МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Е.А. Кузьменкова, В.С. Махнычев, В.А. Падарян

Семинары по курсу "Архитектура ЭВМ и язык ассемблера"

(учебно-методическое пособие)

Часть 1

МАКС ПРЕСС

Москва – 2023

УДК 004.2+004.43(075.8)
ББК 32.973-02я73
К89

Печатается по решению Редакционно-издательского Совета факультета вычислительной математики и кибернетики МГУ имени М.В. Ломоносова

Рецензенты:
С.Ю. Соловьев, профессор
А.Н. Терехин, доцент

Е.А. Кузьменкова, В.С. Махнычев, В.А. Падарян.

К89 **Семинары по курсу «Архитектура ЭВМ и язык ассемблера»: учебно-методическое пособие. Часть 1.** — Издание 2-е, дополненное. М. Издательский отдел факультета ВМиК МГУ им. М.В. Ломоносова (лицензия ИД № 05899 от 24.09.2001); МАКС Пресс, 2014. – 80 с.

ISBN 978-5-89407-529-7

ISBN 978-5-317-04885-3

Учебное пособие содержит задачи и упражнения по первой части семинарских занятий курса «Архитектура ЭВМ и язык ассемблера», прочитанного студентам 1 потока 1 курса факультета Вычислительной математики и кибернетики МГУ в 2010-2023 гг. Пособие предназначено для студентов, изучающих основной курс программирования, а также для преподавателей и аспирантов.
Ключевые слова: архитектура ЭВМ, язык ассемблера, x86, nasm, реализация языка Си.

УДК 004.2+004.43(075.8)

ББК 32.973-02я73

This textbook contains problems and exercises for the first part of the seminar activities of the "Computer architecture and assembly language" course for 1st year 1st stream students of the faculty of Computational Mathematics and Cybernetics of Moscow State University that had been delivered in 2010-2023. The textbook is aimed at students learning the base programming course and at lecturers and postgraduate students.
Key words: computer architecture, assembly language, x86, nasm, C language implementation.

ISBN 978-5-89407-529-7

ISBN 978-5-317-04885-3

Содержание

Введение

Пособие содержит кратко изложенный материал семинарских занятий по курсу «Архитектура ЭВМ и язык ассемблера», читаемого для студентов 1 потока 1 курса факультета ВМК МГУ. Рассматриваются темы занятий первой половины семестра, такие как: организация ассемблерной программы, работа с целочисленными типами данных и указателями, условная и безусловная передача управления. Основной целью семинарских занятий является практическое знакомство с механизмами реализации программ, написанных на языке Си.

Каждая рассматриваемая тема содержит краткий вводный материал, детально разобранные типовые задачи и задачи для самостоятельной работы, часть которых снабжена ответами.

Методическое пособие предназначено для преподавателей, ведущих практические занятия в поддержку лекционного курса «Архитектура ЭВМ и язык ассемблера» для студентов 1 курса, а также рекомендуется студентам при подготовке к письменному экзамену.

В настоящем издании существенно переработаны и дополнены разделы пособия, посвященные описанию операций над битовыми векторами и управляющих конструкций языка, а также расширен набор предлагаемых примеров и задач.

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 $2^{32}-1$ (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.

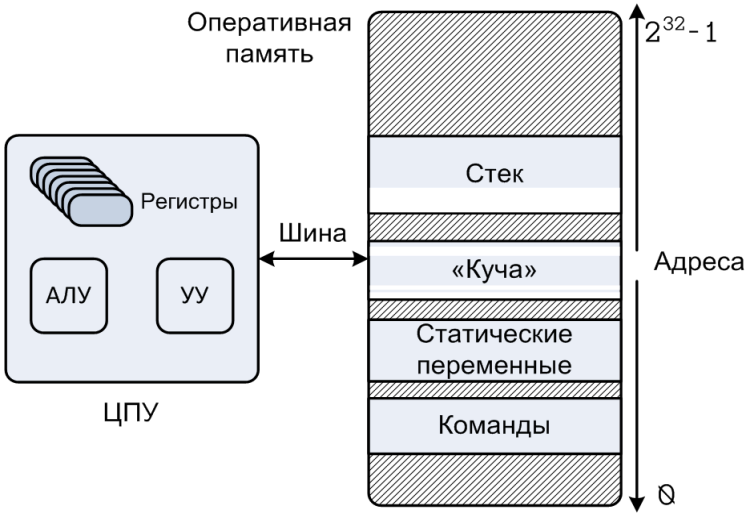


Рисунок 1 - Упрощенная схема виртуальной машины IA-32, используемой прикладной программой.

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 за-

вершатся 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.

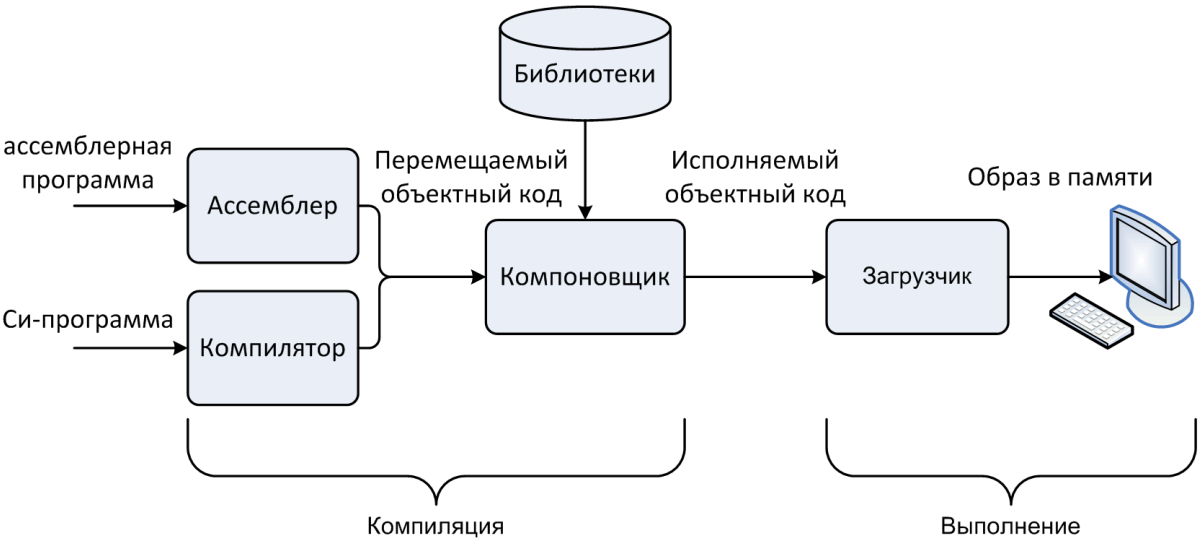


Рисунок 2 - Упрощенная схема компиляции и выполнения прикладной программы.

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 ее 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.

Ассемблерная инструкция

Строка с описанием ассемблерной инструкции имеет следующий вид:

```
метка:    код_опреации  операнды    ; комментарий
```

Имена меток не должны повторяться (за исключением локальных меток, которые будут рассмотрены во второй части пособия). Допустима ситуация, когда ассемблерная инструкция пропущена и строка содержит только метку с двоеточием. Такое форматирование текста практикуют для лучшей читаемости ассемблерного кода. Инструкция отделяется от операндов одним или несколькими пробелами или символами табуляции, а операнды, если они есть, должны быть отделены друг от друга запятыми и, возможно, пробельными символами и символами табуляции. В конце строки может находиться комментарий, начинающийся с символа «точка с запятой». Комментарий продолжается до конца строки.

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 компи-

лятором 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 1-1 Минимальная программа

Требуется написать минимальную ассемблерную программу.

Решение

```
section .text       ; (1)
                    ; (2)
global main         ; (3)
main:               ; (4)
   MOV EAX, 0       ; (5)
   RET              ; (6)
```

Данная ассемблерная программа ничего не делает и при запуске практически сразу возвращает управление операционной системе. Разберем ее построчно. Первая строка содержит директиву, указывающую, что последующие строки относятся к секции кода программы (конкретнее — к секции `.text`). Вторая строка оставлена пустой для наглядности отделения директивы от остального текста ассемблерной программы.

В третьей строке содержится директива, предписывающая сделать имя `main` видимым «снаружи» данного модуля (ассемблерного файла). На четвертой строке это имя используется в метке, которая помечает им инструкции на строке 5. Таким образом, метка `main` начинает описание функции с соответствующим именем. Как и для программ, написанных на языке Си, функция `main` является точкой входа в программу.

На пятой 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 (void) {
   return 0;
}
```

Пара инструкций 5 и 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.

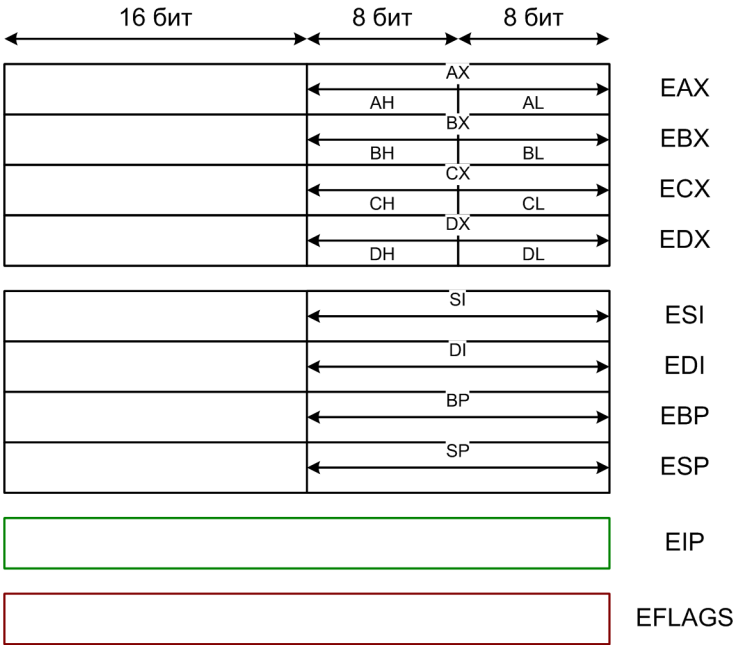


Рисунок 3 – Основные регистры IA-32.

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).

