Практическая работа №2
Тема - Ознакомление с регистрами процессора x86-32 и типами данных доступных для объявления в MASM.

Цель – Получить базовые навыки по объявлению переменных и операциям загрузки значений в регистры процессора.

Сведения

Регистры общего назначения

Регистр — это небольшой (обычно 4 или 8 байт) кусочек памяти в процессоре с чрезвычайно большой скоростью доступа. Регистры делятся на регистры специального назначения и регистры общего назначения. Нас сейчас интересуют регистры общего назначения. Как можно догадаться по названию, программа может использовать эти регистры под свои нужды, как ей вздумается.

На x86 доступно восемь 32-х битных регистров общего назначения — eax, ebx, ecx, edx, esp, ebp, esi и edi. Регистры не имеют заданного наперед типа, то есть, они могут трактоваться как знаковые или беззнаковые целые числа, указатели, булевы значения, ASCII-коды символов, и так далее. Несмотря на то, что в теории эти регистры можно использовать как угодно, на практике обычно каждый регистр используется определенным образом. Так, esp указывает на вершину стека, ecx играет роль счетчика, а в eax записывается результат выполнения операции или процедуры. Существуют 16-и битные регистры ax, bx, cx, dx, sp, bp, si и di, представляющие собой 16 младших бит соответствующих 32-х битных регистров. Также доступны и 8-и битовые регистры ah, al, bh, bl, ch, cl, dh и dl, которые представляют собой старшие и младшие байты регистров ax, bx, cx и dx соответственно.

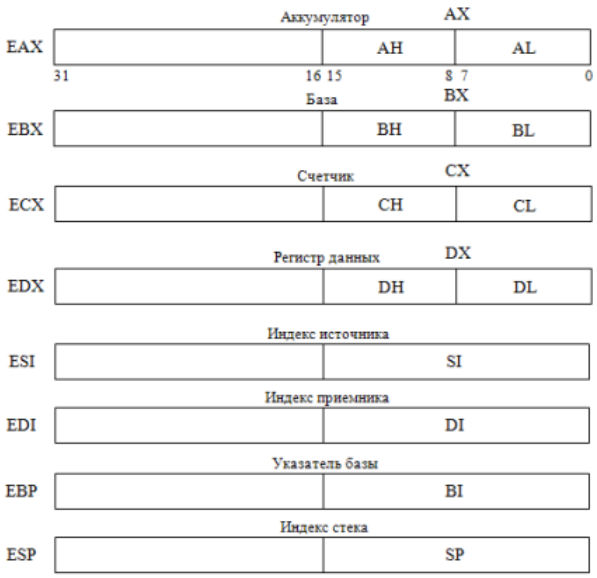

Рис. 1. Регистры общего назначения

Сегментные регистры.

При использовании сегментированных моделей памяти для формирования любого адреса нужны два числа – адрес начала сегмента и смещение искомого байта относительно этого начала. Так как сегменты способны оказаться где угодно, программа обращается к ним, применяя вместо настоящего адреса начала сегмента 16-битное число, называемое селектором. В процессорах Intel предусмотрены шесть 16-битных регистров – CS, DS, ES, FS, GS, SS, где хранятся селекторы. Это означает, что в любой момент можно

изменить параметры, записанные в этих регистрах. В отличие от DS, ES, GS, FS, которые называются регистрами сегментов данных, CS и SS отвечают за сегменты двух особенных типов – сегмента кода и сегмент стека. Первый содержит программу, исполняющуюся в данный момент, следовательно, запись нового селектора в этот регистр приводит к тому, что далее будет исполнена не следующая по тексту программы команда, а команда из кода, находящегося в другом сегменте, с тем же смещением. Смещение очередной выполняемой команды всегда хранится в специальном регистре EIP (указатель инструкции, 16-битная форма IP), запись в который также приведет к тому, что далее будет исполнена какая-нибудь другая команда.

Стек.

Стек – организованный специальным образом участок памяти, который используется для временного хранения переменных, передачи параметров вызываемым подпрограммам и сохранения адреса возврата при вызове процедур и прерываний. Легче всего представить стек в виде стопки листов бумаги (это одно из значений слова «stack» в английском языке) – вы можете класть и забирать листы только с вершины стопки. Поэтому, если записать в стек числа 1, 2, 3, то при чтении они окажутся в обратном порядке – 3, 2, 1. Стек располагается в сегменте памяти, описываемом регистром SS, и текущее смещение вершины стека отражено в регистре ESP, причем во время записи значение этого смещения уменьшается, то есть он «растет вниз» от максимально возможного адреса (см. рис. 2.).

