$ на регистрах ymm0..ymm3 соответственно. Вычисление строки матрицы C оформим в виде макроопределения:

```
StrMat macro Ci,Ai
   pxor xmm5,xmm5; C̄i=ymm5:=0
for j,<0,1,2,3>
   vpbroadcastq ymm4,Ai+8*i; ymm4[3..0]:=Aij
   vfmadd321pd ymm5,ymm4,ymm&j; C̄i:=C̄i+Aij*B̄j
endm
;   C̄i:=Ai0*B̄0+Ai1*B̄1+Ai2*B̄2+Ai3*B̄3
   vmovdqa Ci,ymm5
   endm
```

А теперь напишем саму программу:

```
.data?
  org 1000h
A real8 16 dup (?); A[0..3,0..3] of double;
B real8 16 dup (?); B[0..3,0..3] of double;
C real8 16 dup (?); C[0..3,0..3] of double;
.code
; здесь ввод (получение) матриц A и B
; загрузка строк матрицы B на регистры  ymm0..ymm3:
for i,<0,1,2,3>
    vmovdqa ymm&i,ymmword ptr B+32*i; ymmi:=B̄i
endm
; вычисление строк матрицы C
for i,<0,1,2,3>
    StrMat ymmword ptr C+32*i,A+32*i
endm
```

Задание для самостоятельной работы. Надо исправить эту программу так, чтобы она вычисляла произведение матриц 8x8 из чисел одинарной точности.

17.3.5. Сравнение векторов вещественных чисел

Один день ты перестанешь учиться, и ты начнёшь умирать.

Альберт Эйнштейн

По аналогии со *скалярным* сравнением вещественных чисел (см. разд. 17.1.3) для сравнения двух упакованных *векторов* вещественных чисел служит команда:

vcmpp{**s**,**d**} op1,op2,op3,op4

Буква в конце кода операции задаёт размеры операндов: **s** (single), **d** (double). Операнд op2 сравнивается с op3, отношение сравнения задаётся 4-мя битами в op4=i8[4..0]. Результат сравнения записывается в op1 в виде *векторной* маски (запись результата сравнения вещественных векторов в виде *битовой* маски описана в разд. 17.6.). В качестве результата сравнения двух элементов упакованных векторов в векторной маске выбраны значения из всех нулей (**false**) и всех единиц (**true**). Допустимые форматы операндов:

op1	op2	op3	op4
r128	r128	r128, m128	i8
r256	r256	r256, m256	i8

В четвёртом операнде i8[4..0] задаётся проверяемое отношение между операндами op2 и op3. Пять бит i8[4..0] позволяют закодировать 32 отношения между вещественными числами большая часть из них весьма специфические, практический интерес представляют лишь первые 8:

i8	<op2> отношение <op3>	Псевдокод
0	**eq** – равно	**vcmppEQs**{**s**,**d**}
1	**lt** – меньше	**vcmppLTs**{**s**,**d**}
2	**le** – меньше или равно	**vcmppLEs**{**s**,**d**}
3	**unord** – несравнимы	**vcmppUNORDs**{**s**,**d**}
4	**neq** – неравно	**vcmppNEQs**{**s**,**d**}
5	**nlt**=**ge** – больше или равно	**vcmppNLTs**{**s**,**d**}
6	**nle**=**gt** – больше	**vcmppNLEs**{**s**,**d**}
7	**ord** – сравнимы	**vcmppORDs**{**s**,**d**}

Последняя колонка содержит псевдокод команды, внутрь кода операции включена мнемоника проверяемого отношения, таким образом, вместо 4-х адресной команды:

vcmpps ymm1,ymm2,ymm3,4

можно писать 3-х адресную команду (наличие op4=i8 теперь будет ошибкой):

vcmppNEQs ymm1,ymm2,ymm3

Команда выполняется по такому алгоритму (символом ⊗ обозначено логическое отношение между операндами):

```
for i:=0 to N-1 do {N=8,4,2}
  if op2[i]⊗op3[i]
    then op1[i]:=-1 else op1[i]:=0
```

Отношения с номерами, большими первых 8 базовых, относятся к особым режимам сравнения операндов, среди которых встречаются "не числа", кроме того, отношение определяет, вызывать ли исключение по значению QNaN. Вот начало этой таблицы отношений:

i8	X <отношение> Y	X>Y	X<Y	X=Y	**unord**	IA *
0	**eq**, **eq_OQ** {=, **orded**, **quite**}	F	F	T	F	No
1	**lt**, **lt_OS** {<, **orded**, **signal**}	F	T	F	F	Yes
2	**le**, **le_OS** {<=, **orded**, **signal**}	F	T	T	F	Yes
3	**unord**, **unord_Q** {**unord**, **quite**}	F	F	F	T	No
4	**neq**, **neq_UQ** {<>, **unord**, **quite**}	T	T	F	T	No
5	**nlt**, **nlt_US** {>=, **unord**, **signal**}	T	F	T	T	Yes
6	**nle**, **nle_US** {>, **unord**, **signal**}	T	F	F	T	Yes
7	**ord**, **ord_Q** {**orded**, **quite**}	T	T	T	F	No
8	**eq_UQ** {=, **unord**, **quite**}	F	F	T	T	No

9	**nge_US**	{<, **unord**, **signal**}	F	T	T	T	Yes
10	**ng_US**	{<=, **unord**, **signal**}	F	T	T	T	Yes
11	**false_OQ**	{**false**, **orded**, **quite**}	F	F	F	F	No
12	**neq_OQ**	{<>, **orded**, **quite**}	T	T	F	F	No
13	**ge_OS**	{>=, **orded**, **signal**}	T	F	T	F	No
14	**gt_OS**	{>, **orded**, **signal**}	T	F	F	F	No
15	**true_UQ**	{**true**, **unord**, **quite**}	T	T	T	T	No
16	**eq_OS**	{=, **orded**, **signal**}	F	F	T	F	Yes
17	**lt_OQ**	{<, **orded**, **quite**}	F	T	F	F	No
18	**le_OQ**	{<=, **orded**, **quite**}	F	T	T	F	No
		. . .					

{**O**,**U**}{**Q**,**S**} – {**orded**, **unorded**}{**quite**, **signal**}

IA* – "жёсткое" сравнение, т.е. исключение и по QNan

17.3.5. Условное суммирование элементов вектора вещественных чисел

Человек может познать даже то, что ему не под силу себе представить.
Лев Давидович Ландау

Модифицируем задачу суммирования вектора целых чисел из раздела 17.2.2:

```
const N=1000000;
var X: array[0..4*N-1] of single;
    Y: double; S: double;
S:=0.0;
for i:=0 to 4*N-1 do
    if X[i]<Y then S:=S+X[i]
```

Итак, теперь у нас массив вещественных чисел и в сумму войдут не все элементы, а только удовлетворяющие определённым условиям (у нас просто X[i]<Y). Для суммы отведём переменную S, вдвое большую по размеру, чем элементы массива. После векторизации получится такая программа:

```
.data
  org 1000h
N equ 1000000
X real4 4*N dup (?); X[0..4*N-1] of single;
Y dq  ?;      Для хранения маски X[i]<Y
  org Y+32
S real8 0.0; S:=Сумма(X[i]<Y)
.code
; ↓↓ ymm0=4 частичных суммы double
  pxor xmm0,xmm0; 0.0,0.0,0.0,0.0
; ↓↓ ymm1[3..0]:=Y,Y,Y,Y
  vpbroadcastq ymm1,Y
  mov ecx,N; для цикла
  sub ebx,ebx; индекс массива i:=0
; ↓↓ ymm2[0..3]:=double(X[i..i+3])
L:vcvtps2pd ymm2,xmmword ptr X[ebx]
; построение векторной маски ymm3
; ↓↓ ymm3[0..3]:=(ymm2[0..3]<Y)
  vcmppLTd ymm3,ymm2,ymm1
; ymm3[0..3]:=ymm2[0..3] and ymm3[0..3]
; ↓↓ if ymm2[i]>=Y then ymm2[i]:=0.0
  vpand ymm2,ymm2,ymm3
; ↓↓ +очередные 4 элемента X в ymm0
  vaddpd ymm0,ymm0,ymm2
  add ebx,16; i:=i+4
```

```
  loop L
; ymm0:=[ymm0[3]],[ymm0[2]],[ymm0[1]],[ymm0[0]]
; ↓↓ ymm1:=[?],[?],[ymm0[3]],[ymm0[2]]  [?] – не важно
  vextracea128 xmm1,ymm0,[1]
; ↓↓ ymm0:=[?],[?],[ymm0[3]+ymm0[1]],[ymm0[2]+ymm0[0]]
; vaddpd ymm0,ymm1,ymm0
; ↓↓ ymm1:=[?],[?],[?],[ymm0[3]+ymm0[1]]
  pextrq rax,ymm0,1; rax:=ymm0[3]+ymm0[1]
  pinsrq ymm1,rax,0; ymm1[0]:=[ymm0[3]+ymm0[1]]
; ↓↓ ymm0:=[?],[?],[?],[ymm0[0]+ymm0[1]+ymm0[2]+ymm0[3]]
  vaddpd ymm0,ymm0,ymm1
  pextrq S,ymm0,0; S:=Сумма(X[i]<Y)
```

Как видим, вместо условного суммирования элементов массива выгоднее просто обнулять "ненужные" элементы перед суммированием.

