Санкт-Петербургский политехнический университет Петра Великого

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**Отчёт по лабораторной работе № 4**

**Дисциплина**: Низкоуровневое программирование

**Тема**: Раздельная компиляция

Вариант №19

Выполнил

студент гр.3530901/10005 Стеблецов Р. А.

Преподаватель Коренев Д. А.

“ ”

Санкт-Петербург

2022

1. **Формулировка задачи**

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.

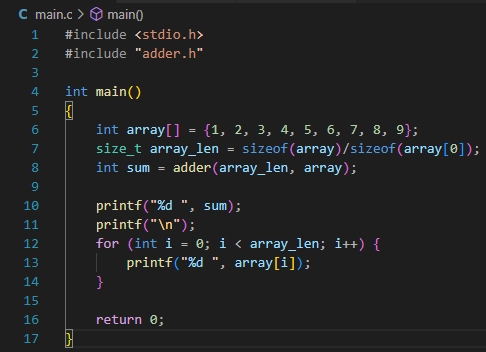
Собрать программу «по шагам». Проанализировать выход препроцессора и компилятора. Проанализировать состав и содержимое секций, таблицы символов, таблицы перемещений и отладочную информацию, содержащуюся в объектных файлах и исполняемом файле.

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.

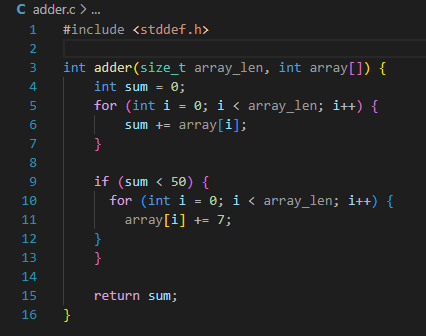
1. **Вариант задания**

Разработать программу для нахождения суммы всех элементов массива. Если сумма меньше 50 – увеличиваем значения всех элементов на 7.

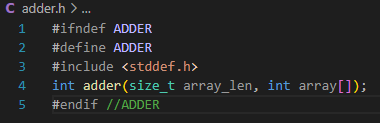
1. **Текст программы**



Файл «main.c»



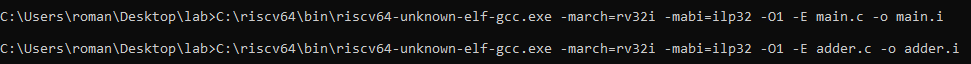
Файл «adder.c»



Заголовочный файл «adder.h»

1. **Сборка программ «по шагам», анализ промежуточных и результирующих файлов**

Первым шагом является препроцессирование файлов исходного текста (файлов “main.c” и “adder.c”), результаты записываются в файлы “main.i” и “adder.i”:



C:\riscv64\bin\riscv64-unknown-elf-gcc.exe -march=rv32i -mabi=ilp32 -O1 -E main.c -o main.i

C:\riscv64\bin\riscv64-unknown-elf-gcc.exe -march=rv32i -mabi=ilp32 -O1 -E adder.c -o adder.i

Параметры:

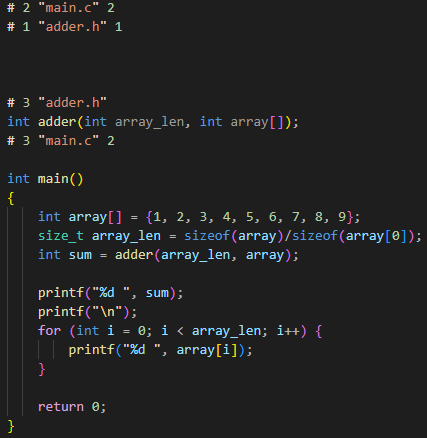
-march=rv32i -mabi=ilp32 – целевым является процессор с базовой архитектурой системы команд RV32I;

-O1 – выполнять простые оптимизации генерируемого кода (мы используем эту опцию в примерах, потому что обычно генерируемый код получается более простым);

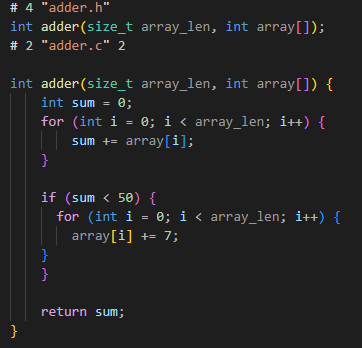
-E – остановить процесс сборки после препроцессирования;

-o – путь к выходному файлу.

Результаты работы препроцессора мало отличаются от исходных версий программ:



Файл «main.i»



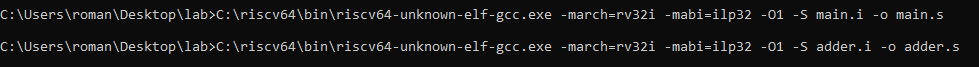
Файл «adder.i»

Результат препроцессирования содержится в файле adder.i и main.i. По причине того, что исходные файлы содержат заголовочные файлы 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 adder.h.

Появившиеся нестандартные директивы, начинающиеся с символа “#”, используются для передачи информации об исходном тексте из препроцессора в компилятор.

