­­­­**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 ЛАБОРАТОРНОЙ РАБОТЫ**

«ВВЕДЕНИЕ В АРХИТЕКТУРУ ARM»

студента 2 курса, группы 22201

Рабецкого Валерия Дмитриевича

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Матвеев А.С.

Новосибирск 2023

**Цель**

Изучить аспекты работы с языком ассемблера; Ознакомиться с архитектурой ARM; Провести детальный анализ ассемблерного кода программы первой лабораторной работы.

**Задание**

Изучить программную архитектуру ARM(Advanced RISC Machine). Ознакомиться с набором регистров процессора, основными отличиями ассемблерного кода архитектуры x86/x86-64 и ARM. Изучить способы адресации памяти, работу со стеком на ARM. Для программы на языке С++ сгенерировать ассемблерный листинг, после чего провести анализ,сопоставляя команды языка С++ с машинными командами. После детального анализа процессорных инструкций продемонстрировать использование ключевых особенностей архитектур ARM на конкретных участках кода.

**Листинг программы**

#include <iostream>

#include <cmath>

double PI(long long n)

{

    double pi;

    pi = 0;

    double temp = 0;

    for (int i = 0; i < n; i++)

    {

        temp = ((pow(-1, i)) / (2 \* i + 1));

        pi += temp;

    }

    pi = 4 \* pi;

    return pi;

}

int main()

{

    long long n = 600000000;

    double pi =  0;

    pi = PI(n);

    printf("pi = %lf", pi);

    return 0;

