Пособие по курсу «Языки системного программирования»

Игорь Жирков
кафедра Вычислительной Техники СПбНИУ ИТМО

24 сентября 2012 г.

Оглавление

Введение

Зачем нужен этот курс? Его основные цели таковы:

- Показать в общем и целом, как работает процессор с точки зрения системного программиста;
- Научить студентов азам программирования на Ассемблере и C;
- Дать студентам кругозор по части стилей программирования в целом, позволяя им выбирать те инструменты, которые наилучшим образом подходят для решения конкретной задачи;
- Осветить работу компьютера и операционной системы настолько, чтобы у студентов сложилось общее представление о том, как запускаются и исполняются программы.

В данном пособии находится необходимый теоретический минимум по курсу, вопросы для самостоятельной проработки, а также список некоторых источников, полезных для самообразования в данной области.

В каждой главе вы найдёте как сведения по языкам программирования, так и более общетеоретический материал, направленный на установление логических связей между разными областями знаний, связанными с программированием вообще и системным программированием в частности.

В пособии вы найдёте в достаточном количестве специальным образом помеченные вопросы. Они оставляются вам на самостоятельную проработку — вы обязательно столкнётесь с ними на защите лабораторных работ, контрольных работах и экзамене.

Любые исправления, мнения и пожелания приветствуются на igorjirkov@gmail.com.

Глава 1

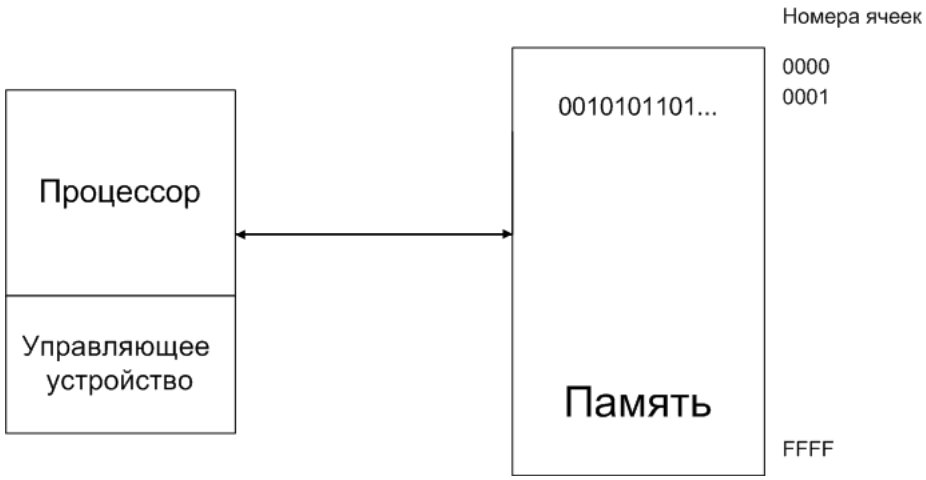
Архитектура компьютера и основы языка Ассемблера

1.1 Архитектура фон Неймана

Представим, что мы перенеслись на много лет назад, когда компьютеров еще не существовало. Существовала лишь необходимость каким-то образом организовать автоматический вычислительный процесс. Тогда на бумаге существовало множество совершенно различных моделей вычислительных систем. Это и машина Тьюринга, и лямбда-исчисление Алонзо Чёрча, и другие вычислительные модели. Это показывает нам, что организовать набор транзисторов или ламп в различные схемы можно совершенно различными способами, и компьютеры отнюдь не обязаны быть именно такими, какие они есть сейчас. В связи с тем, что ранее электронные компоненты были не очень надёжны, а также со своей относительной простотой в создании и написании для неё программ, прижилась концептуальная модель компьютера, получившая название «Архитектура фон Неймана». Её аппаратные реализации также были достаточно надёжны.

Архитектура компьютера — концептуальная структура вычислительной машины, определяющая процесс вычислений, а также то, как взаимодействует аппаратура и программное обеспечение.

Взглянем на схему компьютера с такой архитектурой:



На схеме вы видите процессор, который умеет выполнять команды; память, которая хранит данные и команды, и управляющее устройство, которое указывает процессору, какие команды выбирать из памяти для выполнения.

Какие основополагающие принципы такой архитектуры?

Двоичное кодирование В памяти хранятся только нули и единицы.

Однородность памяти Никаким способом нельзя узнать, команда ли лежит в данной ячейке, или данные. Процессор может попытаться выполнить данные, что, скорее всего, вызовет ошибку, так как произвольные данные врядли соответствуют корректно закодированной команде.

Адресуемость памяти Каждая ячейка имеет свой номер. Ячейки нумеруются последовательно: за нулевой ячейкой идёт первая и т.д. Единица адресации в реальных компьютерах — байт, то есть каждые 8 бит имеют свой адрес.

Последовательное программное управление Программа состоит из набора команд, которые выполняются последовательно.

Исключение – команды перехода, которые явно заставляют компьютер изменять порядок команд.

Принцип жесткости архитектуры Все связи и блоки на схеме остаются собой на протяжении работы компьютера. Никаких новых связей не возникает.

Помимо архитектуры фон Неймана существуют другие похожие архитектуры, например, гарвардская, их рассмотрение мы оставим за рамками нашего курса. Сейчас наша задача – проследить, как из фон Неймановской основы вырос типичный персональный компьютер, каковым он был 20 лет назад.

1.1.1 Модификация фон Неймановской модели

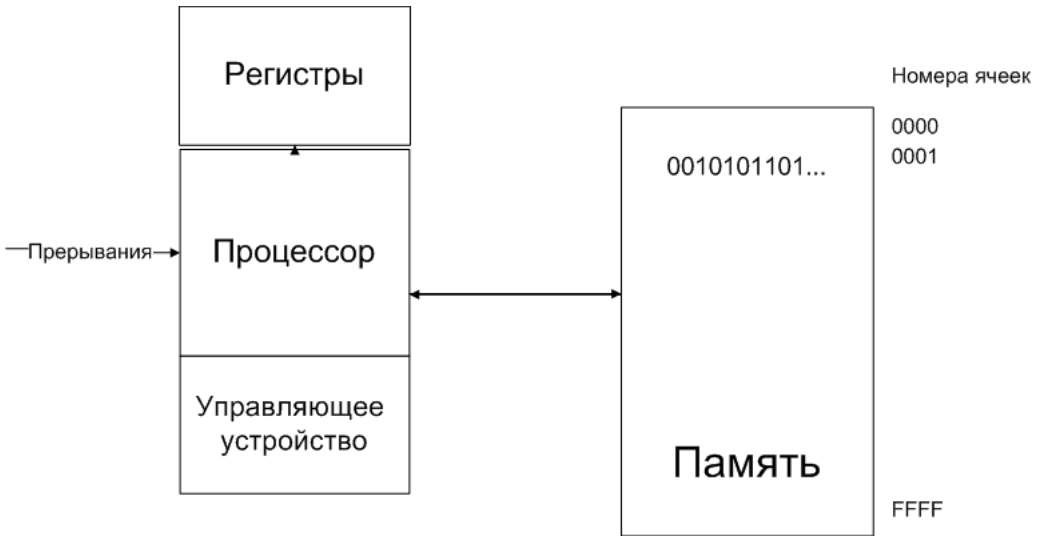
У архитектуры фон Неймана есть несколько серьёзных недостатков. Некоторые из них перечислены ниже:

- Система не интерактивна — никакого взаимодействия человека с системой, кроме прямого редактирования памяти;
- Узкое место системы — канал обмена между памятью и процессором. Для каждой команды необходимо не только обратиться в память, чтобы считать её, но и обращаться туда, чтобы считывать операнды, и, возможно, записывать результат.

Чтобы сгладить эти недостатки, было придумано несколько механизмов.

Прерывания — нарушение исполнения работы программы по специальному сигналу процессору. Процессор останавливает исполнение текущей программы и начинает исполнять программу обработки прерывания;

Регистры — ячейки памяти, расположенные непосредственно на кристалле процессора. Обращение к ним очень быстрое и не использует канал обмена с памятью. Основные операции компьютер осуществляет именно с содержимым регистров.



Разумеется, использование регистров в худшем случае замедляет работу компьютера. Быстрее было бы совершить действие с ячейкой памяти, нежели еще копировать её содержимое в регистр и из регистра. Однако существует объективное свойство компьютерных программ — локальность — которое делает использование регистров оправданным.

Локальность бывает двух видов: пространственная и временная.

Временная локальность заключается в том, что если произошло обращение по некоторому адресу, то следующее обращение по тому же адресу с большой вероятностью произойдет в ближайшее время.

Пространственная локальность заключается в том, что если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.

В программе, обычно, б'ольшую часть времени мы работаем с очень маленьким набором переменных. Если обращение в память за данными понадобится нам лишь в одном случае из десяти, то в этом случае мы потеряем немного производительности, зато в других — значительно ускоримся. Таким образом, в среднем мы получим прирост производительности.

