Архитектура современных ВУ
Лекции 6-7. Организация памяти

П. А. Макаров

октябрь, ноябрь 2024 г.

Содержание

1. Введение

2. Модели управления памятью

3. Адресные пространства

4. Директивы определения данных

5. Виды адресаций операндов инструкций в памяти

6. Стек. Подпрограммы

Введение

Оперативная память доступная процессору Intel 8086

Max объём — 1 МБайт.

Архитектура IBM PC

Max ОЗУ — 640 КБайт
("стандартная память").

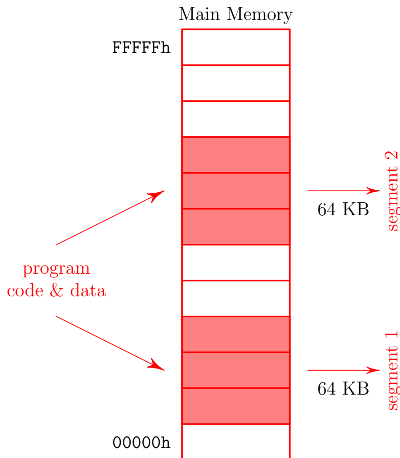


Рис. 1: Организация памяти i8086

На символ отводится один байт памяти, в который записывается код символа — целое от 0 до 255. В x86 используется система кодировки ASCII.

1. Строка (последовательность символов) размещается в соседних байтах памяти (в неперевернутом виде): код первого символа строки записывается в первом байте, код второго символа — во втором байте и т. п. Адресом строки считается адрес её первого байта.

2. В x86 строкой считается также последовательность машинных слов (обычно это последовательность целых чисел). Элементы таких строк располагаются в последовательных ячейках памяти, но каждый элемент представлен в «перевёрнутом» виде.

Адрес — это порядковый номер ячейки памяти (неотрицательное целое число).

1. `OFFSET` — 16-битное смещение.
2. Адресная пара в форме двойного слова `SEGMENT:OFFSET`. Определяет 20-битный абсолютный адрес ячейки памяти

$$\texttt{Absolute Address} = \texttt{SEGMENT} \cdot \texttt{10h} + \texttt{OFFSET}. \quad (1)$$

Пример 1

Пара 1234h:5678h представляется так:

78	56	34	12
Смещение		Сегмент	

Рис. 2: Машинное представление адресных пар

- Сегментация — это средство организации логической памяти на прикладном уровне.
- Страничная трансляция адресов применяется на системном уровне для управления физической памятью.

Реализация виртуальной памяти

Сегменты и страницы могут выгружаться из физической оперативной памяти на жёсткий диск (в так называемый “файл подкачки” или специальный swap-раздел), и по мере необходимости подкачиваться с него обратно в физическую память.

Впервые средства поддержки виртуальной памяти появились в процессорах Intel 80286, но удобный для использования вид приняли только в 32-битных процессорах (начиная с процессора Intel 80386).

- Реальный — Real Address Mode.
- Защищённый — Protected Virtual Address Mode.
- Виртуальный — Virtual 8086 Real Mode.
- SMM — System Management Mode.

Преимущества защищённого режима

- Адресуемость памяти выше предела в 1 МБайт.
- Переключение задач (многозадачный режим).
- Защита памяти программ.
- Виртуализация памяти.
- PAE — Physical Address Extension. Позволяет адресовать более 4 ГБайт (2^{32}) памяти.

Основные модели памяти

1. Сегментная модель.
2. Плоская модель.

Режимы работы процессоров x86-64

- Compatibility mode.
- Long (64-bit) mode.
- SMM — System Management Mode.

В процессорах x86-64 появилась возможность адресации данных относительно текущего значения `RIP`.

Определение 1

Адресное пространство *— это набор адресов, которые умеет формировать процессор.*

Замечание 1

Термины ***адресное пространство*** *и* ***оперативная память*** *не эквивалентны.*

Адресные пространства

1. Логическое (виртуальное).
2. Линейное.
3. Физическое.

