

Практическая работа № 7 Написание программ для эмулятора ЭВМ «ЛамПанель»

Тема: Написание программ для эмулятора ЭВМ «ЛамПанель»

Цель:

- для изучения принципов работы компьютера (процессор, ОЗУ, ПЗУ);
- для начального изучения программирования на языке ассемблера;
- для изучения операций с целыми числами, в том числе поразрядных логических операций и сдвигов.

Программа-тренажёр «ЛамПанель» – это учебная модель компьютера, который управляет панелью лампочек.

Модель компьютера включает:

- процессор,
- оперативную память (ОЗУ),
- постоянную память (ПЗУ)
- устройство вывода – панель лампочек размером 8×16.

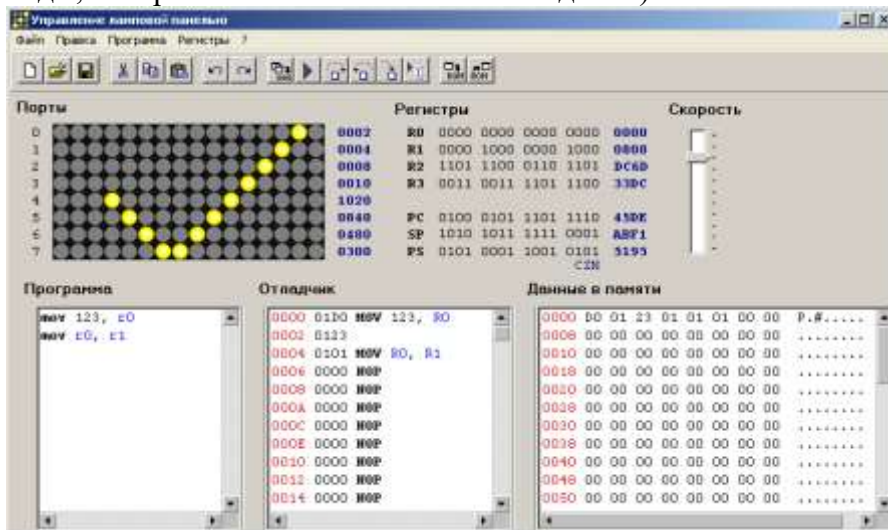
Справочник по языку ассемблера «ЛамПанель»

Система команд процессора в тренажёре «ЛамПанель» – 16-ти -разрядная.

Программа для процессора составляется на языке ассемблера, в котором каждая символьная команда соответствует одной (числовой) команде процессора.

Программа набирается в окне «Программа». Программу можно сохранять в виде файла на диске, а потом загружать в память из файла (с помощью меню «Файл»).

При запуске (по клавише F9) набранная программа транслируется (переводится в машинные коды, которые появляются в окне «Отладчик») и начинается ее выполнение.



В окне «Данные в памяти» показывается содержимое оперативной памяти, в которой расположены программа и данные (принцип однородности).

Структура процессора тренажёра «ЛамПанель»

Процессор имеет 4 регистра общего назначения (РОН), которые называются **R0, R1, R2, R3**. Кроме того, есть еще три внутренних регистра, недоступные программисту (но видимые в окне программы):

- **PC** (англ. *program counter*) – программный счетчик, счётчик команд, указывающий на следующую выполняемую команду;
- **SP** (англ. *stack pointer*) – указатель стека, адрес вершины стека; стек размещается в нижней части оперативной памяти, его содержимое можно просмотреть в нижней части окна «Данные в памяти»;
- **PS** (англ. *processor state*) – регистр состояния процессора; используются только три младших бита: 0 – бит **N** (англ. *negative*, отрицательный результат), 1 – бит **Z** (англ. *zero*, нулевой результат) и 2 – бит **C** (англ. *carry*, перенос).

В качестве устройства вывода используется **панель лампочек** размером 8×16. Каждый ряд лампочек управляется через отдельный порт вывода.

Всего используется **восемь** 16-разрядных портов с именами **P0, P1, P2, P3, P4, P5, P6 и P7**.

Постоянное запоминающее устройство (ПЗУ) предназначено для хранения системных подпрограмм. Код ПЗУ загружается при старте программы из текстового файла, поэтому пользователь может изменять содержимое ПЗУ: добавлять, удалять и изменять любые процедуры.

