**Введение в языке ассемблера**

***Конструирование ПО (часть 1)***

Ассемблер – программа, преобразующая исходный текст программы на языке ассемблера в программу на машинном языке

Язык ассемблера – язык низкого уровня с командами, обычно соответствующими командам машины

Зачем? А вот зачем:

Позволяет максимально полно контролировать поведение программы

Позволяет максимально эффективно использовать возможности платформы

Приучает к разбиению программы на небольшие составные части, из которых строятся более сложные алгоритмы

Демосцена – субкультура и направление компьютерного искусства, ставящее одной из своих целей демонстрацию максимума возможностей различных платформ

В ней ассемблер широко применяется

Ассемблеры существуют для каждой платформы

В курсе «Конструирование ПО» рассматривается ассемблер для платформы Intel IA-32 / Intel 64

В центре внимания – реальный режим (real-address mode)

Protected Mode – защищенный режим

Virtual-8086

(из read-address в protected или через Reset, из virtual-8086 в protected, из virt в real через Reset)

IA-32e Mode (в protected и обратно)

System Management Mode (ваще из любого куда угодно)

Для IA-32 существуют ассемблеры: T/M/F/Y/NASM

Преимущества FASM:

* Простой и продуманный синтаксис (это было целью его разработки)
* Существуют версии для всех основных платформ
  + Которые могут компилировать программы для любых платформ (и не только программы!)
* Мощная система макросов
* И т.д.

Процессор выполняет бесконечный цикл:

* Прочитать очередную команду (Fetch)
* Декодировать команду (Decode)
* Загрузить исходные данные (Read)
* Выполнить команду (Execute)
* Записать результаты (Write-back)
* Повторить для следующей команды

Команды процессора принято называть инструкциями

Инструкция задает

* Операцию (что делать)
* Операнды (с чем это делать)

В машинном коде инструкция – один или несколько байтов

* Ассемблер позволяет записывать инструкции словами

ЗКО ЗКО ЗКО

Любое запоминающее устройство хранить информация в виде байтов

(которые могут группироваться в байты)

Любой бит всегда находится в одном из двух состояний: либо 0, либо 1

Не бывает «пустых» или «чистых» битов и байтов!

* Если там нули – значение равно 0
* Если неизвестно, какое там значение, - значение не определено

Само по себе значение бита/байта или набора байтов бессмысленно

Например, байт со значением C3(16) может быть:

* Беззнаковой 1-байтной величиной 195
* Знаковой 1-байтной величиной -61
* Символом «Г» в кодировке Win1251
* Инструкцией RET

Оно приобретает смысл, когда с ним начинают работать

* Если с ним работают как с числом – это число
* Если с ним работают как с символом – это код символа
* Если его пытаются выполнить – это инструкция
* И т.д.

ЗКО ЗКО ЗКО

*Процессор*

При подаче питания процессор начинает выполнять инструкции

* Инструкции хранятся в ОЗУ
* Первую инструкцию процессор считывает по адресу FFFF:FFF0  
  (При включении (E)IP содержит адрес самой первой инструкции)
* Каждую следующую – там, где закончилась предыдущая  
  (После считывания инструкции (E)IP увеличивается на ее размер в байтах)
* Некоторые инструкции позволяют изменить порядок выполнения  
  (Некоторые инструкции могут изменять значение регистра (E)IP)

*Инструкции*

mov dest, src   
dest – куда  
src – откуда

add dest, src (плюс)

sub dest, src (минус)

and dest, src

or dest, src

not dest

Предназначены для выполнения примитивных операций

Могут иметь 0, 1, 2 и т.д. операнда

Имеют мнемонические обозначения (mov, int, ret и т.п.)

При компиляции преобразуются в последовательность из 1-17 байт

* Теоретически. На практике инструкции длиннее 15 байт могут обрабатываться некорректно

Только 3 вида:

* Регистр (register, **reg**)
* Константа (immediate, **imm**)
* Операнд в памяти (memory, **mem**)

Ограничение!

* Инструкция может явно обращаться только к одному операнду в памяти

Записываются прямо в коде программы

FASM поддерживает следующие способы записи:

* Десятичные константы:  
  5 16 -8 42
* 16-чные константы:  
  $C57D 0xFFEC 21h 0BCh
* 2-чные и 8-чные константы:  
  10111001b 751o

*Операнды в памяти*

Записываются в квадратных скобках

В квадратные скобки помещается выражение, задающее адрес операнда

(адрес - число, задающее место в памяти)

Адрес может задаваться:

* Числом или константным выражением:  
  [$046C] [0x78 + 24 – 1001b]
* Выражением с регистрами:  
  [eax + 4] [eax \* 4 + edx + 8] [bp - 4]

*Операнды-регистры*

В тексте программы задаются своими именами

Используются, когда нужно обратиться к регистрам

*Регистры*

Триггер – устройство, способное хранить 1 бит информации

* В каждый момент времени находится в одном из двух состояний

Регистр – устройство, состоящее из нескольких триггеров

* Разные регистры могут состоять из разного количества триггеров (бит)

Находятся внутри процессора

Самые быстрые устройства для хранения информации

* ОЗУ – внешнее устройства
* Взаимодействие с ОЗУ выполняется по определенным правилам
* Это медленно
* К регистрам процессор имеет прямой доступ, т.к. они являются частью самого процессора

ЗКО ЗКО ЗКО

Пользоваться понятием «ячейка» е рекомендуется!

Всегда нужно уточнять:

* Размер «ячейки» (в битах/байтах)
* Расположение «ячейки»:
  + «ячейкой» может оказаться регистр или его часть, место в кэше, ОЗУ или на носителе вроде жесткого диска или флешки

ЗКО ЗКО ЗКО

*Регистры*

Почти во все регистры можно записать любый значения (по размеру регистра)

Но!

