Московский физико-технический институт
(государственный университет)

Факультет аэромеханики и летательной техники

Архитектура компьютера и язык ассемблера

TEXнические материалы семинаров

2 мая 2024 г.

Жуковский, *2023–2024 уч. гг.*

Оглавление

Цели и задачи

Посмотреть, во что отображается программа на языке C при переходе на более низкий уровень [абстракции], какие действия выполняет компилятор в процессе сборки исполняемого модуля, что умеет непосредственно сам компьютер.

Познакомиться с важными аспектами современной архитектуры, которые необходимо учитывать при написании программ.

Архитектура Intel x86-64, синтаксис языка ассемблера AT&T.

Литература

Основная:

1. *Хэррис Д. М.*, *Хэррис С. Л.* Цифровая схемотехника и архитектура компьютера : пер. с англ. Morgan Kaufman, 2015. 1627 с.
2. *Брайант Р. Э.*, *О'Халларон Д. Р.* Компьютерные системы: архитектура и программирование. М. : ДМК Пресс, 2022. 994 с.

Дополнительная:

3. *Таненбаум Э.*, *Остин Т.* Архитектура компьютера : пер. с англ. СПб. : Питер, 2013. 816 с.
4. *Винокуров Н. А.*, *Ворожцов А. В.* Практика и теория программирования : в 2 т. М. : Физматкнига, 2008
5. *Pacheco P. S.* An introduction to parallel programming. Morgan Kaufmann Publishers, 2011. 370 p.

Материалы и задания

Будут размещены по мере необходимости на яндекс-диске[1] в каталогах:

- **books** — основная и дополнительная литература;
- **asm-lectures** — лекционные материалы;
- **asm-seminars** — семинарские материалы;
  - **/program.pdf** — программа курса, темы зачёта;
  - **/progress.pdf** — планы, успеваемость и проверочные мероприятия;
  - **/seminars.pdf** — вспомогательная методичка по материалам занятий.

Программное обеспечение

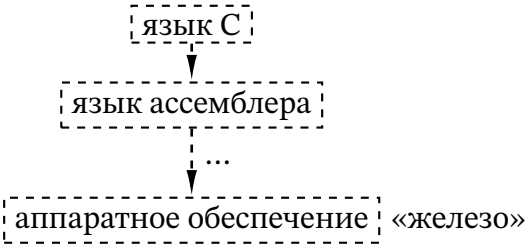
- текстовый редактор,
- компиляторы языков C/C++,
- ассемблер для архитектуры Intel x86-64, поддерживающий синтаксис AT&T,
- отладчик gdb,
- утилита objdump.

Инструкции по установке и настройке рабочей среды размещены в приложении на странице 81.

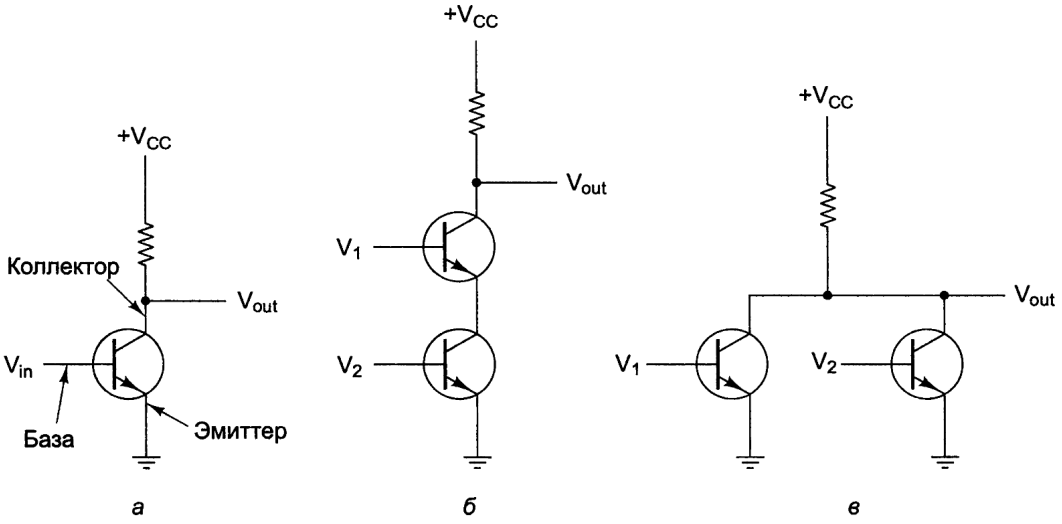
[1] https://yadi.sk/d/XiUj9ZsNf3xHJ

1 Вентили и булева алгебра

1.1. Многоуровневая организация ЭВМ



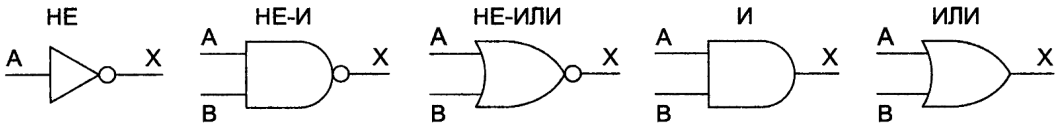
Вентили



Транзисторный инвертор (*а*); вентиль НЕ И (*б*); вентиль НЕ ИЛИ (*в*)

Цифровая схема — это схема, в которой есть только два логических значения: `0` или ложь (сигнал от 0 до 1 В) и `1` или `истина` (сигнал от 2 до 5 В).

Вентили — это электронные устройства, которые позволяют получать различные функции от цифровых (двузначных: `0` или `1`) сигналов. Вентили лежат в основе аппаратного обеспечения, на котором строятся все цифровые компьютеры.



1.2. Булевы функции

Алгебра релейных схем, *булева функция n переменных*, 2^n *значений*, *таблица истинности*.

Пример: $M = f(A,B,C)$ — функция большинства, которая принимает 0, если большинство переменных равны 0, и 1, если большинство переменных равны 1.

СДНФ = Совершенная Дизъюнктивная Нормальная Форма

$$M = \bar{A}BC + A\bar{B}C + AB\bar{C} + ABC$$

1. Составить таблицу истинности.
2. Включить инверторы для каждого входного сигнала.
3. Нарисовать вентиль И для строк таблицы со значением 1.
4. Соединить вентили И с соответствующими входными сигналами.
5. Вывести выходы всех вентилей И и направить на вход вентиля ИЛИ.

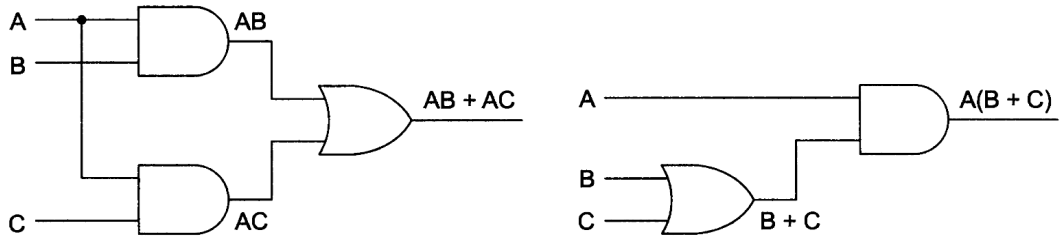
Любую булеву функцию можно реализовать при помощи вентилей НЕ, И и ИЛИ.

Что читать

Хэррис Д. М., *Хэррис С. Л.* глава 1, стр. 2–23, 50—106
Таненбаум Э., *Остин Т.* глава 1, стр. 20–26, глава 3, стр. 172–182

Упражнения

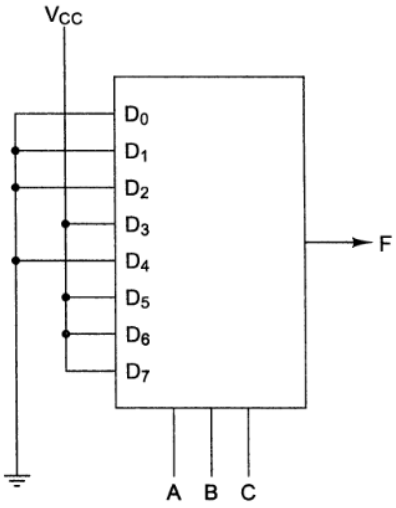
1. Постройте схему для вычисления булевой функции ИСКЛЮЧАЮЩЕЕ-ИЛИ (XOR).
2. Реализуйте вентили НЕ, И и ИЛИ на основе только вентилей НЕ-ИЛИ.
3. Реализуйте вентили НЕ, И и ИЛИ на основе только вентилей НЕ-И.
4. Докажите эквивалентность следующих схем



2 Основные цифровые логические схемы

2.1. Мультиплексор

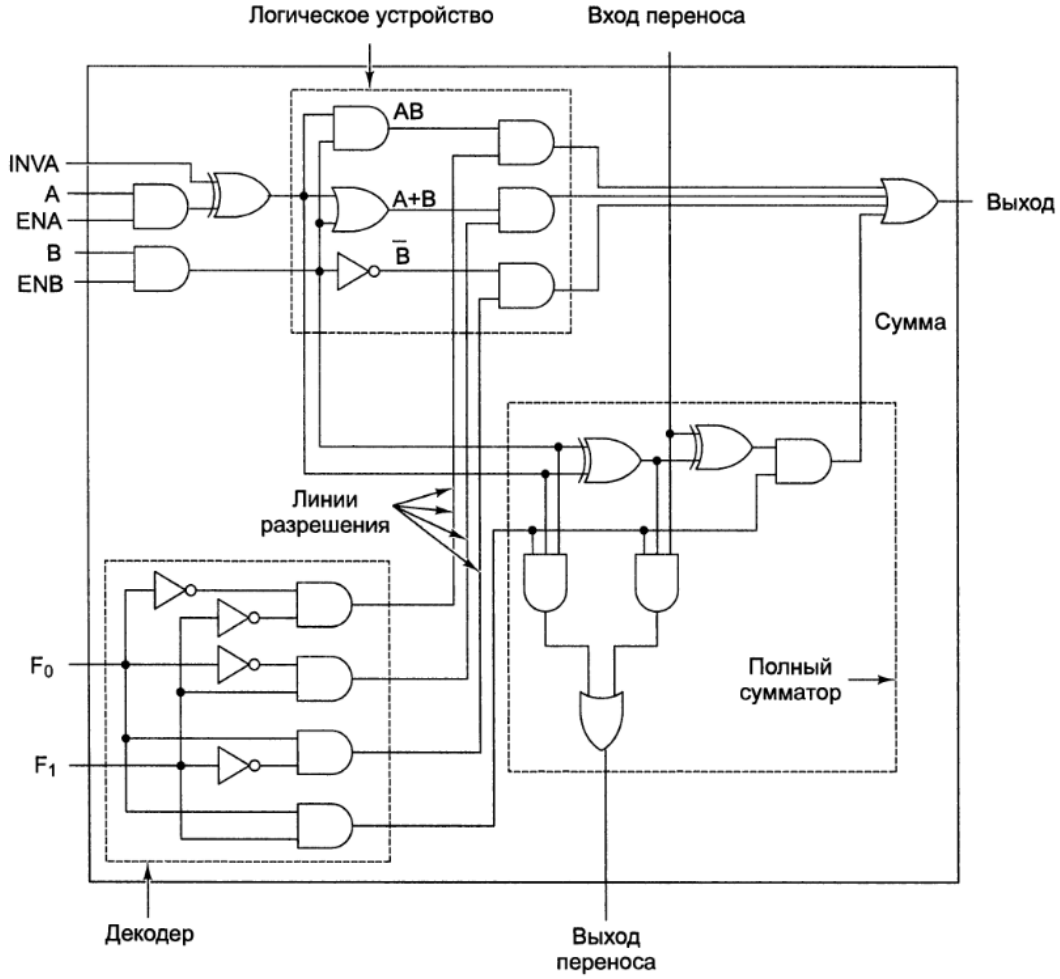
Схема с 2^n входами, одним выходом и n линиями управления, которые позволяют выбрать один из входов. Мультиплексоры можно использовать для реализации булевой функции n переменных.



2.2. Декодер

Схема, которая получает на входе n-разрядное число и использует его для того, чтобы выбрать (то есть установить в 1) одну из 2^n выходных линий.

2.3. Арифметико-логическое устройство (АЛУ)

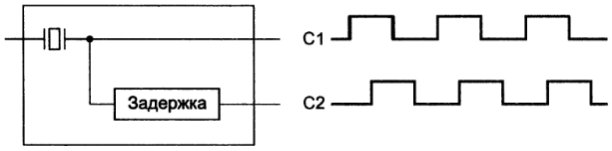


2.4. Сумматор

Полусумматор — схема для вычисления бита суммы и бита переноса. *Сумматор со сквозным переносом*.

2.5. Тактовый генератор

Это схема, которая вызывает серию импульсов одинаковой длительности. Интервалы между последовательными импульсами также одинаковы. Временной интервал между началом одного импульса и началом следующего называется *временем такта*.



Частота импульсов обычно составляет от 1 до 500 МГц (время такта от 1000 до 2 нс). Частота тактового генератора как правило контролируется кварцевым генератором, позволяющим добиться высокой точности. Схемы могут запускаться не только уровнем сигнала, но также фронтом или спадом.

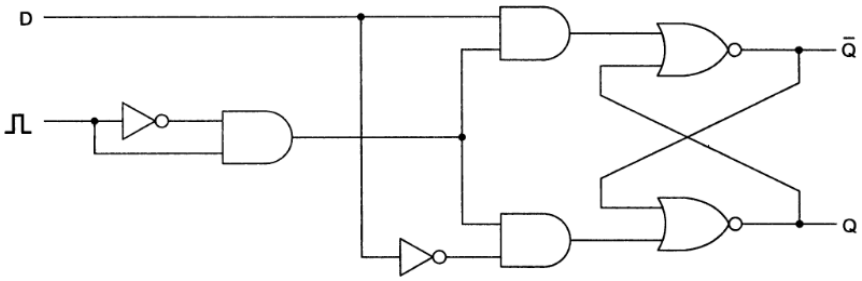
2.6. Элементы памяти

SR-защёлка

синхронная SR-защёлка

триггер

D-триггер



Что читать

Хэррис Д. М., *Хэррис С. Л.* глава 2, стр. 144–200, 220—248, глава 5, стр. 601–638
Таненбаум Э., *Остин Т.* глава 3, стр. 182–208

Упражнения

1. Постройте двухразрядный *компаратор*, то есть схему, которая получает на вход два двухбитовых слова и выдаёт на выход `1`, если слова равны, и `0`, если они не равны.
2. *Схема сдвига* имеет 2^n входных битов. Выходные данные — это входные данные, сдвинутые на один бит. Линия управления определяет направление сдвига: `0` — влево, `1` — вправо. Постройте схему сдвига для четырёхразрядного слова.
3. Нарисуйте логическую схему двухразрядного *кодера*, который содержит 4 входные и 2 выходные линии. Одна из входных линий всегда равна `1`. Двухразрядное двоичное число на двух выходных линиях показывает, какая именно входная линия равна `1`.
4. Нарисуйте логическую схему двухразрядного *демультиплексора*, у которого сигнал на единственной входной линии направляется к одной из четырёх выходных линий в зависимости от значений двух линий управления.

3 Простые исполнители алгоритмов

3.1. Конечные автоматы

Конечные автоматы — это простейшие преобразователи входных данных в выходные данные. Их обычно рассматривают как *потоковые* преобразователи, то есть такие исполнители, которые потенциально могут работать сколь угодно долго, получая на вход поток символов и печатая на выход преобразованный поток символов.

Определение 1. Конечный автомат $\mathcal{M}$ задаётся множеством из пяти элементов: конечное множество состояний K, конечный алфавит B, начальное состояние $s_0 \in K$, подмножество состояний останова $F \subset K$ и функция перехода $\pi : K \times B \to K$:

$$\mathcal{M} = \{B, K, \pi, s_0, F\}, \quad s_0 \in K,\ F \subset K,\ \pi : K \times B \to K.$$

Алгоритм исполнения конечного автомата с действиями:

$x \leftarrow s_0$
while $x \notin F$ **do**
  $b \leftarrow$ считать символ
  $(x, a) \leftarrow \pi(x, b)$
  выполнить действие a
end while

3.2. Машины Тьюринга

Можно считать, что машина Тьюринга — это обобщённый конечный автомат, в который добавлены дополнительные элементы:

- специальное запоминающее устройство — бесконечная лента, состоящая из ячеек памяти, в которых могут хранится символы;
- возможность совершать определённые действия во время переходов между состояниями, а именно, двигаться вдоль ленты, читать содержимое ячеек ленты и записывать в них новые значения.

Алгоритм исполнения машин Тьюринга:

$x \leftarrow s_0$
while $x \notin F$ **do**
  $b \leftarrow$ значение активной ячейки
  $(x, b, a) \leftarrow \pi(x, b)$
  значение активной ячейки $\leftarrow b$
  выполнить действие a
end while

Машины Тьюринга и алгорифмы Маркова приведены далее в нотации консольного эмулятора `turingmarkov`[1], реализованного на языке `Python`.

Пример. Рассмотрим машину Тьюринга из упражнения 1, которая реализует счётчик чётности:

```
     0     1     _
0  _,R,  _,R,1  0,N,!
1  _,R,  _,R,0  1,N,!
```

Машина Тьюринга описывается таблицей, где столбцы соответствуют символу на ленте в активной ячейке, а каждая строка определяет некоторое состояние:

0 – движение вправо с удалением символов, пока количество единиц **чётно**
1 – то же, пока количество единиц **нечётно**

Конец слова достигается, когда в активной ячейке оказывается пробел (обозначаемый символом «_»). Каждая ячейка определяет значение функции перехода — тройку (новый символ, направление движения, новое состояние). Головка может двигаться влево (L), вправо (R) или же остаться на месте (N). Состояние «!» считается состоянием останова.

3.3. Алгорифмы Маркова

Вычисления, основанные на правилах, называются продукционными. Исполнитель алгорифмов Маркова — это простейший исполнитель, осуществляющий продукционные вычисления.

В памяти этого исполнителя хранится одно слово X конечной длины, составленное из букв конечного алфавита B'. Оно изначально равно входному слову, а после останова — выходному слову (результату вычислений).

Алфавит входного слова B является подмножеством алфавита B'. Так же как и в машинах Тьюринга, во время вычислений можно использовать дополнительные буквы из $B' \backslash B$, которые не могут присутствовать во входном слове.

Преобразование слова X на исполнителе алгорифмов Маркова происходит путём последовательности преобразований — замен одного подслова в слове X на другое. Возможные замены описываются **правилами подстановки**.

Упорядоченное множество правил подстановки задаёт логику работы исполнителя. Совокупность правил и алфавитов B и B' называется **нормальным алгорифмом Маркова**, или просто **алгорифмом**.

Пусть b и b' — некоторые слова в алфавите B'. Правило «$b \to b'$» означает, что первое слева вхождение подслова b в слове X следует заменить на слово b'. Это **правило** считается **применимым**, если слово X содержит подслово b. Некоторые правила могут быть помечены как **правила останова**.

Алгоритм работы исполнителя алгорифмов Маркова:

$i \leftarrow 0$
$X \leftarrow X_0$
while true do
  if P_i применимо к X **then**
    $X \leftarrow P_i(X)$
    if P_i — правило останова **then**

[1] https://github.com/vslutov/turingmarkov

```
      завершение работы
    end if
    i ← 0
  else
    i ← i + 1
    if i = n then
      завершение работы
    end if
  end if
end while
```

Пример. Рассмотрим алгорифм из упражнения 4, который приписывает букву x к входному слову в алфавите $B = \{0, 1\}$ справа:

```
x0 -> 0x
x1 -> 1x
x => x
  -> x
```

Поскольку изначально во входном слове буква x не может содержаться, то первым сработает последнее правило. Оно заменит пустое подслово ($b = \varnothing$), которое содержится в начале любого слова, на x. Затем будут срабатывать правила 1 и 2 до тех пор, пока справа от x не останется ни одной цифры. Тогда сработает правило 3, которое является правилом останова. Мы не можем обойтись без него, так как иначе вновь сработает правило 4 и появится ещё одна буква x. В таком случае исполнитель будет работать бесконечно.

