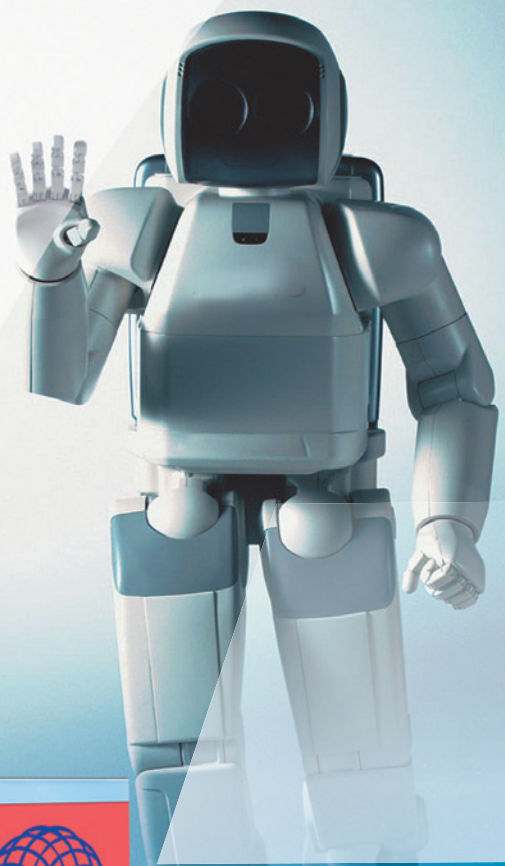
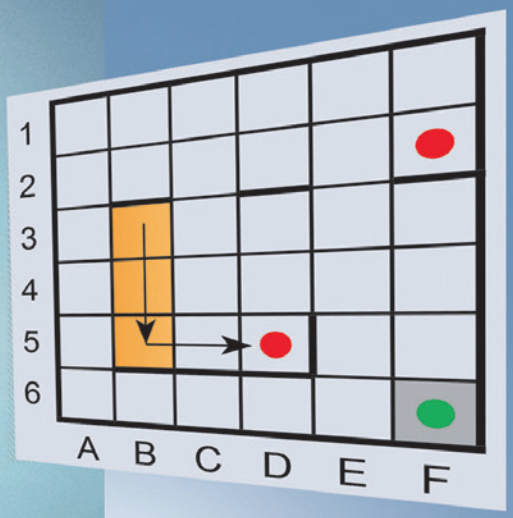
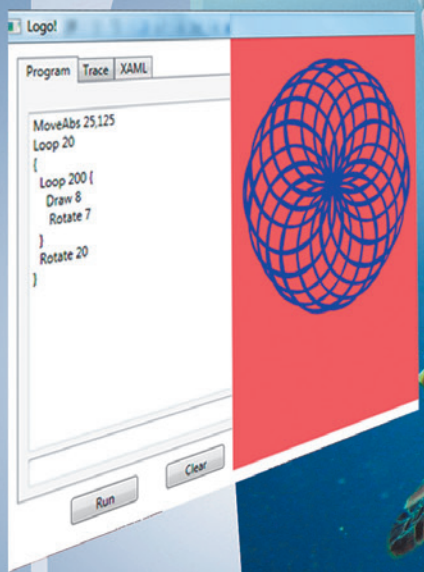
№7 Издание основано в 1995 г. inf.1september.ru
УЧЕБНО-МЕТОДИЧЕСКИЙ ЖУРНАЛ ДЛЯ УЧИТЕЛЕЙ ИНФОРМАТИКИ

ИНФОРМАТИКА







внутри номера CD и код доступа к электронной версии

издательский дом 1september.ru Первое сентября | август 2012
ИНФОРМАТИКА Подписка: «Роспечать» – 32291 (бумажная версия), 19179 (электронная); «Почта России» – 79066 (бумажная версия), 12684 (электронная)

НА ОБЛОЖКЕ

Даже те, кто твердо убежден в “ненастоящести” учебных исполнителей и уверен, что начинать учить алгоритмизации надо на “боевых” современных языках программирования, иногда не удержатся да и напишут какую-нибудь красивую “рекурсивную штучку” на Лого или особо хитрым способом обойдут лабиринт Роботом. Потому что интересно! И детям, и нам. Да не жили бы эти учебные среды так долго (и счастливо!), если бы не были востребованными и продуктивными.

В НОМЕРЕ

НА ДИСКЕ

ЭЛЕКТРОННЫЕ МАТЕРИАЛЫ:

- Программа “Учебный компьютер ЛамПанель”
- Исполнители и методические материалы к статье “Азбука Роботландии. Информация”
- Исходные файлы к статье “Ряды”
- Презентации к статьям номера

ИНФОРМАТИКА

ПОДПИСНЫЕ ИНДЕКСЫ: по каталогу “Роспечати”: 32291 (бумажная версия), 19179 (электронная версия); “Почта России”: 79066 (бумажная версия), 12684 (электронная версия)

http://inf.1september.ru

Учебно-методический журнал для учителей информатики
Основан в 1995 г.
Выходит один раз в месяц

РЕДАКЦИЯ:
гл. редактор С.Л. Островский
редакторы
Е.В. Андреева,
Д.М. Златопольский (редактор вкладки “В мир информатики”)
Дизайн макета И.Е. Лукьянов
верстка Н.И. Пронская
корректор Е.Л. Володина
секретарь Н.П. Медведева
Фото: фотобанк Shutterstock
Журнал распространяется по подписке
Цена свободная
Тираж 25 189 экз.
Тел. редакции: (499) 249-48-96
E-mail: inf@1september.ru
http://inf.1september.ru

ИЗДАТЕЛЬСКИЙ ДОМ “ПЕРВОЕ СЕНТЯБРЯ”
Главный редактор: Артем Соловейчик (генеральный директор)
Коммерческая деятельность: Константин Шмарковский (финансовый директор)
Развитие, IT и координация проектов: Сергей Островский (исполнительный директор)
Реклама, конференции и техническое обеспечение Издательского дома: Павел Кузнецов
Производство: Станислав Савельев
Административно-хозяйственное обеспечение: Андрей Ушков
Главный художник: Иван Лукьянов
Педагогический университет: Валерия Арсланьян (ректор)

ГАЗЕТА ИЗДАТЕЛЬСКОГО ДОМА
Первое сентября – Е.Бирюкова

ЖУРНАЛЫ ИЗДАТЕЛЬСКОГО ДОМА
Английский язык – А.Громушкина
Библиотека в школе – О.Громова
Биология – Н.Иванова
География – О.Коротова
Дошкольное образование – Д.Тюттерин
Здоровье детей – Н.Сёмина
Информатика – С.Островский
Искусство – М.Сартан
История – А.Савельев
Классное руководство и воспитание школьников – М.Битянова
Литература – С.Волков
Математика – Л.Рослова
Начальная школа – М.Соловейчик
Немецкий язык – М.Бузоева
Русский язык – Л.Гончар
Спорт в школе – О.Леонтьева
Управление школой – Е.Рачевский
Физика – Н.Козлова
Французский язык – Г.Чесновицкая
Химия – О.Блохина
Школьный психолог – И.Вачков

УЧРЕДИТЕЛЬ: ООО “ЧИСТЫЕ ПРУДЫ”

Зарегистрировано ПИ № ФС77-44341 от 22.03.2011 в Министерстве РФ по делам печати
Подписано в печать: по графику 12.05.2012, фактически 12.05.2012
Заказ №
Отпечатано в ОАО “Первая образцовая типография” Филиал “Чеховский печатный двор”: ул. Полиграфистов, д. 1, Московская обл., г. Чехов, 142300
Сайт: www.chpk.ru,
E-mail: salas@chpk.ru,
Факс: 8(496) 726-54-10, 8(495) 988-63-87

АДРЕС ИЗДАТЕЛЯ: ул. Киевская, д. 24, Москва, 121165
Тел./факс: (499) 249-31-38
Отдел рекламы: (499) 249-98-70
http://1september.ru

ИЗДАТЕЛЬСКАЯ ПОДПИСКА:
Телефон: (499) 249-47-58
E-mail: podpiska@1september.ru

Учебный компьютер “ЛамПанель”: практикум

К.Ю. Поляков,
д. т. н., Санкт-Петербург

Зачем это нужно?

