

**МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
имени М.В. ЛОМОНОСОВА
Факультет вычислительной математики и кибернетики**

**В.Г. Баула, Е.А. Бордаченкова
Задачи письменного экзамена
по курсу «Архитектура ЭВМ
и язык ассемблера»**

Учебно-методическое пособие
для студентов 1 курса

Москва, 2019

УДК 519.6(075.8)

ББК 32.973-018

Б29

Печатается по решению Редакционно-издательского совета
факультета вычислительной математики и кибернетики
Московского государственного университета имени М.В. Ломоносова

Рецензенты:

Доцент Гуляев Анатолий Викторович

Доцент Кузьменкова Евгения Анатольевна

Баула В.Г., Бордаченкова Е.А. Задачи письменного экзамена по курсу «Архитектура ЭВМ и язык ассемблера». Учебное пособие для студентов I курса. – М. Издательский отдел факультета ВМиК МГУ (лицензия ИД № 05899 от 24.09.2001 г.); МАКС Пресс, 2019, – 48 с.

ISBN 978-5-317-06094-7

Пособие содержит методические рекомендации для студентов по решению задач на письменном экзамене по основному лекционному курсу «Архитектура ЭВМ и язык ассемблера», читаемому на факультете ВМК МГУ. Обсуждаются правила проведения письменного экзамена и разбираются типичные задачи из вариантов экзаменов прошлых лет.

Ключевые слова: архитектура ЭВМ, язык ассемблера

Baula V.G., Bordachenkova E.A. Some Exercises from Computer Architecture and Assembly Language Course Written Examinations.

This handbook is aimed to help students to prepare for the exam on Computer Architecture and Assembly Language course. In the handbook several organizational practices as well as some typical questions from past exams are analyzed. It is addressed to the first-year students of CMC department of Moscow State University.

Keywords: computer architecture, assembly language

Учебное издание

БАУЛА Владимир Георгиевич, БОРДАЧЕНКОВА Елена Анатольевна

ЗАДАЧИ ПИСЬМЕННОГО ЭКЗАМЕНА ПО КУРСУ «АРХИТЕКТУРА ЭВМ И ЯЗЫК АССЕМБЛЕРА»

Учебное пособие для студентов первого курса

© Факультет ВМиК МГУ имени М.В. Ломоносова, 2019

© Баула В.Г., Бордаченкова Е.А., 2019

1. Введение

Письменный экзамен по курсу «Архитектура ЭВМ и язык ассемблера» проводится в весеннем семестре 1 курса. Экзаменационный вариант содержит от 8 до 10 задач, продолжительность экзамена 2 или 2.5 часа. Каждому студенту выдаётся лист с формулировками задач – вариант экзамена; решения требуется написать на этом же листе. В конце экзамена листы-варианты собираются, решения проверяются, и выставляется оценка за экзамен. Предполагается, что каждая задача сначала решается на черновике, а затем переписывается в лист-чистовик, содержащий формулировки задач. Необходимо, однако, учесть, что черновики не проверяются и на оценку не влияют.

Экзаменационные задачи имеют разную сложность, но, независимо от этого, решение каждой задачи оценивается по шкале от 0 до 6 баллов. Максимальное количество баллов не зависит от сложности задачи – и самая лёгкая, и самая сложная правильно решённые задачи дают по 6 баллов. Можно также считать, что задача оценивается от нуля до единицы с шагом 1/6, например, если сказано, что за задачу поставлено 1/3, то за эту задачу засчитано $6 \cdot (1/3) = 2$ балла. Все баллы суммируются; экзаменационная оценка определяется суммой баллов.

Исходя из разной сложности задач, рекомендуется такая последовательность сдачи экзамена. Сначала необходимо внимательно прочитать формулировки всех задач и выбрать из них самую лёгкую (с точки зрения экзаменуемого). Затем выбранная задача решается на черновике, проверяется и переписывается в чистовик. Далее берётся самая лёгкая из оставшихся задач и т.д. Важно следить за временем, в течение которого обдумывается одна задача. Если не получается решить задачу за 5-10 минут, нужно переходить к следующей задаче; а уж если в конце экзамена останется время, можно будет вернуться к отложенной задаче.

Рассматриваемые в пособии задачи разбиваются по темам, соответствующим основным темам курса «Архитектура ЭВМ и язык ассемблера»: представление чисел в ЭВМ, различные архитектуры ЭВМ, основные понятия языка ассемблера, элементы систем программирования, некоторые особенности архитектуры современных компьютеров. В каждом экзаменационном задании предполагается охват большого количества тем, поэтому все они важны. В конце пособия приведён один пример полного варианта экзаменационного задания.

Пособие составлено по материалам экзаменов прошлых лет, когда на лекциях изучалась 16-разрядная архитектура Intel и ассемблер MASM 4.0. В настоящее время в курсе «Архитектура ЭВМ и язык ассемблера» рассматривается архитектура IA-32 и язык ассемблера MASM 6.14. В книгу вошли экзаменационные задачи, сохранившие актуальность; формулировки некоторых задач были откорректированы в соответствии с действующим курсом; были добавлены новые задачи.

При обсуждении синтаксиса команд используются принятые в литературе обозначения операндов *r* – регистр, *m* – ячейка памяти, *i* – непосредственный операнд с указанием размера в битах, например, *r8* – байтовый регистр, *m32* – двойное слово из памяти. Используются следующие обозначения макрокоманд ввода-вывода

inint *r8/r16/r32/m8/m16/m32* – ввод числа;
inch *r8/m8* – ввод символа;
outi *r8/r16/r32/m8/m16/m32/i8/i16/i32* – вывод числа со знаком;
outu *r8/r16/r32/m8/m16/m32/i8/i16/i32* – вывод числа без знака;
outch *r8/m8/i8* – вывод символа;
outstr *i32/r32* – вывод строки.

В последней команде операнд задаёт адрес начала строки; за строкой должен идти байт 0.

В пособии даны краткие сведения по соответствующим темам, полный материал по темам не приводится, пособие не является учебником. Для полноценной подготовки к экзамену нужно обратиться к учебникам.

Рекомендуемая литература

1. Баула В.Г. Введение в архитектуру ЭВМ и системы программирования. – <http://arch32.cs.msu.ru>.
2. Бордаченкова Е.А. Модельные ЭВМ. – М., МАКС Пресс, 2012
3. Пильщиков В.Н. Программирование на языке ассемблера IBM PC. – М., Диалог-МИФИ, 2005
4. Юров В.И. Assembler. Учебник для ВУЗов, 2-е изд., 2003.

2. Разбор задач

2.1. Представление чисел в ЭВМ

В вычислительных машинах используется два способа хранения чисел в ячейке оперативной памяти: так называемые «числа без знака» и «числа со знаком». Разберём их на примере ячейки из k разрядов.

Рассмотрим сначала числа без знака. Будем считать, что в ячейке содержится неотрицательное целое число. При этом каждый разряд ячейки содержит одну цифру двоичной записи числа. У наименьшего представимого таким способом числа все разряды-цифры равны 0, это машинное представление числа 0. У наибольшего числа все разряды-цифры равны 1, это представление числа $2^k - 1$.

Другие представления чисел – числа со знаком – используются для того, чтобы работать и с положительными, и с отрицательными числами. В этом случае можно работать с числами из диапазона от -2^{k-1} до $2^{k-1} - 1$. Числа со знаком представляются в дополнительном коде:

$$\text{Доп}(x) = x \bmod 2^k,$$

другими словами,

$$\text{Доп}(x) = \begin{cases} x, & \text{если } x \geq 0, \\ 2^k - x, & \text{если } x < 0; \end{cases}$$

Таким образом, вместо числа x в ячейке хранится его дополнительный код (разумеется, в двоичной системе). Например, если количество разрядов в ячейке равно $k=3$, в виде чисел со знаком представимы все числа от $-2^{3-1} = -4$ до $2^{3-1} - 1 = 3$. В такой ячейке число -3 представляется в виде:

$$\text{Доп}(-3) = (-3) \bmod 2^3 = 5 = (101)_2$$

Использование дополнительного кода даёт следующую выгоду: сложение и вычитание чисел со знаком можно выполнять по обычным алгоритмам сложения и вычитания в столбик, по которым выполняются эти операции для чисел без знака. В самом деле, по свойству операции взятия остатка **mod**,

$$\text{Доп}(x \pm y) = (x \pm y) \bmod 2^k = (x \bmod 2^k \pm y \bmod 2^k) \bmod 2^k = (\text{Доп}(x) \pm \text{Доп}(y)) \bmod 2^k.$$

Таким образом, в процессоре достаточно иметь одну электронную схему для сложения и одну – для вычитания. С операциями умножения и деления дело обстоит сложнее. Умножение чисел со знаком и умножение чисел без знака выполняются по разным алгоритмам, следовательно, в процессоре должна быть схема, выполняющая умножение чисел со знаком, и схема, выполняющая умножение чисел без знака. Та же ситуация с делением. Собственно, именно поэтому в машинном языке и есть две команды умножения – для чисел со знаком и для чисел без знака – и есть две команды деления.

Задачи

№ 1. Пусть размер машинной ячейки равен 7 разрядам. Выписать в десятичной системе наибольшее и наименьшее числа со знаком, которые можно представить в такой ячейке в дополнительном коде, и наибольшее и наименьшее числа без знака, представимые в такой ячейке.

Решение: Рассмотрим сначала числа без знака. Самое маленькое число в двоичной записи – это $000\,0000_2 = 0_{10}$. Самое большое число равно $2^7 - 1 = 127_{10}$ (т.е. $111\,1111_2$). Перейдём теперь к числам со знаком. Самое маленькое отрицательное число равно $-2^6 = -64_{10}$ (это машинное число $100\,0000_2$); самое большое число $2^6 - 1 = 63_{10}$ (машинное число $011\,1111_2$).

Ответ:	наименьшее число	наибольшее число
Со знаком	-64	+63
Без знака	0	127

№ 2. Пусть размер машинной ячейки равен 9 разрядам. Выписать в десятичной системе наибольшее и наименьшее числа со знаком, которые можно представить в такой ячейке в дополнительном коде, и наибольшее и наименьшее числа без знака, представимые в такой ячейке. Дать (в двоичной системе) машинное представление числа -113 .

