Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский политехнический университет Петра Великого»

Институт компьютерных наук и технологий

Кафедра компьютерных систем и программных технологий

**Отчёт по лабораторной работе №4**

**Дисциплина:** «Низкоуровневое программирование»

**Тема:** «Раздельная компиляция»

**Вариант №17**

Выполнил студент гр. 3530901/90004 \_\_\_\_\_\_\_\_\_\_ Сидоров А.А.

*(подпись)*

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

*(подпись)*

“\_\_\_”\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 г.

Санкт-Петербург

2021

**Цель работы:**

* 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.
* 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 исполнимом файле.
* 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 №12** – Формирование последовательности чисел в коде Грея заданной разрядности.

1. **Программа, реализующая функциональность задания**

**Листинг 1.1.** Заголовочный файл *gray.h*

#include <stdio.h>  
#include <stdlib.h>  
  
void gray(unsigned n);

**Листинг 1.2.** Основной файл *gray.c*

#include "gray.h"  
#include "printBin.h"  
  
int\* gray(unsigned n) {  
 const size\_t size = 1 << n;  
 int \*arr = calloc(size, sizeof(int));  
 for (int i = 0; i < size; ++i) {  
 arr[i] = i ^ i >>1;  
 }  
 return arr;  
}

**Листинг 1.3**. Тестовая программа *main.c*

#include "gray.h"  
  
int main( ) {  
 int n = 3;  
 const size\_t size = 1 << n;  
 int \*arr = gray(n);  
 for (int i = 0; i < size; ++i) {  
 int a = arr[i];  
 a <<= 8 - n;  
 for (int j = 0; j < n; j++) {  
 printf("%c", (a & 128) ? '1' : '0');  
 a <<= 1;  
 }  
 printf("\n");  
 }  
}

1. **Сборка программы «по шагам»**
   1. **Препроцессирование**

Препроцессирование выполнялось следующими командами:

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -E main.c -o main.i

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -E gray.c -o gray.i

Результат препроцессирования содержится в файле *gray.i* и *main.i.* По причине того, что исходные файлы содержат заголовочные файлы нескольких стандартных библиотек С, результат препроцессирования отличается от исходных файлов и имеет достаточно много добавочных строк, среди которых и исходные программы. Также можно заметить, что препроцессор включил содержимое файла *gray.h*.

**Листинг 2.1.1** gray.i. Выход препроцессора (фрагменты)

# 1 "gray.c"  
# 1 "<built-in>"  
# 1 "<command-line>"  
# 1 "gray.c"  
# 1 "gray.h" 1

...

void gray(unsigned n);  
# 2 "gray.c" 2  
  
void gray(unsigned n) {  
 const size\_t size = 1 << n;  
 int arr[size];  
 for (int i = 0; i < size; ++i) {  
 arr[i] = i ^ i >>1;  
 unsigned char a = arr[i];  
 a <<= 8 - n;  
 for (int j = 0; j < n; j++) {  
 printf("%c", (a & 128) ? '1' : '0');  
 a <<= 1;  
 }  
 printf("\n");  
 }  
}

**Листинг 2.1.2** main.i. Выход препроцессора (фрагменты)

# 1 "main.c"  
# 1 "<built-in>"  
# 1 "<command-line>"  
# 1 "main.c"  
# 1 "gray.h" 1

...

# 4 "gray.h"  
void gray(unsigned n);  
# 2 "main.c" 2  
  
int main( ) {  
 gray(3);  
 return 0;  
}

* 1. **. Компиляция**

Компиляция выхода препроцессора осуществляется командой:

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 –S main.i -o main.s

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 –S gray.i -o gray.s

Наибольший интерес представляет файл *main.s*, так как в нем можно заметить обращение к подпрограмме *gray* (значение регистра *ra*, содержащее адрес возврата из *main*, сохраняется на время вызова в стеке). Следует отметить, что символ *gray* используется в файле, но никак не определяется.