```
section .text       ; (1)
                    ; (2)
global main         ; (3)
main:               ; (4)
   MOV EAX, 1       ; (5) EAX := 1
   MOV EBX, EAX     ; (6) EBX := EAX = 1
   MOV CL, 040h     ; (7) CL := 040h = 0x40 = 64
   MOV EAX, 0       ; (8)
   RET              ; (9)
```

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.

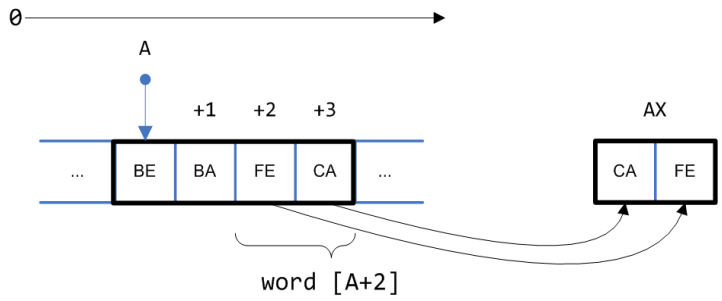
Пример 1-2 Определение значения регистра

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:



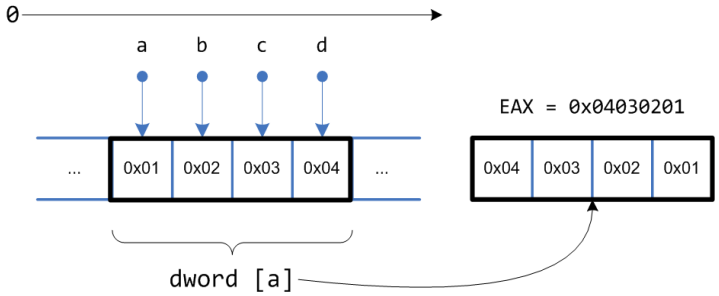
Байт с адресом A+2 будет иметь значение 0xFE, следующий за ним – 0xCA. При пересылке в регистр байты поменяются местами, и регистр AX будет иметь значение 0xCAFE. После того, как к этому значению будет прибавлено 3, оно станет равным 0xCB01.

Пример 1-3 Переворот байтов в двойном слове

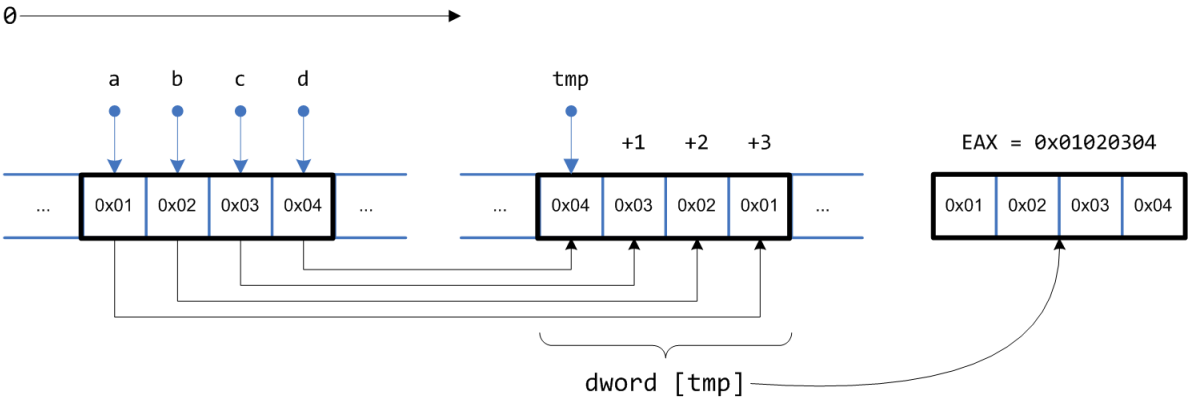
В памяти последовательно расположены 4 переменных a, b, c и d размером 1 байт каждая в заданном порядке. Требуется сформировать 32-битное число в регистре EAX таким образом, чтобы старший байт числа совпадал со значением переменной a, следующий за ним байт – со значением b, следующий – со значением c, 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 функций, позволяющих считывать и печатать числа в шестнадцатеричном и десятичном формате, строки и отдельные символы. Приведенные функции реализованы через вызовы функций стандартной библиотеки ввода/вывода языка Си, поэтому форматы принимаемых и выводимых данных для них полностью аналогичны форматам для вызовов соответствующих функций стандартной библиотеки.

Подробнее ознакомиться с реализацией этих функций можно, изучив текст учебной библиотеки ввода/вывода в соответствующем разделе в конце пособия.

Таблица 1. Сводная таблица функций ввода/вывода.

Функция	**EAX**	**EDX**	**ECX**	**EFLAGS**

`io_get_dec`
`io_get_udec`
`io_get_hex`
	выход: число	-	-	-

`io_get_char`	выход: символ	-	-	-
`io_get_string`	вход: адрес	вход: размер	-	-

`io_print_dec`
`io_print_udec`
`io_print_hex`
	вход: число	-	-	-

`io_print_char`	вход: символ	-	-	-
`io_print_string`	вход: адрес	-	-	-
`io_newline`	-	-	-	-

Значения регистров EBX, EBP, ESP, EDI, ESI не изменяются.

Вызовы произвольных функций будут рассмотрены более подробно во второй части данного пособия. Пока достаточно считать вызовы функций, приведенных в таблице 1, инструкциями, обладающими определенными побочными эффектами то есть изменяющими состояние регистров и памяти в соответствии с их описанием.

Гарантируется, что в результате вызовов функций ввода/вывода значения регистров EBX, EBP, ESP, EDI, ESI не изменяются. Значения EAX, EDX и ECX потенциально могут быть изменены, поэтому если какое-либо из этих значений понадобится в программе после вызова одной из этих функций, необходимо предварительно сохранить его на другом регистре (из множества сохраняемых) либо в памяти.

Пример 1-4 Hello, World!

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

Решение

Программа расширяет Пример 1-1: помимо секции `.text`, используется секция инициализированных статических данных `.data`, в которой размещена последовательность байтов 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 используя имена, не пересекающиеся с мнемоническими именами команд.

```
extern io_print_string              ; (1)
                                    ; (2)
section .data                       ; (3)
   str: db `Hello, World!\n`, 0     ; (4)
                                    ; (5)
section .text                       ; (6)
                                    ; (7)
global main                         ; (8)
main:                               ; (9)
   MOV EAX, str                     ; (10)
   CALL io_print_string             ; (11)
   MOV EAX, 0                       ; (12)
   RET                              ; (13)
```

Директива `extern`, использованная на строке 1, аналогична ключевому слову `extern` в языке Си. Она позволяет объявить имя, не определяемое в данном модуле (в случае `io_print_string`, оно будет связано с определением в учебной библиотеке ввода/вывода).

В записи значения строки используются обратные кавычки, что позволяет внести символ переноса строки непосредственно в последовательность байтов. 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 1-1 – наличие на строке 11 вызова функции `io_print_string`, которой в качестве ее единственного аргумента передается исполнительный адрес, ссылающийся на то место памяти, где расположена строка «`Hello, World!\n`». Как сообщается в таблице 1, эта функция принимает значение аргумента через регистр `EAX`. Команда `MOV` на строке 10 пересылает желаемое значение в регистр.

Задачи

Задачи, название которых подчеркнуто, снабжены ответом, приведенным в конце пособия.

Задача 1-1 Представление чисел

Рассматривается представление целых чисел в формате байта. Заполните все пустые ячейки таблицы, указав для каждого из приведенных чисел его представление в виде десятичного числа без знака, числа со знаком, а также представление в виде шестнадцатеричного и двоичного кодов.

Двоичное представление (байт)	16-ричное представление (байт)	Число десятичное со знаком	Число десятичное без знака
		-67	
01110110			
	28h		
			135

Задача 1-2 Определение переменных

Для каждой из указанных директив привести эквивалентную ей директиву, где начальное значение переменной представлено в 16-ричном виде.

```
section .data
   a dw -2076
   b dd -1
   c db 128
   d dw 256
   e dd -15
```

Задача 1-3 Директивы описания данных

Используя только директивы DB, привести описания, эквивалентные заданным.

```
; a
DW 185Ah
; b
DW 90
; c
DW 9000
; d
DW 10000, -10000
```

```
; e
DW 'AB'
; f
DW "NASM"
; g
DW "Hello"
; h
DW "w", "orld", "!"
```

```
; i
DD 97
; j
DD "x"
; k
DD 'CD'
; l
DD "Hello, world!"
```

Задача 1-4 Ошибки в коде

Даны описания.

```
b resb 1
w resw 1
y resw 1
d resd 1
```

Вычеркнуть синтаксически неверные инструкции.

```
; a
mov b, 1
; b
mov byte [b], 1
; c
mov word [b], 1
; d
mov ax, bx
```

```
; e
mov ecx, cx
; f
mov bh, cl
; g
mov dword [d], esi
; h
mov byte [w], ch
```

```
; i
mov word[w], word[y]
; j
add 15, bx
; k
sub word [y], 8
; l
sub eax, dword [d]
```

Задача 1-5 Определение значения регистра

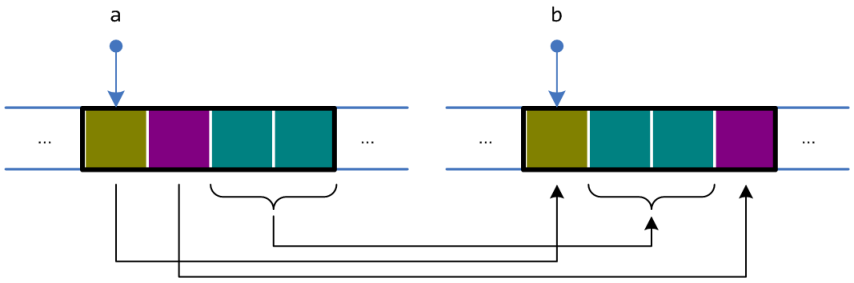
Выписать в шестнадцатеричном виде значение регистра EAX, после выполнения каждой помеченной инструкции. Следует отметить, что в каждой последовательности команд «а»-«ж» и «з»-«к» необходимо учитывать уже имеющееся значение регистра EAX.

```
section .data
   a dw 0xDEAD
   b dw 0xF00D
   c dw 0xCAFE
   d dw 0xBABE

section .text
   movsx eax, word [a + 1] ; (а)
   movsx ax,  byte [b]     ; (б)
   movzx eax, word [c]     ; (в)
   movsx eax, byte [b + 2] ; (г)
   movzx ax,  byte [d + 1] ; (д)
   movsx eax, word [b + 1] ; (е)
   mov   bx,  word [a + 1] ;
   movsx ax,  bh           ; (ж)
```

```
section .data
   a dw 0x0DEC
   b dw 0x4A6F
   c dw 0x7921
   d dw 0xFEFF

section .text
   mov   ebx, dword [b]     ;
   mov   eax, -1            ;
   movzx ax,  bl            ; (з)
   mov   ecx, dword [a + 1] ;
   mov   eax, dword [b + 1] ;
   movsx ax,  cl            ; (и)
   mov   eax, dword [c]     ;
   bswap eax                ; (к)
```

Задача 1-6 Перемещение данных

Привести фрагмент кода, осуществляющего пересылку данных, как это указанно на рисунке.



2. Арифметика и целочисленные типы данных

Основные арифметические операции реализуются в языке ассемблера инструкциями ADD, SUB, NEG, MUL/IMUL, DIV/IDIV. Помимо того, операции уменьшения и увеличения числа на единицу поддержаны инструкциями DEC и INC. Сложение и вычитание чисел большей длины, чем размер регистров поддержано инструкциями ADC и SBB. В определенных случаях для более быстрого вычисления арифметических выражений допустимо использовать инструкцию LEA. Выполнение арифметических операций сопровождается выработкой флагов регистра EFLAGS: CF – перенос, OF – переполнение, SF – знак, ZF – ноль, PF – четность.

Таблица 2. Коды описания флагов.

T	Значение флага влияет на выполнение инструкции
M	Инструкция меняет флаг (устанавливает или сбрасывает, в зависимости от операндов)
-	Влияние инструкции на флаг не определено
Пусто	Инструкция не влияет на флаг

Таблица 3. Перекрестные ссылки регистра флагов.

	OF	SF	ZF	PF	CF
ADC, SBB	M	M	M	M	TM
ADD, SUB, NEG	M	M	M	M	M
MUL, IMUL	M	-	-	-	M
DIV, IDIV	-	-	-	-	-
DEC, INC	M	M	M	M	
MOV, XCHG, MOVSX, MOVZX, LEA					

Машинные данные и типы данных языка Си

Объявление переменной в ассемблерной программе требует указания объема выделяемой памяти. Интерпретация содержимого переменной определяется исключительно кодом операции.

Перевод выражений языка Си на язык ассемблера сопровождается реализацией приведений типов, которые явно или неявно присутствуют в вычислениях.

Приведение целочисленных типов языка Си можно реализовать путем расширения размера данных при пересылке значений с помощью команд MOVZX и MOVSX. При этом команда MOVZX выполняет беззнаковое расширение данных (число до нужного размера дополняется слева нулями), а команда MOVSX – знаковое расширение данных (число до нужного размера дополняется слева битами со значением зна-

кового разряда исходного числа, т.е. нулями для неотрицательного числа и единицами для отрицательного числа).

```
mov     bx, 0xA67B ;  bx  ← 0xA67B
movzx  ebx, bx     ;  ebx ← 0x0000A67B
movsx  eax, bx     ;  eax ← 0xFFFFA67B
movsx  ecx, bl     ;  ecx ← 0x0000007B
```

Пример 2-1 Интерпретация арифметических инструкций

Выпишите значение регистра AL в виде десятичного числа (знакового и беззнакового), а также флаги CF, OF, ZF и SF после выполнения следующих инструкций.

