МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ИНДУСТРИАЛЬНЫЙ УНИВЕРСИТЕТ

В.Ю. Радыгин

Архитектура вычислительных систем и компьютерных сетей

Учебно-методическое пособие

Москва 2011

УДК 004.2
ББК 32.81
Р15

Рецензенты:

Е.А. Роганов, кандидат физико-математических наук,
доцент, зав. каф. информационных систем и технологий
Московского государственного индустриального университета;
И.М. Белова, кандидат физико-математических наук,
доцент Московского государственного индустриального университета

Радыгин В.Ю.

Р15 Архитектура вычислительных систем и компьютерных сетей: учебно-методическое пособие. — М.: МГИУ, 2011. — 45 с.
ISBN 978-5-2760-1974-1

Предназначено для студентов, обучающихся по направлениям «Прикладная математика и информатика», «Информатика и вычислительная техника» и специальности «Математическое обеспечение и администрирование информационных систем». Содержит набор лабораторных проектов, посвящённых решению различных задач с помощью языка ассемблер для микропроцессоров архитектуры Intel 80386 (IA32). Описания проектов включают в себя краткую теоретическую справку, примеры выполнения и различные варианты заданий.

УДК 004.2
ББК 32.81

ISBN 978-5-2760-1974-1

Оглавление

Предисловие

Учебное пособие подготовлено автором на основе многолетнего опыта чтения курсов «Архитектура вычислительных систем и компьютерных сетей» и «Организация ЭВМ и ситем» студентам Московского государственного индустриального университета (направления «Прикладная математика и информатика» и «Информатика и вычислительная техника», а также специальности «Математическое обеспечение и администрирование информационных систем»). При разработке данного курса было использовано только свободное программное обеспечение. Его же рекомендуется применять и в процессе обучения.

1. Общая теория. Основы архитектуры Intel 80386

Микропроцессор Intel 80386 (i80386) был выпущен фирмой Intel в 1985 году и стал первым 32-разрядным микропроцессором в линейке IBM PC-совместимых персональных компьютеров. Большинство последующих микропроцессоров линейки x86 поддерживает весь набор возможностей данного микропроцессора. Это означает, что программы, написанные для работы на архитектуре микропроцессора i80386 (или IA32), будут запускаться и на большинстве современных персональных компьютеров.

Предшественниками микропроцессора Intel 80386 были 16-разрядные IBM PC-совместимые машины, в том числе Intel 8086/8088, Intel 80186, Intel 80286. В связи с этим архитектура i80386 поддерживает весь набор команд и возможностей своих 16-разрядных предшественников. Таким образом, микропроцессор 80386 поддерживает три варианта работы:

1) реальный режим, в котором он деградирует до микропроцессора 8086 и позволяет выполнять программы, написанные для 16-битной архитектуры (причём в случае сбоя происходит полный отказ системы);
2) виртуальный режим 8086, в котором операционная система создает специальную изолированную среды, работающую как процессор 8086 (причём в случае сбоя операционной системе передается соответствующая информация и полного отказа системы не происходит);
3) защищенный режим, в котором он работает как полноценная 32-разрядная система.

В защищённом режиме доступны четыре уровня привилегий, которые управляются битами во флаговом регистре. Уровень ноль соответствует привилегированному режиму и имеет полный доступ к машине. Его использует операционная система. Уровень три предназначен для пользовательских программ. Он блокирует доступ к определённым командам и регистрам управления, чтобы ошибки какой-нибудь пользовательской программы не привели к выходу из строя всей машины. Микропроцессор 80386 поддерживал адресуемое пространство памяти размером до 4 Гб (32-битный адрес). Микропроцессор 8086 мог работать только с адресным пространством памяти в 20 бит (максимальный объём 1Мб). При этом байты памяти нумеровались от 0 до $2^{20}-1$. В дальнейшем в данном методическом пособии будем рассматривать только работу в защищённом режиме.

Микропроцессор Intel 80386, также как и его прародитель микропроцессор Intel 8086 не содержит встроенных средств обработки вычислений для чисел с плавающей запятой. Данные вычисления осуществляются при

помощи специальной микросхемы — сопроцессора (FPU — Floating Point Unit). Поэтому набор команд и регистров при вычислениях с целыми операндами и при вычислениях с дробными операндами разный. Хотя в более старших версиях микропроцессоров линейки IBM PC-совместимых персональных компьютеров блок сопроцессора встроен внутрь самого микропроцессора, минимальный набор команд и регистров остаётся тот же, что был и у 80386.

Таким образом, с точки зрения программирования, из составляющих микропроцессора нас будут интересовать только арифметическо-логическое устройство (АЛУ), блок операций с плавающей запятой (сопроцессор), регистры общего назначения, регистр флагов и стек регистров сопроцессора. С арифметическо-логическим устройством (и связанной с ним схемой сдвига) программист взаимодействует посредством выполнения инструкций. Набор инструкций включает в себя операции перемещения информации, операции управления выполнением программы, вычисления с целыми числами и т.д. Блок операций с плавающей запятой предоставляет программисту набор инструкций по управлению стеком сопроцессора и множество арифметических, логических и трансцендентных операций над числами с плавающей запятой.

Множество регистров общего назначения содержит восемь сверхбыстрых запоминающих ячеек с собственными именами. Микропроцессор Intel 80386 является 32-разрядной машиной. Это означает, что все основные операции могут обрабатывать операнды, двоичное представление которых помещается в 32 бита. Большинство команд имеет три варианта работы: с 32-битными значениями, с 16-битными значениями и 8-битными значениями. Исключение составляют действия с дробными числами. О них будет рассказано позже. Как следствие, регистры общего назначение также имеют размер 32 бита. Множество регистров общего назначения состоит из регистров `eax`, `ebx`, `ecx`, `edx`, `ebp`, `esp`, `esi` и `edi`. Все регистры общего назначения могут быть использованы в 16-битном варианте. В этом случае рассматривается только их младшая часть, а название сокращается до `ax`, `bx`, `cx`, `dx`, `bp`, `sp`, `si` и `di`, соответственно. Первые четыре из регистров общего назначения поддерживают также деление на 8-битный режим. Тогда биты с 0-го по 7-й образуют младшую часть (`al`, `bl`, `cl` и `dl`, соответственно), а биты с 8-го по 15-й образуют старшую часть (`ah`, `bh`, `ch` и `dh`, соответственно). Буквы `l` и `h` соответствуют английским словам low и high.

Отдельный интерес представляет регистр флагов `EFlags`. В данном 32-битном регистре каждый бит отвечает за определённый флаг. Флаги служат двум целям. Одни флаги влияют на выполнение последующих инструкций. Другие флаги содержат характеристики результата последней арифметической или логической операции. Назначение каждого бита в ре-

Таблица 1

Регистр флагов (побитовая карта)

№	Обознач.	Английское название	Название с расшифровкой
0	CF	Carry Flag	Флаг переноса
1	1	-	Зарезервированная единица
2	PF	Parity Flag	Флаг чётности
3	0	-	Зарезервированный ноль
4	AF	Auxiliary Carry Flag	Вспомогательный флаг переноса для вычислений с псевдодесятичными числами
5	0	-	Зарезервированный ноль
6	ZF	Zero Flag	Флаг нуля
7	SF	Sign Flag	Флаг знака
8	TF	Trap Flag	Флаг трассировки
9	IF	Interrupt Enable Flag	Флаг разрешения прерываний
10	DF	Direction Flag	Флаг направления
11	OF	Overflow Flag	Флаг переполнения (знакового переноса)
12	IOPL	I/O Privilege Level	Первый бит уровня приоритета ввода/вывода
13	IOPL	I/O Privilege Level	Второй бит уровня приоритета ввода/вывода
14	NT	Nested Task	Флаг вложенности задачи
15	0	-	Зарезервированный ноль
16	RF	Resume Flag	
17	VM	Virtual 8086 Mode	
18	0	-	Зарезервированный ноль
...	...	...	...
31	0	-	Зарезервированный ноль

гистре `EFlags` описано в табл. 1.

Немного поясним назначения флагов. Флаги `CF` и `OF` отвечают за переполнение. Представление целых чисел на компьютере накладывает ограничения на минимальное и максимальное значения. Например, 32-битное целое число может быть знаковым или беззнаковым. Знаковые числа лежат в диапазоне от -2^{31} до $2^{31}-1$. Беззнаковые числа лежат в диапазоне от 0 до $2^{32}-1$. Флаг `CF` отвечает за переполнение (или перенос) в беззнаковых числах. Флаг `OF` за переполнение в знаковых числах. Оба флага хранят единицу, если в результате последнего действия произошло соответствующее переполнение.

Флаг `PF` характеризует чётность числа. То есть он содержит значение последнего бита из двоичного представления результата последней арифметической (или логической) операции.

Флаг AF относится к переполнению в режиме псевдодесятичных чисел. Микропроцессор Intel 80386 умеет работать в режиме, когда 32 бита представления чисел делятся на группы по 4 бита. Каждая такая группа соответствует одной цифре от 0 до 9. В общем случае, 4 бита позволяют хранить больше информации. Но остальные комбинации (от 10 до 15) просто игнорируются. Таким образом, получается одно число, состоящее из 8-ми десятичных цифр. Операции с такими цифрами приводят в некоторых случаях к переполнениям, не совпадающим с переполнениями, описанными флагами CF и OF. Чтобы понять нужна ли коррекция, используется флаг AF.

Флаг ZF показывает, был ли равен нулю результат последней арифметической (логической) операции или нет. Во флаг SF копируется знаковый бит из двоичного представления результата последней операции.

Флаг TF включает/выключает режим трассировки. В данном режиме после каждого действия генерируется прерывание. Он удобен для отладки программ.

Флаг IF отвечает за резрешение/запрет прерываний.

Флаг DF влияет на направления движения некоторых циклических операций. Движение по массивам данных осуществляется посредством изменения указателей на соответствующие области памяти. Значение указателей на каждом шаге можно уменьшать на единицу (движение из конца в начало), но можно и увеличивать на единицу (движение из начала в конец). Чтобы выбрать направление движения, используется флаг DF.

