



Урок 1

Операционные системы

Для чего нужны операционные системы? Связь между программным и аппаратным обеспечением. Процессор, память, прерывания. Представление о машинных кодах и низкоуровневых языках: Assembler, C. Основные типы операционных систем и их дальнейшее развитие.

Введение

[Программы. Машина Тьюринга](#)

[Первые компьютеры на электромеханических реле](#)

[4 века компьютеров. Лампы, транзисторы, БИС, СБИС](#)

Процессор

[История Intel-процессоров](#)

[Разрядность процессора](#)

[Регистры процессора](#)

[Порядок байтов](#)

Прерывания

[Исключения](#)

[Аппаратные прерывания](#)

[Немаскируемое прерывание](#)

[Прерывание для отладки – breakpoint](#)

[Программные прерывания](#)

[Системные вызовы](#)

[Языки программирования низкого уровня](#)

[Машинный код](#)

[Мнемоники, опкоды, дизассемблер](#)

[Форматы исполняемых файлов](#)

[Си](#)

[Forth](#)

[Ассемблер](#)

[Операционные системы: функции, задачи](#)

[Системные вызовы](#)

[Классификация операционных систем](#)

[История операционных систем](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

Введение

Операционные системы – это настолько распространённое явление, что, казалось бы, вопрос о том, для чего они нужны, не должен возникать. Современные операционные системы реализуют дополнительный уровень абстракции в отношениях между человеком и аппаратом, упрощая не только разработку программного обеспечения, но и работу в целом.

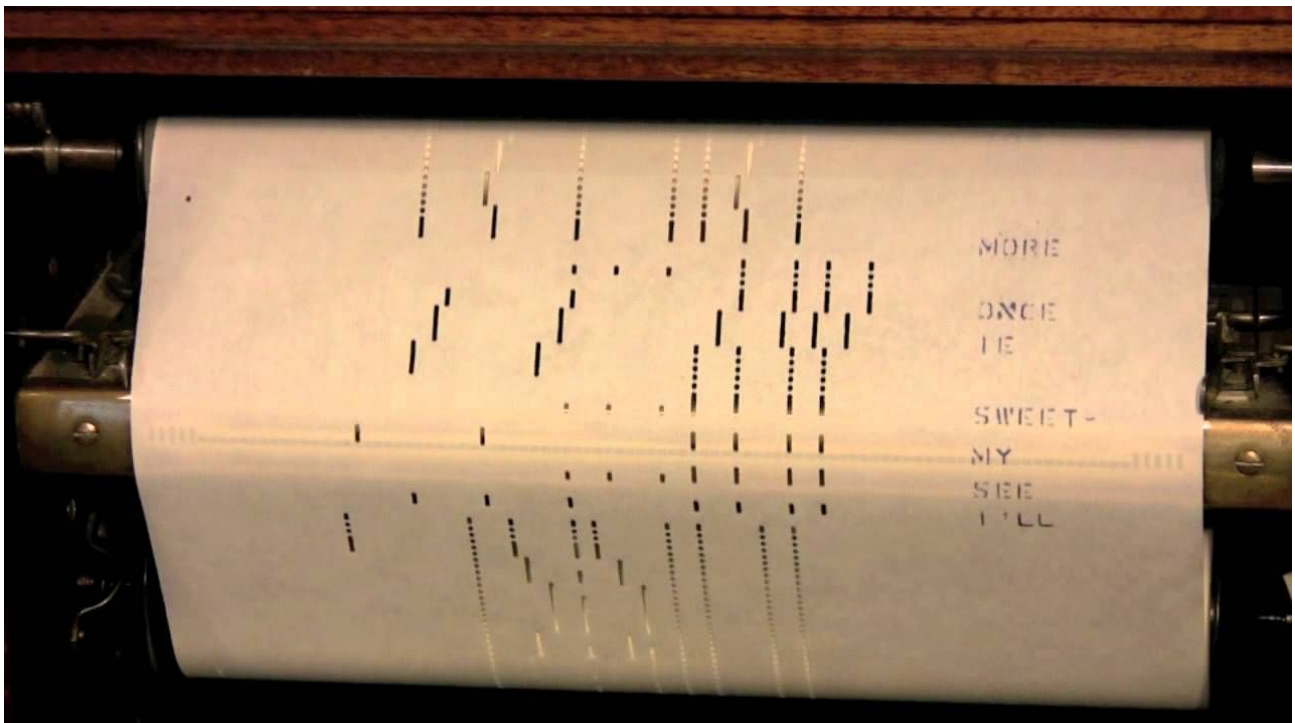
Знание устройства операционных систем жизненно необходимо для работы системного администратора, программиста, инженера в области IT-технологий или встраиваемых систем.

Наша задача – рассмотреть место и роль операционной системы, её связь с аппаратным и программным обеспечением, этапы развития вычислительной техники и собственно операционных систем.

Операционные системы представляют более низкий уровень программного обеспечения по сравнению с прикладным ПО. Поэтому мы разберём также основные положения низкоуровневого программирования, среди которых машина Тьюринга, Ассемблер и Дизассемблер, С. Кроме того, для понимания логики работы операционных систем необходимо знание устройства процессора.

Программы. Машина Тьюринга

Компьютеры зародились в связи с необходимостью расчетов, прежде всего для военных нужд. Сейчас словом «программа» никого не удивишь. Между тем, программы появились задолго до появления вычислительных машин. Механическая пианола, позволяющая воспроизводить музыку с записи, появилась в 1887 году, а шарманки, воспроизводящие запись, известны с XV века.



Запись мелодии для пианолы: https://www.youtube.com/watch?v=Z3ekShc_h-w

Но, разумеется, никаких вычислений такие программы производить не могли. В них предусматривалось только последовательное движение вперёд без возможности произвольного перехода к другому участку программы, а в роли машины выступал музыкальный инструмент.

Теперь представим себе, что лента, считываемая машиной, бесконечна, сама машина может прокручивать ленту вперёд или назад в зависимости от информации, которая была считана, а также записывать на ленту новую информацию. Внимание! Мы «изобрели» машину Тьюринга. Машина Тьюринга – это считывающее устройство, находящееся в одном из нескольких состояний, умеющее читать ленту, изменять состояние в зависимости от прочитанного с ленты, перемещаться по ней вперёд и назад в зависимости от состояний, а также выполнять записи. Число состояний, в которых может находиться машина Тьюринга, конечно и заранее определено.

Концепция машины Тьюринга была предложена Аланом Тьюрингом в 1936 году для формализации понятия алгоритма. Идея машины Тьюринга тесно связана с такими понятиями, как процессор, алгоритм, язык программирования, транслятор, компилятор, интерпретатор, виртуальная машина (машина, выполняющая байт-код).

Языки программирования называются Тьюринг-полными, если на них можно реализовать любую вычислимую функцию. К примеру, регулярные выражения не являются Тьюринг-полным языком, а, например, C или PHP – являются. (Отметим, что языки программирования – это настолько же древний компонент компьютеров, как и операционная система. Первоначально они сами играли роль операционной системы, а в настоящее время входят в её состав в виде командных интерпретаторов, компиляторов для сборки ПО из исходных кодов, библиотек и заголовочных файлов).

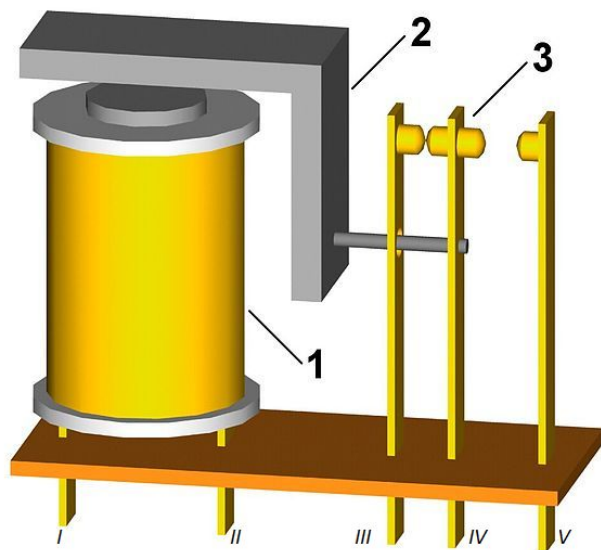
Итак, машину Тьюринга мы с вами «придумали». Осталось реализовать вычислитель – электронное устройство, которое может менять состояния, для чего необходимо найти или разработать соответствующую элементную базу.

Первые компьютеры на электромеханических реле

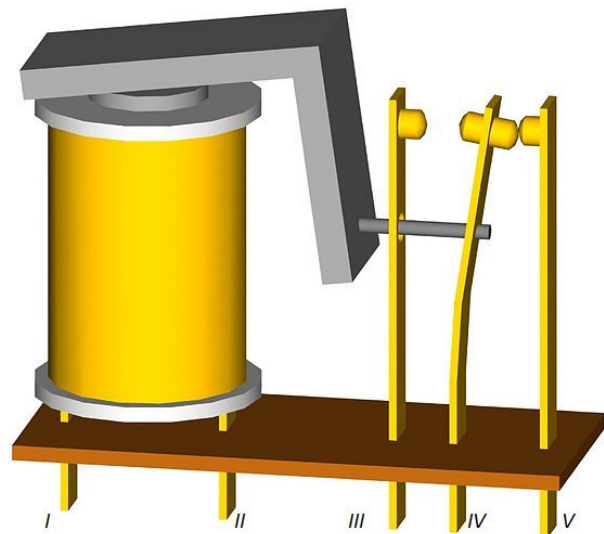
Первые вычислительные машины создавались на базе электромеханических реле. Это не удивительно, поскольку реле – простейшее устройство, позволяющее изменять состояния. Так, одна из первых ЭВМ – машина Z3, разработанная Конрадом Цузе в 1941 году, – действовала на основе телефонных реле. В 1941 году появилась и американская машина MARK-1, действовавшая также на основе электромеханических реле и переключателей. В 1944 году Конрад Цузе уже работал над машиной Z4, одновременно создавая один из первых языков программирования высокого уровня – Планкалкюль.

Конечно, были эксперименты по построению вычислительных машин на других принципах. Например, в 1936 году Владимир Лукьянов создал гидравлический интегратор – единственную в СССР вычислительную машину для решения дифференциальных уравнений в частных производных. Однако это была аналоговая машина, ориентированная на решение неуниверсальных задач. Для построения вычислительных машин, работающих с состояниями, именно реле на тот момент было оптимальным электрическим переключателем.

Электромеханическое реле – это простейший прибор с электромагнитом, позволяющий замыкать, размыкать или даже переключать линию. Чтобы механический переключатель сработал, на управляющий контакт подаётся ток.



А



Б

https://commons.wikimedia.org/wiki/File:Relay_principle_horizontal.jpg?uselang=ru

Схема работы электромагнитного реле:

1 – электромагнит

2 – подвижный якорь

3 – переключатель

В исходном состоянии магнит обесточен, и сигнал, подаваемый на контакт IV, может быть считан с контакта III.

Подача тока на контакт I приводит к возникновению магнитного поля у электромагнита 1, которое притягивает якорь 2, а он, в свою очередь, механически переводит переключатель 3 в отклонённое соединение. Таким образом, цепь между IV и III контактами размыкается, и IV контакт оказывается замкнут на V – соответственно, сигнал идёт на V контакт.

Разумеется, подобный способ управления позволяет задавать определенную логику работы за счёт комбинирования переключателей. Недостатками такого подхода являются громоздкость схемы, медленный темп работы из-за механического движения якоря и переключателя, а также ненадежность. Так, известный термин bug возник из-за жука, который застрял в реле и тем самым вывел переключатель из строя.

Для дальнейшего развития вычислительной техники требовалась более совершенная и надежная база. Такой элементной базой стали вакуумные лампы, или радиолампы.

4 века компьютеров. Лампы, транзисторы, БИС, СБИС

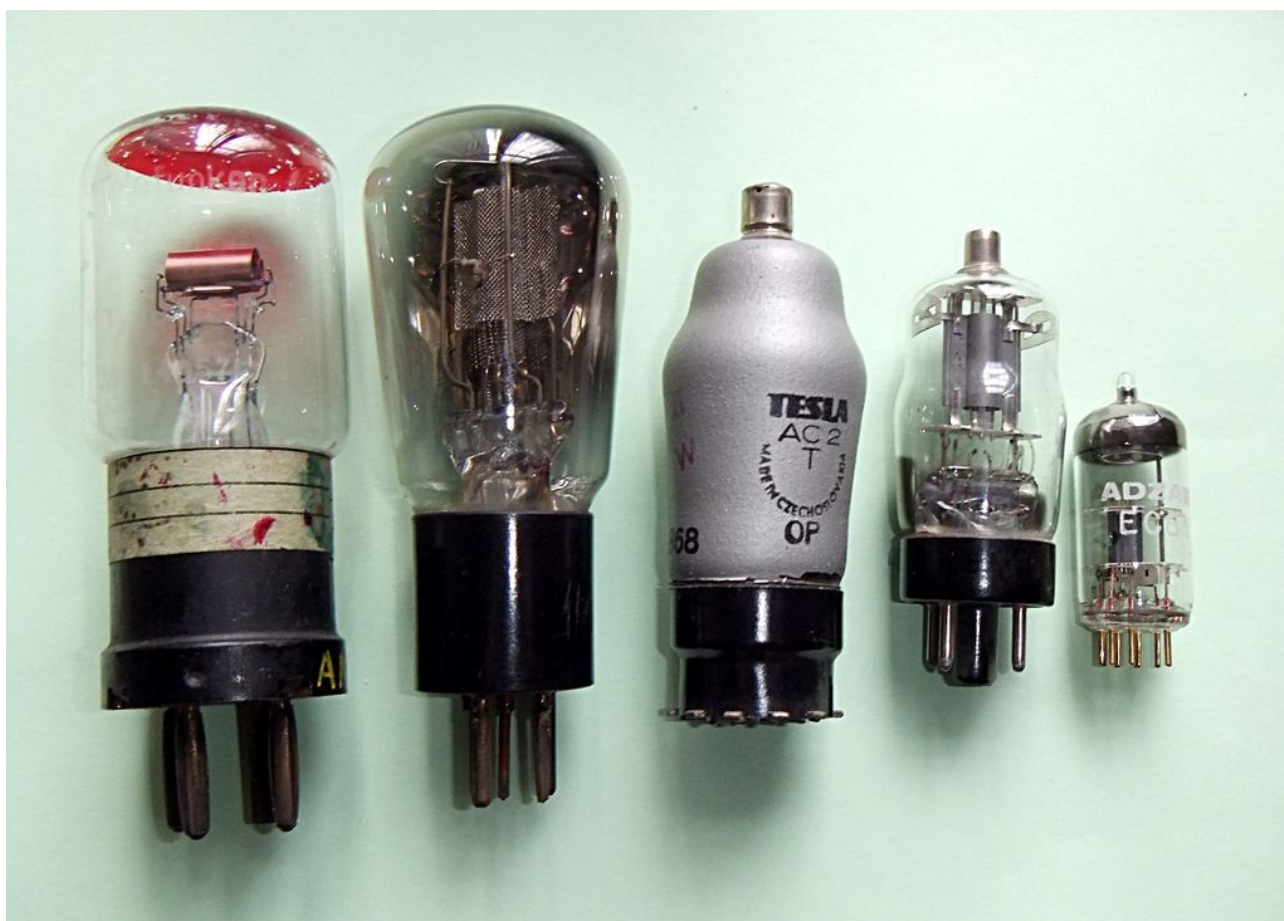
Что позволило лампам заслужить такую любовь пользователей? Возможны три ответа: устойчивость к ЭМИ, тёплый ламповый звук и красивый внешний вид (впрочем, есть мнение, что последние два пункта обозначают почти одно и то же). В своё время вакуумные лампы, или радиолампы, произвели настоящую революцию в электротехнике, позволив создать более совершенные вычислительные машины.

