Информатика. Семинары 1-й курс 2-й семестр. Введение в архитектуру ЭВМ и программирование на языке ассемблера.

Бабичев С. Л.

21 марта 2021 г.

Содержание

Здесь размещены материалы, используемые для занятий по курсу информатики 2-го семестра обучения МФТИ.

Введение

Второй семестр — изучение архитектуры компьютера на более глубоком уровне, чем раньше. Нам придётся познакомиться с составными частями того, что мы называем компьютером.

1 Основные понятия курса

1.1 Цели курса

- Дать представление об архитектуре современных ЭВМ.
- Понять, каким образом то, что мы пишем на каком-либо языке программирования, превращается в исполнимую форму и затем исполняется.
- Узнать о представлении информации в ЭВМ и способах её обработки.
- Узнать, что такое машинный код и какими возможностями он обладает.

1.2 Что нам предстоит на семинарах

- немного изучать теорию по архитектуре ЭВМ;
- писать программы на языке `Си`, иллюстрирующие некоторые положения;
- писать программы на языке ассемблера, реализующие простые алгоритмы;
- восстанавливать код ня языке *Си* по файлу в машинных кодах — объектному и исполнимому.

2 Архитектора ЭВМ

У любого алгоритма должен быть исполнитель. ЭВМ предоставляет такого исполнителя: *процессор*. Отдельные шаги алгоритма есть *машинные команды*, которые хранятся в *оперативной памяти*, извлекаются из неё процессором и исполняются.

Большинство машинных команд имеет *операнды*. Они могут либо размещаться в той же самой оперативной памяти, либо находиться внутри процессора.

Нужно отличать память и процессор — это два совершенно различных устройства, хотя они не могут работать друг без друга.

Операнды машинных команд, находящиеся внутри процессора, называются *регистрами процессора*. Для того, чтобы в регистре что-нибудь оказалось, нужно либо *загрузить* значение регистра из оперативной памяти, либо получить его из других регистров, исполнив *машинную команду*.

Регистры не бесконечны и содержат строго определённое количество битов. Существует несколько режимов работы процессора, в каждом из которых регистры содержат различное количество битов. В зависимости от этого, программы, исполняемые на процессоре делятся на 16, 32 и 64-разрядные. Обычно чем больше битов в регистре, тем скорость исполнения одних и тех же алгоритмов выше.

Память и процессор связаны между собой устройством *шина*. Процессор для получения значения *ячейки памяти* посылает шине *запрос*, который содержит *режим доступа* (чтение или запись) и *адрес*. Каждый *байт* оперативной памяти имеет свой *адрес*, который однозначно описывает данные.

2.1 Представление чисел

Числа представляются в двоичном коде. По теории информации (скорее, по комбинаторике), существует ровно 2^N наборов из N битов. Каждый из таких наборов кодирует ровно одно число.

Напоминаем, что термин число применяется к определению счётного множества предметов. Сами числа системы счисления не имеют. Каким бы образом мы не считали количество овец в стаде, оно останется неизменным. Однако, это число может иметь различное *представление*, вот здесь уже и важно понятие *систем счисления*. Число, соответствующее пальцам на десяти руках может записываться или как 50 (мы подразумеваем десятичную систему счисления), либо как 110010, тогда мы имеем в виду двоичную систему счисления и записываем число как 110010_2, либо как `L` — что в своё время делали римляне.

Для вычисления с помощью автоматических устройств, таких, как процессор, наиболее удобным оказалось двоичное представление чисел. Биты можно кодировать, например, наличием или отсутствием заряда/напряжения/тока, их легко складывать и умножать.

2.2 Дополнительный код

Данные, которые находятся в оперативной памяти, обрабатывать нельзя, там они только хранятся[1]. Для обработки данных они должны быть пересланы в процессор, точнее, в какой-либо из регистров процессора. Как мы уже знаем,

[1] устаревшие ныне модели процессоров могли производить операции и над ячейками памяти. Почему это теперь не применяется, мы узнаем позже

количество битов в регистре ограничено. Обычно это числа, кратные степеням двойки — 8, 16, 32, 64.

Предположим пока, что количество битов в регистре равно 16. В этом случае можно закодировать $2^{16} = 65536$ различных чисел. Например, можно закодировать числа в диапазоне от 0 до 65535 включительно. В языках программирования такие неотрицательные представления чисел называются *беззнаковыми* или *unsigned*.

Перед тем как понять, каким образом кодировать отрицательные числа (кроме знака, с ним мы уже определились), сделаем небольшое отступление.

Предположим, что мы используем беззнаковые числа длиной 16 бит и хотим сложить, например, числа, 40000 и 30000. Мы загружаем их в регистры процессора и исполняем машинную команду сложения регистров (можно просто написать программу на `Си` с использованием типа `unsigned short`). Что получится в результате? Должно получиться 70000. Но как оно может получиться, если мы умеем кодировать 16−ю битами числа, не большие 65535?Это ли число получилось? Разве число 70000 не больше максимального из представимых в 16-битном представлении чисел? Больше. Исполнится ли эта операция? Да, исполнится. Но в результате лишние биты результата будут уничтожены, отрезаны, и мы получим не 70000, а всего навсего 4464. Сложив 32768 и 32768 мы получим 0.

> В ЭВМ все вычисления над беззнаковыми целыми числами, если об этом не сказано особо, производятся по модульной арифметике 2^N , где N — количество бит представления.

Вернёмся к представлению отрицательных чисел. Мы знаем, что есть машинные команды сложения беззнаковых чисел. Плохой практикой было бы для знаковых чисел придумывать свои команды. Так производитель ЭВМ не делают. Поэтому хотелось бы не плодить сущности. Ну а раз так, то представление отрицательных чисел легко вычислить самостоятельно.

Чтобы одной и той же машинной командой можно было складывать и знаковые, и беззнаковые числа, мы всего-навсего хотим, чтобы выражение

$$3 + (-3)$$

давало в результате 0.

Какое из беззнаковых чисел даёт 0 при сложении с тройкой? Результат должен получиться 65536, то есть 0. Это число — 65553.

Двоичное представление числа 65553 есть 1111111111111101_2, то есть оно же будет и двоичным представлением числа -3.

Двоичное представление -1 есть, соответственно, 1111111111111111_2, что легко запомнить и легко проверить.

Такое кодирование чисел называется *дополнительным*.

2.3 Перевод чисел в дополнительный код

Пусть требуется найти представление числа -23 в дополнительном коде в 16-битном представлении. Для этого можно из числа $2^{16} = 65536 = 10000000000000000_2$ вычесть число 23.

Переведём 23 в двоичное представление:

$$23 = 16 + 4 + 2 + 1 = 2^4 + 2^2 + 2^1 + 2^0 = 10111_2.$$

Нужно произвести операцию

$$10000000000000000_2 - 10111_2.$$

Попробовав это сделать, убедимся, что это не очень удобно — приходится помнить все заёмы, которые мы будем делать при вычитании, а их число совпадает с количеством разрядов. Очень много.

Тогда сделаем так (напомним, что мы пока работаем в модульной арифметике по основанию 2^{16}):

$$0 - 23 = 65536 - 23 = 65536 - 1 - 22 = 1111111111111111_2 - 10110_2.$$

А вот вычитание из числа, содержащего все единицы, любого другого делается совсем просто: все биты в двоичном представлении числа 22 меняются на противоположные. Надо только не забыть, что у нас ровно 16 битов и записать ведущие нули у числа 10110_2.

> Для перевода отрицательного числа X в дополнительный код нужно перевести число $X - 1$ в двоичное представление с нужным числом разрядов, и инвертировать все биты.

2.4 Перевод дроби в двоичное представление

Надеемся, что целые числа в двоичное представление умеют переводить все. Поэтому целую часть числа мы переводим отдельно.

С дробной частью всё немного менее привычно.

Основная идея в том, что если дробь 0.1_2 равна 0.5_{10}, то все числа, большие или равные 0.5_{10} будут больше или равны 0.1_2, то есть иметь единицу в первом разряде после запятой[2].

[2]Мы говорим: после десятичной запятой, а сами всё время пишем десятичную точку. Кто виноват? Увы, в ряде стран для разделения дробной части от целой принят крайне неудачный символ — запятая. Российские школьные реалии требуют запятую в качестве разделителя. В языках программирования такое обычно не практикуется. Поэтому мы в числах будем использовать точку, говоря при этом: «запятая».

Узнать, какая первая цифра просто: мы умножает нашу правильную дробь на 2 и смотрим на первую цифру результата. Эта цифра и станет первой цифрой результата. Превратим дробь в правильную, отбросив первую цифру. Повторяем операцию до тех пор, пока или мы не обнаружим, что пришли к нулю, или увидим, что всё началось повторяться, тогда можно определить период этой дроби. Очевидно, что для любой конечной десятичной дроби что-то из этих двых событий сработает.

Это всё можно изложить в следующем виде:

0.263	$0\ (0.263 < 1)$
$0.263 \cdot 2 = 0.526$	$0\ (0.526 < 1)$
$0.526 \cdot 2 = 1.052$	$1\ (1.052 \geqslant 1)$
$0.052 \cdot 2 = 0.104$	$0\ (0.104 < 1)$
$0.104 \cdot 2 = 0.208$	$0\ (0.208 < 1)$
$0.208 \cdot 2 = 0.416$	$0\ (0.416 < 1)$
$0.416 \cdot 2 = 0.832$	$0\ (0.832 < 1)$
$0.832 \cdot 2 = 1.664$	$1\ (1.664 \geqslant 1)$
$0.664 \cdot 2 = 1.328$	$1\ (1.328 \geqslant 1)$
$0.328 \cdot 2 = 0.656$	$0\ (0.656 < 1)$
...	...

2.5 Вещественные числа

Вещественные числа тоже имеют ограниченную информационную ёмкость и, как и все другие числа, записываются в компьютере в двоичном представлении. Любая несократимая дробь, знаменатель которой не является степенью двойки, будет представляться в виде периодической двоичной дроби.

$1/3$	0
$1/3 \cdot 2 = 2/3$	0
$2/3 \cdot 2 = 4/3$	1
$1/3 \cdot 2 = 2/3$	0
...	...

$$\frac{1}{3} = 0.0(01)$$

Это означает, что вычисления с дробями в общем случае будут давать приближённые результаты, в частности,

$$\frac{1}{3} \times 3 \neq 1.$$

Это — фундаментальное свойство всех компьютерных представлений чисел: либо мы обеспечиваем абсолютную погрешность, либо относительную. Третьего не дано.

2.6 Представление вещественных чисел

Любое вещественное число в компьютере представляется в виде $M \times 2^C$. Таких представлений теоретически может быть бесконечное число ($5 = 5 \times 2^0 = 2.5 \times 2^1 = 1.25 \times 2^2 \dots$), поэтому обычно используется *нормализованное* представление. В *нормализованном* представлении $1 \leqslant M < 2$ и представление оказывается единственным.

Для того, чтобы перевести вещественное число в двоичное представление, требуется знать, сколько битов из общего представления выделяется на характеристику (порядок) (степень при двойке) и мантиссу. Для 32-битных *float* чисел на характеристику выделяется 8 бит, на мантиссу — 23 бита. А где же ещё бит? Как обычно, самый левый бит есть знак. Для 64-битных это 10 бит на мантиссу и 53 бита на характеристику.

Правило перевода *float* X в битовое представление:

1. Узнать знак числа. Если число отрицательно, первый слева будет равен единице, иначе — нулю.
2. Нормализовать число. $X = A \times 2^B$, где $1 \leqslant A < 2$
3. Прибавить к числу B число 127. Если получившееся число C больше 254, то число непредставимо и равно бесконечности, тогда записываем 8 единиц и за ними 23 нуля. Иначе переводим C в двоичное представление и записываем 8 разрядов этого представления.
4. Отбросить от A первую единицу. Получится правильная дробь.
5. Перевести дробь в двоичное представление с нужным количеством разрядов (23 для float).
6. Записываем 23 получившихся бита слева направо.

Правило перевода *double* X в битовое представление: см. *float*. Заменить 127 на 511, 254 на 1022, 8 разрядов на 10 разрядов, 23 бита на 53 бита.

Правило перевода *extended* X в битовое представление: см. *float*. Заменить 127 на 16383, 254 на 32766, 8 разрядов на 15 разрядов, 23 бита на 64 бита.

3 Исполнитель программ

3.1 Понятие об исполнителе

Алгоритм для исполнения должен быть преобразован в нечто, удобное для исполнителя. В нашем случае это — машинный код. Машинный код и сопутствующие ему данные, в виде значения переменных, сохраняются во *внешней памяти* компьютера (не путать с оперативной!) — жёсткий или SSD диск, флешка, CD/DVD/BR-диск в виде *файла* с *программой*.

3.2 Язык Си как исполнитель программ

В языке Си мы манипулируем такими терминами, как переменная, цикл, функция, тип данных. Виртуальный компьютер языка Си может сложить две переменные, если их тип допускает это (сложить два целых числа можно, а два указателя — нельзя). Мы делим переменные на

- «долгоживущие», то есть, такие, время жизни которых совпадает с временем жизни всего процесса; это либо глобальные переменные, описанные вне функций, либо статические переменные, описанные с ключевым словом *static*.
- «короткоживущие», время жизни которых совпадает с временем жизни блока, в котором они описаны; это — переменные, описанные внутри функций без слова *static*.
- «динамические», которые мы можем создавать и удалять специальными командами в программе (`calloc`, `malloc`, `new`, `free`, `delete`). Доступ к таким переменным возможен только через указатели.

Как оказывается, такие разновидности переменных хорошо отображаются на реальную процессорную архитектуру. Все такие способы хранения и использования переменных имеют свои особенности, свои плюсы и свои минусы. О них мы расскажем немного позже, когда ознакомимся с архитектурой поподробнее.

3.3 Язык `C++` как исполнитель программ

Всё, что мы говорили про язык `Си`, имеется и в `C++`. Новые возможности языка `C++`, такие, как объекты и методы, как оказывается, есть комбинация всё тех же трёх видов. Поэтому далее мы не будем специально акцентировать наше внимание на том, какой язык мы используем, `Си` или `C++` в качестве исполнителя алгоритмов.

3.4 Язык `Python` как исполнитель программ

При переходе исполнения алгоритмов на язык `Python` волшебным образом исчезают и долгоживущие, и короткоживущие переменные. А что остаётся? Динамические. Как мы вскоре выясним, это — не самый быстрый вид. В частности, именно поэтому не стоит надеяться, что на `Python` можно написать алгоритмы, которые будут исполняться быстрее эквивалентных по реализации на `Си`. Впрочем, возможность из `Python` исполнять код, подготовленный на `Си`, никуда не девается. Но это тема совсем другой книги.

3.5 Программы и процессы

Для исполнения программы требуется дать команду *операционной системе* для загрузки программы в память и для её исполнения. Исполняющаяся программа называется *процессом*.

3.6 Процесс разработки программ

Алгоритм программы записывается на *языке программирования*, не зависящем от компьютера, на котором будет исполняться программа. Есть и языки, сильно зависимые компьютера. Наиболее близкий к машинному языку — *язык ассемблера*, для разных типов компьютеров они могут быть разными.

Текст программы хранится в *исходных файлах*, *исходниках*. Имена исходных файлов на языке `Си` обычно оканчиваются на `.c`, на языке `C++` — на `.cc` или `.cpp`. Тексты программы на языке ассемблера обычно имеют хранятся в файлах, имена, которых оканчиваются на `.s` или `.asm`.

Процесс разработки программы:

1. начинается с разработки алгоритма программы.

2. Алгоритм переводится на язык программирования и набивается в *редакторе*.

3. *Компилятор* преобразует набранный нами код в *объектные файлы*. Некоторые компиляторы вначале преобразуют текст на Си/C++ в текст на языке ассемблера и только потом компилятор с языка ассемблера преобразует их в объектные файлы.

4. *Сборщик* программы собирает все объектные файлы и *библиотеки* в *исполнимый файл*, который мы и называем файлом с *откомпилированной программой*, или, чаще просто с *программой*.

5. Операционная система по нашей команде *запускает* программу на исполнение.

6. Исполняющаяся программа взаимодействует с нами или другими источниками информации и порождает *вывод — результат исполнения*

7. Если результат исполнения нас не устраивает — возвращаемся к пункту 2.

8. Первичная разработка программы закончена.

4 Подробнее об архитектуре

4.1 Принстонская и Гарвардская архитектуры

Как заставить компьютер исполнять какие-либо действия? Как он действует? Почти все современные компьютеры исполняют действия шаг за шагом. Как определить, что нужно делать на очередном шаге? Как определить, какой шаг нужно сделать следующим? Давайте назовём одиночные шаг исполнения *машинной командой*. Что, например, может сделать одиночная машинная команда?

Это может быть, например, следующее:

- Прибавь единицу к какому-то числу, хранящемуся в строго определённом месте.
- Выполни функцию, начинающуюся с какого-то места.
- Скопируй число, находящееся в одном месте в другое место.
- Если последняя исполненная операция дала в отрицательный результат, то следующая команда, которую мы будем исполнять, находится в заданном месте.

Таких команд может быть много, но за один шаг исполняется одна команда. Мы говорили «в определённом месте». Что это такое?

Так как мы имеем дело с компьютерами, которые, в свою очередь, имеют дело с числами, каждое «определённое место» может быть представлено каким-то числом, *адресом*. Таким образом, общий план работы компьютера заключается в следующем:

- По «текущему адресу» взять машинную команду.
- Выполнить предписанные командой действия. У команды могут быть *операнды*, определяющие места, откуда взять входные данные для команды и куда положить выходные данные.
- Установить новый «текущий адрес»
- Повторять, пока нас не остановят.

Самые первые компьютеры исполняли шаг за шагом те программы, которые были в них «зашиты» комбинацией проводов, соединённых штекерами на наборном поле. Готовить такие программы было (по нынешним меркам, конечно) удовольствие ниже среднего. Впрочем, считать всё на бумаге было ещё хуже.