Пара бит, образующих флаг IOPL, отвечает за уровень приоритета операций ввода/вывода, применяемый в защищённом режиме. Флаг NT устанавливается процессором при переходе из одной задачи в другую с использованием инструкции CALL или аппаратного прерывания. Он показывает, была ли выполняемая задача текущей или является вызванной из предыдущей задачи. Флаг RF управляет реакцией процессора на возникающие при отладке исключения.

Флаг VM переключает процессор из защищённого режима в режим эмуляции микропроцессора Intel 8086 (если выставлен в единицу).

2. Проект 1. Простейшие действия с массивами

Язык Ассемблер (Assembler) уникален для каждой архитектуры компьютера. Таким образом, всё, что приводится в данной главе будет работать только на соответствующих моделях микропроцессоров. Поэтому прежде чем начинать работу убедитесь, что Ваш персональный компьютер совместим с Intel 80386. Далее под словом Ассемблер будет подразумеваться Ассемблер для архитектуры 80386 и ни какой иной.

Существуют два основных стандарта написания программ для языка Ассемблер: в синтаксисе фирмы Intel и в синтаксисе фирмы AT&T. Данное пособие описывает только синтаксис фирмы Intel.

2.1. Основы языка Ассемблер для Intel 80386. Программа на языке Ассемблер в общем случае может состоять из разнообразных комбинаций четырёх видов объектов: инструкций, директив, меток и аргументов. Инструкция — это, собственно говоря, буквенная интерпретация основных команд микропроцессора. Обычно инструкция состоит из комбинации английских букв и цифр, образующих либо сокращение от английского слова, либо аббревиатуру словосочетания. Регистр букв значения не имеет. Пример:

```
mov,
CALL,
LOOP
push.
```

Метка в Ассемблере — это способ присвоить имя тому или иному месту программы. Название метки может быть образовано из комбинации английских букв и цифр. При установке метки в программе после её названия указывается двоеточие. При ссылке на метку двоеточие не используется. Пример:

```
start:
      ...
      loop start
```

В качестве аргументов инструкций могут выступать разные объекты. Все виды передачи аргументов называются способами адресации. Микропроцессор Intel 80386 допускает большое число таких способов, но в данном пособии не будут рассмотрены все допустимые варианты, а только основные из них.

1. Непосредственная адресация: аргумент задаётся в виде константного значения. Пример:
   `push 2`
2. Регистровая адресация: аргумент передаётся в виде значения, лежащего в регистре общего назначения. Пример:
   `push eax`
3. Косвенная адресация: аргумент считывается из оперативной памяти по адресу, рассчитываемому по формуле:

$$\text{адрес} = [\text{база} + \textit{масштаб} \times \textit{индекс} \pm \textit{смещение}],$$

   где в качестве базы и индекса могут выступать значения, лежащие в регистрах общего назначения, масштаб принимает только значения из набора {1, 2, 4, 8}, а смещение — это некоторая целочисленная константа. Обязательной составляющей формулы является только база. Масштаб, индекс и смещение могут отсутствовать. Масштаб также не применяется отдельно от индекса. Пример:
   `mov eax, [ebx + 4 * ecx - 16]`

Директивы — это специальные команды, указывающие самой программе Ассемблера (транслятору, переводящему из языка Ассемблер в машинные коды) различные дополнительные аспекты. Например, где располагаются определённые данные, как организовать те или иные секции программы и т.д. К сожалению, общего стандарта для директив нет! В данном пособии рассматривается набор директив для обработки компилятором *as*, который используется в среде *UNIX*. Так как компилятор *gcc* сам вызывает *as* для обработки файлов, то можно использовать непосредственно его.

Все директивы начинаются с точки и состоят из последовательности английских букв, цифр и некоторых спец символов. Например:

```
.intel_syntax
.double
.globl
```

Обычно учебники по языкам программирования в качестве первого примера содержат программу “Hello World!”, печатающую на экран фразу «Здравствуй, мир!» на английском языке. В случае с Ассемблером задача вывода чего-либо на экран не тривиальна. Более того, в современном мире мало кто пишет на Ассемблере полноценные программы. Поэтому и мы будем рассматривать язык Ассемблер не как основное средство программирования, а как механизм, позволяющий более оптимально написать

вспомогательные функции. Реализованные с помощью Ассемблера функции будут потом использоваться в полноценной программе, написанной на языке *C*. Поэтому вместо "Hello World!" рассмотрим пример написания на языке Ассемблер функции сложения двух чисел и пример её использования в программе на языке *C*. Подробно о языке *C* можно прочесть в [3].

Пример 1.1. Функция сложения двух целых четырёхбайтовых чисел на языке Ассемблер.

```
.intel_syntax noprefix
.globl add
.type add,@function

add:
    mov eax, [esp + 8]
    add eax, [esp + 4]
    ret
```

Пример 1.2. Программа на языке *C*, использующая функцию `add`.

```
# include <stdio.h>

extern int add(int, int);

int main(void){
   int a, b;
   scanf("%d", &a);
   scanf("%d", &b);
   printf("%d\n", add(a, b));
   return 0;
}
```

Чтобы запустить написанную нами программу, необходимо разместить текст примера 1.1 в файле `add.s`, текст примера 1.2 в файле `add.c` и скомпилировать их в исполняемый файл при помощи компилятора *gcc*:

```
gcc add.s add.c
```

Полученный запускаемый файл a.out можно выполнить в терминале, добавив спереди точку и слэш:

```
./a.out
```

Чтобы понять содержимое данного примера, рассмотрим основные непривилегированные инструкции микропроцессора Intel 80386, позволяющие работать с целыми числами.

Инструкции для перемещения данных

mov *получатель*, *источник* — инструкция копирует значение из операнда источник в операнд получатель. Оба операнда должны быть одного размера.

Обычно по операндам можно понять какого они размера. Например, команда

```
mov eax, 1
```

подразумевает операнды 32-битного размера. Команда

```
mov ax, 1
```

подразумевает операнды 16-битного размера. Команда

```
mov al, 1
```

подразумевает операнды 8-битного размера. Размер операндов очевиден из названий регистров: `eax`, `ax` и `al`. Если операнды не позволяют определить свой размер, то он может быть задан явно при помощи указателей размерности: `byte ptr` — 8 бит, `word ptr` — 16 бит, `dword ptr` — 32 бита, `qword ptr` — 64 бита. Пример:

```
mov dword ptr [esp + 4], 2
```

Кроме требования равенства размерности команда `mov` (как и большинство других команд Ассемблера) накладывает ограничения на «качество» операндов. Опуская некоторые подробности (и сегментные регистры), можно упрощённо сказать, что один из них всегда должен быть либо регистром общего назначения, либо константой.

xchg *получатель*, *источник* — инструкция меняет местами значения из получателя и источника. Оба операнда должны быть одного размера. Причём один из них обязательно должен быть регистром общего назначения, и они оба не могут быть константами.

lea *получатель*, *источник* — инструкция загрузки эффективного адреса (load effective address). Источником всегда должен быть операнд с косвенной адресацией. Получателем — регистр. Команда вычисляет адрес источника и кладёт его (адрес) в получателя. Вычисление адреса не изменяет значений регистра флагов `EFlags`. Пример:

```
lea eax, [ebx + 4 * ecx - 4]
```

Для дальнейшего знакомства с инструкциями микропроцессора Intel 80386 необходимо рассмотреть понятие стека локальных переменных. Когда мы пишем функцию на языке *C*, то часто используем внутри неё локальные переменные. Пример:

```
int distance2(int x1, int y1, int x2, int y2){
   int d1, d2;
   d1 = (x1 - x2);
   d2 = (y1 - y2);
   return d1 * d1 + d2 * d2;
}
```

В данном примере шесть локальных переменных: `x1`, `y1`, `x2`, `y2`, `d1` и `d2`. Мы не задумывались, когда писали функцию, где они будут лежать. На самом деле, все локальные переменные размещаются в специальной области памяти, носящей название стека локальных переменных. Адрес вершины стека локальных переменных всегда размещается в регистре `esp`. Поэтому мы никогда не будем использовать данный регистр для других целей. Сам стек размещается в оперативной памяти таким образом, чтобы его хвост был направлен в сторону роста адресов, а голова — в сторону уменьшения роста адресов. То есть при добавлении в стек новой локальной переменной его голова смещается в сторону начала оперативной памяти, а хвост всегда стоит на месте. Все объекты в стеке локальных переменных занимают место, кратное четырём байтам.

В показанном примере у функции было четыре аргумента. Если бы мы хотели написать её на Ассемблере, а не на *C*, то как бы выглядел стек локальных переменных в самом начале функции? Соглашение языка *C* подразумевает, что все аргументы функции размещаются на стеке локальных переменных в обратном порядке. То есть глубже всех будет лежать переменная `y2`, а выше всех — переменная `x1`. Помимо переменных на стеке также размещается так называемый адрес возврата. Адрес возврата — это адрес той области памяти, в которой располагается код программы, вызывающей нашу функцию. Он нужен, чтобы по завершении функции можно было понять куда надо вернуть управление, то есть какую команду выполнять следующей. Адрес возврата всегда лежит на самой вершине стека локальных переменных и занимает 4 байта (в случае архитектуры Intel 80386).

Таким образом, в нашем случае стек будет выглядеть, как показано на рис. 1.

Теперь можно рассмотреть набор инструкций, обеспечивающих работу со стеком локальных переменных.

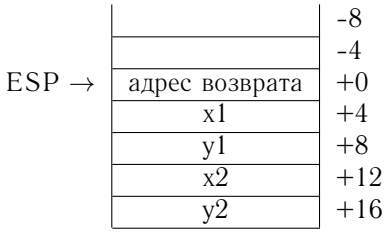

Рис. 1. Размещение переменных на стеке

push *источник* — помещает значение источника на вершину стека локальных переменных. При этом стек сдвигается (изменяется значение регистра `esp`), и все остальные локальные переменные оказываются размещёнными глубже.

pop *получатель* — перемещает значение с вершины стека локальных переменных в получателя. При этом стек сдвигается в обратную сторону (изменяется значение регистра `esp`), и все остальные локальные переменные оказываются размещёнными ближе к началу. Очевидно, что получатель не может быть константой.

pushad — копирует значения всех регистров общего назначения на вершину стека локальных переменных (порядок: `eax`, `ecx`, `edx`, `ebx`, `esp`, `ebp`, `esi`, `edi`). После загрузки каждого регистра значение регистра esp сдвигается (увеличивается) на его размер.

popad — загружает значения всех регистров общего назначения с вершины стека локальных переменных, кроме значения `esp`. То есть сначала загружаются `edi`, `esi` и `ebp`. Затем стек просто сдвигается на четыре байта. А после загружаются оставшиеся `ebx`, `edx`, `ecx` и `eax`.

pushfd — загружает в стек значение регистра флагов.

popfd — загружает значение с вершины стека в регистр флагов.

