КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 10 базовые инструкции IA32

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2022

2022.04.06/13

Оглавление

Базовые инструкции IA32

- инструкции перемещения данных;
- арифметические инструкции (двоичная и двоично-десятичная арифметика);
- логические инструкции;
- инструкции сдвигов;
- инструкции для работы с битами и байтами;
- инструкции для работы со строками;
- инструкции передачи управления;
- инструкции ввода/вывода;
- инструкции управления флагами;
- инструкции для работы с сегментными регистрами;
- другие инструкции.

Инструкции для работы со строками

Строковые инструкции предназначены для доступа к большим структурам данных.

MOVS m, m (MOVSB/MOVSW/MOVSD) — переместить данные из строки в строку;
CMPS m, m (CMPSB/CMPSW/CMPSD) — сравнить строковые операнды;
SCAS m (SCASB/SCASW/SCASD) — сканировать строку (EAX);
LODS m (LODSB/LODSW/LODSD) — загрузить строку (EAX);
STOS m (STOSB/STOSW/STOSD) — сохранить строку (EAX);

MOVS перемещает элемент строки, адресуемый регистром **ESI**, в местоположение, адресуемое регистром **EDI**.

```
ES:(E)DI <- DS:(E)SI)
```

CMPS сравнивает элемент строки назначения и элемент строки источника и обновляет флаги состояния (**CF**, **ZF**, **OF**, **SF**, **PF** и **AF**) в регистре **EFLAGS** в соответствии с результатами.

Ни один из элементов строки обратно в память не записывается.

SCAS сравнивает содержимое регистра **EAX**, **AX** или **AL** в зависимости от длины операнда и элемент строки назначения, после чего обновляет флаги состояния в соответствии с результатами.

Элемент строки и содержимое регистра не изменяются.

LODS загружает элемент строки источника, идентифицируемый регистром **ESI**, в регистр **EAX** (для строки из двойных слов), регистр **AX** (для строки из слов) или регистр **AL** (для строки из байтов).

Эта инструкция обычно используется в цикле, где другие инструкции обрабатывают каждый элемент строки после их загрузки в регистр назначения.

STOS сохраняет элемент строки источника из регистра **EAX** (для строки из двойных слов), регистра **AX** (для строки из слов) или регистра **AL** (для строки из байтов) в ячейке памяти, идентифицированной с помощью регистра **EDI**.

Эта инструкция также обычно используется в цикле. Строка обычно загружается в регистр с помощью инструкции **LODS**, обрабатывается другими инструкциями и затем снова сохраняется в памяти с помощью инструкции **STOS**.

Общие особенности операций

Команды **MOVS** (Переместить строку), **CMPS** (Сравнить строку), **SCAS** (Сканировать строку), **LODS** (Загрузить строку) и **STOS** (Сохранить строку) позволяют перемещать и анализировать большие объемы данных, такие как строки буквенно-цифровых символов.

Инструкции работают с отдельными элементами в строке размером 1, 2, 4 байта.

Элементы, с которыми нужно работать, указываются с помощью регистров **ESI** (элемент строки источника) и **EDI** (элемент строки приемника).

Оба этих регистра должны содержать абсолютные адреса (смещения в сегменте), которые указывают на элемент строки.

По умолчанию регистр **ESI** обращается к сегменту, ассоциированному с регистром сегмента **DS** (**DS:ESI**). **DS (сегмент источника) может быть переопределен.**

Префикс переопределения сегмента позволяет регистру **ESI** быть связанным с сегментными регистрами **CS**, **SS**, **ES**, **FS** или **GS**.

По умолчанию регистр **EDI** обращается к сегменту, ассоциированному с регистром сегмента **ES** (**ES:EDI**). **Переопределение ES (сегмент приемника) не допускается.**

Использование двух разных регистров сегментов в строковых инструкциях позволяет выполнять операции над строками, расположенными в разных сегментах.

Если связать регистр **ESI** с регистром сегмента **ES** вместо используемого по умолчанию **DS**, строки источника и назначения могут быть расположены в одном и том же сегменте.

Это же самое может быть осуществлено загрузкой сегментных регистров **DS** и **ES** одним и тем же селектором сегмента и разрешения регистру **ESI** быть связанным с регистром **DS** по умолчанию.

После выполнения операции над элементом строки регистры **(E)SI** и **(E)DI** автоматически увеличиваются или уменьшаются в соответствии с установкой флага **DF** в регистре **EFLAGS**.

Если флаг **DF** равен 0, регистры **(E)SI** и **(E)DI** инкрементируются (**edi++**, **edi--**).

Если флаг **DF** равен 1, регистры **(E)SI** и **(E)DI** декрементируются (**edi--**, **esi--**).

Регистры изменяются на 1 для байтовых операций, на 2 для операций с словами или на 4 для операций с двойными словами.

Формы строковых инструкций

На уровне кода ассемблера допускаются две формы строковых инструкций — форма с явными операндами и форма без операндов.

Форма с явными операндами использует мнемонику **xxxS** и позволяет явно указывать исходный и целевой операнды:

MOVS m, m (MOVSB/MOVSW/MOVSD) — переместить данные из строки DS:ESI в строку ES:EDI;
LODS m (LODSB/LODSW/LODSD) — загрузить строку из DS:ESI в EAX;

При использовании явной формы операнды в памяти должны быть символами, которые определяют размер и указывают местоположение как источника, так и приемника.

Данная форма явных операндов предоставляется ради возможности самодокументирования кода. Однако, следует обратить внимание, что документирование, предоставляемое этой формой, может вводить в заблуждение — символы операндов, как источника, так и приемника должны указывать правильный тип (размер) операндов (байты, слова или двойные слова), но они не обязаны указывать правильное местоположение. Реальное расположение операндов строковых инструкций всегда указывается регистрами **DS:(E)SI** и **ES:(E)DI**.

Форма без операндов предоставляет «короткие формы» для строковых инструкций, ассоциированные с байтами, словами и двойными словами. Как и в случае формы с явным указанием оперндов данная форма также предполагает, что **DS:(E)SI** и **ES:(E)DI** указывают на операнды источники и приемники, соответственно.

Расположение операндов источника и назначения вне зависимости от использующейся формы всегда указывается регистрами `DS:(E)SI` и `ES:(E)DI`, которые должны быть правильно загружены перед выполнением команды перемещения строки.

Размер операнда источника и/или приемника выбирается с помощью мнемоники — **`xxxSB`** (операция с байтом), **`xxxSW`** (операция со словом) или **`xxxSD`** (операция с двойным словом). Короткие формы строковых инструкций:

Размер элемента строки	MOVS	CMPS	SCAS	LODS	STOS
байт	MOVSB	CMPSB	SCASB	LODSB	STOSB
слово	MOVSW	CMPSW	SCASW	LODSW	STOSW
двойное слово	MOVSD	CMPSD	SCASD	LODSD	STOSD

NASM не поддерживает формы строковых инструкций **`LODS`**, **`MOVS`**, **`STOS`**, **`SCAS`**, **`CMPS`**, **`INS`** или **`OUTS`**, – он поддерживает только формы с явным указанием размеров обрабатываемых элементов, такие, как **`xxxSB`**, **`xxxSW`**, **`xxxSD`** (**`LODSB`**, **`MOVSW`**, **`SCASD`**...).

Причина этого – NASM не «помнит» тип объявленных переменных. Он «помнит» только адрес начала символа. Поэтому нельзя записать **`mov [var],2`** – следует всегда явно указывать размер переменной, если его нельзя вывести из других операндов инструкции. В данном случае следует писать **`mov word [var],2`**. Это же касается любой инструкции с неявными размерами операндов.

MASM	NASM
`rep movs byte ptr es:[edi], byte ptr ds:[esi]` `rep movs byte ptr ds:[edi], byte ptr ds:[esi]`	`rep movs` `rep ds movs`

Повторяющиеся строковые операции

Каждая из строковых инструкций выполняет одну итерацию строковой операции.

Для работы со строками длиннее, чем двойное слово, строковые инструкции могут быть объединены с префиксом повторения (**REP**) с целью создания повторяющейся инструкции, либо помещены в цикл.

При использовании в строковых инструкциях регистры **ESI** и **EDI** автоматически увеличиваются или уменьшаются после каждой итерации, в результате чего они указывают на следующий элемент (байт, слово или двойное слово) в строке.

Строковые операции могут начинать итерации с более высоких адресов и двигаться к более низким, либо они могут начинать с более низких адресов и двигаться к более высоким.

Будут ли регистры **ESI** и **EDI** увеличиваться или уменьшаться определяет флаг **DF** (**D**irection **F**lag) в регистре **EFLAGS**. Установка флага **DF** в 0 приводит к инкременту регистра, установка в 1 — к декременту. Для установки или очистки этого флага используются, соответственно, инструкции **STD** или **CLD**.

Количество повторений строковых операций определяется содержимым регистра-счетчика **ECX**. После каждой итерации регистр счетчика уменьшается на 1.

Для автоматического декремента счетчика в регистре **ECX** следует использовать префиксы повторения:

REP — повторять, пока регистр **ECX** не станет равным нулю;

REPE/REPZ — повторять, пока флаг **ZF == 1** или регистр **ECX** не станет равным нулю.

REPNE/REPNZ — повторять, пока флаг **ZF == 0** или регистр **ECX** не станет равным нулю.

Когда строковая инструкция имеет префикс повторения, операция выполняется до тех пор, пока не будет выполнено одно из условий завершения, указанных префиксом.

Префиксы REPE/REPZ и REPNE/REPNZ используются только с инструкциями CMPS и SCAS.

Следует обратить внимание, что инструкция **REP STOS** — это самый быстрый способ инициализации большого блока памяти.