17.4. Преобразование типов

Существует множество вещей, с которыми мы свыкаемся, не понимая их.
Айзек Азимов. «Конец Вечности»

Программисту часто приходится преобразовывать данных, с которыми работают команды компьютера, из одного типа в другой. На языках высокого уровня для этого используются явные (и неявные) преобразования типов, например, вот на языке Free Pascal:

```
var X: longint; Y: double;
    Y:=X { Это Y:=double(X) };
    X:=longint(trunc(Y));
```

Раньше единственной поддержкой для таких преобразований в языке машины были только команды обмена данными с вещественными регистрами `st(0)-st(7)`. При загрузке целых и вещественных чисел на такие регистры производилось (аппаратное) преобразование в "естественный" для этих регистров вещественный тип `extended`. Для вещественных чисел предусмотрено 3 кода операций (с общей мнемоникой `FLD`) для загрузки, например, на вершину стека `st(0)` чисел типов `single`, `double` и `extended`. Соответственно, предусмотрено (с общей мнемоникой `FILD`) три команды для загрузки целых (знаковых) чисел форматов `smallint`, `longint` и `int64`. Отметим, что загрузка беззнаковых чисел форматов `word`, `longword` и `qword` на вещественные регистры не предусмотрена. Например:

```
fld dword ptr [ebx]; st(0):=extended(single)
fild word ptr [ebx]; st(0):=extended(smallint)
```

И, наоборот, при записи с вещественных регистров в память производилось (аппаратное) преобразование в нужный вещественный или целый тип. Для вещественных чисел предусмотрено 3 кода операций (с общей мнемоникой `FST[P]`) для записи в память вещественного числа, с преобразованием в типы `single` и `double` (тип `extended` записывается без преобразования). Соответственно, предусмотрено (с общей мнемоникой `FIST[P]` [1]) три команды для записи в память целых (знаковых) чисел форматов `smallint`, `longint` и `int64`. Например:

```
fstp dword ptr [ebx]; [ebx]:=single(st(0))
fistp dword ptr [ebx]; [ebx]:=longint(st(0))
```

Для векторных регистров существуют *скалярные* преобразования типов (в преобразовании участвует только одно левое число векторного регистра), и *векторные* преобразования типов (участвуют все числа регистра). Сначала рассмотрим скалярные преобразования типов, вот двухадресные команды для преобразования данных между целыми и вещественными типами:

```
vcvt{sourse}2{destination} op1,op2
```

Когда преобразование невозможно, то возникает соответствующее исключение, а если это исключение замаски́ровано, то возвращается целочисленное значение `-1`, т.е. 2^w-1 (w=32 или 64). При пре-

[1] При использовании буквы `P` число удаляется из стека, такие тонкости мы здесь обсуждать не будем.

образовании из вещесвенного в целое способ такого преобразования задаётся в регистре MXCSR (см. разд. 17.3.1).[1] Допустимые форматы операндов:

op1	op2
r32	r128, m64
r64	r128, m64

```
; var op1: {longint,longword}/{int64,qword};
;     op2: single/double;
; [unsign] scalar int := scalar single/double
; ↓↓ op1:=longword(op2)/qword(op2)
  vcvts{s,d}2{,u}si op1,op2
; var op1: single/double;
;     op2: {longint,longword}/{int64,qword};
; scalar single/double := [unsign] scalar int
; ↓↓ op1:=single/double(op2)
  vcvt{ ,u}si2s{s,d} op1,op2
```

А теперь преобразование между <u>вещественными</u> типами:

```
; var op1: double; op2: single;
; ↓↓ scalar double := scalar single
  vcvtss2sd op1,op2; op1:=double(op2)
; var op1: single; op2: double;
; ↓↓ scalar single := scalar double
  vcvtsd2ss op1,op2; op1:=single(op2)
```

Разумеется, аналогичные команды есть и для преобразования типов данных упакованных векторов. Как Вы можете догадаться, в их кодах операций будет буква **p** (**P**acked). Некоторой проблемой векторных операций является изменение размера элемента векторного регистра, поэтому иногда операндами могут быть векторные регистры разной длины, например, xmm → ymm или ymm → xmm. Сначала преобразование <u>знаковых</u> целых чисел в вещественные числа (для <u>беззнаковых</u> целых есть только <u>скалярные</u> преобразования, описанные выше):

```
; ↓↓ op1[0..7/3] of single:=op2[0..7/3] of longint
      vcvtdq2ps op1,op2; длины операндов совпадают
```

Допустимые форматы операндов:

op1	op2
r128	r128, m128
r256	r256, m256

```
; ↓↓ op1[0..3/1] of double:=op2[0..3/1] of longint
; ↓↓ op1 в два раза больше, чем op2
      vcvtdq2pd op1,op2; dd → dq
```

Допустимые форматы операндов:

op1	op2
r128	r128, m64
r256	r128, m128

А теперь преобразование из вещественного типа в целый тип:

```
; ↓↓ op1[0..7/3] of longint:=op2[0..7/3] of single
      vcvtps2dq op1,op2; длины операндов совпадают
```

Допустимые форматы операндов:

op1	op2
r128	r128, m128
r256	r256, m256

[1] Есть ещё команды преобразования из вещественных чисел в целые

```
rounds{s,d} op1,op2,i8
```

Здесь в третьем операнде i8 прямо задаются применяемые правила преобразования, мы эти команды рассматривать не будем.

```
; ↓↓ op1[0..3/1] of longint:=op2[0..3/1] of double
; ↓↓ op1 в два раза меньше, чем op2
      vcvtpd2dq op1,op2; dq→dd
```

Допустимые форматы операндов:

op1	op2
r128	r128, m128
r128	r256, m256

Далее преобразования между вещественными типами:

```
; ↓↓ op1[0..3/1] of double:=op2[0..3/1] of single
; ↓↓ op1 в два раза больше, чем op2
      vcvtps2pd op1,op2; dd→dq
```

Допустимые форматы операндов:

op1	op2
r128	r128, m128
r256	r128, m128

```
; ↓↓ op1[0..3/1] of single:=op2[0..3/1] of double
; ↓↓ op1 в два раза меньше, чем op2
      vcvtpd2ps op1,op2; dq→dd
```

Допустимые форматы операндов:

op1	op2
r128	r128, m128
r128	r256, m256

При преобразовании из одного типа в другой в регистре MXCSR могут подниматься флаги ошибок:

IE [0] – недействительное значением (NaN, 0/0 и т.д.);
DE [1] – денормализованный операнд (denormalised);
OE [3] – переполнение (overflow, ±∞);
UE [4] – антипереполнение (underflow, денормализованный результат);
PE [5] – потеря точности (precision, 1.0/3.0).

В надёжных программах программист должен после команд преобразования типов сам читать регистр MXCSR (командой **stmxcsr** m32), проверять и сбрасывать в ноль поднятые флаги ошибок (сами эти флаги никогда не сбрасываются), затем записывать регистр MXCSR назад командой **ldmxcsr** m32.

17.5. Логические команды

Познание начинается с удивления.
Аристотель, IV век до н.э.

Для работы на векторных регистрах предоставляются логические команды двух видов. Первые, как обычно, рассматривают свои операнды просто как битовые строки, не имеющие внутренней структуры. Обозначим знаком ⊗ двуместную логическую операцию **or**, **and**, и **xor**:

```
      vp{or,and,xor} op1,op2,op3; op1:=op2⊗op3
```

Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128
r256	r256	r256, m256

Например:

```
.data
  org 1000h
X oword ?,?; X[0..255] of boolean;
Y oword ?,?
.code
```

```
  vmovdqa ymm1,ymmword ptr X
  vmovdqa ymm2,ymmword ptr Y
  vpor   ymm0,ymm1,ymmword ptr X; ymm0:=ymm1 or X
  vpand ymm1,ymm0,ymm2; ymm1:=ymm0 and ymm2
  vpxor ymm0,ymm2,ymmword ptr Y; ymm1:=ymm0 xor Y
; ↓↓ ymm0[255..128]:=0
  vpor   xmm0,xmm1,xmm1; xmm0:=xmm1 or xmm1
; ↓↓ штрих Шеффера nand: X|Y=not(X and Y)
  vpandn ymm0,ymm1,ymm2
```

У команды

vp{**or**,**and**,**xor**} op1,op2,op3; op1:=op2⊗op3

есть синонимы

vp{**or**,**and**,**xor**}{**d**,**q**} op1,op2,op3; op1:=op2⊗op3

Используя эти команды, программист рассматривает операнды этих команд как вектора битовых последовательностей, например, как 4***dq** или 4***dd**.

К сожалению, никакие флаги эти логические команды не устанавливают.

Среди логических команд флаги устанавливает только команда тестирования:

vptest op1,op2

Допустимые форматы операндов

op1	op2
r128	r128, m128
r256	r256, m256

Операнды рассматриваются просто как последовательности битов. Устанавливаются наши любимые флаги: ZF:=(op1 **and** op2)≡0 и CF:=(**not** op1 **and** op2)≡0 (что необычно, и не путать с операцией **nand**, она же штрих Шеффера). Флаги AF, OF, PF, и SF обнуляются.

Отрицание **not** для векторных регистров не реализовано, так как оно может быть выполнено через операцию **vpandn** (штрих Шеффера), так как **not** x ≡ x | x, например:

vpandn ymm0,ymm1,ymm1; ymm0:=**not** ymm1

Пожалуй, к логическим команда стоит отнести и команды подсчёта числа лидирующих "0" в каждом элементе векторного массива. Например, команда:

vplzcnt{**d**,**q**} op1,op2

Допустимые форматы операндов

op1	op2
r128	r128, m128,m32/m64
r256	r256, m256,m32/m64

Вторым операндом может быть как векторный регистр, так и одиночное целое число в памяти, которое в этом случае дублируется необходимое количество раз. В каждом элементе второго операнда подсчитывается число лидирующих "0" и это число записывается в соответствующий элемент первого операнда, например:

```
.data
  org 1000h
A dq 11b,10b,1b,0b; A[0..3] of qword;
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vplzcntq ymm0,ymm1
```

ymm1	0b	1b	10b	11b
ymm1	64	63	62	61

17.5.1. Команды сдвига

Мир изменился. Я чувствую это в воде, чувствую это в земле, ощущаю в воздухе.

