Лексемы, предложения, директивы, выражения

На прошлой лекции мы уже рассматривали, как записываются команды в машинном коде и как они представляются в памяти компьютера. Они делятся на множество групп и для каждой группы необходимо по частям собирать биты и байты её кода и записывать получившуюся конструкцию в правильном порядке, чтобы получить машинную инструкцию. ОДНУ. А таких инструкций даже в простейшей программе может быть несколько десятков.

Поэтому понятно, что программировать на машинном языке сложно, долго и нудно. Одна из причин этого – цифровая форма записи команд и данных, в то время как для людей более привычная буквенно-символьная запись. В любом случае вас никто не обязывает с этим разбираться, и всё подводится к тому, что из-за сложности ассемблера появилась идея написания программы, которая будет транслировать код программы на языке более понятном человеку (он и получил название ассемблер) в бинарный код, выполняющийся на машине.

Для любой ЭВМ можно придумать различные языки ассемблера, хотя бы потому, что можно по-разному обозначать машинные операции. В частности, и для ПК разработано несколько таких языков. В общем и целом, языки ассемблера отличаются в зависимости от архитектуры процессора, под который они делаются. Более того, в настоящее время, помимо компьютерных процессоров (таких, как Intel), существует ещё большое множество микроконтроллеров от разных производителей, и чаще всего в них используются свои архитектуры, свои способы записи машинных кодов и как следствие свои языка ассемблера. В дальнейшем мы будем придерживаться архитектуры процессора x8086.

Лексемы

Изучения ЯА начнём с рассказа о том, как в нём записываются лексемы – такие простейшие конструкции, как имена, числа и строки.

Идентификаторы

Идентификаторы нужны для обозначения различных объектов программы – переменных, меток, названий операций и т.п. В ассемблере идентификатор – это последовательность из латинских букв (больших и малых), цифр и следующих знаков: ? . @ _ $. На эту последовательность накладываются следующие ограничения:

Длина идентификатора может быть любой, но значащими являются только первые 31 символ

Идентификатор не должен начинаться с цифры

Точка может быть только первым символом идентификатора

В идентификаторах одноимённые и большие и малые буквы считаются эквивалентными (AX, aX, Ax и тд – одно и то же)

Также символы $ и ? имеют специальные значения для ассемблера и к использованию не рекомендуются.

В идентификаторах разрешается использовать буквы английского алфавита. Идентификаторы делятся на служебные слова и имена. Служебные слова имеют заранее определённый смысл, они используются для обозначения таких объектов, как регистры (ax, si и тд), названия команд (ADD, NEG и тд). Все остальные идентификаторы называются именами, программист может пользоваться ими по своему усмотрению, обозначая ими переменные, метки и другие объекты программы. В качестве имён

можно использовать и служебные слова, однако крайне не рекомендуется этого делать.

Целые числа

В ЯА имеются средства записи целых и вещественных чисел. Целые числа можно записывать в двоичной, восьмеричной, десятичной или шестнадцатеричной системах счисления. Другие СС не допускаются. При записи чисел в десятичной форме ничего дополнительного указывать не нужно. При записи в двоичной системе, в конце ставится буква b (binary), в восьмеричной – o (octal). При записи в шестнадцатеричной СС ставится буква h (hex). Если число в шестнадцатеричной СС начинается с буквы, то перед ней нужно поставить незначащий 0.

Символьные данные

Символы заключаются в одинарные или двойные кавычки. Естественно, что левая и правая кавычки должны быть одинаковыми. Строки (последовательности символов) также заключаются в одинарные или двойные кавычки. В качестве символов допускается использовать кириллицу. В строках и символах, в отличие от идентификаторов, большие и маленькие буквы не отождествляются. Если в строке должна находиться такая же кавычка, которыми обрамлена строка, то её необходимо просто удвоить.