Что читать

Винокуров Н. А., *Ворожцов А. В.* лекции 1–3 и семинары 1–3, стр. 24–60

Упражнения

Опишите машины Тьюринга, которые реализуют:

1. Счётчик чётности. Выход машины Тьюринга равен `0` или `1` в зависимости от того, чётно или нечётно число единиц в последовательности из `0` и `1`, записанной на ленте. В начальном состоянии головка видит первый левый символ.
2. Инверсию заданного слова в алфавите `{0,1}`, то есть `0` заменяет на `1`, а `1` — на `0`.
3. «Переворачивание» заданного слова в алфавите `{a,b,c}`.

Запишите нормальные алгорифмы Маркова, которые реализуют:

4. Приписывание буквы `x` к входному слову в алфавите `{0,1}` справа.
5. То же, что и в упражнении 2.
6. То же, что и в упражнении 3.
7. Распознавание правильных скобочных выражений. Правильное скобочное выражение — это слово в алфавите `{(,)}`, которое может получиться, если из арифметического выражения удалить все символы, кроме скобок. Примеры правильных скобочных выражений: пустое слово, `()`, `(())()`, `()()`, `((()))`. Примеры неправильных скобочных

выражений: `)(`, `())(()`, `(, ))))`, `((())`. Результат работы: слово `yes`, если скобочное выражение правильное, и слово `no` — иначе.

8. Увеличение данного натурального числа, записанного в двоичной системе счисления, на единицу. Вход: число `n` — слово в алфавите `{0,1}`. Выход: двоичная запись натурального числа `n+1`.

4 Представление данных в ЭВМ

4.1. Числа конечной точности

Память компьютера ограничена, поэтому возможно иметь дело только с такими числами, которые можно представить в фиксированном количестве разрядов:

$$000, 001, 002, \ \dots, 999$$

Сюда не входят:

- числа большие 999
- отрицательные числа
- дроби
- иррациональные числа
- комплексные числа

Свойство замкнутости множества целых чисел:

$$\forall i, j \in \mathbb{N}: \quad i + j, i - j, i \times j \in \mathbb{N}$$

Числа конечной точности незамкнуты относительно всех четырёх операций:

$600 + 600 = 1200$	слишком большое число
$003 - 005 = -2$	отрицательное число
$050 + 050 = 2500$	слишком большое число
$007/002 = 3.5$	нецелое число

Ошибки делятся на два класса:

- ошибки переполнения и ошибки потери значимости
- результат не является членом ряда

Законы алгебры для чисел конечной точности выполняются не всегда:

$a + (b - c) = (a + b) - c$	$700 + (400 - 300) \neq (700 + 400) - 300$	сочетательный з-н
$a \times (b - c) = a \times b - a \times c$	$5 \times (210 - 195) \neq 5 \times 210 - 5 \times 195$	распределительный з-н

4.2. Отрицательные целые числа

Системы представления отрицательных m-разрядных чисел:

1. со знаком
2. дополнение до единицы
3. дополнение до двух (одно представление нуля: +0, но ряд несимметричен, $10 \dots 0 \to 10 \dots 0$)
4. со смещением на 2^{m-1}

4.3. Числа с плавающей точкой

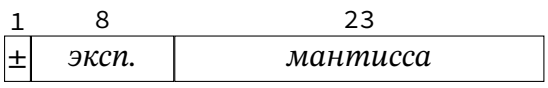
Экспоненциальная форма: $n = f \times 10^e$. Область значений определяется числом разрядов в экспоненте e, а точность — числом разрядов в мантиссе f.

Для хранения чисел в диапазоне $0.1 \leqslant |f| < 1$ с трёхразрядной мантиссой со знаком и двухразрядной экспонентой со знаком требуется всего 5 разрядов и 2 знака. Характерные промежутки на числовой оси:

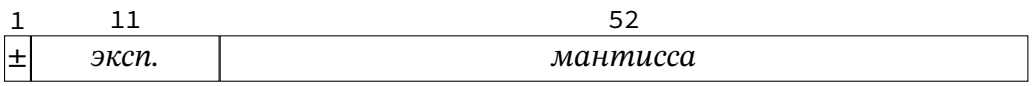
$(-\infty, -0.999 \times 10^{99})$	отрицательное переполнение
$[-0.999 \times 10^{99}, -0.100 \times 10^{-99}]$	выражаемые отрицательные числа
$(-0.100 \times 10^{-99}, 0)$	отрицательная потеря значимости
0	нуль
$(0, +0.100 \times 10^{-99})$	положительная потеря значимости
$[+0.100 \times 10^{-99}, +0.999 \times 10^{99}]$	выражаемые положительные числа
$(+0.999 \times 10^{99}, +\infty)$	положительное переполнение

Числа не формируют континуума. Приходится выполнять *округление*. Плотность представляемых чисел разная, но *относительная погрешность* примерно одинаковая.

4.4. Стандарт IEEE 754



32 бита
одинарная точность



64 бита
двойная точность

Оба формата начинаются со знакового бита для всего числа; `0` указывает на положительное число, `1` — на отрицательное. Затем следует смещённая экспонента. Для формата одинарной точности смещение равно 127, а для формата двойной точности — 1023. Минимальная (0) и максимальная (255 и 2047) экспоненты не используются для нормализованных чисел. В конце идёт мантисса.

нормализованное число	±	$0 < Exp < Max$	любой набор битов
ненормализованное число	±	0	любой ненулевой набор битов
нуль	±	0	0
бесконечность	±	111 ... 1	0
не число	±	111 ... 1	любой ненулевой набор битов

Что читать

Хэррис Д. М., *Хэррис С. Л.* глава 1 стр. 24–50, глава 5 стр. 639–652
Таненбаум Э., *Остин Т.* приложения А и Б стр. 708–728

Упражнения

1. Даны десятичные дроби. Представьте их в формате стандарта `IEEE` для чисел с плавающей точкой одинарной точности и запишите результат восемью шестнадцатеричными разрядами.

   а) `9.0` [Ответ: 41100000_{16}] *б*) `5/32` [Ответ: $3E200000_{16}$] *в*) `-5/32` [Ответ: $BE200000_{16}$] *г*) `6.125` [Ответ: $40C40000_{16}$]

2. Даны числа в формате стандарта `IEEE` для чисел с плавающей точкой одинарной точности записанные восемью шестнадцатеричными разрядами. Запишите их в виде десятичных дробей.

а) $\texttt{42E28000}_{16}$ [Ответ: `113.25`]
б) $\texttt{3F880000}_{16}$ [Ответ: `1.0625`]
в) $\texttt{00800000}_{16}$ [Ответ: `1.17549e-38`]
г) $\texttt{C7F00000}_{16}$ [Ответ: `-122880.0`]

3. Чему равно значение переменной a после выполнения следующего оператора?

```
double a = (1. + 1.e-20) - 1.;
```

Объясните полученный результат.

4. Дано рекуррентное соотношение:

$$x_1 = \frac{1}{e}, x_k = 1 - kx_{k-1}, \quad k = 2, 3, 4, \ldots$$

а)* Докажите, что $\forall k \in \mathbb{N} : \; x_k > 0$.
б) Напишите программу, которая вычисляет первые 15 чисел с одинарной точностью и выводит их на экран.
в) Объясните противоречие между первыми двумя пунктами.

5 Представление программы на машинном уровне

5.1. Процесс компиляции

```
$ gcc -Og -o prog p1.c p2.c
```

GCC на самом деле вызывает целый набор программ, чтобы перевести исходный код C в исполняемый:

- *Препроцессор*. Расширяет исходный код, включая содержимое файлов, указанных посредством директивы `#include`, и разворачивая макросы определяемые директивой `#define`.
- *Компилятор*. Создаёт ассемблерные версии исходных файлов (`p1.s`, `p2.s`). Ключ (опция) `-Og` предписывает компилятору использовать уровень оптимизации, при котором получится машинный код, в целом следующий общей структуре исходного кода на C. Использование более высоких уровней оптимизации (`-O1`, `-O2` и `-O3`) может привести к генерации машинного кода, который настолько сильно изменён, что понять взаимосвязь с исходным кодом затруднительно.
- *Ассемблер*. Переводит ассемблерный код в объектный (`p1.o`, `p2.o`). Объектный код является одной из форм машинного кода — он содержит двоичное представление всех команд, но адреса глобальных переменных ещё не определены.
- *Редактор связей*. Связывает объектные файлы между собой и с библиотеками функций и создаёт файл `prog` (как предписывает ключ `-o prog`) с исполняемым кодом. Исполняемые код — ещё одна форма машинного кода, которая может выполняться непосредственно процессором.

5.2. Программа на машинном уровне

Компьютерные системы используют ряд абстракций, скрывая детали реализации и предоставляя более простую модель. Наиболее значимые абстракции:

- *Архитектура набора команд* (ISA — Instruction Set Architecture). Определяет формат и представление программ на машинном уровне.
- *Виртуальные адреса памяти*. Позволяют представить память на машинном уровне как очень большой массив байтов.

Ассемблерный код очень близок к машинному. Основное отличие в том, что язык ассемблера текстовый и более прост в понимании для человека.

Машинный код позволяет увидеть детали и части системы, которые скрыты от программиста, пишущего на языке C:

- *Счётчик команд*, обозначаемый `%rip` в `x86-64`. Определяет последовательность исполнения инструкций.

- Целочисленный *файл регистров*. Содержит 16 именованных ячеек памяти для хранения 64-битных значений. Эти регистры могут хранить адреса (соответствующие указателям в C) или целочисленные данные.
- Регистр флагов. Хранит информацию о результатах последней выполненной арифметической или логической команды. Флаги используются для реализации условных переходов (`if`, `while`, ...) и пересылки данных по условию.
- Набор векторных регистров. Каждый может хранить одно или несколько целых чисел или чисел с плавающей точкой.

Пример. Рассмотрим следующий код, размещённый в файле `mstore.c`:

```
1 long mult2 (long, long);
2
3 void multstore (long x, long y, long *dest)
4 {
5   long t = mult2 (x, y);
6   *dest = t;
7 }
```

Ассемблерную версию этого кода можно получить, задав компилятору дополнительно ключик `-S`:

```
$ gcc -Og -S mstore.c
$ vim mstore.s          # open file with the vim editor
```

Команды листинга, начинающиеся с точки, являются директивами ассемблера. Директивы вида `.cfi_*` сохраняют дополнительную информацию, полезную при отладке программ. Мы можем полностью их игнорировать. Для отключения генерации этих директив, необходимо добавить ключик `-fno-asynchronous-unwind-tables`:

```
$ gcc -Og -S -fno-asynchronous-unwind-tables mstore.c
```

Набирать длинную команду каждый раз весьма утомительно. Для неё можно объявить синоним (alias):

```
$ alias asmlst="gcc -Og -S -fno-asynchronous-unwind-tables"
```

и далее вызывать, используя новое имя:

```
$ asmlst mstore.c
```

В результате получим ассемблерный листинг из нашего кода на C:

```
5 multstore:
6   pushq   %rbx
7   movq    %rdx, %rbx
8   call    mult2@PLT
9   movq    %rax, (%rbx)
10  popq    %rbx
11  ret
```

Ключ `-c` предписывает `gcc` выполнить и компиляцию, и ассемблирование:

```
$ gcc -Og -c mstore.c
```

Таким образом получается объектный файл (`mstore.o`). С помощью отладчика:

```
$ gdb mstore.o
```

можно увидеть машинный код инструкций функции:

```
(gdb) x/14xb multstore
0x0 <multstore>:    0x53  0x48  0x89  0xd3  0xe8  0x00  0x00  0x00
0x8 <multstore+8>:  0x00  0x48  0x89  0x03  0x5b  0xc3
```

Отсюда понятно, что программа, выполняемая машиной, — это всего лишь последовательность байтов, кодирующих серию машинных инструкций.

Исследовать содержимое файлов с машинным кодом можно при помощи программ называемых *дизассемблерами*:

```
$ objdump -d mstore.o
```

Из машинного кода они создают код в формате похожем на ассемблерный листинг:

```
0000000000000000 <multstore>:
   0:  53                    push   %rbx
   1:  48 89 d3              mov    %rdx,%rbx
   4:  e8 00 00 00 00        call   9 <multstore+0x9>
   9:  48 89 03              mov    %rax,(%rbx)
   c:  5b                    pop    %rbx
   d:  c3                    ret
```

Следует отметить, что:

- Инструкции `x86-64` имеют переменную длину от 1 до 15 байт. Код устроен так, что часто используемые инструкции, а также те, что имеют меньшее количество операндов, занимают меньше байтов.
- Формат спроектирован так, что инструкцию можно однозначно декодировать, начиная с некоторой фиксированной позиции.

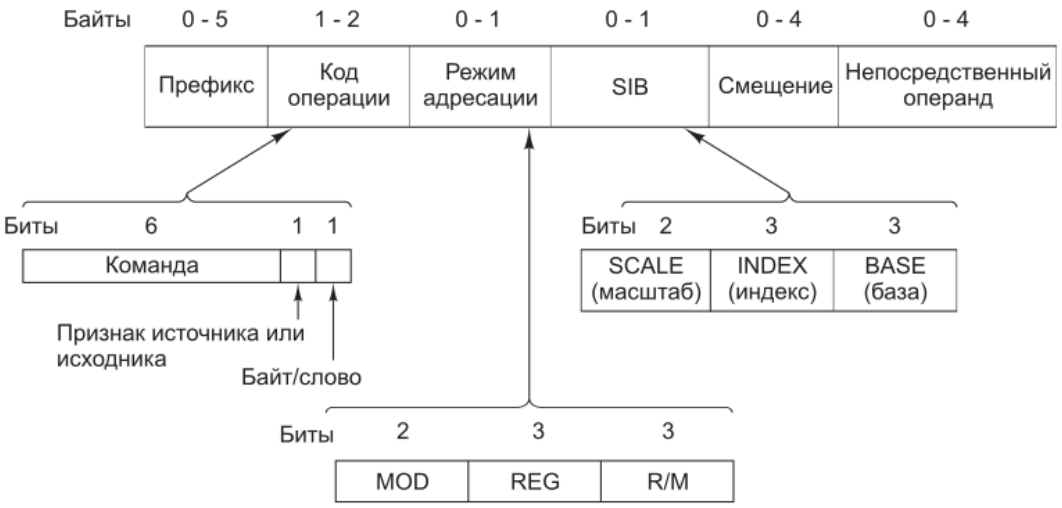


Рис. 5.1 — Формат команд процессора `Core i7`

- Дизассемблер выводит ассемблерный код, исходя только из последовательности байтов в файле с машинным кодом.

– Дизассемблер использует немного иные соглашения об именовании, чем GCC. В частности, он опускает суффукс q у многих инструкций.

Чтобы создать код, который будет выполняться системой, необходимо запустить *компоновщик* для набора объектных файлов, в одном из которых определена функция `main` (`main.cpp`):

```
1 #include <iostream>
2
3 extern "C" void multstore (long, long, long *);
4
5 int main ()
6 {
7   long d;
8   multstore (2, 3, &d);
9   std::cout <<"2 * 3 --> "<< d << std::endl;
10 }
11
12 extern "C" long mult2 (long a, long b)
13 {
14   long s = a * b;
15   return s;
16 }
```

Директива `extern "C"` предписывает компилятору использовать соглашения языка C для именования функций. Язык C++ поддерживает «перегрузку» имени функции и, вследствие этого, придерживается иных правил об именовании: он выполняет декорацию (mangling) исходного имени при помощи имён типов аргументов.

Собрать исполняемый код можно командой:

```
$ g++ -Og -o prog main.cpp mstore.o
```

Объём исполняемого файла значительно больше объёма составляющих его объектных файлов, поскольку он содержит не только код наших процедур, но также информацию, необходимую для запуска и завершения программы и для взаимодействия с операционной системой. Выполнив команду:

```
$ objdump -d prog
```

получим дизассемблерный листинг исполняемого кода:

```
143 00000000000011dd <multstore>:
144     11dd:  53                   push   %rbx
145     11de:  48 89 d3             mov    %rdx,%rbx
146     11e1:  e8 ef ff ff ff       call   11d5 <mult2>
147     11e6:  48 89 03             mov    %rax,(%rbx)
148     11e9:  5b                   pop    %rbx
149     11ea:  c3                   ret
```

Какие изменения произошли в коде функции `multstore` при компоновке?

Что читать

Брайант Р. Э., О'Халларон Д. Р. глава 3, стр. 190–197

Упражнения

1. Настройте рабочую среду, следуя инструкциям из соответствующего подраздела на странице 81, если Вы не сделали этого ранее.

2. Рассмотрим следующий код, который пытается просуммировать элементы массива `arr`, количество элементов задаётся параметром `length`:

```
/* WARNING: This is buggy code */
float sum_elements (float arr[], unsigned length)
{
  float result = 0.;
  for (int i = 0; i <= length - 1; ++i)
    result += arr[i];

  return result;
}
```

   Если вызвать функцию с аргументом `length`, равным `0`, то она должна вернуть нуль. Однако вместо этого происходит ошибка при работе с памятью. Объясните, почему. Покажите, как исправить код.

3. Рассмотрим следующие C-функции:

```
int fun1 (unsigned word)
{
  return (int) ((word << 24) >> 24);
}

int fun2 (unsigned word)
{
  return ((int) word << 24) >> 24;
}
```

   Положим, что они выполняются под управлением 32-разрядной машины, которая использует арифметику на основе дополнения до двух. Также положим, что сдвиг вправо для чисел со знаком выполняется арифметически, а для чисел без знака — логически.

   А. Заполните таблицу ниже, которая показывает действие функций на примере нескольких аргументов. Будет удобнее работать с числами в шестнадцатеричной записи. Помните, что цифры от `8` до `F` имеют старший бит равный `1`.

w	fun1(w)	fun2(w)
0x00000076	_____	_____
0x87654321	_____	_____
0x000000C9	_____	_____
0xEDCBA987	_____	_____

   Б. Опишите словами, какое полезное вычисление выполняет каждая из функций.

4. Вам дали задание написать функцию, которая определяет длиннее ли одна строка, чем другая. Вы решили использовать стандартную функцию `strlen`, которая объявляется следующим образом:

```
/* Prototype for library function strlen */
size_t strlen (const char *s);
```

   Вот первая попытка:

```
/* Determine whether string s is longer than string t */
/* WARNING: This function is buggy */
```

```
int strlonger (char *s, char *t)
{
  return strlen(s) - strlen(t) > 0;
}
```

Во время тестирования на некоторых данных результаты получились неправильные. Вы занялись отладкой и выяснили, что при компиляции для 32-разрядной архитектуры тип данных `size_t` определён (через `typedef`) в заголовочном файле `stdio.h` как `unsigned`.

А. В каких случаях функция выдаёт неверный результат?
Б. Объясните, каким образом этот неверный результат получается.
В. Покажите, как исправить код, чтобы он работал надёжно.

