###### **МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

###### **ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

###### **НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

###### **Факультет информационных технологий**

**Кафедра параллельных вычислений**

**ОТЧЕТ** **О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ**

**Практическая работа №4**

ВВЕДЕНИЕ В АРХИТЕКТУРУ ARM

студента 2 курса, группы 23201

Сорокина Матвея Павловича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

А.С. Матвеев

Новосибирск 2024

**СОДЕРЖАНИЕ**

[**ЦЕЛЬ** 3](#_Toc175850303)

[**ЗАДАНИЕ** 3](#_Toc175850304)

[**ОПИСАНИЕ РАБОТЫ** 4](#_Toc175850305)

[**ЗАКЛЮЧЕНИЕ** 8](#_Toc175850306)

[**ПРИЛОЖЕНИЯ** 9](#_Toc175850307)

[**Приложение 1:Исходный код программы *LogCalculation.cpp*** 9](#_Toc175850308)

[**Приложение 2: Разбивка ассемблерного кода с объяснением размещения переменных функции LogCalculation(long double x, long long n)** 10](#_Toc175850309)

[**Приложение 3: ассемблерный листинг программы LogCalculation.cpp с ключом контроля оптимизации -O0, архитектурой ARM.** 11](#_Toc175850310)

# **ЦЕЛЬ**

Знакомство с программной архитектурой ARM и анализ ассемблерного листинга программы для архитектуры ARM.

# **ЗАДАНИЕ**

1. Изучить основы программной архитектуры ARM.
2. Для программы на языке Си (из лабораторной работы 1) сгенерировать ассемблерные листинги для архитектуры ARM, используя различные уровни комплексной оптимизации.
3. Проанализировать полученные листинги и сделать следующее:

* Сопоставьте команды языка Си с машинными командами.
* Определить размещение переменных языка Си в программах на ассемблере (в каких регистрах, в каких ячейках памяти).
* Описать и объяснить оптимизационные преобразования, выполненные компилятором.
* Продемонстрировать использование ключевых особенностей архитектуры ARM на конкретных участках ассемблерного кода.

1. Составить отчет, отражающий этапы работы, результаты анализа, выводы.

# **ОПИСАНИЕ РАБОТЫ**

1. Реализовано задание 2 - алгоритм вычисления числа Пи метом Монте-Карло на языке *C++*.

Время замерялось перед началом и после окончания работы функции, вычисляющей число Пи. Разность этих двух значений дает общее время выполнения функции. Для проверки точности измерений, код программы запускается несколько раз.

Код программы предоставлен (см. Приложение 1).

1. Ассемблерные листы с различными уровнями оптимизации генерировались с помощью:

* Компилятора:

Установил кросс-компилятор для ARM:

|  |
| --- |
| sudo apt-get install gcc-arm-linux-gnueabi |

Затем использовал следующую команду для генерации ассемблерного листинга:

|  |
| --- |
| arm-linux-gnueabi-g++ -S -o arm32.s PiCalculation.cpp |

***aarch64-linux-gnu-g++:*** используется для вызова компилятора C++ для архитектуры *ARM*, который создает исполняемые файлы для 32-битных систем с *ABI* - *GNU EABI* (*Embedded Application Binary Interface*).

Скомпилированный код будет работать на 32-битной версии ARM.

* Сайта: <https://godbolt.org>

Использовался компилятор ***ARM GCC 14.2.0***

1. Используемые регистры

Архитектура *ARM32* предоставляет 16 регистр **общего назначения**, которые применяются для любых задач, имеют размер 32 бита и называются .

Описание регистров общего назначения архитектуры ARM32:

* + **r0-r3** – регистры аргументов. ***r0*** также выполняет функцию возвратного значения подпрограммы.
  + **r4-r12 – локальные переменные**.
  + **r13 – stack pointer (SP).** Указатель на вершину стека. Стек растёт вниз, а значение ***SP*** указывает на последнее помещённое в него слово.
  + **r14 – link register (LR).** Сохраняет адрес возврата при вызове функции.
  + **r15 – program counter (PC).** Используется для хранения адресов выполняющихся команд. При последовательном выполнении команд он последовательно увеличивается, при переходах и прерываниях в него загружается новое значение.

Регистр ***SP****(****Stack Pointer****)* перекрывает регистр **r13**. Хранит адрес вершины стека. Область стека используется программами для хранения и доступа к данным локальных переменных для данной функции, а также в качестве временной памяти для хранения таких данных, как адреса возврата функции.