Эффект, используемый в вакуумных лампах, был открыт Томасом Эдисоном в 1881 году, что стало мощным стимулом к дальнейшему развитию электронных ламп. Среди нововведений того времени

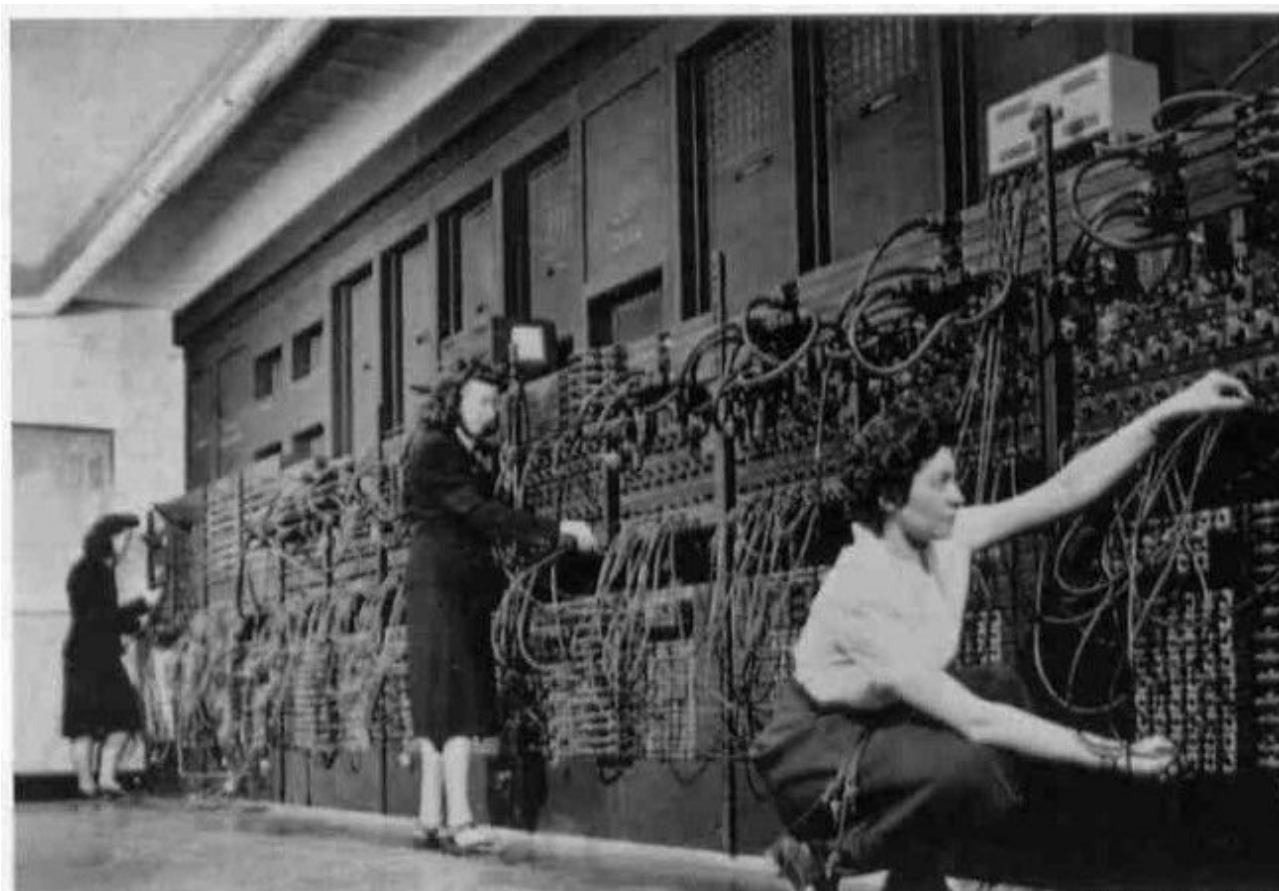
важно отметить вакуумный диод, в котором лампа пропускает ток только в одном направлении, и триод – имея дополнительный вход, эта лампа позволяла управлять током.



Вакуумные диоды. By RJB1 - Own work, GFDL, <https://commons.wikimedia.org/w/index.php?curid=14549203>



Первые известные вычислительные машины, работающие на лампах, были построены в 1943 году. Это были британский Колосс и американский ЭНИАК — предок не менее известного ЭДСАКА, созданного уже в 1948 году.



ЭНИАК

Устройства первого поколения, работающие на радиолампах, были полноценными программируемыми машинами: они обладали процессором, оперативной памятью, позволяли довольно быстро выполнять вычисления. Но и программирование, и, что не менее важно, ввод программ требовали ручного ввода в машинном коде (даже ассемблера тогда не было) путём переключения соответствующих тумблеров (см. фото). Ни о языках программирования, ни об операционных системах речь ещё не шла.

Если основным недостатком машин на электромагнитных реле была откровенная ненадёжность, машины на радиолампах, помимо этого, обладали ещё и внушительными размерами.

В 1947 году физики Уолтер Браттейн и Джон Бардин разработали первый работоспособный точечный транзистор — полупроводниковый прибор, работающий на иных физических принципах, чем радиолампа, но позволяющий выполнять те же задачи. В 1948 — 1950 годах Уильям Шокли создал теорию p-n-перехода и плоскостного транзистора, а уже в 1954 году Texas Instruments выпустила первый кремниевый транзистор.

Транзисторы оказались более надёжными, чем лампы, слабее нагревались и потребляли меньше электроэнергии. Это дало новые возможности для развития вычислительной техники.



Первый транзистор

В машинах второго поколения стали применяться магнитные диски, а для ввода-вывода информации использовались перфоленты и перфокарты. Активно развивается программное обеспечение: появляются библиотеки программ, такие программы, как автокод и монитор. Разрабатываются языки программирования (Фортран, Алгол), трансляторы (интерпретаторы и компиляторы). К наиболее известным машинам второго поколения относятся UNIVAK и Минск-22.

Для машины IBM-704 в Bell Labs (подразделение AT&T) была создана операционная система BESYS. Она предназначалась для эффективного выполнения большого количества коротких задач, динамически загружаемых с использованием перфокарт, для чего использовалось дополнительное оборудование для работы с перфокартами и печати результатов на бумаге. Такой подход послужил прообразом для операционных систем с разделяемым временем исполнения задач.

Интересным экземпляром машин второго поколения, разработанных в СССР, была машина «Сетунь», работающая на троичной логике (использовались триты и трайты вместо битов и байтов). «Сетунь» была создана под руководством Н.П. Брусенцова в 1958 году. С используемым в ней машинным кодом и особенностями его применения можно ознакомиться здесь: [https://ru.wikipedia.org/wiki/Сетунь_\(компьютер\)](https://ru.wikipedia.org/wiki/Сетунь_(компьютер))

В 1958-1959 годах был изобретён способ изолировать компоненты на кристалле полупроводника – таким образом, в одном корпусе мог помещаться не один транзистор, а целая схема, что вело к миниатюризации устройств. Так начался век микросхем.

Под микросхемой понимается интегральная схема – кристалл или пленка со схемой, – заключённая в корпусе. На интегральных схемах (ИС) были реализованы ещё более быстрые и меньшие по размеру ЭВМ третьего поколения. Во второй половине 60-х годов в США небезызвестной компанией IBM начали выпускаться серии машин IBM-360, затем IBM-370, и, наконец, в 70-х годах в СССР стали создавать вычислительные машины серии ЕС ЭВМ (Единая система ЭВМ) по образцу IBM 360/370.



IBM/360, пока еще без Intel Inside

Автор: Ben Franske - DM IBM S360.jpg on en.wiki, CC BY 2.5, <https://commons.wikimedia.org/w/index.php?curid=1189162>

Возможности BESYS для компьютеров третьего поколения уже не удовлетворяли пользователей, поэтому в 1964 году началась разработка новой операционной системы Multics, которой занимались Массачусетский технологический институт и компании General Electric (GE) и Bell Labs. Это уже была полноценная операционная система с разделением времени выполнения программ. Multics была написана на языке высокого уровня PL/I и работала на 36-битных вычислительных машинах GE-600. В системе Multics был реализован ряд идей, которые сейчас применяются в большинстве операционных систем: среди них работа с файлами на уровне операционной системы, использование виртуальной памяти, позволяющей отображать файлы в сегменты памяти, динамическое связывание программ и библиотек, иерархическая файловая система с именами файлов произвольной длины, несколько имён у файлов, символические ссылки на директории. Кроме того, система Multics позволяла на ходу подключать и отключать оборудование, реализуя модульный принцип. Впрочем, хотя Multics и вошла в число первых ОС, написанных на языке высокого уровня, она не стала самой первой из них – первенство принадлежало MCP, написанной на Алголе.

AT&T счёл Multics коммерчески неуспешной, но группа создателей системы продолжила её развивать. Новая созданная ими операционная система получила название Unics, производное от Multics: если Multics расшифровывалось как MULTIplexed Information and Computing Service, то Unics – как UNIplexed Information and Computing System, что подчёркивало простоту новой системы.

К 1969 году Unics была переписана для использования на мини-компьютерах типа PDP-7. Это была первая редакция системы (Edition I) – её первая официально выпущенная версия. С 1 января 1970 года компьютеры с UNIX – так теперь называлась новая операционная система – уже отсчитывали системное время. Началась эпоха UNIX.

Первая редакция UNIX, как и многие (но не все) операционные системы того времени, была написана на ассемблере, не содержала ОС и встроенного компилятора с языка высокого уровня. В 1966 году был разработан интерпретируемый язык BCPL, а в 1969 – язык B (Би), также интерпретируемый. По

сути, это была упрощённая для реализации на миникомпьютерах версия BCPL (который, кстати, в свою очередь сам был развитием языка CPL). Наконец, в 1972 году UNIX была переписана на языке В. Это была вторая редакция (Edition II) Unix.



Мини-компьютер PDP-7

By en:User:Toresbe - From english Wikipedia. Original description was: The Oslo PDP-7, before restoration started. I took the picture., CC SA 1.0, <https://commons.wikimedia.org/w/index.php?curid=1963657>

В 1969 – 1973 годах на основе Би был разработан компилируемый язык, получивший название С (Си). На языке Си и была написана новая версия UNIX, известная как третья редакция – эта версия включала встроенный компилятор С. В том же году вышла и четвёртая редакция. Теперь на Си было переписано и системное ядро (в духе системы Multics, также написанной на языке высокого уровня ПЛ/I). В 1975 году вышла пятая редакция, полностью переписанная на Си. В то время UNIX широко распространяется в учебной и университетской среде, у неё появляются новые версии, разработанные другими организациями. Тогда же, в 1975 году, Bell Labs выпускает шестую редакцию системы, а в 1976 году выходит книга Джона Лайонса «Комментарии к 6-й версии UNIX с исходным кодом», содержащая текст исходного кода ядра 6-й версии AT&T UNIX и комментарии к исходным текстам, которые объясняли функционирование операционной системы UNIX. Последней версией UNIX была седьмая – затем наступила эра множества версий UNIX от разных производителей. В седьмой версии появился интерпретатор командной строки Bourne shell (заново переписанной версией которого является современный bash), интересный также тем, что был создан под влиянием языка Алгол-68 – потомка языка Алгол и непосредственного конкурента другого дочернего языка – известного всем Паскаля.

Таким образом, именно на третье поколение пришлось по-настоящему революционные события в развитии вычислительной техники. Многие из разработанного в тот период в Unix применяется до сих пор.

Дальнейший этап развития компьютера – четвёртое поколение, связанное с ещё большей интеграцией устройства и возможностью реализации уже внутри одной микросхемы целого ряда блоков, до этого реализуемых независимо друг от друга. Многие связывают четвёртое поколение с созданием компанией Intel микропроцессора, реализующего на одной схеме регистры, умножители, арифметико-логическое устройство и управляющее устройство.

Процессор

Рассмотрим историю процессоров (в терминологии периода их появления – микропроцессоров), в настоящее время широко распространенных и ставших стандартом. Речь идет об Intel-совместимых процессорах.

История Intel-процессоров

Исторически первым процессором, предшественником современных Intel-совместимых и некоторых других процессоров был 4004. Он был разработан компанией Intel в 1971 году по заказу японской компании Nippon Calculating Machine, Ltd., впоследствии переименованной в Busicom, для производства печатающих калькуляторов (UNICOM 141-P/BUSICOM 141-PF). Заказчики хотели получить в итоге калькулятор на 12 микросхемах – при этом планировалось прошивкой ПЗУ вносить изменения вместо разработки новых микросхем. Это была удачная находка, но ещё более удачным стало предложение инженера Intel Теда Хоффа сократить число микросхем до четырёх, реализовав арифметико-логическое устройство, регистры, управляющее устройство и т.д. в одной микросхеме.