Быстрая строковая операция (FastString)

Чтобы повысить производительность, более поздние процессоры поддерживают модифицированный алгоритм выполнения операций сохранения строк, инициируемых инструкциями **MOVS**, **MOVSB**, **STOS** и **STOSB**. Если выполнение одной из этих инструкций удовлетворяет определенным начальным условиям, используется оптимизированная операция, называемая быстрой строковой операцией.

Инструкции, использующие быструю строковую операцию, эффективно работают со строкой группами, состоящими из нескольких элементов естественного размера данных (байт, слово, двойное слово или четырехзначное слово). При такой работе процессор распознает прерывания и точки останова данных только на границах между этими группами.

Начальные условия для работы с быстрой строкой зависят от реализации и могут варьироваться в зависимости от собственного размера строки.

Примеры параметров, которые могут повлиять на использование Faststring:

- выравнивание элементов в памяти, указанных в регистрах **EDI** и **ESI**;
- порядок адресов строковой операции;
- значение начального счетчика операций (**ECX**);
- разница между адресами источника и назначения.

MOVS/MOVSB/MOVSW/MOVSD — переместить данные из строки в строку

```
MOVS    m, m    ; 8, 16, 32
MOVSB
MOVSW
MOVSD
```

Операция

```
DEST (ES:EDI) ← SRC (Seg:ESI)
```

Описание

Перемещает байт, слово или двойное слово, указанное вторым операндом (операнд-источник), в место, указанное первым операндом (операнд-приемник).

Операнд-источник и операнд-приемник находятся в памяти.

Адрес операнда-источника считывается из регистров **`DS:ESI`** или **`DS:SI`** (в зависимости от атрибута размера адреса команды, 32 или 16, соответственно).

Адрес операнда-приемника считывается из регистров **`ES:EDI`** или **`ES:DI`** (опять же, в зависимости от атрибута размера адреса в инструкции).

Сегмент `DS` может быть переопределен.

Сегмент `ES` переопределен быть не может.

Флагов не изменяет

CMPS/CMPSB/CMPSW/CMPSD — сравнить строковые операнды

```
CMPS    m, m    ; 8, 16, 32
CMPSB
CMPSW
CMPSD
```

Операция

```
temp ← SRC - DEST
SetStatusFlags(temp);
```

Описание

CMPS вычитает элемент строки назначения из элемента строки источника и обновляет флаги состояния (**CF**, **ZF**, **OF**, **SF**, **PF** и **AF**) в регистре **EFLAGS** в соответствии с результатами. Ни один из элементов строки обратно в память не записывается.

Операнды находятся в памяти.

Адрес операнда-источника считывается из регистров **DS:ESI** или **DS:SI** (в зависимости от атрибута размера адреса команды, 32 или 16, соответственно).

Адрес операнда-приемника считывается из регистров **ES:EDI** или **ES:DI** (опять же, в зависимости от атрибута размера адреса в инструкции). Сегмент **DS** может быть переопределен.

Сегмент ES переопределен быть не может.

Флаги CF, ZF, OF, SF, PF и AF устанавливаются по результатам сравнения.

SCAS/SCASB/SCASW/SCASD — сканировать строку

```
SCAS    m    ; 8, 16, 32
SCASB
SCASW
SCASD
```

Операция

```
temp ← Acc - DEST
SetStatusFlags(temp);
```

Описание

SCAS вычитает элемент строки назначения из содержимого регистра **EAX**, **AX** или **AL** в зависимости от длины операнда и обновляет флаги состояния (**CF**, **ZF**, **OF**, **SF**, **PF** и **AF**) в регистре **EFLAGS** в соответствии с результатами.

Адрес операнда считывается из регистров **ES:EDI** или **ES:DI** (в зависимости от атрибута размера адреса команды, 32 или 16, соответственно).

Сегментный регистр ES переопределен быть не может.

Элемент строки и содержимое регистра не изменяются.

Флаги CF, ZF, OF, SF, PF и AF устанавливаются по результатам сравнения.

LODS/LODSB/LODSW/LODSD — загрузить строку

```
LODS    m    ; 8, 16, 32
LODSB
LODSW
LODSD
```

Операция

```
Acc ← SRC
```

Описание

LODS загружает элемент строки-источника, идентифицируемый регистром **ESI**, в регистр **EAX** (для строки из двойных слов), регистр **AX** (для строки из слов) или регистр **AL** (для строки из байтов).

Адрес операнда-источника считывается из регистров **DS:ESI** или **DS:SI** (в зависимости от атрибута размера адреса команды, 32 или 16, соответственно).

Сегмент DS может быть переопределен.

Эта инструкция обычно используется в цикле, где другие инструкции обрабатывают каждый элемент строки после их загрузки в регистр назначения.

Флагов не изменяет

STOS/STOSB/STOSW/STOSD — сохранить строку

```
STOS    m ; 8, 16, 32
STOSB
STOSW
STOSD
```

Операция

```
DEST ← Acc
```

Описание

STOS сохраняет элемент строки из регистра **EAX** (для строки из двойных слов), регистра **AX** (для строки из слов) или регистра **AL** (для строки из байтов) в ячейке памяти, идентифицированной с помощью пары регистров **ES:EDI**.

Сегментный регистр ES переопределен быть не может.

Эта инструкция также обычно используется в цикле. Строка обычно загружается в регистр с помощью инструкции **LODS**, обрабатывается другими инструкциями и затем снова сохраняется в памяти с помощью инструкции **STOS**.

Флагов не изменяет

Инструкции передачи управления

Команды передачи управления обеспечивают операции перехода, условного перехода, цикла, вызова и возврата с целью **управления потоком управления** (control program flow).

Процессор предоставляет как условные, так и безусловные инструкции передачи управления для управления потоком выполнения программы.

Инструкции передачи управления обычно подразделяются на:

- инструкции безусловной передачи управления **`JMP`**, **`CALL`**, **`RET`**, **`INT`** и **`IRET`**;
- инструкции условной передачи управления **`Jcc`**, **`JCXZ`**, **`JECXZ`**, **`LOOP`**;
- инструкции программных прерываний **`INT`**, **`INTO`**, **`INT3`**.

Инструкции безусловной передачи управления выполняются всегда и передают управление программой в другое место (**адрес назначения**) потока команд. Адрес назначения может находиться в одном и том же сегменте кода (**ближняя передача**) или в другом сегменте кода (**дальняя передача**).

Инструкции условной передачи управления выполняют переходы (**`Jcc`**) или циклы (**`LOOP`**), которые передают управление программой другой инструкции в потоке команд только в том случае, если выполнены определенные условия.

Инструкция безусловного перехода JMP

Инструкция **JMP** в безусловном порядке передает управление программой инструкции по адресу назначения. Переход осуществляется в одну сторону, то есть обратный адрес не сохраняется. Адрес инструкции назначения указывается операндом. Этот операнд может быть непосредственным значением, регистром общего назначения или ячейкой памяти.

```
JMP    rel8       ; относительный короткий
JMP    rel16      ; относительный ближний
JMP    rel32      ; относительный ближний
JMP    r/m16      ; абсолютный ближний косвенный
JMP    r/m32      ; абсолютный ближний косвенный
JMP    ptr16:16   ; абсолютный дальний прямой (адрес в операнде)
JMP    ptr16:32   ; абсолютный дальний прямой (адрес в операнде)
JMP    m16:16     ; абсолютный дальний косвенный (адрес в ОП)
JMP    m16:32     ; абсолютный дальний косвенный (адрес в ОП)
```

Эта инструкция может использоваться для выполнения четырех разных типов переходов:

- ближний переход (near jump) — переход к инструкции в текущем сегменте кода (сегмент, на который в данный момент указывает **CS**), иногда называемый внутрисегментным переходом;

- короткий переход (short jump) — ближний переход, при котором диапазон перехода ограничен от –128 до +127 от текущего значения **EIP**;

- дальний переход (far jump) — переход к инструкции, расположенной в другом сегменте, отличном от текущего сегмента кода, но на том же уровне привилегий, иногда называемый межсегментным переходом;

- переключатель задачи (task switch) — переход к инструкции, находящейся в другой задаче.

Адрес назначения может быть относительным или абсолютным.

Относительный адрес — это смещение (offset) относительно адреса в регистре **EIP**.

Адрес назначения (near pointer — ближний указатель) формируется путем добавления смещения к адресу в регистре **EIP**.

Смещение указывается целым числом со знаком, что позволяет переходить вперед или назад в потоке команд.

Абсолютный адрес — это смещение (offset) от базового адреса сегмента (0).

Абсолютный адрес можно указать одним из следующих способов:

- адрес в регистре общего назначения. Этот адрес рассматривается как ближний указатель (near pointer), который копируется в регистр **EIP**. Затем выполнение программы продолжается по новому адресу в текущем сегменте кода.

- адрес, указанный с использованием стандартных режимов адресации процессора. В этом случае адрес может быть как ближним (near), так и дальним (far) указателем.

Если адрес используется в качестве ближнего указателя, он преобразуется в смещение и копируется в регистр **EIP**.

Если адрес используется в качестве дальнего указателя, адрес преобразуется в селектор сегмента, который копируется в регистр **CS**, и смещение, которое копируется в регистр **EIP**.

В защищенном режиме инструкция **JMP** также позволяет переходить к шлюзу вызова, шлюзу задачи и сегменту состояния задачи.

Близкие и короткие переходы

При выполнении близкого перехода процессор переходит к адресу (в текущем сегменте кода), который указан в целевом операнде. Целевой операнд указывает либо абсолютное смещение (то есть смещение от основания сегмента кода), либо относительное смещение (смещение со знаком относительно текущего значения указателя инструкции в регистре EIP). Ближний переход к относительному смещению 8 бит (rel8) называется коротким переходом. При близких и коротких переходах регистр CS не меняется.