Рис. 2. Стек

Регистр флагов.

Еще один важный регистр, использующийся при выполнении большинства команд, - регистр флагов. Его младшие 16 бит, представлявшие собой весь этот регистр до процессора 80386, называются FLAGS. В E FLAGS каждый бит является флагом, то есть устанавливается в 1 при определенных условиях или установка его в 1 изменяет поведение процессора. Все флаги, расположенные в старшем слове регистра, имеют отношение к управлению защищенным режимом, поэтому будем рассматривать только регистр FLAGS (см. рис. 3.)

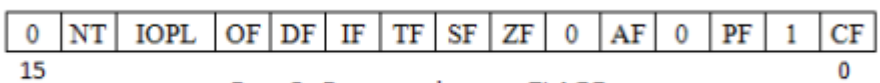

Рис. 3. Регистр флагов FLAGS

- CF – флаг переноса. Устанавливается в 1, если результат предыдущей операции не уместился в приемнике и произошел перенос старшего бита или если требуется заем (при вычитании), в противном случае – в 0. Например, после сложения слова 0FFFFh и 1, если регистр, в который надо поместить результат, - слово, в него будет записано 0000h и флаг CF=1.

- PF – флаг четности. Устанавливается в 1, если младший байт результата предыдущей команды содержит четное число битов, равных 1, и в 0, если нечетное. Это не тоже самое, что делимость на два. Число делится на 2 без остатка, если его самый младший бит равен нулю, и не делится, когда он равен 1.
- AF – флаг полупереноса или вспомогательного переноса. Устанавливается в 1, если в результате предыдущей операции произошел перенос (или заем) из третьего бита в четвертый. Этот флаг используется автоматически командами двоично-десятичной коррекции.
- ZF – флаг нуля. Устанавливается в 1, если результат предыдущей команды – ноль.
- SF – флаг знака. Он всегда равен старшему биту результата.
- TF – флаг ловушки. Он был предусмотрен для работы отладчиков, не использующих
- защищенный режим. Установка его в 1 приводит к тому, что после выполнения каждой
- программной команды управление временно передается отладчику (вызывается прерывание 1 – описание команды INT).
- IF – флаг прерываний. Сброс этого флага приводит к тому, что процессор перестает
- обрабатывать прерывания от внешних устройств (описание команды INT). Обычно его
- сбрасывают на короткое время для выполнения критических участков кода.
- DF – флаг направления. Он контролирует поведения команд обработки строк: когда он
- установлен в 1, строки обрабатываются в сторону уменьшения адресов, когда DF=0 – наоборот.
- OF – флаг переполнения. Он устанавливается в 1, если результат предыдущей арифметической операции над числами со знаком выходит за допустимые для них пределы. Например, если при сложении двух положительных чисел получается число со старшим битом, равным единице, то есть отрицательное, и наоборот.

Флаги IOPL (уровень привилегий ввода-вывода) и NT (вложенная задача) применяются в защищенном режиме.

1. Внутренние типы данных.

В MASM определены несколько внутренних типов данных, значения которых могут быть присвоены переменным, либо они могут являться результатом выполнения выражения. Например, в переменной типа DWORD можно сохранить любое 32-разрядное целое значение. Однако на некоторые типы накладываются более жесткие ограничения. Например, переменной типа REAL4 можно присвоить только вещественную константу.

Таблица 1. *Внутренние типы данных.*

Тип	***Описание***
byte	8-разрядное беззнаковое целое
sbyte	8-разрядное знаковое целое
word	16-разрядное беззнаковое целое
sword	16-разрядное знаковое целое
dword	32-разрядное беззнаковое целое
sdword	32-разрядное знаковое целое
fword	48-разрядное целое
qword	64-разрядное целое

tbyte	80-разрядное (10-байтовое) целое
real4	32-разрядное (4-байтовое) короткое вещественное.
real8	64-разрядное (8-байтовое) длинное вещественное.
real10	80-разрядное расширенное вещественное.

2. Оператор определения данных.

