Практика 1: hello world

Литература.

- ***http://asmworld.ru/uchebnyj-kurs***
- Зубков С.В., Ассемблер для DOS, Windows и UNIX. – М.: ДМК Пресс, 2000. – 608 с.: ил. (Серия «Для программистов»)
- Нортон П., Соухэ Д. Язык ассемблера для IBM PC.– М.: Компьютер, 1992.– 352 с.

Вступление. Что такое ассемблер .

Ассемблер (Assembly) — язык программирования, понятия которого отражают архитектуру электронно-вычислительной машины. Язык ассемблера — символьная форма записи машинного кода, использование которого упрощает написание машинных программ. В отличие от языков высокого уровня – C# или Java - , в которых многие проблемы реализации алгоритмов скрыты от разработчиков, язык ассемблера тесно связан с системой команд микропроцессора. В идеальном случае ассемблер вырабатывает по одному машинному коду на каждый оператор языка; однако на практике для реальных микропроцессоров может потребоваться несколько машинных команд для реализации одного оператора языка, что обусловлено архитектурой ЭВМ.
Язык ассемблера обеспечивает доступ к регистрам, указание методов адресации и описание операций в терминах команд процессора. Он может содержать средства более высокого уровня абстракции

- встроенные и определяемые макрокоманды, соответствующие нескольким машинным командам
- автоматический выбор команды в зависимости от типов операндов
- средства описания структур данных.

Главное достоинство языка ассемблера — «приближенность» к процессору, который является основой используемого программистом компьютера, а главным неудобством — слишком мелкое деление типовых операций, которое большинством пользователей воспринимается с трудом. Однако язык ассемблера в значительно большей степени отражает само функционирование компьютера, чем все остальные языки. Для успешного использования ассемблера необходимы сразу три вещи:

- знание синтаксиса транслятора ассемблера, который используется (например, синтаксис MASM, FASM и GAS отличается), назначение директив языка ассемблер (операторов, обрабатываемых транслятором во время трансляции исходного текста программы);
- понимание машинных инструкций, выполняемых процессором во время работы программы;
- умение работать с сервисами, предоставляемыми операционной системой — в данном случае это означает знание функций конкретной используемой ОС (например, вызовы DOS или Win32 API).

При работе с языками высокого уровня очень часто к API системы программист прямо не обращается; он может даже не подозревать о его существовании, поскольку библиотека языка скрывает от программиста детали, зависящие от конкретной системы. Например, и в Linux, и в Windows, и в любой другой системе в программе на Си/Си++ можно вывести строку на консоль, используя функцию printf() или поток cout, то есть для программиста, использующего эти средства, нет разницы, под какую систему делается программа, хотя реализация этих функций будет разной в разных системах, потому что API систем очень сильно различается. Но если человек пишет на ассемблере, он уже не имеет готовых функций типа printf(), в которых за него продумано, как «общаться» с системой, и должен делать это сам.

В итоге получается, что для написания даже простой программы на ассемблере требуется весьма большое количество предварительных знаний — «порог вхождения» здесь намного выше, чем для языков высокого уровня. Высокий «порог вхождения» компенсируется максимальной

эффективностью написанных на языке ассемблера программ. Им пользуются в тех случаях, когда критически важны:

- **объем используемой памяти** (программы-загрузчики, встраиваемое программное обеспечение, программы для микроконтроллеров и процессоров с ограниченными ресурсами, вирусы, программные защиты и т.п.);
- **быстродействие** (программы, написанные на языке ассемблера выполняются гораздо быстрее, чем программы-аналоги, написанные на языках программирования высокого уровня абстракции. В данном случае быстродействие зависит от понимания того, как работает конкретная модель процессора, реальный конвейер на процессоре, размер кэша, тонкостей работы операционной системы. В результате, программа начинает работать быстрее, но теряет переносимость и универсальность).

Кроме того, знание языка ассемблера облегчает понимание архитектуры компьютера и работы его аппаратной части, то, чего не может дать знание *языков высокого уровня абстракции* (ЯВУ). В настоящее время большинство программистов разрабатывает программы в *средах быстрого проектирования* (Rapid Application Development) когда все необходимые элементы оформления и управления создаются с помощью готовых визуальных компонентов. Это существенно упрощает процесс программирования. Однако, нередко приходится сталкиваться с такими ситуациями, когда наиболее мощное и эффективное функционирование отдельных программных модулей возможно только в случае написания их на языке ассемблера (ассемблерные вставки). В частности, в любой программе, связанной с выполнением многократно повторяющихся циклических процедур, будь это циклы математических вычислений или вывод графических изображений, целесообразно наиболее времяемкие операции сгруппировать в программируемые на языке ассемблера субмодули. Это допускают все пакеты современных языков программирования высокого уровня абстракции, а результатом всегда является существенное повышение быстродействия программ.

