МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М.В. Ломоносова

Факультет вычислительной математики и кибернетики

Е.А. Кузьменкова, В.А. Падарян, М.А. Соловьев

Семинары по курсу "Архитектура ЭВМ и язык ассемблера"

(учебно-методическое пособие)

Часть 2

МАКС ПРЕСС

Москва – 2014

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.А. Падарян, М.А. Соловьев

К89 **Семинары по курсу «Архитектура ЭВМ и язык ассемблера»: учебно-методическое пособие. Часть 2.** — М. Издательский отдел факультета ВМиК МГУ им. М.В. Ломоносова (лицензия ИД № 05899 от 24.09.2001); МАКС Пресс, 2014. – 100 с.

ISBN 978-5-89407-530-3

ISBN 978-5-317-04886-0

Учебное пособие содержит задачи и упражнения по второй части семинарских занятий курса «Архитектура ЭВМ и язык ассемблера», прочитанного студентам 1 потока 1 курса факультета Вычислительной математики и кибернетики МГУ в 2010-2014 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 second 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 2010-2014. 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 978-5-89407-530-3

ISBN 978-5-317-04886-0

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 второй половины семестра, такие как: организация циклов, работа с агрегатными типами данных, различные соглашения вызовов функций, работа с числами с плавающей точкой. Особое внимание уделено двоичному (бинарному) интерфейсу приложений платформы IA-32. На примере компилятора gcc показывается, как в Си-программах реализуется этот интерфейс, и каким образом он может повлиять на скорость работы программы и ее безопасность.

Для разбора задач требуется свободное владение материалом первой части курса. 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.

1. Циклы и многомерные массивы

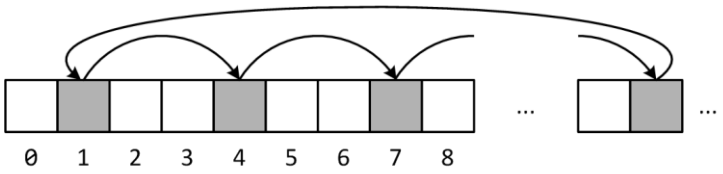
Работа с массивами требует от профессионального разработчика эффективного решения ряда задач: организации циклов, возможно многократно вложенных, кодирования индексных выражений и четкого понимания, как именно данные размещены в памяти. От того, насколько успешно решаются эти задачи, зависит длительность отладки кода и достигаемая программой производительность.

Пример 1-1 Циклический сдвиг в массиве

Дан статический массив.

```
enum { BUF_SIZE = 256 };
static int buf[BUF_SIZE];
```

Требуется привести фрагменты программ на языке Си и на языке ассемблера, которые осуществляют следующее преобразование массива: выполняется циклический сдвиг подмножества элементов массива, расположенных на расстоянии в 3 элемента друг от друга, начиная с элемента с индексом 1.



Решение

В первую очередь приведем реализацию на языке Си.

Вариант «А»

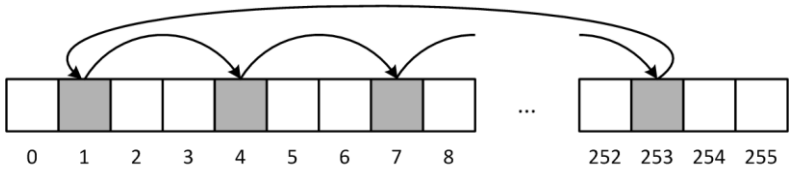
```
int previous = buf[1];
for (int i = 4; i < 256; i += 3) {
   int tmp = buf[i];
   buf[i] = previous;
   previous = tmp;
}
buf[1] = previous;
```

Вариант «Б»

```
int i = 253;
while (i >= 4) {
   int tmp;
   tmp = buf[i];
   buf[i] = buf[i - 3];
   buf[i - 3] = tmp;
   i -= 3;
}
```

Вариант «А» реализует обход элементов массива в прямом порядке – от младших индексов к старшим. Если в теле цикла сразу выполнить присваивание `buf[i] = buf[i-3];` то текущее значение элемента `buf[i]` будет безвозвратно потеряно. На следующей итерации в присваивании будет участвовать значение не предыдущего элемента, а того, чье значение присваивалось на предыдущей итерации. В результате весь массив будет заполнен значением, взятым из элемента `buf[1]`. Чтоб избежать этого, введены вспомогательные переменные `previous` и `tmp`. Первая содержит в себе значение предыдущего элемента массива до присваивания. Вторая (`tmp`) используется для того, чтоб это значение не потерялось при присваивании. Относительной простой такой реализации является простота условия окончания цикла – нельзя выходить за пределы массива.

Вариант «Б» предполагает обход массива в обратном направлении, что позволяет получить более элегантное решение. Особенно заметны отличия вариантов на уровне ассемблерного кода. Идея заключается в том, что на каждой итерации элементы обмениваются своими значениями. Относительной сложностью становится определение наибольшего индекса, участвующего в обмене. Обмен начинается со второго элемента (с индексом `1`). Индексы удовлетворяют условию $0 \le 3 \times n + 1 < 256$, где n – целое число. Поскольку `255` делится на `3` нацело, наибольший индекс – `253`. Два последних элемента массива в обмене не участвуют, а третий с конца – участвует.



В результате обменов значение из `buf[253]` перемещается в `buf[1]`, остальные значения сдвигаются по массиву в конец, как и требовалось в условии задачи.

Имея реализацию на языке Си, переведем код на язык ассемблера.

Вариант «А»

```
; распределяем регистры
; ecx – int i
; eax – int tmp
; edx – int previous

   mov  edx, dword [buf + 4]          ; previous = buf[1];
   mov  ecx, 16                       ; i = 4; не забываем умножать на
                                      ;        sizeof(int) все то, что
                                      ;        относится к адресной
                                      ;        арифметике. В данном случае –
                                      ;        это регистр ecx
 .loop:
   cmp  ecx, 1024                     ; i v 256
   jge  .loop_end                     ; if (i >= 256) goto .loop_end;
   mov  eax, dword [buf + 4 * ecx]    ; tmp = buf[i];
   mov  dword [buf + 4 * ecx], edx    ; buf[i] = previous;
   mov  edx, eax                      ; previous = tmp;
   add  ecx, 12                       ; i += 3;
   jmp  .loop                         ; переходим на следующую итерацию цикла
 .loop_end:
   mov  dword [buf + 4], edx          ; buf[1] = previous;
```

Вариант «Б»

```
; распределяем регистры
; ecx – int i
; eax – вспомогательный регистр при обмене значениями между элементами
;       массива buf[i] и buf[i - 3]

   mov  ecx, 1012                       ; i = 253;
.loop:
   mov  eax, dword [buf + 4 * ecx]      ; инструкция xchg позволяет сократить
   xchg eax, dword [buf + 4 * ecx - 12] ; количество инструкций при обмене
   mov  dword [buf + 4 * ecx], eax      ; значениями
   sub  ecx, 12                         ; i -= 3;
   cmp  ecx, 16                         ; i v 4
   jge  .loop                           ; if (i >= 4) goto .loop;
; поскольку заранее известен размер массива, проверку можно перенести в
; конец тела цикла, избавившись от инструкции безусловной передачи управления
```

Компактное кодирование циклов

Исторически в наборе команд IA-32 присутствует поддержка цикла `do-while` в виде команд `LOOP`, `LOOPNE`, `LOOPE`, `JCXZ/JECXZ`. Они выполняют условные переходы, исходя из состояния регистра `ECX` (`CX`), который используется в качестве счетчика итераций.

Выполнение команды `LOOP/LOOP`*`cc`* уменьшает значение `ECX` на единицу, после чего делается условный переход, если выполняется соответствующее условие. Если `ECX` изначально быть равен `0`, то после выполнения команды из семейства `LOOP` (и передачи управления на метку цикла), `ECX` станет `0xFFFFFFFF`, что (скорее всего) приведет к ошибочной работе программы.

Для компактной записи проверки ECX на 0 следует воспользоваться командой JECXZ, позволяющей «перепрыгнуть» цикл, когда ECX равен 0. Команда JCXZ работает аналогично, но с регистром CX.

Еще одной особенностью является то, что условный переход в командах LOOP/LOOP*cc* может быть только в пределах [-128, 127] байтов от текущей позиции в коде. Закодировать цикл с достаточно объемным телом не получится, т. к. ассемблер не сможет построить код для инструкции, выполняющей переход на слишком удаленную метку.

Таблица 1. Описание команд аппаратной поддержки цикла.

Команда	Описание
JCXZ/JECXZ	Переход выполняется, если значение регистра CX/ECX равно нулю.
LOOP	Переход выполняется, если значение регистра ECX не равно нулю.
LOOPZ/LOOPE	Переход выполняется, если значение регистра ECX не равно нулю и флаг ZF установлен.
LOOPNZ/LOOPNE	Переход выполняется, если значение регистра ECX не равно нулю и флаг ZF сброшен.

Пример 1-2 Аппаратная поддержка цикла

В статическом буфере памяти B хранятся 32-х разрядные числа. Длина буфера (в элементах) задана в регистре ECX, требуется найти в буфере B все нечетные числа.

Решение

Важной особенностью применения команды LOOP является то, что счетчиком цикла обязательно становится регистр ECX, его нельзя использовать для хранения каких-либо иных данных. Помимо того, при использовании команды LOOP счетчик цикла не увеличивается от нуля до некоторого порогового значения, а наоборот, начиная с порогового значения, уменьшается до единицы, что требует аккуратности при использовании ECX в вычислении индексных выражений. Ниже приведено решение, учитывающее эти особенности.

```
   mov   eax, 0                    ; (1)
   jecxz .le                       ; (2)
.l:
   bt    dword [B + 4*ecx -4], 0   ; (3)
   adc   eax, 0                    ; (4)
   loop  .l                        ; (5)
.le:
```

Регистр EAX используем для накопления числа нечетных элементов. Во второй команде происходит проверка длины массива, если длина нулевая – сразу переходим на выход из цикла. В третьей команде проверятся нулевой бит элемента массива.

Поскольку порядок обхода элементов массива не важен, непосредственно используем ECX в вычислении индексного выражения. Умножаем счетчик на размер типа и дополнительно вычитаем из адреса 4, т. к. значения счетчика сдвинуты на единицу относительно индексов массива. Первая итерация цикла будет выполняться с ECX = N, где N – длина массива, а обращение к массиву должно быть B[N-1]. Последняя итерация происходит с ECX = 1, но обращаться необходимо по адресу B + 0.

Выполнившаяся команда BT поместит последний бит числа во флаг CF, который в следующей, четвертой команде будет добавлен к регистру EAX.

Последняя, пятая, команда будет возвращать управление на метку .1 до тех пор, тело цикла не выполнится с ECX = 1.

Многомерный массив

При интерпретации объявления двумерного массива следует помнить о приоритете операций: с именем переменной ассоциируются ближайшие справа квадратные скобки, остальные размерности следует связывать типом элемента многомерного массива.

При объявлении переменной int m[20][14] в памяти необходимо последовательно и без промежутков разместить 20 переменных, имеющих тип int[14]. Если переменная статическая, в ассемблерной программе метка m будет использоваться для обращения к началу этой памяти (в точности, как и в случае с одномерными массивами). Таким образом, в языке Си размещение двумерного массива в памяти происходит по строкам, поскольку первый индекс принято считать номером строки, а второй – номером столбца.

Такая интерпретация позволяет использовать правила умолчания, когда размер массива неизвестен: например, допустимо объявление int q[][14], но недопустимо int qq[14][]. Правила умолчания предписывают в отсутствии размера создать массив из одного элемента. В первом случае массив q состоит из одного элемента полного типа int[14], а во втором тип неполон. Создать массив qq из 14 элементов неполного типа (его размер неизвестен) не получится.

При обращении к элементу матрицы адрес элемента определяется как сумма начального адреса массива и смещений по первому и второму индексам. Как и в случае с одномерным массивом смещение получается умножением значения индексного выражения на размер соответствующего типа. Следовательно, при подсчете смещения, порождаемого первым индексом размер необходимо определять для типа `T[M]`.

Итого, для обращения к элементу `matrix[i][j]` матрицы `T matrix[N][M]`необходимо вычислить адрес $\chi_{matrix} + i \times sizeof(T[M]) + j \times sizeof(T)$, где χ_{matrix} – начальный (базовый) адрес переменной `matrix`.

Пример 1-3 Матрица целых чисел

Напишите фрагмент ассемблерной программы, в которой:

1) в статической памяти выделено пространство для матрицы 32-х разрядных целых чисел размером `N*N`,
2) выполняется поиск максимального по модулю элемента главной диагонали,
3) найденное число печатается на стандартный вывод.

Решение

Для размещения двумерного массива `T array[N][N]` необходимо выделить непрерывный блок памяти длиной в `sizeof(T)*N*N` байтов. Поскольку в условии ничего не говорится о начальных значениях элементов массива, матрица будет размещена в сегменте `.bss`.

Ввод данных во фрагменте пропущен. Для вычислений используются три регистра: `EAX` – для хранения максимального модуля, `ECX` – для извлечения элементов массива, `EDX` – для работы с текущим элементом.

Загрузка очередного элемента соответствует обращению к элементу Си-массива с двойным индексным выражением.

```
int a[N][N];
a[i][i];
```

Двумерный массив размещен в памяти построчно: начиная с базового адреса идет первая строка (с индексом `0`), затем вторая и т. д. Между строками нет промежутков, внутри строки элементы также идут без промежутков, поскольку строка – одномерный массив целых чисел. Обращение к каждой последующей строке требует добавления к базовому адресу массива смещения величиной в размер строки.

Строка – массив из `N` элементов типа `int`, его размер – `4*N` байтов. Обращение к последующему элементу в столбце – требует добавления `4` байтов. На каждой итерации происходит сдвиг на одну строку и один столбец, что означает дополнительное смещение от базового адреса на `4*(N+1)` байтов. Это приращение происходит в конце цикла, после чего проверяется, не вышло ли суммарное смещение от начала массива на его пределы.

```
section .bss
N equ …           ; определяем константу N с некоторым значением
   a resd (N * N) ; выделяем N*N элементов по 4 байта

section .text
global CMAIN
CMAIN:
   ; ввод матрицы

   mov   eax, 0               ; В регистре eax – максимальное по модулю
                              ; значение диагональных элементов
   mov   ecx, 0               ; ecx – счетчик итераций и заодно смещение от
                              ; начала блока памяти

 .l:                          ; метка – начало тела цикла
   mov   edx, dword [a + ecx] ; Загружаем очередной диагональный элемент
   cmp   edx, 0               ; Сравниваем его с 0
   jge   .pos                 ; и если он меньше -
   neg   edx                  ; меняем его знак
 .pos:
   cmp   edx, eax             ; Сравниваем текущий модуль с максимальным и
   cmovg eax, edx             ; если он больше – пересылаем его в eax
   add   ecx, 4 * (N + 1)     ; Увеличиваем смещение на одну строку и один
   cmp   ecx, 4 * (N * N)     ; столбец. Сравниваем текущее смещение с
   jl    .l                   ; границей выделенной памяти.

   PRINT_DEC 4, eax
   NEWLINE
   xor   eax, eax
   ret
```

Пример 1-4 Транспонирование матрицы

Дана матрица целых чисел: `static int A[M][N]`. Требуется написать фрагмент программы, транспонирующий эту матрицу с запоминанием результата в другой матрице: `static int B[N][M]`.

Решение

На языке Си запись требуемого алгоритма имеет следующий вид.

```
static int A[M][N];
static int B[N][M];

for (int i = 0; i < M; i++) {
   for (int j = 0; j < N; j++) {
      B[j][i] = A[i][j];
   }
}
```

Переводим построенный код на язык ассемблера. Используем регистры ESI и EDI для хранения переменных цикла, регистр ECX – для вычисления смещения от начала массива в двойных словах.

Вариант «А»

```
section .bss
   A resd M*N
   B resd N*M

section .text

   mov  esi, 0 ; индекс i

 .l1:                          ; Начинается тело внешнего цикла
   mov  edi, 0 ; индекс j

 .l2:                          ; Начинается тело внутреннего цикла
   imul ecx, esi, N            ; Смещаемся на esi строк по N элементов каждая
   add  ecx, edi               ; … и еще на edi элементов (массив A)
   mov  eax, dword [A + 4*ecx] ; Считываем элемент по вычисленному смещению
   imul ecx, edi, M            ; Аналогично считаем смещение в массиве B
   add  ecx, esi
   mov  dword [B + 4*ecx], eax ; и записываем значение элемента

   inc  edi                    ; Проверка счетчика внутреннего цикла
   cmp  edi, N
   jl   .l2

   inc  esi                    ; Проверка счетчика внешнего цикла
   cmp  esi, M
   jl   .l1
```

Как можно заметить, обращение к элементам массива A приводит к последовательному считыванию содержимого памяти, что позволяет избавиться от одного умножения во вложенном цикле. Идея заключается в том, что смещение для массива A хранится в дополнительно выделенном регистре EDX, на каждой итерации внутреннего цикла EDX будет увеличиваться на единицу. Новые команды в варианте «Б» показаны серым цветом. Замена двойного цикла на одинарный не представляется целесообразной, т. к. преобразование линейного смещения в индексы матрицы B дополнительно потребует вычисления частного и остатка, что является достаточно дорогой операцией.

Вариант «Б»

```
section .bss
   A resd M*N
   B resd N*M

section .text

   mov  esi, 0 ; индекс i
   mov  edx, 0

 .l1:
   mov  edi, 0 ; индекс j

 .l2:
   mov  eax, dword [A + 4*edx]
   imul ecx, edi, M
   add  ecx, esi
   mov  dword [B + 4*ecx], eax

   inc  edx
   inc  edi
   cmp  edi, N
   jl   .l2

   inc  esi
   cmp  esi, M
   jl   .l1
```

Пример 1-5 Восстановление размеров массива

В следующем фрагменте Си-программы используются константы `R` и `S`.

```
int A[R][S];

int extract(int i, int j, int *dest) {
   *dest = A[i][j];
   return sizeof(A);
}
```

Для тела данной функции был получен следующий ассемблерный код. Формальные параметры функции расположены в памяти по следующим адресам: `i` – `EBP+8`, `j` – `EBP+12`, `dest` – `EBP+16`.

```
   imul eax, dword [ebp + 8], 90 ; (1)
   add eax, dword [ebp + 12]     ; (2)
   mov edx, dword [A + eax * 4]  ; (3)
   mov eax, dword [ebp + 16]     ; (4)
   mov dword [eax], edx          ; (5)
   mov eax, 3960                 ; (6)
```

Определите значения констант `R` и `S`.

Решение

```
imul eax, dword [ebp + 8], 90 ; (1) В регистр eax помещается значение
                              ;     выражения i * 90
add  eax, dword [ebp + 12]    ; (2) Добавляем j. итого, в eax находится
                              ;     значение выражения i * 90 + j
mov  edx, dword [A + eax * 4] ; (3) В регистр помещается 32-разрядное
                              ;     значение из памяти по адресу
                              ;     a + 4 * (i * 90 + j)
mov  eax, dword [ebp + 16]    ; (4) Теперь в eax помещается значение
                              ;     параметра dest, являющегося указателем
mov  dword [eax], edx         ; (5) По этому адресу (*dest) кладется
                              ;     содержимое edx
mov  eax, 3960                ; (6) Возвращаемое значение – 3960
```

Для произвольного типа `T` размер двумерного массива `T arrary[`N_1`][`N_2`]` определятся по формуле `sizeof(T)*`N_1`*`N_2. Размер всего массива можно определить из инструкции №6, в регистр `EAX` помещается константа, являющаяся значением `sizeof(A)`. Таким образом,

`sizeof(A) = 3960` ⇨ `sizeof(int)*R*S = 3960` ⇨ `R*S = 990`

Ключевым моментом является определение порядка вычисления адреса памяти, используемого в инструкции №3. В адресе `A+4*(i*90+j)` реализовано обращение к элементу массива `A[i][j]`. Первый индекс означает, что необходимо `i` раз переместится на расстояние из `S` элементов. Таким образом, константа `S = 90`. Из приведенного выше уравнения можно определить, что `R = 11`.

Массив указателей

Массивы указателей позволяют применять два индексных выражения, как и в случае с матрицей. Но массив указателей имеет два существенных отличия. Во-первых, при объявлении такой переменной элементом одномерного массива является указатель, всегда требующий для своего размещения 4 байта. Во-вторых, элемент скалярного типа получается уже после вычисления первого индекса, после чего необходимо обратиться в память и извлечь оттуда элемент-адрес. Далее к этому извлеченному адресу прибавляется смещение, полученное при вычислении второго индексного выражения. Таким образом, для объявления `T* pA[N]` вычисление адреса элемента `p[i][j]` будет соответствовать формуле $M[\chi_{pA} + i \times 4] + j \times sizeof(T)$, где χ_{pA} – начальный (базовый) адрес переменной `pA` , а `M[A]` – содержимое памяти по адресу `A`.