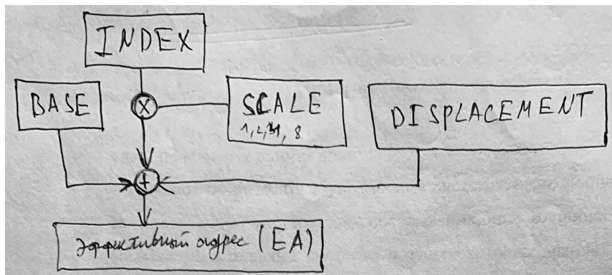


Рис. 3: Формирование эффективного адреса

В зависимости от типа используемой адресации операндов, компоненты эффективного адреса могут быть непосредственными константами в машинной инструкции, находиться в регистрах или размещаться в ячейках памяти.

Определение 2

Логический адрес *— это адресная пара* `Selector:Offset`*, образованная селектором сегмента и смещением, играющим роль эффективного адреса.*

`Selector` — это значение старших 14 битов сегментного регистра, участвующего в адресации данного элемента памяти. По значению селектора из таблиц дискрипторов сегментов, хранящихся в памяти, извлекается начальный адрес сегмента.

Замечание 2

Max пространство виртуальной памяти, доступное программисту в сегментной модели равно 64 ТБайт. Этот объём образуется совокупностью 16 Кбит (2^{14}) селекторов, в то время как размер сегментов ограничен 4 ГБайт (2^{32}).

Логика обработки машинной команды построена так, что при выборке команды, доступе к данным программы или к стеку неявно используются определённые сегментные регистры.

Типы сегментов памяти и адресующие их регистры

- Сегмент кода (`CS`).
- Сегменты данных (`DS`, `ES`, `FS` и `GS`).
- Сегмент стека (`SS`).

Замечание 3

Если программе недостаточно одного сегмента данных, то она имеет возможность использовать ещё три дополнительных сегмента данных. Для этого их адреса требуется указывать явно с помощью специальных префиксов переопределения сегментов в команде.

Каким образом понятие сегментов памяти отражается на структуре программы?

Структура исходного текста программы определяется:

1. Архитектурой процессора (если обращение к памяти возможно только с помощью сегментов, то и программа должна состоять из сегментов).
2. Особенностями той операционной системы, под управлением которой эта программа будет выполняться.
3. Правилами работы выбранного транслятора — разные трансляторы предъявляют несколько различающиеся требования к исходному тексту программы.

Адресные пространства

Сегментная структура программ

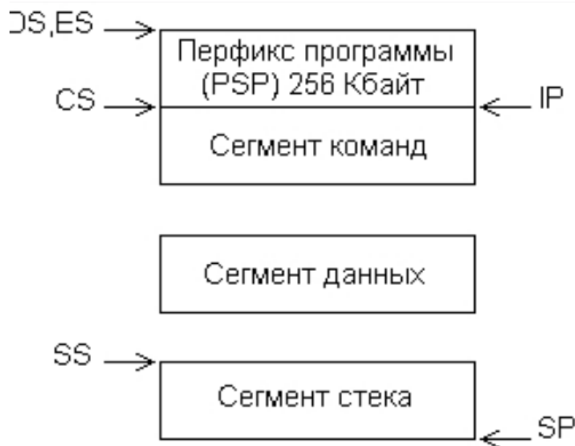


Рис. 4: Образ программы, состоящей из нескольких сегментов, в памяти компьютера

```
segment data
hello:    db 'Hello, world!', 0x0D, 0x0A, '$'

segment code
..start:  mov ax, data
          mov ds, ax
print:    mov dx, hello
          mov ah, 09
          int 0x21
quit:     mov ax, 0x4C00
          int 0x21

segment stack class=stack
          resb 512
```