Языки программирования высокого уровня абстракции разрабатывались с целью возможно большего приближения способа записи программ к привычным для пользователей компьютеров тех или иных форм записи, в частности математических выражений, а также чтобы не учитывать в программах специфические технические особенности отдельных компьютеров. Язык ассемблера разрабатывается с учетом специфики процессора, поэтому для грамотного написания программы на языке ассемблера требуется, в общем, знать архитектуру процессора используемого компьютера. Однако, имея в виду преимущественное распространение PC-совместимых персональных компьютеров и готовые пакеты программного обеспечения для них, об этом можно не задумываться, поскольку подобные заботы берут на себя фирмы-разработчики специализированного и универсального программного обеспечения.

Hello world.

Для процессора x86-x64, имеется более десятка различных ассемблер компиляторов. Они отличаются различными наборами функций и синтаксисом. Из имеющегося разнообразия компиляторов выберем **fasm**. Он обладает рядом преимуществ, и в тоже время достаточно простым интерфейсом (что несомненно достоинство, достаточно взглянуть на masm и убедится в этом). Из преимуществ отметим:

- поддержку 16, 32 и 64 битной разработки
- переносимость программ в том смысле, что компилятор имеет версии под все основные ОС и проект может быть собран в любой из них, если он не использует специфические инструкции конкретной ОС
- качественную документацию.

Рассмотрим простое приложение из примеров fasm – **examples\hello\hello.asm** :

```
include 'win32ax.inc'

.code

  start:
      invoke  MessageBox,HWND_DESKTOP,"May I introduce
myself?",invoke GetCommandLine,MB_YESNO
      .if eax = IDYES
          invoke  MessageBox,HWND_DESKTOP,"Hi! I'm the
example program!","Hello!",MB_OK
      .endif
      invoke  ExitProcess,0
.end start
```

Это пример простого приложения на asm под windows . Общий вид строки программы на ассемблере имеет вид

<метка:> <инструкция/директива/макрос> <операнды> <; комментарий>

где

- **метка** - слово, не совпадающее ни с одним зарезервированным. Чаще всего используется в условном операторе для перехода к этой строке кода.
- **инструкция/директива** - команда на языке ассемблера
- **операнды** - требуемые операнды для команды. Число или вообще их наличие зависит от использованной команды языка
- **комментарий** - пояснение к строке кода в произвольном виде

Строка может состоять и из неполного набора полей, как например первая строка - здесь только комментарий.

Далее следует подключение дополнительного файла с исходным кодом приложения. Рассмотрим его чуть ниже. Следующая строка это объявление сегмента кода. Приложение должно как минимум содержать сегмент кода (раздел исполняемого файла, содержащий инструкции программы) и метку начала исполнения - **start:** - в данном случае. Сегмент кода в нашем примере продолжается до строки **.end start** .
В теле приложения идут вызовы системных процедур ОС Windows, а именно

- **MessageBox** – создать окно сообщения
- **ExitProcess** – завершить работу приложения

Вызов последней обязателен для корректного завершения работы.

Типы приложений на Fasm.

Вообще для программирования под Windows существует три типа приложений

1. .com
2. MZ формат файлов .exe
3. PE фромат файлов .exe

Первый пример приложения типа .com – немодифицированный образ 16-тибитной программы, самый простой из типов.

```
use16
org 100h
mov dx, datastr
mov ah,9
int 21h

mov ax, 4c00h
int 21h
datastr db 'hello world!$'
```

