Глава 1 Введение

Эта глава содержит всю важнейшую информацию, которая понадобится вам, чтобы начать использовать flat assembler. Если у вас уже есть опыт программирования на ассемблере, вам достаточно прочитать лишь первую главу перед использованием этого компилятора.

1.1 Обзор компилятора

Flat assembler - это быстрый компилятор ассемблера для процессоров с архитектурой x86, который делает множественные проходы для оптимизации размера сгенерированного машинного кода. Он способен скомпилировать сам себя и существуют версии для разных операционных систем. Все версии созданы для использования с помощью системной командной строки и в обращении с ними нет разницы.

1.1.1 Системные требования

Для работы всех версий требуется 32-битный процессор с архитектурой x86 (как минимум 80386), хотя также он должен обрабатывать программы для 16-битных процессоров с архитектурой x86. DOS-версия требует ОС, совместимую с MS DOS 2.0, Windows-версия требует консоль Win32, совместимую с версией 3.1.

1.1.2 Работа с компилятором из командной строки

Для запуска FASMa из командной строки вам понадобится ввести два параметра: первый - это путь к файлу с кодом, второй - путь к файлу-адресату информации. Если второй параметр не задан, название файла вывода будет создано автоматически. После показа короткой информации о названии программы и её версии, компилятор считает информацию из файла с кодом и скомпилирует её. После успешной компиляции FASM запишет сгенерированный код в файл-адресат и выведет краткую информацию о завершенном процессе; в противном случае он выведет информацию о первой ошибке.

Исходник должен быть текстовым файлом и может быть создан в любом текстовом редакторе. Обрыв строки допускается и в стандарте DOS, и в стандарте Unix, табуляции обрабатываются как пробелы.

В командную строку вы также можете включить опцию "-m", за которой должно следовать число, указывающее, сколько килобайт памяти может быть максимально задействовано flat assembler'ом. В случае DOS-версии эта опция ограничивает лишь использование расширенной памяти. Опция "-p" со следущим за ним числом может быть использована для того, чтобы ограничить количество проходов, которое будет делать ассемблер. Если код не может быть создан заданным числом проходов, ассемблирование прекратится с сообщением об ошибке. Максимальное значение этой опции равно 65536, а значение по умолчанию равно 100.

Не существует опций, оказывающих воздействие на выходные данные компилятора, вся необходимая flat assembler'y информация должна содержаться в исходном коде. Например, для установки формата файла-адресата используется директива "format" в начале кода.

1.1.3 Сообщения компилятора

Как было сказано выше, после успешной компиляции FASM выводит на экран сводку о проделанной работе. Она включает информацию о том, сколько было сделано проходов, сколько времени это заняло, и сколько байт записано в файл-аресат.

Вот пример такой сводки:

```
flat assembler  version 1.64
38 passes, 5.3 seconds, 77824 bytes.
```

В случае ошибки во время компиляции, программа выведет на экран сообщение об ошибке. Например, когда компилятор не может найти файл ввода, он покажет следующее сообщение:

```
flat assembler  version 1.64
error: source file not found.
```

Если ошибка связана с определенной частью кода, будет выведена строка, которая её вызвала. Также, чтобы помочь вам найти эту ошибку, будет показано место этой строки в коде, например:

```
flat assembler  version 1.64
example.asm [3]:
```

```
        mob     ax,1
error: illegal instruction.
```

Это значит, что в третьей строке файла "example.asm" компилятор встретил неопознанную инструкцию. Если строка, вызвавшая ошибку, содержит макрос, также будет выведена строка в формулировке макроса, которая сгенерировала ошибочную инструкцию:

```
flat assembler  version 1.64
example.asm [6]:
        stoschar 7
example.asm [3] stoschar [1]:
        mob     al,char
error: illegal instruction.
```

Это значит, что макрос в шестой строке файла "example.asm" создал неопознанную инструкцию в первой строке своей формулировки.

1.1.4 Форматы вывода

По умолчанию, если в исходнике нет директивы "format", flat assembler направляет сгенерированный код на вывод, создавая таким образом простой двоичный файл. По умолчанию он создает 16-битный код, но вы всегда можете переключить его в 32-битный или 16-битный режим, используя директивы "use32" или "use16". Выбор некоторых форматов файла-адресата автоматически переключает компилятор в 32-битный режим. Подробнее читайте о форматах, которые можете выбрать, в 2.4.

Весь сгенерированный код в файле-адресате всегда идет в том же порядке, что и написанный в исходнике.

1.2 Синтаксис ассемблера

Информация, изложенная ниже, предназначена главным образом программистам, которые прежде использовали другие компиляторы ассемблера. Если вы новичок, читайте учебники по программированию на ассемблере.

Flat assembler по умолчанию использует Интеловский синтаксис ассемблерных инструкций, однако вы можете переделать их, используя возможности препроцессора (макросы и символьные константы). Он также имеет собственный набор директив - инструкций для компилятора.

Все символы, определенные внутри кода, чувствительны к регистру.

1.2.1 Синтаксис инструкций

Инструкции в ассемблере разделяются разрывами строк, и одна инструкция должна располагаться на одной строке. Если строка содержит точку с запятой, не считая точек с запятой, заключенных в кавычки, остаток её считается комментарием и компилятор её проигнорирует. Если строка содержит символ "\"(в конце концов точка с запятой и комментарий могут за ней следовать), то следующая срока прикрепляется к этой точке. После символа "\" строка не должна содержать ничего, кроме комментариев, следующих за точкой с запятой.

Каждая строка в коде - это последовательность отдельных предметов,каждый из которых может принадлежать к одному из треж типов. Первый тип - это знаки символов, которыми являются специальные знаки, которые трактуются отдельно, даже если не отделены побелами о других символов. Любой из "+-*/=<>()[]{}:,|&~#`" - это знак символа. Последовательность других знаков, отделенная от остальных либо пробелами, либо знаками символов, это символ. Если первый знак такого символа двойная либо одинарная кавычка, он объединяет любую последовательность символов, даже специальных, следующих за ним, в строку. Она должна заканчиваться таким же знаком, каким начиналась (двойной либо одинарной кавычкой). Однако если встречаются две кавычки подряд (без знаков между ними), они также влючаются в строку и она продолжается. Символы, отличные от знаков символов и строк, заключенных в кавычки, могут использоваться как имена, поэтому они также называются символами имен.

Каждая инструкция состоит из мнемоника и различного числа операндов, разделенных запятыми. Операндом может быть регистр, непосредственное значение или адрес в памяти, он также может предварен оператором размера, чтобы определить или изменить его размер (таблица 1.1). Названия возможных регистров вы можете найти в таблице 1.2, их размеры не могут быть изменены. Непосредственные значения могут быть определены любым числовым значением.

Если операнд - это данные в памяти, адрес этих данных (также любого числового выражения, но оно может содержать регистры) должен быть заключен в крадратные скобки или предворен оператором "ptr". Например, инструкция "mov eax,3" поместит число 3 в регистр EAX, а инструкция "mov eax,[7]" поместит 32-битное значение из адреса 7 в EAX, и инструкция "mov byte [7],3" поместит число 3 в байт по адресу 7, это можно записать еще так: "mov byte ptr 7,3". Для того, чтобы установить, какой сегментный регистр будет использоваться для адресации, нужно поставить его название с двоеточием перед адресом внутри квадратных скобок или после оператора "ptr".

Таблица 1.1 Размеры операторов

Оператор	Биты	Байты
byte	8	1
word	16	2
dword	32	4
fword	48	6
pword	48	6
qword	64	8
tbyte	80	10
qword	80	10
dqword	128	16

Таблица 1.2 Регистры

Тип	Биты	Регистры							
General	8	al	cl	dl	bl	ah	ch	dh	bh
	16	ax	cx	dx	bx	sp	bp	si	di
	32	eax	ecx	edx	ebx	esp	esi	edi	ebp
Segment	16	es	cs	ss	ds	fs	gs		

Control	32	cr0		cr2	cr3	cr4			
Debug	32	dr0	dr1	dr2	dr3			dr6	dr7
FPU	80	st0	st1	st2	st3	st4	st5	st6	st7
MMX	64	mm0	mm1	mm2	mm3	mm4	mm5	mm6	mm7
SSE	128	xmm0	xmm1	xmm2	xmm3	xmm4	xmm5	xmm6	xmm7

1.2.2 Описание данных

Чтобы описать данные или зарезервировать для них место, используйте одну из директив, перечисленных в таблице 1.3. За директивой описания данных должно следовать одно или несколько чиловых значений, разделенных запятыми. Эти выражения определяют значения для простейших элементов данных, размер которых зависит от того, какая директива используется. Например "db 1,2,3" описывает три байта со значениями 1, 2 и 3 соответственно.

Директивы "du" и "db" также поддерживают сроки любой длины, заключенные в кавычки, которые будут конвертированы в последовательность байтов, если использована директива "db", или в последовательность слов с нулевым верхним байтом, если использована директива "du". Например, "db 'abc'" определяет три байта со значениями 61, 62 и 63.

Директива "dp" или её синоним "df" допускают, чтобы значения состояли из двух числовых выражений, разделенных двоеточием, где первое значение - это верхнее слово, а второе - это нижнее двойное слово значения дальнего указателя. Также "dd" допускает такие указатели, состоящие из двух слов, разделенных двоеточием, и "dt" допускает слово и четверное слово, разделенные двоеточием, четверное слово запоминается первым. Директива "dt" с одним параметром допускает только значения с плавающей точкой и создает данные в FPU-формате двойной расширенной точности.

Все вышеперечисленные директивы поддерживают использование специального оператора "dup" для сождания копий данных значений. Количество дубликатов должно стоять перед этим оператором, а их значение должно стоять после - это может быть даже цепь значений, разделенных запятыми, но эта цепь должна быть заключена в скобки, например "db 5 dup (1,2)" определяет пять копий данной последовательности из двх байтов.

"file" - это специальная директива и её синтаксис может быть различным. Эта директива включает цепь байтов из файла. В качестве параметра за ней должно идти в кавычках имя файла, далее, опционально, двоеточие и числовое выражение, указывающее начало цепочки байтов, далее, также опционально, запятая и числовое выражение, определяющее количество байтов в этой цепочке (если этот параметр не определен, то будут включены все данные до конца файла). Например, "file 'data.bin'" включит весь файл как двоичные данные, а "file 'data.bin':10h,4" включит только четыре байта, начиная со смещения 10h.

За директивой резервирования данных должно следовать одно числовое выражение, значение которого определяет количество резервируемых ячеек установленного размера. Все директивы описания данных также поддерживают значение "?", которое значит, что этой ячейке не должно быть присвоено какое-то значение. Эффект от этой директивы такой же, как от директивы резервирования данных. Неинициализированные данные не могут быть включены в файл вывода, и, таким образом, их значения всегда будут считаться неизвестными.

Таблица 1.3 Директивы данных

Размер (байты)	Определение данных	Резервирование данных
1	db file	rb
2	dw du	rw
4	dd	rd
6	dp df	rp rf
8	dq	rq
10	dt	rt

1.2.3 Константы и метки

В числовых выражениях вместо чисел вы также можете использовать константы и метки. Чтобы назначить их, используйте специальные директивы. Каждая метка может быть определена только однажды и она будет доступна из любой части кода (даже перед местом, где она была определена). Константа может быть переопределена много раз, но в этом случае она будет доступна только после присвоения значения и всегда будет равна значению из последнего определения перед местом, в котором она использована. Если константа определена лишь однажды, она, так же как и метка, доступна из любой части кода.

Определение константы состоит из имени константы, знака "=" и числового выражения, которое после вычисления становится значением константы. Это значение всегда вычисляется в то же время, что и определение константы. Например, с помощью директивы "count = 17" вы можете определить константу "count" и после использовать её в инструкциях ассемблера, таких как "mov cx,count" - которая превратится в "mov cx,17" во время процесса компиляции.

Существуют разные способы определения меток. Простейший из них - двоеточие после названия метки. За этой директивой на той же строке даже может следовать другая инструкция. Она определяет метку, значение которой равно смещению точки, в которой она определена. Этот метод обычно используется, чтобы пометить места в коде. Другой способ - это следование за именем метки (без двоеточия) какой-нибудь директивы описания данных. Метке присваивается значение адреса начала определенных в директиве данных и запоминается компилятором как метка для данных с размером ячейки, заданной директивой из таблицы 1.3.

Метка может быть обработана как константа со значением, равным смещению помеченного кода или данных. Например, если вы определяете данные, используя помеченную директиву "char db 224", для того, чтобы поместить адрес начала этих данных в регистр BX, вам нужно использовать инструкцию "mov bx,char", а для того, чтобы поместить в регистр DL значение байта, на который ссылается "char", нужно использовать "mov dl,[char]" (или "mov dl,ptr char"). Если вы попытаетесь ассемблировать "mov ax,[char]", FASM выдаст ошибку, так как он сравнивает размеры операндов, которые должны быть равны. Вы можете принудительно проассемблировать эту инструкцию, изменяя размер операнда: "mov ax, word [char]", но помните, что эта инструкция прочитает два байта, начинающихся с адреса "char", тогда как он был определен как один байт.

Последний и самый гибкий способ задания меток - это использование директивы "label". За этой директивой должно следовать имя метки, далее, опционально, размер оператора (может предваряться двоеточием), и далее, также опционально, оператор "at" и числовое выражение, определяющее адрес, на который данная метка должна ссылаться. Например, "label wchar word at char" определяет новую метку для 16-битных данных по адресу "char". Теперь инструкция "mov ax,[wchar]" после компиляции будет выглядеть так же, как "mov ax,word [char]". Если адрес не указан, директива "label" будет ссылаться на текущий адрес. Таким образом, "mov [wchar],57568" скопирует два байта, тогда как "mov [char],224" скопирует один байт на тот же адрес.

Метка, имя которой начинается с точки, обрабатывается как локальная, и её имя прикрепляется к имени последней глобальной метки (с названием, начинающемся с чего угодно, кроме точки) для создания полного имени этой метки. Так, вы можете использовать короткое имя (начинающееся с точки) где угодно перед следующей глобальной меткой, а в других местах вам придется пользоваться полным именем. Метки, начинающиеся с двух точек - исключения. Они имеют свойства глобальных, но не создают новый префикс для локальных меток.

"@@" обозначает анонимную метку, вы можете определить её множество раз. Символ "@b" (или эквивалент "@r") ссылается на ближайшую предшествующую анонимную метку, а символ "@f" ссылается на ближашую после неё анонимною метку. Эти специальные символы нечувствительны к регистру.

1.2.4 Числовые выражения

В предыдущих примерах все числовые выражения были обычными числами, константами или метками. Но они могут быть более сложными, использовать арифметические или логические опреаторы для вычисления во время компиляции. Все эти операторы с их значениями приоритета перечислены в таблице 1.4.

Операции с высшим приоритетом выполняются первыми, однако вы, конечно, можете изменить такой образ действий, заключив некоторые части выражения в скобочки. "+", "-", "*" и "/" - это стандартные арифметические операции, "mod" вычиляет остаток от деления нацело. "and", "or", "xor", "shl", "shr" и "not" совершают те же логичекие операции, что и инструкции ассемблера с такими же названиями. "rva" характерна только для формата вывода PE и производит превращение адреса в RVA.

Числа в выражениях по умолчанию обрабатываются как десятичные, двоичные числа должны иметь "b" в конце, восьмеричные числа должны заканчиваться на букву "o", шестнадцатеричные цифры должны начиаться символами "0x" (как в языке C), или символом "$" (как в языке Pascal) или должны заканчиваться буквой "h". Также заключенная в кавычки строка при включении в выражение будет конвертирована в число - первый символ станет минимальным значащим байтом числа. Числовые выражения, используемые как значения адреса, могут также содержать юбой из общих регистров, используемых для адресации, они могут быть сложены или умножены на подходящие значения так, как это позволено в инструкциях архитектуры x86.

Также есть несколько специальных символов, которые могут быть использованы в числовом выражении. Первое - это "$", которое всегда равно значению текущего смещения, тогда как "$$" равно базовому адресу текущего диапазона адресов. Следующий символ - "%" - это номер текущего повтора в частях кода, которые повторяются, благодаря использованию некоторых специальных директив (смотрите 2.2). Также существует символ "%t", кторый всегда равен текущей отметке времени.

Любое численное выражение также может состоять из одного значения с плавающей точкой (flat assembler не может производить во время компиляции операции с плавающей точкой) в научной записи. Для распознания компилятором, эти значения должны содержать в конце букву "f", либо включать в себя по крайней мере один символ "." или "E". Так, "1.0", "1E0" и "1f" определяют одно и то же значение с плавающей точкой, когда как просто "1" определяет целочисленное значение.

Таблица 1.4 Арифметические и логические операторы в порядке приоритета

Приоритет	Операторы
0	+ -
1	* /
2	mod
3	and or xor
4	shl shr
5	not
6	rva

1.2.5 Переходы и вызовы

Операнд любого перехода или инструкция вызова может предваряться не только операторами размера, но также одним из операторов, определяющих тип перехода: "near" или "far". Например, если ассемблер в 16-битном режиме, инструкция "jmp dword [0]" станет далеким переходом, а если ассемблер в 32-битном режиме, она станет близким переходом. Чтобы заставить эту инструкцию обрабатываться по-разному, используйте формы "jmp near dword [0]" или "jmp far dword [0]".

Если операнд близкого перехода это немедленное значение, ассемблер, если возможно, сгенерирует кратчайший вариант этой инструкции перехода (но не будет создавать 32-битную инструкцию в 16-битном режиме или 16-битную инструкцию в 32-битном режиме, если оператор размера точно её не определит). Заданием оператора размера вы можете заставить ассемблер всегда генерировать длинный вариант (например, "jmp word 0" в 16-битном режиме или "jmp dword 0" в 32-битном режиме) или всегда создавать короткий вариант и завершаться с ошибкой, когда это невозможно (например "jmp byte 0").

