Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**Лабораторная работа № 4**

Раздельная компиляция

по дисциплине «Низкоуровневое программирование»

Выполнил

студент гр. 3530901/90004

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Антунович П.Ю.

(подпись)

Руководитель

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Алексюк А.О.

(подпись)

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

Санкт-Петербург   
2021

**Постановка задачи**

1. Изучить методические материалы, опубликованные на сайте курса.

2. Установить пакет средств разработки “SiFive GNU Embedded Toolchain” для RISC-V.

3. На языке C разработать функцию, реализующую определенную вариантом задания функциональность. Поместить определение функции в отдельный исходный файл, оформить заголовочный файл. Разработать тестовую программу на языке C.

4. Собрать программу «по шагам». Проанализировать выход препроцессора и компилятора. Проанализировать состав и содержимое секций, таблицы символов, таблицы перемещений и отладочную информацию, содержащуюся в объектных файлах и исполняеммом файле.

5. Выделить разработанную функцию в статическую библиотеку. Разработать make-файлы для сборки библиотеки и использующей ее тестовой программы. Проанализировать ход сборки библиотеки и программы, созданные файлы зависимостей.

**Вариант задания:** 13) расчет значения многочлена по схеме Горнера с «длинным» результатом

**Ход работы**

Код программы состоит из 3-х файлов: gorner.c – файл с функциональностью программы, gorner.h – его заголовочный файл и main.c – тестовый файл для проверки работоспособности программы. Ниже приведено содержимое этих файлов.

#include <stdio.h>  
#include "gorner.h"  
  
int main() {  
 int length = 4;  
 int x0 = 2;  
 int array[] = {1, 3, 5, 7, 9};  
 int result = gorner(length, array, x0);  
  
 printf("f(x) = %d\n", result);  
 return 0;  
}

Листинг 1. Main.c

#include "gorner.h"  
  
int gorner(int length, int \*array, int x0)  
{  
 int a\_i = array[0];  
 int b\_i = a\_i;  
 for (int i = 1; i <= length; i++)  
 {  
 b\_i = array[i] + b\_i \* x0;  
 }  
 return b\_i;  
}

Листинг 2. Gorner.c

#include <stddef.h>  
  
#ifndef LAB4\_GORNER\_H  
#define LAB4\_GORNER\_H  
  
int gorner(int length, int \*array, int x0);  
  
#endif //LAB4\_GORNER\_H

Листинг 3. Gorner.h

Запустим программу, чтобы убедиться, что она работает верно.



Рис. 1 результат работы программы

Теперь произведем сборку программы по шагам.

1. **Препроцессирование.**

riscv64-unknown-elf-gcc -march=rv64iac -mabi=lp64 -O1 -v -E main.c -o main.i >log.pp 2>&1

riscv64-unknown-elf-gcc -march=rv64iac -mabi=lp64 -O1 -v -E gorner.c -o gorner.i >log.pp 2>&1

После выполнения соответствующих команд выполнилось препроцессирование, результат был записан в файлы gorner.i и main.i.

# 2 "main.c" 2  
# 1 "gorner.h" 1  
# 1 "c:\\users\\professional\\desktop\\riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86\_64-w64-mingw32\\lib\\gcc\\riscv64-unknown-elf\\10.2.0\\include\\stddef.h" 1 3 4  
# 2 "gorner.h" 2  
  
  
  
  
  
# 6 "gorner.h"  
int gorner(int length, int \*array, int x0);  
# 3 "main.c" 2  
  
int main() {  
 int length = 4;  
 int x0 = 2;  
 int array[] = {1, 3, 5, 7, 9};  
 int result = gorner(length, array, x0);  
  
 printf("f(x) = %d\n", result);  
 return 0;  
}

Листинг 4. Содержимое main.i (фрагмент)

# 1 "gorner.c"  
# 1 "<built-in>"  
# 1 "<command-line>"  
# 1 "gorner.c"  
# 1 "gorner.h" 1  
# 1 "c:\\users\\professional\\desktop\\riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86\_64-w64-mingw32\\lib\\gcc\\riscv64-unknown-elf\\10.2.0\\include\\stddef.h" 1 3 4  
# 143 "c:\\users\\professional\\desktop\\riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86\_64-w64-mingw32\\lib\\gcc\\riscv64-unknown-elf\\10.2.0\\include\\stddef.h" 3 4  
  