Решение: Первая часть решения полностью аналогична решению предыдущей задачи. Рассмотрим, как получить машинное представление числа -113 . Переводить число в двоичную систему проще через

16-ричную систему. Напомним, что при переводе в двоичную систему каждая 16-ричная цифра заменяется четвёркой двоичных цифр. Не забудем, что в результате должно остаться 9 двоичных цифр, лишние нули в начале числа отбросим.

$$\begin{aligned}\text{Доп}(-113) &= (-113) \bmod 512 = 399 = 256 + 128 + 15 \\ &= 1 \cdot 16^2 + 8 \cdot 16 + 15 = 18F_{16} = 0001\ 1000\ 1111_2\end{aligned}$$

Есть другой способ получения дополнительного кода – достаточно выполнить три шага:

1. Записать 113 в двоичной системе (используя 9 разрядов):

$$113 = 16 \cdot 7 + 1 = 0\ 0111\ 0001_2$$

2. Инвертировать все разряды: 1 1000 1110

3. Прибавить 1 к полученному числу: 1 1000 1111

Ответ: 1)	наименьшее число	наибольшее число
Со знаком	-256	+255
Без знака	0	511

2) Машинное представление $\text{Доп}(-113) = 1\ 1000\ 1111$

2.2. Различные архитектуры ЭВМ

В разделе собраны задачи на анализ и сравнение различных архитектур. Для решения задач необходимо знать устройство процессора и оперативной памяти, понимать алгоритм работы процессора, знать основные понятия, достоинства и недостатки различных архитектур ЭВМ. Заметим, что в лекциях на втором и на третьем потоках трёхадресная машина УМ-3 рассказывается в разных вариантах, однако для понимания курса эти различия не имеют существенного значения.

Задачи

№ 1. Для учебной трёхадресной ЭВМ УМ-3 дать определение следующих понятий:

а) адрес ячейки; б) машинное слово; в) код операции.

Ответ:

а) Адрес ячейки есть порядковый номер ячейки в оперативной памяти.

б) Машинное слово есть содержимое ячейки памяти (может быть командой или числом).

в) Код операции есть целое число, он указывается в машинной команде и задаёт выполняемую машинную операцию.

№ 2. Для учебной трёхадресной ЭВМ УМ-3 дать определение следующих понятий:

а) регистр; б) регистр счётчика адреса; в) регистр команд.

Ответ:

а) Регистр есть ячейка, которая находится не в оперативной памяти, а в другом устройстве (ЦП, устройстве ввода/вывода и т.д.)

б) Счётчик адреса – регистр в центральном процессоре, в нём хранится адрес следующей команды, т.е. команды, которая будет выполняться на следующем такте работы процессора (EIP).

в) Регистр команд – регистр центрального процессора, содержит текущую выполняемую команду.

№ 3. Как в ЭВМ определяется, что в данный момент находится в ячейке памяти – данное (число) или команда?

Ответ: Здесь нужно указать два момента.

1. По внешнему виду содержимого ячейки в памяти нельзя различить команды и данные.

2. Если содержимое ячейки поступает в регистр команд (РК), оно трактуется как команда, а когда содержимое ячейки поступает на регистры операндов (в АЛУ), оно трактуется как данное.

№ 4. Как в ЭВМ определяется, какое число – со знаком или без знака – находится в ячейке?

Ответ: В ответе важно указать два факта.

1. По внешнему виду содержимого ячейки невозможно отличить число со знаком от числа без знака.

2. Как трактовать содержимое ячейки – как знаковое или беззнаковое число – определяется кодом операции команды.

№ 5. Указать формат и правила выполнения команды вычитания в одноадресной машине УМ-1. Объяснить используемые обозначения.

Ответ: Формат команды

КОП	A
-----	---

Команда выполняется по правилу:

1) В регистр второго операнда R2 арифметико-логического устройства АЛУ записывается содержимое ячейки оперативной памяти с адресом A: $R2 := ОП[A]$.

2) Из содержимого регистра сумматора S устройства АЛУ вычитается содержимое регистра R2 (результат записывается в S): $S := S - R2$; устанавливается признак результата (флаги).

Можно сформулировать алгоритм выполнения команды короче:

1. $R2 := ОП[A]$

2. $S := S - R2$, установка признака результата (флагов).

При этом обязательно нужно написать, что такое R2, ОП[A] и S.

№ 6. Указать формат команды сложения и правила выполнения этой команды в стековой (безадресной) машине УМ-С (УМ-0). Все используемые обозначения объяснить.

Ответ: Формат команды

КОП

Команда выполняется по правилу

1) ИзСтека (R2). Чтение второго слагаемого из стека на регистр второго операнда R2 устройства АЛУ.

2) ИзСтека (R1). Чтение первого слагаемого из стека на регистр первого операнда R1 устройства АЛУ.

3) $S := R1 + R2$. Сложение, (S – регистр сумматор), установка признака результата (флагов).

4) ВСтек (S). Запись результата из сумматора в стек.

№ 7. В арифметических командах стековой (безадресной) машины УМ-С (УМ-0) не указывается ни одного адреса. Как в этих командах определяются операнды?

Ответ: Есть договорённость, что операнды хранятся в стеке; при выполнении арифметической команды операнды читаются из стека и результат операции записывается в стек.

№ 8. Сравните одноадресную (УМ-1) и трёхадресную (УМ-3) учебные машины по следующим критериям: удобство программирования и быстродействие.

Ответ:

1. В УМ-1 программировать труднее, чем в УМ-3, т.к. у всех команд один из операндов – регистр сумматора, приходится следить, чтобы он содержал нужное значение.

2. Команды в УМ-1 выполняются быстрее, чем аналогичные команды в УМ-3, т.к. в УМ-1 нужно выполнить одно обращение в ОП за операндом, а в УМ-3 – три обращения: чтение первого и второго операндов и запись результата.

№ 9. Выписать для стековой учебной машины фрагмент программы (не более 6 команд), реализующий присваивание $res := a + b * c$ при следующем распределении памяти: res – 0100, a – 0102, b – 0104, c – 0106. Коды операций: 00 – запись в стек, 10 – чтение из стека, 01 – сложение, 03 – умножение.

Решение: Сначала нужно преобразовать формулу в постфиксную форму: $a \ b \ c * +$. Теперь программа выписывается непосредственно по этому выражению: переменной соответствует команда записи этой переменной в стек, знаку операции – машинная команда, выполняющая её

00	0102	в стек a
00	0104	в стек b
00	0106	в стек c
03		$b * c$
01		$a + (b * c)$
10	0100	из стека res

№ 10. Выписать фрагмент программы (не более 8 команд) для безадресной (стековой) учебной машины УМ-С, реализующий присваивание $A := A^2 + B^2$ при следующем распределении памяти: А – 0200, В – 0204. (Коды операций: 00 – запись в стек, 10 – чтение из стека, 30 – дублирование верхнего элемента стека, 01 – сложение, 03 – умножение).

Решение: Выпишем постфиксную форму выражения в правой части присваивания $A^2 + B^2 = A \cdot A + B \cdot B$

При программировании вычислений квадратов вместо пересылки операнда из памяти будем использовать стековую команду дублирования вершины стека – она короче. Получаем программу

			стек, вершина справа
00	0200	в стек А	А
30		дублирование	А, А
03		умножение	A^2
00	0204	в стек В	A^2 , В
30		дублирование	A^2 , В, В
03		умножение	A^2 , B^2
01		сложение	$A^2 + B^2$
10	0200	из стека А	

№ 11. Выписать фрагмент программы (не более 7 команд) для одноадресной учебной машины УМ-1, реализующий присваивание $A := A^2 + B^2$ при следующем распределении памяти: А – 0200, В – 0204. (Коды операций: 00 – запись в сумматор, 10 – чтение из сумматора, 01 – сложение, 03 – умножение).

Решение: Организуем вычисления так

$A := A * A; S := B * B; S := S + A; A := S$

В команде умножения один из операндов должен быть сумматор S, поэтому реализация первого оператора раскладывается на три машинные команды. Результирующая машинная программа выглядит так

00	0200	$S := A$
03	0200	$S := S * A \{=A^2\}$
10	0200	$A := S$
00	0204	$S := B$
03	0204	$S := S * B \{=B^2\}$
01	0200	$S := S + A \{=B^2 + A^2\}$
10	0200	$A := S$

№ 12. Дать определение понятия «сагомодифицирующаяся программа». Привести пример задачи, при решении которой в трёхадресной машине УМ-3 необходима сагомодифицирующаяся программа.

Ответ:

1. Сагомодифицирующаяся программа (сагомодифицирующийся код) – программа, которая изменяет свой код в процессе работы.

2. Пример задачи – любая задача на работу с достаточно большими массивами. Например, вычислить сумму элементов массива из 20 чисел. В машинной команде, которая обрабатывает очередной элемент массива, на новой итерации цикла при переходе к следующему элементу должно изменяться поле адреса операнда.

2.3. Синтаксис Ассемблера

В этом разделе собраны задачи на проверку правильности синтаксиса команд на языке Ассемблера. При решении таких задач нужно учитывать архитектурные требования (в частности, специфику системы команд процессора) и особенности работы транслятора. В системе команд процессора существенны два требования: у команд не может быть двух явных операндов из оперативной памяти; размер операндов команды должен быть одинаковым (если один из операндов – байт, другой тоже должен быть байтом; если один из операндов команды – слово, другой должен быть словом и т.д.).¹ Кроме того, семантика

¹ В 32-битной архитектуре есть специальные команды для записи короткого операнда в более длинный с необходимым (знаковым или беззнаковым) расширением. В задачах данного раздела такие команды не встречаются.

машинной команды накладывает ограничения на операнды, например, у команды сложения первый операнд не может быть константой.

Транслятор при построении машинной команды должен иметь возможность определить размер операндов ассемблерной команды. Таким образом, если один из операндов ассемблерной команды – константное выражение с небольшим значением, а другой операнд – адресное выражение, в котором нет имени переменной, требуется указать тип адресного выражения с помощью оператора **ptr**.

Синтаксис некоторых команд нужно просто запомнить.

Задачи

№ 1. Пусть есть описания

X **dd** ?

B **db** ?