Регистр ***LR****(****Link Register****)* перекрывает регистр **r14**. Основная цель - хранить адреса возврата при вызове функции, однако его также можно использовать для обычных вычислений.

Регистр ***PC****(****Program Counter****)* - содержит адрес текущей инструкции. Перед выполнением каждая инструкция загружается из области памяти, на которую ссылается регистр PC. После выполнения инструкции PC автоматически переходит к следующей инструкции.

Регистр ***CPSR*** (***Current Program Status Register***) – 32-битный, хранит однобитовые флаги состояния, которые могут устанавливаться в зависимости от выполняемых инструкций и их результатов:

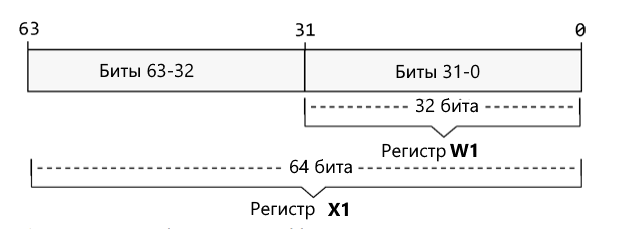
* + N - флаг знака, устанавливается, если результат операции представляет отрицательное число.
  + Z - флаг нуля, устанавливается, если результат операции равен нулю.
  + C - флаг переноса, устанавливается, если при выполнении арифметической операции произошел перенос.
  + V - флаг переполнения, устанавливается, если при выполнении арифметической операции произошло переполнение со знаком.

CPSR всегда доступен процессору и обновляется после каждой операции, которая изменяет флаги состояния, например, после выполнения арифметических операций (сложение, вычитание, сравнение) или операций с плавающей точкой.

Пример изменения флагов состояния (результатов арифметических операций) *CSPR*:

|  |
| --- |
| mov r0, #0 @ заносим в регистр r0 число 0  cmp r0, #0 @ сравниваем r0 с 0  @ если r0 = 0, устанавливается флаг Z  @ если результат отрицательный, флаг N |

В архитектуре *ARM64* доступны 32-битные регистры **w0–w30**, которые представляют младшие 32 бита соответствующих 64-битных регистров x0-x30.



Описание 31 регистра общего назначения архитектуры *ARM64*:

* **x0–x7 - регистры аргументов**. Используются для передачи параметров в функцию и возвращения результата. Также они используются как временные регистры для хранения промежуточных значений. Они не сохраняются при вызове подпрограмм, то есть если функция вызывает другую функцию, эти регистры могут быть перезаписаны (caller-saved).
* **x8** - регистр для **непосредственного результата** (indirect result location). Используется для передачи в функцию адреса блока памяти при непрямой адресации.
* **x9–x15 - временные регистры** для промежуточных значений. Их следует сохранять при вызове функций. Ответственность за сохранение ложится на вызывающую сторону, которая вызывает функцию. Затронутые регистры сохраняются в кадре стека вызывающей функции, что позволяет подпрограмме изменять эти регистры. (caller-saved)
* **x16–x18 -** используются для хранения промежуточных результатов между вызовами функций.
* **x19–x28** - **локальные переменные**. Регистры, сохраняемые вызываемой функцией в стеке, что позволяет функции изменять эти регистры, но также требует их восстановления перед возвратом к вызывающей стороне. (callee-saved)
* **x29 - frame pointer (FP)**. Указатель на фрейм стека.
* **x30 - link register (LR)**. Сохраняет адрес возврата при вызове функции, то есть используется для возврата в вызывающую функцию.

Для операций с числами с плавающей точкой применяется **FPU** или **Floating-point Unit**. Также используется отдельный набор специальных регистров, которые предназначены для другого элемента в архитектуре ARM - **сопроцессора NEON**, который применяется в параллельных вычислениях.

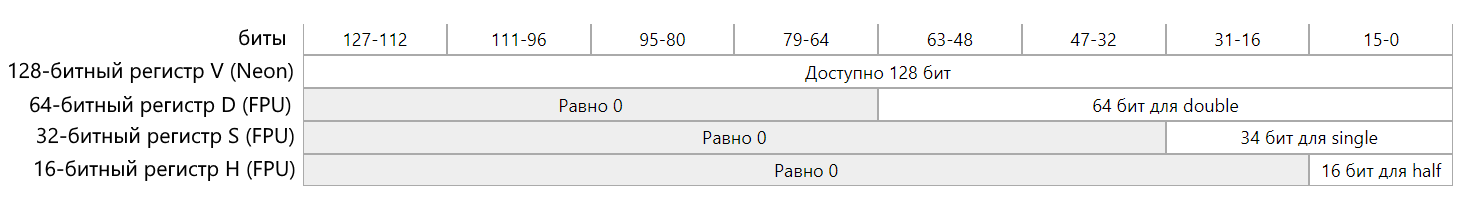
Сопроцессор NEON имеет 32 128-разрядных регистра, которые называются **v0** - **v31.**

Для FPU доступны только младшие 64 бит регистров Vn. Эти 64-разрядные регистры называются **d0 – d31**.

