Лекция 6. Символический язык транслятора (язык ассемблера)

Символический язык для записи исходного текста программы на уровне команд процессора называют языком ассемблера (языком транслятора).

Какие возможности предоставляет транслятор?

- Записывать команды в символическом виде
- Использовать символические обозначения для внутрисегментных адресов данных и команд.
- Выделять место в памяти для данных в нужном формате
- Записывать числовые величины в удобном виде: в 10-ом, 16-ом или 2-ом виде.
- Записывать символьные коды в виде символов, а не обязательно в ASCII-кодах.

Исходный текст программы для транслятора

Исходный текст ассемблерной программы представляет собой описание будущего размещения команд и данных вашей программы в сегментах памяти. Все, что записано внутри одного «бумажного» сегмента, будет так же размещено в реальном сегменте памяти.

Исходный текст состоит из строк 3-х категорий:

- команда процессора
- директива для транслятора
- или строка комментария (тогда она начинается с ;)

Любая строчка исходного текста, включая последнюю, должна заканчиваться нажатием Enter

Директивы транслятора

Это строки специальных указаний (пояснений) для транслятора. О чем они?

- систему команд какого процессора используем
- где начало и конец каждого программного сегмента
- где в сегменте размещать данные и в каком формате
- когда закончить трансляцию и так далее

!! Директивы трансляции (кроме размещения кодов данных) не порождают после трансляции никаких кодов!!

Директивы описания программных сегментов

- .286 или (.386, .486,…)
- имя segment use16 (use32)
- имя ends
- assume sreg : Имя, ...
- end адрес 1-команды

- система команд какого процессора используется
- начало программного сегмента, его символическое имя, разрядность сегмента – use
- конец сегмента
- какие сегментные регистры выбраны в качестве указателей программных сегментов
- закончить трансляцию

- *Директивы SEGMENT и ENDS* отмечают начало и конец описания сегмента в исходном тексте. Символические имена сегментов выбираются произвольные.

- *Директива ASSUME* дает транслятору информацию, какие сегментные регистры будут являться указателями ваших программных сегментов.

- *Директива END* показывает транслятору конец текста исходной программы. В *END* задается символический адрес команды, с которой процессор должен начать чтение и исполнение команд из вашего кодового сегмента.

Структура исходного текста 16-разрядной программы, состоящей из сегмента кода и сегмента данных (синим – директивы для транслятора, красным – команды процессора)

```
data  segment  use16     ; начало описания сегмента данных
 . . .  размещение данных или резервирование места для них . .
data  ends               ; конец  сегмента

cod  segment  use16      ; начало  описания кодового сегмента
assume cs:cod, ds:data
; команды загрузки регистров - указателей  сегментов данных
met:      mov ax, data
          mov ds, ax
;   команды, реализующие ваш алгоритм действий для процессора
            . . .
;  команды для завершения исполнения (вызов в ОС)
          mov ah, 4ch
          int 21h
cod   ends               ;  конец  сегмента
          end   met      ;  конец исходного текста программы
```

Замечания

- Последовательность описания сегментов в исходном тексте не имеет значения;
- Загружать регистр-указатель кодового сегмента CS в программе не надо. Это делает операционная система при загрузке уже исполняемой программы в память;
- Команды для «завершения исполнения» реализуют вызов системного сервиса операционной системы для выгрузки текущего кода из памяти и переключения процессора на исполнение кода другой программы.

Структура исходного текста односегментной программы

```
mycod  segment   use16    ; начало описания кодового сегмента
assume cs:mycod
;   команды, реализующие ваш алгоритм действий для процессора
     met:        . . .
                 . . .
;   команды для завершения исполнения
            mov ah, 4ch
            int 21h

; директивы размещения данных или резервирования места
            . . . .
mycod   ends                  ;  конец  сегмента
            end   met
```

Директивы транслятору для размещения данных

Указываем транслятору, какие значение и в каком формате размещать в сегменте

- db (defined byte) - 1-байтный формат данных
- dw (defined word) - 2-х байтный формат
- dd (defined double word) - 4-х байтный формат

Примеры:	Результат трансляции в код (в hex):
db 4	; 04
dw 1, -4	; 0001 и fffc
dd 1	; 00000001
db 1,2,3,5	; 01 02 03 05

Резервирование места для данных

Вместо конкретных значений данных в директиве записывается символ ?.

Транслятор выделит место в сегменте нужного формата без размещения конкретных кодов

Указание на повторение выделения места транслятору - dup

Примеры использования директив:

```
db  ?              ;  транслятор выделит 1 байт в сегменте
db  ?, ?           ; 2 последовательных байта
dw ?               ; 1 слово (2 байта)
dd ? , ?           ; два двойных слова (по 4-байта)
db 25 dup (?)      ; 25 байт
db 5 dup (3), 10 dup (?) ; транслятор разместит в сегменте 5 байт с кодом 3 и 10 нулевых байт.
```

Синтаксис символических адресов

Требования транслятора к символическим адресам данных и команд:

- латиница
- начинаются с буквы, могут содержать цифры
- по умолчанию, для транслятора нет разницы между строчными и прописными буквами

Примеры символических имен (адресов):

- правильные: a1, vasya, data25, Beg,…
- неправильные: 1f, use16, bx, end, *bb_1,..