Из указанных ниже команд вычеркнуть те, что записаны с ошибкой:

- | | |
|-------------------------------|----------------------------------|
| 1) mul [EBX] | 7) shr CX, CL |
| 2) add EAX, [EBX, ESI] | 8) jmp dword ptr B |
| 3) jmp X | 9) add BX, B |
| 4) xor X, 'X' | 10) jmp B[EBX] |
| 5) adc X, ESP | 11) imul 300 |
| 6) lea EDI, X[EBP+2] | 12) jmp EAX |

Решение:

- 1) Ошибка: неизвестный размер операнда – байт, слово или двойное слово?
- 2) Ошибка – запятая (в записи второго операнда) не может встречаться в выражении.
- 3) Верно, команда косвенного перехода.
- 4) Верно, первый операнд – переменная, второй операнд – константа; формат m32, i32.
- 5) Верно, первый операнд – переменная, второй операнд 32-разрядный регистр общего назначения; формат m32, r32.
- 6) Верно, вычислить исполнительный адрес второго операнда и записать в регистр EDI.
- 7) Верно.
- 8) Верно – косвенный переход.
- 9) Ошибка: разный размер операндов (слово-байт).
- 10) Ошибка: недопустимый размер операнда команды перехода; адрес перехода должен быть либо словом, либо двойным словом.
- 11) Ошибка: у команды умножения операнд не может быть непосредственным.
- 12) Верно – команда косвенного перехода.

№ 2. Пусть есть описания

B **db** ?

W **dw** ?

Из указанных ниже команд вычеркнуть те, что записаны с ошибкой:

- | | |
|---------------------------|-------------------------------|
| 1) mov AX, B-W | 7) add ESP, W-B |
| 2) add ES, W | 8) jmp EIP |
| 3) mov [EBX]+2, 7 | 9) mov DX, 'X' |
| 4) xchg EBX, [EBX] | 10) sub AX, DS |
| 5) sbc AX, 'X' | 11) add 5[ESI][EBX], 5 |
| 6) shl DX, 5 | 12) xchg BL, [BL] |

Решение:

- 1) Верно: запись в 16-разрядный регистр константы со значением -1 (это разность адресов B и W); формат r16, i16.
- 2) Ошибка: сегментный регистр нельзя использовать в арифметических командах (сегментные регистры не относятся к регистрам общего назначения).
- 3) Ошибка: неизвестный размер операндов – байт, слово или двойное слово?
- 4) Верно: обмен значения регистра и двойного слова из памяти; формат r32, m32.
- 5) Ошибка: неверный мнемокод, нет такой команды.

- 6) Верно.
- 7) Верно: прибавление к 32-разрядному регистру общего назначения константы 1.
- 8) Ошибка: регистр EIP нельзя указывать ни в одной команде.
- 9) Верно; формат r16,i16.
- 10) Ошибка: сегментный регистр нельзя использовать в арифметической команде.
- 11) Ошибка: первый операнд – адресное выражение; второй операнд – константа, размер операндов неизвестный.
- 12) Ошибка: неверный второй операнд, BL – не регистр-модификатор.

№ 3. Из указанных ниже команд вычеркнуть те, что записаны с ошибкой:

- | | |
|--|------------------------------|
| 1) cmp 2,CX | 8) shl AX,BX |
| 2) sub AX,['*'] | 9) div AX |
| 3) mov CS,AX | 10) add [ESI] ['*'],1 |
| 4) mov 4[EBX],500 | 11) cmp 20,BX |
| 5) push ESP | 12) add EBX,[5] |
| 6) mul AX | 13) add SP,'?' |
| 7) add [ESI], byte ptr [EBX] | |

Решение:

- 1) Ошибка: первый операнд не может быть константой (у команды **cmp** требования на операнды такие же, как у команды **sub**).
- 2) Верно, операнды формата r16,i16. Запись в скобках ['*'] эквивалентна записи '*'.
- 3) Ошибка: нельзя изменять регистр CS командой **mov**.
- 4) Ошибка: неизвестный размер операндов – слово или двойное слово?
- 5) Верно.
- 6) Верно: <DX:AX> := AX*AX.
- 7) Ошибка: два операнда из памяти.
- 8) Ошибка: второй операнд должен быть константой или регистром CL.
- 9) Верно: AX:=<DX:AX> **div** AX; DX:=<DX:AX> **mod** AX.
- 10) Ошибка: неизвестный размер операндов: 1 длиной 1, 2 или 4 байта ?
- 11) Ошибка: первый операнд не может быть константой.
- 12) Верно: прибавление к 32-разрядному регистру общего назначения константы 5; формат r32,i32.
- 13) Верно; формат r16,i16.

№ 4. Пусть X – имя переменной типа **word**. Вычеркнуть синтаксически неверные команды.

- | | |
|-------------------------------------|---------------------------------------|
| 1) cmp type X,2 | 8) and BL,352 |
| 2) sub X, word ptr AL | 9) mov BP,AX |
| 3) mov DS,[DI] | 10) inc 300[EBX] |
| 4) neg [X] | 11) sub AL, byte ptr BX |
| 5) xchg AX,[EBX] | 12) mov DX,[ECX] |
| 6) shr AX | 13) and BX,'0' |
| 7) dec 400[EDI] | 14) mov ESP,EAX |

Решение:

- 1) Ошибка: первый операнд не может быть непосредственным (константным выражением).
- 2) Ошибка: оператор **ptr** нельзя применять к регистрам.
- 3) Верно ¹.
- 4) Верно. Запись [X] эквивалентна записи X; формат m16.
- 5) Верно; формат r16,m16.
- 6) Ошибка: пропущен второй операнд.

¹ В плоской модели памяти сегментные регистры менять хоть и можно, но опасно; это может привести к непредсказуемым последствиям.

- 7) Ошибка: неизвестный размер операнда – байт, слово или двойное слово?
- 8) Ошибка: разный размер операндов: первый операнд – байт, второй – слово.
- 9) Верно; формат `r16,r16`.
- 10) Ошибка: неизвестный размер операнда.
- 11) Ошибка: оператор `ptr` нельзя применять к регистру.
- 12) Верно; формат `r16,m16`.
- 13) Верно; формат `r16,i16`.
- 14) Верно; формат `r32,r32`.

2.4. Представление чисел в процессоре Intel, арифметические флаги

В архитектуре Intel размер ячейки 8 разрядов (байт). Для хранения данных и команд используют одну, две или четыре расположенные подряд ячейки – байт, слово или двойное слово. Числа хранятся в перевёрнутом виде: в слове и в двойном слове младшие разряды числа находятся в байтах с меньшими адресами. Адресом числа считается адрес его младшего байта. Можно работать с числами без знака и с числами со знаком – в дополнительном коде. При выполнении сложения и вычитания помимо вычисления результата устанавливаются значения арифметических флагов: `CF` сигнализирует о переполнении при работе с числами без знака, `OF` сигнализирует о переполнении при работе со знаковыми числами, флаг `SF` отражает знак машинного результата (равен старшему разряду машинного результата), `ZF` говорит о том, что машинный результат равен нулю. Если в задаче требуется определить значения, которые получают флаги в результате выполнения операции, необходимо учитывать диапазоны беззнаковых и знаковых чисел для того размера чисел, который используется в задаче. Полезно также знать величины соответствующих степеней двойки.

Байт: $2^8 = 256$; числа без знака $[0..255]$, числа со знаком $[-128..127]$.

Слово: $2^{16} = 65\,536$; числа без знака $[0..65\,537]$, числа со знаком $[-32\,768..32\,767]$.

Двойное слово: $2^{32} = 4\,294\,967\,296$; числа без знака $[0..4\,294\,967\,295]$,

числа со знаком $[-2\,147\,483\,648..2\,147\,483\,647]$.

Для двойных слов обычно не требуется знание десятичных границ, достаточно понимать, как выглядит минимальное и максимальное машинное число и уметь записать эти числа в 16-ричной системе.

Флаги используются для проверки выполнимости условия при выполнении команд условных переходов.

Задачи

№ 1. Указать значения регистра `AL` (в виде знакового десятичного числа) и флагов `CF`, `OF`, `SF` и `ZF`, полученные после выполнения следующих двух команд:

```
mov AL, -55
add AL, 170
```

Решение: Команды выполняют операцию сложения над операндами-байтами: $(-55) + 170$.

Определим значение `CF`. Для этого запишем оба операнда в виде чисел без знака. Числу -55 соответствует число $256 - 55 = 201$. Таким образом, для чисел без знака выполняется операция $201 + 170 = 371$. Имеем $371 > 255$, следовательно, значение `CF` = 1. Сейчас удобно определить машинный результат – число без знака: $371 \bmod 256 = 115$. Поскольку $115 < 128$, соответствующее число со знаком тоже будет 115.

Определим значение `OF`, для этого операнды нужно представить как числа со знаком. Числу 170 соответствует отрицательное число $-(256 - 170) = -86$. Имеем операцию $(-55) + (-86) = -141$, это число не представимо в виде числа со знаком в байте (меньше, чем -128), значит, `OF` = 1.

Результат как число со знаком (115) положителен, следовательно, флаг `SF` = 0.

Результат в регистре `AL` $\neq 0$, значит, `ZF` = 0.

Ответ: `AL` = 115, `CF` = 1, `OF` = 1, `SF` = 0, `ZF` = 0

№ 2. Указать значения регистра ВН (в виде знакового десятичного числа) и флагов CF, OF, SF и ZF, которые они будут иметь после выполнения следующих двух команд:

```
mov BH, 160
sub BH, -56
```

Решение: Выполняется вычитание $160 - (-56)$, операнды – байты.

Определим CF. Представляя операнды в виде чисел без знака, получим $\text{Доп}(-56) = 256 - 56 = 200$; пример для чисел без знака будет выглядеть так $160 - 200 = -40$. Результат отрицательный, следовательно, не представим в виде числа без знака, значит, CF=1.

Определим OF. Числу 160 соответствует отрицательное число $-(256 - 160) = -96$. Для чисел со знаком пример запишется как $-96 - (-56) = -40$. Результат принадлежит допустимому диапазону для формата байта $[-128, 127]$, следовательно, OF=0.

Знак машинного результата минус, значит, SF=1.

Результат не равен нулю, значит, ZF=0.

Ответ: ВН=-40, CF=1, OF=0, SF=1

№ 3. Указать значения флагов CF, OF, SF и ZF, которые получатся после выполнения следующих двух команд:

```
mov AL, -56
cmp AL, 170
```

Решение: Команда сравнения **cmp** AL, 170 работает как вычитание **sub** AL, 170, только без сохранения результата. Следовательно, нужно определить, как будут установлены флаги в результате вычитания $(-56) - 170$.