Пример 1-6 Массив указателей vs. Двумерный массив

Дан массив некоторой ненулевой длины, содержащий ненулевые указатели на ограниченные нулем строки символов. Строки преобразуются с помощью массива подстановок. Переведите Си-код на язык ассемблера.

```
static char *strings[N];
static char tr[8][256];

for (int i = 0; i < N; i++) {
   for int (j = 0; '\0' != strings[i][j]; j++) {
      strings[i][j] = tr[i % 8][(unsigned char)strings[i][j]];
   }
}
```

Решение

На языке Си код, обращающийся к массиву указателей, ничем не отличается от обращения к двумерному массиву: дважды используется операция индексирования. Но типы данных массивов требуют реализовывать эти обращения различными способами. Вычисление `strings[i][j]` потребует сначала извлечь из памяти указатель `strings[i]`, а затем использовать его в качестве адреса при обращении к очередному символу в преобразуемой строке. Обращение к двумерному массиву `tr` требует вычислить смещение в непрерывном блоке памяти, выделенном для его размещения. Помимо того, различие между переменными `strings` и `tr` проявляется в выделении памяти: для первой был выделен одномерный массив из `N` двойных слов (хранятся указатели), для второй – `8*256` байтов, в которых последовательно размещены строки двумерного массива.

```
section .bss          ; выделяем память под переменные
   strings resd N
   tr      redb 8*256

section .text
   mov   ecx, 0                       ; В ecx размещаем i. Т.к. массив строк
                                      ; не пустой – предварительно проверять
 .l1:                                 ; i не требуется
   mov   ebx, dword [strings + 4*ecx] ; Помещаем в ebx strings[i] – адрес
                                      ; начала в памяти очередной строки
   cmp   byte [ebx], 0                ; Перед первым выполнением тела for
   jz    .l2e                         ; проверяем условие выхода из цикла,
 .l2:                                 ; поскольку строка может быть пустой
   movzx edx, byte [ebx]              ; Беззнаковое расширение char
   mov   eax, ecx                     ; Вычисляем смещение для массива tr
   and   eax, 7                       ; eax – смещение на (i%8) строк,
   sal   eax, 8                       ; умноженное на 256
   mov   dl, byte [tr + eax + edx]    ; edx – смещение внутри строки
   mov   byte [ebx], dl               ; Сохраняем новое значение
   inc   ebx                          ; Переходим на адрес очередного символа
   cmp   byte [ebx], 0
   jnz   .l2                          ; Продолжаем, если strings[i][j] != 0

 .l2e:
   inc   ecx
   cmp   ecx, N
   jl    .l1                          ; Продолжаем внешний цикл, пока i < N
```

Следует отметить, что условие указывает на то, что в массиве `strings` находятся ненулевые указатели. В противном случае потребуется дополнительная проверка, и корректный Си-код будет выглядеть следующим образом.

```
static char *strings[N];
static char tr[8][256];

for (int i = 0; i < N; i++) {
   if (NULL != strings[i]) {
      for int (j = 0; '\0' != strings[i][j]; j++) {
         strings[i][j] = tr[i % 8][(unsigned char)strings[i][j]];
      }
   }
}
```

Задачи

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

Задача 1-1

Заданы два массива целых чисел: массив `short A[100]` и массив `int B[100]`. Требуется выполнить копирование всех элементов массива `A` в массив `B` со знаковым расширением.

Задача 1-2

В переменной `A` хранится указатель на область памяти, содержащую слова, количество которых указано в переменной `N`. Требуется расширить каждое из этих слов до двойного слова «на месте», то есть после работы программы `A` указывает на массив из `N` двойных слов, значения которых получены знаковым расширением исходных значений. Считать, что выделенной памяти достаточно для хранения `N` двойных слов.

Замечание: следует расширять элементы «справа налево», так как иначе произойдёт перезапись части оригинального массива.

Задача 1-3

Напишите ассемблерную программу, проверяющую симметричность статического массива (`N` элементов типа `int`, `N` – константа, `2 <= N <` 2^{20}): одинаковые значения имеют первый и последний, второй и предпоследний, и т. д. Выполнять ввод значений элементов массива и выравнивать стек не требуется. Если массив симметричный, на выходе из функции `CMAIN` регистр `EAX` должен иметь значение `0`, в противном случае – `1`. Запрещается использовать строковые инструкции и использовать стек. Программа должна содержать не более `16` инструкций. Учитываются все инструкции, включая `ret`.

Задача 1-4

В заданном массиве `short a[50]` поменять местами максимальный и минимальный элементы.

Задача 1-5

Дан массив из `100` элементов типа `int`. Найти сумму элементов массива, превосходящих значение последнего элемента.

Задача 1-6

Даны два массива из `100` элементов типа `short`. Проверить массивы на равенство и напечатать `1` в случае положительного ответа и `0` в противном случае.

Задача 1-7

Дан массив `char x[200]`, содержащий символы некоторого текста. Сформировать массив `char y[200]`, в начале которого расположены все цифры из массива `x` (в порядке их следования в исходном массиве), затем все оставшиеся символы из массива `y` (в произвольном порядке).

Задача 1-8

Написать программу, которая вводит число типа `unsigned long long` и печатает в порядке убывания все десятичные цифры, входящие в это число.

Задача 1-9

Дан текст, состоящий из строчных латинских букв, оканчивающийся точкой (точка в текст не входит). Напечатать:

а) в алфавитном порядке все строчные латинские буквы, встречающиеся в тексте ровно 1 раз;
б) букву, наиболее часто встречающуюся в тексте. В случае если таких букв несколько, напечатать первую по алфавиту.

Задача 1-10

Дана матрица `int A[N][M]`. Выписать код, загружающий элемент `A[i][j]` на регистр `EDX`.

Задача 1-11

Дана матрица `int A[N][N]`. Требуется напечатать сумму всех нечетных элементов ее главной диагонали.

Задача 1-12

Транспонируйте квадратную матрицу «на месте», не используя дополнительной памяти.

Более сложный вариант задачи: Транспонируйте «на месте», не используя дополнительной памяти, матрицу произвольного размера `M×N`. До и после транспонирования матрица хранится в памяти с разверткой по строкам.

Задача 1-13

Дана матрица размера `N×N` (`N = 50`) из элементов типа `int`.

а) Найти сумму элементов матрицы, лежащих ниже главной диагонали.
б) Поменять местами элементы главной и побочной диагоналей матрицы.
в) Подсчитать количество строк матрицы, элементы которых упорядочены по возрастанию.
г) Напечатать номер строки, содержащей наибольшее количество нулевых элементов.

Задача 1-14

Дан массив из `30` ненулевых указателей на ограниченные нулем строки символов.

а) Подсчитать количество строк, начинающихся с ненулевой цифры.
б) Напечатать строку максимальной длины. Если таких строк несколько, напечатать первую из них по порядку следования в массиве.
в) Подсчитать количество строк, являющихся палиндромами.

Задача 1-15

Дана матрица `int A[N][M]`. Требуется удалить из нее все строки, сумма элементов в которых меньше нуля.

Задача 1-16

Дана матрица, заданная в виде массива указателей `int *A[N]`, где для каждой строки уже выделена и заполнена память под `M` элементов. Требуется удалить из нее все столбцы, сумма элементов в которых меньше нуля.

Задача 1-17

Дана последовательность слов, состоящих из строчных латинских букв. Слова отделены друг от друга одним или несколькими пробелами, после последнего слова записана точка, не входящая в текст. Удалите из текста лишние пробелы.

а) Текст вводится и выводится посимвольно.
б) Текст содержится в массиве `char S[]` и выводится на печать посимвольно.
в) Текст содержится в массиве `char S[]`, а обработанный текст размещается в массиве `char R[]`.
г) Тест содержится в массиве `char S[]` и обрабатывается «на месте».

2. Структуры и объединения

Выравнивание данных в памяти

При размещении в памяти массивов все элементы расположены друг за другом, без промежутков. Структуры объединяют в себе поля произвольных типов, размещаемых в памяти в том же порядке, в каком они были объявлены в исходном Си-коде. Размещение неоднородных (различной длины) полей структуры возможно аналогичным образом, без промежутков, но особенности работы аппаратуры будут проявляться в задержках чтения и записи невыровненных в памяти данных. Современные компиляторы используют байты-заполнители, чтобы поля структур всегда размещались в памяти на адресах, кратных некоторым числам, в зависимости от типов полей.

Правила выравнивания полей зависят от платформы: архитектуры аппаратуры и работающей на ней операционной системы (Таблица 1). Число `N` в ячейке таблицы означает, что поле данного типа расположено на расстоянии кратном `N` от начального (базового) адреса структуры. Для 32-разрядного кода правила выравнивания в типовых компиляторов ОС Windows и Linux различаются только для типа `double`. В случае 64-разрядного кода часть изменений в правилах обусловлена увеличением машинного слова с 4 байтов до 8, вследствие чего меняются размеры указателей и типа `long`.

Таблица 2. Правила выравнивания для полей различных типов.

	`char`	`short`	`int`	`long`	`указатель`	`float`	`double`	`long double`
IA-32/Linux	–	2	4	4	4	4	4	4
IA-32/Windows	–	2	4	4	4	4	8	4
x86_64/Linux	–	2	4	8	8	4	8	8
x86_64/Windows	–	2	4	4	8	4	8	8

Правила выравнивания полей являются частью ABI – двоичного (бинарного) интерфейса приложений. В ABI входят соглашения между программами, библиотеками и операционной системой, обеспечивающих взаимодействие этих компонентов на низком уровне на данной платформе. В частности, это дает возможность объединять в единую работающую программу модули, независимо друг от друга скомпилированные, в том числе различными компиляторами, и даже изначально написанные на различных языках программирования. В разделах 3 и 4 будет рассмотрена другая важная часть ABI – соглашения вызова функций.

Пример 2-1 Размещение полей структуры в памяти

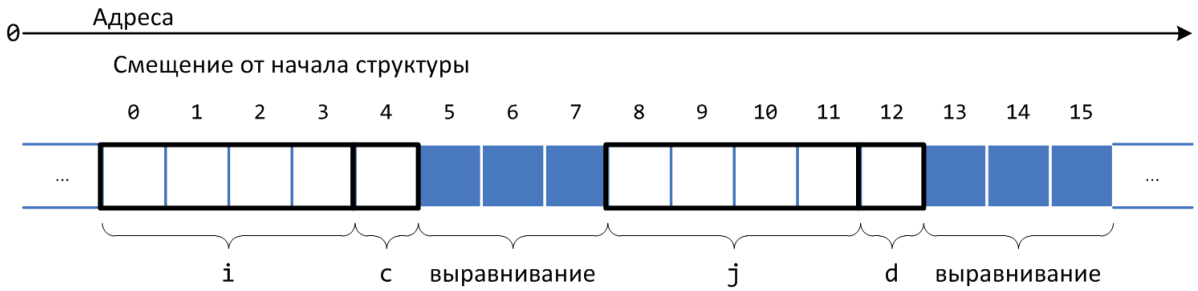
Дан тип данных `P`, описывающий структуру. Компиляция кода выполняется в ОС Linux.

```
typedef struct {
   int i;
   char c;
   int j;
   char d;
} P;
```

Определите размер типа данных `P`. Для каждого поля укажите его смещение относительно начала структуры.

Решение

Приведем рисунок, демонстрирующий размещение полей структуры в памяти.



Первое поле `i` будет размещено в самом начале памяти, выделенной под структуру; смещение от начала у первого поля всегда `0`. Сразу после него будет размещено поле `c`. Поскольку тип `int` занимает 4 байта, смещение, на котором окажется поле `c` – 4; занимает оно один байт, поскольку тип поля `c` – `char`. Следующее поле структуры (поле `j`) имеет тип `int`, его размер 4 байта. Расположено оно будет не сразу после поля `c`, поскольку это нарушает правила выравнивания. Для типа `int` выравнивание составляет 4 байта, поле `j` будет размещено на ближайшем смещении, без остатка делящимся на 4. В данном случае это будет смещение 8. Между полями `c` и `j` размещено 3 «пустых» байта с целью выравнивания. Последнее поле, поле `d`, будет размещено сразу после поля `j`, поскольку его размер один байт (тип `char`), никакие правила выравнивания к однобайтовым полям не применяются. Таким образом, смещение поля `d` составит 12. Однако, после этого (последнего описанного в объявлении) поля будут размещены три байта, их цель – выровнять конец всей структуры по 4-байтной границе. Вся структура `P` будет занимать 4 байта, а смещения ее полей будут следующие: 0, 4, 8, 12.

Пример 2-2 Структуры и объединения

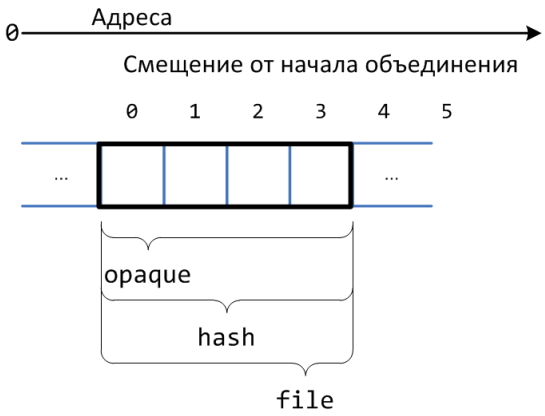
Си-программа, использующая структуру `figure`, была скомпилирована на платформе IA-32/Windows.

```
struct figure {
   int id;
   double scale;
   unsigned char type;
   union {
      void* opaque;
      unsigned char hash[4];
      int file;
   } sourceData;
   struct figure * thumb;
};
```

Определите размер типа данных `figure`. Для каждого поля укажите его смещение относительно начала структуры.

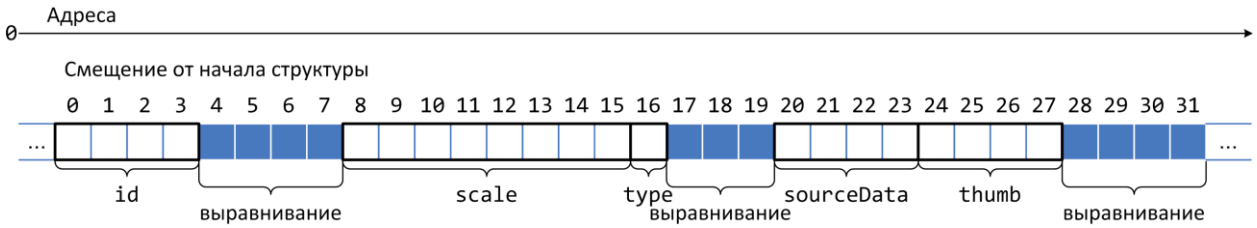
Решение

Одним из полей структуры является представитель другого агрегатного типа – объединения. В свою очередь, в объединении одним из полей является массив. Чтобы определить правила размещения данных придется двигаться «изнутри наружу», поскольку правило выравнивания для агрегатного типа определяется как максимум выравнивания всех его полей. В случае массива используется правило выравнивания типа элемента массива, поэтому поле `unsigned char hash[4]` не создает никаких требований к выравниванию. Вся память, выделенная для объединения `sourceData`, используется полями совместно.



Все три поля занимают одни и те же 4 байта, дополнительных байтов в конце не требуется, т. к. максимальные требования по выравниванию создаются полями `opaque` и `file`.

Для всей структуры карта размещения полей выглядит следующим образом.



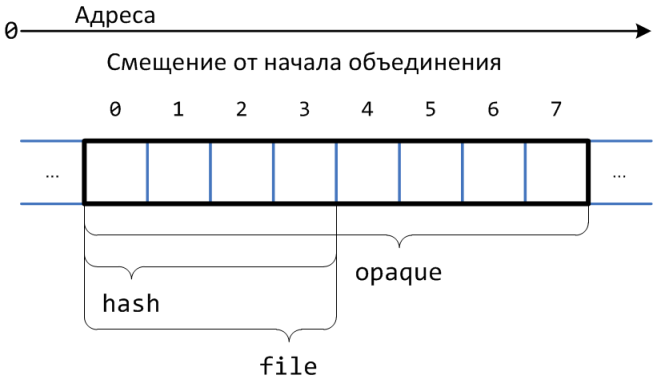
Перед полем `scale` необходим дополнительный промежуток для выравнивания по 8-байтной границе, как того требуют правила компилятора Microsoft. Другие заполняющие байты помещены перед полем `sourceData` (объединение требует выравнивания в 4 байта) и в конце структуры: наличие поля типа `double` требует, чтобы размер структуры был кратен 8.

Пример 2-3 Структуры и объединения в x86_64

Структура `figure` из Примера 2-2 была скомпилирована для 64-х разрядной машины x86_64. Определите ее размер и смещения полей.

Решение

Существенное отличие будет заключаться в том, что размер указателей увеличится с 4 до 8 байтов. В первую очередь изменения затронут объединение `sourceData`.



Поля `hash` и `file` по-прежнему занимают первые 4 байта, но указатель `opaque` требует уже 8 байтов. Все объединение, поле `sourceData`, должно быть выровнено по границе в 8 байтов.

Структура figure в результате увеличится в размере до 40 байтов.

С 3 до 7 байтов будет увеличено заполнение перед полем `sourceData`, чтобы оно размещалось по смещению 24 (кратно 8). Размер поля-указателя `thumb` увеличится

до 8 байтов. Добавлять в конец структуры заполнители не придется, т. к. ее размер сразу же оказывается кратен 8.

Работа с полями

При работе с массивом доступ к элементу требовал вычисления адреса в виде *база + смещение*, где *смещение = масштаб * индекс*. В случае со структурой смещение определятся при построении карты, так, как это было показано в примерах 2-1 – 2-3. Для всех полей объединения смещение нулевое.

Пример 2-4 Чтение и запись полей

Си-программа, использующая структуру `creature`, была скомпилирована на платформе IA-32/Linux.

```
typedef struct t_creature {
   signed char kingdom;
   signed char phylum;
   short       class;
   int         order;
   union {
      int legs;
      signed char wings;
      signed char flippers;
      signed short tentacles;
   } extremity;
   double since;
} creature, *p_creature;
```

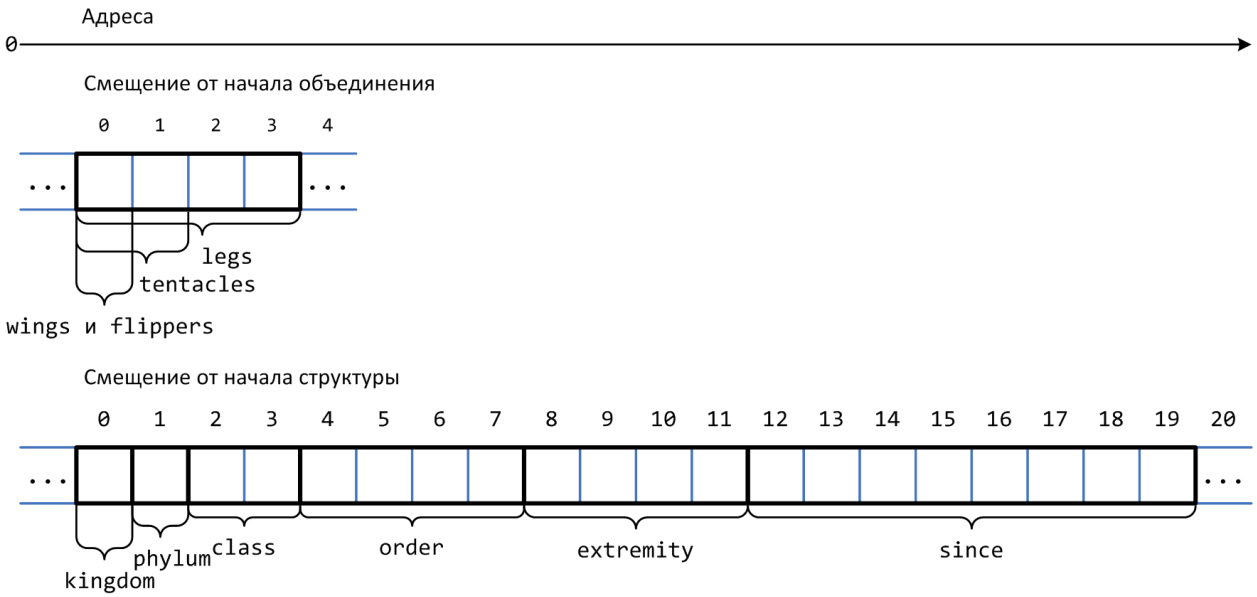
```
static creature grumpyCat;

grumpyCat.extremity.legs = 4;
```

```
static p_creature velociraptor;
static double x;

x = velociraptor->since;
```

Реализуйте приведенный выше Си-код на языке ассемблера.

Решение

В первую очередь определяем размещение полей в структуре `creature`.



Если поля в структуре идут в порядке, когда размер не возрастает, то требования по выравниванию не усиливаются, что гарантированно обеспечивает размещение полей с минимальным числом байтов-заполнителей. Однако в каждом конкретном случае возможности экономии зависят от набора полей. Например, в рассматриваемом примере размер полей, наоборот, не убывает. Тем не менее, промежутки вовсе отсутствуют, и общий размер структуры составляет 20 байтов.

```
section .bss
   grumpyCat    resb 20 ; Выделяем память под всю структуру – 20 байтов
   velocitaptor resd 1  ; Для указателя необходимо 4 байта
   x            resq 1  ; Для double необходимо 8 байтов

section .text
   mov dword [grumpyCat + 8], 4  ; Не забываем использовать спецификатор
                                 ; размера при пересылке констант
   mov eax, dword [velociraptor] ; Сперва загружаем указатель на регистр,
   mov edx, dword [eax+12]       ; после чего прибавляем к нему заранее
   mov dword [x], edx            ; известное смещение
   mov edx, dword [eax+16]
   mov dword [x+4], edx
```

Поле `extremity` расположено по смещению в 8 байтов от начала структуры, все поля объединения `extremity` имеют такое же смещение.

Во втором фрагменте кода происходит копирование 8 байтовых данных, для чего приходится выгружать поле `since` на регистр по частям два раза. Первый раз пересылались младшие байты, располагаемые в структуре по смещению в 12 байтов. У старших четырех байтов это смещение увеличивается на 4 и становится 16.

Использование команд сопроцессора x87 позволит записать второй фрагмент кода более коротко.

```
mov  eax, dword [velociraptor]
fld  qword [eax+12]
fstp qword [x]
```

Битовые поля

Если для целочисленного поля задан битовый размер, способ его размещения стандартом языка Си не специфицирован и зависит от реализации компилятора. Однако в большинстве случаев компиляторы стараются «запаковать» битовые поля, уменьшив общий размер структуры.