* Некоторые регистры недоступны программно (напрямую)
* Некоторые регистры имеют специальное назначение, писать туда произвольные значения не имеет смысла
* Некоторые регистры могут использоваться некоторыми командами неявно

Делятся на:

* Регистры общего назначения
* Сегментные регистры
* Управляющие регистры
* Отладочные регистры
* Регистр программного счетчика (указатель инструкций)
* Регистр флагов
* И др.

*Регистры общего назначения*

31 16 15 8 7 0 16 бит 32 бит

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | AH | AL | AX | EAX |
|  | CH | CL | CX | ECX |
|  | DH | DL | DX | EDX |
|  | BH | BL | BX | EBX |
|  | SP | |  | ESP |
|  | BP | |  | EBP |
|  | SI | |  | ESI |
|  | DI | |  | EDI |

*Сегментные регистры*

Размер – 16 бит

Хранят номер сегмента памяти

* CS – code segment (сегмент кода)
* DS – data segment (сегмент данных)
* SS – stack segment (сегмент стека)
* ES – сегмент данных
* FS – сегмент данных
* GS – сегмент данных

*Регистр (E)IP*

Регистр программного счетчика (указатель инструкций)

Размер – 32 бита (младшая часть IP – 16 бит)

Напрямую из программы недоступен (есть способы получить/изменить его значение косвенно)

Содержит адрес (смещение) команды, которая выполняться следующей

*Регистр флагов (E)FLAGS*

Флаг – один или несколько бит, используемых для хранения состояния какого-либо объекта (обычно 1 бит)

Регистр EFLAGS:

* Размер – 32 бита
* Младшая часть FLAGS – 16 бит

Значения флагов:

* Влияют на поведение инструкций
* Изменяются в ходе выполнения инструкций

Отдельные флаги из (E)FLAGS имеют свои названия

Напрямую из программы недоступен

* Ни весь регистр, ни отдельные флаги

Основные флаги:

* CF – carry flag
* ZF – zero flag:
  + ZF = 1, если результат последний операции равен 0. Иначе ZF = 0
* SF – sign flag:
  + Копия старшего бита результата
* OF – overflow flag:
  + Признак знакового переполнения

Устройство компьютера

CPU – АЛУ,

AX, CX, DX,

EFLAGS, EIP

| |

Адреса Данные

| |

RAM

Fetch - (Из EIP адресу в RAM через данные в [01100101001010101010] в АЛУ, EFLAGS в АЛУ)

Если один из операндов – регистр, его надо подключить к одному из входов в АЛУ.

Какую-то другую часть битов подключаем к АЛУ

Операнд памяти – из инструкции через адрес в RAM получаются данные в АЛУ

Флаги, результат может записан в регистр, или в память (тогда нужен адрес из инструкции, посылается адрес и данные в RAM)

**Основы программирования на языке ассемблер**

*Жизненный цикл программы*

При запуске программы ОС:

* Считывает исполняемый файл в ОЗУ
* Определяет его формат
* Определяет, откуда должно начаться выполнение программы
  + Это место называется «точка входа» (entry point)
* Передает туда управление
  + Т.е. помещает в (E)IP адрес этого места

Понятия код и данные условны

* Код – то, что выполняет процессор
  + А он выполняет то, адрес чего записан в EIP
* Данные – то, с чем работают инструкции
  + Их адреса задаются внутри инструкций
* Для процессора программа – просто массив байтов

Ассемблер (компилятор)

* Генерирует массив байтов, последовательно просматривая текст программы
* Полученный массив записывает «как есть» в выходной файл

Добавлять новые байты могут:

* Инструкции
* Некоторые директивы ассемблера

*С чего начать?*

Программирование сводится к двум основным действиям:

* Подготовка данных
* Вызов стандартных/системных функций

В разных ОС правила вызова системных функций различны

Для вызова системной функции необходимо:

* Подготовить исходные данные (параметры)
* Вызвать функцию

Для каждой функции могут быть свои правила передачи параметров и вызова функции

*Инструкция mov*

Копирует значение операнда-источника в операнд-приемник

Формат:

* mov dest, src

**Не** изменяет регистр **(E)FLAGS**

Оба операнда должны иметь одинаковые размер

* mov ax, bh ; Нельзя!

Одна из немногих команд, операндами которой могут быть сегментные регистры

Нельзя использовать регистр CS в качестве приемника:

* mov cs, ax ; Нельзя!
* mov ax, cs ; Можно

*Соглашение вызова MS-DOS*

В MS-DOS для вызова большинства системных функций используется инструкция int 21h

Большинство параметров передается через регистры

* В регистр AH записывается число, равное номеру вызываемой функции
* В остальные регистры записываются другие параметры функции

org 100h

Start:

mov ah, $09

mov dx, strHello

int 21h

mov ah, $08

int 21h

ret

strHello:

db "hello, world!$"

Директивы db, dw, dd добавляют в файл последовательности байтов, слов, двойных слов.

Примеры:

* db 8, 5, 42, 13, 97
* dw, 19, 10482
* dd 1523231, 1, 453

Символ в апострофах задает число, равное коду этого символа

Эти записи эквивалентны, т.к. код символа ‘A’ равен 65:

db ‘A’

db “A”

db 65

db ‘h’, ‘e’, ‘l’, ‘l’, ‘o’ = db “hello”

strHello: db “Hello, world!” = strHello db “Hello, world!”