Замечание. *Это очень распространённая ситуация в инженерной практике. За счёт дополнительных механизмов мы ухудшаем производительность в худшем случае, но поднимаем её в среднем. С похожей ситуацией вы столкнётесь, когда будете изучать кэши и виртуальную память в курсе «Организация ЭВМ и систем».*

1.2 Процессор Intel 80386

Мы начнём программировать на языке ассемблера для персонального компьютера, каким он был 20 лет назад. Это оправдано, потому что в дальнейшем процессор изменялся эволюционно, почти всегда сохраняя совместимость с более старыми моделями. Поэтому мы начнём с того, что обзорно изучим процессор Intel 80386 со стороны программиста.

Язык ассемблера — самый низкоуровневый язык из доступных программисту. Все команды ассемблера однозначно переводятся в машинные команды и наоборот.

Замечание. *Информация, помещённая в данном разделе, носит лишь обзорный характер. Категорически рекомендуется ознакомиться с самой документацией на процессор. Документация на процессор 80386 доступна онлайн на сайте Массачусетского университета.* `http://css.csail.mit.edu/6.858/2012/readings/i386/toc.htm`

1.2.1 Регистры

Некоторые регистры, которые можно использовать в арифметических операциях, называются регистры общего назначения. Их 8 штук, и все они имеют размер 16 бит. Хотя они и универсальны, у них также есть и особое назначение — в некоторых командах они используются неявно, без указания их имени. Из этого проистекают их названия.

- AX — аккумулятор, обычно используется в арифметических действиях;
- BX — база для базовой адресации;
- CX — счетчик цикла;
- DX — данные, например, прерывания DOS часто получают данные именно из DX;
- SI — Source Index, адрес ячейки-отправителя в командах работы со строками (MOVSB, ...);
- DI — Destination Index, адрес ячейки-получателя в командах работы со строками (MOVSB, ...);
- SP — хранит адрес последнего занесённого в стек элемента;
- BP — база для базовой адресации в случае, когда используются стековые фреймы. Хранит адрес начала текущего стекового фрейма (с этой темой вы подробно познакомитесь позднее).

К младшим и старшим 8 битам первых четырёх регистров (AX, BX, CX, DX) можно обращаться отдельно, используя имена AL, AH и т.п. Например, если DX=0103_{16}, то DH = 01_{16}, DL = 03_{16}.

Помимо этого существуют особые 16-битные регистры, с которыми вы также столкнётесь:

- IP — счётчик команд;
- Flags — в своих битах хранит флаги процессора.

  Вопрос 1. *Обратитесь к документации и прочитайте, какие флаги хранит этот регистр и за что они отвечают. Особое внимание уделите управляющим флагам DF, IF, TF.*

- CS, DS, SS, ES — 16-сегментные регистры, хранят кусок адреса начала сегмента кода, данных и стека текущей программы соответственно. Регистр ES используется для служебных целей программистом, например, в командах работы со строками (MOVSB, MOVSW...).

1.2.2 Сегментация

Так как регистры имеют разрядность 16 бит, а любые операции с адресами ячеек памяти задействуют регистры (хотя бы для хранения самого адреса), то количество ячеек, которые мы можем пронумеровать, ограничивается разрядностью регистра. Так, 16-битными регистрами мы можем перенумеровать $2^{16} = 65536$ байт. Однако программистам всегда не хватало памяти, и инженеры придумывали различные механизмы, чтобы расширить адресацию. Начиная с модели Intel 8086 стало возможным нумеровать 2^{20} ячеек памяти благодаря сегментным регистрам.

Теперь вместо того, чтобы указывать в регистре адрес ячейки относительно начала памяти (нулевого адреса), он указывается относительно начала определённого сегмента, то есть другого адреса.

Хотя для компьютера нет разницы, какие нули и единицы в памяти что кодируют, но программистам удобно держать в памяти раздельно код и данные. В любой программе, обычно, есть три области памяти:

- Код программы, машинные команды;
- Область данных;
- Область памяти, зарезервированная под стек.

По этой причине инженеры выделили три сегментных регистра, в которых программисту предлагается хранить адреса начала областей кода, данных и стека соответственно. Адрес начала сегмента формируется мз соответствующего сегментного регистра путём его умножения на 16.

Так, если CS = 0012_{16}, то предполагаемый компьютером адрес начала сегмента кода – 00120_{16}.

Различные команды подразумевают по умолчанию адреса относительно различных сегментов. Так, команда перемещения данных mov использует, конечно, смещение относительно начала сегмента данных, а команда перехода jmp — сегмента кода. Если из самой сути команды неясно, какой сегмент она использует, следует посмотреть раздел документации по соответствующей команде.

Теперь будем различать линейные адреса (20-разрядные) и адреса в форме сегмент:смещение и всегда обращать внимание на сегмент. Так, адрес следующей исполняемой команды лежит не в IP, а в паре CS:IP.

Например, CS=$042F_{16}$, IP=1212_{16}. Линейный адрес следующей исполняемой команды — $042F_{16} * 16_{10} + 1212_{16}$

```
042f0+
 1212=
```

05502 — адрес следующей исполняемой команды.

1.2.3 Аппаратный стек

Вообще, стек это структура данных типа Last In — First Out, которая ведёт себя подобно стопке тарелок. Новый элемент с помощью команды push помещается на вершину стека, а с помощью команды pop достаётся последний помещённый в стек элемент.

Нечто похожее на эту структуру данных реализовано аппаратно.

Память линейно адресуема, значит, никакой особенной стековой памяти у нас нет. Однако стек эмулируется с помощью машинных команд push, pop и пары регистров SS:SP. SS:SP хранит адрес вершины стека, а команды push и pop в реальном режиме работают по следующим алгоритмам:

- push
  1. SP уменьшается на 2;
  2. По адресу SS:SP кладётся значение (содержимое регистра или число, в зависимости от параметра команды).
- pop
  1. В регистр, указанный, в качестве параметра, кладётся значение, взятое по адресу из SS:SP;
  2. SP увеличивается на 2.

Из этого есть несколько важных выводов:

- Не бывает ситуации, когда "в стеке ничего нет даже если "мы туда ничего не положили". Команда pop всегда будет следовать вышеуказанному алгоритму и выдавать соответствующий результат;
- Стек растёт к младшим адресам (обычно говорят: "вниз но так как когда рисуют память, младшие адреса часто располагают сверху, то стрелка, указывающая на направление роста стека, на картинке направлена вверх. Это может дезориентировать!);
- В реальном режиме работы процессора в стек можно положить или вынуть оттуда только одно слово (2 байта). В защищённом — или два, или четыре байта, соответствующий регистр ESP будет уменьшаться или увеличиваться на 2 или 4.

1.2.4 Режимы работы процессора

Процессор 80386 может работать в реальном, защищённом, специальном или виртульном режимах. Мы поговорим о них позднее, пока что нам нужно запомнить несколько фактов:

1. Мы пока что работаем в реальном режиме работы. Практически все регистры в нём 16-разрядные.
2. В защищённом режиме работы все регистры общего назначения становятся 32-разрядными, к их имени добавляется префикс E. Например, регистр AX расширился до EAX (32-разрядный), но мы по прежнему можем обращаться к младшей половине EAX как к AX, а также к половинам AX как к AH и AL.
3. Специальный режим — режим сна, который вы, скорее всего, часто используете. Для студентов нормально работать в специальном режиме.

1.2.5 Прерывания

Когда процессор получает прерывание, он останавливает выполнение программы чтобы выполнить обработчик прерывания. Каждое прерывание имеет жёстко фиксированный номер. Для нас неважно, как именно процессор получает номер прерывания от контроллера прерываний. Имея номер X, мы обращаемся в память за адресом обработчика прерывания. В начале памяти лежит таблица векторов прерываний, каждый из которых занимает 4 байта (2 байта на адрес начала сегмента, 2 байта на смещение). Всего номеров прерываний 256, поэтому размер таблицы — 1 килобайт.

Алгоритм работы процессора при получении прерывания такой:

- Заносим в стек флаги;
- Сбрасываем флаги прерывания и трассировки;
- Заносим в стек CS и IP;
- Выбираем новые CS и IP из таблицы векторов прерываний.

Если мы не сбросим флаг прерывания, то мы сразу же после начала обработки прерывания можем получить другое прерывание, в то время, как нам нужно гарантированно выполнить начало обработчика прерывания и сориентироваться в ситуации. Если мы не сбросим флаг трассировки, то после каждой выполненной команды процессор будет генерировать прерывание с кодом 03.

Так как мы сбрасываем эти флаги, то нам необходимо прежде всего сохранить регистр флагов, иначе мы не сможем восстановить полное состояние программы в момент прерывания.