```
MOV AL, 70
SUB AL, 130
```

Решение

$AL_{знаковое} = -60$, $AL_{беззнаковое} = 196$, CF = 1, OF = 1, ZF = 0, SF = 1.

Для наглядности формирования значений флагов промоделируем выполнение заданного фрагмента кода над шестнадцатеричным представлением данных.

70 = 46h, 130 = 82h

46h – 82h = C4h, при этом по правилам формирования флага CF (Carry Flag) этот флаг получит значение 1, поскольку первый операнд меньше второго. Результат выполнения операции (C4h) отличен от 0, следовательно, флаг ZF (Zero Flag) получит значение 0. Во флаге SF (Sign Flag) фиксируется значение знакового бита результата, т.е. его старшего бита. В нашем случае это 1, следовательно, флаг SF получит значение 1.

Проинтерпретируем полученный результат как десятичное число без знака и число со знаком:

C4h = 196 – это значение результата, интерпретируемое как десятичное число без знака. Это же значение представляет собой дополнительный код отрицательного числа -60 (196 = 256 – 60 = доп(-60)), поэтому результат выполнения операции, интерпретируемый как десятичное число со знаком, равен -60.

Для определения значения флага OF (Overflow Flag) промоделируем операцию вычитания над знаковыми десятичными числами. Заметим, что второй операнд (число 130) представляет собой дополнительный код отрицательного числа -126 (130 = доп(-126)). Следовательно, для знаковых чисел указанное вычитание приобретает вид:

70 – (-126) = 196

Полученный результат (196) не принадлежит допустимому диапазону значений знаковых чисел в формате байта [-128, 127], следовательно, флаг OF получит значение 1.

Пример 2-2 Объявление переменной

Требуется написать ассемблерную программу, в которой:

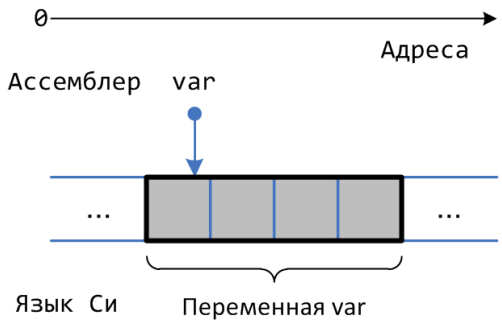
1. Выделяется память согласно следующему объявлению

```
static int var;
```

2. Выделенная память используется для ввода десятичного числа, …
3. у которого знак меняется на противоположный, …
4. а само число после этого печатается в шестнадцатеричном формате.

Решение

Для размещения переменной `var` используем сегмент статических неинициализированных данных `.bss`. Поскольку размер типа `int` в архитектуре IA-32 составляет четыре байта, используем директиву `var resd 1`. Выделенная память показана на рисунке ниже.



С точки зрения языка Си переменная `var` – содержимое четырех байт памяти, но в языке ассемблера символьное имя `var` будет интерпретироваться как адрес, причем адрес первого байта из четырех, выделенных для хранения некоторых произвольных данных. Этот адрес, т.е. его символьное обозначение будет использоваться при обращении к памяти.

```
extern io_get_dec, io_print_hex, io_newline
global main

section .bss
   var resd 1

section .text
main:
   CALL io_get_dec      ; Считываем десятичное число в EAX
   MOV DWORD [var], EAX ; Пересылаем его в выделенные 4 байта памяти
   NEG DWORD [var]      ; Меняем у числа знак на противоположный
   MOV EAX, DWORD [var] ; Загружаем значение переменной var на регистр,
                        ; используя ее имя как адрес
   CALL io_print_hex    ; Печатаем число в шестнадцатеричном формате
   CALL io_newline      ; Не забываем затем перейти на новую строку
   MOV EAX, 0
   RET
```

Пример 2-3 Приведение типа

Записать эквивалентную данному фрагменту на языке Си программу на языке ассемблера.

```
static short int a = 10;
static int b = 20, c;
c = a + b;
```

Решение

Переменные `a` и `b` инициализированы при объявлении – размещаем их в секции `.data`. Переменная `c` помещается в секции неинициализированных статических данных `.bss`.

```
section .data
a dw 10
b dd 20

section .bss
c resd 1

section .text
   movsx  eax, word [a]  ; Знаковое расширение данных
   add    eax, dword [b] ;
   mov    dword [c],eax  ;
```

В вычислении суммы участвуют переменные с разными типами. Прежде чем будет выполнено сложение, типы операндов должны быть приведены к одному общему

типу. В данном случае значение переменной a должно быть расширено до типа `int`.

Пример 2-4 Приведение беззнакового типа

Записать эквивалентную данному фрагменту на языке Си программу на языке ассемблера.

```
unsigned char z = 0xff;
unsigned short a = 0xff00;
unsigned int b;
b = a * z;
```

Решение

Поскольку выполняется арифметическая операция, происходит *integer promotions*, неявное приведение операндов к целому типу стандартного размера. Результат операции в данном примере может быть выражен знаковым целым типом стандартного размера:

$$INT\_MIN < 0 < USHRT\_MAX \times UCHAR\_MAX < INT\_MAX$$

Для 32-разрядной архитектуры IA-32 и компилятора gcc, величины, входящие в неравенство имеют следующие значения.

Таблица 4. Предельные значения некоторых типов данных языка Си.

Константа	Описание	Величина
INT_MIN	Минимальное значение стандартного целого типа	-2147483648
USHRT_MAX	Максимальное значение беззнакового короткого целого типа	65535
UCHAR_MAX	Максимальное значение беззнакового целого типа char	255
INT_MAX	Максимальное значение стандартного целого типа	2147483647

Объявлены эти константы в стандартном заголовочном файле `limits.h`.

Таким образом, выполнение умножения потребует сначала расширить оба операнда до стандартного целого типа, потом выполнить операцию знакового умножения и сохранить результат в переменной b. Причем, последнее присвоение не будет производить никаких преобразований над полученным при умножении числом.

```
section .bss
   b resd 1

section .data
   z db 0xff
   a dw 0xff00

section .text
   movzx eax, byte [z] ; выполняем беззнаковое расширение
                       ; из 8 разрядов в 32
   movzx edx, word [a] ; выполняем беззнаковое расширение
                       ; из 16 разрядов в 32
   imul eax, edx       ; умножаем
   mov dword [b], eax  ; сохраняем полученное значение
```

Пример 2-5 Умножение

Записать эквивалентную данному фрагменту на языке Си программу на языке ассемблера. Описать секции кода, инициализированных и неинициализированных данных.

```
static int a, b = 1, c = -2, d = 3;
...
a = b + c * d;
```

Решение

Переменная a не имеет инициализирующего значения и будет размещена в секции `.bss`. Остальные переменные помещаются в секции `.data`.

```
section .bss
a resd 1

section .data
b dd 1
c dd -2
```

В первую очередь вычисляется произведение переменных c и d. Используется форма инструкции IMUL с одним операндом, второй операнд задан неявно, это регистр EAX. Результат будет сохранен в паре регистров EDX:EAX. Старшие разряды результата, т.е. содержимое регистра EDX, в дальнейших вычислениях не используются, поскольку в языке Си тип результата умножения совпадет с типами операндов-выражений (необходимо не забывать о неявном приведении типов опе-

рандов). После сложения полученный результат выгружается обратно из регистра EAX в память.

```
section .text
   mov  eax, dword [c]
   imul dword [d]
   add  eax, dword [b]
   mov  dword [a], eax
```

Пример 2-6 Деление

Записать эквивалентную данному фрагменту на языке Си программу на языке ассемблера.

```
static int x, y;
...
x /= -y;
```

Решение

Поскольку x и y объявлены как статические переменные, размещаем их в секции неинициализированных статических данных.

```
section .bss
   x resd 1
   y resd 1
```

Реализация выражения, в котором происходит деление, состоит из загрузки значений переменных x и y на регистры, обращения знака, деления и выгрузки полученного результата обратно в память.

```
section .text
   mov  eax, dword [x] ; Записываем в регистр eax значение
                       ; переменной x. Поскольку в дальнейшем
                       ; это значение будет выступать в
                       ; качестве делимого, его необходимо
                       ; разместить в паре регистров edx:eax
                       ; Таким образом, 32-разрядное знаковое
                       ; число будет расширено до 64 разрядов.
                       ; верхние 32 разряда заполнятся знаковым
                       ; битом.
   mov  edx, eax       ; Копируем значение переменной x в edx
   sar  edx, 31        ; Сдвигаем число на 31 разряд – все
                       ; разряды регистра будут заполнены
                       ; знаковым битом, т.к. сдвиг
                       ; арифметический, пара регистров edx:eax
                       ; готова к знаковому делению.
   mov  ecx, dword [y] ; Записываем в регистр ecx значение
                       ; переменной y
   neg  ecx            ; Меняем знак числа, т.е. получаем
                       ; значение (-y)
   idiv ecx            ; делим
   mov  dword [x], eax ; частное из регистра eax записываем в
                       ; память, где размещена переменная x
```

Пример 2-7 Арифметика «длинных» чисел

По заданному фрагменту на языке Си написать эквивалентный код на языке ассемблера.

```
static long long x;
…
x++;
```

Решение

Переменная x будет размещена в секции неинициализированных статических данных в формате учетверенного слова:

```
section  .bss
   x resq 1
```

Поскольку максимальный допустимый размер операнда в арифметических командах – двойное слово (32 бита), для реализации арифметики над 64-разрядными числами их разбивают на две части по 32 бита каждая и выполняют соответствующие операции над этими частями. В нашем случае необходимо увеличить млад-

шую часть x на 1 и учесть возникающий при этом возможный перенос из старшего разряда. Возникающий при сложении младших частей перенос фиксируется во флаге CF, поэтому дальнейшее сложение старших частей надо выполнять с учетом данного флага, для чего следует использовать команду ADC. Принимая во внимание «перевернутое» представление в памяти чисел размером больше байта, получаем следующий код:

```
section  .text
   add  dword[x], 1     ; Сложение младших частей
   adc  dword[x + 4], 0 ; Учитываем возможный перенос из
                        ; старшего разряда
```

Заметим, что реализовывать в данном примере увеличение младшей части x на 1 командой INC нельзя, т.к. она не формирует флаг CF и, следовательно, не позволяет учитывать возможный перенос.

Задачи

Задача 2-1 Присваивание различных типов

Реализовать присваивание `b = a;` (не более двух команд) при условии:

```
// a
static unsigned char a;
static unsigned int b;

// b
static char a;
static short b;
```

Задача 2-2 Интерпретация арифметических инструкций

Выпишите значение регистра AL в виде десятичного числа (знакового и беззнакового), а также флаги CF, OF, ZF и SF после выполнения следующих инструкций.

```
; a
   MOV AL, 199
   ADD AL, -61

; b
   MOV AL, -35
   SUB AL, 216
```

```
; c
   MOV AL, -13
   ADD AL, 179

; d
   MOV AL, 2
   SUB AL, 200
```

Задача 2-3 Реализация вычисления арифметических выражений

Приведите фрагмент программы на ассемблере для вычисления следующих выражений.

```
// a
   static int x, y;
   y = (x / y) * (x % y);

// b
   static unsigned char a;
   static int b;
   b = (a - 500000) % 10;
```

Задача 2-4 Цифры и число

Пусть

```
static unsigned short n; // 100 <= n <= 999
```

Приведите фрагмент ассемблерного кода для записи в n числа, полученного выписыванием в обратном порядке десятичных цифр исходного числа n.

Задача 2-5 64 на 32

Даны 64-разрядные переменные x и y. Реализовать операции.

```
// a
x += y;
// b
x -= y;
```

Задача 2-6 Быстрая арифметика

Используя команду LEA, реализуйте быстрое вычисление следующих арифметических выражений:

```
static int  a, b, c;
// a
   c = 10 * a + b + 14;
// b
   c = 24 * a – 15 + b;
```

Задача 2-7 Ошибки в коде

Зачеркните инструкции, содержащие ошибки.

```
X RESD 1

ADC WORD [X], WORD [EAX]
MUL AL, AH
NEG CF
SBB DWORD [X], 100
MOV EAX, X
DIV 15
SUB EAX, WORD [X]
MOVSX EBX, BL
XCHG WORD[X], 100
IDIV BYTE ⌈EAX]
```

3. Указатели и адресная арифметика

Взятие адреса и разыменование

Статические переменные располагаются в одной из секций статических данных; если у переменных не происходит инициализация при объявлении, то расположить их допустимо в секции `.bss`. В языке ассемблера адрес статической переменной – ее символическое имя. Это имя необходимо присвоить переменной xp.