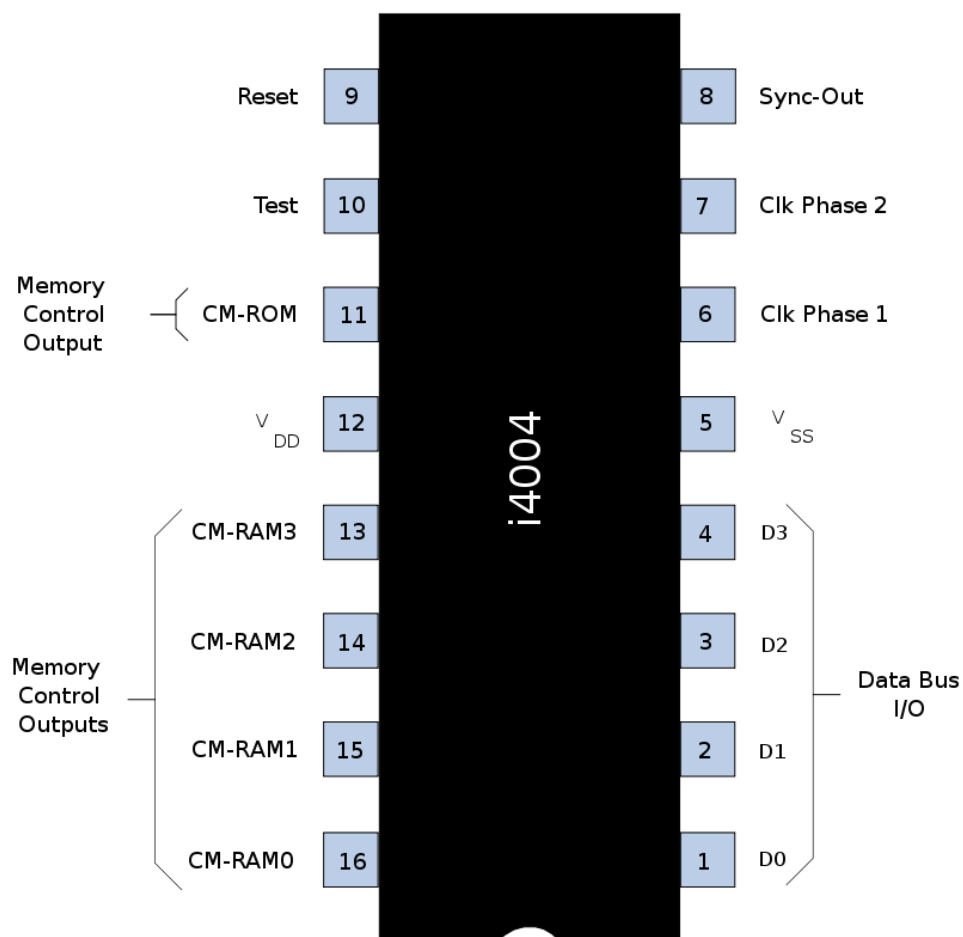
В составе калькулятора использовались следующие микросхемы: 4001 – 256 байтовое ПЗУ, 4002 – 40-байтовое ОЗУ, 4003 – 10-битный расширитель ввода-вывода (сдвиговый регистр, превращающий последовательный код в параллельный для связи с клавиатурой и принтером) и 4004 – процессор. Откуда произошла такая нумерация? Изначально Интел производили ПЗУ, которые имели нумерацию вида 1xxx – соответственно, ОЗУ и сдвиговый регистр относились к сериям 2xxx и 3xxx, а процессорам было присвоено наименование вида 4xxx. Так как остальные три устройства предназначались для работы с процессором 4004, они тоже получили альтернативную нумерацию вида 400x, указывающую на работу с процессором. Однако процессор был разработан позже и потому четвёртый знак получил больший, чем разработанные ранее ПЗУ и ОЗУ.

Конечно, операционной системы в привычном понимании у калькулятора не было, но была прошивка, которая позволяла калькулятору выполнять её функции (с восстановленной прошивкой можно ознакомиться по адресу http://www.4004.com/assets/Busicom-141PF-Calculator_asm_rel-1-0-0.txt)

Подход, при котором контроллер снабжается прошивкой – одной выполняющейся программой, позволяющей устройству выполнять свои функции, – применяется до сих пор. Однако подобно тому, как 4004 эволюционировал до современных процессоров, многие современные устройства также содержат в прошивке уже не одну программу, а целые ОС: в частности, есть и устройства, использующие прошивку на базе ядра Linux (и даже ОС GNU/Linux).

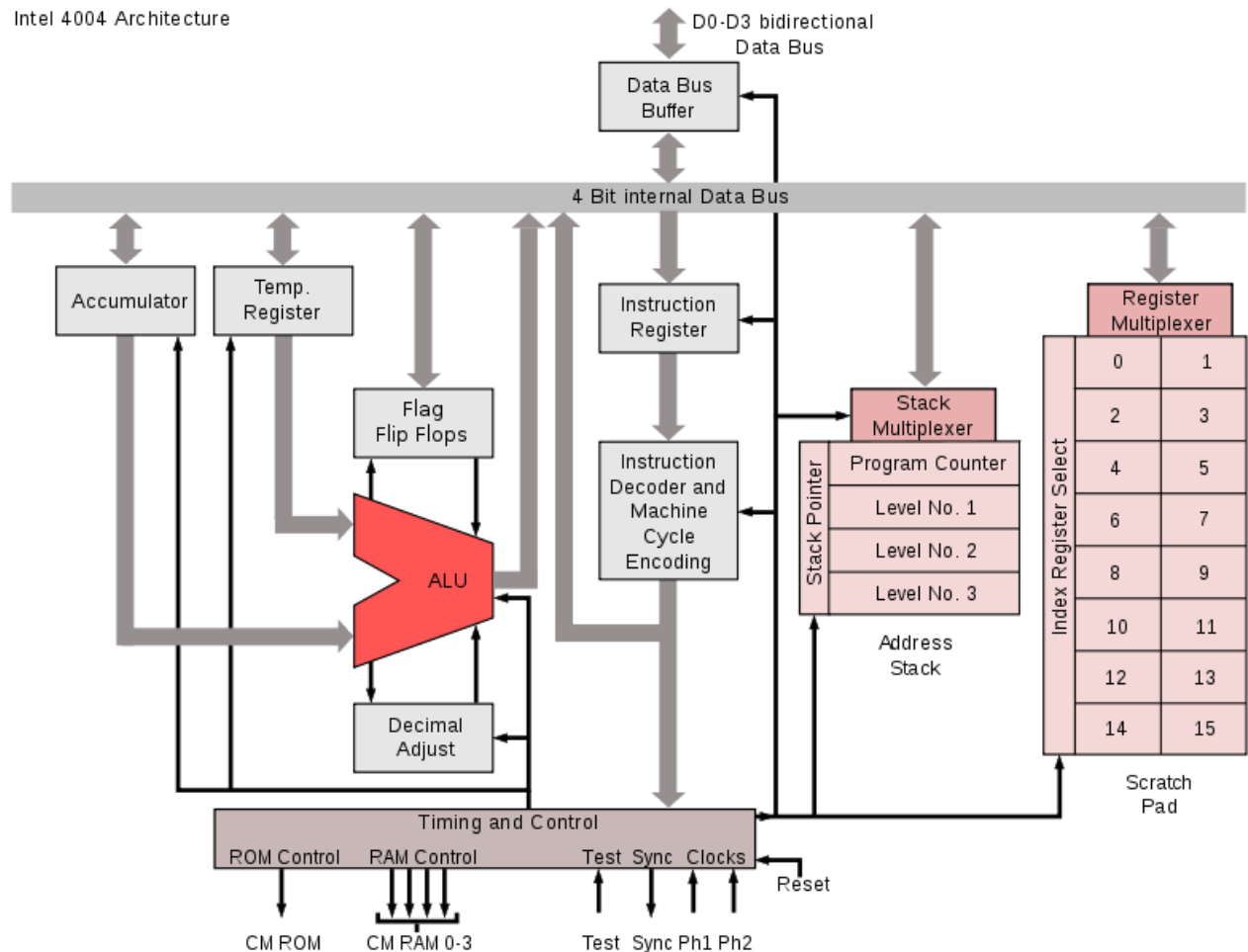


Калькулятор BUSICOM 141-PF



Контакты микросхемы 4004

Intel 4004 Architecture



Блок-схема процессора 4004

Автор: Appaloosa - собственная работа, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=2954987>

Следующий процессор 4040 был улучшенной версией процессора 4004 и вместо 16 включал в себя уже 24 вывода. Кроме того, в отличие от предшественника, в процессоре 4040 появилась поддержка прерываний, а у микросхемы – новые входы. С data sheet для процессора 4040 можно ознакомиться по адресу <http://www.edgar-elsen.de/4040.pdf>

Следующий процессор, выпущенный в 1972 году, был уже 8-битным. Его создатели продолжали придерживаться схемы присвоения наименований по тому же принципу, и новый процессор получил название 8008.

История этого процессора в чём-то повторяет историю 4004. В 1969 году компания Computer Terminal Corporation, в дальнейшем переименованная в Datapoint, заказывает у Intel новый процессор, который планировалось использовать в новом терминале Datapoint 2000. Так же, как и Busicom, Datapoint планировали разместить систему на нескольких микросхемах, но инженер Intel, небезызвестный Тед Хофф, по своему обыкновению предложил разместить все компоненты нового процессора на одной микросхеме. Когда микросхема была готова, Datapoint разорвали контракт, но у процессора нашлись новые заказчики. Примечательно, что на базе микропроцессора 8008 в 1974 году были разработаны мини-компьютеры Mark-8 и Scelbi-8N.

Дальнейшим шагом стало появление 8-битного процессора, получившего наименование 8080. Хотя он был 8-разрядным и содержал семь 8-битных регистров (A, B, C, D, E, H, L), у него были некоторые

ограниченные возможности обработки 16-разрядных чисел, для чего регистры объединялись в пары: BC, DE, HL.

Процессор 8080 оставил не меньший след в истории, чем 4004. На его базе MITS разработали компьютер Альтаир-8800 – первый компьютер, который можно было собрать дома.



Альтаир-8800.

Микрокомпьютеры по своим возможностям и вычислительным мощностям представляли шаг назад по сравнению со своими старшими братьями третьего поколения (впрочем, со временем это упущение было исправлено), что было следствием дешевизны и массовости производимой машины. ЭВМ не имела ни клавиатуры, ни экрана, программы необходимо было вводить в двоичной форме, переключая набором маленьких ключей, которые могли занимать два положения – вверх и вниз; результаты считывались также в двоичных кодах лампочками-индикаторами. Операционная система у таких устройств отсутствовала, взаимодействие оставалось на уровне машин первого поколения, хотя по сравнению с первыми ЭВМ это была более компактная и надежная вычислительная машина.

С Альтаир-8800 началась карьера компании Microsoft (тогда еще Micro soft). Встретившись с Генри Робертсом, разработчиком компьютера, Билл Гейтс и Пол Аллен договорились о новой встрече через пару недель, чтобы познакомиться с интерпретатором языка Basic для Альтаира. На самом деле у Гейтса и Аллена не было не то что интерпретатора, но и самого Альтаира. Для разработки интерпретатора Аллен, пользуясь документацией, написал эмулятор Альтаира на PDP-10. Так Altair Basic стал первым языком программирования, написанным для микрокомпьютеров, и одновременно операционной системой для Альтаир 8800 (поскольку интерпретируемый язык может выступать в роли операционной системы. Более того, для многих микрокомпьютеров в дальнейшем та или иная вариация языка Basic являлась операционной системой, будучи прошитой в ПЗУ и загружаемой при старте. Современные операционные системы также обладают интерфейсом командной строки, предоставляющей, по сути, возможности языка программирования, как, например, уже упомянутый bash). Кроме того, Altair Basic был основным источником дохода Microsoft до разработки MS DOS.

Конец 70-х годов ознаменовался расцветом 8-битных процессоров. Это были как конкуренты (например, Motorola 6800, нашедшая применение в машине MITS Altair 680 – с заменой i8080 на Motorola 6800; её аналог и конкурент от MOS Technology – 6502, использовавшийся в платформах NES, известной в России под именем Dendy; Atari 800, Atari 2600, Commodore 64, Apple I и II, копии

Apple – болгарский Правец и советский Агат), так и совместимые аналоги от компаний NEC, Siemens, Zilog, AMD.



Apple I

Автор: Photo taken by rebelpilot - rebelpilot's Flickr Site, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=183820>