Пример 2-5 Взведение и сброс отдельных битов

Дана структура `omg`. Пусть все три битовых поля размещены компилятором в рамках одного 32-разрядного двойного слова, соответствующему типу `int`, в порядке своего объявления в исходном тексте – от младших битов к старшим.

```
struct omg {
       int a: 3;
       int b: 5;
       int c: 2;
};
```

Требуется поместить в поле `b` значение, составленное из битов полей `a` и `c` (младшие биты из поля `a`, старшие биты из поля `c`), в поле `a` – старшие 3 бита из поля `b`, в поле `c` – младшие 2 бита из поля `b`.

Решение

```
section .bss
    omg resd 1

section .text
    mov eax, dword [omg]
    mov edx, eax
    and edx, 0x300  ; Выделяем поле c в edx
    shr edx, 2      ; Сдвигаем биты c на их место в результате
    mov ebx, eax
    shl ebx, 3      ; Сдвигаем биты a на их место в результате (еще
                    ; останутся другие биты)
    and ebx, 0x38   ; Убираем все биты кроме битов a
    or edx, ebx     ; Объединив биты a и c, получим новое
                    ; значение поля b на его месте
    mov ebx, eax
    and ebx, 0xe0   ; Выделяем старшие 3 бита поля b
    shr ebx, 5      ; Помещаем их на место - в поле a
    mov ecx, eax
    and ecx, 0x18   ; Выделяем младшие два бита поля b
    shl ecx, 5      ; Помещаем их на место поля a
    or ecx, ebx     ; Получаем новые значения полей a и c на их местах
    or edx, ecx     ; Получаем итоговый результат
    mov dword [omg], edx ; Сохраняем его в памяти
```

Задачи

Задача 2-1

Пусть для хранения информации об успеваемости студентов определена структура:

```
struct studentInfo {
   char name [25]; // ФИО
   int aver;       // средний балл за сессию
   int marks [5];  // оценки в сессию
};

static struct studentInfo st;
```

Учитывая, что компиляция производится на платформе IA-32/Windows, выписать фрагмент ассемблерного кода для формирования значения поля `st.aver`.

Задача 2-2

Даны следующие объявления на языке Си:

```
typedef struct t_personInfo {
   char name [20]; // ФИО
   char sex;       // пол ('m' для мужчин, 'w' для женщин)
   struct date {
      int day;
      int month;
      int year;
   } db;           // дата рождения
} personInfo;

static personInfo a[100];
static int m;
```

Учитывая, что компиляция производится на платформе IA-32/Windows, выписать фрагмент ассемблерного кода для печати из массива `a` ФИО всех женщин, день рождения которых приходится на месяц `m`.

Задача 2-3

Пусть дата упакована в виде структуры `date` и все битовые поля структуры компилятор располагает в рамках 16-ти разрядного слова в порядке их объявления в исходном тексте от младших разрядов к старшим.

```
struct date {
   int day   : 5;
   int month : 4;
   int year  : 7;
};

static struct date d1, d2;
```

Написать фрагмент ассемблерного кода для решения следующей задачи:

а) в переменную `d2` записать дату, непосредственно следующую за датой, записанной в переменной `d1`;
б) напечатать `1`, если дата `d1` предшествует дате `d2`, и `0` в противном случае;
в) напечатать `1`, если дата `d1` приходится на летний месяц, и `0` в противном случае.

Задача 2-4

Сформулируйте достаточные требования для того, чтобы заданная структура или объединение имели одинаковое расположение полей для всех рассматриваемых платформ: 32- и 64-разрядных Windows и Linux.

Задача 2-5

Пусть даны статические массивы `char *name[N]`, `double height[N]`, `double weight[N]`, где `name[i]`, `height[i]` и `weight[i]` задают имя, рост и вес `i`-го человека. Опишите

структуру `struct person`, которая позволит хранить всю информацию об одном человеке, и реализуйте фрагмент ассемблерного кода, который заполняет массив struct person `persons[N]` по трем заданным массивам `name`, `height` и `weight`.

Задача 2-6

Даны следующие объявления на языке Си:

```
struct swallow {
   int is_african : 1;
   int is_unladen : 1;
   double velocity;
   int id;
};

static struct swallow s[100];
```

Считая, что компилятор под IA-32/Linux располагает битовые поля от младших битов в пределах 32-разрядного двойного слова, напишите на ассемблере фрагмент кода, который любым известным вам способом отсортирует массив `s` по неубыванию значений поля `velocity`.

Задача 2-7

Пусть массив `s` из задачи 2-6 уже отсортирован по возрастанию значений поля `id` и все эти значения различны. По заданному числу `static int myid`, которое гарантированно встречается в массиве, найдите и напечатайте соответствующее значение `is_unladen`:

а) используя линейный поиск;
б) используя бинарный поиск.

Задача 2-8

Компилятор GCC позволяет при помощи атрибута структуры `packed` указать, что выравнивание полей в структуре не производится. В этом случае все поля структуры располагаются подряд. Этот синтаксис не входит в стандарт языка Си. Пусть даны две идентичные структуры `sp` и `su`, отличающиеся только наличием или отсутствием этого атрибута.

```
struct sp {
   char c;
   short s;
   int i;
} __attribute__((packed));

struct sp sp_data;
```

```
struct su {
   char c;
   short s;
   int i;
};

struct su su_data;
```

Осуществите копирование значений из `sp_data` в `su_data` и обратно.

Задача 2-9

Даны следующие определения на языке Си:

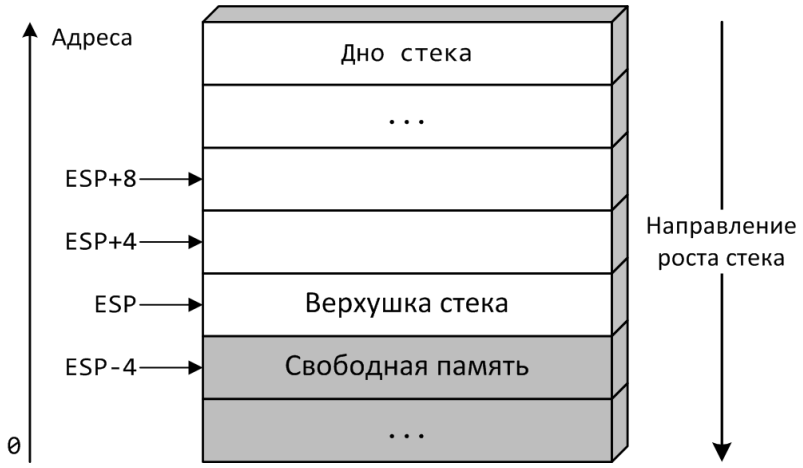
```
enum { RGB, CMYK };

struct color
{
   int type; // RGB или CMYK
   union
   {
      struct { double r, g, b; } rgb;
      struct { double c, m, y, k; } cmyk;
   } u;
};

static struct color clr;
```

Реализуйте фрагмент ассемблерной программы, который печатает `YES`, если в переменной `clr` записан белый цвет (RGB (1, 1, 1) или CMYK (0, 0, 0, 0)), и `NO` иначе.

3. Организация вызова функций

Аппаратный стек

Архитектура IA-32 реализует поддержку аппаратного стека в памяти. Регистр ESP является указателем верхушки стека, команды PUSH и POP выполняют пересылку данных и сдвигают должным образом указатель верхушки стека. Стек растет вниз: при помещении новых данных на стек адрес верхушки уменьшается, при снятии данных – увеличивается. При выполнении 32-разрядного кода естественной единицей данных на стеке является двойное слово. Принято изображать стек в виде «стопки» двойных слов.



В каждый момент времени регистр ESP указывает на верхний элемент стека. Задавая положительное смещение относительно ESP, можно произвольным образом обращаться к содержимому стека. Отрицательные смещения относятся к свободной памяти. После снятия данных со стека содержимое ячеек памяти не меняется, происходит только перемещение указателя стека. Но в рамках логики стека содержимое памяти, лежащей ниже того места, на которое ссылается ESP, не определено.

Пример 3-1 Пересылки без MOV

На стек помещено $N > 1$ целых чисел n_0, n_1, …, n_{N-1}. Последний элемент последовательности – верхушка стека. Переместите n_0 на верхушку стека так, чтобы в результате получилась следующая последовательность: n_1, …, n_{N-1}, n_0. ***Дополнительные ограничения:*** (1) при пересылках данных из памяти или в память разрешается использовать только команды PUSH и POP, (2) не разрешается использовать дополнительную память.

Решение

Будем в цикле сдвигать на один шаг вниз по стеку (и вверх по адресам) элементы последовательности. Элемент i будет извлекаться, а на его место будет помещать-

ся элемент i+1. После того, как будет извлечен последний элемент (n_0), переместим указатель стека на конец последовательности и поместим его на верхушку стека.

```
   mov  ecx, N-1      ; В цикле должно выполниться N-1 записей на стек
   pop  edx           ; В edx помещаем элемент i+1
.l:                   ; Начинаем тело цикла
   pop  eax           ; Выгружаем на eax элемент i
   push edx           ; На его место помещаем элемент i+1
   mov  edx, eax      ; Перемещаем в edx элемент i
   add  esp, 4        ; Принудительно передвигаем верхушку стека
   loop .l            ; Тело цикла выполняется N-1 раз
   sub  esp, 4*(N-1)  ; Возвращаем указатель стека на последний элемент
   push edx           ; Кладем на верхушку стека n0
```

Соглашение вызова cdecl

Любое соглашение вызова функции должно определить, как будут использоваться различными функциями регистры, способ передачи аргументов вызова и возвращаемого значения и т. д. Основным соглашением вызова на платформе IA-32/Linux является cdecl со следующими правилами.

Для вызова функции используется команда `CALL fname`, где операнд `fname` – метка начала вызываемой функции. Для возврата в точку вызова – команда `RET`.

Для передачи аргументов вызова используется стек. Аргументы расположены в «обратном» порядке: в момент вызова на верхушке стека находится первый аргумент, следом за ним – второй и т. д. Для хранения аргументов используются двойные слова. Например, для параметров типа `char`, `short`, `int`, `long` будет использовано 4 байта. Для типа `long long` – 8 байтов.

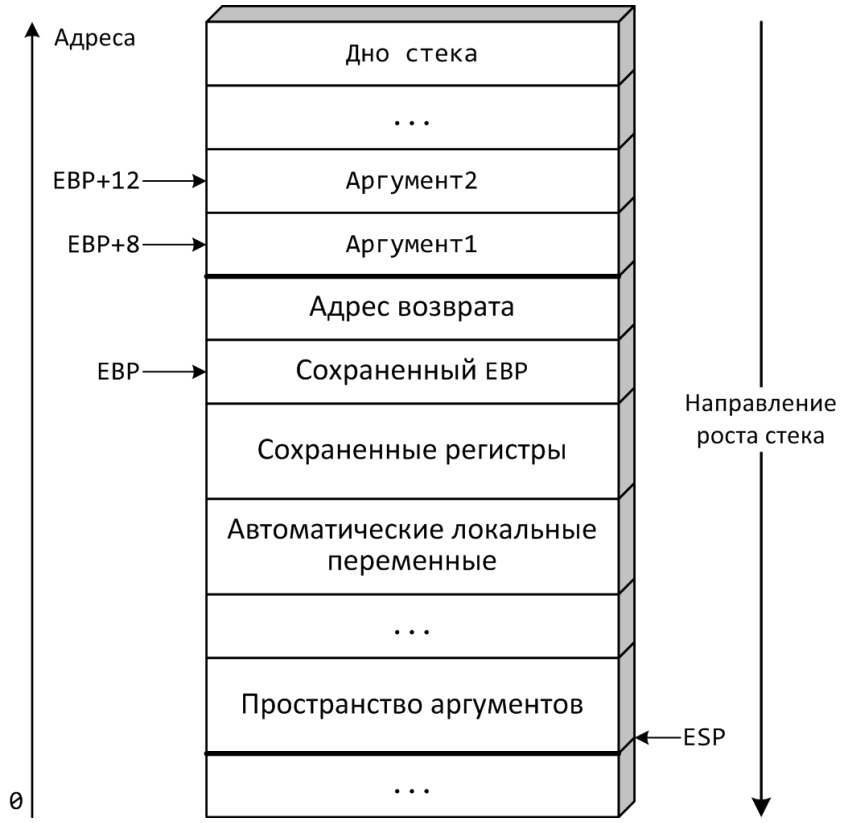
Возвращаемое значение передается в вызвавшую функцию через регистр `EAX`.

Автоматические локальные переменные размещаются на стеке, внутри фрейма вызова функции. Если регистр `ESP` указывает на верхушку стека, что позволяет определить нижнюю границу фрейма, то регистр `EBP` используется для определения верхней границы фрейма.

Регистры `EAX`, `ECX`, `EDX` при необходимости должны сохраняться перед вызовом в вызывающей функции. Регистры `EBX`, `ESI`, `EDI` при необходимости должны сохраняться вызванной функцией. Для хранения используется память во фрейме вызова функции (далее слово «вызов» будет опускаться для краткости изложения). Регистры `ESP` и `EBP` используются для организации работы со стеком.

Код функции можно разделить на три части: пролог, тело функции и эпилог. В прологе и эпилоге происходит создание нового фрейма, сохранение прежнего значе-

ния EBP, в эпилоге освобождение фрейма и восстановление EBP. На рисунке приведено распределение памяти во фрейме, границы фрейма выделены жирными линиями.



Фрейм начинается с адреса возврата, помещенного на стек командой call. Сразу за ним расположен сохраненный регистр EBP, на это место памяти указывает новое значение, помещенное в EBP во время выполнения пролога. Далее в памяти расположены (если есть такая необходимость) сохраняемые в вызванной функции регистры и автоматические локальные переменные. Нижняя часть фрейма отводится для аргументов вызова функций (если из данной функции что-либо вызывается).

Пример 3-2 Скалярное произведение

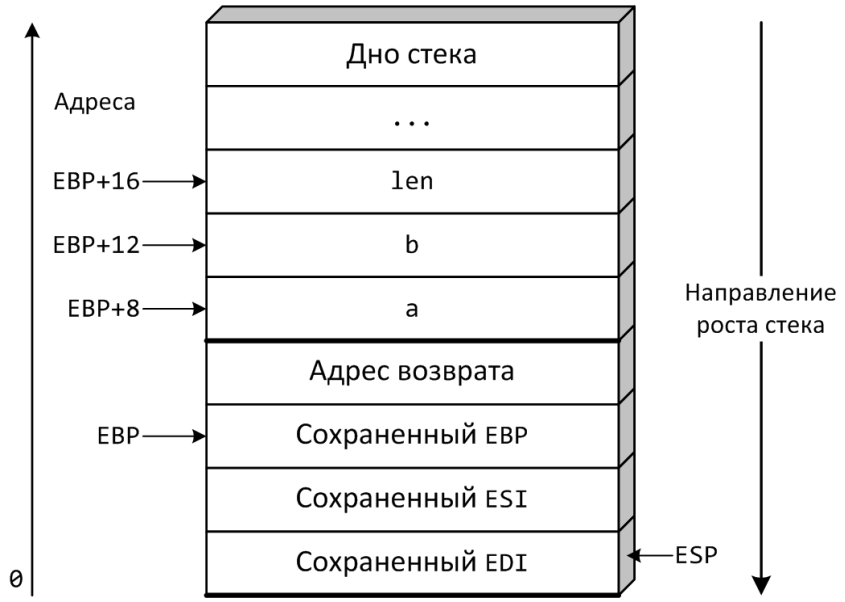
Используя соглашение cdecl, реализуйте функцию conv, вычисляющую скалярное произведение двух массивов длины len.

```
int conv(int *a, int *b, int len);
```

Решение

Приведем рисунок, на котором будет показана структура фрейма функции conv. Над адресом возврата последовательно расположены три параметра. Их значения извлекаются из памяти через задание нужного смещения от регистра EBP. Под сохраненным указателем фрейма EBP расположены ячейки, в которых были сохране-

ны, а затем восстановлены значения регистров ESI и EDI. Указатель стека ESP показан на момент, когда расширенный сохранением регистров пролог закончился, и началось выполнение тела функции.



Само вычисление скалярного произведения организовано следующим образом. В регистре EAX, предварительно обнуленном, накапливается произведение. Для обращения к элементам массива используются регистры ESI, EDI (базовые адреса массивов) и ECX (счетчик цикла). Поскольку при вычислении скалярного произведения неважен порядок обхода элементов массивов, суммируем, начиная с больших индексов.

```
conv:
   push ebp
   mov  ebp, esp            ; Стандартный пролог
   push esi                 ; Сохраняем esi
   push edi                 ; Сохраняем edi
   mov  edi, dword [ebp+8]  ; Извлекаем со стека первый параметр a
   mov  esi, dword [ebp+12] ; Извлекаем со стека второй параметр b
   mov  ecx, dword [ebp+16] ; Извлекаем со стека третий параметр len

   mov  eax, 0              ; В eax будем накапливать скалярное произведение
 .l:
   mov  edx, dword [edi+4*ecx-4]
   imul edx, dword [esi+4*ecx-4]
   add  eax, edx
   loop .l

   pop  edi                 ; Восстанавливаем сохраненный регистр edi
   pop  esi                 ; Восстанавливаем сохраненный регистр esi
   mov  esp, ebp            ; Стандартный эпилог
   pop  ebp
   ret
```

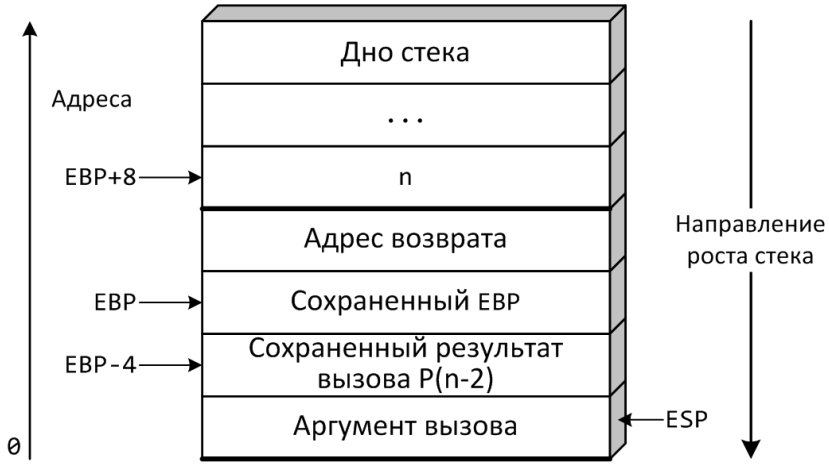
Для организации цикла была использована команда `LOOP`, диапазон изменения регистра `ECX` – от `len` до `1`. Поэтому к базовому адресу массива прибавляется смещение вида `4*ECX-4`, что обеспечивает обращение к индексам `len-1 … 0`.

Пример 3-3 Рекурсивная функция

Используя соглашение cdecl, реализуйте рекурсивную функцию `int P(int n)`, вычисляющую для неотрицательного `n` соответствующий элемент последовательности Падована: $P(0)=P(1)=P(2)=1,\ P(n)=P(n-2)+P(n-3)$.

Решение

Поскольку функция рекурсивная, потребуется формировать фактические аргументы вызова и сохранять во фрейме результат первого вызова, т. к. регистр `EAX` будет «разрушен» во время второго вызова.



Во фрейме должно быть выделено 4 байта для сохранения результатов первого рекурсивного вызова, т. к. по соглашению cdecl регистр `EAX` может быть разрушен во время второго вызова. Помимо того, во фрейме необходимо выделить 4 байта под пространство аргументов (в данном случае для одного аргумента). Первый и второй вызовы будут последовательно использовать это пространство.

```
P:
   push ebp                 ; Стандартный пролог
   mov  ebp, esp
   sub  esp, 8              ; Выделяем во фрейме еще два двойных слова
   mov  eax, dword [ebp+8]  ; Загружаем параметр n в регистр
   cmp  eax, 2              ; Для n большего 2 передаем управление на
   jg   .recur              ; метку .recur
   mov  eax, 1              ; Остальные случаи: P(0)=P(1)=P(2)=1
   jmp  .end                ; Завершаем выполнение
 .recur:
   sub  eax, 2              ; Помещаем в пространство аргументов величину n-2
   mov  dword [esp], eax
   call P                   ; Выполняем первый рекурсивный вызов
   mov  dword [ebp-4], eax  ; Сохраняем результат работы первого вызова
   mov  eax, dword [ebp+8]  ; Помещаем в пространство аргументов величину n-3
   sub  eax, 3
   mov  dword [esp], eax
   call P                   ; Выполняем второй рекурсивный вызов
   add  eax, dword [ebp-4]  ; Складываем P(n-2)+P(n-3)
 .end:
   mov  esp, ebp            ; Стандартный эпилог
   pop  ebp
   ret
```

Пример 3-4 Восстановление прототипа функции

Тело Си-функции `f`, использующей соглашение вызова cdecl, состоит из следующего кода.

```
*b = x;
return c - y;
```

Компилятор сгенерировал для данного фрагмента следующий ассемблерный код.

```
movsx ecx, byte [ebp+8]    ; (1)
mov   eax, dword [ebp+16]  ; (2)
mov   ebx, dword [ebp+20]  ; (3)
mov   edx, dword [ebp+12]  ; (4)
sub   eax, ecx             ; (5)
mov   word [edx], bx       ; (6)
```

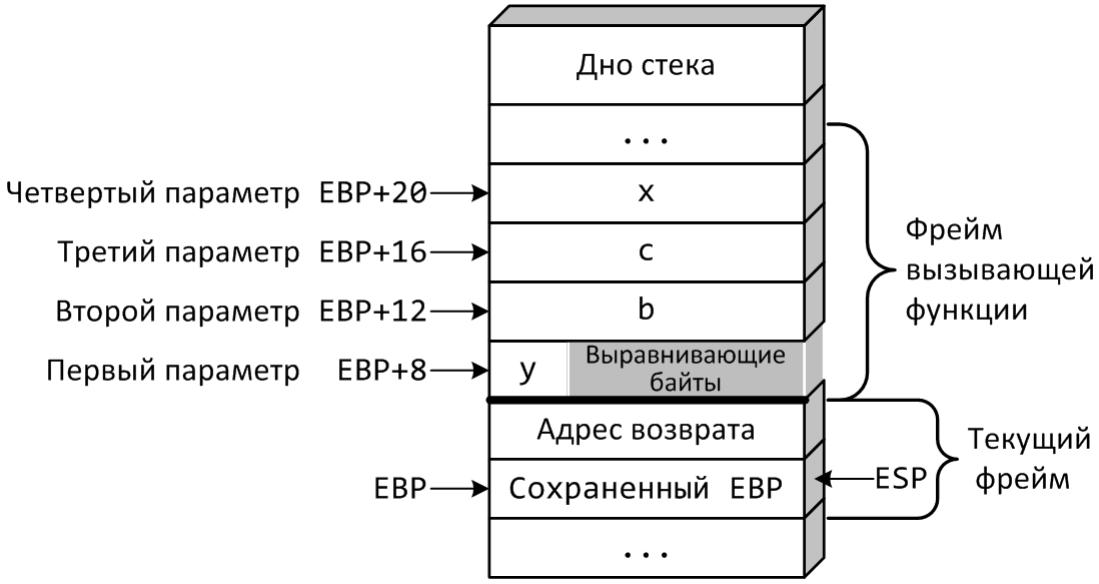
Восстановите прототип функции `f`.