Для получения CF представим оба операнда в виде чисел без знака. Числу (-56) соответствует число $256 - 56 = 200$. Пример запишется как $200 - 170 = 30$. Результат принадлежит допустимому для чисел без знака диапазону $[0, 255]$, следовательно, CF=0.

Для определения флага OF оба операнда нужно представить в виде чисел со знаком. При этом числу 170 соответствует число со знаком $-(256 - 170) = -86$. Имеем $(-56) - (-86) = 30$. Результат принадлежит допустимому для чисел со знаком диапазону $[-128, 127]$, следовательно, OF=0.

Машинный результат как число со знаком (30) положителен, значит, SF=0. Машинный результат не равен нулю, значит, ZF=0.

Ответ: CF = 0, OF = 0, SF = 0, ZF = 0.

№ 4. Объяснить, почему для вывода знаковых и беззнаковых целых чисел используются две разные макрокоманды (**outi** и **outu**), в то время как для ввода целых чисел используется только одна макрокоманда (**inint**)?

Ответ: По содержимому ячейки (регистра) нельзя определить, какое это число – без знака или со знаком. Поэтому программист должен указать при печати, как трактовать содержимое ячейки (регистра), выбирая соответствующую макрокоманду.

При вводе чисел отрицательные числа отличаются от положительных – перед отрицательным числом стоит знак минус. Если встретился минус, значит нужно прочитать отрицательное число со знаком и построить его машинное представление (в дополнительном коде). Для положительных чисел алгоритм построения машинного представления чисел со знаком и чисел без знака один и тот же (это перевод числа в двоичную систему). Таким образом, обработка входной последовательности однозначно определена её видом, поэтому макрокоманда ввода сможет распознать, какой алгоритм запустить. Поэтому достаточно одной макрокоманды ввода.

№ 5. Почему в процессоре Intel есть одна машинная команда перехода "по равно" (**je**), а машинных команд перехода "по меньше" – две (**j1** и **jb**)?

Ответ: Сравнение чисел $a \vee b$ реализовано в процессоре как вычитание $a - b$ и последующее сравнение результата с нулём. Разность $a - b$ равна нулю, если $a = b$, независимо от того, рассматриваются ли операнды как числа со знаком или как числа без знака; о равенстве нулю сигнализирует флаг ZF=1. Поэтому достаточно одной команды перехода по условию "равно" для чисел со знаком и для чисел без знака.

При проверке условия $a < b$ нужно проанализировать, отрицателен ли результат вычитания $a - b$. Если сравниваются числа без знака, об отрицательности математического результата сигнализирует флаг $CF=1$; если же сравниваются числа со знаком, о знаке математического результата сигнализируют флаги OF и SF . Поэтому команда перехода по "меньше" должна анализировать разные комбинации флагов в зависимости от того, с какими числами идёт работа. Соответственно, есть две машинные команды.

2.5. Арифметические команды

В раздел включены задачи на программирование вычислений по формулам, в особенности, на применение команд умножения и деления. Эти команды довольно запутанны, необходимо разобраться в тонкостях их работы. При программировании нужно обращать внимание на размер промежуточных результатов вычислений, чтобы выделить достаточно места для них. Использование команды деления с неподходящим размером операнда считается серьёзной ошибкой при оценке решений задач по теме арифметика.

Задачи

№ 1. Что будет напечатано в результате выполнения последовательности команд

```
mov  AX, -936
cwd
mov  BX, 3
idiv BX
outu AX
```

Решение: После команды **cwd** в регистровой паре $\langle DX:AX \rangle$ содержится двойное слово – отрицательное число -936 . Деление со знаком на 3 даст в AX частное -312 . Макрокоманда **outu** трактует содержимое операнда как число без знака. Если считать содержимое AX числом без знака, то получим $2^{16} - 312 = 65536 - 312 = 65224$.

Ответ: 65224

№ 2. Что будет напечатано в результате выполнения последовательности команд

```
mov  AX, 342
mov  BL, 2
div  BL
cbw
outi AX
```

Решение: Беззнаковое деление числа 342 на $BL=2$ даст частное $AL=171$. Команда **cbw** расширяет содержимое AL до AX , трактуя AL как число со знаком, т.е. как отрицательное число $-(2^8 - 171) = -(256 - 171) = -85$. Таким образом, после команды **cbw** регистр AX содержит число -85 . Макрокоманда **outi** печатает это число.

Ответ: -85

№ 3. Пусть описаны переменные

```
X db ? ; число со знаком
Y dd ?
```

Выписать последовательность команд (не более 8), реализующих присваивание

$Y := 11 * X^2 - X$

Решение: В задачах на программирование вычислений важно учитывать размеры промежуточных результатов, чтобы не потерять старшие разряды суммы или произведения. Организуем вычисление по схеме Горнера $y = 11 * X^2 - X = ((11 * X) - 1) * X$. Результат умножения байта X на число 11 уместится в слово, поэтому можно просто перемножить два байта. Причём произведение будет достаточно маленьким по модулю, так что вычитание $(11 * X) - 1$ не приведёт к переполнению при работе со словами. У последней операции умножения первый сомножитель – слово, значит, второй сомножитель X нужно привести к размеру слова с помощью команды знакового расширения. Результат последнего умножения уже превысит размер слова и будет двойным словом $\langle DX:AX \rangle$, старшие разряды произведения попадут в DX , младшие – в AX . При записи результата в переменную Y нужно учесть, что числа хранятся в перевёрнутом виде, младшие разряды по меньшим адресам. Таким образом, получаем следующую последовательность команд

```

mov     AL, 11
imul    X
sub     AX, 1
movsx   BX, X 1
imul    BX
mov     word ptr Y, AX
mov     word ptr Y+2, DX

```

№ 4. Пусть описаны переменные

```

X dw ?; число со знаком
Y dw ?

```

Выписать последовательность команд, реализующих оператор

```
Y := 100 - (X div 11) * (X mod 11)
```

Решение: В этой задаче, как и в предыдущей, важно учитывать размер промежуточных результатов. Казалось бы, переменная X занимает слово, число 11 уместится в байт, можно записать X в AX, число 11 в BL и применить команду **idiv** BL. Эта команда помещает остаток от деления в AH, а частное – в AL. Однако, если значение X достаточно большое, например, 22000, частное при делении на 11 будет 2000 и не поместится в AL, возникнет ошибка переполнения при делении. Следовательно, чтобы корректно разделить слово X на число 11, нужно выделить для хранения частного регистр AX. Таким образом, нужно записать значение X в двойное слово <DX:AX> как число со знаком, записать 11 в слово (например, в регистр BX) и разделить двойное слово на слово. Далее, поскольку $X \bmod 11 < 11$, получаем

$$(X \bmod 11) * (X \bmod 11) < (X \bmod 11) * 11 = X,$$

значит, результат умножения целиком умещается в слово (в AX). Получаем программу

```

mov     AX, X
cwd
mov     BX, 11
idiv    BX ; AX := X div 11, DX := X mod 11
imul    DX ; <DX:AX> := AX * DX, в DX нет значащих цифр
mov     BX, 100
sub     BX, AX
mov     Y, BX

```

2.6. Массивы, записи, полная программа, стек

При работе с массивами важно учитывать размер элементов массива (тип). От типа зависит адресация элементов относительно начала массива: если X – массив байтов, Y – массив слов, Z – массив двойных слов, то адреса вторых элементов этих массивов будут X+1, Y+2, Z+4. Если в программе происходит перебор элементов массива с помощью регистра-модификатора, для перехода к следующему элементу массива надо изменить значение модификатора на 1 для массива X, на 2 – для массива Y и на 4 – для Z.

Решения задач раздела даны для архитектуры IA-32, используются развитые возможности записи адресных выражений с модификаторами. Варианты решений для 16-разрядной архитектуры Intel-8086 даны в примечаниях.

Задачи

№ 1. Имеется описание

```

N equ 1000
X dw N dup(?); X[1..N]

```

¹ В 16-разрядной архитектуре нет команды записи с расширением; нужно заменить эту строку на последовательность команд

```

mov     BX, AX; сохраняем промежуточный результат
mov     AL, X
cbw

```

В варианте задачи для старого курса ограничение на количество команд в решении – 10.

Пусть i – константа со значением от 1 до N , все числа $X[i]$ лежат в диапазоне от 1 до N . Выписать последовательность команд (не более трёх), реализующих действие $X[X[i]] := -1$.

Решение: Каждый элемент массива X занимает два байта, значит, смещение элемента относительно начала массива в два раза больше индекса элемента. Следует учесть, что нумерация элементов начинается с 1, т.е. смещение элемента $X[1]$ относительно начала массива равно 0. Следовательно, смещение элемента $X[i]$ относительно начала массива равно $2 \cdot (i-1) = 2 \cdot i - 2$. Далее, для того, чтобы использовать значение $X[i]$ в качестве индексного выражения, это значение нужно записать в регистр-модификатор (например, EBX), расширив как число без знака, и преобразовать в смещение элемента относительно начала массива по указанной формуле. Получаем решение

```
movzx  EBX, X+2*i-2
mov     X[2*EBX-2], -1
```

В примечании дано решение для архитектуры Intel-8086.¹

№ 2. Пусть есть описания

```
Time record h:5=0,m:6=0
t      Time <>
```

Написать последовательность команд для решения следующей задачи. Переменная t типа `Time` содержит текущий момент времени – час и минуту. Вычислить и напечатать количество минут, прошедших с начала суток.

Решение: Фактически, нужно вычислить значение выражения $t.h \cdot 60 + t.m$, где $t.h$ и $t.m$ – значения соответствующих полей записи t . Первое произведение превышает размер байта, но уместается в слово, поэтому будем умножать байты. Сама запись t занимает 11 битов, т.е. слово. Для выделения поля $t.h$ запись нужно сдвинуть вправо на h (константа h равна 6) разрядов. Для выделения поля $t.m$ нужно логически умножить запись на маску `mask m` (значение выражения `mask m` равно `3Fh`). Ниже приведено решение для архитектуры IA-32; как изменится решение для архитектуры Intel-8086 сказано в примечании.²

```
mov  AX, t
shr  AX, h      ; AL:=t.h
mov  BL, 60
mul  BL        ; AX:=t.h*60
mov  BX, t
and  BX, mask m; BX:=t.m
add  AX, BX
outu AX
```

№ 3. Выписать полную программу на Ассемблере, которая вводит (с помощью `inint`) целое неотрицательное число N в формате слова и выводит ближайшее к N число, кратное 5. (Считать, что такое число меньше, чем 2^{16} .)