Возник вопрос: как сократить трудоёмкость написания программы, причём таким образом, чтобы компьютер всё ещё мог её исполнять?

Две группы исследователей — в Гарварде и в Принстоне сошлись на одном: программа должна храниться в памяти и данные находиться в памяти. Единица хранения данных называется *ячейка памяти*. Далее пути разошлись: Принстон решил, что и машинные команды должны храниться в таких же самых ячейках памяти, что и данные — вместе с ними. Каждая ячейка памяти могла быть интерпретирована как машинная команда и исполняться. В Гарварде эту схему немного изменили: пусть машинные команды хранятся в отдельной памяти со своей собственной адресацией.

Ячейки памяти в то время были достаточно странные: единицей хранения и обработки были числа, например, длиной 36, 45, 60 бит. Должна ли машинная команда состоять всегда из 45 бит? Принстон считал, что да, должна. Гарвард возражал: зачем так много? Может быть, мы будем хранить машинные команды в ячейках другого размера, более удобного для каких-то целей?

Вторая причина того, что Гарвардская архитектура могла оказаться более удобной — тот факт, что в то время время доступа к памяти, доступной только для чтения, обычно было много меньше, чем время работы с памятью, доступной для чтения и записи. Поэтому хранение программ в более быстрой памяти приводило к меньшему времени *счёта*. Программа как-то записывалась в память, но на момент исполнения запись в эту память блокировалась.

Интересно, какая архитектура удобнее для программистов? Оказывается, Принстонская архитектура позволяла, в том числе, подготавливать машинные команды *во время исполнения* программы, что давало добавочную гибкость. К тому же, как оказалось, Гарвардская архитектура была более дорогой в производстве. Таким образом, обе архитектуры существовали одновременно.

Принстонская архитектура имеет ещё одно название: архитектура фон Неймана.

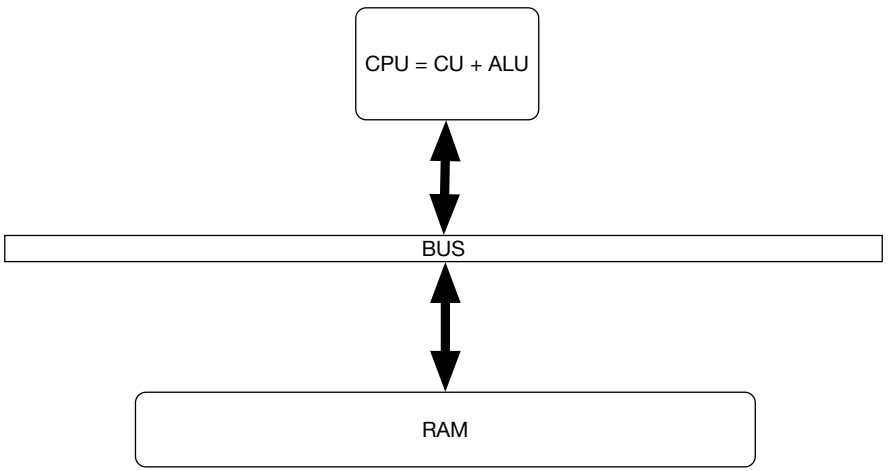
Оперативная память — место, где в компьютере хранятся коды машинных команд процессора во время исполнения процессов а так же оперативные данные, требуемые для исполнения.

Существует ещё понятие *внешняя* память, которая обычно прямым образом для процессора недоступна и перед её использованием из неё нужно *загрузить* данные в оперативную, а после использования — *выгрузить* их обратно.

- *Принстонская*, или архитектура *фон Неймана* — программа вместе со своими данными хранится в единой памяти с единым адресным пространством.
- *Гарвардская архитектура* — машинные команды хранятся отдельно от данных и не могут смешиваться.

А какая архитектура применяется сейчас? Они сейчас настолько переплелись, что программисты видят архитектуру, как фон Неймановскую, а аппаратура — как Гарвардскую.

`RAM` — *память со случайным доступом*. Соединена с исполнителем, *процессором*, устройством связи (*шиной*, `BUS`). Процессор содержит *счётчик команд*. При исполнении команды из памяти извлекаются *операнды*, которые участвуют в операции, результат помещается в память.

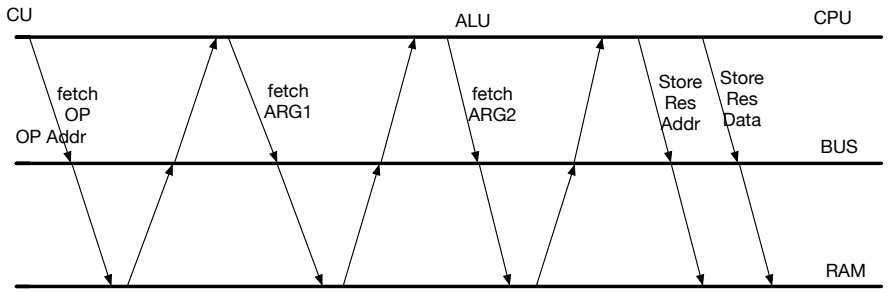


Компоненты процессора:

- `АЛУ`,`ALU` — исполняет операции арифметические и логические
- `УУ`,`CU` — выбирает команду и операнды из памяти, определяет исполняемую команду, передаёт команду и операнды в `АЛУ` и записывает результат операции в `RAM`.

Посредник между RAM и процессором — *шина*, BUS. На первых компьютерах шины представляли из себя простой набор проводников.

4.2 Что происходит в процессоре Принстонской архитектуры при исполнении команд



Предположим, что исполняется инструкция с двумя операндами.

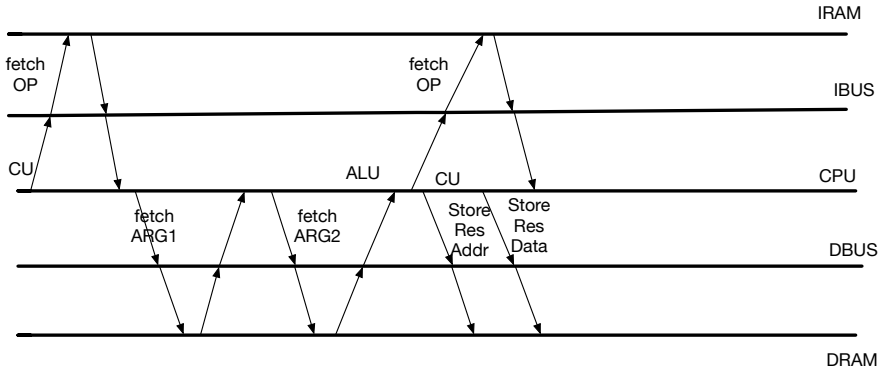
1. Нужно получить инструкцию для исполнения.
2. На шину *выставляется* адрес инструкции.
3. Память отслеживает операцию и выставляет на шину значение по запрошенному адресу.
4. CPU ждёт завершения операции с памятью.
5. Код команды получен. Команда декодирована.
6. Запрашивается операнд 1 (fetch). Ожидание операнда 1.
7. Запрашивается операнд 2. Ожидание операнда 2.
8. Передача исполнения ALU.
9. В память отправляется результат операции.
10. Запрос из памяти очередной инструкции. ...

Пока мы поняли, что нужно не менее трёх устройств. Сейчас они у нас работают работают строго последовательно и невозможно начать действия, не дождавшись результата от других.

4.3 Что происходит в процессоре Гарвардской архитектуры при исполнении команд

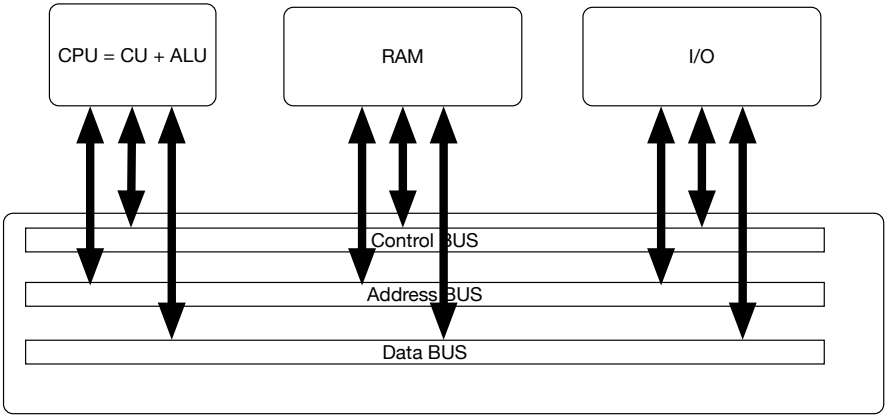
Как мы помним, память программ отличается от памяти данных. Как это может изменить деятельность процессора? Адресные пространства памяти и организация памяти могут быть различными, разрядности адреса и данных могут быть различными.

Давайте будем обслуживать каждое адресное пространство собственной шиной. Например, шина памяти программ будет обслуживать 15 битные адреса, а слова будут 45-битными. Шина памяти данных, скажем, будет обрабатывать 12 битные адреса а слова в памяти будут 18-битными.



Обратите внимание на то, что после того, как `ALU` начало исполнение команды, сразу же после этого можно запросить очередную команду из памяти программ. А это означает, что время исполнения программы, состоящей из нескольких инструкций, можно сократить. Наличие второй шины в Гарвардской архитектуре даёт той несомненное преимущество с точки зрения производительности.

Для программистов интересна неразличимость данных и программ (Принстонская) архитектура. Производители имитируют удобную для программиста архитектуру любым удобным для себя образом.



Самая продуктивная мысль Гарвардской архитектуры: как можно большее количество операций исполнять параллельно. А давайте добавим ещё одну шину адреса. А давайте добавим шину ввода/вывода. А давайте ввод/вывод отдадим периферийным процессорам — чтобы пока идёт ввод/вывод, например, на жёсткий диск, `CPU` был свободен для исполнения программ.

Сейчас так и происходит — для ввода/вывода существуют периферийные процессоры.

Немного забежим вперёд и дадим несколько определений из теории операционных систем:

- **Вычислительное ядро** — компонент центрального процессора, исполняющий машинные команды. Видится как отдельный процессор, исполняющий свой собственный поток инструкций.
- **Вычислительный поток** — абстракция виртуального процессора, исполняющая свой собственный поток инструкций в адресном пространстве, разделяемом с другими вычислительными потоками, исполняющимися в контексте одного процесса.
- **Процесс** — единица исполнения операционной системы, характеризующаяся собственным адресным пространством. Процесс состоит по крайней мере из одного вычислительного потока.

Архитектура: `CPU` Каждый процессор имеет по крайней мере два режима работы:

- **системный**, в котором возможно исполнение любой машинной команды, включая команды ввода/вывода и изменения системных регистров процессора.

- **пользовательский**, в котором использование некоторых *опасных* привилегированных команд приводят к особым ситуациям.

- **Системный вызов** — запрос к временному переводу режима процессора в системный для исполнения особых заявок наших программ.

- **Переключение режимов** процессора происходит при каждом системном вызове. Это — дорогая операция (сотни тактов).

Каждый процесс имеет два *контекста*:

- **пользовательский**, который состоит из *адресного пространства* (виртуальной памяти) и

- **системный**, включающий компоненты процесса, изменяемые только с помощью системных вызовов, такие, как таблицы страниц, таблицы открытых файлов, аутентификационная информация.

Если процесс не исполняет системных вызовов, он может изменять только зафиксированное множество ячеек в своём адресном пространстве.

4.4 Адресация

В каждой исполнимой программе имеются переменные нескольких видов.

- **Долгоживущие** переменные. Они создаются и принимают начальные значения (*инициализируются*) в момент создания. В `Си` это глобальные и статические переменные. Каждая переменная имеет свой адрес и этот адрес не меняется, пока программа живёт. Пример:

```
// p01.c
#include <stdio.h>
int g = 123;

void foo() {
  static int val = 321;
  val++;
  g--;
  printf("foo: val=%d g=%d\n", val, g);
}

int main() {
  foo();
```

```
  printf("main: g=%d\n", g);
  foo();
  printf("main: g=%d\n", g);
  foo();
  printf("main: g=%d\n", g);
}

% cc p01.c
% ./a.out
foo: val=322 g=122
main: g=122
foo: val=323 g=121
main: g=121
foo: val=324 g=120
main: g=120
%
```

Здесь имеется одна *глобальная* переменная `g`, которая с самого начала имеет значение 123. Внутри функции `foo` имеется *статическая* переменная `val`, инициализированная числом 321. Трёхкратный вызов функции `foo` и анализ его вывода показывает, что переменные сохраняют свои значения между вызовами функции. Говорят, что для глобальных и статических переменных адреса *резервируются* перед исполнением программы и по этим адресам ничего другого находиться не может.

> Адрес в исполнимом процессе — уникальное число, однозначно определяющее расположение места в памяти процесса.

В момент исполнения программы, с точки зрения компьютера, эти переменные равнозначны и мы уже в этот момент не можем отличить их — статические ли они или глобальные.

- **Короткоживущие** переменные. Глобальные и статические переменные не так уж часто и встречаются. Современные технологии создания программ советуют их избегать[3]. А что же тогда делать? Оказывается, часто гораздо удобнее оказывается использовать переменные, которые имеют небольшое время жизни. В `Си` такие переменные объявить просто: после любой открывающей фигурной скобки можно объявить несколько таких переменных. Как только исполнение программы дойдёт до парной закрывающей фигурной скобки, эти переменные исчезнут. Зачем это всё?

[3]Почему не рекомендуется массово использовать глобальные и статические переменные? На это есть причины, связанные с тем, что такие переменные создают лишние зависимости в программе. Другие причины связаны с параллельным программированием. И о том и о другом мы узнаем, но не сейчас.

можно спросить, ведь это же очень дорого — создавать и уничтожать переменные.

Оказывается, всё не так уж и плохо. Но перед тем, как двигаться дальше, введём ещё один термин:

> *Регистр* процессора — адресуемая память, находящаяся внутри процессора, для доступа к которой не требуется обращения к оперативной памяти.

Операции над регистрами обычно производятся на десятичные (!) порядки быстрее, чем операции над оперативной памятью. В регистрах могут находиться как данные, используемые в алгоритме, так и адреса. Язык *Си*, это ведь указатели! Да, указатели для доступа к данным часто размещаются именно в регистрах процессора. Здесь мы должны понимать, что указатели могут храниться и в оперативной памяти, но если нужно получить значение по указателю, то придётся скопировать его значение в какой-то регистр процессора и только после этого мы получим к нашим данным доступ.

> Регистров процессора очень ограниченное количество и их правильное использование может многократно ускорить исполнение алгоритма. Компилятор помещает в регистры данные, интенсивно используемые в процессе исполнения алгоритма.

Тот факт, что регистры существуют, позволяет объяснить, почему в `Си` имеются короткоживущие переменные[4]. Компилятор может помещать наши переменные в регистры процессора[5]. Так как после достижения закрывающей фигурной скобки все переменные исчезают, компилятору может повторно использовать те же регистры для других переменных в другом *блоке*.

А что делать, если регистров для переменных не хватает? Мы можем *адресовать* такие переменные с использованием какого-нибудь выделенного регистра, обычно называемого регистром *стека*. При запуске процесса под *стек* выделяется определённое место в оперативной памяти, и регистр стека содержит адреса из этого участка памяти. Мы говорим: переменная расположена в стеке.

```
// p02.c
```

[4] Говоря про `Си`, мы не принижаем роль других языков. Но именно на `Си` видна тесная зависимость языка и архитектуры.

[5] Давайте с этого момента сократим терминологию и *регистры процессора* будем называть просто *регистрами*.

```
#include <stdio.h>

int foo(int x) {
  int bar = x * 2;
  printf("foo::bar: address=%p value=%d\n", &bar, bar);
  return bar;
}

int qwe(int x) {
  int t = foo(x) * 3;
  printf("qwe::t: address=%p value=%d\n", &t, t);
  return t;
}

int main() {
  int x = foo(10);
  int y = qwe(15);
  printf("main::x: address=%p value=%d\n", &x, x);
  printf("main::y: address=%p value=%d\n", &y, y);
}
// end of p02.c
```

```
% cc p02.c
% ./a.out
foo::bar: address=0x16bbb39d8 value=20
foo::bar: address=0x16bbb39a8 value=30
qwe::t: address=0x16bbb39d8 value=90
main::x: address=0x16bbb3a1c value=20
main::y: address=0x16bbb3a18 value=90
%
```

Обратите внимание, что адреса (а мы их выводим в шестнадцатиричной системе) — какие-то большие числа! Неужели так и есть? Да, на современных компьютерах для хранения адреса используется 64 бита. Неужели нам доступно такое большое количество памяти? Увы, нет. Мы в следующем разделе поговорим об этом немного подробнее.

Важнее всего заметить, что адреса переменной `bar` в функции `foo` и переменной `t` в функции `qwe` в точности совпадают! С другой стороны, переменная `foo` в функции `bar` при разных вызовах имеет разные адреса!

Это, как оказывается очень полезно, так как ненужные в настоящее время переменные не занимают памяти. Совсем. А доступ к ним, как уже

говорилось, возможен с помощью регистра стека.

Стек не бесконечен (обычно его размер исчисляется единицами мегабайт), поэтому нужно к нему относиться с уважением и не помещать туда большие структуры данных, например, огромные массивы.