Абсолютное смещение указывается косвенно в регистре общего назначения или в ячейке памяти (r/m16 или r/m32).

Размер целевого операнда (16 или 32 бита) определяется атрибутом размера операнда.

Абсолютные смещения загружаются непосредственно в регистр EIP.

Если атрибут размера операнда равен 16, верхние два байта регистра EIP очищаются, в результате чего максимальный размер указателя инструкции равен 16 битам.

Относительное смещение (rel8, rel16 или rel32) обычно указывается как метка в ассемблерном коде, но на уровне машинного кода оно кодируется как 8-, 16- или 32-битное непосредственное значение со знаком.

Это значение добавляется к значению в регистре EIP (в этом случае регистр EIP содержит адрес инструкции, следующей за инструкцией JMP).

При использовании относительных смещений размер целевого операнда (8, 16 или 32 бита) определяется кодом операции (для коротких и близких переходов) и атрибутом размера операнда (для близких относительных переходов).

(?) Дальние переходы (Far Jumps) в режиме реального адреса или виртуального 8086

При выполнении дальнего перехода в режиме реального адреса или виртуального 8086 процессор переходит к сегменту кода и смещению, который задан в целевом операнде.

В этом случае целевой операнд указывает абсолютный дальний адрес либо напрямую с помощью указателя (ptr16:16 или ptr16:32), либо косвенно с помощью ячейки памяти (m16:16 или m16:32).

При использовании метода указателя сегмент и адрес вызываемой процедуры кодируются в инструкции с использованием 4-байтового (16-битный размер операнда) или 6-байтового (32-битный размер операнда) дальнего адреса.

При косвенном методе целевой операнд указывает область памяти, которая содержит 4-байтовый (16-битный размер операнда) или 6-байтовый (32-битный размер операнда) дальний адрес. Дальний адрес загружается непосредственно в регистры CS и EIP. Если атрибут размера операнда равен 16, верхние два байта регистра EIP очищаются.

(?) Дальние переходы (Far Jumps) в защищенном режиме

Когда процессор работает в защищенном режиме, инструкция JMP может использоваться для выполнения следующих трех типов дальних переходов:

- дальний переход к конформному или неконформному сегменту кода;
- дальний переход через шлюз вызова (call gate);
- переключение задачи (task switching).

Инструкцию JMP нельзя использовать для выполнения дальних переходов между различными уровнями привилегий.

В защищенном режиме процессор всегда использует сегментную часть дальнего адреса (селектор сегмента) для доступа к соответствующему дескриптору в GDT или LDT.

Тип выполняемого перехода определяется типом дескриптора (сегмент кода, шлюз вызова, шлюз задачи или TSS) и правами доступа.

Если выбранный дескриптор предназначен для сегмента кода, выполняется дальний переход в сегмент кода с тем же уровнем привилегий.

Если выбранный сегмент кода находится на другом уровне привилегий и сегмент кода не является конформным, генерируется исключение общей защиты.

(?) Дальний переход к тому же уровню привилегий в защищенном режиме очень похож на тот, который выполняется в режиме реальной адресации или в режиме виртуального-8086.

Целевой операнд указывает абсолютный дальний адрес либо напрямую с помощью указателя (ptr16:16 или ptr16:32), либо косвенно с помощью ячейки памяти (m16:16 или m16:32).

Размер смещения (16 или 32 бита) в случае дальнего адреса определяется атрибутом размера операнда.

Селектор нового сегмента кода и его дескриптор загружаются в регистр CS, а в регистр EIP загружается смещение от инструкции, следующей за инструкцией JMP.

Следует обратить внимание, что для выполнения удаленного вызова в сегмент кода с тем же уровнем привилегий также может использоваться и шлюз вызова.

Использование этого механизма обеспечивает дополнительный уровень косвенности и является предпочтительным методом перехода между 16-битными и 32-битными сегментами кода.

При выполнении дальнего перехода через шлюз вызова селектор сегмента, указанный целевым операндом, идентифицирует шлюз вызова. При этом смещение целевого операнда игнорируется. Процессор переходит к сегменту кода, указанному в дескрипторе шлюза вызова, и начинает выполнение инструкции со смещением, которое указано в шлюзе вызова. Переключение стека при этом не происходит.

В этом случае целевой операнд также может указывать дальний адрес шлюза вызова либо напрямую с помощью указателя (ptr16:16 или ptr16:32), либо косвенно с помощью ячейки памяти (m16: 16 или m16: 32).

Выполнение переключения задач с помощью инструкции JMP в чем-то похоже на выполнение перехода через шлюз вызова. В этом случае целевой операнд указывает селектор сегмента шлюза задачи для задачи, на которую выполняется переключение (и часть смещения целевого операнда игнорируется). Шлюз задачи, в свою очередь, указывает на TSS для задачи, который содержит селекторы сегментов для кода задачи и сегменты стека. TSS также содержит значение EIP для следующей инструкции, которая должна была быть выполнена до того, как задача была приостановлена. Это значение указателя инструкции загружается в регистр EIP, так что задача снова начинает выполняться с этой следующей инструкции.

Инструкция JMP может указывать селектор сегментов TSS напрямую, что исключает косвенное обращение к шлюзу задачи.

Следует обратить внимание, что при переключении задачи с помощью инструкции JMP флаг вложенной задачи (NT) в регистре EFLAGS не устанавливается, а поле ссылки на предыдущую задачу нового TSS не загружается с помощью селектора TSS старой задачи. Таким образом, возврат к предыдущей задаче не может быть осуществлен путем выполнения инструкции IRET.

Переключение задач с помощью инструкции JMP отличается в этом отношении от инструкции CALL, которая устанавливает флаг NT и сохраняет информацию о предыдущей ссылке на задачу, позволяя вернуться к вызывающей задаче с помощью инструкции IRET.

Порядок исполнения инструкций

Команды, следующие за дальним переходом, могут быть выбраны из памяти на выполнение до завершения выполнения более ранних инструкций, но выполняться они (даже спекулятивно) не будут до тех пор, пока все инструкции, предшествующие дальнему переходу, не завершат выполнение.

Тем не менее, до того, как данные, сохраненные более ранними инструкциями, станут глобально видимыми могут выполняться более поздние инструкции.

Определенные ситуации могут привести к спекулятивному выполнению следующей последовательной инструкции после близкого косвенного JMP. Если программному обеспечению необходимо предотвратить это, например, чтобы предотвратить побочный канал спекулятивного исполнения, то после близкого косвенного JMP с целью заблокировать спекулятивное выполнение может быть помещен код операции инструкции INT3 или LFENCE.

Флаги

Если происходит переключение задачи, все флаги изменяются.

Если переключение задачи не происходит, никакие флаги не изменяются

Инструкции вызова и возврата из процедуры CALL/RET

Инструкции **CALL** (процедура вызова) и **RET** (возврат из процедуры) позволяют переходить от одной процедуры (или подпрограммы) к другой с последующим возвратом назад к вызывающей процедуре.

Синтаксис

```
CALL    rel8     ; относительный короткий
CALL    rel16    ; относительный ближний
CALL    rel32    ; относительный ближний
CALL    r/m16    ; абсолютный ближний косвенный
CALL    r/m32    ; абсолютный ближний косвенный
CALL    ptr16:16 ; абсолютный дальний прямой (адрес в операнде)
CALL    ptr16:32 ; абсолютный дальний прямой (адрес в операнде)
CALL    m16:16   ; абсолютный дальний косвенный (адрес в ОП)
CALL    m16:32   ; абсолютный дальний косвенный (адрес в ОП)
```

Инструкция **CALL** передает управление программой из текущей (или вызывающей процедуры) в другую процедуру (вызываемая процедура).

Чтобы разрешить последующий возврат к вызывающей процедуре, инструкция **CALL** перед переходом к вызываемой процедуре сохраняет текущее содержимое регистра **EIP** в стеке.

Регистр **EIP** (до передачи управления) содержит адрес инструкции, следующей за инструкцией **CALL**.

Когда этот адрес помещается в стек, он называется указателем инструкции возврата или **адресом возврата**.

Адрес вызываемой процедуры (адрес первой инструкции в процедуре, на которую выполняется переход) указывается в инструкции **CALL** аналогично тому, как это указывается инструкции **JMP**.

Адрес может быть указан как *относительный* адрес или *абсолютный*. Если указан абсолютный адрес, он может быть ближним или дальним указателем.

Инструкция RET

Инструкция **RET** передает управление программой из выполняемой в данный момент процедуры (вызываемой процедуры) обратно в вызвавшую ее процедуру (вызывающую процедуру).

Передача управления осуществляется путем копирования указателя инструкции возврата из стека в регистр **EIP**, после чего выполнение программы продолжается инструкцией, указанной регистром **EIP**.

Инструкция **RET** имеет необязательный операнд, значение которого добавляется к содержимому регистра **ESP** в рамках операции возврата. Этот операнд позволяет увеличивать указатель стека с целью удаления параметров, которые были помещены в стек вызывающей процедурой.

Инструкцию **RET** можно использовать для выполнения трех различных типов возвратов:

- ближний возврат — возврат к вызывающей процедуре в текущем сегменте кода (сегмент, на который в настоящее время указывает **CS**), иногда называемый внутрисегментным возвратом;

- дальний возврат — возврат к вызывающей процедуре, находящейся в сегменте, отличном от текущего сегмента кода, иногда называемый межсегментным возвратом;

- дальний возврат между уровнями привилегий — дальний возврат к уровню привилегий, отличному от привиелгий выполняющейся в данный момент программы или процедуры.