1.2.6 Установки размера

Если инструкция использует некоторую адресацию в памяти, по умолчанию будет генерироваться кратчайшая 8-битная форма, если значение адреса попадает в нужный диапазон, но он может быть изменен с помощью операторов "word" и "dword" перед адресом в квадратных скобках (или после оператора "ptr"). Такое размещение оператора размера также может быть использовано для установки размера адреса, отличного от размера, установленного в данном режиме по умолчанию.

Инструкции "adc", "add", "and", "cmp", "or", "sbb", "sub" и "xor" с первым 16-ти или 32-битным операндом по умолчанию генерируются в укороченной 8-битной форме, если второй операнд - это непосредственное значение, применимое для предписанных 8-битных значений. Она также может быть изменена операторами "word" и "dword" перед такими значениями. Сходные правила применимы к инструкции "imul" с непосредственным

значениям в качестве последнего операнда.

Непосредственное значение как операнд для инструкции "push" без оператора размера, по умолчанию обрабатывается как слово, если ассемблер 16-битном режиме, и как двойное слово, если FASM в 32-битном режиме. Короткая 8-битная форма используется по возможности, операторы размера "word" и "dword" могут заставить инструкцию "push" быть сгенерированной в более длинной форме. Мнемоники "pushw" и "pushd" указывают ассемблеру сгенерировать 16-битный или 32-битный код без принуждения его использовать длинную форму инструкции.

Глава 2 Описание инструкций

Эта глава содержит детальную информацию об инструкциях и директивах, поддерживаемых FASMом. Директивы определения констант и меток уже описаны в 1.2.3, все остальные директивы будут описаны ниже в этой главе.

2.1 Инструкции архитектуры x86

В этом параграфе вы найдете всю информацию о синтаксисе и назначении инструкций ассемблера. Если вам нужно больше технической информации, смотрите Intel Architecture Software Developer's Manual. Инструкции ассемблера состоят из мнемоника (имени инструкции) и нескольких операндов (от нуля до трех). Если операндов два или три, то обычно первым идет адресат, а вторым источник. Операндом может быть регистр, память или непосредственное значение (подробнее о синтаксисе операндов смотрите в 1.2). После описания каждой инструкции ниже будут примеры разных комбинаций операндов, если, конечно, она содержит операнды.

Некоторые инструкции работают как префиксы и могут быть перед другой инструкцией на той же строке. На одной строке может несколько префиксов. Каждое имя сегментного регистра это тоже мнемоник инструкции-префикса, хотя рекомендуется использовать замещение сегмента внутри квадратных скобок вместо этих префиксов.

2.1.1 Инструкции перемещения данных

"mov" переносит байт, слово или двойное слово из операнда-источника в операнд-адресат. Этот мнемоник может передавать данные между регистрами общего назначения, из этих регистров в память, обратно, но не может перемещать данные из памяти в память. Он также может передавать непосредственное значение в регистр общего назачения или в память, сегментный регистр в регистр общего назначения или в память, регистр общего назначения в сегментный регистр или в память, контрольный или отладочный регистр в регистр общего назначения и назад. "mov" может быть ассемблирована только если размер операнда-источника и размер операнда-адресата совпадают. Ниже приведены примеры каждой из перечисленных комбинаций:

```
mov bx,ax       ; из регистра общего назначения в регистр общего назначения
mov [char],al   ; из регистра общего назначения в память
mov bl,[char]   ; из памяти в регистр общего назначения
mov dl,32       ; непосредственное значение в регистр общего назначения
mov [char],32   ; непосредственное значение в память
```

```
mov ax,ds       ; из сегментного регистра в регистр общего назначения
mov [bx],ds     ; из сегментного регистра в память
mov ds,ax       ; из регистра общего назначения в сегментный регистр
mov ds,[bx]     ; из памяти в сегментный регистр
mov eax,cr0     ; из контрольного регистра в регистр общего назначения
mov cr3,ebx     ; из регистра общего назначения в контрольный регистр
```

"xchg" меняет местами значения двух операндов. Инструкция может поменять два байтовых операнда, операнды размером в слово и размером в двойное слово. Порядок операндов не важен. В их роли могут выступать два регистра общего назначения либо регистр общего назначения и адрес в памяти. Например:

```
xchg ax,bx      ; меняет местами два регистра общего назначения
xchg al,[char]  ; регистр общего назначения и память
```

"push" уменьшает значение указателя стекового фрейма (регистр ESP), потом переводит операнд на верх стека, на который указывает ESP. Операндом может быть память, регистр общего назначения, сегментный регистр или непосредственное значение размером в слово или двойное слово. Если операнд - это непосредственное значение и его размер не определен, то в 16-битном режиме по умолчанию он обрабатывается как слово, а в 32-битном режиме как двойное слово. Мнемоники "pushw" и "pushd" - это варианты этой инструкции, которые сохраняют соответственно слова и двойные слова. Если на одной строке содержится несколько операндов (разделенных пробелами, а не запятыми), компилятор проассемблирует цепь инструкций "push" с этими операндами. Вот примеры с одиночными операндами:

```
push ax         ; сохраняет регистр общего назначения
push es         ; сохраняет сегментный регистр
pushw [bx]      ; сохраняет память
push 1000h      ; сохраняет непосредственное значение
```

Инструкция "pusha" сохраняет в стек содержимое восьми регистров общего назначения. У неё нет операндов. Существует две версии этой инструкции: 16-битная и 32-битная. Ассемблер автоматически генерирует версию, соответствующую текущему режиму, но, используя мнемоники "pushaw" или "pushad",это можно изменить для того, чтобы всегда получать, соответственно, 16- или 32-битную версию. 16-битная версия этой инструкции сохраняет регистры общего назначения в таком порядке: AX, CX, DX, BX, значение регистра SP перед тем, как был сохранен AX, далее BP, SI и DI. 32-битная версия сохраняет эквивалентные 32-битные регистры в том же порядке.

"pop" переводит слово или двойное слово из текущей верхушки стека в операд-адресат и после уменьшает ESP на указатель на новую верхушку стека. Операндом может служить память, регистр общего назначения или сегментный регистр. Мнемоники "popw" и "popd" - это варианты этой инструкции, восстанавливающие соответственно слова и двойные слова. Если на одной строке содержится несколько операндов, разделенных пробелами, компилятор ассемблирует цепочку инструкций с этими операндами.

```
pop bx          ; восстанавливает регистр общего назначения
pop ds          ; восстанавливает сегментный регистр
popw [si]       ; восстанавливает память
```

"popa" восстанавливает регистры, сохраненные в стек инструкцией "pusha", кроме сохраненного значения SP (или ESP)? который будет проигнорирован. У этой инструкции нет операндов. Чтобы ассемблировать 16 или 32-битную версию этой инструкции, используйте мнемоники

"popaw" или "popad".

2.1.2 Инструкции преобразования типов.

Инструкции преобразования типов конвертируют байты в слова, слова в двойные слова и двойные слова в четверные слова. Эти преобразования можно совершить, используя знаковое или нулевое расширение. Знаковое расширение заполняют дополнительные биты большего операнда значением бита знака меньшего операнда, нулевое расширение просто забивает их нулями.

"cwd" и "cdq" удваивают размер регистра AX или EAX соответственно и сохраняет дополнительные биты в регистр DX или EDX. Преобразование делается, используя знаковое расширение. Эти инструкции не имеют операндов.

"cbw" растягивает знак байта AL по регистру AX, а "cwde" растягивает знак слова AX на EAX. Эти инструкции также не имеют операндов.

"movsx" преобразует байт в слово или в двойное слово и слово в двойное слово, используя знаковое расширение. "movzx" делает то же самое, но используя нулевое расширение. Операндом-источником может быть регистр общего назначения или память, тогда как операндом-адресатом должен быть регистр общего назначения. Например:

```
movsx ax,al         ; байт в слово
movsx edx,dl        ; байт в двойное слово
movsx eax,ax        ; слово в двойное слово
movsx ax,byte [bx]  ; байт памяти в слово
movsx edx,byte [bx] ; байт памяти в двойное слово
movsx eax,word [bx] ; слово памяти в двойное слово
```

2.1.3 Двоичные арифметические инструкции

"add" заменяет операнд-адресат суммой операнда-источника и адресата и ставит CF, если было переполнение. Операндами могут байты, слова или двойные слова. Адресатом может быть регистр общего назначения или память, источником регистр общего назначени или непосредственное значение. Также это может быть память, если адресат - это регистр.

```
add ax,bx       ; прибавляет регистр к регистру
add ax,[si]     ; прибавляет память к регистру
add [di],al     ; прибавляет регистр к памяти
add al,48       ; прибавляет непосредственное значение к регистру
add [char],48   ; прибавляет непосредственное значение к памяти
```

"adc" суммирует операнды, прибавляет единицу, если стоит CF и заменяет адресат результатом. Приавила для операндов такие же как с инстукцей "add". "add" со следующими за ней несколькими инструкциями "adc" может быть использована для сложения чисел длиннее, чем 32 бита.

"inc" прибавляет к операнду единицу, он не может изменить CF. Операндом может быть регистр общего назначения или память, размером он может быть в байт, слово или двойное слово.

```
inc ax          ; прибавляет единицу к регистру
inc byte [bx]   ; увеличивает единицу к памяти
```

"sub" вычитает операнд-источник от операнда адресата и заменяет адресат результатом. Если требуется отрицательный перенос, устанавливается CF. Правила для операндов такие же, как с инструкцией "add".

"sbb" вычитает источник из адресата, отнимает единицу, если установлен CF и заменяет адресат результатом. Правила для операндов такие же, как с инструкцией "add". "sub" со следущими за ней несколькими инструкциями "sbb" может быть использована для вычитания чисел длиннее, чем 32 бита.

"dec" вычитает из операнда единицу, не может изменить CF. Правила для операнда такие же, как с инструкцией "inc".

"cmp" вычитает операнд-источник из оператора-адресата. Эта инструкция может устанавливать флаги, как и "sub", но не вносит изменения в операнды. Правила для операндов такие же, как с инструкцией "sub".

"neg" отнимает от нуля целочисленный операнд с знаком. Эффект от этой инструкции - это смена знака операнда с положительного на отрицательный или с отрицательного на положительный. Правила для операндов такие же, как с инструкцией "inc".

"xadd" меняет местами операнд-адресат и операнд-источник, потом загружает сумму двух значений в операнд-адресат. Правила для операндов такие же, как с инструкцией "add". Все вышеперечисленные инструкции изменяют флаги SF, ZF, PF и OF. SF всегда принимает значение, равное биту знака результата, ZF устанавливается, если результат равен нулю, PF устанавливается, если восемь битов нижнего разряда содержат четное число единиц, OF устанавливается, если результат слишком большой для положительного числа или слишком маленький для отрицательного (исключая бит знака) для того, чтобы уместиться в операнде-адресате.

"mul" выполняет беззнаковое перемножение операнда и аккумулятора. Если операнд - байт, процессор умножает его на содержимое AL и возвращает 16-битный результат в AH и AL. Если операнд - слово, процессор умножает его на содержимое AX и возврщает 32-битный результат в DX и AX. Если же операнд - это двойное слово, процессор умножает его на содержимое EAX и возвращает 64-битный результат в EDX и EAX. "mul"устанавливает CF и OF, если верхняя половина результата ненулевая, иначе они очищаются. Правила для операндов такие же, как с инструкцией "inc".

"imul" выполняет знаковое перемножение операндов. У этой инструкции есть три вариации. Первая имеет один операнд и работает так же, как инструкция "mul". Вторая имеет два операнда, и здесь операнд-адресат умножается на операнд-источник и результат заменяет операнд-адресат. Этоим операндом может быть регистр общего назначения, память или непосредственное значение. Третья форма инструкции имеет три операнда, операндом-адресатом должен быть регистр общего назначения, длиной в слово или в двойное слово, операндом-источником может быть регистр общего назначения или память, третьим операндом должно быть непосредственное значение. Источник умножается на непосредственное значение и результат помещается в регистр-адресат. Все три формы вычисляют результат размером в два раза больше размера операндов и ставят CF и OF, если верхняя часть результата ненулевая, но вторая и третья формы усекают результат до размера операндов. Так, их можно использовать для беззнаковых операндов, потому что нижняя половина результата одна и та же для знаковых и беззнаковых операндов. Ниже вы видите примеры всех трех форм:

```
imul bl         ; умножение аккумулятора на регистр
imul word [si]  ; умножение аккумулятора на память
imul bx,cx      ; умножение регистра на регистр
```

```
imul bx,[si]    ; умножение регистра на память
imul bx,10      ; умножение регистра на непосредственное значение
imul ax,bx,10   ; регистр, умноженный на непосредственное значение, в регистр
imul ax,[si],10 ; память, умноженная на непосредственное значение, в регистр
```

"div" производит беззнаковое деление аккумулятора на операнд. Делимое (аккумулятор) размером в два раза больше делителя (операнда), частное и остаток такого же размера , как и делитель. Если делитель - байт, делимое берется из регистра AX, частное сохраняется в AL, а остаток - в AH. Если делитель - слово, верхняя половина делимого берется из DX, а нижняя - из AX, частное сохраняется в AX, а остаток - в DX. Если делитель - двойное слово, верхняя половина делимого берется из EDX, а нижняя - из EAX, частное сохраняется в EAX, а остаток - в EDX. Правила для операндов такие же, как с инструкцией "mul".

"idiv" выполняет знаковое деление аккумулятора на операнд. Инструкция использует те же регистры, что и "div", правила для для операнда тоже такие-же.

2.1.4 Десятичные арифметические инструкции

Десятичная арифметика представлена в виде соединения двоичных арифметических инструкций (описанных в предыдущем параграфе) с десятичными арифметическиими инструкциями. Десятичные арифметические инструкции используются для того, чтобы приспособить результаты предыдущей двоичной арифметической операции для создания допустимого сжатого или несжатого (?) десятичного результата, или приспособить входные данные для последующей двоичной арифметической операции так, чтобы эта операция также давала допустимый сжатый или несжатый десятичный результат.

"daa" прилаживает результат сложения двух допустимых сжатых десятичных числа к AL. "daa" всегда должна следовать за суммированием двух пар сжатых десятичных цифр (один знак в каждой половине байта), чтобы получить как результат пару допустимых сжатых десятичных символов. Если потребуется перенос, будет установлен флаг переноса. У этой инструкции нет операндов.

"das" прилаживает результат вычитания двух допустимых сжатых десятичных числа к AL. "das" всегда должна следовать за вычитанием одной пары сжатых десятичных цифр (один знак в каждой половине байта) из другой, чтобы получить как результат пару допустимых сжатых десятичных символов. Если потребуется отрицательный перенос, будет установлен флаг переноса. У этой инструкции нет операндов.

"aaa" изменяет содержимое регистра AL на допустимое несжатое десятичное число и обнуляет верхние четыре бита. "aaa" всегда должна следовать за сложением двух несжатых десятичных операндов в AL. Если необходим перенос, устанавливется флаг переноса и увеличивается на единицу AH. У этой инструкции нет операндов.

"aas" изменяет содержимое регистра AL на допустимое несжатое десятичное число и обнуляет верхние четыре бита. "aas" всегда должна следовать за вычитанием одного несжатого десятичного операнда из другого в AL. Если необходим перенос, устанавливется флаг переноса и уменьшается на единицу AH. У этой инструкции нет операндов.

"aam" корректирует результат умножения двух допустимых несжатых десятичных чисел. Для создания правильного десятичного результата инструкция должна всегда следовать за умножением двух десятичных чисел. Цифра верхнего регистра передается в AH, а нижнего - в AL.

Обобщенная версия этой инструкции делает возможной подгонку содержимого AX для создания двух несжатых цифр с любым основанием. Стандартная версия этой инструкции не имеет операндов, у обобщенной версии есть один операнд - непосредственное значение, определяющее основание создавакмых чисел.

"aad" модифицирует делимое в AH и AL, чтобы подготовится к делению двух допустимых несжатых десятичных операндов так, чтобы частное стало доустимым несжатым десятичным числом. AH должен содержать цифру верхнего регистра, а AL - цифру нижнего регистра. Эта инструкция корректирует значение и помещает результат в AL, тогда как AH будет содержать ноль. Обобщенная версия этой инструкции делает возможной подгонку двух несжатых цифр с любым основанием. Правила для операнда такие же, как с инструкцией "aam".

2.1.5 Логические инструкции

"not" инвертирует биты в заданном операнде к форме обратного кода операнда. Не оказывает влияния на флаги. Правила для операнда таки же, как с инструкцией "inc".

"and", "or" и "xor" производят стандартные логические операции. Они изменяют флаги SF, ZF и PF. Правила для операнда таки же, как с инструкцией "add".

"bt", "bts", "btr" и "btc" оперируют с единичным битм, который может быть в памяти или регистре общего назначения. Расположения бита определяется как смещение от конца нижнего регистра операнда. Значение смещения берется из второго операнда, это может быть либо регистр общего назначения, либо байт. Эти инструкции первым делом присваивают флагу CF значение выбранного байта. "bt" больше ничего не делает, "bts" присваивает выбранному биту значение 1, "btr" сбрасывает его на 0, "btc" изменяет значение бита на его дополнение. Первый операнд может быть словом или двойным словом.

```
bt  ax,15        ; тестирует бит в регистре
bts word [bx],15 ; тестирует и ставит бит в памяти
btr ax,cx        ; тестирует и сбрасывает бит в регистре
btc word [bx],cx ; тестирует и дополняет бит в памяти
```

Инструкции "bsf" и "bsr" ищут в слове или двойном слове первый установленный бит и сохраняют индекс этого бита в операнд-адресат, которым должен быть регистр общего назначения. Сканируемая строка битов определяется операндом-источником, им может быть либо регистр общего назначения, либо память. Если срока нулевая (ни одного единичного бита), то устанавливается флаг ZF; иначе он очищается. Если не найдено ни одного установленного бита, значение операнда адресата не определено. "bsf" сканирует от нижнего регистра к верхнему (начиная с бита с индексом ноль). "bsr" сканирует от верхнего регистра к нижнему (начиная с бита с индексом 15 в слове или с индекса 31 в двойном слове).

```
bsf ax,bx        ; сканирование регистра по возрастанию
bsr ax,[si]      ; скнирование пмяти в обратном порядке
```

"shl" сдвигает операнд-адресат влево на определенное вторым операндом количество битов. Операндом-адресатом может быть регистр общего назначения или память размером в байт, слово или двойное слово. Вторым операндом может быть непосредственное значение или регистр CL. Процессор "задвигает" нули справа (с нижнего регистра), и биты "выходят" слева. Последний "вышедший" бит запоминается в CF. "sal" - это синоним "shl".

```
shl al,1          ; сдвиг регистра влево на один бит
shl byte [bx],1   ; сдвиг памяти влево на один бит
shl ax,cl         ; сдвиг регистра влево на количество из CL
shl word [bx],cl  ; сдвиг памяти влево на количество из CL
```

"shr" и "sar" сдвигают операнд-адресат вправо на число битов, определенное во втором операнде. Правила для операндо такие же, как с инструкцией "shl". "shr" "задвигает" нули с левой стороны операнда-адресата, биты "выходят" справа. Последний "вышедщий" бит запоминается в CF. "sar" сохраняет знак операнда, "забивая" слева нулями, если значение положительное, и единицами, если значение отрицательное.