Во втором случае компилятор запоминает для метки не только адрес, но и размер данных

Директивы определения данных позволяют использовать операцию dup

Data1 db 5, 5, 5, 5 = Data2 db 4 dup 5

Data3 dw 1, 2, 1, 2, 1, 2 = Data4 dw 3 dup (1, 2)

Если первоначальное содержимое участка памяти не имеет значения, в качестве значения можно указать символ ?

* По возможности компилятор не включает такие байты в исполняемый файл

SomeData db 512 dup(?)

Org – origin

*Директива org*

Указывает ассемблеру значение, с которого следует продолжать вычисление адресов.

Может использоваться много раз.

Для директивы org в файл не добавляются байты

* Но она может влиять на значения некоторых байтов – тех, в которые попадают адреса

*COM- и EXE-программы*

COM-программы:

* При загрузке файл записывается в ОЗУ начиная с адреса 100h
* Управление передается на адрес 100h (т.е. на самый первый байт файла)
* Содержимое COM-файла – дамп памяти, т.е. копия того, что будет находиться в памяти после запуска программы
* Максимальный размер – 64 КБ

EXE-программы:

* Имеет более сложную структуру
* Содержит указания ОС о том:
  + Как производить загрузку и запуск программы
  + Сколько памяти выделить для программы
* Может иметь размер более 64 КБ

***Ветвления и циклы***

*Буфер клавиатурного ввода*

Клавиатурный буфер имеет размер 32 байта

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  |  | … |  |  |

32 байта

*Функции посимвольного ввода*

Функции посимвольного ввода

(01h, 06h, 07h, 08h)

* Для обычных клавиш (символьных):
  + При первом вызове возвращают код символа (не 0)
* Для расширенных клавиш:
  + При первом вызове возвращают 0
  + При втором вызове возвращают расширенный код клавиши
* Если клавиатурный буфер непустой, сразу возвращают управление вызывающей программе
* Если клавиатурный буфер пуст, ожидают нажатия клавиши

JMP = безусловный переход

* Единственный операнд – адрес, с которого нужно продолжить выполнение программы

mov cx, 0

jmp MyLabel

inc cx

MyLabel:

mov ax, cx

Jcc – условный переход

Это семейство команд

ja jb jc

je jg jl

jo jp js

jz jae jbe

jge jle

*Регистр флагов*

Команды, которые вычисляют результат, изменяют биты регистра флагов в зависимости от полученного значения

Команды условного перехода работают в зависимости от состояния тех или иных флагов

Вывод:

* Перед выполнением условного перехода необходимо сделать так, чтобы условие перехода было отражено в одном из флагов (E)FLAGS

Команды условного перехода

Проверяют значения флагов (в зависимости от того, какая команда)

Если соответствующее условия выполняется – перейти по адресу, заданному операндом

В противном случае – продолжить выполнение со следующей команды

And al al

Or al al

Xor al 0 …

CMP и TEST

CMP

* Эквивалентна SUB, но не записывает результат

TEST

* Эквивалентна AND, но не записывает результат

Команды, которые не записывают результат, в общем случае работают быстрее

Метка, имя который начинается с точки, считается локальной

DoSomething:

mov cx, 15

.LocalLabel: (DoSomething.LocalLabel – полное имя)

mov dx, 30

…

jnz .LocalLabel

Метка и именем @@ называется анонимной

jnz @F

mov cx, 20

@@:

mov dx, 40

…

jae @B

*Очистка клавиатурного буфера*

Чтобы гарантировать, что программа будет ожидать нажатия клавиши, необходимо перед началом ожидания очистить клавиатурный буфер.

Соглашение вызова – набор правил, определяющих:

* Способ передачи параметров в функцию
* Способ возврата значения функцией
* Правила использования регистров
* И т. д.

*Соглашение вызова MS-DOS*

В регистр AH помещается номер функции MS-DOS

Параметры передаются в соответствии с описанием функции через регистры (иногда частично)

Вызов производится через int 21h

Результат, если есть, возвращается в соответствии с описанием функции

Значение регистров общего назначения остаются неизменными

Кроме:

* Регистра AX
* Регистров, которые используются функцией для возврата результатов

*Инструкция LOOP*

Используется для организации циклов.

Единственный операнд – адрес, куда надо перейти (по условию).

Принцип работы:

1. CX 🡨 CX – 1
2. Если CX =/ 0 выполнить переход

По сути, делает то же, что и код dec cx; jnz SomeLabel

НО!

* Не изменяет флагов
* Всегда работает с CX

**Арифметические и побитовые операции**

Для сложения и вычитания есть инструкции add и sub

Сложение и вычитание выполняются одинаково для знаковых и беззнаковых чисел

Отличие в работе со знаковыми и беззнаковыми числами в том, какие флаги анализируются

Инструкции условного перехода для беззнаковых чисел:

|  |  |
| --- | --- |
| ja | jna |
| jae | jnae |
| jb | jnb |
| jbe | Jnbe |

Инструкции условного перехода для знаковых чисел:

|  |  |
| --- | --- |
| jg | jng |
| jge | jnge |
| jl | jnl |
| jle | Jnle |

*Операции с переносом*

Для сложения и вычитания больших чисел предусмотрены:

* ADC – Add with Carry
* SBB – SuBtract with Borrow

adc dest, src:

* dest 🡨 dest + (src + CF)

sbb dest, src:

* dest 🡨 dest – (src + CF)

DX:AX – делают одно значение

DX:AX = DX:AX + BX:CX

Add ax, cx

Adc dx, bx

(если в ax и cx переполнение, то в dx + bx + 1)