5. Нам будут встречаться листинги, генерируемые дизассемблером — программой, которая преобразует исполняемый файл обратно в более читабельную ASCII форму. Эти файлы содержат много шестнадцатеричных чисел, обычно представленных в виде дополнения до двух. Способность распознавать эти числа и понимать их смысл (например, положительные они или отрицательные) представляет собой важный навык.
Преобразуйте шестнадцатеричные значения (в 32-битном дополнении до двух), представленные в листинге справа от мнемоник команд (`sub`, `mov` и `add`) в десятичные эквиваленты:

```
4004d0:  48 81 ec e0 02 00 00   sub    $0x2e0,%rsp                А. ____
4004d7:  48 8b 44 24 a8         mov    -0x58(%rsp),%rax           Б. ____
4004dc:  48 03 47 28            add    0x28(%rdi),%rax            В. ____
4004e0:  48 89 44 24 d0         mov    %rax,-0x30(%rsp)           Г. ____
4004e5:  48 8b 44 24 78         mov    0x78(%rsp),%rax            Д. ____
4004ea:  48 89 87 88 00 00 00   mov    %rax,0x88(%rdi)            Е. ____
4004f1:  48 8b 84 24 f8 01 00   mov    0x1f8(%rsp),%rax           Ж. ____
4004f8:  00
4004f9:  48 03 44 24 08         add    0x8(%rsp),%rax
4004fe:  48 89 84 24 c0 00 00   mov    %rax,0xc0(%rsp)            З. ____
400505:  00
400506:  48 8b 44 d4 b8         mov    -0x48(%rsp,%rdx,8),%rax    И. ____
```

6. Положим, что `int` занимает 32 бита и использует дополнение до двух для представления чисел со знаком. Сдвиги вправо выполняются арифметически для значений со знаком и логически для значений без знака. Переменные объявлены и инициализированы следующим образом:

```
int x = foo();  /* Arbitrary value */
int y = bar();  /* Arbitrary value */

unsigned ux = x;
unsigned uy = y;
```

Для каждого из следующих C выражений докажите, что оно истинно для всех значений `x` и `y`, или приведите значения `x` и `y`, для которых оно ложно:

А. `(x > 0) || (x - 1 < 0)` _____
Б. `(x & 7) != 7 || (x << 29 < 0)` _____
В. `x * x >= 0` _____
Г. `x < 0 || -x <= 0` _____
Д. `x > 0 || -x >= 0` _____
Е. `x + y == uy + ux` _____
Ж. `x * ~y + uy * ux == -x` _____

6 Доступ к данным

6.1. Структура модуля на языке ассемблера

```
  .globl    var         # declare var to be visible outside the module
  .bss                  # specific [data] section: better-save-space
var:                    # a [label]
  .zero     4           # store 4 bytes, filled by zeroes

  .data                 # [data] section
var2:
  .byte     'a'         # store byte, filled with ASCII code of symbol 'a'
var3:
  .long     4           # store 4 bytes, filled with an integer 4

  .text                 # [code] section
func:
  mnemonic [op1[, op2[, op3]]]  # an [instruction] format
  # ...
                        # an [empty line] at the end
```

6.2. Регистры

63	31	15	7 0	
%rax	%eax	%ax	%al	– *accumulator, return value*
%rbx	%ebx	%bx	%bl	– *base, callee saved*
%rcx	%ecx	%cx	%cl	– *counter, 4th arg.*
%rdx	%edx	%dx	%dl	– *data, 3rd arg.*
%rsi	%esi	%si	%sil	– *source index, 2nd arg.*
%rdi	%edi	%di	%dil	– *destination index, 1st arg.*
%rbp	%ebp	%bp	%bpl	– *base pointer, callee saved*
%rsp	%esp	%sp	%spl	– *stack pointer*
%r8	%r8d	%r8w	%r8b	– *5th arg.*
%r9	%r9d	%r9w	%r9b	– *6th arg.*
%r10	%r10d	%r10w	%r10b	– *caller saved*
%r11	%r11d	%r11w	%r11b	– *caller saved*
%r12	%r12d	%r12w	%r12b	– *callee saved*
%r13	%r13d	%r13w	%r13b	– *callee saved*
%r14	%r14d	%r14w	%r14b	– *callee saved*

%r15	%r15d	%r15w	%r15b	– *callee saved*
%rip	%eip	%ip		– *instruction pointer*
%rflags	%eflags	%flags		– *condition flags*

В 64-разрядном режиме[1], размер операнда определяет количество действительных бит в регистре-приёмнике:

- 64-битные операнды порождают 64-битный результат в регистре-приёмнике.
- 32-битные операнды порождают 32-битный результат, дополняемый нулями до 64-битного значения в регистре-приёмнике.
- 8- и 16-битные операнды порождают 8- и 16-битные результаты. Старшие 56 или 48 бит, соответственно, в регистре-приёмнике остаются без изменений. Если 8- или 16-разрядная операция планируется для вычисления 64-разрядного адреса, то необходимо в явном виде дополнить знаковым битом до полного 64-битного регистра.

6.3. Формы операндов

Тип	Форма	Значение операнда	Адресация
константа	$Imm	Imm	непосредственная
регистр	r_a	R[r_a]	регистровая
память	Imm	M[Imm]	абсолютная
память	(r_a)	M[R[r_a]]	косвенная
память	Imm(r_b)	M[Imm+R[r_b]]	по базе со смещением
память	Imm(r_b,r_i)	M[Imm+R[r_b]+R[r_i]]	индексная
память	Imm(r_b,r_i,s)	M[Imm+R[r_b]+R[r_i]·s]	индексная с масштабированием

Положим, что в перечисленных ниже адресах памяти и регистрах сохранены указанные значения. Заполните таблицу, вычислив значения операндов.

Адрес	Значение
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Регистр	Значение
%rax	0x100
%rcx	0x1
%rdx	0x3

Операнд	Значение
%rax	_____
0x104	_____
$0x108	_____
(%rax)	_____
4(%rax)	_____
9(%rax,%rdx)	_____
260(%rcx,%rdx)	_____
0xFC(,%rcx,4)	_____
(%rax,%rdx,4)	_____

6.4. Копирование данных

Инструкция	Действие	Пояснение
mov S,D	D ← S	скопировать
movb		байт

[1]Intel® 64 and IA-32 Architectures Software Developer's Manual. In 7 vols. Vol. 1. Basic Architecture. 2014. 3.4.1.1 General-Purpose Registers in 64-Bit Mode.

`movw`			слово
`movl`			двойное слово
`movq`			четверное слово
`movabsq`	`I,R`	`R` ← `I`	
`movz`	`S,R`	`R` ← ZeroExtend(`S`)	скопировать и расширить без учёта знака
`movzbw`			
`movzbl`			
`movzwl`			
`movzbq`			
`movzwq`			
`movs`	`S,R`	`R` ← SignExtend(`S`)	скопировать и расширить с учётом знака
`movsbw`			
`movsbl`			
`movswl`			
`movsbq`			
`movswq`			
`movslq`			
`cltq`		`%rax` ← SignExtend(`%eax`)	

Пример. Рассмотрим следующий код (`move.c`):

```
1 long exchange (long *xp, long y)
2 {
3   long x = *xp;
4   *xp = y;
5   return x;
6 }
```

Вызвав команду[2]:

```
$ asmlst move.c
```

получим ассемблерный листинг:

```
xp: %rdi, y: %rsi
5 exchange:
6   movq   (%rdi), %rax
7   movq   %rsi, (%rdi)
8   ret
```

Примечания:

1. Пользователи `Windows` могут отметить, что у них в коде аргументы передаются через регистры `%rcx` и `%rdx`. Это связано с тем, что разные платформы могут использовать разные соглашения. Чтобы получить листинг в соглашениях, принятых в `Unix`, добавьте ключ `-mabi=sysv`:

   ```
   $ asmlst -mabi=sysv move.c
   ```

2. В `Linux` тип `long` означает длинное целое число (8 байт), а под `Windows` это обычное целое число (4 байта). Используйте здесь и далее `long long` или `int64_t` из библиотеки `<inttypes.h>`.

[2] Команда-синоним `asmlst` рассмотрена ранее в разделе 5.2 на странице 13.

6.5. Работа с программным стеком

Инструкция	Действие	Пояснение
`pushq S`	`R[%rsp] ← R[%rsp] - 8;` `M[R[%rsp]] ← S`	поместить в стек четверное слово
`popq D`	`D ← M[R[%rsp]];` `R[%rsp] ← R[%rsp] + 8`	взять из стека четверное слово

Чтобы разместить в стеке четверное слово, необходимо вначале уменьшить указатель стека на 8, а затем записать целевое значение по новому адресу вершины стека. Таким образом, поведение инструкции `pushq %rbp` эквивалентно следующей паре инструкций:

```
subq $8, %rsp      # decrement stack pointer
movq %rbp, (%rsp)  # store %rbp on stack
```

за исключением того, что инструкция `pushq` представляется в машинном коде всего лишь одним байтом, в то время как пара инструкций, представленных выше, требует 8 байт.

Чтобы изъять четверное слово из стека, необходимо вначале прочитать его с вершины стека, а затем увеличить указатель стека на 8. Таким образом, инструкция `popq %rax` эквивалентна следующей паре инструкций:

```
movq (%rsp), %rax  # read %rax from stack
addq $8, %rsp      # increment stack pointer
```

Что читать

Брайант Р. Э., О'Халларон Д. Р. глава 3, стр. 197–209

Упражнения

1. Заполните пропущенные суффиксы команды `mov`, основываясь на размере операндов:

```
mov  %eax, (%rsp)        # _
mov  (%rax), %dx         # _
mov  $0xFF, %bl          # _
mov  (%rsp,%rdx,4), %dl  # _
mov  (%rdx), %rax        # _
mov  %dx, (%rax)         # _
```

2. Каждая из следующих строк вызывает ошибку ассемблера. Поясните, в чём проблема.

```
movb $0xF, (%ebx)
movl %rax, (%rsp)
movw (%rax), 4(%rsp)
movb %al, %sl
movq %rax, $0x123
movl %eax, %rdx
movb %si, 8(%rbp)
```

3. Предположим, что переменные `sp` и `dp` объявлены как:

```
src_t  *sp;
dest_t *dp;
```

где `src_t` и `dest_t` некоторые типы данных, объявленные при помощи `typedef`. Мы хотим использовать инструкции копирования, чтобы реализовать операцию:

```
*dp = (dest_t) *sp;
```

Положим, что значения `sp` и `dp` размещены в регистрах `%rdi` и `%rsi`, соответственно. Для каждой записи в таблице выпишите две инструкции, которые реализуют указанную операцию присваивания. Первая инструкция должна читать данные из памяти, выполнять требуемое преобразование и записывать соответствующую часть регистра `%rax`. Вторая инструкция затем должна записать эту часть регистра обратно в память. В обоих случаях частями могут быть `%rax`, `%eax`, `%ax`, `%al`, и они могут отличаться друг от друга.
Помните, что при выполнении преобразования в C, которое включает и изменение размера, и изменение «знаковости», операция должна сначала изменить размер.

src_t	dest_t	Инструкция
long	long	`movq (%rdi), %rax` `movq %rax, (%rsi)`
char	int	
char	unsigned	
unsigned char	long	
int	char	
unsigned	unsigned char	
char	short	

4. Функция имеет прототип:

```
void decode (long *xp, long *yp, long *zp);
```

и компилируется в ассемблерный код, приведённый ниже:

```
xp: %rdi, yp: %rsi, zp: %rdx
decode:
  movq  (%rdi), %r8
  movq  (%rsi), %rcx
  movq  (%rdx), %rax
  movq  %r8, (%rsi)
  movq  %rcx, (%rdx)
  movq  %rax, (%rdi)
  ret
```

Параметры `xp`, `yp` и `zp` хранятся в регистрах `%rdi`, `%rsi` и `%rdx`, соответственно. Выпишите эквивалентный исходный C код для функции `decode`.

7 Арифметические и логические операции

Большинство арифметических и логических операций образуют семейства инструкций в зависимости от размеров операндов (байт, слово, двойное слово, четверное слово). Исключение составляет инструкция `leaq`, у неё единственный вариант. (Почему?)

Инструкция		Действие	Пояснение
`leaq`	`S,R`	`R ← &S`	загрузить действительный адрес
`inc`	`D`	`D ← D + 1`	увеличить на 1
`dec`	`D`	`D ← D - 1`	уменьшить на 1
`neg`	`D`	`D ← -D`	изменить знак
`not`	`D`	`D ← ~D`	дополнить до единиц (поразрядное НЕ)
`add`	`S,D`	`D ← D + S`	сложить
`sub`	`S,D`	`D ← D - S`	вычесть
`imul`	`S,D`	`D ← D * S`	умножить
`xor`	`S,D`	`D ← D ^ S`	поразрядное ИСКЛЮЧАЮЩЕЕ ИЛИ
`or`	`S,D`	`D ← D | S`	поразрядное ИЛИ
`and`	`S,D`	`D ← D & S`	поразрядное И
`sal`	`k,D`	`D ← D << k`	сдвиг влево
`shl`	`k,D`	`D ← D << k`	сдвиг влево (то же, что и `sal`)
`sar`	`k,D`	`D ← D` $>>_A$ `k`	арифметический сдвиг вправо
`shr`	`k,D`	`D ← D` $>>_L$ `k`	логический сдвиг вправо

7.1. Загрузка действительного адреса

Пусть регистры `%rbx` и `%rdx` хранят значения p и q соответственно. Заполните таблицу формулами, выражающими значение, которое будет записано в регистр `%rax`:

Инструкция	Результат
`leaq 9(%rdx), %rax`	
`leaq (%rdx,%rbx), %rax`	
`leaq (%rdx,%rbx,4), %rax`	
`leaq 2(%rbx,%rbx,8), %rax`	
`leaq 0xE(,%rdx,2), %rax`	
`leaq 6(%rbx,%rdx,8), %rax`	

7.2. Унарные и бинарные операции

Положим, что в перечисленных ниже адресах памяти и регистрах сохранены указанные значения.

Заполните таблицу, указывая регистр или место в памяти, которые будут изменены в результате отдельного выполнения каждой инструкции, а также результирующее значение.

Адрес	Значение
0x100	0xFF
0x108	0xAB
0x110	0x13
0x118	0x11

Регистр	Значение
%rax	0x100
%rcx	0x1
%rdx	0x3

Инструкция	Адрес назначения	Значение
addq %rcx, (%rax)	_____	_____
subq %rdx, 8(%rax)	_____	_____
imulq $16, (%rax,%rdx,8)	_____	_____
incq 16(%rax)	_____	_____
decq %rcx	_____	_____
subq %rdx, %rax	_____	_____

7.3. Сдвиг разрядов

Ниже приведены C функция:

```c
long shift_left4_rightn (long x, long n)
{
  x <<= 4;
  x >>= n;
  return x;
}
```

и часть её ассемблерного кода, которая выполняет сдвиги и сохраняет результирующее значение в регистре %rax:

```
x: %rdi, n: %rsi
  movq %rdi, %rax  # get x
                   # x <<= 4
  movl %esi, %ecx  # get n
                   # x >>= n
```

Здесь опущены две ключевых инструкции. Выпишите их, опираясь на комментарии. Сдвиг вправо следует выполнить арифметически.

7.4. Пример

Рассмотрим код на C, содержащий арифметические вычисления (arith.c):

```c
long arith (long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0F0F0F0F;
  long t4 = t2 - t3;
  return t4;
}
```

Вызвав GCC[1]:

[1] Команда-синоним asmlst рассмотрена ранее в разделе 5.2 на странице 13.

```
$ asmlst arith.c
```

получим из него ассемблерный листинг:

```
x: %rdi, y: %rsi, z: %rdx
arith:
  xorq  %rsi, %rdi
  leaq  (%rdx,%rdx,2), %rax
  salq  $4, %rax
  andl  $252645135, %edi
  subq  %rdi, %rax
  ret
```

7.5. Специальные арифметические операции

Инструкция	Действие	Пояснение
`imulq S`	`R[%rdx]R[%rax]` ← `S` × `R[%rax]`	полное произведение с учётом знака
`mulq S`	`R[%rdx]R[%rax]` ← `S` × `R[%rax]`	полное произведение без учёта знака
`cqto`	`R[%rdx]R[%rax]` ← SignExtend(`R[%rax]`)	расширить до восьмикратного слова
`idivq S`	`R[%rdx]` ← `R[%rdx]R[%rax]` mod `S` `R[%rax]` ← `R[%rdx]R[%rax]` ÷ `S`	деление с учётом знака
`divq S`	`R[%rdx]` ← `R[%rdx]R[%rax]` mod `S` `R[%rax]` ← `R[%rdx]R[%rax]` ÷ `S`	деление без учёта знака

Примеры. Рассмотрим следующий код (`store_uprod.c`):

```c
#include <inttypes.h>

typedef unsigned __int128 uint128_t;

void store_uprod (uint128_t *dest, uint64_t x, uint64_t y)
{
  *dest = x * (uint128_t) y;
}
```

Используя команду:

```
$ asmlst store_uproduct.c
```

получим ассемблерный листинг. Компилятор использует команду умножения без учёта знака с одним аргументом и извлекает результат из соответствующих регистров:

```
dest: %rdi, x: %rsi, y: %rdx
store_uprod:
  movq  %rsi, %rax
  mulq  %rdx
  movq  %rax, (%rdi)
  movq  %rdx, 8(%rdi)
  ret
```

Теперь рассмотрим код, использующий деление (`remdiv.c`):

```
void remdiv (long x, long y, long *qp, long *rp)
{
  long q = x / y;
  long r = x % y;
  *qp = q;
  *rp = r;
}
```

Как и ранее, вызвав команду:

```
$ asmlst remdiv.c
```

получим ассемблерный листинг. Видно, что компилятор использует команду деления, предварительно заполнив соответствующие регистры:

```
x: %rdi, y: %rsi, qp: %rdx, rp: %rcx
remdiv:
  movq  %rdi, %rax
  movq  %rdx, %r8
  cqto
  idivq  %rsi
  movq  %rax, (%r8)
  movq  %rdx, (%rcx)
  ret
```

Что читать

Брайант Р. Э., О'Халларон Д. Р. глава 3, стр. 209–217

Упражнения

1. Дана функция, в которой опущено вычисляемое выражение:

   ```
   long scale (long x, long y, long z)
   {
     long t = _________________;
     return t;
   }
   ```

   и её ассемблерный код:

   ```
   x: %rdi, y: %rsi, z: %rdx
   scale:
     leaq  (%rdi,%rdi,4), %rax
     leaq  (%rax,%rsi,2), %rax
     leaq  (%rax,%rdx,8), %rax
     ret
   ```

   Выпишите пропущенное в C коде выражение.

2. Ниже приведены C функция:

   ```
   long arith2 (long x, long y, long z)
   {
     long t1 = _______;
     long t2 = _______;
   ```

```
  long t3 = ________;
  long t4 = ________;
  return t4;
}
```

и ассемблерный код её тела:

```
x: %rdi, y: %rsi, z: %rdx
arith2:
  orq   %rsi, %rdi
  sarq   $3, %rdi
  notq   %rdi
  movq   %rdx, %rax
  subq   %rdi, %rax
  ret
```

Основываясь на приведённом коде, восстановите инструкции тела C функции.

3. В ассемблерном коде, полученном из C, часто можно обнаружить строки вида:

```
xorq   %rcx, %rcx
```

хотя в исходном коде никаких операций ИСКЛЮЧАЮЩЕЕ ИЛИ нет.

   А. Объясните действие данной инструкции. Какую полезную операцию она реализует?
   Б. Каким образом можно нагляднее выразить эту операцию в ассемблерном коде?
   В. Сравните количество байт, необходимое для кодирования этих двух реализаций одной и той же операции.

4. Рассмотрим код функции на C, который вычисляет 128-битное произведение двух 64-битных значений *x* и *y*, а затем сохраняет результат в память:

```
#include <inttypes.h>

typedef __int128 int128_t;

void store_prod (int128_t *dest, int64_t x, int64_t y)
{
  *dest = x * (int128_t) y;
}
```

Ниже показан ассемблерный код, который GCC выдаёт для этой функции:

```
dest: %rdi, x: %rsi, y: %rdx
store_prod:
  movq   %rsi, %rax
  movq   %rdx, %r9
  sarq   $63, %r9
  movq   %rsi, %r11
  sarq   $63, %r11
  movq   %r11, %rcx
  imulq   %rdx, %rcx
  movq   %r9, %rsi
  imulq   %rax, %rsi
  addq   %rsi, %rcx
  mulq   %rdx
  addq   %rcx, %rdx
  movq   %rax, (%rdi)
  movq   %rdx, 8(%rdi)
  ret
```

Этот код использует три умножения для арифметики с многократно увеличенной точностью, требуемой для реализации 128-разрядной арифметики на 64-разрядной машине. Опишите алгоритм, который используется для вычисления произведения, и прокомментируйте ассемблерный код, чтобы показать, как он реализует этот алгоритм.