Также FPU может обращаться к младшим 32 битам регистров как к регистрам **s0** - **s31.**

Можно обращаться к младшим 16 битам этих регистров как к регистрам **h0** - **h31.**

Отметим, что на аппаратном уровне имеем 128-разрядные регистры **Vn**, но для FPU доступны только младшие 64-разрядные регистры **Dn** (либо 32-разрядные **Sn**, либо 16-разрядные **Hn**)



1. Сопоставление команд языка Си с машинными командами, размещение переменных языка Си в программах на ассемблере на примере функции *double PiCalculation(long long n).*

Функция на языке *C++:*

|  |
| --- |
| double PiCalculation(long long n) {      long long m = 0;      srand(time(NULL));      for (long long i = 0; i < n; i++) {          double x = static\_cast<double>(rand()) / RAND\_MAX \* 2 - 1.0;          double y = static\_cast<double>(rand()) / RAND\_MAX \* 2 - 1.0;          if (x \* x + y \* y <= 1.0) {              m++;          }      }      return 4 \* m / n;  } |

Разбивка ассемблерного кода с пояснениями размещения переменных предоставлена (см. Приложение 2).

На языке ассемблера 32-битной архитектуре данная подпрограмма образует стековый фрейм следующим образом:

1. Инструкция ***ldrd*** (***load register data***) сохраняет параметр функции в регистр *r0* со стека *[r7+72]* до вызова рассматриваемой функции:

|  |
| --- |
| ldrd    r0, [r7, #72] |

1. С помощью инструкции ***bl*** вызывается подпрограмма *PiCalculation(long long n):*

|  |
| --- |
| bl      PiCalculation(long long) |

Данная инструкция сохраняет адрес последующей за вызовом функции инструкции в регистр *lr*, затем осуществляет безусловный переход к метке *PiCalculation(long long).*

Стек:

|  |
| --- |
| **previous values** |

3.3 – 3.5 – **Пролог функции**

1. В теле самой подпрограммы первым делом осуществляется размещение используемых в функции регистров:

|  |
| --- |
| push    {r4, r5, r7, r8, r9, r10, fp, lr} |

***r4 - r10*** – callee-saved регистры для локальных переменных;

***fp*** - указатель на фрейм стека;

***lr*** – адрес возврата при вызове функции (обновлен в 3.2).

Стек:

|  |
| --- |
| **previous values** |
| **r4-lr** |

1. Резервация места (48 байт) на стеке для локальных переменных подпрограммы:

|  |
| --- |
| sub     sp, sp, #48 |

Стек:

|  |
| --- |
| **previous values** |
| **r4-lr** |
| **local variables (48 bytes)** |

1. Копирование значения регистра ***sp*** (обновлен в 3.4) в регистр ***r7***

|  |
| --- |
| add     r7, sp, #0 |

1. Сохранение параметра функции *n*, хранящегося в регистре ***r0***, на стек со смещением *[r7+8]*

|  |
| --- |
| strd    r0, [r7, #8] |

Стек:

|  |
| --- |
| **previous values** |
| **r4-lr** |
| **local variables (40 bytes)** |
| **n** |

Между этапами 3.6 и 3.7 подразумевается выполнение операций подпрограммы, расположенных между прологом и эпилогом. Подробно с ними можно ознакомиться в Приложении 2.