Большинство школьников старших классов знают, что процессор “понимает” только один язык — язык машинных команд, которые представляют собой числовые коды. Писать программы в кодах очень трудно, поэтому программисты, как правило, составляют программы на алгоритмических языках (языках высокого уровня), которые затем с помощью специальных программ — трансляторов переводятся в коды, выполняемые процессором. Более того, “уровень” учебных языков все больше повышается, и сортировка списка во многих современных языках программирования выполняется одной командой.

Казалось бы, на этом можно и остановиться. Но любознательная человеческая натура неминуемо увидит здесь “белое пятно”: программируя только на языках высокого уровня, мы не понимаем, как же все-таки работает процессор, как он умудряется понимать двоичные коды команд, выполнять их, организовывать ветвления и циклы, выделять память для переменных и вызывать процедуры.

А надо ли в этом разбираться, ведь и так все работает? По сути, этот вопрос аналогичен другому вопросу: “А зачем изучать теорию, если я хорошо умею работать с сенсорным экраном *iPad*?” По мнению автора, для тех, кто хочет стать специалистом в области информационных технологий, необходимо глубокое понимание происходящих процессов. Практика показывает, что только в этом случае удается находить решения в нестандартных ситуациях, не описанных в руководствах, когда “мастера по нажатию на кнопки” оказываются бессильны. Таким образом, тот, кто понимает, как работает компьютер, может более успешно и эффективно решать сложные задачи.

Как же понять и изучить работу компьютера? Конечно, можно экспериментировать с реальным компьютером, используя программы-отладчики, которые позволяют ввести в память коды машинных команд и данных, а потом запустить программу на выполнение, начиная с некоторого адреса [1]. Но отладчикам всегда не хватало наглядности, поэтому

были разработаны учебные модели компьютеров, которые имеют простую (в сравнении с реальными процессорами) систему команд, но при этом позволяют наглядно показать все этапы выполнения программы [2–4]. Иногда строят и полные модели (*эмуляторы*) существующих процессоров [5].

В этой статье описывается еще одна учебная модель компьютера — программа *ЛамПанель* (это название образовано от слов *ламповая панель*). В отличие от других программ этого класса в качестве устройства вывода используется панель (матрица) лампочек размером 8 на 16, причем каждой лампочкой можно управлять независимо от других. Это приближает такую модель к реальным задачам управления ламповыми табло, например, в системах освещения, рекламных установках и информационных табло.

Возможности программы ориентированы на преподавание профильного курса информатики в школе. Основная форма изучения — практические работы. С помощью программы *ЛамПанель* можно изучать

- представление знаковых и беззнаковых целых чисел в памяти компьютера;
- принципы хранения программ и данных в компьютерах с архитектурой фон Неймана;
- алгоритм автоматической работы процессора;
- организацию ветвлений и циклов;
- принципы работы стека;
- организацию вызовов подпрограмм.

Непосредственный предшественник программы *ЛамПанель* — учебная модель компьютера “Е-97”, разработанная Е.А. Ереминым [3]. *ЛамПанель* имеет аналогичную архитектуру и использует похожую систему команд.

Для лучшего понимания материала статьи желательно предварительно познакомиться с основами компьютерной арифметики, например, по работе [6].

Как устроен учебный компьютер?

Процессор обрабатывает данные, используя сверхбыстродействующие ячейки собственной памяти — *регистры*. Процессор учебного компьютера *ЛамПанель* имеет только четыре 16-битных *регистра общего назначения*, которые называются **R0**, **R1**, **R2** и **R3**. В области 1 на *рис.* 1 вы видите двоичные значения этих регистров (они показаны черным цветом), шестнадцатеричные (синий цвет) и десятичные, без учета знака (зеленый цвет) и со знаком (коричневый цвет).

Ниже показаны еще три служебных регистра (**PC**, **SP** и **PS**), о которых мы поговорим позже.

Область 2 — это устройство вывода — ламповая панель, состоящая из 8 рядов, в каждом из которых по 16 лампочек. Состояние лампочек каждого ряда задается 16-битным числом, записанным в *порт* — регистр контроллера ламповой панели. Слева от каждого ряда записан его номер, точнее, номер порта (нумерация начинается с нуля). Единица в каком-то бите означает зажженную лампочку, а 0 — погашенную. Старший бит управляет самой левой лампочкой в ряду, младший — самой правой. Справа от каждого ряда показано значение, которое записано в порт (в шестнадцатеричной системе счисления).

Область 3 — это текстовый редактор, в котором набирается программа для процессора на специальном языке — *языке ассемблера*. Каждая команда этого языка соответствует одной машинной команде, но она записывается в символьном виде, а не как числовой код. Например, команда “*скопировать данные из регистра* **R2** *в регистр* **R3**”, имеющая шестнадцатеричный код 0123_{16}, может быть записана в виде

```
MOV R2, R3
```

где **MOV** — это название команды “*скопировать данные*” (сокращение от англ. *move* — переместить).

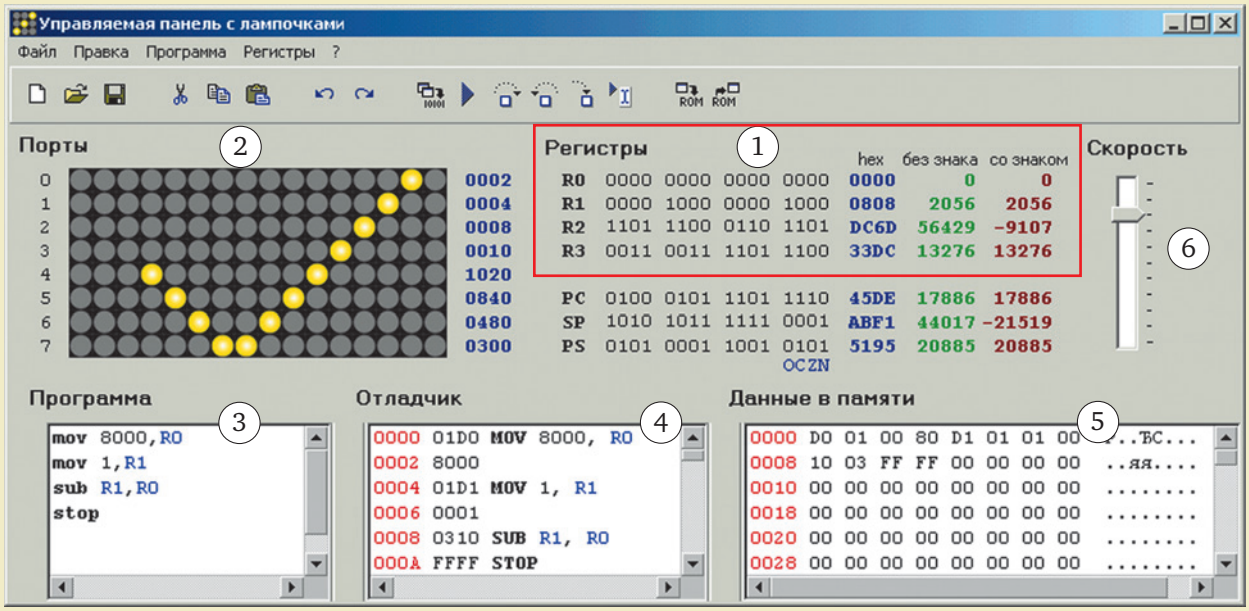


Рис. 1

Очевидно, что символьная запись команды значительно более понятна, чем число 0123_{16}. На псевдокоде эту операцию можно записать так: **R3 := R2**.