Прерывания можно вызвать вручную из вашей программы, для этого есть специальная команда int. Это используется, например, для системных вызовов (см. лабораторную работу №1).

В конце обработчика прерываний используется команда iret, которая восстанавливает из вершины стека IP, CS и FLAGS.

Вопрос 2. *В чём отличие между ret и iret?*

1.3 Цикл компиляции

Компиляция программ происходит в несколько этапов. Мы рассмотрим типичный процесс компиляции программы на языке высокого уровня, таком, как C/C++.

1.3.1 Препроцессор

Изначально мы пишем текст программы, набор символов. Первая стадия компиляции — препроцессинг исходного кода программы. Он сводится к исполнению специальных команд для программы-препроцессора для обработки исходного текста программы. В результате получается какой-то другой исходный текст программы на том же языке программирования.

Типичный пример такой инструкции для препроцессора ассемблера — EQU. Написав в исходном тексте программы:

```
CAT_COUNT TEXTEQU <4>
...
mov ax, CAT_COUNT
```

мы фактически говорим препроцессору: «пройди по тексту программы, замени строчки "CAT_COUNT"на строчку "4»>. После того, как исходный текст пройдёт через препроцессор, он будет выглядеть так:

```
...
mov ax, 4
```

Подобным образом мы можем задавать имена для определённых констант. Зачем это нужно? Пусть нам нужно изменить количество котят в нашей программе. Если мы повсюду писали «4» вместо того, чтобы использовать именованную константу, то нам придётся заменять вручную четвёрки во всей программе на новые значения. Более того, нам каждый раз будет требоваться решать, соответствует ли данное число «4» количеству котят, или чему-то другому.

Замечание. *Хорошим тоном считается именовать все важные константы в вашей программе.*

Препроцессор MASM очень мощен. В нём можно объявлять свои функции, макросы могут повторяться много раз, развёртываться только частично в зависимости от определённых условий и многое другое. В книге [1, стр.121] можно найти подробное описание большинства макростредств MASM и других популярных ассемблеров.

1.3.2 Компилятор

Обычно компилятор принимает исходный файл с кодом и переводит его на язык ассемблера в том или ином виде. Поэтому на этой стадии можно легко получить файл с ассемблерным кодом, в который преобразовался исходный код программы.

Компилятор осуществляет сложное преобразование программы в одну сторону с потерей информации, точное обратное преобразование из ассемблерного кода на язык высокого уровня невозможно.

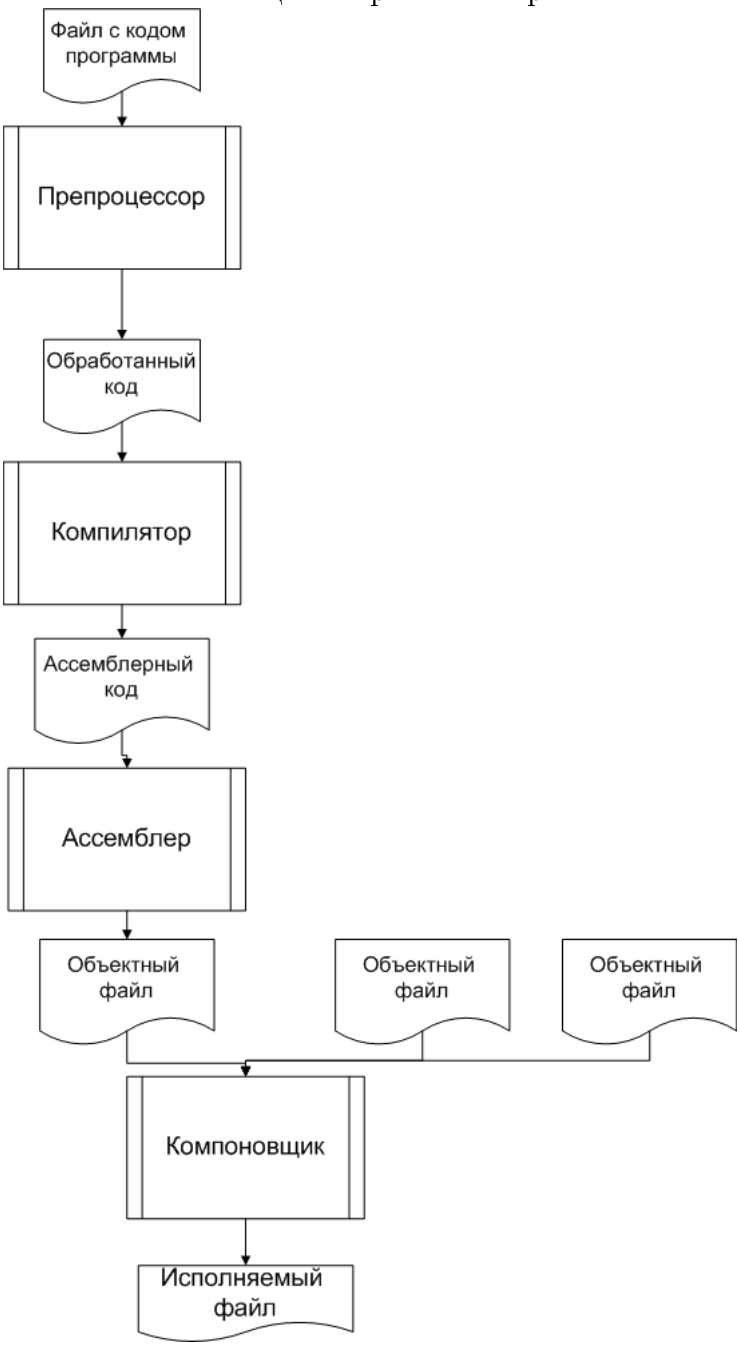
В процессе своей работы, компилятор проходит несколько стадий, таких, как лексический и синтаксический анализ программы, мы познакомимся с ними ближе позднее.

1.3.3 Ассемблер

На этом шаге происходит преобразование ассемблерного кода в объектный файл, содержащий машинные команды. К сожалению, есть некоторая терминологическая путаница: ассемблером называется как язык, так и сама программа, которая переводит его в машинные команды. Объектный файл нельзя исполнять. Причина в том, что файл может, например, вызывать код из другого файла. Чтобы это осуществить, необходима инструкция call, которой нужен, в той или иной форме, адрес начала вызываемой процедуры, который мы на данный момент совершенно не знаем.

Замечание. *Конечно, для нас процедура будет задаваться адресом её первой команды. Концом процедуры будем условно считать момент, когда мы исполняем инструкцию возврата из процедуры — например, ret.*

Для того, чтобы сделать возможным связывание кода в разных файлах, придумали механизм таблиц экспорта и импорта.



1.3.4 Компоновщик

Компоновщик собирает исполняемый файл, анализируя таблицы экспорта и импорта и заполняя их корректными адресами.

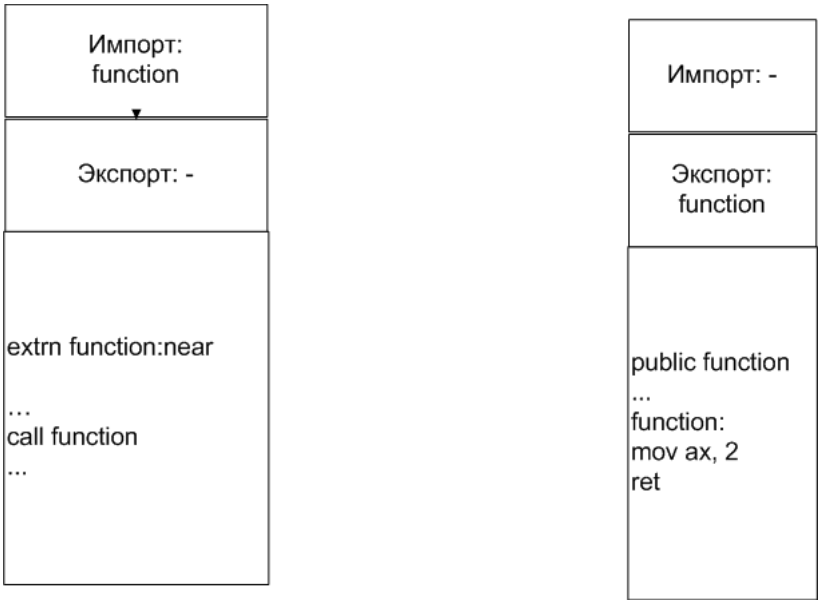
Чтобы понять, что такое таблицы экспорта и импорта, рассмотрим пример. Пусть у нас есть два файла с ассемблерным кодом:

