Mikl___
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

Мануал по flat assembler

10.08.2014, 06:58. **Просмотров** 96964. **Ответов** 50
Метки нет (Все метки)

flat assembler 1.71
Мануал программера

перевод "flat assembler 1.71 Programmer's Manual" by Tomasz Grysztar

перевод выполнили **Paranoik** и **Mikl___**

Содержание

2. PE (Portable Executable)
3. COFF (Common Object File Format)
4. ELF (Executable and Linkable Format)

Глава 3. Windows-программирование на FASM

1. Основные заголовочные файлы
   1. Структуры
   2. Импорт
   3. Процедуры (32-разрядные)
   4. Процедуры (64-разрядные)
   5. Customizing procedures
   6. Экспорт
   7. COM (Component Object Model)
   8. Ресурсы
   9. Кодировка текста
2. Расширенные заголовочные файлы
   1. Параметры процедур
   2. Структурирование исходников

10

Вложения

FASM.PDF (505.8 Кб, 804 просмотров)

Лучшие ответы (1)

Сообщение: #6561513

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 06:58 **[ТС]** **2**

Сообщение было отмечено R71MT как решение

Решение

Глава 1. Введение

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

1.1 Обзор компилятора

FASM написан на самом себе, обладает небольшими размерами и очень высокой скоростью компиляции, имеет богатый и емкий макро-синтаксис, позволяющий автоматизировать множество рутинных задач. Поддерживаются как объектные форматы, так и форматы исполняемых файлов. Это позволяет в большинстве случаев обойтись без линкера. В остальных случаях нужно использовать сторонние линкеры, поскольку линкер вместе с FASM не распространяется.

Помимо базового набора инструкций процессора x86 и сопроцессора FASM поддерживает наборы инструкций MMX, SSE, SSE2, SSE3, SSSE3, SSE4.1, SSE4.2, SSE4a, AVX и 3DNow!, а также EM64T и AMD64 (включая AMD SVM и Intel SMX). FASM — это быстрый компилятор языка ассемблер, который выполняет несколько проходов компиляции для оптимизации генерируемого машинного кода. FASM способен работать в следующих операционных системах:

- DOS, расширенные DOS
- Windows 9X, NT, XP, Vista, Seven
- Основанные на Linux — напрямую, через системные вызовы
- FreeBSD
- Другие, основанные на libc (UNIX-подобные).
- MenuetOS, KolibriOS — не поставляется в стандартном пакете FASM, поставляется вместе с этими операционными системами.

Версии для разных операционных систем предназначены для использования из системной командной строки и не отличаются поведением.

Этот документ описывает также IDE версию, разработанную для операционной системы Windows, которая использует графический интерфейс взамен консольного и обладает встроенным текстовым редактором. С точки зрения компиляции она обладает в точности такой же функциональностью как и все консольные версии, и дальнейшие части (начиная с 1.2) этого документа являются общими для всех версий компилятора. Исполняемый файл IDE версии называется **fasmw.exe**, в то время как консольная версия называется **fasm.exe**.

2

Вложения

FASM.PDF (505.8 Кб, 182 просмотров)

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 08:11 **[ТС]** **3**

1.1.1 Системные требования

Все версии требуют для своей работы 32-битного процессора архитектуры x86 (как минимум 80386), хотя могут также генерировать программы и для 16-битных процессоров. Консольная версия компилятора требует для работы любую Win32 совместимую операционную систему, в то время как версия с графическим интерфейсом требует Win32 GUI операционную систему версии 4.0 и выше, так что компилятор будет работать на любой системе начиная с Windows 95 и выше.

Исходные коды программ-примеров поставляемые с этой версией компилятора для успешной компиляции требуют, чтобы переменная окружения **INCLUDE** хранила полный путь к папке **include**, которая является частью пакета. Если такая переменная уже существует в вашей системе и содержит пути используемые другой программой, достаточно добавить к ней новый путь (разные пути разделяются точкой с запятой). Если вы не хотите определять такую переменную в системе, или не знаете как это сделать, то для IDE версии вы можете установить её редактированием файла **fasmw.ini** в папке с исполняемым файлом компилятора (этот файл создаётся компилятором fasmw.exe при запуске, но может также быть создан вручную). В этом случае вам необходимо добавить значение Include в секцию

Environment. Например, если вы распаковали файлы FASM'а в папку **c:\fasmw**, вы должны добавить следующие строки в файл **c:\fasmw\fasmw.ini**:

Код Выделить код

```
[Environment]
Include = c:\fasmw\include
```

Если вы не определите переменную окружения **INCLUDE** правильно, вам придется вручную указывать полный путь к заголовочным файлам Win32 в каждой программе, которую вы хотите скомпилировать.