Команда **MOV** может не только выполнять копирование данных из регистра в регистр, но и присваивать новое значение регистру. Например, команда

```
MOV 12, R0
```

запишет число 12_{16} = 18 в регистр **R0** (все числа в программе записываются в шестнадцатеричной системе счисления). Эта команда кодируется в виде двух 16-битных кодов, которые записываются в память последовательно: сначала код команды $01D0_{16}$, а затем — число, пересылаемое в регистр:

```
01D0 MOV 12, R0  ; R0 := 12
0012
```

Точка с запятой означает начало *комментария*: ассемблер игнорирует (не обрабатывает) все символы в строке, расположенные после точки с запятой.

Нажав на клавишу F1, можно посмотреть описание всех команд языка ассемблера.

Чтобы программа остановилась, процессор должен выполнить команду **STOP**. Таким образом, простейшая программа состоит из одной команды **STOP**.

Каждая команда программы записывается в отдельной строке. Поэтому полная программа, которая копирует содержимое регистра **R2** в регистр **R3**, будет выглядеть так:

```
MOV R2, R3
STOP
```

При нажатии сочетания клавиш Ctrl + F9 запускается *ассемблер* (англ. *assembler* — сборщик) — программа, которая переводит программу на языке ассемблера в машинные коды, понятные процессору. Результат работы ассемблера — программа в машинных кодах — записывается в оперативную память (область 5 на *рис.* 1), размер которой в программе *ЛамПанель* составляет 256 байт. При нажатии на клавишу F9 программа начинает выполняться, скорость ее выполнения регулируется ползунком в правой части окна программы (область 6 на *рис.* 1).

Клавиша F8 позволяет выполнять программу по шагам — после выполнения очередной команды процессор останавливается и ждет следующего нажатия на клавишу F8 (или на клавишу F9, чтобы дальше выполнить программу без остановки). Во время паузы можно посмотреть, какие данные находятся в регистрах и в памяти; этот режим служит для отладки программ.

Компьютерная арифметика

Особенности компьютерной арифметики на уровне профильного школьного курса информатики подробно рассмотрены в статье [6], поэтому здесь мы не будем повторяться, а расскажем только о возможностях программы *ЛамПанель*.

Для того чтобы сложить два числа, применяют команду **ADD** (от англ. *add* — сложить). Например, команда

```
ADD 15, R0 ; R0 := R0 + 15
```

добавляет число 15_{16} = 23 к регистру **R0**. Можно добавить значение одного регистра к значению другого:

```
ADD R2, R3 ; R3 := R3 + R2
```

Существуют аналогичные команды, выполняющие другие арифметические действия:

SUB — вычитание (от англ. *subtract* — вычесть);

MUL — умножение (от англ. *multiply* — умножить);

DIV — целочисленное деление (от англ. *divide* — делить), остаток отбрасывается.

Команда **NOT** выполняет *инверсию* всех битов регистра, то есть меняет все нули на единицы, а единицы — на нули. Например, команда

```
NOT R0
```

выполнит инверсию регистра **R0**. С помощью этой команды можно, например, получить дополнительный двоичный код числа, находящегося в регистре **R0**, по классическому алгоритму [6]:

```
NOT R0
ADD 1, R0
```

Если первоначально в **R0** было записано число 21 = 15_{16}, то после выполнения первой команды (инверсии битов) значение регистра будет равно $FFEA_{16}$, а после добавления единицы — число $FFEB_{16}$, которое совпадает с дополнительным кодом числа (–21). Эти преобразования можно наглядно увидеть с помощью отладчика, запуская программу в пошаговом режиме (клавиша F8).

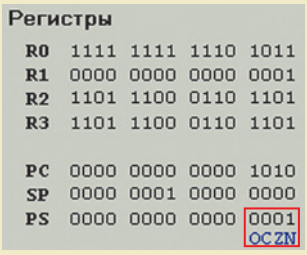


Рис. 2

Кроме регистров общего назначения, с которыми мы уже работали, в процессоре есть служебные регистры. Один из них — *регистр состояния процессора* **PS** (англ. *Processor State register* — регистр состояния процессора), биты которого называются также *флагами состояния* (флаг может быть "спущен" или "поднят", эти состояния обозначаются как 0 и 1 соответственно).

Фактически используются только четыре младших бита регистра **PS** (см. *рис.* 2), которые показывают, какой результат был получен в результате последней операции:

- **бит O** (от англ. *overflow* — переполнение) установлен (равен 1), если произошло **переполнение** разрядной сетки [6], то есть результат вычислений неверный; в остальных случаях бит O сброшен (равен 0);
- **бит C** (от англ. *carry* — перенос) установлен, если произошел **перенос** бита из разрядной сетки [6]; в остальных случаях сброшен;

• **бит Z** (от англ. *zero* — ноль) установлен, если результат последней операции — ноль; в остальных случаях сброшен;

• **бит N** (от англ. *negative* — отрицательный) установлен, если результат последней операции отрицательный; в остальных случаях сброшен.

Эти биты могут учитываться при выполнении ветвлений. Например, для организации цикла используют команду перехода

JNZ *метка*

Если результат предыдущей операции НЕ равен нулю, то происходит переход на указанную метку. Вот пример программы, которая вычисляет сумму натуральных чисел от 1 до 5 в регистре **R0**:

```
MOV 0, R0    ; начальное значение суммы
MOV 5, R1    ; количество шагов цикла
m:           ; метка обозначает начало цикла
  ADD R1, R0 ; R0 := R0 + R1
  SUB 1, R1
  ; R1 := R1 - 1 — оставшееся число шагов
JNZ m        ; переход на метку "m",
             ; если получился не ноль
STOP
```

Здесь в каждой строчке после точки с запятой записан комментарий, объясняющий ее действие. Давайте проследим, как выполняется эта программа:

Команда	R0	R1	бит Z	переход
MOV 0,R0	0		1	
MOV 5,R1		5	0	
ADD R1,R0	5		0	
SUB 1,R1		4	0	
JNZ m			0	да
ADD R1,R0	9		0	
SUB 1,R1		3	0	
JNZ m			0	да
ADD R1,R0	12		0	
SUB 1,R1		2	0	
JNZ m			0	да
ADD R1,R0	14		0	
SUB 1,R1		1	0	
JNZ m			0	да
ADD R1,R0	15		0	
SUB 1,R1		0	1	
JNZ m			1	нет
STOP				

Цикл заканчивается, потому что бит **Z** равен 1 (результат последней операции вычитания — ноль) и перехода по команде **JNZ** не происходит.

Кроме команды **JNZ**, существуют и другие команды перехода:

JMP *метка* — безусловный переход;

JGE *метка* — переход, если результат больше или равен нулю;

JL *метка* — если результат меньше нуля;

JZ *метка* — если результат равен нулю;

JLE *метка* — если результат меньше или равен нулю;

JG *метка* — если результат больше нуля.

Для проверки условий без изменения значений регистров можно применять команду сравнения **CMP** (от англ. *compare* — сравнить). Сравнивать можно число с регистром:

```
CMP 12, R0          ; сравнить 12_16 и R0
```

или регистр с регистром

```
CMP R2, R3          ; сравнить R2 и R3
```

Смысл сравнения состоит в том, чтобы установить *флаги* — биты регистра состояния **PS** — по результату: разности второго и первого операндов. При этом значения регистров не изменяются. Например, вторая приведенная команда сравнения вычисляет разность **R3-R2** и устанавливает флаги состояния по этой разности (например, если эти регистры равны, будет установлен бит **Z**).

В программе *ЛамПанель* можно использовать битовые логические операции [6]: “НЕ” (уже знакомая нам команда **NOT**), “И” (команда **AND**), “ИЛИ” (команда **OR**) и “исключающее ИЛИ” (команда **XOR**). В последних трех командах после названия команды сначала указывается маска, а затем через запятую — регистр, к которому применяется логическая операция. Например, команда

```
AND FF,R0
```

обнуляет старшие 8 бит (старший байт) регистра **R0**. Маска может находиться в регистре, например, последовательность команд

```
MOV FF,R1
OR R1,R0
```

устанавливает в единицу 8 младших бит регистра **R0**, а остальные оставляет без изменений.