1. **Эпилог функции** – деаллокация локальных переменных и восстановление регистра ***sp*** - указателя на вершину стека

|  |
| --- |
| adds    r7, r7, #48  mov     sp, r7  pop     {r4, r5, r7, r8, r9, r10, fp, pc} |

Стек:

|  |
| --- |
| **previous values** |

1. Размещение переменных функции *double PiCalculation(long long n)* в ассемблереном листинге

Входные параметры:

* *long long n*: значение передается в регистре r0 и сохраняется на стеке по адресу *[r7+8].*

Локальные переменные:

* *m* (счетчик попаданий): сохраняется по адресу *[r7+40]* с помощью команды *vstr.64*;
* *i* (счетчик всех точек): сохраняется по адресу *[r7+32]* командой *vstr.64*;
* *x* (сгенерированное случайное значение): адрес: *[r7+24];*
* *y* (сгенерированное случайное значение): адрес: *[r7+16].*

1. Использование ключевых особенностей архитектуры *ARM* на конкретных участках ассемблерного кода.
   * + - **Использование регистров для передачи параметров и возврата значений**

Передача параметров в функцию часто осуществляется через регистры, а не через стек, что ускоряет вызов функции.

Пример из функции *double PiCalculation(long long n):*

|  |
| --- |
| vstr.64 d16, [r7, #16] |

В примере выше *y* (локальная переменная функции) передается через регистр *d16* и сохраняется в стеке по адресу *[r7+16]* для последующего использования.

* + - * **Инструкции для работы с числами с плавающей точкой**

Инструкции ***vadd.f64, vmul.f64***, ***vdiv.f64***, позволяют нам выполнять арифметические операции с числами с плавающей точкой.

Именно суффикс ***.f64*** указывает на то, что эти операции выполняются с 64-битными числами с плавающей точкой (double в *C++*).

Пример:

|  |
| --- |
| vadd.f64 d16, d16, d16 ; d16 \*= 2 ([0, 2]) |
| vmul.f64 d17, d16, d16 ; d17 = x^2 |
| vdiv.f64 d16, d17, d18 |

В примере 3 два значения с плавающей точкой, хранящиеся в регистрах *d17* и *d18*, делятся друг на друга, результат сохраняется в регистре *d16*.

* + - * **Инструкции управления стеком**

*ARM* использует стек для сохранения контекста при вызове функций, примером являются инструкции **push** и **pop**.

Пример:

|  |
| --- |
| push {r4, r5, r7, r8, r9, r10, fp, lr} ; сохраняем регистры общего назначения, адрес возврата на стеке |
| pop {r4, r5, r7, r8, r9, r10, fp, pc} ; восстановление регистров и возврат из функции |

Когда вызывается функция, адрес следующей инструкции записывается в регистр ***lr***. Это позволяет программе знать, куда нужно вернуться после выполнения функции.

Когда функция завершает выполнение, нужно вернуться к точке, откуда она была вызвана. Для этого из стека восстанавливается значение ***pc*** регистра, который указывает на адрес следующей выполняемой инструкции. При записи значения в ***pc*** происходит переход на этот адрес.

Важно заметить, что инструкция

|  |
| --- |
| pop {pc} |

Равносильна двум инструкциям

|  |
| --- |
| pop {lr}  mov pc, lr |

1. Оптимизация ассемблерного кода при оптимизации ***-O1***

Разбивка ассемблерного кода с объяснением размещения переменных функции *double PiCalculation(long long n)* с уровнем компиляции *-O1* представлена (см. Приложение 3).

Основные положительные изменения:

* + Используются регистры для хранения промежуточных значений и переменных. Больше не резервируется место на стеке.
  + Для локальных переменных используются разделенные регистры.
  + Удаление ненужных инструкций и регистров, например – регистр ***fp***, который больше не используется в подпрограмме.
  + Перед началом цикла сравниваем параметр функции *n* с 1, если *n < 1*, то основная часть цикла пропускается.

1. Оптимизация ассемблерного кода при оптимизации ***-O2***

Разбивка ассемблерного кода с объяснением размещения переменных функции *double PiCalculation(long long n)* с уровнем компиляции *-O1* представлена (см. Приложение 3).

На данном уровне оптимизации замечено одно существенное отличие: в случае если параметр функции *n <1*, то мы переходим в эпилог в отдельной метке *.L6*, в которой приписываем искомому числу Пи значение 0 и выполняем эпилог функции, отличный от основного эпилога из метки *.L3*, в то время как при уровне ***-O1***, сначала переходил в метку .*L6*, в которой присваивали локальной переменной m значение 0, затем выполнял ненужные рифметические операции: *4 \* m / n*, после чего уже выполняли эпилог функции.

# **ЗАКЛЮЧЕНИЕ**

В ходе данной лабораторной работы я познакомился с программной архитектурой ARM и научился анализировать ассемблерный листинг программы для архитектуры ARM.