Решение

В приведенном фрагменте Си-кода фигурируют четыре переменные. Поскольку все обращения к памяти используют положительные смещения относительно EBP (+8, +12, …) можно утверждать, что в Си-коде присутствуют только формальные параметры, автоматических локальных переменных не заводилось. Осталось опреде-

лить в каком порядке параметры расположены, какие типы у параметров и возвращаемого функцией значения.

Согласно соглашению, возвращаемое значение передается через регистр EAX. В инструкции 5 происходит вычитание (единственное в приведенном коде): из регистра EAX вычитают регистр ECX, что позволяет соотнести переменные c и y с теми значениями, которые в этих регистрах в данный момент находятся. В инструкциях 1 и 2 происходит загрузка значений на регистры из стека, используя смещения относительно EBP +8 и +16. После стандартного пролога функции два верхних элемента фрейма содержат сохраненное значение регистра EBP и адрес возврата.



Это позволяет определить, что первый параметр был загружен на регистр ECX, третий – на регистр EAX. Совместив это с порядком операндов в инструкции вычитания, устанавливаем, что c – третий параметр, y – первый. Результирующее значение было получено операцией над 32-разрядными числами, но при загрузке первого параметра y было выполнено знаковое расширение с 8 до 32 разрядов. Однозначно можно восстановить тип параметра y – это 8-разрядное знаковое число. Таким образом, первый параметр – `signed char y`. Тип третьего параметра однозначно восстановлен быть не может, вычитание допускает операнды как целых типов, так и указатели. У параметра c допустим тип `int` и любой тип-указатель. Тип возвращаемого значения должен совпадать с типом параметра c.

Оставшиеся инструкции 3, 4 и 6 реализуют присваивание `*b = x;`. Адрес, по которому осуществляется запись, был помещен в инструкции 4 на регистр EDX. Чтение происходило из места расположения второго параметра (EBP+12), это означает, что b – второй параметр. Четвертый параметр (x) был загружен на регистр EBX в инструкции 3, после чего для последующей записи в памяти была использована только младшая половина BX. Это означает, что параметр b указывает на 16 разрядные

данные, которые не могут быть указателем, но могут быть как знаковыми, так и беззнаковыми. Типом переменной `x` не может быть указатель, но может быть любой целочисленный тип разрядностью 16 или 32. Знаковость типов параметров `x` и `b` может быть любой, в том числе и не совпадающей.

Итого, приведем один из допустимых прототипов функции `f`.

```
int f(signed char y, short *b, int c, int x);
```

Пример 3-5 Нарушение соглашения вызова

Функция `h` использует соглашение cdecl. Найти ошибки в реализации, если они есть, и объяснить их, указав, какие именно правила были нарушены.

```
int h(int a, int b) {
   return a % b;
}
```

```
h:
   push ebp
   mov  ebp, esp
   mov  edx, dword [ebp+8]
   mov  eax, edx
   sar  edx, 31
   mov  ebx, dword [ebp+12]
   idiv ebx
   pop  ebp
   ret
```

Решение

По соглашению cdecl возвращаемое функцией значение передается через регистр `EAX`. После выполнения инструкции `IDIV` в регистре `EAX` окажется результат деления, а не взятия остатка. Для исправления данной ошибки необходимо добавить инструкцию копирования регистра `EDX` в `EAX`. Помимо того, в вычислениях используется регистр `EBX`, прежнее значение которого не было сохранено. Следует либо добавить сохранение и восстановление, либо отказаться от использования `EBX`. Например, команда `IDIV` позволяет напрямую обращаться к памяти в своем явном операнде-делителе.

Вызов функций по указателю

Пример 3-6 Просто указатель

В статической переменной `p` хранится указатель на функцию, принимающей в качестве параметров два числа типа `short` и возвращающей `int`. Реализуйте вызов функции по указателю `p` на языке ассемблера, включая объявление переменной `p`. Фактическими аргументами вызова являются числа `0x16` и `0x12`. Для вызова функции используйте соглашение cdecl.

Решение

```
section .bss                ; Выделяем 4 байта под указатель
   p resd 1

section .text
   sub  esp, 8              ; Выделяем на стеке место для двух аргументов
   mov  [esp], word 0x16    ; Кладем на стек первый аргумент
   mov  [esp+4], word 0x12  ; Кладем на стек второй аргумент
   call dword [p]           ; Вызываем функцию по указателю p
```

Пример 3-7 Массив в качестве возвращаемого значения

Реализуйте приведенный Си-код, включая объявление переменной x. Для вызова функции используйте соглашение cdecl. Значение выражения поместите в регистр EAX.

```
static short (*(*x)(int))[1870];
...
(*(*x)(22))[4];
```

Решение

Основной сложностью задания является правильная интерпретация объявления переменной x. Первым рассматривается ключевое слово static, затем имя переменной и в порядке должного приоритета все операции, применяемые к имени переменной. Существуют открытые сервисы* в сети Интернет, позволяющие перевести подобные объявления на естественный язык.

static short (*(***x**)(int))[1870]	x – статическая переменная
static short (*(***x**)(int))[1870]	x – статическая переменная, являющаяся указателем
static short (***(*x)(int)**)[1870]	x – статическая переменная, являющаяся указателем на функцию, у которой один параметр типа int, а возвращаемое значение – указатель
static short **(*(*x)(int))[1870]**	x – статическая переменная, являющаяся указателем на функцию, у которой один параметр типа int, а возвращаемое значение – указатель на массив из 1870 элементов
static short (*(*x)(int))[1870]	x – статическая переменная, являющаяся указателем на функцию, у которой один параметр типа int, а возвращаемое значение – указатель на массив из 1870 элементов типа short

* http://www.cdecl.org

Стандарт Си не позволяет возвращать из функции массив, но можно вернуть указатель на массив. К полученному значению можно сразу же применять операции индексирования и адресную арифметику.

Первое, что следует выполнить – вызов функции по указателю. Для этого необходимо положить на стек фактический аргумент и выполнить команду CALL. В результате в EAX будет помещен начальный адрес массива из 1870 элементов. Останется только обратиться к 4-му элементу (смещение +8) и расширить его до 32 разрядов.

```
section .bss
   x resd 1
section .text
   push  dword 22
   call  dword [x]
   movsx eax, word [eax+8]
```

Пример 3-8 Функциональный тип у возвращаемого значения

Реализуйте приведенный Си-код. Для вызова функции используйте соглашение cdecl. Значение выражения поместите в регистр EAX.

```
static int index, param;
int (*(*f())[256])(int);
...
(*f())[index](param);
```

Решение

Как в предыдущей задаче, начать следует с интерпретации объявления переменной f. То, что переменная – статическая, будет опущено.

`int (*(*f())[256])(int)`	Объявлена функция `f()`
`int (*(*f())[256])(int)`	Объявлена функция `f()`, возвращающая указатель
`int (*(*f())[256])(int)`	Объявлена функция `f()`, возвращающая указатель на массив из `256` элементов
`int (*(*f())[256])(int)`	Объявлена функция `f()`, возвращающая указатель на массив из `256` элементов, тип которых – указатели
`int (*(*f())[256])(int)`	Объявлена функция `f()`, возвращающая указатель на массив из `256` элементов, тип которых – указатели на функции с одним параметром типа `int`
`int (*(*f())[256])(int)`	Объявлена функция `f()`, возвращающая указатель на массив из `256` элементов, тип которых – указатели на функции с одним параметром типа `int` и возвращающие `int`

Таким образом, вычисляющееся выражение представляет собой вызов функции `f`. То, что она вернет – адрес массива, в котором надо выбрать элемент с номером `index`. Выбранный элемент – указатель на функцию, для вызова которой необходимо передать ей параметр `param`. Другими словами, `f()` возвращает таблицу функций, требуется вызвать функцию с заданным индексом и параметром.

```
call f                   ; В eax базовый адрес массива указателей на функции
mov  edx, dword [param] ; Кладем на стек переменную param
mov  dword [esp], edx
mov  edx, dword [index] ; В edx помещаем index
call dword [eax+edx*4]  ; Вызываем функцию из таблицы
```

Использование библиотечных функций

Выполнение правил, заданных в соглашении вызова, позволяет пользоваться не только своим кодом, но и любыми другими функциями. Однако требуется совершить дополнительные действия: (1) объявить внешнюю (не определенную в данном файле) метку кода, на которую будет передавать управление команда `CALL`, (2) выровнять фрейм по 16-байтной границе, (3) при сборке исполняемой программы включить код библиотечной функции.

Объявление внешних библиотечных функций в ассемблерном коде требует знания правил перевода имен функций из языка высокого уровня в имена меток. К счастью, для языка Си и платформы Linux/IA-32 это правило тривиально – имя функции никак не меняется, достаточно задать директиву `extern fname`, где `fname` – имя нужной функции. Для удобства в файле `io.inc` определена команда `CEXTERN fname`, которая должным образом объявляет имя внешней Си-функции `fname`

в таких ОС, как Windows, MacOS, Linux. Помимо того, в `io.inc` уже объявлены некоторые функции стандартной библиотеки языка Си: `printf`, `scanf`, `putchar`, `fgets`, `puts`, `fputs`. В случае включения в исходный текст программы файла `io.inc` самостоятельно объявлять перечисленные функции не требуется.

Цель выравнивания – улучшение производительности программы, поскольку чтение выровненных данных может выполняться быстрее. Компилятор gcc производит выравнивание границ фрейма по кратным 16 адресам, статически распределяя в нем память под переменные, сохраняемые значения и аргументы вызова. Последним (с наибольшим адресом) двойным словом во фрейме является адрес возврата. Помимо того, стандартный пролог сохраняет регистр `EBP`. Таким образом, размер дополнительно выделяемой под фрейм памяти берется как минимальное допустимое число из ряда 8, 24, 40, …

Последняя задача – включение библиотечных функций в исполняемый код. Скрипт сборки учебных программ `build_asm.sh` включает в исполняемую программу стандартную библиотеку языка Си, что позволяет свободно вызывать любые функции этой библиотеки (при соблюдении перечисленных правил). Вопрос включения кода произвольных библиотек в собираемую программу в данном разделе не рассматривается.

Некоторые функции стандартной библиотеки используют параметры типа `FILE*`. Предопределенные переменные этого типа, связанные со стандартным входом и выходом, доступны через вызов Си-функций

```
FILE *get_stdin(void);
FILE *get_stdout(void);
```

Объявление и подключение кода этих функций к исполняемой программе обеспечивается скриптом `build_asm.sh` и файлом `io.inc`.

Пример 3-9 Динамическая память и переменное число параметров

Реализуйте на языке ассемблера функцию, перемещающую целые числа, свои параметры, в динамическую память.

```
int* dynamo(int n, ...);
```

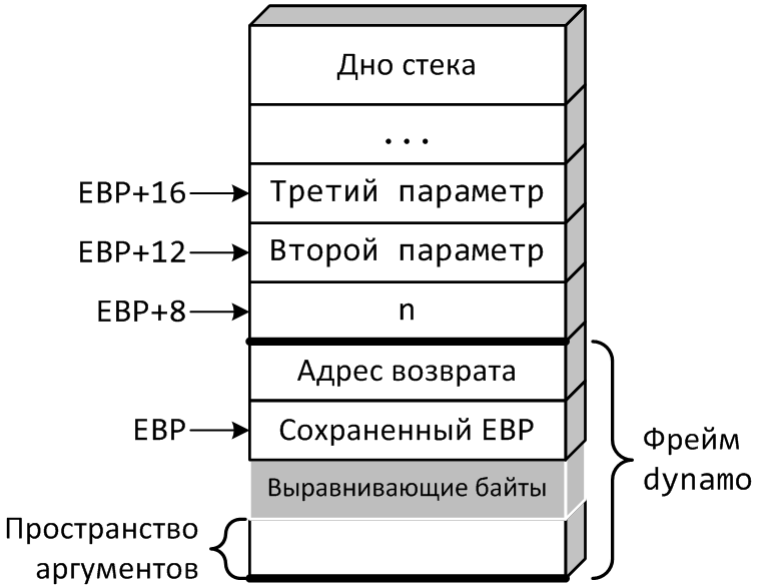
У функции `dynamo` переменное число параметров. Первый параметр – число идущих следом целых чисел. Для этих чисел выделяется динамическая память, куда они копируются. Возвращаемое значение – адрес выделенной динамической памяти.

Если параметр `n` равен `0` возвращается `NULL`. Считать, что при вызове функции `dynamo` стек уже выровнен на границе в 16 байтов.

Решение

Первый параметр вызванной функции размещен по адресу `EBP+8`. Адреса следующих параметров будут идти следом за ним с шагом в 4 байта: `EBP+8+4*(i-1)`, где `i` – номер параметра, при нумерации параметров функции начиная с `1`.

Особенностью кода будет то, что вызов функции `malloc` необходимо будет выполнять на выровненном фрейме. Для остальных действий будет достаточно трех регистров, поэтому закажем на стеке 8 байтов, чтобы суммарный объем стека составил 16 байтов. Из этих 8 байтов 4 байта на верхушке стека потребуются для вызова функции `malloc`, другие 4 байта в вычислениях не принимают участия и используются исключительно для выравнивания стека. Границы выровненного 16-байтного фрейма показаны на рисунке жирными линиями.



```
section .text
dynamo:
   push    ebp                 ; Пролог
   mov     ebp, esp
   mov     eax, dword [ebp+8] ; Проверяем не нулевое ли количество элементов
   test    eax, eax            ; было передано
   jz      .end
   sub     esp, 8              ; Выделяем 8 байт во фрейме
   sal     eax, 2              ; Переводим элементы в байты
   mov     dword [esp], eax    ; Кладем нужное число байт на стек
   call    malloc              ; и вызываем malloc. Теперь eax - адрес буфера
   mov     ecx, dword [ebp+8] ; Снова берем число элементов. Оно не нулевое.
   lea     esp, [ebp+12]       ; Временно переставляем указатель стека на
                               ; последовательность чисел-параметров
   xor     edx, edx            ;
.copy:                         ;
   pop     dword [eax+4*edx]  ; n раз выталкиваем в буфер содержимое со стека
   inc     edx                 ; Явно сдвигать надо только edx
   loop    .copy
   mov     esp, ebp            ; Возвращаем указатель стека на место
 .end:
   pop     ebp                 ; Эпилог
   ret
```

Замечание. Показанный прием следует рассматривать как *«hack»* в изначальном смысле этого термина, но более уместен здесь термин *«kludge»*: проблема решается с нарушением правил, но, тем не менее, код работает и, возможно, имеет какие-то преимущества, например, занимает меньше места, быстрее выполняется. В данном случае особенностью является то, что не используются регистры, требующие сохранения (`EBX`, `ESI`, `EDI`) и тело цикла занимает всего 6 байтов: `8f 04 90 42 e2 fa`. Важно понимать, что приведенный код нарушает принятый порядок работы с регистром `ESP`. Перестановка указателя стека на произвольные пользовательские данные может вызывать проблемы при работе отладчика. В случае возникновения ошибки нет никакой гарантии, что отладчик будет давать корректную диагностику.

Пример 3-10 Стандартный ввод/вывод своими силами

Реализовать на языке ассемблера заданную Си-функцию `f`.

```
void f(void) {
   int x;
   scanf("%d", &x);
   printf("%d @ %p\n", x, &x);
}
```

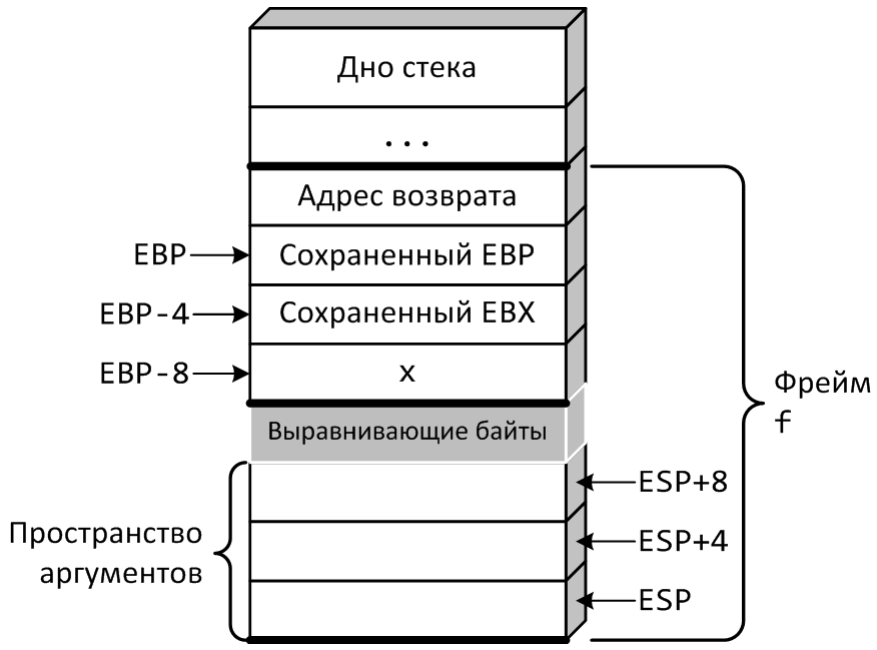
Считать, что при вызове функции `f` стек уже выровнен на границе в 16 байтов. Пользоваться командами ввода/вывода из библиотеки `io.inc` запрещено.

Решение

Основная сложность данной задачи заключается в правильном распределении памяти во фрейме функции `f`. Это необходимо для того, чтоб можно было корректно вызвать библиотечные функции `scanf` и `printf` (их использование обусловлено ограничением, заданном в условии). Фрейм функции `f` должен содержать автоматическую локальную переменную `x` (4 байта), сохраненное значение регистра `EBP` (4 байта), фактические аргументы вызываемых функций, а также адрес возврата (4 байта). Помимо того, значение `&x` используется в Си-функции дважды, вычислим и разместим его в регистре `EBX` (значение должно «пережить» вызов `scanf`), а для сохранения прежнего значения регистра выделим еще 4 байта.

Пространство аргументов будет совместно использоваться функциями `scanf` и `printf`. При вызове первой потребуется два двойных слова: адрес форматной строки и адрес переменной, в которой будет размещено введенное число. Для вызова `printf` потребуется три двойных слова: адрес форматной строки, значение `x` и адрес, по которому эта переменная размещена. Таким образом, под пространство аргументов потребуется 12 байтов.

Суммарный объем размещенных во фрейме данных составит 28 байтов, что не кратно 16. Необходимо дополнительно выделить 4 байта для выравнивания, тогда суммарный объем фрейма достигнет 32 байтов, что уже кратно 16. Компилятор GCC традиционно размещает выравнивающие байты непосредственно выше пространства аргументов. Границы блоков памяти, выровненных по 16 байтов, показаны жирной чертой.



```
section .rodata
   .LC0 db "%d", 0        ; Форматная строка для scanf
   .LC1 db `%d @ %p\n`, 0 ; Форматная строка для printf

section .text
f:
   push    ebp
   mov     ebp, esp
   push    ebx
   sub     esp, 20
   lea     ebx, [ebp-8]
   mov     dword [esp+4], ebx
   mov     dword [esp], .LC0
   call    scanf
   mov     dword [esp+8], ebx
   mov     eax, dword [ebp-8]
   mov     dword [esp+4], eax
   mov     dword [esp], .LC1
   call    printf
   mov     ebx, dword [ebp-4]
   leave
   ret
```

Пример 3-11 Дамп файла

Реализуйте на языке ассемблера функцию `hdump`, которая печатает на стандартный вывод содержимое файла `fname` в шестнадцатеричном виде. Печатать следует по одному байту, разделяя их пробелами, не более 16 байтов в одной строке.

```
void hdump(const char * fname);
```

Считать, что при вызове функции `hdump` стек уже выровнен на границе в 16 байтов. Пользоваться командами ввода/вывода из библиотеки `io.inc` запрещено.