Арифметические операции

add *получатель*, *источник* — складывает два знаковых или беззнаковых числа по принципу: получатель = получатель + источник. Оба операнда могут быть образованы как при помощи косвенной, так и при помощи регистровой адресации. Источник может также быть константой. Аргументы должны быть одного размера (1, 2 или 4 байта). Два операнда не могут одновременно адресоваться косвенно.

sub *получатель*, *источник* — осуществляет вычитание для двух знаковых или беззнаковых чисел по принципу: получатель = получатель - источник. Оба операнда могут быть образованы как при помощи косвенной, так и при помощи регистровой адресации. Источник может также быть константой. Аргументы должны быть одного размера (1, 2 или 4 бай-

та). Два операнда не могут одновременно адресоваться косвенно.

cmp *источник*1, *источник*2 — осуществляет «сравнение» двух знаковых или беззнаковых чисел по принципу: источник1 - источник2. Аргументы должны быть одного размера (1, 2 или 4 байта). Два операнда не могут одновременно адресоваться косвенно. Результат никуда не записывается! Основная цель инструкции — изменить флаги.

adc *получатель*, *источник* — складывает два числа и значение флага переноса по принципу: получатель = получатель + источник + `CF`. Оба операнда могут быть образованы как при помощи косвенной, так и при помощи регистровой адресации. Источник может также быть константой. Аргументы должны быть одного размера (1, 2 или 4 байта). Два операнда не могут одновременно адресоваться косвенно.

sbb *получатель*, *источник* — осуществляет вычитание для двух чисел с учётом флага переноса по принципу: получатель = получатель - источник - `CF`. Оба операнда могут быть образованы как при помощи косвенной, так и при помощи регистровой адресации. Источник может также быть константой. Аргументы должны быть одного размера (1, 2 или 4 байта). Два операнда не могут одновременно адресоваться косвенно.

mul *источник* — умножение беззнаковых чисел. Значение источника умножается на значение регистра `eax` (`ax` или `al` в зависимости от размера), и полученный результат записывается в `edx`:`eax` (`dx`:`ax` или `ah`:`al`). Для источника может применяться как регистровая, так и косвенная адресации.

imul *источник* — умножение знаковых чисел. Значение источника умножается на значение регистра `eax` (`ax` или `al` в зависимости от размера), и полученный результат записывается в `edx`:`eax` (`dx`:`ax` или `ah`:`al`). Для источника может применяться как регистровая, так и косвенная адресации. Доступны также ещё два варианта работы этой инструкции: `imul` получатель, источник и `imul` получатель, источник1, источник2. В первом из них осуществляется умножение по принципу: получатель = источник × получатель. Во втором — по принципу: получатель = источник1 × источник2.

div *источник* — деление беззнаковых чисел. Значение источника делится на значение регистра `eax` (`ax` или `al` в зависимости от размера). Частное записывается в `eax` (`ax` или `al`), а остаток в `edx` (`dx` или `ah`). Для источника может применяться как регистровая, так и косвенная адресации.

Важно! До выполнения деления необходимо обнулить регистр `edx` (`dx` или `ah`). В противном случае — при выполнении программы либо произойдёт ошибка, либо результат деления будет неверен!

Замечание. Обнулять регистр `edx` при помощи команды `mov`:

```
mov edx, 0
```

неэффективно! Более правильно использовать либо не описанное в данном пособии исключающее или:

```
xor edx, edx,
```

либо рассматриваемую ниже инструкцию `cltd`.

idiv ***источник*** — деление знаковых чисел. Значение источника делится на значение регистра `eax` (`ax` или `al` в зависимости от размера). Частное записывается в `eax` (`ax` или `al`), а остаток в `edx` (`dx` или `ah`). Для источника может применяться как регистровая, так и косвенная адресации.

inc ***получатель*** — увеличивает значение получателя на единицу, не изменяя флага `CF`. Для получателя может применяться как регистровая, так и косвенная адресации.

dec ***получатель*** — уменьшает значение получателя на единицу, не изменяя флага `CF`. Для получателя может применяться как регистровая, так и косвенная адресации.

neg ***получатель*** — инвертирует знак получателя. Для получателя может применяться как регистровая, так и косвенная адресации.

cbw — конвертирует знаковое 8-битное число в 16-битное. Начальное значение должно лежать в регистре `al`. Результат помещается в регистр `ax`.

cwde — конвертирует знаковое 16-битное число в 32-битное. Начальное значение должно лежать в регистре `ax`. Результат помещается в регистр `eax`.

cltd (или **cdq**) — конвертирует знаковое 32-битное число в 64-битное. Начальное значение должно лежать в регистре `eax`. Результат помещается в пару регистров `eax` и `edx`.

Управляющие инструкции

call ***адрес*** **(*метка*)** — передача управления по адресу. Говоря более простым языком, инструкция `call` соответствует вызову функции. Прежде чем перейти в указанное место, `call` размещает на стеке адрес возврата (адрес следующей за собой инструкции). Если `call` используется для вызова функции стандартной библиотеки *C*, то аргументы функции должны быть предварительно размещены на стеке в обратном порядке.

ret [*кол-во байт для удаления*] — возврат управления назад. Данная инструкция ищет на вершине стека адрес возврата и переходит на соответствующую ему инструкцию, удаляя при этом его со стека. Если указан необязательный аргумент — целое число, то перед осуществлением перехода со стека удаляется указанное число байт.

iret — отличается от ret только тем, что дополнительно загружает со стека значения регистра флагов.

jmp *адрес* (*метка*) — безусловный переход по адресу (метке), то есть переход к выполнению той инструкции, которая расположена по данному адресу в памяти или стоит после данной метки.

j*xx адрес* (*метка*) — набор инструкций условного перехода. Вместо символов *xx* подставляются краткие обозначения флагов или их комбинация с символом *n*, означающим отрицание. Иногда краткое обозначение флагов заменяется сокращением их смыслового назначения. Например, сокращённому обозначению флага `ZF` (`z`) соответствует сокращение от слова *equal* — `e`. Основные примеры:

jc — переход в случае если флаг `CF` равен единице;
jnc — переход в случае если флаг `CF` равен нулю;
js — переход в случае если флаг `SF` равен единице;
jns — переход в случае если флаг `SF` равен нулю;
jz, **je** — переход в случае если флаг `ZF` равен единице;
jnz, **jne** — переход в случае если флаг `ZF` равен нулю.

loop *адрес* (*метка*) — простейший цикл. Каждый раз при достижении инструкции `loop` значение регистра ecx уменьшается на единицу. Если после этого `ecx` становится равным нулю, то происходит переход к следующей за `loop` инструкции, иначе происходит переход по указанной метке (адресу). Обычно цикл выглядит следующим образом:

```
присвоение значения ecx
...
label:
   действия в теле цикла
   loop label
```

При таком подходе надо быть осторожным! Цикл всегда выполняется хотя бы один раз! Более того, при стартовом значении `ecx`, равном нулю, цикл будет работать «бесконечно».

clc — устанавливает флаг `CF` в ноль.
stc — устанавливает флаг `CF` в единицу.
cld — устанавливает флаг `DF` в ноль.
std — устанавливает флаг `DF` в единицу.

Более подробно с набором инструкций микропроцессора Intel 80386 для целочисленных вычислений можно ознакомиться в [1, 2, 4].

Основные директивы

Кроме набора инструкций нам придётся кратко познакомиться с набором директив Ассемблера *as*. Важно понимать, что данный набор директив будет работать только в той среде, для которой он предназначен.

В частности, рассматриваемые директивы будут работать при компиляции программой *as* на 32-битной архитектуре в среде *Linux*.

.intel_syntax noprefix — директива, включающая синтаксис языка Ассемблер фирмы Intel. Ключ *noprefix* подразумевает, что названия регистров будут писаться без дополнительного символа процента перед ними.

.globl метка — директива, экспортирующая название во внешнее пространство имён.

.type метка,тип — директива, указывающая тип экспортируемого имени. Например, для функций:

```
.type label,@function
```

.byte, .word, .int — директивы определения константных переменных для однобайтового, двухбайтового и четырёхбайтового целых чисел. Пример:

```
...
var1: .int 32
...
   mov eax, var1
```

.string — директива определения константной текстовой строки. Для получения указателя на данную строку можно использовать конструкцию offset. Пример:

```
...
var1: .string "Hello World!"
...
   mov eax, offset var1
```

Теперь можно вернуться к примеру 1.1. Первые три строки примера (будем называть их в дальнейшем «шапкой») содержат подготовочные директивы. Включается нужный синтаксис языка. Экспортируется название функции во внешнее пространство имён.

Содержательная часть примера начинается с метки `add:`, соответствующей названию формируемой нами функции. Аргументы функции по соглашению языка C размещаются на стеке локальных переменных следом за адресом возврата. Адрес возврата занимает четыре байта, переменные типа `int` тоже. Ситуация на стеке локальных переменных показана на рис. 2.

Из рисунка видно: для того чтобы добраться до первого аргумента функции, надо к указателю, хранящемуся в регистре `esp`, прибавить четыре, а до второго — восемь.

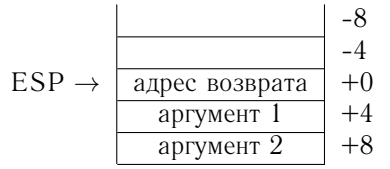

Рис. 2. Размещение переменных на стеке для задачи сложения двух чисел

Команда сложения не позволяет просуммировать два объекта с косвенной адресацией. То есть один из аргументов необходимо переложить со стека локальных переменных в любой из регистров общего назначения. Это и делается при помощи инструкции `mov`. Следующий шаг — это непосредственное сложение (`add`).