С помощью оператора определения данных в программе резервируется область памяти соответствующей длинны для размещения переменной. При необходимости этой переменной можно назначить имя. Синтаксис оператора следующий:

[имя] директива инициализатор [, инициализатор]…

Инициализаторы. При определении данных должен быть указан хотя бы один инициализатор, даже если переменной не назначается какого-то конкретного значения (в этом случае значение инициализатора равно ?). Все дополнительные инициализаторы перечисляются через запятую. Для целочисленных типов данных инициализатор является целочисленной константой либо выражением, значение которого соответствует размеру определяемых данных (BYTE, WORD, и т.д.).

Независимо от используемого формата чисел, все инициализаторы автоматически преобразовываются ассемблером в двоичную форму. Другими словами, в результате компиляции инициализаторов 00110010b, 32h и 50d будет получено одинаковое двоичное значение.

3. Определение переменных типа BYTE и SBYTE

Директивы BYTE (определяет беззнаковый байт) и SBYTE (определяет знаковый байт) используются в операторах определения данных, с помощью которых в программе выделяется память под одну или несколько знаковых или беззнаковых переменных длиной 8 битов. Каждый инициализатор должен быть либо 8-разрядным целочисленным выражением или символьной константой. Например:

```
value1    BYTE      'A'    Символьная константа
value2    BYTE       0     Наименьшее беззнаковое байтовое значение
value3    BYTE      255    Наибольшее беззнаковое байтовое значение
value4    SBYTE    -128    Наименьшее знаковое байтовое значение
value5    SBYTE    +127    Наибольшее знаковое байтовое значение
```

Чтобы оставить переменную неинициализированной (т.е. при выделении под нее па мяти не присваивать ей никакого значения), вместо инициализатора используется знак вопроса. Такая форма записи предполагает, что значение данной переменной будет на значено во время выполнения программы с помощью специальных команд процессора. Вот пример:

```
value6    BYTE    ?
```

Имена переменных. На самом деле имя переменной является меткой, значение кото рой соответствует смещению данной переменной относительно начала сегмента, в кото ром она расположена. Например, предположим, что переменная valuel расположена в сегменте данных со смещением 0 и занимает один байт памяти. Тогда переменная value2 будет располагаться в сегменте со смещением 1:

```
.data
valuel    BYTE    10h
value2    BYTE    20h
```

Директива DB. В предыдущих версиях MASM для определения байта данных исполь зовалась директива DB. В нынешней версии компилятора MASM вы можете по-прежнему использовать эту директиву, но тогда компилятор не сможет отличить, к како му типу (знаковому или беззнаковому) относится ваша переменная:

```
vail      DB      255    Беззнаковое байтовое значение
val2      DB     -128    Знаковое байтовое значение
```

4. Множественна инициализация.

Если в одном и том же операторе определения данных используется несколько ини циализаторов, то присвоенная этому оператору метка относится только к первому байту данных. В приведенном ниже примере подразумевается, что метке list соответствует смещение 0. Тогда значение 10 располагается со смещением 0 относительно сегмента данных, значение 2 0 — со смещением 1, 3 0 — со смещением 2 и 4 0 — со смещением 3:

```
.data
list      BYTE     10,20,30,40
```

Метки нужны далеко не для всех операторов определения данных. Например, если нам нужно определить непрерывный массив байтов, начинающийся с переменной list, то дополнительные операторы определения данных могут быть введены в последующих стоках программы:

```
list      BYTE     10,20,30,40
          BYTE     50,60,70,80
          BYTE     81,82,83,84
```

В одном операторе определения данных могут использоваться инициализаторы, за данные в разных системах счисления. Кроме того, могут использоваться вперемешку как символы, так и строковые константы. В приведенном ниже примере списки listl и list2 эквивалентны:

```
listl    BYTE    10,   32,   41h,  00100010b
list2    BYTE    OAh,  20h,  'A',  22h
```

5. Определение строк.

Чтобы определить в программе текстовую строку, нужно составляющую ее последова тельность символов заключить в кавычки. Чаще всего в программах используются так называемые нуль-завершенные (null-terminated) строки, или строки, оканчивающиеся нулевым байтом, т.е. байтом, значение которого равно двоичному нулю. Пример:

```
greetingl BYTE "Добрый день!",O
```

Каждый символ данной строки занимает один байт памяти. К строкам символов не от носится правило, согласно которому значения отдельных байтов инициализатора разделя ются между собой запятой.