Не верите, что стек не очень большой? Смотрите.

```
// p03.c
#include <stdio.h>

int foo(int size) {
  int bar[size];
  for (int i = 0; i < size; i++) {
    bar[i] = i * i;
  }
  int sum = 0;
  for (int i = 0; i < size; i++) {
    sum += bar[i];
  }
  return sum;
}

int main() {
  for (int i = 1; ; i += i) {
    int sum = foo(i);
    printf("foo(%d)=%d\n", i, sum);
  }
}
// end of p03.c

% cc p03.c
% ./a.out
foo(1)=0
foo(2)=1
foo(4)=14
foo(8)=140
foo(16)=1240
foo(32)=10416
foo(64)=85344
foo(128)=690880
foo(256)=5559680
foo(512)=44608256
foo(1024)=357389824
```

```
foo(2048)=-1433752576
foo(4096)=1423267840
foo(8192)=-1465208832
foo(16384)=1297440768
foo(32768)=-1968521216
foo(65536)=-715816960
foo(131072)=-1431633920
foo(262144)=1431699456
foo(524288)=-1431568384
foo(1048576)=1431830528
zsh: segmentation fault  ./a.out
%
```

То, что функция выводит отрицательные числа, уже понятно — 32 бита типа `int` не хватает для представления сумм.

Но то, что программа «упала» при попытке использовать массив размером 8 мегабайт (`sizeof (int) * 2097152`) говорит нам о том, что при запуске программы стеку выделили не такую уж большую память. Почему бы стеку не выделять неограниченную память? Подумайте над плюсами и минусами предложения.

А что же делать, если нам нужно действительно много данных? Тогда мы используем *динамические* переменные.

- **Динамические** переменные. Это то, что доступно только через указатели. Мы можем попросить выделить нам кусок памяти определённого размера и вернуть его адрес. Этот адрес запоминается в указателе и его мы должны беречь, как зеницу ока.

Итак, каждая используемая переменная какого-то языка в программе может иметь свой адрес. Умные компиляторы могут программу оптимизировать. Чтобы не использовать медленную память компилятор для ускорения работы может назначить какой-то переменной регистр процессора. Но это относится только к короткоживущим переменным! Если к переменной где-нибудь была применена операция взятия адреса &, то место резервируется обязательно на время жизни этой переменной.

4.5 Виртуальное и физическое адресные пространства

Предположим, что мы написали простую программу:

```
#include <stdio.h>
```

```
int a,b;

int main() {
   scanf("%d", &a);
   printf("Address of a=%p, value of a=%d\n", &a, a);
   scanf("%d", &b);
   printf("Address of a=%p, value of a=%d\n", &a, a);
}
```

Мы откомпилировали её и запустили на исполнение. Первое, что делает эта программа, это считывание глобальной переменной a. Введём какое-нибудь значение, например, 123.

```
123
Address of a=0x40020, value of a=123
```

Программа остановилась, чтобы считать значение переменной b. Пока ничего не будем вводить и запустим в другом окне ещё копию этой же программы. Введём число 567.

```
567
Address of a=0x40020, value of a=567
```

Интересно, первая программа положила в ячейку с адресом 0x40020 число 123, а вторая в ячейку с точно таким же адресом — число 567.

Вопрос: если мы переключимся сейчас на окно с первой программой и введём число b, что она выведет? Она должна вывести адрес переменной a и её значение. Адрес переменной не изменился, но ведь вторая программа положила по этому адресу число 567? Проверим.

```
888
Address of a=0x40020, value of a=123
```

Ничего не изменилось! Как это?! Мы же только что в другой программе положили по этому адресу другое значение.

Ответ заключается в том, что каждая программа использует так называемой *виртуальное* адресное пространство. Одно и то же значение адреса в различных копиях одной и той же программы будет соответствовать различным адресам *физической* памяти.

Программа при исполнении использует виртуальные адреса. Микросхемы памяти имеют физические адреса. Преобразованием виртуальных адресов в физические управляет операционная система. Для пользовательской программы невозможно узнать физический адрес памяти, он может

изменяться во время исполнения программы.

Ещё интересный факт: несмотря на то, что для хранения адреса используется 64 бита, не все такие адреса доступны для работы программы. Поэтому для исполнения программы требуется помощь операционной системы: программа запрашивает у той требуемое ей количество памяти. Выделяемая системой память становится частью *адресного пространства* процесса. Обращения к этой памяти разрешены, к остальной — запрещены и такое обращение приведёт к аварийному завершению исполняемого процесса[6].

4.6 Процессор как исполнитель программ

Итак, исполнитель *процессор* ничего не знает про переменные. Он умеет работать только с ячейками памяти и с регистрами. Если переменная находится в памяти, главное для него — адрес ячейки. Все адреса в исполняемой программе есть виртуальные адреса. Если при исполнении одной программы мы положили в ячейку по конкретному адресу какое-нибудь значение и не производили с этим адресом других операций записи в нашем процессе[7], то операция чтения обязательно получит то значение, которое мы положили, независимо от того, через какое время мы это значение запросим и исполняются ли на компьютере другие процессы. Это означает, что

При исполнении нашего процесса одновременно с другими, память нашего процесса для других процессов недоступна.

Процессор и память — два независимых устройства. Для их взаимодействия применяется ещё одно устройство — *шина*. Шина нужна для подключения к связке память/процессор добавочных устройств, например, графического адаптера или жёстких дисков. Быстродействие шины во многом определяет быстродействие всей вычислительной системы.

4.7 Физическая память. Шина

Физическая память располагается в микросхемах, как-то соединённых с процессором. Мы применяем термин *шина*, хотя всё может оказаться не так просто. Классическая шина представляла набор проводников, соединяющих устройства компьютера между собой. Сейчас шина, скорее, набор *виртуальных* соединений и на современных вычислительных системах это тоже набор управляющих микросхем. Конечно, проводники никуда не делись и они попрежнему соединяют все компоненты системы.

[6] Подробнее об этом будет на втором курсе, когда мы будем изучать операционные системы.

[7] но мы можем сделать это в другом вычислительном потоке, об этом вы узнаете на втором курсе.

4.8 Регистры процессора. Разрядность процессора

Итак, мы установили тот факт, что обращение к чему-либо, что находится *вне* процессора — будь то оперативная память или внешние устройства относительно медленно. Если операции производятся *внутри* процессора, за единицу времени можно обработать больше данных. Каждая ячейка памяти вне процессора имеет свой номер, по которому к ней можно обращаться. Внутренняя память процессора тоже имеет свои ячейки, которые мы называли *регистрами*, и которые тоже имеют свою нумерацию.

В каждой машинной команде наряду с кодом операции (*что* будет делаться) обычно присутствуют адреса *операндов* — *над чем* производятся действия. И код операции, и операнды нужно *закодировать*, то есть привести в набор битов. Всё это определяет *длину машинной команды*. Адреса, как мы знаем, в современных вычислительных системах очень большие, до 64 бит. Поэтому машинная команда, которая содержит такой адрес, тоже может содержать очень много бит.

Для того, чтобы закодировать номер регистра, много бит не требуется. Если, например, в процессоре используется 32 регистра, то для их кодирования требуется всего 5 бит. Короче машинная команда — больше команд поместится в одном и том же количестве памяти.

Различают *регистры общего назначения*, *РОН*, которые могут применяться в машинных командах самого разного применения — арифметических и логических операциях, хранения адресов и пр. Вторая важная группа — регистры *вещественной арифметики*, которые точно требуются в вычислительной математике. Все современные ЭВМ имеют и *векторные* регистры, позволяющие за одну машинную команду обработать сразу несколько десятков операндов. Обычно имеются и *специализированные регистры*, обращение к которым возможно из небольшого количества команд. К ним относятся *управляющие* регистры, регистры *отладки*, *сегментные* и подобные. Какие-то из специализированных регистров мы упомянем в дальнейшем, но фактически они требуются только для использования в коде операционной системы и мы его не увидим.

Размер регистров общего назначения — количество бит, которые они содержат — во многом определяет архитектуру и возможности вычислительной системы. Если регистр способен хранить и обрабатывать 8 бит информации, то за одну машинную команду с участием такого регистра мы не сможем обрабатывать числа большие 127 или 255. Чтобы сложить, скажем, два числа, не превосходящие 10000, нам придётся применять навыки из школы — сложить младшие 8 бит, понять, не было ли переноса в старшие разряды, запомнит этот перенос, сложить старшие разряды и добавить значение переноса. Это займёт две или три машинные команды. Если бы регистры могли хранить 16 бит, такая операция заняла бы одну машинную команду. По сути, размер регистров определяет систему счисления для операций.

Бóльшее количество бит в регистре обычно означает меньшее количество команд, требуемое для реализации одного и того же алгоритма на похожих системах с различным размером регистра.

4.9 Организация исполнения

Наличие памяти и регистров не означает, что все процессоры устроены одинаково. Наоборот, каждый процессор уникален.

Чтобы процессор исполнил машинную команду, он должен определить, что внутри неё находится, как она закодирована. Давайте приведём несколько примеров кодировки.

В качестве подопытного кролика попробуем взять простой фрагмент алгоритма.

```
x = 3*a - (2*b/3)%2;
```

4.9.1 Стековый безадресный процессор

Вначале рассмотрим гипотетический компьютер, у которого имеется стек с операциями `push` и `pop`, исполнительное устройство которого работает с двумя верхними элементами стека — оно снимает их со стека, производит операцию и кладёт результат опять в стек. Пусть переменные хранятся в памяти и адрес такой переменной может быть закодирован 16-ю битами.

Операция	Код	Описание
push	1	положить в стек переменную
pushi	2	положить в стек константу
pop	3	извлечь из стека
add	4	сложить второй элемент с верхушкой
sub	5	из второго элемента вычесть верхушку
mul	6	умножить второй элемент на верхушку
div	7	второй элемент разделить на верхушку
mod	8	остаток от деления второго на верхушку

Тогда наш алгоритм при переводе на язык ассемблера примет такой вид:

```
push a
pushi 3
add
push b
pushi 2
mul
pushi  3
div
pushi 2
```

```
mod
sub
pop x
```

Как это будет закодировано? Каждая переменная имеет свой адрес. Пусть были назначены такие адреса:

Переменная	Адрес
a	120
b	157
x	300

Тогда всё будет закодировано так:

```
00000001 0000000001100100 ; push a
00000010 0000000000000011 ; pushi 3
00000100                  ; add
00000001 0000000010011101 ; push b
00000010 0000000000000010 ; pushi 2
00000110                  ; mul
00000010 0000000000000011 ; pushi  3
00000111                  ; div
00000010 0000000000000010 ; pushi 2
00001000                  ; mod
00000101                  ; sub
00000011 0000000100101100 ; pop x
```

Обратили внимание на то, что мы должны передавать адрес каждой переменной как составную часть машинной команды? Как передавать в машинную команду константы, вроде 2 и 3? Как тогда отличать те биты в команде, которые кодируют адрес, от тех битов, которые кодируют константу? Исполнитель, процессор, должен знать это в момент, когда исполняется программа. Это можно кодировать различными способами, например, различными машинными командами (как у нас), или добавлять в машинную команду биты, кодирующие режим адресации.

Константы, содержащиеся в машинном коде, называются *непосредственными* операндами.

4.9.2 Трёхадресный процессор

В этом процессоре каждая команда содержит четыре поля: код и три адреса.

Операция	Код	Описание
add	4	сложить второй и третий, поместить в первый
sub	5	из второго элемента вычесть третий, поместить в первый
mul	6	умножить второй и третий, поместить в первый
div	7	разделить второй на третий, поместить в первый
mod	8	остаток второго на третий поместить в первый

А как кодировать константы? Можно добавить новые коды операций. Давайте добавим один бит, который будет кодировать наличие константы вместо адреса. Код операции теперь у нас будет занимать 7 бит, следующий бит будет единица, если биты третьего операнда — непосредственный операнд и ноль, если биты третьего операнда — адрес в памяти.

Тогда наша программа на языке ассемблера станет такой:

```
mul t1,a,3
mul t2,b,2
div t2,t2,3
mod t2,t2,2
add x,t1,t2
```

Нам пришлось добавить две переменные, `t1` и `t2`! Иначе нам некуда девать промежуточные результаты. Пусть они имеют адреса 10 и 11.

Если мы захотим закодировать это в машинные команды, мы получим следующее:

```
0000110 1 0000000000001010 0000000001100100 0000000000000011 ; mul t1,a,3
0000110 1 0000000000001011 0000000010011101 0000000000000010 ; mul t2,b,2
0000111 1 0000000000001011 0000000000001011 0000000000000011 ; div t2,t2,3
0001000 1 0000000000001011 0000000000001011 0000000000000010 ; mod t2,t2,2
0000100 0 0000000100101100 0000000000001010 0000000000001010 ; add x,t1,t2
```

Количество команд в программе уменьшилось. Только пришлось использовать дополнительную память и общая длина программы в битах увеличилась!

4.9.3 Двухадресный процессор с регистрами

Добавим в процессор 16 регистров от `r0` до `r15`. Очевидно, что каждый можно закодировать 4-мя битами. Заставим процессор производить операции только над регистрами. Если с переменной нужно что-то сделать, нужно загрузить её в регистр, поработать над ней и затем выгрузить обратно в память.

Операция	Код	Описание
load	1	загрузить из памяти в регистр
loadi	2	загрузить в регистр константу
store	3	выгрузить из регистра в память
add	4	`r1 += r2`
sub	5	`r1 -= r2`
mul	6	`r1 *= r2`
div	7	`r1 /= r2`
mod	8	`r1 %= r2`

```
loadi r2,2
loadi r3,3
load  r0,a
mul   r0,r3
load  r1,b
mul   r1,r2
div   r1,r3
mod   r1,r2
sub   r0,r1
store r0,x
```

Будем кодировать каждую команду 8-мью битами и константы, в том числе и адреса, — 20 битами.

```
00000010 0010 00000000000000000010      ; loadi r2,2
00000010 0011 00000000000000000011      ; loadi r3,3
00000001 0000 00000000000001100100      ; load  r0,a
00000110 0000 0011                      ; mul   r0,r3
00000001 0001 00000000000010011101      ; load  r1,b
00000110 0001 0010                      ; mul   r1,r2
00000111 0001 0011                      ; div   r1,r3
00001000 0001 0010                      ; mod   r1,r2
00000101 0000 0001                      ; sub   r0,r1
00000011 0000 00000000000100101100      ; store r0,x
```

В качестве временных переменных мы использовали регистры. Это быстро. Код стал меньше. Адресное пространство расширилось от 2^{16} ячеек до 2^{20} ячеек. Если программу писать хорошо и использовать принцип локальности, от большого количества операций с памятью можно вообще отказаться, если у нас много регистров.

4.10 Подробнее об архитектуре памяти и кэш-памяти

В 2021 году массово применяются два вида памяти:

- **статическая,** `SRAM` — значение бита определяется состоянием ячейки, типично 6 или 8 транзисторов. Состояние устойчивое, регенерация не требуется, дорогая (очень!), быстрая.

- **динамическая,** `DRAM` — значение бита определяется зарядом конденсатора. Состояние неустойчивое, считывание значение разрушает заряд, требуется регенерация, дешёвая, медленная.

Пусть процессор запросил у памяти 4 байта. Предположим, что мы используем модули физической памяти памяти с частотой 2400 `MHz`.

Логически `DRAM` есть двумерный массив столбцов и строк. Для получения значения нужно подать на шину управляющие сигналы выбора номера строки ($RAS\sharp$). Значение сохраняется в промежуточной быстрой статической памяти, располагаемой рядом с микросхемами динамической. В следующем цикле подаётся сигнал выбора номера столбца ($CAS\sharp$), вместе с ним сигнал направления передачи (чтение или запись). Каждая операция требует фиксированного количества тактов. Для нашего модуля это, например, `27-9-9-9` – общая *латентность* (задержка) на разных этапах запроса данных. Общая задержка не может быть меньше суммы (27+9+9+9=54) что даёт не менее 25 наносекунд. В реальности задержки оцениваются в 40-60 наносекунд.

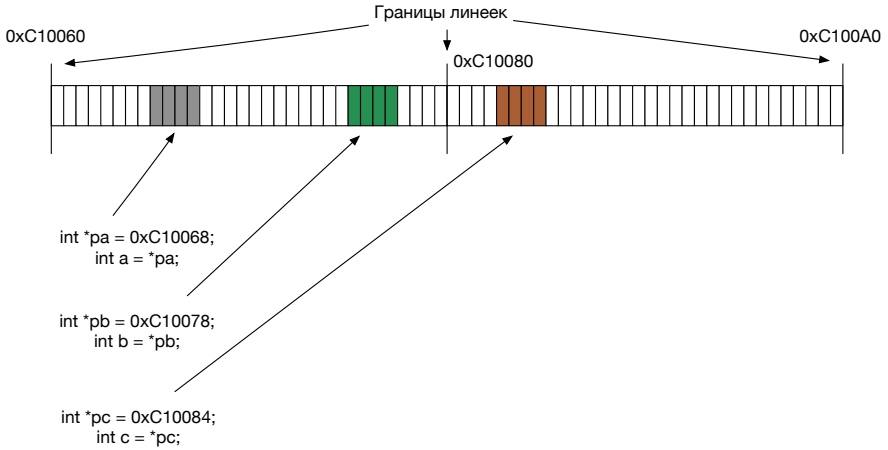
Для ускорения работы адресные сигналы подаются на группу модулей. Передача из модулей происходит параллельно. Из памяти считывается не запрошенные 32 бита, а целая линейка из 256-512-1024 бит. Количество битов определяется шириной шины памяти. Куда это всё доставляется? В *кэш-память*. И только оттуда данные направляются в `ALU`. Кэш-память — необходимая часть современных процессоров. Зачем это нужно?

Память — узкое место современных вычислительных систем. Ширина шины памяти увеличилась от 8 до 128 битов, пропускная способность (сколько бит можно передать за такт) тоже увеличилась. Латентность изменилась незначительно, она уже почти 30 лет составляет те же 40-60 наносекунд.

Введение кэш-памяти помогает решить две основных проблемы:

1. При запросе первого 32-х битного слова общая задержка определяется только латентностью. Контроллер кэша запрашивает *линейку* в 32-64 байта. Время доставки такой *кэш-линейки* в процессор практически совпадает со временем доставки одного слова. Эффективная латентность на байт уменьшается.