```
static int *xp;
static int x;
xp = &x;
```

```
section .bss
   xp resd 1
   x  resd 1

section .text
   mov    dword [xp], x
```

Отображение оператора разыменования в язык ассемблера

Располагаем все переменные в сегменте статической памяти `.bss`.

```
static int *xp;
static int x, y;

x = *xp;

*xp = y;
```

```
section .bss
   xp resd 1
   x  resd 1
   y  resd 1

section .text
   mov edx, dword [xp]   ; помещаем в EDX значение переменной xp
   mov eax, dword [edx]  ; помещаем в EAX значение ячейки
                         ; памяти, на которую ссылается xp
   mov dword [x], eax    ; присваиваем это значение x

   mov eax, dword [y]    ; помещаем в EAX значение переменной y
   mov dword [edx], eax  ; в регистре EDX уже находится значение
                         ; переменной xp. Это значение
                         ; используется как адрес, по которому
                         ; будет записано содержимое EAX
```

Пример 3-1 Двукратное разыменование

Дана статическая переменная p:

```
static int **p;
```

Требуется на языке ассемблера написать фрагмент программы, который вычисляет выражение `**p + 1` и печатает его значение на стандартный вывод, используя функцию `io_print_dec`.

Решение

Заданное выражение предполагает двойное разыменование указателя и увеличение полученного значения на единицу.

```
extern io_print_dec
global main

section .text
main:
   MOV  EAX, DWORD [p]    ; Записываем в регистр EAX значение
                          ; переменной p
   MOV  EAX, DWORD [EAX]  ; Используем это значения для доступа
                          ; к памяти, теперь в регистре EAX
                          ; не p, а *p
   MOV  EAX, DWORD [EAX]  ; Повторяем – теперь в EAX находится **p
   INC  EAX               ; Увеличиваем это значение на единицу
   CALL io_print_dec      ; Печатаем
   XOR  EAX, EAX          ;
   RET                    ;
```

Пример 3-2 Разыменование и побочные эффекты

Дан фрагмент кода на языке Си.

```
static short *px, *py;
...
*px++ = --*py;
```

Требуется привести эквивалентную программу на языке ассемблера.

Решение

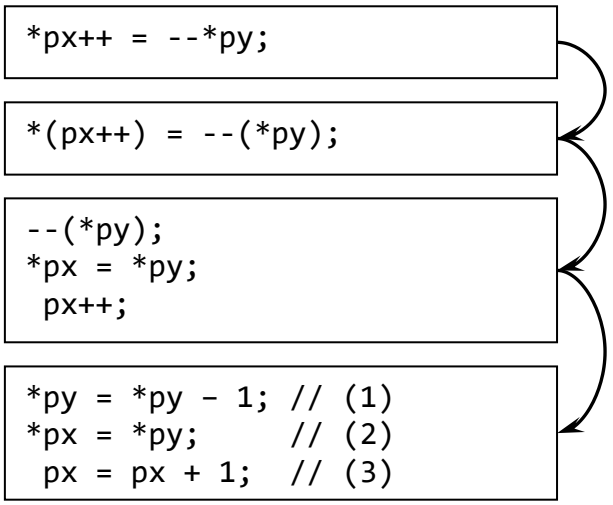
В первую очередь требуется определить, какие данные должны быть определены в ассемблерной программе. Во фрагменте Си-кода объявлены две переменные типа `short*`. Несмотря на то, что тип `short` занимает два байта, указатель будет занимать четыре байта, поскольку адресация в IA-32 32-х разрядная, т.е. под адрес

памяти необходимы 4 байта. Выписываем следующие директивы ассемблера Nasm:

```
section .bss
   px resd 1
   py resd 1
```

Статические переменные располагаются в секциях статических данных. Поскольку нет кода инициализации этих переменных, их допустимо разместить в секции статических неинициализированных данных `.bss`, в противном случае переменные были бы объявлены и проинициализированы в секции `.data`.

Для того чтоб было проще корректно реализовать на ассемблере заданное выражение, преобразуем его, избавившись от побочных эффектов. Помимо того, расставим скобки для явного указания порядка выполняемых операций.



Теперь последовательно переводим каждый оператор в соответствующий код на языке ассемблера.

```
section .text
mov eax, dword [py] ; (1) записываем в регистр eax значение
                    ;     переменной py
dec word [eax]      ;     уменьшаем на единицу 16-разрядную
                    ;     величину, которую адресует регистр
                    ;     eax, т.е. переменная py
mov cx, word [eax]  ; (2) записываем в регистр cx эту измененную
                    ;     величину
mov eax, dword [px] ;     записываем в регистр eax значение
                    ;     переменной px
mov word [eax], cx  ;     используя это значение, записываем в
                    ;     то место памяти, которое адресуется
                    ;     указателем px, текущее значение *py
add dword [px], 2   ; (3) Увеличиваем указатель px. Согласно
                    ;     правилам адресной арифметики языка Си
                    ;     значение указателя будет увеличено на
                    ;     1 * sizeof(short), т.е на 2
```

Пример 3-3 Восстановление кода

Функция, имеющая следующий прототип

```
void decode1(int *xp, int *yp, int *zp)
```

была скомпилирована в ассемблерный код. Тело функции выглядит следующим образом.

```
mov edi, dword [ebp + 8]  ; (1)
mov edx, dword [ebp + 12] ; (2)
mov ecx, dword [ebp + 16] ; (3)
mov ebx, dword [edx]      ; (4)
mov esi, dword [ecx]      ; (5)
mov eax, dword [edi]      ; (6)
mov dword [edx], eax      ; (7)
mov dword [ecx], ebx      ; (8)
mov dword [edi], esi      ; (9)
```

Параметры xp, yp и zp находятся в памяти по смещениям 8, 12 и 16 относительно адреса, содержащегося в регистре ebp. Напишите код тела функции decode1 на языке Си, который был бы эквивалентен представленному ассемблерному коду.

Решение

В первых трех инструкциях выполнилась загрузка значений формальных параметров функции на регистры.

```
mov edi, dword [ebp + 8]  ; (1) edi ← xp
mov edx, dword [ebp + 12] ; (2) edx ← yp
mov ecx, dword [ebp + 16] ; (3) ecx ← zp
```

В следующих трех инструкциях эти регистры использовались для доступа к содержимому памяти, т.е. параметры-указатели разыменовывались, и соответствующие значения из памяти пересылалась в регистры ebx, esi, eax.

```
mov ebx, dword [edx]      ; (4) ebx ← *yp
mov esi, dword [ecx]      ; (5) esi ← *zp
mov eax, dword [edi]      ; (6) eax ← *xp
```

В последних трех инструкциях регистры edi, edx, ecx, содержащие адреса, снова использовались для доступа к памяти, но в этот раз происходила запись тех значений, которые были ранее размещены на регистрах ebx, esi, eax.

```
mov dword [edx], eax      ; (7) *yp ← eax ← *xp
mov dword [ecx], ebx      ; (8) *zp ← ebx ← *yp
mov dword [edi], esi      ; (9) *xp ← esi ← *zp
```

Если ввести три вспомогательные переменные для хранения значений считываемых из памяти, получим следующий Си-код.

```
void decode1(int *xp, int *yp, int *zp) {
    int y = *yp;
    int z = *zp;
    int x = *xp;
    *yp = x;
    *zp = y;
    *xp = z;
}
```

Переменная y соответствует ebx, z – esi, x – eax.

Указатели и массивы

В языке Си обращение к указателям и массивам происходит единообразно. К ним можно применять адресную арифметику и индексное выражение. Однако имеется существенное отличие в ассемблерном коде, реализующем текстуально одинаковые выражения. Отличие обусловлено тем, как выделятся память при объявлении указателей и массивов.

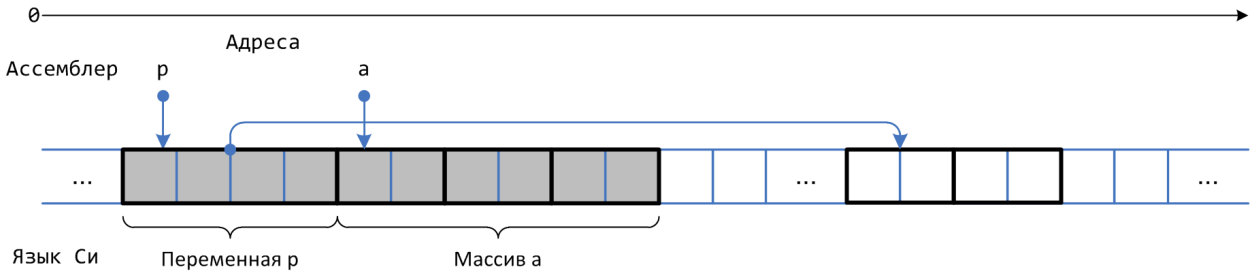
Пример 3-4 Адресная арифметика и массивы

Для приведенного фрагмента Си-кода требуется написать соответствующий фрагмент ассемблерной программы.

```
   static short *p;
   static short a[3];
…
   p[1] = *(a + 2);
```

Решение

В приведенном фрагменте массив используется как указатель, а указатель – как массив, с индексным выражением. Тем не менее, ассемблерный код будет отражать особенности фактического выделения памяти. На рисунке ниже показано распределение данных в памяти: черные линии показывают группировку байт в базовые типы, серым цветом показаны выделенные байты. Снизу памяти данные подписаны в терминах языка Си, сверху – адреса, используемые в ассемблерном коде.



Память выделена для указателя p, но не для тех ячеек, на которые он указывает. Поэтому, что вычислить адрес первого элемента последовательности int-ов, на которые указывает p, необходимо загрузить адрес (значение переменной p) из памяти в регистр. В случае с массивом память была выделена для всех элементов, а имя массива интерпретируется как адрес начала выделенной памяти. Поэтому при извлечении элемента с индексом два имя массива следует сразу же использовать в адресном коде в качестве базы.

```
mov   dx, word [a + 4]     ; *(a + 2) – то же, что и a[2]
                           ; a – адрес, начиная с которого
                           ; в памяти размещены элементы массива
mov   eax, dword [p]       ; Значение переменной p – адрес,
mov   word [eax + 2], dx   ; который указывает на начало
                           ; массива
```

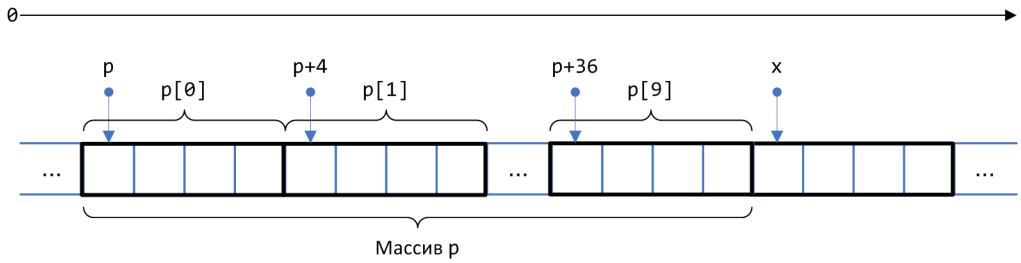
Пример 3-5 Массив указателей

Для приведенного фрагмента Си-кода требуется написать соответствующий фрагмент ассемблерной программы.

```
static int *p[10];
static int x;
x = *p[8] + 1;
```

Решение

В первую очередь определим, как данные были расположены в памяти.



Элементами массива p являются указатели на целые числа. Массив занимает 40 байт, элементы массива расположены непрерывно, каждый элемент занимает 4 байта.

Обращение к 8-му элементу означает выборку данных по адресу p+32, далее происходит разыменование, т.е. значение этого элемента трактуется как адрес, по которому лежит требуемое число. После того, как это число извлечено из памяти к нему прибавляется 1, результат сложения записывается в память по адресу x.

```
mov     eax, dword [p + 32]
mov     eax, dword [eax]
add     eax, 1
mov     dword [x], eax
```

Задачи

Задача 3-1 Восстановление Си-кода

Функция, имеющая следующий прототип

```
void decode2(int *xp, int *yp, int *zp)
```

была скомпилирована в ассемблерный код. Тело функции выглядит следующим образом.