Оператор определения строк может занимать несколько строчек в программе. При этом для каждой строчки программы совершенно не обязательно присваивать отдельную метку, как показано в следующем примере:

```
greetingl BYTE "Вас приветствует демо-версия программы шифрования, "
          BYTE "созданная Кипом Ирвином.",ODh,OAh
          BYTE "Если вы внесете изменения в эту программу, "
          BYTE "пожалуйста, пришлите мне ее копию.",ODh,OAh,O
```

Напомню, что шестнадцатеричные значения байтов ODh и OAh, указанные в этом примере, называются символами конца строки и сокращенно обозначаются CR/LF. При их посылке на стандартное устройство вывода, курсор монитора будет автоматически пе реходить в первую позицию следующей строки.

В MASM предусмотрена возможность разделения одного длинного оператора про граммы на нескольких строчек. Для этого в месте разрыва текущей строчки оператора ставится специальный знак продолжения — символ обратной косой черты (\). Другими словами, если оператор не помещается в одной строчке исходного кода, то в конце текущей строчки ставится символ \ и набор кода продолжается со следующей строчки програм мы. Например, приведенные ниже два оператора определения данных эквивалентны:

```
greeting1 BYTE " Вас приветствует демо-версия программы шифрования, "
и
   greeting1 \
 BYTE " Вас приветствует демо-версия программы шифрования, "
```

6. Использование оператора DUP.

Оператор DUP используется для создания переменных, содержащих повторяющиеся значения байтов. В качестве счетчика байтов используется константное выражение. Этим оператором обычно пользуются при выделении памяти под строку символов или массив, которые могут быть инициализированы или нет. Например:

```
BYTE  20  DUP(0)          ; 20 байтов, все равны нулю
BYTE  20  DUP(?)          ; 20 байтов, значение которых не определено
BYTE   4  DUP("СТЕК ")  / 20 bytes: "СТЕК СТЕК СТЕК СТЕК "
```

7. Определение переменных типа WORD и SWORD

С помощью директив WORD (определить слово) и SWORD (определить слово со знаком) в программах выделяется память для хранения 16-разрядных целых значений. Например:

```
word1   WORD   65535  ;  Наибольшее беззнаковое значение
word2   SWORD   -32768  ;  Наименьшее знаковое значение
word3   WORD   ?  ;  Не инициализированное беззнаковое значение
```

В прежних версиях ассемблера для определения как знаковых, так и беззнаковых 16-разрядных целых переменных использовалась директива DW. В новой версии MASM вы также можете пользоваться этой директивой:

```
vail   DW   65535  ;  Беззнаковое
val2   DW   -32768  ;  Знаковое
```

Массив слов. Для создания массива 16-разрядных слов можно воспользоваться либо оператором DUP, либо явно перечислить значения каждого элемента массива через запя тую. Вот пример массива слов, содержащего определенные значения:

```
myList      WORD   1,2,3,4.5
```

Для выделения памяти под массив слов удобно пользоваться оператором DUP:

```
array  WORD 5 DUP(?)   ; Массив из 5 не инициализированных слов
```

8. Определение переменных типа DWORD и SDWORD

С помощью директив DWORD (определить двойное слово) и SDWORD (определить двой ное слово со знаком) в программах выделяется память для хранения 32-разрядных целых значений. Например:

```
vail   DWORD    12345678b ;    Беззнаковое
val2  SDWORD   -2147483648 ;    Знаковое
val3   DWORD    20 DUP(?) ;    Не инициализированный массив беззнаковых чисел
```

В прежних версиях компилятора ассемблера для определения как знаковых, так и без знаковых 32-разрядных целых переменных использовалась директива DD. В новой версии MASM вы также можете пользоваться этой директивой:

```
val1    DD    12345678h  ;  Беззнаковое
val2    DD     -2147483648  ;  Знаковое
```

Массив двойных слов. Для создания массива 32-разрядных слов можно воспользоваться либо оператором DUP, либо явно перечислить значения каждого элемента массива че рез запятую. Вот пример массива слов, содержащего определенное значения:

```
myList    DWORD       1,2,3,4.5
```

9. Определение переменных типа QWORD

С помощью директивы QWORD (определить учетверенное слово) в программах выделя ется память для хранения 32-разрядных целых значений. Например:

```
quad1    QWORD       1234567812345678h
```

Кроме того, для определения учетверенного слова в программах можно использовать устаревшую директиву DQ:

```
quad1    DQ    1234567812345678h
```

10. Определение переменных типа TBYTE.