2. Линейка хранится в кэше и при повторном обращении к тому же участку, физической передачи информации по шине не происходит — нагрузка на шину уменьшается.



При первичном обращении к переменной a загружается линейка кэша размером 32 байта, большая латентность (`cache-miss`, кэш-промах).

Обращение к переменной b, которая располагается недалеко от a уже не требует обращения к физической памяти (`cache-hit`, кэш-попадание). Первое обращение к переменной c из другой линейки опять вызовет кэш-промах.

Возможен ещё один способ увеличения производительности: при записи в кэш-память управление процессору передаётся немедленно, сама же операция записи произойдёт в удобное для контроллера время. Опять работает принцип, что дополнительное устройство может работать параллельно с основным. Это позволяет комбинировать запросы на запись и сливать их в одну транзакцию, что ещё более увеличивает производительность.

Итак, зафиксируем общий принцип работы кэш-памяти:

- Вся память делится на линейки фиксированного размера. Адреса выровнены $\rightarrow$ младшие биты адреса линейки содержат нули (если в линейке 64 байта то младшие 6 бит — нули).

- Первое чтение (*cache-miss*) заставляет всю линейку, содержащую данный адрес, загрузиться в кэш. Если часть переменной находится в одной линейке, а часть — в другой, то такие переменные называются *невыровненными*, это плохо, и при обращении к такой переменной в кэш загружаются две линейки. После загрузки линейка *присутствует* в кэше.

- При любом запросе к памяти он проходит черех контроллер кэша, который проверяет наличие адреса в *тегах* кэша. Если линейка с нужным адресом принадлежит кэшу, то обращения к физической памяти не производится и (*cache-hit*) запрос исполняется из кэша.

- Если кэш уже заполнен данными, но требуется новая загрузка, возникает

ситуация вытеснения (*evict*) какой-либо линейки. Обычно реализуется по LRU[8])-подобному алгоритму.

Полностью ассоциативный кэш может назначать любому адресу памяти любую линейку. Как быстро найти в большом массиве нужное значение? Это очень дорого если делать быстро и медленно, если делать дёшево.

Поэтому каждый адрес может отображаться не на все возможные линейки, а только на определённое количество линеек. Например, адрес `0xF0040` при размере кэш-памяти в 512 линеек мог бы отображаться только на 1-ю, 5-ю, ... $4n + 1$ линейки. Соответственно, каждая линейка может обслуживать не все адреса, а только содержащие определённый набор битов. Например, 1-я линейка могла бы содержать только адреса, остаток от деления начала которых на 2^8 равен `0x40`.

Это называется *ассоциативностью* кэша.

Кэширование прозрачно для большинства программ. Знаем про кэширование → увеличиваем производительность. Иногда в 10 раз. Иногда в 100.

Эффективность кэша определяется несколькими факторами:

1. Отношением количества попаданий в кэш к общему числу запросов.

$$HitRatio = \frac{CacheHits}{CacheHits + CacheMisses}$$

2. Скоростью обработки.

   - T_R — среднее время доступа к памяти.
   - T_C — среднее время доступа к кэшу.
   - $H = \frac{CacheHits}{CacheHits + CacheMisses}$ — вероятность попадания в кэш.

   Тогда математическое ожидание времени доступа к элементу памяти будет равно

$$T_{eff} = T_C \times H + (T_R + T_C) \times (1 - H)$$

Пусть $T_C = 1$ и $T_R = 30$. Тогда

H	0.90	0.91	0.92	0.93	0.94	0.95	0.96	0.97	0.98	0.99	1.00
T_{eff}	4	3.7	3.4	3.1	2.8	2.5	2.2	1.9	1.6	1.3	1.0

При 95% попадании в кэш вместо 100% эффективная производительность уменьшается, как мы видим, в 2.5 раза!

Для максимизации попадания кода и данных в кэш, есть подходы:

[8]LRU — Least Recently Used, алгоритм, обнаруживающий данные, которые не использовались дольше всего.

1. обеспечить высокую локальность данных:

   - хорошие компиляторы, которые помещают переменные в регистры;
   - хорошие программисты стараются использовать алгоритмы, не прыгающие по памяти;
   - поменьше использовать объектную модель — обычно в ней много указателей, следовательно, много прыжков по памяти. Любители объектно-ориентированного программирования, не обижайтесь! ООП не призвано ускорять *работу* программ, оно призвано ускорять их *разработку*.

2. Увеличить количество кэша:

   - к первому уровню, самому быстрому, но и самому маленькому, добавим второй, третий и т. д. уровни, каждый из которых будет всё больше, но и всё медленнее.
   - `L1`: десятки килобайт, время обслуживания 2-3 такта
   - `L2`: сотни килобайт, время обслуживания 5-10 тактов, HitRatio=99%.
   - `L3`: сервера и многоядерные процессоры: десятки и сотни мегабайт.

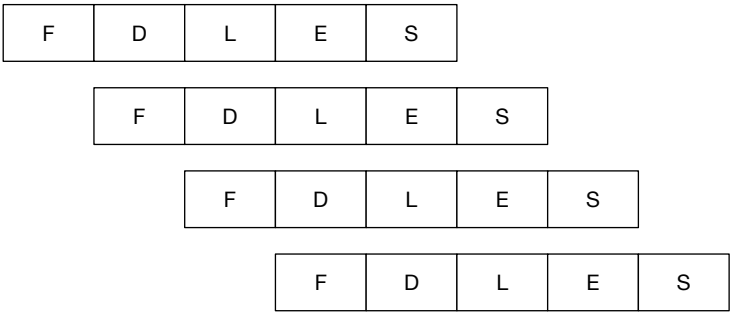
4.11 Конвейерное и суперскалярное исполнение

4.11.1 Конвейерное исполнение

Конвейер — один из способов увеличить производительность процессора. Мы уже знаем, что при исполнении инструкции проходит много этапов. Предположим, что наш процессор имеет пять таких этапов, *ступеней конвейера*, и каждая ступень исполняется ровно один такт:

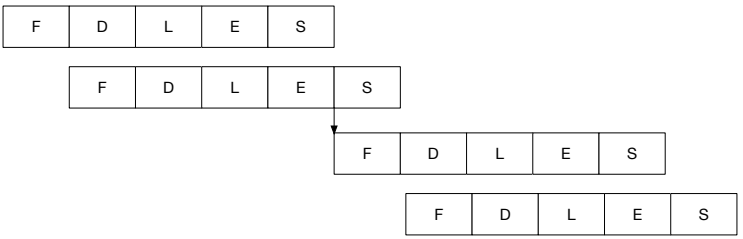
1. `F` — выборка команды
2. `D` — декодирование команды
3. `L` — выборка операндов из памяти
4. `E` — исполнение
5. `S` — занесение результатов в память.

При последовательном коде от начала исполнения до её конца пройдёт пять тактов. Вроде бы как много. Но на выходе конвейера каждый такт будет появляться результат исполнения какой-то команды! Можно ли определить *латентность* команды, время от начала её исполнения до её завершения? А требуется ли это нам? Нас больше интересует темп исполнения команд — сколько команд за единицу времени конвейер способен выдать.



Конвейер команд при последовательном исполнении.

Однако, не всё так просто. Предположим, что в поток исполняемых команд затесалась команда условного перехода. Произойдёт переход или нет — мы можем узнать только начав исполнение команды, получив её операнды и исполнив. Если переход произойдёт, то придётся считывать новую инструкцию, то есть, запустив конвейер с самого начала. Печально.



Конвейер команд при наличии команд перехода.

Переходы очень плохо влияют на производительность конвейера. Процессор вынужден прервать конвейерную цепочку. Для борьбы с этим применяют предсказание переходов. Производится выборка команды по наиболее ожидаемому маршруту. Если переход не осуществился — откат.

Чем больше ступеней конвейера, тем меньше действий эта ступень должна успеть завершить за фиксированное время — тем меньше это фиксированное время можно сделать, то есть, можно увеличить тактовую частоту.

Однако, чем больше ступеней конвейера, тем больнее по производительности бьют условные переходы. А их в программах много.

4.11.2 Суперскалярное исполнение

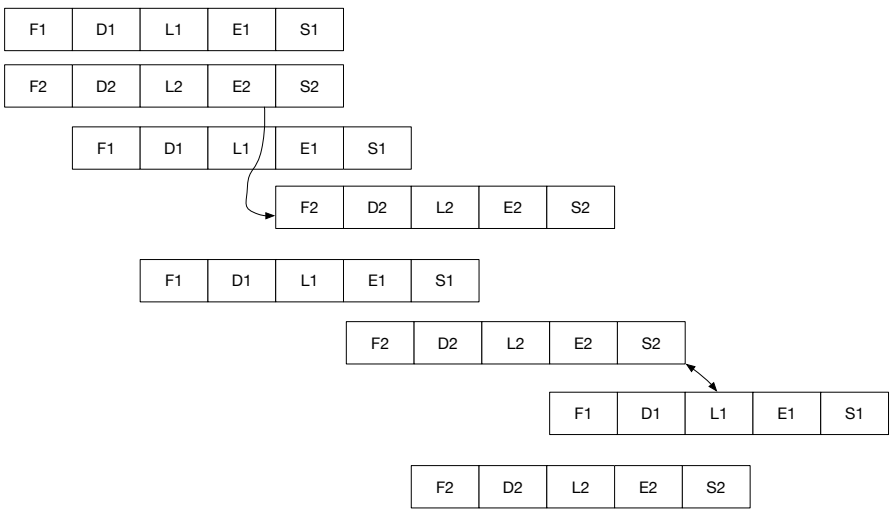
На разных исполнительных устройствах в один момент времени могут исполняться различные машинные команды. В начале очередного такта на исполнение могут быть переданы несколько команд. Процессор `Pentium` имел

два конвейера — `u` и `v`. Конвейер `u` мог исполнять любую команду. При определённых условиях, следующую команду мог взять на исполнение конвейер `v`. Сейчас имеется 3-8 конвейеров и одновременно могут запускаться до 8 команд, исполняющихся параллельно.

Возможны несколько экземпляров каждого исполнительного устройства. Они могут исполнять работу одновременно. Если одно устройство исполнения должно дожидаться завершения другого, то это называется *зависимостью по данным*.

Предположим, что в нашем гипотетическом процессоре имеется по два исполнительных устройства каждого типа. Очевидно, что при идеальных условиях, такой процессор сможет выдавать результаты до двух инструкций за такт.

Однако, здесь может вмешаться тот факт, что для того, чтобы исполнить инструкцию `X`, нужно дождаться результатов инструкции `Y`, которая сейчас тоже исполняется. Это, конечно, не позволяет добиться идеальных результатов, но, тем не менее, общее время исполнения заданного набора инструкций сокращается.



Суперскалярное исполнение с зависимостью по данным.

5 RISC и CISC архитектуры

Проектировщики машинных команд — люди с фантазией. Мы уже заметили, что один и тот же алгоритм можно реализовать с помощью разного набора машинных команд. Ещё мы заметили, что каждая команда может кодироваться различным числом бит.

А какие бы команды вы добавили, если былы бы проектировщиком процессора? Было время, когда языки программирования, хотя они уже появились, служили лишь средством написать программу, затратив меньшее количество времени, чем написать её же на языке ассемблера, и, тем более, в машинных кодах. Те, кому нужен был результат в виде эффективной программы, предпочитали ассемблер. Производители компьютеров заметили, что если добавлять удобные для использования в программах машинные команды, то это пригодится как авторам компиляторов, так и программирующим на ассемблере. Нужны команды работы с десятичными числами — давайте их добавим! Нужны тригонометрические функции — вот, держите. В процессоре `VAX-11`, например, была команда, которая принимала три операнда — длину массива, содержащего коэффициенты полинома, его адрес и значение переменной, для которой полином должен быть вычислен. Представьте себе — всего одна команда вместо цикла. Выполнялась ли эта команда быстрее, чем написанный вручную цикл? Может быть. Но кто будет писать такой цикл, если всё можно сделать одной командой?

Вы видите какие при таком подходе проблемы? А ведь они имеются.

Если, например, процессор содержит 200 машинных команд, для их кодирования достаточно 8 бит. А если 300? Использовать 9 бит? Или может быть, удобнее использовать 16 бит и зарезервировать пространство для кодирования ещё новых команд? А что делать со старым кодом? Если изменится формат команды, то старые программы могут перестать исполняться.

Другая проблема состоит в следующем. Рассмотрим следующий фрагмент кода:

```
int x = a*2;
int y = b + c;
```

Предположим, что переменные `x` и `a` компилятор разместил в памяти, а переменные `y`, `b` и `c` — в регистрах. Тогда, чтобы исполнить первую строку нужно сначала обратиться к памяти за операндом, затем произвести операцию и потом положить результаты операции в память. Как мы уже знаем, память — медленная штука и в традиционной архитектуре вторая команда должна начать исполнение только после того, как исполнилась первая. Но ведь ни операнды второй команды, ни её результаты, не зависят от первой и вторая команда могла бы выполниться независимо от первой и её результаты были бы получены раньше, чем результаты первой. Такое исполнение называется *Out-of-band*, или *внеочередное*. Однако, как процессор сможет это узнать? Если бы он только мог загрузить себе несколько *инструкций* одновременно... Но как он может это сделать быстро, если первая команда, скажем, занимает 9 байт, а вторая — 3? Длину первой команды можно определить, только считав код операции, и проанализировав его. Вот если бы сделать бы все команды одной длины, чтобы всё это проводить с лёгкостью...

А что ещё хотелось бы сделать для того, чтобы можно было попытаться улучшить производительность процессора? Конечно, стоило бы сделать так, чтобы инструкции выполнялись как можно быстрее, лучше всего, за один такт. Но тогда придётся отказаться от сложных инструкций. Может, действительно, стоит от них отказаться, если всё можно значительно ускорить?

Ещё возникло мнение, что нужно избавиться от тех машинных команд, которые могут производить операции над ячейками памяти. Например, если у нас имеется команда увеличения ячейки памяти на единицу

```
inc x
```

то при выполнении этой команды приходиться дождаться, пока старое значение ячейки будет доставлено в процессор, затем увеличено и отправлено назад в память. Команда может исполняться сравнительно долго, если память не блещет быстродействием, и фазы исполнения этой команды трудно совместить с исполнением других команд.

Если мы написали

```
load r0,x
inc  r0
store r0,x
add  r1,r2
```

то во время ожидания загрузки значения переменной `x` в память можно было бы исполнить те инструкции, которые находятся дальше в коде и которые не зависят от результатов исполнения команды. Важно, что внутренний параллелизм исполнения команд мог бы увеличиться.

Сказано — сделано. Одним из разработчиков процессора, основанного на подобных идеях был Стенфордский университет. Такая модель процессора получила название `RISC` — Reduced Instructions Set Computer, в отличие от прежних идей, которые теперь стали называться `CISC` — Complex Instructions Set Computer. Вот изложение принципов первых `RISC` процессоров:

- Каждая инструкция исполняется строго один такт процессора.
- Для работы с памятью используются только инструкции `load` и `store`.
- Имеется много регистров, типично 32.
- Все инструкции кодируется однотипно и занимают одно и то же число бит.

Этот процессор был создан университетской командой разработчиков, воплощён в жизнь и оказалось, что имея в несколько раз меньшее количество

транзисторов (а чем больше транзисторов, тем дороже процессор), он исполняет типичные программы *быстрее*, чем существовавшие в то время *CISC* процессоры.

Сейчас, конечно, трудно представить, что в то время имелись десятки совсем различных процессорных архитектур, и имелось не меньшее число их производителей. Те из производителей, которые не приняли архитектуру RISC всерьёз, давно ушли из процессорного бизнеса. Все те, кто остался, либо являются чистыми представителями RISC архитектуры (ARM, SPARC, POWER) либо используют эту архитектуру невидимо для пользователя, на лету преобразуя CISC команды процессоров, которые видимы пользователям, во внутренние RISC команды (Intel, AMD).

Почему матёрые производители процессоров не восприняли всерьёз RISC-архитектуру? Как мы уже заметили, машинных команд для исполнения одних и тех же алгоритмов на RISC-процессорах могло оказаться больше. На первых процессорах даже операцию умножения, которая требовала одной или около того машинных команд на CISC, приходилось реализовывать в виде функции. Потребовались новые компиляторы. Внезапно оказалось, что качество компилятора определяет быстродействие компьютера, причём очень сильно!

Могу подтвердить это своим опытом. В разгар появления процессоров Intel i486, работающих на тактовой частоте 66 MHz, (о нём ещё будет идти разговор) на место моей работы был доставлен компьютер на RISC-архитектуре SPARC, основанный на разработках университета Беркли. Тактовая частота его составляла всего 40 MHz. Так как получили мы его для решения расчётных задач проектирования, первое, что было сделано, это попытка скомпилировать и запустить наши расчётные программы, написанные на языке Си. Поработав на нём, мы удивились его производительностью — расчётные программы, использующие вычисления с вещественными числами, требовали примерно в 5 раз меньше времени, чем i486. Задачи, не требующие вещественной арифметики, исполнялись в 2-3 раза быстрее. Исследования, что можно ещё сделать с этим компьютером, привели к тому, что был найден другой компилятор с языка Си. Когда мы заменили им штатный, все вновь откомпилированные программы стали работать ещё быстрее: вещественная арифметика ускорилась примерно в 1.6 раза, а целочисленная примерно в 2! По-сути, мы приобрели новый компьютер.

6 Немного о работе с битами

Те процессоры, с которыми мы будем работать в этом семестре, используют двоичную арифметику, с представлением данных в которой мы уже познакомились.

Однако, для более глубокого понимания машинных команд неплохо бы научиться работать с битами более продвинуто. Например, нужно узнать, какое

значение закодировано в битах [5..7] по адресу 0x3345.