15  
+ 16  
 (2+CF)1

*Инкремент и декремент*

inc dest

* dest 🡨 dest + 1

dec dest

* dest 🡨 dest – 1

Есть важное отличие от add/sub в работае с флагами

* Найти самостоятельно в Intel SDM

*Умножение и деление*

Умножение и деление разделяют на знаковое и беззнаковое

Умножение

* MUL – беззнаковое
* IMUL – знаковое

Деление:

* DIV – беззнаковое
* IDIV – знаковое

Поведение команд зависит:

* От разрядности операндов
* От количества операндов

*Инструкция MUL*

|  |  |
| --- | --- |
| Операнд | Принцип работы |
| mul r/m8 | AX 🡨 AL \* r/m8 |
| mul r/m16 | DX:AX 🡨 AX \* r/m16 |
| mul r/m32 | EDX:EAX 🡨 EAX \* r/m32 |
| mul r/m64 | RDX:RAX 🡨 RAX \* r/m64 |

*Инструкция DIV*

|  |  |
| --- | --- |
| Операнд | Принцип работы |
| div r/m8 | AL 🡨 AX / src (ост. AH) |
| div r/m16 | AX 🡨 DX:AX / src (ост. DX) |
| div r/m32 | EAX 🡨 EDX:EAX / src (ост. EDX) |
| div r/m64 | RAX 🡨 RDX:RAX / src (ост. RDX) |

*Инструкции IMUL и IDIV*

Работают аналогично, но учитывают знак.

Кроме того, у команды IMUL есть:

* 2-операндная форма
* 3-операндная форма

Подробности в документации Intel

Расширение (extension) – преобразование в эквивалентное число большей разрядности.

Может быть знаковым и беззнаковым.

* Беззнаковые числа дополняются нулями в старших разрядах
* Для знаковых чисел в старшие разряды копируется знаковый бит исходного числа

*Преобразование типов. 1/2*

Инструкции:

* CBW – Convert Byte to Word
* CWD – Convert Word to DWord
* CWDE – Convert Word to DWord Ext.
* CDQ – Convert DWord to QuadWord

Примеры:

* cbw ; AX 🡨 SignExtend(AL)
* cwd ; DX:AX 🡨 SignExtend(AX)

*Преобразование типов. 2/2*

Инструкции:

* MOVZX – Move with Zero-Extend
* MOVSX – Move with Sign-Extend

Примеры:

* movzx dx, bl
* movsx si, ch

Инструкции преобразования типов не изменяют регистр FLAGS

*Команды изменения флагов*

Инструкции:

* STC CF 🡨 1
* CLC CF 🡨 0
* CMC CF 🡨 not(CF)
* CLD DF 🡨 0
* STD DF 🡨 1
* CLI IF 🡨 0
* STI IF 🡨 1

Все инструкции – без операндов

*Побитовые сдвиги:*

* Арифметические
* Логические
* Циклические

Направления сдвигов:

* Влево – в сторону старшего разряда
* Вправо – в сторону младшего разряда

Логический сдвиг влево:

* Освобождающий разряд – 0

CF 🡨 ст. …….приемник….. мл. 0 🡨

Логический сдвиг вправо:

* Освобождающий разряд – 0

🡪 0 ст. ….. приемник …. мл. 🡪 CF

Арифметический сдвиг влево:

* Освобождающий разряд – 0

CF 🡨 ст. …….приемник….. мл. 0 🡨

Арифметический сдвиг вправо:

* Освобождающий разряд – старший разряд исходного значения

🡪🡨 0 ст. ….. приемник …. мл. 🡪 CF (1000 🡪 1100 🡪 1110 🡪 1111)

Циклический сдвиг влево:

* Освобождающий разряд – старший разряд исходного значения

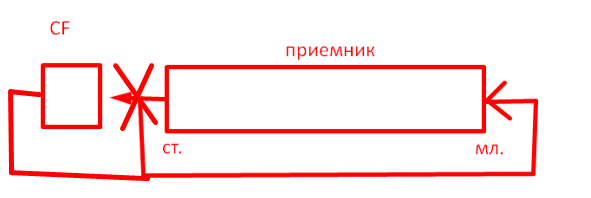


Циклический сдвиг вправо:

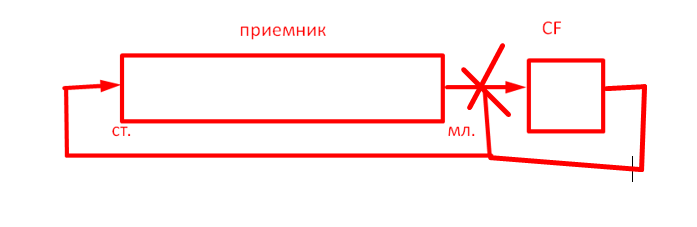
* Освобождающий разряд – младший разряд исходного значения



Циклический сдвиг влево через CF:



Циклический сдвиг вправо через CF:



*SAL SAR SHL SHR*

Инструкции:

* SAL – Shift Arithmetic Left
* SAR – Right
* SHL – Shift logical Left
* SHR – Right

Формат:

* Инструкция dest, count

Count – imm8 или регистр Cl

Флаги:

* Если count = 0, FLAGS не изменяется
* CF – значение последнего выдвинутого («shifted out») бита:
  + Для SHL/SHR, если count больше разрядности dest, значение CF не определено

Начиная с 80286 на значение count накладывается маска 1Fh, поэтому количество бит для сдвига – от 0 до 31

RCL RCR ROL ROR

Инструкции:

* RCL – Rotate through Carry Left
* RCR – Right
* ROL – Rotate Left
* ROR – Right

Формат:

* Инструкция dest, count

Count – imm8 или регистр CL

Флаги:

* CF – значение оставшегося в нем бита
* SF, ZF, PF, AF – не изменяются
* OF – изменяются только для count = 1 (см. документацию Intel)

Начиная с 80286 на значение count накладывается маска 1Fh, поэтому количество бит для сдвига – от 0 до 31

Shld operand 1, operand2, count

Shrd operand1, operand2, count

**Адресация в реальном режиме**

*Понятие сегмента*

Сегмент 0 (64 кб)

Сегмент 1 (64 кб)

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Сегмент0 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| Сегмент0  Сегмент1 | 11 | 11 | 11 | 11 | 11 | 11 | 111 | 11 |
| … | | | | | | | | |
| Сегмент0  Сегмент1 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
| Сегмент1 | 11 | 11 | 11 | 11 | 11 | 11 | 11 | 11 |
|  | … | | | | | | | |

Адрес в виде «сегмент» и «смещение» - это логический адрес, причем полный адрес.

Просто порядковый номер в памяти – физический адрес ( равен Seg\*16 + Ofs (смещение) )

*Сегмент и смещение*

Сегмент – участок памяти размером 64 КБ

Смещение – порядковый номер байта относительно начала сегмента (начиная с 0)

Полный адрес состоит из:

* Номера сегмента (16 бит)
* Смещения (16 бит)

В инструкции обычно задается смещение:

mov ax, [bx + si + $150]

Номер сегмента находится в сегментном регистре.

* Какой сегментный регистр используется, зависит от того, какое действие выполняет процессорв

CS:IP задают полный адрес следующей команды

* Т.е. CS содержит номер сегмента, IP – смещение внутри сегмента

Операнды в памяти по умолчанию задают смещение в сегменте DS

* Если при адресации используется регистр BP, то сегмент – SS  
  mov dx, [bp – 4]
* Можно переопределить используемый сегментный регистр явно: [es:bx + $875]

Один и тот же байт может иметь несколько различных адресов:

$0000:$046C = $0001:$045C = $0046:$000C

При запуске программы ОС настраивает сегментные регистры на свободные области памяти, выбранные для запускаемой программы

Например, COM-программа при запуске будет загружена в произвольный сегмент, но по смещению $0100, т.е. по адресу: $XXXX:$0100

*Эффективный адрес*

Эффективный адрес – смещение операнда в памяти.

При обращении к операнду в памяти вычисляется значение выражения в квадратных скобках

* Это значение и есть эффективный адрес

mov bx, 8

mov si, 6

mov ax, [bx+si+5]

*Инструкция Lea*

LEA:

* Вычисляет эффективный адрес и помещает его в операнд-приемник

Пример:

* lea dx, [dx + di + 42]

Операнды:

* Приемник (первый) – всегда регистр
* Источник (второй) – всегда память

LEA:

* Не выполняет обращение к памяти.
* Это позволяет использовать LEA для вычисления значений выражений

Mov ebx, 13

Mov esi, 48

lea eax, [ebp \* 4 + esi + 63]

Смещение может быть:

* Число
* Один из регистров BX, Bp, SI, DI + число
* Регистр BX или BP  
  + регистр SI или DI  
  + число

*Стек*

Стек – структура данных, работающая по принципу LIFO (Last In, First Out)

В Intel IA-32 стек – область памяти, на которую указывают регистры SS:SP

* Может использоваться для временного хранения значений
* Используется при работе с процедурами

16 бит

|  |  |  |
| --- | --- | --- |
| SS |  | 64 кб |
|  |  |
|  |  |
|  | … |
|  |  |
|  |  |
|  |  |
| SP | TopValue |

Команда push помещает значение операнда на вершину стека

push src

* SP 🡨 SP – 2
* Memory[SS:SP] 🡨 src

push 1

16 бит

|  |  |  |
| --- | --- | --- |
| SS |  | 64 кб |
|  |  |
|  |  |
|  | … |
|  |  |
|  |  |
| SP | 1 |
|  | TopValue |

Инструкция pop извлекает значение с вершины стека

pop dest

* dest 🡨 Memory[SS:SP]
* SP 🡨 SP + 2

pop ax

16 бит

|  |  |  |
| --- | --- | --- |
| SS |  | 64 кб |
|  |  |
|  |  |
|  | … |
|  |  |
|  |  |
|  | 1 |
| SP | TopValue |

ax 🡨 1

При запуске COM-программы в вершине стека находится значение 0

Правила работы со стеком:

* Все помещенные на стек значения должны быть извлечены
* Не следует извлекать больше значений, чем было помещено в стек

PUSHF: (push FLAGS)

* Помещает значение из регистра флагов на вершину стека

POPF: (pop FLAGS)

* Перемещает значение с вершины стека в регистр флагов

PUSHA

* Сохраняет значения всех регистров общего назначения на стек

POPA

* Считывает значения всех регистров общего назначения из стека

*Стек и регистр (E)SP*

push sp

* В Intel 8086 в стек помещается новое значение sp (после уменьшения)
* Начиная с Intel 80286 – старое значение sp (до уменьшения)

pop sp

* Увеличивает значение sp до того, как данные со старой вершины стека будут помещены в sp

Pop [esp]

* Эффективный адрес вычисляется после увеличения esp

**Процедуры**

Процедура – участок кода, который может вызываться из различных мест программы

После выполнения процедуры управление возвращается в точку вызова

*Инструкция CALL*

Выполняет вызов процедуры

Пример:

Call MyProc

…

MyProc:

…

push ip

jmp MyProc

Выполняет 2 действия:

* Помещает в стек значение регистра IP
* Выполняет безусловный переход по адресу, заданному операндом

Инструкция RET

Выполняет 2 действия:

* Извлекает из стека значение
* Помещает его в регистр IP

Если есть операнд (необязательный)

* Извлекает из стека значение (адрес возврата)
* Извлекает из стека N байт (N – операнд)
  + Фактически – просто увеличивает значение SP
* Помещает в регистр IP адрес возврата

В итоге:

…

call MyProc // Вызывающий код

…

MyProc: //

… // Процедура (вызываемый код)

ret //

*Соглашение вызова*

Важно, чтобы обе стороны – вызывающая и вызываемая – соблюдали одно и то же **соглашение вызова**.

