Практическая работа №6

Тема: Построение алгоритмов для ассемблерных программ

Цель занятия: научиться писать простые программы на языке Ассемблер.

Основные требования по технике безопасности при выполнении практической работы: изучить правила техники безопасности, руководствоваться ими и обеспечить их строгое соблюдение при проведении учебного процесса.

Краткие теоретические сведения, необходимые для выполнения практической работы:

1. Организация ассемблерной программы

Пользовательская (прикладная) программа, написанная для платформ IA-32/Linux или IA-32/Windows, рассчитана на выполнение в машине, реализующей принципы фон Неймана (Рис. 1). Машинные команды и данные хранятся совместно в оперативной памяти, а процессор в автоматическом режиме последовательно читает из памяти команды и исполняет их. Оперативная память разбита на ячейки размером по 8 двоичных разрядов (1 байт), ячейки пронумерованы числами от 0 до 232-1 (номер ячейки называют ее адресом). Процессор содержит набор 32-х разрядных регистров, состоящий из восьми регистров общего назначения, счетчика команд, регистра состояния. Взаимодействие процессором с оперативной памятью осуществляется через шину, позволяющей процессору считывать значения ячеек оперативной памяти, записывать в ячейки новые значения.

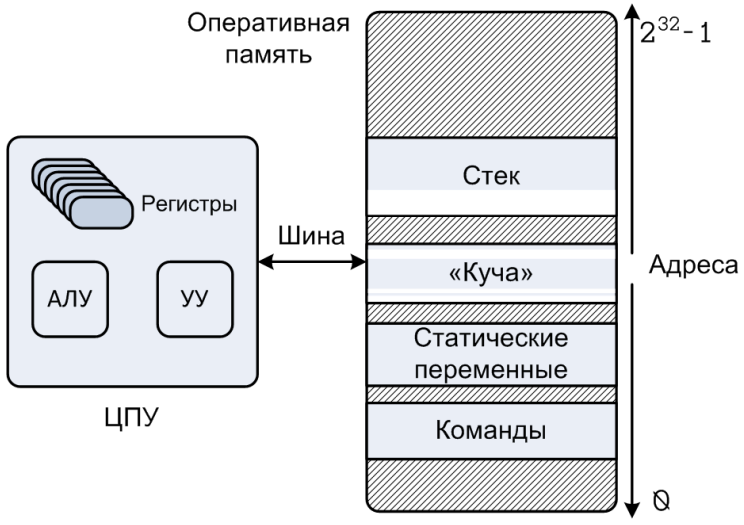


Рисунок 1 - Упрощенная схема виртуальной машины IA-32, используемой прикладной программой

Программа, которая готова к исполнению процессором, представляет собой определенный набор значений ячеек оперативной памяти. Эти значения заносит в оперативную память загрузчик — компонент операционной системы, отвечающий за загрузку исполняемого файла (копирование его содержимого) в память. Для формирования исполняемого файла используются инструменты системы программирования; основными инструментами являются компилятор, ассемблер, компоновщик (Рис. 2).

Машина IA-32 позволяет обращаться к произвольному месту оперативной памяти, однако не все ячейки доступны, обращения к одним адресам завершатся успешно, к другим – приведут к аварийной остановке программы. Причиной является то, что пользовательская программа работает в защищенном режиме процессора: фактически ей доступны только определенные диапазоны адресов, содержащие образ программы в памяти. Управление памятью, доступной пользовательской программе, осуществляет операционная система, этот вопрос выходит за рамки данного курса.