# **ПРИЛОЖЕНИЯ**

# **Приложение 1:Исходный код программы *PiCalculation.cpp***

|  |
| --- |
| #include <iostream>  #include <sys/times.h> // for times  #include <unistd.h> // for sysconf  #include <cstdlib> // for atoi and atof  double PiCalculation(long long n) {      long long m = 0;      srand(time(NULL));      for (long long i = 0; i < n; i++) {          double x = static\_cast<double>(rand()) / RAND\_MAX \* 2 - 1.0;          double y = static\_cast<double>(rand()) / RAND\_MAX \* 2 - 1.0;          if (x \* x + y \* y <= 1.0) {              m++;          }      }      return 4 \* m / n;  }  int main(int argc, char \*argv[]) {      if (argc != 2) {          std::cerr << "Invalid number of arguments" << std::endl;          return 0;      }        struct tms start, end;      long clocks\_per\_sec = sysconf(\_SC\_CLK\_TCK);      long long n = atoll(argv[1]);      std::cout << "N = " << n << std::endl;      int runs = 5;      double min\_time = 10000;      for (long long i = 0; i < runs; i++) {          times(&start);          double pi = PiCalculation(n);          times(&end);          double taken\_time = static\_cast<double>(end.tms\_utime - start.tms\_utime) / static\_cast<double>(clocks\_per\_sec);          std::cout << "pi = " << pi << std::endl;          std::cout << "Run time: " << taken\_time << "\n" << std::endl;          min\_time = std::min(min\_time, taken\_time);      }      std::cout << "Min time: " << min\_time << " seconds" << std::endl;      return 0;  } |

# **Приложение 2: Разбивка ассемблерного кода с объяснением размещения переменных функции *double PiCalculation(long long n) с уровнем компиляции -O0***