Подсказка: при расширении аргументов x и y до 128 бит, их можно представить как:

$$x = 2^{64} \cdot x_h + x_l \quad \text{и} \quad y = 2^{64} \cdot y_h + yl,$$

где x_h, x_l, y_h и y_l являются 64-битными значениями. Подобным образом 128-битное произведение может быть записано как:

$$p = 2^{64} \cdot p_h + p_l,$$

где p_h и p_l являются 64-битными значениями. Покажите, как вычисляются значения p_h и p_l через значения x_h, x_l, y_h и y_l.

5. Рассмотрим следующую функцию, которая вычисляет частное и остаток от деления двух 64-битных целых чисел без знака:

```
void uremdiv (unsigned long x, unsigned long y,
              unsigned long *qp, unsigned long *rp)
{
  unsigned long q = x / y;
  unsigned long r = x % y;
  *qp = q;
  *rp = r;
}
```

Реализуйте эту функцию, взяв за основу ассемблерный код для деления чисел со знаком, приведённый в качестве примера на странице 26.

8 Управление потоком выполнения

8.1. Флаги условий

В дополнение к целочисленным регистрам ЦП имеет *регистр флагов*, описывающих свойства последней выполненной арифметической или логической операции. Например, предположим, что используется инструкция `add` для выполнения следующей операции:

```
int t = a + b;
```

- CF: `Carry Flag = (unsigned)t < (unsigned)a` Последняя операция породила перенос из самого старшего бита. Используется для обнаружения переполнения при выполнении операций без учёта знака.
- ZF: `Zero Flag = (t == 0)` Результатом последней операции стал нуль.
- SF: `Sign Flag = (t < 0)` В результате последней операции получено отрицательное значение.
- OF: `Overflow Flag = (a < 0 == b < 0) && (t < 0 != a < 0)` Последняя операция над числами с учётом знака вызвала переполнение — отрицательное или положительное.

Инструкция `leaq` не изменяет никакие флаги, поскольку подразумевает использование в адресной арифметике. Остальные операции, приведённые в таблице на странице 23, производят установку флагов. Логические операции, такие как `xor`, устанавливают флаги переноса и переполнения в нуль. Операции сдвига устанавливают флаг переноса в значение последнего «вытесненного» бита, а флаг переполнения в нуль. Инструкции `inc` и `dec` устанавливают флаг переполнения и флаг нуля, но флаг переноса оставляют без изменений.

В дополнение к упомянутым арифметическим и логическим инструкциям есть ещё две инструкции, которые выставляют флаги, не меняя при этом никакие другие регистры.

Инструкция	Основана на	Пояснение
`cmp` S_1, S_2	S_2 - S_1	сравнить
`test` S_1, S_2	S_1 & S_2	протестировать

Вместо непосредственного доступа к значениям флагов существуют три распространённых способа использовать их. В зависимости от комбинаций установленных флагов можно: 1) установить байт в `0` или `1`, 2) перейти к выполнению другой части программы, 3) скопировать данные.

Примечания:

1. Флаг переноса `CF` означает как *перенос* из старшего разряда, так и *заём*.
2. Подробнее о расстановке флагов командами изложено в руководстве Intel® 64 and IA-32 Architectures Software Developer's Manual. In 7 vols. Vol. 1. Basic Architecture. 2014. A.1 EFLAGS AND INSTRUCTIONS.

8.2. Установка байта по условию

Инструкция	Синоним	Действие	Условие
`sete D`	`setz`	`D ← ZF`	равно/нуль
`setne D`	`setnz`	`D ← ~ZF`	не равно/не нуль
`sets D`		`D ← SF`	отрицательное
`setns D`		`D ← ~SF`	неотрицательное
`setg D`	`setnle`	`D ← ~(SF ^ OF) & ~ZF`	больше (знаковое $>$)
`setge D`	`setnl`	`D ← ~(SF ^ OF)`	больше или равно (знаковое $\geqslant$)
`setl D`	`setnge`	`D ← SF ^ OF`	меньше (знаковое $<$)
`setle D`	`setng`	`D ← (SF ^ OF) | ZF`	меньше или равно (знаковое $\leqslant$)
`seta D`	`setnbe`	`D ← ~CF & ~ZF`	выше (беззнаковое $>$)
`setae D`	`setnb`	`D ← ~CF`	выше или равно (беззнаковое $\geqslant$)
`setb D`	`setnae`	`D ← CF`	ниже (беззнаковое $<$)
`setbe D`	`setna`	`D ← CF | ZF`	ниже или равно (беззнаковое $\leqslant$)

Пример. Код функции, имеющей следующий прототип:

```
int comp (data_t a, data_t b);
```

вынуждает компилятор генерировать инструкции семейства `set`:

```
a: %rdi, b: %rsi
comp:
  cmpq  %rsi, %rdi
  setg  %al
  movzbl  %al, %eax
  ret
```

Упражнение. Код на C:

```
int comp (data_t a, data_t b)  { return a COMP b; }
```

выполняет сравнение аргументов `a` и `b`. Тип аргументов задаётся синонимом `data_t` (через `typedef`), а операция сравнения — макросом `COMP` (через `#define`).

Положим, что значение параметра `a` располагается в соответствующей части регистра `%rdi` и значение параметра `b` — регистра `%rsi`. Определите какой тип `data_t` и операция `COMP` могли бы заставить компилятор создать код, приведённый ниже. (Правильных ответов может быть несколько, перечислите все.)

а)
```
cmpl  %esi, %edi
setl  %al
```
б)
```
cmpw  %si, %di
setge %al
```
в)
```
cmpb  %sil, %dil
setbe %al
```
г)
```
cmpq  %rsi, %rdi
setne %al
```

Упражнение. Код на C:

```
int test (data_t a)  { return a TEST 0; }
```

выполняет сравнение аргумента `a` с нулём. Тип аргумента задаётся синонимом `data_t` (через `typedef`), а операция сравнения — макросом `TEST` (через `#define`). Значение параметра `a` располагается в соответствующей части регистра `%rdi`. Определите какой тип `data_t` и

операция TEST могли бы заставить компилятор создать код, приведённый ниже. (Правильных ответов может быть несколько, перечислите все.)

а)
```
testq %rdi, %rdi
setge %al
```
б)
```
testw %di, %di
sete  %al
```
в)
```
testb %dil, %dil
seta  %al
```
г)
```
testl %edi, %edi
setle %al
```

8.3. Инструкции перехода

```
1   movq  $0, %rax      set %rax to 0
2   jmp .L1             goto .L1
3   movq (%rax), %rax   null pointer dereference (skipped)
4 .L1:
5   popq %rdx           jump target
```

Инструкция `jmp .L1` заставляет программу пропустить инструкцию `movq` и вместо этого возобновить выполнение, начиная с инструкции `popq`. Во время генерации объектного кода, ассемблер определяет адреса всех инструкций с метками и кодирует целевой адрес перехода (значение метки `.L1`) как часть инструкции перехода.

Инструкция	Синоним	Условие	Пояснение
`jmp метка`		`1`	прямой переход
`jmp *операнд`		`1`	косвенный переход
`je метка`	`jz`	`ZF`	равно/нуль
`jne метка`	`jnz`	`~ZF`	не равно/не нуль
`js метка`		`SF`	отрицательное
`jns метка`		`~SF`	неотрицательное
`jg метка`	`jnle`	`~(SF ^ OF) & ~ZF`	больше (знаковое >)
`jge метка`	`jnl`	`~(SF ^ OF)`	больше или равно (знаковое ⩾)
`jl метка`	`jnge`	`SF ^ OF`	меньше (знаковое <)
`jle метка`	`jng`	`(SF ^ OF) | ZF`	меньше или равно (знаковое ⩽)
`ja метка`	`jnbe`	`~CF & ~ZF`	выше (беззнаковое >)
`jae метка`	`jnb`	`~CF`	выше или равно (беззнаковое ⩾)
`jb метка`	`jnae`	`CF`	ниже (беззнаковое <)
`jbe метка`	`jna`	`CF | ZF`	ниже или равно (беззнаковое ⩽)

Инструкция `jmp` выполняет безусловный переход. Он может быть *прямым*, когда целевой адрес кодируется как часть инструкции (например, `jmp .L1`), или *косвенным*, когда целевой адрес перехода читается из регистра (например, `jmp *%rax`) или из памяти (например, `jmp *(%rax)`). Остальные инструкции в таблице представляют собой условный переход. Суффиксы определяются как и для семейства инструкций `set`.

Кодирование инструкций перехода

Существует несколько способов представления переходов на машинном уровне. Один из наиболее часто используемых — запомнить смещение *относительно* адреса следующей команды (program counter relative). Это смещение может быть представлено, используя 1, 2 или 4 байта. Вторым способом является задание «абсолютного» адреса, используя 4 байта для непосредственного указания целевого адреса.

В качестве примера относительной адресации рассмотрим следующий код:

```
1   movq  %rdi, %rax
2   jmp  .L2
3 .L3:
4   sarq  %rax
5 .L2:
6   testq  %rax, %rax
7   jg  .L3
8   ret
```

Он содержит два перехода: `jmp` в строке 2 прыгает вперёд к старшим адресам, в то время как `jg` в строке 7 прыгает назад к младшим адресам.

Дизассемблерная версия объектного кода:

```
1    0:  48 89 f8              mov    %rdi,%rax
2    3:  eb 03                 jmp    8 <loop+0x8>
3    5:  48 d1 f8              sar    $1,%rax
4    8:  48 85 c0              test   %rax,%rax
5    b:  7f f8                 jg     5 <loop+0x5>
6    d:  c3                    ret
```

В аннотациях справа дизассемблер приводит целевой адрес инструкции перехода в строке 2 как `0x8` и в строке 5 — как `0x5` (дизассемблер выводит все числа в шестнадцатеричной системе счисления). Однако если взглянуть на второй байт машинного кода, то в первом случае видим `0x03`. Добавляя это смещение к `0x5`, адресу следующей инструкции, получим целевой адрес `0x8`, равный адресу инструкции в строке 4.

Аналогично, целевой адрес второй инструкции перехода представляется в машинном коде как `0xf8` (десятичное `-8`), используя одно-байтовое дополнение до двух. Добавляя его к `0xd` (десятичное `13`), адресу инструкции в строке 6, получим `0x5` — адрес инструкции в строке 3.

Эти примеры показывают, что значение счётчика команд при выполнении относительной адресации равно адресу следующей инструкции, а не адресу самой инструкции перехода. Это соглашение восходит к ранним реализациям, когда процессор обновлял счётчик команд перед выполнением инструкции.

Ниже представлена дизассемблерная версия программы после компоновки:

```
1    1131:  48 89 f8              mov    %rdi,%rax
2    1134:  eb 03                 jmp    1139 <loop+0x8>
3    1136:  48 d1 f8              sar    $1,%rax
4    1139:  48 85 c0              test   %rax,%rax
5    113c:  7f f8                 jg     1136 <loop+0x5>
6    113e:  c3                    ret
```

Инструкции размещены по смещённым адресам, но кодировка целевых адресов перехода в строках 2 и 5 осталась той же. Использование относительной адресации для переходов позволяет выполнять компактное кодирование инструкций (всего 2 байта) и перемещать объектный код в памяти без каких-либо изменений.

Что читать

Брайант Р. Э., О'Халларон Д. Р. глава 3, стр. 218–225

Упражнения

1. Ниже приведены фрагменты из дизассемблерного листинга исполняемого файла. Некоторые детали заменены `X`-ми. Ответьте на следующие вопросы:
   А. Каков целевой адрес инструкции `je`? (Об инструкции `callq` ничего знать не нужно.)

```
4003fa:  74 02                 je     XXXXXX
4003fc:  ff d0                 callq  *%rax
```

   Б. Каков целевой адрес инструкции `je`?

```
40042f:  74 f4                 je     XXXXXX
400431:  5d                    pop    %rbp
```

   В. Каковы адреса инструкций `ja` и `pop`?

```
XXXXXX:  77 02                 ja     400547
XXXXXX:  5d                    pop    %rbp
```

   Г. В приведённом ниже фрагменте, целевой адрес перехода представлен в виде 4-х байтового относительного смещения в формате дополнения до двух. Байты следуют в порядке от младшего к старшему, отражая обратный порядок байт архитектуры `x86-64`. Каков целевой адрес инструкции перехода?

```
4005e8:  e9 73 ff ff ff        jmpq   XXXXXX
4005ed:  90                    nop
```

9 Ветвления и циклы

9.1. Реализация условных ветвлений

Общая форма конструкции `if-else` в C задаётся шаблоном:

```
if (test-expr)
  then-statement
else
  else-statement
```

где *test-expr* является целочисленным выражением, результатом которого является либо нуль (интерпретируется как `false`), либо ненулевое значение (интерпретируется как `true`). Выполняется только одна из ветвей: *then-statement* или *else-statement*.

Эту форму ассемблер обычно реализует в виде, который можно изобразить, используя синтаксис C, следующим образом:

```
  t = test-expr;
  if (!t)
    goto false;
  then-statement
  goto done;
false:
  else-statement
done:
```

Упражнение. Дана C функция:

```
void cond (long a, long *p)
{
  if (p && a > *p)
    *p = a;
}
```

для которой GCC генерирует следующий ассемблерный код:

```
  a: %rdi, p: %rsi
cond:
  testq  %rsi, %rsi
  je  .L1
  cmpq  %rdi, (%rsi)
  jge  .L1
  movq  %rdi, (%rsi)
.L1:
  ret
```

А. Напишите `goto`-версию на C, которая выполняет те же самые вычисления и отражает поток выполнения ассемблерного кода.
Б. Объясните, почему ассемблерный код содержит два условных ветвления, хотя в исходном коде C только одна ветвь.

9.2. Копирование по условию

Альтернативной стратегией для реализации условных ветвлений является пересылка данных по условию. В этом подходе вычисляются выражения в обеих ветвях условного оператора, а затем выбирается одно в зависимости от выполнения условия. Такая стратегия имеет смысл только в ограниченных случаях, но она может быть реализована простой инструкцией *копирования по условию*, которая имеет более высокую производительность на современных процессорах.

Инструкция		Синоним	Условие	Пояснение
`cmove`	`S,R`	`cmovz`	`ZF`	равно/нуль
`cmovne`	`S,R`	`cmovnz`	`~ZF`	не равно/не нуль
`cmovs`	`S,R`		`SF`	отрицательное
`cmovns`	`S,R`		`~SF`	неотрицательное
`cmovg`	`S,R`	`cmovnle`	`~(SF ^ OF) & ~ZF`	больше (знаковое >)
`cmovge`	`S,R`	`cmovnl`	`~(SF ^ OF)`	больше или равно (знаковое $\geqslant$)
`cmovl`	`S,R`	`cmovnge`	`SF ^ OF`	меньше (знаковое <)
`cmovle`	`S,R`	`cmovng`	`(SF ^ OF) | ZF`	меньше или равно (знаковое $\leqslant$)
`cmova`	`S,R`	`cmovnbe`	`~CF & ~ZF`	выше (беззнаковое >)
`cmovae`	`S,R`	`cmovnb`	`~CF`	выше или равно (беззнаковое $\geqslant$)
`cmovb`	`S,R`	`cmovnae`	`CF`	ниже (беззнаковое <)
`cmovbe`	`S,R`	`cmovna`	`CF | ZF`	ниже или равно (беззнаковое $\leqslant$)

Чтобы понять, как условные операции могут быть реализованы через пересылку данных, рассмотрим следующую общую форму условного выражения с присваиванием:

```
v = test-expr ? then-expr : else-expr;
```

Стандартный способ скомпилировать это выражение, используя условные переходы, имеет следующий вид:

```
  if (!test-expr)
    goto false;
  v = then-expr;
  goto done;
false:
  v = else-expr;
done:
```

Этот код содержит две последовательности команд — одну для вычисления *`then-expr`* и вторую для вычисления *`else-expr`*. Комбинация условных и безусловных переходов гарантирует, что будет выполнена только одна из этих последовательностей.

Что касается кода, основанного на пересылке по условию, то необходимо вычислять оба выражения: *`then-expr`* и *`else-expr`*. Финальный результат выбирается, в зависимости от значения *`test-expr`*:

```
v  = then-expr;
ve = else-expr;
t  = test-expr;
if (!t) v = ve;
```

Последняя строка этого кода реализуется операцией пересылки по условию: зачение `ve` копируется в `v`, только если условие `t` не выполняется.

Не все условные выражения могут быть скомпилированы, используя пересылку данных по условию. Если одно из выражений условного оператора может вызывать ошибку или имеет побочные эффекты, такой код может привести к ошибочному поведению:

```
long cread (long *xp)  { return xp ? *xp : 0; }
```

Использование пересылки по условию не всегда повышает производительность кода. Компиляторы должны принимать во внимание соотношение затрат на ненужные вычисления и стоимости ошибки в предсказании перехода (потери на перезапуск конвейера).

Упражнение. Ниже приведена функция на C, в которой определение операции `OP` не завершено:

```
#define OP _____  /* Unknown operator */
long arith (long x)  { return x OP 8; }
```

При компиляции `gcc` генерирует следующий ассемблерный код:

```
x: %rdi
arith:
  leaq  7(%rdi), %rax
  testq  %rdi, %rdi
  cmovns  %rdi, %rax
  sarq  $3, %rax
  ret
```

А. Что использовано в качестве операции `OP`?
Б. Объясните, как работает этот код.

Упражнения

1. Компилируя код на C вида:

```
long test (long x, long y, long z)
{
  long val = __________;

  if (__________)
  {
    if (__________)
      val = __________;
    else
      val = __________;
  }
  else if (__________)
    val = __________;
```

```
14
15  return val;
16 }
```

GCC генерирует следующий ассемблерный листинг:

```
x: %rdi, y: %rsi, z: %rdx
5 test:
6   leaq   (%rdx,%rsi), %rax
7   subq   %rdi, %rax
8   cmpq   $5, %rdx
9   jle   .L2
10  cmpq   $2, %rsi
11  jle   .L3
12  movq   %rdx, %rax
13  imulq   %rdi, %rax
14  ret
15 .L3:
16  movq   %rsi, %rax
17  imulq   %rdi, %rax
18  ret
19 .L2:
20  cmpq   $2, %rdx
21  jg   .L1
22  movq   %rdx, %rax
23  imulq   %rsi, %rax
24 .L1:
25  ret
```

Заполните пропущенные в C коде выражения.

2. Компилируя код на C вида:

```
1 long test (long x, long y)
2 {
3   long val = ________;
4
5   if (________)
6   {
7     if (________)
8       val = ________;
9     else
10      val = ________;
11  }
12  else if (________)
13    val = ________;
14
15  return val;
16 }
```

GCC, запущенный с ключом `-O1`, генерирует следующий ассемблерный листинг:

```
x: %rdi, y: %rsi
5 test:
6   testq   %rdi, %rdi
7   js   .L6
8   leaq   12(%rsi), %rax
9   addq   %rsi, %rdi
10  cmpq   $10, %rsi
11  cmovg   %rdi, %rax
```

```
12   ret
13 .L6:
14   movq  %rsi, %rax
15   subq  %rdi, %rax
16   movq  %rsi, %rdx
17   andq  %rdi, %rdx
18   cmpq  %rdi, %rsi
19   cmovle  %rdx, %rax
20   ret
```

Заполните пропущенные в C коде выражения.

9.3. Циклы

Язык C предоставляет несколько различных циклов: `do-while`, `while` и `for`. В машинном коде нет соответствующих инструкций. Вместо этого для реализации таких конструкций используются комбинации условных и безусловных переходов. `gcc` и другие компиляторы генерируют код для реализации циклов, придерживаясь двух основных шаблонов, которые будут рассмотрены ниже.