Решение: Число N представимо в виде $N = 5q + r$, где $0 \leq r < 5$; q – частное, а r – остаток от деления N на 5. Ближайшим к N числом, кратным 5, будет либо $5q$, либо $5(q+1)$, в зависимости от значения r . Если $r < 3$, искомым числом будет $5q$, иначе – число $5(q+1)$.

Для вычисления остатка r надо разделить N на 5. Определим размер операндов деления. Если N достаточно велико, частное `N div 5` может быть больше 255, оно не уместится в байт. Значит, частное нужно записать в слово. Следовательно, делить надо двойное слово на слово. Получаем следующую программу ("решение в лоб")

```
include console.inc      |      cmp  DX, 3
```

¹ В архитектуре Intel-8086 использовались 16-разрядные регистры-модификаторы; множители при модификаторах запрещены. Решение в этом случае выглядит так

```
mov  BX, X+2*i-2
add  BX, BX
mov  X[BX-2], -1
```

² В процессоре Intel-8086 команды сдвигов не допускают произвольное число в качестве второго операнда, поэтому вместо команды `shr AX, h` нужно использовать две команды

```
mov  CL, h
shr  AX, CL
```

<pre>.code Start: inint AX mov DX, 0 mov CX, 5 div CX; DX:=N mod 5</pre>	<pre> jb L inc AX L: mul CX outu AX exit end Start</pre>
---	--

Искомый ответ, однако, можно вычислить и по формуле $K := ((N+2) \text{ div } 5) * 5$, тогда

<pre>include console.inc .code Start: inint AX mov DX, 0 mov CX, 5 add AX, 2</pre>	<pre> adc DX, 0 div CX; AX:=(N+2) div 5 mul CX outu AX exit end Start</pre>
--	--

В примечании дано решение для старого курса «Архитектура ЭВМ и язык ассемблера». ¹

№ 4. Пусть стек содержит не менее одного двойного слова. Написать фрагмент программы на Ассемблере (не более 3-х команд), изменяющий верхний элемент стека X (двойное слово, число без знака) по правилу $X := X \text{ div } 2 + 1$. Регистры не менять, остальные данные в стеке не портить.

Решение: На верхний элемент (вершину) стека указывает регистр ESP. Для доступа к верхнему двойному слову используем модификацию по ESP; напомним, что в командах необходимо указывать размер операнда **dword ptr**. Деление на 2 проще всего реализуется с помощью команды сдвига вправо **shr**. Получаем фрагмент

```
shr dword ptr [ESP], 1; X:=X div 2
inc dword ptr [ESP]
```

2.7. Процедуры и функции

Предварительное замечание: в ассемблере процедуры и функции описываются одинаково, директивами **proc** и **endp**. Разница между ними в том, что функция возвращает значение. Поэтому далее в тексте слово «процедура» используется как обобщающее название процедур и функций.

При решении задач по теме «процедуры» важно понимать механизм передачи параметров-значений и механизм передачи параметров-переменных; уметь обрабатывать оба способа передачи. Неверная работа в процедуре с параметром, переданным по ссылке, является серьезной ошибкой.

В новой программе курса формулируется соглашение о связях между программой и процедурой на основе соглашения **stdcall**:

- 1) Процедура не меняет своего окружения – не портит значения регистров, не работает с переменными программы по именам.
- 2) Параметры передаются в стеке.
- 3) Программа помещает параметры в стек в обратном порядке – сначала последний параметр, затем предпоследний и т.д., первый параметр оказывается верхним в стеке.
- 4) Локальные переменные процедуры размещаются в стеке.

¹ Решение задачи 3 для старого курса. Различия состоят в оформлении программы – в записи программных сегментов и в названиях макрокоманд ввода-вывода.

<pre>include io.asm St segment stack db 64 dup (?) St ends Code segment assume cs:Code, ss:St H: inint AX mov DX, 0 mov CX, 5</pre>	<pre> div CX cmp DX, 3 jb L inc AX L: mul CX outword AX finish Code ends endH</pre>
---	---

5) Функция возвращает результат в регистре-аккумуляторе AL, AX, EAX или в <EDX:EAX> в зависимости от типа результата.

6) Процедура очищает стек от параметров.

Для доступа к параметрам и локальным переменным в стеке используется регистр EBP. Кадр стека, относящийся к процедуре, имеет следующий вид:

база кадра →	Используемые регистры	создаёт процедура
	Локальные переменные	
	Старое EBP	создаёт программа
	Адрес возврата EIP	
	Параметры	

Необходимо аккуратно вычислять смещения параметров относительно базы стекового кадра. Для этого достаточно нарисовать, какое содержимое будет иметь стек во время работы процедуры и отметить смещения, учитывая, что элементы стека имеют размер 4 байта.

В формулировках задач используется синтаксис языка Free Pascal; соответствие между типами Free Pascal и типами Ассемблера:

int64 – **qword** (четверное слово, число со знаком);
longword, longint – **dword** (двойное слово, числа без знака и со знаком);
word, integer – **word** (слово, числа без знака и со знаком);
boolean, char, shortint, byte – **byte** (байт).

В качестве значения **false** используется число 0, а в качестве значения **true** в Ассемблере лучше использовать число 0FFh (хотя сам Free Pascal использует для **true** значение 1, и, кроме того, считает **true** любое ненулевое значение).

В программе при передаче параметров в стек нужно помещать двойные слова (расширяя, при необходимости, переменные). В процедуре из стека нужно брать параметры соответствующего размера, учитывая перевёрнутое размещение чисел в памяти.

Задачи

№ 1. Описать процедуру с именем ADD3, которая получает в качестве параметра адрес беззнаковой переменной размером в четверное слово (**dq**) и увеличивает на 3 значение этой переменной. Процедура должна выполнять стандартные соглашения о связях.

Решение. В этой задаче важно определить способ передачи параметра (по значению или по ссылке) и корректно реализовать обработку параметра. Поскольку значение параметра должно измениться в результате работы процедуры, следовательно, параметр передаётся по ссылке. Во время работы процедуры в стеке хранится адрес параметра. В процедуре надо использовать этот адрес для доступа к параметру – переменной в секции данных.

Далее, в 32-битном Ассемблере нет команд для работы с четверными словами. С данными длиной 4 слова приходится работать по частям – по двойным словам: сначала прибавить 3 к младшему двойному слову, затем учесть возможный перенос в старшие разряды. При этом в реализации сложения необходимо учитывать перевёрнутое представление чисел в памяти. Получаем процедуру

```
ADD3 proc
    push EBP
    mov EBP, ESP; База стекового кадра
    push EBX
    mov EBX, [EBP+8]; EBX:=Адрес параметра
    add dword ptr [EBX], 3; Переменная в секции данных
    adc dword ptr [EBX+4], 0
    pop EBX
    pop EBP
    ret 4
ADD3 endp
```

№ 2. Описать на Ассемблере процедуру LAST3, которая заменяет на 3 младшую десятичную цифру своего параметра. Параметр процедуры – беззнаковая переменная длиной в байт. Процедура должна удовлетворять стандартным соглашениям о связях. Привести пример вызова этой процедуры.

Решение. При программировании подобной процедуры на Паскале мы передавали бы параметр по ссылке, следовательно, и в процедуре на Ассемблере нужно работать с параметром, переданным по ссылке. При передаче параметра по ссылке программа помещает в стек адрес фактического параметра; процедура записывает этот адрес из стека в регистр-модификатор и работает с переменной в секции данных, используя косвенную адресацию.

Значит, при вызове процедуры нужно поместить в стек адрес параметра, а в самой процедуре использовать этот адрес для доступа к параметру. Возможный вызов процедуры

```
X db ?
. . .
push offset X
call LAST3
```

В процедуре вычисления организуем так: $n := (n \text{ div } 10) * 10 + 3$, здесь n обозначает параметр процедуры. Для деления байта n на байт 10 необходимо расширить n до слова. Возможный вид процедуры

<pre>LAST3 proc push EBP mov EBP, ESP push EAX push EBX push ECX mov EBX, [EBP+8]; Адрес X movzx AX, byte ptr [EBX]; AX:=X mov CL, 10 div CL</pre>	<pre>mul CL add AL, 3 mov [EBX], AL pop ECX pop EBX pop EAX pop EBP ret 4 LAST3 endp</pre>
--	--

№ 3. Описать на Ассемблере процедуру, эквивалентную процедуре

```
procedure P(var X:word; Y:word);
begin X:= X div 16 + (23 + Y) mod 32 end;
```

Параметры – числа без знака, команды деления не использовать. Процедура должна удовлетворять стандартным соглашениям о связях.

Решение. Воспользуемся тем, что деление числа без знака на степени двойки можно реализовать командами **shr** и **and**. Частное $X \text{ div } 16$ получится, если сдвинуть X вправо на 4 разряда; остаток $\text{mod } 32$ – это просто последние 5 разрядов числа.

Первый параметр процедуры передаётся по ссылке, второй – по значению. Во время работы процедуры кадр стека будет иметь вид

	Используемые регистры	
EBP→	EBP _{старое}	
EBP+4	Адрес возврата EIP	
EBP+8	Адрес X	
EBP+12	Значение Y	0

Описание процедуры

<pre>P proc push EBP mov EBP, ESP push EAX push EBX push EDX mov EBX, [EBP+8]; Адрес X mov AX, [EBX]; Значение X shr AX, 4; AX:=X div 16 mov DX, [EBP+12]; Значение Y</pre>	<pre>add DX, 23; Y:=Y+23 and DX, 1Fh; DX:=(23+Y) mod 32 add AX, DX mov [EBX], AX; запись результата в X pop EDX pop EBX pop EAX pop EBP ret 8 P endp</pre>
---	--

№ 4. **const** N=1000; **type** Mas=array[1..N] of byte;

Описать на Ассемблере функцию **Sym(X, N)** для проверки симметричности массива X длины N , соответствующую заголовку

```
function Sym(var X:Mas; N:Longword):boolean;
```

Функция должна удовлетворять стандартным соглашениям о связях. Привести на Ассемблере пример вызова функции, описав используемые переменные.