"shld" сдвигает биты операнда-адресата влево за заданное в третьем операнде число битов, в то время как справа "задвигаются" биты верхних регистров операнда-источника. Операнд-источник не изменяется. Операндом-адресатом может быть регистр общего назначения или память размером в слово или двойное слово, операндом-источником должен быть регистр общего назначения, третьим операндом может быть непосредственное значение либо CL.

```
shld ax,bx,1      ; сдвиг регистра влево на один бит
shld [di],bx,1    ; сдвиг памяти влево на один бит
shld ax,bx,cl     ; сдвиг регистра влево на количество из CL
shld [di],bx,cl   ; сдвиг памяти влево на количество из CL
```

"shrd" сдвигает биты операнда-адресата вправо, в то время как слева "задвигаются" биты нижних регистров операнда-источника. Операнд-источник остается неизменным. Правила для операндов такие же, как с инструкцией "shld".

"rol" и "rcl" циклически сдвигают байт, слово или двойное слово влево на число битов, заданное во втором операнде. Для каждой заданной ротации старший бит, выходящий слева, появляется справа и становится самым младшим битом. "rcl" дополнительно помещает в CF каждый бит высшего регистра, выходящий слева, перед тем, как он возвратится в операнд как младший бит во время следующего цикла ротации. Правила для операндов такие же, как с инструкцией "shl".

""ror" и "rcr" циклически сдвигают байт, слово или двойное слово вправо на число битов, заданное во втором операнде. Для каждой заданной ротации младший бит, выходящий справа, появляется слева и становится самым старшим битом. "rcr" дополнительно помещает в CF каждый бит низшего регистра, выходящий слева, перед тем, как он возвратится в операнд как старший бит во время следующего цикла ротации. Правила для операндов такие же, как с инструкцией "shl".

""test" производит такое же действие, как инструкция "and", но не изменяет операнд-адресат, только обновляет флаги. Правила для операндов такие же, как с инструкцией "and".

""bswap" переворачивает порядок битов в 32-битном регистре общего назначения: биты от 0 до 7 меняются местами с битами от 24 до 31, а биты от 8 до 15 меняются с битами от 16 до 23. Эта инструкция предусмотрена для преобразования значений с прямым порядком байтов к формату с обратномым порядком и наоборот.

```
bswap edx         ; перестановка байтов в регистре
```

2.1.6 Инструкции передачи управления

"jmp" безоговорочно передает управление а заданное место. Адрес назначения может быть определен непосредственно в инструкции или косвенно через регистр или память, допустимый размер адреса зависит от того, какой переход, близкий или дальний, а также от того, какая инструкция, 16-битная или 32-битная. Операнд для близкого перехода должен быть размером "word" для 16-битной инструкции и размером "dword" для 32-битной инструкции. Операнд для дальнего перехода должен быть размером "dword" для 16-битной инструкции и размером "pword" для 32-битной инструкции. Прямая инструкция "jmp" содержит адрес назначения как часть инструкции, операндом, определяющим этот адрес, должно быть числовое выражение для близкого перехода и два числа, разделенные двоеточием, для дальнего перехода, первое определяет номер сегмента, второе - смещение внутри сегмента. Непрямая инструция "jmp" получает адрес назначения через регистр или переменную-указатель, операндом должен быть регистр общего назначения или память. Для более подробной информации смотрите также 1.2.5.

```
jmp 100h          ; прямой близкий переход
jmp 0FFFFh:0      ; прямой дальний переход
jmp ax            ; непрямой близкий переход
jmp pword [ebx]   ; непрямой дальний переход
```

"call" передает управление процедуре, сохраняя в стеке адрес инструкции, следующей за "call", для дальнейшего возвращения к ней инструкцией "ret". Правила для операндов такие же, что с инструкцией "jmp", но "call" не имеет короткого варианта в виде прямой инструкции, и поэтому не оптимизирована.

"ret", "retn" и "retf" прекращают выполнение процедуры передают управление назад программе, которая изначально вызвала эту процедуру, используя адрес, который был сохранен в стеке инструкцией "call". "ret" это эквивалент "retn", которая возвращает из процедуры, которая была вызвана с использованием близкого перехода, тогда как "retf" возвращает из процедуры, которая была вызвана с использованием дальнего перехода. Эти инструкции подразумевают размер адреса, соответствующий текущей установке кода, но этот размер может быть изменен на 16-битный с помощью мнемоников "retw", "retnw" и "retfw" и на 32-битный с помощью "retd", "retnd" и "retfd". Все эти инструкции могут опционально предписывать непосредственный операнд, если добавить константу к указателю стека, они эффективно удаляют любые аргументы, которые вызвающая программа положила в стек перед выполнением инструкции "call".

"iret" возвращает управление прерванной процедуре. Эта инструкция отличается от "ret" в том, что она также выводит из стека флаги в регистр флагов. Флаги сохраняются в стек механизмом прерывания. Инструкция подразумевает размер адреса, соответствующий текущей установке кода, но этот размер может быть изменен на 16-битный или на 32-битный с помощью мнемоников "iretw" или "iretd".

Условные инструкции перехода осуществляют или не осуществляют передачу управления в зависимости от состояния флагов CPU во время вызова этих инструкций. Мнемоники условных переходов могут быть получены добавлением условных мнемоников (смотри таблицу 2.1) к символу "j", например инструкция "jc" осуществляет передачу управления, если установлен флаг CF. Условные переходы могут быть только близкие и прямые и могут быть оптимизированы (смотри 1.2.5), операндом должно быть число, определяющее адрес назначения.

Таблица 2.1 Условия

Мнемоника	Тестируемое условие	Описание
o	OF = 1	переполнение
no	OF = 0	нет переполнения
c		перенос

Мнемоника	Тестируемое условие	Описание
b nae	CF = 1	меньше не больше и не равно
nc ae nb	CF = 0	нет переноса выше или равно не ниже
e z	ZF = 1	равно ноль
ne nz	ZF = 0	не равно не ноль
be na	CF or ZF = 1	ниже или равно не выше
a nbe	CF or ZF = 0	выше не ниже и не равно
s	SF = 1	знаковое
ns	SF = 0	беззнаковое
p pe	PF = 1	четное
np po	PF = 0	нечетное
l nge	SF xor OF = 1	меньше не больше и не равно
ge nl	SF xor OF = 0	больше или равно не меньше
le ng	(SF xor OF) or ZF = 1	меньше или равно не больше
g nle	(SF xor OF) or ZF = 0	больше не меньше и не равно

"loop" - это условные переходы, которые используют значение из CX (или ECX) для определения количества повторений цикла. Все инструкции "loop" автоматически уменьшают на единицу CX (или ECX) и завершают цикл, когда CX (или ECX) равно нулю. CX или ECX используется в зависимости от от текущей утановки битности кода - 16-ти или 32-битной, но вы можете принудительно использовать CX с помощью мнемоника "loopw", или ECX с помощью мнемоника "loopd".

"loope" и "loopz" это синонимы этой инструкции, которые работают так же, как стандартный "loop", но еще завершают работу, если установлен ZF. "loopew" и "loopzw" заставляют использовать регистр CX, а "looped" и "loopzd" заставляют использовать регистр ECX.

"loopne" и "loopnz" это тоже синонимы той же инструкции, которые работают так же, как стандартный "loop", но еще завершают работу, если ZF

сброшен на ноль. "loopnew" и "loopnzw" заставляют использовать регистр CX, а "loopned" и "loopnzd" заставляют использовать регистр ECX.

Каждая инструкция "loop" требует операндом число, определяющее адрес назначения, причем это может быть только близкий переход (в пределах 128 байт назад и 127 байт вперед от адреса инструкции, следующей за "loop").

"jcxz" переходит к метке, указанной в инструкции, если находит в CX ноль, "jecxz" делает то же, но проверяет регистр ECX. Правила для операндов такие же, как с инструкцией "loop".

"int" активирует стандартный сервис прерывания, который соответствует числу, указанному как операнд в мнемонике. Это число должно находиться в пределах от 1 до 255. Стандартный сервис прерывания заканчивается мнемоником "iret", которая возвращает управление инструкции, следующей за "int". "int3" кодирует короткое (в один байт) системное прерывание, которое вызывает прерывание 3. "into" вызывает прерывание 4, если установлен флаг OF.

"bound" проверяет, находится ли знаковое число, содержащееся в указанном регистре в нужных пределах. Если число в регистре меньше нижней границы или больше верхней, вызывается прерывание 5. Инструкция нуждается в двух операндах, первый - это тестируемый регистр, вторым должен быть адрес в памяти для двух знаковых чисел, указывающих границы. Операнды могут быть размером "word" или "dword".

```
bound ax,[bx]    ; тестирует слово на границы
bound eax,[esi]  ; тестирует двойное слово на границы
```

2.1.7 Инструкции ввода-вывода

"in" переводит байт, слово или двойное слово из порта ввода в AL, AX или EAX. Порты ввода-вывода могут быть адресованы либо напрямую, непосредственно с помощью байтового значения, либо непрямо через регистр DX. Операндом-адресатом должен быть регистр AL, AX или EAX. Операндом-источником должно быть число от 0 до 255 либо регистр DX.

```
in al,20h        ; ввод байта из порта 20
in ax,dx         ; ввод слова из порта, адресованного регистром DX
```

"out" переводит байт, слово или двойное слово из порта вывода в AL, AX или EAX. Программа может может определить номер порта, используя те же методы, что и в инструкции "in". Операндом-адресатом должен быть регистр AL, AX или EAX. Операндом-источником должно быть число от 0 до 255 либо регистр DX.

```
out 20h,ax       ; вывод байта в порт 20
out dx,al        ; вывод слова в порт, адресованный регистром DX
```

2.1.8 Строковые операции

Строковые операции работают с одним элементом строки. Этим элементом может быть байт, слово или двойное слово. Строковые элементы адресуются регистрами SI и DI (или ESI и EDI). После каждой сроковой операции SI и/или DI (или ESI и/или EDI) автоматически обновляются до указателя на следующий элемент строки. Если DF (флаг направления) равен нулю, регистры индекса увеличиваются, если DF равен единице, они

уменьшаются. Число, на которое они увеличиваются или уменьшаются равно 1, 2 или 4 в зависимости от размера элемента строки. Каждая инструкция строковой операции имеет короткую форму без операндов, использующую SI и/или DI если тип кода 16-битный, и ESI и/или EDI если тип кода 32-битный. SI и ESI по умолчанию адрес данных в сегменте, адресованном регистром DS, DI и EDI всегда адресует данные в сегменте, выбранном в ES. Короткая форма образуется добавлением к мнемонику строковой операции буквы, определяющей размер элемента строки, для байта это "b", для слова это "w", для двойного слова это "d". Полная форма инструкции требует операнды, указывающие размер оператора, и адресы памяти, которыми могут быть SI или ESI с любым сегментным префиксом, или DI или EDI всегда с сегментным префиксом ES.

"movs" переводит строковый элемент, на который указывает SI (или ESI) в место, на которое указывает DI (или EDI). Размер операнда может быть байтом, словом или двойным словом. Операндом-адресатом должна быть память, адресованная DI или EDI, операндом-источником должна быть память, адресованная SI или ESI с любым сегментным префиксом.

```
movs byte [di],[si]        ; переводит байт
movs word [es:di],[ss:si]  ; переводит слово
movsd                      ; переводит двойное слово
```

"cmps" вычитает строковый элемент-адресат из строкового элемента-источника и обновляет флаги AF, SF, PF, CF и OF, но не изменяет никакой из сравниваемых элементов. Если стрововые элементы эквивалентны, устанавливается ZF, иначе он очищается. Первым операндом этой инструкции должен быть строковый элемент, адресованный SI или ESI с любым сегментным префиксом, вторым операндом должен быть строковый элемент, адресованный DI или EDI.

```
cmpsb                      ; сравнение байтов
cmps word [ds:si],[es:di]  ; сравнение слов
cmps dword [fs:esi],[edi]  ; сравнение двойных слов
```

"scas" вычитает строковый элемент-адресат из AL, AX или EAX (в зависимости от размера этого элемента) и обновляет флаги AF, SF, ZF, PF, CF и OF. Если значения эквивалентны, устанавливается ZF, иначе он очищается. Операндом должен быть строковый элемент, адресованный DI или EDI.

```
scas byte [es:di]          ; сканирует байт
scasw                      ; сканирует слово
scas dword [es:edi]        ; сканирует двойное слово
```

"stos" помещает значение AL, AX или EAX в строковый элемент-адресат. Правила для операндов такие же, как с инструкцией "scas".

"lods" строковый элемент в AL, AX или EAX. Операндом должен быть строковый элемент, адресованный SI или ESI с любым префиксом сегмента.

```
lods byte [ds:si]          ; загружает байт
lods word [cs:si]          ; загружает слово
lodsd                      ; загружает двойное слово
```

"ins" переводит байт, слово или двойное слово порта ввода, адресованного регистром DX в строковый элемент-приемник. Операндом-адресатом должна быть память, адресованная DI или EDI, операндом-источником должен быть регистр DX.

```
insb                       ; ввод байта
```

```
ins word [es:di],dx         ; ввод слова
ins dword [edi],dx          ; ввод двойного слова
```

"outs" переводит строковый элемент-источник в порт вывода, адресованный регистром DX. Операндом-адресатом должн быть регистр DX, а операндом-источником должна быть память, адресованная SI или ESI с любым префиксом сегмента.

```
outs dx,byte [si]           ; вывод байта
outsw                       ; вывод слова
outs dx,dword [gs:esi]      ; вывод двойного слова
```

Префиксы повторения "rep", "repe"/"repz" и "repne"/"repnz" определяют повторяющуюся строковую операцию. Если инструкция строковой операции имеет префикс повторения, операция выполняется повторно, каждый раз используя другой элемент строки. Повторение прекратится, когда будет выполнено одно из условий, указанных префиксом. Все три префикса автоматически уменьшают регистр CX или ECX (в зависимости от того, какую адресацию использует инструкция строковой операции, 16-битную или 32-битную) после каждой операции и повторяют ассоциированную операцию, пока CX или ECX не станет равным нулю. "repe"/"repz" и "repne"/"repnz" используются только с инструкциями "scas" и "cmps" (описанными выше). Когда используются эти префиксы, повторение следующей инструкции зависит также от флага нуля (ZF), "repe" и "repz" прекращают выполнение, если ZF равен нулю, "repne" и "repnz" прекращают выполнение, если ZF равен единице.

```
rep  movsd        ; переводит несколько двойных слов
repe cmpsb        ; сравнивает байты, пока эквивалентны
```

2.1.9 Инструкции управления флагами

Инструкции управления флагами обеспечивают метод прямого изменения состояния битов во флаговом регистре. Все инструкции, описанные в этом разделе, не имеют операндов.

"stc" устанавливает CF (флаг переноса) в 1, "clc" обнуляет CF, "cmc" изменяет CF на его дополнение.

"std" устанавливает DF (флаг направления) в 1, "cld" обнуляет DF.

"sti" устанавливает IF (флаг разрешения прерываний) в 1 и таким образом разрешает прерывания, "cli" обнуляет IF и таким образом запрещает прерывания.

"lahf" копирует SF, ZF, AF, PF и CF в биты 7, 6, 4, 2 и 0 регистра AH. Содержание остальных битов неопределено. Флаги остаются неизменными.

"sahf" переводит биты 7, 6, 4, 2 и 0 из регистра AH в SF, ZF, AF, PF и CF.

"pushf" уменьшает ESP на два или на четыре и сохраняет нижнее слово или двойного слово флагового регистра в вершине стека. Размер сохраненной информации зависит от текущей настройки кода. Вариант "pushfw" сохранят слово, независимо от настройки кода, "pushfd" также независимо от настройки кода сохраняет двойное слово.

"popf" переводит определенные биты из слова или двойного слова в вершине стека и увеличивает ESP на два или на четывре, в зависимости от текущей настройки кода. Вариант "popfw" сохранят слово, независимо от настройки кода, "popfd" также независимо от настройки кода сохраняет двойное слово.

2.1.10 Условные операции

Инструкции, образванные с помощью добавления условного мнемоника (смотрите таблицу 2.1) к мнемонику "set" присваивают байту единицу, если условие истинно, и ноль, если если условие не выполняется. Операндом должен быть 8-битный регистр общего назначения либо байт в памяти.

```
setne al          ; единицу в al, если флаг нуля пустой
seto byte [bx]    ; единицу в байт, если есть переполнение
```

"calc" присваивает всем битам регистра AL единицу, если стоит флаг переноса, и нули в другом случае. У этой инстркции нет аргументов.