Инструкция возврата из прерывания IRET

Когда процессор обрабатывает прерывание, он выполняет неявный вызов процедуры обработки прерывания. Инструкция **IRET** (возврат из прерывания) возвращает управление программой от обработчика прерывания к прерванной процедуре (то есть процедуре, которая выполнялась, когда произошло прерывание).

Инструкция **IRET** выполняет операцию, аналогичную инструкции **RET**, за исключением того, что она также восстанавливает регистр **EFLAGS** из стека. Содержимое регистра **EFLAGS** автоматически сохраняется в стеке вместе с указателем инструкции возврата, когда процессор обслуживает прерывание.

Вызов процедур с использованием CALL и RET

Синтаксис

```
CALL   rel      ; 8, 16, 32 ближний, относительно следующей инструкции
CALL   r/m      ; 16, 32 ближний, абсолютный косвенный
CALL   ptr16:16 ; дальний, абсолютный прямой (адрес в операнде)
CALL   ptr16:32 ; дальний, абсолютный прямой (адрес в операнде)
CALL   m16:16   ; дальний, абсолютный косвенный (адрес в памяти)
CALL   m16:32   ; дальний, абсолютный косвенный (адрес в памяти)
RET
RET    n
```

CALL позволяет передавать управление процедурам как в текущем сегменте кода (ближний вызов), так и в другом сегменте кода (удаленный вызов).

Ближние вызовы обычно используются для доступа к локальным процедурам в рамках текущей выполняющейся программы или задачи.

Удаленные вызовы обычно используются для доступа к процедурам операционной системы или процедурам в другой задаче.

Инструкция **RET** обеспечивает ближний и дальний возврат, соответствующие ближнему и дальнему вариантам инструкции **CALL**. Кроме того, инструкция **RET** позволяет программе увеличивать указатель стека при возврате, с целью освободить стек от параметров. Количество байтов, освобождаемых в стеке, определяется необязательным аргументом (n) инструкции RET.

Как работает

CALL сохраняет информацию о возврате из процедуры в стеке и выполняет переход к вызываемой процедурой, указанной с помощью операнда-назначения.

Операнд указывает адрес первой инструкции в вызываемой процедуре.

Операндом может быть непосредственное значение, регистр общего назначения или ячейка памяти.

Инструкция **CALL** может использоваться для выполнения четырех типов вызовов:

Ближний вызов — вызов процедуры в текущем сегменте кода (сегмент, на который в данный момент указывает регистр **CS**), иногда называемый *внутрисегментным вызовом*.

Дальний вызов — вызов процедуры, расположенной в другом сегменте, иногда называемый *межсегментным вызовом*.

Дальний вызов со сменой уровня привилегий — дальний вызов процедуры в сегменте с уровнем привилегий, отличным от уровня выполняемой в данный момент программы или процедуры.

Переключение задач — вызов процедуры, расположенной в другой задаче.

Последние два типа вызовов (вызов со сменой уровня привилегий и переключение задач) могут выполняться только в защищенном режиме.

Ближний вызов: детали

При выполнении ближнего вызова процессор помещает значение регистра **EIP**, который содержит смещение инструкции, следующей за инструкцией **CALL**, в стек для последующего использования его в качестве указателя для инструкции возврата. Затем процессор переходит к адресу в текущем сегменте кода, указанному целевым операндом.

Целевой операнд указывает либо абсолютное смещение в сегменте кода — смещение от базового адреса сегмента кода, либо относительное смещение — смещение со знаком относительно текущего значения указателя команды в регистре **EIP**, указывающего на инструкцию, следующую за инструкцией **CALL**.

При ближних вызовах регистр **CS** не изменяется.

В случае абсолютного ближнего вызова абсолютное смещение указывается косвенно в регистре общего назначения или в ячейке памяти (**r/m16** или **r/m32**). Размер целевого операнда (16 или 32 бита) определяется атрибут размера операнда.

Абсолютное смещение загружается непосредственно в регистр **EIP**.

Если атрибут размера операнда равен 16, верхние два байта регистра **EIP** очищаются, в результате чего максимальный размер указателя команд составляет 16 битов. При обращении по абсолютному смещению, косвенно использующему указатель стека **ESP** в качестве базового регистра, используемым значением базы является значение **ESP** до выполнения инструкции.

Относительное смещение (**`rel16`** или **`rel32`**) обычно указывается как метка в коде ассемблера, но на уровне машинного кода оно кодируется как 16-битное или 32-битное непосредственное значение со знаком. Оно добавляется к значению в регистре **`EIP`**.

Как и в случае абсолютных смещений, размер целевого операнда (16 или 32 бита), определяется атрибутом размера операнда.

Дальние вызовы в режиме реального адреса или виртуального 8086

При выполнении удаленного вызова в режиме реального адреса или виртуального 8086 процессор помещает текущее значение обоих регистров **`CS`** и **`EIP`** в стек для использования в качестве указателя инструкции возврата. Затем процессор выполняет «far переход», используя сегмент кода и смещение, указанные в качестве целевого операнда для вызываемой процедуры.

Целевой операнд указывает абсолютный дальний адрес либо **напрямую**, используя указатель (**`ptr16:16`** или **`ptr16:32`**), либо **косвенно**, указывая ячейку памяти (**`m16:16`** или **`m16:32`**).

В случае использования прямого метода сегмент и смещение вызываемой процедуры кодируются в инструкции с использованием 4-байтового (16-битного размера операнда) или 6-байтового (32-битного размера операнда) непосредственного адреса.

В случае использования косвенного метода целевой операнд определяет ячейку памяти, которая содержит дальний адрес 4-байтового (размер 16-битного операнда) или 6-байтового (размер 32-битного операнда).

Размер смещения (16, 32) определяется атрибутом размера операнда.

Дальний адрес загружается непосредственно в регистры **CS** и **EIP**. Если атрибут размера операнда равен 16, верхние два байта регистра **EIP** очищаются.

Дальние вызовы в защищенном режиме

Когда процессор работает в защищенном режиме, инструкция **CALL** может использоваться для выполнения следующих типов удаленных вызовов:

- дальний вызов на тот же уровень привилегий;
- дальний вызов на другой уровень привилегий (со сменой уровня привилегий);
- переключение задач (дальний вызов другой задачи).

В защищенном режиме процессор всегда использует соответствующую часть селектора сегмента дальнего адреса для доступа к соответствующему дескриптору в **GDT** или **LDT**. Тип операции вызова, которая должна быть выполнена, определяются типом дескриптора (сегмент кода, шлюз вызова, шлюз задач или TSS) и правами доступа.

Если выбранный дескриптор является сегментом кода, выполняется дальний вызов к сегменту кода с тем же уровнем привилегий.

Если выбранный сегмент кода находится на другом уровне привилегий, а сегмент кода не является конформным, генерируется исключение общей защиты.

Дальний вызов на одном уровне привилегий в защищенном режиме очень похож на вызов, выполняемый в режиме реальной адресации или в виртуальном режиме 8086. Целевой операнд указывает абсолютный дальний адрес либо напрямую, используя указатель (**ptr16:16**, **ptr16:32**), либо косвенно, указывая ячейку памяти (**m16:16**, **m16:32**). Размер смещения (16 или 32 бита) для дальнего адреса определяет ся атрибутом размера операнда. Новые селектор сегмента кода и его дескриптор загружаются в регистр **CS**, а смещение — в регистр **EIP**.

Для выполнения удаленного вызова к сегменту кода с тем же уровнем привилегий также может использоваться **шлюз вызова**, что обеспечивает дополнительный уровень косвенности и является предпочтительным методом выполнения вызовов между 16-битным и 32-битным сегментами кода.

Дальние вызовы в режиме совместимости

Когда процессор работает в режиме совместимости, инструкция **CALL** может использоваться для выполнения следующих типов удаленных вызовов:

- дальний вызов на тот же уровня привилегий, оставаясь в режиме совместимости;
- дальний вызов на тот же уровня привилегий с переходом в 64-битный режим;
- дальний вызов на другой уровень привилегий (вызов с изменением уровня привилегий) с переходом в 64-битный режим.

Флаги

Флаги затрагиваются только в случае при переключении задачи.

Инструкции условной передачи управления

Инструкции условной передачи управления выполняют переходы (**Jcc**) или циклы (**LOOP**), которые передают управление программой другой инструкции в потоке команд только в том случае, если выполнены определенные условия.

Условия для передачи управления задаются набором кодов условий, которые определяются различными состояниями флагов (**CF**, **ZF**, **OF**, **PF** и **SF**) в регистре **EFLAGS**.

Инструкции условного перехода **Jcc** передают управление программой инструкции назначения, если выполняются условия, указанные с помощью кода условия (**cc**), из приведенных в таблице. Все режимы адресации инструкций **Jcc** — относительные.

Если условие не выполняется, выполнение продолжается с инструкции, следующей за инструкцией **Jcc**.

Как и в случае с инструкцией **JMP**, передача является односторонней, то есть обратный адрес (адрес возврата) не сохраняется.

Синтаксис

```
Jcc    rel8  ; кроме JECXZ
Jcc    rel16 ; кроме JECXZ, JCXZ
Jcc    rel32 ; кроме JECXZ, JCXZ
```

Операнд-адресат указывает **относительный** адрес (смещение со знаком относительно адреса в регистре **EIP**), который указывает на целевую инструкцию в текущем сегменте кода.

Относительное смещение (rel8, rel16 или rel32) обычно указывается как метка в коде ассемблера, но на уровне машинного кода оно кодируется как 8-битное или 32-битное непосредственное значение со знаком, которое добавляется к указатель инструкции.