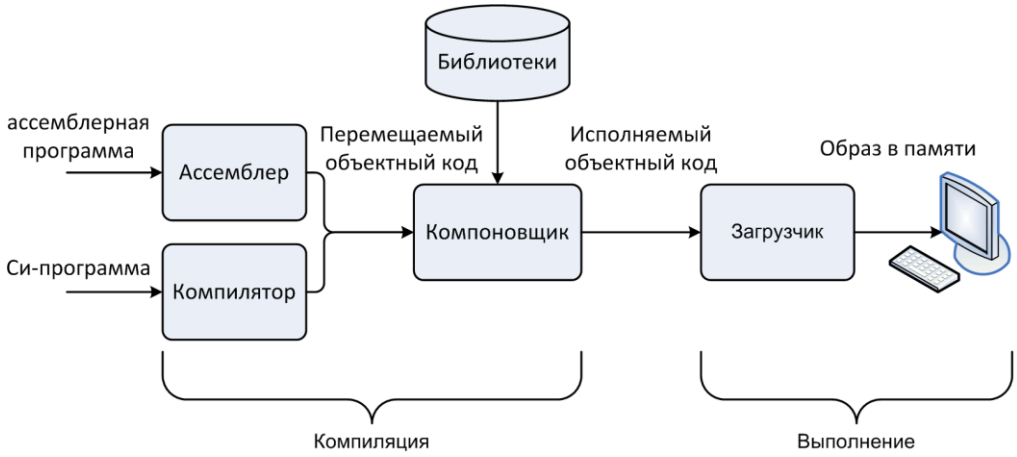


Рисунок 2 - Упрощенная схема компиляции и выполнения прикладной программы.

Язык ассемблера позволяет программисту описывать содержимое ячеек оперативной памяти не в виде двоичных чисел, а в виде более удобных для человека конструкций и мнемонических обозначений. Программа на языке ассемблера представляет собой текстовый файл. Программист определяет в этом текстовом файле набор секций. В результате перевода (трансляции) текста программы каждая секция будет превращена в последовательность байт. При запуске программы каждая такая последовательность байт будет загружена в оперативную память и размещена в выделенных для нее ячейках памяти. Операционная система выделяет отдельные блоки памяти под машинные команды, константы, переменные трех классов памяти (в терминологии языка Си): статические, автоматические, динамические. Секции ассемблерного файла содержат только машинные команды, константы и статические переменные, память для «кучи» и автоматических переменных в начальный момент работы

программы выделяется операционной системой. Эта память доступна для использования, но изначально ничем не заполнена, т.е. содержит произвольные значения.

Способ записи машинных команд и определения данных в секциях исходного (текстового) файла определяется синтаксисом ассемблера. Для архитектуры IA-32 существует два основных синтаксиса: AT&T и Intel. Каждая конкретная реализация ассемблера вносит свои изменения в синтаксис, образуя диалект.

В рамках данного курса рассматривается ассемблер NASM (Netwide Assembler), версии которого существуют под большое количество платформ, включая Microsoft Windows и различные виды UNIX-систем. Скачать ассемблер и документацию к нему можно с официального сайта http://www.nasm.us/.

Программы на языке ассемблера представляют собой текстовые файлы с расширением .asm. Как уже говорилось, файл состоит из нескольких секций, в которых размещается код программы (последовательность ассемблерных инструкций) и её данные.

В простых программах набор секций, как правило, ограничен тремя (не считая служебных):

- секция кода, которая обычно называется .text;
- секция инициализированных данных (то есть тех, для которых определено начальное значение), которая обычно называется .data;
- секция неинициализированных данных, значения которых обнуляются операционной системой перед запуском программы; эта секция обычно называется .bss.

В ассемблерном файле могут присутствовать и другие секции. Например, может быть несколько секций кода или данных, однако мы такие программы рассматривать не будем. Программа воспринимается построчно. Каждая строка, как и во многих других ассемблерах, представляет собой последовательность следующих полей:

- метка — присваивает имя данному месту в программе, вне зависимости от того, что на этом месте расположено (код/данные);
- ассемблерная инструкция, состоит из кода операции и операндов (если они есть), перечисляемых через запятую;
- директива определения данных;
- комментарий.

Допустимы пустые строки. Кроме того, на отдельных строках могут быть записаны служебные директивы, указывающие ассемблеру, каким образом следует размещать код и данные в памяти.

2. Директивы определения данных

Строка с описанием данных имеет следующий вид:

```
имя_переменной [:]    директива_определения_данных    ; комментарий
```

При описании данных, после имени переменной может присутствовать двоеточие; фактическое использование ассемблером символьного имени переменной в точности совпадает с использованием меток, помечающих код.