Осталось определиться, каким образом функция будет возвращать полученный результат. Язык *C* подразумевает, что для этой цели (в случае если возвращаемое значение по размеру меньше или равно четырём байтам) служит регистр `eax`, а именно в нём сейчас и лежит сумма. Поэтому завершаем нашу функцию и возвращаем управление назад посредством `ret`.

2.2. Работа с массивами. Наиболее часто используемое представление массивов в языке *C* предусматривает размещение элементов массива единым куском в памяти. Обычно для работы с ними необходимо знать указатель на начало массива и количество элементов в нём.

Рассмотрим пример решения задачи на обработку массивов: напишите на языках *C* и Ассемблер функцию, заменяющую значения элементов массива целых двухбайтовых чисел (тип `short`) на их абсолютную величину (модуль). Прототип функции будет следующим:

```
short* array_abs(short* array, int size);
```

Сначала рассмотрим реализицию на *C*.

Пример 2.1.

```
# include <stdlib.h>

short* array_abs(short* array, int size)
{
   while(size > 0){
      size--;
      array[size] = abs(array[size]);
   }
   return array;
```

```
}
```

Давайте рассмотрим, что размещается на стеке локальных переменных в начале работы функции (рис. 3).

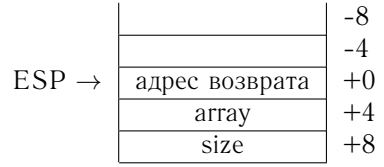

Рис. 3. Размещение переменных на стеке для задачи с массивом

Подумаем, какие регистры придётся занять. Во-первых, нужен регистр для указателя на массив, так как Ассемблер не поддерживает двойную операцию взятия значения по адресу. Во-вторых, понадобится регистр `ecx` для счётчика цикла. В него мы положим переменную `size`. В Ассемблере нет инструкции для вычисления модуля целого числа. Делать эту операцию придётся в два шага: сначала проверим знак числа, а затем, если оно отрицательно, то инвертируем при помощи инструкции `neg`.

Наш план решения содержит один недостаток. Если мы будем использовать регистр `ecx`, то испортим значение, лежавшее в нём до вызова функции, что может привести к краху или неправильной работе вызывающего процесса. Поэтому значения всех используемых нами регистров необходимо предварительно сохранить на стеке локальных переменных. Почему же мы не делаем это для регистра *eax*? Это дозволяется, так как данный регистр предназначен для возвращения функцией результата (в языке *C*). Следовательно, он не может быть не изменён. Стек при сохранении значений регистров изменится (рис. 4).

Рис. 4. Размещение переменных на стеке для задачи с массивом после сохранения регистра *ecx*

Теперь по аналогии с примером 2.1 можно написать и реализацию на Ассемблере.

Пример 2.2.

```
.intel_syntax noprefix
.globl array_abs
.type array_abs,@function

array_abs:
   push ecx
   mov eax, [esp + 8]
   mov ecx, [esp + 12]
   cmp ecx, 0
   jz end
step:
   cmp word ptr [eax + ecx * 2 - 2], 0
   jns next
   neg word ptr [eax + ecx * 2 - 2]
next:
   loop step
end:
   pop ecx
   ret
```

Осталось только написать тестовую программу на языке *C*, которая позволяла бы продемонстрировать использование реализованных функций.

Пример 2.3.

```
# include <stdio.h>

extern short* array_abs(short*, int);

int main(void)
{
   short b[10] = {1, -1, 2, -2, -3, 3, -4, 4, 5, -5}, *a;
   a = array_abs(b, 10);
   printf("%hd %hd %hd %hd ...\n", a[0], a[1], a[2], a[3]);
   return 0;
}
```

2.3. Задачи для самостоятельного решения. Напишите на языках *C* и Ассемблер функции, описание и прототип которых приведены ниже.

1. `int* reverse(int* array, int size);`
   Функция переставляет местами элементы массива `array` из конца в начало. Массив состоит из целочисленных элементов размерностью в четыре байта. Их количество указанно в переменной `size`.
2. `char* invert(char* array, int size);`
   Функция инвертирует знак элементов массива `array`. Массив состоит из целочисленных элементов размерностью в один байт. Их количество указанно в переменной `size`.
3. `short sum(const short* array, int size);`
   Функция считает сумму элементов массива `array`. Массив состоит из целочисленных элементов размерностью в два байта. Их количество указанно в переменной `size`.
4. `int scalar_mult(const int* a, const int* b, int size);`
   Функция вычисляет скалярное произведение элементов двух массивов: `a` и `b`. Массивы состоят из целочисленных элементов размерностью в четыре байта. Их количество указанно в переменной `size`.
5. `char* minus(char* a, const char* b, int size);`
   Функция вычисляет разность двух векторов по принципу $a = a - b$. Векторы представлены в виде массивов, состоящих из целочисленных элементов размерностью в один байт. Их количество указанно в переменной `size`.
6. `short* vector_mult(const short* a,`
   `const short* b, short* res);`
   Функция вычисляет векторное произведение двух векторов размерностью три по принципу: $res = a \times b$. Векторы представлены в виде массивов, состоящих из целочисленных элементов размерностью в два байта.
7. `int* mod(int* a, int size, int unit);`
   Функция вычисляет остатки от деления компонент вектора `a` на целое число `unit`: $a = a \,\%\, unit$. Вектор представлен в виде массива, состоящего из целочисленных элементов размерностью в четыре байта. Их количество указанно в переменной `size`.
8. `int* max(int* a, int* b, int* r, int size);`
   Функция вычисляет для каждой пары элементов с одинаковыми индексами из массивов `a` и `b` максимальное значение и записывает его в элемент с тем же индексом массива `r`. Векторы представлены в виде массивов, состоящих из целочисленных элементов размерностью в четыре байта. Их количество указанно в переменной `size`.

9. `int* min(int* a, int* b, int size);`
   Функция вычисляет для каждой пары элементов с одинаковыми индексами из массивов `a` и `b` минимальное значение и записывает его в элемент с тем же индексом массива `a`. Массивы состоят из целочисленных элементов размерностью в четыре байта. Их количество указанно в переменной `size`.
10. `int max(int* a, int size);`
    Функция находит максимальный элемент массива `a`. Массив состоит из целочисленных элементов размерностью в четыре байта. Их количество указанно в переменной `size`.
11. `int* sort(int* a, int size);`
    Функция сортирует пузырьковым методом элементы массива `a` по убыванию. Массив состоит из целочисленных элементов размерностью в четыре байта. Их количество указанно в переменной `size`.

3. Проект 2. Вычисления с длинными числами

В своей жизни человек при необходимости легко оперирует в вычислениях бесконечными множествами чисел (конечно, имеются в виду не вычисления в уме). Единственное ограничение — это Ваше терпение и способности. Возможности же компьютера в обычной ситуации ограничены. Архитектура любого современного ЭВМ не позволяет осуществлять непосредственные вычисления с числами произвольной длины. В частности, архитектура Intel 80386 для целых чисел ограничена 32-мя битами.

С другой стороны, любой хороший программист легко может, используя ограниченные числа, реализовать методы обработки длинных чисел. Как это делается? Чтобы найти решение данного вопроса нужно прежде всего рассмотреть, как представлются целые числа на компьютере. Причём важно понимать, что представление только неотрицательного ряда целых чисел (будем называть их беззнаковые) и представление всех целых чисел, включая отрицательные (будем называть их знаковыми), — это разные задачи.

3.1. Представление целых чисел в архитектуре 80386. Ещё в середине XX века Фон-Нейман показал преимущество использования двоичной системы представления чисел на компьютерах. Двоичная система применяется на сегодняшний день для представления как целых, так и вещественных чисел. Рассмотрим для начала представление целых беззнаковых чисел. Например, архитектура Intel 80386 поддерживает 8-битный режим работы с целыми числами. Это означает, что для представления целого числа используется не более 8-ми двоичных цифр. Минимальное число, соответственно, — ноль, максимальное — $11111111_2 = 2^0 + 2^1 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7 = 255_{10}$.

Вспомним материалы школьного курса. Как перевести число из десятичной системы счисления в двоичную? Например, мы хотим перевести число 133_{10} из десятичной системы счисления в двоичную. Можно воспользоваться делением в столбик. Но мы используем степени двойки. Известно, что $2^7 = 128$, $2^6 = 64$, $2^5 = 32$, $2^4 = 16$, $2^3 = 8$, $2^2 = 4$, $2^1 = 2$, $2^0 = 1$. Число 133 больше, чем 128. Следовательно, в позиции с номером семь мы записываем единицу и уменьшаем 133 на 128. Остаётся 5. Числа 64, 32, 16 и 8 больше 5. Следовательно, в позиции шесть, пять, четыре и три записывается ноль. Число 4 меньше 5. Поэтому записываем в позицию два единицу и уменьшаем число на четвёрку. Остаётся единица. Два больше, чем единица. Поэтому записываем в позицию один ноль. Единица совпадает с единицей нулевой степени двойки, следовательно, в

позиции ноль у нас будет единица. Процесс завершён! Получаем: $133_{10} = 10000101_2$.

Представление знаковых чисел на компьютере гораздо сложнее. Во-первых, старший бит числа теперь играет роль знака. Неотрицательные числа начинаются с нуля, отрицательные — с единицы. Таким образом, неотрицательных чисел становится меньше. Доступны числа от 0 до 127. Отрицательные числа доступны в диапазоне от -1 до -128. Для получения представления отрицательного числа недостаточно к 7-ми битам двоичной формы его абсолютного значения добавить спереди 8-ю единичку. Всё гораздо сложнее. Для начала модуль числа уменьшается на единицу, а затем к нему применяется побитовое инвертирование. Например, посмотрим, как будет записано число -3.