Директива «use 16» указывает на генерирование 16-битного кода. «org 100h» объявляет пропуск 256 байт (адреса 0000h – 00FFh). Указанные адреса зарезервированы под служебные данные (PSP).
Далее следуют команды. Завершение работы программы осуществляется вызовом функции 4C прерывания 21h. Обращение к прерыванию 21h это обращение к функциям ОС MS-DOS. Соответственно, приложение пригодно для запуска только под MS DOS и непереносимо на другие ОС. MS Windows эмулирует работу MS DOS при помощи процесса cmd.exe. Последней строкой указан пример размещения в памяти текстовой строки. Строка это просто набор символов, завершающийся специальным – терминальным – символом, обозначающим конец строки. Если его не указать, то строка продолжится и будет содержать «лишние» данные. В MS-DOS терминальным символом является $.
MZ — стандартный формат 16-битных исполнимых файлов с расширением .EXE для DOS. Назван так по сигнатуре — ASCII-символам MZ (4D 5A) в первых двух байтах.

```
format MZ
entry code_seg:start
stack 200h
segment code_seg
start:
mov ax,data_seg
mov ds,ax
mov dx, datastr
mov ah,9
int 21h
mov ax, 4c00h
int 21h

segment data_seg
```

datastr db 'hello world!$'

Для создания нужно использовать директиву «format MZ». По умолчанию код для этого формата 16-битный.

- **«segment»** определяет новый сегмент, за ним должна следовать метка, чьим значением будет номер определяемого сегмента. Опционально за этой директивой может следовать «use16» или «use32», чтобы указать разрядность кода в сегменте. Начало сегмента выровнено по параграфу (16 байт). Все метки, определенные далее, будут иметь значения относительно начала этого сегмента. В примере выше объявляются 2 сегмента: «data_seg» и «code_seg».
- **«entry»** устанавливает точку входа для формата MZ, за ней должен следовать дальний адрес (имя сегмента, двоеточие и смещение в сегменте) желаемой точки входа. В нашем случае объявлена метка «start».
- **«stack»** устанавливает стек для MZ. За директивой может следовать числовое выражение, указывающее размер стека для автоматического создания, либо дальний адрес начального стекового фрейма, если вы хотите установить стек вручную. Если стек не определен, он будет создан с размером по умолчанию в 4096 байт.

Неопределенная в примере директива **«heap»** со следующим за ней значением определяет максимальный размер дополнительного места в параграфах (это место в добавление к стеку и для неопределенных данных). Используйте «heap 0», чтобы всегда отводить только память, которая программе действительно нужна.

Следует помнить, что программы типа COM и Формат MZ не поддерживаются 64-разрядными ОС Windows. Для запуска таких программ под этими операционными системами следует использовать программу DOSBox – эмулятор среды MS DOS.

Далее стоит упомянуть про программирование под Linux на FASM. В этом случае код приложения Hello World выглядит так :

```
format ELF


section '.data' writeable
msg db 'Hello, world!', 0
formatStr db "%s", 0


section '.text' executable
public main
extrn printf
main:
mov ebp, esp; for correct debugging
push msg
push formatStr
call printf
add esp, 8
xor eax, eax
ret
```

Формат ELF является стандартным форматом исполняемых файлов под Unix совместимые ОС. По структуре не сильно отличается от ранее рассмотренного MZ. В данном примере содержит два блока

- data — данные
- text — исполняемый код

и использует вызовы posix в своей работе, конкретнее вызов printf — форматированный вывод строки в консоль. Для завершения работы приложения используется вызов ret — возврат из процедуры.
Существенных отличий в плане изучения языка ассемблера под Windows и Linux нет, так что воспользуемся пока более

распространенной платформой — Windows. Желающие могут свободно адаптировать все приведенные далее примеры под Linux, используя любую доступную оболочку для программирования, например SipmleASM.

Последний пример это формат PE — сокращение от Portable Executable, т.е. переносимый (универсальный) исполняемый файл. Он понимается всеми версиями Windows, начиная с Windows 3.1.

```
format PE console          ;Исполняемый файл Windows EXE
entry start                ;Точка входа в программу
include 'win32a.inc'
section '.text' code executable
start:
   push hello
   call [printf]
   push 0
   ccall   [getchar]
   call [ExitProcess]
section '.rdata' data readable
   hello db 'Hello World!', 0
section '.idata' data readable import
   library kernel32, 'kernel32.dll', \
      msvcrt,  'msvcrt.dll'
   import kernel32, ExitProcess, 'ExitProcess'
   import msvcrt, printf, 'printf', getchar,'_fgetchar'
```

Чтобы выбрать формат PE, нужно использовать директиву «format PE», за ней могут следовать дополнительные настройки формата:

- console» - консольное приложение
- «GUI» - графическое

- «native» - чтобы выбрать целевую подсистему (далее может следовать значение с плавающей точкой, указывающее версию подсистемы)
- «DLL» помечает файл вывода как динамическую связывающую библиотеку. Далее может следовать оператор «at» и числовое выражение, указывающее базу образа PE, и опционально оператор «on» со следующей за ним строкой в кавычках, содержащей имя файла, выбирающей заглушку MZ для PE программы (если указанный файл не в формате MZ, то он трактуется как простой двоичный исполняемый файл и конвертируется в формат MZ). По умолчанию код для этого формата 32-битный.