Для выполнения сдвигов (см. [6]) используются следующие команды:

```
SHL 1,R0 ; логический сдвиг
         ; влево на 1 бит
SHR 2,R0 ; логический сдвиг вправо
         ; на 2 бита
SAR 1,R0 ; арифметический сдвиг
         ; вправо на 1 бит
ROL 2,R0 ; циклический сдвиг влево
         ; на 2 бита
ROR 3,R0 ; циклический сдвиг вправо
         ; на 3 бита
```

Конечно, сдвиг может применяться к любому регистру общего назначения, а не только к **R0**.

Практикум

1. Используя команду **MOV**, напишите программу, которая заполнит регистры так, как на рисунке. Не забудьте закончить программу командой **STOP**.

```
Регистры
 R0  1111 0000 0000 0000
 R1  1111 1111 0000 0000
 R2  1111 1111 1111 0000
 R3  1111 1111 1111 1111
```

Запишите, какие десятичные числа были только что записаны в регистры:

Регистр	Десятичные значения	
	без учета знака	с учетом знака
R0		
R1		
R2		
R3		

2. Выполните программу

```
SUB 1, R0
NOT R0
STOP
```

при различных начальных значениях регистра **R0** и запишите десятичные значения, полученные в **R0** после выполнения программы:

До	После	
	без учета знака	с учетом знака
5		
10		
25		

Какую операцию выполняет этот алгоритм?

3. Используя программу *ЛамПанель*, вычислите арифметические выражения и запишите результаты в таблицу. Объясните полученные результаты.

Выражение	Результат	
	без учета знака	с учетом знака
65530 + 9		
32760 + 9		
8 - 10		

Подсказка: **65535 = FFFF**$_{16}$, **32767 = 7FFF**$_{16}$

4. Вычислите приведенные выражения с помощью программы. Запишите в таблицу результаты, значения знакового (старшего) бита полученного числа и битов состояния:

Выражение	Результат		Знаковый бит	Биты состояния			
	без учета знака	с учетом знака		**O**	**C**	**Z**	**N**
32760 + 32752							
-32760 - 32752							
256 - 256							

5. С помощью программы, приведенной в теоретической части, вычислите сумму натуральных чисел от 1 до 100.

6. Напишите программу, которая вычисляет значение факториала — произведения всех натуральных чисел от 1 до заданного числа. Например, факториал числа 5 равен 5! = 1 · 2 · 3 · 4 · 5. С помощью программы заполните таблицу:

N	**N!**	
	без учета знака	с учетом знака
5		
6		
7		
8		
9		

Объясните полученные результаты.

1. Напишите программу, которая решает следующую задачу, используя логические операции:

В регистрах **R1**, **R2** *и* **R3** *записаны коды трех десятичных цифр, составляющих трехзначное число (соответственно сотни, десятки и единицы). Построить в регистре* **R0** *это число. Например, если* **R1**=31_{16}, **R2**=32_{16} *и* **R3**=33_{16}, *в регистре* **R0** *должно получиться десятичное число 123.*

2. Используя программу *ЛамПанель*, определите и запишите в таблицу значения регистра **R0** после выполнения каждой из следующих команд, которые выполняются последовательно:

	Команда	**R0**
1	**MOV 1234, R0**	
2	**XOR ABCD, R0**	
3	**XOR ABCD, R0**	

Ответьте на вопросы:

- как изменится результат выполнения программы, если в команде 1 записать в **R0** другое число?
- как изменится результат выполнения программы, если в командах 2 и 3 заменить маску на другую, например, на CB24$_{16}$?
- как изменится результат выполнения программы, если маску в команде 2 изменить, а маску в команде 3 не менять?

3. Запишите в таблицу десятичные числа, которые будут получены в регистре **R0** после выполнения каждой команды этой программы при разных начальных значениях **R0** (две команды выполняются последовательно одна за другой):

Начальное значение	**255**	**254**	**252**	**-255**	**-254**	**-252**
SHR 2,R0						
SHL 2,R0						

В каком случае последовательное выполнение этих двух команд не изменяет данные?

4. Напишите программу, которая решает следующую задачу, используя логические операции и сдвиги:

При кодировании цвета используются 4-битные значения составляющих R (красная), G (зеленая) и B (синяя). Коды этих составляющих записаны в регистрах **R1**, **R2** *и* **R3**. *Построить в регистре* **R0** *полный код цвета. Например, если* **R1** = A_{16}, **R2** = B_{16} *и* **R3** = C_{16}, *в регистре* **R0** *должно получиться число* ABC_{16}.

5. Напишите программу, которая умножает число в регистре **R0** на 10, не применяя команду умножения. Используйте арифметические операции и сдвиги.

Программа и данные

Итак, мы научились работать с регистрами, используя арифметические и логические операции. Теперь пришло время разобраться, как компьютер работает с памятью и как организуется автоматическое выполнение программы.

Память (см. область 5 на *рис.* 1) разбита на ячейки размером 1 байт (8 бит). Значение каждой ячейки записывается в виде двух шестнадцатеричных цифр — каждая из них представляет ровно четыре бита.

Каждая строчка в окне *Данные в памяти* содержит значения 8 байтов памяти; число слева, выделенное красным цветом, — это *адрес* (номер) первой ячейки, показанной в этой строке. Справа от шестнадцатеричных кодов выведена символьная строка из 8 символов — те же данные, только представленные как символы.

Данные можно записывать в память напрямую, используя команду **DATA**, например, можно набрать такую программу:

```
DATA 3132
DATA FFFF
```

Если теперь нажать клавиши Ctrl + F9, происходит *ассемблирование* (“сборка”) — перевод программы в машинные коды, и эти коды записываются в память (см. *рис.* 3).

Посмотрим на окно отладчика. В память записаны два 16-битных слова (4 байта), 3132_{16} и $FFFF_{16}$, причем эти слова процессор распознал как две команды:

```
MOV R3,R2
STOP
```

Такой обратный перевод из числовых кодов команд в их символьное обозначение называется *дисассемблирование* (обратное ассемблирование, “разборка”).

Любая машинная команда в компьютере *ЛампПанель* состоит из целого числа 16-битных слов, то есть из четного числа байтов. Поэтому в окне отладчика нумерация содержит только четные адреса ячеек памяти (команда не может начинаться с ячейки, имеющей нечетный адрес).

Теперь посмотрим на окно *Данные в памяти* (см. *рис.* 3). Видим, что байты 16-битного слова 3132_{16} расположены в памяти “наоборот” — сначала младший байт 32_{16}, а затем — старший 31_{16}. Кроме того, в правой части окна видно, что эти коды соответствуют символам “21яя”. Все специальные коды (не соответствующие каким-то принятым изображениям символов) обозначены точками. Таким образом, компьютер, основанный на архитектуре фон Неймана, не может самостоятельно различить, где данные, а где команды.

