Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа информатики и вычислительной техники

**Лабораторная работа № 4**

**Дисциплина**: Низкоуровневое программирование

**Тема:** Раздельная компиляция

Выполнил студент гр. 3530901/10005 Шилоносов А. П.

Преподаватель Коренев Д. А.

«\_\_» \_\_\_\_\_\_\_\_\_\_\_ 2022 г.

Санкт-Петербург

2022

Оглавление

[**1.** **Формулировка задачи** 3](#_Toc121873093)

[**2.** **Вариант задания** 3](#_Toc121873094)

[**3.** **Ход решения** 3](#_Toc121873095)

[**4.** **Создание статической библиотеки и Makefile** 18](#_Toc121873096)

# **Формулировка задачи**

1) На языке C разработать функцию, которая находит минимальный элемент массива и из всех элементов массива вычитает этот минимальный элемент. Поместить определение функции в отдельный исходный файл, оформить заголовочный файл. Разработать тестовую программу на языке C.

2) 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 исполняемом файле.

3) Выделить разработанную функцию в статическую библиотеку. Разработать Makefile для сборки библиотеки и использующей ее тестовой программы. Проанализировать ход сборки библиотеки и программы, созданные файлы зависимостей.

# **Вариант задания**

Найти минимальный элемент массива. Из всех элементов массива вычесть минимальный.

# **Ход решения**

Файл *main.c*:

Изображение выглядит как текст

Автоматически созданное описание

Файл *submin.h*:

Изображение выглядит как текст

Автоматически созданное описание

Файл *submin.c*:

Изображение выглядит как текст

Автоматически созданное описание

Для сборки программы выполним следующую команду:

*riscv64-unknown-elf-gcc --save-temps -march=rv32i -mabi=ilp32 -O1 -v main.c submin.c >log 2>&1*

После выполнения команды, в файлах *main.i* и *submin.i* появится результат препроцессирования – первый шаг сборки программы.

Второй шаг сборки программы - ассемблирование: на основе файлов *main.i* и *submin.i* создадутся текстовые файлы *main.s* и *submin.s*, содержащие код на языке ассемблера.

Третий шаг сборки программы – компилирование: создадутся объектные файлы *main.o* и *submin.o*, сгенерированные ассемблером в результате обработки файлов *main.s* и *submin.s*

В итоге мы получим файл *a.out*: исполняемый файл, сгенерированный компоновщиком в результате обработки файлов *main.o*, *submin.o*, а также других объектных файлов и файлов библиотек, входящих в состав пакета средств разработки.

Препроцессирование:

Содержание файла *main.i*:

В программе используется директива *#include <stdio.h>,* поэтому в самом файла *main.i* в начале файла содержится порядка 1200 строк с инструкциями по линковке *stdio.h* к проекту, а затем следует код на C, который соответствует программе

Изображение выглядит как текст

Автоматически созданное описание

Файл *submin.i*:

Изображение выглядит как текст

Автоматически созданное описание

Появившиеся нестандартные директивы, начинающиеся с символа «#», используются для передачи информации об исходном тексте из препроцессора в компилятор. Так, файл *submin.i* в четвёртой строчке (*#* *1 "submin.c"*) информирует компилятор о том, что следующая строка является результатом обработки строки 1 исходного файла *submin.c*. В этой строке стояла команда *#include "submin.h"*, поэтому препроцессор произвел вставку содержимого этого заголовочного файла, то есть определение функции *subMin()*. Далее же начинается описание самого содержимого файла, что происходит после директивы *# 2 "submin.c" 2*. Исходный код тестирующей функции main() после работы препроцессора остался без изменений, как и исходный код функции *subMin()*.

Компиляция:

Компилятором сгенерирован код на языке низкого уровня – Assembly.

Файл *main.s*:

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Здесь с метки *.main* начинается выполнение разработанного алгоритма. Так же можно заметить, что на 40ой строчке идёт обращение к разработанной функции *subMin()*. Но *subMin* ещё нигде не определен, потому что не проведено ассемблирование.

Файл *submin.s*:

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

Здесь разработанная изначально функция *subMin()* представлена в метке с таким же названием - *.subMin*.