main.asm

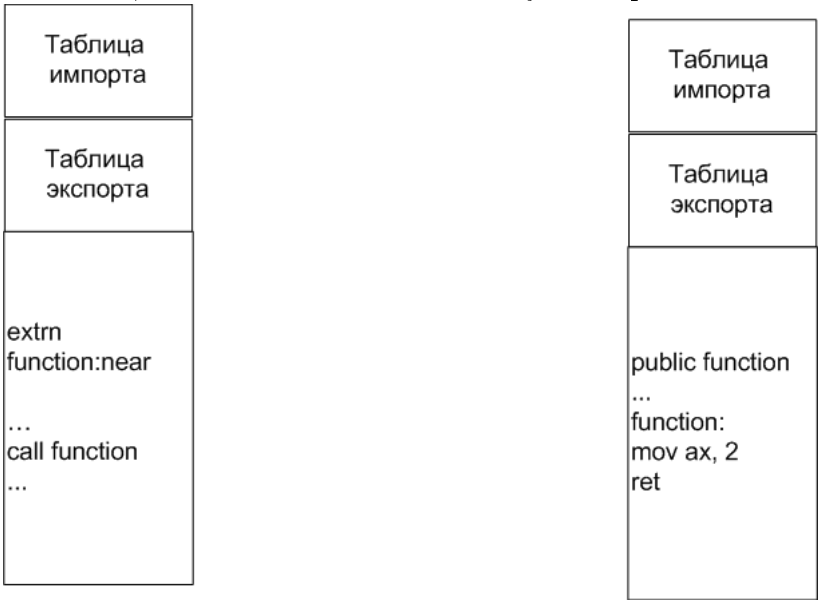
```
...
call function
...
```

func.asm

```
...
function:
mov ax, 10d
ret
...
```



В каждом объектном файле после компиляции появляются таблицы экспорта и импорта. Если мы хотим использовать в файле код из другого файла, то название соответствующей процедуры мы помещаем в таблицу импорта. Если мы хотим выставить на обозрение другим модулям процедуру из данного файла, чтобы они могли её вызывать, мы добавляем её в таблицу экспорта.



Чтобы заставить ассемблер добавить в таблицу экспорта запись о процедуре, которая начинается по определённой метке, нужно описать её где-нибудь в начале файла с пометкой public. В нашем случае это будет выглядеть так:

func.asm

```
public function

...
function:
mov ax, 10d
ret
...
```

Таким образом, public указывает компилятору добавить запись в таблицу экспорта объектного файла. В таблицу импорта запись можно добавить следующим образом:

main.asm

```
extrn function
...
call function
...
```

Компилятор вместо вызова процедуры просто по адресу вставит вызов процедуры с косвенной адресацией. В таблице импорта выделяется ячейка под адрес вызываемой процедуры из другого файла. Инструкция call с косвенной адресацией заберёт адрес из этой ячейки и перейдёт по нему. Задача компоновщика — проставить в неё корректный адрес, взятый из таблицы экспорта соответствующего модуля.

В таблицах импорта и экспорта в том или ином виде фигурируют именно имена процедур, какими их назвал программист, и их относительные адреса от начала их модулей. Формат этих имён зависит от компилятора и языка программирования.

1.4 Конечные автоматы

Конечный автомат — некая абстрактная машина, которая выполняет операции над входным потоком символов в соответствии с заданными правилами.

Конечный автомат полностью задаётся следующими параметрами:

1. Множество состояний, в которых может находиться автомат. В начале работы автомат находится в одном из них, которое помечено, как «начальное», а завершает работу тогда, когда попадает в конечное состояние.
2. Начальным состоянием;
3. Конечным состоянием или состояниями;
4. Набором символов, которые могут встречаться на входе;
5. Правилами перехода между состояниями.

Алгоритм работы автомата следующий:

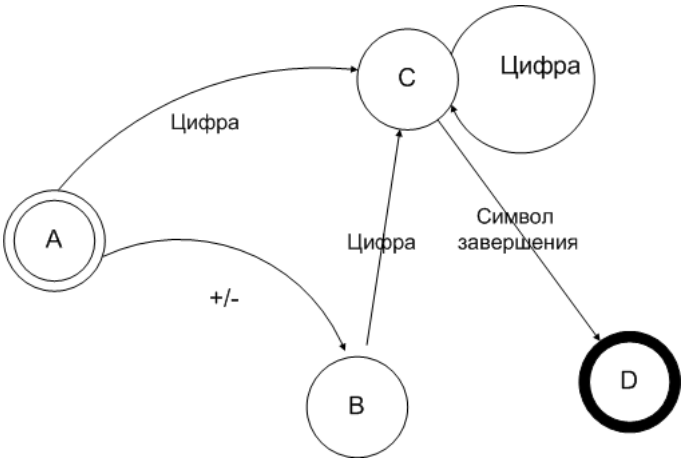
1. Старт в начальном состоянии;
2. Проверяем символ на входе и, в зависимости от правил перехода, переходим в одно из состояний (не обязательно отличных от текущего).

3. Если мы попали в конечное состоянии, завершить работу;

4. Иначе повторяем начиная с пункта 2.

Если в какой-то момент на вход автомата попал такой символ, для которого из текущего состояния автомата не было предусмотрено правил перехода, можно считать, что с точки зрения автомата эта последовательность входных символов ошибочна.

Описывая автомат мы вместе с этим описываем некоторое множество входных строчек, которые, с точки зрения автомата, корректны. Чтобы продемонстрировать это, рассмотрим автомат, который принимает на вход все строчки, соответствующие целым числам.



В автомате 4 состояния:

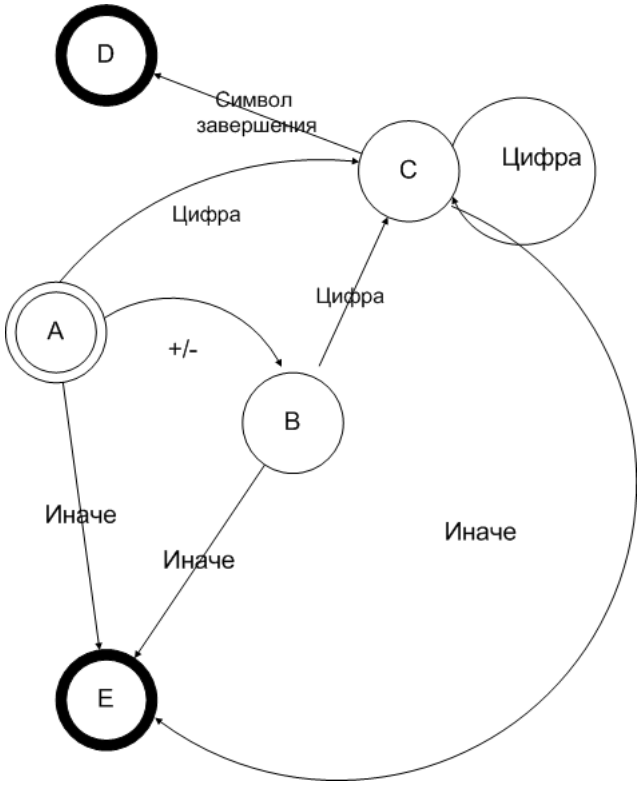
$\{A, B, C, D\}$

A — начальное состояние, D — конечное состояние.

На каждый момент времени автомат находится в одном из состояний. Каждый следующий символ, принятый им, побуждает его переходить по стрелкам между состояниями. Если он входит в конечное состояние D, то завершает свою работу.

В принципе, выходных состояний может быть много. Мы можем с помощью различных выходных состояний кодировать результат работы автомата. Например, можно предусмотреть ситуацию, когда введено некорректное число (например, «45hd333»), и в таком случае переходить в иное конечное состояние E.

Следует обратить внимание на то, что теперь все входные строчки для автомата являются корректными — он может обрабатывать любую.



Конечные автоматы очень легко реализовывать на ассемблере. Достаточно завести метки для каждого состояния и по каждой метке предусмотреть считывание следующего элемента из «входа» и переходы на другие состояния.

1.4.1 Пример реализации конечного автомата на ассемблере

Реализуем вышеуказанный автомат на ассемблере.

fsm.asm

```
.model small

.stack 128

.data
okmsg db 10,13,"I enjoyed parsing this one, man. Man, I'm good", 10, 13,"$"
errormsg db 10,13, "Invalid number here, but I ate some oranges and it was k", 10, 13,"$"

.code

newline_code EQU <13>
get_symbol MACRO
mov ah, 01h
int 21h
ENDM

main:
mov ax, @data
mov ds, ax

_A:
        get_symbol
        cmp al, '+'
```

```
        jmp _B
        cmp al, '9'
        ja _E
        cmp al, '0'
        jb _E
        jmp _C

_B:
        get_symbol
        cmp al, '9'
        ja _E
        cmp al, '0'
        jb _E
        jmp _C

_C:
        get_symbol
        cmp al, newline_code
        je _D
        cmp al, '9'
        ja _E
        cmp al, '0'
        jb _E
        jmp _C

_D:
        lea dx, okmsg
        mov ah, 09h
        int 21h
        mov ax, 4c00h
        int 21h
_E:

        lea dx, errormsg
        mov ah, 09h
        int 21h
        mov ax, 4c01h
        int 21h

end main
```

1.5 Команды работы со строками

Команда mov не умеет перемещать данные из памяти в память. Существуют специализированные команды для того, чтобы копировать массивы данных в памяти. Рассмотрим для начала команду movsb.