Цикл `do-while`

Общая форма инструкции `do-while` в C задаётся шаблоном:

```
do
  body-statement
  while (test-expr);
```

Цикл выполняет тело *`body-statement`*, затем вычисляет условие *`test-expr`* и продолжает работу повторно, пока результатом вычисления условия является не нуль. Эта форма может быть преобразована в `goto`-версию следующим образом:

```
loop:
  body-statement
  t = test-expr;
  if (t)
    goto loop;
```

Упражнение. Дана C функция:

```
1 long loop_do_while (long x)
2 {
3   long y = x * x;
4   long *p = &x;
5   long n = 4 * x;
6
7   do
8   {
9     x += y;
10    *p += 5;
11    n -= 2;
12  }
13  while (n > 0);
```

```
14
15   return x;
16 }
```

для которой GCC генерирует следующий ассемблерный код:

```
  x initially in %rdi
5 loop_do_while:
6   movq   %rdi, %rax
7   movq   %rdi, %rcx
8   imulq   %rdi, %rcx
9   leaq   0(,%rdi,4), %rdx
10 .L2:
11  leaq   5(%rcx,%rax), %rax
12  subq   $2, %rdx
13  testq   %rdx, %rdx
14  jg   .L2
15  ret
```

А. Какие регистры используются для хранения значений переменных x, y и n?
Б. Каким образом компилятор устранил необходимость в указателе p и его разыменовании, которое подразумевает наличие в коде выражения `(*p)+=5`?
В. Поясните, как ассемблерная версия отображает код на C.

Цикл `while`

Общая форма инструкции `while` в C задаётся шаблоном:

```
while (test-expr)
  body-statement
```

Он отличается от цикла `do-while` тем, что *`test-expr`* вычисляется вначале, и цикл может закончится, не выполнив *`body-statement`* ни разу. Существует много способов перевода цикла `while` в машинный код, два из которых используются в коде, генерируемом GCC. Оба способа используют ту же структуру, что и для `do-while`, но отличаются реализацией начальной проверки условия.

Первый способ «переход в середину».

```
  goto test;
loop:
  body-statement
test:
  t = test-expr;
  if (t)
    goto loop;
```

Упражнение. Ниже приведена общая структура кода на C:

```
long loop_while (long a, long b)
{
  long result = _____________________;
  while (_____________________)
  {
    result = _____________________;
```

```
    a = ____________________;
  }
  return result;
}
```

для которой GCC выдаёт следующий код:

```
a: %rdi, b: %rsi
loop_while:
  movl  $0, %eax
  jmp  .L2
.L3:
  leaq  (%rdi,%rsi), %rdx
  addq  %rdx, %rax
  subq  $1, %rdi
.L2:
  cmpq  %rsi, %rdi
  jg  .L3
  ret
```

Видно, что компилятор использовал переход-в-середину (jump-to-middle), применив инструкцию `jmp` в строке 7 для перехода к проверке условия, которое начинается с метки `.L2`. Заполните пропущенные части в C коде.

Второй способ «защищённый-do». Код преобразуется в форму `do-while` и использует условный переход, чтобы пропустить выполнение тела, если условие сразу не выполняется:

```
  t = test-expr;
  if (!t)
    goto done;
  do
    body-statement
    while (test-expr);
done:
```

Упражнение. Ниже приведена общая структура кода на C:

```
long loop_while2 (long a, long b)
{
  long result = ____________;
  while (____________)
  {
    result = ____________;
    b = ____________;
  }
  return result;
}
```

для которой GCC, запущенный с ключом `-O1`, выдаёт следующий код:

```
a: %rdi, b: %rsi
loop_while2:
  testq  %rsi, %rsi
  jle  .L4
  movq  %rsi, %rax
```

```
.L3:
  imulq  %rdi, %rax
  subq  %rdi, %rsi
  testq  %rsi, %rsi
  jg  .L3
  ret
.L4:
  movq  %rsi, %rax
  ret
```

Видно, что компилятор использовал защищённый-do (guarded-do), применив инструкцию `jle` в строке 7 для пропуска выполнения тела, если условие оказалось невыполненным. Заполните пропущенные части в C коде.

Что читать

Брайант Р. Э., О'Халларон Д. Р. глава 3, стр. 225–242

Упражнения

1. Функция `fun_a` имеет следующую структуру:

```
long fun_a (unsigned long x)
{
  long val = 0;

  while (...)
  {
    ...

  }

  return ...;
}
```

GCC генерирует для этой функции ассемблерный код, показанный ниже:

```
x: %rdi
fun_a:
  movl  $0, %eax
  jmp  .L2
.L3:
  xorq  %rdi, %rax
  shrq  %rdi
.L2:
  testq  %rdi, %rdi
  jne  .L3
  andl  $1, %eax
  ret
```

Разберитесь, как работает этот код, и выполните следующее:

А. Определите, какой способ трансляции циклов был использован.
Б. При помощи ассемблерного кода восстановите пропущенные части в C.
В. Опишите словами, что вычисляет функция.

10 Ветвления и циклы (продолжение)

10.1. Цикл `for`

Общая форма инструкции `for` в C задаётся шаблоном:

```
for (init-expr; test-expr; update-expr)
  body-statement
```

Стандарт языка C утверждает (за одним исключением, освещаемом в упражнении 1), что поведение такого цикла идентично следующему коду, использующему цикл `while`:

```
init-expr;
while (test-expr)
{
  body-statement
  update-expr;
}
```

Код, генерируемый `gcc` для цикла `for`, следует одной из двух стратегий трансляции, рассмотренных для цикла `while`, в зависимости от уровня оптимизации:

переход-в-середину

```
  init-expr;
  goto test;
loop:
  body-statement
  update-expr;
test:
  t = test-expr;
  if (t)
    goto loop;
```

защищённый-do

```
  init-expr;
  t = test-expr;
  if (!t)
    goto done;
loop:
  body-statement
  update-expr;
  t = test-expr;
  if (t)
    goto loop;
done:
```

Упражнение. Функция `fun_b` имеет следующую структуру:

```
long fun_b (unsigned long x)
{
  long val = 0;
  for (...; ...; ...)
  {
    ...

  }
  return val;
}
```

GCC генерирует для этой функции ассемблерный код, показанный ниже:

```
  x: %rdi
5 fun_b:
6   movl   $64, %edx
7   movl   $0, %ecx
8 .L2:
9   leaq   (%rcx,%rcx), %rax
10  movq   %rdi, %rcx
11  andl   $1, %ecx
12  orq   %rcx, %rax
13  movq   %rax, %rcx
14  shrq   %rdi
15  subq   $1, %rdx
16  jne   .L2
17  ret
```

Разберитесь, как работает этот код, и выполните следующее:

А. При помощи ассемблерного кода восстановите пропущенные части в C.

Б. Опишите словами, что вычисляет функция.

10.2. Реализация оператора `switch`

Оператор `switch` обеспечивает множественное ветвление на основе целочисленного значения и особенно полезен, когда имеется большое количество вариантов. Он не только делает код на языке C более читабельным, но также допускает эффективную реализацию на основе структуры данных, называемой *таблицей переходов*. Таблица переходов — это массив, в котором i-ый элемент хранит адрес сегмента кода, реализующего некоторое действие программы, соответствующего i-му варианту в операторе `switch`. Для определения целевого адреса перехода код выполняет обращение к таблице переходов, используя индекс i. Преимущество использования таблицы переходов перед длинной последовательностью иструкций `if-else` в том, что время на выполнение переключения не зависит от количества вариантов. GCC выбирает метод трансляции оператора `switch`, руководствуясь количеством вариантов (cases) и разреженностью значений `case`-меток. Таблицы переходов используются, когда имеется несколько вариантов (например, четыре и более), и они укладываются в небольшой диапазон значений.

Упражнение. В функции на C, которая показана ниже, пропущено тело оператора `switch`. `case`-метки не образуют непрерывный диапазон, а некоторые варианты имеют несколько меток.

```
void switch2 (long x, long *dest)
{
  long val = 0;
  switch (x)
  {
    ...  Body of switch statement omitted
  }
  *dest = val;
}
```

При компиляции этой функции GCC генерирует следующий ассемблерный код для начальной части процедуры:

```
x: %rdi
switch2:
  addq   $1, %rdi
  cmpq   $8, %rdi
  ja     .L2
  jmp    *.L4(,%rdi,8)
```

а также код для таблицы переходов:

```
.L4:
  .quad   .L9
  .quad   .L5
  .quad   .L6
  .quad   .L7
  .quad   .L2
  .quad   .L7
  .quad   .L8
  .quad   .L2
  .quad   .L5
```

Опираясь на эти данные, ответьте:

А. Какие значения меток были в операторе `switch`?

Б. Какие варианты имеют несколько меток в C коде?

Что читать

Брайант Р. Э., О'Халларон Д. Р. глава 3, стр. 242–250

Упражнения

1. Оператор `continue` в C заставляет программу перейти в конец текущей итерации цикла. Изложенное выше правило перевода цикла `for` в цикл `while` требует некоторого уточнения, если в теле цикла встречается этот оператор. Для примера рассмотрим следующий код:

```c
/* Example of for-loop containing a continue statement */
/* Sum even numbers between 0 and 9 */
long sum = 0;

for (long i = 0; i < 10; ++i)
{
  if (i & 1)
    continue;

  sum += i;
}
```

   А. Что получится, если наивно применить упомянутое на странице 42 правило перевода цикла `for` в цикл `while`? Что будет неверным?

   Б. Как следует заменить оператор `continue` на оператор `goto`, чтобы гарантировать, что цикл `while` корректно воспроизводит поведение цикла `for`?

2. Для функции `switcher`, написанной на C и имеющей следующую структуру:

```
1 void switcher (long a, long b, long c, long *dest)
2 {
3   long val;
4
5   switch (a)
6   {
7   case _____:       /* Case A */
8     c = _____;
9     /* Fall through */
10
11  case _____:       /* Case B */
12    val = _____;
13    break;
14
15  case _____:       /* Case C */
16  case _____:       /* Case D */
17    val = _____;
18    break;
19
20  case _____:       /* Case E */
21    val = _____;
22    break;
23
24  default:
25    val = _____;
26  }
27
28  *dest = val;
29 }
```

GCC, запущенный с ключом `-fno-pie`, генерирует ассемблерный код и таблицу переходов, как показано ниже:

```
a: %rdi, b: %rsi, c: %rdx, d: %rcx
5 switcher:
6   cmpq  $7, %rdi
7   ja  .L2
8   jmp  *.L4(,%rdi,8)
9   .section  .rodata
10  .align 8
11  .align 4
12 .L4:
13  .quad  .L7
14  .quad  .L2
15  .quad  .L3
16  .quad  .L2
17  .quad  .L6
18  .quad  .L5
19  .quad  .L2
20  .quad  .L3
21  .text
22 .L6:
23  movq  %rdi, %rsi
24  jmp  .L2
25 .L5:
```

```
26   movq   %rsi, %rdx
27   xorq   $15, %rdx
28 .L7:
29   leaq   112(%rdx), %rsi
30 .L2:
31   movq   %rsi, (%rcx)
32   ret
33 .L3:
34   addq   %rdx, %rsi
35   salq   $2, %rsi
36   jmp   .L2
```

Заполните пропущенные части в C коде. Существует только один способ разместить варианты в шаблоне, за исключением порядка `case`-меток `C` и `D`.

11 Процедуры

Процедуры являются ключевой абстракцией в разработке программного обеспечения. Они предоставляют способ оформления кода, который реализует некоторую функциональность с заданным набором аргументов и, в случае необходимости, возвращаемым значением. Такая функция может быть вызвана из другой точки программы. Хорошо спроектированное программное обеспечение использует процедуры как механизм абстракции, скрывая детали реализации и, в то же время, предоставляя краткий и ясный интерфейс, определяющий, какие значения будут вычислены и какое действие процедура окажет на состояние программы. Процедуры принимают разные формы в разных языках программирования — функции, методы, подпрограммы, обработчики и т. п. — но все они обладают общим набором особенностей.

Для организации процедур на машинном уровне, необходимо учесть разные моменты. Предположим, что процедура `P` вызывает процедуру `Q`, затем `Q` выполняется и возвращается обратно в `P`. Эти действия включают в себя один или несколько следующих механизмов:

Передача управления. Счётчик команд должен быть установлен в адрес первой команды процедуры `Q` при входе и затем переведён в адрес первой инструкции `P`, следующей за вызовом `Q`, при возврате.

Передача данных. `P` должна быть способна передать один или более параметров в `Q`, а `Q` — вернуть значение обратно в `P`.

Выделение и освобождение памяти. `Q` может потребоваться выделить память для локальных переменных до начала исполнения и, затем, освободить её перед возвратом.

11.1. Программный стек

Во время выполнения процедуры `Q` процедура `P`, наряду со всеми остальными процедурами в цепочке вызовов, временно приостанавливается. Пока выполняется `Q`, только ей необходимо выделять память под локальные переменные или аргументы для вызова другой процедуры. Следовательно, программа может управлять памятью, необходимой для процедур, используя стек.

В архитектуре `x86-64` стек растёт в направлении младших адресов. Указатель стека `%rsp` указывает на вершину стека. Данные могут быть размещены в стеке и изъяты оттуда, используя команды `pushq` и `popq`. Место под данные без начального значения может быть выделено в стеке просто путём уменьшения значения указателя стека на необходимую величину. Освобождение памяти выполняется таким же образом, путём увеличения указателя стека.

Когда процедуре необходимо больше памяти, чем могут хранить регистры, она выделяет место в стеке. Эта область называется *кадром стека* процедуры. Кадр текущей процедуры всегда расположен на вершине стека. Когда процедура `P` вызывает `Q` она сохраняет в стек *адрес возврата*, который указывает, откуда следует возобновить выполнение процедуры `P` после возврата из `Q`. Адрес возврата будем считать частью кадра стека `P`, поскольку он хранит состояние, имеющее значение для `P`. Кадр стека для большинства процедур имеет фиксированный размер, и память выделяется в самом начале процедуры. Тем не менее, для некото-

рых процедур необходим стек переменного размера. Процедура P может передать до шести целочисленных значений (т. е. указателей и целых чисел) в регистрах, но если Q требует больше аргументов, то они должны быть сохранены в стеке процедурой P перед вызовом Q.

11.2. Передача управления

Передача управления из функции P в функцию Q осуществляется установкой указателя команд в адрес первой команды Q. Однако, когда настаёт момент возврата из Q, процессору необходима информация, откуда следует возобновить исполнение процедуры P.

Инструкция	Пояснение
call метка	вызов процедуры (прямой)
call *операнд	вызов процедуры (косвенный)
ret	возврат из процедуры

В машинах с архитектурой x86-64 эта информация записывается при вызове процедуры Q командой call Q. Данная инструкция помещает адрес A в стек и устанавливает счётчик команд на начало процедуры Q. Адрес A называют *адресом возврата* — это адрес инструкции, следующей непосредственно за командой call. В дополнение к команде call есть инструкция ret, которая извлекает A из стека и устанавливает счётчик команд в это значение.

11.3. Передача данных

Первые шесть целочисленных аргументов процедуры размещаются в регистрах согласно их порядку в списке аргументов. Аргументы, меньшие 64-х бит, помещаются в соответствующие части указанных 64-битных регистров.

Размер операнда, бит	Номер аргумента					
	1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Упражнение. Функция на C procprob имеет четыре аргумента u, a, v и b. Каждый из них представляет собой либо число со знаком, либо указатель на число со знаком. Аргументы имеют разные размеры. Ниже приведено тело функции:

```
3   *u += a;
4   *v += b;
5   return sizeof(a) + sizeof(b);
```

Функция компилируется в следующий код для архитектуры x86-64:

```
5 procprob:
6   movslq  %edi, %rdi
7   addq  %rdi, (%rdx)
8   addb  %sil, (%rcx)
9   movl  $6, %eax
10  ret
```

Определите порядок и типы параметров функции. Возможны два верных ответа.

11.4. Размещение локальных переменных в стеке

Причинами размещения локальных переменных в стеке часто оказываются:

- Недостаточное количество регистров, чтобы поместить все локальные данные.
- К локальной переменной применяется оператор взятия адреса &, так что компилятор вынужден генерировать адрес для неё.
- Некоторые локальные переменные являются массивами или структурами, и, следовательно, компилятор должен обращаться к их элементам, используя адресную арифметику и косвенный доступ.

11.5. Размещение локальных переменных в регистрах

Набор программных регистров действует как разделяемый между всеми процедурами ресурс. Хотя, в каждый момент времени активна лишь одна из них, необходима гарантия, что когда одна процедура (caller) вызывает другую (callee), последняя не перезапишет значения некоторых регистров, которые первая планировала использовать далее. По этой причине в архитектуре `x86-64` принят ряд соглашений об использовании регистров, которые должны выполняться всеми процедурами, включая библиотечные функции.

По соглашению, регистры `%rbx`, `%rbp` и `%r12` – `%r15` классифицируются как callee-saved, то есть их значения обязана сохранить вызываемая процедура. То есть, когда `P` вызывает `Q`, `Q` обязана сохранить значения этих регистров такими, какие они были до её вызова. Процедура `Q` может выполнить это либо не изменяя их вообще, либо предварительно сохранив их в стеке, а затем восстановив оттуда обратно. Используя это соглашение код функции `P` может сохранить значение в любом из callee-saved регистров, вызвать `Q`, и затем использовать значение этого регистра без риска его повреждения за время вызова.

Все остальные регистры, за исключением указателя стека `%rsp`, классифицируются как caller-saved. Это означает, что они могут изменяться любой функцией. Если процедура `P` хранит некоторые данные в таком регистре и вызывает процедуру `Q`, то, поскольку `Q` может свободно менять этот регистр, то `P` обязана сама позаботиться о сохранении этого значения перед вызовом.

11.6. Соглашения о вызовах в `Intel x86-64`

Опускаясь на всё более и более низкие уровни абстракции, код становится всё менее переносимым. Так, в 64-битных операционных системах `Linux` и `Windows` приняты разные соглашения о вызовах процедур и использовании регистров в архитектуре `Intel x86-64`.

`Linux x64`. Соглашения в `Linux` подробно изложены выше (в данном разделе) и кратко приведены в таблице раздела 6.2 ранее. Для удобства повторим эту таблицу ещё раз:

Регистр	Соглашение	Пояснение
%rax	return value	аккумулятор
%rbx	callee saved	база
%rcx	4th argument	счётчик
%rdx	3rd argument	данные
%rsi	2nd argument	индекс источника
%rdi	1st argument	индекс приёмника
%rbp	callee saved	указатель начала кадра стека процедуры
%rsp	stack pointer	

%r8	5th argument	регистр общего назначения
%r9	6th argument	регистр общего назначения
%r10	caller saved	регистр общего назначения
%r11	caller saved	регистр общего назначения
%r12	callee saved	регистр общего назначения
%r13	callee saved	регистр общего назначения
%r14	callee saved	регистр общего назначения
%r15	callee saved	регистр общего назначения

Windows x64. Аналогичная таблица для Windows:

Регистр	Соглашение	Пояснение
%rax	return value	аккумулятор
%rbx	callee saved	
%rcx	1st argument	
%rdx	2nd argument	
%rsi	callee saved	регистр общего назначения
%rdi	callee saved	регистр общего назначения
%rbp	callee saved	указатель начала кадра стека процедуры
%rsp	stack pointer	
%r8	3rd argument	
%r9	4th argument	
%r10	caller saved	используется в syscall/sysret инструкциях
%r11	caller saved	используется в syscall/sysret инструкциях
%r12	callee saved	регистр общего назначения
%r13	callee saved	регистр общего назначения
%r14	callee saved	регистр общего назначения
%r15	callee saved	регистр общего назначения

Первые 4 целочисленных аргумента, размер которых не превосходит 8 байт, передаются через регистры %rcx, %rdx, %r8, %r9. Остальные аргументы сохраняются в стек перед вызовом в обратном порядке, как и в Linux.

На момент входа в функцию указатель стека обязан быть выровнен по границе 16 байт. Это означает, что перед выполнением команды вызова call указатель стека должен быть кратен 8 байтам, но не 16 одновременно. (Она записывает на вершину стека адрес возврата, а это те самые 8 байт, которые сделают указатель стека кратным 16.)