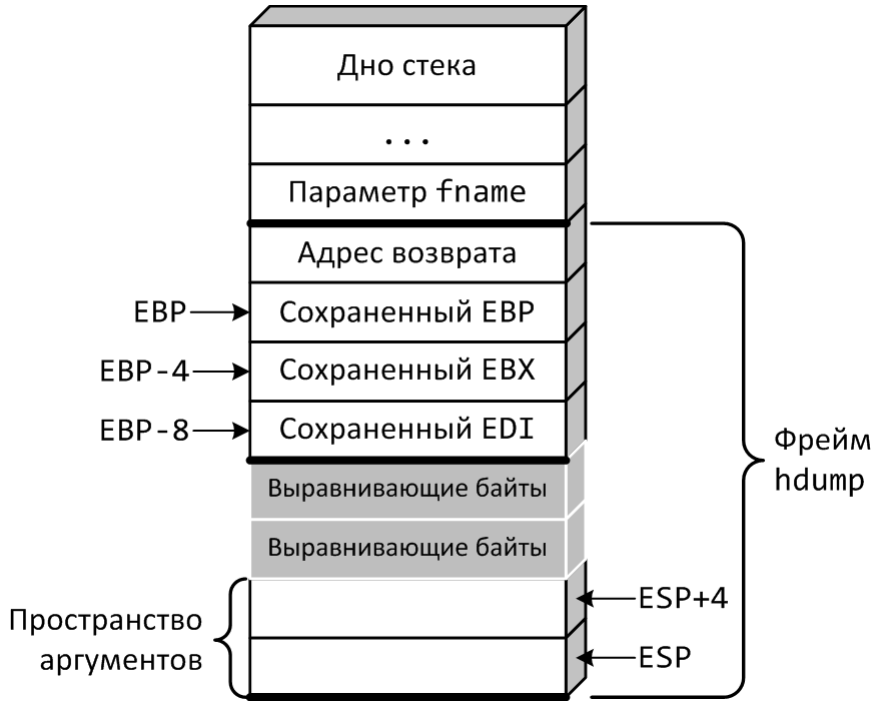
Решение

В функции `hdump` будут вызываться `fopen`, `getc`, `printf`, `fclose`. `printf` объявлять не требуется, объявление остальных функций обязательно. Для работы потребуются три переменные: описатель открытого файла, значение текущего байта, номер текущей позиции в строке. Значение байта будет передаваться из `getc` в `printf`, сохранение во фрейме не требуется. В свою очередь описатель и номер позиции необходимо сохранять между вызовами функций, для этих переменных будут выделены регистры `ebx` и `edi` соответственно.

```
%include 'io.inc'

CEXTERN fopen
CEXTERN fclose
CEXTERN getc

section .rodata
   LC0 db "rb", 0      ; параметр fopen – открыть двоичный файл на чтение
   LC1 db "%02x ", 0   ; параметр printf для печати считанного байта
   LC2 db `\n`, 0      ; параметр printf для перевода строки
```

Для работы библиотечных функций потребуются три строки, которые заданы в секции `.rodata`.

В функции `hdump` присутствуют типовые пролог и эпилог. По смещениям `EBP-4` и `EBP-8` сохраняются регистры `EBX` и `EDI` соответственно.



```
hdump:
   push ebp                   ; Пролог
   mov  ebp, esp
   sub  esp, 24
   mov  dword [ebp-4], ebx
   mov  dword [ebp-8], edi

   mov  eax, dword [ebp+8]    ; Открываем файл fname на чтение
   mov  dword [esp], eax
   mov  dword [esp+4], LC0
   call fopen
   cmp  eax, 0                ; В случае неудачи – сразу выходим
   je   .end
   mov  ebx, eax              ; Сохраняем описатель файла в ebx

   mov  edi, 0                ; Обнуляем номер позиции на строке

 .loop:
   mov  dword [esp], ebx      ; Считываем один байт
   call getc
   cmp  eax, -1               ; -1 – признак конца файла
   je   .le

   inc  edi                   ; Увеличиваем счетчик позиции на строке
   mov  dword [esp], LC1      ; Печатаем байт. Обязательно выводим два символа
   mov  dword [esp+4], eax
   call printf

   and  edi, 0xf              ; Если достигли 16 позиции – сбрасываем счетчик
   jnz  .currLine             ; в ноль и печатаем функцией printf перевод
   mov  dword [esp], LC2      ; строки
   call printf
 .currLine:
   jmp  .loop                 ; Переходим на считывание очередного байта
  .le:

   test edi, edi              ; Если номер позиции не ноль – надо перевести
   jz   .nl                   ; не до конца оконченную строку
   mov  dword [esp], LC2
   call printf
 .nl:

   mov  dword [esp], ebx      ; Не забываем закрыть файл
   call fclose

 .end:
   mov  edi, dword [ebp-8]
   mov  ebx, dword [ebp-4]
   leave
   ret
```

Устройство фрейма функции

Фрейм содержит как пользовательские, так и служебные данные. Помимо автоматических локальных переменных в нем сохраняются регистры и размещен адрес возврата, переход по которому происходит при завершении функции. Язык Си не предусматривает контроля границ памяти, выделенной переменным. Ошибки в

индексации массивов и адресной арифметике могут привести не только к порче других переменных пользователя, но и, что гораздо хуже, к порче служебных данных. При определенных условиях ошибка работы с памятью становится уязвимостью: специально подобранные входные данные (их принято называть эксплойтом) вызывают срабатывание ошибки в таком виде, что поведение программы принципиально меняется. Например, до того, как программа аварийно завершится, происходит целенаправленная порча определенных данных или отправка данных на вывод. Наихудшая ситуация – когда программа в результате срабатывания уязвимости начинает выполнять заданный извне код.

Задачей разработчика является не только обеспечение высокой производительности программы, но и ее безопасность – минимальное число ошибок. В тех случаях, когда происходит порча данных в памяти предпочтительно аварийно завершать выполнение как только всплывает факт срабатывания ошибки. Определенную помощь разработчику оказывает современный компилятор, встраивающий в программу проверки в наиболее опасных с точки зрения безопасности местах.

Пример 3-12 Карта фрейма

Для данной Си-функции компилятор построил следующий ассемблерный код.

```
#include <stdio.h>

int f(int i) {
   int array[4];
   scanf("%x", (unsigned*)&array[i]);
   return array[i];
}
```

```
section .rodata
   LC0 db "%x", 0

section .text
f:
   push ebp                        ; (1)
   mov  ebp, esp                   ; (2)
   push ebx                        ; (3)
   sub  esp, 36                    ; (4)
   mov  ebx, dword [ebp+8]         ; (5)
   mov  dword [esp], LC0           ; (6)
   lea  eax, [ebp-24+ebx*4]        ; (7)
   mov  dword [esp+4], eax         ; (8)
   call scanf                      ; (9)
   mov  eax, dword [ebp-24+ebx*4] ; (10)
   add  esp, 36                    ; (11)
   pop  ebx                        ; (12)
   pop  ebp                        ; (13)
   ret                             ; (14)
```

В момент вызова функции (перед выполнением ассемблерной инструкции №1) регистры имели следующие значения: `ESP = 0xbfffff1c`, `EBP = 0xbfffff50`.

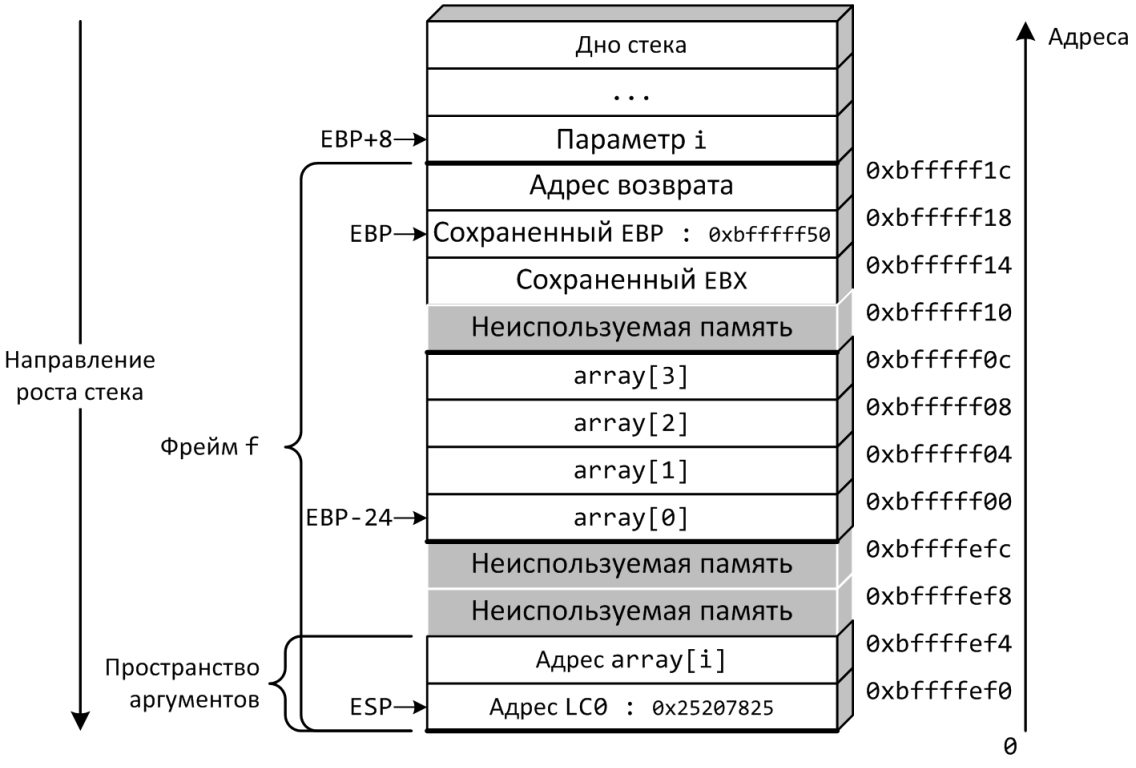
При вызове функции `scanf` со стандартного потока ввода было считано число `0x23`.

Строка `LC0` размещена в памяти по адресу `0x25207825`.

1. Какое значение регистр `EBP` получит после выполнения инструкции №2?
2. Какое значение регистр `ESP` получит после выполнения инструкции №4?
3. По какому адресу размещена переменная `array`?
4. Нарисуйте карту фрейма, указав на рисунке: границы фрейма, аргументы вызова, сохраненные регистры, автоматические локальные переменные, пространство аргументов, неиспользуемую память. Укажите содержимое ячеек памяти, когда их значение гарантированно известно.

Решение

Начнем решение с того, что определим, как происходило распределение памяти во фрейме функции.



Границы выровненных блоков показаны на рисунке жирными линиями. Командами `PUSH` были сохранены регистры `EBP` и `EBX`, после чего на стеке было выделено 9 двойных слов. Общий размер фрейма – 48 байтов, два нижних двойных слова – пространство аргументов для вызова функции `scanf`. Как видно по командам №7 и №10, массив array размещен во фрейме по адресу `EBP-24` и занимает целиком выровненный блок из 16 байтов.

Однозначно можно утверждать, что сохраненный EBP и первый аргумент вызова scanf (адрес форматной строки) будут иметь значения 0xbfffff50 и 0x25207825 соответственно. Куда именно будет помещено введенное число, и, какое значение у второго аргумента scanf, зависит от параметра i и не может быть определено только по заданному фрагменту кода.

Значения регистров в указанные в условиях моменты времени приведены в таблице. После выполнения второй команды EBP будет указывать на место, где сохранено его прежнее значение. После вычитания ESP переместится на нижнее двойное слово занятое фреймом функции (верхушка стека).

Таблица 3. Значения регистров и адрес размещения переменной.

	Значение
Регистр EBP после команды №2	0xbfffff18
Регистр ESP после команды №4	0xbffffef0
Адрес размещения array	0xbfffff00

Пример 3-13 Эксплойт

Приведенная функция f и ее ассемблерная реализация содержат ошибки, которые порождают эксплуатируемую уязвимость.

Пусть в момент вызова функции f адрес возврата был размещен в стеке по адресу 0xbffdb3cc. Постройте такой эксплойт, чтобы в результате его обработки функция f возвращала управление на адрес 0xdeadbeef.

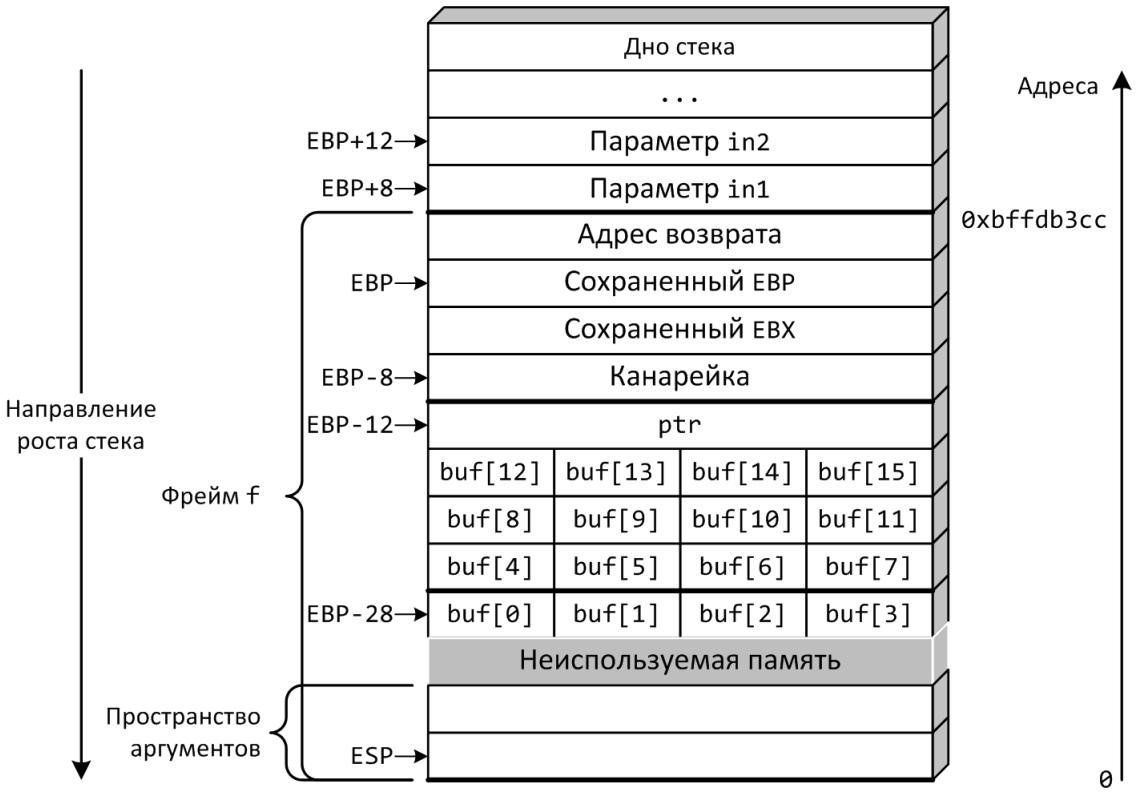
```
#include <string.h>

void f(char *in1, char *in2) {
   char *ptr;
   char buf[16];
   strcpy(buf, in1);
   strcpy(ptr, in2);
}
```

```
f:
   push    ebp                 ; (1)
   mov     ebp, esp            ; (2)
   push    ebx                 ; (3)
   sub     esp, 36             ; (4)
   mov     ebx, dword [ebp+12] ; (5)
   mov     eax, dword [gs:20]  ; (6)
   mov     dword [ebp-8], eax  ; (7)
   xor     eax, eax            ; (8)
   mov     eax, dword [ebp+8]  ; (9)
   mov     dword [esp+4], eax  ; (10)
   lea     eax, [ebp-28]       ; (11)
   mov     dword [esp], eax    ; (12)
   call    strcpy              ; (13)
   mov     dword [esp+4], ebx  ; (14)
   lea     eax, [ebp-12]       ; (15)
   mov     dword [esp], eax    ; (16)
   call    strcpy              ; (17)
   mov     eax, dword [ebp-8]  ; (18)
   xor     eax, dword [gs:20]  ; (19)
   jne     .L5                 ; (20)
   add     esp, 36             ; (21)
   pop     ebx                 ; (22)
   pop     ebp                 ; (23)
   ret                         ; (24)
.L5:
   call    __stack_chk_fail    ; (25)
```

Решение

Как и в предыдущей задаче, определим, как происходило распределение памяти во фрейме функции. Помимо всего, компилятор поместил во фрейм функции «канарейку» – контрольное значение, которое будет затираться при переполнении массива buf (Команды №6-8). Значение «канарейки» проверяется перед выходом из функции (Команды №18-20). Если «канарейка» изменилась – управление передается на функцию, выполняющую аварийное завершение работы программы.

«Канарейка» защищает от изменения адрес возврата, поскольку массив `buf` расположен ниже (`ebp-28`). Но указатель `ptr` расположен также ниже «канарейки» и его значение может быть изменено в результате переполнения. Для этого необходимо задать первый параметр в виде строки из 16 произвольных ненулевых байтов (в приведенном решении они выделены курсивом), а затем – адрес `0xbffdb3cc`, по которому размещен адрес возврата. Байты адреса, как и любые другие целые числа, следует развернуть в обратном порядке (начиная с младших). Важно, чтобы сразу после адреса строка заканчивалась, т. к. в противном случае будет изменено значение «канарейки».



Второй вызов функции `strcpy` перепишет адрес возврата; в параметре `in2` должен быть задан адрес `0xdeadbeef`, на который требуется передать управление. Поскольку других действий предпринимать не требуется, параметр `in2` можно этим ограничить.

in1	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	*ff*	**cc**	**b3**	**fd**	**bf**	**00**
in2	**ef**	**be**	**ad**	**de**	**00**																

Задачи

Задача 3-1

В стеке записаны 20 указателей типа `char*` (последний из указателей находится на вершине стека). Не меняя состояние стека, обнулить значения по данным указателям.

Задача 3-2

В стеке записано 50 чисел типа `int` (последнее число находится на вершине стека).

а) Среди этих чисел подсчитать количество чисел, имеющих равных «соседей» (предыдущее и последующее числа), и записать ответ в регистр `EAX`.

б) Считая, что числа заносились в стек в порядке `x1`, `x2`, …, `x50`, записать в регистр `EAX` число `1`, если последовательность `x1`, `x2`, …, `x50` является возрастающей, или `0` в противном случае.

Состояние стека менять не разрешается.

Задача 3-3 Максимум двух чисел

Требуется реализовать функцию max, соответствующую прототипу `int max(int a, int b)`, которая возвращает максимальное значение своих параметров.

Задача 3-4 Поменять местами

Требуется оформить функцию `swap`, соответствующую прототипу `void swap(int *a, int *b)`, которая меняет значения своих аргументов.

Задача 3-5 Восстановить объявление функции

Тело Си-функции `f` имеет следующий вид:

```
*p = d;
return x-a;
```

Этому телу соответствует следующий ассемблерный код.

```
movsx edx, byte [ebp + 12]
mov   eax, dword [ebp + 16]
mov   dword [eax], edx
movsx eax, byte [ebp + 8]
mov   edx, dword [ebp + 20]
sub   edx, eax
mov   eax, edx
```

Требуется восстановить прототип функции `f`.

Задача 3-6

Реализовать функцию с прототипом `int f(unsigned n)` для решения следующей задачи:

а) функция возвращает значение `1`, если количество единиц в двоичном представлении числа `n` превышает количество нулей;
б) функция возвращает значение `1`, если в двоичном представлении числа `n` имеется 5 стоящих подряд единиц.

Реализация должна удовлетворять соглашению cdecl.

Задача 3-7

Описать функцию `int max(short *a, unsigned n)`, которая возвращает в качестве результата значение максимального элемента массива `a` из `n` элементов. Функция должна удовлетворять соглашению cdecl.

Задача 3-8 Факториал

Реализовать в виде функции вычисление факториала.

Задача 3-9 Только рекурсия

Описать рекурсивную функцию для решения следующих задач. Функция должна удовлетворять соглашению cdecl:

1. На входном потоке задан текст в виде последовательности символов, заканчивающейся точкой.
   а) Вывести символы текста в обратном порядке.
   б) Подсчитать количество вхождений цифр в указанный текст.
2. На входном потоке задана последовательность ненулевых чисел типа `int` (признаком окончания последовательности является число `0`). Вывести сначала все отрицательные числа этой последовательности, а затем все остальные (в любом порядке).
3. Для числа `unsigned n` найти:
   а) Сумму цифр в его десятичной записи.
   б) Максимальное значение цифры в его десятичной записи.

Задача 3-10 Выравнивание стека

Требуется должным образом оформить функцию `CMAIN` для последующего вызова функций стандартной библиотеки. Необходимо выровнять стек по границе 16 байтов.

Задача 3-11 «Сколько время»

Требуется напечатать на стандартный вывод текущее время. Пример вывода:

```
Tue Mar 29 23:00:33 2011
```

Указание: воспользуйтесь функциями стандартной библиотеки языка Си:

```
#include <time.h>
time_t time(time_t *timer);
char *ctime(const time_t *timer);
```

Задача 3-12

Реализовать функцию стандартной библиотеки языка Си `memcpy` в соответствии с соглашением cdecl.

```
void *memcpy(void *destination, const void *source, size_t num);
```

Функция `memcpy` не допускает пересечение между областями памяти, указываемыми в `source` и `dest`. ***Примечание:*** Тип `size_t` — 32-битное беззнаковое целое.

Задача 3-13

Реализовать функцию стандартной библиотеки языка Си `memmove` в соответствии с соглашением cdecl.

```
void *memmove(void *destination, const void *source, size_t num);
```

Функция `memmove` допускает пересечение между областями памяти, указываемыми в `source` и `dest`.

Задача 3-14

Написать полную программу на языке ассемблера, которая вводит с клавиатуры два целых числа (`int`) и выводит:

а) максимальное из этих чисел;
б) среднее арифметическое этих чисел.

Для ввода/вывода использовать функции стандартной библиотеки языка Си: `scanf` и `printf`. Для расположения вводимых чисел использовать стек. Обеспечить выравнивание стека в функции `main`.