Для определения переменных с начальными значениями используются директивы DB, DW, DD и DQ. Например:

```
имя_переменной DD значение1[, значение2, ... ]
```

Директива DB предназначена для определения данных размером в байт, DW, DD и DQ определяют данные размером соответственно в слово (2 байта), двойное слово (4 байта) и учетверенное слово (8 байтов). Например:

```
x  dw  -1        ; Определение переменной x в формате слова с
                 ; начальным значением -1
y  dd  1, 2, 3   ; Определение трех двойных слов с начальными
                 ; значениями 1, 2, 3
```

Последняя директива описывает тот факт, что в памяти, начиная с адреса y, последовательно размещаются три двойных слова с указанными значениями, при этом первое двойное слово располагается в памяти по адресу y, второе – по адресу y+4, третье – по адресу y+8.

В общем случае, через запятую перечисляется набор значений, который будет размещен в памяти, начиная с адреса, помеченного как имя_переменной.

NASM не позволяет указывать неопределенное начальное значение (в MASMe для этого служил знак ?). Если начальное значение переменной не важно, ее следует располагать в секции .bss, все байты которой инициализируются нулем. Такая особенность позволяет экономить место в файле с исполняемым кодом: содержимое секции заранее известно, достаточно хранить только ее размер. При объявлении переменных в секции .bss необходимо использовать соответствующие директивы: RESB, RESW, RESD и RESQ, имеющие следующий формат.

```
имя_переменной RESB количество_ячеек
```

Под количеством ячеек понимается число байт, слов, двойных или четверных слов, в зависимости от использованной директивы.

```
a  resd  1   ; Выделено место для одной переменной, ее размер –
             ; двойное слово, начальное значения – 0
b  resb  20  ; Выделено место для последовательно размещенных
             ; 20 байт
c  resw  256 ; Выделено место для 256 слов
```

Имена a, b и c являются адресами, начиная с которых размещены обнуленные данные.

В качестве конструкции повторения в ассемблере NASM используется префикс TIMES (в отличие от DUP в MASMe):

```
TIMES количество_раз повторяемая конструкция
```

Пример – требуется объявить переменную с именем zerobuf, представляющую собой буфер размером в 64 байта и заполненный нулями.

```
zerobuf times 64 db 0
```

Аргумент конструкции TIMES не константа, а вычисляемое выражение, что позволяет реализовывать, например, следующее:

```
buffer: db 'hello, world!'
        times 64-$+buffer db ' '
```

Начиная с метки buffer будет выделено 64 байта, первые байты будут заполнены заданной строкой, остальные – пробелом. Лексема $ соответствует текущей позиции в транслируемом коде. Выражение $-buffer, записанное непосредственно после первой строки, будет содержать длину строки 'hello, world!'.

Константы

Ассемблер NASM поддерживает несколько типов констант: целочисленные, символы, строки и числа с плавающей точкой.

У целочисленных констант поддерживаются различные основания: десятичное, двоичное, восьмеричное, шестнадцатеричное. Для явного задания основания следует воспользоваться соответствующими суффиксами: d, b или y, o или q, h. По умолчанию последовательность цифр рассматривается как десятичное число. Помимо того, допустимы формы задания основания в виде префиксов: 0d – десятичное 0b – двоичное, 0o – восьмеричное, 0h – шестнадцатеричное. Допускается запись шестнадцатеричных чисел как в Си-программах, с префиксом 0x.

Целочисленные константы могут содержать символ подчеркивания для разделения длинных последовательностей цифр.

```
        mov     ax,200          ; десятичное
        mov     ax,0200d        ; явно указанное десятичное
        mov     ax,0c8h         ; шестнадцатеричное
        mov     ax,0xc8         ; шестнадцатеричное
        mov     ax,310q         ; восьмеричное
        mov     ax,11001000b    ; двоичное
        mov     ax,1100_1000b   ; двоичное
```

Во всех случаях приведен один и тот же код.

Символьная константа содержит от одного до восьми символов, заключенных в прямые, обратные или двойные кавычки. Тип кавычек для NASM несущественен, поэтому если используются одинарные кавычки, двойные могут выступать в роли символа и, соответственно, наоборот. Обратные кавычки позволяют использовать специальные символы языка Си.