Иннструкции, образованные добавлением условного мнемоника к "cmov" переводят слово или двойное слово из регистра общего назначения или памяти в регистр общего назначения только если условие верно. Операндом-адресатом должен быть регистр общего назначения, операндом-источником - регистр общего назначения либо память.

```
cmove ax,bx       ; переводит, если установлен флаг нуля
cmovnc eax,[ebx]  ; переводит, если очищен флаг переноса
```

"cmpxchg" сравнивает значение в регистре AL, AX или EAX с операндом-адресатом. Если значения равны, операнд-источник загружается в операнд-адресат, иначе операнд-адресат загружается в регистр AL, AX или EAX. Операндом-адресатом должен быть регистр общего назначения или память, операндом-источником - регистр общего назначения.

```
cmpxchg dl,bl     ; сравнивает и меняет с регистром
cmpxchg [bx],dx   ; сравнивает и меняет с памятью
```

"cmpxchg8b" сравнивает с операндом 64-битное значение в регистрах EDX и EAX. Если значения раны, 64-битное значение в регистрах EDX и EAX сохраняется в операнде, иначе значение из операнда загружается в эти регистры. Операндом должно быть четверное слово в памяти.

```
cmpxchg8b [bx]    ; сравнивает и меняет 8 битов
```

2.1.11 Разные инструкции

"nop" занимает один бит, но ничего не значит, кроме как указатель инструкции. У неё нет операндов и она ничего не совершает.

"ud2" генерирует недопустимый опкод. Эта инструкция создана для тестирования прграммного обеспечения, чтобы недвусмысленно генерировать недопустимый опкод. У инструкции нет операндов.

"xlat" заменяет байт в регистре AL байтом, индексированным его значением в таблице перевода, адресованной BX или EBX. Операндом должен

быть байт памяти, адресованный регистром BX или EBX с любым сегментным префиксом. Эта инструкция также имеет короткую форму "xlatb", которая не использует операнды и использует адрес из BX или EBX (в зависимости от настройки кода) в сегменте, адресованном DS.

"lds" переводит переменный указатель из операнда-источника в DS и регистр-адресат. Операндом-источником должна быть память, а операндом-адресатом - регистр общего назначения. Регистр DS получает селектор сегмента указателя, а регистр-адресат получает его смещение. "les", "lfs" "lgs" и "lss" работают точно так же, как "lds", только вместо регистра DS используются соответственно ES, FS, GS или SS.

```
lds bx,[si]      ; загружает указатель в ds:bx
```

"lea" переводит смещение операнда-источника (вместо его значения) в операнд-адресат. Операндом-источником должна быть память, а операндом-адресатом должен быть регистр общего назначения.

```
lea dx,[bx+si+1] ; загружает исполнительный адрес в dx
```

"cpuid" возвращает идентификацию процессора и информацию о его свойствах в регистры EAX, EBX, ECX и EDX. Выдаваемая информация выбирается вводом нужного значения в регистр EAX перед тем, как выполнить иструкцию. У этой инструкции нет операндов.

"pause" задерживает выполнение следующей инструкции, реализуясь оперделенное количество времени. Эта инструкция может быть использована, чтобы улучшить выполнение циклов ожидания. У инструкции нет операндов.

"enter" создает стековый фрейм, который может быть использован для реализации свода правил блочных языков высокого уровня. Инструкция "leave" в конце процедуры дополняет "enter" в начале процедуры, чтобы упростить управление стеком и контролировать доступ к переменным для вложенных процедур. Инструкция "enter" имеет два параметра. Первый параметр определяет количество байт динамической памяти, которое должно быть отведено для введенной подпрограммы. Второй параметр соответствует лексическому уровню вложенности подпрограммы, может находится в области от 0 до 31. Указанный лексический уровень устанавливает, сколько наборов указателей стековых фреймов CPU копирует в новый стековый фрейм из предыдущего фрейма. Этот список указателей стековых фреймов иногда называется дисплеем. Первое слово (или двойное слово, если код 32-битный) дисплея - это указатель на последний стековый фрейм. Этот указатель делает возможным для инструкции "leave" совершить в обратном порядке действия предыдущей инструкции "enter", эффективно сбрасывая последний стековый фрейм. После того, как "enter" создает новый дисплей для процедуры, инструкция выделяет для неё место в динамической памяти, уменьшая ESP на количество байтов, определенных в первом параметре. Чтобы процедура могла адресовать свой дисплей, "enter" передает указатель BP (или EBP) в начало нового стекового фрейма. Если лексический уровень равен нулю, "enter" сохраняет BP (или EBP), копирует SP в BP (или ESP в EBP) и далее вычитает первый операнд из SP (или ESP). Для уровней вложенности больших нуля процессор сохраняет дополнительные указатели фреймов в стек перед подгонкой указателя стека.

```
enter 2048,0     ; ввод и выделение 2048 байтов в стеке
```

2.1.12 Системные инструкции

"lmsw" загружает операнд в слово машинного статуса (биты от 0 до 15 регистра CR0), тогда как "smsw" сохраняет слово машинного статуса в операнд-адресат. Операндом может быть 16-битный или 32-битный регистр общего назначения или слово в памяти.

```
lmsw ax          ; загружает машинный статус из регистра
```

```
smsw [bx]        ; сохраняет машинный статус в память
```

"lgdt" и "lidt" загружают значения из операнда в регистр таблицы глобальных дескрипторов или в регистр таблицы дескриптров прерываний соответственно. "sgdt" и "sidt" сохраняют содержимое регистра таблицы глобальных дескрипторов или регситра таблицы дескриптров прерываний в операнд-адресат. Операндом должны быть 6 байтов в памяти.

```
lgdt [ebx]       ; загружает таблицу глобальных дескрипторов
```

"lldt" загружает операнд в поле селектора сегмента регистра таблицы локальных дескрипторов, а "sldt" сохраняет селектор сегмента из регистра таблицы локальных дескрипторов в операнд. "ltr" загружает операнд в поле селектора сегмента регистра задачи, а "str" сохраняет селектор сегмента из регистра задачи в операнд. Правила для операндов такие же, как в инструкциях "lmsw" и "smsw".

"lar" загружает права доступа из сегментного дескриптора, указанного селектором в операнде-источнике, в операнд-адресат и ставит флаг ZF. Операндом-адресатом может быть 16-битный или 32-битный регистр общего назначения. Операндом-источником должен быть 16-битный регистр общего назначения или память.

```
lar ax,[bx]      ; загружает права доступа в слово
lar eax,dx       ; загружает права доступа в двойное слово
```

"lsl" загружает сегментный предел из сегментного дескриптора, указанного селектором в операнде-источнике, в операнд-адресат и ставит флаг ZF. Правила для операндов такие же, как в инструкции "lsl".

"verr" и "verw" проверяют, поддаётся ли чтению или записи на данном уровне привилегий сегмент кода или данных, заданный в операнде. Операндом должно быть слово, это может быть регистр общего назначения или память. Если сегмент доступен и читаем (для "verr") или изменяем, устанавливается флаг ZF, иначе он очищается. Правила для операндов такие же, как в инструкции "lldt".

"arpl" сравнивает поля RPL (уровень привилегий запрашивающего) двух селекторов сегментов. Первый операнд содержит один селектор сегмента, второй содержит другой. Если поле RPL операнда-адресата меньше, чем поле RPL операнда-источника, то устанавливается флаг ZF, и поле RPL операнда-адресата увеличивается до соответствия операнду-источнику. Иначе флаг ZF очищается и в операнде никаких изменений не производится. Операндом-адресатом должен быть регистр общего назначения или память длиной в слово, операндом-источником должен быть регистр общего назначения тоже длиной в слово.

```
arpl bx,ax       ; подгоняет RPL селектора в регистре
arpl [bx],ax     ; подгоняет RPL селектора в памяти
```

"clts" очищает флаг TS (переключение задач) в регистре CR0. У этой инструкции нет операндов.

Префикс "lock" заставляет процессор объявить сигнал "bus-lock" (или LOCK#) во время выполнения сопутсвующей инструкции. В многопроцессорной среде сигнал "bus-lock" гарантирует, что пока он объявлен, процессор эксклюзивно использует любую общую память. Префикс "lock" может быть присоединен только к следующим инструкциям и причем только к тем их формам, в которых операндом-адресатом является память: "add", "adc", "and", "btc", "btr", "bts", "cmpxchg", "cmpxchg8b", "dec", "inc", "neg", "not", "or", "sbb", "sub", "xor", "xadd" и "xchg". Если этот

префикс используется с одной из этих инструкций, но операндом-источником является память, может быть сгенерирован не определенный ошибочный опкод. Он может быть сгенерирован также, если префикс "lock" используется с инструкцией, не перечисленной выше. Инструкция "xchg" всегда объявляет сигнал "bus-lock", независимо от отсутствия или присутствия преикса "lock".

"hlt" прекращает выполнение инструкции и переводит процессор в состояние остановки. Запущенное прерывание, отладочное исключение, BINIT, INIT или RESET продолжат выполнение. У этой инструкции нет операндов.

"rdmsr" загружает содержимое 64-битного MSR (модельно-специфический регистр) по адресу, определенному в ECX, в EDX и EAX. "wrmsr" загружает содержимое регистров EDX и EAX в 64-битный MSR по адресу, определенному в ECX. "rdtsc" загружает текущее значение счетчика времени процессора из 64-битного MSR в регистры EDX и EAX. Процессор увеличивает значение счетчика времени MSR каждый цикл тактового генератора и сбрасывается на 0, когда процессор перезагружается. "rdpmc" загружает содержимое 40-битного счетчика событий производительности, заданного в ECX, в EDX и EAX. Эти инструкции не имеют операндов.

"wbinvd" совершает обратную запись модифицированных строк внутреннего кэша процессора в основную память и аннулирует (очищает) внутренние кэши. Далее инструкция запускает специальный цикл шины, который предназначает внешним кэшам также совершить обратную запись модифицированных данных и другой цикл шины, который указывает, что внешние кэши должны аннулироваться. Эта инструкция не имеет операндов.

"rsm" возвращает программное управление из из системного режима управления программе, которая была прервана, когда процессор получил прерывание SMM. Эта инструкция не имеет операндов.

"sysenter" выполняет быстрый вызов системный вызов процедуры уровня 0, "sysexit" выполняет быстрый возврат к коду пользователя уровня 3. Адреса, использованные этими инструкциями, сохраняются в MSR-ах. Эти инструкции не имеют операндов.

2.1.13 Инструкции FPU

Инструкции FPU (модуль операций с плавающей точкой) оперируют со значениями с плавающей точкой в трех форматах: одинарная точность (32-битная), двойная точность (64-битная) и расширенная точность (80-битная). Регистры FPU формируют стек и каждый из них вмещает значение с плавающей точкой расширенной точности. Если некоторые значения задвигаются в стек или вытаскиваются из вершины, регистры FPU сдвигаются, таким образом ST0 - это всегда значение в вершине стека FPU, ST1 - это первое значение ниже вершины и т. д.. Название ST0 имеет также синоним ST.

"fld" задвигает значение с плавающей точкой стек регистров FPU. Операндом может быть 32-битное, 64-битное или 80-битное расположение в памяти или регистр FPU, его значение загружается в вершину стека регистров FPU (регистр ST0) и автоматически конвертируется в формат расширенной точности.

```
fld dword [bx]   ; загружает значение одинарной точности из памяти
fld st2          ; загружает значение st2 в вершину стека
```

"fld1", "fldz", "fldl2t", "fldl2e", "fldpi", "fldlg2" и "fldln2" загружают часто ипользуемые константы в стек регистров FPU. Эти константы +1.0, +0.0, lb 10, lb e, pi, lg 2 и ln 2 соответственно. Эти инструкции не имеют операндов.

"fild" конвертирует знаковый целочисленный операнд-источник в расширенный формат с плавающей точкой и задвигает результат в стэк регистров FPU. Операндом-источником может быть 16-битное, 32-битное или 64-битное расположение в памяти.

```
fild qword [bx]  ; загружает 64-битное целое число из памяти
```

"fst" копирует значение из регистра ST0 в операнд-адресат, которым может быть 32-битное или 64-битное расположение в памяти или другой регистр FPU. "fstp" совершает ту же операцию, но далее выдвигает стек регистров, освобождая ST0. "fstp" поддерживает те же операнды, что и "fst" и ещё может сохранять 80-битное значение в память.

```
fst st3          ; копирует значение ST0 в регистр ST3
fstp tword [bx]  ; сохраняет значение в память и выдвигает стэк
```

"fist" конвертирует значение из ST0 в знаковое целое число и сохраняет результат в операнд-адресат. Операндом может быть 32-битное или 64-битное расположение в памяти. "fistp" совершает ту же операцию, но далее выдвигает стек регистров. Инструкция поддерживает те же операнды, что и "fist" и ещё может сохранять 64-битное целочисленное значение в память, таким образом, у неё правила для операндов такие же, как с инструкцией "fild".

"fbld" конвертирует сжатое целое число BCD в в расширенный формат с плавающей точкой и задвигает это значение в стек FPU. "fbstp" конвертирует значение из ST0 в 18-знаковое сжатое число BCD, сохраняет результат в операнд-адресат и выдвигает стэк регистров. Операндом должно быть 80-битное расположение в памяти.

"fadd" складывает операнд-источник и операнд-адресат и сохраняет сумму в адресате. Операндом-адресатом всегда должен быть регистр FPU, если источник - это расположение в памяти, то адресат это регистр ST0 и нужно указать только источник. Если обоими операндами являются регистры FPU, то одним из них должен быть ST0. Операндом в памяти может быть 32-битное или 64-битное значение.

```
fadd qword [bx]  ; прибавляет значение двойной точности к ST0
fadd st2,st0     ; прибавляет ST0 к ST2
```

"faddp" складывает операнд-источник и операнд-адресат, сохраняет сумму в адресате и далее выдвигает стэк регистров. Операндом-адресатом должен быть регистр FPU, а операндом-источником - ST0. Если операнды не указаны, то в качестве операнда-адресата используется ST1.

```
faddp            ; прибавляет st0 к st1 и выдвигает стэк
faddp st2,st0    ; прибавляет st0 к st2 и выдвигает стэк
```

"fiadd" конвертирует целочисленный операнд-источник в расширенный формат с плавающей точкой и прибавляет его операнде-адресату. Операндом должно быть 32-битное или 64-битное расположение в памяти.

```
fiadd word [bx]  ; прибавляет целочисленное слово к st0
```

"fsub", "fsubr", "fmul", "fdiv" и "fdivr" похожи на "fadd", имеют такие же правила для операндов и различаются только в совершаемых вычислениях. "fsub" вычитает операнд-источник из операнда-адресата, "fsubr" вычитает операнд-адресат из операнда-источника, "fmul" перемножает источник и адресат, "fdiv" делит операнд-адресат на операнд-источник, "fdivr" делит операнд-источник на операнд-адресат. "fsubp", "fsubrp", "fmulp" "fdivrp" и

"fdivrp" совершают те же операции и выдвигают стек регистров, правила для операнда такие же, как с инструкцией "faddp". "fisub", "fisubr", "fimul" "fidivr" и "fidivr" совершают те же операции после конвертации целочисленного операнда-источника в формат с плавающей точкой, они имеют такие же правила для операндов, как и инструкция "fiadd".

"fsqrt" вычисляет квадратный корень из значения в регистре ST0, "fsin" вычисляет синус этого значения, "fcos" вычисляет его косинус, "fchs" дополняет его знаковый бит, "fabs" очищает знак, чтобы создать абсолютное значение, "frndint" округляет до ближайшего целого значения, зависящего от текущего режима округления. "f2xm1" вычисляет экспоненциальное значение 2 в степени ST0 и вычитает из результата 1.0 (2^x-1), значение в ST0 должно лежать в пределах от -1.0 до +1.0. Все вышеперецисленные инструкции сохраняют значение в ST0 и не имеют операндов.

"fsincos" вычисляет синус и косинус значения в ST0, сохраняет синус в ST0 и задвигает косинус в вершину стека регистров FPU. "fptan" вычисляет тангенс значения в ST0, сохраняет результат в ST0 и задвигает 1.0 в вершину стека регистров FPU. "fpatan" вычисляет арктангенс значения в ST1, деленного на значение в ST0, сохраняет результат в ST1 и выдвигает стек регистров FPU. "fyl2x" вычисляет двоичный логарифм ST0, умножает его на ST1, сохраняет результат в ST1 и выдвигает стек регистров FPU; "fyl2xp1" совершает ту же операцию, перед вычислением логарифма помешает в ST0 значение 1.0. "fprem" вычисляет остаток, зависящий от деления значения из ST0 на значение из ST1, и сохраняет результат в ST0. "fprem1" совершает ту же операцию, что и "fprem", но вычисляет остаток способом, указанном в стандарте IEEE 754. "fscale" оставляет целую часть значения в ST1 и увеличивает экспоненту ST0 на полученное число. "fxtract" разделяет значение в ST0 на экспоненту и мантиссу, сохраняет экспоненту в ST0 и задвигает мантиссу в стек регистров. "fnop" не делает ничего. Эти инструкции не имеют операндов.

"fxch" меняет местами содержимое регистра ST0 и другого регистра FPU. Операндом должен служить регистр FPU, а если он не указан, меняются местами регистры ST0 и ST1.

"fcom" и "fcomp" сравнивают содержимое ST0 и операнда-источника и в зависимости от результатов ставят флаги статуса FPU. "fcomp" дополнительно после сравнения выдвигает стек регистров. Операндом может служить значение одинарной или двойной точности в памяти или регистр FPU. Если операнд не определен, в этой роли используется ST1.

```
fcom            ; сравнивает st0 с st1
fcomp st2       ; сравнивает st0 с st2 и выдвигает стек
```

"fcompp" сравнивает содержимое ST0 и ST1, устанавливает флаги вслове статуса FPU и дважды выдвигает стек регистров. У этой инструкции нет операндов.

"fucom", "fucomp" и "fucompp" совершают неупорядоченное сравнение двух регистров FPU. Правила для операндов такие же, как с инструкциями "fcom", "fcomp" и "fcompp", но операндом-источником должен быть регистр FPU.