Адресные пространства

Преобразования адресов в процессорах IA-32

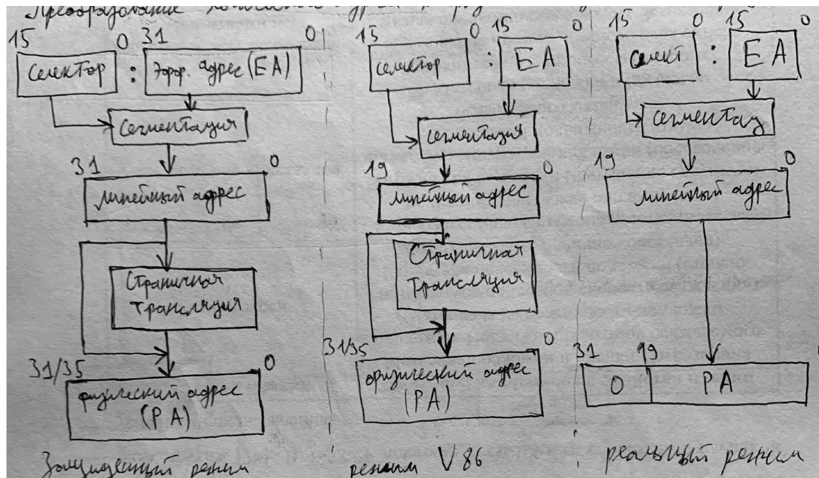


Рис. 5: Преобразования адресов в различных режимах IA-32

Рис. 6: Образование физического адреса в реальном режиме

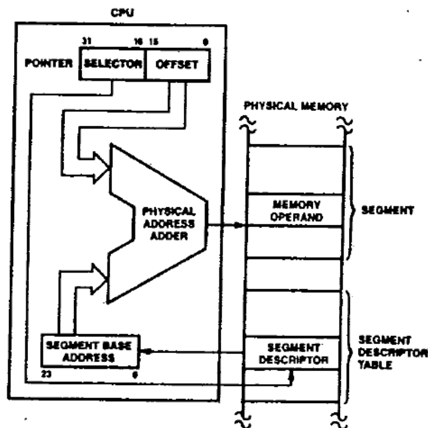


Рис. 6: Образование физического адреса в защищённом режиме i80286

Блок страничной трансляции позволяет использовать разрядность физического адреса, отличающуюся от разрядности линейного адреса.

1. Процессор i80386SX при 32-битном линейном адресе имел физический адрес 24 бита (до 16 МБайт ОЗУ).
2. В большинстве 32-битных процессоров до 6 поколения использовался 32-битный физический адрес, что позволяло подключать до 4 ГБайт ОЗУ.
3. В процессорах 6 – 8 поколений используется расширение физического адреса (PAE — Physical Address Extension): 32-битный линейный адрес транслируется в 36-битный физический адрес (до 64 ГБайт ОЗУ).
4. Процессоры x86-64 имеют линейный адрес 48 бит, а физический — до 52 бит.

Адресные пространства

Преобразование адресов в процессорах x86-64

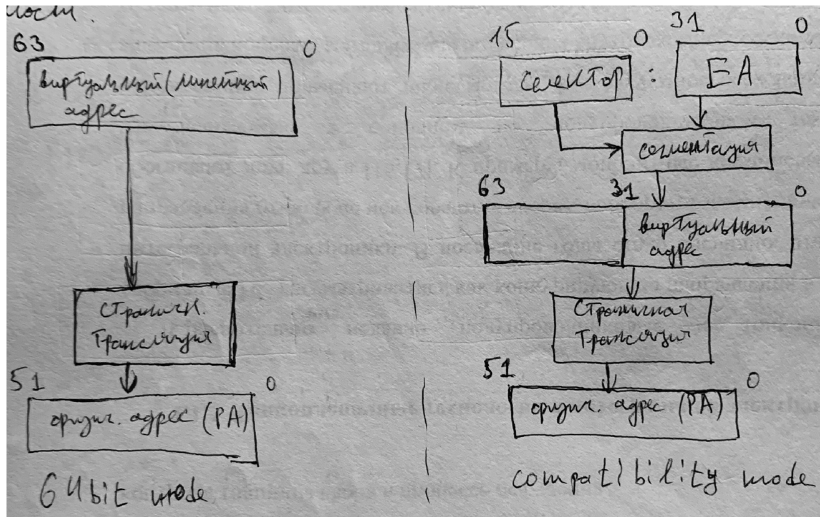


Рис. 7: Преобразования адресов в процессорах x86-64

Замечание 4

В режиме совместимости адреса должны соответствовать канонической форме, т. е. их старшие биты, выходящие за пределы поддерживаемой разрядности должны быть нулевыми. В противном случае сработает исключение защиты.