Символьная константа, состоящая из одного символа, эквивалентна целому числу, равному коду этого символа. Символьная константа, содержащая более одного символа, будет транслирована посимвольно в обратном порядке следования байтов: 'abcd' эквивалентно не 0x61626364, а 0x64636261. Эта особенность обусловлена порядком хранения байтов целого числа в памяти: сначала хранятся младшие байты, за ними — старшие. Таким образом, если записать эту константу в память, а затем прочитать побайтово, получится снова abcd, но не dcba.

Строковые константы допустимы только в директивах db/dw/dd/dq. От символьных констант они отличаются только отсутствием ограничения на длину и интерпретируются как сцепленные друг с другом символьные константы максимального допустимого размера.

```
dd 'ninechars'       ; строковая константа – последовательность
                     ; двойных слов
dd 'nine','char','s' ; явно заданы три двойных слова
db 'ninechars',0,0,0 ; последовательность байт
```

Во всех случаях определены одни и те же данные.

Классы памяти

В стандарте языка Си определены три класса памяти (storage duration): статическая, автоматическая, динамическая. К статическому классу памяти относятся глобальные и статические переменные. В зависимости от того, как выполняется инициализация, переменные этого класса помещаются компилятором либо в секцию .data, либо в секцию .bss. Такие вопросы, как размещение автоматических локальных переменных и работа с динамической памятью, в этой части пособия не рассматриваются.

Пример 1 Минимальная программа

Требуется написать минимальную ассемблерную программу.

Решение

```
%include 'io.inc'  ; (1)
section .text      ; (2)
                   ; (3)
global CMAIN       ; (4)
CMAIN:             ; (5)
   MOV EAX, 0      ; (6)
   RET             ; (7)
```

Данная ассемблерная программа ничего не делает и при запуске практически сразу возвращает управление операционной системе. Разберём её построчно. Первая строка является директивой ассемблера, она требует включения в текст программы текста файла io.inc, в котором содержатся команды ввода/вывода. Эту строчку следует рассматривать как аналог строки “#include <stdio.h>” в Си- программах, однако существует некоторое отличие. Стандарт языка Си определяет набор функций, осуществляющих ввод/вывод. Для языка ассемблера таких стандартных функций нет, для облегчения разработки учебных программ был подготовлен файл io.inc, содержащий набор команд ввода/вывода.

Следующая, вторая, строка содержит директиву, указывающую, что последующие строки относятся к секции кода программы (конкретнее — к секции .text). Третья строка оставлена пустой для наглядности отделения директивы от остального текста ассемблерной программы.

В четвертой строке содержится директива, предписывающая сделать имя CMAIN видимым «снаружи» программы. В данном случае указывается имя CMAIN, которое определено в файле io.inc и является точкой входа в программу. На пятой строке это имя используется в метке, управление при запуске будет передано помеченной этим именем инструкции на строке 6. Таким образом, метка CMAIN начинает описание функции с соответствующим именем. Эту функцию следует считать полным аналогом функции main в Си-программах.

На шестой строке выполняется инструкция MOV, в которой в регистр EAX помещается число 0. На следующей строке выполняется инструкция RET, завершающая выполнение функции.

Рассмотренный пример можно соотнести со следующей Си-программой.

```
#include <stdio.h>

int main () {
   return 0;
}
```

Инструкция 6 в ассемблерной программе соответствует оператору return 0; Она необходима для правильной обработки программы в системе автоматического приема задач – правильно отработавшая программа должна возвращать число 0.

Регистры общего назначения

Процессор IA-32 содержит восемь 32-разрядных регистров общего назначения: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP (Рис. 3). Каждый из них допускает непосредственный доступ к своей младшей половине по имени соответственно AX, BX, CX, DX, SI, DI, BP, SP. Таким образом, программист имеет возможность работать с 16-разрядными регистрами с указанными именами. Кроме того, регистры AX, BX, CX, DX в свою очередь допускают независимый доступ к своей младшей и старшей половине по именам соответственно AL, AH, BL, BH, CL, CH, DL, DH, обеспечивая возможность работать уже с 8-разрядными регистрами. Буква L в имени регистра обозначает младшую половину соответствующего 16-разрядного регистра, буква H – старшую.