**Листинг 2.2.** Выход компилятора *main.s*

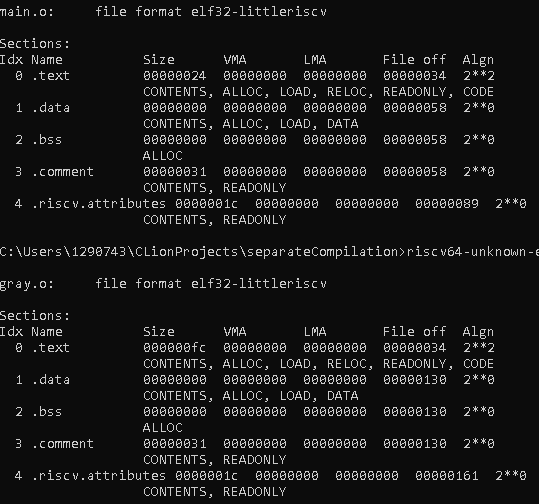
.file "main.c"  
 .option nopic  
 .attribute arch, "rv32i2p0"  
 .attribute unaligned\_access, 0  
 .attribute stack\_align, 16  
 .text  
 .align 2  
 .globl main  
 .type main, @function  
main:  
 addi sp,sp,-16  
 sw ra,12(sp)  
 li a0,3  
 call gray  
 li a0,0  
 lw ra,12(sp)  
 addi sp,sp,16  
 jr ra  
 .size main, .-main  
 .ident "GCC: (SiFive GCC-Metal 10.2.0-2020.12.8) 10.2.0"

**Листинг 2.2.** Выход компилятора gray*.s*

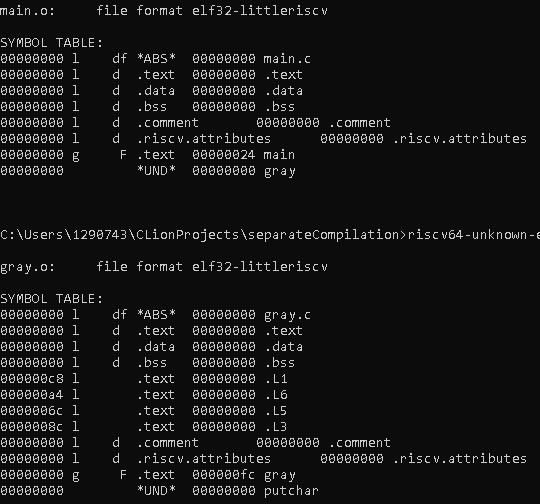
.file "gray.c"  
 .option nopic  
 .attribute arch, "rv32i2p0"  
 .attribute unaligned\_access, 0  
 .attribute stack\_align, 16  
 .text  
 .align 2  
 .globl gray  
 .type gray, @function  
gray:  
 addi sp,sp,-48  
 sw ra,44(sp)  
 sw s0,40(sp)  
 sw s1,36(sp)  
 sw s2,32(sp)  
 sw s3,28(sp)  
 sw s4,24(sp)  
 sw s5,20(sp)  
 sw s6,16(sp)  
 sw s7,12(sp)  
 sw s8,8(sp)  
 addi s0,sp,48  
 li s6,1  
 sll s6,s6,a0  
 li a5,4  
 sll a5,a5,a0  
 addi a5,a5,15  
 andi a5,a5,-16  
 sub sp,sp,a5  
 beq s6,zero,.L1  
 mv s4,sp  
 mv s5,a0  
 li s3,0  
 li s7,8  
 sub s7,s7,a0  
 mv s2,a0  
 j .L6  
.L3:  
 li a0,10  
 call putchar  
 addi s3,s3,1  
 addi s4,s4,4  
 beq s6,s3,.L1  
.L6:  
 srai s1,s3,1  
 xor s1,s1,s3  
 sw s1,0(s4)  
 andi s1,s1,255  
 sll s1,s1,s7  
 andi s1,s1,0xff  
 beq s5,zero,.L3  
 li s8,0  
.L5:  
 srli a0,s1,7  
 addi a0,a0,48  
 call putchar  
 slli s1,s1,1  
 andi s1,s1,0xff  
 addi s8,s8,1  
 bne s8,s2,.L5  
 j .L3  
.L1:  
 addi sp,s0,-48  
 lw ra,44(sp)  
 lw s0,40(sp)  
 lw s1,36(sp)  
 lw s2,32(sp)  
 lw s3,28(sp)  
 lw s4,24(sp)  
 lw s5,20(sp)  
 lw s6,16(sp)  
 lw s7,12(sp)  
 lw s8,8(sp)  
 addi sp,sp,48  
 jr ra  
 .size gray, .-gray  
 .ident "GCC: (SiFive GCC 8.3.0-2020.04.1) 8.3.0"