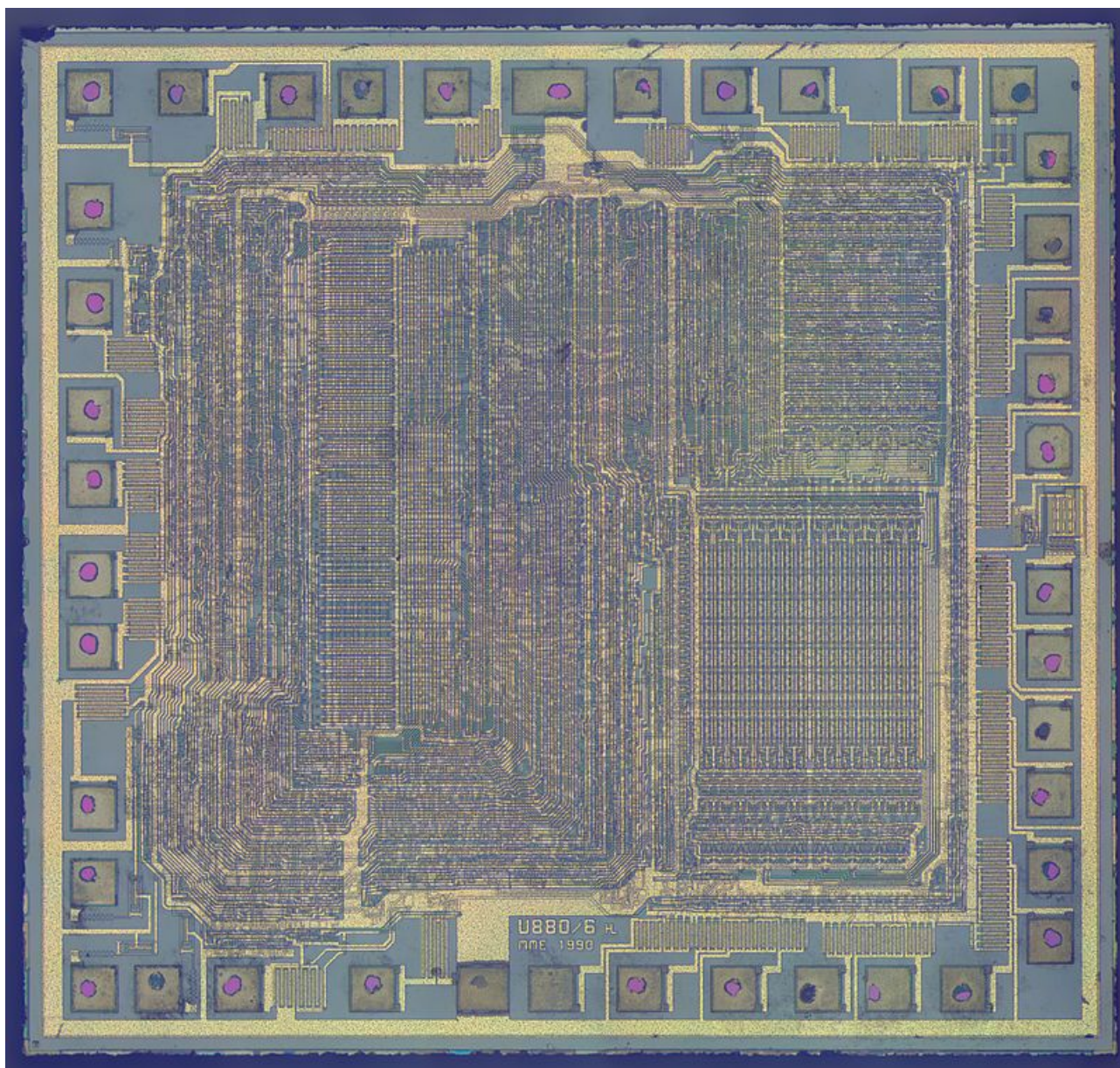
Отдельно следует упомянуть процессор Z80 от компании Zilog. Этот восьмибитный процессор разработали в 1978 году инженеры, непосредственно участвовавшие в создании процессора 4004. За основу был взят процессор 8080, и Z80 был с ним двоично совместим, что позволяло выполнять на процессорах Z80 программы, написанные для 8080, без изменений (среди них была и дисковая операционная система CP/M). При этом процессор обладал рядом улучшений: среди них было увеличенное число регистров (что в принципе позволяло реализовывать микроконтроллеры без ОЗУ), один источник питания с напряжением +5В вместо трёх (+5В, -5В и +12В) у Intel 8080. На базе Z80 производились известные микрокомпьютеры, подключаемые к телевизору и бытовому магнитофону в качестве запоминающего устройства и получившие название «бытовых компьютеров» – например, ZX80, ZX81 и по-настоящему известный ZX Spectrum. Примечательно, что, несмотря на поддержку процессором Z80 прерываний, эти машины прерываний не использовали.

У машин Z80 встроенной в ПЗУ системой был язык программирования Sinclair Basic – весьма оригинальный диалект Бейсика с однобайтовыми (при хранении, но не при печати на экран) командами. В составе команд были инструкции LOAD и SAVE для загрузки и сохранения программ с магнитофона, RUN – для запуска (многие программы в машинных кодах изначально содержали загрузчик на Sinclair Basic), PEEK и POKE – непосредственно для чтения и записи ячейки памяти, IN и OUT – для доступа к портам. Для запуска программы в машинных кодах использовалась функция USR, которая возвращала значение регистра BC. Таким образом загружались программы в машинных кодах (что можно сравнить с COM-файлами в дисковых ОС), и точно так же можно было использовать встроенные функции ОС, вызывая их по прямому адресу. Привычного механизма вызова функции через прерывания устройство не имело. Кроме того, ассемблер Z80 обладал некоторыми отличиями от Intel-ассемблера, среди которых была, например, мнемоника LD вместо MOV. Подробнее с работой устройства можно ознакомиться по адресу

4

A

К
К
К
К
К
К
К
К
К
К



Микросхема КР1858ВМ1 — Автор: ZeptoBars - <http://zeptobars.ru/en/read/KR1858VM1-Z80-MME-Angstrom>, CC BY 3.0, <https://commons.wikimedia.org/w/index.php?curid=33458670>

Следующим процессором от Intel был 8086. В отличие от своих предшественников, это был уже 16-битный процессор. Любопытно, что и современные процессоры могут выполнять инструкции, предназначенные для 8086.

Процессор 8086 содержал четырнадцать 16-разрядных регистров: 4 регистра общего назначения (AX, BX, CX, DX), 2 индексных регистра (SI, DI), 2 указательных регистра (BP, SP), 4 сегментных регистра (CS, SS, DS, ES). Для нужд процессора использовались указатель команды (IP) — фактически программный счётчик — и регистр флагов FLAGS, состоявший из 9 битовых флагов, которые можно было воспринимать как битовые регистры и управлять ими особым образом. При этом регистры данных (AX, BX, CX, DX) допускали адресацию не только целых регистров, но и их младшей половины (регистры AL, BL, CL, DL) и старшей половины (регистры AH, BH, CH, DH), что позволяло работать с ними как с 8-битными регистрами. Последнее давало возможность выполнять на процессоре не только новое 16-разрядное программное обеспечение, но и сохранять совместимость с 8-битными программами, хотя непосредственный перенос был невозможен. Аналогичные команды процессора имели другие байтовые последовательности команд. Так, в 8080 и ZX80 команда NOP («ничего не делать»), служившая для заполнения и выравнивания адресов, имела код 0x00, в 8086 и

далее – 0x90, а 0x00 является составной частью одной из команд сложения. Таким образом, несмотря на совместимость программ на уровне мнемоник, для переноса программы её пришлось бы перекомпилировать.

16-разрядный регистр позволял распоряжаться $2^{16}=65536$ байт памяти, то есть порядка 64Кб, несмотря на то, что системная шина позволяла адресовать до 1 Мб памяти. Для этого была разработана сегментная адресация памяти с использованием до 4-х сегментов по 64 Кб.

В процессоре 8086 был реализован прототип конвейерной обработки: в то время, когда процессор выполнял одну инструкцию, уже происходило чтение следующей команды.

Процессор 8086 применялся в первом переносном компьютере – Xerox Note Tracker, предшественнике ноутбуков. Он использовал целых три процессора 8086: в качестве основного ЦПУ, в качестве графического процессора и для управления вводом-выводом.



Xerox NoteTracker

Модификация 8088 применялась в революционной серии компьютеров IBM PC/XT. В погоне за сохранением рынка компания IBM решила использовать в новой ЭВМ узлы других производителей, фактически создав конструктор, что предопределило дальнейшую судьбу персональных компьютеров.

Отечественные клоны 8086 и 8088 – К1810ВМ86 и К1810ВМ88 – использовались в советских машинах ЕС ПЭВМ начиная с ЕС-1840. Машины были IBM-совместимыми.

Шагом в дальнейшем развитии машин стали 16-битные процессоры 80186 и 80286. Если первый практически не применялся в ЭВМ, то на базе второго появилось новое поколение компьютеров IBM AT. В отличие от 8086, в 80286 адреса и данные передавались через разные контакты процессора, что позволяло устройству действовать быстрее. В нём появился защищённый режим памяти и новые регистры для его обслуживания. 80386 уже был 32-байтным процессором, в котором появились 32-битные регистры (EAX, EBX), страничная организация памяти, аппаратная поддержка многозадачности.

Разрядность процессора

В предыдущем тексте упоминаются 4-разрядные, 8-разрядные, 16-разрядные процессоры... Что это означает? Разрядность процессора – это показатель числа бит, с которыми процессор может манипулировать одновременно, то есть размер машинного слова.

Что означает понятие «4-битный процессор»? Число, с которым может он работать, лежит в диапазоне от 0000 до 1111, т.е. от 0 до 15 – ровно 16 значений (2^4). Возможна ли работа с большими числами? Возможна. Но для этого потребуется большее число инструкций процессора.

Стоит увеличить число бит в два раза, и мы получим уже 256 значений (2^8)! Таким образом, 8-битный процессор уже мог работать с символами ASCII. 16-битный процессор (2^{16}) – с 65 536 значений. Для 32-битного и 64-битного значения несложно рассчитать самостоятельно.

Количество адресуемой памяти зависит от разрядности процессора. Адрес ячейки памяти задаётся одним или парой регистров. Технически возможно реализовать, например, 8-битный доступ к памяти на 4-битном компьютере с использованием комбинации двух регистров и других решений. Тем не менее известно, что 32-битные компьютеры, как правило, адресуют до 4-х гигабайт памяти. Почему? $2^{32}=4$ Гб. Поэтому 64-битные процессоры могут использовать оперативную память, превышающую 4 Гб.

Регистры процессора

В состав процессора входит сверхбыстрая оперативная память (СОЗУ), которая располагает именованными ячейками. Работа с регистрами – наиболее быстрая, так как происходит внутри ЦПУ и не требует обращения к внешним по отношению к нему устройствам. СОЗУ является более дорогой и, соответственно, ограниченной, потому в СОЗУ может размещаться лишь небольшой объем данных. Как правило, это данные, необходимые для обеспечения работы процессора, и промежуточные данные вычислений (как арифметических, так и предназначенных для передачи номеров функций и аргументов при вызове программных прерываний).

Регистры могут быть сгруппированы таким образом, что к двум можно обращаться как к одному. Рассмотрим это на примере регистра А (аккумулятор).

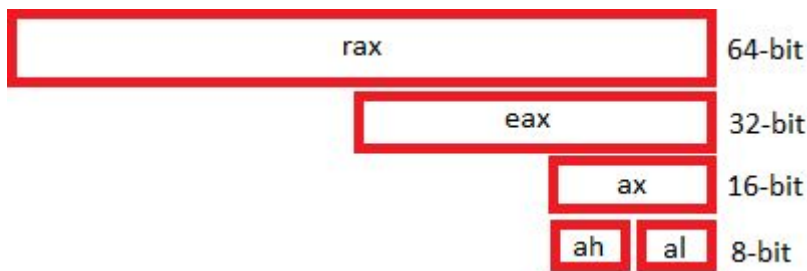
Процессор 8080 (результат дальнейшего развития восьмибитного 8008, также 8-битный) насчитывал регистры А, В, С, D, Е, H, L, но имел ограниченные возможности обработки 16-разрядных чисел, для чего регистры объединялись в пары BC, DE, HL.

В 16-битном процессоре 8086 уже имелся 16-битный регистр AX, который можно было рассматривать как пару восьмибитных регистров AH и AL (high и low).

В 32-битном процессоре 80386 появились 32-битные регистры, к младшей части которых можно было обращаться по-прежнему. Так, если аккумулятор в 32-битной версии имел название EAX, то к его младшей 16-битной части можно было обратиться как к AX, а к её половинкам в свою очередь как к AH и AL. Старшая часть регистра такого расположения не получила – отдельный доступ имелся только к младшей части.

Аналогично сложилась история и с 64-битными процессорами. 64-битный регистр аккумулятора имеет наименование RAX, младшая 32-битная часть которого по-прежнему доступна как EAX. В свою очередь, ее младшая 16-битная часть доступна как AX, а обе ее части – как AH и AL.

Всё вместе это выглядит следующим образом:



Обратите внимание: изменяя, например, регистр AL, вы изменяете и AX, и EAX, и RAX (при этом AH не изменится). Изменяя AX, с большой долей вероятности вы поменяете и AH, и AL (это зависит от того, совпадает ли старое и новое число в старшей и младшей своей части или нет). Таким образом, несмотря на возможность работы с отдельными «частями» регистра, вы работаете со всем регистром в целом! При этом иногда бывает удобно работать, например, с AH и AL как с отдельными регистрами, не воспринимая их как единое целое.

Какие бывают регистры

Что такое регистры? На высоком уровне их можно воспринимать как особые переменные. Более того, если написать строчку:

```
for (i; i++; i<100) { s=s+2;}
```

– при компиляции программы роль переменной *i* будет играть регистр CX. Кстати, и в роли переменной *S* также может выступить регистр AX – впрочем, не везде и не всегда, так как программа на высоком уровне будет выглядеть довольно сложно после компиляции. Одна и та же переменная может находиться в разных участках памяти и выполнять функцию регистра.

Регистры общего назначения

Содержат 4 регистра счётчиков-аккумуляторов и 4 регистра для работы с памятью (кроме того, для работы программы с памятью, есть ещё внутренние регистры, напрямую недоступные для изменения программой).

Регистры счётчиков-аккумуляторов:

- RAX/EAX/AX: Accumulator register – аккумулятор, применяется для хранения результатов промежуточных вычислений;
- RBX/EBX/BX: Base register – базовый регистр, применяется для хранения адреса (указателя на объект в памяти);
- RCX/ECX/CX: Counter register – счетчик, его неявно используют некоторые команды для организации циклов (см. loop);
- RDX/EDX/DX: Data register – регистр данных, используется для хранения результатов промежуточных вычислений и ввода-вывода.

Нижняя 16-битная часть каждого из вышеперечисленных регистров также состоит из половинок (AH:AL, BH:BL, CH:CL, DH:DL).

Следующие четыре регистра, как правило, не делятся на 8-битные половинки:

- RSP/ESP/SP: Stack pointer register – указатель стека, содержит адрес вершины стека;
- RBP/EBP/BP: Base pointer register – указатель базы кадра стека (англ. stack frame), предназначен для организации произвольного доступа к данным внутри стека;

- RSI/ESI/SI: Source index register – индекс источника, в цепочечных операциях содержит указатель на текущий элемент-источник;
- RDI/EDI/DI: Destination index register – индекс приемника, в цепочечных операциях содержит указатель на текущий элемент-приемник.

Сегментные регистры

- CS: Code segment – содержит текущий сегмент кода. Процессор может загружать инструкции только из указанного в CS сегмента. Регистр CS используется процессором в связке CS:IP. Не может меняться непосредственно, командой MOV – тем не менее процессорные команды вызовов и переходов (JMP, CALL, RET) фактически могут изменять этот регистр;
- DS: Data segment – содержит текущий сегмент данных. Может использоваться в комбинации DS:BX, DS:SI, DS:DI (в формате сегмент:смещение) для манипуляции с данными;
- SS: Stack segment – содержит текущий сегмент стека. Фактически регистры SP и BP указывают смещение относительно сегмента, заданного в SS (то есть работа идет с памятью в SS:SP и SS:BP);
- ES: Extra segment – дополнительный сегмент, часто используется неявно в строковых командах как сегмент-получатель (работа идет с ES:DI);
- FS: F segment – дополнительный сегментный регистр без специального назначения;
- GS: G segment – дополнительный сегментный регистр без специального назначения.

В ОС Linux используется плоская модель памяти (flat memory model), в которой все сегменты описаны как использующие всё адресное пространство процессора и, как правило, явно не используются. Таким образом, каждый адрес представляется в виде 32-битных смещений. Это позволяет одним 32-битным регистром адресовать до 4 Гб памяти и упрощает программирование. Тем не менее возможность работы с сегментами есть и может использоваться.