Джон Рональд Руэл Толкин.
«Властелин колец»

А теперь наши любимые команды сдвига ☺. В простейших случаях битовые вектора на регистрах общего назначения можно было сдвигать (налево или направо) на заданное количество бит. С векторными регистрами дело обстоит немного веселее, третий операнд задаёт либо константу, на которую сдвигаются все элементы вектора, либо вектор, каждый элемент которого определяет число сдвигов соответствующего элемента векторного регистра на своё количество разрядов. Итак, *логический сдвиг* элементов вектора на переменное число бит:

vpsl{**l**,**r**}**v**{**w**,**d**,**q**} op1,op2,op3

Буквы в коде операции: первая **l** → **l**ogical, далее буквы {**l** → **l**eft, **r** → **r**ight}, **v** → переменное число бит. Последняя буква задаёт размер одного элемента: **w** → **dw**, **d** → **dd**, **q** → **dq**, как видим, байты не сдвигаются ☹. Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128/m128, i8
r256	r256	r256/m256, i8

Итак, каждый элемент вектора из op2 сдвигается (влево или вправо) на число разрядов, заданное либо константой i8, либо в соответствующем элементе вектора op3 и результат записывается в соответствующую позицию op1. Когда число сдвигов больше 15 для **dw**, 31 для **dd** и 63 для **dq**, то результат сдвига чистый ноль (для регистров общего назначения это не совсем так).

Например:

```
.data
  org 1000h
A dd 0h,1h,2h,3h,4h,5h,6h,7h; A[0..7] of dd
B dd 7,6,5,4,3,2,1,0; B[0..7] of dd
.code
  vmovdqa ymm1,A; что сдвигаем
  vmovdqa ymm2,B; на сколько сдвигаем
; ↓↓ тогда для команды
  vpsllvd ymm0,ymm1,ymm2
; ↓↓ будет такой результат в ymm0
```

ymm1	7h	6h	5h	4h	3h	2h	1h	0h
ymm2	0	1	2	3	4	5	6	7
ymm0	7h	0Ch	14h	20h	30h	40h	40h	0h

Далее у нас следует команда *арифметического сдвига вправо* на переменное число разрядов:

vpsrav{**w**,**d**,**q**} op1,op2,op3

Сдвиг только вправо, так как логический и арифметический сдвиги влево совпадают. Отличие от команды логического сдвига вправо такое же, как и для сдвига на регистрах общего назначения, т.е. на освобождающееся место копируется знаковый бит. Когда число сдвигов больше 15 для **dw**, 31 для **dd** и 63 для **dq**, то результат сдвига полностью заполнен знаковым битом.

И, наконец, циклические сдвиги (вращения по кольцу бит):

vpro{**l**,**r**}**v**{**d**,**q**} op1,op2,op3

Как видим, вращать можно только элементы типов **dd** и **dq**. Другое отличие от "обычных" сдвигов регистров общего назначения заключается в том, что величина сдвига предварительно берётся по модулю 32 для **dd** и 64 для **dq**, так как вращение на большее количество бит просто бессмысленно. К сожалению, так любимых нами вращений через флаг переноса CF здесь по очевидным причинам не предвидится ☹.

Непривычными являются команды, сдвигающие операнд на заданное число байт (а не бит):

vpsl{**l**,**r**}**dq** op1,op2,op3

Буквы в коде операции: первая **l** → **l**ogical, далее буквы {**l** → **l**eft, **r** → **r**ight}, **dq** (**d**ouble **q**uadro) означает, что сдвигается весь 128-битный регистр (для операнда XMM) или независимо две линии по 128 бит (для операнда YMM). Как мы уже значем, некоторые команды вырабатывают *вектор-*

ные маски, элементами которых являются целые числа 0 (**false**) и –1 (**true**). Как и обычные битовые сдвиги, байтовые сдвиги позволяют ставить элементы векторной маски в нужные позиции этого вектора. Допустимые форматы операндов:

op1	op2	op3
r128	r128, m128	i8
r256	r256, m256	i8

На освободившееся место помещаются нулевые байты. При сдвиге регистра XMM и записи результата в YMM старшая часть YMM обнуляется. Например:

```
   vpslldq ymm0,ymm1,2
```

; ↓↓ будет такой результат в ymm0

ymm1	8h	7h	6h	5h	4h	3h	2h	1h
ymm0	6h	5h	0	0	2h	1h	0	0

Ещё более непривычным является сдвиг, названный "выравнивание вправо", он тоже сдвигает на заданное число байт (а не бит):

```
      vpalignr op1,op2,op3,op4
```

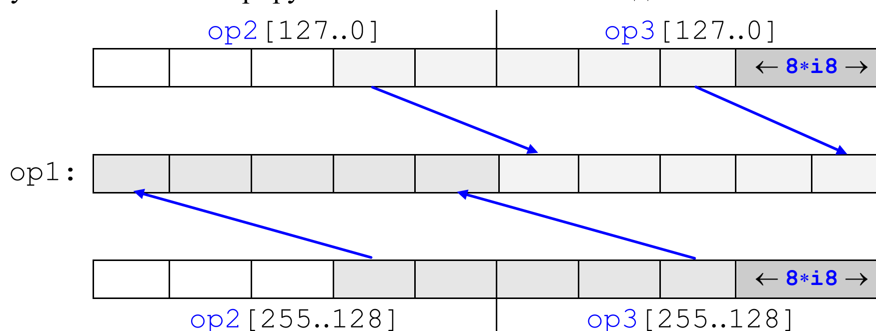
Буквы в коде операции: первая **p** → **p**acked, **align** → выравнивание, **r** → **r**ight. Каждая линия op1 формируется из конкатенации соответствующих линий op2 и op3, которые вместе сдвигаются на число байт, заданное операндом op4. Для YMM регистра операция применяется независимо к двум линиям этого регистра. Величина сдвига в байтах задаётся op4=i8. Некоторым аналогом этой операции для регистров общего назначения служит команда SHRD.(см. разд. 8.8). Допустимые форматы операндов:

op1	op2	op3	op4
r128	r128	r128, m128	i8
r256	r256	r256, m256	i8

Алгоритм выполнения команды:

```
{N=128,256}
temp[255..0]:=(op2[127..0] shl 128) or op3[127..0]) shr op4*8;
op1[127..0]:=temp[127..0];
if N=256 then begin
   temp[255..0]:=(op2[255..128] shl 128) or op3[255..128]) shr op4*8;
   op1[255..128]:=temp[127..0]
end
```

Рисунок ниже иллюстрирует выполнение этой команды.



17.6. Работа со строками

Чтобы усовершенствовать ум, надо больше размышлять, а не заучивать.
Рене Декарт

При работе со строками символов можно эффективно использовать 128-битовые xmm регистры, на которых размещается и за одну команду обрабатывается сразу до 16 символов (байт) или 8 слов, в дальнейшем будем говорить о байтовых строках. Описывая работу с символами, мы будем предполагать, что строки располагаются в оперативной памяти и символы в них нумеруются слева направо, на-

чиная с нуля. Конечно, мы знаем, что при чтении строки (или части этой строки) на регистр символы, в противоположность памяти, нумеруются справа налево, начиная с нуля:

xmm	s_{15}	s_{14}	s_{13}	s_{12}	s_{11}	s_{10}	s_9	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0

На векторных регистрах можно обрабатывать строки символов с явно и неявно заданной длиной. Строка с неявной длиной, как обычно, оканчивается нулём, явная длина строки должна задаваться в регистре EAX для op1 и в регистре EDX для op2. Для строк с неявной длиной 16-байтная часть строки на регистре или в памяти может содержать некорректные символы, расположенные в строке после нулевого байта, эти символы обрабатываются по особым правилам.

Команды обработки строк на векторных регистрах XMM по праву считаются одними из самых сложных в языке машины.

Команды сравнения строк символов **pcmpistri** и **pcmpistrm** используют неявную длину строк, а команды **pcmpestri** и **pcmpestrm** – явную. Формат этих команд:

pcmp{**i**,**e**}**str**{**i**,**m**} op1,op2,op3

Буквы {**i** – **i**mplicit length, **e** – **e**xplicit length} определяют неявную или явную длину строки, буквы {**i** – **i**ndex, **m** – **m**ask} задают способ возврата результата: в виде индекса в регистре ECX и в виде битовой или векторной маски в регистре XMM0.

К сожалению, использовать в этих командых 256-битные YMM регистры нельзя, они "неполноценные", так как реализованы в виде двух независимых 128-битных линеек (lane). Векторная команда параллельно обрабатывает обе линейки и не может переходить с одной линейки на другую. Для арифметических команд это не важно, а вот для движения по строке существенно.

Допустимые форматы операндов:

op1	op2	op3	ответы	явная длина
r128	r128, m128	i8	ecx, xmm0	eax, edx

Как видим, кроме явных параметров, команды имеют и неявные (заданные по умолчанию). При явном задании длин строк эти длины равны abs(EAX) для op1 и abs(EDX) для op2.[1] При выполнении таких команд длины в этих регистрах автоматически округляются вверх до границы 16 (байт) или 8 (слов), если конечно, эти длины больше 16 (или 8). Это позволяет делать "аккуратные" циклы на N **div** 16/8 посторений, оставляя "хвост" строки для отдельной обработки.

Результат команды возвращается в неявных операндах, это регистр ECX и регистр XMM0. Первые два явных операнда op1 и op2 задают обрабатываемые строки, а третий (непосредственный) операнд i8 является так называемым *управляющим байтом*, он определяет способ выполнения команды. По существу, этот байт входит в код операции команды, просто задаваемые им действия настолько специфичны, что их невозможно выразить "в буквах" и включить в имя кода операции.

По сравнению с уже известными Вам строковыми командами (**movs**, **cmps** и т.е.), элементами "векторных" строк могут быть только байты и слова (вероятно, для кодировки Unicode), а двойные и четверные слова не предусмотрены.

Надо отметить, что эти команды выполнются в три этапа. Сначала вычисляется первый промежуточный результат, обозначенный как IntRes1, затем второй промежуточный результат IntRes2. Промежуточные результаты всегда битовые маски. На третьем этапе IntRes2 преобразуется в битовую или байтовую маску в регистре в XMM0 и в индекс символа [0..15] в регистр ECX.

Теперь структура управляющего байта:

№ бита	Описание
7	Зарезервирован
6="0"	Регистр xmm0[15..0] содержит IntRes2 как битовую маску
	Самый младший найденный индекс символа строки (в памяти) в регистре ECX
6="1"	Регистр xmm0 содержит IntRes2 как байтовую маску (из 0 и -1)
	Самый старший найденный индекс символа строки (в памяти) в регистре ECX
5..4="00"	Операция плюс (тождественная), IntRes2:=IntRes1
5..4="01"	Операция минус, IntRes2:=~IntRes1=**not**(IntRes1)
5..4="10"	Маскированный плюс (тождественная), IntRes2:=IntRes1

[1] Вероятно, это сделано, чтобы длина была заведомо неотрицательной?