Задача 3-15

Написать полную программу на языке ассемблера, которая вводит с клавиатуры целое число (`int`), вычисляет его абсолютное значение и выводит его. Программа должна использовать функции стандартной библиотеки языка Си: `scanf`, `abs` и `printf`. Вводимое число расположить в стеке. Обеспечить выравнивание стека в функции `main`. Прототип функции `abs` имеет вид `int abs(int n);`

Задача 3-16

Реализовать функцию `strpfx`, которая возвращает длину наибольшего общего префикса строк `s1` и `s2`, в соответствии с соглашением cdecl. Строки:

а) не могут полностью совпадать;
б) могут полностью совпадать.

```
size_t strpfx(const char *s1, const char *s2);
```

Примечание: данная функция не входит в стандартную библиотеку языка Си.

Задача 3-17

При компьютерной обработке звука используется дискретное представление звукового сигнала: звук рассматривается как массив дискретных сэмплов (моментальных значений амплитуды), снятых с источника с определённой частотой (частота дискретизации). Эти значения могут храниться, например, в виде знаковых 8-разрядных целых чисел.

Таким образом, монофонический звук может быть представлен массивом `signed char *M`, где `M[i]` — значение амплитуды в момент времени `i*T`, где `T` — период дискретизации.

Стереофонический звук, как правило, представляется в «переплетённом» (interleaved) формате: в массиве `signed char *S` чётные элементы соответствуют амплитуде в левом канале, а нечётные — в правом. Иными словами, для одного и того же стереофонического звука, представленного в таком формате и в виде двух монофонических массивов `L` и `R` верны соотношения:

```
S[0] == L[0];
S[1] == R[0];
S[2] == L[1];
S[3] == R[1];
...
S[i] == (i % 2 ? R : L)[i / 2];
```

Требуется реализовать функции преобразования:

а) двух монофонических сигналов для левого и правого уха `L` и `R` в interleaved stereo `S`;
б) interleaved stereo сигнала `S` в два монофонических `L` и `R`.

```
/* Преобразование из L и R в S.  Число сэмплов передаётся в n.  */
void
interleave(size_t n, const signed char *L, const signed char *R,
   signed char *S);

/* Преобразование из S в L и R.  Число сэмплов передаётся в n.  */
void
deinterleave(size_t n, const signed char *S, signed char *L,
   signed char *R);
```

Задача 3-18

С использованием функции стандартной библиотеки языка Си `qsort` реализуйте:

а) функцию с прототипом `void sort1(int *x, int n)`, сортирующую по неубыванию массив `x`, в котором содержатся `n` целочисленных значений;

б) функцию с прототипом `void sort2(unsigned long long *x, int n)`, сортирующую по невозрастанию массив `x`, в котором содержатся `n` целочисленных значений;

в) функцию с прототипом `void sort3(char **x, int n)`, сортирующую в лексикографическом порядке массив строк `x` длины `n` с использованием библиотечной функции `strcmp`.

Задача 3-19

Измените решение Примера 3-9 таким образом, чтобы в теле цикла, выполняющего копирование данных, вместо команды `POP` использовалась команда `PUSH`.

Задача 3-20

Функция `g` и ее ассемблерная реализация содержат ошибки, которые порождают эксплуатируемую уязвимость.

Пусть в момент вызова функции `g` адрес возврата был размещен в стеке по адресу `0xbfc1d80c`. Постройте такой эксплойт, чтобы в результате его обработки функция `g` возвращала управление на адрес `0xdabbad00`.

```
int g(int i, int val, int dest) {
   int *lp;
   int buf[3];

   buf[i] = val;
   *lp = dest;

   return *lp + buf[i];
}
```

```
g:
   push    ebp
   mov     ebp, esp
   sub     esp, 16
   mov     eax, dword [ebp+8]
   mov     edx, dword [ebp+12]
   mov     dword [ebp-12+eax*4], edx
   mov     eax, dword [ebp-16]
   mov     edx, dword [ebp+16]
   mov     dword [eax], edx
   mov     eax, dword [ebp-16]
   mov     edx, dword [eax]
   mov     eax, dword [ebp+8]
   mov     eax, dword [ebp-12+eax*4]
   lea     eax, [edx+eax]
   leave
   ret
```

4. Различные соглашения вызова функций

Оптимизация вызова функций

Деление программы на функции упрощает разработку (и делает в принципе возможным разрабатывать большие программы), но вносит накладные расходы на организацию вызова функций. Компилятор по возможности оптимизирует эти расходы, как с помощью классических оптимизаций кода, так и применением более эффективных соглашений вызова.

После того, как была завершена отладка программы и она собирается с включенной оптимизацией, появляется возможность отказаться от использования указателя фрейма (“omit frame pointer”): все обращения к содержимому стека выполняются только через регистр ESP, что делает дальнейшую отладку затруднительной. Зато с точки зрения эффективности такой подход дает два преимущества: (1) исчезает необходимость сохранять и восстанавливать EBP в прологе и эпилоге функции, и (2) у компилятора появляется дополнительный регистр для организации вычислений.

Еще одна возможность улучшить производительность кода появляется в случае, когда формальных параметров достаточно мало. Тогда их можно передавать не через память (стек), а через сохраняемые в вызывающей функции регистры (fastcall). Общего стандарта на такой способ вызова функций нет, но компиляторы GCC и Microsoft используют схожие правила.

Пример 4-1 “Omit frame pointer”

Реализуйте заданную функцию на языке ассемблера для платформы IA-32/Linux. ***Дополнительные требования:*** (1) реализация должна использовать рекурсивный вызов, (2) код должен отражать особенности компиляции с ключом `-fomit-frame-pointer`.

```
typedef struct t_link {
   short payload;
   struct t_link *next;
} link;

int f(link* chain) {
   return chain? chain->payload + f(chain->next) : 0;
}
```

Решение

Поскольку необходимо организовывать рекурсивный вызов, придется помещать двойное слово на стек. Будем делать это непосредственно перед рекурсивным вызовом, а сразу после вернем указатель стека на место. Таким образом, при вычис-

лении выражений ESP будет указывать на адрес возврата, а формальный параметр достижим по адресу ESP+4.

```
f:
   mov   eax, dword [esp+4]
   test  eax, eax           ; Проверяем, не 0 ли указатель chain
   jz    .false             ; Если 0 – сразу выходим, eax уже содержит нужное
                            ; число
   push  [eax+4]            ; Кладем на стек chain->next
   call  f                  ; Рекурсивный вызов
   add   esp, 4             ; Очищаем стек от фактического аргумента
   mov   ecx, dword [esp+4] ; Берем параметр chain
   movsx ecx, word [ecx]    ; Расширяем chain->payload до типа int
   add   eax, ecx           ; Добавляем ecx к тому, что вернул рекурсивный
                            ; вызов
.false:
   ret
```

Пример 4-2 Соглашение fastcall

Реализуйте приведенный ниже Си-код. ***Дополнительные требования:*** (1) реализация должна использовать рекурсивный вызов, (2) код должен отражать особенности компиляции с ключом `-fomit-frame-pointer`.

```
#include <stdio.h>

typedef struct t_dlink {
   int          payload;
   struct t_dlink *next;
   struct t_dlink *prev;
} dlink;

__attribute__((fastcall)) void f(dlink *p, _Bool direction) {
   if (!p) {
      return;
   }

   if (p->payload) {
      printf("Payload %d @ link %p\n", p->payload, (void*)p);
   } else {
      if (direction) {
         p = p->next;
      } else {
         p = p->prev;
      }
      f(p, direction);
   }
}
```

Решение

Соглашение fastcall предполагает передачу первого параметра через регистр ECX, второго – через EDX (если размера регистров хватает для их размещения). В данном

случае указатель p передается через ECX, а direction – через EDX (более точно – через DL). Для вызова printf на стеке потребуется 3 двойных слова. Для рекурсивного вызова f на стеке ничего размещать не придется, аргументы должны быть помещены в ECX и EDX, причем второй регистр обновлять не требуется. В свою очередь ECX должен быть перезаписан адресом, считанным из ячейки с адресом либо ECX+4, либо ECX+8.

```
section .rodata
   .LC0 db `Payload %d @ link %p\n`, 0 ; форматная строка для функции printf

section .text
f:
   test    ecx, ecx
   jz      .L6
   mov     eax, dword [ecx]            ; eax : p->payload
   test    eax, eax
   jnz     .L8
   test    dl, dl
   jnz     .L9                         ; Если direction == false
   mov     ecx, dword [ecx+8]          ; ecx : p = p->prev
   jmp     .L5
.L9:                                   ; Если direction == true
   mov     ecx, dword [ecx+4]          ; ecx : p = p->next
.L5:
   call    f                           ; Рекурсивный вызов с подготовленными
   ret                                 ; ecx и edx
.L8:                                   ; Сюда попадаем, когда p->payload != 0
   sub     esp, 12                     ; Выделяем место под три аргумента:
   mov     dword [esp+8], ecx          ; Третий аргумент ecx : p
   mov     dword [esp+4], eax          ; Второй аргумент eax : p->payload
   mov     dword [esp], .LC0           ; Первый аргумент – форматная строка
   call    printf
   add     esp, 12                     ; Освобождаем место аргументов
 .L6:
   ret
```

Очистка стека от аргументов вызова

Соглашение cdecl предполагает, что вызывающая функция размещает фактические аргументы на стеке, а потом освобождает стек. Другое соглашение вызова – stdcall предписывает вызванной функции самой освободить место на стеке, занятое ее параметрами. Это соглашение используется при вызове функций Win32 API ОС Windows. Возврат из функции делается командой RET N, где N – число освобождаемых байтов на стеке, помимо адреса возврата. В качестве N используют суммарный размер памяти на стеке, занятой аргументами вызова. Как следствие, это соглашение не позволяет реализовывать функции с переменным числом параметров, передаваемых по значению через стек.

Еще одной ситуацией, когда вызванная функция очищает стек от параметров, является возврат из функции структуры в компиляторе GCC. В такой ситуации задей-

ствуются правила, из-за которых реализацию cdecl в GCC называют гибридной. Поскольку значение структуры имеет произвольный размер, для его возврата в общем случае регистра EAX не хватает. Компилятор при вызове такой функции помещает первым на стек служебный параметр – адрес памяти, в которой должно быть размещено содержимое структуры. При выходе из функции этот служебный параметр освобождается командой RET 4. Если вызывающая функция изначально разметила стек так, что этот служебный параметр находится внутри ее фрейма, то при необходимости будет добавлен код компенсации, возвращающий ESP на должное место, нижнюю границу фрейма.

Пример 4-3 Соглашение stdcall

Реализуйте на языке ассемблера заданную функцию.

```
unsigned __attribute__((stdcall)) f(unsigned x, unsigned y, int pos) {
   unsigned t = ~(x & y) & (1 << pos);
   return t;
}
```

Решение

Единственной особенностью функции будет то, что в момент выхода из нее со стека будет дополнительно снято 12 байтов – память, занятая фактическими аргументами вызова.

```
f:
   push    ebp                 ; Стандартный пролог
   mov     ebp, esp

   mov     edx, dword [ebp+12]
   and     edx, dword [ebp+8]  ;   x & y
   not     edx                 ; ~(x & y)
   mov     eax, 1
   mov     ecx, dword [ebp+16]
   sal     eax, cl             ; 1 << pos
   and     eax, edx            ; ~(x & y) & (1 << pos)

   pop     ebp                ; Восстанавливаем ebp
   ret     12                 ; Возвращаемся и очищаем три двойных слова
```

Пример 4-4 Передача структуры в функцию в качестве параметра

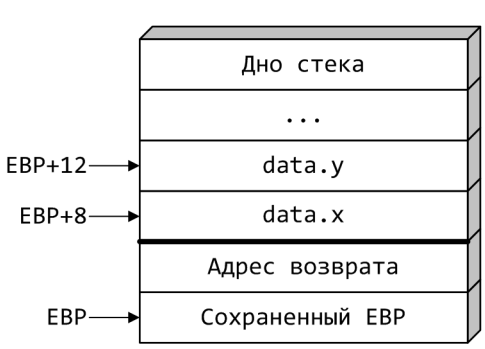
Реализуйте на языке ассемблера заданную функцию.

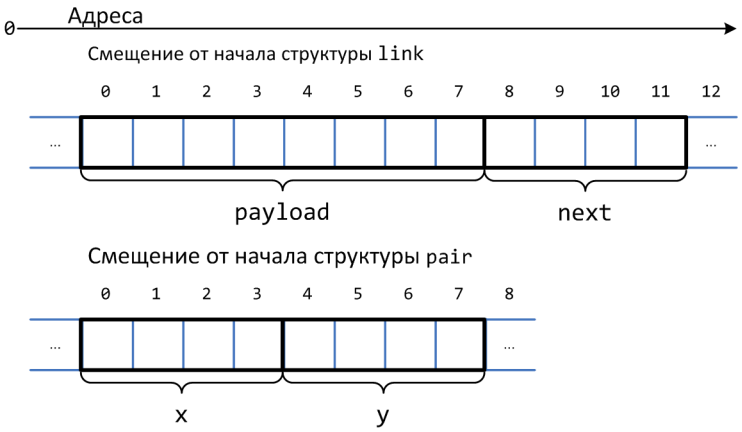
```
#include <stdlib.h>

typedef struct t_pair {
   int x;
   int y;
} pair;

typedef struct t_link {
   pair payload;
   struct t_link *next;
} link;
```

```
link* h(pair data) {
   link *p = (link*)malloc(sizeof(link));
   p->payload = data;
   p->next = (link*)0;
   return p;
}
```

Решение

Перед вызовом функции `malloc` стек должен быть выровнен по 16-байтной границе: 4 байта заняты адресом возврата, в 4 байтах сохранен `EBP`, 4 байта пропущены и не используются, 4 байта – аргумент функции `malloc`.



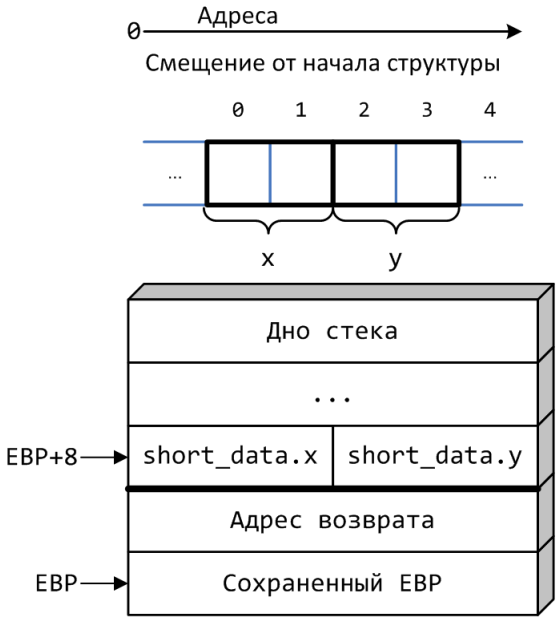



```
h:
   push    ebp
   mov     ebp, esp
   sub     esp, 4
   push    12
   call    malloc                ; В eax – адрес выделенной памяти
   mov     ecx, dword [ebp+12]   ; Копируем data по частям, а именно по
   mov     edx, dword [ebp+8]    ; отдельным полям
   mov     dword [eax+4], ecx
   mov     dword [eax], edx
   mov     dword [eax+8], 0      ; Заполняем последнее поле p->next = (link*)0
   leave
   ret
```

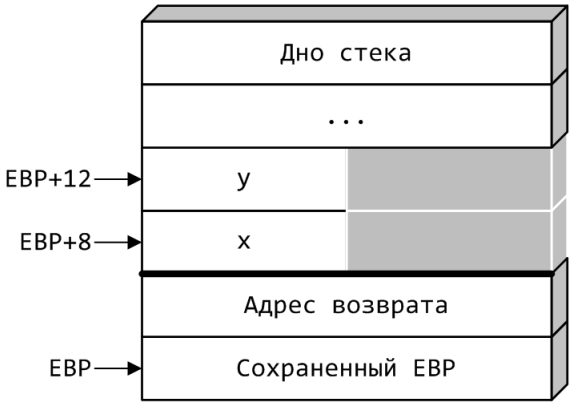
Структуры передаются в функцию по значению: на стеке будет выделено место (кратное 4 байтам), достаточное для того, что бы скопировать в него всю структуру. Размер `pair` – 8 байтов, содержимое параметра `data` будет располагаться сразу после адреса возврата. Поскольку структура состоит всего из двух полей типа `int`, об-

ращения к ним происходят по смещениям EBP+8 и EBP+12, что принципиально неотличимо от ситуации, когда в функцию передаются два параметра типа int.

Следует отметить, что если бы размеры полей были по 2 байта (например, тип short), размещение данных на стеке станет характерным для структуры. При передаче параметров для каждого из них выделятся память кратная 4 байтам. В случае, когда используются два параметра типа short, для них выделяется по 4 байта (двойное слово), причем 2 старших байта в каждом двойном слове будут использоваться в качестве заполнителя для выравнивания параметров. Для одного параметра – структуры будет выделено 4 байта, которые будут все заполнены содержимым структуры.



Передача структуры в качестве параметра



Передача двух параметров типа short

Пример 4-5 Возвращаемое значение – структура

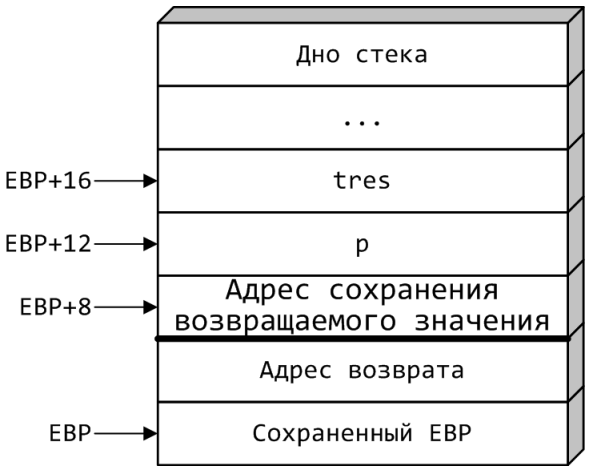
Реализуйте на языке ассемблера заданную функцию.

```
typedef struct t_pair {
   int x;
   int y;
} pair;

typedef struct t_link {
   pair payload;
   struct t_link *next;
} link;
```

```
pair f(link* p, int tres) {
   while (p) {
      int dist = p->payload.x - p->payload.y;
      if (dist < 0) {
         dist = -dist;
      }
      if (tres < dist) {
        return p->payload;
      }
      p = p->next;
   }
   return (pair){0, 0};
}
```

Решение

Основным изменением, по сравнению с привычным соглашением cdecl, будет то, что на стеке, непосредственно выше адреса возврата появится неявный служебный параметр, помещенный туда компилятором. Через этот параметр будет передаваться адрес для возвращаемого значения, являющегося структурой. Где именно будет выделена память под возвращаемое значение – «забота» вызывающей функции.



По завершении работы вызванной функции управление возвращается командой RET 4, которая освободит стек от служебного параметра, оставив на нем только те, что явно заданы пользователем в объявлении функции.

```
f:
   push ebp                  ; Стандартный для соглашения cdecl пролог
   mov  ebp, esp
   push esi                  ; Регистр esi будет использоваться в вычислениях
   mov  eax, dword [ebp+8]   ; Неявный служебный параметр – адрес для
                             ; возвращаемого значения
   mov  edx, dword [ebp+12]  ; Параметр link* p
   mov  esi, dword [ebp+16]  ; Параметр int tres
   jmp  .L2                  ; Сразу переходим на проверку условия while
 .L5:                        ; Начало тела цикла
   mov  ecx, dword [edx]     ; ecx| p->payload.x
   sub  ecx, dword [edx+4]   ; ecx| p->payload.x - p->payload.y
   jge  .L4
   neg  ecx                  ; Если число в ecx отрицательное – меняем знак
 .L4:
   cmp  esi, ecx             ; Сравниваем tres и модуль разности
   jge  .L3
   mov  ecx, dword [edx+4]   ; Возвращаем p->payload. Копируем структуру pair
   mov  edx, dword [edx]     ; в регистры edx, ecx
   mov  dword [eax+4], ecx   ; eax указывает на место размещения возвращаемого
   mov  dword [eax], edx     ; значения
   jmp  .L1                  ; Переходим на эпилог
 .L3:
   mov  edx, dword [edx+8]   ; edx| p = p->next
 .L2:
   test edx, edx             ; Проверка условия цикла while: продолжаем
   jne  .L5                  ; итерироваться, пока edx|p не равно нулю
   mov  dword [eax], 0       ; Возвращаемое значение – (pair){0, 0}
   mov  dword [eax+4], 0
 .L1:                        ; Конец тела функции – эпилог
   pop  esi                  ; Восстанавливаем регистр esi и указатель фрейма
   pop  ebp
   ret  4                    ; Удаляем со стека служебный параметр
```

Пример 4-6 Вызов функции с возвращаемой структурой

Реализуйте на языке ассемблера заданную функцию. Вызываемая функция `f` – листовая, выравнивать стек для ее вызова не обязательно. ***Дополнительное ограничение***: при реализации тела функции запрещено использовать команды `MOV`.