Кодирование инструкций наиболее эффективно для смещений от –128 до +127. Если атрибут размера операнда равен 16, верхние два байта регистра **EIP** очищаются, в результате чего максимальный размер указателя команды составляет 16 бит.

```
first   dd   ?
secnd   dd   ?
...
        push  secnd
        push  first
        call  foo   ; int foo(first, secnd)
        jnz   L1    ;
        jmp   error ; процедура завершения по ошибке
L1:     ....
```

Мнемоника инструкции формируется из символа **J** (**J**ump) и мнемоники кода проверяемого условия. Инструкции делятся на две группы — условные переходы без знака и со знаком. Эти группы соответствуют результатам операций, выполняемых с целыми числами без знака и со знаком, соответственно.

Инструкции, перечисленные в виде пар (например, **JA/JNBE**), являются альтернативными именами для одной и той же инструкции. Ассемблеры предоставляют альтернативные имена, чтобы упростить чтение листингов программ.

Инструкции **JCXZ** и **JECXZ** проверяют регистры **CX** и **ECX**, соответственно, вместо одного или нескольких флагов состояния.

Знаковые условные переходы

JG/JNLE	ZF=0 & SF=OF	**G**reater / **N**ot (**L**ess or **E**qual)	больше / не (меньше или равно)
JGE/JNL	SF=OF	**G**reater or **E**qual / **N**ot **L**ess	больше или равно / не меньше
JL/JNGE	SF≠OF	**L**ess / **N**ot **G**reater or **E**qual	меньше / не (больше или равно)
JLE/JNG	ZF=1 | SF≠OF	**L**ess or **E**qual / **N**ot **G**reater	меньше или равно / не больше
JNO	OF=0	**N**o **O**verflow	отсутствие переполнения
JNS	SF=0	**N**o **S**ign (non-negative)	отсутствие знака
JO	OF=1	**O**verflow	переполнение
JS	SF=1	**S**ign (negative)	знак

Беззнаковые условные переходы

JA/JNBE	CF=0 & ZF=0	**A**bove / **N**ot (**B**elow or **E**qual)	выше / не (ниже или равно)
JAE/JNB	CF=0	**A**bove or **E**qual / **N**ot **B**elow	выше или равно / не ниже
JB/JNAE	CF=1	**B**elow / **N**ot (**A**bove or **E**qual)	ниже/ не (выше или равно)
JBE/JNA	CF=1 | ZF=1	**B**elow or **E**qual / **N**ot **A**bove	ниже или равно / не выше
JC	CF=1	**C**arry	перенос
JE/JZ	ZF=1	**E**qual / **Z**ero	равно / ноль
JNC	CF=0	**N**o **C**arry	отсутствие переноса
JNE/JNZ	ZF=0	**N**ot **E**qual / **N**ot **Z**ero	не равно / не ноль
JNP/JPO	PF=0	**N**o **P**arity / **P**arity **O**dd	отсутствие паритета / нечетно
JP/JPE	PF=1	**P**arity / **P**arity **E**ven	паритет / четно
JCXZ	CX=0	Register **CX** is **Z**ero	в регистре CX ноль
JECXZ	ECX=0	Register **ECX** is **Z**ero	в регистре ECX ноль

Детали использования

Инструкция **Jcc** не поддерживает дальний переход (переход в другие сегменты кода), однако, дальние передачи могут быть выполнены с помощью комбинации **Jcc** и инструкции **JMP**.

Если целевая инструкция для условного перехода находится в другом сегменте, следует использовать условие, противоположное условию, которое тестируется инструкцией **Jcc**, после чего можно получить доступ к целевому адресу с помощью безусловного перехода в (**JMP**) в другой сегмент. Например, следующий условный дальний переход недопустим:

```
        jz FarLabel
```

Чтобы выполнить этот дальний переход, следует использовать следующие две инструкции:

```
        jnz .beyond
        jmp FarLabel
.beyond:
```

LOOP / LOOPcc — цикл по счетчику ECX

Инструкции **LOOP**, **LOOPE** (цикл, пока равно), **LOOPZ** (цикл, пока ноль), **LOOPNE** (цикл, пока не равно) и **LOOPNZ** (цикл, пока не ноль) являются инструкциями условного перехода, которые используют значение регистра **ECX** в качестве счетчика количества итераций цикла.

Синтаксис

```
LOOP            rel8
LOOPE/LOOPZ     rel8
LOOPNE/LOOPNZ   rel8
```

Все инструкции цикла декрементируют содержимое счетчика в регистре **ECX** при каждом их выполнении и завершают цикл при достижении счетчиком **ECX** нуля.

Команды **LOOPE**, **LOOPZ**, **LOOPNE** и **LOOPNZ** используют флаг **ZF** в качестве условия для завершения цикла до того, как счетчик достигнет нуля.

Инструкция **LOOP** сначала уменьшает содержимое регистра **(E)CX** (атрибут *address-size сегмента*), и только потом проверяет регистр на условие завершения цикла.

Если счетчик в регистре **ECX** не равен нулю, управление программой передается на адрес инструкции, указанный в операнде. Операндом является относительный адрес (то есть смещение относительно содержимого регистра **EIP**), и он обычно указывает на первую инструкцию в блоке кода, который должен быть выполнен в цикле.

Когда счетчик в регистре ECX достигает нуля, управление программой передается инструкции, следующей сразу за инструкцией **LOOP**, которой завершается цикл.

Если при первом выполнении команды **LOOP** содержимое регистра **ECX** равно нулю, регистр предварительно будет декрементирован до FFFFFFFFH, в результате чего цикл выполнится 2^{32} раза.

Инструкции **LOOPE** и **LOOPZ** выполняют одну и ту же операцию (они являются разными мнемониками для одной и той же инструкции). Эти инструкции работают так же, как инструкция **LOOP**, за исключением того, что они также проверяют флаг **ZF**.

Если содержимое счетчика в регистре **ECX** не равно нулю и установлен флаг **ZF**, управление программой передается операнду-адресату.

Когда счетчик достигает нуля или флаг **ZF** будет сброшен, цикл прерывается путем передачи управления программой инструкции, которая следует непосредственно за инструкцией **LOOPE/LOOPZ**.

Инструкции **LOOPNE** и **LOOPNZ** (две мнемоники для одной и той же инструкции) работают так же, как инструкции **LOOPE/LOOPPZ**, за исключением того, что они завершают цикл до обнуления **ECX**, если флаг **ZF** будет установлен.

Инструкция Jump if zero

Инструкции **JECXZ** и **JCXZ** отличаются от других инструкций **Jcc**, поскольку они не проверяют флаги состояния. Вместо этого они проверяют **ECX** или **CX** на 0.

Проверяемый регистр определяется атрибутом **address-size**.

Эта инструкция может использоваться в сочетании с инструкцией цикла (**LOOP**, **LOOPE**, **LOOPZ**, **LOOPNE**, **LOOPNZ**) для проверки регистра **ECX** перед началом цикла с целью предотвращения входа в цикл, если **ECX** или **CX** равен 0.

Как было описано ранее, инструкции цикла уменьшают содержимое регистра **ECX** перед проверкой на ноль. Если значение в регистре **ECX** изначально равно нулю, оно будет уменьшено до FFFFFFFFH в первой инструкции цикла, в результате чего цикл будет выполнен 2^{32} раза. Чтобы предотвратить эту проблему, в начало блока кода для цикла может быть вставлена инструкция **JECXZ**, вызывая обход цикла, если счетчик — регистр **ECX** изначально равен нулю.

При использовании с повторяющимися инструкциями сканирования/сравнения строк, команда **JECXZ** может определить, завершился ли цикл по причине достижения счетчиком нуля или потому что были выполнены условия сканирования/сравнения.

Все условные переходы преобразуются в выборки кода размером в одну или две строки кэша независимо от адреса перехода или возможностей кэширующей подсистемы.

Примеры

1) Цикл for()

C

```
...
for(int i = 0; i < 5 ; i++) {
    printf(".");       // тело цикла
}
printf("\n");
...
```

asm

```
    ...
            mov    ecx, 5   ; инициализация счетчика
l1:
            PUTCHAR "."     ; тело цикла
            loop   l1
over_loop:
            PUTCHAR 10      ; \n
            ...             ; продолжение прграммы
```

2) счетчик может быть случайно проинициализирован нулем

C

```
#include <stdlib.h> // <cstdlib>
    ...
    size_t cnt;     // счетчик
    ...
    for (size_t i = 0; i < cnt; i++) {
        printf(".");
    }
    printf("\n");
```

asm

```
            ...
            mov    ecx, ebx ; инициализация счетчика
l1:
            PUTCHAR "."     ; если ECX = 0 тело цикла выполнится FFFFFFFF раз
            loop   l1
over_loop:
            PUTCHAR 10      ; \n
            ...             ; продолжение прграммы
```

```
            ...
            mov    ecx, ebx  ; инициализация счетчика
            jcxz   over_loop ; если ECX = 0 тело цикла будет обойдено
l1:
            PUTCHAR "."      ;
            loop   l1
            PUTCHAR 10       ; \n
over_loop:
            ...              ; продолжение прграммы
```

3) Вечный цикл

```
while(1) { // warning !!! некоторые компиляторы могут ругаться
    ...
    if (условие) break;
    ...
}
...
for(;;) { // правильная форма вечного цикла
    ...
    if (условие) break;
    ...
}

CRITICAL_VALUE equ 12345678h
            ...
loop_start:
            ...       ; eax
            cmp    eax, CRITICAL_VALUE
            je     behind
            ...
            jmp    loop_start
behind:
```

3) Конструкция while()

```
while (condition) {
    тело
}
```

```
            ...
while_loop: call   condition ; eax
            cmp    eax, 0
            je     behind    ; eax = 0 (false)
            ...              ; тело цикла
            jmp    while_loop
behind:
```

или

```
while_loop: call   condition ; eax
            cmp    eax, 0    ;
            jne    .L1       ; eax != 0 (true)
            jmp    behind    ; eax = 0 (false)
.L1:
            ...              ; тело цикла
            jmp    while_loop
behind:
```

4) Конструкция do...while()

```
do {
    тело
} while (condition)
```

```
            ...
while_loop:
            ...                 ; тело цикла
            call   condition    ; eax
            cmp    eax, 0
            jne    while_loop ; eax != 0 (true)
behind:
```

или

```
while_loop:
            ...                 ; тело цикла
            call   condition    ; eax
            cmp    eax, 0
            je     behind       ; eax = 0 (false)
            jmp    while_loop
behind:
```

Прерывания и исключения

Прерывание (Interrupt) — сигнал, сообщающий процессору о наступлении какого-либо события. При этом выполнение текущей последовательности команд приостанавливается, и управление передаётся обработчику прерывания, который реагирует на событие и обслуживает его, после чего возвращает управление в прерванный код.