1.1.2 Использование компилятора

Для того, чтобы начать работать с FASM'ом, просто запустите файл fasmw.exe, или перетащите иконку исходного файла на ярлык программы fasmw.exe в проводнике. Также вы можете открывать новые исходные файлы с помощью команды **Open** в меню **File**, или перетаскивая файлы в окно редактора. В редакторе могут быть открыты несколько файлов одновременно, каждый из них представляется закладкой в нижней части окна редактора. Выбрать файл для редактирования можно щёлкнув левой кнопкой мыши на соответствующей закладке. По умолчанию компилятор работает с редактируемым в данный момент файлом, но вы можете принудить его работать с конкретным файлом щёлкнув на соответствующей закладке правой кнопкой мыши и выбрав в контекстном меню пункт **Assign**. Одновременно к компилятору может быть привязан только один файл.

Когда ваш исходный файл готов, вы можете выполнить компиляцию выбрав команду **Compile** из меню **Run**. Если компиляция пройдет успешно, компилятор отобразит окно результатов компиляции; иначе он выведет информацию о произошедших ошибках. Окно результатов компиляции содержит информацию о количестве проходов, длительности компиляции и количестве байт записанных в результирующий файл. Оно также содержит текстовое поле называемое Display, в котором отображаются все сообщения от директив display в исходном коде (см. 2.2.3). Сводка ошибок содержит как минимум сообщение об ошибке и текстовое поле **Display**, того же назначения. Если ошибка связана с конкретной строкой исходного кода, сводка содержит также текстовое поле **Instruction**, которое содержит препроцессированную форму инструкции вызвавшей ошибку если ошибка произошла после стадии препроцессора (иначе поле пустое) и список **Source**, который показывает расположение всех строк исходного кода связанных с этой ошибкой, если вы выберете строку из этого списка, она одновременно выбрана в окне редактора (если соответствующий файл не открыт, он будет автоматически загружен).

Команда **Run** также выполняет компиляцию, и в случае успешного её завершения запускает скомпилированную программу в том случае, если она относится к одному из форматов, запускаемых в среде Windows, иначе выводится сообщение о том, что такой тип файла не может быть запущен. Если при компиляции возникают ошибки, выводится сводка по ошибкам, как для команды **Compile**.

Если компилятору не хватает памяти, вы можете увеличить используемый её объём, открыв окно **Compiler Setup** из меню **Options**. В нем вы можете указать объём памяти в килобайтах который компилятор должен использовать и также задать приоритет потока компилятора.

1.1.3 «Горячие клавиши» для работы с редактором

В этом разделе перечислены все команды доступные с клавиатуры при работе с редактором. Клавиши, перечисленных как «специфические исключения», являются общими с FASM DOS-IDE.

Перемещение курсора по тексту программы:	
Left	перейти на один символ назад
Right	перейти на один символ вперед
Up	перейти на одну строку выше
Down	перейти на одну строку ниже
Ctrl+Left	перейти на одно слово назад
Ctrl+Right	перейти на одно слово вперед
Home	перейти на первый символ строки
End	перейти на последний символ строки
PageUp	перейти на один экран вверх
PageDown	перейти на один экран вниз
Ctrl+Home	перейти на первую строку экрана
Ctrl+End	перейти на последнюю строку экрана
Ctrl+PageUp	перейти на первую строку программы
Ctrl+PageDown	перейти на последнюю строку программы

Each of the movement keys pressed with Shift selects text.