# 143 "c:\\users\\professional\\desktop\\riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86\_64-w64-mingw32\\lib\\gcc\\riscv64-unknown-elf\\10.2.0\\include\\stddef.h" 3 4  
typedef int ptrdiff\_t;  
# 209 "c:\\users\\professional\\desktop\\riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86\_64-w64-mingw32\\lib\\gcc\\riscv64-unknown-elf\\10.2.0\\include\\stddef.h" 3 4  
typedef unsigned int size\_t;  
# 321 "c:\\users\\professional\\desktop\\riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86\_64-w64-mingw32\\lib\\gcc\\riscv64-unknown-elf\\10.2.0\\include\\stddef.h" 3 4  
typedef int wchar\_t;  
# 415 "c:\\users\\professional\\desktop\\riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86\_64-w64-mingw32\\lib\\gcc\\riscv64-unknown-elf\\10.2.0\\include\\stddef.h" 3 4  
typedef struct {  
 long long \_\_max\_align\_ll \_\_attribute\_\_((\_\_aligned\_\_(\_\_alignof\_\_(long long))));  
 long double \_\_max\_align\_ld \_\_attribute\_\_((\_\_aligned\_\_(\_\_alignof\_\_(long double))));  
# 426 "c:\\users\\professional\\desktop\\riscv64-unknown-elf-toolchain-10.2.0-2020.12.8-x86\_64-w64-mingw32\\lib\\gcc\\riscv64-unknown-elf\\10.2.0\\include\\stddef.h" 3 4  
} max\_align\_t;  
# 2 "gorner.h" 2  
  
  
  
  
  
# 6 "gorner.h"  
int gorner(int length, int \*array, int x0);  
# 2 "gorner.c" 2  
  
int gorner(int length, int \*array, int x0)  
{  
 int a\_i = array[0];  
 int b\_i = a\_i;  
 for (int i = 1; i <= length; i++)  
 {  
 b\_i = array[i] + b\_i \* x0;  
 }  
 return b\_i;  
}

Листинг 5. Содержимое gorner.i

В файлах видно, что препроцессор включил содержимое заголовочных файлов. Из-за того, что файл main.c также включает в себя заголовочные файлы из стандартных библиотек, результат работы препроцессора для него гораздо больше, чем для gorner.c.

1. **Компиляция**

riscv64-unknown-elf-gcc -march=rv64iac -mabi=lp64 -O1 -v -S -fpreprocessed main.i -o main.s >log.cc 2>&1

riscv64-unknown-elf-gcc -march=rv32iac -mabi=lp64 -O1 -v -S -fpreprocessed gorner.i -o gorner.s >log.cc 2>&1

Следующими командами запустим компиляцию выхода препроцессора.

.file "main.c"  
 .option nopic  
 .attribute arch, "rv64i2p0\_a2p0\_c2p0"  
 .attribute unaligned\_access, 0  
 .attribute stack\_align, 16  
 .text  
 .section .rodata.str1.4,"aMS",@progbits,1  
 .align 2  
.LC1:  
 .string "f(x) = %d\n"  
 .text  
 .align 2  
 .globl main  
 .type main, @function  
main:  
 addi sp,sp,-48  
 sw ra,44(sp)  
 lui a5,%hi(.LANCHOR0)  
 addi a5,a5,%lo(.LANCHOR0)  
 lw a1,0(a5)  
 lw a2,4(a5)  
 lw a3,8(a5)  
 lw a4,12(a5)  
 lw a5,16(a5)  
 sw a1,12(sp)  
 sw a2,16(sp)  
 sw a3,20(sp)  
 sw a4,24(sp)  
 sw a5,28(sp)  
 li a2,2  
 addi a1,sp,12  
 li a0,4  
 call gorner  
 mv a1,a0  
 lui a0,%hi(.LC1)  
 addi a0,a0,%lo(.LC1)  
 call printf  
 li a0,0  
 lw ra,44(sp)  
 addi sp,sp,48  
 jr ra  
 .size main, .-main  
 .section .rodata  
 .align 2  
 .set .LANCHOR0,. + 0  
.LC0:  
 .word 1  
 .word 3  
 .word 5  
 .word 7  
 .word 9  
 .ident "GCC: (SiFive GCC-Metal 10.2.0-2020.12.8) 10.2.0"