5..4="11"	Маскированный минус, IntRes2[i]:=~IntRes1[i] для "обычных" символов и IntRes2[i]:=IntRes1[i] для недействительных символов из op2
3..2="00"	Поиск в операнде op2 любых символов (equal any), входящих в набор (множество) op1, при совпадении символа из op2 с любым символом из op1 соответствующий бит в IntRes1 устанавливается в "1" [1]
3..2="01"	Поиск в операнде op2 любых символов, входящих в диапазон (range) символов op1[0]..op1[1] (значения op1[15..2] не важны). При попадании символа из op2 в этот диапазон op1[1]<=op2[i]<=op1[0], соответствующий бит в IntRes1 устанавливается в "1". Здесь важно, знаковые или беззнаковые числа!
3..2="10"	Сравнение соответствующих символов строк в op1 и op2 на равенство в каждой позиции (equal each), при совпадении этот бит в IntRes1 устанавливается в "1", при первом несовпадении далее одни "0"
3..2="11"	Поиск в op2 подстроки, заданной в op1(equal ordered), для всех найденных вхождений первый бит подстроки в IntRes1 устанавливается в "1" [2]
1..0="00"	Вектор из упакованных беззнаковых байтов
1..0="01"	Вектор из упакованных беззнаковых слов
1..0="10"	Вектор из упакованных знаковых байтов
1..0="10"	Вектор из упакованных знаковых слов

Сначала команда выполняет выборку и сравнение символов из op1 и op2 в соответствии правилами, задаваемыми битами 3..2 управляющего байта, записывая промежуточный результат в IntRes1. Затем к этому промежуточному результату применяется операция, заданная битами 5..4 управляющего байта, записывая промежуточный результат в IntRes2. И, наконец, промежуточный результат из IntRes2 сохраняется в регистре xmm0 как битовая или байтовая маска, в зависимости от бита 6 управляющего байта, при этом формируется нужный индекс в регистре ECX.

Описываемые команды устанавливают флаги по таким правилам:

Установка флагов для команд со строками неявной длины	
CF	CF := IntRes2<>0
ZF	ZF := в op2 ∃ нулевой байт/слово, т.е. в op2 конец строки
SF	SF := в op1 ∃ нулевой байт/слово, т.е. в op1 конец строки
OF	OF := IntRes2[0]=0
AF, PF	AF := 0, PF := 0

Установка флагов для команд со строками явной длины	
CF	CF := IntRes2<>0
ZF	ZF := abs(EDX)<16/8
SF	SF := abs(EAX)<16/8
OF	OF := IntRes2[0]=0
AF, PF	AF := 0, PF := 0

Например (считаем, что строки расположены не на регистрах, а в памяти и оканчиваются нулём): [3]

```
; Пример 0. i8[3..2]=0000b
  op1:=   "i!"; op2={'i','!'}
  op2:=   "this is a joke!!"
  IntRes1="0010010000000011"
; Пример 1. i8[3..2]=0100b
  op1:=   "aj"; op2=['a'..'j']
  op2:=   "this is a joke!!"
  IntRes1="0010010010100100"
; Пример 2. i8[3..2]=1000b
```

[1] Сразу до 256 сравнений!

[2] Это более мощный аналог функции pos(op1,op2) в языке Free Pascal.

[3] Примеры этих "красивых" текстовых строк взяты из книги: Йо Ван Гуй. "Программирование на ассемблере x64 от начального уровня до профессионального использования AVX", М.: ДМК Пресс, 2021. 332 с.

```
    op1:=   "this is no joke!"
    op2:=   "this is a joke!!"
    IntRes1="1111111100000000"
; Пример 3. i8[3..2]=1100b
    op1:=   "is"; pos(op1,op2)
    op2:=   "this is a joke!!"
    IntRes1="0010010000000000"
```

Приведённые примеры хорошо описывают суть дела и легко воспринимаются человеком, посмотрим, однако, как, скажем, Пример 1 реализуется на Ассемблере (символ ˽ использован для "явного" задания вида пробела):

Пример 1. i8[3..2]=0100b

```
.data
;    без выравнивания на 16 байт → movdqu
op1 db "aj",14 dup(0)
    align 16; теперь с выравниванием → movdqa
op2 db "this˽is˽a˽joke!!"
 code
; ↓↓ xmm1[15..2]:=#0; xmm1[1..0]:="ja"
    movdqu xmm1,xmmword ptr op1
; ↓↓ xmm2[15..0]:="!!ekoj˽a˽si˽siht"
    movdqa xmm2,xmmword ptr op2
    pcmpistrm xmm1,xmm2,0100b
;     xmm1=0000 0000 0000 00ja
;     xmm2=!!ek oj˽a ˽si˽ siht
; IntRes1=0010 0101 0010 0110b
; IntRes2=0010 0101 0010 0110b
; xmm0=00…00 2526h; ecx=1 (позиция 'h')
; ↓↓ для pcmpistri xmm1,xmm2,0100b
; xmm0=0,0,1,0,0,1,0,1,0,0,1,0,1,1,0,0
; CF=1(IntRes2<>0), ZF=0(в op2 нет нулевого байта)
; SF=1(в op1 есть нулевой байт), OF=IntRes2[0]=0
```

17.6.1. Определение длины строки

Возможность писать красивые программы даже на Ассемблере – именно это в первую очередь сделало меня приверженцем программирования.

Дональд Эрвин Кнут

Теперь примеры, без которых во всём этом не разберёшься ☹. Сначала определение длины строки, оканчивающейся нулём, для простоты максимальная длина строки кратна 16.

Начало нашей строки выровнено на 16 байт, а шаг движения по строке тоже 16. Когда длина строки не кратна 16, есть опасность выйти за конец строки (за нулевой символ). Это опасно только тогда, когда мы при этом выходим за границу текущей страницы памяти и попадаем в "чужую" страницу, не принадлежащую нашей программе (будет исключение). Простейший выход – отвести под строку память, длина которой кратна 16 байт.

```
; Определение длины строки, конец #0
.data?
N equ 1000000
  align 16
S db 16*N dup (?); строка
.code
; здесь ввод (получение) строки S
  mov ebx,-16; +16 будет в начале цикла
```

```
; ↓↓ xmm1="" – не содержит ни одного символа!
  pxor xmm1,xmm1; op1:=16 dup (#0)=""
  align 16; для опримизации работы кеша
L:add ebx,16; после pcmpistri флаги не портятся
; ↓↓ i8=1000b – сравнение строк в op1 и op2 ⚠
; ↓↓ байты, а не слова; в регистре ECX младший индекс
; ↓↓ ZF=1, когда нашли #0 в op2
  pcmpistri xmm1,xmmword ptr S[ebx],1000b
  jnz L; пока не нашли #0; ECX=16
; в ECX индекс первого #0 <=15
  add ebx,ecx; длина строки в ebx>=0
  outwordln ebx,,'Длина строки S='
```

Работа команды **pcmpistri** не так проста, как это может показаться с первого взгляда. Дело в том, что эта команда с `i8[3..2]=10b` сравнивает две строки с *неявно* заданной длиной. Первая из строк содержится в `op1` (у нас это регистр `xmm1`) и оканчивается нулём, а вторая в `op2` (у нас это 16-байтная область памяти) и тоже должна оканчиваться нулём! Но на регистре `xmm1` пустая строка, со всеми некорректными символами, там и искать то нечего, а в очередных 16-ти байтах памяти тоже может быть, *некорректная* строка, с символами после конечного нуля!

Работа строковых команд базируется на сравнении символов между собой, но результат такого сравнения неопределён, если хотя бы один из символов недействительный (расположен за концом строки). В этом случае сравнения двух символов (**true**/**false**) надо доопределить, как это делают наши строковые команды, показано в следующей таблице (`K=i8[3..2]`):

op1	op2	K=00b equal any	K=01b range	K=10b equal each	K=11b equal ordered
invalid	invalid	**false**	**false**	**true**	**true**
invalid	valid	**false**	**false**	**false**	**true**
valid	invalid	**false**	**false**	**false**	**false**

В такое доопределение сравнения символов заложена "разумная" семантика, которую мы здесь разбирать не будем. [1]

Результатами таких сравнений в наших простых программах мы пользоваться не будем. В нашем предыдущем примере мы просто воспользовались тем, что, когда в `op2` нет нуля, то будет `ZF=1`, а регистр `ECX:=16` и цикл продолжится. В особом случае (для операнда `i8[3..2]=10b`), когда в регистре `xmm1` пустая строка, а в 16-ти байтах памяти *некорректная* строка (с символами после нуля), то команда возвращает в регистре `ECX` не число 16, а индекс этого нуля `ECX<16` и тоже будет `ZF=1`. Заметим, что для команды с операндом `i8[3..2]=01b` в этом случае на регистре `ECX` останется число 16 и длина строки будет подсчитана неправильно!

Когда нуля в строке, хранящейся в памяти, нет, будет нехорошо ☺, попытайтесь это исправить, применив другой цикл. Задача также усложняется, если начало строки не выровнено на границу 16 байт (у нас **align** 16!), настоящая" программа определения длины строки должна учитывать такую ситуацию.