}

**Ассемблерный листинг для O0**

**PI(long long):**

**push {r4, r5, r7, lr} // Поместите регистры r4, r5, r7, lr в стек**

**r4, r5,r7 – регистры общего назначения**

**lr – содержит адрес возврата**

**sub sp, sp, #32 // Выделите 32 байта свободного места в стеке**

**add r7, sp, #0 // загружаем текущее значение стекового указателя в r7**

**strd r0, [r7] // Сохранить значение из регистра r0 в память по адресу, указанному в регистре r7**

**mov r2, #0 // Установить значение регистра r2 в 0**

**mov r3, #0 // Установить значение регистра r3 в 0**

**strd r2, [r7, #24] // Сохранить r2 (0) в память по адресу, вычисленному как r7 + 24**

**mov r2, #0 // Установить значение регистра r2 в 0**

**mov r3, #0 // Установить значение регистра r3 в 0**

**strd r2, [r7, #8] // Сохранить r2 (0) в память по адресу, вычисленному как r7 + 8**

**movs r3, #0 // Установить значение регистра r3 в 0 и установить флаги состояния**

**str r3, [r7, #20] // Сохранить r3 (0) в память по адресу, вычисленному как r7 + 20**

**b .L2 // переход к метке .L2**

**.L3:**

**ldr r3, [r7, #20] // Загрузить данные из памяти (r7 + 20) в регистр r3**

**vmov s15, r3 // Перемещение данных из регистра r3 в SIMD-регистр s15 (регистр с плавающей точкой)**

**vcvt.f64.s32 d16, s15 // Конвертация 32-битного целого числа из s15 в 64-битное число с плавающей точкой в d16**

**vmov.f64 d1, d16 // Перемещение значения из с плавающей точкой d16 в d1**

**vmov.f64 d0, #-1.0e+0 // Перемещение значения -1.0 в регистр d0**

**bl pow // вызов функции pow с аргументами d1 и d0**

**vmov.f64 d18, d0 // Перемещение результата из d0 в d18**

**ldr r3, [r7, #20] // Загрузка 32-битного значения из памяти по адресу [r7 + 20] в регистр r3**

**lsls r3, r3, #1 // Логический сдвиг влево значения в r3 на 1 бит (умножение на 2)**

**adds r3, r3, #1 // Увеличение значения в r3 на 1**

**vmov s15, r3 // Перемещение значения из целочисленного r3 в регистр с плавающей точкой s15**

**vcvt.f64.s32 d17, s15 // Конвертация 32-битного числа из s15 в 64-битное число с плавающей точкой в d16**

**vdiv.f64 d16, d18, d17 // деление: d16 = d18 / d17**

**vstr.64 d16, [r7, #8] // сохранить результат по адресу r7 + 8 в памяти**

**vldr.64 d17, [r7, #24] // загрузить значение из памяти по адресу r7 + 24 в d17**

**vldr.64 d16, [r7, #8] // загрузить значение из памяти по адресу r7 + 8 в d16**

**vadd.f64 d16, d17, d16 // d16 = d17 + d16**

**vstr.64 d16, [r7, #24] // сохранить результат по адресу r7 + 24 в памяти**

**ldr r3, [r7, #20] // загрузить значение из памяти по адресу r7 + 20 в r3**

**adds r3, r3, #1 // прибавить 1 к r3**

**str r3, [r7, #20] // Сохранение результата из r3 по адресу [r7 + 20] в памяти**

**.L2:**

**ldr r3, [r7, #20] // Загрузить данные из памяти (r7 + 20) в регистр r3**

**asrs r2, r3, #31 // Арифметический сдвиг r3 вправо на 31 бит и сохранение результата в r2**

**получаем значение знака числа r3**

**mov r4, r3 // Копировать значение из r3 в r4**

**mov r5, r2 // Копировать значение из r2 в r5**

**ldrd r2, [r7] // Загрузить 64-битное значение из памяти по адресу, указанному в r7, в r2 и, возможно, в r3**

**cmp r4, r2 // сравнить r4 и r2, устанавливает флаги состояния**

**sbcs r3, r5, r3 // вычитание r3 из r5 с учетом флага переноса (r5 - r3 – флаг)**

**blt .L3 // Переход к метке .L3, если r4 меньше r2**

**vldr.64 d16, [r7, #24] // загрузить значение из памяти по адресу r7 + 24 в d16**

**vmov.f64 d17, #4.0e+0 // Загрузка значения 4.0 в регистр d17**

**vmul.f64 d16, d16, d17 // d16 = d17 \* d16**

**vstr.64 d16, [r7, #24] // сохранить результат в r7 + 24**

**ldrd