11.7. Рекурсивные процедуры

Упражнение. Ниже приведена общая структура функции на C:

```
long rfun (unsigned long x)
{
  if (_______) return _______;

  unsigned long nx = _______;
  long rv = rfun (nx);

  return _______;
}
```

для которой GCC выдаёт следующий код:

```
rfun:
  testq  %rdi, %rdi
  jne  .L8
  movl  $0, %eax
  ret
.L8:
  pushq  %rbx
  movq  %rdi, %rbx
  shrq  $2, %rdi
  call  rfun
  addq  %rbx, %rax
  popq  %rbx
  ret
```

А. Какое значение функция `rfun` сохраняет в callee-saved регистре `%rbx`?
Б. Заполните пропущенные в C коде выражения.

Что читать

Брайант Р. Э., О'Халларон Д. Р. глава 3, стр. 250–264

Упражнения

Напишите функцию, которая принимает текстовую строку (в стиле C) и возвращает `1`, если она оканчивается заглавной латинской буквой, которая более не встречается в тексте, и `0` в противном случае.

Реализация на языке Си. Начать следует с комментария в шапке файла `check.c`, где полезно привести формулировку проверяемого свойства:

```
// property:
//   text ends with a capital Latin letter, which is unique
```

Алгоритм проверки прост и состоит из трёх отдельных действий. Его можно записать следующим образом:

```
__attribute__((sysv_abi)) int check_property (const char* s)
{
  char c = last_of(s);

  if (!isupper(c))
    return 0;

  return count(s, c) == 1;
}
```

Отметим, что далее предполагается использование компилятора `gcc` из пакета `MinGW-64` для ОС `Windows`. Чтобы оставаться в соглашениях `SYSV ABI`, принятых в ОС `Linux` и которых мы до сих пор придерживались, перед заголовком функции добавлен соответствующий атрибут. При компиляции в `Linux` эти атрибуты излишни, их можно опустить.

Вспомогательная функция `last_of()` возвращает последний символ строки `s`:

```
__attribute__((sysv_abi)) static char last_of (const char* s)
{
  assert(s);

  char c = '\0';
  for ( ; *s != '\0'; ++s)
    c = *s;

  return c;
}
```

Мы добавили квалификатор `static`, указав компилятору, что функция должна быть видна только в пределах данного модуля (файла) и недоступна извне. Это одно из средств сокрытия деталей реализации, которые не входят в интерфейс и могут измениться в новой версии.

Следует помнить о проверке указателей! Разыменование нулевого указателя недопустимо! Обработка ошибок усложняет логику, и хотелось бы избежать этого в ассемблерной версии. Простым решением может служить вызов функции `abort()` из `<stdlib.h>`. Однако это не позволяет простым образом выяснить, где произошла ошибка, и требует вывода дополнительного сообщения на экран.

В языке C++ можно использовать механизм исключений. Хотя, повторить подобные действия компилятора в ассемблерной версии потребует определённых усилий, в том числе по изучению особенностей конкретной платформы. Это остаётся за рамками нашего курса.

Мы решили вставить проверку предусловия через макрос `assert()` из стандартной библиотеки `<assert>`, чтобы напомнить о важном аспекте при работе с указателями. Релиз-версия, как правило, игнорирует такие проверки, объявляя макрос `NDEBUG`.

Функция `isupper()` реализована в стандартной библиотеке `<ctype.h>` и возвращает `1`, если символ в верхнем регистре. А функция `count()` подсчитывает количество вхождений символа `c` в строку `s` следующим образом:

```
__attribute__((sysv_abi)) static int count (const char* s, char c)
{
  assert(s);

  int n = 0;
  for ( ; *s != '\0'; ++s)
    if (*s == c) ++n;

  return n;
}
```

Тестирование. Как всегда, чтобы убедиться в работоспособности нашей реализации, необходимо написать небольшую «тестовую оснастку» (`main.cpp`). Для удобства и выразительности кода определим вспомогательный тип `PropertyUnitTest`:

```
using PropertyUnitTest = std::pair<std::string, bool>;
```

Напомним, что тип `pair` из `<utility>` способен хранить пару элементов разного типа. Теперь функция `main()` может выглядеть так:

```
int main ()
{
  std::vector<PropertyUnitTest> tests{
      {"", false},
```

```
39        {"a", false},
40        {"A", true},
41        {"z", false},
42        {"Z", true},
43        {"\\\"", false},
44        {"arjk402-1l5,cvFD.,", false},
45        {"arjk402-1l5,cvFD", true},
46        {"aA", true},
47        {"zZ", true},
48        {"AQ", true},
49        {"A1", false},
50        {"AAAAAAAAAAAAAAAAAA", false},
51        {"ACDEFGHIJKLMNOPAAB", true},
52        {"#{}()<>,`__../\\|\"'", false},
53        {"A#{}()<>,`../\\|\"'A", false},
54        {"a#{}()<>,`../\\|\"'A", true},
55        {"'A'", false},
56        {"'A'B", true},
57        {"qDCBABCDA", false},
58    };
59    size_t mw = max_width(tests);
60
61    for (const auto& t : tests)
62      run(t, mw);
63  }
```

Вспомогательная функция `max_width()` возвращает размер наиболее длинной строки среди данных тестов:

```
24  size_t max_width (const std::vector<PropertyUnitTest>& tests)
25  {
26    auto it = std::max_element(tests.begin(), tests.end(),
27                               [] (const auto& p, const auto& q)
28                               { return p.first.size() < q.first.size(); });
29    if (it != tests.end())
30      return it->first.size();
31
32    throw std::runtime_error{"empty test sequence"};
33  }
```

Она использует `max_element()` из стандартной библиотеки шаблонов `<algorithm>`. В данном случае нам необходимо указать, каким образом сравнивать элементы.

И, наконец, функция `run()` запускает тест, передавая данные нашей функции

```
9   extern "C" __attribute__((sysv_abi)) int check_property (const char* s);
```

и выводит краткий отчёт об успешности его выполнения:

```
13  void run (const PropertyUnitTest& test, size_t mw = 20)
14  {
15    const auto& [s, ans] = test;  // since C++17
16    std::cout << std::setw(mw) << std::left << s << "  " << std::setw(5)
17              << std::boolalpha << ans << "  " << std::flush;
18
```

```
19   bool res = check_property(&s[0]);
20
21   std::cout << "[" << (res == ans ? "OK" : "FAILED") << "]" << std::endl;
22 }
```

Отметим, что со времени выхода стандарта C++11, контейнер `std::string`[1] обязан хранить нулевой байт в качестве завершающего элемента с индексом `s.size()`.

Сборка. Собрать и запустить исполняемый файл можно следующими командами:

```
$ gcc -c check.c && g++ -o prog -std=c++17 main.cpp check.o
$ ./prog
```

При успешном выполнении всех тестов мы получим примерно такой вывод на экране:

```
                    false  [OK]
a                   false  [OK]
A                   true   [OK]
z                   false  [OK]
Z                   true   [OK]
\"                  false  [OK]
arjk402-1l5,cvFD.,  false  [OK]
arjk402-1l5,cvFD    true   [OK]
aA                  true   [OK]
zZ                  true   [OK]
AQ                  true   [OK]
A1                  false  [OK]
AAAAAAAAAAAAAAAAA   false  [OK]
ACDEFGHIJKLMNOPAAB  true   [OK]
#{}()<>,`__../\|"'  false  [OK]
A#{}()<>,`../\|"'A  false  [OK]
a#{}()<>,`../\|"'A  true   [OK]
'A'                 false  [OK]
'A'B                true   [OK]
qDCBABCDA           false  [OK]
```

Реализация на языке ассемблера. Код, показанный выше, помогает нам легко и быстро реализовать ассемблерную версию. Мы уже проработали и отладили алгоритм, а также подготовили тесты. Язык ассемблера даёт максимальную свободу при написании кода, но всю ответственность с этого момента несёт сам программист, ибо ассемблер не отслеживает некорректную работу с типами данных и памятью.

В шапку файла `check_win-sysv.s` также добавим комментарий с формулировкой проверяемого свойства:

```
1 ## property:
2 ##   text ends with a capital Latin letter, which is unique
```

Интерфейсная функция должна быть доступна извне (директива `.globl`):

```
33 .globl check_property
34
35 check_property:
```

[1] https://en.cppreference.com/w/cpp/string/basic_string

```
36   pushq %rdi          # store s on stack
37   call last_of
38
39   pushq %rax          # store c on stack
40   # NB! Under MinGW standard functions use MS ABI
41   movsbl %al, %ecx    # char -> int
42   call isupper
43   popq %rsi           # restore c
44   popq %rdi           # and s from stack
45
46   testl %eax, %eax    # !isupper(c)
47   jnz .L6
48   ret                 # return 0
49
50 .L6:
51   call count
52   cmpl $1, %eax
53   sete %al
54   movzbl %al, %eax
55   ret                 # return n == 1
```

Следует помнить о порядке использования регистров. В функции `check_property()` мы используем соглашения `SYSV ABI`, в то время как функция `isupper()` из стандартной библиотеки уже скомпилирована под `Windows`, и, соответственно, необходимо вызывать её, передавая первый параметр в регистре `%ecx`, как это принято в ОС `Windows`.

Как видно, мы имеем право смешивать соглашения, ассемблер даёт максимальную гибкость. Но, проверять нас никто не будет, поэтому необходимо обо всех деталях помнить самому программисту, что значительно усложняет разработку кода. В противном случае, ошибки неизбежны и заканчиваются неожиданным крахом при выполнении.

Оставшиеся функции пишем по шаблону версий на `Си` в соглашениях `Linux`. Первая функция `last_of()`:

```
4  last_of:
5    xorb %al, %al       # c = 0
6  .L1:
7    movb (%rdi), %r8b
8    testb %r8b, %r8b    # *s != '\0'
9    je .L2
10   movb %r8b, %al      # c = *s
11   incq %rdi           # ++s
12   jmp .L1             # for - iterate to last letter
13 .L2:
14   ret
```

и вторая функция `count()`:

```
17 count:
18   xorl %eax, %eax     # n = 0
19 .L3:
20   movb (%rdi), %r8b
21   testb %r8b, %r8b    # *s != '\0'
22   je .L5
23   cmpb %r8b, %sil     # *s == c
24   jne .L4
```

```
25   incl %eax           # ++n
26 .L4:
27   incq %rdi           # ++s
28   jmp .L3             # for - count if same
29 .L5:
30   ret
```

Обе функции локальные, об этом говорит квалификатор `static` при объявлении в Си. Поэтому не нужно давать директиву `.globl` для них. И если не указано иначе, по умолчанию мы работаем в сегменте кода (директива `.text`).

Тестирование. Собрать исполняемый файл можно одной командой:

```
$ g++ -o prog -std=c++17 main.cpp check_win-sysv.s
```

Тестовая оснастка уже готова. Нам остаётся лишь добиться работоспособности кода, чтобы все тесты были успешно выполнены.

12 Массивы и структуры данных

12.1. Массивы и адресная арифметика

```
T A[N];
```

Обозначим через x_A адрес начала массива. Во-первых, объявление выделяет непрерывную область памяти размером $L \cdot N$ байт, где L — размер (в байтах) данных типа T. Во-вторых, оно вводит в область видимости идентификатор A, который может использоваться как указатель на начало массива. Доступ к элементам осуществляется по индексу в диапазоне от 0 до $N - 1$. Элемент с индексом i размещается по адресу $x_A + L \cdot i$.

Упражнение. Рассмотрим следующие объявления:

```
1 int P[5];
2 short Q[2];
3 int **R[9];
4 double *S[10];
5 short *T[2];
```

Заполните таблицу, описывающую размер элементов, суммарный размер и адрес i-го элемента для каждого массива.

Массив	Размер элемента	Суммарный размер	Адрес начала	Адрес i-го элемента
P			x_P	
Q			x_Q	
R			x_R	
S			x_S	
T			x_T	

Упражнение. Положим, что x_P — адрес массива P типа `short` и индекс i загружены в регистры `%rdx` и `%rcx`, соответственно. Для каждого из следующих выражений выпишите тип, формулу, определяющую значение, и ассемблерную реализацию. Результат должен быть записан в регистр `%rax` (или его часть `%ax`).

Выражение	Тип	Значение	Ассемблерный код
P[1]			
P + 3 + i			
P[i*4 - 5]			
P[2]			
&P[i+2]			

Упражнение. Приведённая ниже функция транспонирует элементы массива $M \times M$, где M определяется при помощи `#define`:

```
3 void transpose (long A[M][M])
4 {
5   for (int i = 0; i < M; ++i)
6   for (int j = 0; j < i; ++j)
7   {
8     long tmp = A[i][j];
9     A[i][j] = A[j][i];
10    A[j][i] = tmp;
11  }
12 }
```

При оптимизации с ключом `-O1` `gcc` генерирует для внутреннего цикла следующий код:

```
20 .L3:
21   movq   (%rdx), %rcx
22   movq   (%rax), %rsi
23   movq   %rsi, (%rdx)
24   movq   %rcx, (%rax)
25   addq   $8, %rdx
26   addq   $120, %rax
27   cmpq   %r8, %rax
28   jne    .L3
```

Очевидно, что обращение к элементам по индексу `gcc` преобразовал в код с указателями.

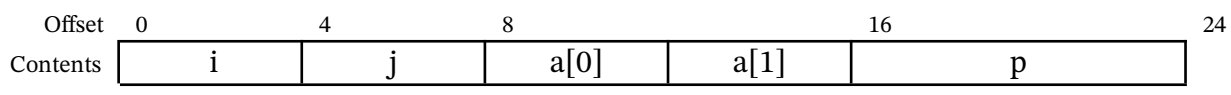
А. Какой регистр содержит указатель на элемент массива `A[i][j]`?
Б. Какой регистр содержит указатель на элемент массива `A[j][i]`?
В. Чему равно значение M?

12.2. Структуры

Объявление `struct` в C создаёт тип данных, который группирует объекты возможно различных типов в один объект. Доступ к различным компонентам структуры осуществляется по имени. Реализация структур подобна реализации массивов в том, что все компоненты структуры размещаются в непрерывном участке памяти и указатель на структуру хранит адрес её первого байта. Компилятор сохраняет информацию о каждом типе-структуре, запоминая смещения в байтах каждого поля. Он генерирует ссылки на элементы структуры, используя эти смещения в качестве смещений для инструкций обращения в память.

В качестве примера рассмотрим следующее объявление:

```
1 struct Record
2 {
3   int i;
4   int j;
5   int a[2];
6   int *p;
7 };
```



и реализацию инструкции:

```
12  r->p = &r->a[r->i + r->j];
```

полагая, что в начальный момент значение переменной `r` типа `Record*` записано в регистр `%rdi`:

```
6   movl  4(%rdi), %eax
7   addl  (%rdi), %eax
8   cltq
9   leaq  8(%rdi,%rax,4), %rax
10  movq  %rax, 16(%rdi)
```

Упражнение. Рассмотрим объявление структуры:

```
1 struct Test
2 {
3   int *p;
4   struct
5   {
6     int x;
7     int y;
8   } s;
9   struct Test *next;
10 };
```

Это объявление показывает, что одна структура может быть встроена в другую так же, как массивы могут быть встроены внутрь структур и массивов [большей размерности].

Следующая ниже процедура (с некоторыми опущенными выражениями) выполняет некоторые операции над приведённой выше структурой:

```
void st_init (struct Test *st)
{
  st->s.y  = _________;
  st->p    = _________;
  st->next = _________;
}
```

А. Чему равны смещения (в байтах) следующих полей?

p: _____
s.x: _____
s.y: _____
next: _____

Б. Сколько байтов всего занимает экземпляр структуры?

В. Компилятор генерирует следующий ассемблерный код:

```
5 st_init:
6   movl  8(%rdi), %eax
7   movl  %eax, 12(%rdi)
8   leaq  12(%rdi), %rax
9   movq  %rax, (%rdi)
10  movq  %rdi, 16(%rdi)
11  ret
```

На основе этой информации заполните пропущенные выражения в коде функции `st_init()`.

12.3. Выравнивание

Многие компьютерные системы накладывают ограничения на допустимые значения адресов для примитивных (базовых) типов данных, требуя, чтобы адрес объекта был кратен некоторому значению K (как правило, 2, 4 или 8). Такое *ограничение на выравнивание* упрощает проектирование аппаратного обеспечения, образующего интерфейс между процессором и системой памяти.

Аппаратное обеспечение `x86-64` будет работать корректно вне зависимости от выравнивания данных. Тем не менее, `Intel` рекомендует выравнивать данные, чтобы улучшить производительность системы памяти. Правило выравнивания основывается на принципе, что любой примитивный объект, занимающий K байт, должен иметь адрес, кратный K. Из этого правила вытекает следующая таблица выравниваний:

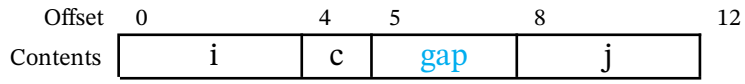
K	Типы
1	`char`
2	`short`
4	`int, float`
8	`long, double, char*`

Выравнивание осуществляется путём проверки, что каждый тип данных устроен таким образом, что все объекты внутри него удовлетворяют своим ограничениям на выравнивание. Компилятор размещает директивы в ассемблерном коде, указывая требуемое выравнивание для глобальных данных. Например, для таблицы переходов оператора `switch` на странице 45:

```
.align 8
```

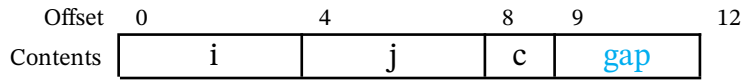
При работе со структурами компилятор может выделять больше памяти и оставлять между полями промежутки, чтобы гарантировать, что каждый элемент удовлетворяет своим ограничениям на выравнивание. Рассмотрим пример:

```
struct S1
{
  int i;
  char c;
  int j;
};
```



Экземпляр структуры также имеет ограничение на выравнивание начального адреса. Например, компилятор может добавить промежуток в конце, чтобы удовлетворить ограничение на выравнивание элементов в массиве:

```
struct S2
{
  int i;
  int j;
  char c;
} d[4];
```



Упражнение. Для каждого из объявлений структуры, определите смещение всех полей, суммарный размер и требование к выравниванию в архитектуре x86-64:

А. `struct P1 { short i; int c; int *j; short *d; };`
Б. `struct P2 { int i[2]; char c[8]; short s[4]; long *j; };`
В. `struct P3 { long w[2]; int *c[2]; };`
Г. `struct P4 { char w[16]; char *c[2]; };`
Д. `struct P5 { struct P4 a[2]; struct P1 t; };`

12.4. ⋆Выход за границы памяти и переполнение буфера

Напомним, что ни язык C, ни C++ не выполняют никаких проверок выхода за границы массива, а локальные переменные размещаются в стеке наряду с *записью активации* — информацией о состоянии программы, такой как значение регистров и адрес возврата. Эта комбинация может привести к серьёзным ошибкам в программе, если состояние, сохранённое в стеке, будет повреждено записью в «элемент» вне массива. Иными словами, когда программа попытается восстановить значение регистра или выполнить инструкцию `ret` при повреждении состояния, дальнейшее выполнение может привести к серьёзным ошибкам.

Очень распространённый источник повреждения состояния программы известен как *переполнение буфера* (buffer overflow). При этом, как правило, создаётся локальный символьный массив для хранения строки, но размер последней превышает выделенную память. Рассмотрим следующий пример:

```
#include <stdio.h>

/* Implementation of library function gets() */
char * gets (char *s)
{
  int c;
  char *dest = s;

  while ((c = getchar()) != '\n' && c != EOF)
    *dest++ = c;

  if (c == EOF && dest == s)  /* no characters read */
    return NULL;

  *dest++ = '\0';  /* terminate string */
  return s;
}

/* Read input line and write it back */
void echo ()
{
  char buf[8];  /* way too small */
  gets (buf);
  puts (buf);
}
```

Этот код показывает реализацию библиотечной функции `gets`, чтобы выявить серьёзную проблему, связанную с ней. Она читает символы из стандартного потока ввода в `s` и останавливается, только когда встречает переход на новую строку или происходит ошибка

ввода. Затем завершает считанную строку нулевым байтом. Функция `echo` показывает пример использования `gets`.