* 1. **Выход ассемблера – объектный файл**

**Листинг 2.3.1** Заголовки секций файлов *main.o gray.o*

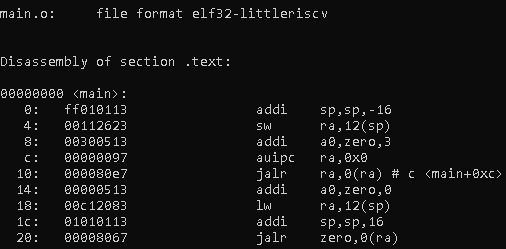
****

**Листинг 2.3.2** Таблица символов



В таблице символов *main.o* имеется запись: символ “gray” типа \*UND\*. Эта запись означает, что символ “gray” использовался в ассемблерном коде, из которого был получен данный объектный файл, но не был определен; ассемблер сделал вывод о том, что символ должен быть определен где-то еще, и отразил это в таблице символов.

**Листинг 2.3.2** Таблица перемещений файла *main.o*



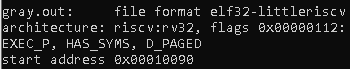
Результат дизассемблирования “zero.o” интереса не представляет, в отличие от результата дизассемблирования “main.o”: сравнивая его с “main.s”, легко понять, что псевдоинструкция вызова подпрограммы “zero”, транслировалась ассемблером в следующую пару инструкций:



* 1. **Результат компоновки**

Сформированный компоновщиком файл “a.out является «бинарным», и для изучения его содержимого будем пользоваться утилитой objdump.

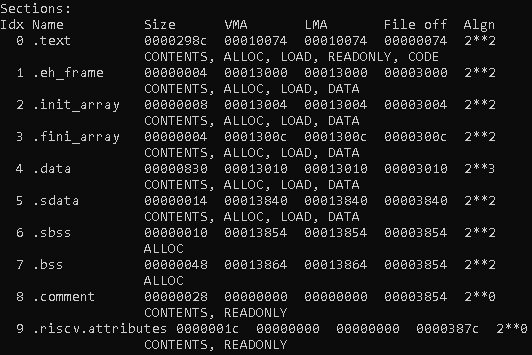
riscv64-unknown-elf-objdump -f gray.out



Можно видеть, что файл имеет формат ELF и действительно является исполняемым (флаг “EXEC\_P”), после загрузки его выполнение должно начаться с адреса 0x10090.

Далее изучим секции файла:

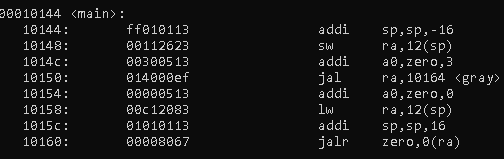
riscv64-unknown-elf-objdump -h a.out



Сразу можно заметить, что состав секций “gray.out” значительно расширен по сравнению с “main.o”. Следует также обратить внимание на то, что размеры секций “.text”, “.data”, “.bss” (и “.comment”) увеличились. Это связано с тем, что на вход компоновщика было подано несколько объектных файлов.

Изучим содержимое секции “.text”(фрагмент) :

riscv64-unknown-elf-objdump –j .text –d –M no-aliases a.out >a.ds



Можно видеть, что подпрограмма main имеет на одну инструкцию меньше: пара инструкций auipc+jalr заменена компоновщиком после оптимизации одной инструкцией jal.

1. **Создание статической библиотеки**

Выделим из программы *gray.c* функцию печати двоичных чисел *printBin* в отдельную программу и объединим эти программы в статическую библиотеку gray, тестовую программу main оставим без изменений.