Предложения

Программа на ЯА – это последовательность предложений, каждое из которых записывается в отдельной строке. Переносить предложение на следующую строку или записывать несколько предложений на одной строке нельзя. Если в предложении более 131 символа, то 132-ой и все последующий символы игнорируются. При записи предложений действуют следующие правила расстановки пробелов:

Пробел обязателен между рядом стоящими идентификаторами и/или числами (чтобы отделить их друг от друга)

Внутри идентификаторов и чисел пробелы недопустимы

В остальных местах пробелы можно ставить или не ставить

Там, где допустим один пробел, допустимо ставить и более

Эти правила не относятся к пробелам внутри строк, где пробел – обычный значащий символ.

По смыслу все предложения делятся на три группы: комментарии, команды, директивы (приказы ассемблеру).

Комментарии

Комментарии не влияют на смысл программы, так как при трансляции, ассемблер игнорирует их. Они предназначены для людей, они поясняют смысл программы. Однострочный комментарий идёт после знака «точки с запятой». Также есть возможность определить многострочный комментарий. Для этого он должен начинаться со строчки: COMMENT <маркер> <текст>. Маркер, который по совместительству является операндом директивы COMMENT - любой символ, который будет считаться концом комментария: весь участок текста вплоть до следующего появления этого символа будет ассемблером игнорироваться.

Команды

Предложения-команды – это символьная форма записи машинных команд, то есть непосредственных ассемблерных инструкций. Общий синтаксис этого типа предложений таков (все, что в квадратных скобках, не является обязательным):
[<метка>:]<мнемокод> [<операнд>][;<комментарий>]

Метка

Синтаксически, метка – это имя. Если метка есть, то после неё обязательно ставится двоеточие. Метка нужна для ссылок на команду из других мест программы: например, для того, чтобы осуществить инструкции условного или безусловного перехода или вычислить адрес в памяти, по корому которому находится та инструкция или данные, перед которыми стоит метка. По сути, установка после идентификатора двоеточия указывает ассемблеру, что нужно создать переменную с именем данного идентификатора, содержащую адрес текущей команды, что и является меткой.

Мнемокод

Мнемокод (мнемонический код) является обязательной частью команды. Это служебное слово, указывающее в символьной форме операцию, которую должна выполнить команда. В ЯА не используются цифровые коды операций, операции указываются только своими символьными названиями, которые, конечно, легче запомнить (слово мнемонический означает легко запоминающийся).Простые примеры - xor ax, ax; mov bx, ax. В качестве операндов, как могли заметить по первой инструкции, могут выступать даже одни и те же регистры. Более того, маленькое отклонение – помните ли, что такое побитовое исключающее или, почему оно дает именно такой результат и почему, по-вашему, нельзя просто написать mov ax, 0 и для обнуления этого регистра используется такая инструкция? Дело в том, что любые побитовые операции процессором выполняются быстрее, чем простая пересылка.

Операнды

Операнды команды, если они есть, отделяются друг от друга запятыми. Операнды обычно записываются в виде выражений. В частных случаях, выражениями являются числа и имена переменных. В отличие от машинного языка, в ассемблере подставляют не адреса, а имена переменных.

Каждая программа на языке ассемблера помимо команд процессора содержит еще и специальные инструкции, указывающие самому ассемблеру, как организовывать различные секции программы, где располагаются данные, а где команды, позволяющие создавать макроопределения, выбирать тип используемого процессора, налаживать связи между процедурами и т. д. Единого стандарта на эти команды нет, но два самых популярных ассемблера для DOS и Windows, TASM и MASM, поддерживают общий.

Директивы

Помимо машинных команд в программе на ЯА надо указывать и другие вещи. Например, надо сообщать, какие константы и переменные используются в программе и какие имена мы им дали. Это делается с помощью предложений, называемых приказами ассемблеру или директивами. Синтаксис директив следующий:
[<имя>] <название директивы> [<операнды>] [;<комментарий>]
X db 10,-5,0xFF

Директивы определения данных

Для описания переменных, с которыми работает программа, в ЯА используются директивы определения данных. Одна из них предназначена для описания данных размером в байт, вторая – для описания данных размером в слово, а третья – для описания данных размером в двойное слово, также есть директивы для учетверенного слова (8 байт) и для размера данных в 10 байт, хотя две последние отсутствуют в некоторых вариациях языка ассемблера. В остальном эти директивы практически не отличаются друг от друга.