Выполним программу в пошаговом режиме. После первого нажатия на клавишу F8 в регистр **PC** (англ. *Program Counter* — программный счетчик) записывается стартовый адрес 0 (*рис.* 4), с которо-

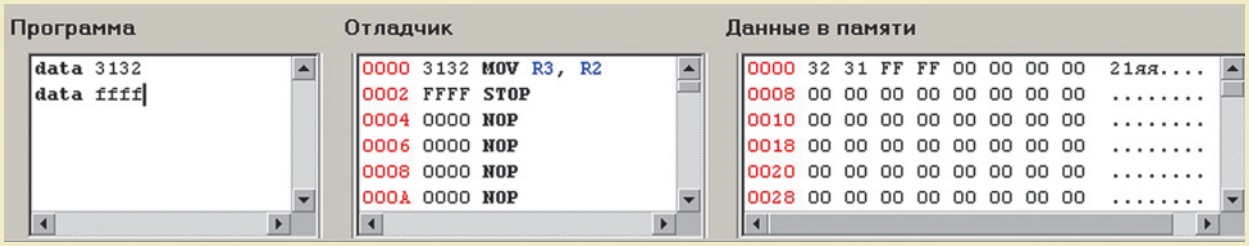


Рис. 3

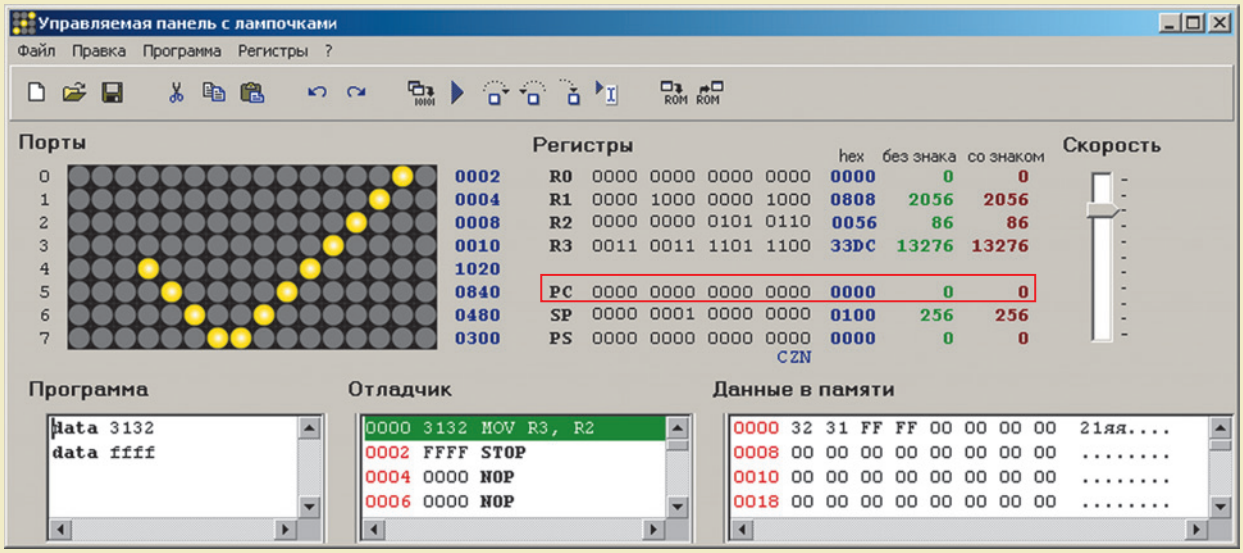


Рис. 4

го начинается выполнение программы. В окне *Отладчик* зеленым цветом выделена первая команда. Она еще не выполнялась, но будет выполнена при повторном нажатии F8. При этом регистр **PC** будет указывать на начало следующей команды (которая еще не выполнялась).

Таким образом:

• регистр **PC** всегда содержит адрес следующей команды;

• процессор воспринимает байты, расположенные по этому адресу, как код команды (а не как данные);

• программа всегда начинает выполняться с некоторого известного адреса, который "вшит" в компьютер и автоматически заносится в регистр **PC** при его включении;

• программа останавливается, когда будет выполнена команда **STOP** с кодом $FFFF_{16}$.

Заметим, что команды, содержащие числовые данные, могут занимать в памяти два 16-битных слова (см. *рис.* 5).

Для того чтобы обрабатывать данные из оперативной памяти, процессор должен загрузить их в регистры. Поскольку программа и данные расположены в одной области памяти, размещать данные можно сразу после команды **STOP**:

```
...
STOP
D:
  DATA 1234
```

Метка **D** нужна для того, чтобы удобно было загружать адрес блока данных в регистр, например, так:

```
MOV @D, R0 ; загрузить адрес метки D в R0
```

Можно считать, что **D** — это переменная программы. После этого легко загрузить в регистр данные из памяти:

```
MOV (R0),R1 ; загрузить в R1 данные,
            ; адрес которых записан в R0
```

Запись **(R0)** означает "значение, находящееся в памяти по адресу, записанному в **R0**", это так называемый *косвенный способ адресации*, когда в регистре находится адрес данных, а не значение. Аналогичной командой можно изменить содержимое ячейки памяти:

```
MOV R2,(R0) ; записать данные из R2
            ; в ячейку, адрес которой
            ; записан в R0
```

Заметим, что так можно сразу обратиться к любой ячейке памяти, поэтому такой вид памяти называется *памятью с произвольным доступом* (англ. *RAM = random access memory*).

Как вы знаете, минимальная ячейка памяти, имеющая собственный адрес, называется байтом. В современных компьютерах 1 байт состоит из 8 битов. Поэтому компьютер должен иметь возможность работать не только с 16-битными словами, но и с отдельными байтами. Покажем, как это происходит на примере задач обработки символьных строк, записанных в однобайтной кодировке.

Для того чтобы разместить символьную строку в памяти, используется команда **DATA**:

```
D:
  DATA "ABCDEFG"
```

Символы строки **"ABCDEFG"** записываются в память последовательно, начиная с первого.

Теперь запишем адрес строки в какой-нибудь регистр, например, в **R0**:

```
MOV @D,R0
```

Чтобы работать с отдельными байтами, используют байтовые версии команд, которые заканчиваются на латинскую букву B. Например, байтовый вариант команды **MOV** называется **MOVB**. Команда

```
MOVB (R0), R2
```

загружает один байт из памяти (расположенный по адресу, который записан в **R0**) в регистр **R2**. Полный список команд можно посмотреть в справочной системе, нажав на клавишу F1.

Рассмотрим такую задачу — преобразовать все заглавные латинские буквы в строчные. Для этого нужно посмотреть, чем отличаются двоичные коды заглавных и строчных букв:

```
A: 01000001   C: 01000011
a: 01100001   c: 01100011

B: 01000010   D: 01000100
b: 01100010   d: 01100100
```

Оказывается, коды заглавных и соответствующих строчных букв отличаются одним битом (этот бит выделен красным цветом в таблице). Поэтому, для того чтобы получить из заглавной буквы строчную букву, нужно установить 5-й бит (биты нумеруются справа налево, начиная с нулевого). Для этого можно, например, использовать логическую операцию "ИЛИ" с маской 0020_{16}, в которой 5-й бит установлен, а остальные — сброшены:

```
OR 20,R2
```

Затем нужно записать результат обратно в память, по адресу, находящемуся в **R0**:

```
MOVB R2,(R0)
```

Для перехода к следующему символу просто увеличиваем **R0** на единицу, сдвигаясь к следующему байту:

```
ADD 1,R0
```

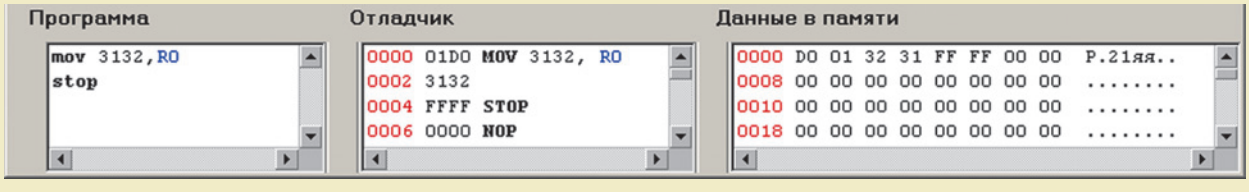


Рис. 5

и выполняем те же самые команды. Чтобы обработать 6 символов, можно организовать цикл со счетчиком в регистре **R1**:

```
MOV @D,R0 ; адрес данных — в R0
MOV 6,R1  ; счётчик шагов цикла
          ; (сделать 6 раз)
M:
  MOVB (R0),R2 ; прочитать байт из памяти
  OR 20,R2     ; заглавную — в строчную
  MOVB R2,(R0) ; записать байт в память
  ADD 1,R0     ; к следующему байту
  SUB 1,R1     ; уменьшить счетчик
               ; оставшихся шагов
JNZ M          ; если не все сделали —
               ; переход на метку M
STOP
D:
  DATA "ABCDEFG"
```

Отметим, что две команды

```
MOVB R2,(R0) ; записать байт в память
ADD 1,R0     ; перейти к следующему байту
```

можно заменить на одну, которая делает то же самое:

```
MOVB R2,(R0)+ ; записать байт в память
              ; и перейти к следующему
              ; байту
```

Практикум

1. Введите программу

```
DATA 01D0
DATA 3536
DATA 0101
DATA FFFF
```

Используя дисассемблер, запишите эту программу на языке ассемблера. Запишите содержимое памяти, в которой располагается эта программа, в виде последовательности символов.