Замечание 5

Из сегментных регистров процессор использует только регистры `CS`, `FS` и `GS`. В дискрипторе сегмента, на который указывает `CS`, задействуются лишь следующие атрибуты: признак 64-битного режима, размер операнда по-умолчанию и уровень привилегий. Регистры `FS` и `GS` требуются для нового режима адресации — в дискрипторе сегмента, на который они ссылаются базовый адрес, может применяться как смещение при вычислении адреса (эффективного, виртуального и линейного — теперь это одно и то же).

Адресные пространства

Плоская модель памяти IA-32

Эта модель подразумевает, что каждому приложению отводится линейное адресное пространство объёмом 2 ГБ, а остальные 2 ГБ предоставляются ОС.

Базовый адрес в дескрипторах всех сегментов приложения устанавливается равным 0. В результате все сегменты приложения «перекрываются». Код, данные и стек размещаются в разных местах памяти за счёт различных смещений.

Разделение памяти между приложениями осуществляется ОС, которая размещает страницы приложений с одинаковыми линейными адресами в разных местах ОЗУ. Как следствие, — защита сегментов при этой модели не работает.

Чтобы предотвратить взаимное влияние выполняющихся программ друг на друга им выделяются изолированные участки памяти (т. е. код и данные программ находятся во взаимно несмежных сегментах).

Адресные пространства

Структура программ NASM в плоской модели памяти

```
; write (01) to stdout (01) and exit (60)
; nasm -f elf64 example.asm && ld example.o -o example

global _start

section .text
_start:   mov rax, 01
          mov rdi, 01
          mov rsi, message
          mov rdx, 14
          syscall
          mov rax, 60
          mov rdi, 0
          syscall

section .data
message:  db "Hello, world!", 0x0A
```

Адресные пространства

Загрузка объектного файла в память под управлением ОС защищённого режима

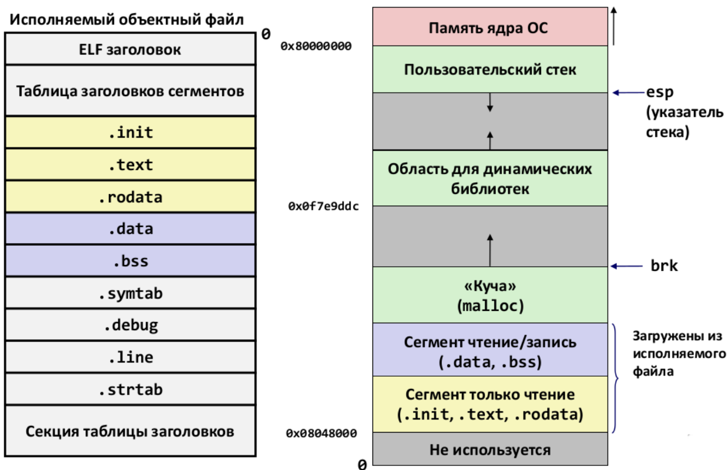


Рис. 8: Структура файла формата ELF и его образ в памяти

Определение 3

Директивы *— это предложения программы, которыми ассемблеру сообщается информация или предписывается сделать что-то дополнительное, кроме непосредственного перевода символьных команд на машинный язык.*

Директива	DB	DW	DD	DQ	DT	DDQ или DO
Размер, Байт	1	2	4	8	10	16

Таблица 1: Директивы определения данных NASM

```
A DB 162    ; константа-байт 162 с именем A
B DB 0A2h   ; такая же константа, но с именем B
C DW -1     ; константа-слово -1 с именем C
D DW 0xFFFF ; такая же константа-слово, но с именем D
E DD -1     ; константа -1 с именем E как двойное слово
```