Нам достаточно использовать язык Си. Вскоре мы узнаем, что по программе на языке Си можно достаточно неплохо представить, что будет происходить на компьютере.

Мы будем использовать шестнадцатиричную систему счисления для записи чисел. Почему? Потому, что её использование позволит нам увидеть каждый из битов числа, не переводя его явным образов в двоичное представление целиком.

Интересно ещё посмотреть, каким образом алгоритмы влияют на время исполнения программы.

Задача будет состоять в том, чтобы определить, сколько единичных битов в двоичном представлении заданного числа. Мы придумаем несколько алгоритмов и испытаем их — определим самый быстрый.

Разделим задачу на подзадачи. Их может оказаться несколько.

- Определить время, прошедшее между двумя событиями.
- Получить количество бит в числе, используя первый алгоритм.
- Получить количество бит в числе, используя второй алгоритм.
- ...

Первая задача — посчитать время исполнения достаточно быстропротекающего события. Имеется много инструментов, чтобы определить это время. Например, можно воспользоваться функцией clock из файла <time.h>, которая возвращает количество *тиков*, которое процесс затратил с момента начала его исполнения. Эти тики можно потом перевести в секунды.

```
#include <time.h>
    ....
    clock_t start = clock();
    // Что-то делаем
    clock_t end = clock();
    double running_time = (double)(end - start) / CLOCKS_PER_SEC;
```

Чтобы уменьшить влияние измерителя времени на точность, повторим эксперимент несколько раз. Сколько? Это зависит от точности измерения времени выбранными свойствами. В Visual Studio в Windows, например, CLOCKS_PER_SEC равен 100, так что точность измерения времени — 10 миллисекунд.

Второе, что нам нужно понять — с одними и теми же данными мы будем проводить измерения, или с разными. Современные оптимизирующие компиляторы обладают значительным интеллектом. Они могут уже при компиляции вычислить значение функции от известного аргумента, поэтому не дадим им развернуться и будем вычислять много значений нашей функции.

Модуль испытаний у нас спроектирован. Теперь добавим удобств в программировании. Испытанием с получением времени займётся функция `bench`. А что ей передать в качестве аргументов? А давайте передадим, в том числе, саму функцию, которую нужно тестировать, и, заодно, параметры тестирования. Здесь мы пользуемся тем, что современные процессоры представляются нам в виде архитектуры фон Неймана — мы можем получать указатели на функции и передавать их в качестве данных.

Функция `main` у нас готова:

```
int main() {
    const int LIM = 100000000;
    bench("simple", run_simple, LIM);
    // ... Другие bench
}
```

В функции `run_simple` мы запустим нашу функцию столько раз, сколько попросят:

```
int run_simple(int lim) {
    int sum = 0;
    for (unsigned x = 0; x < lim; x++) {
     unsigned count = 0;
        // Для x вычислим count - количество бит в нём.

        }
        sum += count;
    }
    return sum;
}
```

Мы сделали прототип, его можно менять в соответствии с придуманным нами алгоритмом.

А сама функция `bench` будет едина для всех алгоритмов, которые мы будем тестировать:

```
typedef int (* run_func)(int); // Указатель на функцию

void bench(const char *name, run_func func, int lim) {
    clock_t start = clock();
    int answ = func(lim);
    clock_t end = clock();
    printf("test %10s runs for %.3f secs with result %d\n",
      name, (double)(end - start) / CLOCKS_PER_SEC);
}
```

Наконец, мы добрались до самого алгоритма: имеется число `x`. Сколько в нём единичных бит?

Хотя алгоритм, который мы ходим разработать, на вход принимает 32-битное число, для иллюстрации мы ограничимся 8-ми битным. Биты в числе нумеруются справа налево. Почему? Потому, что это показатели степени при соответствующей двойке.

$$1001101101_2 = 1\cdot 2^9 + 0\cdot 2^8 + 0\cdot 2^7 + 1\cdot 2^6 + 1\cdot 2^5 + 0\cdot 2^4 + 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0$$

Пусть наш x будет равен $44 = 101100_2$.

Дополним его слева нулями до 8.

7	6	5	4	3	2	1	0
0	0	1	0	1	1	0	0

Новая подзадача: как получить значение $i-$го бита в числе? Для этого нужно создать число, в котором биты во всех позициях равны нулю, кроме $i-$й. Для этого необходимо и достаточно взять единицу, которая имеет следующее представление:

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	1

и сдвинуть её влево на i. Для $i = 3$ результат операции `1 << i` будет равен

7	6	5	4	3	2	1	0
0	0	0	0	1	0	0	0

Теперь операция `&` (побитовое и) с числом x даст следующее:

	7	6	5	4	3	2	1	0
`x`	0	0	1	0	1	1	0	0
`1 << i`	0	0	0	0	1	0	0	0
`x & (1 << i)`	0	0	0	0	1	0	0	0

Это число отлично от нуля. Если бы в третьей позиции числа x был бы ноль, оно бы было равно нулю. Вот и первый способ получения количества бит в числе. У нас готова функция для `bench`.

```
int run_dummy(int lim) {
    int sum = 0;
    for (unsigned x = 0; x < lim; x++) {
        int count = 0;
        for (unsigned i = 0; i < 32; i++) {
            if ((x & (1 << i)) != 0) {
```

```
                count++;
            }
        }
        sum += count;
    }
    return sum;
}
```

Собираем целую программу:

```
// Файл bench.c
#include <stdio.h>
#include <time.h>

int run_dummy(int lim) {
    int sum = 0;
    for (unsigned x = 0; x < lim; x++) {
        int count = 0;
        for (unsigned i = 0; i < 32; i++) {
            if ((x & (1 << i)) != 0) {
                count++;
            }
        }
        sum += count;
    }
    return sum;
}


typedef int (* run)(int);

void bench(const char *name, run func, int lim) {
    clock_t start = clock();
    int answ = func(lim);
    clock_t end = clock();
    printf("test %10s runs for %.3f secs with result %d\n",
      name, (double)(end - start) / CLOCKS_PER_SEC, answ);
}


int main() {
    const int LIM = 100000000;
    bench("dummy",  run_dummy, LIM);
```

```
}
```

Компилируем её и запускаем. Все значения времени, которые мы здесь увидим, получены на процессоре `Intel Xeon W3520`.

```
$ gcc bench.c
$ ./a.out
test      dummy runs for 14.125 secs with result 1314447104
$
```

Сто миллионов вычислений функции исполнилось за примерно 14 секунд. Одно вычисление, следовательно, заняло около 140 наносекунд. Много это или мало? Один такт на используемом процессоре длится $\frac{1}{2.67 \cdot 10^9} \approx 0.37$ наносекунд. Всего вычисление заняло в среднем $14.125 * 10 / 0.37 \approx 370$ тактов.

Что у нас пока не так? Мы не разрешили компилятору языка `Си` воспользоваться принципом локальности и он не стал размещать локальные переменные в регистрах процессора. Дадим ему такую возможность.

```
$ gcc -O bench.c
$ ./a.out
test      dummy runs for 3.621 secs with result 1314447104
$
```

Время исполнения теста уменьшилось в 3.9 раза! Регистры — великая сила.

Почему мы назвали функцию `dummy`? Потому, что этот алгоритм самый простой, но не самый умный. Давайте попытаемся убрать операцию сравнения `if ((x & (1 << i)) != 0)` и заменим её на что-то другое. Заметим, что при успехе сравнения мы увеличиваем переменную `count` на 1. Но эта единица — это число, в котором все биты нулевые, кроме самого правого. Это число просто надо прибавить к `count`. Поступим ещё хитрее: чтобы не пробегать все 32 бита, мы будем «забирать» младший бит и сдвигать число на один бит направо. Как только число обнулится, цель достигнута.

```
int run_simple(int lim) {
    int sum = 0;
    for (unsigned x = 0; x < lim; x++) {
        for (unsigned y = x; y != 0; y >>= 1) {
            sum += y & 1;
        }
    }
    return sum;
}
```

Добавляем функцию, добавляем строку в `main`, проверяем результат:

```
$ gcc -O bench.c
$ ./a.out
test      dummy runs for 3.621 secs with result 1314447104
test     simple runs for 2.263 secs with result 1314447104
$
```

Подсчёт всё ещё правильный, и время улучшилось.

Количество операций в алгоритме `dummy` всегда было равно 32, сейчас оно стало равно номеру самого старшего бита.

Улучшаем алгоритма дальше. Что будет, если из числа вычесть 1?

	7	6	5	4	3	2	1	0
`x`	0	0	1	0	1	1	0	0
`x - 1`	0	0	1	0	1	0	1	1

Интересно! Все младшие нули превратились в единицы, а самая младшая единица обнулилась. Что, если над этими числами произвести операцию побитового и?

	7	6	5	4	3	2	1	0
`x`	0	0	1	0	1	1	0	0
`x - 1`	0	0	1	0	1	0	1	1
`x & (x - 1)`	0	0	1	0	1	0	0	0

Операция привела к тому, что самый младший бит в числе `x` обнулился, и только он один[9].

Оформляем алгоритм в виде функции и проверяем.

```
int run_smart(int lim) {
    int sum = 0;
    for (unsigned x = 0; x < lim; x++) {
        int count = 0;
        for (unsigned y = x; y != 0; y &= y-1) {
            count++;
        }
        sum += count;
    }
    return sum;
}
```

[9]Этот и подобные факты при побитовых операциях служат основой структуры данных *дерево Фенвика*.

```
$ gcc -O bench.c
$ ./a.out
test      dummy runs for 3.602 secs with result 1314447104
test     simple runs for 2.311 secs with result 1314447104
test      smart runs for 1.680 secs with result 1314447104
$
```

Время ещё немного уменьшилось.

А вот теперь — внимание. Следующий алгоритм сложен для быстрого понимания.

До сих пор мы работали с единичными битами. Так, как их в числе может быть до 32, мы в эту константу при таком подходе упёрлись. Сила побитовых операций в том, что они одновременно производятся над всеми битами числа. Можно сказать, что мы можем рассматривать 32-битное число как вектор, содержащий 32 единичных бита. Или как вектор, содержащий 16 маленьких 2-битных чисел. Или как вектор из 8 4-битных чисел. Эти числа можно извлекать из вектора с помощью тех же самых побитовых операций. Например, чему равно число, закодированное в битах `[2..4]` в нашем числе? Создадим *маску*, содержащую единичные биты в позициях `[2..4]` и произведём операцию `AND` над маской и `x`.

	7	6	5	4	3	2	1	0
`x`	0	0	1	0	1	1	0	0
`mask`	0	0	0	1	1	1	0	0
`x & mask`	0	0	0	0	1	1	0	0

Теперь если маску сдвинуть вправо на два разряда, получим желаемый результат.

```
unsigned v24 = (x & (0x1C)) >> 2;
```

Читается не очень хорошо. На практике делают по-другому.

```
unsigned v24 = (x >> 2) & ((1 << 3) - 1);
```

Убедитесь сами, что если сдвинуть единицу влево на n бит и и вычесть 1, получится число с ровно n единицами. Изменив порядок — сначала сдвиг числа вправо, затем наложение маски, получаем нужный результат. Почему мы применяем слово *маска*? Потому, что при операции побитового и в результирующем значении останутся только те биты, которые содержатся в маске, а остальные будут стёрты.

Хорошо, мы умеем извлекать группу бит из числа. Ну и что?

А это означает, что теперь мы знаем, как производить операции над группой бит параллельно и как получать результаты!

Вначале рассмотрим число x как вектор 1-битных значений. Нам тогда нужно найти сумму элементов этого вектора.

Разобьём вектор на чётные (красные) и нечётные (зелёные) элементы.

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0

Теперь нам нужно сложить все пары. Так как результат не превосходит двух, мы положим то, что получилось после сложения пар (а их 4), в четыре двухбитных числа. Красные компоненты отделим от зелёных и отправим их по разным четырёмэлементным 2-х битным векторам. Чёрным цветом обозначим получившиеся после операций биты.

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
x & 01010101	0	0	0	0	0	1	0	0
(x & 10101010) >> 1	0	0	0	1	0	1	0	0

В четырёх элементах 2-х битных векторов содержатся счётчики количества бит для зелёных и красных половинок отдельно.

Теперь осталось сложить вектора. Простой операцией +, чем ещё? Переполнения элементов векторов не произойдёт, так как результаты сложения не превосходят двух и поместятся в двухбитовые числа.

	7	6	5	4	3	2	1	0
x	0	0	1	0	1	1	0	0
u = x & 01010101	0	0	0	0	0	1	0	0
v = (x & 10101010) >> 1	0	0	0	1	0	1	0	0
x = u + v	0	0	0	1	1	0	0	0

Пока в результате мы закодировали вектор (0,1,2,0), каждое значение хранится в двух битах. Повторим операцию, разбив вектор на чётные и нечётные элементы и сложив уже пары двухбитных чисел. Опять чётные пары выделим красным цветом, нечётные — зелёным.

	7	6	5	4	3	2	1	0
x	0	0	0	1	1	0	0	0

Снова раскинем их по векторам — на сей раз это будут два 4-битных вектора и сложим:

	7	6	5	4	3	2	1	0
x	0	0	0	1	1	0	0	0
u = x & 0x33	0	0	0	1	1	0	0	0
v = (x & 0xCC) >> 2	0	0	0	0	0	0	1	0
x = u + v	0	0	0	1	0	0	1	0

И, напоследок, сложим элементы 4-х битных векторов.

	7	6	5	4	3	2	1	0
x	0	0	0	1	0	0	1	0
u = x & 0x0F	0	0	0	1	0	0	1	0
v = (x & 0xF0) >> 4	0	0	0	0	0	0	0	1
x = u + v	0	0	0	0	0	0	1	1

Если мы изменим порядок наложения маски и сдвига, то уменьшим число различных масок.

Применим алгоритм к 32-битному числу вместо 8-битного. Теперь операций будет 5 вместо трёх.

```
int run_mask(int lim) {
    int sum = 0;
    for (unsigned x = 0; x < lim; x++) {
        unsigned y = x;
        y = (y & 0x55555555) + ((y >> 1) & 0x55555555);
        y = (y & 0x33333333) + ((y >> 2) & 0x33333333);
        y = (y & 0x0F0F0F0F) + ((y >> 4) & 0x0F0F0F0F);
        y = (y & 0x00FF00FF) + ((y >> 8) & 0x00FF00FF);
        y = (y & 0x0000FFFF) + ((y >> 16) & 0x0000FFFF);
        sum += y;
    }
    return sum;
}
```

Запуск программы даст нам такой результат:

```
$ gcc -O bench.c
$ ./a.out
test      dummy runs for 3.621 secs with result 1314447104
test     simple runs for 2.263 secs with result 1314447104
test      smart runs for 1.769 secs with result 1314447104
test       mask runs for 0.459 secs with result 1314447104
$
```

Мы ускорили алгоритм в 4 раза! Не странно? Это особенность современных процессоров, о которых мы говорили. Они умеют исполнять несколько инструкций одновременно. В результате 20 с лишним машинных команд исполняется за время, чуть большее 12 тактов!

Если мы решим оптимизировать ещё больше, то получим следующий результат:

```
$ gcc -O3 bench.c
$ ./a.out
test      dummy runs for 3.453 secs with result 1314447104
test     simple runs for 2.257 secs with result 1314447104
test      smart runs for 1.666 secs with result 1314447104
test       mask runs for 0.117 secs with result 1314447104
```

Здесь уже поработал компилятор на славу! Он заметил, что можно одновременно вычислять значения функции от разных аргументов (да здравствуют *чистые* функции, которые зависят только от аргументов!) и воспользовался так называемыми *векторными* инструкциями процессора, когда за одну машинную команду обрабатывается не только вектор бит, как в нашем примере, а вектор, скажем, из 8 32-битных чисел. Но об этом будет отдельный разговор.

7 Введение в архитектуру x86-x64

Эта архитектура — одна из двух наиболее распространённых в 2021 году. Именно на ней работает большинство серверов, настольных компьютеров, ноутбуков и значительная часть суперкомпьютеров. Другая популярная архитектура — `ARM`, по численности процессоров, на которых она исполняется, превосходит первую. Но это, в основном мобильные телефоны, планшеты и пока не очень большой число тех компьютерами, которые называются персональными. К слову сказать, этот текст набирается автором на настольном компьютере и ноутбуке, работающим на архитектуре `ARM`. Сервер, с которого вы скачиваете этот файл — на архитектуре `X64`. Так что нужно быть знакомым и с той, и с другой архитектурами.

7.1 16-битная архитектура `X86`. Общее понятие

Начнём мы с архитектуры `X86`. Она была изобретена фирмой `Intel` и развивалась ей и фирмой `AMD`.

Первый процессор этой архитектуры назывался `Intel 8086` и на его основе производились первые действительно массовые *персональные компьютеры*, `PC`.

Этот процессор умел адресовать 2^{20} 8-ми битных байтов. Правда, в первых моделях такое большое количество памяти на ставили, считали это излишним.

Этот процессор был оснащён следующими регистрами:

- `ax` — accumulator, регистр-аккумулятор
- `bx` — base/index регистр индекса/базы
- `cx` — counter, регистр-счётчик
- `dx` — data, регистр данных
- `si` — source index, регистр-источник

di — destination index, регистр-приёмник
bp — base pointer, регистр базы стека
sp — stack pointer, регистр стека
ip — instruction pointer, регистр адреса команды

Почему такие дикие названия? Потому, что перед процессором 8086 Intel выпускал коммерчески популярные процессоры серии 8080 (совсем печальные по нынешним меркам). Так как 8080 был 8-ми битным и мог использовать максимум 64 килобайта памяти, большинство программного обеспечения для него писалось на языке ассемблера. Intel, введя такие названия регистров, полагал, что перенести программы с процессора 8080 на процессор 8086 будет достаточно просто. Такой простоты не получилось, а имена регистров и их роли остались надолго. До сегодняшних дней.