Напишем теперь макрофункцию, для вычисления длины строки, которая оканчивается нулём и состоит не более, чем из 15 символов:

```
Len macro S
    local L
    vmovdqu xmm2,xmmword ptr S
    pxor xmm1,xmm1; op1:=""
    pcmpistri xmm1,xmm2,1000b
    jz  L; длина строки <16
    mov  ecx,-1; длина строки >=16
L:
```

[1] См., например, "Randall Hyde. The Art of 64-Bit Assembly Language, vol.", разд. 14.3.5.

```
    exitm <ecx>
    endm
```

Попробуем решить эту же задачу с помощью регистров YMM, рассматривая строку просто как массив беззнаковых байт, и будем искать в нём первое вхождение числа ноль.

```
; Определение длины строки, конец #0
.data?
N equ 1000000
  org 1000h
S db 32*N dup (?); строка
.code
; здесь ввод (получение) строки S
  pxor xmm1,xmm1; ymm0:=32 нуля db
  xor  ebx,ebx; индекс массива
; ↓↓ ymm0<>0 - есть нулевой байт!
L:vcmpEQb ymm0,ymm1,S[eax]
  vptest ymm0,ymm0
  jnz Z; Нашли ноль!
  add ebx,32
  jmp L; Продолжение поиска
Z:
; Как более эффективно найти младшую -1 в ymm0[31..0] ?
; Поиск нуля в S[ebx]..S[ebx+31]
  mov  ecx,32
P:cmp  S[ebx],0
  je   КОН; нашли
  inc  ebx
  loop P
КОН: ; ebx=Длина строки>=0
```

Данный вариант заведомо быстрее для длинных строк, хотя его и тормозит "хвостовой" цикл. Более сложные операции со строками с помощью регистров YMM сделать не получится.

17.6.2. Вхождение одной строки в другую

Мы тогда уверены в познании всякой вещи, когда узнаём её первые причины, первые начала...

Аристотель, IV век до н.э. «Физика»

Далее рассмотрим задачу определения первого вхождения одной строки (не длиннее 15 символов) в состав другой строки. Заметим, что это упрощённая версия стандартной функции pos(X,Y) в языке Free Pascal для строк типа AnsiString.

```
; Определение первого вхождения X в Y
; пусть pos("",Y)=pos("","")=1 !
N  equ 1000000
.data?
  align 16
Y  db N dup (?);  строка Y
.data
  align 16
X  db "Что ищем",0; строка X
.code
; здесь ввод (получение) строки Y
   vmovdqu xmm0,xmmword ptr Y
   mov   ebx,16
; ↓↓ шаг поиска=ebx=16-Len(X)
```

```
    sub   ebx,Len(X); шаг поиска
    xor   eax,eax; позиция X в Y
    sub   eax,ebx; +ebx будет в начале цикла
L: add    eax,ebx; после pcmpistri флаги не портятся
; ↓↓ i8=1100b - pos(op1,op2)
; ↓↓ байты, а не слова; в ECX индекс младшей "1"
; ↓↓ ZF=1, когда нашли #0 в op2
    pcmpistri xmm0,xmmword ptr Y[eax],1100b
```

```
    pushfd; запомнить флаги (нам надо ZF)
    or    ecx,ecx
    jz    L1; не нашли вхождение
    add   eax,ecx; pos вхождения
    popfd; очистить стек
    jmp   L2; Выход pos>0
L1:popfd; востановить флаги (ZF)
    jnz   L; пока не нашли #0 в op2
    mov   eax,-1; будет pos=0 - не нашли
L2:inc    eax; 0 -> не нашли
    outwordln eax,,"pos(X,Y)="
```

Как видим, эффективность алгоритма обратно пропорциональна длине строки X (и прямо пропорциональна шагу поиска).

17.6.3. Сравнение двух строк

Порой мы видим многое, но не замечаем главного.

Конфуций, V век до н.э.

Сначала рассмотрим задачу сравнения двух строк с неявной длиной:

```
; Сравнение двух строк, концы #0
.data?
N equ 1000000
  align 16
X db 16*N dup (?); строка X
Y db 16*N dup (?); строка Y
.code
; здесь ввод (получение) строк X и Y
  mov   eax,-16; +16 будет в начале цикла
  align 16
L:add   eax,16; после pcmpistri флаги не портятся
  movdqu   xmm1,X[eax]; очередная порция символов
; ↓↓ 01 10 00b=00 беззнаковые байты,
  ↓↓ 10 сравнение строк, 01 IntRes2:=~IntRes1
  pcmpistri xmm1,Y[eax],01 10 00b
  jc  NotEquel; не все символы (до #0) совпали
  jnz L; продолжение поиска
Equel: ; все символы до #0 совпали
  outstrln "Строки совпали"
  exit
NotEquel:
  lea eax,[eax+ecx+1]; индекс несовпадения
  outwordln eax,,"Строки различны с позиции:"
  exit
```

Директива `align 16` перед меткой начала "главного" цикла позволяет более эффективно использовать кеш память.

А теперь задача сравнения двух строк с явной (и, возможно различной) длиной. Как уже говорилось, первый операнд (op1) должен содержать длину строки в регистре EAX, а второй операнд (op2) в регистре EDX. Будем предполагать, что перед строкой с явно заданной длиной при её хранении в памяти расположены 4 байта её длины:

```
N equ 1000000; Макс. длина строки
  dd ?; Длина строки S
S db N dup (?)
```

Тогда длину строки можно получить, скажем, так:

```
  mov eax,dword ptr S-4; Длина строки
```

Мы не будем предполагать, что текст в таких "строках" выровнен в памяти на границу 16 байт. Символы строк сравниваются, пока не будет достигнута длина хотя бы одной из них. Итак, программа сравнения строк X и Y:

```
.data?
N equ 16*1000000-4
  dd ?; длина строки X
X db N dup (?); строка X
  dd ?; длина строки Y
Y db N dup (?); строка Y
.code
; здесь ввод (получение) строк X и Y
  xor ecx,ecx; Для пустой строки
  mov edx,dword ptr Y-4; Длина строки Y
  cmp edx,dword ptr X-4; Длина Y=Длина X ?
  jne NotEquel; Строки разной длины
  mov eax,-16; +16 будет в начале цикла
L:add eax,16; после pcmpestri флаги не портятся
  movdqu xmm1,X[eax]; очередная порция символов
; ↓↓ 01 10 00 b – справа налево, 00 беззнаковые байты,
; ↓↓ 10 IntRes1[i]:="1", пока совпадают, далее "0"
  ↓↓ 01 IntRes2:=~IntRes1
  pcmpestri xmm1,Y[eax],01 10 00 b
  ↓↓ CF:=IntRes2<>0
  jc  NotEquel; не все символы до конца строки совпали
  ↓↓ ZF:=abs(edx)<16
  jz Equel; Строки совпали
  sub edx,16; Программа сама уменьшает длину строки!
  jmp L; продолжение поиска
Equel:
  outstrln "Строки совпали"
  exit
NotEquel:
  lea eax,[eax+ecx+1]; индекс несовпадения
```

17.7. Использование регистров масок

Мы те, кем мы притворяемся. Осторожнее выбирайте свою маску.

Курт Воннегут. «Матерь Тьма»

Начиная с версии AVX2, во многих командах работы с векторными регистрами реализованы новые возможности. В коде операции три бита были отведены под номер так называемого регистра маски записи (writemask или opmask register), эти 64-битовые регистры обозначаются как k0–k7. Регистр k0 фиктивный, по умолчанию его номер (000b) всегда находится в коде операции, а значение состоит из одних "1", что, как мы вскоре увидим, не влияет на выполнение команды. Задание регистров с номерами k1–k7 может существенно изменить результат выполнение команды. Кроме того, в коде опе-

рации появился бит так называемой маски изменения z (merging-masking), присваивание ему значения "1" тоже меняет результат выполнения команды.

Сначала рассмотрим уже известную нам операцию рассылки значения **vpbroadcast** с использованием маски. Номер регистра маски k и (единичная) битовая маска изменения z записываются (если они заданы) в фигурных скобках после первого операнда:

vpbroadcast{**b**,**w**,**d**,**q**} op1 {k}{z},op2

Принимая в маске значение "0" за **false** и "1" за **true**, алгоритм выполнения этой команды с регистром маски k и битом z можно записать в виде:

```
for i:=0 to N-1 do
  if (k=0){k не задан} or k[i]
    then op1[i]:=op2 else
      if z=1 {z задан в команде}
         then op1[i]:=0
         else {op1[i] не изменяется}
```

Итак, задание маски и бита изменения позволяет выборочно для каждого элемента вектора op1 записать в него результат выполнения операции, оставить без изменения либо обнулить. Ещё пример использования этого механизма на примере команды сложения векторов целых чисел двойной точности:

vpaddd op1 {k}{z},op2,op3

Теперь алгоритм выполнения этой команды можно формально записать в виде:

```
for i:=0 to 7 do
if (k=0){k не задан} or k[i]
  then op1[i]:=op2[i]+op3[i] else
    if z=1 {z задан в команде}
       then op1[i]:=0
       else {op1[i] не изменяется}
```

Итак, рассмотрим пример:

```
.data
  org 1000h
A dd 10,11,12,13,14,15,16,17; A[0..7] of dd
B dd 20,21,22,23,24,25,26,27; B[0..7] of dd
C dd 40,41,42,43,44,45,46,47; C[0..7] of dd
.code
  vmovdqa ymm1,ymmword ptr A; первое слагаемое
  vmovdqa ymm2,ymmword ptr B; второе слагаемое
  vmovdqa ymm0,ymmword ptr C; что сначала было в ymm0
```

ymm1	17	16	15	14	13	12	11	10
ymm2	27	26	25	24	23	22	21	20
ymm0	47	46	45	44	43	42	41	40

```
; ↓↓ если маска k1=10011010b, то для команды
  vpaddd ymm0 {k1}{z},op2,op3
; ↓↓ будет такой ответ в ymm0 (верхний для z=0, нижний для z=1):
```

ymm0 z=0	44	46	45	38	36	42	33	40
ymm0 z=1	44	0	0	38	36	0	33	0

По существу, маски образуют ещё одно пространство регистров, над которым реализованы команды пересылок, логические команды и сдвиги. По аналогии с регистром общего назначения, с регистром маски можно работать целиком (со всеми 8 байтами), с младшими четырьмя, двумя и одним байтом, что задаётся соответствующей буквой в коде операции.