Основные правило написания программ:

- Программа должна заканчиваться командой **stop**. Например, самая простая программа: **stop**
- Команда **NOP** (*no operation*, нет операции) – это пустая команда, она ничего не делает.
- Комментарий начинается символом «точка с запятой»: **nop ; пустая команда, stop**

Инструменты отладки программы

Программу можно выполнить всю целиком (клавиша F9) или в пошаговом режиме (F8). В пошаговом режиме в окне отладчика зелёным цветом выделяется текущая строка, которая будет выполнена при следующем нажатии F8. Сочетание клавиш Ctrl+F8 позволяет отменить только что сделанную команду.

Клавиша F7 (вместо F8) позволяет войти в подпрограмму и выполнить ее пошагово (см. раздел «Подпрограммы» ниже).

Если установить курсор в какую-то строчку программы и нажать клавишу F4, программа будет выполняться до этой строчки и затем остановится.

Движок «Скорость» изменяет скорость выполнения программы.

Все команды отладки включены в меню «Программа».

Кроме того, они могут выполняться с помощью кнопок панели инструментов:

	Трансляция в машинные коды без выполнения (Ctrl+F9).
	Трансляция и выполнение (F9).
	Выполнить один шаг (F8).
	Отменить один шаг (Ctrl+F8).
	Войти в подпрограмму (F7).
	Выполнить до курсора (F4).

С помощью меню «Регистры» можно изменить значения любого регистра во время выполнения программы в пошаговом режиме.

Работа с регистрами и портами

Для простейшей обработки данных можно использовать 4 регистра процессора и 8 портов ламповой панели.

Основные операции:

- записать данные в регистр, например,

ассемблер	псевдокод
mov 1234, R0	R0:=1234₁₆

Все числа записываются в шестнадцатеричной системе счисления.

- скопировать значение из одного регистра в другой, например,

ассемблер	псевдокод
mov R0, R1	R1:=R0

- вывести значение регистра в порт

ассемблер	псевдокод
out R0, P1	P1:=R0

- прочитать значение из порта в регистр

ассемблер	псевдокод
in P2, R0	R0:=P2

Пример программы:

ассемблер	псевдокод
mov 1234, R0	R0:= 1234₁₆
mov R0, R2	R2:= R0

out R2, P1 stop	P1:= R2 стоп
----------------------------------	-------------------------------

Арифметические операции

Арифметические операции могут выполняться с числами (константами) и значениями регистров. Результат записывается по адресу второго операнда-регистра (это не может быть число).

- сложение

ассемблер	псевдокод
add 1, R1 add R2, R3	R1:= R1 + 1 R3:= R3 + R2

- вычитание

ассемблер	псевдокод
sub 2, R1 sub R2, R3	R1:= R1 - 2 R3:= R3 - R2

- умножение

ассемблер	псевдокод
mul 5, R1 mul R2, R3	R1:= R1 * 5 R3:= R3 * R2

- деление

ассемблер	псевдокод
div 12, R1 div R2, R3	R1:= R1 div 12 R3:= R3 div R2

Логические операции

Логические операции могут выполняться с числами (константами) и значениями регистров. Результат записывается по адресу второго операнда-регистра (это не может быть число).

- отрицание («НЕ»)

ассемблер	псевдокод
not R1	R1:= not R1

- логическое умножение («И»)

ассемблер	псевдокод
and R0, R1 and 1234, R1	R1:= R1 and R0 R1:= R1 and 1234₁₆

- логическое сложение («ИЛИ»)

ассемблер	псевдокод
or R0, R1 or 1234, R1	R1:= R1 or R0 R1:= R1 or 1234₁₆

- сложение по модулю 2 («исключающее ИЛИ»)

ассемблер	псевдокод
xor R0, R1 xor 1234, R1	R1:= R1 xor R0 R1:= R1 xor 1234₁₆

Операции сдвига

В командах сдвига первый операнд – это величина сдвига (от 1 до 10₁₆), а второй – регистр.