"ficom" и "ficomp" сравнивают значение в ST0 с целочисленным операндом-источником и устанавливают флаги в слове статуса FPU в зависимости от результатов. "ficomp" дополнительно после сравнения выдвигает стек регистров. Перед операцией сравнения целочисленный операнд-источник конвертируется в расширенный формат с плавающей точкой. Операндом должно служить 16-битное или 32-битное расположение в памяти.

```
ficom word [bx]  ; сравнивает st0 с 16-битным целым числом
```

"fcomi", "fcomip", "fucomi", "fucomip" сравнивают ST0 с другим регистром FPU и ставят, в зависимости от результатов, флаги ZF, PF и CF. "fcomip"

и "fucomip" ещё выдвигают стек регистров после завершения сравнения. Инструкции, образованные добавлением условного мнемоника FPU (смотрите таблицу 2.2) к мнемонику "fcmov", переводят указанный регистр FPU в регистр ST0, если данное условие выполняется. Эти инструкции поддерживают два разных синтаксиса, первый с одним операндом, определяющим регистр FPU, второй с двумя операндами, где операнд-адресат - регистр ST0, а операнд-источник, идущий вторым, - нужный регистр FPU.

```
fcomi st2         ; сравнивает st0 с st2 и устанавливает флаги
fcmovb st0,st2    ; переводит st2 в st0 если меньше
```

Таблица 2.2 Условия FPU

Мнемоник	Тестируемое условие	Описание
b	CF = 1	меньше
e	ZF = 1	равно
be	CF or ZF = 1	меньше или равно
u	PF = 1	неномализованное
nb	CF = 0	не меньше
ne	ZF = 0	не равно
nbe	CF and ZF = 0	не меньше и не равно
nu	PF = 0	нормализованное

"ftst" сравнивает значение в ST0 с 0.0 и в зависимости от результатов устанавливает флаги в слове статуса FPU. "fxam" проверяет содержимое регистра ST0 и устанавливает флаги в слове статуса FPU, показывая класс значения в регистре. Эти инструкции не имеют операндов.

"fstsw" и "fnstsw" сохраняют текущее значение слова статуса FPU в указанном месте. Операндом-адресатом может быть либо 16 бит в памяти, либо регистр AX. "fstsw" перед сохранением слова проверяет на подвешенные немаскируемые численные исключения, "fnstsw" этого не делает.

"fstcw" и "fnstcw" сохраняют текущее значение управляющего слова FPU в указанном месте в памяти. "fstcw" перед сохранением слова проверяет на подвешенные немаскируемые численные исключения, "fnstcw" этого не делает. "fldcw" загружает операнд в управляющее слово FPU. Операндом должно быть 16-битное расположение в памяти.

"fstenv" и "fnstenv" сохраняют текущий контекст FPU в расположении в памяти, указанном в операнде-адресате, и далее маскируют все исключения операций с плавающей точкой. "fstenv" перед совершением операции проверяет на подвешенные немаскируемые численные исключения, "fnstenv" этого не делает. "fldenv" загружает полный контекст FPU из памяти в FPU. "fsave" и "fnsave" сохраняют текущий статус FPU (контекст и регистры стека) в указанном месте в памяти и затем ре-инициализируют FPU. "fsave" перед совершением операции проверяет на подвешенные немаскируемые численные исключения, "fnsave" этого не делает. "frstor" загружает статус FPU из указанного места в памяти. Все эти инструкции в качестве операнда требуют роасположение в памяти.

"finit" и "fninit" устанавливают контекст FPU в его значение по умолчанию. "finit" перед совершением операции проверяет на подвешенные немаскируемые численные исключения, "fninit" этого не делает. "fclex" and "fnclex" очищают флаги исключений FPU в слове статуса FPU."fclex" перед совершением операции проверяет на подвешенные немаскируемые численные исключения, "fnclex" этого не делает. "wait" и "fwait" - это синонимы

одной и той же инструкции, которая указывает процессор проверить наличие подвешенных немаскируемых численных исключений и разобраться с ними до продолжения работы. Эти инструкции не имеют операндов.

"ffree" помечает тэг, асооциированный с указанным регистром FPU, как пустой. Операндом должен служить регистр FPU.

"fincstp" и "fdecstp" вращают стек FPU на единицу, прибавляя или отнимая единицу от поля TOP слова статуса FPU. У этих инструкций нет операндов.

2.1.14 Инструкции MMX

Инструкции MMX оперируют со сжатыми целочисленными типами и используют регистры MMX, которыми являются нижние 64-битные части 80-битных регистров FPU. Поэтому инструкции MMX не могут использоваться в одно и то же время с инструкциями FPU. Они могут оперировать со сжатыми байтами (восемь 8-битных целых чисел), сжатыми словами (четыре 16-битных целых чисел) или сжатыми двойными словами (два 32-битных целых числа). Использование сжатых форматов позволяет совершать операции одновременно над многими данными.

"movq" копирует четверное слово из операнда-источника в операнд-адресат. По крайней мере одним из операндов должен являться регистр MMX, вторым может быть либо регистр MMX, либо 64-битное расположение в памяти.

```
movq mm0,mm1     ; копирует четверное слово из регистра в регистр
movq mm2,[ebx]   ; копирует четверное слово из памяти в регистр
```

"movd" копирует двойное слово из операнда-источника в операнд-адресат. Одним из операндов должен быть регистр MMX, вторым может быть регистр общего назначения либо 32-битное расположение в памяти. Используется только нижнее двойное слово регистра MMX.

Все основные операции MMX имеют два операнда, где операндом-адресатом должен быть регистр MMX, а операндом-источником может быть либо регистр MMX, либо 64-битное расположение в памяти. Операция совершается на соответствующих элементах данных источника и адресата и сохраняется элементах данных адресата. "paddb", "paddw" и "paddd" совершают сложение сжатых байтов, сжатых слов и сжатых двойных слов. "psubb", "psubw" и "psubd" совершают вычитание соответсвующих типов. "paddsb","paddsw", "psubsb" и "psubsw" совершают сложение или вычитание сжатых байтов или сжатых слов со знаковым насыщением. "paddusb", "paddusw", "psubusb", "psubusw" - это аналоги, но без знакового насыщения. "pmulhw" и "pmullw" совершают знаковое умножение сжатых слов и сохраняют верхние или нижние слова результатов в операнде-адресате. "pmaddwd" совершает умножение сжатых слов и складывает четыре промежуточных продукта в виде двойных слов в парах, чтобы получить результат в виде сжатых двойных слов. "pand", "por" и "pxor" совершают логические операции над четверными словами, "pandn" также производит логическое отрицание перед операцией "and". "pcmpeqb", "pcmpeqw" и "pcmpeqd" сравнивают на эквиваленитность сжатые байты, сжатые слова или сжатые двойные слова. Если пара элементов данных эквивалентна, то соответствующий элемент данных операнда-адресата покрывается единичными битам, иначе нулевыми. "pcmpgtb", "pcmpgtw" и "pcmpgtd" совершают похожую операцию, но они проверяют, больше ли элементы данных в операнде-адресате, чем соответствующие элементы данных в операнде-источнике. "packsswb" конвертирует сжатые знаковые слова в сжатые знаковые байты, "packssdw" конвертирует сжатые знаковые двойные слова в сжатые знаковые слова, используя насыщение, чтобы удовлетворить условиям переполнения. "packuswb" конвертирует сжатые знаковые слова в сжатые беззнаковые байты. Сконвертированные элементы данных из операнда-источника сохраняются в нижней части операнда-адресата, тогда как сконвертированные элементы данных операнда-адресата сохраняются в его верхней части. "punpckhbw", "punpckhwd" и "punpckhdq" чередуют элементы данных из верхних частей источника и адресата и сохраняют результат в

операнд-адресат. "punpcklbw", "punpcklwd" и "punpckldq" совершают те же операции, но с нижними частями операндов.

```
paddsb mm0,[esi] ; складывает сжатые байты со знаковым насыщением
pcmpeqw mm3,mm7  ; проверяет сжатые слова на эквивалентность
```

"psllw", "pslld" и "psllq" совершают логический сдвиг влево сжатых слов, сжатых двойных слов или одиночных четверных слов в операнде-адресате, на число битов, указанное в операнде-источнике. "psrlw", "psrld" и "psrlq" совершают логический сдвиг вправо сжатых слов, сжатых двойных слов или одиночных четверных слов. "psraw" и "psrad" совершают арифметический сдвиг сжатых слов или двойных слов. Операндом-адресатом должен быть регистр MMX, а операндом-источником может быть регистр MMX, 64-битное расположение в памяти или 8-битное непосредственное значение.

```
psllw mm2,mm4    ; сдвигает слова влево логически
psrad mm4,[ebx]  ; сдвигает двойные слова вправо арифметически
```

"emms" делает регистры FPU используемыми для инструкций FPU. Эта инструкция должна быть примененена перед использованием инструкций FPU, если в ход пускались инструкции MMX.

2.1.15 Инструкции SSE

Расширение SSE добавляет больше инструкций MMX, а также представляет операции со сжатыми значениями одинарной точности с плавающей точкой. 128-битный сжатый формат одинарной точности содержит четыре значения одинарной точности с плавающей точкой. 128-битные регистры SSE созданы для поддержки операций этого типа данных.

"movaps" и "movups" переводят операнд размером в двойное четверное слово, содержащий значения одинарной точности из операнда-источника в операнд-адресат. По крайней мере одним из операндов должен быть регистр SSE, вторым может быть либо тоже регистр SSE, либо 128-битное расположение в памяти. Операнды в памяти для "movaps" должны быть выровнены по 16-битной меже, для "movups" этого не требуется.

```
movups xmm0,[ebx]  ; переводит невыровненное двойное четверное слово
```

"movlps" переводит два сжатых значения одинарной точности из нижнего четверного слова регистра-источника в верхнее четверное слово регистра-адресата. "movhlps" переводит два сжатых значения одинарной точности из верхнего четверного слова регистра-источника в нижнее четверное слово регистра-адресата. Обоими операндами должны быть регистры SSE.

"movmskps" переводит знаковые биты всех значений одинарной точности в регистре SSE в нижние четыре бита регистра общего назначения. Операндом-источником должен быть регистр SSE, операндом-адресатом должен быть регистр общего назначения.

"movss" переводит значение одинарной точности между источником и адресатом (переводится только нижнее двойное слово). По крайней мере одним из операндов должен быть регистр SSE, вторым может быть либо тоже регистр SSE, либо 32-битное расположение в памяти.

```
movss [edi],xmm3   ; переводит нижнее двойное слово из xmm3 в память
```

Каждая арифметическая операция SSE имеет два варианта. Если мнемоник заканчивается на "ps", операндом-источником может быть 128-битное расположение в памяти или регистр SSE, операндом-адресатом должен быть регистр SSE, и операция производится над четыремя сжатыми значениями одинарной точности, для каждой пары соответствующих элементов данных отдельно, и результат сохраняется в регистре-адресате. Если мнемоник заканчивается на "ss", то операндом-источником может быть 32-битное расположение в памяти или регистр SSE, операндом-адресатом должен быть регистр SSE, и операция производится над одним значением одинарной точности, используются только нижние двойные слова регистров SSE, и результат сохраняется в нижнем двойном слове регистра-адресата. "addps" и "addss" складывают значения, "subps" и "subss" вычитают источник из адресата, "mulps" и "mulss" перемножают значения, "divps" и "divss" делят адресат на источник, "rcpps" и "rcpss" вычисляют аппроксимированную обратную величину источника, "sqrtps" и "sqrtss" вычисляют квадратный корень источника, "rsqrtps" и "rsqrtss" вычисляют аппроксимированную обратную величину квадратного корня источника, "maxps" и "maxss" сравнивают источник и адресат и возвращают большее значение, "minps" и "minss" сравнивают источник и адресат и возвращают меньшее значение.

```
mulss xmm0,[ebx]    ; перемножает значения одинарной точности
addps xmm3,xmm7     ; складывает сжатые значения одинарной точности
```

"andps", "andnps", "orps" и "xorps" производят логические операции над сжатыми значениями одинарной точности. Операндом-источником может быть 128-битное расположение в памяти или регистр SSE, операндом-адресатом должен быть регистр SSE.

"cmpps" сравнивает сжатые значения одинарной точности и возвращают маскируемый результат в операнд-адресат, которым должен быть регистр SSE. Операндом-источником может быть либо регистр SSE, либо 128-битное расположение в памяти, третьим операндом должно быть непосредственное значение, выбирающее код одного из восьми условий сравнения (таблица 2.3). "cmpss" совершает ту же над значениями одинарной точности, изменяется только нижнее двойное слово регистра-адресата, таким образом операндом-источником должен быть либо регистр SSE, либо 32-битное расположение в памяти. Эти две инструкции имеют также варианты с двумя операндами и условие, закодированным в мнемоник. Их мнемоники образуются путем добавления мнемоников из таблицы 2.3 к "cmp" и после добавления к ним в конце "ps" илои "ss".

```
cmpps xmm2,xmm4,0  ; сравнивает сжатые значения одинарной точности
cmpltss xmm0,[ebx] ; сравнивает значения одинарной точности
```

Table 2.3 Условия SSE

Код	Мнемоник	Описание
0	eq	равно
1	lt	меньше
2	le	меньше или равно
3	unord	ненормализованное
4	neq	не равно
5	nlt	не меньше
6	nle	не меньше и не равно
7	ord	нормализованное

"comiss" и "ucomiss" сравнивают значения одинарной точности и ставят в зависимости от результата флаги ZF, PF и CF. Операндом-адресатом

должен быть регистр SSE, операндом-источником может быть 32-битное расположение в памяти или регистр SSE.

"shufps" переводит некоторые два из четырех значений одинарной точности из операнда-адресата в нижнее четверное слово операнда-адресата и некоторые два из четырех значений одинарной точности из операнда-источника в верхнее четверное слово операнда-адресата. Операндом-адресатом должен быть регистр SSE, операндом-источником может быть 128-битное расположение в памяти или регистр SSE, а третьим операндом должно быть 8-битное непосредственное значение, какие конкретно значения будут задействованы. Биты 0 и 1 указывают значение из адресата, которое должно быть в нижнем двойном слове результата, биты 2 и 3 указывают значение из адресата, которое должно быть во втором двойном слове результата, биты 4 и 5 указывают значение из источника, которое должно быть в третьем двойном слове результата, биты 6 и 7 указывают значение из источника, которое должно быть в нижнем верхнем слове результата.

```
shufps xmm0,xmm0,10010011b ; перемешивает двойные слова
```

"unpckhps" совершает перемежающуюся распаковку значений из верхних частей источника и адресата и сохраняет результат в адресат, которым должен быть регистр SSE. Операндом-источником может быть 128-битное расположение в памяти или регистр SSE. "unpcklps" совершает перемежающуюся распаковку значений из нижних частей источника и адресата и сохраняет результат в адресат, правила для операндов такие же.

"cvtpi2ps" конвертирует два сжатых целых числа размером в двойное слово в два сжатых значения с плавающей точкой одинарной точности и сохраняет результат в нижнем четверном слове адресата, которым должен быть регистр SSE. Операндом-источником может быть 64-битное расположение в памяти или регистр MMX.

```
cvtpi2ps xmm0,mm0  ; конвертирует целые числа в значения одинарной точности
```

"cvtsi2ss" конвертирует целое число размером в двойное слово в сжатое значение с плавающей точкой одинарной точности и сохраняет результат в нижнем двойном слове адресата, которым должен быть регистр SSE. Операндом-источником может быть 32-битное расположение в памяти или 32-битный регистр общего назначения.

```
cvtsi2ss xmm0,eax  ; конвертирует целое число в значение одинарной точности
```

"cvtps2pi" конвертирует два сжатых значения с плавающей точкой одинарной точности в два сжатых целых числа размером в двойное слово и сохраняет результат в адресате, которым должен быть регистр MMX. Операндом-источником может быть 64-битное расположение в памяти либо регистр SSE, в котором будет использовано только нижнее четверное слово. "cvttps2pi" совершает похожую операцию, но для округления здесь используется отбрасывание дробной части, правила для операндов у этой инструкции такие же.

```
cvtps2pi mm0,xmm0  ; конвертирует значения одинарной точности в целые числа
```

"cvtss2si" конвертирует сжатое значение с плавающей точкой одинарной точности в сжатое целое число размером в двойное слово и сохраняет результат в адресате, которым должен быть 32-битный регистр общего назначения. Операндом-источником может быть 32-битное расположение в памяти либо регистр SSE, в котором будет использовано только нижнее двойное слово. "cvttss2si" совершает похожую операцию, но для округления здесь используется отбрасывание дробной части, правила для операндов у этой инструкции такие же.

```
cvtss2si eax,xmm0  ; конвертирует значение одинарной точности в целое число
```

"pextrw" копирует слово, указанное третьим операндом, из источника в адресат. Операндом-источником должен быть регистр MMX, операндом-адресатом должен быть 32-битный регистр общего назначения (но используется только нижнее его слово), третьим операндом должно быть 8-битное непосредственное значение.

```
pextrw eax,mm0,1   ; извлекает слово в eax
```

"pinsrw" вставляет слово из источника в место в адресате, указанное третьим операндом, которым должно быть 8-битное непосредственное значение. Операндом-адресатом должен быть регистр MMX, операндом-источником должен быть 32-битный регистр общего назначения (но используется только нижнее его слово).

```
pinsrw mm1,ebx,2   ; вставляет слово из ebx
```

"pavgb" and "pavgw" вычисляют среднее сжатых байтов или слов. "pmaxub" возвращает максимум сжатых беззнаковых байтов, "pminub" возвращает минимум сжатых беззнаковых байтов, "pmaxsw" возвращает максимум сжатых знаковых слов, "pminsw" возвращает минимум сжатых знаковых слов. "pmulhuw" совершает беззнаковое умножение сжатых слов и сохраняет верхние слова результатов в операнд-адресат. "psadbw" вычисляет абсолютные разности сжатых беззнаковых байтов, суммирует эти разности и сохраняет результат в нижнее слово операнда-адресата. Все эти инструкции следуют тем же правилам для операндов, что и основные операции MMX, описанные в предыдущем параграфе.