Директивы определения данных

Символьные данные и перечисления в NASM

```
CH1 DB 02Ah
CH2 DB '*'
CH3 DB "*"
```

```
array1 dw -7, 132, 0, 1097
array2 dd 1, 2, 3, 4, 5
```

Имя, указанное в директиве, считается именующим первую из констант. Для ссылок на остальные в NASM используются выражения вида `<имя> + <целое>`.

```
string: db 'Hello, world!', 0Ah, 0Dh, '$'
```

Замечание 6

Любой размер больше, чем `DQ` *не допускает целочисленных строк в качестве операндов.*

Директивы определения данных

Неинициализированные данные

Неинициализированные данные (данные такого типа размещаются в BSS-секции объектного модуля) можно объявить в программе с помощью директив NASM `RESB`, `RESW`, `RESD`, `RESQ`, `REST`, `RESDQ` или `RESO`. Каждая из этих директив должна сопровождаться одним операндом, являющимся числом резервируемых элементов.

```
buffer      resb 64
shortvar    resw 1
doublearray resq 10
```

Замечание 7

NASM не поддерживает синтаксис резервирования неинициализированного пространства, реализованный в MASM/TASM, где можно делать `DW ?` или подобное.

Префикс TIMES

```
zerobuf: times 64 db 0
buffer:  db 'Hello, world'
         times 64-$+buffer db ' '
```

Во втором случае будет зарезервировано строго определённое пространство, расширяющее полную длину buffer до 64 байт.

Константные указатели и директива SEG

```
ADR1 DW CH1

ADR2 DW CH2, SEG CH2
```

Определение 4

Адресацией *называется обращение к операнду, указание на который содержится в команде.*

Операнды, в зависимости от места своего хранения, могут указываться разными способами, которым соответствуют разные типы адресации. При описании различных адресаций операндов используют понятия адресного кода и исполнительного адреса.

Определение 5

Адресный код *— это информация об адресе операнда, содержащаяся в команде.*

Определение 6

Исполнительный адрес *— это номер физической ячейки памяти, к которой производится обращение.*

Виды адресаций операндов инструкций в памяти

Первая группа типов адресаций

Устанавливает исполнительный адрес по адресному коду.

- Непосредственная адресация.

  ```
  mov ax, 512
  ```

- Регистровая адресация.

  ```
  mov ax, cx
  ```

- Прямая адресация.
  - Относительная прямая адресация.

    ```
    jmp 200
    ```

  - Абсолютная прямая адресация.

    ```
    mov ax, [200]
    ```

- Косвенная регистровая адресация.

  ```
  mov ax, [ecx]
  ```

- Автоинкрементная и автодекрементная адресация.

Виды адресаций операндов инструкций в памяти

Пример автоинкрементной адресации — цепочечные команды

```
segment data
  str:  db "Example of string$"
  len:  dw $-str
  buf:  resb 20    ; Change to 10
  size: dw $-buf

segment code
..start:  mov ax, data
          mov ds, ax
          mov es, ax

          cld   ; From Start
          lea si, [str]
          lea di, [buf]
          mov cx, [len]
          cmp cx, [size]
          jg quit
      rep movsb
```

```
print:    mov ah, 09
          lea dx, [buf]
          int 0x21

quit:     mov ax, 0x4C00
          int 0x21
```

Устанавливает исполнительный адрес по адресному коду и содержимому дополнительных регистров процессора.

- Индексация.
- Базирование.

Замечание 8

Все виды адресации, кроме непосредственной, регистровой и прямой относятся к косвенным. Слово косвенный в названии этих видов адресации означает, что в самой команде может находиться лишь часть эффективного адреса, а остальные его компоненты находятся в регистрах.