1. Копируем данные с адреса DS:SI в память по адресу ES:DI
2. Если флаг DF=0, то SI и DI увеличиваются на единицу, иначе уменьшаются на единицу.

Если мы после этого выполним команду movsb еще раз, то скопируем следующий байт по порядку. Существуют вариации этой команды: movsw и movsd (для

защищённого режима). Они перемещают соответственно 2 и 4 байта, и изменяют регистры (E)SI и (E)DI на 2 или 4.

Как этим пользоваться?

1. Сбрасываем флаг DF командой CLD;
2. Заносим в DS:(E)SI и ES:(E)DI корректные значения, формирующие адреса начала первой ячейки отправителя и первой ячейки получателя;
3. Выполняем MOVSB/MOVSW/MOVSD столько раз, сколько нам необходимо, чтобы скопировать весь массив.

Если нам необходимо копировать массив с последнего элемента до первого, а не в прямом порядке, то мы вначале устанавливаем флаг DF командой STD, а на шаге 2 заносим в соответствующие регистры адреса последних элементов в массиве-приёмнике и массиве-отправителе.

Вопрос 3. *Как работают команды CMPSB, CMPSW, LODSB, LODSW, STOSW, STOSB, SCASB, SCASW? Посмотрите документацию.*

Вопрос 4. *В чём разница между MOVS и MOVSB?*

1.6 Префиксы

Формат машинной команды подразумевает, что в начале команды может стоять один из префиксов, определённым образом изменяющих поведение команды. Каждый префикс имеет смысл не для всех команд, которые вообще существуют в процессоре.

1.6.1 Префиксы повторения операций

Префиксы повторения операций определены для строковых команд:

- rep — повторять, пока (E)CX $\neq 0$; декрементировать (E)CX;
- repe, repz — повторять, пока установлен ZF и (E)CX $\neq 0$; декрементировать (E)CX;
- repne, repnz— повторять, пока не установлен ZF и (E)CX $\neq 0$; декрементировать (E)CX;

Поведение префиксов определено в следующих случаях:

- когда rep используется с командами SCAS, INS, OUTS, MOVS, LODS, STOS;
- когда repe/repz/repne/repnz исползуются с командами CMPS и SCAS.

В иных случаях поведение префиксов не задано, а транслятор с ассемблера запрещает использовать префиксы отличным от вышеуказанных способом.

Пример использования:

```
cld
rep movsw
```

Это означает «скопировать количество слов, заданное регистром (E)CX, из памяти в память начиная с адреса DS:(E)SI по адресу ES:(E)DI».

1.6.2 Префикс блокировки шины

Префикс lock можно использовать с командами: ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, DEX, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG. Он означает, что во время выполнения помеченной им команды блокируется шина данных. Если в системе больше одного процессора, он не сможет обращаться к памяти, пока не завершится выполнение команды.

1.6.3 Префикс замены сегмента

С помощью него можно заменять используемый в команде по умолчанию сегмент на какой-либо иной. Например:

```
mov ax, es:[0002h]
```

Это возможно не всегда, следует проконсультироваться с документацией в случае возникновения проблем. Например, в командах типа movsb нельзя переопределить сегмент назначения (всегда ES), а сегмент источника — можно.

1.6.4 Префикс изменения размера операнда !

В случае, когда мы находимся, например, в защищённом режиме, и хотим положить в стек не двойное слово, а одинарное (не 4 байта, а 2), что является «неродной» операцией для данного режима, то перед командой автоматически ставится префикс изменения размера операнда. Он означает

1.7 Защищённый режим работы процессора !

1.7.1 Особенности защищённого режима работы процессора

1.7.2 Сегментация

В защищённом режиме существовала усовершенствованная сегментация с таблицами локальных и глобальных дескрипторов сегментов. Однако, так как операционные системы не поддерживали её, то мы имеем дело исключительно с «вырожденным» случаем сегментации: размер каждого сегмента — 4 Гбайт и они совпадают между собой, начинаясь с нулевого адреса. В 64-разрядных процессорах существовавший сложный механизм сегментации был упразднён, так как распространённые операционные системы всё равно его не использовали.

1.8 Адресные пространства

Адресное пространство, интуитивно — это некоторая последовательность адресов, от минимального до максимального. Мы сталкиваемся с тремя типами адресных пространств: логическим, физическим и адресным пространством процесса.

1.8.1 Логическое, физическое адресное пространство

То, сколько реальной памяти имеет компьютер, определяет его физическое адресное пространство. Разрядность шины адреса определяет его логическое адресное пространство (сколько всего ячеек памяти он может адресовать). Бывает, что логическое адресное пространство больше физического (например, в 32-разрядной машине стоит 512 Мбайт оперативной памяти, а потенциально она способна работать почти с четырьмя гигабайтами). Возможна и обратная ситуация: если мы поставим в неё 8 гигабайт памяти, то она сможет работать только с четырьмя из них.

Если логическое адресное пространство больше, то некоторые его адреса становятся запрещёнными. При попытке обращения к ним процессор генерирует ошибку, ведь им не соответствует реальных ячеек физической памяти.

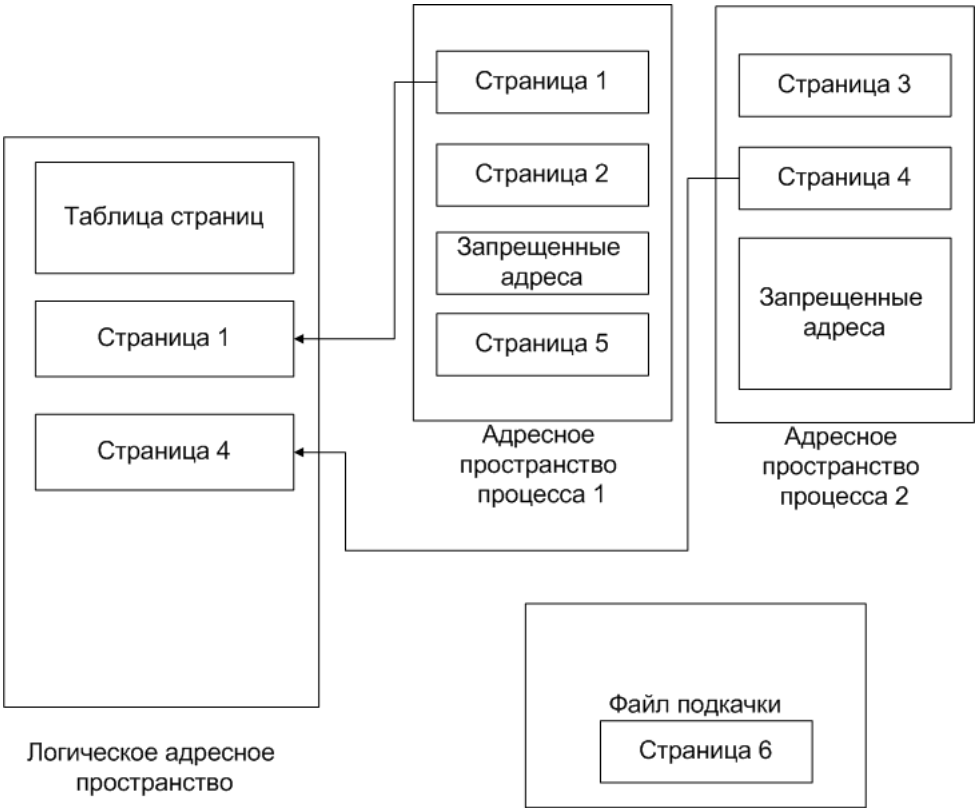
1.8.2 Адресное пространство процесса и виртуальная память

Адресное пространство процесса – это то, что доступно программисту. Когда он пишет программу для реального режима работы, например, под DOS, он может легко добраться до кода операционной системы, ничто не запрещает ему даже переписать её части прямо в памяти. Когда же он пишет прикладное программное обеспечение в современном окружении, у него есть иллюзия того, что вся память принадлежит именно ему. В какую бы ячейку памяти он не обратился, он никогда не увидит другие программы, которые, на деле, параллельно или псевдопараллельно выполняются на процессоре вместе с его собственной и делят с ней память.

Как это происходит? С помощью виртуальной памяти.

Адресное пространство процесса делится на страницы фиксированного размера. В каждый момент времени только некоторые страницы находятся в физической памяти компьютера. Благодаря программно-аппаратному механизму виртуальной памяти (за который ответственны операционная система и аппаратура) необходимые страницы отображаются из адресного пространства процесса на логическое адресное пространство.