1. Необходимо перевести -3_{10} в двоичную систему счисления. Для этого получим модуль числа: $|-3_{10}| = 3_{10} = 0000011_2$.
2. Уменьшим на единицу: $|3_{10}| - 1 = 0000010_2$.
3. Применим побитовое инвертирование: $!(0000010_2) = 1111101_2$.
4. Добавим знаковую единицу: $-3_{10} = 11111101_2$.
5. Ответ получен: $-3_{10} = 11111101_2$.

Зачем так сложно? Преимущество такого представления легко понять на примере. Давайте сложим числа -3 и 3, используя алгоритм сложения беззнаковых чисел. То есть будем считать, что двоичное компьютерное представление числа $-3_{10} = 11111101_2$ является обычным беззнаковым числом. Тогда вместо -3 мы получаем число $253_{10} = 2^0 + 2^2 + 2^3 + 2^4 + 2^5 + 2^6 + 2^7$. Сложим: $3_{10} + 253_{10} = 256_{10} = 100000000_2$. Обратите внимание, что результат такого сложения не поместился в требуемые восемь бит. Если отрезать лишние старшие биты, то получаем ноль! Что и требовалось получить в исходной задаче. Таким образом, данное представление отрицательных чисел позволяет использовать те же самые алгоритмы сложения и вычитания, что и для беззнаковых чисел. К сожалению, умножение и деление такого простого подхода не позволяют.

3.2. Перенос, переполнение и заём. Как видно из последнего примера, при вычислениях на компьютере результат может не поместиться в отведённое для представления числа место. Причём для знаковых и беззнаковых чисел такая ситуация наступает для различных значений. Например, для 8-битных чисел результат знаковых вычислений перестаёт помещаться уже при превышении величины 127. Для беззнаковых чисел предел — 255. Будем называть превышение допустимого знакового значения переполнением, а превышение допустимого беззнакового значения — переносом. Переносу соответствует флаг *Carry* (`CF`), а переполнению — флаг *Overflow* (`OF`).

Перенос в случае операции сложения приводит к потере старшего бита, так как его некуда записать. Поэтому флаг `CF` как раз играет роль недостающей цифры. В случае переполнения мы получаем более сложную ситуацию. Потеря старшего бита приводит к смене знака числа. В частности, если к 8-битному знаковому числу 127 добавить единицу, мы получим минимальное отрицательно число — -128.

Похожая ситуация происходит и в случае операции вычитания. Например, если попробовать вычесть в беззнаковом режиме что-либо из числа ноль, то произойдёт заём недостающего бита. То есть вычитание осуществится так, как будто у нас был не ноль, а число $100000000_2 = 256_{10}$. Для знаковых чисел переход через минимально допустимую границу значений приводит к смене знака. Например, $-128 - 1 = 127$.

Чтобы лучше понять смысл флага переноса, вспомним, как нас учили складывать столбиком в школе. Например, сложим два числа: 1567 и 4488 (рис. 5).

$$\begin{array}{cccccc} & 1 & 1 & 1 & \\ & 1 & 5 & 6 & 7 \\ + & & & & \\ & 4 & 4 & 8 & 8 \\ \hline & 6 & 0 & 5 & 5 \end{array}$$

Рис. 5. Сложение столбиком чисел 1567 и 4488

Аналогичным образом можно складывать в любой системе счисления, начиная с двоичной (рис. 6) и заканчивая системами с большими основаниями, например, с основанием 256 (рис. 7).

$$\begin{array}{cccccccc} & 1 & 1 & 0 & 0 & 0 & & \\ & 1 & 1 & 1 & 1 & 0 & 1 & \\ + & & & & & & & \\ & 1 & 0 & 1 & 0 & 1 & 0 & \\ \hline 1 & 1 & 0 & 1 & 1 & 1 & 1 & \end{array}$$

Рис. 6. Сложение столбиком чисел 111101_2 (61_{10}) и 101010_2 (42_{10})

Из рассмотренных примеров видно, что «единичка в уме», которой нас учили в школе, — это упрощённая модель флага переноса. Причём последний пример наглядно показывает, как данный флаг можно использовать. Если есть возможность производить вычисления с 8-битовыми числами, то, используя флаг переноса, можно из обычного сложения получить «длинное». Каждое длинное число при этом делится на 8-битовые блоки, а

$$\begin{array}{rccc} & 1 & 1 & \\ & 214 & 128 & 1 \\ + & & & \\ & 1 & 127 & 255 \\ \hline & 216 & 0 & 0 \end{array}$$

Рис. 7. Сложение столбиком чисел 214 128 1_{256} (14057473_{10}) и 1 127 255_{256} (98303_{10})

вычисления осуществляются по аналогии со школьным алгоритмом «столбика» с учётом «единицы в уме», хранящейся во флаге `CF`. Для обобщения алгоритма будем считать, что при сложении младших цифр двух чисел у нас «в уме» хранится ноль.

3.3. Сложение двух 128-битных чисел. Рассмотрим пример решения задачи на обработку длинных чисел. Пусть есть два числа, двоичное представление которых помещается в 128 бит, но не помещается (хотя бы у одного из них) в 96 бит. Числа поделены на кусочки в 32 бита (соответствует типу *int* языка *C*), которые сложены в массивы. Результат сложения тоже будем считать 128-битным числом (пренебрежём возможным переполнением), хранящимся в аналогичном массиве.

Прототип функции на языке *C* будет следующим:

```
int* add128(int* a, int* b, int* res);
```

Замечание: третий аргумент функции нужен для передачи указателя на массив, в котором будет размещён результат сложения. Тот же самый указатель необходимо вернуть по завершении вычислений в качестве вычисляемого функцией значения.

Рассмотрим, что будет храниться на стеке локальных переменных (рис. 8).

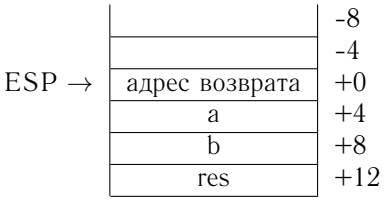

Рис. 8. Размещение переменных на стеке

Подумаем, какие регистры нам придётся занять. Ассемблер не поддерживает двойную операцию взятия значения по адресу. Поэтому нам по-

требуются три регистра для хранения указателей на массивы `a`, `b` и `res`. Пусть это будут `esi`, `edi` и `eax`. Также необходимо завести счётчик элементов, чтобы следить сколько уже обработано элементов массива. Счётчик положим в регистр `ecx`. Ассемблер не позволяет сложить два элемента с косвенной адресацией. По этой причине нам потребуется ещё один регистр для промежуточного сложения. Выберем для этой цели регистр `ebx`.

Значения всех используемых нами регистров, кроме регистра *eax*, необходимо предварительно сохранить на стеке локальных переменных. После таких преобразований стек примет вид, показанный на рис. 9.

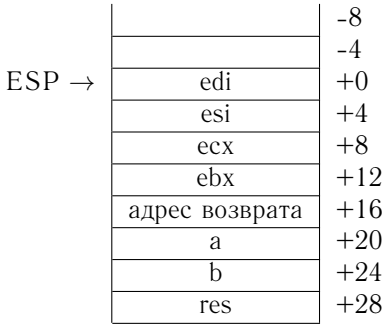

Рис. 9. Размещение переменных на стеке после сохранения значений используемых регистров

Если части представления числа уложены в массивы в обычном (для человека) порядке, то старшие биты находятся в начале массива, а — младшие в конце. Алгоритм сложения чисел в столбик выполняет обработку чисел от младших цифр к старшим. Следовательно, нам будет необходимо двигаться по массиву с конца в начало. Как получить адрес последнего элемента массива, зная адрес его начала и общее число элементов? Пусть, для однозначности, мы рассматриваем массив `a`. Тогда, согласно нашей схеме, адрес его начала лежит в регистре `esi`, а номер элемента в регистре `ecx`. Один 32-битный элемент занимает четыре байта памяти. Тогда n-й элемент массива доступен по формуле:

```
[esi + 4 * ecx].
```

Замечание. Количество элементов массива на единицу больше номера его последнего элемента. Поэтому если в `ecx` лежит общее количество элементов массива, то адрес последнего элемента массива будет получаться по следующей формуле:

```
[esi + 4 * ecx - 4].
```

Роль «запоминаемой в уме» цифры у нас будет играть флаг `CF`. На первом шаге его надо обнулить! Это можно сделать командой `clc`.

Всё готово, чтобы написать реализацию функции на языке Ассемблер. Пример 3.1.

```
.intel_syntax noprefix
.globl add128
.type add128,@function

add128:
        push ebx
        push ecx
        push esi
        push edi
        mov esi, [esp + 20]
        mov edi, [esp + 24]
        mov eax, [esp + 28]
        mov ecx, 4
        clc
step:
        mov ebx, [esi + 4 * ecx - 4]
        adc ebx, [edi + 4 * ecx - 4]
        mov [eax + 4 * ecx - 4], ebx
        loop step

        pop edi
        pop esi
        pop ecx
        pop ebx
        ret
```

Осталось только написать тестовую программу на языке *C*, которая позволяла бы продемонстрировать использование реализованной функции. Пример 3.2.

```
# include <stdio.h>

extern int* add128(int*, int*, int*);

int main(void)
{
        int a[4] = {1, 0xffffffff, 1, 0xffffffff},
```

```
        b[4] = {0, 1, 0, 1},
        c[4] = {0, 0, 0, 0}, *res;
    res = add128(a, b, c);
    printf("%x %x %x %x\n", res[0], res[1],
                            res[2], res[3]);
    return 0;
}
```

3.4. Задачи для самостоятельного решения. Напишите на языках *C* и Ассемблер функции, описание и прототип которых приведены ниже.