- логический сдвиг влево и вправо

ассемблер	псевдокод
shl 2, R1 shr F, R1	R1:= R1 shl 2₁₆ R1:= R1 shr F₁₆

- арифметический сдвиг вправо

ассемблер	псевдокод
sar 2, R1	R1:= R1 sar 2₁₆

- циклический сдвиг влево и вправо

ассемблер	псевдокод
rol 2, R1 ror F, R1	R1:= R1 rol 2₁₆ R1:= R1 ror F₁₆

- циклический сдвиг влево и вправо через бит переноса

ассемблер	псевдокод
rcl 2, R1	R1:= R1 rcl 2₁₆
rcr F, R1	R1:= R1 rcr F₁₆

Операции: метки, сравнения и переходы

Команды перехода используются для выполнения разветвляющихся алгоритмов. Различают безусловный переход (переходить всегда) и условные переходы (переход при выполнении какого-то условия).

Чтобы обозначить место перехода, необходимо создать метку. **Метка** – это произвольное имя, за которым следует двоеточие. **После двоеточия не должно быть никаких символов** (метка – это отдельная строка программы).

Безусловный переход имеет формат

jmp метка

Пример программы (бесконечный цикл):

```
qq:
por
jmp qq
```

Условные переходы зависят от битов состояния процессора, которые определяются результатом последней операции:

```
jge метка    ; если больше или равно
jl метка     ; если меньше
jnz метка    ; если не ноль
jz метка     ; если ноль
jle метка    ; если меньше или равно
jg метка     ; если больше
```

Пример программы (цикл из 5 шагов):

ассемблер	псевдокод
mov 5, R1	R1:=5
qq:	нц пока R1 <> 0
sub 1, R1	R1:= R1 - 1
jnz qq	кц

Существует команда сравнения, которая изменяет только биты состояния процессора:

ассемблер	значение
cmp 2, R1	установка битов состояния по значению 2₁₆-R1
cmp R0, R1	установка битов состояния по значению R0-R1

Пример программы:

ассемблер	псевдокод
cmp 5, R0	если R0=5 то
jnz aaa	R0:= R0 + 1
add 1, R0	все
aaa:	

Подпрограммы

Подпрограммы – это вспомогательные алгоритмы, которые можно вызывать по имени. В языке ассемблера имя подпрограммы – это метка. Для вызова подпрограммы используется команда

call метка

Подпрограмма должна заканчиваться командой возврата из подпрограммы

ret

Подпрограммы располагаются в программе ниже основной программы, после команды **stop**.

Пример программы, которая использует подпрограмму **divMod** для деления с остатком:

ассемблер	псевдокод
mov 1234, R0	R0:= 1234₁₆
mov 10, R1	R1:= 10₁₆
call divMod	вызвать divMod
stop	стоп
divMod:	

mov R0, R2	R2:= R0
div R1, R0	R0:= R0 div R1
mul R0, R1	R1:= R1 * R0
sub R1, R2	R2:= R2 – R1
mov R2, R1	R1:= R2
ret	возврат

Чтобы при отладке выполнять по шагам не только основную программу, но и подпрограмму, при выполнении команды **call** нужно нажать не F8, а F7.

Работа со стеком

Стек – это структура типа LIFO (англ. *Last In – First Out*, последним пришел – первым ушел). В современных компьютерах стек размещается в памяти, специальный регистр SP (англ. *stack pointer*) указывает на начало стека. Для работы со стеком используются всего две команды:

ассемблер	псевдокод
push R0	сохранить R0 в стеке
pop R0	«снять» данные с вершины стека в R0

Конечно, сохранять в стеке можно не только **R0**, но и другие регистры.

Стек используется:

- для временного хранения данных
- для хранения адресов возврата из подпрограмм
- для размещения локальных переменных подпрограмм

Пример программы (обмен значений регистров R0 и R1):

ассемблер	псевдокод
push R0	R0 – в стек
push R1	R1 – в стек
pop R0	со стека – в R0 (старое значение R1)
pop R1	со стека – в R1 (старое значение R0)

Если подпрограмма использует какой-то регистр, которые не содержит исходные данные и не предназначен для записи результата, она должна сохранить его в стеке при входе и восстановить старое значение из стека при выходе. Например:

ассемблер	псевдокод
proc:	начало подпрограммы
push R0	R0 – в стек
...	основное тело подпрограммы
pop R0	со стека – в R0 (старое значение R0)
ret	возврат из подпрограммы

Заметьте, что подпрограмма, приведенная в предыдущем пункте, не совсем грамотно написана – она не сохраняет значение регистра R2, хотя «портит» его во время работы.