Общий формат эффективного адреса

```
mov <reg>, [B + N*I + D]  ; NASM
mov <reg>, D[B][I*N]      ; MASM, DEBUG
```

- Косвенная базовая адресация со смещением.
  ```
  mov ax, [ebx + 3]
  ```
- Косвенная индексная адресация со смещением.
  ```
  mov al, [4*esi + 3]
  ```
- Косвенная базовая индексная адресация.
  ```
  mov eax, [ebx + 2*esi]
  ```
- Косвенная базовая индексная адресация со смещением.
  ```
  mov al, [ebx + 4*esi + 3]
  ```

Исключение — режим 16-битной адресации

```
mov <reg>, [bp|bx] +  [si|di] + disp0/8/16 ; NASM
mov <reg>, [bp|bx] + N[si|di] + disp0/8/16 ; DEBUG
```

Исключение — режим 16-битной адресации

```
mov <reg>, [bp|bx] +  [si|di] + disp0/8/16 ; NASM
mov <reg>, [bp|bx] + N[si|di] + disp0/8/16 ; DEBUG
```

Замечание 9

Т. к. сегментирование — это разновидность модификации адресов, то адрес, указываемый в команде x86, в общем случае модифицируется по трём регистрам — сегментному, базовому и индексному. Это происходит в два этапа.

1. *Сначала учитываются только базовый и индексный регистры. Полученный в результате адрес называется исполнительным (или эффективным) адресом.*
2. *Если же нужен доступ к памяти, тогда на втором этапе исполнительный адрес рассматривается как смещение, и к нему прибавляется начальный адрес необходимого сегмента.*

1. В командах перехода адрес перехода сегментируется только по регистру `CS`.
2. Адреса во всех других командах, кроме цепочечных (`stos`, `movs`, `scas` и `cmps`), сегментируются по умолчанию, а именно:
   - по регистру `DS`, если среди указанных регистров-модификаторов нет регистра `BP`;
   - по регистру `SS`, если один из модификаторов — регистр `BP`.
3. В цепочечных командах, имеющих два операнда-адреса, на которые указывают индексные регистры `SI` и `DI`, один из операндов (на который указывает `SI`) сегментируется по регистру `DS`, а другой (на него указывает `DI`) — по регистру `ES`.

Виды адресаций операндов инструкций в памяти

Пример базовой индексной адресации — инвертирование строки

```
segment data
buffer: db 0xFF
  resb 255
segment code
..start:
  mov ax, data
  mov ds, ax
  mov ah, 0x0A
  mov dx, buffer
  int 0x21
  xor cx, cx
  xor si, si
  xor di, di
  mov cl, [buffer+1]
  mov si, cx
  mov byte [buffer+2+si], '$'
  dec si
  shr cl, 1
  jz print
swap:
  mov al, [buffer+2+si]
  mov ah, [buffer+2+di]
  mov [buffer+2+di], al
  mov [buffer+2+si], ah
  dec si
  inc di
  loop swap
print:
  mov word [buffer], 0x0A0D
  mov ah, 9
  mov dx, buffer
  int 0x21
  mov ax, 0x4C00
  int 0x21
segment stack class=stack
  resb 512
```

Определение 7

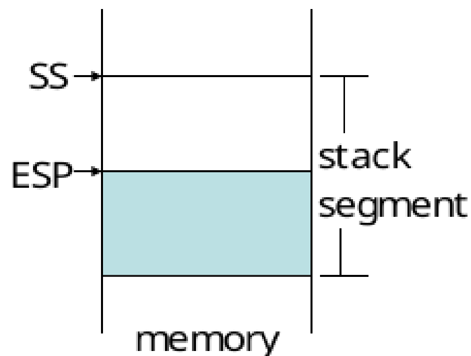
Стек *— область памяти программы для временного хранения произвольных данных, доступ к которым осуществляется по принципу LIFO (Last Input — First Output) «последним записан — первым считан».*

Применения стека, связанные с процедурами

- Сохранение состояний всех регистров перед вызовом процедуры или обработчиков прерываний.
- Автоматическое сохранение адреса возврата из подпрограмм.
- Передача фактических параметров при вызове процедуры, а также приём результатов работы функций.
- Хранение локальных переменных подпрограммы.