Такие директивы называются псевдокомандами: они приводят к включению в программу каких-либо данных или кода, но сами никаким командам и инструкциям процессора не соответствуют. Директивы определения данных указывают ассемблеру, что в соответствующем месте программы располагается переменная, устанавливают её тип, задают начальное значение и ставят в соответствие переменной метку, которая будет использоваться для обращения к этим данным. Все они записываются в виде <имя> d* <значение>, вместо *один из предопределенных символов. Поле значения может содержать одно или несколько чисел, строк символов, операторов ? и DUP.

Например, директива text_string db ‘Hello, world!’заполнит данными 13 байт, которые будут содержат ASCII-коды символов соответствующей строки, и text_string будет указывать на первую букву строки, так что последующая команда mov al, text_string поместит в al число 48h, соответствующее коду буквы H.

Директива DB

[<имя>] <DB> операнд {,операнд…}

Встречаю такую директиву, ассемблер вычисляет операнды и записывает из значения в последовательные байты памяти. Первому из этих байтов даётся указанное имя, по которому на этот байт можно будет ссылаться из других мест программы. Существует два основных способа задания операндов директивы DB:

? (знак неопределённого значения: переменная считается неинициализированной и её значение на момента запуска может оказаться каким угодно)

Константное выражение со значением от -128 до 255

X DB ?.

Ассемблер вставляет соответствующий байт прямиком в код. Поэтому, нельзя использовать эту директиву между командами, если, это делается не специально. Адрес ячейки, выделенной переменной X под хранение байта, принято называть ЗНАЧЕНИЕМ ИМЕНИ X (&X). Также, ассемблер запоминает размер переменной, который берёт из объявления. Этот размер называется ТИПОМ ИМЕНИ переменной (в ассемблере тип – это всего лишь размер, т.е. число, в отличие от того же C, например, где переменные одного размера могут иметь разный тип). Также есть оператор TYPE. Им можно задать тип. Пример:

Q DB ?

W DB TYPE Q

Это эквивалентно: W DB 1, потому, что размер типа Q == 1.

Можно использовать данную директиву определения, чтобы задать последовательность нужных данных, либо просто зарезервировать место. DB можно использовать так:

DB 1,2,3,4,5,….n

DB ‘a’,’s’,’s’,-1, 9
DB ‘asdasdasd’,0
Операнд – конструкция повторения DUP
Довольно часто в директиве DB нужно указывать повторяющиеся элементы (одинаковые операнды). Например, если мы хотим описать байтовый массив R из 8 элементов с начальным значением 0 для каждого из них, то это можно сделать так:
R DB 0,0,0,0,0,0,0,0
Но, можно сделать и так:
R DB 8 DUP(0).
Здесь в качестве операнда использована так называемая конструкция повторения, в которой сначала указывается коэффициент повторения, затем – служебное слово DUP (duplicate), а за ним в круглых скобках – повторяемая величина.
В общем случае эта конструкция имеет следующий вид:
count dup (p1, p2, p3, …, pn)
Где count – константное выражение, count >=1, pi – любой допустимый операнд директивы DB (значения, разделенные запятыми, в том числе даже вложенные операторы DUP). Данная запись является сокращением для count раз повторенной последовательности указанных в скобках операндов. Например:
X db 2 dup (‘ab’, ?, 1) ; x db ‘ab’,?,1,’ab’,?,1
Y db -7, 3 DUP(0, 2 DUP(?)) ; y db -7, 0,?,?,0,?,?,0,?,?
Вложенность конструкций DUP можно использовать для описания многомерных массивов.
A DB 20 DUP (30 DUP (?)) можно рассматривать как матрица 20x30.
С директивами DW и DD всё тоже самое, за исключением того, что нужно будет учитывать перевёртывание байт в памяти.
Директивы эквивалентности и присваивания
Мы рассмотрели, как в ЯА описываются переменные. Теперь рассмотрим, как в этом языке описываются константы. Это делается с помощью директивы эквивалентности – директивы EQU (equal), имеющей синтаксис:
<имя> EQU <операнд>
Эквивалент данной директивы на С - директива препроцессора #define <имя> <операнд>. Имени присваивается значение операнда, причем это может быть и целое число, и адрес, и любая строка символов. Применяется, как и #define, с целью введения параметров, общих для всей программы.
Например, само по себе имя регистра AX ничем не говорит, но если мы в нём храним сумму, можно сделать так: SUM EQU AX, тогда вместо “AX” можно будет писать SUM. Также константы полезны при объявлении длины массивов, в общем, всё тоже самое, как и в ЯВУ.
Теперь рассмотрим ещё одну директиву ЯА, похожую на директиву EQU – директиву присваивания:
<имя> = <константное выражение>
Эта директива определяет константу с именем, указанным в левой части, и с числовым значением, равным значению выражения справа. Но в отличие от констант, определённых по директиве EQU, данная константа может менять своё значение, обозначая в разных частях текста программы разные числа. Например:

```
K=10
A DW K ;A DW 10
K = K+4;
B DB K ;B DB 14
```

Подобного рода константы можно использовать «ради экономии имён»: если в разных частях текста программы используются разные константы и области использования этих констант не пересекаются, тогда, чтобы не придумывать новые имена, этим константам можно дать одно и то же имя (другими словами, поменять значение константы с таким именем). Однако, главное применение таких констант – в макросредствах, о которых мы поговорим позднее.

Если с помощью директивы EQU можно определить имя, обозначающее не только число, но и другие конструкции, то по директиве присваивания можно определить только числовую константу. Кроме того, если имя указано в левой части директивы EQU, то оно не может появляться в левой части других директив (его нельзя переопределять). А вот имя, появившееся в левой части директивы присваивания, может снова появиться в начале другой такой директивы (но только такой!).

```
K=1
N EQU K
A DW N ;A=1
K=2
B DW N ;B=2
K=1
N EQU K+10
C DW N ;c=11
K=2
D DW N ;d=11
```

Другими словами, при директиве EQU значение слева – синоним значения справа и ассемблер будет подставлять значение справа вместо встретившегося синонима слева. При директиве присваивания, константа слева вычисляется сразу.

Директива end.

```
end start_label
```

Этой директивой завершается любая программа на ассемблере. В роли операнда выступает метка или выражение, определяющее адрес, с которого выполнение программы начинается. Если в программе несколько модулей, только в одном файле может быть начальный адрес, равно как и в многофайловых программах на C только в одном может быть функция main().

Операторы

Арифметические: сложение +, вычитание -, умножение *, целочисленное деление /, остаток от деления Mod

```
tab_size equ 50 ;размер массива в байтах
size_el equ 2 ;размер элементов
…
;вычисляется число элементов массива
;и заносится в регистр cx
```