2. Как вы думаете, какой код будет иметь команда **MOV R3,R2**? Проверьте свой ответ с помощью программы.

3. Блок данных программы выглядит так.

```
A:
  DATA 1234
B:
  DATA 4321
SUM:
  DATA 0
```

Напишите программу, которая складывает переменные **A** и **B** и записывает результат в переменную **SUM**.

4. Напишите программу, которая преобразует строчные буквы в заглавные, используя байтовые операции. Что произойдет, если среди исходных данных уже есть заглавные буквы?

5. Усовершенствуйте программу так, чтобы цикл останавливался не после заданного количества букв, а тогда, когда очередной прочитанный байт равен 0.

6. Поскольку в компьютере с архитектурой фон Неймана программа и данные расположены в одной области памяти, программа может менять свой собственный код. Напишите какую-нибудь программу, которая изменяет сама себя во время работы.

Ламповая панель

Наконец, мы подошли к самой интересной возможности программы *ЛамПанель* — управлению ламповой панелью. Ламповая панель (область 2 на *рис.* 1) — это устройство вывода.

Обмен данными процессора и внешнего устройства происходит через *порты* — регистры контроллера панели. У ламповой панели 8 портов, которые называются **P0**, **P1**, **P2**, **P3**, **P4**, **P5**, **P6** и **P7**. Каждый порт "отвечает" за одну строку лампочек; например, для того чтобы "зажечь" всю верхнюю строку, нужно записать в порт **P0** код $FFFF_{16}$ (все 16 бит — единичные, все лампочки горят). Для этого можно использовать, например, команды

```
MOV FFFF, R0
OUT R0, P0 ; запись значения R0 в порт P0
```

К сожалению, записать число сразу в порт нельзя — сначала нужно записать его в регистр общего назначения (в данном примере — в **R0**), а потом — из регистра в порт.

Для того чтобы изменить второй сверху ряд лампочек, нужно записать новое значение в **P1** и т.д.; последний ряд управляется портом **P7**. Например, для того чтобы все ряды лампочек горели одинаково, можно сначала записать нужный код в регистр:

```
MOV AAAA, R0
```

а затем из этого регистра — во все порты:

```
OUT R0, P0
OUT R0, P1
…
OUT R0, P7
```

Здесь многоточие обозначает аналогичные команды записи содержимого регистра **R0** в порты **P2**…**P6**. Однако вместо последней серии из восьми команд можно использовать всего одну:

```
SYSTEM 2
```

Эта команда вызывает *системную подпрограмму* с номером 2, находящуюся в постоянном запоминающем устройстве (ПЗУ) компьютера.

Так же, как и у реального компьютера, ПЗУ — это неизменяемая область памяти, то есть после запуска компьютера *ЛамПанель* изменить содержимое ПЗУ нельзя.

Для того чтобы увидеть все подпрограммы, которые записаны в ПЗУ, нужно щелкнуть по кнопке 

или выбрать пункт верхнего меню *Программа — Просмотр ПЗУ*. После этого появляется окно, в левой части которого перечислены все системные подпрограммы (с их номерами), а в правой части показывается текст выбранной подпрограммы и ее машинные коды:

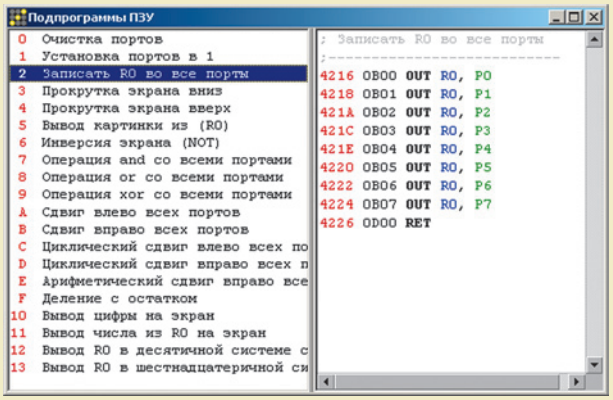


Рис. 6

В этом списке есть много полезных подпрограмм для работы с ламповой панелью, в том числе

0 — очистка экрана (погасить все лампочки);

1 — зажечь все лампочки на панели;

3–4 — прокрутка изображения вниз и вверх;

6–9 — логические операции;

A_{16}–E_{16} — сдвиги битов;

12_{16} — вывод числа, записанного в регистр **R0**, в десятичной системе счисления;

13_{16} — вывод числа, записанного в регистр **R0**, в шестнадцатеричной системе счисления.

Обратите внимание, что номер системной подпрограммы задается в шестнадцатеричной системе счисления.

Щелкнув мышкой на названии подпрограммы в левой части окна, мы увидим в правой части ее содержимое и комментарии, объясняющие, в каких регистрах должны быть расположены параметры подпрограммы и как она возвращает значения-результаты.

Рассмотрим еще одну задачу: вывести на экран рисунок, закодированный в виде шестнадцатеричных чисел (бит, равный единице, обозначает горящую лампочку). Для этого нужно сначала записать коды рисунка в память. Поскольку наш компьютер основан на архитектуре фон Неймана, в нем программа и данные находятся в одной области памяти. Поэтому данные можно записать с помощью специальной команды **DATA** после команды **STOP**:

```
...         ; здесь будет программа
STOP
M:          ; метка — начало блока данных
  DATA AAAA  ; код первой строчки
  DATA 5555
  DATA AAAA
  DATA 5555
  DATA AAAA
  DATA 5555
  DATA AAAA
  DATA 5555  ; код последней строчки
```

Для того чтобы вывести этот рисунок на экран, нужно записать его адрес в регистр **R0** и вызвать системную подпрограмму с номером 5:

```
MOV @M, R0  ; записать адрес метки M
            ; в регистр R0
SYSTEM 5 ; вывести на экран рисунок,
         ; адрес которого в R0
STOP
M:
  DATA AAAA; код первой строчки
  ...
```

ПЗУ учебного компьютера *ЛамПанель* хранится в текстовом файле, который загружается в память при запуске программы (так же, как и у учебной модели компьютера “E97” [3]). Поэтому такое ПЗУ можно изменять! Если требуется добавить новую подпрограмму в ПЗУ, нужно сохранить ее в специальном формате с помощью кнопки 

(или команды меню *Программа — Сохранить как ПЗУ*). Затем текст подпрограммы просто добавляется в конец файла **lampanel.rom**, в котором хранится ПЗУ.

Практикум

1. Составьте программу, после выполнения которой ламповая панель выглядит так:

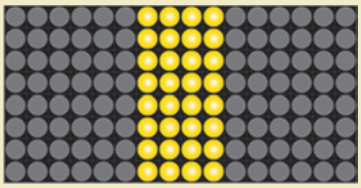


Рис. 7

2. Как вы думаете, что выведет приведенная выше программа, которая вызывает системную процедуру с номером 5? Проверьте ваш ответ с помощью программы *ЛамПанель*.

3. Закодируйте изображение домика и выведите его на экран.