Вызов подпрограмм из ПЗУ

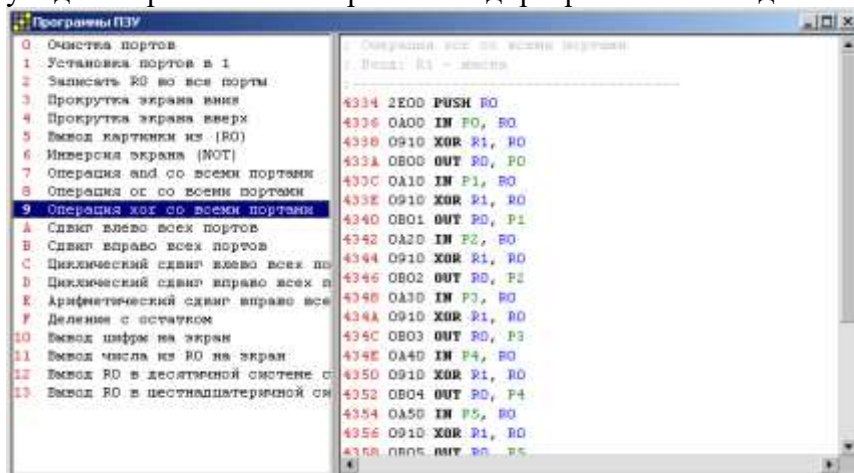
ПЗУ в данной модели компьютера – это набор подпрограмм, каждая из которых заканчивается командой **ret**. Всего в ПЗУ может быть до 256 подпрограмм.

ПЗУ загружается при запуске тренажёра «ЛамПанель» из файла **lampanel.rom**, который должен находиться в том же каталоге, что и сама программа. Это обычный текстовый файл, который можно редактировать в редакторах типа *Блокнота* (если, конечно, вы понимаете, что вы делаете). В настоящей версии в ПЗУ включены следующие подпрограммы:

Номер	Описание
0	Очистить все порты панели (выключить все лампочки).
1	Установить в FF ₁₆ все порты панели (включить все лампочки).
2	Записать значение R0 во все порты панели.
3	Прокрутить изображение на панели вниз.
4	Прокрутить изображение на панели вверх.
5	Вывести на панель массив данных, адрес которого находится в R0.
6	Выполнить инверсию экрана (применить NOT).
7	Операция «И» со всеми портами (R1 – маска).
8	Операция «ИЛИ» со всеми портами (R1 – маска).

9	Операция «исключающее ИЛИ» со всеми портами (R1 – маска).
A ₁₆	Логический сдвиг влево всех портов (R1 – величина сдвига).
B ₁₆	Логический сдвиг вправо всех портов (R1 – величина сдвига).
C ₁₆	Циклический сдвиг влево всех портов (R1 – величина сдвига).
D ₁₆	Циклический сдвиг вправо всех портов (R1 – величина сдвига).
E ₁₆	Арифметический сдвиг вправо всех портов (R1 – величина сдвига).
F ₁₆	Деление с остатком (R0:=R0 div R1, R1:=R0 mod R1).
10 ₁₆	Вывод цифры на экран (R0 – цифра, R1 – позиция, от 0 до 2)
11 ₁₆	Вывод числа из R0 на экран (R1 – система счисления, от 2 до 16).
12 ₁₆	Вывод числа из R0 на экран в десятичной системе счисления.
13 ₁₆	Вывод числа из R0 на экран в шестнадцатеричной системе счисления.

Просмотреть содержимое ПЗУ можно с помощью пункта меню «Программа-Просмотр ПЗУ» или кнопки  на панели инструментов. Выделив какую-нибудь строчку в левой части окна, мы увидим справа текст выбранной подпрограммы и ее коды:



Для вызова подпрограмм из ПЗУ нужно использовать команду:

system номер_подпрограммы

Пример программы:

асемблер	псевдокод
system 0	выключить панель
system 1	включить все лампочки
mov 123, R0	R0:= 123 ₁₆
system 12	вывести R0 в десятичной системе
system 6	инверсия
mov 1, R1	R1:= 1 ; величина сдвига
system A	сдвиг экрана влево
system B	сдвиг экрана вправо
system 13	вывести R0 в шестнадцатеричной системе
stop	стоп