```
mov     eax, dword [ebp + 12]
mov     edi, dword [eax]
mov     eax, dword [ebp + 8]
mov     edx, dword [eax]
mov     eax, dword [ebp + 16]
mov     eax, dword [eax]
add     edx, eax
mov     eax, dword [ebp + 12]
mov     dword [eax], edx
mov     eax, dword [ebp + 8]
mov     dword [eax], edx
mov     eax, dword [ebp + 16]
mov     dword [eax], edi
```

Параметры xp, yp и zp находятся в памяти по смещениям 8, 12 и 16 относительно адреса, содержащегося в регистре ebp. Напишите код тела функции decode2 на языке Си, который был бы эквивалентен представленному ассемблерному коду.

Задача 3-2 Побочный эффект

Написать фрагмент ассемблерного кода, эквивалентный данному фрагменту на языке Си.

```
// a
int x, *px;
*px++ = x -10;

// b
short x, *px;
x = *(--px + 4);
```

Задача 3-3 Среднее арифметическое

Дан массив из 100 целых чисел. Написать фрагмент ассемблерного кода, который помещает в регистр eax среднее арифметическое первого и последнего элементов массива при условии, что элементы массива имеют тип:

a) `char`,
b) `short`,
c) `int`.

Задача 3-4 Разность указателей

Для приведенного фрагмента Си-кода написать соответствующий фрагмент ассемблерной программы.

```
static int a[50];
static int *p, *q;
static int n;

p = &a[10];
q = &a[25];
n = q - p;
```

Задача 3-5 Реализовать Си-код

Для приведенного фрагмента Си-кода написать соответствующий фрагмент ассемблерной программы.

```
// a
short **p;
++*(*(p+=3) -= 2);

// b
char **q, **r, **t;
r = q++;
*t = (*r)++;
**q = (**t)++;
```

4. Операции над битовыми векторами

Поразрядные (побитовые) операции

Набор команд IA-32 поддерживает поразрядные команды, работающие с векторами битов: NOT, AND, XOR, OR. Помимо получения результата, команды формируют флаги, наибольший интерес среди которых обычно представляет флаг ZF.

Приведенные команды естественным образом реализуют побитовые логические операции языка Си над целыми числами, тогда как работу с логическими величинами необходимо реализовывать другими способами.

Команда TEST работает аналогично команде AND с той лишь разницей, что результат выполнения команды никуда не заносится и, следовательно, значение первого операнда не изменяется. Обычно команда TEST применяется для проверки на 0 отдельных битов первого операнда. Например, проверка на четность содержимого регистра eax выглядит следующим образом:

```
test  eax,  1
jz    .even   ;  Переходим на метку, если eax четно
```

Приведем еще два распространенных примера использования побитовых операций:

```
xor  eax, eax  ; Обнуление регистра (вместо  mov  eax, 0)
```

```
test  eax, eax ; Проверка на 0 содержимого регистра
               ; (вместо  cmp  eax, 0)
```

Заметим, что в силу специфики выполнения побитовых операций команды в приведенных примерах выполняются быстрее, чем аналогичные действия, выполненные без использования побитовых операций.

Пример 4-1 Логические операции над битовыми векторами

Даны четыре статические переменные целого типа a, b, c, d. Написать фрагмент кода на языке ассемблера, вычисляющий значение переменной a.

```
static int a, b, c, d;

a = ~(a & b) | ((~c & d) | (c & ~d));
```

Решение

Переменные a, b, c и d размещаем в статической памяти, в сегменте `.bss`. При вычислении будем следовать заданному порядку вычисления выражения, хотя используемые в выражении операции позволяют этот порядок менять.

Следует заметить, что при вычислении подвыражения `(~c & d) | (c & ~d)` необходимо пользоваться регистрами для выполнения побитового отрицания, т.к. непосредственное выполнение этой операции над ячейками памяти «испортит» значение переменной, расположенное в этих ячейках.

```
section .bss
   a resd 1
   b resd 1
   c resd 1
   d resd 1

section .text
   mov  eax, dword [c]  ; eax ← c
   mov  edx, dword [d]  ; edx ← d
   mov  ecx, eax        ; создаем копию переменной c в ecx
   not  eax             ; eax ← ~c
   and  eax, edx        ; eax ← ~c & d
   not  edx             ; edx ← ~d
   and  ecx, edx        ; ecx ← c & ~d
   or   eax, ecx        ; eax ← (~c & d) | (c & ~d)
   mov  edx, dword [a]  ; edx ← a
   and  edx, dword [b]  ; edx ← a & b
   not  edx             ; edx ← ~(a & b)
   or   eax, edx        ; eax ← ~(a & b) | ((~c & d) | (c & ~d))
   mov  dword [a], eax  ;   a ← ~(a & b) | ((~c & d) | (c & ~d))
```

Сдвиги и вращения

К побитовым операциям относятся также команды сдвигов. Основные виды сдвигов реализуются командами SHR, SHL, SAR, SAL, ROR, ROL, RCR, RCL.

С помощью сдвигов можно реализовать быстрое умножение и деление на 2^n (см. таблицу 5).

Все указанные команды выполняются значительно быстрее традиционных команд умножения и деления (MUL, IMUL, DIV, IDIV), поэтому целесообразно использовать именно их для реализации соответствующих операций при вычислении арифметических выражений.

Таблица 5. Реализация быстрого умножения и деления на степени 2.

Выражение	op *= 2^n	op /= 2^n	op %= 2^n, op >= 0
Команда	SHL op, n	числа без знака: SHR op, n числа со знаком: SAR op, n	AND op, 2^n-1

Пример 4-2 Двигаем и вращаем

Даны две статические переменные a и b размером 2 байта. Требуется регистр EAX заполнить следующим образом: верхние 2 байта должны содержать значение переменной a, а нижние 2 байта – значение переменной b, циклически сдвинутое на 5 бит вправо.

Решение

```
section .bss
   a resw 1
   b resw 1

section .text
   mov ax, word [a]  ; помещаем значение переменной a в младшие
                     ; 16 разрядов
                     ; регистра EAX
   shl eax, 16       ; сдвигаем это значение в верхние байты
   mov ax, word [b]  ; помещаем значение переменной b в
                     ; освободившиеся разряды регистра EAX и …
   ror ax, 5         ; … выполняем над ними циклический сдвиг
```

Пример 4-3 Реализация умножения регистра на константу через сложения и сдвиги

Реализовать умножение значения, лежащего в регистре eax на 6, используя только операции логического сдвига и сложения. Результат поместить в eax.

Решение

```
mov ebx, eax ; копируем значение в регистр ebx
shl ebx, 2   ; сдвигаем значение ebx влево на 2 бита
             ; (эквивалентно умножению на 4)
shl eax,1    ; сдвигаем значение eax влево на 1 бит
             ; (эквивалентно умножению на 2)
add eax, ebx ; складываем получившиеся значения,
             ; результат в регистре eax
```

Пример 4-4 Восстановление выражения

Дан код на языке Си, в котором пропущены выражения, присваивающие значения локальным переменным.

```
int arith(int x, int y, int z) {
   int t1 = _________;
   int t2 = _________;
   int t3 = _________;
   int t4 = _________;
   return t4;
}
```

Дан ассемблерный код, который был сгенерирован компилятором. Параметры расположены следующим образом: x лежит по адресу `[ebp + 8]`, y – по адресу `[ebp + 12]`, z – по адресу `[ebp + 16]`.

```
mov eax, dword [ebp + 12]  ; (1)
xor eax, dword [ebp + 8]   ; (2)
sar eax, 3                 ; (3)
not eax                    ; (4)
sub eax, dword [ebp + 16]  ; (5)
```

Требуется восстановить пропущенные выражения.

Решение

Инструкция в первой строке ассемблерного кода выполняет пересылку значения переменной-параметра y на регистр eax. Остальные четыре инструкции преобразуют данные, распределим эти операции по выражениям в Си-программе.

Во второй строке выполняется операция xor над значениями переменных y и x. Полученный результат сохраняется в регистре eax. Далее (строка №3) это значение

было сдвинуто вправо на 3 разряда. Причем сдвиг арифметический, не логический, что согласуется со знаковым типом сдвигаемой переменной. В четвертой строке выполняется побитовое отрицание, в последней, строке №5, из этого значения был вычтен параметр z.

Следует отметить, что все результаты вычислений не записывается в память, а продолжают сохраняться на одном и том же регистре eax. Все автоматические локальные переменные не были размещены в памяти, что позволяет характер выполняемых вычислений – никаких действий с адресами этих переменных не происходило.

```
int arith(int x, int y, int z) {
   int t1 = y ^ x;
   int t2 = t1 >> 3;
   int t3 = ~t2;
   int t4 = t3 - z;
   return t4;
}
```

Возвращаемое значение в данной функции – переменная t4, после выполнения инструкции в строке №5 ее значение уже расположено в регистре EAX, который используется для передачи возвращаемого функцией значения.

Пример 4-5 Обращение операций

После выполнения последовательности команд в регистре EAX было получено некоторое значение. Восстановите содержимое ячеек памяти.

```
x db ____________________
movsx eax, word [x]
ror   eax, 4
; EAX = 0xfffffbee
```

Решение

Первое необходимое действие – определить значение EAX перед командой ror. Поскольку вращение циклически сдвинуло содержимое регистра на 4 разряда все шестнадцатеричные цифры остались неизменны, но последняя оказалась крайней слева. Таким образом, значение EAX перед вращением – 0xffffbeef.

Выполнение команды movxs привело к заполнению двух старших байтов в регистре EAX единицами, что согласуется с содержимым младших байтов: при переводе старшей шестнадцатеричной цифры в числе 0xbeef в последовательность бит получаем старший (знаковый) бит – 1. Величина 0xbeef была размещена в памяти,

начиная с адреса x. При размещении использовались директивы db, что привело к разделению числа на два отдельных байта и явному развороту их порядка: первым идет 0xef, затем – 0xbe.

```
x db 0xef, 0xbe
movsx eax, word [x]
ror   eax, 4
; EAX = 0xfffffbee
```

Пример 4-6 Безусловный модуль

Требуется вычислить модуль числа, не используя команд условной передачи данных и условной передачи управления.

Решение

Пусть число, для которого будет вычисляться модуль, уже загружено в регистр eax. Воспользуемся следующими фактами. Арифметический сдвиг влево на 31 разряд заполнит знаковым битом весь битовый вектор. В зависимости от того, какое знак у числа был изначально, будет получено либо 0, либо -1.

Другой факт заключается в том, что побитовое обращение устанавливает взаимно однозначное соответствие между отрицательными и неотрицательными числами:

$$00\text{K}\ 00_b = 0 \xleftrightarrow{not} -1 = 11\text{K}\ 11_b$$
$$00\text{K}\ 01_b = 1 \xleftrightarrow{not} -2 = 11\text{K}\ 10_b$$
$$\text{K}$$
$$01\text{K}\ 10_b = 2147483646 \xleftrightarrow{not} \text{-}2147483647 = 10\text{K}\ 01_b$$
$$01\text{K}\ 11_b = \text{INT\_MAX} \xleftrightarrow{not} \text{INT\_MIN} = (\text{-INT\_MAX -1}) = 10\text{K}\ 00_b$$

Таким образом, NOT(x) = -(x+1).

Третий факт касается свойств Исключающего Или: $a \oplus 0 \equiv a, a \oplus 1 \equiv \neg a$.

Эти три факта в совокупности позволяют составить следующий код, реализующий формулу $sign(x) = (x \oplus sb) - sb$, где sb – результат арифметического сдвига на 31 разряд. Для положительных чисел xor и sub берут в качестве второго операнда 0, что никак не меняет eax. Для отрицательного числа формула приобретает вид $(x \oplus 1\text{K}\ 1_b) - (-1) = NOT(x) + 1 = -x$.