Прерывания — неотъемлемая часть любой вычислительной системы. Без прерываний операционная система не сможет работать с внешними устройствами и защищать свои данные от приложений.

Процессор генерирует прерывания:

1) при нарушениях защиты;

2) при получении сигналов от контроллера прерываний, который принимает сигналы от внешних и системных устройств.

При генерации прерывания процессором текущая программа прерывается (отсюда и название), при этом сохраняются адрес следующей инструкции и регистр флагов, а управление передаётся обработчику. После своей работы обработчик возвращает управление прерванной программе, и её выполнение возобновляется.

В зависимости от источника возникновения сигнала прерывания делятся на:

- асинхронные, или внешние (аппаратные),
- синхронные, или внутренние;
- программные (частный случай внутреннего прерывания).

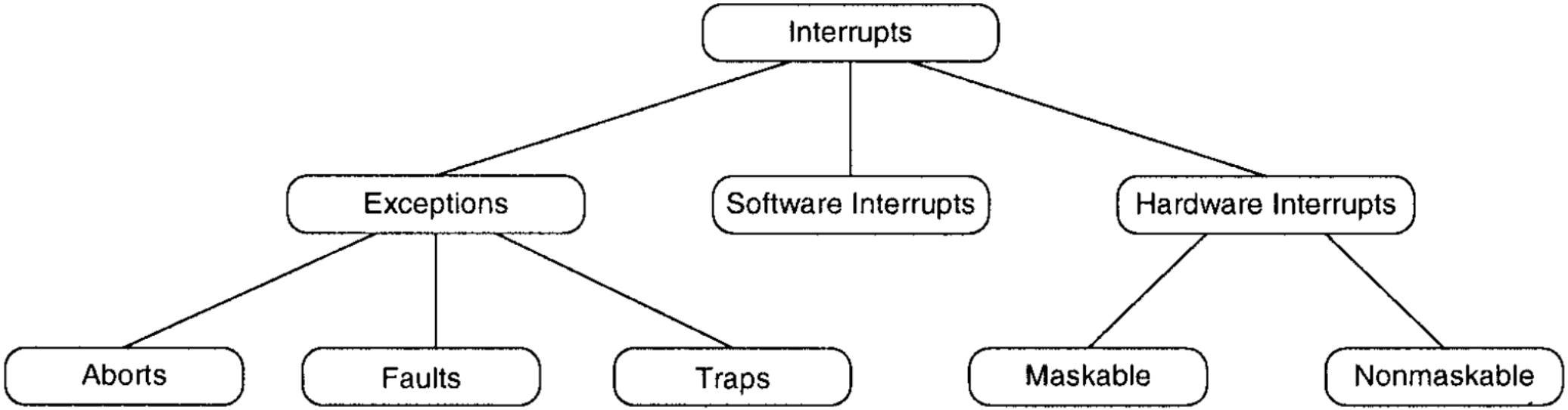
асинхронные — события, которые исходят от внешних источников и могут произойти в любой произвольный момент — сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши. Факт возникновения в системе такого прерывания трактуется как запрос на прерывание **(Interrupt request, IRQ)**;

синхронные — события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода — деление на ноль или переполнение стека, обращение к недопустимым адресам памяти или недопустимый код операции;

программные — инициируются исполнением специальной инструкции в коде программы. Программные прерывания, как правило, используются для обращения к функциям встроенного программного обеспечения, драйверов и операционной системы.

Систематика прерываний IA-32

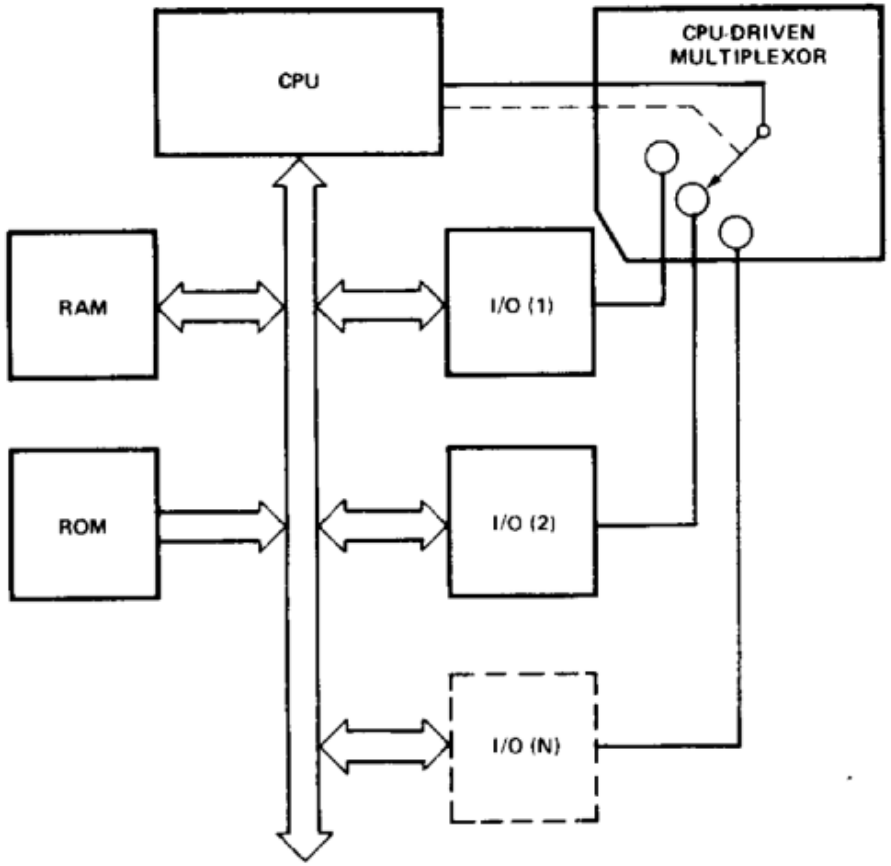
Прерывания бывают трёх видов:
1) аппаратные или внешние (Hardware Interrupts);
2) программные (Software Interrupts);
3) исключения (Exceptions).



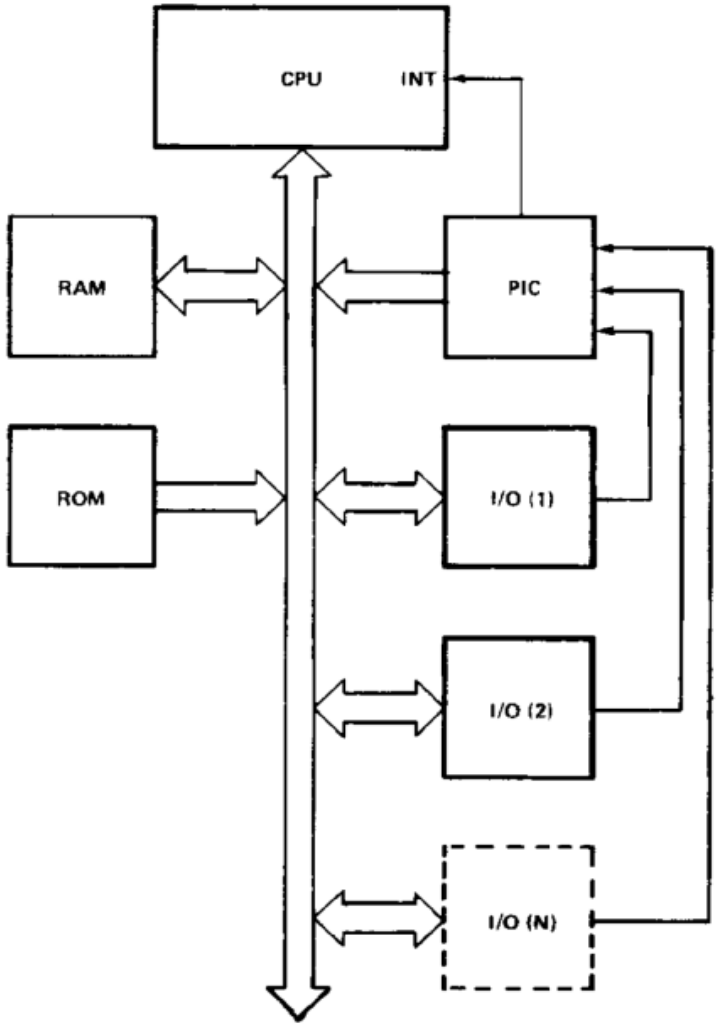
Aborts — Аварии;
Faults — Ошибки ;
Traps — Ловушки;

(¤) Аппаратные (внешние) прерывания

Метод опроса и метод прерываний (Polling vs Interrupt)



231468–3



231468–4

Аппаратные прерывания генерируются контроллером прерываний — их количество зависит от самого контроллера прерываний.