Обмен данными между 1, 2, 4 или 8 байтами регистра маски и регистра общего назначения производится командой

```
kmov{b,w,d,q} op1,op2
```

Допустимые форматы операндов

op1	op2
k (**b**)	k, r8, m8
k (**w**)	k, r16, m16
k (**d**)	k, r32, m32
k (**q**)	k, r64, m64
m8 (**b**)	k
m16 (**w**)	k
m32 (**d**)	k
m64 (**q**)	k

Например:

```
mov   al,10011010b
kmovb k2,al
```

Логические команды

```
knot{b,w,d,q} k1,k2; k1:=not k2
k{⊗}{b,w,d,q} k1,k2,k3; k1:=k2⊗k3
```

Здесь операции ⊗={**or**,**and**,**xor**,**or**,**andn**,**xnor**} такие же, как и для битовых строк на векторных регистрах, где экзотика **kxnor** это `k1:=not(k2 xor k3)`.

Команды сдвига

```
kshift{l,r}{b,w,d,q} k1,k2,i8; k1:=shift{l,r}(k2,i8)
```

Команды сложения

```
kadd{b,w,d,q} k1,k2,k3; k1:=k2+k3
```

Иногда из двух масок надо сформировать одну (третью), для этого служат команды

```
kunpck{bw,wd,dq} k1,k2,k3
```

Команды выполняется по такому алгоритму:

```
{N=8(bw), 16(wd), 32(dq)}
k1:=0; k1:=(op1 shr N) or op2
```

В разд. 17.3.4 была описана команда

```
vpcmp{eq,gt}{b,w,d,q} op1,op2,op3
```

Она формировала векторную маску, как результат сравнения двух векторов знаковых целых чисел. А вот аналогичное формирование битовой маски на регистре масок по результату сравнения двух векторов целых чисел:

```
vpcmp{ ,u}{b,w,d,q} op1,op2,op3,op4
```

Суффикс **u** задаёт *беззнаковое* сравнение, а отсутствие этого суффикса – *знаковое*. Буква в конце кода операции задаёт размеры операндов: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Допустимые форматы операндов:

op1	op2	op3	op4
k1	r128	r128, m128	i8
k1	r256	r256, m256	i8

Операция сравнения задаётся тремя последними битами `i8[2..0]`:

i8	<op2> отношение <op3>	Псевдокод
0	**eq** – равно	**vpcmpEQ**{ ,**u**}{**b**,**w**,**d**,**q**}
1	**lt** – меньше	**vpcmpLT**{ ,**u**}{**b**,**w**,**d**,**q**}
2	**le** – меньше или равно	**vpcmpLE**{ ,**u**}{**b**,**w**,**d**,**q**}
3	**false** – всегда **false**	
4	**ne** – неравно	**vpcmpNEQ**{ ,**u**}{**b**,**w**,**d**,**q**}
5	**nlt**=**ge** – больше или равно	**vpcmpNLT**{ ,**u**}{**b**,**w**,**d**,**q**}
6	**nle**=**gt** – больше	**vpcmpNLE**{ ,**u**}{**b**,**w**,**d**,**q**}
7	**true** – всегда **true**	

В третьей колонке указан псевдокод, позволяющий задавать четвёртый операнд внутри кода операции, например, вместо команды:

```
vpcmpb k1,ymm2,ymm3,4
```

можно писать

```
vpcmpNEb k1,ymm2,ymm3
```

Результат сравнения двух векторов записывается в младшие разряды регистра маски, заданной первым операндом (старшие разряды обнуляются). Ну, и, наконец, эта команда тоже может выполняться под управлением другого (или этого же самого) регистра маски и управляющего бита z ☺:

```
vpcmp{,u}{b,w,d,q} op1 {k1}{z},op2,op3,op4
```

Далее рассмотрим команду преобразование значения векторного регистра в регистр маски:

```
vpmov{b,w,d,q}2m op1,op2
```

Буква задаёт длину элемента вектора: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Допустимые форматы операндов:

op1	op2
k1	r128, r256

Команда выполняется по такому алгоритму:

```
for i:=0 to N-1 do k1[i]:=SF(op2[i])
```

Обратная команда для преобразования значения регистра маски в векторный регистр:

```
vpmovm2{b,w,d,q} op1,op2
```

Буква задаёт длину элемента вектора: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Допустимые форматы операндов:

op1	op2
r128, r256	k1

Команда выполняется по такому алгоритму:

```
for i:=0 to N-1 do
  if k1[i]=0
    then op2[i]:=0 else op2[i]:=-1
```

К сожалению, команды формирования битовой маски по сравнению двух векторов <u>вещественных</u> чисел нет. Приходится сначала формировать <u>векторную</u> маску описанной ранее командой

```
vcmpp{s,d} op1,op2,op3,op4
```

а затем преобразовывать векторную маску в битовую маску командой

```
vpmov{b,w,d,q}2m op1,op2
```

Например:

```
.data
  org 1000h
A real8 10.0,21.0,12.0,13.0; A[0..3]
B real8 20.0,11.0,22.0,13.0; B[0..3]
.code
  vmovdqa ymm1,ymmword ptr A; ymm1:=A
  vmovdqa ymm2,ymmword ptr B; ymm2:=B
; ↓↓ Маска ymm0[i]:=ymm1[i]<=ymm2[i]
  vcmppLEsd ymm0,ymm1,ymm2
```

ymm1	10.0	21.0	12.0	13.0
ymm2	20.0	11.0	22.0	13.0
ymm0	-1	0	-1	-1

```
; ↓↓ Маска k1[i]:=SF(ymm0[i])
  vpmovq2m k1,ymm0; k1=1010b
```

Можно формировать регистр маски не только как результат арифметического сравнения элементов двух векторов, но и как результат логических операций над элементами векторов. Например, вот

команда попарного логического произведения элементов двух векторов и запись результата в регистр маски:

```
vptestm{b,w,d,q} op1,op2,op3
```

Буква задаёт длину элемента вектора: **b** → **byte**, **w** → **word**, **d** → **dword**, **q** → **qword**. Допустимые форматы операндов:

op1	op2	op2
k1	r128, r256	r128, r256, m128, m256

Команда выполняется по такому алгоритму:

```
for i:=0 to N-1 do
    k1[i]:=(op2[i] and op3[i])≡0
```

Нам известен аналог этого механизма масок для скалярных операций. Для нашей ЭВМ это только команды условного присваивания, например:

```
cmovz op1,op2; if ZF=1 then op1:=op2
```

А вот в процессорах ARM, часто используемых в планшетах и смартфонах, есть возможность условного выполнения почти любой команды. Например, просто представьте, что в нашем Ассемблере есть такие "условные" команды **cadd**{xx}, **csub**{xx}, **ccall**{xx} и т.д.

Для векторных операций механизм масок позволяет сделать условными операции для каждого элемента векторного регистра по отдельности. По большому счёту, это позволяет исключить при программировании использование многих команд условных переходов **if then else**.

17.7.1. Операции с масками, устанавливающие флаги

Но, увы, самые очевидные вещи как раз хуже всего и доходят до сознания людей.
Айзек Азимов. «Я, робот»

Все предыдущие команды для работы с регистрами масок не устанавливают флагов в EFLAGS, что неудобно для программиста. Теперь рассмотрим команды, устанавливающие такие флаги. Команды тестирования:

```
ktest{b,w,d,q} k1,k2
```

Это, как обычно, команда логического умножения без изменения первого операнда. Тестирование устанавливает наши любимые флаги:

`ZF:=(op1 and op2)≡0; CF:=(not op1 and op2)≡0` {необычно}

Флаги AF, OF, PF, и SF при этом просто обнуляются.

Аналогичное тестирование, но операцией логического сложения:

```
kortest{b,w,d,q} k1,k2
```

Первый операнд здесь тоже не меняется (в отличие от рассмотренной ранее команды **kor**). Устанавливает флаги:

`ZF:=(op1 or op2)≡0; CF:=(op1 or op2)=-1` {все "1"}

Остальные флаги при этом не меняются.

17.7.2. Векторизация плоского цикла

Чтобы поверить в алгоритм, его нужно увидеть.
Дональд Эрвин Кнут

Приведём примеры использования механизма масок при решении задач. В качестве примера рассмотрим задачу векторизации простого (так называемого *плоского*) цикла,[1] содержащего условные операторы, вот этот алгоритм на языке Free Pascal:

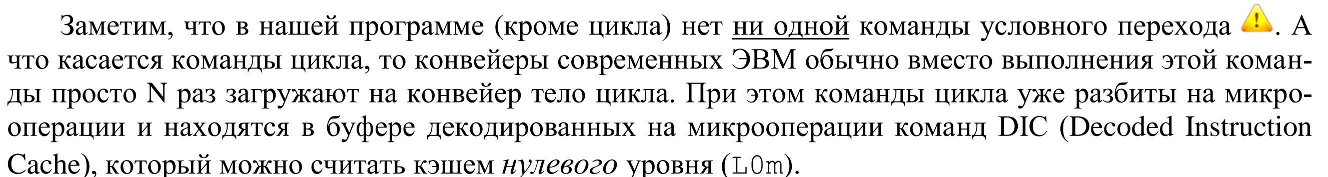
```
const N=100000000;
var X,Y,Z: array[1..8*N] of longint;
```

[1] В языках высокого уровня цикл обычно называется плоским, если в нём один параметр цикла, отсутствуют побочные эффекты, обращение ко всем массивам производится по одному параметру цикла и операции для всех значений параметра цикла можно выполнять параллельно.

```
    i: longint;
begin {ввод массивов X и Y}
  for i:=1 to 8*N do begin Z[i]:=0;
    if X[i]<Y[i] then Z[i]:=X[i]+Y[i];
    if X[i]>Y[i] then Z[i]:=X[i]*Y[i];
  end
```

После векторизации получается примерно такая программа:

```
.data?
  org 1000h
N equ 100000000
X dd  8*N dup (?)
Y dd  8*N dup (?)
Z dd  8*N dup (?)
.code
; ввод векторов X и Y
  mov ecx,N
  xor ebx,ebx; индекс массивов
  align 16
L:vmovdqa ymm0,ymmword ptr X[ebx]; 8 очередных X[i]
  vmovdqa ymm1,ymmword ptr Y[ebx]; 8 очередных Y[i]
  pxor xmm2,xmm2; ymm2:=8 очередных Z[i]
  vpcmpLTd k1,ymm0,ymm1; k1[i]:=X[i]<Y[i]
; ↓↓ if k1[i] then ymm2[i]:=X[i]+Y[i]
  vaddpd ymm2{k1},ymm0,ymm1
  knotd  k1,k1; k1:=not k1
; ↓↓ if k1[i] then ymm2[i]:=X[i]*Y[i]
  vmulpd ymm2{k1},ymm0,ymm1
  vmovdqa ymmword ptr Z[ebx],ymm2; 8 очередных Z[i]
  add ebx,32
  loop L
```



Заметим, что в нашей программе (кроме цикла) нет ни одной команды условного перехода ⚠. А что касается команды цикла, то конвейеры современных ЭВМ обычно вместо выполнения этой команды просто N раз загружают на конвейер тело цикла. При этом команды цикла уже разбиты на микрооперации и находятся в буфере декодированных на микрооперации команд DIC (Decoded Instruction Cache), который можно считать кэшем *нулевого* уровня (L0m).