Каждый регистр содержит ровно 16 бит. Первые 4 регистра — ax,bx,cx и dx умеют разбиваться на половинки. Например, 16-разрядный регистр bx состоит из двух 8-ми разрядных половинок — регистров bh и bl. Положив данные в регистр bx, мы заодно и изменяем данные в подрегистрах bh и bl (похоже, что именно это послужило поводом заявить о лёгком переносе программ с процессора 8080).

Это — регистры общего назначения, РОН. Помимо них имеется ещё несколько *специализированных* регистров.

Как обращаться к ячейке памяти? Можно поместить нужный адрес как часть инструкции в виде *непосредственного* операнда. Можно адрес поместить в какой-то регистр, и только после этого заказать обращение к памяти, используя регистр. Это называется *регистровая* адресация, которую можно отнести к *косвенным*.

Сразу отметим, что система команд архитектуры x86 весьма странная и неоднородная. Сейчас мы в этом убедимся и будем убеждаться в этом всё время, пока мы будем изучать эту архитектуру.

Что мы пока увидим странного? А то, что для операций регистровой адресации не все регистры, оказывается, одинаково полезны. Регистры bx, si, di, bp и sp могут содержать адреса, по которым можно обращаться к памяти. Регистры ax, cx и dx могут содержать какие-то адреса, но только в виде копий — чтобы ими воспользоваться, нужно скопировать в правильные регистры. Здесь нам стоит остановиться на некоторое время, чтобы сделать выбор, каким образом мы будем записывать инструкции и их операнды. Для этого мы будем использовать *язык ассемблера*.

7.2 Язык ассемблера

Программа на языке ассемблера состоит из отдельных строк. Блоков, как в Си/Паскале/C++ в языке ассемблера обычно нет.

Бывают строки директив, управляющие процессом компиляции (ассемблирования) и строки с машинными командами.

Каждая строка с машинной командой содержит имя команды и возможные операнды.

Примеры машинных команд:

```
movl    $4, %eax
nop
incl    %eax
```

Перед любой из машинных команд может находиться *метка*, означающая адрес данной точки в машинном коде. Существуют команды, передающие управление выбранной команде в коде.

```
    cmpl $1,%eax
    jb lab1
    incl %ebx
lab1: incl  %ecx
```

В данном примере имеется метка lab1. Обращение к метке — обращение к адресу того места, где метка расположена.

Язык ассемблера служит переходным мостиком от языков высокого уровня (Си, C++,...) к машинному коду.

Если наша основная цель — написание программ *вручную* на языке ассемблера, то синтаксис языка должен быть удобен для чтения, для добавления в него своих удобных конструкций (макросов) и прочее. Он может быть многословным, относительно медленно переводиться в машинный код. По этому пути пошла фирма Intel.

Если наша основная цель — посредничество между языками высокого уровня и машинным кодом, то он должен быть максимально лаконичным, быстро переводиться в машинный код и он не обязательно обязан быть красивым по синтаксису — его будут редко читать и ещё реже на нём писать. По этому пути пошла фирма AT&T, автор операционной системы UNIX. Все компиляторы с языков высокого уровня преобразуют код в текст на языке ассемблера и затем используют компилятор ассемблера для производства машинного кода.

7.2.1 Синтаксис Intel и синтаксис AT&T

Для одних и тех же машинных команд различают синтаксис Intel и синтаксис AT&T.

Их основные различия:

- AT&T — регистры начинаются со знака процента, %eax, %esp. Intel — регистры пишутся как они есть.
- AT&T — перед константами пишется знак доллара, $15. Intel — ничего не пишется.

- AT&T — в двухоперандных командах источник — слева, приёмник — справа. Например

```
movl    %eax, %ebx
```

  копирует содержимое регистра eax в регистр ebx.

  Intel — источник справа, приёмник — слева.

```
mov     ebx, eax
```

- Команды, которые могут в качестве операндов принимать данные разной длины, в Intel кодируются одной и той же мнемоникой, в AT&T команды имеют соответствующий *суффикс*.

```
inc ax ;16 бит, Intel
inc al ; 8 бит, Intel
incw %ax ; 16 бит, AT&T
incb %al ; 8 бит, AT&T
```

- Обращение по адресу, содержащемуся в регистре %bx в ассемблере AT&T кодируется заключением регистра в круглые скобки, (%bx), в ассемблере Intel — в квадратные [bx]. Intel при этом использует более многословный формат команды.

```
mov ax, WORD PTR -4[bp]  ; Intel
movw -4(%bp), %ax        ; AT&T
```

Так как наша задача в первую очередь понимать, что происходит на том или ином процессоре, нужно сделать выбор из одного из способов записи. Мы выберем синтаксис AT&T.

7.3 16-битная архитектура. Продолжаем.

Итак, адрес какого-то места в памяти может быть использован с помощью какого-то регистра. Из этого адреса можно что-то забрать и туда можно что-то положить. Самая первая команда, которую мы рассмотрим подробно — команда mov. Название говорит о том, что мы что-то откуда-то перемещаем (moving). Не верьте этому! Это команда просто копирует *источник* в *приёмник*.

```
movw %ax,%bx  ; Скопировать регистр ax в регистр bx
movw $1,%ax   ; Скопировать непосредственный операнд 1 в регистр ax
movw %ax,(%si); Скопировать регистр ax в память по адресу в регистре si
movb (%si),%al; Скопировать 8 бит по адресу в регистре si, в регистр al
```

Ничего странного не заметили? А странно то, что мы говорили, что память может достигать 2^{20} байт, а сами обращаемся к ней, используя всего лишь 16-битные регистры. А ведь в таком регистра могут быть адреса от 0 до 65535, что никак не дотягивает до предела. Проблема здесь в том, что для того, чтобы добраться до всей памяти, требуется помощь. Здесь в её качестве выступают так называемые *сегментные* регистры. Из 4 штуки и они все 16-битные.

Это регистры

`ds` — data segment, сегментный регистр для данных.
`cs` — code segment, сегментный регистр для кода.
`ss` — stack segment, сегментный регистр для стека.
`es` — extended segment, дополнительный сегментный регистр.

Имена этих регистров используются в качестве *префиксов* перед адресными выражениями. Ассемблерные команды становятся такими:

```
movw %ax,es:(%si)
movb ss:(%si),%al
```

Так писать не очень удобно, поэтому каждому из адресных регистров приписан свой сегментный регистр по умолчанию.

`bx` — приписан `ds`
`si` — приписан `ds`
`di` — приписан `ds`
`bp` — приписан `ss`
`sp` — приписан `ss`
`ip` — приписан `cs`

Таким образом, следующие две команды — синонимы.

```
movw %cx,ss:(%bp)
movw %cx,(%bp)
```

Осталось понять, как всё же получить 20-битный *полный* адрес из двух 16-битных регистров.

Ничего сложного. Например, для команды `movw %cx,ss:(%bp)` нужно просто умножить на 16 содержимое регистра `ss` и прибавить к нему содержимое регистра `bp`. Вот и всё.

Умножение на 16 есть сдвиг влево на 4 бита. Поэтому всё выглядит так:

	19	18	17	16	15	14	13	12	11	10	09	08	07	06	05	04	03	02	01	00
ss					1	1	0	0	1	0	1	1	0	0	1	0	0	1	0	0
ss*16	1	1	0	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0	0
bp					0	0	0	1	1	1	0	0	1	0	0	1	0	0	1	0
ss:bp	1	1	0	0	1	1	0	0	1	1	1	0	1	1	0	1	0	0	1	0

Мы смогли адресовать все 2^{20} байт памяти. Но удобно ли это? Как работать с массивами? Можно, например, загрузить в сегментный регистр начало массива (увы, оно должно быть кратным 16 байтам) и потом с помощью другого

регистра, который пробегает все индексы от 0 до размера массива с нужным шагом, получать доступ к требуемой ячейке памяти.

А что, если размер массива в байтах больше 65536? Один регистр с индексом с работой не справится, он не дотянется от начала до конца массива. А это означает, что при работе с таким массивом придётся постоянно менять значения в сегментном регистре, с помощью которого этот массив обрабатывается. И это, поверьте, крайне неудобно!

`Intel` приобрёл то, что называлось *проклятьем 64 килобайт*. Хотя некоторые компиляторы могли допустить работу с большими массивами, скорость работы с ними была в несколько раз меньше, чем с маленькими.

7.4 16-битная архитектура. Защищённый режим

1 мегабайт памяти очень быстро оказалось недостаточно. Реально в программах могло использоваться не более 640 килобайт, вся память свыше использовалась для взаимодействия с устройствами, включая видеопамять, для программ, исполняющихся при запуске компьютера (Basic Input/Output System — BIOS). Какие-то кусочки памяти свыше 640-килобайтной границы могли использоваться для программ (Expanded Memory), но это не было особенно удобно.

Ещё одна проблема состояла в том, что изменение сегментного регистра было обычной и привычной операцией, в больших программах количество команд загрузки сегментных регистров достигало 10-15% от общего количества исполненных команд. Любая программа могла загрузить в сегментный регистр любое значение — и ей становилась доступна *любая* ячейка памяти. На таком мощном компьютере было невозможно запустить несколько одновременно исполняющихся программ (пусть они и были бы маленькими), в таком режиме, чтобы они были изолированы друг от друга. Любая программа могла прочитать данные другой программы, которая исполнялась бы с ней одновременно, изменить её код и тому подобное. Вы скажете: ну и что? Памяти ведь совсем мало? А ведь на рынке уже имелись 16-разрядные машины, на которых был возможен многозадачный режим. Например, на машине СМ1420, с 248 килобайтами оперативной памяти (правда, с двумя огромными высокоскоростными 650-мегабайтными дисками), работало одновременно до 16 пользователей и каждый исполнял свою задачу — кто-то набирал текст программы, кто-то отлаживал свою программы, кто-то часами вычерчивал на графопостроителе схемы размером 2 на 4 метра. А вот `Intel 8086` этого не умел.

Выпуск процессора `Intel 80286` должен был это исправить.

Набор регистров у процессора остался тем же. Самая главная новинка: появился так называемый *защищённый* режим процессора. В нём адресуемая память могла составлять до 2^{24} байт, то есть 16 мегабайт! Фантастика. За счёт чего это произошло? За счёт того, что сегментные регистры поменяли смысл. Раньше в них хранился кусок адреса. Теперь в 16 битах в них стал

храниться 14-битный номер *дескриптора* и 2-битный уровень привилегий. Всё вместе называлось *селектор*.

Появилось несколько режимов работы процессора, формально 4 (как раз эти два бита и кодировали номер этого режима), фактически два — системный и пользовательский. Теперь программы, работающие в пользовательском режиме, должны были запрашивать у операционной системы всё, что только можно — и память, и ввод/вывод. Это означало, что старые программы, которые *жонглировали* сегментными регистрами, стало невозможно исполнять. Самая популярная операционная система предыдущих компьютеров — ДОС, в защищённом режиме исполняться не могла. Для нового процессора спешно была выпущена операционная система `OS/2`, вначале совместно фирмами `Microsoft` и `IBM`. Но где популярные программы под DOS? Они исполняться под `OS/2` не могут, а если могут то только в специальном режиме.

А что же с проклятьем 64 килобайт? Оно не только не осталось, оно стало гораздо хуже. Проблема было в том, что команда смены сегментного регистра теперь стала исполняться во много раз медленнее — процессору нужно было проверять права данного процесса на этот сегмент и загружать так называемые*теневые* регистры, в которых находились 24-битные адреса, которые прибавлялись к адресным регистрам для получения исполнительного адреса. Если до того с большими массивами работали медленно, то сейчас стали работать мееееедленнооооо. Практически, подобным решением `Intel` запретил большие массивы. А для чего нужно много памяти? Правильно, для того, чтобы обрабатывать много данных. Это оказалось нереальным. Поэтому процессоры `80286` выпускались и были даже более популярны, чем старые, но работали они в *реальном* режиме процессора, то есть как более быстрый `8086`. Нужно было что-то делать. И в 1985-м году `Intel` выпустил процессор `80386`.

7.5 32-битная архитектура. Общее понятие

Этот процессор был первым представителем 32-битной архитектуры у `Intel`, но далеко не первым в ряду других микропроцессоров. Почтим память 32-битных архитектур `Motorola MC68XXX`, `Fairchild Semiconductor Clipper`, `AT&T`, `Western Electric 32XXX`, море их, и посмотрим, получилось ли у `Intel` на этот раз.

Ура!

Регистры стали 32-битными! Это уже значит, что нет проклятья 64 килобайт.

Старые регистры остались, но стали больше. Если к имени старых регистров добавить букву `e`, то мы получим имя нового, 32-битного регистра. Например, `ax` → `eax`. И так для всех. Под именем `ax` теперь доступны младшие 16 бит регистра `eax`, и если `ax` что-то присвоить, то старшие 16 бит останутся неизменными.

Старые сегментные регистры остались, к ним добавились ещё два регистра: `fs` и `gs`. В *защищённом* режиме работа с этими регистрами всё ещё осталась на усмотрении операционной системы. Впрочем, сейчас про сегментные регистры не системному программисту можно и не вспоминать.

Много инструкций стали выполняться несколько быстрее предшественника. Появился 16-байтный *буфер предвыборки команд*, упрощённый предшественник кэш-памяти.

Появилась *страничная виртуальная память*. Подробнее об этом — на 2-м курсе. В двух словах — это механизм, позволяющий процессу использовать большее количество памяти чем установлено физически, за счёт того, что память делится на *страницы* типичным размером 4096 байт и часть страниц памяти процесса может находиться в *физической* памяти, а часть — на *устройстве страничного обмена*, типично жёстком диске в то время.

Далее, 1989 году, был выпущен процессор *i486*, в котором появилась настоящая кэш-память размером 8192 байта, большая часть команд стала исполняться за один такт, вместо двух, как было у `80386`. Ещё одним достижением стало добавление *сопроцессора*, исполняющего инструкции с вещественными операндами длиной до 80 бит. Для прежних процессоров такой сопроцессор нужно было покупать отдельно — для него было предусмотрено место на материнских платах. Наличие такого сопроцессора ускоряло операции с вещественными числами в десятки раз. Это был первый процессор, который получил `RISC`-подобное ядро, сохранив систему команд `CISC`.

Следующий значимый шаг — выпуск процессора `Pentium` в 1993 году. Он получил суперскалярную архитектуру. Его конвейеры `u` и `v` могли в большинстве случаев исполнять две различные команды одновременно. Внутри он получил Гарвардскую архитектуру, разделив кэши на кэш инструкций и кэш данных. Дебютировали векторные расширения, `MMX`, пока не очень востребованные.

`Pentium Pro` довёл количество операций, которые он мог выполнить за один такт, до трёх, впервые реализовав (для `Intel`, конечно), режим *внеочередного исполнения команд*, `OOB`.

После выпуска процессоров `Pentium II` и `Pentium III`, которые отличались с программисткой точки зрения только расширением набора векторных команд, появился `Pentium 4` (2000-й год), основной идеей которого была: максимальное увеличение тактовой частоты невзирая на последствия. Было резко увеличено количество ступеней конвейера, вначале до 20, затем до 31! Штраф за исполнение команд перехода возрос в несколько раз (у `Pentium III` был конвейер в 12 стадий). Чтобы добиться той же производительности, что и предшественник, частота должна была быть выше примерно в полтора раза!

Мы всё это время говорили про `Intel`. `Intel` разработал эту систему команд, `Instruction Set Architecture`, `ISA X86` и предложил эталонную реализацию. Вначале данную `ISA` бросились реализовывать много фирм — `Cyrix`,

AMD, VIA, Transmeta, ... Не у всех получилось это сделать достаточно эффективно — ведь ISA подразумевает *что* надо реализовать, но не указывает *как* это надо сделать.

Неудачи Intel с разработкой и продвижением Pentium 4 привели к тому, что главный конкурент, AMD, вырвался вперёд, предлагая более дешёвые и более производительные процессоры. К тому же именно AMD предложил расширение команд ISA x86, добавившее 64-битные возможности. AMD называл это расширение AMD64, Intel согласился с этим расширением, немного его модифицировав, назвав EM64T. Сейчас для неё в ходу много названий — AMD64, X64, Intel 64, x86-64. Всё это обозначает одно и то же и сейчас мы это рассмотрим подробнее.

7.6 64-битная архитектура. Общее понятие

7.6.1 Регистры

Первое, что мы заметим, это, конечно, регистры. Они действительно стали 64-битными. Конечно, изменились и названия. Регистр eax теперь занимает младшие 32 бита регистра rax.

8 регистров общего назначения предыдущей архитектуры было мало. Добавилось ещё 8 64-битных регистров с именами от r8 до r15. Младшие 32 бита каждого из новых регистров называются r8d, r9d и так далее, младшие 16 бит носят суффикс w: r8w, ... а самые младшие 8 — суффикс b: r8b.

7.6.2 Структура программы на ассемблере

Как же знакомиться с ассемблером? Один из простых способов — сочетать его с языком Си. Можно писать маленькие программы на Си, переводить их в код ассемблера и смотреть, что получилось.

Начнём с простой задачи. Имеется регистр eax. Нужно его обнулить.

Как понять, выполнили ли мы задачу?

Сделаем маленькую функцию foo на языке ассемблера, вызывать которую будет обычная функция main.

Нам потребуются два файла: foo.s и main.c.

Вот файл main.c:

```
// main.c
#include <stdio.h>

extern int foo();
int main() {
    printf("%d\n", foo());
}
```

и файл `foo.s`:

```
; foo.s
    .text
    .globl  foo
foo:
    ;
    ; Здесь будет наш код.
    ;
    ret
```

Файл `main.c` будет посредником между языком ассемблера и операционной системой — нам пока непонятно, как что-то выводить из программы на ассемблере (да и пока совсем ничего не понятно).