Обратите внимание: сегментные регистры – 16-битные. Почему? Ответьте на этот вопрос самостоятельно.

Регистры, связанные с состоянием процессора

- RIP/EIP/IP – instruction pointer. Данный регистр указывает на адрес текущей команды и не может напрямую изменяться командами MOV, ADD и т.д. Выполнение каждой команды либо изменяет этот регистр на размерность команды, либо меняет на указанный адрес. Фактически команды JMP, RET, REP, LOOP, CALL изменяют состояние данного регистра. Команду NOP, которая ничего не делает, можно считать командой INC IP. Команду JMP XX можно изобразить как MOV IP, XX, но обычно так не делают, хотя сам процессор именно так воспринимает эти команды;
- RFLAGS/EFLAGS/FLAGS – регистр флагов. Если AX можно воспринимать как два 8-битных регистра AH и AL, чтобы работать с ними напрямую, регистр флагов состоит из однокбитных регистров. Работа с ними напрямую невозможна, но ряд команд по-разному выполняются, в зависимости от значения данных флагов.

Некоторые значимые флаги:

ZF: zero flag – флаг нуля:

- 1 – результат последней операции нулевой;
- 0 – результат последней операции ненулевой;

IF: interrupt flag – флаг прерываний:

- 1 – в этом случае процессор обрабатывает внешние (аппаратные прерывания);
- 0 – в этом случае процессор игнорирует внешние прерывания (кроме NMI);

CF: carry flag – флаг переноса:

- 1 – во время арифметической операции был произведен перенос из старшего бита результата;
- 0 – переноса не было;

OF: overflow flag – флаг переполнения:

- 1 – во время арифметической операции произошёл перенос в/из старшего (знакового) бита результата;
- 0 – переноса не было;

DF: direction flag – флаг направления. Указывает направление просмотра в строковых операциях:

- 1 – направление «назад», от старших адресов к младшим;
- 0 – направление «вперед», от младших адресов к старшим.

Пример использования:

```
if (a==7)
{ // обработка кода 7
}
else
{
    //иначе
}
```

На ассемблере это будет выглядеть так:

```
CMP AX, 0x07 ; сравниваем регистр AX и число 7

JZ xx ; если ZF=00 переходим к xx – обработка кода 7

здесь иначе.

xx: обработка кода 7
```

Есть еще вариант записи:

```
CMP AX, 0x07

JE xx
```

Эта запись передаёт то же самое (синоним), но в ней мы ориентируемся уже не на флаг нуля, а на равенство, то есть результат Compare – Equal. Точно так же дело обстоит с переходом в случае ненулевого результата: JNZ – если ZF!=0 или, что то же самое, JNE – результат Compare – Not Equal

Несмотря на то, что у каждого флага свои механизмы работы (например, для установки и сброса IF есть команды STI и CLI), существует общий способ работы с регистром флагов: PUSHF запишет значение регистра FLAGS на вершину стека, а POPF – извлечет значение с вершины стека и запишет в регистр FLAGS.

Порядок байтов

Существует как минимум два способа хранения и передачи байтов, которые могут различаться в разных реализациях процессоров: так называемые `big-endian` и `little-endian`. Названия этих способов взяты из книги Дж. Свифта «Путешествие Гулливера», где упоминаются жители страны, воевавшие между собой из-за того, как правильно разбивать вареное яйцо – с тупого конца (`big-endians`, тупоконечники) или с острого (`little-endians`, остроконечники). (Кстати, на похожую тему существует мультфильм «Хроники бутербродной войны»: <https://www.youtube.com/watch?v=b1jFOizRoAY>).

`Big-endian` («тупоконечный», от старшего к младшему) – это интуитивно понятный, привычный порядок байтов. Он также называется сетевым порядком байтов (`network byte order`) и используется в процессорах SPARC, MIPS*. Пример: число 256 в 16-ричной системе счисления – 01 00.

`Little-endian` («остроконечный», от младшего к старшему) – это противоположность вышеописанному порядку байтов. Он более удобен для арифметических операций, но хуже считывается при чтении или правке машинного кода. `Little-endian` также называется интеловским порядком байт (`intel byte order`) и применяется в процессорах Intel. Пример: число 256 в 16-ричной системе счисления (`little-endian`) – 00 01.

Существуют линейки процессоров с поддержкой обоих типов либо с возможностью переключения переключкой на плате. К ним относятся ARM и MIPS* (автор сталкивался с реализацией MIPS `Big-endian`, но в общем случае они могут быть как `big-endian`, так и `little-endian`).

Для преобразования одного порядка байтов в другой существуют разные механизмы. Так, в C для этого служат функции `ntohs` (`network to host short`), `htons` (`host to network short`), `ntohl` (`network to host long`), `htonl` (`host to network long`), которые производят соответствующее преобразование, если порядок байтов, отличается сетевого, или ничего не делают, если порядок совпадает.

В ассемблере 32-битных процессоров Intel имеется инструкция `bswap` (например, `bswap eax`), которая преобразует порядок байт регистра в противоположный. Её аналог для 16-битных процессоров – команда `xchg` (например, `xchg al,ah`), которая меняет два регистра местами. Почему именно эта команда преобразует порядок байтов, вам предлагается ответить самостоятельно исходя из информации о регистрах Intel-совместимых процессоров.

В файлах Unicode-16 в начале файла может использоваться Byte order mark (BOM) – двухбайтовый символ, который имеет значение 0xFEFF. Символа 0xEFFF не существует, поэтому если файл начинается с 0xEFFF, это знак, что файл необходимо преобразовать из `little-endian` в `big-endian`.

Прерывания

Прерывание – это функция, имеющая номер, по которому производится её вызов. Отличие такой функции от обычных состоит в том, что вызов обычной функции осуществляется по адресу (например, `CALL 0x0110`), а вызов прерывания – по номеру (например, `INT 3`). Из самого названия «прерывание» следует, что прерывание останавливает работу программы и процессор переходит к обработчику прерывания. Пример с `INT 3` – это случай, когда прерывание явно вызывается из программы. Но программа может быть остановлена и процессор переходит к обработчику прерывания и без соответствующей инструкции – например, при наступлении какого-либо события.

Операционная система включает таблицу векторов прерываний, сопоставляя номера прерываний и фактические адреса нахождения обработчиков прерываний. Обработчики прерываний могут перекрываться другими, сохраняя адрес старого прерывания и вызывая его в процессе выполнения собственного кода.

Исключения

Исключения – это прерывания, которые возникают в случае определенных событий, связанных с выполнением программы (деление на ноль, ошибка сегментации, переполнение стека и т.д.) В таких случаях программа останавливается и управление передается обработчику. Как правило, программа может переопределить прерывание и продолжить работу, но обычно происходит снятие программы с выполнения и вывод соответствующей ошибки. Например, прерывание по делению на ноль вызовет инструкция DIV.

При обработке прерывания регистры IP и FLAGS сохраняются в стеке, после чего управление переходит к обработчику прерывания. После завершения прерывания IRET, если управление будет возвращено назад, IP и FLAGS восстанавливаются – фактически восстановление IP и есть переход к возобновлению инструкции.

Аппаратные прерывания

Аппаратные прерывания обрабатываются при поступлении сигнала IRQ на вход процессора. Они служат для взаимодействия процессора с устройствами: например, нажатие кнопки на клавиатуре приводит к поступлению соответствующего прерывания на процессор, и обработчик прерывания считывает код введенного символа, который затем может быть запрошен соответствующим вызовом операционной системы. Номера IRQ в настоящее время не совпадают с номерами INT.

Для управления обработкой прерываний служит флаг FI. Инструкции установки флага соответствует флаг STI (Set Interrupt-Enable Flag), инструкции сброса флага – CLI (Clear Interrupt-Enable Flag). Если флаг сброшен, внешние прерывания, за исключением немаскируемого (NMI), игнорируются. Примечательно, что команда HLT служит для остановки процессора до поступления внешнего прерывания. Комбинация CLI HLT, таким образом, полностью останавливает процессор.

Немаскируемое прерывание

INT 2 – немаскируемое прерывание NMI – поступает на вход процессора NMI, и в этом случае его обработчик будет выполнен, даже если IF сброшен. Название данной операции означает, что прерывание не может быть маскировано сбросом флага IF. NMI генерируется, например, при ошибке чётности памяти. Память в ОЗУ хранится в банках по 9 бит, девятый бит – контроль четности, который позволяет на аппаратном уровне отслеживать корректность работы ОЗУ и тем самым используется, чтобы количество единиц в 8 битах было нечетным. Сбой чётности приводит к вызову прерывания NMI и, как правило, к остановке машины. В других процессорах и микроконтроллерах NMI может служить для других целей: для удалённого управления процессором, вывода его из HLT по событию и т.д. Интересно, что первые ОЗУ давали сбои из-за альфа-частиц, образующихся при распаде примесей урана или тория, который мог присутствовать в устройстве из-за технического несовершенства микросхем.

Прерывание для отладки – breakpoint

Большинство прерываний кодируются двумя байтами. Прерывание INT 3 кодируется одним байтом 0xCC. Это специально вставленная точка останова, которая вызывает прерывание-обработчик для отладки.

Программные прерывания

Программные прерывания – это прерывания, вызываемые инструкцией INT для доступа к функциям системы. Обработчики прерываний в разных устройствах предоставлялись BIOS, операционной

системой (системные вызовы), драйверами. В реальном режиме процессора любая программа могла осуществлять обработку программных прерываний. В настоящее время это позволено делать только компонентам операционной системы.

Системные вызовы

Изначально для системных вызовов использовалась инструкция INT, однако в i586 появилась инструкция sysenter, а в 64-битных системах – уже syscall. Системные вызовы представляют собой механизм, похожий на прерывания, но в нём есть и свои отличия.

Языки программирования низкого уровня

Машинный код

Приведём пример машинного кода для x86, для ОС MS DOS. Формат исполняемого файла – COM (простой файл без заголовков, код помещается в память начиная с адреса 0x100):

BB 11 01 B9 0D 00 B4 0E 8A 07 43 CD 10 E2 F9 CD 20 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21

В машинном коде нет таких привычных вещей, как перевод строки и отступы. Это двоичный файл. Команды в нём имеют переменную длину, и удаление любого символа приведёт к тому, что код будет непредсказуемым.

В приведённом примере команды отделены друг от друга цветом. На самом деле никакого разделения нет: читая очередной байт, процессор в зависимости от содержания этого байта может прочитать ещё несколько байт. В целом аргументы команды не отделены от команды.

Для удобства запишем код построчно и разберём его:

```
BB 11 01
B9 0D 00
B4 0E
8A 07
43
CD 10
E2 F9
CD 20
48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21
```

- BB 11 01 – команда поместить число 0x0111 (шестнадцатеричное, little-endian) в регистр BX. Это смещение строки «Hello, world!».
- B9 0D 00 – команда поместить число 0x00D (шестнадцатеричное, little-endian) в регистр CX. Это длина строки «Hello, world!».

- B4 0E – поместить число 0x0E в регистра AH. Это номер запроса, который мы будем отправлять операционной системе (печать на экран).
- 8A 07 – поместить в регистр AL значение из памяти, адрес которой находится в регистре BX. Это второй аргумент для системного вызова – адрес печатаемого символа. Каждый раз мы будем переходить сюда.
- 43 – увеличить BX на 1. Это действие – для следующей итерации. Таким образом мы переходим к следующему байту строки.
- CD 10 – вызвать прерывание 0x10. Это обращение к ОС с запросом функции. В качестве аргументов вызова фигурируют регистр AH, содержащий номер функции 0x0E вызова 0x10, и содержащийся в регистре AL адрес очередного байта, включающего в себя печатаемый символ. Таким образом, мы печатаем по одному символу в позицию, где находится курсор. На самом деле 0x10 – это даже не функция MS DOS, а функция, реализованная в BIOS и представляющая собой пусть и медленный, но всё же доступ к памяти. Для доступа к многочисленным функциям MS DOS использовалось прерывание 0x21.
- E2 F9 – если CX не равно нулю, перейти на 7 байт назад, начиная с адреса следующей с E2 F9 команды. (F9 в десятичной системе счисления означает -7. Почему именно F9? Потому что речь идёт об отрицательном числе в дополнительном коде размером в один байт. Тем самым команда LOOP поддерживает переходы в диапазоне от -128 до 127).
- CD 20 – завершить выполнение программы, выгрузить ее из памяти и передать управление операционной системе.
- 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21 – Hello, world!

Мнемоники, опкоды, дизассемблер

На самом деле команды не имеют определенных имён – у них есть только номера кодов. Изначально процесс программирования выглядел так: программисты смотрели по справочнику номер команды и вписывали по очереди байты. Процессор до сих пор исполняет команды именно таким образом, однако разработка изменилась за счёт появления более высоких уровней абстракции (мнемоники, собственно ассемблеры, язык C, языки высокого уровня и т.д.)

Пока машинный код не сложен, номера команд можно даже помнить наизусть:

- CD 10 – вызвать прерывание 10.
- 43 – инкрементировать BX.

Впрочем, программистам всё равно приходилось пользоваться справочниками. Это тоже создавало неудобства, записи программ даже на бумаге были громоздкими, потому возникли мнемоники.

Ведь:

- INT 10 – нагляднее, чем CD 10.
- INC BX – нагляднее, чем 43.

Таким образом, в мнемониках код можно было бы записать так:

```
MOV BX, 0111
MOV CX, 000D
MOV AH, 0E
MOV AL, [BX]
INC BX
```

```
INT 10
LOOP -7
INT 20
Hello, world!
```

Такая запись получалась удобнее и производительнее, но переводить код всё равно приходилось по таблицам, кроме того, точно рассчитывая позиции в байтах.

Следующей итерацией в обеспечении удобства кодирования стали программы под названием Автокод и Монитор. Они позволяли вводить код в мнемониках, что по существу было продолжением программирования в машинных кодах. Безусловно, удобнее было бы ввести метки, автоматический расчёт адресов, куда необходимо сделать переход, добавить переменные, макросы и т.д. Такие программы появились, а сам язык, принадлежащий к более высокому уровню, чем машинный код или мнемоники, получил название Ассемблер.

Кроме того, существует и обратная ситуация, когда машинные коды необходимо перевести в мнемоники. Уровень подобного преобразования может быть разным – от простого проставления мнемоник до подсчёта точек вхождения и даже именования вызовов. Такая процедура называется дизассемблированием, близким к этому понятием является декомпиляция – перевод на более высокоуровневый язык.