Стек. Подпрограммы

Организация стека



- **PUSH** syntax:
  - **PUSH** ***r/m16***
  - **PUSH** ***r/m32***
  - **PUSH** ***imm32***
- **POP** syntax:
  - **POP** ***r/m16***
  - **POP** ***r/m32***

Рис. 9: Косвенная адресация элементов стека и синтаксис основных команд для работы с ним в процессорах IA-32

Стек. Подпрограммы

Состояния стека

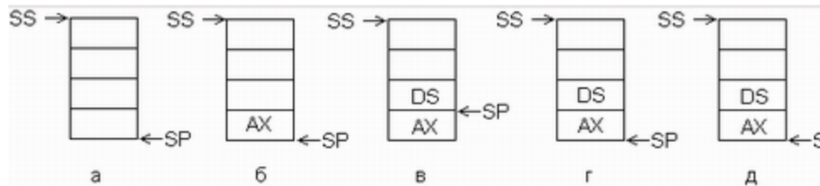


Рис. 10: Состояния стека

Замечание 10

1. *Значение «0» в регистре SP свидетельствует о том, что стек полностью заполнен.*
2. *Для пустого стека значение SP равно размеру стека, т. е. пара SS:SP указывает на байт, следующий за последним байтом области стека.*

Дополнительные комментарии к рис. 10

Замечание 11

После выгрузки сохраненных в стеке данных, они не стираются физически, а остаются в области стека на своих местах. Однако, при «стандартной» работе со стеком они оказываются недоступными. Действительно, поскольку указатель стека SP указывает под дно стека, стек считается пустым. Очередная команда push поместит новое данное на место сохраненного ранее содержимого, затерев его. Однако пока стек физически не затёрт, сохранёнными и уже выбранными из него данными можно пользоваться, если помнить, в каком порядке они расположены в стеке. Этот приём часто используется при работе с подпрограммами.

Пример 2

Рекомендуемый вариант.

```
segment stack class=stack
  resb 512
```

Пример 3

Вариант описания сегмента стека с самостоятельной настройкой.

```
segment code
  mov ax, stack
  mov ss, ax
  mov sp, stacktop

segment stack
  resb 512
stacktop:
```

Замечание 12

Пример 3 работоспособен, но на этапе компоновки программы, линкер может выводить автоматические предупреждения (warnings) о том, что сегмент стека программы не обнаружен.

Замечание 13

В прикладных программах, использующих плоскую модель памяти, описание стека в исходном тексте, как правило, не требуется, т. к. все необходимые действия по его инициализации и обслуживанию возлагаются на ОС.

Стек. Подпрограммы

Стековые команды x86

- Запись в стек: `PUSH op`, `PUSHA/PUSHAD` (исключены в x86-64), `PUSHF`.
- Чтение из стека: `POP op`, `POPA/PUSHAD` (исключены в x86-64), `POPF`.
- Вызов подпрограммы: `CALL op`.
    - Внутрисегментный относительный длинный переход.
    - Внутрисегментный абсолютный косвенный переход.
    - Межсегментный абсолютный прямой переход.
    - Межсегментный абсолютный косвенный переход.
- Возврат из подпрограммы: `RET op`.
- Автоматическая подготовка стека вызванной подпрограммы: `ENTER frmSize, Count`.
- Автоматическая очистка стека вызванной подпрограммы: `LEAVE`.

Стек. Подпрограммы

Пример использования стека — инвертирование строки

```
segment data
buffer: db 0xFF
        resb 255

segment stack class=stack
        resb 512

segment code
..start: mov ax, data
         mov ds, ax
         mov ah, 0x0A
         mov dx, buffer
         int 0x21
         mov ah, 02
         mov dl, 0x0D
         int 0x21
         mov dl, 0x0A
         int 0x21
```

```
          xor cx, cx
          mov cl, [buffer+1]
          jcxz quit
          xor si, si
          xor dx, dx
pushing:  mov dl, [buffer+2+si]
          push dx
          inc si
          loop pushing

          mov cx, si
          mov ah, 02
poping:   pop dx
          int 0x21
          loop poping

quit:     mov ax, 0x4C00
          int 0x21
```