|  |
| --- |
| PiCalculation(long long):  **; пролог**  push {r4, r5, r7, r8, r9, r10, fp, lr} ; сохраняем регистры общего назначения на стеке, ими позже будем пользоваться в функции  sub sp, sp, #48 ; выделяем 48 байт на стеке для локальных переменных  add r7, sp, #0 ; (указатель на стек сохраняется в r7  strd r0, [r7, #8] ; сохранение параметра n (в r0) на стеке по адресу [r7 + 8]  ; **long long m = 0;**  vmov.i32        d16, #0  @ di ; инициализация переменной di (счетчик внутри круга) **(m = 0)**  vstr.64 d16, [r7, #40]    @ int ; сохраняем переменную di в [r7 + 40] (m в [r7 + 40])    **; srand(time(NULL));**  movs r0, #0 ; устанавливаем r0 в 0 для вызова time()  bl time ; вызов функции time() для инициализации генератора случайных чисел  mov r3, r0 ; сохраняем текущее время (результата time()) в r3  mov r0, r3 ; передача результата time() как аргумент для srand  bl srand ; вызов srand() для инициализации генератора случайных чисел    **; long long i = 0;**  vmov.i32        d16, #0  @ di ; инициализация переменной di (счетчик всех точек)  vstr.64 d16, [r7, #32] ; сохраняем переменную di в [r7 + 32] (i в [r7 + 32])  b .L2 ; переход к основной части цикла  .L5:  **; double x = static\_cast<double>(rand()) / RAND\_MAX \* 2 - 1.0;**  bl rand ; генерация случайного числа  vmov s15, r0 ; перемещение результата в s15 (преобразовали в 32-битный регистр)  vcvt.f64.s32 d17, s15 ; преобразование случайного числа в double  vldr.64 d18, .L8 ; загрузка RAND\_MAX из метки .L8 в регистр d18  vdiv.f64 d16, d17, d18 ; **d16 = static\_cast<double>(rand()) / RAND\_MAX** ([0, 1])  vadd.f64 d16, d16, d16 ; d16 \*= 2 ([0, 2])  vmov.f64 d17, #1.0 ; загружаем 1.0 в d17  vsub.f64 d16, d16, d17 ; d16 -= 1.0 (приведение в диапазон [-1, 1])  vstr.64 d16, [r7, #24] ; сохраняем значение d16 (x) в [r7 + 24] на стеке    **; double y = static\_cast<double>(rand()) / RAND\_MAX \* 2 - 1.0;**  bl rand  vmov s15, r0  vcvt.f64.s32 d17, s15  vldr.64 d18, .L8  vdiv.f64 d16, d17, d18  vadd.f64 d16, d16, d16  vmov.f64 d17, #1.0  vsub.f64 d16, d16, d17  vstr.64 d16, [r7, #16] ; сохранение y на стеке в [r7 + 16]  **; if (x \* x + y \* y <= 1.0) {**  vldr.64 d16, [r7, #24] ; загружаем x в d16  vmul.f64 d17, d16, d16 ; d17 = x^2  vldr.64 d16, [r7, #16] ; загружаем y в d16  vmul.f64 d16, d16, d16 ; d16 = y^2  vadd.f64 d16, d17, d16 ; d16 = x^2 + y^2  vmov.f64 d17, #1.0 ; загружаем 1.0 в d17  vcmpe.f64 d16, d17 ; сравниваем d16 и 1.0  vmrs APSR\_nzcv, FPSCR ; копирование флагов состояния из регистра состояния плавающей точки (FPSCR — Floating Point Status and Control Register) в регистр состояния программы APSR (Application Program Status Register)  bhi .L3 ; если x^2 + y^2 > 1, переход к .L3    **; если x^2 + y^2 <= 1**  ldrd r2, [r7, #40] ; загружаем счетчик попаданий (m) в r2  adds r8, r2, #1 ; **m++**  adc r9, r3, #0  strd r8, [r7, #40] ; сохраняем обновленный счетчик попаданий в [r7 + 40]  .L3:  ldrd r2, [r7, #32] ; загружаем i в r2  adds r10, r2, #1 ; **i++**  adc fp, r3, #0  strd r10, [r7, #32] ; сохраняем обновленный счетчик i в [r7 + 32]  .L2:  **; i < n;**  ldrd r0, [r7, #32] ; загружаем i в r0  ldrd r2, [r7, #8] ; загружаем n в r2  cmp r0, r2 ; сравниваем i с n  sbcs r3, r1, r3  blt .L5 ; если i < n, переходим к .L5 (продолжаем цикл)  **; return 4 \* m / n;**  ldrd r2, [r7, #40] ; загружаем m в r2  adds r4, r2, r2 ; r4 = 2 \* m  adc r5, r3, #0 ; r5 = 2 \* всех точек (n)  adds r3, r4, r4 ; r3 = 4 \* m (r4 + r4 = 2\*m + 2\*m = 4\*m)  str r3, [r7] ; сохраняем 4\*m в r7  adc r3, r5, r5 ; увеличиваем r3 на 2 \* n  str r3, [r7, #4] ; сохраняем 2 \* n в [r7, #4]  ldrd r0, [r7] ; загружаем 4\*m в r0  ldrd r2, [r7, #8] ; загружаем n в r2  bl \_\_aeabi\_ldivmod ; делим 4\*m на n (с остатком)  mov r2, r0 ; сохраняем целую часть деления в r2  mov r3, r1 ; сохраняем остаток в r3  mov r0, r2 ; готовим r0 для преобразования в double  mov r1, r3 ; готовим r2 для преобразования в double  bl \_\_aeabi\_l2d ; преобразование целого числа в double  mov r2, r0 ; загружаем результат в r2  mov r3, r1 ; загружаем результат в r3  vmov d16, r2, r3 ; загружаем результат в d16  vmov.f64 d0, d16  **; возвращаем результат в d0**    **; эпилог**  adds r7, r7, #48 ; освобождаем память на стеке  mov sp, r7 ; восстанавливаем указатель стека к изначальному состоянию  pop {r4, r5, r7, r8, r9, r10, fp, pc} ; восстановление pc (счетчика команд/program counter) из стека значит, что выполнение программы вернется к адресу, который был сохранен ранее, туда, откуда была вызвана функция. |

# **Приложение 3: Разбивка ассемблерного кода функции *double PiCalculation(long long n)* с комментариями с уровнем компиляции *-O1***