"pmovmskb" создает маску из знаковых битов всех байтов в источнике и сохраняет результат в нижнем байте адресата. Операндом-источником должен быть регистр MMX, операндом-адресатом должен быть 32-битный регистр общего назначения.

"pshufw" копирует слова, указанные третьим операндом, из источника в адресат. Операндом-адресатом долже быть регистр MMX, операндом-источником может быть 64-битное расположение в памяти или регистр MMX, третьим операндом должно быть 8-битное непосредственное значение, выбирающее, какие значения будут помещены в адресат, таким же образом как третий операнд в инструкции "shufps".

"movntq" переводит четверное слово из операнда-источника в память, используя "не-временное малое количество" (non-temporal hint), чтобы минимизировать загрязнение кэша. Операндом-источником должен быть регистр MMX, операндом-адресатом должно быть 64-битное расположение в памяти. "movntps" сохраняет сжатые значения одинарной точности из регистра SSE в память, используя "не-временное малое количество". Операндом-источником должен быть регистр SSE, операндом-адресатом должно быть 128-битное расположение в памяти. "maskmovq" сохраняет выбранные байты из первого операнда в 64-битное расположение в памяти, используя "не-временное малое количество". Обоими операндами должны служить регистры MMX, второй операнд указывает, какие байты из первого операнда должны быть записаны в память. Расположение в памяти указывается регистром DI (или EDI) в сегменте, определенном в DS.

"prefetcht0", "prefetcht1", "prefetcht2" and "prefetchnta" помещает строку данных из памяти, которая содержит байт, указанный в операнде, в определенное место в иерархии кеша. Операндом должно быть 8-битное расположение в памяти.

"sfence" переводит в последовательный режим все предыдущие команды, совершающие запись в память. У этой инструкции нет операндов.

"ldmxcsr" загружает 32-битный операнд в памяти в регистр MXCSR. "stmxsr" сохраняет содержимое MXCSR в 32-битный операнд в памяти.

"fxsave" сохраняет текущий статус FPU, регистр MXCSR и все регистры FPU и SSE в 512-байтное расположение в памяти, указанное в операнде-адресате. "fxstor" перезагружает данные, ранее сохраненные инструкцией "fxsave" из 512-байтного расположения в памяти. Операнд для обеих этих инструкций должен быть выровнен по 16 байтам, нужно объявить операнд неопределенного размера.

2.2 Директивы управления

Этот параграф описывает директивы, которые управляют процессом ассемблирования. Эти директивы выполняются во время ассемблирования и могут делать так, чтобы некоторые блоки инструкций ассемблировались по-разному или не ассемблировались вовсе.

2.2.1 Условное ассемблирование

С помощью директивы "if" можно ассемблировать или не ассемблировать блок инструкций в зависимости от выполнения условия. За ней должно следовать логическое выражение, определяющее условие. Инструкции на следующих строках ассемблируются, только если это условие выполняется, иначе они пропускаются. Опциональная директива "elseif" со следущим за ней логическим выражением, определяющим дополнительное условие, начинает следующий блок инструкций, который ассемблируется, если предыдущие условия не выполняются, а данное дополнительное условие выполняется. Опциональная директива "else" начинает блок инструкций, которые ассемблируются, если не выполняется ни одно из условий. "end if" заканчивает последний блок инструкций.

Вы должны помнить, что директива "if" обрабатывается на стадии ассемблирования и поэтому не влияет на директивы препроцессора, такие как определения символьных констант и макроинструкции - когда ассемблер распознает директиву "if", весь препроцессинг уже закончен.

Логическое выражение состоит из логических значений и логических операторов. Логические операторы выглядят так: "~" для логического отрицания, "&" для логического И, "|" для логического ИЛИ. Отрицание имеет высший приоритет. Логическое значение может быть числовым выражением, оно будет считаться ложным в случае равенства нулю, иначе оно будет истинным. Для создания логического значения можно сравнить два числовых выражения, используя один из следующих операторов: "=" (равно), "<" (меньше), ">" (больше), "<=" (меньше или равно), ">=" (больше или равно), "<>" (не равно).

"used" со следующим за ним символом имени, это логическое значение, которое проверяет, использовался ли где-нибудь данный символ (он возвращаетправильный результат даже если символ используется только после этой проверки). За оператором "defined" может следовать любое выражение, обычно это только одно символьное имя; этот оператор проверяет, содержит ли данное выражение исключительно символы, определенные в коде, и доступные из текущей позиции.

Следующий простой пример использует константу "count" которая должна быть определена где-то в коде:

```
if count>0
    mov cx,count
    rep movsb
end if
```

Эти две инструкции будут ассемблированы только если константа "count" больше нуля. Следующий пример показывает более комплексную

условную структуру:

```
if count & ~ count mod 4
    mov cx,count/4
    rep movsd
else if count>4
    mov cx,count/4
    rep movsd
    mov cx,count mod 4
    rep movsb
else
    mov cx,count
    rep movsb
end if
```

Первый блок инструкций ассеблируется, если константа "count" не равна нулю и кратна четырем, если это условие не выполняется, оценивается второе логическое условие, следующее за "else if", и если оно верно, ассемблируется второй блок инструкций, иначе ассемблируется последний блок, который следует за строкой, содержащей только "else".

Также есть операторы, которые позволяют сравнивать значения, которые представляют собой последовательности символов. "eq" проверяет такие значения на тождественность. Оператор "in" проверяет, принадлежит ли данное значение к списку значений, следующему за оператором. Список должен быть заключен между символами "<" и ">", а его члены должны быть разделены запятыми. Символы считаются одинаковыми, если они имеют одно и то же значение для ассемблера - например, "pword" и "fword" для ассемблера одинаковы поэтому не различаются вышеуказанными операторами. Так же "16 eq 10h" является истиной, однако "16 eq 10+4" нет.

Оператор "eqtype" имеют ли сравниваемые значения одинаковую структуру, и принадлежат ли структурные элементы одному типу. Различаемые типы включают в себя числовые выражения, строки, заключенные в кавычки, значения с плавающей точкой, адресные выражения (выражения в квадратных скобках или предваренные оператором "ptr"), мнемоники инструкций, регистры, операторы размера, операторы перехода и операторы типа кода. И каждый из специальных символов, действующих как разделители, такой как запятая или двоеточие, это отдельный тип сам по себе. Например, два значения, каждое из которых состоит из имени регистра и числового выражения, разделенных запятой, будут распознаны как один тип, независимо от вида регистра и сложности числового выражения; за

исключением строк, заключенных в кавычки и значений с плавающей точкой, которые относятся к специальным видом числовых выражений и распознаются как разные типы. Поэтому условие "eax,16 eqtype fs,3+7" является истиной, но "eax,16 eqtype eax,1.6" - ложь.

2.2.2 Повторение блоков инструкций

"times" повторяет одну инструкцию указанное количесво раз. За ней должно следовать числовое выражение, определяющее количество повторений, и инструкция, которую нужно повторять (опционально для того, чтобы отделить число и инструкцию, можно использовать двоеточие). Специальный символ "%", использующийся внутри инструкции, эквивалентен номеру текущего повтора. Например, "times 5 db %" определит пять байтов со значениями 1, 2, 3, 4, 5. Поддерживается также рекурсивное использование директивы "times", например, "times 3 times % db %" определит шесть байтов со значениями 1, 1, 2, 1, 2, 3.

"repeat" повторяет целый блок инструкций. За ней должно следовать числовое выражение, определяющее количество повторений. Инструкции для повторения предполагаются на следующих строках, а заканчиваться блок должен директивой "end repeat", например:

```
repeat 8
    mov byte [bx],%
    inc bx
end repeat
```

Сгенерированный код сохраняет байты со значениями от одного до восьми в памяти, адресованной регистром BX.

Количество повторений может быть равным нулю, и в таком случае инструкции не будут ассемболироваться вовсе.

"break" позволяет остановить повторение раньше и продолжить ассемблирование с первой строки после "end repeat". В сочетании с директивой "if" она позволяет остановить повторение при выполнении некоторого особого условия, например:

```
s = x/2
repeat 100
    if x/s = s
        break
    end if
    s = (s+x/s)/2
end repeat
```

"while" повторяет блок инструкций, пока выполняется следующее за ней условие, определенное логическим выражением. Блок инструкций для повторения должен заканчиваться директивой "end while". Перед каждым повторением логическое выражение вычисляется и если его значение ложь, ассемблирование продолжается, начиная с первой строки после "end while". Также в этом случае символ "%" содержит номер текущего повторения. Директива "break" может быть использована для остановки этого типа цикла так же , как с директивой "repeat". Предыдущий пример может быть переписан с использованием "while" вместо "repeat" таким образом:

```
s = x/2
while x/s <> s
    s = (s+x/s)/2
    if % = 100
        break
    end if
end while
```

Блоки, определенные с использованием "if", "repeat" и "while" могут быть вложены в любом порядке, однако и закрыты в обратном. Директива "break" всегда останавливает обработку бока, который был начат последним либо директивой "repeat", либо "while".

2.2.3 Адресные пространства

"org" устанавливает адрес, по которому следующий за ней код должен появиться в памяти. За ней должно следовать числовое выражение,

указывающее адрес. Эта директива начинает новое адресное пространство, следующий код сам по себе никуда не двигается, но все метки, определенные в нем и значение символа "$" изменяются как если бы он был бы помещен по этому адресу. Тем не менее обязанность поместить во время выполнения код по правильному адресу лежит на программисте.

"load" позволяет определить константу двоичным значением, загруженным из уже сассемблированного кода. За директивой должно следовать имя константы, затем опционально оператор размера, затем оператор "from" и числовое выражение, определяющее валидный адрес в текущем адресном пространстве. Оператор размера здесь имеет необычное значение - он определяет, сколько байтов (до 8) должно быть загружено из двоичного значения константы. Если оператор размера не определен, загружается один байт (таким образом значение оказывается в пределах от 0 до 255). Загруженные данные не могут превосходить текущее смещение.

"store" может модифицировать уже сгенерированный код заменой некоторых ранее сгенерированных байтов значением, задаваемым следующим за инструкцией числовым выражением. Перед этим выражением может идти оператор размера, определяющий, насколько длинное значение оно задает, то есть сколько будет сохранено байт. Если оператор размера не задан, подразумевается длина в один байт. Далее должен следовать оператор "at" и числовое выражение, указывающее валидный адрес в текущем адресном пространстве кода. По этому адресу будет сохранено задаваемое значение. Это директива для продвинутого применения и её следует использовать осторожно.

Обе директивы "load" и "store" ограничены оперированием только в пределах текущего адресного пространства. Символ "$$" всегда равен базовому адресу в текущем адресном пространстве, а символ "$" - это адрес текущей позиции в нём, то есть эти два значения определяют границы действия директив "load" и "store".

Сочетая директивы "load" и "store" можно делать вещи, такие как шифрование некоторого из уже сгенерированного кода. Например, для шифрования всего кода, сгенерированного в текущем адресном пространстве вы можете использовать такой блок директив:

```
repeat $-$$
    load a byte from $$+%-1
    store byte a xor c at $$+%-1
end repeat
```

Каждый байт кода будет проксорен со значением, определенным константой "c".

"virtual" определяет виртуальные данные по указанному адресу. Эти данные не будут включены в файл вывода, но но метки, определенные здесь, могут использоваться в других частях кода. За этой директивой может следовать оператор "at" и числовое выражение, определяющее адрес виртуальных данных, иначе будет использован текущий адрес, что равносильно директиве "virtual at $". Инструкции определяемых данных должны быть расположены на следующих строках и заканчиваться директивой "end virtual". Блок виртуальных инструкций сам по себе независимое адресное пространство, и после того, как оно заканчивается, восстанавливается контекст предыдущего адресного пространства.

Директива "virtual" может быть использована для создания объединения нескольких переменных, например:

```
GDTR dp ?
virtual at GDTR
    GDT_limit dw ?
```

```
    GDT_address dd ?
end virtual
```

Здесь определяются две части 48-битной переменной по адресу "GDTR".

Директива также может быть использована для определения меток некоторых структур, адресованных регистром, например:

```
virtual at bx
    LDT_limit dw ?
    LDT_address dd ?
end virtual
```

С таким определением инструкция "mov ax,[LDT_limit]" будет сассемблирована в "mov ax,[bx]".

Также может быть полезно объявление инструкций и значений данных внутри виртуально блока, так как директиву "load" можно использовать для загрузки в константы значений из виртуально сгенерированного кода. Эта директива должна быть использована после загружаемого кода, но до окончания виртуального блока, так как она может загружать значения только из того же адресного пространства. Например:

```
virtual at 0
    xor eax,eax
    and edx,eax
    load zeroq dword from 0
end virtual
```

Этот кусок кода определяет константу "zeroq", которая будет содержать четыре байта машинного кода инструкций, указанных внутри виртуального блока. Этот метод также может быть использован для загрузки некоторых бинарных значений из внешнего файла. Например этот код:

```
virtual at 0
    file 'a.txt':10h,1
    load char from 0
end virtual
```

Загружает один байт со смещением 10h из файла "a.txt" в константу "char".

Все директивы "section", описанные в 2.4, также начинают новое адресное пространство.

2.2.4 Другие директивы

"align" выравнивает код или данные по указанной границе. За ней должно следовать числовое выражение, определяющее количество байтов, на кратность которому должен быть выровнен текущий адрес. Значение границы должно быть степенью двойки.

Директива "align" заполняет байты, которые должны быть пропущены, чтобы совершить выравнивание, инструкциями "nop", и в это же время маркирует эту область как неинициализированные данные, то есть если её поместить среди других неинициализированных данных, это не займет

места в файле вывода, выравнивание байтов происходит таким же образом. Если вам нужно заполнить область выравнивания какими-то другими значениями, вы можете сочетать "align" и "virtual", чтобы получить требуемый размер выравнивания и далее создать выравнивание самостоятельно, например:

```
virtual
    align 16
    a = $ - $$
end virtual
db a dup 0
```

Константа "a" определяется как разница между адресом после выравнивания и адресом блока "virtual" (смотрите предыдущий параграф), то есть она равна размеру требуемого пространства выравнивания.

"display" во время ассемблирования показывает сообщение. За ней должны следовать строка в кавычках или значения байтов, разделенные запятыми. Директива может быть использована для показа значений некоторых констант, например:

```
bits = 16
display 'Current offset is 0x'
repeat bits/4
    d = '0' + $ shr (bits-%*4) and 0Fh
    if d > '9'
        d = d + 'A'-'9'-1
    end if
    display d
end repeat
display 13,10
```

Этот блок директив рассчитывает четыре цифры 16-битного значения и конвертирует их в знаки для показа. Помните что это не будет работать, если адреса в текущем адресном пространстве перемещаемы (как это может быть с объектным форматом вывода и форматом PE), так как таким образом могут быть использованы только абсолютные значения. Абсолютное значение может быть получено вычислением относительного адреса, например "$-$$" или "rva $" в случае формата PE.

2.2.5 Множественные проходы

Так как ассемблер позволяет ссылаться на некоторые метки и константы перед тем, как они фактически определены, приходится прогнозировать значения этих меток и если есть даже подозрение, что прогноз окажется неверным хотя бы один раз, далается еще один проход, ассемблирующий весь код, и в это время делается лучший прогноз, базирующееся на значениях меток, полученных в предыдущий проход.

Изменение значений меток может быть причиной того, что некоторые иструкции перекодируются с другими длинами и это снова повлечет изменение меток. И так как метки и константы ещё могут использоваться внутри выражений, которые влияют на поведение директив управления, весь блок инсрукций в новый проход может ассемблироваться абсолютно по-другому. Поэтому ассемблер делает проходы снова и снова, каждый раз пытаясь создать лучшие прогнозы, чтобы приблизиться к финальному решению, когда все значения спрогнозированы правильно. Для прогнозов

используются разные методы, которые выбираются с тем, чтобы найти с как можно меньшим количеством проходов решение наименьшей возможной длины для большинства программ.

О некоторых ошибках, таких как непопадание значений в заданные границе, не сигнализируется во время этих промежуточных проходов, пока может случиться такое, что если какие-то значения будут спрогнозированы лучше, эти ошибки исчезнут сами собой. Однако, если ассемблер встречает какую-то недопустимую синтаксическую конструкцию или неизвестную инструкцию, он всегда останавливается немедленно. Такую же ошибку вызывает определение метки более, чем один ра, так как это делает прогнозы необоснованными.

Если в коде встречается директива "display", фактически отображаются только сообщения, созданные в последний совершённый проход. В случае, если ассемблер остановился из-за ошибки, эти сообщения могут отражать спрогнозированные значения, которые еще не разрешены правильно.

Разрешение иногда может не создаться и в таких случаях ассемблер никогда не сумеет создать правилные прогнозы - по этой причине существует предел количесва походов, и когда ассемблер исчерпает этот лимит, он остановится отобразит сообщение, что невозможно сгенерировать корректный вывод. Рассмотрим следующий пример:

```
if ~ defined alpha
    alpha:
end if
```

Если оператор "defined" выдает значение истина, если выражение, следующее за ним, в этом месте может быть вычислено, что в даном случае означает, что метка "alpha" где-то определена. Но блок выше определяет эту метку только, если значение, данное оператором "defined" ложь, что ведет к противоречию и делает невозможным разрешить такой код. Если, обрабатывая директиву "if" ассемблер должен прогнозировать, будет ли где-нибудь определена метка "alpha" (этого делать не приходится только если метка уже определена раньше), то какой бы ни был прогноз, всегда присходит противоположное. Поэтому ассемблирование остановится,если только метка "alpha" не определена где-то в коде перед вышеукзанным блоком - в этом случае, как уже было отмечено прогнозирование не требуется и блок просто будет пропущен.