Решение: В соответствии со стандартными соглашениями о связях, параметры функции *Sym* передаются в стеке, сначала второй параметр, затем первый; функция должна вернуть логический результат в регистре *AL*, в качестве значения **false** используется 0, а в качестве значения **true** – 0FFh.¹ Для организации вызова опишем некоторый массив *A*

```
N equ 10000
A db N dup (?)
```

Вызов функции *Sym* (*A*, *N*) реализуется последовательностью команд

```
push N
push offset A
call Sym
```

Опишем теперь саму функцию

<pre>Sym proc push EBP mov EBP, ESP push EBX push ECX push EDX mov EBX, [EBP+8]; Адрес массива X mov ECX, [EBP+12]; Длина массива N lea EBP, [EBX+ECX-1]; Адрес X[N] shr ECX, 1; Макс. колич. итераций N/2 jecxz T; Для N<2 xor AL, AL; Sym:=false L:mov DL, [EBX]</pre>	<pre>cmp DL, [EBP] jne KON inc EBX dec EBP loop L T:mov AL, 0FFh; не различий, Sym:=true KON: pop EDX pop ECX pop EBX pop EBP ret 2*4; очистка стека от параметров Sym endp</pre>
---	---

Обратите внимание на важный момент: хотя внутри процедуры используется регистр *AL*, мы *не* сохраняем в стеке *EAX* в начале процедуры и *не* восстанавливаем его из стека в конце процедуры. Дело в том, что, в соответствии со стандартными соглашениями о связях, функции возвращают результат в регистре *AL* (*AX*, *EAX*); собственно, цель процедуры *Sym* – записать результат в *AL*.

2.8. Макросредства языка MASM

При решении задач по теме «макросредства» важно чётко понимать, как ассемблер взаимодействует с макрогенератором (макропроцессором). Каждое предложение программы обрабатывает Ассемблер, и, если в этом предложении есть макросредства, то вызывается макрогенератор. Макрогенератор порождает макрорасширение (новый текст), затем этот текст возвращается для обработки Ассемблером. Макрогенератору доступна информация, собранная Ассемблером при обработке расположенной выше части программы.

Одна из задач, решаемых макросредствами – перенести, насколько возможно, вычисления с этапа выполнения программы на этап компиляции. Макрогенератор может вычислять значения выражений (без ссылок вперёд). Конечно, значения переменных из секции данных или регистров макрогенератору недоступны – они становятся известны и изменяются в момент выполнения программы процессором. Это обстоятельство важно иметь в виду, решая задачи на макросы. Путаница с этапами вычисления значений – попытка анализа значений переменных на этапе макрогенерации или реализация в виде программного кода вычисления выражений, который может вычислить макрогенератор – является грубой ошибкой.

Если в условии указано ограничение на количество команд в макрорасширении, каждая лишняя команда будет штрафоваться при проверке решения.

В данном разделе формулировки задач и решения приведены в редакции для нового курса, на языке ассемблера MASM 6.14.

¹ В Ассемблере в качестве константы **true** удобнее брать байт со значением 11111111b=-1, так как тогда машинная команда **not true=false**, хотя язык Free Pascal, как и язык C, считают любое ненулевое значение **true**.

В Ассемблере есть служебные константы **byte** = 1, **word** = 2, **dword** = 4, **qword** = 8. Можно использовать их в выражениях для повышения наглядности, например, **type X EQ byte**.

Задачи

№ 1. Описать макрос **max X**, где **X** – имя знаковой переменной. Макрос реализует присваивание **X := max (X, 57)**, если тип **X** – байт или слово, иначе порождает пустое макрорасширение.

Решение: При обработке макровывода в зависимости от типа указанного фактического параметра макрогенератор должен построить либо пустое макрорасширение, либо макрорасширение вида

```
    cmp X, 57
    jle L
    mov X, 57
L:
```

Как заставить макрогенератор строить разные макрорасширения? Это делается с помощью директивы условной генерации

```
if type X EQ 1 OR type X EQ 2
```

Макрогенератор подставит фактический параметр на место **X**, вычислит его тип. Нужный нам тип – это 1 (байт) и 2 (слово), только для этих значений требуется включить в текст макрорасширения нужный фрагмент. Для того, чтобы имя метки **L** не конфликтовало с именами, описанными в программе, нужно объявить его локальным.

Приходим к следующему решению

```
max macro X
    local L
    if type X EQ 1 OR type X EQ 2
        cmp X, 57
        jle L
        mov X, 57
    L:
    endif
endm
```

№ 2. Описать макрос **SHRight Y**, где **Y** – имя переменной размером в байт, слово, двойное слово или четверное слово. Макрос реализует логический сдвиг переменной **Y** на один разряд вправо. Диагностики об ошибках обращения не выдавать. Макрорасширение не должно содержать более двух предложений Ассемблера.

Решение: Для всех типов, кроме четверного слова, сдвиг реализуется одной командой **shr Y, 1**. Четверное слово нужно сдвигать по частям: сначала сдвинуть вправо старшее двойное слово, затем, подхватив выдвинувшийся разряд, сдвинуть младшее двойное слово. В теле макроопределения нужно определить, является ли параметр четверным словом с помощью директивы условной генерации. Возможное решение:

```
SHRight macro Y
    if type Y NE 8
        shr Y, 1
    else
        shr dword ptr Y+4, 1
        rcr dword ptr Y, 1
    endif
endm
```

№ 3. Описать макрос **OddBQ X**, реализующий оператор Паскаля **if odd (x) then X := 2 * X**. Параметр макроса **X** – беззнаковая переменная размером в байт или четверное слово. Считать, что значение произведения **2 * X** не превзойдет размера переменной **X**.

Решение: Проверить, является ли значение переменной нечетным числом, проще всего по значению младшего разряда, например, применив команду **test**. Далее, в нашем Ассемблере нет команды умножения и сложения, допускающей в качестве операнда четверное слово. Реализовать умножение **2 * X** можно сложением **X + X** или с помощью сдвига **X** влево на 1 разряд. Мы будем использовать сдвиг. Если **X** – байт, достаточно одной команды, если же **X** – четверное слово, будем сдвигать его по частям: сначала сдвигаем влево младшее двойное слово, затем сдвигаем старшее двойное слово, подхватывая разряд,

который выдвинулся из младшего двойного слова. Для определения типа переменной-параметра используем директиву **if**.

```
OddBQ macro X
    local Not_Odd
    test byte ptr X,1
    jz Not_Odd
if type X EQ 8
    shl dword ptr X,1
    rcl dword ptr X+4,1
else
    shl X,1
endif
Not_Odd:
endm
```

№ 4. Описать макрос JGT *v*, *L*. Смысл параметров: фактическое значение *L* будет меткой, а в качестве значения *v* будет передаваться список $\langle v_1, v_2, \dots, v_k \rangle$, где каждое v_i – имя переменной типа **byte**, **word** или **dword**. Макрос должен делать переход на метку *L*, если значения всех переменных типа **word** из списка положительны (числа знаковые) или если в списке нет переменных типа **word**. Выписать макрорасширение для макровывоза JGT $\langle z, a, y, w \rangle$, Con, если переменные имеют следующие типы: *a* – **byte**, *y, z* – **word**, *w* – **dword**.

Решение: Сначала разберёмся с управлением. Какой текст на Ассемблере должен получить макрогенератор, раскрывая макровывозы? Представим, что все переменные имеют тип **word**. Мы должны последовательно проверить значения всех переменных; если окажется, что какое-то значение не положительно, проверку нужно прекращать, т.к. известно, что перехода на *L* не будет. На Ассемблере это выглядит следующим образом

```
cmp v1,0
jle NO
cmp v2,0
jle NO
cmp v3,0
jle NO
.
.
.
cmp vk,0; все переменные v1, v2, ..., vk-1 были > 0
jg L
NO:
```

Фрагменты, обрабатывающие все переменные, кроме последней переменной, совпадают, значит, можно записать текст короче с помощью блока повторения. Преобразуем обработку последней переменной так, чтобы внести её в блок повторения:

```
cmp vk,0      cmp vk,0
jg L          ⇒ jle NO
               jmp L
```

Теперь можно записать текст короче:

```
for P, <v1, v2, ..., vk>
    cmp P,0
    jle NO
endm
jmp L
NO:
```

Вернёмся к исходной задаче – вспомним, что анализировать значения нужно только для переменных-слов. Проверку типа переменной реализуем с помощью директивы **if**, включающей в макрорасширение соответствующие команды сравнения и перехода. Ещё момент – как передать в блок повторения список переменных? В условии сказано, что список – это значение параметра *v*, поэтому достаточно передать в блок *v*. И, наконец, вспомогательную метку *NO* необходимо объявить локальной. В итоге, приходим к описанию

```
JGT macro v, L
    local NO
    for P, <v>
        if type P EQ 2
```

```

        cmp P, 0
        jle NO
    endif
endm
        jmp L
NO:
    endm

```

Теперь решим вторую часть задачи, надо раскрыть макровывозы JGT <z, a, y, w>, Con, где a – байт, y, z – слова, w – двойное слово. Не будем разбирать пошагово процесс получения макрорасширения, напомним только, что макрогенератор вместо локального имени вставляет в макрорасширение специально построенное имя вида ??XXXX. Результат

```

    cmp z, 0
    jle ??0000
    cmp y, 0
    jle ??0000
    jmp Con
??0000:

```

№ 5. Описать макрос RSub R, X для вычитания из 32-разрядного регистра общего назначения R значения беззнаковой переменной X. Тип X – байт, слово или двойное слово. Другие регистры не портить.