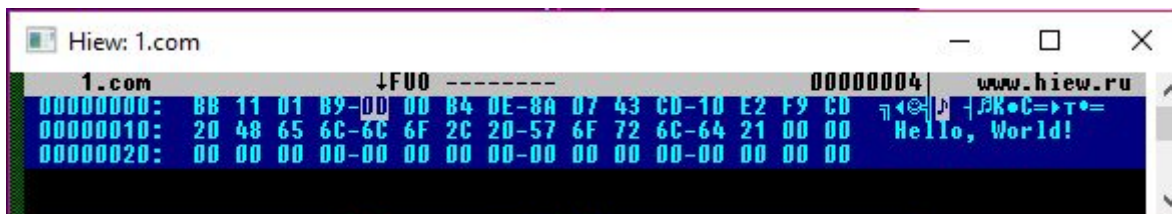
Рассмотрим дизассемблирование программы в Hiew.

[illegible]

Niew – довольно интеллектуальная программа: обратите внимание на указание ссылки в LOOP. Тем не менее 32-битная версия Niew уже не понимает формат COM и не опознает ссылку на «Hello,

world!» (далее увидим, что Hiew умеет делать это для MZ). Зато адрес LOOP рассчитан верно, потому что задан относительно текущей инструкции (в дополненном коде это отрицательное число).

А вот то, что Hiew пытается дизассемблировать и «Hello, world!», получая непредсказуемый код, – нормально. Более того, в общем случае невозможно определить, данные это или код.



Код в шестнадцатеричном виде

Анализ машинного кода и внесение изменений в машинный код применяются и сейчас – например, если в программе обнаружена ошибка, а исходных файлов нет. Иногда исправление ошибки осуществляется непосредственно в машинном коде – фактически это тоже своего рода программирование в машинных кодах. Несмотря на то, что в дизассемблерах доступны мнемоники, код инструкции приходится вводить непосредственно. Очень часто происходят замены JZ на JNZ (чтобы исправить ошибку в ветвлении), исправление адресов в JMP, CALL, замена JMP на NOP. Команда NOP оказывается особенно полезной, так как просто удалить, например, ошибочный JMP не получится: смещение даже одного байта приведёт в лучшем случае к изменению всех адресов (команды перехода не будут работать), а в худшем – к полной потере смысла в дальнейшем коде: с учётом переменной длины инструкций аргументы превратятся в совершенно случайные команды, а код станет непредсказуемым. В этом случае NOP используется в качестве заполнителя.

Форматы исполняемых файлов

Исполняемый файл хранится на диске.

.COM

.COM – простейший формат выполняемого файла, применяемый в операционных системах CP/M и DOS.

При старте программы операционная система выделяет сегмент памяти в первые 0x100 байт, после которых помещается исполняемый код. Поэтому первый байт адреса – 0x100, и этот факт должны учитывать дальнейшие переходы и ссылки. В остальном код помещается в область памяти без изменений. Размер программы ограничен размером сегмента за вычетом служебной области в 0x100 байт, то есть составляет не более 65280.

Такие файлы имели расширение .com и особой сигнатуры не имели. Позже в MS DOS появилась тенденция именовать MZ-файлы как .com, причём операционная система распознавала и выполняла их. Примером может быть command.com в последних версиях MS DOS, а Windows был уже MZ-файлом.

MZ и другие .exe

Данный формат использовался и используется в ОС OS/2, MS DOS, Windows и имеет расширение .exe. Для него существует несколько стандартов (MZ, NE, LE, PE и т.д.) Файл содержит несколько заголовков, первый из которых (за исключением формата MZ) – заглушка в формате MZ, выводящая в консоль сообщение, что программа не может быть исполнена в DOS. Интересно, что формат PE

является разновидностью формата COFF, изначально разработанного для Unix на смену формату a.out.

.ELF

Этот формат, популярный в Linux и некоторых других системах, был также разработан на смену a.out. Он может сводиться к одному из следующих типов:

1. Перемещаемый файл (relocatable file) хранит инструкции и данные, которые могут быть связаны с другими объектными файлами. Результатом такого связывания может быть исполняемый файл или разделяемый объектный файл.
2. Разделяемый объектный файл (shared object file) содержит инструкции и данные, но может быть использован двумя способами. В первом случае он может быть связан с другими перемещаемыми файлами и разделяемыми объектными файлами, в результате чего будет создан новый объектный файл. Во втором случае при запуске программы на выполнение операционная система может динамически связать его с исполняемым файлом программы, в результате чего будет создан исполняемый образ программы. В этом случае речь идет о разделяемых библиотеках.
3. Исполняемый файл хранит полное описание, позволяющее системе создать образ процесса. Он содержит инструкции, данные, описание необходимых разделяемых объектных файлов, а также необходимую символьную и отладочную информацию.

Си

Си — язык, занимающий промежуточное положение между языками высокого и низкого уровня. Несмотря на более привычные записи арифметических операций и алгоритмических конструкций, нежели в Ассемблере, Си — язык довольно низкоуровневый, требующий аккуратной работы с памятью и переменными. Ошибка при написании программы в Си может приводить к неверному завершению программы (например, наиболее частая ошибка — Segmentation fault — влечёт за собой попытку обращения к не своей памяти), а для ядра или модуля ядра — к Kernel panic.

Ассемблер и Си — два языка, используемые для программирования не только программ и компонентов операционной системы, но и микроконтроллеров. При этом в работе с микроконтроллерами, как правило, необходимо ознакомиться с Data sheet на соответствующее устройство и, даже программируя на Си, напрямую работать с портами ввода-вывода, памятью и т.д.

Forth

Пример ещё одного низкоуровневого языка программирования — Форт. Этот язык ориентирован на использование стека, постфиксная (обратная польская) запись для него так же непривычна, как выполнение арифметических операций для ассемблера. Существуют форт-процессоры, в случае которых форт используется как встраиваемый язык. На базе Forth был разработан PostScript, фактически встроенный язык во многих современных принтерах и МФУ.

Пример сложения 2+3 и вывода на экран:

```
2 3 + .
```

Пример расчета факториала:

```
: FACTORIAL recursive
dup 1 > if
  dup 1 - FACTORIAL *
```

```

else
    drop 1
endif ;

: OURLOOP
swap 1 + swap
do
    i . i ." ! = " i FACTORIAL . cr
loop ;

```

Запуск:

```
17 0 Ourloop
```

Пример реализации задачи Ханойской башни можно посмотреть по адресу: <http://www.kernelthread.com/projects/hanoi/html/4th.html>.

Ассемблер

INTEL-запись

С INTEL-записью мы уже знакомы, так как она используется для дизассемблеров. Регистры в ней называются AX, BX и т.д., порядок операндов следующий: КОМАНДА приемник, источник. Например: MOV AX, BX – переместить содержимое регистра BX в регистра AX.

Источником такой записи, как ясно из названия, является компания Intel.

На то, что берётся не значение регистра, а значение из адреса памяти в регистре, в записи указывается квадратными скобками.

```
MOV AX, [BX]
```

Числа в дизассемблере записываются изначально в шестнадцатеричном виде, в ассемблере – в виде 10h.

В качестве символа комментария используется «;»

AT&T-запись

В Unix и Linux применяется AT&T-синтаксис, и это не случайно, так как UNIX-подобные системы зародились в недрах AT&T.

Регистры в данном типе синтаксиса именуются следующим образом: %eax, %ebx.

Значение с префиксом \$xx означает число xx. Запись xx без префикса означает использование ячейки с адресом xx (в Intel-нотации это было бы [xx]).

К мнемоникам добавляются префиксы: b – byte, w – word, l – long (например: movb, movw).

Запись имеет вид КОМАНДА префикс источник, приемник – например: movl %ebx, %eax (в противовес MOV EAX, EBX в Intel-нотации).

Шестнадцатеричные числа имеют вид \$0x00.

Если работа идёт с ячейкой памяти, адрес которой содержится в регистре, регистр берётся в круглые, а не квадратные скобки – например: `movw (%bx), %ax`.

В качестве символа комментария используется `#`, а `;` служит для разделения нескольких команд.

Простейшие команды

Открыв в дизассемблере бинарный файл, вы можете увидеть команды `mov` и `int`, `call` и `ret`.

Что они означают?

- `mov` – поместить в регистр или память значение (либо записанное константой, либо взятое из другого регистра или из памяти);
- `int` – вызвать программное прерывание: указывается номер прерывания (эта команда похожа на вызов функции, аргументы передаются в регистрах и в них же возвращаются);
- `call` – вызвать функцию (подпрограмму): указывается адрес команды, на которую передаётся управление;
- `ret` – вернуться из функции (подпрограммы);
- `por` – ничего не сделать (заполнитель).

Простейшая программа

Простейшая программа на языке C выглядит так:

hello.c

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[])
{
    printf("Hello, world!\n");
    exit(0);
}
```

Уже из кода мы видим подключение библиотеки. На самом деле `printf` – это обращение не напрямую к операционной системе, а к библиотеке, то есть функция, которая уже обращается к операционной системе. В этой программе нам не нужны возможности форматирования `printf`, потому создадим программу печати на экран непосредственно на ассемблере, но прежде сделаем это и на C (что тоже возможно):

hello2.c

```
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char* argv[])
{
    char str[] = "Hello, world!\n";
    write(1, str, sizeof(str) - 1);
    exit(0);
}
```

Строки в C заканчиваются `\0`. Если `\0` перезаписать, мы получим непредсказуемый результат и вывод на экран тех байтов, которые следовали в памяти после ячейки, где хранился `\0`. Наверняка случайно где-то далее в области данных нуль всё-таки найдется, но на экран при этом будет выведен объём

нечитаемого мусора. Поэтому при вызове write, указывая длину строки, мы уменьшаем ее на 1 (на тот самый нулевой байт).

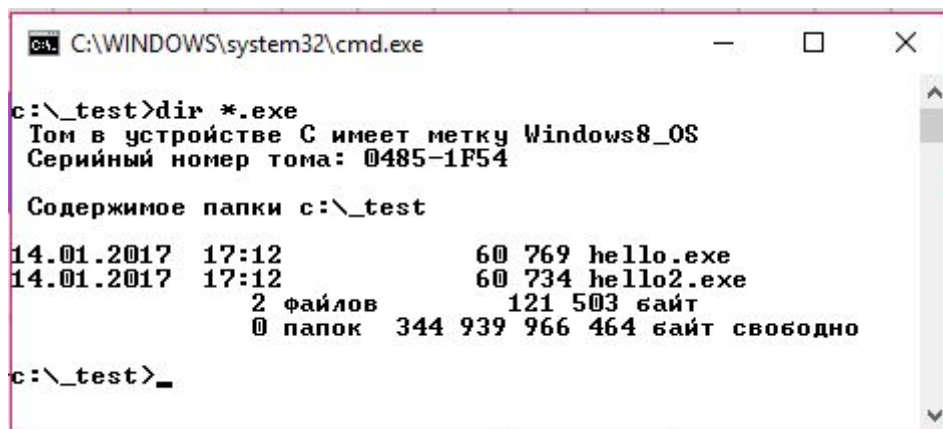
Обратите внимание, что в данном случае происходит запись в файл. Стандартный выходной поток – stdout – также является файлом с файловым дескриптором 1, который уже открыт операционной системой при старте программы.

Обе программы можно скомпилировать и в Windows с помощью gcc из ПО MinGW.

```
c:\>cd c:\MinGW\bin
c:\>gcc c:\tmp\hello.c -o c:\tmp\hello
c:\>c:\tmp\hello
Hello, world!
```

Кстати, расширение .exe компилятор добавит сам.

Оба файла мало отличаются по размеру.



```
C:\WINDOWS\system32\cmd.exe

c:\_test>dir *.exe
Том в устройстве C имеет метку Windows8_OS
Серийный номер тома: 0485-1F54

Содержимое папки c:\_test

14.01.2017  17:12                60 769 hello.exe
14.01.2017  17:12                60 734 hello2.exe
                2 файлов                121 503 байт
                0 папок   344 939 966 464 байт свободно

c:\_test>_
```

Посмотрим содержимое в Hiew. Нажимаем F4 и для начала hex.

Давайте посмотрим, как выглядит код в `hello2`, без применения библиотеки.

[illegible]

Код без дизассемблера:

Найдите здесь «Hello, world!»

Теперь скомпилируем те же программы в GNU/Linux.

```
$ nano hello.c
$ gcc hello.c -o hello
$ hello ./hello
Hello, world!
$
```

Здесь исполняемый файл не имеет разрешения .exe, зато права на исполнение (chmod +x hello) назначать не понадобится: в данной операционной системе компилятор сделал это сам, точно так же, как в Windows добавил расширение .exe, чтобы файл мог исполняться.

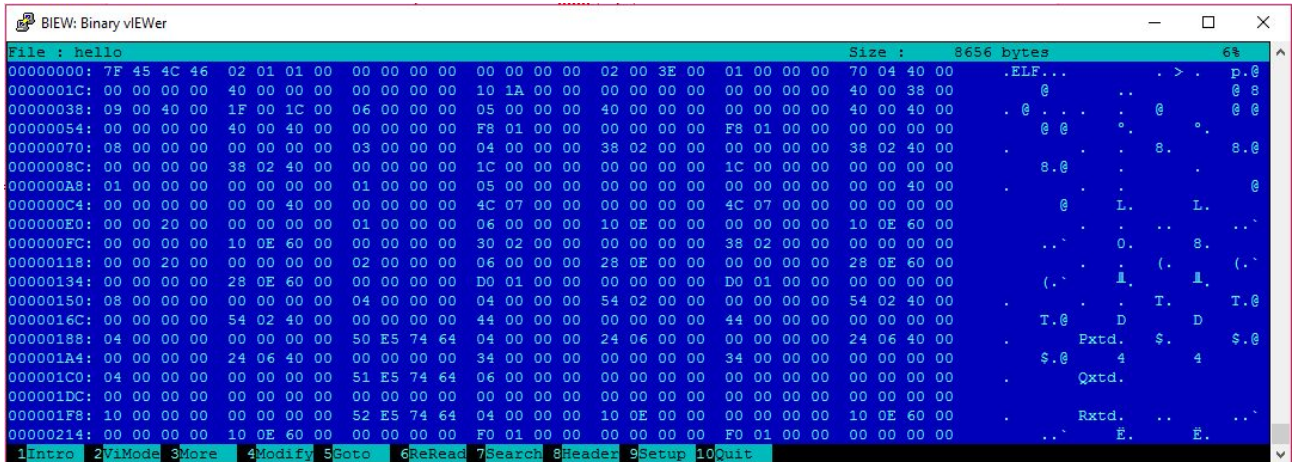
Так как hiew работает под Windows и понимает файлы MZ, но не понимает формат ELF, запускать под Wine его смысла нет. Поэтому нам понадобится biew (beye).