4. Добавьте в предыдущую программу команды, которые сначала шифруют изображение, используя операцию “исключающее ИЛИ” с маской $BCA7_{16}$, а затем — восстанавливают исходное изображение. При изменении маски программа не должна изменяться. Изучите текст системной процедуры, которую вы используете.

5. Напишите программу, которая делает “бегущую строку” из рисунка-домика. *Подсказка*: используйте команды циклического сдвига.

6. Напишите программу, которая организует “обратный отсчет” от 100 до 0, а затем выводит рисунок с домиком и останавливается.

Подсказка: для вывода чисел используйте системную подпрограмму с номером 12_{16}.

Как вызываются подпрограммы

Вызов подпрограмм — это достаточно сложная операция для процессора. Действительно, при этом нужно:

- запомнить *адрес возврата* — адрес команды, на которую нужно перейти после завершения работы подпрограммы;
- передать параметры в *подпрограмму*;

• разместить в памяти локальные переменные подпрограммы;

• для подпрограммы-функции: передать в вызывающую программу результаты работы;

• обеспечить возврат из подпрограммы по правильному адресу.

В решении этих задач важнейшую роль играет *стек* (англ. *stack* — кипа, стопка) — специальная область оперативной памяти, в которой хранятся адреса возврата из подпрограмм и локальные переменные.

Стек в программе *ЛамПанель* размещается в оперативной памяти вместе с программой и данными. Оперативная память имеет размер 256 байт, адреса ячеек (байтов) изменяются от 0 до 255 = $\mathbf{FF}_{16}$. Стек находится в самом конце оперативной памяти и "растет вверх". Это значит, что первое записанное в стек 16-битное слово имеет адрес $\mathbf{FE}_{16}$ и занимает последние два байта памяти с адресами $\mathbf{FE}_{16}$ и $\mathbf{FF}_{16}$. Следующее записанное слово расположено по адресу $\mathbf{FC}_{16}$ и т.д. Как же компьютер разбирается, где начинается стек и сколько чисел туда записано?

В процессоре есть специальный регистр **SP** (от англ. *Stack Pointer* — указатель стека), в котором хранится адрес *вершины стека*, то есть последнего записанного в стек 16-битного значения. При запуске программы в регистр **SP** записывается значение 100_{16} = 256. Этот адрес находится за пределами оперативной памяти и говорит о том, что стек пуст. При добавлении в стек первого значения указатель стека уменьшается на 2 (до $\mathbf{FE}_{16}$), а потом по этому адресу записывается число.

Для того чтобы записать в стек значение из регистра общего назначения, используется команда **PUSH** (от англ. *push* — втолкнуть). Например, при выполнении этих команд в стек будет записано значение регистра **R0**, равное 1234_{16}:

```
MOV 1234,R0
PUSH R0
```

После выполнения команды **PUSH** значение регистра **SP** стало равно $\mathbf{FE}_{16}$, и теперь он указывает на последнее слово памяти, в котором записано число 1234_{16} — значение регистра **R0** (проверьте это с помощью программы).

Добавим в конец программы команду POP (англ. *pop* — вытолкнуть), которая "снимает" 16-битное значение с вершины стека и записывает его в регистр **R2**:

```
POP R2
```

После этого наблюдаем следующее:

• в регистре **R2** находится то же значение 1234_{16}, которое вначале было в **R0**;

• регистр **SP** содержит значение 100_{16}, которое говорит о том, что стек пуст;

• в последних двух байтах памяти осталось значение 1234_{16}, которое было записано в стек, но теперь оно уже не является частью стека, поскольку регистр **SP** изменен.

Как же используется стек при вызове подпрограмм? Напишем подпрограмму, которая возводит в квадрат значение регистра **R0**. Эта подпрограмма содержит всего одну команду умножения (умножить **R0** на **R0**, записать результат в **R0**):

```
MUL R0, R0
```

В начале подпрограммы нужно поставить метку (имя подпрограммы), а в конце — команду возврата **RET** (от англ. *return* — возврат), по которой процессор возвращается в точку вызова. Таким образом, вся подпрограмма, которую мы назовем **SQR**, выглядит так:

```
SQR:
  MUL R0, R0
RET
```

"Паспорт" этой подпрограммы такой:

Имя: SQR

Вход: число в регистре **R0**

Выход (результат): квадрат числа в регистре **R0**

Подпрограмма располагается ниже основной программы. Чтобы вызвать подпрограмму, используют команду **CALL** (от англ. *call* — вызвать), после которой записывают имя подпрограммы — метку, с которой она начинается, адрес точки входа. Вот вся программа вместе с подпрограммой:

```
MOV 12,R0 ; R0 := 12
CALL SQR ; вызвать подпрограмму SQR
STOP  ; конец основной программы
SQR:  ; точка входа подпрограммы
  MUL R0, R0 ; тело подпрограммы –
             ; R0 := R0 * R0
RET          ; возврат из подпрограммы
```

Остается один вопрос: как процессор определяет адрес возврата из подпрограммы, когда выполняется команда **RET**? Заметим, что в самой команде **RET** адрес не указан. Дело в том, что адрес перехода заранее определить нельзя (нельзя поставить команду безусловного перехода **JMP**), потому что подпрограмма может вызываться из разных мест программы, в том числе из других подпрограмм. Оказалось, что эта проблема просто решается с помощью стека:

• команда **CALL** записывает в стек адрес возврата из подпрограммы, то есть адрес команды, следующей за командой **CALL**; поскольку регистр **PC** (программный счетчик) всегда содержит адрес следующей команды, процессору достаточно просто "втолкнуть" содержимое регистра **PC** в стек;

• после этого в регистр **PC** записывается адрес подпрограммы (метки), указанной в команде **CALL**, и ей передается управление;

• команда **RET** снимает с вершины стека адрес возврата и записывает его в регистр **PC**, таким образом, управление передается следующей команде вызывающей программы.

Итак, при вызове подпрограммы адрес, содержащийся в регистре **PC**, временно запоминается в стеке, а по команде **RET** восстанавливается. Это обеспечивает возврат в то самое место программы, откуда вызывалась подпрограмма (на следующую команду после команды **CALL**).

Напишем более сложную подпрограмму, которая возводит **R0** в куб. Теперь для вычисления

придется задействовать еще один регистр, например, **R1**:

```
MOV R0,R1   ; R1 := R0
MUL R0,R0   ; R0 := R0 * R0
MUL R1,R0   ; R0 := R1 * R0
```

Все хорошо, но… в первой команде мы стерли значение регистра **R1**, которое было до вызова подпрограммы. Чтобы при вызове подпрограммы регистры не стирались, их желательно сохранять при входе в подпрограмму и восстанавливать перед самым выходом. В этом случае, вызывая подпрограмму, мы можем не беспокоиться, что она испортит нужные нам данные.

Где сохранять значения регистров? Самый простой выход — использовать стек: сохранять командой **PUSH** и восстанавливать командой **POP**. Таким образом, полный текст подпрограммы **CUBE** с сохранением регистра **R1** выглядит так:

```
CUBE:
  PUSH R1          ; сохраняем R1
  MOV R0,R1        ; R1 := R0
  MUL R0,R0        ; R0 := R0 * R0
  MUL R1,R0        ; R0 := R1 * R0
  POP R1           ; восстанавливаем R1
RET
```

В предыдущих примерах вы уже увидели, что параметры (дополнительные данные) могут передаваться в подпрограмму через регистры общего назначения **R0-R3**. Но этих регистров всего четыре, поэтому таким способом можно передать только четыре 16-битных числа. А что, если нужно передать, например, массив из 100 элементов? В этом случае снова на помощь приходит стек.

Перед вызовом подпрограммы в стек записываются все передаваемые параметры. Рассмотрим сначала "игрушечную" задачу — написать подпрограмму, которая возводит число в квадрат, причем это число передается через стек. Результат должен быть помещен в регистр **R0**.