С помощью директивы TBYTE (определить 10 байтов) в программах выделяется па мять для хранения 80-разрядных целых значений. Этот тип данных в основном использу ется для хранения десятичных упакованных целых чисел (двоично-кодированных целых чисел). Для работы с этими числами используется специальный набор команд математи ческого сопроцессора. Вот пример определения:

```
val1    TBYTE    100000000123456789Ah
```

Кроме того, для определения десятибайтовой переменной в программах можно ис пользовать устаревшую директиву DT:

```
val1      DT       100000000123456789Ah
```

11. Определение переменных вещественного типа.

Директива REAL4 определяет в программе 4-байтовую переменную вещественного типа одинарной точности. Директива REAL8 определяет 8-байтовую переменную веще ственного типа двойной точности, а REAL10 — 10-байтовую переменную вещественного типа расширенной точности. После каждой из директив необходимо указать один или несколько инициализаторов, значение которых должно соответствовать длине выделяемого участка памяти под переменную:

```
rVal1      REAL4    -2.1
rVal2      REAL8    3.2E-260
rVal3      REAL10    4.6E+4096
ShortArray REAL4    20 DUP(O.O)
```

В табл. 2 перечислены характеристики, такие как количество значащих цифр и диа пазоны возможных значений для каждого из трех основных вещественных типов данных.

Таблица 2. *Характеристики основных вещественных типов данных*

Тип	Количество значащих десятичных цифр	Приблизительный диапазон значений
Короткое вещественное	6	1,18x10-38...3,40x1038
Длинное вещественное	15	2,23x10-308...1,79x10308
Расширенное вещественное	19	3,37x10-4932...1,18x104932

В предыдущих версиях компилятора ассемблера для определения вещественных чисел использовались директивы DD, DQ и DT. Их можно использовать и в современной версии ассемблера:

```
rVal1   DD   -1.2
rVal2   DQ   3.2E-260
rVal3   DT   4.6E+4096
```

12. Символические константы.

Идентификатор языка ассемблера (или символ), которому поставлено в соответствие целочисленное выражение или текстовая строка, называется символической константой (symbolic constant) или определением символа (symbol definition). В отличие от переменных, для которых во время объявления ассемблер резервирует память в программе, при опре делении символической константы память не выделяется. Символические константы используются только во время компиляции программы, их нельзя изменить во время вы полнения программы.

Директива присваивания (=) связывает символическое имя с целочисленным выражением. Ее синтаксис следующий:

```
имя =  выражение
```

Как правило, значением выражения является 32-разрядное целое число. Все имена заменяются соответствующими им выражениями на этапе ассемблирования программы, точнее, во время ее первой фазы — обработки исходного текста программы препроцессо ром. Например, если препроцессор встречает в программе следующие строки

```
COUNT = 500 mov ax,COUNT
```

он заменит их на следующую команду:

```
mov  ax,500
```

13. Директива EQU.

Эта директива используется для назначения символического имени целочисленному выражению или произвольной текстовой строке. Существует три формата директивы EQU:

```
имя   EQU   выражение
имя   EQU   символ
имя   EQU   <текст>
```

В первом случае значение выражения должно иметь целый тип и находиться в допусти мых пределах. Во втором случае символ должен быть определен ранее с помощью директивы присваивания (=) или другой директивы EQU. В третьем случае

между угловыми скобками <. . . > может находиться произвольный текст. Если после определения символа с указанным именем оно встретится компилятору в программе, то вместо этого символа будет подставлено соответствующее ему целочисленное значение или текст.

Директивы EQU используются в случае определения символов, которым не обязатель но должно соответствовать целочисленное значение. Например, с помощью этой дирек тивы можно определить вещественную константу:

```
PI    EQU   <3.1415926>
```

Пример. Символ можно легко связать с текстовой строкой, а затем на основе этого символа создать переменную в программе:

```
pressKey    EQU <"Для продолжения  нажмите любую клавишу. ..",0>
.data
prompt    BYTE pressKey
```

Пример. Предположим, что нам нужно определить символическую константу, значе ние которой равно количеству ячеек а матрице размерности 10x10. Мы можем опреде лить в программе две символические константы двумя разными способами. Значение первой из них будет являться целым числом, а второй — текстовым выражением. Затем оба этих символа можно использовать а операторах определения данных:

```
matrix1    EQU    10   *   10
matrix2    EQU    <10   *   10>
.data
M1    WORD    matrix1
M2    WORD    matrix2
```