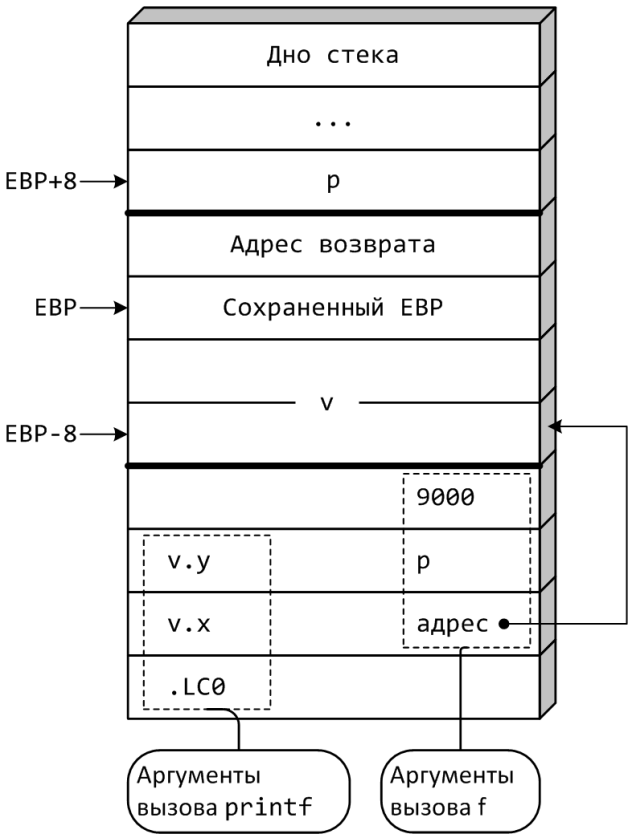
```
typedef struct t_pair {
   int x;
   int y;
} pair;

typedef struct t_link {
   pair payload;
   struct t_link *next;
} link;
```

```
pair f(link* p, int tres);

void g(link *p) {
   static const int treshold = 9000;
   pair v = f(p, treshold);
   printf("Pair (%d, %d)\n", v.x, v.y);
}
```

Решение

В отсутствии возможности пересылать данные командой `MOV`, аргументы на стек будут помещаться естественной в такой ситуации командой `PUSH`. Размер фрейма

будет меняться в процессе работы функции g. Другой, более важной особенностью рассматриваемого далее кода является то, что фактическим первым аргументом вызова функции f будет адрес автоматической локальной переменной v (неявный аргумент). Для переменной static const int treshold места в памяти не выделяется, в выражении непосредственно используется ее значение. Пространство аргументов совместно используется при вызове функций printf и f. Поскольку для вызова f стек не требуется выравнивать, будем класть на стек аргументы сразу после выделения 8 байтов памяти для хранения переменной v. Таким образом, фрейм функции g имеет следующую структуру.



После возврата из функции f двойное слово с адресом v будет освобождено. Для того, что бы функция printf вызывалась на выровненном стеке, освободим еще одно двойное слово, содержащее значение p. В результате фреймом функции g будет занято 20 байтов, пять двойных слов: адрес возврата, сохраненный регистр EBP, переменная v, оставшийся на стеке аргумент вызова 9000. Поместив тремя командами PUSH аргументы вызова printf, получим фрейм размером в 32 байта.

```
section .rodata
 .LC0 db `Pair (%d, %d)\n`, 0

section .text
g:
   push    ebp               ; Первые две команды – стандартный пролог
   mov     ebp, esp
   sub     esp, 8            ; Выделяем 8 байт для pair v
   push    9000              ; Кладем второй аргумент функции f – treshold
   push    dword [ebp+8]     ; Кладем первый аргумент функции f - p
   lea     eax, [ebp-8]      ; Вычисляем адрес переменной v и кладем его на
   push    eax               ; стек в качестве неявного аргумента
   call    f
   add     esp, 4            ; Освобождаем одно двойное слово на стеке
   push    dword [ebp-4]     ; Кладем на стек три аргумента вызова printf
   push    dword [ebp-8]
   push    .LC0
   call    printf
   leave                     ; Эпилог
   ret
```

Задачи

Задача 4-1

Для работы с однонаправленным списком без заглавного звена используется объявление:

```
typedef struct t_listnode{
   short  key;
   struct  t_listnode *next;
} listnode;
```

Учитывая, что для вывода значений, а также для работы с динамической памятью используются функции стандартной библиотеки языка Си, реализовать следующие функции:

1) `void printlist(listnode *p);`
   Печать ключей из звеньев списка `p`.
2) `listnode * addhead(listnode *p, short n);`
   Добавление нового звена с ключом `n` в голову списка `p`, возвращается указатель на головное звено списка.
3) `listnode * dellast(listnode *p);`
   Удаление последнего звена из непустого списка `p` с освобождением занятой им памятью, возвращается указатель на головное звено списка.

При реализации рассмотреть два случая:

a) реализация удовлетворяет соглашению cdecl;

б) реализация удовлетворяет соглашению fastcall.

Задача 4-2

Для работы с двоичным деревом используется объявление:

```
typedef struct t_treenode {
   short key;
   struct  t_treenode *left, *right;
} treenode;
```

Учитывая, что для вывода значений, а также для работы с динамической памятью используются функции стандартной библиотеки языка Си, рекурсивно реализовать следующие функции:

1) `int eq(treenode *t1, treenode *t2);`
   Проверка на равенство деревьев `t1` и `t2`, в случае равенства возвращается значение `1`, и `0` в противном случае.
2) `treenode * insert(treenode *t, short n);`
   Вставка новой вершины с ключом `n` в дерево поиска `t`, возвращается указатель на корень дерева.
3) `void deltree(treenode *t);`
   Удаление дерева с освобождением занятой им памятью.

При реализации рассмотреть два случая:

а) реализация удовлетворяет соглашению cdecl;
б) реализация удовлетворяет соглашению fastcall.

Задача 4-3

Пусть имеются две функции с одинаковыми прототипами:

1) `int wrapper(int a, int b, int c)`, `int actual(int a, int b, int c);`
2) `struct pair wrapper(struct pair *a, int x, int y)`, `struct pair actual(struct pair *a, int x, int y)`, где `struct pair` содержит два поля `int n` и `int m`.

Реализуйте на языке ассемблера функцию `wrapper`, которая должна вызвать `actual` с идентичным собственному набором параметров и вернуть полученное значение. При этом:

а) и `wrapper`, и `actual` используют соглашение cdecl;
б) и `wrapper`, и `actual` используют соглашение fastcall;
в) и `wrapper`, и `actual` используют соглашение stdcall;
г) `wrapper` использует соглашение cdecl, а `actual` – stdcall;
д) `wrapper` использует соглашение stdcall, а `actual` – cdecl;

е) то же, что в пункте г), но используется ключ `-fomit-frame-pointer`;

ж) то же, что в пункте д), но используется ключ `-fomit-frame-pointer`.

Задача 4-4

Реализуйте следующую функцию на языке ассемблера. Обратите внимание на то, что возвращаемую структуру можно целиком упаковать в регистр EAX.

```
struct result {
   short count;
   short min;
};

struct result f(short n, short x[n])
{
   struct result ret = { 0, 0 };

   for (short i = 0; i < n; ++i) {
      if (ret.min == x[i] && ret.count) {
         ++ret.count;
      } else if (ret.min > x[i] || !ret.count) {
         ret.min = x[i];
         ret.count = 1;
      }
   }

   return ret;
}
```

5. Сопроцессор x87 и обработка чисел с плавающей точкой

Представление вещественных чисел – числа с плавающей точкой

Для работы с вещественными числами используется представление с плавающей точкой. Число представляется в виде произведения трех множителей $(-1)^S \times M \times 2^E$. Степень `S` определяет, является число положительным или отрицательным. Мантисса `M` – дробное двоичное число в заданном полуинтервале. В стандарте IEEE 754, задающего правила работы с числами с плавающей точкой, используются полуинтервалы: `[1.0, 2.0)` для нормализованных чисел и `[0.0, 1.0)` для денормализованных. Порядок `E` определяет степень 2 в третьем множителе.

Для представления числа в машине необходимо определить правила кодирования этих трех величин. Стандарт IEEE 754 задает следующий формат:

Наибольший значащий бит `S` непосредственно кодирует знак числа. Поле `EXP` кодирует порядок числа `E`. Поле `FRAC` кодирует мантиссу `M`. Количество битов, выделяемых для кодировки мантиссы и порядка, определяют тип данных (Таблица ???) и характеризуют точность представления чисел.

Таблица 4. Размеры полей для некоторых типов данных.

	Тип	Размер	Знак S	Мантисса M	Порядок E
Одинарная точность	`float`	32 бита	1	23	8
Двойная точность	`double`	64 бита	1	52	11
Расширенная точность		80 битов	1	64	15

Типы `float` и `double` определены в стандарте языка Си, расширенная точность реализована в аппаратуре архитектуры IA-32.

Число получает нормализованное представление в заданном типе данных, если размеров полей хватает для кодировки `S`, `M` и `E` по следующим правилам.

Пусть для кодировки порядка имеется k битов. В поле `EXP` кодируется целое число $E + bias$, где $bias = 2^{k-1} - 1$. Поле `EXP` не может содержать одни нули или единицы. При этом мантисса должна принадлежать полуинтервалу `[1.0, 2.0)`. В поле `FRAC` пишется последовательность битов, кодирующая дробную часть мантиссы (т. е. ве-

дущая 1 отбрасывается). Если мантисса является периодической двоичной дробью, то при выделении битов, помещаемых в поле FRAC, происходит округление.

Если число слишком мало (по модулю), оно представляется в денормализованном виде. Поле EXP заполняется нулями, порядок E принимается равным $1-bias$, а мантисса должна в этом случае принадлежать полуинтервалу [0.0, 1.0). В поле FRAC пишется последовательность битов, кодирующая дробную часть мантиссы (т. е. ведущий 0 отбрасывается). При необходимости выполнятся округление.

Для подавления ошибок округления стандарт IEEE 754 предлагает округление к ближайшему четному числу. В случае если из двоичной дроби отбрасывается одна единица, то округляют к тому числу, у которого наименьшая значащая цифра четная, т. е. 0.

Пример 5-1 Перевод числа в модельную кодировку

Используется 10-битный формат, удовлетворяющий требованиям стандарта IEEE 754: знаковый бит, 4 бита – порядок, 5 битов – мантисса. Требуется представить в данном формате числа -⅕ и 105.

Решение

$-\frac{1}{5}=(-1)^1\times\frac{8}{5}\times 2^{-3}$, таким образом S = 1, M = $\frac{8}{5}$, E = -3. Поскольку для кодировки порядка выделено 4 бита $bias=2^{4-1}-1=7$, а EXP = -3 + $bias|_7$ = 4 = 0100_2.

Переводим $\frac{8}{5}$ в периодическую двоичную дробь разложением по степеням двойки.

$$\frac{8}{5}=1\times 2^0+\frac{6}{5}\times 2^{-1}=1\times 2^0+1\times 2^{-1}+\frac{2}{5}\times 2^{-2}=1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+\frac{4}{5}\times 2^{-3}=$$

$$=1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+0\times 2^{-3}+\frac{8}{5}\times 2^{-4}=1\times 2^0+1\times 2^{-1}+0\times 2^{-2}+0\times 2^{-3}+1\times 2^{-4}+\frac{6}{5}\times 2^{-5}=\ldots$$

Собираем коэффициенты перед степенями двоек и получаем периодическую дробь: $\frac{8}{5}=1.100(1100)_2$. Т. к. ведущая 1 в поле FRAC кодироваться не будет, записаны будут следующие, выделенные серым цветом, биты $1.1001100(1100)_2$. Не вмещающаяся в поле последовательность битов $00(1100)_2$ меньше $1(0)_2$ поэтому округление выполняется к меньшему число, т. е. к 1.10011_2.

Таким образом, искомая кодировка для -⅕ имеет вид 1_0100_10011.

Представляем 105 в виде суммы степеней двойки и переводим в двоичное представление $105_{10} = 64 + 32 + 8 + 1 = 1101001_2$. В виде трех множителей число представится как $105_{10} = (-1)^0 \times 1.101001_2 \times 2^6$. EXP = 6 + *bias*|$_7$ = 13 = 1101_2.

В записи мантиссы последняя единица не помещается в поле FRAC, что означает необходимость искать ближайшее четное число. Выпишем (в порядке убывания), какие представимые заданной кодировкой числа окружают мантиссу.

1.101010_2

$M = 1.101001_2$

1.101000_2

Как видно, после приведения к предоставленным размерам большее число оказывается оканчивающимся на 1, а меньшее – на 0. Таким образом, округляем мантиссу к 1.101000_2 и заносим последовательность выделенных серым цветом битов в поле FRAC.

Собрав все поля, получаем для числа 105 искомую кодировку $0\_1101\_10100_2$.

Сопроцессор x87

Обработка чисел с плавающей точкой в архитектуре IA-32 выполняется либо командами сопроцессора x87, либо векторными командами расширений SSE, позволяющими одновременно обрабатывать несколько чисел. На момент 2014 года было последовательно произведено пять расширений набора команд: SSE, SSE2, SSE3, SSSE3, и SSE4. Современные версии компилятора GCC отдают предпочтение именно этой группе команд, как более производительной, но если пользователь (в целях совместимости с более ранними версиями процессоров) укажет в качестве целевого процессора i386, в построении исполнимой программы будут использоваться команды сопроцессора x87.

Изучение команд SSE выходит за рамки данного курса, тогда как сопроцессор x87 реализует ряд идей, делающих его кардинально отличающимся от x86. Исторически сопроцессор был реализован отдельным кристаллом, но уже в Pentium он был окончательно интегрирован с основным процессором в рамках одного кристалла. Однако интеграция не затронула набор команд, изначально разработанный набор команд функционирует до сих пор. В наборе команд можно выделить подмножество, позволяющее работать с x87 как с безадресным, стековым процессором. Основное внимание в дальнейшем будет уделено именно этим командам.

Сопроцессор x87 содержит 8 регистров для размещения чисел с плавающей точкой. Помимо этих регистров есть еще три служебных регистра: управляющий регистр, регистр состояния и регистр признаков.

Регистры для данных организованы в виде аппаратно поддерживаемого стека. Регистр признаков содержит биты, описывающие состояние регистров данных: заняты они или свободны. Изначально все восемь регистров считаются свободными. Именование регистров меняется после того, как на верхушку стека было помещено новое значение (команда FLD) или, наоборот, значение было извлечено (команда FSTP). Обмен данными возможен только с памятью: на стек регистров x87 невозможно поместить непосредственно закодированную в команде величину-константу или переслать данные из регистров x86. Верхушка стека регистров именуется как ST0, следующий за ним регистр – ST1 и так далее, до последнего занятого регистра. После помещения на стек нового значения оно оказывается в ранее свободном регистре ST0, прежний регистр ST0 начинает называться ST1 и т. д.

Операции над числами, расположенными на стеке, выполняются командами, которые не содержат в себе явно закодированных операндов – неявными операндами являются значения из верхних регистров стека.

Пример 5-2 Взятие модуля числа

Требуется взять по модулю число двойной точности, расположенное в статической памяти.

Решение

Для взятия модуля воспользуемся безадресной командой FABS, а для пересылок данных командами FLD и FSTP. Все команды x87 имеют префикс F. Суффикс P указывает на то, что при выполнении команды на верхушке стеке освобождается один регистр и все имена оставшихся регистров «сдвигаются» описанным выше образом.

```
section .bss
   var resq 1          ; резервируем в статической памяти 8 байт

section .text
   finit               ; инициализируем сопроцессор
   fld   qword [var]   ; помещаем на стек регистров значение из памяти
   fabs                ; берем по модулю значение из верхнего регистра и
                       ; помещаем туда же полученный результат
   fstp  qword [var]   ; пересылаем в память содержимое занятого регистра
                       ; и освобождаем этот регистр
```

После выполнения приведенных команд состояние стека регистров вернется к начальному состоянию – все регистры будут свободны.

Особенности кодировки чисел с плавающей точкой позволяют брать модуль без использования x87, поразрядными битовыми командами x86.

Пример 5-3 Разность чисел

Требуется вычислить разность двух чисел с плавающей точкой одинарной точности. Разность необходимо расширить до двойной точности и сохранить в третьей переменной соответствующего размера. Все переменные расположены в статической памяти.

Решение

Для вычисления разности будет использована команда FSUBP, которая вычисляет величину ST1 – ST0. Поскольку до вычисления разности на стеке было занято два верхних элемента, а для размещения результата требуется всего один, в результате выполнения команды один элемент стека освобождается, во второй (ставший верхним) записывается результат.

Другой важной особенностью x87 является то, что вычисления по умолчанию происходят в пределах расширенной точности (80 битов на число). Точность вычислений задается в управляющем регистре и на практике не меняется после того, как сопроцессор был инициализирован командой FINIT. Преобразования типов происходят в момент пересылок данных между регистрами и памятью: команды FLD и FSTP выполняют эти преобразования согласно явно заданным размерам операнда. Допустимые спецификаторы размера операндов: DWORD, QWORD, TWORD. Переменные размера TWORD в рассматриваемых примерах не используются; данный формат уникален для сопроцессора x87 и в других распространенных архитектурах не применяется.

```
section .bss
   x resd 1          ; резервируем 4 байта для первой переменной x
   y resd 1          ; резервируем 4 байта для второй переменной y
   z resq 1          ; резервируем 8 байт для сохранения результата

section .text
   finit             ; инициализируем сопроцессор
   fld   dword [x]   ; st0 = x
   fld   dword [y]   ; st0 = y, st1 = x
   fsubp             ; Вычисляем st1 – st0, освобождаем один регистр,
                     ; результат записываем в верхний элемент стека
   fstp  qword [z]   ; Снимаем со стека регистров x87 верхнее значение
                     ; и записываем его в переменную z
```

Вычисление разности не потребовало явного задания имен регистров x87. Большинство учебных задач, связанных с x87, можно решить именно в таком стиле.

Пример 5-4 Ввод и вывод чисел с плавающей точкой

Требуется ввести с клавиатуры число с плавающей точкой двойной точности, вычислить обратную величину и напечатать ее значение с точностью до 12 знака. Используйте для хранения числа автоматическую память (стек).

Решение

Напрямую использовать команды ввода-вывода из файла io.inc для работы с плавающей точкой не представляется возможным – поддержки плавающей точки у них нет. Поэтому ввод-вывод будет организован через функции `scanf` и `printf`. Использование этих функций потребует предварительно выровнять стек, подготовить форматные строки и вычислить место на стеке, где будет размещена переменная. Для помещения на стек регистров величины `1.0` используется специальная команда `FLD1`. Вычисление обратной величины будет осуществлено командой `FDIVP`, которая делит `ST1` на `ST0`, освобождает один регистр, а во второй регистр, ставший верхушкой стека, помещает частное.

```
%include 'io.inc'
section .rodata
   input_fmt  db `%lf`, 0     ; Вводим 8-ми байтовое число с плавающей точкой
   output_fmt db `%.12f\n`, 0 ; Требуем вывести 12 знаков после точки

CEXTERN scanf
CEXTERN printf

section .text
global CMAIN
CMAIN:
   mov  ecx, esp
   and  esp, 0xfffffff0
   push ecx
   sub  esp, 28           ; 12 байт не хватает для размещения во фрейме
                          ; переменной и аргументов вызова– добавляем еще 16
   lea  eax, [esp+20]     ; esp+20 – адрес, по которому будет размещено
                          ; веденное число. Для хранения числа потребуется
                          ; 8 байт.
   mov  [esp+4], eax
   mov  dword [esp], input_fmt
   call scanf
   cmp  eax, 1
   jne  .epilog           ; Ничего не было введено

   finit
   fld1                   ; Поместили на стек регистров 1.0
   fld  qword [esp+20]    ; Поместили на стек регистров введенное число
   fdivp                  ; Поделили 1.0 на введенное число
   fstp qword [esp+20]    ; Выгрузили обратную величину в память

   mov  eax, [esp+20]     ; Копируем вычисленную величину в пространство
   mov  [esp+4], eax      ; аргументов. Поскольку число состоит из 8 байт
   mov  eax, [esp+24]     ; копируем его двумя частями по 4 байта каждый
   mov  [esp+8], eax
   mov  dword [esp], output_fmt
   call printf

 .epilog:
   add esp, 28            ; восстанавливаем исходное состояние стека
   pop ecx
   mov esp, ecx
   xor eax, eax
   ret
```

Пример 5-5 Сравнение чисел с плавающей точкой

Реализовать функцию `int cmp(double a, double b)`, возвращающую `1`, если `a` больше `b`, `-1`, если `b` больше `a`, и `0`, если параметры равны.

Решение

Сравнение чисел с плавающей точкой можно провести двумя способами.

Первый способ предполагает использование команды `FUCOMPP`, которая изначально присутствовала в x87. Результаты ее работы запоминаются в регистре состояния x87 и не могут быть сразу же использованы для изменения порядка выполнения

команд. Для этого необходимо последовательно передать состояние в регистр AX (команда FNSTSW AX), а затем переслать старший байт в EFLAGS (команда SAHF). В итоге биты, показывающие результаты сравнения, окажутся помещенными в ZF и CF, что позволяет использовать коды условий для беззнаковых чисел. Для практического кодирования такой способ сравнения относительно неудобен, поскольку приходится запоминать мнемоники крайне редко используемых команд. Положительной стороной является то, что команда FUCOMPP после сравнения освобождает оба регистра (на что указывает суффикс PP), приводя стек регистров к первоначальному состоянию. Следует отметить, что существуют две другие разновидности команды сравнения: с одним суффиксом P FUCOMP и без суффикса FUCOM.

Второй способ использует команду FUCOMIP STi (FUCOMI STi), которая появилась только в Pentium Pro. Команда сравнивает ST0 с явно заданным операндом-регистром и сразу же помещает результаты в ZF и CF. Неудобство вызывает то, что нет возможности сразу очистить оба сравниваемых регистра. Ниже приведено решение на основе второго способа.

```
section .text
cmp:
   push   ebp              ; Стандартный пролог для соглашения cdecl
   mov    ebp, esp

   xor    eax, eax         ; Предварительно обнуляем возвращаемое значение
   fld    qword [ebp+16]   ; Помещаем на стек второй параметр b
   fld    qword [ebp+8]    ; Помещаем на стек первый параметр a
   fucomi st1              ; Сравниваем st0|a vs. st1|b
                           ; Результаты сравнения сразу попадают в ZF и CF
   finit                   ; Через инициализацию очищаем стек регистров

   setb   al               ; Если a больше b, помещаем в al 1
   jbe    .1               ; Только если a меньше b …
   mov    al, 0xff         ; … помещаем в al -1
 .1:
   movsx  eax, al          ; Расширяем al до всего регистра eax

   pop    ebp
   ret
```

Пример 5-6 Вычисление площади треугольника

Реализовать функцию `double herons(double a, double b, double c)`, вычисляющую площадь треугольника по формуле Герона.