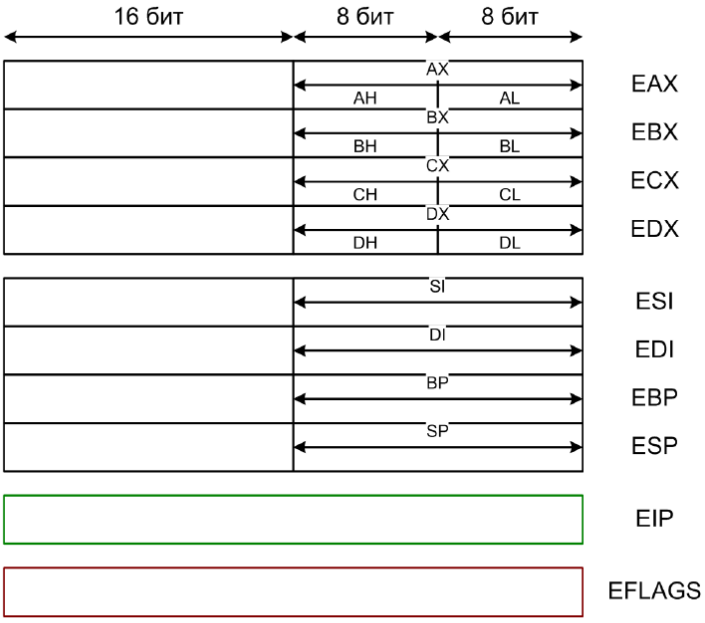


Рисунок 3 – Основные регистры IA-32.

Пересылка данных

Самая простая и часто используемая инструкция архитектуры IA-32 — инструкция пересылки MOV. Она имеет всегда два операнда, которые должны подходить под один из следующих форматов:

- MOV регистр, константа — запись значения константы в регистр;
- MOV регистр-1, регистр-2 — запись значения из регистра-2 в регистр-1;
- MOV регистр, память — запись в регистр значения из памяти;
- MOV память, регистр — запись в память значения из регистра;
- MOV память, константа — запись значения константы в память.

Необходимо обратить внимание на то, что целевой операнд находится слева (как бы перед «присваиванием»), а исходный — справа. Данный порядок операндов является одной из характерных черт синтаксиса Intel. Кроме того, важно, что в инструкции MOV размеры операндов обязаны совпадать (то есть нельзя переслать в регистр AX из регистра EBX).

```
%include 'io.inc'  ; (1)
section .text      ; (2)
                   ; (3)
global CMAIN       ; (4)
CMAIN:             ; (5)
   MOV EAX, 1      ; (6) EAX := 1
   MOV EBX, EAX    ; (7) EBX := EAX = 1
   MOV CL, 040h    ; (8) CL := 040h = 0x40 = 64
   MOV EAX, 0      ; (9)
   RET             ; (10)
```

Команда XCHG выполняет обмен значений своих операндов (операнды обязательно должны быть одинакового размера). Допустимы следующие форматы этой команды:

- XCHG регистр-1, регистр-2
- XCHG регистр, память
- XCHG память, регистр

Обращение к памяти

Обращение к операнду в памяти в простейшем случае имеет вид: спецификатор размера [имя переменной], где спецификатор размера (dword, word или byte) задает размер соответствующей переменной в памяти. Например, dword [a] – обращение к переменной a в формате двойного слова (32 бита).

В записи операнда важную роль играют квадратные скобки вокруг имени переменной. Именно такая запись трактуется в NASM как содержимое ячейки памяти по указанному адресу, т.е. значение по данному адресу, тогда как просто имя переменной трактуется как адрес соответствующей ячейки памяти. Сравните:

```
mov eax, dword [a] ; В регистр eax помещается значение переменной a
mov eax, a         ; В регистр eax помещается адрес переменной a
```

При выполнении второй команды обращения к памяти не происходит.