Редактирование:	
Insert	переключение режима вставка/замена
Alt+Insert	переключение между вертикальными/горизонтальным выделением
Delete	удалить текущий символ
Backspace	удалить предыдущий символ
Ctrl+Backspace	удалить предыдущее слово
Alt+Backspace	отменить предыдущее действие (также Ctrl+Z)
Alt+Shift+Backspace	повторить отмену предыдущего действия (также Ctrl+Shift+Z)

Редактирование:	
Ctrl+Y	удалить текущую строку
F6	дублировать текущую строку

Операции с выделенными фрагментами:	
Ctrl+Insert	Копировать выделенный фрагмент в Буфер Обмена (также Ctrl+C)
Shift+Insert	Вставить содержимое Буфера Обмена в заданное место (также Ctrl+V)
Ctrl+Delete	Удалить выделенный фрагмент
Shift+Delete	"Вырезает" выделенный фрагмент и помещает его в Буфер Обмена (также Ctrl+X)
Ctrl+A	Выделить весь текст программы

Поиск:	
F5	перейти на заданную строку (также Ctrl+G)
F7	поиск (также Ctrl+F)
Shift+F7	поиск следующего (также F3)
Ctrl+F7	замена (также Ctrl+H)

Компиляция:	
F9	компилировать и запустить
Ctrl+F9	только компилировать
Shift+F9	сделать выбранный файл основным для компиляции
Ctrl+F8	компиляция с добавлением отладочной информации

Другие клавиши:	
F2	сохранить текущий файл
Shift+F2	сохранить файл под новым именем
F4	загрузить файл
Ctrl+N	создать новый файл
Ctrl+Tab	переключиться на следующий файл
Ctrl+Shift+Tab	переключиться на предыдущий файл
Alt+[1-9]	переключиться на файл с заданной вкладки
Esc	закрыть текущий файл
Alt+X	закрыть все файлы и выйти из программы
Ctrl+F6	вызвать калькулятор
Alt+Left	arrow scroll left
Alt+Right	arrow scroll right
Alt+Up	arrow scroll up
Alt+Down	arrow scroll down

Специфические клавиши:	
F1	поиск ключевого слова в выбранной справочной системе
Alt+F1	вызов контекстной помощи

1.1.4 Настройка IDE

В меню **Options** также содержится список опций редактора, который может быть включен или выключен в зависимости от текущего состояния редактора. В этом разделе описываются эти опции.

Secure selection — Это вариант, когда вы включаете, выделенный фрагмент никогда не будет удален, если вы начинаете набирать текст на клавиатуре. Когда вы делаете любой текстовый-операцию изменения, выбор будет отменен, ни в коем случае, влияющие Это был текст выбран, и тогда команда выполняется. Когда эта опция выключена, и вы начинаете набирать, текущее выделение отбрасывается, просто Также клавиша Del удаляет выделенный блок (когда выбор будет на безопасной Вы должны использовать Ctrl + Del).

Автоматические скобки — когда вы набираете открытую скобку любого типа, закрыв один будет автоматически переведен только после вставки.

Автоматические отступы — При нажатии Enter, чтобы начать новую строку, каретка перемещается в новую строку в том же положении, в предыдущей строке, где первый непустой символ помещается. Если вы нарушаете строку, и были некоторые непустые символы после вставки, когда вы нажимаете Enter, они перемещаются в новую строку в положении отступа, ни пробелов, которые между кареткой и их игнорируются.

Смарт табуляция — при нажатии Tab, он перемещает вас в положение чуть ниже следующей последовательности

символов, кроме пробелов в указанном выше начиная с позиции чуть выше, где вы были строка. Если такой последовательности не найдена в строке выше, стандартный размер табуляции 8 символов используется.

Оптимальное сохранение — после включения опции, когда файл сохраняется, все пустые места заполнены оптимальному сочетанию вкладок и пробелов, чтобы получить меньший размер файла. Если эта опция выключена, то пустые участки сохраняются как заполненные пробелами (но пробелы в концах строк не сохраняются).

«Оживление мертвых клавиш» — когда эта опция включена, она отключает окна редактора так называемые «мертвые ключи» (ключи, которые не сразу генерируют символ, но ждать следующего нажатия клавиши, чтобы решить, какой знак надо поставить — как правило, вы вводите символ мертвого ключа при нажатии на клавишу пробела после него). Это может быть полезно, если ключ для ввода некоторых персонажей, которые вам нужно ввести часто в источнике сборки мертвый ключ и вам не нужно эту функцию для написания программ.