Байтовые команды

Все рассмотренные выше команды работают с 16-битными данными (словами). Часто, например, при обработке текстов, нужно использовать однобайтные данные. Для этого предназначены следующие команды, которые полностью аналогичны соответствующим командам без буквы «b» (от англ. *byte*) на конце:

команда	значение
movb	копирование байта
cmpb	сравнение двух байтов
shlb	логический сдвиг влево
shrb	логический сдвиг вправо
sarlb	арифметический сдвиг вправо
rolb	циклический сдвиг влево
rorb	циклический сдвиг вправо

rcld	циклический сдвиг влево через бит переноса
rcrb	циклический сдвиг вправо через бит переноса

Команда **movb** очищает старший байт регистра, в который копируются данные. Например,

ассемблер	псевдокод
mov 1234, R0	R0:= 1234₁₆
movb 12, R0	R0:= 12₁₆

Остальные команды никак не изменяют старший байт регистра-приемника.

Существует специальная команда для обмена старшего и младшего байтов слова:

swapb регистр

Пример программы:

ассемблер	псевдокод
mov 1234, R0	R0:= 1234₁₆
swapb R0	R0:= 3412₁₆

Работа с данными

Согласно принципу однородности памяти фон Неймана, данные размещаются в той же области памяти, что и программа (обычно сразу после команды **stop**).

В тренажере «ЛамПанель» данные – это 16-битные слова (вводятся как числа в шестнадцатеричной системе счисления) или символьные строки, заключенные в двойные кавычки. Для размещения данных в памяти применяется команда **data**.

Например:

```
... ; основная программа
stop
ddd: ; метка начала блока данных
data 1234 ; слово 123416
data 5678 ; слово 567816
data "Ехал Грека через реку" ; строка
```

Для того, чтобы работать с этими данными, нужно как-то к ним обратиться. Для этого используется косвенная адресация – в регистре находятся не сами данные, а их адрес в памяти.

Рассмотрим пример:

ассемблер	псевдокод
mov @ddd, R0	R0:= адрес метки ddd
swapb (R0)	переставить байты слова под адресу ddd
add 2, R0	увеличить адрес на 2 (байта)
swapb (R0)	переставить байты слова под адресу ddd+2
stop	стоп
ddd:	начало блока данных
data 1234	здесь будет 3412₁₆
data 5678	здесь будет 7856₁₆

Запись **@метка** означает «адрес метки». Запись **(R0)** означает «данные, адрес которых находится в R0» – это и есть косвенная адресация.

Косвенную адресацию можно использовать и в других командах, работающих с регистрами.

Обработка массивов

Пусть в блоке данных, который начинается на метке **ddd**, записан массив, который нужно обработать в цикле. В этом случае удобно использовать косвенную адресацию с автоматическим увеличением адреса. Запись «**(R0)+**» означает «работать с данными, адрес которых находится в R0, и после выполнения операции увеличить R0». Если команда работает со словом, R0 увеличится на 2, а если с байтом – на 1.

Пример программы:

ассемблер	псевдокод
mov @ddd, R0	R0:= адрес метки ddd
mov 3, R1	записать в R1 количество шагов цикла
loop:	начало цикла
swapb (R0)+	переставить байты слова под адресу из R0
sub 1, R1	уменьшить счетчик оставшихся шагов

jnz loop stop ddd: data 1234 data 5678 data 9ABC	если счетчик не ноль, перейти в начало цикла стоп начало блока данных здесь будет 3412 ₁₆ здесь будет 7856 ₁₆ здесь будет BC9A ₁₆
---	---

Пример программы обработки байтов:

<i>ассемблер</i>	<i>псевдокод</i>
mov @ddd, R0 loop: movb (r0),r1 or 20,r1 movb r1,(r0)+ cmpb 0,(r0) jnz loop stop ddd: data "ABCD"	R0:= адрес метки ddd начало цикла R1:= байт слова под адресу из R0 из заглавной буквы сделать строчную записать результат в память сравнить код следующего байта с 0 если не ноль, перейти в начало цикла стоп начало блока данных здесь будет "abcd"

Самомодифицирующиеся программы


Поскольку данные находятся в той же области памяти, что и программы, программа может изменять свой код во время выполнения.

Например, для защиты от взлома может быть использовано шифрование: основной код программы зашифрован, и она сама себя расшифровывает при запуске.