Сложение и вычитание

Арифметические инструкции сложения и вычитания для целых чисел называются ADD и SUB. У них нет отдельных версий для знаковых и беззнаковых целых чисел, они применимы в обоих случаях.

```
ADD EBX, EAX
; EBX := EBX + EAX = 1 + 1 = 2, EAX не меняется

ADD EBX, EBX
; EBX := EBX + EBX = 2 * EBX = 4

SUB EAX, 2
; EAX := EAX - 2 = 1 - 2 = -1 = 0FFFFFFFFh

ADD AX, 1
; AX := AX + 1 = 0FFFFh + 1 = -1 + 1 = 0
; EAX стал равен 0FFFF0000h
```

Первый операнд ADD и SUB — целевой. Это тот операнд, к значению которого будет прибавлено (или от которого будет отнято) значение второго операнда. Результат операции также записывается в первый операнд. Целевой операнд может быть регистром или операндом в памяти. Второй операнд (то, что прибавляется или отнимается) может быть регистром, операндом в памяти или константой.

Важно: ни в какой инструкции, кроме строковых, не может быть два операнда в памяти.

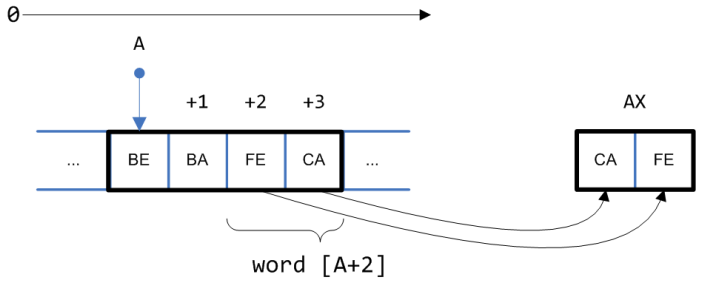
Пример 2 Определение значения регистра

Пусть ассемблерная переменная A имеет значение 0x CAFE BABE. Требуется выписать в шестнадцатеричном виде значение регистра AX после выполнения следующих инструкций.

```
MOV AX, WORD [A + 2]
ADD AX, 3
```

Решение

Рассмотрим расположение в памяти переменной A и определим, что будет в регистре AX, после выполнения первой инструкции. Поскольку в архитектуре IA-32 используется обратное расположение байтов в памяти, то получаем следующее:



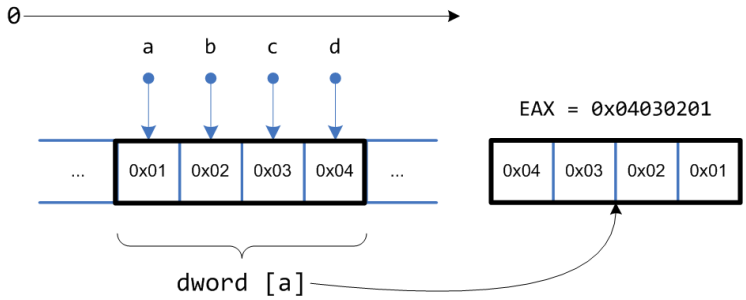
Байт с адресом A+2 будет иметь значение 0xFE, следующий за ним – 0xCA. При пересылке в регистр байты поменяются местами, и регистр AX будет иметь значение 0xCAFE. После того, как к этому значению будет прибавлено 3, оно станет равным 0xCB01.

Пример 3 Переворот байтов в двойном слове

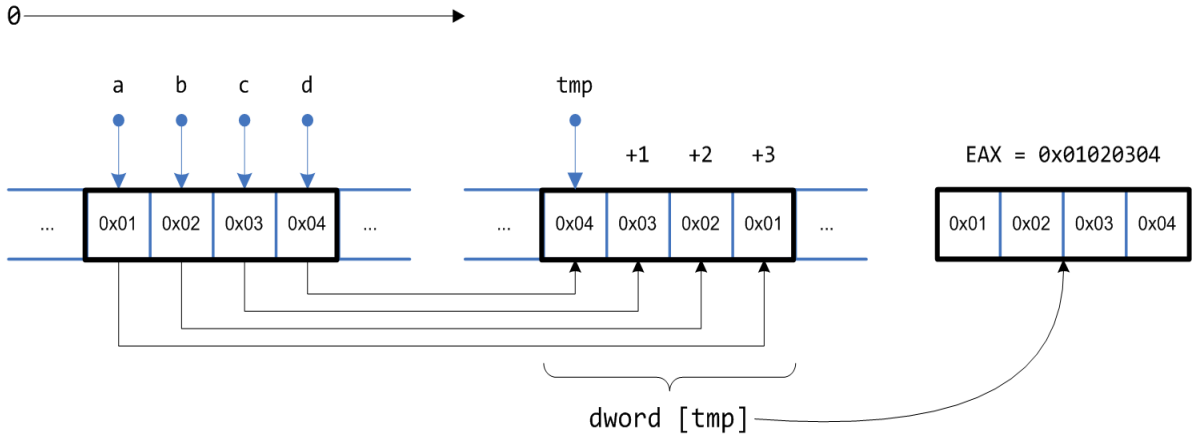
В памяти последовательно расположены 4 переменных a, b, c и d размером 1 байт каждая в заданном порядке. Требуется сформировать 32-битное число в регистре EAX таким образом, чтобы старший байт числа совпадал со значением переменной a, следующий за ним байт – со значением b, следующий – со значением c, и, наконец, младший байт – со значением d. Пусть, для примера, значения a, b, c и d равны 1, 2, 3, 4 соответственно. Тогда в регистре EAX будет размещено число 0x01020304.