Листинг 6. Вывод компилятора main.s

.file "gorner.c"  
 .option nopic  
 .attribute arch, "rv64i2p0\_a2p0\_c2p0"  
 .attribute unaligned\_access, 0  
 .attribute stack\_align, 16  
 .text  
 .globl \_\_muldi3  
 .align 1  
 .globl gorner  
 .type gorner, @function  
gorner:  
 mv a5,a1  
 lw a1,0(a1)  
 ble a0,zero,.L6  
 addi sp,sp,-32  
 sd ra,24(sp)  
 sd s0,16(sp)  
 sd s1,8(sp)  
 sd s2,0(sp)  
 mv s2,a2  
 addi s0,a5,4  
 addiw s1,a0,-1  
 slli a4,s1,32  
 srli s1,a4,30  
 addi a5,a5,8  
 add s1,s1,a5  
.L3:  
 mv a0,s2  
 call \_\_muldi3  
 lw a1,0(s0)  
 addw a1,a1,a0  
 addi s0,s0,4  
 bne s0,s1,.L3  
 mv a0,a1  
 ld ra,24(sp)  
 ld s0,16(sp)  
 ld s1,8(sp)  
 ld s2,0(sp)  
 addi sp,sp,32  
 jr ra  
.L6:  
 mv a0,a1  
 ret  
 .size gorner, .-gorner  
 .ident "GCC: (SiFive GCC-Metal 10.2.0-2020.12.8) 10.2.0"

Листинг 7. Вывод компилятора gorner.s

Из листинга 6 замечаем обращение компилятора к подпрограмме gorner а также определение исходных данных. В листинге 7 можно заметить цикл.

1. **Ассемблирование**

riscv64-unknown-elf-gcc -march=rv64iac -mabi=lp64 -v -c main.s -o main.o >log.as 2>&1

riscv64-unknown-elf-gcc -march=rv64iac -mabi=lp64 -v -c gorner.s -o gorner.o >log.as 2>&1

Проводим ассемблирование и проверяем содержимое таблиц символов объектных файлов gorner.o и main.o.

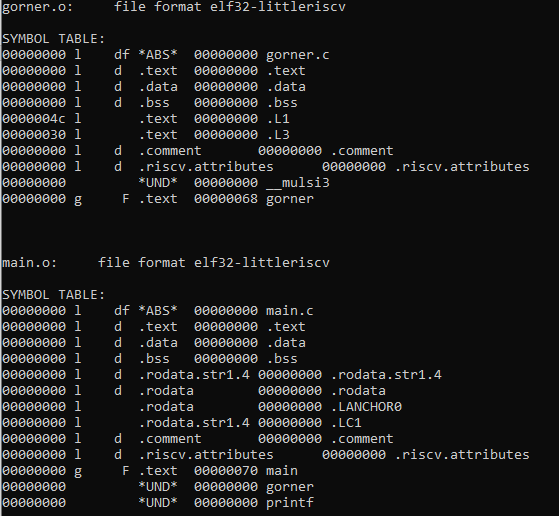


Рис. 2 вывод утилиты

В таблице символов main.o можем увидеть запись: символ “gorner” типа \*UND\*. Эта запись означает, что символ использовался в ассемблерном коде, но не был определен; ассемблер сделал вывод о том, что символ должен быть определен где-то еще, поэтому отразил это в таблице символов.

Изучим содержимое секции “.text” объектных файлов.