Решение

Условие задачи требует реализации вычисления по формуле $\sqrt{s(s-a)(s-b)(s-c)}$, где $s=\frac{a+b+c}{2}$. Сначала вычислим полупериметр *s*, а затем будем использовать это значение для вычисления остальных подкоренных множителей. Согласно требованиям соглашения cdecl, возвращаемое значение должно остаться в ST0, остальные регистры данных x87 должны быть свободными.

```
section .rodata
   float_two dd 2.0       ; 2.0 потребуется для вычисления полупериметра

section .text
herons:
   push   ebp             ; Стандартный пролог для соглашения cdecl
   mov    ebp, esp
   sub    esp, 8          ; Выделяем место для хранения полупериметра

   fld    qword [ebp+8]
   fld    qword [ebp+16]
   fld    qword [ebp+24]  ; Загрузили на стек регистров все три параметра
   faddp
   faddp                  ; Сложили параметры и получили периметр
   fld    dword [float_two]
   fdivp                  ; Поделили на 2.0 – получили полупериметр s
   fst    qword [esp]     ; Сохранили полупериметр в памяти, не освобождая
                          ; регистр st0 – он будет использоваться при
                          ; вычислении подкоренного произведения
   fld    qword [esp]     ; Загрузили s
   fld    qword [ebp+8]   ; Загрузили a
   fsubp                  ; Вычислили второй множитель (s-a)
   fld    qword [esp]
   fld    qword [ebp+16]
   fsubp                  ; Вычислили третий множитель (s-b)
   fld    qword [esp]
   fld    qword [ebp+24]
   fsubp                  ; Вычислили четвертый множитель (s-c)
   fmulp
   fmulp
   fmulp                  ; Все перемножили и …
   fsqrt                  ; … взяли корень. Результат остался в st0,
                          ; все остальные регистры данных x87 свободны.
   add    esp, 8
   pop    ebp
   ret
```

Задачи

Задача 5-1 Перевод числа в модельную кодировку

Используется 9-битный формат, удовлетворяющий требованиям стандарта IEEE 754: знаковый бит, 4 бита – порядок, 4 бита - мантисса. Требуется представить в данном формате числа ⅝ и -89.

Задача 5-2 Вычисление длины окружности

На стандартный вход подается число в экспоненциальной форме записи, задающее длину радиуса окружности. Требуется вычислить длину окружности и напечатать ее на стандартный вывод с точностью три десятичных знака после запятой.

Замечание: допустимо оформлять программу в виде отдельной функции, выполняющейся на уже выровненном стеке.

Задача 5-3 Приближение Чебышева

Требуется вычислить расстояние между двумя точками на плоскости, используя приближение Чебышева вместо дорогостоящей операции взятия квадратного корня.

```
sqrt(dx^2 + dy^2) = max(|dx|, |dy|) + 0.35 * min(|dx|, |dy|)
```

Ответ напечатайте на стандартный вывод, входом программ являются непосредственно `dx` и `dy`.

Задача 5-4 Теорема синусов

Требуется рассчитать длину одной из сторон треугольника по длине другой стороны и синусам противолежащих углов: $\frac{\sin\alpha}{a} = \frac{\sin\beta}{b}$.

Ниже дан фрагмент программы, которая выполняет ввод данных, требуемые вычисления и вывод результата.

Дополните приведенный текст:

1) реализуйте функцию `lawOfSin`,
2) обеспечьте ее вызов из функции `CMAIN`.

Дополнительные требования к реализации:

1. Функция `lawOfSin` соответствует объявлению
   `void lawOfSin(float a, float alpha, float beta)`.
2. Печатается результат вычислений с одинарной точностью.
3. Для печати используется функция `printf`, что дополнительно требует должного выравнивания стека.

```
%include "io.inc"

section .rodata
fmt1 db 'a=%f alpha=%f beta=%f', 0

CEXTERN scanf

section .text
global CMAIN
CMAIN:
   push ebp
   mov ecx, esp
   and esp, 0xfffffff0
   sub esp, 32
   mov [esp+28], ecx

   mov dword [esp], fmt1
   lea eax, [esp+24] ; &a
   mov [esp+4], eax
   lea eax, [esp+20] ; &alpha
   mov [esp+8], eax
   lea eax, [esp+16] ; &beta
   mov [esp+12], eax
   call scanf

   ; подготовка фактических аргументов
   ; ...
   call lawOfSin

   mov ecx, [esp+28]
   mov esp, ecx
   pop ebp
   xor eax, eax
   ret

lawOfSin:
   ; тело функции
   ret
```

Задача 5-5 Из полярных – в декартовы

Даны полярные координаты точки на плоскости, требуется перевести их в декартовы координаты. Реализуйте преобразование в виде функции, удовлетворяющей следующему объявлению на языке Си и гибридному соглашению вызова cdecl компилятора GCC.

```
typedef struct {
   double x;
   double y;
} 2DPoint;

typedef 2DPoint C_Point;
typedef 2DPoint P_Point;

D_Point fromPolarToCartesian(C_Point);
```

Более сложная версия этой задачи – обратное преобразование, из декартовых в полярные координаты.

Задача 5-6 Параллельное соединение

n сопротивлений подключены параллельно в электрическую цепь. Требуется рассчитать их суммарное сопротивление R по формуле $\frac{1}{R} = \frac{1}{R_1} + \frac{1}{R_2} + \ldots + \frac{1}{R_n}$.

Реализуйте вычисление в виде функции, удовлетворяющей следующему объявлению на языке Си.

```
double parallelCircuitResistor(int n, double resistors[n]);
```

Задача 5-7

Пусть задан массив вещественных чисел с плавающей точкой, которые необходимо просуммировать. Зависит ли получаемый результат от порядка суммирования? Если да, то какой порядок суммирования позволяет получить наиболее точный результат?

Задача 5-8

Реализуйте функцию, вычисляющую скалярное произведение двух векторов, удовлетворяющую следующему объявлению на языке Си.

```
double dot(int n, double v[n], double u[n]);
```

Задача 5-9

Реализуйте функцию, вычисляющую результат перемножения двух матриц, удовлетворяющую следующему объявлению на языке Си.

```
void mult(int n, int m, int k, double a[n][m], double b[m][k], double r[n][k]);
```

Задача 5-10

Даны следующие определения переменных на языке Си.

```
short s;
int i;
long long ll;
```

Какие из последующих выражений верны при любых начальных значениях переменных?

1) `s == (short) (float) s`,
2) `s == (short) (double) s`,
3) `i == (int) (float) i`,

4) `i == (int) (double) i,`

5) `ll == (long long) (float) ll,`

6) `ll == (long long) (double) ll.`

Ответы и решения

Задача 1-1

Приведенные ниже и в задаче 1-2 решения используют строковые инструкции IA-32. В данном пособии они не рассматриваются, ознакомиться с этой группой инструкций можно, например, в работе [6].

```
  cld                                ; просмотр в прямом порядке

  mov         esi, A                 ; загружаем адрес массива A
  mov         edi, B                 ; загружаем адрес массива B

  mov         ecx, 100               ; устанавливаем число повторов

.1:
  lodsw                              ; теперь в ax очередное слово из a
  movsx       eax, ax                ; расширение (альтернатива: cwde)
  stosd                              ; записываем двойное слово в b

  loop        .1                     ; продолжаем цикл
```

Задача 1-2

```
  std                              ; просмотр в обратном порядке
  mov   ecx, dword [N]             ; считываем количество элементов
  lea   esi, [A + 2 * ecx - 2]     ; указатель на последний элемент типа
                                   ; слово
  lea   edi, [A + 4 * ecx - 4]     ; указатель на последний элемент типа
                                   ; двойное слово
  jecxz .2                         ; пропускаем цикл, если элементов нет
.1:
  lodsw                            ; теперь в ax очередное слово из a
  movsx  eax, ax                   ; расширение (альтернатива: cwde)
  stosd                            ; записываем двойное слово
  loop  .1                         ; продолжаем цикл
.2:
```

Задача 1-3

```
section .bss
   N equ …
   a resd N

section .text
CMAIN:
   mov eax, 0                ; (1)
   mov ecx, 0                ; (2)
   mov ebx, 4 * (N - 1)      ; (3)
 .loop:
   mov edx, dword [a + ebx]  ; (4)
   cmp edx, dword [a + ecx]  ; (5)
   je  .l1                   ; (6)
   mov  eax, 1               ; (7)
   jmp .lool_end             ; (8)
 .l1:
   add ecx, 4                ; (9)
   sub ebx, 4                ; (10)
   cmp ecx, ebx              ; (11)
   jl  .loop                 ; (12)
 .loop_end:
   ret                       ; (13)
```

Задача 1-10

```
mov  eax, dword [i] ; Можно считать, что i и j из Си-программы соответствуют
mov  ebx, dword [j] ; статические переменные, расположенные в секции .data
imul ecx, eax, M
add  ecx, ebx
mov  edx, dword [A + 4 * ecx]
```

Задача 1-11

```
%include 'io.inc'

section .bss
   N equ ... ; Вместо многоточия некоторая константа
   A resd N * N

section .text
global CMAIN
CMAIN:
   ; этот блок кода с инициализацией матрицы необязателен
   mov  ecx, N * N
.init:
   mov dword [A + 4 * ecx - 4], ecx
   loop .init

   xor  eax, eax
   xor  ecx, ecx
.loop:
   cmp  ecx, N
   jge  .exit
   imul ebx, ecx, N + 1 ; это вместо
                        ; imul ebx, ecx, N
                        ; add  ebx, ecx
   mov  edx, dword [A + 4 * ebx]
   test edx, 1
   jz   .even
   add  eax, edx
.even:
   inc  ecx
   jmp  .loop
.exit:
   PRINT_DEC 4, eax
   NEWLINE
   xor eax, eax
   ret
```

Задача 3-3

```
%include 'io.inc'

section .text
global CMAIN
CMAIN:
   push ebp
   mov ebp, esp
   sub esp, 8
   GET_DEC 4, [esp] ; input a
   GET_DEC 4, [esp + 4] ; input b
   call max
   PRINT_DEC 4, eax
   NEWLINE
   xor eax, eax
   add esp, 8
   pop ebp
   ret

max:
   push ebp
   mov ebp, esp
   mov eax, dword [ebp + 8] ; get a
   cmp eax, dword [ebp + 12] ; a vs. b
   cmovl eax, dword [ebp + 12] ; if a < b eax = b
   pop ebp
   ret
```

Другой вариант функции max – без создания нового фрейма:

```
max:
   mov eax, dword [esp + 4]   ; Извлекаем a
   cmp eax, dword [esp + 8]   ; a vs. b
   cmovl eax, dword [esp + 8] ; if a < b eax = b
   ret
```

Задача 3-4

В предложенном ниже решении используются инструкции PUSH/POP для пересылки данных.

```
%include 'io.inc'

section .text
global CMAIN
CMAIN:
   push ebp
   mov ebp, esp
   sub esp, 16
   GET_DEC 4, [esp + 8]    ; input a
   GET_DEC 4, [esp + 12]   ; input b
   lea eax, [esp + 8]
   mov dword [esp], eax
   lea eax, [esp + 12]     ; add eax, 4
   mov dword [esp + 4], eax
   call swap
   PRINT_DEC 4, [esp + 8]  ; print a
   NEWLINE
   PRINT_DEC 4, [esp + 12] ; print b
   NEWLINE
   xor eax, eax
   add esp, 16
   pop ebp
   ret

swap:
   push ebp
   mov ebp, esp
   mov eax, dword [ebp + 8]  ; get a
   mov edx, dword [ebp + 12] ; get b
   push dword [eax]          ; push *a
   push dword [edx]          ; push *b
   pop  dword [eax]          ; *a <- *b
   pop  dword [edx]          ; *b <- *a
   pop ebp
   ret
```

Задача 3-5

```
int f(signed char a, signed char d, int *p, int x);
```

Задача 3-8

```
%include 'io.inc'

section .text
global CMAIN
CMAIN:
    sub esp, 4
    GET_DEC 4, [esp]
    call fact
    PRINT_DEC 4, eax
    NEWLINE
    xor eax, eax
    add esp, 4
    ret

fact:
    push ebp
    mov ebp, esp
    sub esp, 4
    mov edx, dword [ebp + 8]
    cmp edx, 1
    jg .L1
    mov eax, 1
    jmp .L2
 .L1:
    dec edx
    mov dword [esp], edx
    call fact
    imul eax, dword [ebp + 8]
 .L2:
    add esp, 4
    pop ebp
    ret
```

Допустимо слегка усложнить задачу, потребовав проверку на переполнение. В этом случае программа может выглядеть следующим образом.

```
%include 'io.inc'

section .bss
   overflow resd 1

section .text
global CMAIN
CMAIN:
    sub esp, 4
    GET_DEC 4, [esp]
    call fact
    cmp dword [overflow], 0
    jne .L1
    PRINT_DEC 4, eax
    jmp .L2
 .L1:
    PRINT_STRING "overflow detected"
 .L2:
    NEWLINE
    xor eax, eax
    add esp, 4
    ret

fact:
    push ebp
    mov ebp, esp
    sub esp, 4
    mov edx, dword [ebp + 8]
    cmp edx, 1
    jg .L1
    mov eax, 1
    jmp .L2
 .L1:
    dec edx
    mov dword [esp], edx
    call fact
    imul eax, dword [ebp + 8]
    jno .L2
    mov dword [overflow], 1
 .L2:
    add esp, 4
    pop ebp
    ret
```

Однако в этом случае более интересным вариантом было бы вычислить факториал «в обратную сторону», т. е. от меньшего к большему, обрывая рекурсию сразу, как только текущий результат перестает помещаться в 32 разряда.

Задача 3-10

```
global CMAIN
CMAIN:
   lea     ecx, [esp+4]        ; Положили в ecx указатель на первый параметр
   and     esp, -16            ; Выровняли стек, спустившись "вниз"
   push    dword [ecx-4]       ; Скопировали адрес возврата
   push    ebp                 ; Сохранили ebp и ...
   mov     ebp, esp            ; переместили его на новое место
   push    ecx                 ; Сохранили ecx и ...
   sub     esp, 20             ; заказали оставшуюся часть фрейма.
                               ; Т.к. выравнивание начинается с адреса
                               ; возврата, то уже израсходовано
                               ; 12 байт = адрес возврата + ecx + ebp
                               ; Заказываем 20 байт, чтоб получилось два
                               ; блока по 16
                               ; Далее свободно работаем с имеющимся
                               ; пространством фрейма
   mov     eax, dword [ecx+4]  ; Копируем параметры в низ фрейма
   mov     dword [esp+4], eax
   mov     eax, dword [ecx]
   mov     dword [esp], eax    ; дабы вызвать какую-нибудь функцию
;    call  ...
   mov     eax, 0
   add     esp, 20             ; Возвращаем указатель фрейма на место
   pop     ecx                 ; Восстанавливаем регистры
   pop     ebp
   lea     esp, [ecx-4]        ; Ставим указатель стека на исходную позицию
   ret
```

Задача 3-11

```
%include 'io.inc'

section .text
global CMAIN
CEXTERN time
CEXTERN ctime
CEXTERN puts

CMAIN:
   lea     ecx, [esp+4]
   and     esp, -16
   push    dword [ecx-4]
   push    ebp
   mov     ebp, esp
   push    ecx
   sub     esp, 20
   mov     dword [esp], 0
   call    time
   mov     dword [ebp-8], eax
   lea     eax, [ebp-8]
   mov     dword [esp], eax
   call    ctime
   mov     dword [esp], eax
   call    puts
   mov     eax, 0
   add     esp, 20
   pop     ecx
   pop     ebp
   lea     esp, [ecx-4]
   ret
```

Задача 3-12

```
memcpy:
   push        ebp                     ; стандартный пролог
   mov         ebp, esp

   push        esi                     ; сохраняем esi и edi,
                                       ; т.к. они нам понадобятся
   push        edi

   mov         edi, dword [ebp + 8]    ; edi := destination
   mov         esi, dword [ebp + 12]   ; esi := source
   mov         ecx, dword [ebp + 16]   ; ecx := num

   mov         eax, edi                ; должны вернуть destination

   cld                                 ; сбрасываем df
   rep         movsb                   ; копируем побайтово

   pop         edi                     ; восстанавливаем esi и edi
   pop         esi

   leave                               ; стандартный эпилог
   ret
```

Задача 3-13

```
memmove:
   push        ebp                     ; стандартный пролог
   mov         ebp, esp

   push        esi                     ; сохраняем esi и edi,
                                       ; т.к. они нам понадобятся
   push        edi

   mov         edi, dword [ebp + 8]    ; edi := destination
   mov         esi, dword [ebp + 12]   ; esi := source
   mov         ecx, dword [ebp + 16]   ; ecx := num

   mov         eax, edi                ; должны вернуть destination

   cmp         esi, edi                ; сравниваем указатели, чтобы выбрать
                                       ; направление
   jb          .1

   cld                                 ; source >= destination, копировать
                                       ; будем слева направо
   jmp         .2

.1:
   std                                 ; source < destination, копировать
                                       ; будем справа налево

   lea         esi, [esi + ecx - 1]    ; переставляем указатели на конец
                                       ; блоков
   lea         edi, [edi + ecx - 1]

.2:
   rep         movsb                   ; выполняем побайтовое копирование

   pop         edi                     ; восстанавливаем esi и edi
   pop         esi

   leave                               ; стандартный эпилог
   ret
```

Задача 3-16

Решение для варианта задачи а):

```
strpfx:
   push        ebp                     ; стандартный пролог
   mov         ebp, esp

   push        esi                     ; сохраняем esi и edi,
                                       ; т.к. они нам понадобятся
   push        edi

   mov         esi, dword [ebp + 8]    ; esi := s1
   mov         edi, dword [ebp + 12]   ; edi := s2

   mov         ecx, 0xffffffff         ; инициализируем ecx наибольшим числом

   cld                                 ; просмотр слева направо (*)
   repe        cmpsb                   ; сравниваем побайтово, пока равно (*)

   neg         ecx                     ; ecx := n + 1, где n - длина префикса
   lea         eax, [ecx - 1]          ; готовим возвращаемое значение

   pop         edi                     ; восстанавливаем esi и edi
   pop         esi

   leave                               ; стандартный эпилог
   ret
```

В данном решении используется строковая команда `CMPSB` с префиксом повтора (строки, помеченные звездочкой). Если строки-параметры в точности совпадают, то организованный этой командой цикл продолжится за пределами строк, что является ошибкой. Для того чтобы исправить этот недостаток, чего требует вариант б), необходимо отказаться от использования строковой команды, заменив помеченные звездочкой строки на явный цикл:

```
.1:
   mov         al, byte [esi]      ; явно сравниваем байты
   cmp         al, byte [edi]
   jne         .2

   test        al, al              ; проверяем на нуль-терминатор
   je          .2                  ; выходим из цикла по нуль-терминатору

   inc         esi
   inc         edi
   dec         ecx
   jmp         .1
```

Задача 5-2

```
%include 'io.inc'

section .rodata
   input_format  db '%g', 0
   output_format db 'c = %.3f', 10, 0

global cmain
cmain:
   lea     ecx, [esp+4]
   and     esp, -16
   push    dword [ecx-4]
   push    ebp
   mov     ebp, esp
   push    ecx
   sub     esp, 4
   call    circumference
   add     esp, 4
   pop     ecx
   pop     ebp
   lea     esp, [ecx-4]
   ret

circumference:
   push    ebp
   mov     ebp, esp
   sub     esp, 24

   mov     dword [esp], input_format
   lea     eax, [esp + 20]
   mov     dword [esp + 4], eax
   call    scanf
   fld     dword [esp + 20]
   fld     dword [esp + 20]
   faddp
   fldpi
   fmulp
   fstp    qword [esp + 4]
   mov     dword [esp], output_format
   call    printf

   leave
   ret
```

Замечание. Приведенное решение может быть ассемблировано и выполнено. Содержательная часть – функция `circumference` и секция `.rodata`.

Задача 5-3

```
%include 'io.inc'

section .data
   _dx  dq 42.0
   _dy  dq 23.0
   __k  dq 0.35

section .rodata
   output_format db 'ca = %.3f', 10, 0

global CMAIN
CMAIN:
   lea     ecx, [esp+4]
   and     esp, -16
   push    dword [ecx-4]
   push    ebp
   mov     ebp, esp
   push    ecx
   sub     esp, 4
   call    cheba_approx
   add     esp, 4
   pop     ecx
   pop     ebp
   lea     esp, [ecx-4]
   ret

cheba_approx:
   push    ebp
   mov     ebp, esp
   sub     esp, 24

   fld     qword [_dx]
   fld     qword [_dy]
   fucom
   fnstsw  ax
   sahf
   jc .1
   fxch
.1:
   fld     qword [__k]
   fmulp
   faddp

   fstp    qword [esp + 4]
   mov     dword [esp], output_format
   call    printf

   leave
   ret
```

На стек регистров значения следует помещать в должном порядке – сверху меньшее, под ним большее. Тогда можно выполнить умножение верхушки стека на `0.35` и сложить результат с нижним числом. Если изначально порядок у чисел был иной – меняем их местами.

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.

6. А.В. Столяров. Архитектура ЭВМ и системное программное обеспечение: язык ассемблера в ОС Unix. Часть I. / Московский государственный технический университет ГА, ISBN 978-5-86311-769-0, 2010.

7. А.В. Столяров. Программирование на языке ассемблера NASM для ОС Unix. / МАКС Пресс. ISBN 978-5-317-03627-0, 2011.