Решение

Требуемый порядок размещения байтов является обратным по отношению к тому, который получится при считывании 32-разрядного числа с адреса a.



Необходимо сформировать число в регистре с обратным порядком байтов. Воспользуемся для этого вспомогательной статической переменной и переместим отдельные байты таким образом, что при считывании из нее двойного слова будет получаться необходимый порядок байтов в регистре.



```
section .bss
   tmp resd 1

section .data
   a db 1
   b db 2
   c db 3
   d db 4

section .text
   mov al, byte [a]
   mov byte [tmp + 3], al
   mov al, byte [b]
   mov byte [tmp + 2], al
   mov al, byte [c]
   mov byte [tmp + 1], al
   mov al, byte [d]
   mov byte [tmp], al
   mov eax, dword [tmp]
```

Средства ввода/вывода

Для считывания данных, вводимых с клавиатуры, и печати данных на экран (консоль) предлагается набор команд, определенных в файле io.inc. Они позволяют считывать и печатать числа в шестнадцатеричном и десятичном формате, строки и отдельные символы. Фактически, приведенные команды являются макросами, содержащими обращения к функциям стандартной библиотеки ввода/вывода языка Си.

Таблица 1. Сводная таблица макросов ввода/вывода.

Имя команды	**Описание макроса**
`PRINT_UDEC` *size, data* `PRINT_DEC` *size, data*	Вывод числовых данных заданных параметром *data* в 10-чном представлении. Параметр *size* – число, указывающее размерность данность в байтах; допускаются значения 1, 2, 4. В качестве параметра *data* может выступать числовая константа, символьная константа, имя переменной, имя регистра или адресное выражение (без спецификатора размера данных в памяти). Если задается регистр большего размера, то берется заданное параметром *size* количество младших разрядов. `PRINT_UDEC` интерпретирует число как беззнаковое, `PRINT_DEC` — как знаковое.
`PRINT_HEX` *size, data*	Аналогично предыдущему, но данные выводятся в 16-чном представлении.

PRINT_CHAR *ch*	Печатается символ, заданный параметром *ch*. В качестве параметра может выступать числовая константа, символьная константа, имя переменной, имя регистра или адресное выражение (без спецификатора размера данных в памяти). Печатается всегда содержимое 8 младших разрядов.
PRINT_STRING *data*	Печать строки текста, оканчивающейся символом с кодом 0. В качестве параметра можно передавать строковую константу, имя переменной или адресное выражение (без спецификатора размера данных в памяти). В случае печати строковой константы, наличие символа с кодом 0 в конце строки необязательно.
NEWLINE	Макрос переводит печать на новую строку.
GET_UDEC *size, data* GET_DEC *size, data*	Ввод числовых данных в 10-чном представлении с клавиатуры. Размер вводимых данных ограничен параметром *size*, который задается числом (1, 2, 4). Введенные данные обрезаются соответствующим образом. Параметр *data* – либо имя переменной, либо имя регистра, либо адресное выражение (без спецификатора размера данных в памяти). Если задается регистр большего размера, то старшие разряды заполняются знаковым битом в случае GET_DEC и нулями в случае GET_UDEC. GET_UDEC считывает беззнаковое число, GET_DEC — знаковое. Запрещается использовать в качестве параметра регистр ESP.
GET_HEX *size, data*	Аналогично предыдущему, но данные задаются в 16-чном представлении с префиксом 0x.
GET_CHAR *data*	Аналогично предыдущему, но происходит считывание одного символа, нажатие Enter не требуется. Более того, нажатие Enter будет расцениваться как ввод управляющих символов 0xD 0xA. Если параметр – регистр, размер которого больше 1 байта, значение считанного символа будет дополнено нулями.
GET_STRING *data, maxsz*	Ввод последовательности символов, оканчивающейся символом перевода строки (включая этот символ), но не более чем *maxsz* символов. Параметр *data* – либо имя переменной, либо адресное выражение (без спецификатора размера данных в памяти). Параметр *maxsz* – числовая константа. В конец строки добавляется символ с кодом 0.