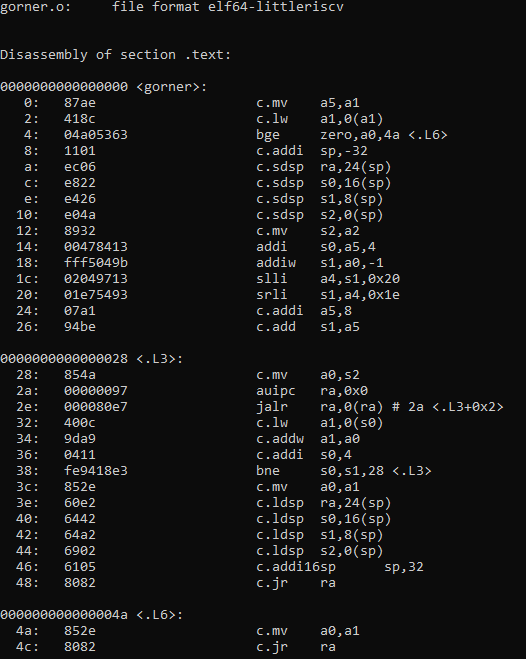


Рис. 3 результат дизассемблирования “gorner.o”

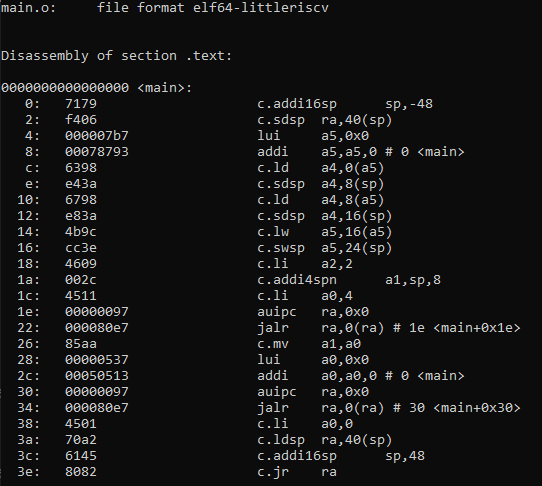


Рис. 4 результат дизассемблирования “main.o”

Из рис. 4 видно, что если сравнить вывод дизассемблера с выводом main.s, то можно увидеть, что вызов подпрограммы gorner осуществляется следующей парой инструкций:

1e: 00000097 auipc ra,0x0

22: 000080e7 jalr ra,0(ra) # 1e <main+0x1e>

Результатом работы этих инструкций является переход к адресу 1e и зацикливание. Это объясняется тем, что ассемблер не имел возможности определить целевой адрес перехода, поэтому не мог сформировать корректную инструкция передачи управления.

1. **Компоновка**

riscv64-unknown-elf-gcc -march=rv64iac -mabi=lp64 -v main.o gorner.o -o main >log.ld 2>&1

Произведем компоновку и просмотрим информацию обо всех незаконченных инструкциях в таблице перемещений.

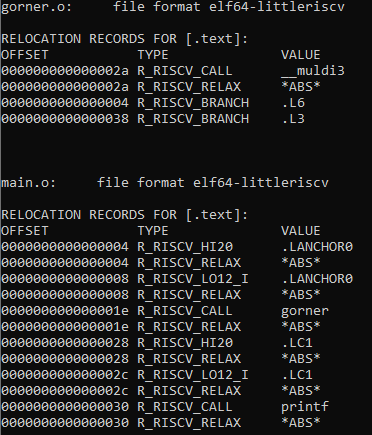


Рис. 5 таблица перемещений

В файле main.o видим 2 записи, относящиеся к тому самому адресу 1e. В первой записи указано, что по адресу 1e следует исправить пару инструкций так, чтобы результат соответствовал вызову подпрограммы.

Теперь изучим содержимое секции “.text” компоновщика.

0000000000010156 <main>:  
 10156: 7179 c.addi16sp sp,-48  
 10158: f406 c.sdsp ra,40(sp)  
 1015a: 67f5 c.lui a5,0x1d  
 1015c: b3078793 addi a5,a5,-1232 # 1cb30 <\_\_clzdi2+0x4e>  
 10160: 6398 c.ld a4,0(a5)  
 10162: e43a c.sdsp a4,8(sp)  
 10164: 6798 c.ld a4,8(a5)  
 10166: e83a c.sdsp a4,16(sp)  
 10168: 4b9c c.lw a5,16(a5)  
 1016a: cc3e c.swsp a5,24(sp)  
 1016c: 4609 c.li a2,2  
 1016e: 002c c.addi4spn a1,sp,8  
 10170: 4511 c.li a0,4  
 10172: 018000ef jal ra,1018a <gorner>  
 10176: 85aa c.mv a1,a0  
 10178: 6575 c.lui a0,0x1d  
 1017a: b2050513 addi a0,a0,-1248 # 1cb20 <\_\_clzdi2+0x3e>  
 1017e: 1c0000ef jal ra,1033e <printf>  
 10182: 4501 c.li a0,0  
 10184: 70a2 c.ldsp ra,40(sp)  
 10186: 6145 c.addi16sp sp,48  
 10188: 8082 c.jr ra  
  