r2, [r7, #24] // загрузить значение из памяти по адресу r7 + 24 в dr2**

**vmov d16, r2, r3 // Перемещение целочисленные значения r2 и r3 в регистр с плавающей точкой d16**

**vmov.f64 d0, d16 // Перемещение значения из d16 в d0**

**adds r7, r7, #32 // Увеличили указатель стека, чтобы освободить пространство стека**

**mov sp, r7 // Копирование значения из регистра r7 в регистр sp**

**pop {r4, r5, r7, pc} // Восстановление значений из стека в регистры r4, r5, r7 и pc, возврат из подпрограммы**

**или переход по адресу, хранящемуся в pc**

**pc содержит адрес следующей инструкции для выполнения**

**.LC0:**

**.ascii "pi = %lf\000"**

**main:**

**push {r7, lr} //** **Сохранение значений регистров r7 и lr в стеке**

**sub sp, sp, #16 // Выделите 16 байта свободного места в стеке**

**add r7, sp, #0 // загружаем текущее значение стекового указателя в r7**

**adr r3, .L7 //** **Загрузка адреса метки .L7 в регистр r3**

**ldrd r2, [r3] // Загрузка 64-битного значения из памяти по адресу, который содержится в r3, в r2**

**strd r2, [r7, #8] //** **Сохранение 64-битного значения из r2 в память по адресу r7 + 8**

**mov r2, #0 // Установить значение регистра r2 в 0**

**mov r3, #0 // Установить значение регистра r3 в 0**

**strd r2, [r7] // Сохранение 64-битного значения из r2 в память по адресу r7**

**ldrd r0, [r7, #8] // Загрузка 64-битного значения из памяти по адресу, который содержится в r7 + 8, в r0**

**bl PI(long long) //**

**.64 d0, [r7] // сохранить результат в r7**

**ldrd r2, [r7] // Загрузка 64-битного значения из памяти по адресу, который содержится в r7, в r2**

**movw r0, #:lower16:.LC0 //** **Загрузка младших 16 бит адреса метки .LC0 в регистр r0**

**movt r0, #:upper16:.LC0 //** **Загрузка старших 16 бит адреса метки .LC0 в регистр r0**

**bl printf //**

**movs r3, #0 // Загрузка нулевого значения в регистр r3 с обновлением флагов состояния**

**mov r0, r3 //** **Копирование значения из регистра r3 в регистр r0**

**adds r7, r7, #16 // Увеличили указатель стека, чтобы освободить пространство стека**

**mov sp, r7 // Копирование значения из регистра r7 в регистр sp**

**pop {r7, pc} // Восстановление значений из стека в регистры r7 и pc, возврат из подпрограммы**

**.L7:**

**.word 600000000**

**.word 0**

**Ассемблерный листинг для архитектуры ARM с оптимизацией O1**

**PI(long long):**

**push {r4, r5, r6, lr} // Сохранение значений регистров r4, r5, r6 и lr в стеке**

**vpush.64 {d8, d9} // Сохранение значений двойных регистров d8 и d9 в стеке, чтобы позднее их можно было восстановить**

**cmp r0, #1 // Сравнение значения r0 с 1**

**sbcs r1, r1, #0 // вычитание из r1 0 с учетом флага переноса (r1 – 0 – флаг)**

**blt .L4 // Переход на .L4, если r0 меньше 1**

**mov r6, r0 // Копирование значения r0 в r6**

**movs r5, #1 // Загрузка значения 1 в r5**

**movs r4, #0 // Загрузка значения 0 в r4**

**vmov.i64 d8, #0 // Загрузка 64-битного значения 0 в d8**

**vmov.f64 d9, #-1.0e+0 // Загрузка значения -1.0e+0 в d9**

**.L3:**

**vmov s15, r4 @ int // Перемещение значения r4 в одинарный регистр с плавающей запятой s15**

**vcvt.f64.s32 d1, s15 // Конвертация значения из одинарного регистра в двойной**

**vmov.f64 d0, d9 // Загрузка значения из d9 в d0**

**bl pow // Вызов функции pow с аргументами d0 и d1**

**vmov s15, r5 @ int // Перемещение значения r5 в s15**

**vcvt.f64.s32 d17, s15 // Конвертация значения из одинарного регистра в двойной**

**vdiv.f64 d16, d0, d17 // Выполнение операции деления с плавающей запятой d16 = d0 / d17**

**vadd.f64 d8, d8, d16 // Выполнение операции сложения с плавающей запятой d8 = d8 + d16**

**adds r4, r4, #1 // Увеличение значения r4 на 1**

**adds r5, r5, #2 // Увеличение значения r5 на 2**

**cmp r4, r6 // Сравнение r4 и r6**

**bne .L3 // Переход к .L3, если r4 не равно r6**

**.L2:**

**vmov.f64 d0, #4.0e+0 // Загрузка значения 4.0e+0 в d0**

**vmul.f64 d0, d8, d0 // Выполнение операции умножения с плавающей запятой**

**vldm sp!, {d8-d9} // Восстановление значений двойных регистров d8 и d9 из стека**

**загружает значения из памяти sp в d8 и d9. После sp увеличивается на размер загруженных данных**

**! указывает, что sp меняется после выполнения операции**

**pop {r4, r5, r6, pc} // Восстановление значений регистров r4, r5, r6 и pc из стека**

**.L4:**

**vmov.i64 d8, #0 // Загрузка 64-битного значения 0 в d8**

**b .L2 // Переход к .L2**

**.LC0:**

**.ascii "pi = %lf\000" // Строка для форматированного вывода**

**main:**

**push {r3, lr} // Сохранение значений регистров r3 и lr в стеке**

**adr r1, .L9 // Загрузка адреса .L9 в r1**

**ldrd r0, [r1] // Загрузка значений, расположенных по адресу в r1, в r0**

**bl PI(long long) // Вызов функции PI**

**vmov r2, r3, d0 // Копирование значений из d0 в r2 и r3**

**movw r0, #:lower16:.LC0 // Загрузка младших 16 бит адреса .LC0 в r0**

**movt r0, #:upper16:.LC0 // Загрузка старших 16 бит адреса .LC0 в r0**

**bl printf // Вызов функции printf для вывода значения**

**movs r0, #0 // Загрузка значения 0 в r0 (код возврата)**

**pop {r3, pc} // Восстановление значений регистров r3 и pc из стека**

**.L9:**

**.word 600000000**

**.word 0**

**Ассемблерный листинг для архитектуры ARM с оптимизацией O2, O3**

**PI(long long):**

**cmp r0, #1 // Сравнение r0 с 1**

**sbcs r1, r1, #0 // вычитание из r1 0 с учетом флага переноса (r1 – 0 - флаг)**

**blt .L4 // Переход к .L4, если r0 < 1**

**push {r4, r5, r6, lr} // Помещение регистров r4, r5, r6, lr в стеке**

**mov r6, r0 // Помещение значения r0 в r6**

**movs r5, #1 // Установка r5 в 1**

**vpush.64 {d8} // Сохранение значения двойной точности d8 в стеке**

**vmov.i64 d8, #0 // Инициализация d8 как 0.0**

**movs r4, #0 // Установка r4 в 0**

**.L3:**

**vmov s15, r4 // int // Помещение значения r4 в s15**

**vmov.f64 d0, #-1.0e+0 // Установка двойного регистра d0 в -1.0**

**adds r4, r4, #1 // Увеличение r4 на 1**

**vcvt.f64.s32 d1, s15 // Преобразование значения s15 в двойной регистр d1**

**bl pow // Вызов функции pow**

**vmov s15, r5 // int // Помещение целочисленного значения r5 в s15 (одинарная точность)**

**cmp r6, r4 // Сравнение r6 и r4**

**add r5, r5, #2 // Увеличение r5 на 2**

**vcvt.f64.s32 d16, s15 // Преобразование целочисленного значения s15 в двойной d16**

**vdiv.f64 d17, d0, d16 // Деление d0 на d16**

**vadd.f64 d8, d8, d17 // Сложение d8 и d17**

**bne .L3 // Переход к .L3, если r6 не равно r4**

**vmov.f64 d0, #4.0e+0 // Установка двойного регистра d0 в 4.0**

**vmul.f64 d0, d8, d0 // Умножение d8 на d0**

**vldm sp!, {d8} // Восстановление d8 из стека**

**загружает значение из стека в регистр d8 и одновременно увеличивает указатель стека**

**pop {r4, r5, r6, pc} // Восстановление регистров r4, r5, r6, и переход на адрес, сохраненный в pc**

**.L4:**

**vmov.i64 d0, #0 // Инициализация d0 как 0.0**

**bx lr // Возврат из функции**

**.LC0:**

**.ascii "pi = %lf\000" // Формат строки вывода**

**main:**

**adr r1, .L13 // Получение адреса .L13 в r1**

**ldrd r0, [r1] // Загрузка значения, хранящегося по адресу r1, в r0**

**push {r3, lr} // Сохранение регистров r3 и lr в стеке**

**bl PI(long long) // Вызов функции PI**

**movw r0, #:lower16:.LC0 // Загрузка младших 16 бит адреса .LC0 в r0**

**movt r0, #:upper16:.LC0 // Загрузка старших 16 бит адреса .LC0 в r0**

**vmov r2, r3, d0 // Перемещение значения d0 в r2**

**bl printf // Вызов printf**

**movs r0, #0 // Установка r0 в 0**

**pop {r3, pc} // Восстановление регистров r3 и pc, завершение работы программы**

**.L13:**

**.word 600000000**

**.word 0**

**Ассемблерный листинг для архитектуры ARM с оптимизацией Ofast**

**PI(long long):**

**cmp r0, #1 // Сравнение r0 с 1**

**sbcs r3, r1, #0 // вычитание из r1 0 с учетом флага переноса (r1 – 0 - флаг)**

**blt .L6 // Переход к .L6, если меньше**

**push {r4, r5, r6, lr} // Помещение регистров {r4, r5, r6, lr} в стек**

**subs r3, r0, #1 // Вычитание 1 из r0 и сохранение в r3**

**mov r6, r0 // Перемещение значения из r0 в r6**

**orrs r3, r3, r1 // Побитовое ИЛИ r3 с r1 и сохранение в r3**

**vpush.64 {d8} // Помещение двойного регистра {d8} в стек**

**beq .L13 // Переход к .L13, если равно**

**movs r5, #3 // Загрузка значения 3 в r5**

**movs r4, #1 // Загрузка значения 1 в r4**

**vmov.f64 d8, #1.0e+0 // Перемещение значения 1.0 в двойной регистр d8**

**.L4:**

**vmov s15, r4 @ int // Перемещение значения из r4 в s15**

**vmov.f64 d0, #-1.0e+0 // Перемещение -1.0 в двойной регистр d0**

**adds r4, r4, #1 // Добавление 1 к r4**

**vcvt.f64.s32 d1, s15 // Преобразование целочисленного значения из s15 в двойной регистр d1**

**bl pow // Вызов функции pow (предположительно, вычисление степени)**

**vmov s15, r5 @ int // Перемещение значения из r5 в скалярный регистр s15**

**cmp r4, r6 // Сравнение r4 и r6**

**add r5, r5, #2 // Добавление 2 к r5**

**vcvt.f64.s32 d16, s15 // Преобразование целочисленного значения из s15 в двойной регистр d16**

**vdiv.f64 d17, d0, d16 // Деление d0 на d16 и сохранение результата в d17**

**vadd.f64 d8, d8, d17 // Добавление d17 к d8 и сохранение результата в d8**

**bne .L4 // Переход к .L4, если не равно**

**.L5:**

**vmov.f64 d0, #4.0e+0 // Перемещение значения 4.0 в двойной регистр d0**

**vmul.f64 d0, d8, d0 // Умножение d8 на d0 и сохранение результата в d0**

**vldm sp!, {d8} // Восстановление d8 из стека**

**загружает значение из стека в регистр d8 и одновременно увеличивает указатель стека**

**pop {r4, r5, r6, pc} // Восстановление значений регистров из стека и переход на адрес, сохраненный в pc**

**.L6:**

**vmov.i64 d0, #0 // Перемещение значения 0 в двойной регистр d0**

**bx lr // Возврат из функции**

**.L13:**

**vmov.f64 d8, #1.0e+0 // Перемещение значения 1.0 в двойной регистр d8**

**b .L5 // Переход к .L5**

**.LC0:**

**.ascii "pi = %lf\000" // Определение строки "pi = %lf" с завершающим нулем**

**main:**

**adr r1, .L16 // Загрузка адреса .L16 в r1**

**ldrd r0, [r1] // Загрузка двойного слова из памяти по адресу, на который указывает r1, в r0**

**push {r3, lr} // Помещение регистров {r3, lr} в стек**

**bl PI(long long) // Вызов функции PI(long long)**

**movw r0, #:lower16:.LC0 // Загрузка младших 16 бит адреса .LC0 в r0**

**movt r0, #:upper16:.LC0 // Загрузка старших 16 бит адреса .LC0 в r0**

**vmov r2, r3, d0 // Перемещение значения из d0 в r2**

**bl printf // Вызов функции printf**

**movs r0, #0 // Загрузка значения 0 в r0**

**pop {r3, pc} // Восстановление значений регистров из стека и возврат**

**.L16:**

**.word 600000000**

**.word 0**

**Выводы**

По результатам проведенного анализа было установлено, что в ассемблерном коде ARM обращение к памяти, работа с математическими функциями, работа с регистрами существенно отличаются от ассемблерного кода на x86/x86-64. Изучены основные отличия этих архитектур и изучены основы ARM.