Передача параметров:

* Через регистры (кол-во регистров ограничено)
* Через память (глобальные переменные)
* Через стек (наш слоняра).

*Возврат значений*

* Через регистры (одни плюсы, еще один слоняра)
* Через память (press F to рекурсия)
* Через стек (особо плюсов нет)

Процедуры

Будем использовать:

* Передачу параметров через стек;
* Возврат значения функции через регистр.

Напишем функцию, эквивалентную…

function Sum(a, b: Integer): Integer;

begin

Result := a – b;

end;

Пролог – часть процедуры, которая подготавливает процедуру к работе с параметрами.

Эпилог – часть процедуры, которая восстанавливает состояние, необходимое для продолжения выполнения вызывающего кода.

(((Mov bx, sp

Mov ax, [ss:bx+4]

=

Mov bp, sp

Mov ax, [bx+4])))

MyProc:

Push bp

Mov bp, sp

Sub sp, sizeof(localVars)

…

Mov sp, bp

Pop bp

Ret sizeof(params)

Для приведенного кода состояние стека будет таким:

…

[bp+6] Параметр 2

[bp+4] Параметр 1

[bp+2] Адрес возврата

[bp] Старое значение BP

[bp-2] Локальная переменная 1

[bp-4] Локальная переменная 2

…

*Соглашение вызова*

Соглашение вызова – набор правил, определяющих:

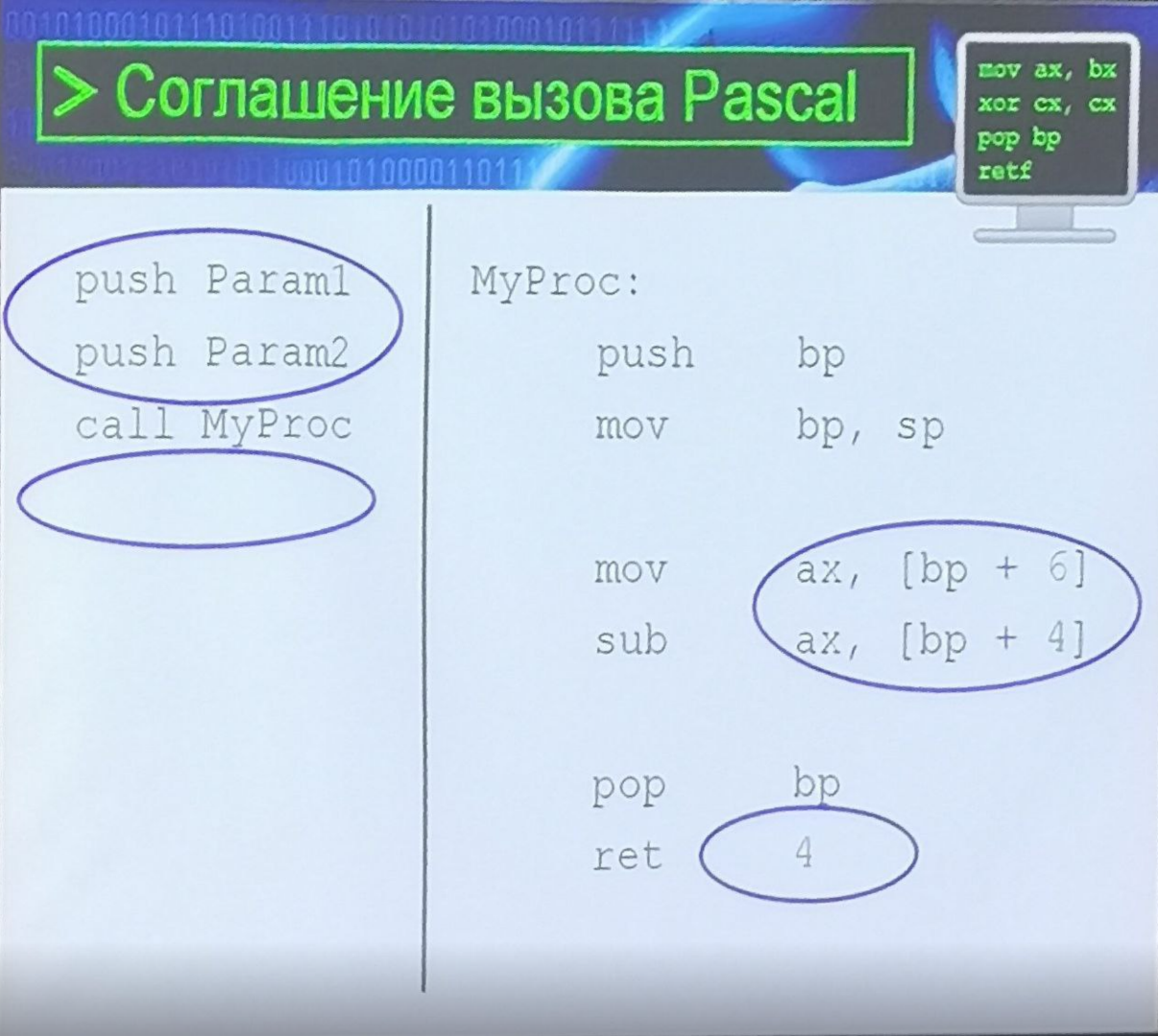
* Способ передачи параметров в функцию
* Способ возврата значения функцией
* Правила использования регистров
* Правила удаления параметров со стека (если передаются через стек);
* И т.д.