000000000001018a <gorner>:  
 1018a: 87ae c.mv a5,a1  
 1018c: 418c c.lw a1,0(a1)  
 1018e: 04a05163 bge zero,a0,101d0 <gorner+0x46>  
 10192: 1101 c.addi sp,-32  
 10194: ec06 c.sdsp ra,24(sp)  
 10196: e822 c.sdsp s0,16(sp)  
 10198: e426 c.sdsp s1,8(sp)  
 1019a: e04a c.sdsp s2,0(sp)  
 1019c: 8932 c.mv s2,a2  
 1019e: 00478413 addi s0,a5,4  
 101a2: fff5049b addiw s1,a0,-1  
 101a6: 02049713 slli a4,s1,0x20  
 101aa: 01e75493 srli s1,a4,0x1e  
 101ae: 07a1 c.addi a5,8  
 101b0: 94be c.add s1,a5  
 101b2: 854a c.mv a0,s2  
 101b4: 020000ef jal ra,101d4 <\_\_muldi3>  
 101b8: 400c c.lw a1,0(s0)  
 101ba: 9da9 c.addw a1,a0  
 101bc: 0411 c.addi s0,4  
 101be: fe941ae3 bne s0,s1,101b2 <gorner+0x28>  
 101c2: 852e c.mv a0,a1  
 101c4: 60e2 c.ldsp ra,24(sp)  
 101c6: 6442 c.ldsp s0,16(sp)  
 101c8: 64a2 c.ldsp s1,8(sp)  
 101ca: 6902 c.ldsp s2,0(sp)  
 101cc: 6105 c.addi16sp sp,32  
 101ce: 8082 c.jr ra  
 101d0: 852e c.mv a0,a1  
 101d2: 8082 c.jr ra

Листинг 8. Фрагмент результирующего файла main.ds

Инструкции программы gorner начинаются с адреса 1018a, и пара инструкций auipc+jalr были откорректированы: результатом jalr будет переход на адрес 1018a.

**Создание статической библиотеки**

Объединим все объектные файлы в библиотеку:

riscv64-unknown-elf-ar –rsc libgorner.a gorner.o

Результирующим файлом является архив libgorner.a. Проверим его содержимое.



Рис. 6 содержимое архива

Используем стат. библиотеку для сборки программ:

riscv64-unknown-elf-gcc -march=rv64iac -mabi=lp64 -O1 --save-temps main.c libgorner.a -o main1

Изучим таблицы символов исполняемых файлов:

main1: file format elf64-littleriscv  
  
SYMBOL TABLE:

…

000000000001018a g F .text 000000000000004a gorner

…

Листинг 9. Содержимое таблицы символов

Видим, что в состав программы main1 вошло содержимое файла gorner.o.

Создадим make-файл, чтобы создавать библиотеку и собирать программу без выполнения команд.

CC=riscv64-unknown-elf-gcc  
AR=riscv64-unknown-elf-ar  
CFLAGS=-march=rv64iac -mabi=lp64  
  
all: lib  
lib: gorner.o  
 $(AR) -rsc libgorner.a gorner.o  
gorner.o: gorner.c  
 $(CC) $(CFLAGS) -c gorner.c -o gorner.o  
clean:  
 rm -f \*.o \*.a  
app:  
 $(CC) $(CFLAGS) main.c libgorner.a -o main

Листинг 10. Содержимое make-файла

**Вывод**

В ходе работы была создана программа на языке C, проведена пошаговая ее сборка. Разработанные функции были выделены в статическую библиотеку, а также были разработан make-файл для создания библиотеки и сборки программы из нее.