1. `char* minus(const char* a, const char* b, char* c, int size_a, int size_b);`
   Функция вычитает длинное число `b` из длинного числа `a` и кладёт ответ в длинное число `c`. Длинные числа представлены массивами из 16-битных элементов. Размер массива `b` — `size_b` вхождений, массивов `a` и `c` — `size_a` вхождений. Функция возвращает указатель на начало массива `c`.
2. `short* minus(short* a, const short* b, int size);`
   Функция вычитает длинное число `b` из длинного числа `a` и кладёт ответ назад в длинное число `a`. Длинные числа представлены массивами из 16-битных элементов. Размер массивов `a` и `b` — `size` вхождений. Функция возвращает указатель на начало массива `a`.
3. `unsigned int* avg(unsigned int* a, const unsigned int* b, int size);`
   Функция вычисляет среднее арифметическое двух длинных чисел `a` и `b` и кладёт ответ назад в длинное число `a`. Длинные числа представлены массивами из 32-битных элементов. Размер массивов `a` и `b` — `size` вхождений. Функция возвращает указатель на начало массива `a`. Для деления на два рекомендуется использовать операцию сдвига `shr`.
4. `unsigned int* avg(unsigned int* a, const unsigned int* b, unsigned int* c, int size);`
   Функция вычисляет среднее арифметическое двух длинных чисел `a` и `b` и кладёт ответ в длинное число `c`. Длинные числа представлены массивами из 32-битных элементов. Размер массивов `a`, `b` и `c` — `size` вхождений. Функция возвращает указатель на начало массива `c`. Для деления на два рекомендуется использовать операцию сдвига `shr`.
5. `int* longdiv(const int* a, const int* b, int* c, int size);`

Функция осуществляет целочисленное деление длинного числа `a` на длинное число `b` и кладёт ответ в длинное число `c`. Длинные числа представлены массивами из 32-битных элементов. Размер массивов `a`, `b` и `c` — `size` вхождений. Деление длинных чисел реализуйте путём их вычитания. Функция возвращает указатель на начало массива `c`.

6. `int* longmod(const int* a, const int* b, int* c, int size);`

Функция осуществляет целочисленное деление длинного числа `a` на длинное число `b` и кладёт остаток от деления в длинное число `c`. Длинные числа представлены массивами из 32-битных элементов. Размер массивов `a`, `b` и `c` — `size` вхождений. Деление длинных чисел реализуйте путём их вычитания. Функция возвращает указатель на начало массива `c`.

7. `int* longmul(const int* a, const int* b, int* c, int size);`

Функция осуществляет умножение длинного числа `a` на длинное число `b` и кладёт результат в длинное число `c`. Длинные числа представлены массивами из 32-битных элементов. Размер массивов `a`, `b` и `c` — `size` вхождений. Умножение длинных чисел реализуйте путём их сложения. Функция возвращает указатель на начало массива `c`.

8. `int* longmul(const int* a, const int* b, int size);`

Функция осуществляет умножение длинного числа `a` на длинное число `b` и кладёт результат назад в длинное число `a`. Длинные числа представлены массивами из 32-битных элементов. Размер массивов `a`, и `b` — `size` вхождений. Умножение длинных чисел реализуйте путём их сложения. Функция возвращает указатель на начало массива `a`.

4. Проект 3. Операции с числами с плавающей запятой

4.1. Представление чисел с плавающей запятой на архитектуре Intel 80386. Важным аспектом при работе с числами с плавающей запятой (далее дробные числа) является тот факт, что их представление на компьютере ограничено, в отличие от целых чисел, не только по максимальному и минимальному значеням. Ограничения накладываются также и на точность представления чисел, то есть на количество значащих цифр в их записи. По этой причине допустимы только дробные числа, имеющие конечную форму записи. Например, дробь $\frac{1}{3}$ будет округлена до конечного числа знаков.

Математический сопроцессор Intel 80387 умеет обрабатывать три основных типа дробных чисел 32-битное число, 64-битное число и 80-битное число. Первым двум из них соответствуют типы языка *C float* и *double*, соответственно.

В предыдущей главе был рассмотрен алгоритм преобразования целого числа из десятичной системы счисления в двоичную систему счисления. Но как быть с дробными числами? Разберём преобразование дробных чисел из десятичной системы счисления в двоичную на примере. Пусть в десятичной системе задано число $12,625_{10}$. Преобразуем целую часть числа по старому алгоритму: $12_{10} = 10100_2$. Остаётся преобразовать дробную часть. Рассмотрим по убыванию степени числа два, начиная со степени -1 (табл. 1). Будем по очереди сравнивать десятичную запись числа со значениями отрицательных степеней двойки. Если значение степени больше оставшейся части числа, то в соответствующую позицию после запятой вписывается ноль, иначе число уменьшается на значение степени, и в соответствующую позицию вписывается единица. Процесс повторяется до тех пор, пока число не превратится в ноль. Для числа $12,625_{10}$ получаем:

$$12,625_{10} = 10100,\ldots_2 \quad \text{остаток: } 0,625;$$

$$0,625 > 0,5 \Rightarrow 12,625_{10} = 10100,1\ldots_2 \quad \text{остаток: } 0,125;$$

$$0,125 < 0,25 \Rightarrow 12,625_{10} = 10100,10\ldots_2 \quad \text{остаток: } 0,125;$$

$$0,125 = 0,125 \Rightarrow 12,625_{10} = 10100,101_2.$$

Известно, что любое число в десятичной системе счисления может быть записано в экспоненциальной форме. Или, говоря другим языком, любое

Показатель степени	Полная запись	Десятичная дробь
-1	2^{-1}	0,5
-2	2^{-2}	0,25
-3	2^{-3}	0,125
-4	2^{-4}	0,0625
-5	2^{-5}	0,03125
-6	2^{-6}	0,015625

Таблица 1. Соответствие отрицательных степеней двойки десятичным дробям

число может быть записано в такой форме, когда перед десятичной запятой стоит всего одна цифра, и она отлична от нуля (исключение составляет число ноль). Например:

$$1234,56 = 1,23456 \times 10^3;$$

$$120 = 1,2 \times 10^2;$$

$$-0,000123456 = -1,23456 \times 10^{-4}.$$

Аналогичный подход применим и для двоичной системы счисления. Например, наше число $12,625_{10} = 10100,101_2$ может быть записанно в виде: $1,0100101_2 \times 2^{100_2}$. Степень двойки (в данном примере 100_2) называется экспонентой, а предшествующее знаку умножения число (в данном примере $1,0100101_2$) — мантиссой.

В десятичной системе счисления цифра перед запятой в экспоненциальной форме записи числа может принимать значения от 1 до 9. В двоичной форме возможен только один вариант — 1. По этой причине разработчики фирмы Intel приняли решение её не хранить. Таким образом, в представлении дробных чисел хранится не вся мантисса, а только её часть после запятой. Экспонента в общем случае может быть как положительной, так и отрицательной. Это не очень удобно. Разработчики Intel всегда прибавляют к экспоненте некоторое большое положительное число, что позволяет избежать необходимости отслеживания знака. Размеры мантиссы и экспоненты для разных типов дробных чисел показаны в табл. 2.

4.2. Работа с математическим сопроцессором Intel 80387.

Сопроцессор Intel 80387 содержит собственный набор регистров и предоставляет пользователю собственный набор команд. Группа основных, необходимых для работы программиста регистров включает в себя набор регистров данных (`R0-R7`), регистр состояния (`SR`) и регистр управления (`CR`).

Общее кол-во бит	Тип данных в языке *C*	Длина мантиссы	Длина экспоненты
32	*float*	23 бита + 1 бит для знака	8 бит
64	*double*	52 бита + 1 бит для знака	11 бит
80	–	64 бита + 1 бит для знака	15 бит

Таблица 2. Основные поддерживаемые архитектурой Intel 80386 типы дробных чисел

Регистры данных аналогичны регистрам общего назначения основного микропроцессора с двумя отличиями. Во-первых, они все полностью равнозначны и служат одинаковым целям. Во-вторых, они не могут быть доступны по непосредственному имени. Вместо этого регистры данных образуют циклический стек. Вершина стека адресуема по имени st(0), остальные регистры по аналогичным именам с большими индексами: st(1)-st(7). Все регистры данных имеют размер в 80 бит, но могут использоваться и для 32- и 64-битных операций.

Регистры состояния и управления играют роль аналогичную регистру флагов (`EFlags`) основного микропроцессора. Флаги регистра `CR` управляют выполнением последующих инструкций. Разберём некоторые из них (табл. 3). Как уже говорилось ранее, сопроцессор Intel 80387 поддерживает три режима вычислений: 32-, 64- и 80-битные. Выбор точности вычислений осуществляется посредством флага `PC`, состоящего из 8-го и 9-го битов регистра управления. Два бита образуют четыре варианта значений. Ноль соответствует 32-битной точности. Двойка — 64-битной точности. Тройка — 80-битной точности. Единица зарезервирована и не используется.

При работе с дробными числами часто возникает задача округления до целого числа. Сопроцессор Intel 80387 предлагает четыре алгоритма округления. Каждому из них соответствует двухбитовое число. Для выбора алгоритма округления его номер записывается в 10-й и 11-й биты регистра управления, образующие флаг `RC`. Доступны следующие алгоритмы округления: ноль — округление к ближайшему числу, один — округление к отрицательной бесконечности, два — округление к положительной бесконечности, три — округление к нулю.

В регистре состояния интерес представляют биты 8, 9, 10 и 14. Они образуют набор условных флагов (`C0-C3`). Команды сопроцессора не изменяют значений регистра флагов `EFlags`. Вместо этого выставляются значения условных флагов регистра состояния. Их можно легко скопировать в регистр `EFlags`. Например, набор команд

```
fstsw ax
sahf
```

копирует значения C0 в CF, C2 в PF, C3 в ZF.

Номер бита	Флаг	Описание
8	PC	Первый бит управления точностью
9	PC	Второй бит управления точностью
10	RC	Первый бит управления округлением
11	RC	Второй бит управления округлением

Таблица 3. Основные поддерживаемые архитектурой Intel 80387 типы дробных чисел

Все инструкции сопроцессора начинаются с литеры `f` (от английского названия сопроцессора — floating point unit). Рассмотрим наиболее важные из них.