Предыдущий пример может быть создан как попытка определить ментку, только если этого все ещё не сделано. Эти строк неправильны, поскольку операто "defined" проверяет определена ли метка где-либо вообще, и это включает определение внутри этого условного блока. Однако есть способ бойти эту проблему:

```
if ~ defined alpha | defined @f
    alpha:
    @@:
end if
```

"@f" это всегда та же метка, что ближайший следующий за ним символ "@@", поэтому предыдущий пример значит то же, как если бы вместо анонимной метки было определено любое уникальное имя. Если метка "alpha" ещё не определена, ассемблер спрогнозирует значение "defined alpha" как ложь, это будет однако значить, что будут определены обе метки. Но на следующем проходе ассемблер спрогнозирует, что определены обе метки, что затавит определить их вновь - так прогноз будет совпадать с результатом и процесс ассемблиования придет к правильному решению. Анонимная метка выступает здесь как маркер того, что метка "alpha" определена в этом месте.

Из этого примера вы можете заключить, что прогноз для оператора "defined" очень прямолинейный - метка прогнозируется как определенная только если она была определена в предыдущий проход (а если она была определена в текущий проход, прогноз не требуется). То же самое относится к оператору "used". Однако прогнозы для значений меток не так просты и вам никогда не следует полагать, что ассембер работает таким способом.

2.3 Директивы препроцессора

Все директивы препроцессора выполняются перед основным ассемблированием, и таким образом директивы управления на них никак не влияют. В это время также удаляются все комментарии.

2.3.1 Включение файлов-исходников

"include" включает указанный файл-исходник туда, где эта директива испльзуется. За ней должно следовать в кавычках имяфайла, который должен быть включен, например:

```
include 'macros.inc'
```

Весь включенный файл обрабатывается препроцессором перед обработкой строк, следущих за содержащей директиву "include". Нет предела для количества включаемых файлов, пока они умещаются в память.

Путь, заключеннй в скобки, может содержать окружающие переменные, заключенные в знаки "%", они будут заменены на их значения внутри пути. Знаки "\" и "/" трактуются как разделители пути. Если не указан абсолютный путь, сначала файл ищется в директории, содержащей файл, в который он включается, и, далее, если его там нет, в директории, содержащей главный файл-исходник (указаннй в командной строке). Эти правила так же относятся к путям, которые указываются в директиве "file".

2.3.2 Символьные константы

Символьные константы отличаются от числовых констант тем, что перед процессом ассемблирования они заменяются на их значения во всех строках кода, следующих за их определением, и все может стать их значением.

Определение символьных констант состоит из имени константы, за которой следует директива "equ". Все, что следует за этой дирекивой, станет значением константы. Если значение символьной константы содержит другие символьные константы, они заменяются на их значения перед присвоением значения новой константе. Например:

```
d equ dword
NULL equ d 0
d equ edx
```

После этих трех определений значение "NULL" будет "dword 0", а значение "d" будет "edx". Так, например, "push NULL" будет сассемблировано как "push dword 0", а "push d" как "push edx". А, например, в такой строке:

```
d equ d,eax
```

константе "d" будет присвоено новое значение "edx,eax". Таким образом могут определяться растущие списки символов.

"restore" позволяет присвоить назад предыдущее значение переопределенной константы. За ней должно следовать одно или больше имен символьных констант, разделенных запятыми. Так, "restore d" после предыдущего

переопределения вернет константе значение "edx", следующее применение этой директивы вернет ей значение "dword", а ещё одно применение восстановит первоначальное значение, как будто такая константа не опредеяась. Если не константа с заданным именем не определена, то "restore" не вызовет ошибку, а будет просто проигнорирована.

Символьные константы могут использоваться для адаптации синтаксиса ассемблера к персональным предпочтениям. Например, следующие определения создают удобные ярлыки для всех операторов размера:

```
b equ byte
w equ word
d equ dword
p equ pword
f equ fword
q equ qword
t equ tword
x equ dqword
```

Так как символьная константа может так же иметь пустое значение, она может использоваться для того, чтобы допустить синтаксис со словом "offset" перед каким-нибудь значением адреса:

```
offset equ
```

После такого определения "mov ax, offset char" будет правильной конструкцией, которая будет копировать смещение переменной "char" в регистр "ax",так как "offset" заменяется пустым значением, и поэтому игнорируется.

Символьные константы могут также быть определены директивой "fix", которая имеет такой же синтаксис, как "equ", но определяет константы высшего приоритета - они заменяются их символическим значением даже перед совершением директив препроцессора и макроинструкций. Исключением является сама директива "fix", которая имеет наивысший возможный приоритет, и поэтому допускает переопределение констант, заданных таким путем. Но если такие константы высшего приоритета находятся внутри значения, следующего за директивой "fix", они заменяются их значениями перед присвоением этого значения новой константе.

Директива "fix" может использоваться дл адаптирования директив препроцессора, что нельзя сделать директивой "equ". Например:

```
incl fix include
```

определяет короткое имя для директивы "include", тогда как такое же определение директивой "equ" не даст такого результата, так как стандартные символьные константы заменяются на из значения после поиска строк с директивами препроцессора.

2.3.3 Макроинструкции

"macro" позволяет вам определить собственный комплекс инструкций, называемых макроинструкциями. Их использование может существенно упростить процесс программирования. В своей простейшей форме директива похожа на описание символьной константы. Например, следующая строка определяет ярлык для инструкции "test al,0xFF":

```
macro tst {test al,0xFF}
```

После директивы "macro" должно идти имя макроинструкции и далее её содержание, заключенное между знаками "{" и "}". Вы можете использовать инструкцию "tst" в любом месте после её определения и она будет ассемблирована как "test al,0xFF". Определение символьной константы с таким значением даст похожий результат, различие лишь в том, что имя макроинструкции будет распознаваться только как мнемоник инструкции. Также, макроинструкции заменяются соответствующим кодом даже перед заменой символьных констант на их значения. То есть, если вы вы определите макроинструкцию и символьную константу под одним и тем же именем и используете это имя как мнемоник инструкции, оно будет заменено на содержание макроинструкции, но если вы используете его внутри операндов, имя будет заменено на значение символьной константы.

Определение макроинструкции может сотоять из нескольких строк, потому что знаки "{" и "}" не обязательно должны находиться на одной строке директивой "macro". Например:

```
macro stos0
 {
    xor al,al
    stosb
 }
```

Макроинструкция "stos0" будет заменена на эти две инструкции ассемблера, где бы он не использовался.

Как и инструкции, которым требуются несколько операндов, для макроинструкции можно задать требование нескольких аргументов, разделяя их запятыми. Имена этих аргументов должны следовать за именем макроинструкции на строке с директивой "macro". В любом месте в макроинструкции, где эти имена появятся, они будут заменены соответствующими значениями, указанными там, где макроинструкция используется. Вот пример макроинструкции, которая делает выравнивание данных для двоичного формата вывода:

```
macro align value { rb (value-1)-($+value-1) mod value }
```

Когда инструкция "align 4" встречается после этого задания макроинструкции, она заменяется на его содержание, и здесь "value" станет 4, а результат будет "rb (4-1)-($+4-1) mod 4".

Если в определении макроинструкции встречается её же имя, то используется предыдущее значение этого имени. Таким образом могут быть сделаны полезные переопределения макросинструкций, например:

```
macro mov op1,op2
 {
  if op1 in  & op2 in
```

```
      push  op2
      pop   op1
    else
      mov   op1,op2
    end if
   }
```

Эта макроинструкция расширяет синтаксис инструкции "mov", позволяя обоим операндам быть сегментными регистрами. Например, "mov ds,es" будет ассемблировано как "push es" и "pop ds". Во всех других случаях будет использована стандартная инструкция "mov". Синтаксис этого "mov" может быть расширен далее определением следующей макроинструкции с таким именем, который будет использовать предыдущий:

```
macro mov op1,op2,op3
 {
  if op3 eq
    mov   op1,op2
  else
    mov   op1,op2
    mov   op2,op3
  end if
 }
```

Это позволяет инструкции "mov" иметь три операнда, но она так же все ещё может иметь два операнда, так как если макроинструкции задается меньше аргументов, чем ему требуется, оставшиеся заполняются пустми значениями. Если заданы три операнда, то макроинструкция превратится в две ранее определенных, то есть "mov es,ds,dx" будет ассемблировано как "push ds", "pop es" и "mov ds,dx".

Если требуется создать макроинструкцию с аргументом, который содержит запятые, этот аргумент следует заключить между "<" и ">". Если он содержит больше одного знака "<", то для окончания его описания должно быть использовано такое же количество ">".

"purge" позволяет отменить последнее определение указанной макроинструкции. За директивой должно следовать одно или больше имен макроинструкций, разделенных запятыми. Если указанная макроинструкция не определена, это не вызовет ошибку. Например, после расширения синтаксиса "mov" вышеуказанными макроинструкциями вы можете отключить синтаксис с тремя операндами, используя директиву "purge mov". Следующее "purge mov" отключит синтаксис для сегментных регистров, а дальнейшее применение этой директивы не возымеет эффекта.

Если после директивы "macro" вы заключаете некоторую группу аргументов в квадратные скобки, это позволит при использовании макроинструкции задать данной группе аргументов больше значений. Любой следующий аргумент данный после последнего аргумента данной группы начнет новую группу и станет её первым членом. Пэтому после закрытия квадратных скобок не должно быть имен аргументов. Содержание макроинструкции будет обрабатываться для каждой такой группы аргументов отдельно. Простейший пример - это заключение одного имени аргумента в квадратные скобки:

```
macro stoschar [char]
 {
    mov al,char
    stosb
 }
```

Эта макроинструкция допускает неограниченное число аргументов, и каждый будет обработан этими двумя инструкциями отдельно. Например, "stoschar 1,2,3" будет ассемблирован как следующие инструкции:

```
mov al,1
stosb
mov al,2
stosb
mov al,3
stosb
```

Существуют некоторые специальные директивы, возможные только внутри определений макроинструкций. Директива "local" задает локальные имена, которые будут заменены уникальными значениями каждый раз, когда используется макроинструкция. За ней должны следовать имена, разделенные запятыми. Эта директива обычно требуется для внутренних констант или меток макроинструкции. Например:

```
macro movstr
 {
    local move
  move:
    lodsb
    stosb
    test al,al
    jnz move
 }
```

Каждый раз, когда используется эта макроинструкция, "move" заменяется новым уникальным именем. То есть вы не получите ошибку, это обычный случай, когда метка определяется больше, чем один раз.

"forward", "reverse" и "common" делят макроинструкцию на блоки, каждый из которых обрабатывается после окончания обработки предыдущего. Они различаются в поведении только если макроинструкция поддерживает много групп аргументов. Блок инструкций, следующий за "forward" будет обрабатываться для каждой группы аргументов от первой до последней, как блок по умолчанию (без этих директив). Блок, идущий за "reverse" будет обрабатываться для каждой группы аргументов в обратном порядке - от последней до первой. Блок за директивой "common" обрабатывается лишь один раз, просто для всех групп аргументов. Локальное имя, определеное в одном блоке, доступно во всех следующих блоках при обработке той же группы аргументов. Если оно было определено в блоке "common", оно доступно во всех следующих блоках, независимо от обрабатываемой группы.

Вот пример макроинструкции, которая создает таблицу адресов строк и следующих за ними строк.

```
macro strtbl name,[string]
 {
  common
    label name dword
  forward
    local label
    dd label
  forward
```

```
    label db string,0
  }
```

Первый аргумент, задавемый этой макроинструкции, станет меткой для таблицы адресов, следующими аргументами должны быть строки. Первый блок обрабатыается однажды и определяет метку, второй блок назначает локальную метку для каждой строки и определяет запись в таблице, содержащий адрес этой строки. Третий блок определяет данные каждой строки с соответствующей меткой.

Первая инструкция, следующая за директивой, начинающей блок в макроинструкции, может идти с ней на той же строке, как на следующем примере:

```
macro stdcall proc,[arg]
 {
  reverse push arg
  common call proc
 }
```

Это макрос может применяться для вызова процедур, используя соглашение STDCALL, аргументы сохраняются в стеке в обратном порядке. Нпример, "stdcall foo,1,2,3" будет ассемблировано так:

```
push 3
push 2
push 1
call foo
```

Если некоторе имя внутри макроинструкции имеет несколько значений (это либо один из аргументов, заключенных в квадратные скобки, либо локальное имя, определенное в блоке, следующем за директивой "forward" или "reverse") и используется в блоке, следующем за директивой "common", оно будет заменено на все значения, разделенные запятыми. Например, следующий макрос передать все дополнительные аргументы ранее определенной макроинструкции "stdcall":

```
macro invoke proc,[arg]
 { common stdcall [proc],arg }
```

Он может применяться для непрямого вызова (через указатель в памяти) процедуры, используя соглашение STDCALL.

Внутри макроинструкции также может быть использован специальный оператор "#". Этот оператор сцепляет два имени в одно. Это может быть полезно, так как делается после того, как аргументы и локальные имена заменяются на свои значения. Следующая макроинструкция генерирует условный переход в зависимости от аргумента "cond":

```
macro jif op1,cond,op2,label
 {
    cmp op1,op2
    j#cond label
 }
```

Например, "jif ax,ae,10h,exit" будет ассемблировано как инструкции "cmp ax,10h" и "jae exit".

Оператор "#" может также использоваться для объединения двух строк, заключенных в кавычки.

Возможно преобразование имени в строку в кавычках с помощью оператора "`", который также может быть ипользован внутри макроинструкции. Он конвертирует следующее за ним имя в строку, заключенную в скобки, но имейте в виду, что если за ним следует аргумент, который заменяется на значение, содержащее больше, чем один символ, будет преобразован только первый из них, так как оператор "`" конвертирует только символ, идущий непосредственно за ним. Здесь пример использования этих двух свойств:

```
macro label name
 {
    label name
    if ~ used name
      display `name # " is defined but not used.",13,10
    end if
 }
```

Если метка, определенная таким макросом, не используется в коде, он известит вас об этом сообщением, указывающим, к какой метке это относится.

Чтобы создать макроинструкцию, ведущую себя по-разному в зависимости от типа аргументов, например если это строки в кавычках, вы можете использовать оператор сравнения "eqtype". Вот пример его использования для отделения строки в кавычках от других типов аргументов:

```
macro message arg
 {
  if arg eqtype ""
    local str
    jmp   @f
    str   db arg,0Dh,0Ah,24h
    @@:
    mov   dx,str
  else
    mov   dx,arg
  end if
    mov   ah,9
    int   21h
 }
```

Вышеописанный макрос создан для показа сообщений в программах DOS. Если аргумент этого макроса некоторое число, метка или переменная, показывается строка из этого адреса, но если аргумент - это строка в кавычках, то созданный код покажет её после ... и ... .

Также возможно объявить макроинструкцию внутри другой макроинструкции, то есть один макрос может определить другой, но с такими определениями есть проблема, вызванная тем, что знак "}" не может появляться внутри макроинструкции, он всегда означает конец его определения. Чтобы обойти эту проблему, можно избавиться от мешающих символов. Это делается путем подстановки одного или боьше обратных слэшей перед

любыми другими символами (даже специальными знаками). Препроцессор видит эту последовательность как один символ, но каждый раз, когда он видит такой символ во время обработки макроса, он обрезает обратные слэши с его начала. Например, "\{" трактуется как один символ, но во время обработки макроса он станет символом "\{". Это позволит вам определить одну макроинструкцию внутри другой:

```
macro ext instr
 {
  macro instr op1,op2,op3
   \{
    if op3 eq
      instr op1,op2
    else
      instr op1,op2
      instr op2,op3
    end if
   \}
 }

ext add
ext sub
```

Макрос "ext" определен корректно, но когда он используется, символы "\{" и "\}" становятся "{" и "}". То есть когда обрабатывается "ext add", содержание макроса становится действительным определением макроинструкции, и таким образом определяется макрос "add". Так же "ext sub" определяет макрос "sub". Использование символа "\{" не было здесь действительно необходимо, но сделано таким образом для того, чтобы определение было боее ясным.

Если некоторые директивы, специфические для макроинструкций, такие как "local" или "common", требуются в некотором макросе, включенном таким образом, то их можно избежать таким же путем. Исключение символа больше чем одним обратным слэшем так же поддерживается, это позволяет допустить множественные уровни вложения определений макросов.

Другая техника определения макроинструкций внутри других состоит в использовании директивы "fix", которая становится полезной, когда некоторый макрос только начинает определение другого, без его закрытия. Например:

```
macro tmacro params
 {
  macro params {
 }

MACRO fix tmacro
ENDM fix }
```

Определяет альтернативный синтаксис определения макросов, который выглядит как:

```
MACRO stoschar char
    mov al,char
    stosb
```

```
ENDM
```

Имейте в виду, что таким образом заданное определение должно быть создано с применением директивы "fix", так как перед тем, как процессор ищет знак "}" во время определения макроса, обрабатываются только символьные константы высшего приоритета! Может возникнуть проблема, если требуется выполнить некоторые дополнительные задания в конце такого определения, но есть еще одно свойство, которое в таких случаях поможет вам. А именно возможно поместить любую директиву, инструкцию или макроинструкцию сразу после символа "}", который заканчивает макроинструкцию и она будет обработана так же, как если бы была на следующей строке.

2.3.4 Структуры

"struc" - это специальный вариант директивы "macro", который используется для определения структур данных. Макроинструкции, определенные директивой "struc", когда используются, должны предваряться меткой (как директивы определения данных). Эта метка будет также присоединена к началу каждого имени, начинающегося с точки, в содержании макроинструкции. Макроинструкция, определенная с использованием директивы "struc", может иметь такое же имя, как макросы, определенные с использованием директивы "macro". Структурная макроинструкция не будет мешать обычному макросу, выполняющемуся без метки перед ним и наоборот. Все правила и свойства, касающиеся стандартных макросов, применимы к структурным макроинструкциям.

Вот пример структуры:

```
struc point x,y
 {
    .x dw x
    .y dw y
 }
```

Например "my point 7,11" определит структуря, помеченную "my", содержащую две переменные: "my.x" со значением 7 и "my.y" со значением 11.