**Листинг 1.1.** Заголовочный файл *gray.h*

#include <stdio.h>  
#include <stdlib.h>  
  
void gray(unsigned n);

**Листинг 1.2.** Основной файл *gray.c*

#include "gray.h"  
  
void gray(unsigned n) {  
 const size\_t size = 1 << n;  
 int arr[size];  
 printf("%d\n", arr[0]);  
 for (int i = 0; i < size; ++i) {  
 arr[i] = i ^ i >>1;  
 printBin(arr[i], n);  
 }  
}

**Листинг 1.3**. программа *printBin.c*

#include "printBin.h"  
  
void printBin(int a, int n) {  
 a <<= 8 - n;  
 for (int j = 0; j < n; j++) {  
 printf("%c", (a & 128) ? '1' : '0');  
 a <<= 1;  
 }  
 printf("\n");  
}

Для создания статической библиотеки получим объектные файлы всех используемых программ:

Для создания статической библиотеки получим объектные файлы всех используемых программ:

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -c gray.c -o gray.o

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -c printBin.c -o printBin.o

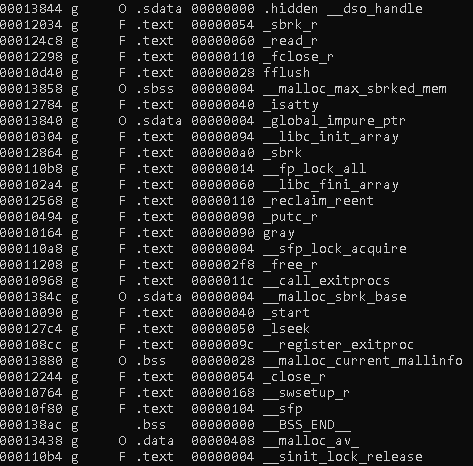
Объединим их в одну библиотеку gray:

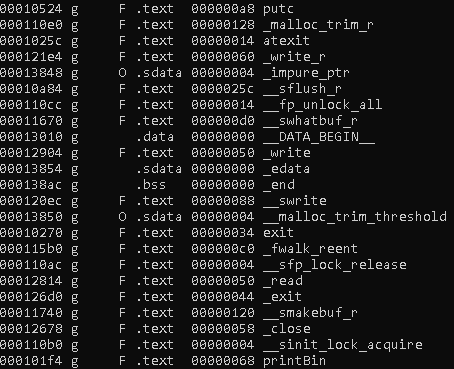
riscv64-unknown-elf-ar –rsc libgray.a printBin.o gray.o

Используем получившуюся библиотеку для сборки программ:

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 --save-temps main.c libgray.a -o main

**Листинг 3.4.** Таблица символов исполняемого файла

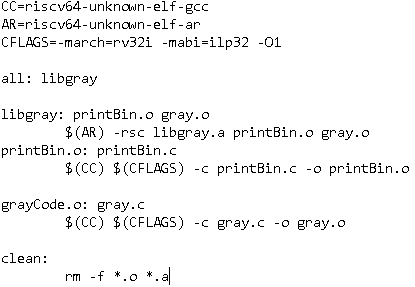


****

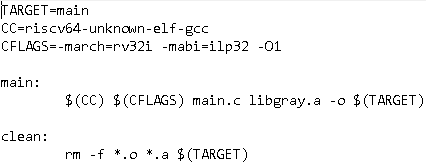
Легко заметить, что в состав программы main вошло содержимое объектных файлов *gray.o* и *printBin.o*.

Процесс выполнения команд выше можно заменить make-файлами, которые произведут создание библиотеки и сборку программы.

**Листинг 3.5**. make-файл для создания библиотеки



**Листинг 3.6**. make-файл для сборки



**Выводы**

В ходе выполнения лабораторной работы были получены навыки работы с препроцессором, компилятором, компоновщиком пакета GCC и драйвером компилятора riscv64-unknown-elf-gcc.

Также были изучены особенности каждого этапа пошаговой сборки набора программ. Разработанные функции были выделены в статическую библиотеку с помощью пакета средств разработки “SiFive GNU Embedded Toolchain”, а также с помощью make-файлов.