3

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 08:11 **[ТС]**

1.1.5 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
Если второй параметр не задан, название файла вывода будет создано автоматически. После показа короткой информации о названии прог
компилятор считает информацию из файла с кодом и скомпилирует её. После успешной компиляции FASM запишет сгенерированный код в
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
табуляции обрабатываются как пробелы.
В командную строку вы также можете включить опцию "**-m**", за которой должно следовать число, указывающее, сколько килобайт памяти
задействовано FASM'ом. В случае DOS-версии эта опция ограничивает лишь использование расширенной памяти. Опция "**-p**" со следующим
использована для того, чтобы ограничить количество проходов, которое будет делать ассемблер. Если код не может быть создан заданным
ассемблирование прекратится с сообщением об ошибке. Максимальное значение этой опции равно 65536, а значение по умолчанию равно
Не существует опций, оказывающих воздействие на выходные данные компилятора, вся необходимая FASM'у информация должна содержат
Например, для установки формата файла-адресата используется директива "**format**" в начале кода.

1.1.6 Сообщения компилятора

Как было сказано выше, после успешной компиляции FASM выводит на экран сводку о проделанной работе. Она включает информацию о то
проходов, сколько времени это заняло, и сколько байт записано в файл-адресат.
Вот пример такой сводки:

Код Выделить код

```
flat assembler  version 1.71
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 фа
следующее сообщение:

Код Выделить код

```
flat assembler  version 1.71
error: source file not found.
```

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

Код Выделить код

```
flat assembler  version 1.71
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, со
будет выведена строка в формулировке макроса, которая сгенерировала ошибочную инструкцию:

Код Выделить код

```
flat assembler  version 1.71
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.

1.1.7 Форматы вывода

По умолчанию, если в исходнике нет директивы "format", flat assembler направляет сгенерированный код на вывод, создавая таким образом
умолчанию он создает 16-битный код, но вы всегда можете переключить его в 32-битный или 16-битный режим, используя директивы "use3
некоторых форматов файла-адресата автоматически переключает компилятор в 32-битный режим. Подробнее читайте о форматах, которые

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 ассемб
читайте учебники по программированию на ассемблере.
Flat assembler по умолчанию использует Intel-синтаксис ассемблерных инструкций, однако вы можете переделать их, используя возможност
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
точек с запятой, заключенных в кавычки, остаток её считается комментарием и компилятор её проигнорирует. Если строка содержит симв
с запятой и комментарий могут за ней следовать), то следующая срока прикрепляется к этой точке. После символа "\" строка не должна с
комментариев, следующих за точкой с запятой.
Каждая строка в коде - это последовательность отдельных предметов,каждый из которых может принадлежать к одному из трех типов. Пер
символов, которыми являются специальные знаки, которые трактуются отдельно, даже если не отделены пробелами о других символов. Лю
{}:,|&~#`" - это знак символа. Последовательность других знаков, отделенная от остальных либо пробелами, либо знаками символов, это
такого символа двойная либо одинарная кавычка, он объединяет любую последовательность символов, даже специальных, следующих за н
заканчиваться таким же знаком, каким начиналась (двойной либо одинарной кавычкой). Однако если встречаются две кавычки подряд (без
также включаются в строку и она продолжается. Символы, отличные от знаков символов и строк, заключенных в кавычки, могут использов
они также называются символами имен.
Каждая инструкция состоит из мнемоника и различного числа операндов, разделенных запятыми. Операндом может быть регистр, непосред
адрес в памяти, он также может предварен оператором размера, чтобы определить или изменить его размер (таблица 1.1). Названия возмо
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 б
скобки или предварен оператором "ptr". Например, инструкция "mov eax,3" поместит число 3 в регистр EAX, а инструкция "mov eax,[7]" пом
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
какой сегментный регистр будет использоваться для адресации, нужно поставить его название с двоеточием перед адресом внутри квадра
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
tword	80	10
dqword	128	16
xword	128	16
qqword	256	32
yword	256	32
zword	512	64

Таблица 1.2 Регистры

Тип	Биты	Регистры																
General	8	al	cl	dl	bl	spl	bpl	sil	dil	ah	ch	dh	bh	r8b	r9b	r10b	r11b	r12b	r13b
	16	ax	cx	dx	bx	sp	bp	si	di	-	-	-	-	r8w	r9w	r10w	r11w	r12w	r13w
	32	eax	ecx	edx	ebx	esp	ebp	esi	edi	-	-	-	-	r8d	r9d	r10d	r11d	r12d	r13d
	64	rax	rcx	rdx	rbx	rsp	rbp	rsi	rdi	-	-	-	-	r8	r9	r10	r11	r12	r13
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	-	-	-	-	xmm8	xmm9	xmm10	xmm11	xmm12	xmm1
AVX	256	ymm0	ymm1	ymm2	ymm3	ymm4	ymm5	ymm6	ymm7	-	-	-	-	ymm8	ymm9	ymm10	ymm11	ymm12	ymm1
AVX2	512	zmm0	zmm1	zmm2	zmm3	zmm4	zmm5	zmm6	zmm7	-	-	-	-	zmm8	zmm9	zmm10	zmm11	zmm12	zmm1

1

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

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 числовых 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 создания 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 двух 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
16	ddq	rdq

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 ближайшую 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 операторы 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" вычисляет остаток от деления нацело. "and", "or", "xor", "shl", "shr" и "not" совершают те же логические 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 начинаться символами "0x" (как в языке C), или символом "$" (как в языке Pascal) или должны заканчиваться буквой "h". Также заключенная в кавычки строка при включении в выражение будет конвертирована в число - первый символ станет минимальным значащим байтом числа. Числовые выражения, используемые как значения адреса, могут также содержать любой из общих регистров, используемых для адресации, они могут быть сложены или умножены на подходящие значения так, как это позволено в инструкциях архитектуры x86.
Также есть несколько специальных символов, которые могут быть использованы в числовом выражении. Первое - это "$", 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", который 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.

4

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 08:21 **[ТС]** **6**

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.

1

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 08:44 **[TC]** **7**

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.

3

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 08:44 **[TC]** **8**

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 назначения 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:

Assembler Выделить код