|  |
| --- |
| PiCalculation(long long):          push    {r3, r4, r5, r6, r7, r8, r9, lr} ; cохранение регистров r3–r9 и lr (ссылка на возвращение из функции) в стек          vpush.64        {d8, d9, d10} ; cохранение векторных регистров d8–d10 (для чисел с плавающей точкой)    mov     r7, r0 ; r7 = n (количество итераций цикла)          mov     r6, r1 ; r6 = старшая часть n, если n > 32 бит (разделенный регистр)  **; srand(time(NULL));**          movs    r0, #0 ; r0 = 0, установка r0 в 0 перед вызовом функции time          bl      time ; вызов time          bl      srand ; вызов srand для инициализации генератора случайных чисел (аргумент передан через r0, который сейчас содержит время)          cmp     r7, #1 ; cравниваем n с 1 (вычитание без сохранения результата) + установка флага состояния          sbcs    r3, r6, #0 ; вычитаем из r6 значение 0 с учётом флага заимствования и сохраняем результат в r3. Если флаг C установлен то вычиатния с заимствованием не происходит (r7=n > 1), если флаг C не установлен (r7 = n < 1) то r6 = r6 - 1  blt     .L6 ; если n < 1 переходим к .L6 (пропускаем основную часть цикла)  **; long long i = 0**          movs    r4, #0 ; r4 = 0, счётчик итераций цикла i (младшая часть)          mov     r9, r4 ; r9 = 0, старшая часть счётчика итераций    **; long long m = 0;**  mov     r5, r4 ; r5 = 0, младшая часть результата (количество точек внутри круга m)          mov     r8, r4 ; r8 = 0, старшая часть результата    vldr.64 d10, .L11 ; загружаем константу 2.0 в d10          vmov.f64        d9, #1.0e+0 ; константу 1.0 в d9    b       .L5 ; переход к наччалу основного цикла  .L10: **; m++**          adds    r5, r5, #1 ; увеличение младшей части m          adc     r8, r8, #0 ; увеличение старшей если младшая переполнена  .L3: ; метка для увеличения общего счётчика итераций i  **; i++**          adds    r3, r4, #1 ; увеличиваем младшую часть i r4 (общий счётчик)          adc     ip, r9, #0 ; увеличиваем старшую часть i r9 (если r4 переполнен)          mov     r4, r3 ; r4 = r3          mov     r9, ip ; r9 = ip          cmp     r6, ip ; сравнение старшей части n и ip (счётчика i)          it      eq ; если равны, выполняем далее          cmpeq   r7, r3 ; сравнение младшей части n и r3 (счётчика i)          beq     .L2 ; если обе части n совпадают с i то переходим в метку .L2 где завергаем цикл  .L5:  **; double x = static\_cast<double>(rand()) / RAND\_MAX \* 2 - 1.0;**          bl      rand ; вызов функции rand          vmov    s15, r0 @ int ; результат rand - случайное число в регистр s15          vcvt.f64.s32    d16, s15 ; преобразуем это число в 64-битное с плавающей точкой (в d16)          vdiv.f64        d8, d16, d10 ; d8 = rand() / RAND\_MAX          vadd.f64        d8, d8, d8 ; d8 \*= 2          vsub.f64        d8, d8, d9 ; d8 -= 1.0  **; double y = static\_cast<double>(rand()) / RAND\_MAX \* 2 - 1.0;**          bl      rand          vmov    s15, r0 @ int          vcvt.f64.s32    d17, s15          vdiv.f64        d16, d17, d10          vadd.f64        d16, d16, d16          vsub.f64        d16, d16, d9 ; d16 -= 1.0 (значение переменной y хранится здесь)          vmul.f64        d16, d16, d16 ; y^2          vmla.f64        d16, d8, d8 ; x^2: x^2 + y^2          vcmpe.f64       d16, d9 ; сравниваем x^2 + y^2 с 1.0          vmrs    APSR\_nzcv, FPSCR          bhi     .L3 ; Если x^2 + y^2 > 1, переходим к .L3 (следующая итерация цикла)          b       .L10 ; иначе m++  **; m = 0**  .L6:          movs    r5, #0 ; r5 = 0          mov     r8, r5 ; r8 = 0;  **; return 4 \* m / n;**  .L2:          adds    r5, r5, r5 ; m \*= 2          adc     r1, r8, r8 ; умножаем старшую часть m на 2          adds    r0, r5, r5 ; m \*= 4          mov     r2, r7 ; копируем n в r2 (аргумент для деления)          mov     r3, r6 ; копируем старшую часть n в r3          adcs    r1, r1, r1 ; r1 += r1 = 4r1          bl      \_\_aeabi\_ldivmod ; вызов \_\_aeabi\_ldivmod для деления чисел          bl      \_\_aeabi\_l2d ; конвертация в 64-битный double (fpu число)          vmov    d0, r0, r1 **; в d0 результат всей функции**  **; эпилог функции**          vldm    sp!, {d8-d10} ; восстанавливаем векторные регистры d8–d10 из стека          pop     {r3, r4, r5, r6, r7, r8, r9, pc} ; восстанавливаем регистры и возвращаемся из функции  .L11:          .word   -4194304          .word   1105199103 |