Проблема заключается в том, что не существует способа определить, достаточно ли памяти выделено, чтобы вместить считываемую строку.

Исследуя сгенерированный ассемблерный код, можно выяснить структуру кадра стека функции `echo`:

```
42 echo:
43   pushq  %rbx
44   subq   $16, %rsp
45   leaq   8(%rsp), %rbx
46   movq   %rbx, %rdi
47   call   gets
48   movq   %rbx, %rdi
49   call   puts@PLT
50   addq   $16, %rsp
51   popq   %rbx
52   ret
```

По мере увеличения строки ввода, следующая информация окажется поврежденной:

Количество введённых символов	Дополнительное повреждённое состояние
0 — 7	ничего
8 — 15	состояние вызывающей функции, регистр `%rbx`
16 — 23	адрес возврата
24+	состояние вызывающей функции

В общем случае, использование `gets` или любой другой функции, которая может переполнить память, считается плохой практикой программирования. К несчастью, некоторые часто используемые библиотечные функции, включая `strcpy`, `strcat` и `sprintf`, генерируют последовательность символов в отсутствие информации о размере целевого буфера. Такие условия порождают уязвимость к переполнению буфера.

Более опасное использование переполнения буфера — дать программе выполнить нежелательную функцию. Это один из распространённых приёмов взломать безопасность системы по сети. Обычно программе скармливается строка, которая содержит выполнимый код, называемый *вредоносным кодом* (exploit code), и несколько дополнительных байтов, перезаписывающих адрес возврата указателем на этот код. Выполнение инструкции `ret` вызывает переход к выполнению вредоносного кода.

12.5. ⋆Способы защиты от атак с переполнением буфера

Атаки с переполнением буфера стали настолько распространёнными и вызывали так много проблем с компьютерными системами, что современные компиляторы и операционные системы внедрили механизмы, чтобы усложнить проведение таких атак и ограничить способы, которыми злоумышленник может захватить управление системой через атаки с переполнением буфера. Ниже представлены механизмы предоставляемые современными версиями `gcc` под `Linux`.

Рандомизация стека. Чтобы внедрить вредоносный код в систему, взломщику необходимо вставить не только сам код, но и указатель на него как часть заражённой строки. Создание

такого указателя требует знания адреса в стеке, где будет расположена строка. Исторически, адреса в программном стеке были в высокой степени предсказуемыми.

Идея *рандомизации стека* заключается в его размещении по различным адресам от одного запуска программы к другому. Таким образом, даже если множество машин запускают идентичный код, все они будут использовать различные адреса в стеке. Такое поведение достигается за счёт выделения случайного количества памяти в стеке в промежутке от 0 до n при старте программы, например, при помощи функции `alloca`, которая выделяет указанное ей количество памяти в стеке. Выделенная память программой не используется, но вызывает смещение всех последующих адресов в стеке от запуска к запуску. Диапазон выделяемой памяти должен быть достаточно велик, чтобы получить существенные изменения адресов, но, в то же время, и достаточно мал, чтобы не расходовать память программы впустую.

Следующий код демонстрирует простой способ определения «типичного» адреса в стеке (`stack_address.cpp`):

```
#include <iostream>

int main ()
{
  long local{};
  std::cout <<"stack address is "<< &local << std::endl;
}
```

Собрать и запустить (10 раз) программу можно следующим образом:

```
$ g++ -o prog -Og stack_address.cpp
$ for (( i = 0; $i < 10; ++i )) do ./prog; done
stack address is 0x7ffcaf551740
stack address is 0x7fff203c31f0
stack address is 0x7ffdf1b0aea0
stack address is 0x7ffd2a5c0270
stack address is 0x7fff0067d450
stack address is 0x7ffcdc471290
stack address is 0x7fff63c3ff20
stack address is 0x7ffd0b84a910
stack address is 0x7ffd29d81c90
stack address is 0x7fffaae0b310
```

Рандомизация стека стала стандартной практикой в системах `Linux`. Она принадлежит более широкому классу приёмов, известному как *рандомизация разметки адресного пространства* (ASLR ”— Address-Space Layout Randomization). С ASLR различные части программы, включая код, библиотеки, стек, глобальные переменные и кучу, при каждом запуске загружаются в различные области памяти. Это может предотвратить некоторые формы атак, хотя и не может гарантировать полной безопасности.

Аналогично примеру со стеком, можно определить «типичный» адрес в куче:

```
  auto heap = std::make_unique<long>();
  std::cout <<"heap address is "<< heap.get() << std::endl;
```

и в сегменте кода:

```
  auto code = reinterpret_cast<void*>(&main);
  std::cout <<"code address is "<< code << std::endl;
```

Обнаружение повреждения стека. На второй линии защиты стоит механизм обнаружения, что стек был испорчен. Повреждение стека обычно происходит, когда программа выходит за границу локального буфера. В языке C, а также в C++, нет надёжного способа предотвращения записи вне границ массива. Вместо этого, программа может попытаться обнаружить, когда такая запись произошла, прежде, чем это нанесёт какой-либо вред.

Современные версии GCC встраивают механизм, известный как *защита стека* (stack protector), в генерируемый код, чтобы обнаружить переполнение буфера. Идея заключается в размещении специального *канареечного*[1] *значения* (canary value, guard value) в кадре стека между локальным буфером и его остальной частью. Это значение генерируется случайным образом каждый раз при запуске программы, так что не существует простого способа для взломщика определить его.

GCC пытается выяснить уязвима ли функция к переполнению буфера и вставляет код, реализующий описанную выше стратегию. Отключить такое поведение можно, задав ключ `-fno-stack-protector`.

Ограничение областей с исполняемым кодом. Заключительный шаг — устранить способность взломщика встроить выполняемый код в систему. Одним из приёмов является ограничение областей памяти, которые могут выполняться.

В типичных программах только часть памяти содержит код, сгенерированный компилятором, который необходимо исполнять. Остальная память может быть ограничена только чтением и записью.

Пространство виртуальной памяти логически разделено на *страницы*, обычно по 2048 или 4096 байт. Аппаратное обеспечение поддерживает различные формы *защиты памяти*, предоставляя права доступа к этим страницам как для пользовательских программ, так и для ядра операционной системы. Многие современные системы поддерживают три формы доступа: чтение (чтение данных из памяти), запись (запись данных в память) и выполнение (трактовка содержимого памяти как машинного кода). Таким образом, стек может иметь права на чтение и запись, но не иметь прав на выполнение. Проверка, может ли страница выполняться или нет, осуществляется аппартным обеспечением без потери эффективности.

Что читать

Брайант Р. Э., О'Халларон Д. Р. глава 3, стр. 264–271, 273—276, 279—284, *284–295

Упражнения

1. Ниже объявлена структура ACE и приведён прототип функции `test`:

```
struct ACE
{
  long v;
  struct ACE *p;
};

long test (struct ACE *ptr);
```

При компиляции функции, GCC выдаёт следующий ассемблерный листинг:

[1] Термин «канареечный» сложился по историческим причинам, поскольку этих птиц использовали для обнаружения присутствия опасных газов в угольных шахтах.

```
test:
  movl  $1, %eax
  jmp  .L2
.L3:
  imulq  (%rdi), %rax
  movq  8(%rdi), %rdi
.L2:
  testq  %rdi, %rdi
  jne  .L3
  ret
```

А. Восстановите код на C для функции `test`.
Б. Опишите, какую структуру данных реализует `ACE` и какая операция выполняется функцией `test`.

2. Ответьте на следующие вопросы для данного объявления структуры:

```
struct
{
  int    *a;
  float  b;
  char   c;
  short  d;
  long   e
  double f;
  int    g;
  char   *h;
} rec;
```

А. Каковы смещения (в байтах) всех полей в структуре?
Б. Каков суммарный размер структуры?
В. Перегруппируйте поля так, чтобы минимизировать пустое пространство, и затем ещё раз ответьте на вопросы для полученной структуры.

3. Рассмотрите следующий код, где *R*, *S* и *T* константы, объявленные при помощи `#define`;

```
long A[R][S][T];

long store_ele (long i, long j, long k, long *dest)
{
  *dest = A[i][j][k];
  return sizeof(A);
}
```

При компиляции этой функции с ключом `-fno-pie` GCC генерирует код:

```
store_ele:
  movq  %rdx, %r8
  leaq  (%rsi,%rsi,2), %rax
  leaq  (%rsi,%rax,4), %rax
  movq  %rdi, %rdx
  salq  $6, %rdx
  addq  %rdi, %rdx
  leaq  (%rax,%rdx), %rdi
  addq  %r8, %rdi
  movq  A(,%rdi,8), %rax
  movq  %rax, (%rcx)
```

```
16  movl  $3640, %eax
17  ret
```

А. Выпишите формулу для вычисления одномерного индекса, определяющую положение элемента `A[i][j][k]` в массиве.

Б. Основываясь на приведённом выше ассемблерном коде, определите значения R, S и T.

13 Ассемблирование и компоновка

В качестве примера рассмотрим следующую программу, состоящую из двух модулей:

main.c

```c
void swap();

int buf[2] = {1, 2};

int main ()
{
  swap();
  return 0;
}
```

swap.c

```c
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap ()
{
  bufp1 = &buf[1];

  int tmp = *bufp0;
  *bufp0 = *bufp1;
  *bufp1 = tmp;
}
```

13.1. Стадии сборки

Ключ `-v` заставляет `gcc` выводить подробную информацию о совершаемых действиях:

```
$ gcc -Og -g -o p -v main.c swap.c
...
cc1 main.c [other arguments] -g -Og -o /tmp/ccU1HP45.s
as --64 -o /tmp/cc7TbUce.o /tmp/ccU1HP45.s
...
collect2 -m elf_x86_64 -o p [system object files and args] /tmp/cc7TbUce.o
/tmp/cc6OWLom.o -lc [other libraries]
```

Замечание. В современном `gcc` препроцессор встроен в сам компилятор `cc1`, поэтому нет необходимости вызывать его в явном виде. Результат работы препроцессора (`main.i` и `swap.i`) можно получить отдельно, используя опцию `-save-temps`.

Утилита `collect2` на самом деле является «обёрткой» компоновщика `ld` и вызывает его, выполнив перед этим (в некоторых случаях) дополнительные действия, например, генерацию кода для конструкторов.

13.2. Структура объектного файла

`ELF` = Executable and Linkable Format

`ELF header`	тип файла; архитектура; смещение, размер и число записей табл. заголовков
`.text`	сегмент кода
`.rodata`	сегмент данных с доступом только на чтение

`.data`	сегмент данных с доступом на чтение/запись
`.bss`	неинициализированные глобальные переменные
`.symtab`	таблица символов
`.rel.text`	адреса в коде, которые необходимо скорректировать при компоновке
`.rel.data`	данные для корректировки глобальных переменных, хранящих адреса
`.debug`	таблица символов для отладки (с ключом `-g`)
`.line`	таблица соответствия номеров строк исх. и машинного кодов (с ключом `-g`)
`.strtab`	таблица строк
section header table	смещения и размеры различных сегментов объектного файла

13.3. Таблица символов

Символические имена, или *символы*, либо являются метками, либо определяются явным образом (при помощи директив). В каждой строке таблицы символов содержится само имя (или указатель на него), его численное значение и иногда некоторая дополнительная информация. Она может включать:

- длину поля данных, связанного с символическим именем;
- биты перераспределения памяти;
- сведения о том, можно ли получить доступ к имени извне процедуры/модуля.

Например, символ таблицы в `ELF` определяется так:

```
typedef struct {
  int name;         /* string table offset */
  int value;        /* section offset, or VM address */
  int size;         /* object size in bytes */
  char type:4,      /* data, func, section, or src file name (4 bits) */
       binding:4;   /* local or global (4 bits) */
  char reserved;    /* unused */
  char section;     /* section header index, ABS, UNDEF, or COMMON */
} Elf_Symbol;
```

С точки зрения компоновщика, существуют следующие виды символов:

- *глобальные*, определяются в модуле и доступны извне. В языке `C` они соответствуют нестатическим функциям и глобальным переменным, объявленным без квалификатора `static`.
- *внешние* — глобальные символы, к которым обращается модуль, но которые определены где-то в другом модуле.
- *локальные*, определение и обращение к которым происходит только внутри модуля. Некоторые локальные символы соответствуют `C` функциям и глобальным переменным, объявленным с квалификатором `static`.

Упражнение. Выведите таблицу символов для объектных файлов.

```
$ gcc -Og -c main.c swap.c
$ readelf --syms main.o swap.o

Num:    Value          Size Type    Bind   Vis      Ndx Name
File: main.o
...
  8: 0000000000000000    24 FUNC    GLOBAL DEFAULT    1 main
...
```

```
 10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND swap
 11: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 buf
File: swap.o
...
  9: 0000000000000000    41 FUNC    GLOBAL DEFAULT    1 swap
 10: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND buf
 11: 0000000000000008     8 OBJECT  GLOBAL DEFAULT  COM bufp1
 12: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    5 bufp0
```

Для каждого символа в таблице, следующей ниже, укажите, находится ли он в символьной таблице модуля `swap.o`; тип: глобальный, локальный или внешний; модуль, где находится определение символа; и сегмент, в котором расположен символ:

Символ	запись в swap.o .symtab?	тип символа	модуль, где определён	сегмент
buf	_____	_____	_____	_____
bufp0	_____	_____	_____	_____
bufp1	_____	_____	_____	_____
swap	_____	_____	_____	_____
tmp	_____	_____	_____	_____

13.4. Ассемблирование за два прохода

Проблема опережающей ссылки: символическое имя используется до своего определения (то есть выполняется обращение к символическому имени, определение которого появится позднее).

Первый проход. Основная задача — построить *таблицу символических имён*.

```
1  void pass_one ()
2  {
3    constexpr int END_STATEMENT {-2};
4    int location_counter {0};
5    int type {0};
6
7    initialize_tables();
8
9    do {
10     string line = read_next_line();
11     int length {0};
12     string opcode;
13
14     if (line_is_not_comment (line))
15     {
16       string symbol = check_for_symbol (line);
17       if (!symbol.empty())
18         enter_new_symbol (symbol, location_counter);
19
20       string literal = check_for_literal (line);
21       if (!literal.empty())
22       enter_new_literal (literal);
23
24       opcode = extract_opcode (line);
```

```
      type = search_opcode_table (opcode);

      if (type < 0)
        type = search_pseudo_table (opcode);

      switch (type)
      {
        case 1: length = get_length_of_type1 (line); break;
        case 2: length = get_length_of_type2 (line); break;
        // ...
      }
    }

    write_temp_file (type, opcode, length, line);

    location_counter += length;

  } while (type != END_STATEMENT);

  rewind_temp_for_pass_two();
  sort_literal_table();
  remove_redundant_literals();
}
```

Второй проход. Основная задача — создание объектного кода и информации, необходимой для компоновки в исполняемый файл. Попутно вывод протокола ассемблирования, если нужно.

```
void pass_two ()
{
  constexpr int END_STATEMENT {-2};
  constexpr int MAX_CODE {16};
  char code[MAX_CODE];
  int location_counter {0};

  do {
    int type = read_type();
    int opcode = read_opcode();
    int length = read_length();
    string line = read_line();

    if (type != 0)  // if not a comment
    {
      switch (type)
      {
        case 1: eval_type1 (opcode, length, line, code); break;
        case 2: eval_type2 (opcode, length, line, code); break;
        // ...
      }
    }

    write_output (code);
    write_listing (code, line);

```

```
27      location_counter += length;
28
29    } while (type != END_STATEMENT);
30
31    finish_up();
32  }
```

13.5. Компоновка

Основные задачи:

– *Идентификация символов.* Объектные файлы определяют символы и обращаются к символам. Цель идентификации, или разрешения, символов — поставить каждому обращению к символу в соответствие единственное определение этого символа.

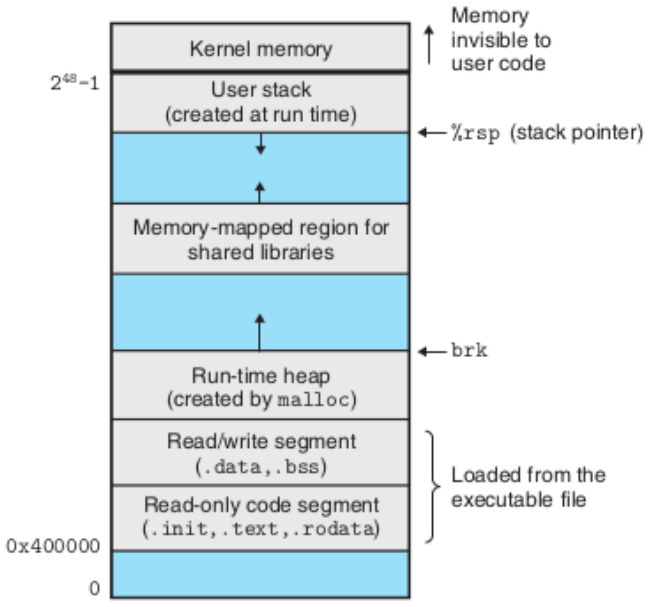
– *Перераспределение адресов.* Компиляторы и ассемблеры создают сегменты кода и данных, которые начинаются с адреса `0`. Компоновщик объединяет и перераспределяет, или перемещает, эти сегменты, определяя местоположение в памяти каждого символа, а затем корректирует все обращения к символам так, чтобы они ссылались в соответствующую область памяти.

13.6. Загрузка на исполнение

Чтобы запусить исполняемый файл `prog` на выполнение, необходимо набрать его имя в командной оболочке:

```
$ ./prog
```

При этом командная оболочка вызовет системный код, называемый загрузчиком. Он скопирует код и данные из исполняемого файла в память, и, затем, запустит программу, передав управление первой инструкции, или *точке входа*. Этот процесс называется *загрузкой* на исполнение.

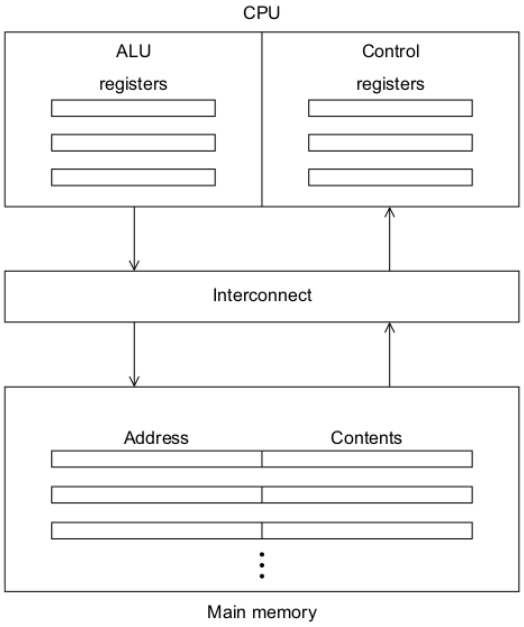


Что читать

Брайант Р. Э., О'Халларон Д. Р. глава 7, стр. 634–660
Таненбаум Э. глава 7, стр. 571–592

14 Архитектура компьютера

14.1. Архитектура фон Неймана



Центральный процессор (ЦП), или central processor unit (CPU), — это мозг компьютера. Его задача — выполнять программы, расположенные в основной памяти. Он вызывает команды из памяти, определяет их тип, а затем выполняет одну за другой. Компоненты соединены *шиной* (bus), представляющей собой набор параллельно связанных проводов, по которым передаются адреса, данные и сигналы управления. Шины могут быть внешними (связывающими процессор с памятью и устройствами ввода–вывода) и внутренними.

Системы RISC (Reduced Instruction Set Computer) и CISC (Complex Instruction Set Computer).

14.2. Процессы, многозадачность и нити

Операционная система (ОС) — существенная часть программного обеспечения, задача которого управлять аппаратными и программными ресурсами компьютера.