mov cx,tab_size / size_el ;оператор “/”
Сдвиг: shr - правый, shl - левый. Сдвиг может быть использован для более быстрого деления или умножения на степени двойки. «Выдвинутый» байт помещается во флаг CF.
Пример:
mask_b equ 10111011
…
mov al,mask_b shr 3 ;
Сравнение: eq ==, ne !=, < lt, <= le, > gt, > ge. Возвращает 1, если условие выполняется, иначе 0.
tab_size equ 30 ;размер таблицы
…
mov al,tab_size ge 50 ;загрузка размера таблицы в al
cmp al,0 ;если tab_size меньше 50, то
je m1 ;переход на m1
…
m1:
…
Команда cmp сравнивает значение al с нулем и устанавливает соответствующие флаги в flags/eflags. Команда je на основе анализа этих флагов передает или не передает управление на метку m1.
Индексный: base[index] - сложить базовый адрес со смещением.
mov ax,mas[si] ;пересылка слова по адресу mas+(si) в регистр ax
Переопределение/явное указание типа: new_type ptr
new_type: byte 1, word 2, dword 4, qword 8, tword 10. near и far - ближние и дальние указатели
d_wrd dd 0
…
mov al,byte ptr d_wrd+1 ;пересылка второго байта из двойного слова
Длина массива: length. Определяет количество элементов в нем (!).
Array dw 10 dup (0)
mov ax, length Array; ax = 10
Размер массива: size. Определяет его размер в байтах (!).
Array dw 10 dup (0);
mov ax, size Array; ax = 20
Определение размера типа: type. Определяет тип одного элемента (тип в ассемблере = размер в байтах).
A db 0
Array dw 100 dup (?)
mov ax, type A; ax = 1
mov bx, type array; bx = 2
Оператор () (круглые скобки). Часть выражения, заключенная в них, вычисляется в первую очередь.
Унарный минус ставится перед отрицательным числом.
Побитовые операторы and, not, or, xor (и, не, или, исключающее или).

mov ax, 1234h and 4321h ; mov ax, 0220h
seg и offset: возвращают соответствующую часть своего аргумента (сегментную часть для seg и смещение для offset):
mov dx, offset msg ; занести в dx смещение переменной msg в её сегменте
Приоритет операторов: определяет, что выполнится первым, если нет скобок.
1: length, size, (), []
4: ptr, offset, seg
6: - (унарный).
7: *, /, mod, shl, shr
8: +, - (бинарные)
9: eq, ne, lt, le, gt, ge
10: not
11: and
12: or, xor
13: type
Выражения
Операнды директив, как правило, описываются в виде выражений. Выражения используются для описания операндов команд. В целом выражения ЯА похожи на арифметические выражения языка высокого уровня, однако между ними есть и отличия. Наиболее важное отличие заключается в том, что выражения ЯА вычисляются не во время выполнения программы, а во время её трансляции: встретив в тексте программы выражение, ассемблер вычисляет его и полученное значение записывает в машинную программу. Поэтому, когда программа начинает выполняться, от выражений не остаётся ни следа. В связи с этим, в выражениях можно использовать только значения, известные на этапе трансляции программы, например, адреса и типы имён. Нельзя использовать величины, например, содержимое регистров или ячеек памяти, которые станут известными лишь во время выполнения программы.
Выражения делятся на константные и адресные. Константные:
K EQU 30
X DB (3*K-1)/2 DUP(?) ;44 байта

len = 10
i = (len/2) + 1 ; i = 6
Адресные:
В ассемблере разрешено вычитать адреса и из этого получать число.
К простейшим адресным выражениям относятся:
Метка и имя переменной, описанной директивой DX
Счётчик размещения, он записывается как $ и обозначает адрес текущего предложения: это единственная предопределенная метка, которая поддерживается большинством ассемблеров.
100: A dw $
102: B dw $-2 ; B = A
Конструкция
jmp $

выполняет безусловный переход на саму себя, что создает вечный цикл из одной команды.

Обобщая информацию о том, как записываются команды и директивы, структуру программы можно представить следующим образом: она состоит из строк, имеющих следующий вид:

метка команда/директива операнды ; комментарий

Причем все эти поля необязательны. Метка есть идентификатор; команда транслируется в исполняемый код (а директива управляет работой самого ассемблера, новый код не создавая). Если метка стоит перед командой, то после неё ставится двоеточие, которое указывает ассемблеру, что мы создаем переменную с именем метки, содержащую адрес текущей команды. Если метка стоит перед директивой, то она обычно является операндом этой директивы, тогда двоеточие не ставится.