Маскирование

Внешние прерывания в зависимости от возможности запрета делятся на:

- маскируемые;
- немаскируемые (Non-Maskable Interrupt, NMI).

Маскируемые — прерывания, которые можно запрещать установкой соответствующих битов в регистре маскирования прерываний (в x86 — сбросом флага IF в регистре флагов);

Маскируемые прерывания называются так именно потому, что на них можно поставить маску и запретить некоторые из них (в контроллере прерываний).

Бит **IF** регистра флагов отвечает за маскирование аппаратных прерываний.

— если IF очищен, то маскируемые прерывания игнорируются (**`CLI`**).

— если IF выставлен, то маскируемые прерывания обрабатываются (**`STI`**).

При сбросе флага IF запрещаются все маскируемые прерывания.

Немаскируемые прерывания обрабатываются всегда, независимо от запретов на другие прерывания. К примеру, такое прерывание может быть вызвано сбоем в микросхеме памяти.

(¤) Приорити[1]зация аппаратных прерываний (PIC/APIC)

До окончания обработки прерывания обычно устанавливается запрет на обработку этого типа прерывания, чтобы процессор не входил в цикл обработки одного прерывания.

Приоритизация означает, что все источники прерываний делятся на классы и каждому классу назначается свой уровень приоритета запроса на прерывание.

Приоритеты могут обслуживаться как **относительные** и **абсолютные**.

Относительное обслуживание прерываний означает, что если во время обработки прерывания поступает более приоритетное прерывание, то это прерывание будет обработано только после завершения текущей процедуры обработки прерывания.

Абсолютное обслуживание прерываний означает, что если во время обработки прерывания поступает более приоритетное прерывание, то текущая процедура обработки прерывания вытесняется, и процессор начинает выполнять обработку вновь поступившего более приоритетного прерывания. После завершения этой процедуры процессор возвращается к выполнению вытесненной процедуры обработки прерывания.

---

1 gramota.ru

Программные прерывания

Программные прерывания генерируются инструкцией процессора **`INT n`**.

Программные прерывания используются для вызовов многих API функций, в том числе и системных вызовов на платформе IA-32.

Исключения

Исключения генерируются самим процессором при попытке нарушить ограничения защиты или при возникновении ошибок во время выполнения программы.

Ограничения защиты:

- контроль предела;
- контроль типа;
- контроль уровня привилегий;
- контроль выравнивания;
- ограничение адресного пространства;
- ограничение точек входа в процедуры (для шлюзов);
- ограничение набора команд (привилегированные инструкции).

Прикладные программы, выполняющиеся даже на нулевом уровне привилегий, не могут повлиять на исключения, либо замаскировать их.

Исключения процессора генерируются вне зависимости от флага IF.

Это аппаратный контроль защиты.

Исключения делятся на три типа в зависимости от условий их возникновения — ошибка, ловушка, авария.

1. Ошибка — это исключение, возникающее в ситуации ошибочных действий программы (подразумевается, что такую ошибку можно исправить). Такой тип исключения допускает рестарт команды, которая вызвала исключение после исправления ситуации, для чего в стеке обработчика адрес возврата из прерывания указывает на команду, вызвавшую исключение.

Примером такого исключения может быть исключение отсутствующей страницы.

Благодаря этому реализуется механизм виртуальной памяти, в частности подкачка данных с диска.

2. Ловушка — это исключение, возникающее сразу после выполнения некоторой «отлавливаемой» команды. Это исключение позволяет продолжить выполнение программы со следующей команды. На ловушках строится механизм отладки программ.

3. Авария — это исключение, которое не позволяет продолжить выполнение прерванной программы и сигнализирует о серьёзных нарушениях целостности системы.

Примером аварии служит исключение двойного нарушения (прерывание 8), когда сама попытка обработки одного исключения вызывает другое исключение.

С целью корректного определения источника ошибки для некоторых исключений процессор помещает в стек 2-байтовый код ошибки.

Обработка прерываний

Обработчик прерываний (или процедура обслуживания прерываний) — специальная процедура, вызываемая по прерыванию для выполнения его обработки. Обработчики прерываний могут выполнять множество функций, которые зависят от причины, которая вызвала прерывание.

Обработчик прерываний — это низкоуровневый эквивалент обработчика событий. Обработчики вызываются либо по аппаратному прерыванию, либо соответствующей инструкцией в программе, и соответственно обычно предназначены для взаимодействия с устройствами или для осуществления вызова функций операционной системы.

На IBM PC/AT обработчики основных аппаратных и программных прерываний находятся в памяти BIOS.

Современная операционная система реального режима, во время своей загрузки, заменяет эти обработчики своими в процессе загрузки и инициализации драйверов устройств.

Операционная система распределяет управление обработкой прерывания между драйверами.

Таблица прерываний

Обработчики прерываний связываются с устройствами посредством векторов прерываний.

Вектор прерывания — закреплённый за устройством номер, который идентифицирует соответствующий обработчик прерываний.

Векторы прерываний объединяются в **таблицу векторов прерываний**, содержащую адреса обработчиков прерываний. Местоположение таблицы зависит от типа и режима работы процессора.

В реальном режиме таблица прерываний располагается в области памяти BIOS.

В защищенном режиме адрес таблицы прерываний может быть любым.

Прерывания реального режима

В реальном режиме может существовать 256 различных прерываний. В начале памяти по адресу **`0000:0000`** расположена таблица прерываний, состоящая из 256 4-байтовых адресов в виде

`<dw сегмент>:<dw смещение>.`

Каждый такой адрес указывает на процедуру, обрабатывающую прерывание.

Такие процедуры называются **обработчиками прерываний**, а адреса в этой таблице — **векторами**.

В режиме реальной адресации исключений не существует.

(¤) Прерывания защищенного режима

В защищённом режиме количество прерываний такое же, но таблица прерываний теперь может находиться по любому адресу и называется **IDT (Interrupt Descriptor Table, таблица дескрипторов прерываний**).

В защищённом режиме каждое прерывание описывается не 4-байтовым адресом, а **дескриптором шлюза прерываний**.

Шлюз (Gateway) — это специальный системный объект, который обеспечивает защиту и позволяет программам на разных уровнях привилегий использовать системные ресурсы, в частности, прерывания.

Обычно обработчик прерывания находится на нулевом уровне привилегий, а вызывающий, как правило, выше, поэтому процессор как бы проходит через шлюз и опускается вниз на нулевой уровень привилегий, а после обработки поднимается вверх на уровень вызывающего.

Различают три типа шлюзов:

1. шлюз задачи;
2. шлюз прерывания;
3. шлюз ловушки.

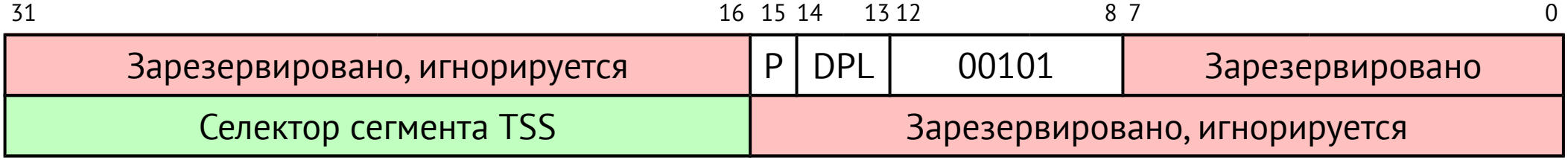
(¤) Шлюз задачи

При использовании шлюза задачи в случае, если генерируется прерывание, которое обрабатывается шлюзом задачи, происходит автоматическое переключение задачи.

Шлюз содержит селектор TSS (Task State Segment) задачи, на которую будет производиться переключение при использовании этого прерывания.

В поле «тип шлюза» будут стоять значения 1, 0, 1 в битах 8, 9, 10 соответственно. Все остальные биты за исключением бита присутствия всегда равны нулю либо зарезервированы.

Формат дескриптора шлюза задачи



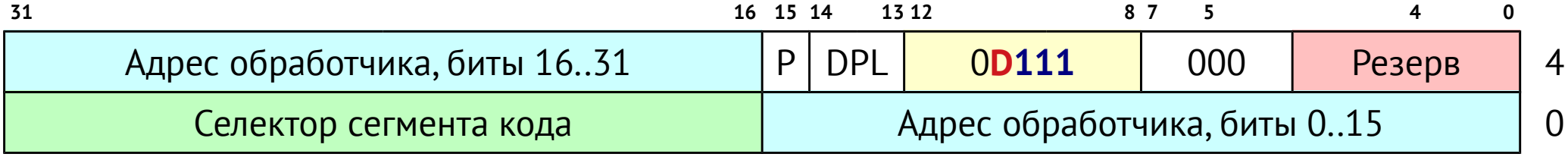
Поле **DPL** содержит максимальный уровень привилегий, с которого можно обратиться в этому шлюзу. Например, если поле **DPL** содержит 1, то вызвать данный шлюз могут только программы, выполняющиеся на уровнях привилегий 0 и 1.

Вызов этого шлюза на уровнях 2 и 3 приведёт к исключению общей защиты.

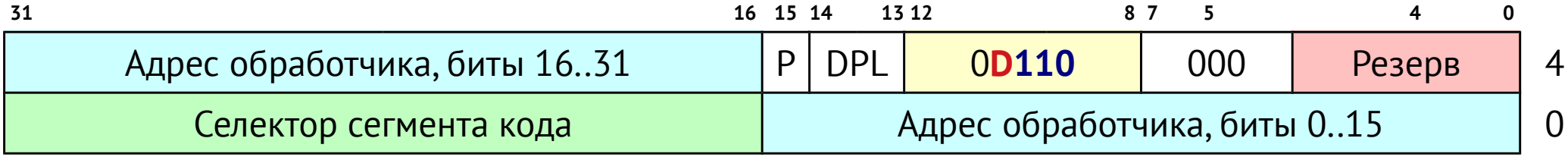
(¤) Дескрипторы шлюза ловушки и дескрипторы шлюза прерывания

Дескрипторы шлюза ловушки и дескрипторы шлюза прерывания отличаются лишь в том, что в битах типа у дескриптора шлюза ловушки будет стоять значение **111**, а у шлюза прерывания - **110**.