```
mov ecx, eax
sar ecx, 31
xor eax, ecx
sub eax, ecx
```

Манипуляции с отдельными битами

Для манипуляции с отдельными битами используются команды BTS, BTR, BTC, позволяющие устанавливать нужные значения указанному биту операнда. Например:

```
bts  op,  5 ; Установить в 1 пятый бит операнда op
```

Пример 4-7 Установка и сброс отдельных битов

Дана 32-битная статическая переменная a. Требуется установить в этой переменной бит 3 в значение 1, сбросить бит 17 в значение 0, а значение бита 23 заменить на противоположное.

Решение

Для манипуляций с отдельными битами используем инструкции `bts`, `btr`, `btc`.

```
section .bss
    a resd 1

section .text
    bts   dword [a], 3
    btr   dword [a], 17
    btc   dword [a], 23
```

Задачи

Задача 4-1 Знак числа

Пусть переменная x содержит некоторое целочисленное значение. Не используя инструкции условного перехода, поместить в регистр eax 1, если `x < 0`, и `0` в противном случае.

```
// a
static int x;
```

```
// b
static short x;
```

```
// c
static char x;
```

Задача 4-2 Обмен битами

В регистре eax заменить 5 левых битов на 5 правых битов.

Задача 4-3 Арифметика без арифметических команд

Приведите фрагмент ассемблерного кода для вычисления выражений.

```
// a
static unsigned x, y;
y = 32 * x - x / 8 + x % 16;

// b
static long long x;
x *= 2;

// c
static int x, y;
y = x / 64 + x * 4;
```

Задача 4-4 Интерпретация кода

Укажите значения регистров eax (как десятичное число со знаком) и edx (как десятичное число без знака) после выполнения следующего фрагмента кода.

```
section .data
   x dd 0xfeff0201
section .text
   mov cx, word [x + 1]
   mov ax, cx
   imul ah
   shl ax, cl
   movsx eax, ax
   movzx edx, ax
```

Задача 4-5 Подсчет числа единиц

Подсчитать количество двоичных единиц в двоичном представлении заданного целого неотрицательного числа.

Задача 4-6 Подряд идущие единицы

В регистр eax поместить значение 1, если в двоичном представлении заданного целого числа встречается подряд 5 единиц, и 0 в противном случае.

Задача 4-7 Проверка четности

Не используя команд деления и не меняя значения переменной a, реализовать переход на метку L в том и только в том случае, если значение a четно.

Задача 4-8 Битовый разворот

"Перевернуть" содержимое регистра EAX в битовом представлении.

Задача 4-9 Sign

Реализуйте функцию `sign`, используя только инструкции MOV, SHR, NEG и OR.

Задача 4-10 Крайний правый

Для данного машинного слова, сформируйте в регистре слово, с единственным установленным в 1 битом в позиции крайнего правого 0 исходного слова. Например: 10100111 → 00001000

Задача 4-11 Замена инструкций

Не используя команд ROL, ROR, RCL, RCR и SAR, выписать фрагмент программы на ассемблере, реализующий действие указанной команды.

```
a) sar eax, 2     b) rol bx, 1      c) rcr ecx, 1
d) rol ax, 8      e) rcl ebx, 31
```

Задача 4-12 Обращение операций

После выполнения последовательности команд в регистре EAX было получено некоторое значение. Восстановите содержимое ячеек памяти.

```
z dw ___________
xor  eax, eax
or   ax, word [z]
imul eax, eax, 16
; EAX = 0x1ee70
```

5. Управляющие конструкции

Безусловная и условная передача управления

Набор машинных команд IA-32 содержит одну команду безусловного перехода `jmp` и одну команду условного перехода `J`*`cc`*, где *cc* – код условия, связанный с состоянием флагов регистра EFLAGS (см. Табл. 6).

Таблица 6. Связь кодов условий и арифметических флагов.

J*cc*	**Условие**	**Описание**
JE	ZF	Равно / Ноль
JNE	~ZF	Не равно / Не ноль
JS	SF	Отрицательное число
JNS	~SF	Неотрицательное число
JG	~(SF^OF)&~ZF	Больше (знаковые числа)
JGE	~(SF^OF)	Больше либо равно (знаковые числа)
JL	(SF^OF)	Меньше (знаковые числа)
JLE	(SF^OF)|ZF	Меньше либо равно (знаковые числа)
JA	~CF&~ZF	Больше (числа без знака)
JB	CF	Меньше (числа без знака)

Основным способом организации ветвлений и циклов в ассемблерной программе является последовательность из двух действий: (1) выработка значений флагов регистра EFLAGS и (2) выполнение условного перехода. Первый пункт, как правило, реализуется командой CMP, выполняющей с EFLAGS те же действия, что и команда SUB, но не запоминающей результата вычитания, т.е. в результате ее выполнения первый операнд не «разрушается».

```
cmp  eax, 0
je   .12     ; Переход на метку .12 при eax = 0
```

Значение флагов регистра EFLAGS может быть сформировано не только командой CMP. Для реализации реакции программы на значение какого-то определенно флага предусмотрены команды JZ, JS, JC, JO, JP (в коде условия указана первая буква проверяемого флага), выполняющие переход при значении флага 1, и соответ-

ственно команды JNZ, JNS, JNC, JNO, JNP, выполняющие переход при нулевом значении флага.

В качестве примера рассмотрим реализацию реакции на возможное переполнение при выполнении сложения двух беззнаковых целых чисел, расположенных в регистрах eax и ebx.

```
add  eax, ebx
jc  .uns_ov   ; Переход на метку .uns_ov при CF = 1
```

Операции над булевыми переменными

Единственными допустимыми значениями логического типа являются значения 0 и 1, они представляют ложь и истину соответственно. Для хранения логического значения компилятор gcc использует один байт.

Формально, вычисление управляющих условий в операторах должно приводить получению булевской величины, которая затем должна проверяться. Но архитектура IA-32 позволяет сразу же воспользоваться флагами, выработанными арифметическими действиями, что позволяет не использовать лишние, с точки зрения производительности, инструкции.

Тем не менее, в некоторых ситуациях необходимо явно использовать тип _Bool.

```
#include <stdbool.h>

static _Bool p = true;
...
p = !p;
```

```
section .data
   p db 1

section .text
   xor  byte [b], 1
```

В архитектуре IA-32 присутствует инструкция SET*cc*, которая помещает в свой единственный операнд либо 0, либо 1, в зависимости от выполнения условия, указанного в суффиксе кода операции. В качестве операнда может выступать как регистр, так и память, но обязательно размером в один байт.

Пример 5-1 Восстановление типа и операции сравнения

Ниже приведен код на языке Си, где data_t – некоторый целочисленный тип данных, а COMP – некоторая операция сравнения. Пусть переменная a расположена в регистре EDX, а переменная b в регистре ECX. По заданному ассемблерному коду, реализующему данную операцию сравнения, требуется восстановить data_t и COMP (возможно несколько вариантов решения).

```
int comp(data_t a, data_t b) {
   return a COMP b;
}
```

```
CMP ECX, EDX
SETL AL
```

Решение

Поскольку в инструкции SET*cc* используется код L, данные, расположенные в регистрах EAX и EDX рассматриваются как 32 разрядные знаковые числа. В языке Си соответствующим типом может выступать `int` или `long`, причем допустим явно добавленный спецификатор `signed`.

Операцией сравнения является «>», поскольку код L требует выполнения строгого неравенства ОП1 < ОП2 в предыдущей инструкции.

$$ОП1 < ОП2 \Rightarrow ECX < EDX \Rightarrow b < a \Rightarrow a > b$$

Итого, имеем следующий вариант допустимого кода.

```
int comp(int a, int b) {
   return a > b;
}
```

Реализация ветвления и цикла

Типовым приемом при организации ветвления в операторе `if` является вычисление условия и последующая проверка условия в команде условного перехода; если условие не выполнилось – переходим на метку, расположенную сразу после фрагмента кода, который реализует тело `if`.

```
static int *p, a;

if (p) {
   a = *p;
}
```

```
   mov eax, dword [p]
   test eax, eax
   jz .l
   mov  eax, dword [eax]
   mov  dword [a], eax
.l:
```

При реализации оператора `if-else` можно воспользоваться этим же приемом, слегка его дополнив, а можно закодировать в условной передаче уравнения переход на ветку «истина», разместив сразу после условной передачи управления ветку «ложь». Ниже показаны два эквивалентных варианта.

```
static int *p, a, b;

if (p) {
   a = *p;
} else {
   b = -*p;
}
```

```
   mov eax, dword [p]
   test eax, eax
   jz .false
   mov  eax, dword [eax]
   mov  dword [a], eax
   jmp  .if_end
 .false:
   mov  eax, dword [eax]
   neg  eax
   mov  dword [b], eax
 .if_end:
```

```
   mov eax, dword [p]
   test eax, eax
   jnz .true
   mov  eax, dword [eax]
   neg  eax
   mov  dword [b], eax
   jmp  .if_end
 .true:
   mov  eax, dword [eax]
   mov  dword [a], eax
 .if_end:
```

Если переход на метку, расположенную ниже в тексте означает ветвления, то переход на ветку, расположенную выше может привести к коду, который уже выполнялся, что позволяет организовывать циклы.

Ниже приведен фрагмент кода, который подсчитывает число символов введенной строки, включая последний символ – перевод строки.

```
   xor  ebx, ebx     ; Обнулили счетчик
 .loop_cycle:        ; Сюда будем возвращаться
   inc  ebx          ; Увеличиваем счетчик
   call io_get_char  ; Считываем что-то со стандартного входа
   cmp  al, `\n`     ; Сравниваем с переводом строки
   jne  .loop_cycle  ; Если не конец строки – возвращаемся
```

Пример 5-2 Ветви прорастают

Реализуйте на языке ассемблера приведенный Си-код.

```
static long *a, b, c;
...

if (a) {
   if (b > c) {
      *a += b;
   }
} else {
      a = &c;
      *a -= b;
}
```

Решение

Каждая переменная требует для хранения 4 байта памяти, которая должна быть выделена в одной из секций статических данных: `.data` или `.bss`.

Вычисления должны начаться с проверки первого условия. Если условие не будет выполняться – следует передать управление на блок кода `else`, помеченного `.l1`. В противном случае, т.е. когда условие выполняется, проверяем второе условие. Если оно не выполняется, следует сразу же выходить из объемлющего оператора `if-else`. Также и из тела вложенного оператора `if` следует выполнить безусловный переход на эту же метку `.l2`, чтобы не началось выполнение ветки `else` объемлющего оператора `if-else`.

```
section .bss              ; Выделяем необходимую память
   a resd 1
   b resd 1
   c resd 1

section .text
   mov  edx, dword [b]    ; Упреждающе загружаем на регистр
                          ; переменную b. Она потребуется в любом
                          ; случае.
   mov  eax, dword [a]    ; Загружаем на регистр указатель a
   test eax, eax          ; Если указатель нулевой -
   jz   .l1               ; переходим на ветку else
   cmp  edx, dword [c]    ; Если b меньше или равно c -
   jle  .l2               ; сразу покидаем оба оператора
   add  dword [eax], edx  ; Прибавляем к *a значение переменной b
   jmp  .l2               ; Переходим на конец if-else
 .l1:                     ; Начало ветки else
   mov  dword [a], c      ; Присваиваем a адрес переменной c
   mov  eax, c            ; Этот же адрес потребуется на регистре
   sub  dword [eax], edx  ; Вычитаем из *a значение переменной b
 .l2:
```

Пример 5-3 Геометрическая прогрессия

Напишите программу, вычисляющую i-ый элемент геометрической прогрессии. На вход программе дают три целых числа: b, q, i. На стандартный выход требуется напечатать b_i. Нумерация элементов начинается с `0`. Для хранения чисел используйте тип `int`, отслеживать переполнение не требуется.

Решение

Разместим входные данные в регистрах. i будем хранить в `ecx`. Вычисление i-го элемента будет проводиться в цикле, условие окончания которого – нулевой `ecx`.

Если ecx изначально нулевой, тело цикла не должно выполняться. b будем хранить в ebx. Этот же регистр будет использоваться для сохранения результата вычислений i-го элемента. В случае, когда $i = 0$, ebx уже содержит нужное число. Для хранения q будет использоваться edi.