Ассемблирование

Объектные файлы не являются текстовым, поэтому для изучения их содержимого надо использовать утилиту objdump.

Файл *main.o*:

Отображение заголовков секций:

riscv64-unknown-elf-objdump.exe -h main.o

Изображение выглядит как текст

Автоматически созданное описание

По этим данным можно сделать выводы о существующих секциях, сколько памяти они занимают и прочую заголовочную информацию. Например, можно сказать, что размер секции *.text* составляет 244 байт.

Таблица символов файла:

riscv64-unknown-elf-objdump.exe –t main.o

Изображение выглядит как текст

Автоматически созданное описание

Как и следовало ожидать, таблица содержит один глобальный (флаг “g”) символ типа «функция» (“F”) – символ *main*. Так же можно заметить 2 символа \*UND\*. UND означает, что символы *printf* и *subMin* использовались в ассемблерном коде, из которого был получен данный объектный файл, но не были определены. Ассемблер сделал вывод о том, что эти символы должны быть определены где-то ещё, и отразил это в таблице символов.

Изучим содержимое секции *.text*:

riscv64-unknown-elf-objdump –s –j .text main.o

Изображение выглядит как текст

Автоматически созданное описание

Как мы уже видели, секция содержит 244 байт. Учитывая, что целевой архитектурой является RV32I (без каких-либо расширений), в которой инструкция всегда имеет размер 32 разряда – 4 байта, это соответствует 61 инструкциям.

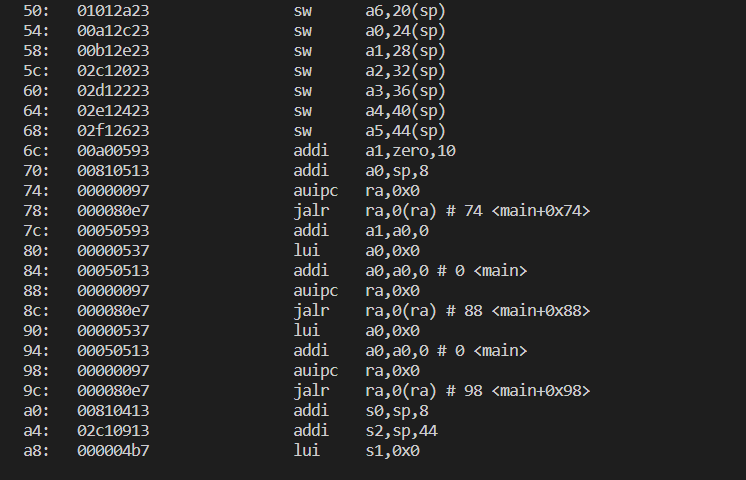
Коды инструкций программы, сформированные ассемблером и размещенные в секции *.text*, легко декодировать, пользуясь сводными таблицами, приведенными в конце спецификации “The RISC-V Instruction Set Manual Volume I: User-Level ISA”. Разумеется, процедура декодирования кодов инструкций является «механической» (иначе как бы ее реализовывало техническое устройство – процессор), следовательно, разумно поручить ее выполнение ЭВМ.

Декодирование кодов инструкции (дизассемблирование):

riscv64-unknown-elf-objdump -d -M no-aliases -j .text main.o

Изображение выглядит как текст

Автоматически созданное описание



Изображение выглядит как текст

Автоматически созданное описание

Можно заметить, что дизассемблированный код практически идентичен сгенерированному (за исключением псевдоинструкций).

Изучим содержимое секции *.comment*:

riscv64-unknown-elf-objdump –s –j .comment main.o

Изображение выглядит как текст

Автоматически созданное описание

Секция *.comment* – секция данных о версиях. Содержит сведения о GCC версии 8.3.0 от SiFive.

Аналогичным образом можно проанализировать файл *submin.o*.

Компоновка

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

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

Изображение выглядит как текст

Автоматически созданное описание

Можно видеть, что файл имеет формат ELF и действительно является исполняемым (флаг “EXEC\_P”), после загрузки его выполнение должно начаться с адреса 0x00010090 (entry point).