Пример 4 Hello, World!

Требуется написать программу «Hello, World!» на языке ассемблера.

Решение

Программа расширяет Пример 1: помимо секции .text, используется секция инициализированных статических данных .data, в которой размещена последовательность байт с текстом (строка 4). Для ссылки на эту последовательность используется имя str, после которого идет двоеточие. Наличие двоеточия в данном случае является обязательным, если его убрать ассемблер при трансляции кода выдаст ошибку со следующей диагностикой.

```
hello.asm:4: error: comma, colon or end of line expected
```

Объясняется эта ошибка тем, что набор команд IA-32 содержит команду с мнемоническим именем str. Запись объявления данных с именем str без символа двоеточия делает для ассемблера эту строку неотличимой от строки с командой str. Исправить ситуацию можно либо поместив двоеточие после имени, либо использовать имена, не пересекающиеся с мнемоническими именами команд.

```
%include 'io.inc'                ; (1)
                                 ; (2)
section .data                    ; (3)
   str: db `Hello, World!\n`, 0  ; (4)
                                 ; (5)
section .text                    ; (6)
                                 ; (7)
global CMAIN                     ; (8)
CMAIN:                           ; (9)
   PRINT_STRING [str]            ; (10)
   MOV EAX, 0                    ; (11)
   RET                           ; (12)
```

В записи значения строки используются обратные кавычки, что позволяет внести символ переноса строки непосредственно в последовательность байт. Следует отметить, что символ-ограничитель 0 должен явно дописываться в конец строк. В случае Си-кода этот символ автоматически добавляется компилятором, на уровне языка ассемблера это обязанность программиста.

Последнее отличие от Примера 1 – наличие на строке 10 вызова команды PRINT_STRING. Ее аргументом является исполнительный адрес, ссылающийся на то место памяти, где расположена строка «Hello, World!\n».

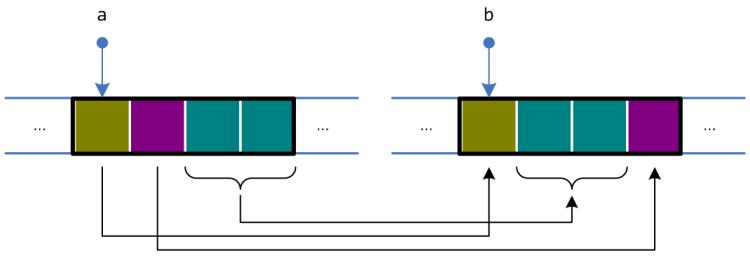
Перечень необходимого для выполнения практического занятия оснащения: задание, тетрадь для практических работ, компьютер

Порядок выполнения практической работы:

Задание 1. Запустить в среде разработки SASM программу из Примера 4 и выполнить ее в режиме отладки.

Задание 2. Запустить в среде разработки SASM программу из Примера 3 и выполнить ее в режиме отладки.

Задание 3. Написать и запустить в режиме отладки программу в среде разработки SASM по аналогии с программой из Примера 3 которая осуществляет пересылку данных, как это указано на рисунке.



Оценки выставляются в соответствии с полученным результатом.

№ задания	Результат
Не выполнены задания 1,2,3	2 (неудовлетворительно)
Выполнено задание 1	3 (удовлетворительно)
Выполнены задания 1,2	4 (хорошо)
Выполнены задания 1,2,3	5 (отлично)