# **Приложение 4: Разбивка ассемблерного кода функции *double PiCalculation(long long n)* с комментариями с уровнем компиляции *-O2***

|  |
| --- |
| PiCalculation(long long):  **; пролог функции**  push {r3, r4, r5, r6, r7, r8, r9, lr} ; сохраняем регистры r3-r9 и lr (ссылка на возвращение из функции) на стек  mov r8, r0 ; сохраняем аргумент (n) в r8  movs r0, #0 ; обнуляем r0 для вызова srand  vpush.64 {d8, d9, d10} ; сохраняем регистры с плавающей запятой d8-d10  mov r9, r1  bl time ; вызываем функцию time  bl srand ; вызов srand для инициализации генератора случайных чисел (аргумент передан через r0, который сейчас содержит время)  cmp r8, #1 ; сравниваем n с 1  sbcs r3, r9, #0 ; вычисляем заём и сохраняем в r3 (если n < 1)  blt .L6 ; если n < 1, переходим к .L6  movs r4, #0 ; обнуляем r4 (младшая часть i)  vldr.64 d9, .L11 ; загружаем RAND\_MAX в d9  mov r6, r4 ; обнуляем r6 (старшая часть i)  mov r5, r4 ; обнуляем r5 (младшая часть m)  mov r7, r4 ; обнуляем r7 (старшая часть m)  vmov.f64 d8, #1.0e+0 ; загружаем 1.0 в регистр d8  .L5:  bl rand ; генерация случайного числа  vmov s15, r0 @ int ; перемещаем результат rand в регистр s15  vcvt.f64.s32 d16, s15 ; преобразуем s15 в число с плавающей точкой (суффикс .f64) в d16  vdiv.f64 d10, d16, d9 ; нормализуем случайное число: d10 = static\_cast<double>(rand()) / RAND\_MAX  bl rand ; ещё одно случайное число  vmov s15, r0 @ int ; сохраняем результата rand в s15  vcvt.f64.s32 d17, s15 ; преобразование s15 в float (теперь в s17)  vdiv.f64 d16, d17, d9 ; нормализуем s17 (static\_cast<double>(rand()) / RAND\_MAX  )  vadd.f64 d10, d10, d10 ; x \*= 2  vsub.f64 d10, d10, d8 ; x -= 1.0  vadd.f64 d16, d16, d16 ; y \*= 2  vsub.f64 d16, d16, d8 ; y -= 1.0  vmul.f64 d16, d16, d16 ; y \*= y  vmla.f64 d16, d10, d10 ; y^2 += x^2  vcmpe.f64 d16, d8 ; сравниваем x^2 + y^2 с 1.0  vmrs APSR\_nzcv, FPSCR ; сохраняем флаги сравнения  bhi .L3 ; если x^2 + y^2 > 1 => .L3  adds r5, r5, #1 ; m в r5 ++  adc r7, r7, #0 ; учитываем перенос для старшей части счётчика r7  .L3:  adds r4, r4, #1 ; увеличиваем общий счётчик точек i (r4)  adc r6, r6, #0 ; учитываем перенос для старшей части i  cmp r9, r6 ; сравниваем старшие части i с n  it eq ; следующая команда выполнится только если i и n равны  cmpeq r8, r4 ; сравниваем младшую часть i с n  bne .L5 ; если счётчики не равны, повторяем цикл  adds r5, r5, r5 ; m \*= 2 (младшая часть)  mov r2, r4 ; сохраняем младшую часть i в r2  adcs r7, r7, r7 ; m \*= 2 (старшая часть)  adds r0, r5, r5 ; удваиваем младшую часть m (m \*= 2 в r0)  adc r1, r7, r7 ; m \*= 2 (старшая часть)  mov r3, r6 ; сохраняем старшую часть i в r3  bl \_\_aeabi\_ldivmod ; вызов \_\_aeabi\_ldivmod для деления чисел  bl \_\_aeabi\_l2d ; преобразование результата в 64-битный double  vmov d0, r0, r1 ; результат в регистр d0  **; эпилог функции**  vldm sp!, {d8-d10} ; восстанавливаем регистры с плавающей точкой d8-d10  pop {r3, r4, r5, r6, r7, r8, r9, pc} ; восстанавливаем регистры и возвращаемся из функции  **; эпилог в отдельной метке если n < 1**  .L6:  vldm sp!, {d8-d10} ; восстанавливаем регистры d8-d10  vmov.i64 d0, #0 @ float ; результат pi = 0, если n < 1  pop {r3, r4, r5, r6, r7, r8, r9, pc} ; восстанавливаем регистры и возвращаемся из функции  .L11: ; RAND\_MAX  .word -4194304  .word 1105199103 |