При этом ассемблер создаст два разных оператора определения данных для перемен ных M1 и M2. Вначале он вычислит значение символической константы matrix1, а затем присвоит ее значение переменной M1. Во втором случае он просто скопирует текст, соответ ствующий символу matrix2, в оператор определения данных для второй переменной M2:

```
M1    WORD   100
M2    WORD   10   *   10
```

Невозможность переопределения. Директива EQU отличается от директивы присваива ния (=) тем, что определенный с ее помощью символ нельзя переопределить в одном и том же исходном файле. На первый взгляд это может показаться недостатком. Однако данный недостаток может обернуться преимуществом, поскольку вы не сможете случай но изменить значение однажды определенного символа.

14. Директива TEXTEQU.

Эта директива впервые появилась в шестой версии MASM. По сути, она очень похожа на директиву EQU и создает так называемый текстовый макрос (text macro). Существует три формата директивы textequ:

```
имя    TEXTEQU    <текст>
имя    TEXTEQU    текстовый__макрос
```

```
имя   TEXTEQU   %константное_выражение
```

В первом случае символу присваивается указанная в угловых скобках <. . . > тексто вая строка. Во втором случае — значение заранее определенного текстового макроса. В третьем случае — символической константе присваивается значение целочисленного выражения.

В приведенном ниже примере переменной prompt1 присваивается значение текстового макроса continueMsg:

```
continueMsg      TEXTEQU  <"Хотите  продолжить  (Y/N)?">
. data
prompt1      BYTE continueMsg
```

При определении текстовых макросов можно использовать значения других текстовых макросов. В приведенном ниже примере символу count присваивается значение целочисленного выражения, в котором используется символ rowSize. Затем определяется символ move, значение которого равно mov. После этого определяется символ setupAL на основе символов move и count:

```
rowSize  =  5
count      TEXTEQU  % (rowSize  *  2)   ;  Тоже  самое,  что  и  count  TEXTEQU
<10>
move   TEXTEQU   <mov>
setupAL   TEXTEQU   <move   al,count>
;  Тоже  самое,  что  и  setupAL   TEXTEQU   <mov   al,10>
```

Символ, определенный с помощью директивы TEXTEQU, можно переопределить в программе в любой момент. Этим она отличается от директивы EQU.

Замечание по поводу совместимости. Директива TEXTEQU появилась только в шестой версии MASM. Поэтому, если вы хотите, чтобы ваша программа была совместима с предыдущими версиями компилятора MASM, а также с другими типами ассемблеров, используйте вместо директивы TEXTEQU директиву EQU.

Чтобы разделить код и разного рода данные, в exe-файлах существуют секции.

У каждой секции свои атрибуты и они тоже напрямую связаны с защищённым режимом. Это позволяет стабилизировать работу программ и упорядочить процесс написания. Считается, что машинный код менять не нужно, значит, для него подойдёт атрибут read only (только чтение). А данные (как теперь считается) выполнять не нужно, значит, можно в секции данных выключить атрибут исполняемого кода. Слава Богу, что всё это только теоретически (пока). Любая секция может содержать и код, и данные, и соответствующие атрибуты. Так что сейчас можно воспринимать секции только как полезную для программистов вещь.

Итак, секция данных объявляется директивой ".data", и это выставляет в ней нужные для данных атрибуты.

Так же, как мы вписывали строки текста в конце файла (после кода программы), эта секция по умолчанию будет размещаться в exe-файле после кода.

.DATA –секция с данными
Здесь объявляем переменные
.CODE
Здесь пишем код

Задание:

Используя код программы первой работы

1. Объявить переменные следующих типов:

Байт, беззнаковое слово, целое (4 байта) со знаком. Строка из 10 символов, число с плавающей точкой одинарной точности (аналог float из c++). Строка символов заканчивающаяся символом с кодом #0. Для каждого типа необходимо объявить инициализированную и неинициализированную переменную. Значения для инициализации переменных выбрать по своему усмотрению.

2. Для числовых инициализированных переменных в секции кода загрузить их значения в один из регистров общего назначения, и затем выгрузить значение регистра в неинициализированную переменную.

3. Для строковых значений проделать те же действия используя 32 битные и 16 битные регистры общего назначения.