Вопрос 5. *Можно ли говорить, что страницы отображаются в физическое адресное пространство?*



Если свободной памяти не осталось, то в соответствии с одним из правил (называемых стратегии вытеснения) некоторые страницы вытесняются из логического адресного пространства процессора в файл подкачки, который хранится во внешней памяти. В *nix системах для этого часто выделяют отдельный раздел на диске, в Windows этот файл называется PageFile.sys.

Что происходит, когда мы обращаемся к виртуальным адресам, которые находятся в страницах, не находящихся в физической памяти? Тут возможны два исхода:

- С точки зрения операционной системы, процессу не были выделены адреса, к которым он обращается. Они «запрещены» для него, в этом случае генерируется ошибка доступа. Когда мы будем программировать на C, мы вплотную с этим столкнёмся.

  Замечание. *Процесс может обратиться к операционной системе с просьбой выделить ему память. На самом деле он, таким образом, получает не физическую память, а дополнительные адреса в своё распоряжение. Динамическое выделение памяти, в конечном счёте, происходит именно так.*

- Процесс обращается к адресам, которые ему принадлежат. Тогда генерируется ошибка «Page Fault», означающая отсутствие нужной страницы в физической памяти, а операционная система осуществляет необходимую работу по управлению страницами, чтобы страница, в конце концов, оказалась в физической памяти.

Почему такой сложный и громоздкий механизм оказался не только эффективным по затратам памяти, но и достаточно быстрым?

1. Работа по трансляцию адресов осуществляется, преимущественно, аппаратурой;

2. Благодаря уже упоминающемуся свойству локальности, механизм подгрузки недостающих страниц запускается достаточно редко, в сравнении с доступом к уже загруженным. Как и в случае с регистрами, в худшем случае у нас падение производительности, но в среднем это работает быстро.

1.9 Режимы адресации

1.9.1 Косвенная адресация

В реальном режиме работы для косвенной адресации можно использовать только bx, si, di и bp. В защищённом — любые 32-разрядные регистры.

```
mov ax, [bx]
```

1.9.2 Базовая адресация

При базовой адресации со смещением мы используем bx, si, di или bp для задания базы, к которой прибавляем фиксированное число — смещение. Они складываются, и получается адрес. В защищённом режиме можно использовать любые регистры для базы.

```
mov ax, [bx+4]
```

1.9.3 Базово-индексная адресация со смещением и масштабированием

Этот тип адресации включает в себя все другие как частные случаи. Адрес вычисляется как сумма нескольких компонент:

```
mov ebx, [eax+ ecx*4 +42]
```

Полная формула выглядит так:

$$[base + index * size + offset]$$

, где:

$base = \{eax, ebx, ecx, edx, ebp, esp, edi, esi\}$;

$index = \{eax, ebx, ecx, edx, ebp, esi, edi\}$

$size = \{1, 2, 4, 8\}$

$offset$ – произвольное число.

С помощью команды lea, используя этот способ адресации, часто можно выполнять быстрые арифметические операции, например, умножение регистра на 5 или 9 в одну команду.

Вопрос 6. *Объясните, как.*

1.10 Лабораторная работа №1

В данной лабораторной работе вы найдёте примеры простейших программ на MASM, а затем на основе имеющегося материала напишете свою программу. В теоретической части вы встретите вопросы, ответы на которые вы должны найти сами. К защите лабораторной работы вы должны быть готовы ответить на любой из них (разумеется, не ограничиваясь только ими).

Для выполнения работы вам понадобятся как минимум следующие команды:

- mov
- lea
- jmp
- jg/jb/ja/jl
- cmp
- test
- push
- pop
- add
- sub
- call
- ret

Документацию по ним следует посмотреть в Intel 80386 reference manual.

1.10.1 Hello, world!

По традиции начнём изучение языка ассемблера с программы "Hello, world!".

```
.model small
.stack 128

.data
mess db 'Hello world!$'

.code

main:

mov ax,@data
mov ds,ax

lea dx,mess
mov ah,09h
int 21h
```

```
mov ax,4c00h
int 21h

end main
```

1.10.2 Комментарий

```
.model small
```

Выбираем модель памяти small, что означает один сегмент кода и один сегмент данных.

Вопрос 7. *Что такое модель памяти?*

Вопрос 8. *Какие есть еще модели памяти?*

```
.stack 128
```

Выделяем 128 байт под стек. Это значит, что нам гарантировано, что если мы положим в стек не более 128 байт, то программа будет корректно работать.

Вопрос 9. *Что произойдёт, если размер стека не указать явно?*

Вопрос 10. *Может ли программа работать без стека?*

Вопрос 11. *Корректна ли команда и если нет, то почему?*

```
push al
```

```
.data
mess db 'Hello world!$'
```

Объявляем начало сегмента данных и метку mess. Метка в ассемблере – имя для определённого адреса, так что в дальнейшем каждый раз при использовании имени mess вместо него будет подставляться смещение относительно начала сегмента данных.

Директива db обозначает байтовые данные. Можно задавать данные:

- db – байтами;
- dw – словами, или двойными байтами;
- dd – двойными словами, или по четыре байта;
- df – по 6 байт;
- dq – учетверёнными словами, или по 8 байт;
- dt – по 10 байт.

Несколько примеров:

```
.data
example1 db 10
example2 dw 10 dup (?)
example3 dw 4 dup(42)
```

В первой строчке мы задаем байт, хранящий число 10. Затем мы задаем 10 двойных слов подряд, значение каждого из которых в момент запуска может быть любым, т.е. не определено. example3 задает 4 слова, каждое из которых равно 42.

Замечание. *В терминологии Intel есть некоторая путаница, которая связана с тем, что вообще машинным словом называется «родной» для машины формат данных. С появлением защищённого режима и расширением регистров до 32 бит размер машинного слова стал равен 32 битам, а не 16, но чтобы сохранить преемственность 32битные данные продолжили называть удвоенными словами.*

```
.code

main:
```

Начало сегмента кода и метка main. В ассемблере можно произвольно смешивать описания кода и данных, каждый раз указывая начало соответствующего сегмента. Метка main это такой же адрес, впоследствии она будет нам важна – см. комментарий к последней строчке.

```
mov ax,@data
mov ds,ax
```

Нам необходимо настроить регистр DS, потому что иначе все смещения в сегменте данных будут отсчитываться от неправильного начала сегмента. В начале работы программы в DS лежит мусор. Вместо @data компилятор подставит корректный адрес начала сегмента данных, т.е. число. Например, команда:

```
lea dx,mess
```

использует адрес DS:mess (в формате адреса сегмент:смещение), поэтому для неё критично, чтобы значение DS было корректным. Напрямую переслать значение в DS нельзя, это не предусмотренно процессором, поэтому мы пересылаем его через регистр общего назначения.

Следующая команда загружает в DX смещение mess от начала сегмента данных.

```
lea dx,mess
```

Команда lea позволяет вычислить эффективный адрес ячейки и загрузить его куда-либо. Адресация может быть достаточно сложной, например, базово-индексной, и благодаря lea мы сможем запомнить значение адреса.

Вопрос 12. *В чем разница между командами:*

```
lea dx,mess
mov dx, mess
```

? Что будет в регистре DX после выполнения второй команды?

Вопрос 13. *Что такое Little Endian и Big Endian?*

```
mov ah,09h
int 21h
```

С помощью этих двух команд мы перемещаем в регистр AH значение 09_{16} и вызываем вручную прерывание с номером 21h.

Замечание. *С помощью суффиксов в ассемблере мы можем задавать основание системы счисления для чисел.*

- *10h* = 16_{10};
- *10d* = 10_{10} – *если не указать суффикс, то будет тоже использоваться десятиричная система счисления;*
- *10b* = 2_{10};
- *10o* = 8_{10}

Прерывание с номером 21h закреплено за DOS, а не за внешними устройствами. При его вызове происходит передача управления операционной системе, которая может выполнить одно из четко определённых действий. Каждое из этих действий имеет свой номер. Выбирается то действие, номер которого находится в регистре AH. В данном случае требуемая операция – вывод строки по адресу DS:DX до тех пор, пока не встретится символ доллара.

Замечание. *Вниманию тех, кто помнит про нуль-терминированные строки в языках высокого уровня. Доллар не заменяется на символ с кодом 0, доллар это просто доллар , символ экономической мощи Америки .*

Аналогичная операция по вызову прерывания DOS'а с требуемыми параметрами происходит в следующих двух строчках.