```
extern io_get_dec, io_print_dec

section .text
global main
main:
   call io_get_dec      ; Вводим b
   mov  ebx, eax        ; (Значение ebx не изменится
                        ;  в результате следующих вызовов)
   call io_get_dec      ; Вводим q
   mov  edi, eax        ; (Значение edi не изменится
                        ;  в результате следующего вызова)
   call io_get_dec      ; Вводим i
   mov  ecx, eax        ;
   jmp  .check_condition
 .loop_body:
   imul ebx, edi        ; Вычисляем следующий элемент
   dec  ecx             ; Уменьшаем счетчик не единицу
 .check_condition:
   cmp  ecx, 0          ; Итерируемся, пока счетчик не обнулился
   jne  .loop_body
   mov  eax, ebx
   call io_print_dec    ; Печатаем результат
   mov  eax, 0
   ret
```

Вариант «А»

Тело цикла содержит одно умножение с накоплением результата в регистре ebx и уменьшение счетчика цикла – регистра ecx – на единицу.

В рассмотренном примере для организации цикла были использованы команды условного и безусловного переходов. Приведем фрагмент другой реализации того же примера, где цикл организован с помощью команды LOOP. Считаем, что исходные значения регистров ecx, ebx и edi уже заданы.

```
   jecxz .print         ; Не выполняем тело цикла при ecx = 0
 .loop_body:
   imul ebx, edi        ; Вычисляем следующий элемент
   loop  .loop_body
 .print:
   mov  eax, ebx
   call io_print_dec    ; Печатаем результат
```

Вариант «Б»

Тело цикла будет повторено i раз (именно такое значение находится в регистре ecx перед началом цикла), причем команда jecxz обеспечит корректное выполнение данного фрагмента при ecx = 0.

Короткая логика

Выполнение операций короткой логики && и || позволяет сэкономить время за счет того, что если общий результат заранее понятен, исходя из значения левого операнда – правый операнд не будет вычисляться. Реализация таких операций должна пропускать выполнение части инструкций, что можно обеспечить условной передачей управления.

Рассмотренный выше пример, в котором указатель проверяется на ноль и только потом происходит разыменование, может быть переписан без использования оператора if. Реализующий его код не изменится.

```
static int *p, a;

(p) && (a = *p);
```

```
   mov eax, dword [p]
   test eax, eax
   jz .l
   mov  eax, dword [eax]
   mov  dword [a], eax
 .l:
```

Пример 5-4 Восстановление управляющих конструкций

Требуется записать соответствующий данному фрагменту на ассемблере фрагмент кода на языке Си.

```
SECTION .text
GLOBAL main
main:
   MOV  ESI, DWORD [a]    ; (1)
   TEST ESI, ESI          ; (2)
   JE   .1                ; (3)
   MOV  ECX, DWORD [b]    ; (4)
   TEST ECX, ECX          ; (5)
   JE   .1                ; (6)
   MOV  EDX, DWORD [ESI]  ; (7)
   MOV  EAX, EDX          ; (8)
   SAR  EDX, 31           ; (9)
   IDIV ECX               ; (10)
   SUB  DWORD [ESI], EDX  ; (11)
 .1:
   XOR  EAX, EAX          ; (12)
   RET                    ; (13)
```

Решение

Просматривая ассемблерный код можно заметить обращения к двум статическим переменным a и b. В первый момент можно определить только их разрядность – обе переменные 32-разрядные. Определить, являются они целыми числами или адресами, можно только просмотрев остальной код. Поэтому временно выпишем объявление переменных в следующем виде.

```
static int a, b;
```

Далее рассмотрим поток управления: происходит условная передача управления из инструкций 3 и 6 на помеченную инструкцию 12. Причем перед условной передачей происходит только загрузка значений переменных на регистры и проверка этих значений на равенство с нулем (инструкции TEST и JE) . Таким образом блок инструкций 1-6 может быть представлен в следующем виде:

```
if ((0 != a) && (0 != b)) {
   ...
}
```

Вариант «А»

```
if (0 != a) {
   if (0 != b) {
      ...
   }
}
```

Вариант «Б»

Оба варианта допустимы, однако следует отметить необходимость использовать во варианте «А» именно оператор && короткой логики, поскольку инструкции 4-6 выполняются только в том случае, если не сработал условный переход в инструкции 3.

Теперь рассмотрим блок инструкций 7-11. В первой же инструкции этого блока регистр ESI, в котором размещено значение переменной a используется в адресном коде, т.е. переменная a является указателем, а объявление переменных следует скорректировать.

```
static int *a, b;
```

Значение, взятое из памяти, размещается в паре регистров eax и edx, сразу после чего происходит арифметический сдвиг регистра edx – типовые действия перед делением целых чисел, которое, в свою очередь, выполняется в 10 инструкции. Следует отметить, что используется инструкция знакового деления – в объявление переменных следует внести модификатор signed, но поскольку он задействован по умолчанию для типа int, объявление можно не менять.

Операндами инструкции `idiv` выступили величины `*a` и `b`, поскольку в момент выполнения этой инструкции на регистре `ecx` было расположено именно значение переменной `b`. Однако эта инструкция может применяться для реализации двух операций языка Си – деления и взятия остатка. Выяснить, какая именно операция была реализована можно по тому, какой регистр в дальнейшем использовался в вычислениях. В данном случае это регистр `edx` (он используется в инструкции 11), что приводит к заключению – реализована операция взятия остатка. Использование этого значения заключается в том, что оно вычитается из числа, на которое указывает переменная `a`. В итоге будет составлено следующее выражение на языке Си.

```
*a = *a – *a % b;
```

Собирая весь код вместе и выполняя некоторое преобразование для выражения, содержащегося в теле оператора `if`, получаем следующий код.

```
static int *a, b;
if ((0 != a) && (0 != b)) {
   *a -= *a % b;
}
```

Условная передача данных

Помимо условной передачи управления архитектура IA-32 позволяет непосредственно присваивать значения только в случае выполнения заданного условия. Примером такого присвоения является рассмотренная выше инструкция `SET`*`cc`*. Но ее возможности ограничены – присваивается не произвольное целое число, а только true/false.

В процессор Pentium Pro была добавлена инструкция `CMOV`*`cc`*, позволяющая условно присваивать любое целое число. При выполнении закодированного в инструкции условия происходит пересылка в операнд-регистр содержимого второго операнда, им может быть только регистр или память, непосредственно закодированная величина недопустима.

В приведенном ниже примере порядок выполнения инструкций неизменен, но присвоение в `eax` уменьшенной на 1 величины выполнится только когда `a > 0`.

```
static int a;

(a > 0) && (a--);
```

```
    mov     eax, dword [a]
    mov     edx, eax
    dec     edx
    test    eax, eax
    cmovg   eax, edx
    mov     dword [a], eax
```

Задачи

Задача 5-1 Восстановление типа и операции сравнения

Дан следующий код на языке Си

```
int comp(data_t a) {
   return a COMP 0;
}
```

где data_t – некоторый целочисленный тип данных, а COMP – некоторая операция сравнения. Пусть переменная a расположена в регистре ECX. По заданному ассемблерному коду, реализующему данную операцию сравнения, требуется восстановить data_t и COMP (возможно несколько вариантов решения).

```
(A)
TEST ECX, ECX
SETNE AL

(Б)
TEST CX, CX
SETE AL
```

```
(В)
TEST CL, CL
SETG AL

(Г)
TEST CX, CX
SETA AL
```

Задача 5-2 Восстановление типа и операции сравнения

Ниже приведен код на языке Си, где data_t – некоторый целочисленный тип данных, а COMP – некоторая операция сравнения. Пусть переменная a расположена в регистре EDX, а переменная b в регистре ECX. По заданному ассемблерному коду, реализующему данную операцию сравнения, требуется восстановить data_t и COMP (возможно несколько вариантов решения).

```
int comp(data_t a, data_t b) {
   return a COMP b;
}
```

```
(A)
CMP DX, CX
SETGE AL

(Б)
CMP DL, CL
SETB AL
```

```
(B)
CMP EDX, ECX
SETNE AL
```

Задача 5-3 Реализация тернарного оператора на языке ассемблера

Дан фрагмент Си-кода.

```
int x, y = 100, a = 0, b = 1;
x = y <= 0 ? a : b;
```

Требуется реализовать тернарный оператор `x = (y <= 0? a : b)` на языке ассемблера, используя инструкцию `CMOV`*`cc`*.

Задача 5-4 Модуль числа

Требуется составить программу, которая для заданного с клавиатуры числа печатает модуль этого числа.

Задача 5-5 Реализация тела функции

Дана Си-функция.

```
void cond(int a, int *p) {
   if (p && a > 0) {
      *p += a;
   }
}
```

Требуется написать соответствующий код на языке ассемблера. Размещение формальных параметров функции считать следующим:

```
mov edx, dword [ebp + 8]  ; int a
mov eax, dword [ebp + 12] ; int *p
```

Задача 5-6 Вычисление факториала

Требуется написать ассемблерную программу, вычисляющую факториал заданного числа.

Задача 5-7 Вычисление суммы арифметической прогрессии

Требуется написать ассемблерную программу, вычисляющую сумму первых членов арифметической прогрессии. С клавиатуры задаются: первый член прогрессии, шаг, количество членов прогрессии, которые требуется суммировать.

Задача 5-8 Восстановление структуры программы: ветвления

Дан ассемблерный код, реализующий код Си-функции.

```
 mov eax, dword [ebp + 8]  ; параметр x
 mov edx, dword [ebp + 12] ; параметр y
 cmp eax, -3
 jge l2
 cmp eax, edx
 gle l3
 imull eax, edx
 jmp l4
l3:
 lea eax, dword [eax + edx]
 jmp l4
l2:
 cmp eax, 2
 jg l5
 xor eax, edx
 jmp l4
l5:
 sub eax, edx
l4:
```

Сама Си-функция задана с пропущенными выражениями в операторах.

```
int test(int x, int y) {
   int  val = __________;
   if (______) {
      if (______) {
         val = __________;
      } else {
         val = __________;
      }
   } else if (______) {
         val = __________;
   }
   return val;
}
```

Требуется восстановить пропущенные выражения.

Задача 5-9 Восстановление структуры программы: ветвления

Дан ассемблерный код, реализующий код на языке Си.

```
     mov  ebx, dword [a]
     mov  ecx, dword [b]
     cmp  ebx, ecx
     jg   .L7
     jl   .L8
     xor  eax, eax
     jmp  .L9
.L8:
     mov  edx, dword [c]
     mov  eax, edx
     sar  edx, 31
     sub  ecx, ebx
     idiv ecx
     jmp  .L9
.L7:
     mov  edx, dword [c]
     mov  eax, edx
     sar  edx, 31
     sub  ebx, ecx
     idiv ebx
.L9:
     mov  dword [d], eax
```

```
...
   static ____________________ a;
   static ____________________ b;
   static ____________________ c;
   static ____________________ d;
   if (__________) {
      d = _________________;
   } else if (__________) {
      d = _________________;
   } else {
      d = _________________;
   }
...
```

Требуется восстановить пропуски в Си-коде.

Задача 5-10 Числа с различными цифрами

Напечатать все двузначные числа, у которых цифры не равны между собой.

Задача 5-11 Восстановление структуры программы: циклы

Дан код на языке Си

```
int dw_loop(int x, int y, int n) { //1
{                                  //2
  do {                             //3
     x += n;                       //4
     y *= n;                       //5
     n--;                          //6
  } while ((n  > 0) && (y < n));   //7
  return x;                        //8
}
```

компилятор генерирует следующий ассемблерный код:

```
   mov eax, dword [ebp + 8]  ; (1) переменная x
   mov ecx, dword [ebp + 12] ; (2) переменная y
   mov edx, dword [ebp + 16] ; (3) переменная n
L2:                          ; (4)
   add eax, edx              ; (5)
   imul ecx, edx             ; (6)
   sub edx, 1                ; (7)
   test edx, edx             ; (8)
   jle L5                    ; (9)
   cmp ecx, edx              ; (10)
   jl L2                     ; (11)
L5:                          ; (12)
```

Разметить ассемблерный код комментариями, выделив в нем выражение-условие, проверяемое в цикле, и операторы, составляющие тело цикла в Си-коде.

Задача 5-12 Переход на метку

Указать, на какую метку будет осуществлен переход при выполнении следующего фрагмента ассемблерного кода:

```
mov al, -80
cmp al, 150
jb M1
jl M2
jmp M3
```

Задача 5-13 Сумма цифр числа

Найти сумму цифр заданного неотрицательного целого числа типа `unsigned int`.

Задача 5-14 Число Фибоначчи

Найти первое по порядку число Фибоначчи, превосходящее заданное неотрицательное значение N ($N > 1$). Числа Фибоначчи задаются соотношением: $F_0 = F_1 = 1$, $F_k = F_{k-1} + F_{k-2}$, $k \geq 2$.

Задача 5-15 Максимум последовательности

Дана последовательность из 30 целых чисел. Найти:

a) максимум данной последовательности,
b) номер максимального элемента последовательности (считать, что последовательность не содержит повторяющихся элементов),
c) количество вхождений максимального элемента в последовательность.

Текст учебной библиотеки ввода/вывода

Файл `macro.c`[1], содержимое которого приведено ниже, распространяется вместе со скриптом для сборки учебных программ `build_asm.sh` (который помимо прочего включает определенные в нем функции в итоговый исполнимый файл), а также в составе учебной среды разработки с открытым исходным кодом SASM[2].