Перед вызовом подпрограммы запишем в стек значение **R0**:

```
0000 MOV 12,R0    ; R0 := 12
0004 PUSH R0      ; запишем R0 в стек
0006 CALL SQR     ; вызов подпрограммы
000A STOP
000C SQR:
     ...    ; здесь будет подпрограмма
```

Если посмотреть на стек (в нижней части оперативной памяти), то после выполнения команды **PUSH R0** он выглядит так:

```
SP → 00FE  0012
```

Указатель стека **SP** содержит адрес $\mathbf{FE}_{16}$ и указывает на последнее 16-битное слово памяти. В нем записано число 12_{16} — это значение, передаваемое в подпрограмму.

Во время выполнения команды **CALL** в стек запишется адрес возврата из подпрограммы, то есть адрес $\mathbf{000A}_{16}$, по которому расположена команда **STOP**. Регистр **SP** будет указывать на ячейку памяти, содержащую этот адрес:

```
SP → 00FC  000A
     00FE  0012
```

Теперь займемся подпрограммой: как "достать" переданное значение? Сначала нужно скопировать содержимое указателя стека в какой-то регистр общего назначения, например, в **R0**:

```
MOV SP,R0
```

Теперь в **R0** находится адрес вершины стека, но там лежит адрес возврата. Чтобы получить адрес переданного параметра, нужно увеличить **R0** на 2:

```
ADD 2,R0
```

Теперь в **R0** записан адрес параметра, переданного подпрограмме. Можно взять значение по этому адресу и записать его в тот же регистр **R0**:

```
MOV (R0),R0
```

Теперь в **R0** уже получено переданное число и можно возвести его в квадрат. Вот полная подпрограмма:

```
SQR:
  MOV SP,R0
  ADD 2,R0
  MOV (R0),R0
  MUL R0,R0
RET
```

Остается один вопрос — кто же будет освобождать стек, удаляя из него параметры подпрограммы? Тут есть два варианта. Во-первых, этим может заниматься вызывающая программа — после вызова подпрограммы нужно использовать команду **POP**. Кроме того, это может делать и процедура — для этого нужно применить команду **RET** с параметром, обозначающим количество байт, которые нужно "сбросить" со стека. Например, в нашем случае можно применить команду

```
RET 2
```

которая освободит 2 байта (удалит один параметр). Отметим, что параметр команды **RET** — четное число, записанное в шестнадцатеричной системе счисления.

Таким образом, если параметров мало, их удобно передавать через регистры **R0-R3**. Кроме того, параметры можно передавать через стек. Если подпрограмма обрабатывает большой массив, лучше передать ей адрес этого массива, вместо того чтобы записывать его в стек целиком.

При изучении программирования *рекурсивные* подпрограммы (которые вызывают сами себя) выделяются в особый класс подпрограмм и изучаются отдельно. Однако для процессора такая подпрограмма ничем не отличается от "обычной", нерекурсивной. Отличие состоит только в том, что внутри рекурсивной подпрограммы есть вызов **CALL** по адресу той же самой подпрограммы.

Напишем рекурсивную подпрограмму для вычисления факториала числа, находящегося в регистре **R0**. Результат должен быть помещен в тот же регистр **R0**.

Факториал числа N вычисляется как произведение всех натуральных чисел от 1 до N: $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot$

$\cdot (N-1) \cdot N$. Вспомним рекурсивное определение факториала:

$$\begin{cases} 1! = 1 \\ N! = (N-1)! \cdot N, \quad N > 1 \end{cases}$$

Это значит, что факториал числа, большего единицы, определяется *рекурсивно* — через факториал другого числа, которое меньше данного на 1. Поэтому основная идея подпрограммы может быть записана на псевдокоде так:

```
R1 := R0
R0 := R0 - 1
; вычисляем факториал R0
; (вызов подпрограммы)
R0 := R1 * R0
```

Перевод на язык ассемблера дает:

```
FACT:
  MOV R0,R1    ; R1 := R0
  SUB 1,R0     ; R0 := R0 - 1
  CALL FACT    ; R0 := факториал R0
  MUL R1,R0    ; R0 := R1 * R0
FINISH:
  RET
```

Однако выполнение этой подпрограммы никогда не закончится, потому что при каждом обращении к подпрограмме она снова вызывает сама себя, и эти вызовы никогда не остановятся. Нужно добавить условие выхода: если **R0** = 1, нужно выйти из подпрограммы, то есть перейти на последнюю команду **RET** (перед ней должна быть метка):

```
FACT:
  CMP 1,R0      ; если R0 = 1, то выход
  JZ FINISH
  MOV R0,R1     ; R1 := R0
  SUB 1,R0      ; R0 := R0 - 1
  CALL FACT     ; R0 := факториал R0
  MUL R1,R0     ; R0 := R1 * R0
FINISH:
  RET
```

Остается еще один недостаток — подпрограмма изменяет значение регистра **R1**. Нужно при входе в подпрограмму сохранить его (в стеке), а перед выходом — восстановить. Как это сделать, мы рассматривали выше.

Практикум

1. Напишите и отладьте программу, которая меняет местами значения регистров **R2** и **R3** с помощью стека (не используя других регистров общего назначения).

2. Введите текст программы

```
MOV 12,R0
CALL SQR
STOP
SQR:
  MUL R0,R0
RET
```

Заполните таблицу, выполнив программу пошагово с помощью клавиши F7 (пошаговое выполнение с входом в подпрограммы):

Адрес	Команда	Регистры после ее выполнения		
		R0	**PC**	**SP**
		?	**0000**	**0100**
	MOV 12,R0			
	CALL SQR			
	MUL R0,R0			
	RET			
	STOP			

3. Напишите и отладьте программу с подпрограммой, которая вычисляет куб числа, записанного в регистр **R0**.

4. Напишите и отладьте программу с подпрограммой, которая и строит RGB-код цвета, 4-битные составляющие которого (R, G и B) записаны, соответственно, в регистры **R0**, **R1** и **R2**. Результат должен быть получен в регистре **R0**.

5. Выполните предыдущее задание при условии, что параметры передаются через стек, а значения регистров **R1** и **R2** не должны измениться.

6. Отладьте программу с рекурсивной подпрограммой, которая вычисляет факториал числа, записанного в регистр **R0**. При выполнении в пошаговом режиме (клавиша F7) наблюдайте, как изменяется регистр **SP** и содержимое стека.

7. Решите предыдущую задачу, применив подпрограмму без рекурсии.

Выводы

В статье представлена новая учебная модель компьютера *ЛамПанель*. Ее главное отличие от существующих — возможность управления ламповой панелью (матрицей, состоящей из лампочек) и современный интерфейс. В профильном курсе информатики с помощью программы *ЛамПанель* можно в наглядной форме изучать основные принципы работы современного компьютера.

Список использованных источников

1. *Еремин Е.А.* Debug и язык ассемблер // Информатика, 2007, № 5, с. 37–39; № 6, с. 37–38; № 7, с. 42–44.

2. *Еремин Е.А.* Учебные модели компьютера. Электронный ресурс (*http://educomp.runnet.ru/*).

3. *Еремин Е.А.* Учебный компьютер “E97” для курса основ информатики и вычислительной техники // VIII Международная конференция “Информационные технологии в образовании”. Тезисы докладов. Направление B, с. 26–27. Москва, 1998 (*http://ito.edu.ru/1998-99/b/eremin.html*).

4. *Леонов А.Г.* ЭВМ-практикум // Информатика, 2011, № 11, с. 4–12.

5. EMU8086: 8086 microprocessor emulator integrated disassembler. Электронный ресурс (*http://ziplib.com/emu8086*).

6. *Поляков К.Ю., Шестаков А.П., Еремин Е.А.* Компьютерная арифметика // Информатика, № 1, 2011, с. 2–21.

Автор благодарит Е.А. Еремина за обсуждение этой статьи и полезные замечания.