Соглашения вызова:

* Pascal (юзает паскаль)
* Ccall (Используется компиляторами C/C++)
* Stdcall (Используется функциями Windows)
* Fastcall (Используется компиляторами для оптимизации)
* Comcall/safecall (Используется при вызове методов COM-итерфейсов)
* И др.

Соглашение вызова pascal:

* Параметры в стек помещаются слева направо
* Очистку стека выполняет вызываемая процедура

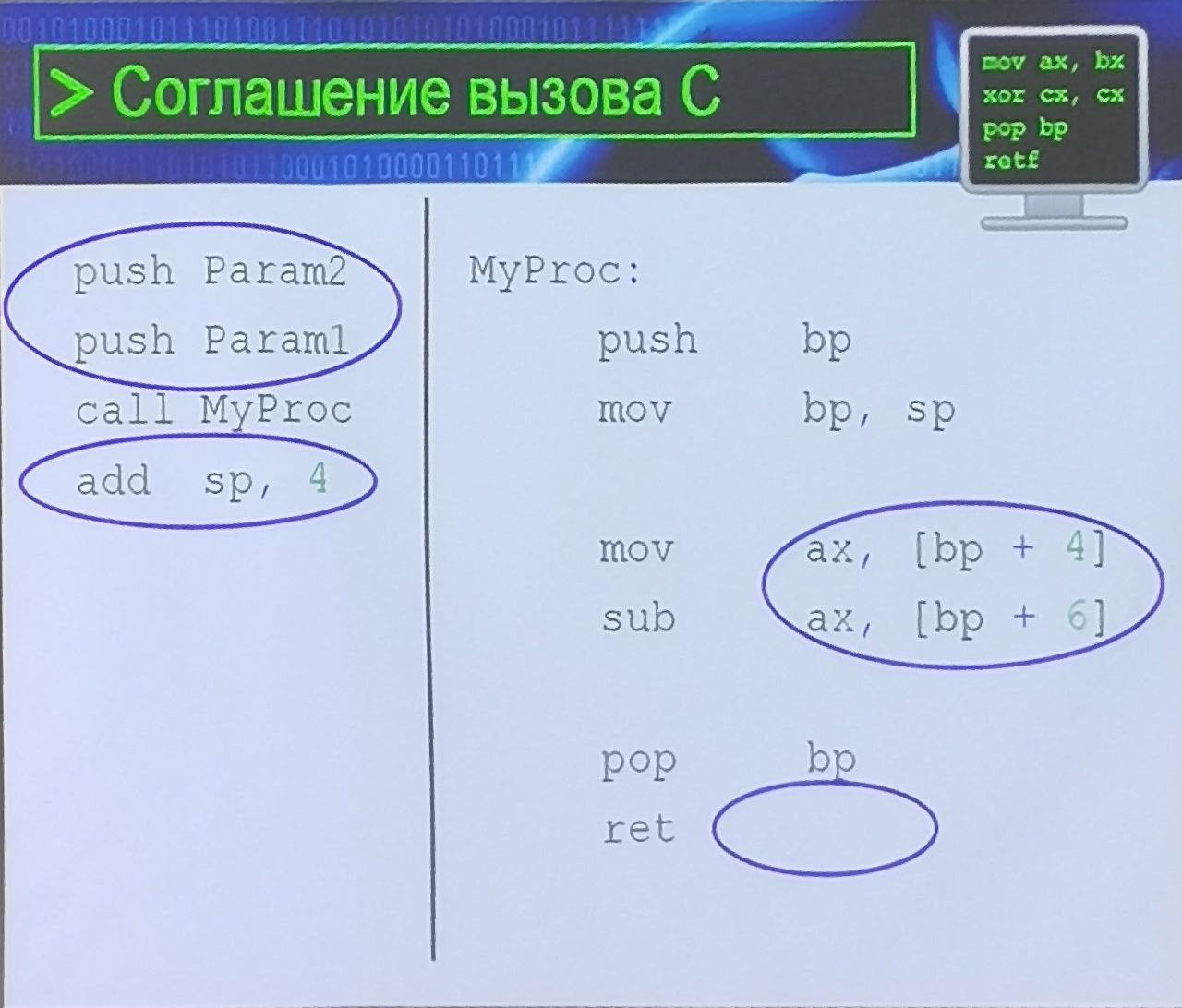


|  |  |
| --- | --- |
| Push Param1  Push Param2  Call MyProc | MyProc:  Push bp  Mov bp, sp  Mov ax, [bp + 6]  Sub ax, [bp + 4]  Pop bp  Ret 4 |

Соглашение вызова C:

* Параметры в стек помещаются справа налево.
* Очистку стека выполняет вызывающая процедура.

|  |  |
| --- | --- |
| Push Param2  Push Param1  Call MyProc  Add sp, 4 | MyProc:  Push bp  Mov bp, sp  Mov ax, [bp + 4]  Sub ax, [bp + 6]  Pop bp  Ret |



Stdcall:

Параметры в стек помещаются справа налево.