Когда пользователь запускает программу, операционная система создаёт *процесс* — копию программы, которая может быть выполнена. Он состоит из нескольких элементов:

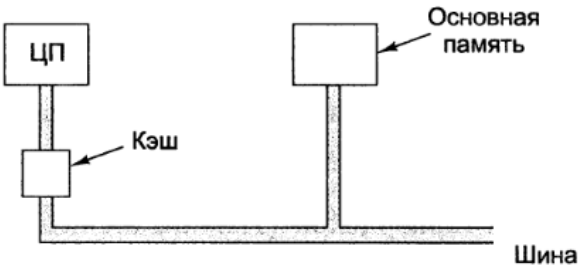
- Код на машинном языке.
- Блок памяти, которая будет включать исполняемый код, *стек вызовов* (call stack), динамически распределяемую память (кучу, heap) и некоторые другие части.
- Дескрипторы ресурсов, которые операционная система выделяет для процесса.
- Права доступа — информация, определяющая какие аппаратные и программные ресурсы доступны данному процессу.
- Информация о состоянии: готов ли процесс к выполнению или ожидает ресурс, содержимое регистров, распределение занимаемой памяти.

Большинство современных операционных систем *многозадачные* (multitasking). Это означает, что они обеспечивают поддержку одновременного выполнения нескольких программ (даже на одном ядре). После того, как одна запущенная программа отработала небольшой интервал времени (несколько миллисекунд), называемый *квантом* (time slice), операционная система может запустить другую программу.

Если в многозадачной ОС процесс ожидает ресурса, то он *блокируется* (block). Это означает, что его выполнение приостанавливается, и операционная система может запустить

другой процесс. *Нити* (threading) исполнения обеспечивают механизм для разделения программы на более или менее независимые задачи так, что когда одна нить блокируется, другая может выполняться. Нить создаётся внутри процесса и разделяет большую часть его ресурсов. Два наиболее важных исключения — счётчик команд и стек вызовов. Поэтому создание и переключение между нитями происходит быстрее, чем между процессами.

14.3. Модификации модели фон Неймана



Основы кэширования. Кэш-память, или *кэш* (cache), — набор блоков памяти, обращение к которым может быть выполнено за меньшее время, чем к основной памяти.

С появлением кэша, сразу возникает вопрос, какие данные и инструкции следует там хранить. В основе универсальной стратегии лежит идея, что программа стремится использовать данные и инструкции, которые физически расположены рядом с теми, что использовались недавно. Эту стратегию часто называют *принципом локальности*. За обращением по некоторому адресу для извлечения инструкции или данных в следующий момент (*временная* локальность) обычно происходит обращение к соседнему адресу (*пространственная* локальность). Ярким примером служит работа с массивами.

Для реализации принципа локальности на практике система использует *шину*, или соединительные провода, достаточно большой ширины. То есть при обращении к памяти в действительности происходят операции с блоками данных и инструкций, которые называют *кэш-блоками* или *строками кэша*.

Концептуально, зачастую удобно думать о кэше ЦП, как о единой структуре. Однако на практике его обычно разделяют на *уровни*: первый уровень (L1) самый маленький и самый быстрый, более высокие уровни (L2, L3, …) больше и медленнее.

Когда центральному процессору необходимо загрузить инструкции или данные, он просматривает кэш: в начале проверяет первый уровень, затем второй и т. д. Если необходимая информация найдена, это называют *кэш-попаданием* (cache hit). А если нет, то это *кэш-промах* (cache miss), и процессор обращается к основной памяти, что может приостановить выполнение текущей программы до тех пор, пока не будут получены данные.

Для измерения эффективности вводят *коэффициент кэш-попаданий* h (hit ratio), который показывает соотношение числа обращений к кэш-памяти и общего числа всех обращений к памяти, и *коэффициент кэш-промахов* (miss ratio), равный $1 - h$.

Пусть c — время доступа к кэш-памяти, m — время доступа к основной памяти. Тогда среднее время доступа:

$$t_{\text{ave}} = c + (1 - h)\, m.$$

Если $h \to 1$, то есть все обращения делаются только к кэш-памяти, то время доступа стремится к c. С другой стороны, если $h \to 0$, то есть каждый раз нужно обращаться к основной памяти, то время доступа стремится к $c + m$: сначала требуется время c для проверки кэш-памяти (в данном случае безуспешной), а затем — время m для обращения к основной памяти.

Когда ЦП пишет данные в кэш, значение в кэше и основной памяти становятся различными, или *несогласованными* (inconsistent). Существует два подхода устранить несогласованность. Немедленное обновление значения в основной памяти называется *сквозной записью* (write-through). Этот подход обычно гораздо проще реализуется и, к тому же, более надёжен, поскольку современная память при ошибке может восстановить своё предыдущее состояние. К сожалению, при этом приходится передавать больше данных в память, поэтому в сложных проектах стремятся использовать альтернативный подход — *отложенную запись* (write-back). Обновлённые данные в кэше помечаются как «грязные» (dirty), и когда строка кэша заменяется новой строкой из памяти, «грязная» строка записывается обратно в память.

Отображение кэш-памяти на основную память:

- *полностью ассоциативный* кэш: строка может быть помещена в любую позицию;
- кэш *прямого отображения*: строка связана с фиксированной позицией определяемой, например, младшими битами адреса в основной памяти;
- *n-входовый ассоциативный* кэш являет собой промежуточный вариант.

Виртуальная память. Современные операционные системы поддерживают одновременную работу множества задач, которые вынуждены разделять ресурс памяти меж собой. Кроме того, необходимо защищать данные и инструкции каждого процесса от возможного повреждения со стороны других процессов.

Виртуальная память (virtual memory) разработана так, чтобы основная память могла быть использована как кэш для вторичного хранилища (обычно в дисковом пространстве). Она использует принцип пространственной и временной локальности, сохраняя в основной памяти только активные части запущенных программ; те части, что простаивают, хранятся в блоке вторичного хранилища, называемом *файлом подкачки* (swap space). Подобно кэшу, виртуальная память оперирует с блоками данных и инструкций. Эти блоки обычно называют *страницами* (pages). Поскольку доступ ко вторичному хранилищу может быть в сотни тысяч раз медленнее, чем к основной памяти, страницы делают относительно большими — большинство систем имеют фиксированный размер страниц, который на данный момент лежит в диапазоне от 4 до 16 Кб.

При компиляции программы используются виртуальные адреса. Когда программа запускается, создаётся *таблица страниц* (page table) для отображения виртуальных страниц на физические адреса. Обычно создание таблиц осуществляет операционная система, что позволяет гарантировать, что участки памяти, используемые одним процессом, не пересекается с памятью других процессов.

Недостаток использования таблиц в удвоении времени доступа к основной памяти (обращение к таблице плюс обращение в память по целевому физическому адресу). Для решения этой проблемы процессоры содержат специальный кэш для преобразования адресов, называемый *буфером быстрого преобразования* (translation-lookaside buffer, TLB).

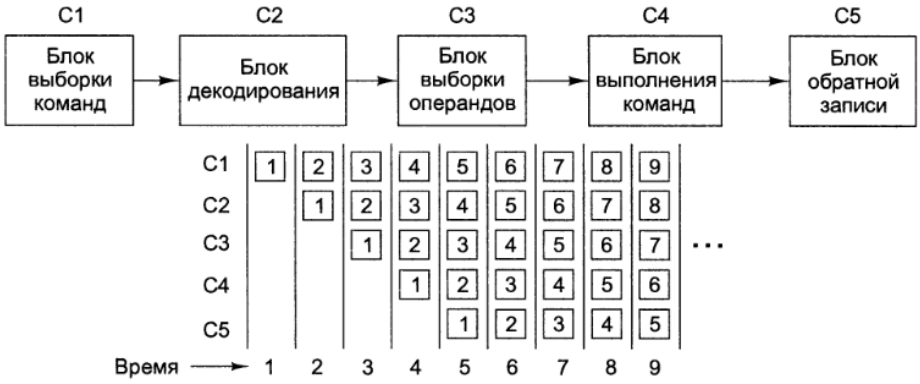
Вследствие медленной работы диска виртуальная память всегда использует отложенную запись, если страница была изменена в процессе работы программы. А управление таблицей страниц и обращением к диску можно поручить операционной системе (совместно с аппаратным обеспечением).

Параллелизм выполнения на уровне инструкций позволяет улучшить производительность процессора за счёт разделения на компоненты, или *функциональные блоки* (functional units), одновременно выполняющие инструкции.

Существует два основных подхода:

- *конвейер* (pipeline), принцип работы схож с заводским конвейером;

– *суперскалярная архитектура* (superscalar architecture), один конвейер с большим количеством функциональных блоков.



Многонитевое аппаратное обеспечение предоставляет средства для продолжения выполнения полезной работы, когда задача, выполняющаяся в данный момент, была приостановлена, например, в ожидании загрузки данных из памяти. В этот момент, возможно, имеет смысл просто запустить другую нить. Конечно, для этого система должна поддерживать очень быстрое переключение между нитями.

14.4. Параллельное аппаратное обеспечение

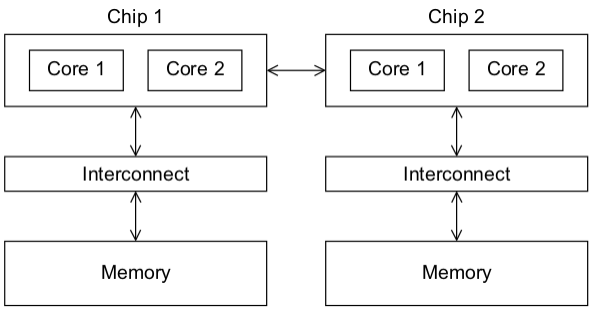
Суперскалярную архитектуру и конвейер также можно рассматривать как параллельное аппаратное обеспечение, поскольку функциональные блоки дублируются. Однако, эта форма параллелизма обычно не видна программисту, и её можно рассматривать как расширение исходной модели фон Неймана. Под *параллельным обеспечением* будем понимать обеспечение, которое требует от программиста изменений в исходном коде.

SIMD системы. Single Instruction-stream Multiple Data-stream:

– *векторные процессоры* (vector processors);
– *графические процессоры* (graphics processing units, GPU)

MIMD системы. Multiple Instruction-stream Multiple Data-stream:

– *системы с общей памятью* (shared-memory systems), используют один или более многоядерных процессоров, связанных с единой памятью. Выделяют системы с *однородным доступов к памяти* (uniform memory access, UMA) и *неоднородным доступом к памяти* (nonuniform memory access, NUMA);



– *системы с распределённой памятью* (distributed-memory systems). Обычно *узлами* (nodes) таких систем являются системы с общей памятью, поэтому их иногда называют *гибридными* (hybrid systems). *Глобальная сеть* (grid) обеспечивает инфраструктуру, необходимую для объединения больших сетей географически распределённых компьютеров в одно целое. В общем, такие системы будут *гетерогенными* (heterogeneous), то есть индивидуальные узлы могут иметь разную архитектуру.

Упражнения

Рассмотрим влияние кэширования на производительность программы. Для этого создадим структуру данных, которая хранит целочисленный идентификатор и массив символов фиксированного размера.

```
13 struct Data
14 {
15   const int id;
16   std::array<char, 8192> line {};
17
18   Data (const int an_id) : id{ an_id }  { }
19 };
```

Тестовая оснастка. Напишем функцию, которая создаёт тестовый массив заданного размера, заполняя поле `id` элементов псевдослучайными значениями в диапазоне от `0` до `high`.

```
22 auto generate_data (size_t n, int high = std::numeric_limits<int>::max())
23 {
24   std::default_random_engine ran{};
25   std::uniform_int_distribution<> uni{ 0, high };
26
27   std::vector<Data> data;
28   data.reserve (n);
29   for (size_t i = 0; i < n; ++i)
30   {
31     int id = uni (ran);
32     data.emplace_back (id);
33   }
34
35   return data;
36 }
```

Здесь используются возможности стандартной библиотеки `<random>`. При желании, можно выбрать другое распределение, отличное от равномерного. Обратим внимание на пустые скобки при указании шаблонного типа. Объявление класса выглядит примерно так:

```
template<class IntType = int>
class uniform_int_distribution;
```

Видно, что параметр по умолчанию — тип `int`. Именно поэтому мы можем его не указывать явно. А угловые скобки говорят о том, что это шаблон.

Напишем тестовую функцию, которая запускает два различных варианта упорядочивания элементов в массиве заданного размера и измеряет время выполнения каждого алгоритма. Добавим проверку, что оба варианта приводят к одинаковому порядку. В качестве таймера можно использовать класс-обёртку `Timer`, описание которого приведём ниже.

```
80 void test (size_t n)
81 {
82   auto data = generate_data (n);
83
84   std::vector<const Data*> ord;
85
```

```
  std::cout << sizeof(Data) <<" "<< data.size() << std::flush;
  for (auto& get_order : {get_order_1, get_order_2})
  {
    Timer timer;
    auto tmp = get_order (data);
    timer.stop();

    if (ord.empty())
      ord = tmp;
    else if (ord != tmp)
      throw std::runtime_error{"sorting order not the same"};

    std::cout <<" "<< timer.elapsed() << std::flush;
  }
  std::cout << std::endl;
}
```

Функцию `main()` нашей тестовой программы можно записать следующим образом:

```
int main ()

{
  std::vector<size_t> lengths {
    200, 500, 1'000,
    5'000, 10'000, 50'000, 100'000,
    200'000, 500'000, 800'000, 1'000'000,
  };

  for (size_t n : lengths)
    test (n);
}
```

Класс `Timer`. Это небольшой класс-обёртка, использующий стандартную библиотеку `<chrono>`. Его реализация проста и очевидна, не требует дополнительных комментариев. Код следует разместить в файле `timer.h` и подключить к коду остальной тестовой оснастки.

```
using namespace std::chrono;

class Timer
{
public:
  Timer () { start(); }

  void start () { s = f = high_resolution_clock::now(); }
  void stop ()  { f = high_resolution_clock::now(); }

  double elapsed () const;  // elapsed time in milliseconds


private:
  high_resolution_clock::time_point s,  // start
                                    f;  // finish
};

```

```
25
26 inline
27 double Timer::elapsed () const
28 {
29   return duration_cast<microseconds>(f - s).count() / 1000.;
30 }
```

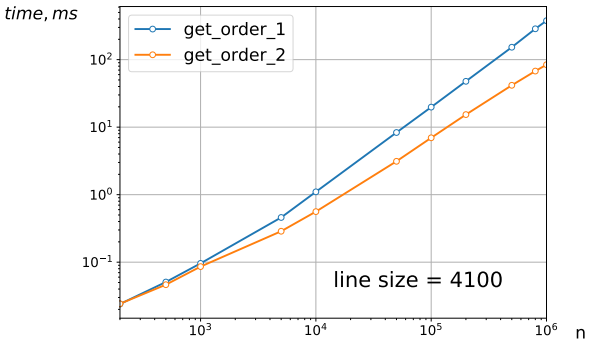
Теперь рассмотрим первый вариант сортировки. Упорядочивать будем по полю `id`. Положим, что элементы массива достаточно велики, чтобы пренебречь временем копирования при обменах. Для решения этой проблемы создадим массив указателей и упорядочим его. (В случае необходимости легко получить новый, упорядоченный, массив исходных элементов, используя указатели.)

```
39 auto get_order_1 (const std::vector<Data>& data)
40 {
41   std::vector<const Data*> ord;
42   ord.reserve (data.size());
43
44   std::transform (data.begin(), data.end(), std::back_inserter(ord),
45                   [] (const Data& x) { return &x; });
46
47   std::sort (ord.begin(), ord.end(),
48              [] (const auto& p, const auto& q) { return p->id < q->id; });
49
50   return ord;
51 }
```

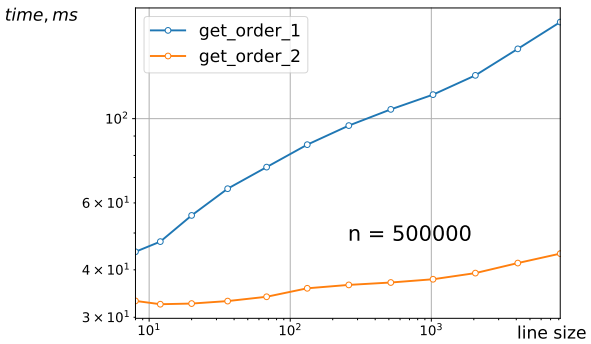
Второй вариант сделаем чуть хитрее. Добавим вспомогательный массив пар (`указатель, id`) и выполним упорядочивание без косвенного доступа к исходным элементам.

```
54 auto get_order_2 (const std::vector<Data>& data)
55 {
56   std::vector<std::pair<const Data*, size_t>> ord;
57   ord.reserve (data.size());
58
59   std::transform (data.begin(), data.end(), std::back_inserter(ord),
60                   [] (const Data& x)
61   {
62     return std::make_pair (&x, x.id);
63   });
64
65   std::sort (ord.begin(), ord.end(),
66              [] (const auto& p, const auto& q)
67   {
68     return p.second < q.second;
69   });
70
71   std::vector<const Data*> out;
72   out.reserve (ord.size());
73   for (const auto& p : ord)
74     out.emplace_back (p.first);
75
76   return out;
77 }
```

Проведите серию запусков программы и постройте несколько графиков подобных тем, что приведены на рисунке 14.1.



а) в зависимости от количества элементов



б) в зависимости от размера самого элемента

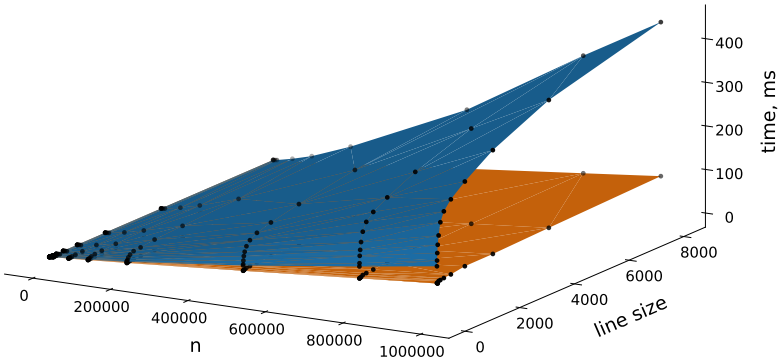


Рис. 14.1 — Влияние кэширования на производительность программы.

Объясните полученное поведение кривых. Используйте знания об устройстве и работе кэша, изложенные выше.

Что читать

Хэррис Д. М., *Хэррис С. Л.* глава 3, стр. 390–398
Хэррис Д. М., *Хэррис С. Л.* глава 7, стр. 1091–1131, глава 8, стр. 1163–1240
Pacheco P. S. глава 2, стр. 15–46

Список литературы

1. *Хэррис Д. М.*, *Хэррис С. Л.* Цифровая схемотехника и архитектура компьютера : пер. с англ. — Morgan Kaufman, 2015. — 1627 с.
2. *Брайант Р. Э.*, *О'Халларон Д. Р.* Компьютерные системы: архитектура и программирование / пер. с англ. А. Киселева. — 3-е изд. — М. : ДМК Пресс, 2022. — 994 с.
3. *Таненбаум Э.*, *Остин Т.* Архитектура компьютера : пер. с англ. — 6-е изд. — СПб. : Питер, 2013. — 816 с.
4. *Винокуров Н. А.*, *Ворожцов А. В.* Практика и теория программирования : в 2 т. — М. : Физматкнига, 2008. — (Информатика).
5. *Pacheco P. S.* An introduction to parallel programming. — Morgan Kaufmann Publishers, 2011. — 370 p.
6. Ассемблер в Linux для программистов С. — URL: https : / / ru . wikibooks . org / wiki / Ассемблер_в_Linux_для_программистов_С.
7. *Зубков С. В.* Assembler для DOS, Windows и UNIX. — М. : ДМК Пресс, 2000. — 608 с. — (Для программистов).
8. Intel® 64 and IA-32 Architectures Software Developer's Manual. In 7 vols. Vol. 1. Basic Architecture. — 2014.
9. *Таненбаум Э.* Архитектура компьютера : пер. с англ. — 5-е изд. — СПб. : Питер, 2007. — 844 с.

Приложение

Установка и настройка рабочей среды

Добавить материал