Инструкции для перемещения данных

fld ***источник*** — загружает на вершину стека сопроцессора дробное число. При этом число размещается на месте регистра `st(7)`, после чего стек циклически сдвигается на один шаг. Таким образом, `st(7)` превращается в `st(0)`, то есть в новую вершину стека. Источник может быть размещён в оперативной памяти или в другом регистре стека сопроцессора.

fst ***получатель*** — загружает дробное число с вершины стека сопроцессора и копирует его в получателя. Стек при этом не изменяется. Получатель может быть размещён в оперативной памяти или в другом регистре стека сопроцессора.

fstp ***получатель*** — перемещает дробное число с вершины стека сопроцессора и в получателя. Стек при этом циклически сдвигается в обратную сторону, а ставший после этого регистром `st(7)` бывший регистр `st(0)` помечается свободным. Говоря простым языком, с вершины стека удаляется одно значение. Получатель может быть размещён в оперативной памяти или в другом регистре стека сопроцессора.

Замечание. Если инструкция сопроцессора оканчивается на литеру `p`, то это означает, что помимо основного действия осуществляется удаление вершины стека. Если в конце инструкции стоит две литеры `p`, то с вершины стека удаляются два подряд идущих значения.

fldz — загружает на вершину стека число ноль. При этом ноль размещается на месте регистра `st(7)`, после чего стек циклически сдвигается

на один шаг. Таким образом, `st(7)` превращается в `st(0)`, то есть в новую вершину стека.

fld1 — загружает на вершину стека единицу. При этом единица размещается на месте регистра `st(7)`, после чего стек циклически сдвигается на один шаг. Таким образом, `st(7)` превращается в `st(0)`, то есть в новую вершину стека.

fild ***источник*** — загружает на вершину стека целое число. Число при этом размещается на месте регистра `st(7)`, после чего стек циклически сдвигается на один шаг. Таким образом, `st(7)` превращается в `st(0)`, то есть в новую вершину стека. Источник может быть размещён только в оперативной памяти.

fldpi — загружает на вершину стека приближённое значение числа π. При этом оно размещается на месте регистра `st(7)`, после чего стек циклически сдвигается на один шаг. Таким образом, `st(7)` превращается в `st(0)`, то есть в новую вершину стека.

fxch ***источник*** — меняет местами значения источника и регистра `st(0)`. Источник может быть размещён только в другом регистре стека сопроцессора. Если источник не указан, то меняются местами `st(0)` и `st(1)`.

Арифметические инструкции

fadd ***получатель***, ***источник*** — складывает значения источника и получателя. Результат записывается в получателя. Если один из операндов размещён в оперативной памяти, то получателем всегда выступает `st(0)`. В этом случае указывается только источник. Один из операндов всегда лежит в `st(0)`. Если операнды лежат в `st(0)` и `st(1)`, то аргументы можно не указывать вовсе.

faddp ***получатель***, ***источник*** — инструкция аналогична `fadd`, только помимо вычислений она дополнительно выталкивает одно значение с вершины стека.

Замечание: данная инструкция очень удобна, когда необходимо убрать после сложения «мусор». Вместо двух аргументов остаётся только результат.

fmul ***получатель***, ***источник*** — умножает значение источника на значение получателя. Результат записывается в получателя. Если один из операндов размещён в оперативной памяти, то получателем всегда выступает `st(0)`. В этом случае указывается только источник. Один из операндов всегда лежит в `st(0)`. Если операнды лежат в `st(0)` и `st(1)`, то аргументы можно не указывать вовсе.

fmulp ***получатель***, ***источник*** — инструкция аналогична `fmul`, только помимо вычислений она дополнительно выталкивает одно значение с вершины стека.

Замечание. Операции умножения и сложения обладают свойством коммутативности, то есть их аргументы можно менять местами без изменения результата. Операции вычитания и деления таким свойством не обладают. Их результат зависит от порядка следования аргументов.

fsub ***получатель, источник*** — вычитает значение источника из получателя. Результат записывается в получателя. Если один из операндов размещён в оперативной памяти, то получателем всегда выступает `st(0)`. В этом случае указывается только источник. Один из операндов всегда лежит в `st(0)`. Если операнды лежат в `st(0)` и `st(1)`, то аргументы можно не указывать вовсе.

fsubp ***получатель, источник*** — инструкция аналогична `fsub`, только помимо вычислений она дополнительно выталкивает одно значение с вершины стека.

fsubr ***получатель, источник*** — вычитает значение получателя из источника. Результат записывается в получателя. Если один из операндов размещён в оперативной памяти, то получателем всегда выступает `st(0)`. В этом случае указывается только источник. Один из операндов всегда лежит в `st(0)`. Если операнды лежат в `st(0)` и `st(1)`, то аргументы можно не указывать вовсе.

fsubrp ***получатель, источник*** — инструкция аналогична `fsubr`, только помимо вычислений она дополнительно выталкивает одно значение с вершины стека.

fdiv ***получатель, источник*** — делит значение получателя на значение источника. Результат записывается в получателя. Если один из операндов размещён в оперативной памяти, то получателем всегда выступает `st(0)`. В этом случае указывается только источник. Один из операндов всегда лежит в `st(0)`. Если операнды лежат в `st(0)` и `st(1)`, то аргументы можно не указывать вовсе.

fdivp ***получатель, источник*** — инструкция аналогична `fdiv`, только помимо вычислений она дополнительно выталкивает одно значение с вершины стека.

fdivr ***получатель, источник*** — делит значение источника на значение получателя. Результат записывается в получателя. Если один из операндов размещён в оперативной памяти, то получателем всегда выступает `st(0)`. В этом случае указывается только источник. Один из операндов всегда лежит в `st(0)`. Если операнды лежат в `st(0)` и `st(1)`, то аргументы можно не указывать вовсе.

fdivrp ***получатель, источник*** — инструкция аналогична `fdivr`, только помимо вычислений она дополнительно выталкивает одно значение с вершины стека.

fabs — вычисляет абсолютное значение (модуль) числа, лежащего на вершине стека сопроцессора (в st(0)), и записывает полученный результат на то же самое место (в st(0)).

fchs — инвертирует знак числа, лежащего на вершине стека сопроцессора — в `st(0)`, и записывает полученный результат на то же самое место — в `st(0)`.

frndint — округляет до целого (по одному из рассмотренных ранее алгоритмов округления) значение числа, лежащего на вершине стека сопроцессора — в `st(0)`, и записывает полученный результат на то же самое место — в `st(0)`.

fscale — умножает значение, лежащее в `st(0)`, на число два, возведённое в степень. Показатель степени берётся из `st(1)`, значение которого предварительно округляется в сторону нуля до целого числа.

fsqrt — вычисляет квадратный корень числа, лежащего на вершине стека сопроцессора — в `st(0)`, и записывает полученный результат на то же самое место — в `st(0)`.

Инструкции для сравнения

fcom ***источник*** — сравнивает значение, лежащее в `st(0)`, с источником. Источник может быть размещён как в оперативной памяти, так и в одном из регистров сопроцессора. Если источник не указан, то сравниваются значения `st(0)` и `st(1)`. В результате сравнения выставляются условные флаги регистра состояния `SR` (табл. 4).

Замечание. Помимо трёх основных результатов сравнения возможен ещё четвёртый — числа несравнимы. Этот вариант возможен в режиме маскировки недопустимых операций. В этом случае операции, вызывающие деление на ноль, операции, приводящие к превышению допустимых рамок представления чисел, или другие подобные ошибочные ситуации не завершаются возникновением исключения. Вместо этого формируются несравнимые числа. Например, *NaN* (от англ. not a number). Если один из операндов был таким числом, и по-прежнему включён режим маскировки, то три условных флага выставляются в единицу.

Ситуация	Флаг C0	Флаг C2	Флаг C3
Источник < st(0)	0	0	0
Источник = st(0)	0	0	1
Источник > st(0)	1	0	0
Операнды несравнимы	1	1	1

Таблица 4. Изменение условных флагов в результате применения инструкции `fcom`

fcomp ***источник*** — инструкция аналогична `fcom`, только помимо сравнения она дополнительно выталкивает одно значение с вершины стека.

fcompp — инструкция аналогична `fcom`, только помимо сравнения она дополнительно выталкивает два значение с вершины стека. Аргументов нет. Всегда сравниваются `st(0)` и `st(1)`.

Замечание. Инструкции `fcom`, `fcomp` и `fcompp` заполняют только флаги регистра состояния `SR`, но не изменяют флагов регистра `EFlags`. Поэтому для того чтобы можно было пользоваться результатами сравнения в операциях условных переходов, необходимо предварительно скопировать флаги из `SR` в `EFlags`.

fcomi ***источник*** — инструкция аналогична `fcom`, только помимо сравнения она дополнительно копирует флаги из регистра `SR` в регистр `EFlags`. Флаг `C0` переходит в `CF`, `C2` — в `PF`, а `C3` — в `ZF`.

fcomip ***источник*** — инструкция аналогична `fcomp`, только помимо сравнения и выталкивания она дополнительно копирует флаги из регистра `SR` в регистр `EFlags`. Флаг `C0` переходит в `CF`, `C2` — в `PF`, а `C3` — в `ZF`.

Управляющие инструкции

ffree ***аргумент*** — инструкция помечает указанный в качестве аргумента регистр сопроцессора как пустой.

fincstp — циклически меняет нумерацию регистрового стека сопроцессора в большую сторону по аналогии с выталкиванием.

Замечание. Полноценное выталкивание вершины стека получается только при использовании комбинации инструкций `ffree` и `fincstp`.

fdecstp — циклически меняет нумерацию регистрового стека сопроцессора в меньшую сторону.

finit — обнуление состояния сопроцессора до значения по умолчанию. Значения регистров данных не изменяются.

Трансцендентные операции

fsin — вычисляет синус угла (заданного в радианах), записанного в `st(0)`, и кладёт его на то же самое место — в `st(0)`.

fcos — вычисляет косинус угла (заданного в радианах), записанного в `st(0)`, и кладёт его на то же самое место — в `st(0)`.

fsincos — вычисляет синус и косинус угла (заданного в радианах), записанного в `st(0)`. Значение синуса кладётся на старое место — в `st(0)`. После чего в стек добавляется значение косинуса. Таким образом, стек продвигается, и значение синуса перемещается в `st(1)`. Косинус же лежит на вершине стека — в `st(0)`.