Пример самомодифицирующейся программы:

<i>ассемблер</i>	<i>псевдокод</i>
jmp decode main: data ba6b data ba98 data 27a8 data 4444 decode: mov @main,r0 mov 4,r1 loop: xor bbbb,(r0)+ sub 1, r1 jnz loop jmp main	переход на блок расшифровки начало основной части в этом и следующем словах будет "mov 123, R0" здесь будет "system 13" здесь будет "stop" начало блока расшифровки R0:= начало зашифрованного блока R1:= 4 ; нужно расшифровать 4 слова начало цикла расшифровка: хог с маской BBBB ₁₆ уменьшить счетчик если счетчик не ноль, перейти на начало цикла перейти на основную программу

Расширение ПЗУ

Пользователь может добавить свои подпрограммы в ПЗУ. Для этого нужно сначала отладить подпрограмму, а затем сохранить ее в специальном формате с помощью кнопки  или пункта меню «Программа – Сохранить как ПЗУ».

Например, напишем подпрограмму, которая переставляет биты числа в обратном порядке, используя циклический сдвиг через бит переноса:

<i>ассемблер</i>	<i>псевдокод</i>
mov 1234, R0 call reverse stop reverse: push R1 push R2 mov 10, R2 xor R1, R1 next-bit:	R0:= 1234 ₁₆ вызов подпрограммы стоп начало подпрограммы сохранить R1 в стеке сохранить R2 в стеке R2:= 16 = 10 ₁₆ R1:= 0

rel 1, R0 rcr 1, R1 sub 1, R2 jnz next-bit mov R1, R0 pop R2 pop R1 ret	старший бит R0 попадает в бит переноса бит переноса попадает в старший бит R1 R2:= R2 – 1 если R2≠0, перейти к метке next-bit R0:= R1 восстановить R2 из стека восстановить R1 из стека возврат из подпрограммы
--	---

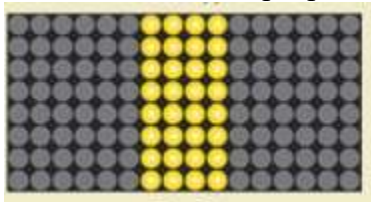
Отладив эту программу, уберем верхние три строчки, оставив только процедуру, и применяем команду меню «Программа – Сохранить как ПЗУ». Полученный файл (он будет иметь расширение **.rom**) открываем в любом текстовом редакторе (например, в Блокноте) и добавляем в начало комментариев:

```
; Перестановка битов R0
; в обратном порядке
;-----
2E10 PUSH R1
2E20 PUSH R2
01D2 MOV 10, R2
0010
0911 XOR R1, R1
9E00 RCL 1, R0
AE01 RCR 1, R1
03D2 SUB 1, R2
0001
4D0D JNZ 000A
FFF4
0110 MOV R1, R0
3E20 POP R2
3E10 POP R1
0D00 RET
```

Теперь остается добавить (также в текстовом редакторе) этот фрагмент в конец файла **lampanel.rom**. Новая процедура будет доступна при следующем запуске программы «ЛамПанель».

Практическая часть

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



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. Напишите программу, которая делает “бегущую строку” из рисунка-домика. Подсказка: используйте команды циклического сдвига.
5. 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 12₁₆.
6. 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		

7. 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?

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

Выражение	Результат	
	без учета знака	с учетом знака
65 530 + 9		
32 760 + 9		
8 - 10		

Подсказка: $65\,535 = \text{FFFF}_{16}$, $32\,767 = 7\text{FFF}_{16}$

9. 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
32 760 + 32 752							
-32 760 - 32 752							
256 - 256							

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

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

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

Объясните полученные результаты.

12. 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.

13. 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	

14. 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?

15. 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}

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

Самостоятельная работа

Заполните таблицу обозначение элементов и команд программы

Элемент	Назначение	Принадлежность
R0, R1, R2, R3		
PC;		
PS		
SP		
Команда		
NOP		
STOP		
MOV		
ADD		
SUB		
CMP		
MUL		
DIV		
NOT		
AND		
OR		
XOR		
SHL		
SHR		
SAR		
ROL		
ROR		
RCL		
RCR		
JMP		
JGE		
SYSTEM		
CALL		
RET		