Далее изучим секции файла:

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

Изображение выглядит как текст

Автоматически созданное описание

Состав секций *a.out* значительно расширен по сравнению с *main.o*. Также увеличились размеры секций *.text*, *.data*, *.bss* и *.comment*. Дополнительные секции появились из других объектных файлов, переданных на вход компоновщика.

Таблица символов после компоновки также значительно расширилась:

riscv64-unknown-elf-objdump –t a.out

(фрагмент)

Изображение выглядит как текст

Автоматически созданное описание

Изучим содержимое секции *.text*:

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

Результирующий файл “a.ds” весьма велик (21405 строк), поэтому здесь мы рассмотрим только его фрагменты.

Изображение выглядит как текст

Автоматически созданное описание

Код, начинающийся с метки “\_start” обеспечивает инициализацию памяти, регистров процессора и среды времени выполнения, после чего передает управление определенной нами функции *main*.

После возврата из функции *main* управление будет передано блоку *exit*.

Изображение выглядит как текст

Автоматически созданное описание

А по завершению работы блока exit программа перейдёт в блок *\_exit*,

Изображение выглядит как текст

Автоматически созданное описание

в котором программа завершает свою работу.

# **Создание статической библиотеки и Makefile**

Статическая библиотека (static library) является, по сути, архивом (набором, коллекцией) объектных файлов, среди которых компоновщик выбирает «полезные» для данной программы: объектный файл считается «полезным», если в нем определяется еще не разрешенный компоновщиком символ.

В нашем примере функционал по решению задачи представлен только в одном файле (*submin.o*). Однако, это не помешает поместить нам его в статическую библиотеку:

riscv64-unknown-elf-ar –rsc submin\_lib.a submin.o

Результирующим файлом является *submin\_lib.a*. Проверим его содержимое:

riscv64-unknown-elf-ar –t submin\_lib.a



Используем полученную статическую библиотеку для сборки программы:

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 --save-temps main.c submin\_lib.a -o main.out

Изучим таблицы символов полученного исполняемого файла *main.out*:

riscv64-unknown-elf-objdump –t main.out >symtabl

Содержание файла *symtabl* (фрагмент):

Изображение выглядит как текст, табличка

Автоматически созданное описание

Здесь, на 191 строчке, мы можем заметить функцию subMin из библиотеки submin\_lib.a, значит сборка произошла корректно.

Здесь мы создавали библиотеку и собирали программу уже по готовому объектному файлу *submin.o*. Однако, если нам надо будет выполнять сборку с самого начала, например если мы изменили *submin.c*, для это придётся вводить множество команд. В таких случаях для облегчения процесса сборки программы используется Makefile.

Создадим Makefile для автоматической сборки проекта, он будет выполнять следующие действия:

1. Получение объектного файла *submin.o* из *submin.c*

2. Архивация объектного файла *submin.o* и создание статической библиотеки *submin\_lib.a*

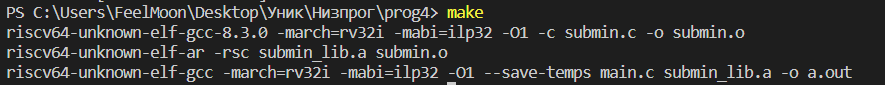
3. Компоновка объектного файла main.c и библиотеки *submin\_lib.a* для получения исполняемого файла *a.out*

Makefile:

Изображение выглядит как текст, снимок экрана, экран

Автоматически созданное описание

Запускаем Makefile с помощью команды *make*:



На выходе получаем исполняемый файл *a.out*.

riscv64-unknown-elf-objdump –t a.out >symtabl

Сравнив таблицы символов предыдущего и текущего файлов, можно сделать вывод, что содержание файлов одно и то же.

Вывод

В ходе лабораторной работы была разработана программа на языке С, проведена её пошаговая сборка: препроцессирование, компиляция, ассемблирование и компоновка.

Были закреплены знания языка C, ассемблера RISC-V, получены навыки работы с препроцессором, компилятором, ассемблером и компоновщиком пакета GCC и драйвером компилятора riscv64-unknown-elf-gcc. Были изучены особенности каждого этапа пошаговой сборки набора программ, а также инструменты, позволяющие выделить разработанные программы в статическую библиотеку и автоматизировать сборку этой библиотеки (Makefile).