Мы видим в этом простейшем файле `foo.s` строки нескольких типов:

- всё, что содержится после точки с запятой в программе на ассемблере — комментарии.
- Слова, начинающиеся с точки (`.text`, `.globl`) — *директивы* компилятора. Первая директива просит компилятор всё, что будет генерировать с этого момента, направлять в *секцию* с кодом программы, она называется `.text`. Директива `.globl` приказывает компилятор пометить заданный адрес внутри секции как *глобальный*, то есть тот, к которому могут обращаться извне.
- После слова *foo* следует двоеточие. Это — объявление *метки*, поименованного адреса.
- строка `ret` — единственная (пока) машинная команда в файле.

В секцию `.text` попадает машинный код (команды). Иногда в эту секцию попадают и глобальные константы, например, строки `Си`. Иногда для строк заводится своя секция `.strings`. Иногда константы помещаются в секцию `.const`. Сейчас это нам не важно.

В секцию `.data` попадают глобальные и статические переменные `Си`/`C++`, имеющие в начале программы ненулевые значения.

В секцию `.bss` попадают глобальные и статические переменные, имеющие в начале программы нулевые значения.

7.6.3 Компиляция

Как с этим работать? Мы будем начинать со знака доллара те строки, которые будем вводить в командной строке.

```
$ cc main.c foo.s
$ ls -l
total 36
-rwxr-xr-x 1 root root 25184 мар  9 06:13 a.out
```

```
-rw-r--r-- 1 root root    57 мар  9 06:12 foo.s
-rw-r--r-- 1 root root    81 мар  9 06:12 main.c
$
```

Появился исполнимый файл `a.out`, операции компиляции и сборки программы в единое целое удались.

Нашу функцию `foo` вызовет `main` и то, то функция возвратила, будет распечатано.

7.6.4 Инструкции и флаги

А что в ассемблере означает *функция возвратила*? Где она должна оставить своё значение, чтобы им воспользовалась функция `main`? И вообще, мы сказали, что регистров, которые можно использовать, 16 штук. Как ими распоряжаться?

В любой операционной системе сегодня в памяти находятся сотни исполняемых процессов, большинство из них системные. Количество процессов обычно намного больше, чем число процессоров, которые их могут исполнять. Поэтому происходит *квантование* — из тех процессов, которые ничего не ждут и готовы исполняться, операционная система устраивает *очередь*. Процесс исполнялся *квант* времени (скажем, 10 миллисекунд) — у него отбирается процессор и отдаётся следующему ожидающему. К счастью, когда управление переходит от нашего процесса к другому, операционная система сохраняет все регистры в область памяти, называемую *контекстом процесса*. Дали процессу исполняться дальше — регистры восстановились. Это абсолютно прозрачно, поэтому можно сказать, что регистры — глобальные объекты для нашего процесса.

> Для возвращаемого значения из целочисленной функции используется регистр `rax`.

Если в нашем ассемблерном коде мы обнулим этот регистр, то `main` должен распечатать нуль.

А как его обнулить? Оказывается, это можно сделать большим количеством способов и все немного отличаются.

Проще всего, видимо, просто положить в регистр непосредственное значение.

```
movl %0, %eax
```

Мы можем попробовать сделать так, и убедиться, что всё работает.

```
; foo.s
    .text
    .globl  foo
```

```
foo:
    movl %0, %eax
    ret

$ cc main.c foo.s
$ ./a.out
$ 0
$
```

Нужно ли искать другие решения? Попробуем. Можно, например, вычесть из регистра `eax` значение ... этого же регистра.

```
subl %eax, %eax
```

А можно воспользоваться и логическими операциями. Например, сделать побитовое `AND` с нулём:

```
andl $0, %eax
```

А можно и побитовое `XOR` с самим собой.

```
xorl %eax, %eax
```

Зачем столько возможностей? Действительно эти команды делают одно и то же? Оказывается, они немного различаются. В регистре `eax`, действительно, оказывается нуль. Но это не всё. Есть ещё один регистр, который изменяется при исполнении операций. Именно этот регистр отвечает за все условия, которые могут возникнуть при исполнении инструкции. Это регистр `eflags`. Мы не можем присвоить этому регистру конкретное значение, таких команд нет. Но при исполнении инструкций отдельные биты этого регистра могут изменяться и другие команды могут использовать изменение этих бит. Не все биты этого регистра нам интересны, отметим некоторые из них:

Некоторые биты регистра eflags:
- `O` — произошло арифметическое переполнение.
- `S` — результат операции отрицательный.
- `Z` — результат операции нулевой.
- `C` — произошёл перенос разряда в бит за пределами разрядной сетки.

Почему нам важны эти флаги? По ним мы принимаем решение, исполнять ли следующую инструкцию или перейти в другое место.

Какие флаги изменяются при исполнении команды? Об этом нужно читать в документации на команду! При описании каждой команды перечисляются флаги, которые она может изменить. Попробуем посмотреть, что изменяется в наших командах:

```
    .text
    .globl  foo
foo:
    movl    $0, %eax            ; no flags affected
    subl    %eax, %eax          ; flags O,S,Z,C according result
    andl    $0, %eax            ; O,C <= 0. S,Z according result
    xorl    %eax, %eax          ; O,C <= 0. S,Z according result
    ret
```

7.6.5 Дизассемблирование

Теперь интересно посмотреть, какие значения имеют машинные команды этих инструкций? Как получить машинный код?

Для этого проще всего воспользоваться *утилитой* objdump. Но перед этим нужно сформировать *объектный* файл.

```
$ cc -c foo.s
$ ls -l
total 40
-rwxr-xr-x 1 root root 25184 мар  9 07:08 a.out
-rw-r--r-- 1 root root   704 мар  9 08:03 foo.o
-rw-r--r-- 1 root root   171 мар  9 08:03 foo.s
-rw-r--r-- 1 root root    81 мар  9 06:12 main.c
$
```

Файл `foo.o`, который образовался при этой операции, содержит *объектный код*, в котором есть и машинные команды, и описания меток, и описания сегментов. Мы можем получить *дизассемблер* сегмента кода этого файла:

```
$ objdump -d foo.o

foo.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foo>:
   0: b8 00 00 00 00        mov    $0x0,%eax
   5: 29 c0                 sub    %eax,%eax
   7: 83 e0 00              and    $0x0,%eax
   a: 31 c0                 xor    %eax,%eax
   c: c3                    retq
$
```

Вся адресация внутри объектного файла начинается с нуля. В процессе *сборки* программы (другой термин — *линковка*) объектный файл будет помещён на нужное место и все адреса в нём получат правильные значения.

Опять же, интересно, что похожие инструкции могут кодироваться различным числом байт. Посмотрите на инструкции `mov` и `and`. Какая-то часть команды — код операции, какая-то — аргументы. Каким образом кодировать аргументы, зависит от команды. Пока можем заметить, что, как будто, первый байт команды — её код. Проверим? Присвоим разным регистрам разной длины ноль.

```
0000000000000000 <foo>:
   0: b0 00                 mov    $0x0,%al
   2: b4 00                 mov    $0x0,%ah
   4: b3 00                 mov    $0x0,%bl
   6: b7 00                 mov    $0x0,%bh
   8: 66 b8 00 00           mov    $0x0,%ax
   c: 66 bb 00 00           mov    $0x0,%bx
  10: b8 00 00 00 00        mov    $0x0,%eax
  15: bb 00 00 00 00        mov    $0x0,%ebx
  1a: 48 c7 c0 00 00 00 00  mov    $0x0,%rax
  21: 48 c7 c3 00 00 00 00  mov    $0x0,%rbx
  28: c3                    retq
```

Интересно, что для работы с байтами каждая команда — своя. А вот для работы с 16-битными регистрами процессор переключает на время, на одну команду, исполнение в 16-битный режим. Это — *команда-префикс* с кодом `0x66`. Непосредственный операнд у команды с таким префиксом занимает 2 байта, а у такой же команды без префикса — 4 байта.

Мы видим и второй префикс — `0x48`. Он сообщает, что в этой машинной команде регистры будут 64-битными. Исследуем дальше.

Ассемблерный код

```
    .text
    .globl  foo
foo:
    movq $0x12,%rax
    movq $0x1234,%rax
    movq $0x123456,%rax
    movq $0x12345678,%rax
    movq $0x123456789A,%rax
```

```
movq $0x123456789ABC,%rax
movq $0x123456789ABCDE,%rax
movq $0x123456789ABCDEF0,%rax
ret
```

превращается в

```
0000000000000000 <foo>:
   0: 48 c7 c0 12 00 00 00  mov    $0x12,%rax
   7: 48 c7 c0 34 12 00 00  mov    $0x1234,%rax
   e: 48 c7 c0 56 34 12 00  mov    $0x123456,%rax
  15: 48 c7 c0 78 56 34 12  mov    $0x12345678,%rax
  1c: 48 b8 9a 78 56 34 12  movabs $0x123456789a,%rax
  23: 00 00 00
  26: 48 b8 bc 9a 78 56 34  movabs $0x123456789abc,%rax
  2d: 12 00 00
  30: 48 b8 de bc 9a 78 56  movabs $0x123456789abcde,%rax
  37: 34 12 00
  3a: 48 b8 f0 de bc 9a 78  movabs $0x123456789abcdef0,%rax
  41: 56 34 12
  44: c3                    retq
```

Программа `objdump` не желает распечатывать машинные команды, занимающие более 8 байт, в одну строку. Тем не менее, мы видим, что:

1. существуют и инструкции с непосредственными операндами длиной 8 байт;
2. непосредственные операнды кодируются от младшего разряда к старшему;
3. код смены разрядности `0x48` никуда не делся.

7.6.6 Арифметические инструкции

Сложение/вычитание. Мы отличаем арифметические инструкции, которые рассматривают содержимое регистра как число, от логических, для которых регистры — просто набор бит. Арифметических инструкций не так уж и много.

Например, инструкция

```
addq $10,%r8
```

добавит 10 к тому, что было в регистре r8. Изменятся все интересные нам флаги, в зависимости от результата. Например, если регистр `r8` станет равным нулю, будет установлен флаг `Z` иначе он будет сброшен. Флаг `C` установится,

если регистр r8 перед операцией находился в диапазоне $[-10 \dots -1]$. Действительно, именно в этом случае произойдёт перенос разряда в 64-й бит. Флаг O установится, если регистр r8 перед операцией содержал в самом старшем бите 0, а после операции стал содержать 1 и наоборот.

Подобным образом работает инструкция sub, вычитания. И add, и sub работают и со знаковыми, и с беззнаковыми числами.

В таких инструкциях нельзя смешивать регистры разной разрядности, например, rdx и esi. Их можно смешивать в операциях преобразования типов, о которых будет рассказано дальше.

В примерах мы будем оперировать 8-ми битными операндами.

addb %ah,%cl								
	7	6	5	4	3	2	1	0
ah	1	0	1	0	0	1	0	0
cl	1	1	0	0	1	0	1	1
cl	0	1	1	0	1	1	1	1
Z=0		S=0		C=1		O=1		

После сложения регистр-приёмник не равен нулю (Z=0), неотрицательный в дополнительном коде (S=0), произошёл перенос в 8-й бит (C=1, произошло переполнение при знаковом сложении: регистр ah был равен -92, регистр cl был равен -53, результат оказался равен 111.

Ещё одна полезная инструкция — увеличения аргумента на единицу — inc. Парная инструкция уменьшает аргумент на единицу и называется dec.

```
0000000000000000 <foo>:
   0: fe c0                 inc    %al
   2: 04 01                 add    $0x1,%al
   4: 66 ff c0              inc    %ax
   7: 66 83 c0 01           add    $0x1,%ax
   b: ff c0                 inc    %eax
   d: 83 c0 01              add    $0x1,%eax
  10: 48 ff c0              inc    %rax
  13: 48 83 c0 01           add    $0x1,%rax
  17: 49 ff c0              inc    %r8
  1a: 49 83 c0 01           add    $0x1,%r8
  1e: ff cb                 dec    %ebx
  20: 83 eb 01              sub    $0x1,%ebx
  23: c3                    retq
```

Опять мы видим 16-разрядный префикс `0x66` и 64-разрядный префикс `0x48`. Добавился префикс `0x49` для возможности оперировать с регистрами `r8-r15`. Команды `add` и `sub`, в которых есть непосредственный операнд, обычно несколько длиннее соответствующих `inc` и `dec`. Есть и другое отличие: `inc` и `dec` не изменяют флага `C`.

Умножение. Инструкций умножения несколько, имеются и знаковый вариант `imul`, и беззнаковый `mul`. Они отличаются не только по названию, но не будем сейчас увлекаться деталями и рассмотрим первый вариант.

Сколько двоичных разрядов может потребоваться для результата *сложения* 32-битных чисел? 33 разряда. При сложении ничего не теряется, самый старший разряд сохраняется в флаге `C`.

А сколько двоичных разрядов нужно для умножения 32-битных чисел? Очевидно, 64 разряда. Флагов на всех не хватит, поэтому для хранения старших 32 битов результата используют регистр. Когда нужен результат именно 64-битный, полный, применяется форма с одним операндом.

```
imul %rsi
```

Интересно, что на что был умножен регистр `rsi`? Одноперандный вариант умножения категоричен: это только регистр `rax`. Результат окажется в паре регистров: `rdx:rax`. В `rax` окажутся младшие 64 бита произведения, в регистре `rdx` — старшие 64 бита. Двухоперандный вариант более похож на операции сложения/вычитания:

```
imul %rsi,%rcx
```

Здесь всё по-старому: в регистре `rcx` — результат умножения его на регистр `rsi`.

Команда беззнакового умножения `mul` имеет только одноперандный вариант.

Ещё имеется и трёхоперандный вариант! Но он может что-то умножать только на константу.

```
movl    $12345679, %edi
imul    $9,%edi,%eax
```

В регистре `eax` появится результат умножения `edi` на 9.

С флагами всё интересно. Меняются только флаги `C` и `O`, остальные не определены. Значения флагов зависят от того, что должно содержаться в старшей половине. Они устанавливаются, если в ней что-нибудь появилось, отличное от нулей. Просто в одноперандном варианте старшая половина сохраняется, в остальных — отбрасывается.

Пример: умножим -92 на 55

imul %cl								
	7	6	5	4	3	2	1	0
al	1	0	1	0	0	1	0	0
cl	0	0	1	1	0	1	1	1
ah	1	1	1	0	1	1	0	0
al	0	0	1	1	1	1	0	0
Z=?		S=?		C=1		O=1		

Флаги `C` и `O` приняли значение 1, так как результат не поместился в один регистр.

Деление. Команд деления тоже две — знаковая и беззнаковая. Но форма у них одна — однооперандная. Это означает, что делимое должно находиться на строго определённом месте — в паре регистров `dx:ax`, соответствующей длины. Если мы хотим разделить отрицательное число на что-то, в старшем регистре должно быть отрицательное значение.

Предположим, мы захотели разделить нечто со знаком, находящееся в регистре `rcx` на содержимое регистра `rsi`.

```
movq %rcx,%rax
cdq
idiv %rsi
```

Инструкция `cdq` копирует знаковый бит из регистра `rax` и заполняет его значением регистр `rdx`. Убедитесь, что получившееся 128-битное число имеет то же самое значение в дополнительном коде, что и исходное 64-битное. Если нужно разделить беззнаково — регистр `rdx` достаточно обнулить.

После операции деления в младшем регистре — частное, в старшем — остаток.

Впрочем, знаковое деление отрицательных чисел в рассматриваемой нами системе команд противоречит всем математическим принципам. Считается, что при делении отрицательного числа на положительное, знак остатка должен быть отрицательным! До свидания, арифметика вычетов! $-5 \pmod 2$ равен не 1, как в теории чисел, а -1. Поэтому будьте осторожны!

Ещё несколько нюансов: все флаги после команды деления становятся неопределенными. Если результат деления не помещается в отведённый регистр (или делитель равен нулю), то возникает специальная ситуация процессора *деление на ноль*, при котором будет *возбуждено исключение* соответствующего типа. Мы пока про исключения ничего не знаем, хотя они — важная часть архитектуры, и будем считать, что процесс просто аварийно завершится.

Пример: разделим 25 на 8.

idiv %cl								
	7	6	5	4	3	2	1	0
ah	0	0	0	1	1	0	0	1
al	0	0	0	0	0	0	0	0
cl	0	0	0	0	0	1	0	0
ah	0	0	0	0	0	0	0	1
al	0	0	0	0	0	0	1	1
Z=?		S=?		C=?		O=?		

Перед делением помещаем 25 в `al`, `ah` обнуляем. В `cl` помещаем 8. Результат: в регистре `ah` — остаток, равный 1, в регистре `al` — частное, равное 3. Все флаги стали неопределёнными.

7.6.7 Логические инструкции

Их много. Во-первых, это три классические двуместные инструкции алгебры логики: `or`, `and` и `xor`. Все эти инструкции очищают флаги `C` и `O`, устанавливая остальные по значению результата. В дополнение к ним идёт одноместная `not`, которая меняем каждый бит своего аргумента на противоположный. Она никакие флаги не трогает.

Имеется несколько инструкций сдвига на несколько бит. Инструкции *логического* сдвига сдвигают либо влево `shl` либо вправо `shr` на указанное число бит. Логический сдвиг вправо заполняет *освободившиеся* биты нулями, *арифметический* — знаком.

Имеются также команды *циклического* сдвига, которые не изменяют количество единичных и нулевых битов в числе, *поворачивая* их на нужное количество налево или направо. Проще рассмотреть несколько примеров:

```
shll    $4, %esi      ; esi <<= 4
shrl    $5, %esi      ; esi >>= 5
andl    $255, %esi    ; esi &= 255
orl     $8192, %esi   ; esi |= 8192
negl    %esi          ; esi = ~esi --- побитовое НЕ
shll    %cl,%eax      ; сдвинуть на cl & 0x1F бит влево
shlq    %cl,%r8       ; сдвинуть на cl & 0x3f бит влево
sall    $5,%edx       ; сдвинуть на 5 вправо.
```

Внимание! При сдвиге на содержимое регистра `cl` используются не все биты регистра! Для 32-х и меньше разрядных регистров сдвижка производится на значение в младших 5 битах регистра `cl`. Для 64-х разрядных — на младшие 6 бит.