Решение: У команды вычитания операнды имеют одинаковый размер. Если X является двойным словом, задача решается командой **sub R, X**. Если же тип X байт или слово, перед вычитанием нужно расширить X до 32 разрядов как число без знака. В Ассемблере есть команда **movzx**, позволяющая это сделать; первый операнд команды является регистром. Разумеется, перед использованием надо сохранить регистр в стеке, а после использования – восстановить. Вопрос – какой регистр взять в качестве первого операнда? Если возьмём, например, регистр EAX для записи X, получится ошибка для макровывоза RSub EAX, X: при записи расширения X сотрётся значение уменьшаемого. А при восстановлении регистра из стека будет стёрто вычисленное значение. Подобная ошибка возникнет, какой бы регистр мы ни взяли в качестве операнда **movzx**. Как справиться с этой трудностью? Запрограммируем два варианта вычислений. Сначала предположим, что параметр макровывоза не может быть регистром EAX и используем EAX как вспомогательный. Затем предположим, что параметр макровывоза является именно регистром EAX, в этом случае в качестве вспомогательного возьмём регистр EBX. Объединим эти варианты в директиве условной генерации; тогда макрогенератор, раскрывая конкретный макровывоз, зная, какой регистр передан в качестве параметра, выберет для включения в макрорасширение нужную ветвь.

```

RSub macro R, X
    if type X EQ 4
        sub R, X
    else
        ifdif <R>, <EAX>; R<>EAX
            push EAX
            movzx EAX, X
            sub R, EAX
            pop EAX
        else; Здесь R=EAX
            push EBX
            movzx EBX, X
            sub R, EBX
            pop EBX
        endif
    endif
endm

```

2.9. Модули

Решая задачи по теме «модули» важно не упускать из виду несколько моментов. Как на Ассемблере описывают модули? Модуль на Ассемблере выглядит так же, как обычная программа. По сути, программа – это частный случай модуля. То есть модуль на Ассемблере может состоять из секции данных, секции кода, заканчивается директивой **end**, может включать другие необходимые предложения. Го-

ловной модуль отличается от остальных модулей программы тем, что в директиве **end** стоит метка – точка входа в программу.

Модули транслируются отдельно, независимо друг от друга. Каждый модуль имеет своё пространство имён; это значит, что имена в разных модулях могут совпадать. Взаимодействие модулей, связь между ними реализуется с помощью внешних и общих имён. Формулировки условий задач и решения даны для Ассемблера MASM 6.14.

Задачи

№ 1. Как в Ассемблере объявить имя внешним; зачем нужны такие имена?

Ответ:

Необходимо сказать следующее.

- 1) Внешние имена объявляются в директиве **extrn**;
- 2) Внешние имена позволяют осуществлять доступ к именам, описанным в других модулях программы.

№ 2. Как в Ассемблере объявить имя общедоступным (общим); зачем нужны такие имена?

Ответ:

Необходимо сказать следующее.

- 1) Общие имена переменных, меток, имён процедур и констант объявляются в директиве **public**.¹
- 2) Общие имена доступны другим модулям программы. При трансляции модуля ассемблер сохраняет информацию о значениях общих имён в объектном модуле; компоновщик использует эту информацию при объединении объектных модулей в единую программу.

№ 3. Описать модуль M1, в секции данных модуля находится переменная A типа слово; описать модуль M2, в секции данных которого находится переменная B типа слово. Выписать *неголовной* модуль M3, в котором реализуется присваивание $A := A - B$.

Решение:

<pre>; модуль M1 public A .data A dw ? end</pre>	<pre>; модуль M2 public B .data B dw ? end</pre>	<pre>; модуль M3 include console.inc extrn A:word, B:word .code public P P proc mov BX, B sub A, BX ret P endp end</pre>
--	--	--

Обратите внимание на директиву **public** P, без неё выполнение кода в не головном модуле невозможно. Поскольку модуль M3 не головной, в директиве **end** нельзя указывать метку (директива **end** P будет ошибкой). Процедура P не выполняет стандартные соглашения о связях (в частности, не сохраняет регистры).

№ 4. Программа на Ассемблере состоит из двух модулей. В головном модуле описана общедоступная (**public**) переменная X размером в слово. Выписать второй модуль, содержащий процедуру с именем PechX. Процедура печатает значение X как число со знаком и как число без знака.

Решение. Для того, чтобы процедуру можно было вызвать из других модулей, имя PechX нужно объявить общедоступным. Далее, чтобы модуль мог использовать имя X, имя переменной X следует объявить внешним. Поскольку процедура использует команды печати, нужно включить в модуль файл console.inc. Получаем решение

```
include console.inc
extrn X:word
public PechX
.code
```

¹ Можно объявить общие имена и в директиве **extrn**.

```

PechX proc
    outi X
    outu X
    ret
PechX endp
end

```

2.10. Трансляция, компоновка, загрузка

Задачи этого раздела носят теоретический характер. Прежде, чем приступить к разбору решений задач, желательно прочесть соответствующую главу лекционного курса.

Задачи

№ 1. В программе на Ассемблере есть предложения **extrn X:abs** и **mov AX, X**. Какая служебная программа подставляет конкретное значение на место операнда X в команде **mov AX, X**?

Решение: Текст программы на Ассемблере первоначально обрабатывает транслятор, он переводит предложения программы на машинный язык; вместо каждой ассемблерной команды транслятор строит машинную команду – выбирает нужный КОП, заполняет операнды. Поскольку имя X – внешнее, транслятор не знает его значение, поэтому оставляет соответствующее поле машинной команды незаполненным. После транслятора программу обрабатывает компоновщик (редактор внешних связей), который объединяет несколько модулей в единую программу, заполняя значения внешних имён.

Ответ: Компоновщик (редактор внешних связей).

№ 2. Практически все программы работают со стеком, хотя бы для того, чтобы вызвать процедуры, реализующие ввод данных и печать результатов. Однако ни одна программа не загружает начальное значение в регистр ESP – указатель на вершину стека. Как же ESP получает значение?

Ответ: Загрузчик после размещения программы в ОП, и перед запуском её на счёт, отводит место в ОП для стека и записывает нужное значение в ESP.¹

№ 3. Одна из функций компоновщика (редактора внешних связей) – редактирование (разрешение) межмодульных связей. Объяснить:

- а) в чем смысл этой функции;
- б) почему ее не может выполнить транслятор Ассемблера.

Ответ:

а) Редактирование межмодульных связей – это замена внешних имён (имён из других модулей) их значениями (адресами для имён переменных, меток и числовыми значениями для имён констант).

б) Поскольку Ассемблер транслирует каждый модуль программы по отдельности, то он не знает значения имён из других модулей (внешних имён) и потому не может сделать такую замену.

2.11. Особенности современных архитектур: конвейер

Для увеличения быстродействия ЦП используют параллелизм, встраивая в процессор конвейер: процесс выполнения одной машинной команды разделяют на этапы (микрооперации); каждая микрооперация выполняется на отдельном устройстве; команда завершается, когда выполнены все её микрооперации. Устройства (порты), реализующие отдельные микрооперации, работают параллельно.

Таким образом, в ЦП в процессе выполнения одновременно находится несколько команд, каждая на своём этапе обработки. На конвейер команды программы загружаются последовательно, в том порядке, в каком они записаны в программе.²

¹ Допускается и более точный ответ: загрузчик создаёт и заполняет поле сохранения программы TSS (Task State Segment), где на месте хранения регистра ESP записывается нужное значение. При запуске программы на счёт диспетчером задач значения всех регистров загружаются из TSS.

² Современные процессоры разлагают команды на микрооперации, которые могут выполняться не в том порядке, как команды записаны в программе. Однако в данном разделе мы предполагаем, что этого не происходит.

Основная проблема в использовании конвейера – простои, вызванные тем, что команды программы зависимы друг от друга: результат выполнения одной команды может быть операндом другой, или вслед за выполнением одной команды нужно выполнить не команду, следующую за ней по тексту программы, а сделать переход. При этом устройства внутри ЦП будут простаивать. Существуют разные способы уменьшить простои конвейера.

Задачи

№ 1. Пусть в центральном процессоре используется конвейерный способ выполнения команд, при котором параллельно выполняются следующие 6 этапов: 1) чтение команды, 2) дешифровка кода операции, 3) вычисление исполнительных адресов, 4) чтение операндов, 5) выполнение операции, 6) запись результата. Считая, что на каждый из этих этапов тратится 1 единица времени, и что вначале конвейер был свободен, определить, сколько времени будет затрачено на выполнение следующего фрагмента программы:

```
mov bx, Y
add si, [bx]
xor ax, [bx]
```

Решение: Проще всего решить задачу, изобразив занятость устройств конвейера. На первом шаге на этап чтения команды попадает первая команда; на втором шаге первая команда перемещается на этап дешифровки, а вторая команда попадает на этап чтения команды и т.д. Нужно учесть, что вторая команда не может завершить этап вычисления исполнительного адреса, пока первая команда не выполнит запись результата. Схематически процесс выполнения команд изображён ниже в виде таблицы; первая колонка – единицы времени.

№	Чтение команды	Дешифровка	Исполнит. адреса	Чтение операндов	Выполнение операции	Запись результата
1	mov bx, Y					
2	add si, [bx]	mov bx, Y				
3	xor ax, [bx]	add si, [bx]	mov bx, Y			
4		xor ax, [bx]	add si, [bx]	mov bx, Y		
5		xor ax, [bx]	add si, [bx]		mov bx, Y	
6		xor ax, [bx]	add si, [bx]			mov bx, Y
7		xor ax, [bx]	add si, [bx]			
8			xor ax, [bx]	add si, [bx]		
9				xor ax, [bx]	add si, [bx]	
10					xor ax, [bx]	add si, [bx]
11						xor ax, [bx]

Ответ: 11 единиц времени.

№ 2. Пусть конвейер состоит из 5 этапов: 1) чтение команды, 2) дешифровка, 3) чтение операндов, 4) выполнение операции, 5) запись результата, каждый этап выполняется за две единицы времени. Сколько единиц времени займёт выполнение последовательности

```
sub BX, X
add BX, 4
mul CX
sub AX, 8
```

Переставить команды так, чтобы минимизировать простои конвейера. Сколько единиц времени будет выполняться преобразованная последовательность команд?

Решение: Сначала разберёмся, в какие моменты происходят простои конвейера

шаг	Чтение команды	Дешифровка	Чтение операндов	Выполнение операции	Запись результатов
1	sub BX, X				
2	add BX, 4	sub BX, X			
3	mul CX	add BX, 4	sub BX, X		
4	sub AX, 8	mul CX	add BX, 4	sub BX, X	
5	sub AX, 8	mul CX	add BX, 4		sub BX, X
6	sub AX, 8	mul CX	add BX, 4		
7		sub AX, 8	mul CX	add BX, 4	
8			sub AX, 8	mul CX	add BX, 4
9			sub AX, 8		mul CX
10			sub AX, 8		
11				sub AX, 8	
12					sub AX, 8

Простои возникают из-за того, что вторая команда ждёт чтения операндов, пока первая команда не завершится; также четвёртая команда ждёт на этапе чтения операндов, когда завершится третья команда (т.к. результат умножения записывается в регистр AX, который является первым операндом четвёртой команды). С другой стороны, первые две команды не зависят от третьей и четвёртой. Поэтому уменьшить простои можно, переставив команды так: 1-3-2-4. Преобразованная последовательность команд

```

sub BX, X
mul CX
add BX, 4
sub AX, 8

```

Состояние конвейера для преобразованной последовательности:

шаг	Чтение команды	Дешифровка	Чтение операндов	Выполнение операции	Запись результатов
1	sub BX, X				
2	mul CX	sub BX, X			
3	add BX, 4	mul CX	sub BX, X		
4	sub AX, 8	add BX, 4	mul CX	sub BX, X	
5		sub AX, 8	add BX, 4	mul CX	sub BX, X
6		sub AX, 8	add BX, 4		mul CX
7			sub AX, 8	add BX, 4	
8				sub AX, 8	add BX, 4
9					sub AX, 8

Ответ: 24 единицы времени выполняется исходная последовательность, 18 – переупорядоченная.