```
$ lynx https://sourceforge.net/projects/beye/files/latest/download
$ tar xvfj biew-610-src.tar.gz
$ cd biew-610
$ ./configure
$ sudo make
$ sudo make install
```

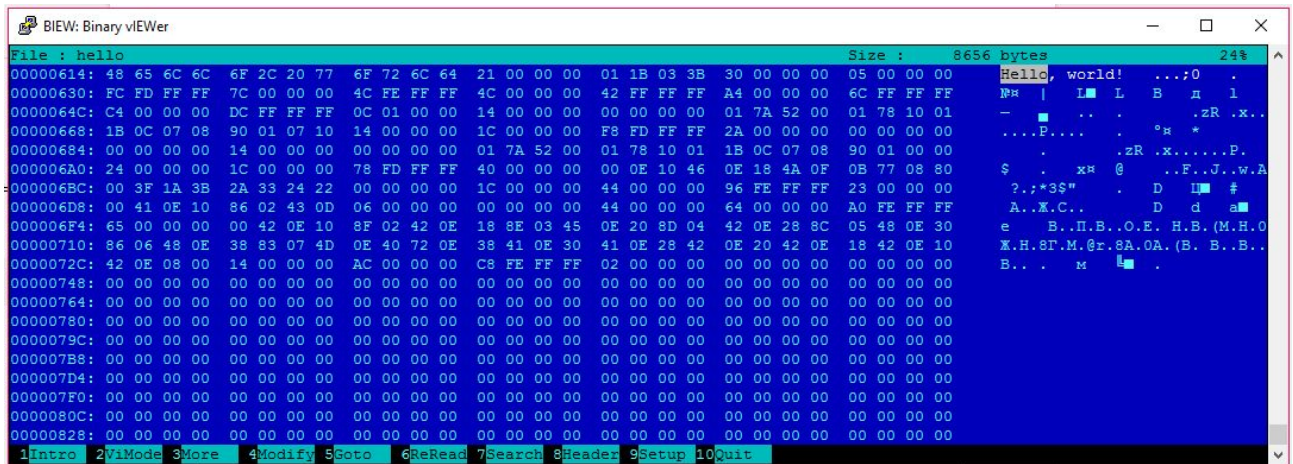
Запускаем:

```
$ biew -h hello
```

Перед нами совсем другой формат (сигнатура ELF).



Через F7 найдем «Hello, world!»

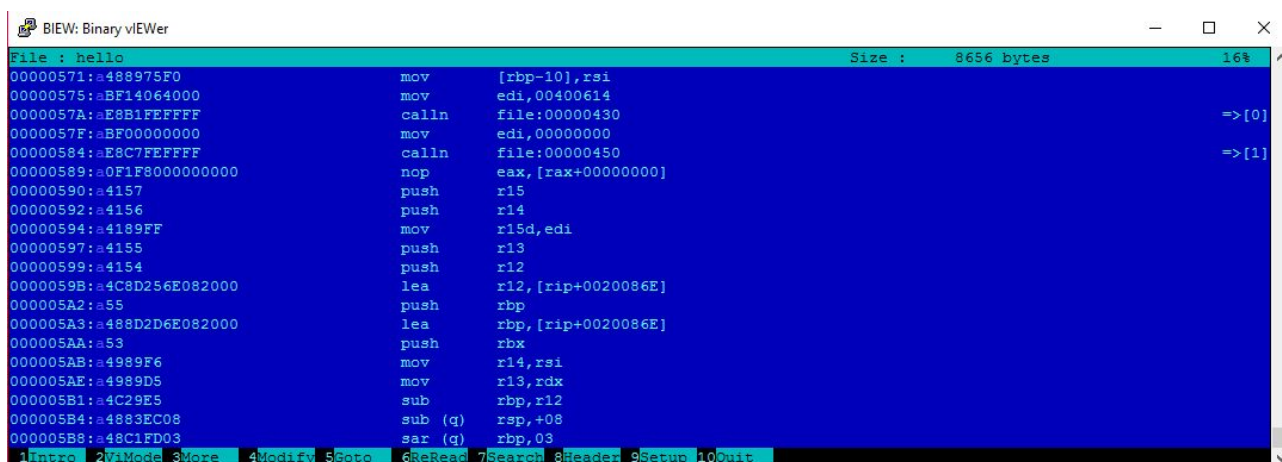


Нажимаем F2 и выбираем Disassembler или сразу запускаем:

```
$ biew -d hello
```

Невероятно похоже на hiew, не правда ли?

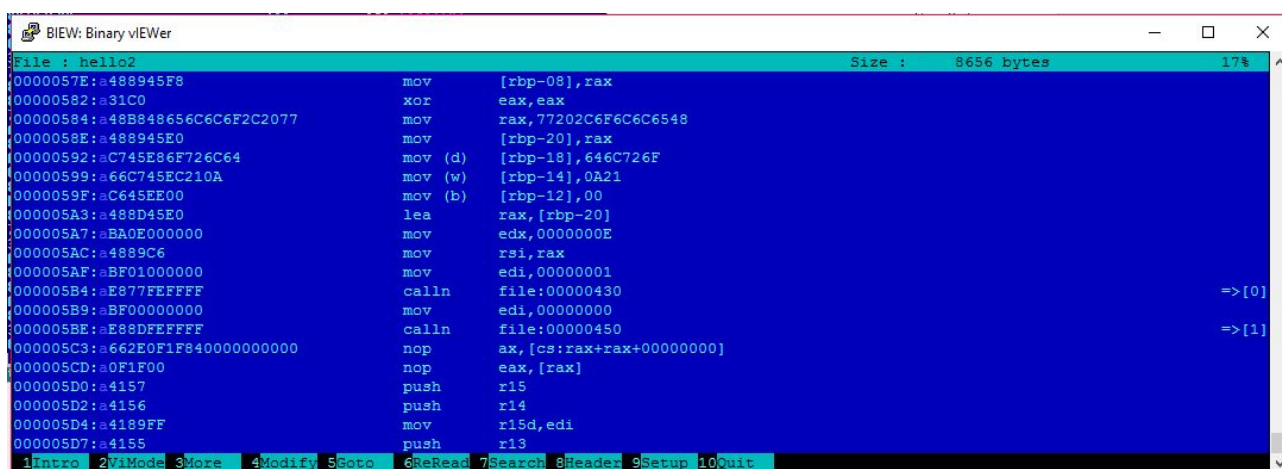
Хотя hiew и более дружелюбный, старые версии выглядели так же. Чтобы перейти в начало исполняемой части, нажимаем F8 и Enter – и мы находим наш адрес. Он не полностью совпадает, но младшая часть у него такая же. В целом код получается схожий. Помещаем в регистр edi адрес строки, вызываем функцию. Обнуляем регистр и вызываем функцию. Это и есть наши печать и exit.



Теперь откомпилируем и посмотрим hello2.c и сразу же запустим его в дизассемблере:

```
$ nano hello2.c
$ gcc hello2.c -o hello2
$ hello ./hello2
Hello, world!
$ biew -h hello2
```

Переходим в начало исполняемого кода:F8 и Enter:



Похожая картина?

Для надёжности попробуем через F7 (сразу здесь, или в Hex-режиме) найти «Hello».

Вот что мы видим:

Точно так же частями загружается фраза «Hello, world!»: часть через регистр `rax`, часть в ссылке, находящуюся в регистре `rbp` со смещением. Присутствуют два вызова `call` для печати и выхода.

Сравним размер файлов:

Теперь попробуем создать аналог `hello2` на Ассемблере (в стандарте AT&T).

hello3.s

```
.data                                /* поместить следующее в сегмент
данных */

hello_str:                          /* наша строка
*/
    .string "Hello, world!\n"

/*
    */                                /* длина строки
*/
    .set hello_str_length, . - hello_str - 1

.text                                /* поместить следующее в сегмент кода
*/

.globl main                          /* main - глобальный символ, видимый за
пределами текущего файла */
.type main, @function               /* main - функция (а не данные) */
main:
    movl    $4, %eax                /* поместить номер системного вызова write = 4 в
регистр %eax */
    movl    $1, %ebx                /* первый параметр - в регистр %ebx; номер файлового
дескриптора stdout - 1 */
    movl    $hello_str, %ecx        /* второй параметр - в регистр %ecx; указатель на
строку */
    movl    $hello_str_length, %edx /* третий параметр - в регистр %edx; длина
строки */
    int     $0x80                   /* вызвать прерывание 0x80 */
    movl    $1, %eax                /* номер системного вызова exit - 1 */
    movl    $0, %ebx                /* передать 0 как значение параметра */
    int     $0x80                   /* вызвать exit(0) */
    .size   main, . - main          /* размер функции main */
```

В EBX мы поместили дескриптор (идентификатор) консоли – stdout. В ECX и EDX содержатся адрес начала сообщения (адрес первого байта) и длина сообщения в байтах. Таким образом, этот системный вызов должен выполнить вывод строки, находящейся по адресу msg, на консоль.

- `mov eax, 1` – в EAX помещается 1 – номер системного вызова `exit`,
- `mov ebx, 0` – в EBX помещается 0 – параметр вызова `exit` означает код, с которым завершится выполнение программы,
- `int 0x80` – системный вызов. После системного вызова `exit` выполнение программы завершается.

Компилируем, запускаем и сразу в дизассемблер:

```
$ nano hello3.s
$ gcc hello3.c -o hello3
$ hello ./hello3
Hello, world!
$ biew -d hello3
```

F8 и Enter. Теперь наш код выглядит так:

BIEW: Binary vIEWer

File : hello3

```

000004D6:aB804000000    mov     eax,00000004
000004DB:aBB01000000    mov     ebx,00000001
000004E0:aB930106000    mov     ecx,00601030
000004E5:aBA0E000000    mov     edx,0000000E
000004EA:aCD80      int     80
000004EC:aB801000000    mov     eax,00000001
000004F1:aBB00000000    mov     ebx,00000000
000004F6:aCD80      int     80
000004F8:a0F1F84000000000    nop     eax,[rax+rax+00000000]
00000500:a4157      push    r15
00000502:a4156      push    r14
00000504:a4189FF     mov     r15d,edi
00000507:a4155      push    r13
00000509:a4154      push    r12
0000050B:a4C8D25FE082000    lea     r12,[rip+002008FE]
00000512:a55       push    rbp
00000513:a488D2DFE082000    lea     rbp,[rip+002008FE]
0000051A:a53       push    rbx
0000051B:a4989F6     mov     r14,rsi
1Intro 2ViMode 3More 4Modify 5Goto 6ReRead 7Search 8Header 9Setup 10Quit

```

Знакомые строки?

А вот и наша строка.

BIEW: Binary vIEWer

File : hello3

Size : 8656 bytes

54%

```

00001030: 48 65 6C 6C 6F 2C 20 77 6F 72 6C 64 21 0A 00 47 43 43 3A 20 28 55 62 75 6E 74 75 20
0000104C: 35 2E 34 2E 30 2D 36 75 62 75 6E 74 75 31 7E 31 36 2E 30 34 2E 34 29 20 35 2E 34 2E
00001068: 30 20 32 30 31 36 30 36 30 39 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00001084: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 02 01 00 38 02 40 00 00 00 00 00
000010A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 02 00 00 54 02 40 00 00 00 00 00
000010BC: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 74 02 40 00 00 00 00 00 00 00 00 00
000010D8: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 98 02 40 00 00 00 00 00 00 00 00 00
000010F4: 03 00 05 00 B8 02 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 06 00
00001110: 00 03 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 07 00 38 03 40 00
0000112C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 08 00 40 03 40 00 00
00001148: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 09 00 60 03 40 00 00 00 00 00 00
00001164: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 78 03 40 00 00 00 00 00 00 00 00 00
00001180: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 90 03 40 00 00 00 00 00 00 00 00 00
0000119C: 03 00 0C 00 B0 03 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 0D 00
000011B8: D0 03 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 0E 00 E0
000011D4: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 00 0F 00 74 05 40 00
000011F0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 10 00 80 05 40 00 00 00 00 00 00
0000120C: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 11 00 84 05 40 00 00 00 00 00 00
00001228: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 03 12 00 B0 05 40 00 00 00 00 00 00 00

```

1Intro 2ViMode 3More 4Modify 5Goto 6ReRead 7Search 8Header 9Setup 10Quit

Осталось сравнить размеры файлов:

```

oga@uho: ~
oga@uho:~$ ls -la | grep hello
-rwxrwxr-x 1 oga oga 8656 янв 14 18:59 hello
-rwxrwxr-x 1 oga oga 8656 янв 14 19:18 hello2
-rw-rw-r-- 1 oga oga 161 янв 14 19:16 hello2.c
-rwxrwxr-x 1 oga oga 8656 янв 14 19:36 hello3
-rw-rw-r-- 1 oga oga 2163 янв 14 19:36 hello3.s
-rw-rw-r-- 1 oga oga 120 янв 14 18:14 hello.c
oga@uho:~$

```


Размер файлов одинаковый.

В этом месте может возникнуть вопрос: где же здесь обращение к операционной системе? Разве мы работаем не напрямую с процессором?

В данном случае задействуется вызов прерывания 0x80. В Linux данное прерывание является шлюзом для системных вызовов (в Windows это 0x2e). Таким образом, если скомпилировать C-код, приведённый в самом первом примере, то работать он будет, а вот код на ассемблере уже ОС-независимым не окажется. Впрочем, зная системные вызовы соответствующей операционной системы, его (пока он ещё небольшой) не сложно переделать.

Программа вычисления факториала

Вычислять факториал будем рекурсивно.

Команды, которые нам понадобятся:

- `push` – поместить регистр в стек;
- `pop` – извлечь регистр из стека;
- `jmp` – безусловный переход;
- `dec` – уменьшить на единицу (противоположность `inc`);
- `add` – сложение;
- `mul` – умножение;
- `movl 8(%ebp), %eax` – адресация со сдвигом. В `%eax` поместить значение из памяти, находящейся по адресу `EBP+8`. Аналог в Intel записи: `MOV EAX, [EBP+8]`.

Также обратите внимание, что `printf` мы вызываем из библиотеки `libc` через `call`.