```
1       mov bx,ax       ; поместить содержимое регистра общего назначения AX в регистр общего назначения BX
2       mov [char],al   ; поместить содержимое регистра общего назначения AL в ячейку памяти размером 8 бит
3       mov bl,[char]   ; поместить содержимое ячейки памяти в регистр общего назначения BL
4       mov dl,32       ; поместить непосредственное значение в регистр общего назначения DL
5       mov [char],32   ; поместить непосредственное значение в ячейку памяти
6       mov ax,ds       ; поместить содержимое сегментного регистра DS в регистр общего назначения AX
7       mov [bx],ds     ; поместить содержимое сегментного регистра в ячейку памяти адресуемую через регистр BX
8       mov ds,ax       ; поместить содержимое регистра общего назначения AX в сегментный регистр DS
9       mov ds,[bx]     ; поместить содержимое ячейки памяти, адресуемую через регистр BX, в сегментный регистр
10      mov eax,cr0     ; поместить содержимое контрольного регистра CR0 в регистр общего назначения EAX
11      mov cr3,ebx     ; поместить содержимое регистра общего назначения EBX в контрольный регистр CR3
```

- "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:

Assembler Выделить код

```
1       xchg ax,bx      ; обмен содержимым регистров общего назначения AX и BX
2       xchg al,[char]  ; обмен содержимым регистра общего назначения AL и ячейки памяти
```

- "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:

Assembler Выделить код

```
1       push ax         ; сохранить содержимое регистра общего назначения AX в стеке
2       push es         ; сохранить содержимое сегментного регистра в стеке
3       pushw [bx]      ; сохранить содержимое ячейки памяти в стеке
4       push 1000h      ; поместить в стек 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 операнд-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.