Дескриптор шлюза ловушки (8 байт)



Дескриптор шлюза прерывания (8 байт)



При переходе через **шлюз прерывания** процессор автоматически сбрасывает флаг IF и тем самым не допускает генерации других прерываний на время работы обработчика, а для шлюза ловушки - не меняет состояние флага IF.

Ловушки используются для отладки программ.

Бит D - это размер шлюза: если он установлен, то 32 бита; иначе 16 бит. Размер шлюза определяет размер стека, используемый процессором по умолчанию.

Перед вызовом обработчика процессор помещает в стек значения регистров CS, EIP, EFLAGS и SS, ESP (если переход осуществился на другой уровень привилегий) и 16-битный код ошибки. Если размер шлюза - 32 бита, то значения размером в 16 бит будут расширены нулями до 32.

Бит присутствия P. Если сброшен, то процессор понимает это как отсутствие обработчика и генерирует исключение общей защиты.

Дескриптор обработчика исключения общей защиты находится в IDT[13].

При любых непредвиденных ошибках и для большинства стандартных ошибок генерируется исключение общей защиты. Если генерируется конкретное исключение и его обработчика нет в IDT или в его дескрипторе ошибки, то генерируется исключение отсутствующего сегмента;

если его дескриптора нет, то процессор «зависнет» или перезагрузится.

Все дескрипторы объединяются в таблицу **IDT**. В этой таблице может быть от 0 до 256 дескрипторов. Параметры **IDT** задаются в регистре **IDTR** аналогично регистру **GDTR**.

Инструкции программных прерываний

Инструкции **INT n** (программное прерывание), **INTO** (прерывание при переполнении) и **BOUND** (обнаружение значения вне диапазона) позволяют программе явным образом вызывать указанное прерывание или исключение, что, в свою очередь, приводит к тому, что будет вызвана подпрограмма обработчика прерывания или исключения.

Инструкция **INT n** может вызвать любое из прерываний или исключений процессора, указывая в инструкции вектор прерывания или исключения.

Данная инструкция может использоваться для поддержки программных прерываний или для проверки работы обработчиков прерываний и исключений.

Для возврата программного управления из обработчика прерывания в прерванную процедуру используется инструкция **IRET** (возврат из прерывания).

IRET выполняет операцию, аналогичную инструкции **RET**.

Когда процессор обрабатывает прерывание, вместе с указателем на инструкцию возврата в стеке автоматически сохраняется содержимое регистра **EFLAGS**.

Инструкция **INTO** вызывает исключение переполнения, если установлен флаг **OF**. Если флажок снят, выполнение продолжается без вызова исключения. Эта инструкция позволяет программному обеспечению явно обращаться к обработчику исключений переполнения для проверки условий переполнения.

Инструкция **BOUND** сравнивает значение со знаком с верхней и нижней границами и вызывает исключение «BOUND range exceeded — превышен диапазон BOUND», если значение меньше нижней границы или больше верхней границы. Эта инструкция полезна для таких операций, как проверка индекса массива, чтобы убедиться, что он попадает в диапазон, определенный для него.

INT n/INTO/INT 3 — Вызов процедуры прерывания

```
INT     imm8
INT3
INTO
```

Инструкция **INT n** генерирует вызов обработчика прерываний или исключений, указанного в операнде.

Операнд задает вектор от 0 до 255, закодированный как 8-битное промежуточное значение без знака.

Каждый вектор предоставляет индекс дескриптора шлюза в **IDT**.

Первые 32 вектора зарезервированы Intel для системного использования. Некоторые из этих векторов используются для исключений, генерируемых внутри процессора.

Инструкция **INT n** является общей мнемоникой для выполнения программно-сгенерированных вызовов обработчиков прерываний.

Инструкция **INTO** является специальной мнемоникой для вызова исключения переполнения (**#OF**), исключение 4. Данное прерывание проверяет флаг **OF** в регистре **EFLAGS** и, если флаг **OF** установлен в 1, вызывает соответствующий обработчик прерывания.

Инструкция **INT3** генерирует специальный однобайтовый код операции (**CC**), который предназначен для вызова обработчика исключений отладки.

Данная однобайтовая форма полезна для отладки, поскольку ее можно использовать для замены первого байта любой инструкции в точке останова, включая другие однобайтовые инструкции, без перезаписи кода других инструкций.

Отладочное прерывание, генерируемое с помощью кода операции **CC**, отличается от обычных программных прерываний следующим образом:

- в режиме VME[2] не происходит перенаправление прерываний — прерывание обрабатывается обработчиком защищенного режима.

- проверка IOPL в режиме virtual-8086 не выполняется — прерывание выполняется без сбоев на любом уровне IOPL.

Регулярный 2-байтовый код операции для `INT 3` (`CD03`) не имеет этих специальных функций.

Некоторые ассемблеры, скорее всего (Intel и Microsoft точно) не будут генерировать код операции **CD03** из какой-либо мнемоники, тем не менее данный код операции может быть создан путем прямого определения числового кода.

Nasm генерирует этот код.

```
24 0000001D CD03        int 3
25 0000001F CC          int3
```

2 VME -- Virtual-8086 Mode Extensions(Расширение виртуального режима 8086). Виртуальный режим (V86) -- это особое состояние задачи защищенного режима, в котором процессор использует модель реального адреса для формирования линейных адресов.

Действие инструкции INT n

Инструкция **INT n** (включая инструкции **INTO** и **INT 3**) работает аналогично дальнему вызову, выполненному с помощью инструкции **CALL**. Основное отличие состоит в том, что с помощью инструкции **INT n** в стек перед адресом возврата помещается регистр **EFLAGS**.

Адрес возврата — это дальний адрес, состоящий из текущих значений регистров **CS** и **EIP**. Возвраты из процедур прерывания обрабатываются инструкцией **IRET**, которая извлекает информацию **EFLAGS** и адрес возврата из стека.

Вектор указывает дескриптор прерывания в таблице дескрипторов прерываний **IDT**, то есть он является индексом в **IDT**. В свою очередь, выбранный дескриптор прерывания содержит указатель на процедуру обработки прерывания или исключения.

В защищенном режиме IDT содержит массив 8-байтовых дескрипторов, каждый из которых является шлюзом прерывания, шлюзом ловушки или шлюзом задачи.

В режиме реального адреса IDT представляет собой массив 4-байтовых дальних указателей (2-байтовый селектор сегмента кода и 2-байтовый указатель инструкций), каждый из которых указывает непосредственно на процедуру в выбранном сегменте.

В режиме реального адреса **IDT** называется таблицей векторов прерываний, а ее указатели называются векторами прерываний.

Флаги

Регистр **EFLAGS** помещается в стек.

Флаги **IF**, **TF**, **NT**, **AC**, **RF** и **VM** могут быть очищены в зависимости от режима работы процессора при выполнении команды **INT**. Если прерывание использует шлюз задачи, любые флаги могут быть установлены или удалены модифицированием образа **EFLAGS** в **TSS** новой задачи.

IRET/IRETD — возврат из прерывания

```
IRET
IRETD
```

Описание

Возвращает программное управление из обработчика исключений или прерываний программе или процедуре, которая была прервана:

- исключением;
- внешним прерыванием;
- программным прерыванием.

IRET и **IRETD** являются мнемоникой для одного и того же кода операции. Мнемоника **IRETD** (двойной возврат прерывания) предназначена для использования при возврате из прерывания при использовании размера 32-битного операнда, однако большинство ассемблеров используют мнемонику **IRET** взаимозаменяемо для обоих размеров операндов.

В режиме реального адреса команда **IRET** выполняет дальний возврат к прерванной программе или процедуре. Во время этой операции процессор извлекает указатель инструкции возврата, селектор сегмента кода возврата и образ **EFLAGS** из стека в регистры **EIP**, **CS** и **EFLAGS** соответственно, после чего выполнение прерванной программы или процедуры возобновляется.

(¤) BOUND — проверка индекса массива по границам

```
BOUND    r16, m16&m16
BOUND    r32, m32&m32
```

Описание

Инструкция BOUND определяет, находится ли первый операнд (индекс массива) в пределах границ массива, указанных вторым операндом (операнд границ).

Индекс массива — это целое число со знаком, расположенное в регистре.

Операнд границ — это местоположение в памяти, которое содержит пару целых чисел со знаком в размере двойного слова, если атрибут размера-операнда равен 32, или пару целых чисел со знаком в размере слова, когда атрибут размера-операнда равен 16.

Первое двойное слово (слово) является нижней границей массива, второе двойное слово (слово) — верхней.

Индекс массива должен быть больше или равен нижней границе и меньше или равен верхней границе плюс размер операнда в байтах. Если индекс находится за пределами границ, сигнализируется исключение превышения диапазона **BOUND** (**#BR**). Когда генерируется это исключение, указатель сохраненной инструкции возврата указывает на инструкцию **BOUND**.

Структура с данными о границах (пара слов или двойных слов, содержащих нижний и верхний пределы массива) обычно размещается непосредственно перед самим массивом, делая пределы адресуемыми с помощью постоянного смещения от начала массива. Поскольку адрес массива уже будет присутствовать в регистре, эта практика позволяет избежать лишних циклов шины для получения эффективного адреса границ массива.