```

.data
printf_format:
.string "%d\n"

.text
/* int factorial(int) */
factorial:
    pushl %ebp
    movl %esp, %ebp
    /* извлечь аргумент в %eax */
    movl 8(%ebp), %eax
    /* факториал 0 равен 1 */
    cmpl $0, %eax
    jne not_zero
    movl $1, %eax
    jmp return

not_zero:
    /* следующие 4 строки вычисляют выражение %eax = factorial(%eax - 1) */
    decl %eax
    pushl %eax
    call factorial
    addl $4, %esp
    /* извлечь в %ebx аргумент и вычислить %eax = %eax * %ebx */
    movl 8(%ebp), %ebx
    mull %ebx
    /* результат в паре %edx:%eax, но старшие 32 бита нужно отбросить, так
как они не помещаются в int */
return:
    movl %ebp, %esp
    popl %ebp
    ret

.globl main
main:
    pushl %ebp
    movl %esp, %ebp
    pushl $5
    call factorial
    pushl %eax
    pushl $printf_format
    call printf
    /* стек можно не выравнивать, это будет сделано во время выполнения
эпилога */
    movl $0, %eax
    /* завершить программу */
    movl %ebp, %esp
    popl %ebp
    ret

```

Программа выводит таблицу умножения:

```
.data
input_prompt:
    .string "enter size (1-255): "
scanf_format:
    .string "%d"
printf_format:
    .string "%5d "
printf_newline:
    .string "\n"
size:
    .long 0
.text
.globl main
main:
    /* запросить у пользователя размер таблицы */
    pushl $input_prompt    /* format */
    call printf            /* вызов printf */
    /* считать размер таблицы в переменную size */
    pushl $size            /* указатель на переменную size */
    pushl $scanf_format    /* format*/
    call scanf             /* вызов scanf */
    addl $12, %esp         /* выровнять стек одной командой сразу после
двух функций */
    movl $0, %eax          /* в регистре %ax команда mulb будет выдавать
результат, но мы печатаем всё содержимое %eax, поэтому два старших байта %eax
должны быть нулевыми */
    movl $0, %ebx          /* номер строки */
print_line:
    incl %ebx              /* увеличить номер строки на 1 */
    cmpl size, %ebx
    ja print_line_end     /* если номер строки больше запрошенного
размера, завершить цикл */
    movl $0, %ecx          /* номер колонки */
print_num:
    incl %ecx              /* увеличить номер колонки на 1 */
    cmpl size, %ecx
    ja print_num_end      /* если номер колонки больше запрошенного
размера, завершить цикл*/
    movb %bl, %al          /* команда mulb ожидает второй операнд в %al */
    mulb %cl               /* вычислить %ax = %cl * %al */
    pushl %ebx             /* сохранить используемые регистры перед
вызовом printf*/
    pushl %ecx
    pushl %eax             /* данные для печати */
    pushl $printf_format  /* format */
    call printf            /* вызов printf */
    addl $8, %esp          /* выровнять стек */
    popl %ecx              /* восстановить регистры */
    popl %ebx
    jmp print_num          /* перейти в начало цикла */
print_num_end:
    pushl %ebx             /* сохранить регистр */
    pushl $printf_newline /* напечатать символ новой строки */
    call printf
    addl $4, %esp
    popl %ebx              /* восстановить регистр */
    jmp print_line        /* перейти в начало цикла */
```

```
print_line_end:
    movl    $0, %eax          /* завершить программу */
    ret
```

Операционные системы: функции, задачи

Может ли программа работать без операционной системы? Может. Так обстоит дело в микроконтроллерах (обычно говорят о прошивке), и получается узкая в применении система. Если же требуется выполнение разных программ, решение разных задач, возникает необходимость не только в запуске программ, но и в управлении ими. Необходимо реализовать такие функции, как загрузка/выгрузка программ, обработка ошибок, выделение ресурсов, предоставление доступа к устройствам, предоставление доступа к стандартным функциям, чтобы программы на низком уровне не реализовывали одни и те же действия самостоятельно. Кроме того, операционные системы решают задачи межпроцессного и межсетевого взаимодействия.

В состав операционной системы, как правило, входит следующее:

- ядро;
- драйвера;
- сервисы (демоны);
- системные библиотеки;
- оболочка;
- утилиты;
- средства разработки.

Ядро – это основа операционной системы. Именно на уровне ядра решаются вопросы взаимодействия программного и аппаратного обеспечения, доступа к файлам и сети, межпроцессное взаимодействие, управление задачами и распараллеливание.

Драйвера, как правило, являются независимыми компонентами, но в зависимости от операционной системы они могут быть реализованы как сервисы или как компоненты ядра.

Сервисы (демоны) – постоянно выполняющиеся службы, обслуживающие операционную систему.

Системные библиотеки – компоненты, используемые разными программами и динамически подключаемые при выполнении (DLL, .so).

Оболочка – основной механизм взаимодействия с пользователем. Это может быть интерпретатор командной строки (sh, bash, command.com) или графическая среда (explorer для Windows, X11-сервер и соответствующая среда рабочего стола для Linux).

Утилиты – набор инструментов, запускаемых в скриптах, из командной строки или даже из графического режима для управления операционной системой.

Средства разработки – компиляторы и файлы библиотек, необходимые для компиляции и/или разработки для данной операционной системы.

Системные вызовы

Как правило, ядро и остальные программы выполняются в разных окружениях. Ошибка в ядре приводит к краху системы (BSOD для Windows, Kernel panic для Linux). Ошибка в пользовательской программе приводит к снятию программы. (Если ядро и программы выполняются в одном окружении, как например в MS DOS, ошибка в программе может привести к зависанию всей системы). Для обмена между ядром и программами предусмотрен интерфейс – системные вызовы.

Как уже выше было указано, классический вариант обращения приложения к ядру с вызовом функции выполнялся через прерывание. В настоящее время в ядре Linux используется другой, хотя и похожий подход – обращение через VDSO (Virtual Dynamically Shared Object) и машинные инструкции sysenter для 32-битных систем и syscall для 64-битных (подробнее: <http://rflinux.blogspot.ru/2008/03/linux-syscalls-linux.html>, <http://eax.me/linux-assembler/>).

По существу, sysenter и syscall аналогичны вызову соответствующего прерывания, однако некоторые действия, которые при вызове прерывания происходят автоматически (сохранение значений адреса возврата и регистров в стеке), теперь необходимо делать вручную.

Sysenter

```
.data
msg:
    .ascii "Hello, world!\n"
    len = . - msg
.text
.globl _start
_start:
    # write
    mov    $4,    %eax
    mov    $1,    %ebx
    mov    $msg,  %ecx
    mov    $len,  %edx
    push   $write_ret
    push   %ecx
    push   %edx
    push   %ebp
    mov    %esp, %ebp
    sysenter
write_ret:
    # exit
    mov    $1,    %eax
    xor    %ebx, %ebx
    push   $exit_ret
    push   %ecx
    push   %edx
    push   %ebp
    mov    %esp, %ebp
    sysenter
exit_ret:
```

Syscall (для 64-битных систем)

```
.data
msg:
    .ascii "Hello, world!\n"
    .set len, . - msg
.text
.globl _start
_start:
    # write
    mov $1, %rax
    mov $1, %rdi
    mov $msg, %rsi
    mov $len, %rdx
    syscall
    # exit
    mov $60, %rax
    xor %rdi, %rdi
    syscall
```

Здесь действует тот же принцип, но есть важные отличия: номера системных вызовов нужно брать из `unistd_64.h`, а не из `unistd_32.h` – они совершенно другие. Так как это 64-битный код, то и регистры в нём используются 64-битные. Номер системного вызова помещается в `rax`. До шести аргументов передаётся через регистры `rdi`, `rsi`, `rdx`, `r10`, `r8` и `r9`. Возвращаемое значение помещается в регистр `rax`. Значения, сохранённые в остальных регистрах, при возвращении из системного вызова остаются прежними, за исключением регистров `gsx` и `r11`.

Классификация операционных систем

Операционные системы (ОС) могут классифицироваться по-разному:

- однозадачные и многозадачные;
- однопользовательские и многопользовательские;
- с поддержкой сети и без неё;
- системы с графическим интерфейсом (встроенным в ядро, реализованным как отдельная служба) или поддерживающие только консольный интерфейс;
- универсальные или специализированного назначения (используемые в качестве прошивки для определенного оборудования).

Отдельно стоит выделить операционные системы реального времени, ориентированные на предоставление ресурсов в рамках требуемого временного ограничения. Кроме того, существуют операционные системы, реализующие на своём уровне гипервизор для гостевых операционных систем.

ОС могут также разделяться на проприетарные и свободные, классифицироваться по происхождению (OS/2, Unix, GNU/Linux и т.д.) и по соответствию POSIX. POSIX – стандарт ISO/IEC 9945:2009, описывающий взаимодействие между ОС и приложением, т.е. системные API, библиотеку языка C, набор приложений и интерфейсов. Он обеспечивает совместимость с UNIX-операционными системами.

История операционных систем

Рассмотрим некоторые известные операционные системы.

CP/M – однозадачная дисковая ОС, написанная в 1973 году Гэри Килдаллом (Gary Kildall) на языке программирования PL/M (Programming Language for Microcomputers). Широко применялась на 8-битных компьютерах.

MS DOS – 16-разрядный клон CP/M, однозадачная дисковая ОС. Изначально она называлась QDOS (англ. Quick and Dirty Operating System – быстрая и грязная операционная система), затем получила название 86-DOS, была куплена Microsoft и основательно доработана.

Windows 3.0 – полноценная операционная система, использующая собственное ядро и системные вызовы. Тем не менее DOS служил для неё загрузчиком и консольным режимом, пригодным для операций восстановления. Та же ситуация повторилась с Windows 95 и Windows 98, хотя переход от DOS к Windows был более сглаженным.

OS/2 – система, которая разрабатывалась IBM при участии Microsoft. Одной из её задач была конкуренция с DESQView – многозадачной надстройкой для DOS. Сначала система была текстовой, затем у неё появился графический интерфейс. Версия OS/2 2.0 (1992) могла запускать DOS и Windows 3.11 приложения. Выполнение Win-приложений было лицензировано у Microsoft.

Windows NT является преемником не Windows 3, а OS/2. После прекращения сотрудничества с IBM в Microsoft продолжили разрабатывать свою версию OS/2, дальнейшее развитие которой привело к созданию Windows 2000, Windows XP и т.д. Изначально система называлась NT OS/2. Что интересно, причиной разногласий, а потом и разрыва IBM и Microsoft стало стремление последних добавить в OS/2 поддержку WinAPI (изначально в OS/2 планировалась поддержка собственного API, а потом POSIX).

Предысторию системы UNIX мы уже рассматривали.

BSD (англ. Berkeley Software Distribution) – созданная в университетской среде, данная ОС являлась разновидностью UNIX и распространялась свободно. Её известными потомками являются FreeBSD, OpenBSD и MAC OS X.

Система GNU/Linux в представлении не нуждается: это серия Unix-подобных ОС, состоящих из ядра Linux и окружения GNU (успешная попытка воспроизведения окружения UNIX на основе GNU GPL принципов). Отдельно стоит отметить связанные с GNU и Linux ядра ОС. Спор Эндрю Таненбаума с Линусом Торвалдсом по поводу архитектуры ядра и привел к созданию ядра Linux. При этом сам Линус использовал в качестве среды разработки Minix – учебную операционную систему, созданную Таненбаумом. Кроме того, у GNU имелась собственная попытка разработки ядра GNU/Hurd – из-за успешной разработки Linux процесс вялотекущий и, вероятно, бесперспективный. Тем не менее такое ядро было выполнено, причём в сборке Debian/kHurd (впрочем, как и Debian/kFreeBSD – на основе ядра FreeBSD).

Практическое задание

При работе над практическим заданием для части заданий понадобится Linux. Рекомендуется установить виртуальную машину VirtualBox или VMWare Player и, скачав образ, например, Ubuntu, установить операционную систему. Часть работ будут выполняться в ОС Linux.

1. Какие минусы у реле?
2. Какие плюсы у транзисторов?
3. Что такое порядок байт? Какой порядок байт в intel?
4. Что такое прерывания?

Дополнительные материалы

1. <http://blog.nintech.net/anatomy-of-the-eicar-antivirus-test-file/>
2. <http://rus-linux.net/MyLDP/algol/asm-lin.html>
3. https://ru.wikibooks.org/wiki/Ассемблер_в_Linux_для_программистов_C
4. <http://nongnu.askapache.com/pgubook/ProgrammingGroundUp-1-0-booksize.pdf>
5. MICROPROCESSORS: THE 8086/8088, 80186/80286, 80386/80486 AND THE PENTIUM FAMILY
Авторы: NILESH B. BAHADURE
https://books.google.ru/books?id=QZ9nM0nRQcYC&pg=PA625&lpg=PA625&dq=BPH+BPL+%D0%B2+80386&source=bl&ots=Vq4rRXtOiO&sig=b3FkT0zJadtMAhSUV-8zJzAgY8U&hl=ru&sa=X&ved=0ahUKEwj1_K7ycHRAhVDP5oKHx4iBwIQ6AEIJDAAB#v=onepage&q=BPH%20BPL%20%D0%B2%2080386&f=false
6. <https://habrahabr.ru/post/146080/>
7. <http://asmworld.ru/spravochnik-komand/>
8. http://rjaan.narod.ru/docs/gnu/cs/assembler/gnu-assembler.html#mov_ins
9. <https://habrahabr.ru/post/208176/>
10. <http://rflinux.blogspot.ru/2008/03/linux-syscalls-linux.html>
11. <http://www.programmersclub.ru/assembler/>
12. <http://mar.ugatu.su/assets/files/Arc/lab/asm.pdf>

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Разомнем мозг при помощи Forth? <https://habrahabr.ru/post/159351/>
2. ASM в Unix <https://habrahabr.ru/post/146080/>
3. <https://unixforum.org/index.php?showtopic=209>
4. https://ru.wikipedia.org/wiki/%D0%9C%D0%B0%D1%88%D0%B8%D0%BD%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4
5. <http://chernykh.net/content/view/230/245/>
6. http://www.4004.com/assets/Busicom-141PF-Calculator_asm_rel-1-0-0.txt
7. <https://habrahabr.ru/post/233245/>
8. <https://ru.wikipedia.org/wiki/Multics>
9. <http://rflinux.blogspot.ru/2008/03/linux-syscalls-linux.html>
10. <http://eax.me/linux-assembler/>
11. <http://ctfhacker.com/ctf/pwnable/2015/08/13/defcon-openctf-sigil.html>