3. Разбор варианта письменного экзамена

Ниже приведён вариант письменного экзамена, содержащий 8 задач, время выполнения 2 часа.

3.1. Вариант письменного экзамена

Фамилия И.О. _____ Группа _____

1. Что такое такт работы процессора в машине Фон-Неймана? Перечислить основные этапы такта работы.

Ответ:

Такт –

Этапы –

2. Что будет напечатано в результате выполнения последовательности команд

```
mov  AX, -254
shl  AH, 1
imul AH
shl  AL, 1
outi AX
```

Ответ:

3. Написать полную программу на Ассемблере, которая вводит (по макрокомандам **inint**) два числа со знаком формата **dd** и печатает частное от деления меньшего числа на большее. При равенстве чисел вывести "Числа равны", вместо деления на ноль вывести "Деление на ноль". (Привести ответ на обратной стороне листа.)

4. Вычеркнуть синтаксически неверные команды

```
mul AL  jmp EAX  cmp CF, 1  mov EBX, 'A'  add 5000[ESI], 0  mov EAX[EBX], 5
```

5. Выписать справа фрагмент на Ассемблере (не более 4-х команд), который реализует операцию над флагом переноса CF

```
CF := notCF
```

Ответ:

Можно использовать регистр AL.

6. Пусть есть описания на языке Free Pascal

```
type MAS=array[1..k] of char;
function LastDigit(var z:MAS; n:longword):char;
```

Описать на Ассемблере функцию LastDigit, которая возвращает последнюю цифру (символ из диапазона '0'..'9') этого массива или значение 0FFh, если в массиве нет символов-цифр. Функция должна удовлетворять стандартным соглашениям о связях. Привести пример вызова этой функции, сделав на Ассемблере необходимые описания. (Дать ответ на обратной стороне листа.)

7. Написать макроопределение с заголовком

```
M11 macro v
```

где v – имя переменной. Если параметр v имеет формат m8 или m16, то надо реализовать присваивание

```
v:= 11-v
```

иначе должно получаться пустое макрорасширение. Макрорасширение должно содержать не более двух команд. Диагностику о возможных ошибках в переданных параметрах не выводить.

Ответ:

8. Указать значения регистра CL (в виде знакового и беззнакового десятичных чисел) и флагов CF, OF, SF, ZF после выполнения команд

```
mov CL, -70
add CL, 183
```

Ответ:

```
CL = (зн.),
CL= (беззн.)
CF =   OF =
SF =   ZF =
```

3.2. Решение варианта письменного экзамена

1. Такт работы процессора – выполнение одной машинной команды. Такт состоит из следующих этапов

1) Выборка команды из оперативной памяти по адресу, находящемуся в регистре счётчик адреса и запись команды в регистр команд, схематично $RK := \text{Пам}[RA]$ или $RK := \langle RA \rangle$.

2) Увеличение значения регистра счётчик адреса на единицу $RA := RA + 1$.

3) Выполнение команды, находящейся в регистре команд.

2. Выпишем, какие значения получают регистры в результате выполнения команд

mov AX, -254	$AX := -254 = \langle AH:AL \rangle := \langle -1, 2 \rangle$
shl AH, 1	$AH := AH * 2 = -2$
imul AH	$AX := AL * AH = 2 * (-2) = -4 = \langle AH:AL \rangle := \langle -1, -4 \rangle$
shl AL, 1	$AL := AL * 2 = -8; AX := \langle AH:AL \rangle := \langle -1, -8 \rangle = -8$

Если печатать AX как число со знаком, получим -8.

Ответ: -8

3. Возможное решение:

```
include console.inc
.data
Estr db 'Числа равны', 0
Zstr db 'Дел. на ноль', 0
.code
Start:
    inint eax
    inint ebx
    cmp eax, ebx
    je Eql
    jl Cnt
    xchg eax, ebx
; eax < ebx
Cnt: cmp ebx, 0
    je Zero
    cdq
    idiv ebx
    outi eax
    jmp Fin
Eql: mov ebx, offset Estr
    jmp Prn
Zero: mov ebx, offset Zstr
Prn: outstr ebx
Fin: exit
end Start
```

Комментарий: В этой задаче необходимо обратить внимание на размер делимого. Поскольку делитель – двойное слово, делимое должно быть четверное слово. Следовательно, перед делением нужно расширить `eax` до пары `<EDX:EAX>` как число со знаком (в условии задачи сказано, что работаем со знаковыми числами).

4. Неверные команды подчёркнуты.

`mul AL jmp EAX cmp CF, 1 mov EBX, 'A' add 5000[ESI], 0 mov EAX[EBX], 5`

Комментарий:

1) Верно, $AX := AL * AH$.

2) Верно, $EIP := EAX$; это косвенный переход по адресу из EAX.

3) Ошибка: флаг не может быть операндом команды.

4) Верно, $EBX := \text{ord}('A')$, формат операндов `r32, i32`.

5) Ошибка: неизвестный размер операнда, байт, слово или двойное слово?

6) Ошибка, неверный формат операндов: регистр EAX – не переменная, его нельзя модифицировать.

5. Для анализа значения флага используем команду условного перехода `jc`. Приведём одно из возможных решений

```
mov AL, 0h
jc L
mov AL, 0FFh
L: add AL, 1
```

При оценке решений задач, в которых есть ограничение на длину решения, лишние команды штрафуются снижением баллов за задачу. Тем не менее, на экзамене лучше написать решение, пусть длиннее требуемого, чем не написать решение совсем.

6. В процедуре нужно реализовать следующий алгоритм: просматривать массив с конца, от последнего элемента к первому; как только найдена цифра, прекратить просмотр и вернуть найденное значение. Символьному типу на Ассемблере соответствует тип `byte`, значит, массив будет байтовый; адрес последнего элемента массива `z` вычисляется по формуле *адрес `z` + количество элементов `n` – 1*. Адрес начала массива запишем в регистр `EBX`, количество элементов – в регистр `ECX`, он будет счётчиком цикла и модификатором для просмотра массива одновременно. Возможное решение:

<pre> LastDigit proc push ebp mov ebp, esp push ebx push ecx mov ecx, [ebp+12]; n mov ebx, [ebp+8]; Адрес z L: mov al, [ebx+ecx-1] cmp al, '0' jbe L1 </pre>	<pre> cmp al, '9' jbe L2 L1: loop L ; нет цифр в массиве mov al, 0FFh; Ответ L2: pop ecx pop ebx pop ebp ret 2*4 LastDigit endp </pre>
---	---

Пример вызова:

```

n equ 40000
z db N dup(?)

```

```

push n
push offset z
call LastDigit

```

Заметим, что в задаче используются два способа передачи параметров: первый параметр `z` передаётся по ссылке, второй параметр `n` – по значению. При вызове процедуры в стек помещается значение `n` и адрес переменной `z`. В процедуре значение `n` из стека записывается в `ECX`, а адрес `z` – в регистре `EBX` и затем `EBX` используется для доступа к массиву, расположенному в секции данных. Обратим внимание также на то, что результат возвращается в регистре `AL`, поэтому сохранять в стеке `EAX` не нужно.

7. Возможное решение:

```

M11 macro v
    if type v EQ 1 or type v EQ 2
        neg v
        add v, 11
    endif
endm

```

8. Сначала определим `SF` и результат в виде числа без знака. Доп(-70) = $256 - 70 = 186$. Выполняя операцию $186 + 183$, получим результат больше, чем 256, следовательно, `SF`=1. Значение `CL` как число без знака равно $(186 + 183) \bmod 256 = 113$. Далее, если рассматривать это значение `CL` как число со знаком, тоже получится 113 (т.к. $113 \leq 127$). Флаг `ZF`=0, поскольку результат операции – не ноль. Наконец, получим флаги `OF` и `SF`. Для этого представим операнды в виде чисел со знаком. Числу 183 соответствует отрицательное число со знаком -73 : $183 = \text{Доп}(-73)$. Выполним сложение $(-70) + (-73) = -143$, результат меньше, чем -128 , значит, `OF`=1. Математический результат отрицательный, но было переполнение, следовательно, `SF` неверно показывает знак результата, поэтому `SF`=0.

Ответ:

```

CL = 113 (зн.),
CL = 113 (беззн.)
CF=1   OF=1
SF=0   ZF=0

```

В заключение подчеркнём, что данное пособие не является учебником по курсу «Архитектура ЭВМ и язык ассемблера», играет вспомогательную роль. Для полноценного освоения курса нужно разобрать материал лекций, прочитать учебники и потренироваться в решении задач, предлагаемых на семинарских занятиях.

Отметим, что письменный экзамен отличается строгостью: оценивается то, что написано студентом, а не то, что он имел в виду. Поэтому нужно внимательно относиться к проверке своих решений, отвести на это время в конце экзамена.

Желаем всем читателям пособия успешной сдачи экзамена.

Содержание

1. Введение.....	3
2. Разбор задач	4
2.1. Представление чисел в ЭВМ	4
2.2. Различные архитектуры ЭВМ	5
2.3. Синтаксис Ассемблера	7
2.4. Представление чисел в процессоре Intel, арифметические флаги	10
2.5. Арифметические команды	12
2.6. Массивы, записи, полная программа, стек	13
2.7. Процедуры и функции	15
2.8. Макросредства языка MASM	18
2.9. Модули	21
2.10. Трансляция, компоновка, загрузка	23
2.11. Особенности современных архитектур: конвейер	23
3. Разбор варианта письменного экзамена	25
3.1. Вариант письменного экзамена	26
3.2. Решение варианта письменного экзамена	27