Символические адреса данных

- Чтобы не высчитывать конкретное значение внутрисегментного адреса данных, начало их размещения в сегменте можно «пометить» символическим адресом и использовать его в командах.
- При трансляции символические внутрисегментные адреса транслятор заменяет на числовые внутрисегментные адреса

Пример:

```
a1   db   34          ; a1 – символический внутрисегментный адрес
                          байта
. . . . . .
  add  bl, ds: a1     ; используем символический
                        внутрисегментный адрес - a1 в команде
```

!! Важно

- Если размещение нескольких байтов в сегменте задано транслятору одной директивой , символический адрес относится к ее первому байту

```
a1   db  5, 1, 0,  4     ; a1 – адрес байта 05h
```

- Адрес любого последующего байта можно задать транслятору как арифметическое выражение с использованием символического адреса

будущее выполнение команды процессором:

```
mov al,  ds: a1+2        ; al ←00h
mov bl,  ds: a1+3        ; bl ←04h
```

Использование символических и числовых прямых внутрисегментных адресов

Пример. Содержимое сегмента данных в исходном тексте

```
dseg   segment
    x       db   5
    y       dw  10
    mass  db   1,  -2,  13,  4
dseg  ends
```

- После трансляции в сегменте будут созданы коды (в hex):

с адреса ds:0000 - 05 0A 00 01 FE 0D 04

- Надо прочитать 5-й байт (FE) из сегмента данных в регистр al.

Прямой адрес байта можно написать транслятору разными способами:

```
mov al, ds: [4]
mov al, ds: mass+1
mov al, ds: y+3
```

и т.д.

Директивы транслятору для указания длины операнда

Иногда по тексту исходной программы транслятор не понимает, с какой длиной операнда из памяти должна выполняться команда. И дает сообщения об ошибках при трансляции.

В этих случаях перед адресом операнда для транслятора пишут указание его длины : byte ptr, word ptr или dword ptr.

Пример 1. В директиве размещения данных вы использовали один формат данных, а в команде надо работать с другим форматом.

```
data   dw  5        ;  формат данных - слово
    . . . .
    add  bh, ds: data   ;  при трансляции команды в код получите
```

предупреждение, т.к. это сложение байтов, а вы размещали число в формате слова

Поэтому, надо добавить указание «длины операнда»

```
add  bh,  byte ptr ds: data     ;
```

Пример 2.

Вы хотите сложить двухбайтные коды: один - в памяти и адрес задан косвенно, второй – непосредственный операнд.

```
add   ds:[si],  4
```

В такой символической команде транслятор не может понять длину используемых операндов. Даст сообщение об ошибке трансляции команды в код

Решение: добавим в команду явное указание «длины операнда»

```
add  word ptr  ds:[si], 4
```

Символические адреса команд (метки команд)

Место размещения команды в кодовом сегменте можно «пометить» символическим адресом (записывается с двоеточием :) и использовать этот адрес в командах передачи управления.

Пример:

```
adr:   add  ds:[si], bx
       .  .  .  .  .
       jmp  adr       ; переход на команду с
                      внутрисегментным адресом adr
```

Запись числовых величин в исходном тексте

Числовые непосредственные операнды для транслятора можно записать в 2-м, 8-м, 16-м или 10-м виде. Он переведет число в машинный код. Для числа в любой системе, кроме 10-й, надо указать транслятору его признак:

b – записано в 2-м виде (binary), q - 8-м виде, h - 16-м виде (hex)

- Если число в hex начинается с буквы (например, F7), перед буквой надо писать ноль - 0F7h, чтобы транслятор понял, что это число, а не символическое имя

Примеры записи чисел в командах процессора и директивах транслятора:

```
mov bl, 31        ; число без знака в 10-м виде
add  bx, -9       ;  число со знаком в 10-м виде
add  bl, 11111b   ; число в 2-м  виде
mov bl, 0F7h      ; число в 16-м виде
                  db  1, 1Fh, -4, 5
                  dw  10 dup (0FFh)
```

Запись символьных данных

Символ для транслятора надо записывать в кавычках .
Транслятор преобразует его в однобайтный ASCII код.

Пример:

```
mov al,  'T'      ; 'T' будет заменен на 54h  (ASCII код)

mov ax,  '25'     ;  ax ← 32 35h  (32h - код  ASCII  символа «2»
                                   35h - код  ASCII символа «5» )
```

Размещение последовательности символов в памяти

Разместить в сегменте последовательность из 4 символов. Как записать для транслятора?

```
;  можно записать каждый символ в отдельных кавычках
      st   db   ‘1’, ’!’, ’3’, ’*’

;  можно последовательностью в общих кавычках
      st   db    ‘1!3* ’
```

Каков символический внутрисегментный адрес символа * ?

st+3

Пример: Исходный текст программы с символическими внутрисегментными адресами данных и команд (синим)

Цель: Получить сумму двух однобайтных беззнаковых чисел.

Наш личный выбор и решение:

1) Структура исходного кода: пусть будет односегментная - только кодовый сегмент

2) Какие данные размещаем в памяти:

- числовые байты (их адреса cs:a1 и cs:a2)
- двухбайтную сумму (адрес cs:sum)

3) Алгоритм для процессора:

- расширить байты беззнаково до 2-х байтных форматов пересылкой в двухбайтные регистры (AX, BX)
- сложить содержимое регистров и сумму оставить в BX
- записать сумму из регистра BX в память по адресу cs:sum

```
.386
cseg segment  use16
assume cs: cseg
m1:   movzx ax, cs:a1
      movzx bx, cs:a2
      add   bx, ax
      mov cs:sum, bx
;завершение исполнения вызовом в ОС
      mov ah, 4ch
      int 21h
; размещение данных
a1    db  3
a2    db  75h
sum   dw   ?
cseg ends
      end  m1
```