Регистр BP/EBP/RBP фиксируется для произвольной адресации в стеке параметров процедур и их локальных переменных.

```
; Data
arg0 dw 100
arg1 dw 200
argN dw 500

; Code
push [argN]
push ...
push [arg1]
push [arg0]
call myproc
```

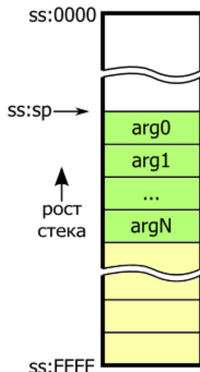


Рис. 11: Стек перед выполнением инструкции `call myproc`

Стек. Подпрограммы

Обращение к параметрам внутри процедуры

```
myproc:
  push bp
  mov bp, sp
  ...
  mov ax, [bp + 4]  ; AX = arg0
  mov bx, [bp + 6]  ; BX = arg1
  add ax, [bp + 8]  ; AX += arg2
  ...
```

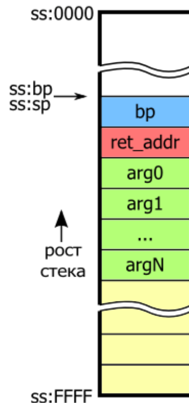


Рис. 12: Стек в начале выполнения процедуры myproc

Стек. Подпрограммы

Очистка стека после выполнения процедуры

```
push [arg1]
push [arg0]
call myproc
...

myproc:
  push bp
  mov bp, sp
  ...
  pop bp
  ret 4      ; Clear by proc
```

Стек. Подпрограммы

Очистка стека после выполнения процедуры

```
push [arg1]
push [arg0]
call myproc
...

myproc:
  push bp
  mov bp, sp
  ...
  pop bp
  ret 4      ; Clear by proc
```

```
push [arg1]
push [arg0]
call myproc2
add sp, 4     ; Clear by caller
...

myproc2:
  push bp
  mov bp, sp
  ...
  pop bp
  ret
```

Стек. Подпрограммы

Локальные переменные и автоматизация создания/очистки фрейма стека

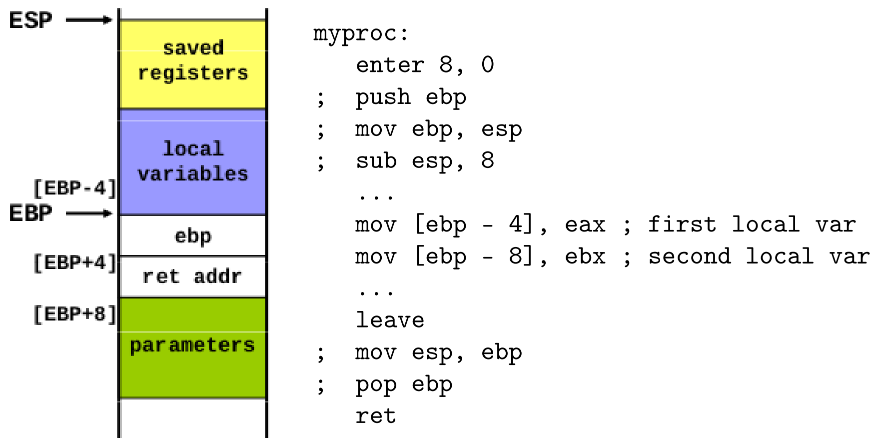


Рис. 13: Фрейм стека текущей процедуры для процессоров IA-32

Какого размера должен быть стек?

Какого размера должен быть стек?

1. Зависит от того, насколько интенсивно он используется.
2. В ряде случаев стек автоматически используется ОС, в частности, при выполнении команды прерывания `int 21h`. Поэтому, даже если программа совсем не использует стек, он все же должен присутствовать в программе и иметь размер не менее нескольких десятков (а лучше — сотен) машинных слов.

Замечание 14

Если для стека определён слишком маленький сегмент, то возможно его переполнение (stack overflow). Переполнение в ОС защищённого режима вызывает срабатывание защиты. В ОС реального режима переполнение стека приводит к "загадочным" вылетам и зависаниям программы.