Директива end означает конец ассемблирования файла. Всё, что идёт после неё, игнорируется. Кроме того, её параметр указывает точку входа в программу – адрес, с которого она должна начать своё выполнение.

```
end main
```

Вопрос 14. *Что такое директива и чем она отличается от ассемблерной команды?*

Вопрос 15. *Объясните, что такое системный вызов и чем он отличается от обычной процедуры?*

Как вы видите, прерывания можно вызвать вручную с помощью команды int. Здесь они используются чтобы обратиться к системному вызову операционной системы DOS. Механизм системных вызовов DOS прост:

- В регистр AH заносится номер системного вызова;
- В другие регистры, возможно, заносятся параметры (зависит от системного вызова);

- Выполняется инструкция int 21h, вызывающая прерывание номер 21_{16}, закреплённое за операционной системой DOS;
- Готово, некоторые системные вызовы возвращают в регистрах различные значения.

Вопрос 16. *Объясните, что такое системный вызов и чем он отличается от обычной процедуры?*

Вопрос 17. *В чем отличие инструкций call и int? Что такое call near и call far?*

Мы столкнулись уже с двумя системными вызовами:

- AH = 09h – вывод строки. Параметры: DS:DX = адрес начала строки. Концом строки считается символ доллара, то, что следует за ним, не выводится.

  Вопрос 18. *Попробуйте убрать символ доллара из строки, скомпилировать и запустить программу. Объясните результат.*

  Вопрос 19. *Как можно вывести на экран символ доллара, используя это прерывание с AH=09h? А используя другие прерывания или другие значения AH?*

- AH = 4ch – завершить выполнение программы и передать управление DOS.

  Параметры: AL = код возврата.

Вопрос 20. *Что такое код возврата?*

Рассмотрим также вызовы для вывода и считывания символов из консоли.

- AH = 01h – считывание символа из стандартного ввода, дублируя его в стандартный вывод.

  Возвращает: AL = считанный символ.

- AH = 02h – вывести символ на стандартный вывод.

  Параметры: DL = символ для вывода.

  Возвращает: AL = Последний выведенный символ.

Например, такая программа при запуске дублирует каждый введённый символ.

```
.model small
.code
start:
mov ah, 01h
int 21h
mov dl, al
mov ah, 02h
int 21h
jmp start
end start
```

Вопрос 21. *В каком случае после такого системного вызова AL не будет равен DL? (Подсказка: ищите полное описание системного вызова.)*

1.10.3 Задание

Ваша задача – написать калькулятор, работающий со стеком. Он должен реагировать на нажатия клавиш пользователем следующим образом:

1. Ввод числа от 0 до 9 — число (не код введённой цифры!) кладётся на вершину стека, счётчик элементов в стеке увеличивается;

2. Ввод символа +/-

   - если в стеке есть как минимум два числа, то последние два числа вынимаются оттуда, и результат их сложения/вычитания помещается в стек. Затем счётчик элементов в стеке изменяется так, чтобы отражать их реальное число. Затем вершина стека выводится на экран;
   - если в стеке слишком мало чисел, то программа выдаёт на экран строчку "error"и игнорирует этот символ.

3. Ввод символа 'p' — вывод на экран числа, находящегося в вершине стека, или ошибка, если стек пуст. Для этого вы можете пользоваться процедурой вывода содержимого регистра AX на экран, которая находится в приложении.

4. Ввод символа 'q' — выход из программы. Желательно при этом очистить стек.

Несколько советов:

- Обратите внимание на то, что для того, чтобы очистить ваш стек, совершенно не обязательно вынимать оттуда каждое лежащее там число. Такие решения неразумны и неэффективны, и не будут приниматься!

- В ассемблере можно писать символы в кавычках подразумевая их ASCII-коды. Например, вместо 's' подставится ASCII-код символа 's', а вместо '9' – 39h, т.е. код символа девятки.

- Для организации условных переходов используются команды cmp и test, которые совершают вычитание и логическое И операндов соответственно, выставляя соответствующие результату флаги. Сам результат, однако, они не сохраняют. Затем, имея флаги, отражающие результат, используется соответствующая команда условного перехода, например, ja.

  С помощью test можно быстро проверять содержимое регистра на ноль.

Вопрос 22. *Чтобы программу можно было начать выполнять, нужно сначала загрузить её в основную память. Это работа загрузчика – части операционной системы. Но, помимо того, что программа должна оказаться в памяти, некоторые регистры должны быть установлены загрузчиком в определённые значения прежде, чем программа начнёт исполняться. О каких регистрах идёт речь?*

Состав работы

- Отчёт
- Листинг

- Таблица трассировки
- Блок-схема алгоритма.

Для таблицы трассировки важны только те регистры, которые изменяются в процессе работы программы. Такие регистры, как ES, можно не включать в неё. На каком примере входных данных делать таблицу трассировки — решать вам. Он не должен быть большим, например, положите в стек несколько чисел, сложите и вычтите их, а затем выйдите из программы. Трассировать процедуру AX_OUT необязательно, если вы всё-таки вошли в неё в процессе отладки можете пропустить все те команды, которые находятся внутри неё.

Вопрос 23. *Как только с помощью команды xor обменять две ячейки памяти местами?*

Вопрос 24. *Как с помощью двух команд взять модуль целого числа? (подсказка: одна из них – NEG).*

Вопрос 25. *Как узнать адрес текущей команды с помощью команды call с относительной адресацией? Можно уложиться в две команды.*

1.10.4 Приложение

Процедура для вывода регистра AX в стандартный вывод. Скопировать как есть в начало сегмента кода. Пример использования — call ax_out.

```
;-------------------------------------
ax_out  proc  uses dx cx
; Outputs AX
;-------------------------------------

.code
push ax

mov ah,02h
mov cx, 12d

mainloop:

        pop dx
        push dx
        shr dx,cl
        and dl, 0Fh
        cmp dl, 10d
        jge add_37h
        add dl, 30h

        jmp print_4

        add_37h:
                add dl, 37h

        print_4:

                int 21h
        sub cx,3

loop mainloop

pop dx
push dx

and dl, 0fh
cmp dl, 10d
jge add_37h_last
add dl, 30h

jmp print_last

add_37h_last:
        add dl, 37h

print_last:
                int 21h

pop ax
ret
ax_out endp
```

Глава 2

Основы языка C !

2.1 Чем особен C? !

2.2 Типы данных !

2.2.1 Что такое типы данных? !

2.2.2 Типизация в языках программирования !

Статическая и динамическая !

Строгая и нестрогая !

2.2.3 Полиморфизм в широком смысле !

2.2.4 Виды полиморфизма

2.2.5 Типы данных в C

2.2.6 Массивы в C

2.3 Структура программ в C !

2.3.1 Процедуры и функции !

2.3.2 Побочные эффекты выражений !

2.3.3 Связанность и связность !

2.3.4 Структуры, перечисления, объединения !

2.4 Модель памяти языка C !

2.4.1 Выделение памяти !

Автоматическое выделение памяти

Статическое выделение памяти

Динамическое выделение памяти

2.5 Модель вычислений !

2.5.1 Абстрактный вычислитель и модель вычислений !

2.5.2 Модель вычислений языка C !

2.6 Стили программирования !

2.6.1 Стили программирования, поддерживаемые C !

2.7 Указатели !

2.7.1 Принцип работы !

2.7.2 Адресная арифметика !

Операции над указателями !

2.7.3 Указатели на функции !

2.8 Препроцессор C !

2.8.1 Include Guard !

2.9 Синтаксис, семантика и прагматика языков !

2.9.1 Грамматики !

2.9.2 Прагматика C и выравнивание !

2.10 Потоки данных в C!

2.11 Лабораторная работа №2

Необходимо написать программу на языке C (не C++!) с ассемблерными вставками. Программа должна принимать в качестве аргумента имя файла-изображения в формате BMP любого разрешения и переворачивать картинку.

Поворот картинки на 180 градусов необходимо реализовать тремя способами:

1. На C;
2. На ассемблере с использованием команд работы со строками;
3. На ассемблере с использованием MMX-команд.

В процессе работы необходимо совершить каждое преобразование достаточно большое одинаковое нечётное количество раз и замерить его суммарное время. Время выводится в поток вывода stderr.

2.12 Формат BMP-файла

Формат файла BMP (сокращенно от BitMaP) - это формат растровой графики близкий Windows, поскольку он лучше всего соответствует внутреннему формату Windows, в котором эта система хранит свои растровые массивы. В файлах BMP информация о цвете каждого пикселя кодируется 1, 4, 8, 16 или 24 бит (бит/пиксель). Числом бит/пиксель, называемым также глубиной представления цвета, определяется максимальное число цветов в изображении. Изображение при глубине 1 бит/-пиксель может иметь всего два цвета, а при глубине 24 бит/пиксель - более 16 млн. различных цветов.