Эти инструкции легки тем, что, при соблюдении правил можно и предсказать код, который будет сгенерирован компилятором Си, и после реализации алгоритма на Си можно его легко перевести на язык ассемблера.

7.7 Адресация памяти. Указатели

Мы до сих пор работали только с регистрами. Большое количество команд в качестве одного из операндов готовы принять и ячейку в памяти. Для этого в команде должно быть *адресное выражение*. Почему *выражение*, а не просто адрес в виде номера ячейки или первого байта ячейки?

Во-первых, современные процессоры — 64-битные. Использование каждый раз 64 бит для адреса в команде — роскошь, которая приведёт к очень большому коду.

Во-вторых, имеется много рекурсивных алгоритмов, то есть таких, что имеется несколько экземпляров вызова одной и той же функции и, соответственно, несколько экземпляров каждой локальной переменной и аргументов. Невозможно им заранее назначить адреса и эти адреса будут меняться[10].

Таким образом, адреса сейчас редко (а в 64-битном коде — никогда) появляются в виде констант в машинных командах. Основной метод работы с адресами — *относительная адресация*. Во время исполнения программы известны значения каких-то регистров. Например, в данной конкретно исполняемой инструкции известен адрес этой инструкции — он находится в регистре `rip`. Ассемблеру известно место, где располагаются данные. Если вычислить разницу между ними и использовать её в машинной команде, то данные можно легко найти, сократив код этой команды.

Давайте посмотрим на код.

```
// file g.c
int g;

void foo() {
    g = 12345678;
}
```

Откомпилируем и посмотрим код. Мы уберём из вывода компилятора незначащие для нашей задачи строки. То, что получите вы при компиляции, будет отличаться от нашего вывода именно на такие строки.

```
$ gcc -c g.c
```

[10]Между прочим, в первом промышленном языке программирования, FORTRAN рекурсия была запрещена. Для чего? Для того, чтобы адреса всех переменных были известны после компиляции и до начала исполнения — тогда это было возможно, адреса были короткими, и даже необходимо, так как на тех процессорах отсутствовала относительная адресация.

Мы получаем следующий код:

```
01     .globl  g
02     .bss
03 g:
04    .zero   4
05    .text
06    .globl  foo
07 foo:
08     movl    $12345678, g(%rip)
09     ret
```

В коде на Си имеется глобальная переменная g. Компилятор поместил её в сегмент bss (строка 02), назначил ей метку g (строка 03) и зарезервировал 4 байта, равные нулю (строка 04). При этом он установил ей атрибут globl для того, чтобы к этой переменной могли обращаться из других объектных файлов.

Далее компилятор открыл секцию с кодом (строка 05), завёл глобальную метку foo (строки 06 и 07) и присвоил переменной g значение 12345678. В качестве адреса он не использовал саму метку g, которая у нас имеет длину в 64 бита, а *адресное выражение* g(%rip). Если в команде присутствует какой-то регистр в скобках — мы увидели адресное выражение.

Посмотрим на дизассемблер:

```
0: c7 05 00 00 00 00 4e  movl   $0xbc614e,0x0(%rip)  # a <foo+0xa>
7: 61 bc 00
a: c3                    retq
```

В правой части машинной команды мы видим 4e 61 bc 00 — константу, которую мы присваиваем. А до неё идёт код команды c7. Это один из кодов команда mov. Следующий байт команды — 05 — описывает режим адресации. В данном случае это косвенная адресация относительно регистра rip с добавленной константой. Следующие 4 байта — нули. Странно? Нет. Сейчас неизвестны ни адрес, который будет назначен переменной g после сборки программы, ни адрес функции foo. Пока эти байты — placeholder, зарезервированное место, которое будет заполнено сборщиком по информации, которая содержится в объектном файле.

Чтобы увидеть, что сделает сборщик (*линкер*), переименуем функцию foo в main, чтобы сборка дошла до конца и посмотрим на результат:

```
0000000000401106 <main>:
401106: c7 05 10 2f 00 00 4e  movl   $0xbc614e,0x2f10(%rip) # 404020 <g>
40110d: 61 bc 00
401110: b8 00 00 00 00        mov    $0x0,%eax
401115: c3                    retq
```

Линкер назначил переменной g адрес 0x404020. Адрес инструкции, следующей за исполняемой, 0x401110. В машинную команду попал код *смещения*, равный 0x404020-0x401110=0x2f10.

Чуть изменим исходник на Си:

```
int g;

int main() {
    g = 12345678;
    g = 87654321;
}
```

```
0000000000401106 <main>:
401106: c7 05 10 2f 00 00 4e  movl $0xbc614e,0x2f10(%rip) # 404020 <g>
40110d: 61 bc 00
401110: c7 05 06 2f 00 00 b1  movl $0x5397fb1,0x2f06(%rip) # 404020 <g>
401117: 7f 39 05
40111a: b8 00 00 00 00        mov  $0x0,%eax
40111f: c3                    retq
```

> Регистр rip в команде movl $0xbc614e,0x2f10(%rip) является *базовым регистром*, а константа $0xbc614e — *смещением*.

Обратите внимание что, несмотря на то, что две машинных команды обращаются к одной и той же переменной, поле смещения у них разное!

Глобальные переменные — дело нечастое в современных программах. Их использование мешает многопоточному программированию. А для переменных локальных переменных и переменных, полученных вследствие заказа памяти, применяется адресация с помощью регистров. В качестве базового регистра для переменных в стеке применяется обычно два регистра — rsp и rbp. После операций заказа памяти calloc/new возвращается указатель на выделенную память. Этот указатель может храниться в какой угодно памяти, но для того, чтобы получить доступ к заказанной памяти, его используют в качестве базового регистра.

Несмотря на то, что инструкции изменения памяти существуют, современные компиляторы их не любят. Не потому, что они неудобные для использования, а потому, что они обычно исполняются медленнее, чем более длинные последовательности кода, использующие операции загрузки из памяти в регистр, выполнение операции и выгрузки из регистра в память.

Давайте посмотрим, сколько времени займут как будто эквивалентные программы. Возьмём типичный процессор Intel 2020 года. Буквой l обозначается латентность, то есть количество тактов от начала исполнения команды до её

завершения (*выпуска*, `issue`). Буква `t` означает в первом приближении обратную величину темпа исполнения инструкций. `t=0.25` означает, что таких инструкций может быть исполнено 4 штуки за один такт.

```
movq    8(%rbp),%rax    ; l=2 t=0.5
addq    $10,%rax        ; l=1 t=0.25
movq    %rax,8(%rbp)    ; l=2 t=1

addq    $10,8(%rbp)     ; l=5 t=1
```

Какое количество тактов исполняется первая последовательность? Трудно сказать точно — ведь имеется конвейер. В любом случае, если каждая команда будет дожидаться предыдущей, то больше 5 тактов на это не уйдёт. Мы сейчас не учитываем время реальной пересылки в память/из памяти, так как оно зависит от того, находится ли ячейка в кэше, и в кэше какого уровня. Это — задержка в идеальном случае!

Вторая последовательность в любом случае не выполнится быстрее 5 тактов.

Вроде бы, как первая последовательность длиннее. Но! После исполнения первой последовательности мы уже имеем в регистре `rax` копию переменной! Если значение этой переменной мы будем использовать вскоре ещё раз, мы не должны будем терять время (и машинные команды) на её загрузку из памяти. Так как компилятор способен отследить все использования регистров и переменных, он старается минимизировать операции загрузки из памяти и он в большинстве случаев выберет первую последовательность.

Поэтому давайте сосредоточимся на инструкциях загрузки из памяти в регистр и выгрузки из регистра в память и не будем применять операции над памятью, если это явно не понадобится.

В простом примере, которые мы только что видели, адрес операнда `8(%rbp)` равен `8 + rbp`.

Кроме базового регистра в вычислении адреса может использоваться и *индексный*. В простейшем случае его содержимое прибавляется к базовому регистру.

```
movq    $120, %rax
movl    -160(%rbp,%rax), %rbx
```

На что это похоже? На получение значения по индексу в массиве! Адрес массива определяется здесь базовым регистром и смещением, индекс в массиве — индексным регистром. Так как размер элемента — 4, то мы извлекаем элемент 30 из массива. В цикле регистр `rax` будет изменяться, например, на 4, а начало массива — нет.

Можно и немного по-другому, если применять *масштабирование* индекса.

```
movq    $30, %rax
movl    -160(%rbp,%rax,4), %rbx
```

Во второй команде адрес памяти, по которому произойдёт обращение, равен `-160+rbp+4*rax`. Третий операнд в адресном выражении может принимать значения 1, 2, 4 и 8.

Итак, процессор может вычислять достаточно сложные адресные выражения. Ещё одна любопытная команда использует этот же синтаксис, но обращения к памяти не производит! Это команда `lea`, Load Executive Address. Вычисленный адрес (не значение по адресу!) копируется в регистр. Вот несколько примеров:

```
leaq    (%rbx),%rsi          ; rsi = rbx
leaq    5(%rbx),%rsi         ; rsi = 5+rbx
leaq    (%rbx,%rax),%rdx     ; rdx = rbx + rax --- сложение
leaq    10(%rax,%rdx),%rdx   ; rdx += rax + 10
leaq    (%rax,%rbx,4),%rdx   ; rdx = rax + 4*rbx
leaq    (%rax,%rax,2),%rdx   ; rdx = rax * 3
leaq    (%rax,%rax,4),%rdx   ; rdx = rax * 5
leaq    (%rax,%rax,8),%rdx   ; rdx = rax * 9
leaq    4(%rax,%rax,2),%rdx  ; rdx = rax * 3  + 4
```

То, что можно было бы сделать в несколько инструкций, получается сделать одной. Видите, наряду с умножением на степени двойки, которые можно было совершить сдвигом влево, есть быстрое умножение на 3,5 и 9, с возможным добавлением константы.

7.8 Переходы

Флаги, о которых мы так долго говорили, наконец-то будут использоваться!

Давайте решим простую задачу: *Если регистр* **`ax`** *равен регистру* **`dx`**, *поместить в регистр* **`dx`** *единицу, а если не равен — ноль.*

Основная, но не единственная команда сравнения так и называется — `cmp`. Она вычитает первый аргумент из второго (не изменяя операндов) и устанавливает флаги.

Сами флаги могут использоваться в командах перехода. Они начинаются на `j` (в документации — `Jcc`) и имеют много вариантов, несколько из которых мы приведём. Общая идея состоит в том, что при каких-то установленных флагах может произойти *переход* — передача управления на метку в инструкции. Если переход не произошёл — команда игнорируется. Вот примеры:

```
jeq  zero       ; переход, если ==   Z=1
jne  nonzero    ; переход, если !=   Z=0
jl   less       ; переход, если signed <    S!=O
```

```
    jle  lesseq     ; переход, если signed <=   Z=1 и S!=0
    jg   greater    ; переход, если signed >    Z=0 и S==0
    jge  greatereq  ; переход, если signed >=   S==0
    jb   uless      ; переход, если unsigned <  C==1
    jbe  ulesseq    ; переход, если unsigned <= C==1 и Z=-1
    ja   ugreater   ; переход, если unsigned >  C==0 и Z==0
    jae  ugreatereq ; переход, если unsigned >= C==0
```

Обычно не требуется запоминать все реакции на флаги, мнемоника команд подсказывает правильное решение.

Вернёмся к задаче.

Вариант 1. Первая команда очевидна — `cmpw %ax,%bx`. А вот что делать дальше? Давайте перескочим, если регистры равны на метку, скажем, `zero`. Тогда после метки `zero` можно обнулить регистр `dx`. А если не равны? Нужно установить в `dx` единицу. А потом что? Мы должны *слить* две ветви сравнения так, чтобы выполнился код после нашего. Но, если ничего не сделать, мы наткнёмся на метку `zero` и регистр `dx` обнулится. Значит, и и метку, и что следует за ней надо перепрыгнуть — совершить *безусловный* переход.

```
    cmpw    %ax,%bx
    je      zero
    movw    $1,%dx
    jmp     end
zero:
    xorw    %dx,%dx
end:
```

Какая проблема в этом коде? Он не очень эффективный. В любом случае хотя бы один переход произойдёт — и при равенстве, и при неравенстве. А ведь помните, мы говорили о том, что переходы плохо влияют на конвейер.

Вариант 2. Чтобы избавиться от одного перехода можно схитрить. Будем оптимистами и предположим, что переменные чаще всего *не* равны. Тогда, чтобы переход не произошёл, мы вначале присвоим `dx` единицу, а затем проверим условие. Если они неравны, то перехода не должно произойти. Ну а если они равны, то перейдём.

```
    movw    $1,%dx
    cmpw    %ax,%bx
    je      end
    movw    $0,%dx
end:
```

Теперь переход может вообще не произойти и это чаще, чем переход будет происходить (мы знаем, что равенство будет редким. А если мы знаем, что переменные будут чаще равны, то можно инвертировать условие и присваивания:

```
    movw    $0,%dx
    cmpw    %ax,%bx
    je      end
    movw    $1,%dx
end:
```

Можно немного оптимизировать код, воспользовавшись той идеей, что современные процессоры могут исполнять команды вне очереди, если те не зависят от результатов предыдущих. В нашем случае команда `movw $0,%dx`, очевидно, выполнится быстро, а вот команда сравнения `cmpw %ax,%bx` может немного задержать конвейер, так как следующая команда зависит от неё. А что если их поменять местами?

```
    cmpw    %ax,%bx
    movw    $0,%dx
    je      end
    movw    $1,%dx
end:
```

Тогда, когда `cmpw %ax,%bx` всё ещё будет исполняться, команда `movw $0,%dx` может уже исполниться. Почему это корректно? Потому, что команда `mov` на меняет флаги! А вот если бы мы обнуляли регистр арифметическими или логическими командами, то результат выполнения был бы неверным — флаг `Z` установился бы.

Компиляторы автоматически делают такую оптимизацию за нас.

Вариант 3. Впрочем, переход всё равно может произойти. Для того, чтобы этого избежать, имеется семейство команд, `SETcc`. Аргумент команды — 8-ми битный регистр. Если условие, аналогично условию перехода, соблюдено — регистр устанавливается в 1, иначе в 0. Нам повезло, нам и нужны 0 или 1. Поэтому:

```
    movb    $0,dh
    cmpw    %ax,%bx
    sete    %dl
```

Если `ax` и `bx` равны, в `dl` будет единица. А так как в старших 8 битах регистра `dx` мы уже положили 0, то и весь регистр станет равным 1.

Вариант 4. Предыдущий вариант отлично подходит для присвоения логических значений. Но если нужно присвоить что-то, отличное от 0 и 1, он не очень подходит. Имеется ещё одна группа команд условного присваивания: `CMOVcc`. Это условное присвоение одного регистра другому. Если условие сработало — происходит присвоение. Не сработало — команда игнорируется.

```
cmoveqq %rax,%rbx  ; rbx = rax, если ==   Z=1
cmovnq  %rax,%rbx  ; rbx = rax, если !=   Z=0
cmovq   %rax,%rbx  ; rbx = rax, если signed <    S!=O
cmovlq  %rax,%rbx  ; rbx = rax, если signed <=   Z=1 и S!=O
cmovq   %rax,%rbx  ; rbx = rax, если signed >    Z=0 и S==O
cmovgq  %rax,%rbx  ; rbx = rax, если signed >=   S==O
cmovq   %rax,%rbx  ; rbx = rax, если unsigned <  C==1
cmovbeq %rax,%rbx  ; rbx = rax, если unsigned <= C==1 и Z=-1
cmovaq  %rax,%rbx  ; rbx = rax, если unsigned >  C==0 и Z==O
cmovaeq %rax,%rbx  ; rbx = rax, если unsigned >= C==0
```

Применим это к нашей задаче:

```
movw    $0,dx
cmpw    %ax,%bx
movw    $1,%bx
cmove   %bx,dx
```

Здесь мы воспользовались нашей идеей: команда `mov` не меняет флаги. Поэтому инструкция `cmove` заберёт флаги, установленные командой `cmpw`. При равенстве регистров произойдёт присвоение, при неравенстве содержимое регистра останется.

Впрочем, заметили ли вы, что мы нарушили условие задачи? Мы поменяли регистр `bx`, хотя условие это запрещает. Что же, хотя мы задачу не решили, но узнали новую команду.

7.8.1 Функции. Аргументы функции

7.8.2 Стек и регистр стека

7.8.3 Фрейм вызова

7.8.4 Локальные переменные

7.9 Системные вызовы

Системный вызов — способ для программы каким-либо образом взаимодействовать с окружающим миром. Программа, не исполняющая ни одного системного вызова бесполезна — она не сможет ни напечатать результаты вычислений, ни передать что-нибудь по сети.

Для того, чтобы совершить системный вызов в linux требуется занести в определённые регистры соответствующие значения и выполнить команду *syscall*:

```
mov $1,%rax          #системный вызов
```

linux:

`/usr/include/asm/unistd_64.h`

7.10 Операции с вещественными числами

7.11 Векторные операции

8 Введение в reverse engineering

9 Введение в архитектуру aarch64

9.1 Язык ассемблера

9.2 Регистры

9.3 Адресация памяти

9.4 Основные арифметические и логические операции

9.5 Операции условного и безусловного перехода

9.6 Вызов функции

9.6.1 Регистр связи

9.6.2 Соглашения о регистрах при вызове. Аргументы функции

9.6.3 Использование стека

9.6.4 Фрейм вызова и локальные переменные

9.7 Операции с вещественными числами

9.8 Векторные операции

Предметный указатель