До реализации набора векторный команд AVX2 (2013 год) эту задачу приходилось делать не с помощью регистрами битовых масок, а с векторными масками на векторных регистрах, что было значительно сложнее.

17.7.3. Задача подсчёта числа цифр в массиве символов

Программирование – такое же сложное искусство, как и литературное творчество. И там и здесь предпочтение отдаётся краткости.
Р.В. Хэмминг «Тьюринговская лекция»

В качестве ещё одного примера рассмотрим задачу подсчёта числа цифр в массиве символов. Предполагаем, что число символов в массиве кратно 32, иначе последнюю порцию дополним символами #0. Вот эта программа на языке Free Pascal:

```
const N=100000000;
var X: array[1..32*N] of char;
    i: longword; S: longword:=0;
begin {ввод массива X}
  for i:=1 to 32*N do
```

```
    if X[i] in ['0'..'9'] then inc(S)
```

Как обычно, рассматриваем символы как беззнаковые однобайтные целые числа (номера символов в алфавите). После векторизации получается примерно такая программа:

```
.data?
  org 1000h
N equ 100000000
X db  32*N dup (?)
.data
S dd 0
Z db '0'
D db '9'
.code
; ввод массива символов X
  mov ecx,N
  xor ebx,ebx; индекс массива
  vpbroadcastb ymm1,Z; все 32 '0'
  vpbroadcastb ymm2,D; все 32 '9'
  align 16
L:vmovdqa ymm0,ymmword ptr X[ebx]; очередные 32 символа
; ↓↓ маска k1[i]:=X[i]>='0'
  vpcmpuNLTb k1,ymm0,ymm1
; ↓↓ маска k2[i]:=X[i]<='9'
  vpcmpuLEb  k2,ymm0,ymm2
; ↓↓ k1[i]:=(X[i]>='0') and (X[i]<='0')
  kandd k1,k1,k2; k1[i]:=X[i] in ['0'..'9']
; ↓↓ число "1" в k1=число цифр в ymm0 ⚠
  kmovd eax,k1
  popcnt eax,eax; eax:=число "1" в k1
  add S,eax
  add ebx,32
  sub ecx,1
  jnz L
  outwordln S,,"Число цифр в X="
```

Итак, регистры масок (как и регистры общего назначения, вещественные и векторные регистры) образуют ещё одно линейное адресное пространство ячеек памяти. Маски "заточены" под работу с множествами, каждый бит маски задаёт, обладает ли соответствующий элемент векторного регистра нужным свойством (принадлежит ли он к определённому *множеству*).

Так, в нашем последнем примере регистр маска `k1` задавала множество символов цифр в очередной порции из массива символов. Многие команды для работы с векторными регистрами могут выполняться под "фильтром" регистров масок. Соответственно, как Вы уже знаете, над регистрами масок определены как арифметические, так и логические операции (объединения **`or`**, пересечения **`and`** и т.д.).

17.7.4. Векторизация цикла с внутренним подциклом

Скоро останутся лишь две группы людей: те, кто контролирует компьютеры, и те, кого контролируют компьютеры. Постарайтесь попасть в первую.

Льюис Д. Эйген

Рассмотрим программу для обработки массива вещественных чисел на языке Free Pascal:

```
const N=100000000;
var X: array[1..4*N] of double;
    i: longword;
```

```
begin {ввод массива X}
  for i:=1 to 4*N do begin
    while X[i]>1.0 do X[i]:=X[i]/3.0;
    X[i]:=-X[i]
  end
```

В этой программе цикл с параметром включает в себя цикл с неизвестным числом повторений (в нашем примере это цикл **while**). После векторизации получается примерно такая программа:

```
.data?
  org 1000h
N equ 1000000
X real8 4*N dup (?); X[1..4*N] of double
.data
  org 1000h
T real8 3.0
E real8 1.0
.code
; ввод массива X
  mov ecx,N
  xor ebx,ebx
  vpbroadcastsd ymm1,T; все четыре 3.0
  vpbroadcastsd ymm2,E; все четыре 1.0
  pxor xmm3,xmm3; ymm3:=все четыре 0.0
  align 16
L:vmovdqa ymm0,ymmword ptr X[ebx]; очередные 4 double
; ↓↓ маска ymm4[i]:=X[i]>1.0
@@:vcmppGTsd ymm4,ymm0,ymm2
  vptest ymm4,ymm4; ZF:=(ymm4=0)
  jz F@
; ↓↓ маска k1[i]:=X[i]>1.0
  vpmovq2m k1,ymm4
; ↓↓ if k1[i] then ymm0[i]:=ymm0[i]/3.0
  vdivpd ymm0 {k1},ymm0,ymm1
  jmp @B
; ↓↓ ymm0[i]:=0.0-ymm0[i]
@@:vsubpd ymm0,ymm3,ymm0
  vmovdqa ymmword ptr X[ebx],ymm0; готовы 4 double
  add ebx,32; i:=i+4
  sub ecx,1
  jnz L
```

Итак, механизм масок позволяет избежать при программировании многих условных переходов **if then else**.

17.8. Сложные векторные команды

Только самые мудрые и самые глупые не поддаются обучению.
Конфуций, V век до н.э.

С каждым годом векторные процессоры оснащаются всё более сложными и изощрёнными командами. Эти команды призваны ускорить обработку данных в определённых, достаточно узких предметных областях.

Например, вот команды из набора AVX-512VNNI (Vector Neural Network Instructions) предназначенные для так называемого глубокого обучения нейросетей (accelerating inner convolutional neural network):

```
vpdpbusd op1{k1}{z},op2,op3
```

Буквы {**busd**} обозначают: **b** → байты, **u** → беззнаковые, **s** → знаковые, умножаются в **d** и суммируются (по четыре) в двойные слова. Допустимые форматы операндов:

op1	op2	op3
r128	r128	r128, m128, m32bcst
r256	r256	r256, m256, m32bcst
r512	r512	r512, m512, m32bcst

Алгоритм выполнения команды (первый операнд – вектор беззнаковых, а второй – знаковых байтов):

```
{N=4,8,16}
{var op2: array[0..4*N-1] of byte;
     op3: array[0..4*N-1] of shortint;
     op1: array[0..N-1]   of longint;
     t: longint;}
for i:=0 to N-1 do begin t:=0;
  for j:=0 to 3 do
    t:=t+word(op2[4*i+j])*smallint(op3[4*i+j]);
  op1[i]:=t
end
```

Итак, сначала **byte** → **word**, потом **word**∗**word** → **dword**, потом сумму четырёх **dword** → op1[i]. Это позволяет в одной команде умножать значения входных сигналов нейрона на (знаковые) веса этих сигналов, и суммировать полученные величины. Раньше это приходилось делать тремя командами.

Большой проблемой в программировании задач обработки вещественных данных является наличие таких "особых" вещественных чисел, как ±0, ±∞, QNaN, SNaN и денормализованных чисел. Такие вещественные значения часто требуют отдельной и особой обработки. Соответственно, должны быть векторные команды, позволяющие выявлять и обрабатывать такие величина. Команда:

vpclassp{**s**,**d**} k1{k2},op2,op2

позволяет построить в регистре k1 битовую маску, определяющую принадлежность элементов вектора op2 к указанному в параметре op3 классу величин. Допустимые форматы операндов:

op1	op2	op3
k1	r128, m128, m32bcst/m64bcst	i8
k1	r256, m256, m32bcst/m64bcst	i8
k1	r512, m512, m32bcst/m64bcst	i8

Операнд op3=i8 задаёт множество классов величин, каждый элемент вектора op2 проверяется на принадлежность этому множеству.

i8	Класс величин
0	QNaN
1	+0.0
2	-0.0
3	+∞
4	-∞
5	денормализованное
6	"нормальное" < 0
7	SNaN

Алгоритм выполнения команды

```
for i:=0 to N-1 do
  if (k2=0){k2 не задан} or k2[i]
    then k1[i]:=op2[i]∈i8
```

Далее рассмотрим очень сложную команду

vfixupimmp{**s**,**d**} op1{k1}{z},op2,op3,op4

Эта команда может не только определять "нехорошие" значения элементов вектора op2, но и заменять их в векторе op1 на "хорошие" значения, определённые по некоторым правилам. Выбор "хорошего" значения производится для каждого элемента независимо! Допустимые форматы операндов:

op1	**op2**	**op3**	**op4**
r128	r128	r128, m128, m32bcst/m64bcst	i8
r256	r256	r256, m256, m32bcst/m64bcst	i8
r512	r512	r512, m512, m32bcst/m64bcst	i8

Сначала определяется, в какую позицию второго столбца приведённой ниже таблицы попадает значение op2[i]. Затем из нужных позиций op3[i] из первого столбца находится индекс ind=0..15:

ind берётся из битов op3[i]	**Это определяет ind, когда op2[i]**
[3..0]	QNaN
[7..4]	SNaN
[11..8]	±0.0
[15..12]	1.0
[19..16]	-∞
[23..20]	+∞
[27..24]	конечный <0.0
[31..28]	конечный >0.0

Затем по индексу ind выбирается значение из приведённой ниже таблицы, которое и записывается на место элемента op1[i]: [1]

ind	**Ret[i]**
0	op1[i] (не меняется)
1	op2[i]
2	QNaN
3	QNaN Indefinite
4	-∞
5	+∞
6	∞ со знаком op2[i]
7	-0.0
8	+0.0
9	-1.0
10	+1.0
11	0.5
12	90.0
13	π/2
14	MaxSingle/MaxDouble
15	-MaxSingle/MaxDouble

И, наконец, если некоторый бит в op4=i8 установлен в "1", то каждый элемент op2 проверяется на заданное ниже значение, и, если проверка показала истину, то устанавливается определённый флаг в регистре MXCSR:

i8	**op2[i]**
0	0.0 → ZE:=1
1	0.0 → IE:=1
2	1.0 → ZE:=1
3	1.0 → IE:=1
4	SNaN → ZE:=1
5	-∞ → IE:=1
6	конечное <0.0 → IE:=1
7	+∞ → IE:=1

[1] Такой механизм называется двухуровневой табличной подстановкой (two-level look-up table).