Замечание. Данная команда позволяет быстро вычислить значение котангенса угла. Достаточно дополнить её делением.

fptan — вычисляет тангенс угла (заданного в радианах), записанного в `st(0)`, и кладёт его на то же самое место — в `st(0)`. После этого в

стек добавляется единица. Таким образом, стек продвигается, и значение тангенса перемещается в `st(1)`.

Замечание. Данная команда тоже позволяет быстро вычислить значение котангенса угла. Вновь достаточно только дополнить её делением.

fpatan — вычисляет арктангенс угла (заданного в радианах). Причём аргумент арктангенса получается из значений регистров `st(0)` и `st(1)` по формуле: $\frac{\texttt{st(1)}}{\texttt{st(0)}}$. Одно из значений впоследствии выталкивается из стека, а на место второго кладётся результат. Таким образом, результат оказывается в `st(0)`.

fil2x — вычисляет произведение значения регистра `st(1)` и двоичного логарифма от значения регистра `st(0)` ($\texttt{st(1)} \times \log_2 \texttt{st(0)}$). Одно из значений впоследствии выталкивается из стека, а на место второго кладётся результат. Таким образом, результат оказывается в `st(0)`.

4.3. Вычисление объёма шара. Рассмотренный минимальный набор инструкций позволяет решать простейшие задачи. В качестве примера разберём задачу о нахождении объёма шара по заданному радиусу. Прототип функции на языке *C* выглядит следующим образом:

```
double volume(double radius);
```

Для начала вспомним формулу объёма шара:

$$V = \frac{4}{3}\pi r^3,$$

где r — радиус шара.

Теперь нам потребуется вычислить константу $\frac{4}{3}$. Сделаем это в лоб, поделив число четыре на три. В процессе будем использовать промежуточную переменную в памяти. Свободное место для размещения переменной есть на стеке. Чтобы стек не смещался, будем хранить числа в его отрицательной части. В частности, целое 4-байтовое число можно разместить по адресу `[esp - 4]`.

Важно также решить, как будет передаваться ответ вызывающему функцию процессу. Соглашение языка *C* подразумевает, что функции, тип возвращаемого значения у которых *double* или *float*, перед завершением работы размещают результат своих вычислений на вершине стека сопроцессора. То есть в `st(0)`.

Кроме того, функция не должна оставлять после себя «мусор». Стек сопроцессора должен содержать только ответ. Таким образом, необходимо позаботиться об удалении результатов всех промежуточных вычислений!

Замечание. Если стек сопроцессора не чистить, то после нескольких вызовов функции вычисления перестанут осуществляться правильно. Не

забывайте тестировать свои решения задач на нескольких подряд идущих вызовах!

Пример 3.1. Реализация функции нахождения объёма шара на языке *C*.

```
# include <math.h>

double volume(double radius)
{
      return 4.0 / 3.0 * atan(1) * 4 * radius * radius * radius;
}
```

Замечание. Функция арктангенса потребовалась для получения числа π. Для этого использовалась формула: $\pi = 4 \times \mathrm{arctg}(1)$.

Пример 3.2. Реализация функции нахождения объёма шара на языке ассемблер.

```
.intel_syntax noprefix
.globl volume
.type volume,@function

volume:
      mov dword ptr [esp - 4], 4
      fild dword ptr [esp - 4]
      mov dword ptr [esp - 4], 3
      fild dword ptr [esp - 4]
      fdivp st(1), st(0)
      fldpi
      fmulp
      fld qword ptr [esp + 4]
      fld qword ptr [esp + 4]
      fld qword ptr [esp + 4]
      fmulp
      fmulp
      fmulp
      ret
```

Пример 3.3. Тестовая программа на языке *C*, позволяющая продемонстрировать использование реализованных функций.

```
# include <stdio.h>
```

```
extern double volume(double);

int main(void)
{
      double r = 5;
      printf("%g\n", volume(r));
      return 0;
}
```

4.4. Задачи для самостоятельного решения. Напишите на языках *C* и Ассемблер функции, описание и прототип которых приведены ниже.

1. `double logxy(double x, double y)`
   Функция вычисляет $\log_y x$.
2. `double dist(double xa, double ya,`
   `double xb, double yb)`
   Функция вычисляет расстояние между двумя точками $A(xa, ya)$ и $B(xb, yb)$ на плоскости.
3. `int isright(double xa, double ya,`
   `double xb, double yb,`
   `double xc, double yc)`
   Функция проверяет, является ли треугольник с вершинами $A(xa, ya)$, $B(xb, yb)$ и $C(xc, yc)$ правильным или нет. Возвращает единицу, если треугольник правильный, и ноль иначе. Проверку правильности осуществляйте в рамках некоторой точности. Задаётся некоторая маленькая величина ε, с которой по абсолютному значению сравнивается разность между длинами сторон треугольника. Если разность меньше, чем ε, стороны считаются равными.
4. `int roots(double a, double b, double c,`
   `double* x1, double* x2)`
   Функция находит корни уравнения $ax^2 + bx + c = 0$. Константы a, b и c могут быть любыми. Функция возвращает количество корней. Если корней бесконечно много, то возвращается число -1. В переменные по указателям $x1$ и $x2$ размещаются значения корней (если они есть).
5. `double cube_root(double x)`
   Функция вычисляет $\sqrt[3]{x}$.
6. `double square(double a, double num)`
   Функция вычисляет по заданным длине a и числу сторон num площадь соответствующего правильного многоугольника.

7. `double square(double xa, double ya,`
   `double xb, double yb,`
   `double xc, double yc)`
   Функция вычисляет площадь треугольника с вершинами $A(xa, ya)$, $B(xb, yb)$ и $C(xc, yc)$.
8. `double square(double a)`
   Функция вычисляет площадь окружности, вписанной в правильный треугольник со сторонами, длина которых равна a.
9. `double square(double angle, double r)`
   Функция вычисляет площадь сектора в $angle$ радиан, вырезанного из окружности радиусом r.
10. `double is_triangle(double ax, double ay,`
    `double bx, double by,`
    `double cx, double cy)`
    Функция проверяет, является ли треугольник с вершинами $A(xa, ya)$, $B(xb, yb)$ и $C(xc, yc)$ нормальным треугольником, или он вырожден в точку или отрезок. Возвращает единицу, если треугольник не вырожден, и ноль иначе.
11. `double cube_in_four_dimensions(double a)`
    Функция вычисляет площадь поверхности тессеракта (аналога куба в четырёхмерном пространстве) по заданной длине ребра a.

5. Дополнительные задачи для самостоятельного решения

1. Напишите на языках *C* и Ассемблер функции, описание и прототип которых приведены ниже.
   a) `int* find(int* source_array, int* sub_array, int source_size, int sub_size);`
   Функция находит вхождения подмассива *sub_array* в массиве *source_array*. В качестве результата возвращается указатель на первое найденное вхождение. Размеры массивов *sub_array* и *source_aray* указаны в переменных *sub_size* и *source_size*, соответственно.
   b) `double* intersect(double* array1, double* array2, int size);`
   Функция находит пересечение двух массивов: *array*1 и *array*2. Результирующий массив создаётся посредством вызова стандартного метода malloc. Размер обоих входных массивов одинаковый и равен *size* элементов. Результирующий массив создаётся таким же. Лишние элементы заполняются нулями.
   c) `double volume(double r1, double r2, double r3);`
   Функция вычисляет объём эллипсоида с радиусами *r*1, *r*2 и *r*3.
   d) `double volume(double r1, double r2);`
   Функция вычисляет объём тороида с радиусами внешней и внутренней окружностей «бублика» *r*1 и *r*2.
   e) `double sum(int n, int size);`
   Функция вычисляет сумму ряда $\frac{1}{n^2}$ для всех $n = 1 \ldots size$.
   f) `int det(int** matrix);`
   Функция вычисляет определитель матрицы *matrix* размером $3 \times \times 3$.
   g) `double sum(double x0, double d, int n);`
   Функция вычисляет сумму элементов геометрической прогрессии $x_i = \frac{x_0}{d^i}$ для всех $i = 1 \ldots n$.
2. Реализуйте самостоятельно на языках *C* и Ассемблер функции стандартной библиотеки `string.h`. Информацию о функциях можно посмотреть, выполнив в терминале операционной системы *Linux* команду `man имя функции`, или найти в [3].
   a) `strcmp` — сравнение двух строк .
   b) `strncmp` — сравнение *n* символов из двух строк.
   c) `strcpy` — копирование строки.
   d) `memmove` — перемещение области памяти.

e) `strlen` — вычисление длины строки.
f) `strncpy` — копировние *n* символов строки.
g) `memcpy` — копирование области памяти.
h) `memset` — присвоение заданного значения всем байтам некоторой области памяти.
i) `strchr` — поиск символа в строке.
j) `strcat` — склеивание строк.
k) `strncat` — приклеивание к строке *n* символов другой строки.

Список литературы и интернет-ресурсов

1. *Зубков С.В.* Assembler для DOS, Windows и UNIX. — СПб.: Питер-Маркет, 2004.

2. Архитектура и программирование микропроцессора Intel 80386. — М., 1992.

3. *Керниган Б.*, *Ритчи Д.* Язык программирования С. —М.: Вильямс, 2010.

4. `http://microsym.com/editor/assets/386intel.pdf` Intel 80386 Programmer's Reference Manual 1986

Учебное издание

Радыгин Виктор Юрьевич

АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ И КОМПЬЮТЕРНЫХ СЕТЕЙ

Учебно-методическое пособие

Редактор К.В. Шмат

Санитарно-эпидемиологическое заключение
№77.99.60.953.Д.006314.05.07 от 31.05.2007

Подписано в печать 28.02.11
Формат бумаги 60x84/16. Изд. № 10/11-у
Усл.печ.л. 3. Уч.-изд.л. 3,25. Тираж 100. Заказ № 10

Издательство МГИУ, 115280, Москва, Автозаводская, 16
www.izdat.msiu.ru; e-mail: izdat@msiu.ru; тел. (495) 620-39-90

По вопросам приобретения продукции
издательства МГИУ обращаться по адресу:
115280, Москва, Автозаводская ул., 16
www.izdat.msiu.ru; e-mail: izdat@msiu.ru; тел. (495) 620-39-90

Отпечатано в типографии издательства МГИУ