Очистку стека выполняет вызываемая процедура.

|  |  |
| --- | --- |
| Push Param2  Push Param1  Call MyProc | MyProc:  Push bp  Mov bp, sp  Mov ax, [bp + 4]  Sub ax, [bp + 6]  Pop bp  Ret 4 |

Pascal, ccall, stdcall:

* Процедура должна оставить нетронутыми значения регистров BX, SI, DI, BP.
* Процедура может изменять значения AX, CX, DX.
* Регистр SP изменяется в результате манипуляций со стеком.
* Для возврата значения из функции используется AX

**Строковые команды**

Строка – последовательность (массив) символов

Строка (как и любой массив) имеет 2 основные характеристики:

* Адрес (где находится?)
* Длина (сколько места занимает?)

Проще всего работать с данными, размер которых равен размеру регистра общего назначения

**Как хранить строковые данные?**

Unicode – стандарт кодирования символов, позволяющий представить знаки почти всех письменных языков.

Существуют различные способы представления (Unicode Transformation Formats):

* UTF-8
* UTF-16 (UTF-16LE, UTF-16BE)
* UTF-32 (UTF-32LE, UTF-32BE)
* UTF-7

Отличие UCS-2 и UTF-16 – …

BOM-байты это – …

Строковые команды:

* CMPS
* INS
* LODS
* LODS
* STOS
* MOVS
* SCAS
* CMPS
* INS
* OUTS

Могут работать с данными размером 1, 2, 4 или 8 байт.

Загружает элемент из памяти в регистр.

* LODSB: mov al, [ds:si]
* LODSW: mov ax, [ds:si]

Загружает:

* Всегда в AL/AX;
* Всегда из памяти по адресу DS:SI

Записывает элемент из регистра в память.

* STOSB: mov [es:di], al
* STOSW: mov [es:di], ax

Записывает:

* Всегда из AL/AX
* Всегда в память по адресу ES:DI

Копирует элементы в память.

* MOVSB: mov [es:di], [ds:si]
* MOVSW: mov [es:di], [ds:si]

Записывает:

* Всегда из памяти по адресу DS:SI
* Всегда в память по адресу ES:DI

Сравнивает значение регистра с элементом в памяти.

* SCASB: cmp al, [es:di]
* SCASW: cmp ax, [es:di]

Сравнивает:

* Всегда значение регистра AL/AX
* Всегда с элементом по адресу ES:DI

Сравнивает значения двух элементов в памяти.

* CMPSB: cmp [ds:si], [es:di]
* CMPSW: cmp [ds:si], [es:di]

Сравнивает:

* Всегда элемент по адресу DS:SI
* Всегда с элементом по адресу ES:DI

Кроме копирования/сравнения, все строковые команды изменяют значения регистров SI и/или DI.

* LODS: SI
* STOS: DI
* MOVS: SI и DI
* SCAS: и DI
* CMPS: SI и DI

Значения SI и/или DI изменяются в большую или меньшую сторону в зависимости от флага DF:

* DF = 0: увеличивается
* DF = 1: уменьшается

Значения изменяются на количество байт, с которым работает инструкция.

|  |  |  |
| --- | --- | --- |
| LODS: (максимально бесполезен) | | |
| mov reg, [ds:si] | SI 🡨 SI +- n |  |
| STOS: (заполнение) | | |
| mov [es:di], reg |  | DI 🡨 DI +- n |
| MOVS: (перемещение из одного в другое) | | |
| mov [es:di], [ds:si] | SI 🡨 SI +- n | DI 🡨 DI +- n |
| SCAS: (поиск первого значения которые не/совпадет) | | |
| cmp reg, [es:di] |  | DI 🡨 DI +- n |
| CMPS: (сравнить двух областей памяти) | | |
| cmp [ds:si], [es:di] | SI 🡨 SI +- n | DI 🡨 DI +-n |

Перед строковыми инструкциями могут указываться префиксы.

* REP
* REPE/REPZ (пока равно)
* REPNE/REPNZ (пока не равно)

REP:

* Перед LODS, STOS и MOVS

REPE/REPZ и REPNE/REPNZ:

* Перед SCAS и CMPS

Если перед стоковой командой указан REP-префикс, команда будет повторяться несколько раз

|  |  |  |
| --- | --- | --- |
| Префикс | Условие 1 | Условие 2 |
| REP | CX = 0 | НЕТ |
| REPE/REPZ | CX = 0 | ZF = 0 |
| REPNE/REPNZ | CX = 0 | ZF = 1 |

С помощью строковых инструкций можно реализовать работу:

* Со строками
* С массивами скалярных типов

**Работа с видеопамятью**

Видеоадаптер – устройство, преобразующее графической образ в ОЗУ или памяти самого адаптера в форму, пригодному для вывода на экран.

* Современные видеокарты умеют намного больше

*Видеорежимы*

Видеоадаптер находится в одном из поддерживаемых им режимов.

Режимы:

* Текстовые
* Графические

Все режимы пронумерованы.

* Для некоторых режимов характеристики известны заранее
* Для отдельных режимов их можно узнать, взаимодействуя с видеоадаптером

Видеопамять – область памяти, отведенная для хранения данных, которые используются для формирования изображения на экране.

* В первых видеоадаптерах хранилось растровое изображение
* В современных видеокартах могут храниться также векторные данные (например, в виде массивов треугольника)

*int 10h*

В зависимости от выбранного режима видеоадаптер интерпретирует содержимое видеопамяти по-разному.

Для смены режимов и управления другими параметрами видеоадаптера в реальном режиме используют int 10h

Соглашение вызова у функций предоставляемых через int 10h:

* Параметры передаются через регистры, для крупных типов – указатель на область памяти
* Значение всех регистров, кроме BX, CX и DX, могут быть изменены.

Функции int 10h – функции

Для работы с режимами видеоадаптера есть 2 функции:

00h – установить требуемый режим

0Fh – получить информацию о выбранном