Assembler Выделить код

```
1       pop bx          ; восстанавливает регистр общего назначения
2       pop ds          ; восстанавливает сегментный регистр
3       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".

1

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 08:44 **[ТС]** **9**

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:

Assembler Выделить код

```
1       movsx ax,al         ; байт в слово
2       movsx edx,dl        ; байт в двойное слово
3       movsx eax,ax        ; слово в двойное слово
4       movsx ax,byte [bx]  ; байт памяти в слово
5       movsx edx,byte [bx] ; байт памяти в двойное слово
6       movsx eax,word [bx] ; слово памяти в двойное слово
```

1

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 10:32 **[ТС]** **10**

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 назначения или непосредственное значение. Также это может быть память, если адресат - это регистр.

Assembler Выделить код

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

- "adc" суммирует операнды, прибавляет единицу, если стоит CF и заменяет адресат результатом. Правила для операндов такие же как с инструкцией "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.

Assembler Выделить код

```
1       inc ax          ; прибавляет единицу к регистру
2       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 следующими 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 возвращает 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. Этим 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:

Assembler Выделить код

```
1      imul bl         ; умножение аккумулятора на регистр
2      imul word [si]  ; умножение аккумулятора на память
3      imul bx,cx      ; умножение регистра на регистр
4      imul bx,[si]    ; умножение регистра на память
5      imul bx,10      ; умножение регистра на непосредственное значение
6      imul ax,bx,10   ; регистр, умноженный на непосредственное значение, в регистр
7      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.

3

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 10:32 **[ТС]** **11**

2.1.4 Двоично-десятичные арифметические инструкции

Десятичная арифметика представлена в виде соединения двоичных арифметических инструкций (описанных в предыдущем параграфе) с десятичными арифметическими инструкциями. Десятичные арифметические инструкции используются для того, чтобы приспособить результаты предыдущей двоичной арифметической операции для создания допустимого упакованного или неупакованного десятичного результата (BCD-числа), или приспособить входные данные для последующей двоичной арифметической операции так, чтобы эта операция также давала допустимый упакованный или неупакованный 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, устанавливается 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, устанавливается 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 создаваемых чисел.
- "aad" модифицирует делимое в AH и AL, чтобы подготовится к делению двух допустимых несжатых десятичных операндов так, чтобы частное стало допустимым 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".

1

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 10:51 **[ТС]** **12**

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 битом, 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.

Assembler Выделить код

```
1      bt  ax,15        ; тестирует бит в регистре
2      bts word [bx],15 ; тестирует и ставит бит в памяти
3      btr ax,cx        ; тестирует и сбрасывает бит в регистре
4      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).

Assembler Выделить код

```
1      bsf ax,bx        ; сканирование регистра по возрастанию
2      bsr ax,[si]      ; сканирование памяти 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 не синоним "shl".

Assembler Выделить код

```
1      shl al,1         ; сдвиг регистра влево на один бит
2      shl byte [bx],1  ; сдвиг памяти влево на один бит
3      shl ax,cl        ; сдвиг регистра влево на количество из CL
4      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 операндов такие же, как с инструкцией "shl". "shr" "задвигает" нули с левой стороны операнда-адресата, биты "выходят" справа. Последний "вытолкнутый" 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.

Assembler Выделить код

```
1      shld ax,bx,1     ; сдвиг регистра влево на один бит
2      shld [di],bx,1   ; сдвиг памяти влево на один бит
3      shld ax,bx,cl    ; сдвиг регистра влево на количество из CL
4      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 обратным порядком и наоборот.

Assembler Выделить код

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

4

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 10:51 **[ТС]** **13**

2.1.6 Инструкции передачи управления

"jmp" 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 инструкция "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.

Assembler Выделить код

```
1      jmp 100h         ; прямой близкий переход
2      jmp 0FFFFh:0     ; прямой дальний переход
3      jmp ax           ; непрямой близкий переход
4      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 вызывающая 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 3) 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 3.1 Условия

Мнемоник	Тестируемое условие	Описание
o	OF = 1	переполнение
no	OF = 0	нет переполнения
c b nae	CF = 1	перенос меньше не больше и не равно
nc ae nb	CF = 0	нет переноса выше или равно не меньше
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	отрицательное
ns	SF = 0	положительное
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 установки разрядности

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 мнемоникой "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".

Assembler Выделить код

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

2

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 10:51 **[TC]** **14**

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.

Assembler Выделить код

```
1       in al,20h        ; ввод байта из порта 20
2       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.

Assembler Выделить код

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

2

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 17:21 **[TC]** **15**

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 строковой 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 адрес 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.

Assembler Выделить код

```
1       movs byte [di],[si]        ; переводит байт
2       movs word [es:di],[ss:si]  ; переводит слово
3       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 строковые 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.