Еслигде-то в определении структуры находится имя, состоящие из одной лишь точки, оно заменяется на имя метки для данного примера структуры и эта метка таким образом не будет определена автоматически, позволяя полностью задать определение. Следующий пример использует это свойство, чтобы расширить определение директивы "db" с возможностью вычисления размера определяемых данных:

```
struc db [data]
 {
   common
    . db data
    .size = $ - .
 }
```

Таким образом строка "msg db 'Hello!',13,10" определит так же константу "msg.size", арвную размеру определяемых данных в байтах.

Определение структур данных, адресованных регистрами или абсолютными значениями может быть сделано структурными макроинструкциями с использованием директивы "virtual" (смотрите 2.2.3).

"restruc" удаляет последнее определение структуры, так же как "purge" делает с макросами и "restore" с символьными константами. Директива имеет тот же синтаксис - за ней должно следовать одно или несколько имен структурных макросов, разделенных запятыми.

2.3.5 Повторение макроинструкций

Директива "rept" - это специальный вид макроинструкций, который делает заданное число дубликатов блока, заключенного в фигурные скобки. Простой синтаксис - число, следующее за "rept" (это не может быть выражение, так как препроцессор не совершает вычисления, если вам нужны повторения, базирующиеся на выражениях, вычисленных ассемблером, используйте одну из директив, обрабатываемых ассемблером, смотрите 2.2.2), и блок кода, заключенный между знаками "{" и "}". Простейший пример:

```
rept 5 { in al,dx }
```

создает пять дубликатов строки "in al,dx". Блок инструкций определяется таким же образом, как для стандартных макросов, и допускаются все специальные операторы и директивы, которые могут использоваться только внутри макроинструкций. Если заданное число равно нулю, блок просто пропускается, как если бы вы определили макрос, но не использовали его. За количеством повторений может следовать имя символа-счетчика, который символьно будет заменяться на номер текущего повторения. Таким образом:

```
rept 3 counter
 {
    byte#counter db counter
 }
```

Сгенерирует строки:

```
byte1 db 1
byte2 db 2
byte3 db 3
```

Механизм повторения, применяемый к блокам "rept" такой же, как тот, что используется для обработки множественных групп аргументов макросов, то есть директивы, такие как "forward", "common" и "reverse" могут использоваться их обычном значении.

Итак, такой макрос:

```
rept 7 num { reverse display `num }
```

покажет символы от 7 до 1 как текст. Директива "local" работает так же, как внутри макросов с несколькими группами аргументов, то есть:

```
rept 21
 {
   local label
   label: loop label
 }
```

сгенерируетуникальную метку для каждого дубликата. Символ-счетчик обычно начинает с 1, но вы можете объявить другое базовое значение, предваренное запятой, сразу же после имени счетчика. Например:

rept 8 n:0 { pxor xmm#n,xmm#n }

Сгенерирует код, очищающий содержимое регистров SSE. Вы можете определить несколько счетчиков, разделенных запятыми, и каждый может иметь свою базу.

"irp" итерирует один аргумент через данный список параметров. Синтаксис такой: за "irp" следует имя аргумента, далее запятая и далее список параметров. Параметры определяются таким же образом, как в вызове стандартного макроса, то есть они должны разделяться запятыми и каждый может быть заключен между знаками "<" и ">". Так же за именем аргумента может следовать "*" для обозначения того, что он не может иметь пустое значение. Такой блок:

```
irp value, 2,3,5
 { db value }
```

Сгенерирует строки:

```
db 2
db 3
db 5
```

"irps" итерирует через данный список символов, за директивой должно следовать имя аргумента, далее запятая и далее последовательность любых символов. Каждый символ в последовательности, независимо от того, символы ли это имен, знаки символов или строки в кавычках, становится значением аргумента на одну итерацию. Если за запятой никаких символов не следует, то итераций не производится вообще. Этот пример:

```
irps reg, al bx ecx
 { xor reg,reg }
```

сгенерирует строки:

```
xor al,al
xor bx,bx
xor ecx,ecx
```

Блоки, определенные директивами "irp" и "irps", обрабатываются так же, как макросы, то есть операнды и директивы, специфичные для макросов могут в них свободно использоваться.

2.3.6 Условный препроцессинг

При применении директивы "match" некоторый блок кода обрабатывается препроцессором и передаётся ассемблеру, только если заданная последовательность символов совпадает с образцом. Образец идет первым, заканчивается запятой, далее идут символы, которые должны подходить

под образец, и далее блок кода, заключенный в фигурные скобки, как макроинструкция.

Есть несколько правил для построения выражения для сравнения, первое - это любые смвольные знаки и строки в кавычках должны соответствовать абсолютно точно. В этом примере:

```
match +,+ { include 'first.inc' }
match +,- { include 'second.inc' }
```

Первый файл будет включен, так как "+" после запятой соответствует "+" в образце, а второй файл не будет включен, так как совпадения нет.

Чтобы соответствовать любому другому символу буквально, он дооложен предварятья знаком "=" в образце. Также чтобы привести в соответствие сам знак "=", или запятую должны использоваться конструкции "==" и "=,". Например, образец "=a==" будет соответствовать последовательности "a=".

Если в образце стоит некоторый символ имени, он соответствует любой последовательности, содержащей по крайней мере один символ и его имя заменяется на поставленную в соответствие последовательность везде в следующем блоке, аналогично параметрам в макроинструкции. Напимер:

```
match a-b, 0-7
 { dw a,b-a }
```

сгенерирует инструкцию "dw 0, 7-0". Каждое имя всегда ставится в соответствие как можно меньшему количеству символов, оставляя оставшиеся, то есть:

```
match a b, 1+2+3 { db a }
```

имя "a" будет соответствовать символу "1", оставляя последоватеьность "+2+3" в соответствие с "b". Но, таким образом:

```
match a b, 1 { db a }
```

для "b" ничего не остается, и блок вообще не будет обработан.

Блок кода, определенный директивой "match" обрабатывается так же, как любая макроинструкция, поэтому здесь могут использовтся любые операторы, специфичные для макроинструкций.

Что делает директиву "match" очень полезной, так это тот факт, что она заменяет символьные константы на их значения в поставленной в соответствие последовательности символов (то есть везде после запятой до начала блока кода) перед началом сопоставления. Благодаря этому директива может использоваться, например, для обработки некоторого блока кода в зависимости от выполнения условия, что данная символьная константа имеет нужное значение, например:

```
match =TRUE, DEBUG { include 'debug.inc' }
```

здесь файл будет включен, только если символьноая константа "DEBUG" определена со значением "TRUE".

2.3.7 Порядок обработки

При сочетании разных свойств препроцессора важно знать порядок, в котором они обрабатываются. Как уже было отмечено, высший приоритет имеет директива "fix" и замены, ею определенные. Это полностью делается перед совершением любого другого препроцессинга, поэтому такой кусок кода:

```
V fix {
  macro empty
   V
V fix }
   V
```

делает допустимое определение пустого макроса. Можно сказать, что директива "fix" и приоритетные константы обрабатываются на отдельной стадии, и весь остальной препроцессинг делается на результирующем коде.

Стандартный препроцессинг, который начинается после, на каждой строке начинается с распознавания первого символа. Сначала идет проверка на директивы препроцессора, и если ни одна из них не опознана, препроцессор проверяет, является ли первый символ макроинструкцией. Если макроинструкция не найдена, препроцессор переходит ко второму символу на строке, и снова начинает с проверки на директивы, список которых в этом случае ограничивается лишь "equ", так как только она может оказать вторым символом на строке. Если нет директивы, второй символ поверяется на структурную макроинструкцию, и если ни одна из этих проверок не дала положительного результата, символьные константы заменяются на их значения, и строка передается ассемблеру.

Продемонстрируем это на примере. Пусть "foo" - это макрос, а "bar" - это структура. Эти строки:

```
foo equ
foo bar
```

Обе будут интерпретированы как вызовы макроса "foo", так как значение первого символа берет верх над значением второго.

Макроинструкции генерируют новые строки от их блоков определения, заменяя параметры на их значения и далее обрабатывая операторы "#" и "`". Оператор конверсии имеет высший приоритет, чем оператор сцепления.

После завершения этого, заново сгенерированная строка проходит через стандартный препроцессинг, как описано выше.

Хотя обычно символьные константы заменяются исключительно в строках, нет ни директив препроцессора, ни макроинструкций, встречается несколько особых ситуаций, где замены проводятся в частях строк, содержащих директивы. Первая - это определение символьной константы, где замены производятся везде после слова "equ" и результирующее значение присваивается новой константе (смотрите 2.3.2). Вторая такая ситуация - это директива "match", где замены производятся в символах, следующих за запятой перед сопоставлением их с образцом. Эти свойства могут использоваться, например, для сохранения списков, как,например, эта совокупность определений:

```
list equ
```

```
macro append item
 {
   match any, list \{ list equ list,item \}
   match , list \{ list equ item \}
 }
```

Здесь константа "list" инициализируется с пустым значением, и макрос "append" может использоваться для добавления новых пунктов к списку, разделяя их запятыми. Первое сопоставление в этом макросе происходит, только если значение списка непусто (смотрите 2.3.6), таким образом новое его значение - это предыдущее с запятой и новым пунктом, добавленным в конец. Второе сопоставление происходит, только если список все еще пуст, и таким образом список определяется как содержащий только лишь новый пункт. Так, начиная с пустого списка, "append 1" определит "list equ 1", а "append 2", следующий за ним, определит "list equ 1,2". Может потребоваться использовать этот список как параметры к некоторому макросу. Но это нельзя сделать прямо - если "foo" это макрос, то в строке "foo list" символ "list" просто прошел бы как параметр к макросу, поскольку символьные константы на этой стадии ещё не развернуты. Для этой цели снова оказывается удобна директива "match":

```
match params, list { foo params }
```

Значение "list", если оно не пустое, соответствует ключевому слову "params", которое далее во время генерации строк, заключенных в фигурные скобки, заменяется на соответственное значение. Так, если "list" имеет значение "1,2", строка, указанная выше, сгенерирует строку, содержащую "foo 1,2", которая далее пройдет стандартный препроцессинг.

Есть ещё один особый случай - когда препроцессор собирается проверить второй символ и натыкается на двоеточие (что далее интерпретируется асемблером как определение метки), он останавливается в этом месте и заканчивает препроцессинг первого символа (то есть если это символьная константа, она развертывается) и если это все еще выглядит меткой, совершается стандартный препроцессинг, начиная с места после метки. Это позволяет поместить директивы препроцессора и макроинструкции после меток, аналогично инструкциям и директивам, обрабатываемым ассемблером, например:

```
start: include 'start.inc'
```

Однако если метка во время препроцессинга разрушается (например, если у символьной константы пустое значение), происходит только замена символьных констант до конца строки.

2.4 Директивы форматирования

"format" со следующим за ним идентификатором формата позволяет выбрать формат вывода. Эта директива должна стоять в начале кода. Формат вывода по умолчанию - это простой двоичный файл, он может быть также выбран директивой "format binary".

"use16" и "use32" указывают ассемблеру генерировать 16-битный или 32-битный код, пренебрегая настройкой по умолчанию для выбранного формата вывода. "use64" включает генерирование кода для длинного режима процессоров x86.

Ниже описаны разные форматы вывода со специфичными для них директивами.

2.4.1 MZ

Чтобы выбрать формат вывода MZ, используйте директиву "format MZ". По умолчанию код для этого формата 16-битный.

"segment" определяет новый сегмент, за ним должна следовать метка, чьим значением будет номер ределяемого сегмента. Опционально за этой директивой может следовать "use16" или "use32", чтобы указать битность кода в сегменте. Начало сегмента выровнено по параграфу (16 байт). Все метки, определенные далее, будут иметь значения относительно начала этого сегмента.

"entry" устанавливает точку входа для формата MZ, за ней должен следовать дальний адрес (имя сегмента, двоеточие и смещение в сегменте) желаемой точки входа.

"stack" устанавливает стек для MZ. За директивой может следовать числовое выражение, указывающее размер стека для автоматического создания, либо дальний адрес начального стекового фрейма, если вы хотите установить стек вручную. Если стек не определен, он будет создан с размером по умолчанию в 4096 байт.

"heap" со следующим за ней значением определяет максимальный размер дополнительного места в параграфах (это место в добавление к стеку и для неопределенных данных). Используйте "heap 0", Чтобы всегда отводить только память, которая программе действительно нужна.

2.4.2 PE

Чтобы выбрать формат вывода PE, используйте директиву "format PE", за ней могут следовать дополнительные настройки формата: используйте "console", "GUI" или оператор "native", чтобы выбрать целевую субсистему (далее может следовать значение с лавающей точкой, указаывающее версию субсистемы), "DLL" помечает файл вывода как динамическую связывающую библиотеку. Далее может следовать оператор "at" и числовое выражение, указывющее базу образа PE и далее опционально оператор "on" со следующей за ним строкой в кавычках, содержащей имя файла, выбирающей заглушку MZ для PE программы (если указанный файл не в формате MZ, то он трактуетс как простой двоичный исполняемый файл и конвертируется в формат MZ). По умолчанию код для этого формата 32-битный. Пример объявления формата PE со всеми свойствами:

```
format PE GUI 4.0 DLL at 7000000h on 'stub.exe'
```

"section" определяет новую секцию, за ней должна следовать строка в кавычках, определяющая имя секции, и далее могут следовать один или больше флагов секций. Возможные флаги такие: "code", "data", "readable", "writeable", "executable", "shareable", "discardable", "notpageable". Начало секции выравнивается по странице (4096 байт). Пример объявления сиекции PE:

```
section '.text' code readable executable
```

Вместе с флагами также может быть определен один из специальных идентификаторов данных PE, отмечающий всю секцию как специальные данные, возможные идентификаторы: "export", "import", "resource" и "fixups". Если секция помечена для содержания настроек адресов, они генерируются автоматически, и никаких данных определять больше не требуется. Также данные ресурсов могут быть сгенерированы автоматически из файлов ресурсов, этого можно добиться, написав после идентификатора "resourse" оператор "from" и имя файла в кавычках. Ниже вы можете увидеть примеры секций, содержащих некоторые специальные данные:

```
section '.reloc' data discardable fixups
section '.rsrc' data readable resource from 'my.res'
```

"entry" создает точку входа для PE, далее должно следовать значение точки входа.

"stack" устанавливает размер стека для PE, далее должно следовать значение зарезервиронного размера стека, опционально может следовать отделенное запятой значение начала стека. Если стек не определен, ему присваивается размер по умолчанию, равный 4096 байт.

"heap" выбирает размер дополнительного места для PE, далее должно следовать значение для зарезервированного для него места, опционально ещё может быть значение его начала, отделенное запятой. Если дополнительное место не определено, оно ставится по умолчанию равным 65536 байт, если не указано его начало, то оно устанавливается равным 0.

"data" начинает определение специальных данных PE, за директивой должен следоваь один из идентификаторов данных ("export", "import", "resource" или "fixups") или номер записи данных в заголовке PE. Данные должны быть определены на следующих строках и заканчиваться директивой "end data". Если выбрано определение настроек адресов, они генерируются автоматически, и никаких данных определять больше не требуется. То же самое относится к ресурсам, если за идентификатором "resourse" следует оператор "from" и имя файла в кавычках - в этом случае данные берутся из этого файла ресурсов.

2.4.3 COFF

Чтобы выбрать COFF (Common Object File Format), используйте директиву "format COFF" или "format MS COFF", если вы хотите создать классический мелкософтофский файл COFF. По умолчанию код для этого формата 32-битный. Чтобы создать микросФтовский формат COFF для архитектуры x86-64, используйте установку "format MS64 COFF", в этом случае автоматически будет генерироваться код длинного режима.

"section" определяет новую секцию, за директивой должна следовать строка в кавычках, определяющая имя новой секции, и ещё может следовать один или более флагов секций. Возможные флаги такие: "code" и "data" для обоих вариантов COFF, "readable", "writeable", "executable", "shareable", "discardable" и "notpageable" толко для микросовтофского варианта COFF. По умолчанию секция выровнена по двойному слову (четыре байта), но микросовтовский вариант COFF можно выровнять еще как-нибудь по-другому с помощью оператора "align" и следующим за ним значением выравнивания (любая степень двойки от двух до 8192) среди фагов секций.

"extrn" определяет внешний символ, за ним должно следовать имя символа и опционально оператор размера, указывающий размер данных, помеченных этим символом. Имя символа также может предваряться строкой в кавычках, содержащей имя внешнего символа и оператор "as". Пара примеров объявления внешних символов:

```
extrn exit
extrn '__imp__MessageBoxA@16' as MessageBox:dword
```

"public" объявляет существующий символ как общедоступный, за ним должно следовать имя символа, и далее опционально оператор "as" и строка в кавычках, содержащая имя, под которым символ будет действителен как общедоступный. Пара примеров объявления общедоступных символов:

```
public main
```

```
public start as '_start'
```

2.4.4 ELF

Чтобы выбрать формат вывода ELF, используйте директиву "format ELF". По умолчанию код для этого формата 32-битный. Чтобы создать формат ELF для архитектуры x86-64, используйте установку "format ELF", в этом случае автоматически будет генерироваться код длинного режима.

"section" определяет новую секцию, за директивой должна следовать строка в кавычках, определяющая имя новой секции, и ещё может следовать один или оба флага "executable" и "writeable", опционально также может идти оператор "align" со следующим за ним числом, определяющим выравнивание секции (это должна быть степень двойки), если выравнивание не указано, используется значение по умолчанию, которое равно 4 или 8, в зависимости от варианта выбранного формата.

"extrn" и "public" имеют те же значения и синтаксис у них, как в случае формата COFF (описанного в предыдущем параграфе).

Чтобы создать исполняемый файл, придерживайтесь директивы выбора формата со словом "executable".Это позволяет использовать директиву "entry" со следующим за ним значением, чтобы создать точку входа в программу. С другой стороны это делает недоступными директивы "extrn" и "public". За директивой "section" в этом случае может следовать только один или более флагов секций и начало секции будет выровнено по странице (4096 байт). Доступные флаги секций такие: "readable", "writable" и "executable".

Translate made by Paranoik
Converted to HTML by Alexandr Kolodkin

-=EOF=-

Последнее обновление : 25 января 2008