Пример объявления формата PE со всеми свойствами:

format PE GUI 4.0 DLL at 7000000h on 'stub.exe'

«section» определяет новую секцию, за ней должна следовать строка в кавычках, определяющая имя секции, и далее могут следовать один или больше флагов секций. Возможные флаги такие:

- «code»
- «data»
- «readable»
- «writeable»
- «executable»
- «shareable»
- «discardable»
- «notpageable».

Начало секции выравнивается по странице (4096 байт). Пример объявления секции PE:

section '.text' code readable executable

Секций кода или данных может быть несколько внутри одного исполняемого файла.
Заметим, что в нашем примере большая часть синтаксиса отсутствует. Связанно это с включением файла win32ax.inc, а не win32a.inc. Файл win32ax.inc содержит большое число макросов, автоматизирующих основные процедуры в asm программе для Windows.
Объявление сегмента повторно приведет к его переопределению. Например, если в конце программы написать

```
section '.idata' data readable import
        library kernel32, 'kernel32.dll', msvcrt, 'msvcrt.dll'
        import kernel32, ExitProcess, 'ExitProcess'
        import msvcrt, printf, 'printf', scanf, 'scanf',
getchar,'getchar'
```

то этим вы переопределяете секцию .idata , определенную в включенном файле win32ax.inc . Фактически это приводит к тому, что все импортированные из библиотек функции вам становятся недоступны и пользоваться вы можете только теми вызовами, которые явно укажете в вашей секции .idata . В приведенном примере подключены только системный вызов ExitProcess и функции printf, scanf и getchar, причем из библиотеки, используемой языком С, а не операционной системой, что добавляет лишние трансляции вызовов.
Чтобы посмотреть какие именно системные вызовы доступны в fasm, можно посмотреть содержимое файлов в директории include/api . Там в явном виде указаны все подключаемые системные вызовы.

Запишем аналогичную программу, но более детально определив системные вызовы и используя синтаксис FASM. Более подробное определение системных вызовов необходимо

только в целях обучения, в работе использование специфических конструкций FASM не несёт никаких замедлений в работе приложения.

```
Format PE console
include `win32ax.inc`
.data
  hellostr DB `Hello world!`,0
  inputnum DD 0
.code
  start:

    push STD_OUTPUT_HANDLE
    call [GetStdHandle]
    push 0
    push inputnum
    push 13
    push hellostr
    push eax
    call [WriteConsole]

    push STD_INPUT_HANDLE
    call [GetStdHandle]
    push 0
    push inputnum
    push 3
    push hellostr
    push eax
    call [ReadConsole]

    push 0
    call [ExitProcess]
.end start
```

По сравнению с предыдущим приложением здесь можно заметить отличия:

- нет объявления entry start
- нет объявлений section .data и section .text
- полностью отсутствует секция .idata

Всё это скрыто в инструкции include . Мы включаем в состав файл win32ax.inc в котором — если обратить внимание на его исходный код - включены макросы .text и .data. В них то и определяются соответствующие блоки файла. Так же можно обратить внимание на макрос .end в котором объявляется блок .data с включение всех базовых функций win32api , в отличии от предыдущего примера, не включает функции библиотеки msvcrt.dll . Рекомендованным способом написания приложений является именно использование библиотеки win32api.

Последним вариантом написания этого приложения был бы вариант 64 кода, но в конкретно данном случае он бы отличался только включением другого файла заголовков

include `win64ax.inc`

что не существенно. В дальнейшем обучении существенных различий между 32бит и 64бит нет, так что продолжим рассмотрение в варианте 32бит.

Домашнее задание

Написать приложение, аналогичное hello.asm, на любом языке высокого уровня (C# к примеру) и сравнить объем полученного исполняемого файла. Объяснить разницу.