Заголовок файла растровой графики (14 байт)	Сигнатура файла BMP (2 байт) – Символы “BM” Размер файла (4 байт) Не используется (2 байт) Не используется (2 байт) Местонахождение данных растрового массива (4 байт)
Инфор-мационный заголовок растрового массива (40 байт)	Длина этого заголовка (4 байт) Ширина изображения (4 байт) Высота изображения (4 байт) Число цветовых плоскостей (2 байт) Бит/пиксел (2 байт) Метод сжатия (4 байт) Длина растрового массива (4 байт) Горизонтальное разрешение (4 байт) Вертикальное разрешение (4 байт) Число цветов изображения (4 байт) Число основных цветов (4 байт)
Данные растрового массива	RGBRGBRGB… R=1 байт, G=1 байт, B=1 байт

На приведенной схеме показана структура BMP-файла без палитры. Файл разбит на 3 основных раздела: заголовок файла растровой графики, информационный заголовок растрового массива и данные растрового массива. Заголовок файла растровой графики содержит информацию о файле, в том числе адрес, с которого начинается область данных растрового массива. В информационном заголовке растрового массива содержатся сведения об изображении, хранящемся в файле, например, его высоте и ширине в пикселях. Формат данных растрового массива в файле BMP зависит от числа бит, используемых для кодирования данных о цвете каждого пикселя. Файлы с глубиной 16 и 24 бит/пиксель не имеют таблиц цветов; в этих файлах значения пикселей растрового массива непосредственно характеризуют значения цветов RGB.

Примерный алгоритм чтения BMP файла:

- Открыть файл для чтения;
- Прочитать первые 54 байта заголовка в структуру, соответствующую его формату;
- Проанализировать первые два байта заголовка, они должны быть равны “BM”;
- Прочитать и проанализировать глубину цвета, она должна быть равна 24;
- Прочитать размер изображения: ширину и высоту;
- Вычислить размер изображения (всего 3 слоя R, G и B) = 3*ширину*высоту;
- Прочитать изображение в буфер целиком;
- Закрыть файл.

Содержание отчёта:

1. Титульный лист;
2. Листинг программы с комментариями;
3. Описание использованных команд MMX;
4. Результаты замеров скорости.

Некоторые дополнительные требования:

1. Нельзя читать заголовок BMP файла в массив. Только структура!
2. Пользуемся безопасными функциями для работы с файлами, например, fopen_s, а не fopen;
3. Компилируем программу как C код, а не как C++ код.

2.12.1 Инструментарий

Вы можете пользоваться компилятором и средой разработки на ваше усмотрение. Например, Visual Studio/ Visual C++ Express.

Компилятор, идущий в составе Visual C++ Express умеет компилировать как C++ код, так и C код. Необходимы некоторые пояснения по работе со средой разработки:

Конфигурация — набор параметров, используемых для компиляции.

Если в первой лабораторной работе вы задавали его вручную каждый раз, когда компилировали программу (например, ключ /Zi), то теперь вы можете автоматизировать данный процесс. Обычно используют два набора параметров для компиляции, две конфигурации: Debug и Release.

В Visual Studio используются следующие термины для ваших «проектов»:

Решение (Solution) соответствует конечной программе

Проект (Project) соответствует одному скомпилированному файлу (исполняемому файлу или библиотеке).

В конфигурации Debug:

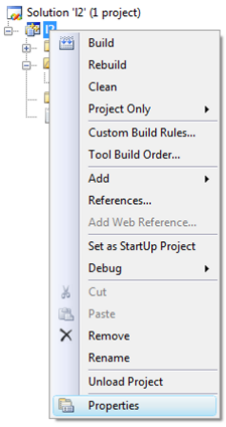
- Отключены оптимизации, поэтому код на C максимально соответствует сгенерированному машинному коду;
- Добавляется диагностическая информация, облегчающая отладку;
- Исполняемый код получается медленнее и большего размера.

В конфигурации Release:

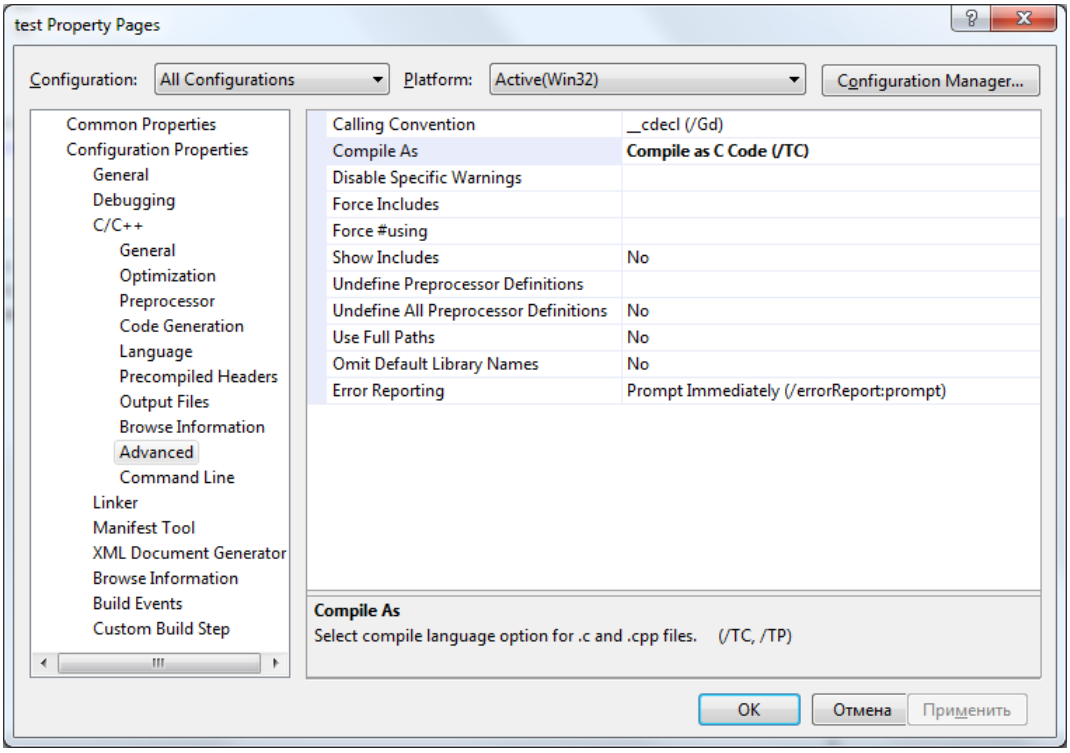
- Включают оптимизации, поэтому код на C уже не очень соответствует сгенерированному машинному коду;
- Отладка затруднена;
- Исполняемый файл получается меньше и выполняется быстрее.

Разумеется, замеры производительности надо производить в конфигурации Release:)

Вам также обязательно надо переключить компилятор на режим компилирования кода, как C кода.



В проекте должен быть уже добавлен хотя бы один файл с расширением .c . Зайдя в свойства проекта, выберите сверху All configurations, далее C/C++ -> Advanced ->Compile as: C code.


test Property Pages
Configuration: All Configurations
Platform: Active(Win32)
Configuration Manager...
Common Properties
Configuration Properties
General
Debugging
C/C++
General
Optimization
Preprocessor
Code Generation
Language
Precompiled Headers
Output Files
Browse Information
Advanced
Command Line
Linker
Manifest Tool
XML Document Generator
Browse Information
Build Events
Custom Build Step
Calling Convention __cdecl (/Gd)
Compile As Compile as C Code (/TC)
Disable Specific Warnings
Force Includes
Force #using
Show Includes No
Undefine Preprocessor Definitions
Undefine All Preprocessor Definitions No
Use Full Paths No
Omit Default Library Names No
Error Reporting Prompt Immediately (/errorReport:prompt)
Compile As
Select compile language option for .c and .cpp files. (/TC, /TP)
OK
Отмена
Применить

Глава 3

Компиляция и запуск программы!

3.1 Работа компилятора!

3.1.1 Лексический анализ!

3.1.2 Синтаксический анализ!

3.1.3 Оптимизации!

3.2 Статический и динамический контекст программы !

3.2.1 Стековые фреймы !

3.3 Точки следования !

3.4 Соглашения вызова !

3.5 Статическое и динамическое связывание !

3.6 Структура исполняемого файла !

3.7 Оптимизация компилятором!

Глава 4

Безопасность программ!

4.1 Атаки типа Buffer overflow!

4.2

Литература

[1] Зубков С.В., *Assembler для DOS, Windows и Unix*. СПб:Питер, 3-е изд. стер., 2004.

[2] Керниган Б.В., Ричи Д.М., *Язык С*.

[3] Непейвода Н.Н., Скопин И.Н. *Основания программирования*. Москва-Ижевск: Институт компьютерных исследований, 2003, 864 стр.

[4] Cardelli L., Wegner P. *On understanding Types, Data Abstracion and Polymorphism*. Computing Surveys, Vol. 17, No.4, December 1985

[5] Ахо А., Сети Р., Ульман Д. *Компиляторы: Принципы, технологии, инструменты*. Москва: Вильямс, 2003