Фантастически сложная команда, по сути – целая процедура в кремнии от 4-х параметров.

17.4.1. Задачи обработки изображений

Три пути ведут к знанию: путь размышления – это путь самый благородный, путь подражания – это путь самый лёгкий, и путь опыта – это путь самый горький.

Конфуций, V век до н.э.

В простых случаях, при обработки изображения (получение негатива, изменение яркости и контрастности, размытие и повышение чёткости и т.д.) некоторые преобразования применяются к каждому пикселю исходного изображения независимо. Пусть каждый пиксель, как обычно, описывается тремя величинами типа `byte`, это интенсивности красного, зелёного и синего цвета:

```
type Point=array[(R,G,B)] of byte;
     Image=array[0..N,0..M] of Point;
var  X: Point; { X[R]+X[G]+X[B]=1.0 }
```

Многие такие преобразования задаются вещественной матрицей, преобразовывающей старое значение точки (R_0,G_0,B_0) в новое значение (R_1,G_1,B_1):

$$\begin{pmatrix} R_1 \\ G_1 \\ B_1 \end{pmatrix} = \begin{pmatrix} R_0 \\ G_0 \\ B_0 \end{pmatrix} * \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

Рассмотрим преобразование, которое называется фильтр "Сепия". После этого преобразование изображение будет похоже на старые фотографии, сделанные в тёплых, светло-коричневых тонах. Вообще говоря, это преобразование определяется некоторым коэффициентов – интенсивностью этго светло-коричневого тона, но вы не будем обращать на это внимание, возьмём такую матрицу для "среднего" коэффициента:

$$\begin{pmatrix} R_1 \\ G_1 \\ B_1 \end{pmatrix} = \begin{pmatrix} R_0 \\ G_0 \\ B_0 \end{pmatrix} * \begin{pmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{pmatrix}$$

Отсюда получам формулы для элементов вектора (R_1,G_1,B_1):

R_1= (R_0,G_0,B_0) *(0.393,0.769,0.189)

G_1= (R_0,G_0,B_0) *(0.349,0.686,0.168)

B_1= (R_0,G_0,B_0) *(0.272,0.534,0.131)

Здесь у нас намечаются программистские трудности. Во-первых, надо получить целый вектор как результат скалярного произведения целого (типа `byte`) и вещественного (типа `single`) векторов, а во-вторых, компоненты итогового вектора (типа `byte`) надо получить с насыщением, т.е. каждый элемент вектора (R_1,G_1,B_1) не должен превысить 255. Напишем фрагмент программы для преобразования одного пикселя:

```
.data
N equ 1000
M equ 1000; изображение N*M точек
X db  N*M dup (?,?,?); X[0..N*M-1] of Point
A real4 0.393,0.769,0.189; A[0..2,0..2] of singlt
  real4 0.349,0.686,0.168
  real4 0.272,0.534,0.131
.code
; получение изображения A, цикл по всем пикселям
  movdqu xmm0,xmmword ptr A;    xmm0:=?,A[0]
  movdqu xmm1,xmmword ptr A+12; xmm1:=?,A[1]
  movdqu xmm2,xmmword ptr A+24; xmm2:=?,A[2]
   . . .
; ↓↓ RBX – адрес очередной точки
; ↓↓ xmm3:=?,longword(B,G,R)
```

```
  vpmovzxbd xmm3,[rbx]
; ↓↓ xmm3:=?,single(B,G,R)
  vcvtupi2ps xmm3,xmm3
; ↓↓ xmm4:=?,B*A02,G*A01,R*A00
  vmulps xmm4,xmm3,xmm0
; ↓↓ xmm4:=?,longword(B*A02,G*A01,R*A00)
  vcvtps2upi xmm4,xmm4
```

Вопросы и упражнения

Я не доверяю компьютеру, который не могу понять.

Стив Джобс

1. Какие достоинства и недостатки у векторных регистров по сравнению с вещественными регистрами `st(0)-st(7)` ?
2. Почему для векторных команд нет формата `r128,i128` ?
3. Почему операнды векторных команд нужно выравнивать в памяти на границу 16 или 32 байт ?
4. Как выровнить переменную `X` в секции данных на границу 64-х байт?
5. Какие три команды загружают выровненные на границу 32-х байт данные из **`.data`** в YMM регистр, и когда нужно использовать ту или иную из этих команд?
6. Напишите фрагмент программы, который загружает в регистр `AL` `n`-й байт регистра `YMM0` (`0<=n<16`).
7. В чём разница между обычным (вертикальным) и горизонтальным сложением векторных регистров?
8. Чем может быть опасна работа со строками, не выровненными на границу 16 байт?
9. Как задаётся длина строк для векторных команд обработки строк?

Список литературы

Сделай первый шаг, и ты поймёшь, что не всё так страшно.
Луций Анней Сенека (младший)

1. Йо Ван Гуй. Программирование на ассемблере x64 от начального уровня до профессионального использования AVX. – М.: ДМК Пресс, 2021. – 332 с.
2. Д. Куссвюрм. Профессиональное программирование на ассемблере x64 с расширениями AVX, AVX2 и AVX-512 (2-изд.). – М.: ДМК Пресс, 2021. – 628 с.
3. Методическая документация. «Руководство разработчика по использованию набора инструкций AVX-512». – М.: Межведомственный суперкомпьютерный центр Российской академии наук, 2019. – 58 с.
4. Intel Architecture Instruction Set Extensions Programming Reference. − Intel Corporation, 2015.
5. Microsoft Macro Assembler reference. – https://docs.microsoft.com/en-us/cpp/assembler/masm/microsoft-macro-assembler-reference
6. Randall Hyde. The Art of 64-Bit Assembly Language, vol., 2022. 1002 p.
7. Irvine K.R. Assembly language for x86 processors, 8d Ed, 2019. 999 p.

---

[i] Для продвинутых читателей. Попытка использовать младшие части YMM регистров как самостоятельные приводит к большим накладным расходам в работе процессора. Действительно, по команде с использованием XMM регистра процессору приходится где-то запоминать старшую часть соответствующего YMM регистра, а для последующей команды с этим регистром восстанавливать эту старшую часть. Напоминаем, что большая часть работы в конвейере производится на с настоящими (физическими) регистрами процессора, а с временными (теневыми) регистрами. Получается, что на самом деле, скажем, для регистра YMM0 приходится выделять два теневых регистра (XMM0 и YMM0). Регистры XMM часто используются в программах для хранения вещественных чисел, поэтому при переключении контекста с одного программного потока на другой операционная система вынуждена сохранять, а затем восстанавливать старшие части регистров YMM, даже если сам поток использует только регистры XMM.

[ii] Для продвинутых читателей. В этой главе нам часто будут встречаться команды-синонимы, которые, с точки зрения программиста, выполняются одинаково. Тем не менее, эти команды имеют разные коды операций, и встаёт вопрос, зачем так сделано. Причина заключается в универсальности векторных регистров. Мы понимаем, что целочисленный регистр общего назначения (скажем, RAX) должен быть "устроен" совсем не так, как вещественный регистр st(0), к нему призязан еэг и "припаяны" разные логические схемы для выполнения принципиально разных (целых и вещественных) операций.

Ясно, что так же надо себя вести и при работе с векторными регистрами. Когда векторный регистр загружается из памяти командой **vmovaps**, программист "намекает" компьютеру, что в дальнейшем он будет выполнять на этом регистре операции с вещесвенными числами одинарной точности. Наоборот, используя для загрузки регистра команду **vmovdqa**, программист как бы говорит процессору, что намеревается работать на этом регистре с вектором целых чисел.

Соответственно, готовясь к выполнению векторной команды, планировщик конвейера выбирает из регистрового файла (см. разд. 14.2.1) векторный регистр, "прикреплённый" к исполнительному устройству для работы либо с вещественными, либо же с целыми числами. Спрашивается, а что будет, если загрузить на регистр (скажем, YMM0) вектор целых чисел, а потом использовать для этого регистра, операцию сложения вещественных чисел?

В этом случае планировщик конвейера будет вынужден отменить выполнение команды, так как выделенный регистр прикреплён к устройству, которое не умеет складывать вещественные числа. Придётся выделить новый теневой регистр, прикрепить его к устройству для операций с вещественными числами и перегрузить данные со старого на новый регистр. Это потеря времени, в экономике похожая на сбой логистической цепочки (сырьё привезли не на то предприятие). Но, с другой стороны, наш процессор достаточно "умный", планировщик работы конвейера анализирует Вашу программу на десятки команд вперёд и, увидев такое "безобразие", может и сам исправить Вашу ошибку, заранее прикрепив регистр к нужному устройству. Так что программист на Ассемблере (да и не очень хороший компилятор с языков высокого уровня) может уже не заботиться о таких "мелочах".

Отметим, что некоторые ЭВМ могут решать эту задачу "по-простому", привязывая векторные регистры с определёнными номерами к типу обрабатываемых на этих регистрах данных. Тогда в коде операции уже не надо указывать тип обрабатываемых данных, он определяется по номеру регистра.