```
#include <stdio.h>

FILE *get_stdin(void) { return stdin; }
FILE *get_stdout(void) { return stdout; }

// I/O functions easily callable from asm
// - Accept under-aligned stack
// - Return value (if any) in EAX
// - Up to three arguments in EAX, EDX, ECX
// - Preserve EBX, EBP, ESP, EDI, ESI
//
// Extends macro.c from SASM (usually /usr/share/sasm/NASM/macro.c)

#define IO_ATTR __attribute__((regparm(3),force_align_arg_pointer))

// Input facilities

// Read a 32-bit number using %d/%u/%x format, return value in EAX
IO_ATTR int      io_get_dec(void);
IO_ATTR unsigned io_get_udec(void);
IO_ATTR unsigned io_get_hex(void);

// Read a character via getchar(), return value in EAX
IO_ATTR int io_get_char(void);

// Read a string, 'buf' in EAX, 'size' in EDX
IO_ATTR void io_get_string(char *buf, int size);

// Output facilities

// Print a 32-bit number in EAX using %d/%u/%x format
IO_ATTR void io_print_dec(int v);
IO_ATTR void io_print_udec(unsigned v);
IO_ATTR void io_print_hex(unsigned v);

// Print a character in AL
IO_ATTR void io_print_char(char c);

// Print a string addressed by EAX
IO_ATTR void io_print_string(const char *s);
```

[1] Название файла обусловлено историческими причинами.
[2] https://dman95.github.io/SASM/

```
// Start a new line
IO_ATTR void io_newline(void);

IO_ATTR
int io_get_dec(void)
{
      int r;
      scanf("%d", &r);
      return r;
}

IO_ATTR
unsigned io_get_udec(void)
{
      unsigned r;
      scanf("%u", &r);
      return r;
}

IO_ATTR
unsigned io_get_hex(void)
{
      unsigned r;
      scanf("%x", &r);
      return r;
}

IO_ATTR
int io_get_char(void)
{
      return getchar();
}

IO_ATTR
void io_get_string(char *buf, int size)
{
      fgets(buf, size, stdin);
}

IO_ATTR
void io_print_dec(int v)
{
      printf("%d", v);
}

IO_ATTR
void io_print_udec(unsigned v)
{
      printf("%u", v);
}
```

```
IO_ATTR
void io_print_hex(unsigned v)
{
     printf("%x", v);
}

IO_ATTR
void io_print_char(char c)
{
     putchar(c);
}

IO_ATTR
void io_print_string(const char *s)
{
     fputs(s, stdout);
}

IO_ATTR
void io_newline(void)
{
     putchar('\n');
}

__attribute__((constructor))
static void unbuffer_stdout(void)
{
     setbuf(stdout, 0);
}
```

Ответы и решения

Задача 1-1

Двоичное представление (байт)	16-ричное представление (байт)	Число десятичное со знаком	Число десятичное без знака
10111101	BDh	-67	189
01110110	76h	118	118
00101000	28h	40	40
10000111	87h	-121	135

Задача 1-5

(а) EAX = 0xdde

(б) EAX = 0xd

(в) EAX = 0xcafe

(г) EAX = 0xfffffffe

(д) EAX = 0xffff00ba

(е) EAX = 0xfffffef0

(ж) EAX = 0xffff000d

(з) EAX = 0xffff006f

(и) EAX = 0xff79000d

(к) EAX = 0x2179fffe

Задача 1-6

```
section .bss
   a resd 1
   b resd 1

section .text
   mov al, byte [a]
   mov byte [b], al
   mov al, byte [a + 1]
   mov byte [b + 3], al
   mov ax, word [a + 2]
   mov word [b + 1], ax
```

Задача 2-1

```
; a
movzx eax, byte [a]
mov dword [b], eax

; b
movsx eax, byte [a]
mov word [b], eax
```

Задача 2-2

(a) $AL_{знаковое}$ = -118, $AL_{беззнаковое}$ = 138, CF = 1, OF = 0, ZF = 0, SF = 1.

(b) $AL_{знаковое}$ = 5, $AL_{беззнаковое}$ = 5, CF = 0, OF = 0, ZF = 0, SF = 0.

(c) $AL_{знаковое}$ = -90, $AL_{беззнаковое}$ = 166, CF = 1, OF = 0, ZF = 0, SF = 1.

(d) $AL_{знаковое}$ = 58, $AL_{беззнаковое}$ = 58, CF = 1, OF = 0, ZF = 0, SF = 0.

Задача 3-1

```
void decode2(int *xp, int *yp, int *zp) {
    int y = *yp;

    *yp = *xp + *zp;
    *xp = *yp;
    *zp = y;
}
```

Задача 4-4

```
eax = -8
edx = 65528
```

Задача 4-12 Обращение операций

```
z dw 0x1ee7
xor  eax, eax
or   ax, word [z]
imul eax, eax, 16
; EAX = 0x1ee70
```

Задача 5-1

```
(A)
data_t – int, unsigned, указатель,
         для целочисленных типов допустим спецификатор long
COMP - !=

(Б)
data_t – short или unsigned short
COMP - ==

(В)
data_t – char
COMP - >

(Г)
data_t – unsigned short
COMP - >
```

Задача 5-2

```
(А)
data_t – short
COMP - >=

(Б)
data_t – unsigned char
COMP - <

(В)
data_t – int, unsigned, указатель;
         для целочисленных типов допустим спецификатор long
COMP - !=
```

Задача 5-3

```
section .bss
    x resd 1

section .data
    y dd 100
    a dd 0
    b dd 1

section .text
global main
main:
    cmp dword [y], 0
    mov eax, dword [b]
    cmovle eax, dword [a]
    mov dword [x], eax
```

Задача 5-4

```
extern io_get_dec, io_print_dec, io_newline

section .text
global main
main:
   call io_get_dec
   cmp eax, 0
   jge label
   neg eax
 label:
   call io_print_dec
   call io_newline
   xor eax, eax
   ret
```

Задача 5-5

```
test eax, eax
 jz label
 test edx, edx
 jle label
 add dword [eax], edx
label:
```

Задача 5-6

```
extern io_get_dec, io_print_dec

section .text
global main
main:
   call io_get_dec
   mov ecx, eax
   mov eax, 1
   jecxz end_label
  loop_body:
   imul eax, ecx
   loop loop_body
  end_label:
   call io_print_dec
   mov eax, 0
```

Задача 5-7

```
extern io_get_dec, io_print_dec, io_newline

section .text
global main
main:
   call io_get_dec ; первый член
   mov esi, eax
   call io_get_dec ; шаг прогрессии
   mov edi, eax
   call io_get_dec ; количество шагов
   mov ecx, eax

   xor eax, eax
   jecxz .end_label
 .loop_body:
   mov eax, esi
   add esi, edi
   loop .loop_body
 .end_label:
   call io_print_dec
   call io_newline
   xor eax, eax
   ret
```

Задача 5-8

```
int test(int x, int y) {
   int  val = x^y;
   if (x < -3) {
      if (y < x) {
         val = x * y;
      } else {
         val = x + y;
      }
   } else if (x > 2) {
      val = x - y;
   }
   return val;
}
```

Восстанавливать можно по следующей методике. В первую очередь выявляются линейные блоки. Строится граф потока управления для ассемблерной программы. Этот граф сопоставляется с предложенным шаблоном программы. В концах базовых блоков выявляются пары инструкций `CMP Jcc`, которые позволяют восстановить условия операторов `if` и их ветви. Следует отметить, что для заданного ассемблерного кода будет корректно и такое восстановление.

```
int test(int x, int y) {
   int val;
   if (x < -3) {
      if (y < x) {
         val = x * y;
      } else {
         val = x + y;
      }
   } else if (x > 2) {
     val = x - y;
   } else {
     val = x^y;
   }
   return val;
}
```

Задача 5-9

```
...
   static int a;
   static int b;
   static int c;
   static int d;
   if (a > b) {
      d = c / (a - b);
   } else if (a < b) {
      d = c / (b - a);
   } else {
      d = 0;
   }
...
```

Задача 5-10

```
extern io_print_dec, io_newline

section .text
global main
main:
        mov esi, 1
.loop1:                     ; цикл по старшей цифре
        cmp esi, 10
        jge .exit1
        xor edi, edi
.loop2:                     ; цикл по младшей цифре
        cmp edi, 10
        jge .exit2
        cmp esi, edi        ; проверка цифр на неравенство
        je .lc
        mov eax, esi
        call io_print_dec
        mov eax, edi
        call io_print_dec
        call io_newline
.lc:
        inc edi
        jmp .loop2
.exit2:
        inc esi
        jmp .loop1
.exit1:
        xor esi, esi
        ret
```

Задача 5-11

```
 mov eax, dword [ebp + 8]  ; (1) переменная x
 mov ecx, dword [ebp + 12] ; (2) переменная y
 mov edx, dword [ebp + 16] ; (3) переменная n
L2:                        ; (4)
 add eax, edx              ; (5)  x += n;
 imul ecx, edx             ; (6)  y *= n;
 sub edx, 1                ; (7)  n--;
 test edx, edx             ; (8)  n v 0
 jle L5                    ; (9)  n <= 0 – выходим из цикла
 cmp ecx, edx              ; (10) y v n
 jl L2                     ; (11) y < n
                           ;      переходим на новую итерацию
L5:                        ; (12)
```

Задача 5-12

Ответ: M3

Решение:

```
mov al, -80; доп(-80) = 176
cmp al, 150; 150 = доп(-106)
jb M1
jl M2
jmp M3
```

Переход на метку M1 не будет осуществлен, поскольку 176 > 150; переход на метку M2 также не будет осуществлен, т.к. -80 > -106. Будет выполнен переход на метку M3.

Литература

1. Рэндал Э. Брайант, Дэвид О'Халларон. Компьютерные системы: архитектура и программирование (Computer Systems: A Programmer's Perspective). Издательство: БХВ-Петербург, 2005 г. — 1186 стр.

2. Henry S. Warren. Hacker's Delight (2nd Edition). / Addison-Wesley Professional; 2 edition (October 5, 2012) — p. 512

3. А.А. Белеванцев, С.С. Гайсарян, Л.С. Корухова, Е.А. Кузьменкова, В.С. Махнычев. Семинары по курсу "Алгоритмы и алгоритмические языки" (учебно-методическое пособие для студентов 1 курса). М.: Издательский отдел факультета ВМК МГУ имени М.В. Ломоносова

4. А.А. Белеванцев, С.С. Гайсарян, В.П. Иванников, Л.С. Корухова, В.А. Падарян. Задачи экзаменов по вводному курсу программирования (учебно - методическое пособие). М.: Изд. отдел ф-та ВМК МГУ имени М.В. Ломоносова, 2012.

5. К.А. Батузов, А.А. Белеванцев, Р.А. Жуйков, А.О. Кудрявцев, В.А. Падарян, М.А. Соловьев. Практические задачи по вводному курсу программирования (учебное пособие). М.: Изд. отдел ф-та ВМК МГУ имени М.В. Ломоносова, 2012.