Assembler Выделить код

Assembler Выделить код

```
1      cmpsb                    ; сравнение байтов
2      cmps word [ds:si],[es:di]  ; сравнение слов
3      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.

Assembler Выделить код

```
1      scas byte [es:di]          ; сканирует байт
2      scasw                      ; сканирует слово
3      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.

Assembler Выделить код

```
1      lods byte [ds:si]          ; загружает байт
2      lods word [cs:si]          ; загружает слово
3      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.

Assembler Выделить код

```
1      insb                       ; ввод байта
2      ins word [es:di],dx        ; ввод слова
3      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.

Assembler Выделить код

```
1      outs dx,byte [si]          ; вывод байта
2      outsw                      ; вывод слова
3      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.

Assembler Выделить код

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

3

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

10.08.2014, 17:38 **[TC]** **16**

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 четыре, в зависимости от текущей настройки кода. Вариант "popfw" сохранят слово, независимо от настройки кода, "popfd" также независимо от настройки кода сохраняет двойное слово.

3

Mikl
Ушел с форума
13953 / 6974 / **804**

10.08.2014, 17:38 **[TC]** **17**

2.1.10 Условные операции

Регистрация: 11.11.2010
Сообщений: 12,545

Инструкции, образованные с помощью добавления условного мнемоника (смотрите таблицу 3) 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.

Assembler Выделить код

```
1       setne al         ; единицу в al, если флаг нуля пустой
2       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 инструкции нет аргументов.
- Инструкции, 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.

Assembler Выделить код

```
1       cmove ax,bx      ; переводит, если установлен флаг нуля
2       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.

Assembler Выделить код

```
1       cmpxchg dl,bl    ; сравнивает и меняет с регистром
2       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.

Assembler Выделить код

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

1

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

11.08.2014, 04:33 **[ТС]** **18**

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 программного 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.

Assembler Выделить код

```
1       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.

Assembler Выделить код

```
1       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 инструкцию. У этой инструкции нет операндов.
- "pause" задерживает выполнение следующей инструкции, реализуясь определенное 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 нее 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.

Assembler Выделить код

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

2

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

11.08.2014, 05:00 **[ТС]** **19**

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.

Assembler Выделить код

```
1       lmsw ax          ; загружает машинный статус из регистра
2       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 регистра таблицы дескриптров прерываний в операнд-адресат. Операндом должны быть 6 байтов в памяти.

Assembler Выделить код

```
1       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.

Assembler Выделить код

```
1       lar ax,[bx]      ; загружает права доступа в слово
2       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.

Assembler Выделить код

```
1       arpl bx,ax       ; подгоняет RPL селектора в регистре
2       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 сопутствующей 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 префикса "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

Mikl
Ушел с форума
13953 / 6974 / **804**
Регистрация: 11.11.2010
Сообщений: 12,545

11.08.2014, 05:11 **[ТС]** **20**

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 так далее. 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.

Assembler Выделить код

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

- "fld1", "fldz", "fldl2t", "fldl2e", "fldpi", "fldlg2" и "fldln2" загружают часто используемые константы в стек регистров FPU. Это константы +1.0, +0.0, $log_2(10)$, $log_2(e)$, 3,14159265358979323846264338327 95, $lg(2)$=0,30102999566398119521373889472449 и $ln(2)$=0,69314718055994530941723212145818 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.

Assembler Выделить код

```
1      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.

Assembler Выделить код

```
1      fst st3          ; копирует значение ST0 в регистр ST3
2      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.

Assembler Выделить код

```
1      fadd qword [bx]  ; прибавляет значение двойной точности к ST0
2      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.

Assembler Выделить код

```
1      faddp            ; прибавляет st0 к st1 и выдвигает стек
2      faddp st2,st0    ; прибавляет st0 к st2 и выдвигает стек
```

- "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.

Assembler Выделить код

```
1      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 вышеперечисленные 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.

Assembler Выделить код

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

- "fcompp" сравнивает содержимое ST0 и ST1, устанавливает флаги в слове 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.

2

Ответить Создать тему

1 из 3 **1** 2 3 Ctrl > »