**Компиляция**

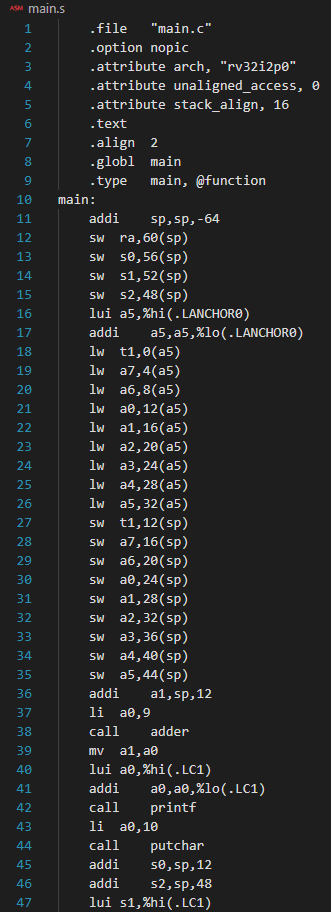
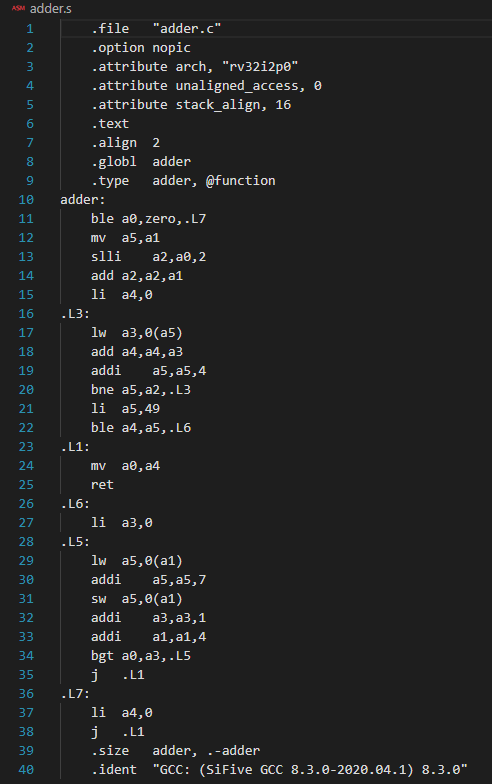
Далее необходимо выполнить компиляцию файлов “main.i” и “adder.i”, сохранив результат – сгенерированный код на языке ассемблера – в файлы “main.s” и “adder.s”.

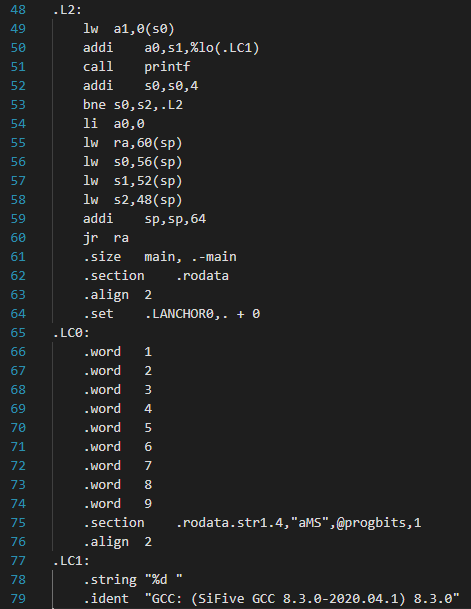


Параметры:

-S – остановить процесс сборки после компиляции, не запуская ассемблер;

Все остальные параметры из прошлого пункта означают тоже самое.





Файл «main.s» Файл «adder.s»

В файле main.s, можно заметить обращение к подпрограмме adder (значение регистра ra, содержащее адрес возврата из main, сохраняется на время вызова в стеке). Следует отметить, что символ adder используется в файле, но никак не определяется.

**Ассемблирование**

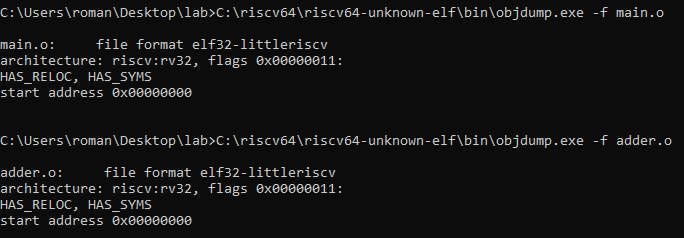
Ассемблирование файлов “main.s” и “adder.s” выполняется по следующей команде:

**

Параметры:

-c – остановить процесс сборки после ассемблирования.

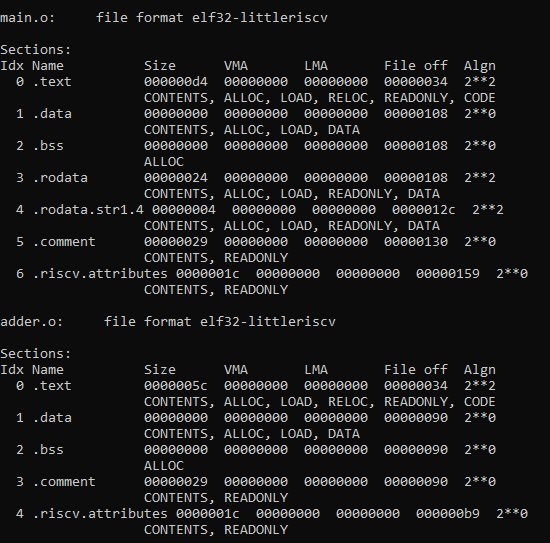
На выходе получили объектные файлы main.o и adder.o. Они содержат коды инструкций, таблицу символов и таблицу перемещений. В отличие от ранее рассмотренных файлов, объектный файл не является текстовым, для изучения его содержимого используем утилиту objdump, отображающую содержимое бинарных файлов в текстовом виде.



Как известно, содержательная часть объектного файла разбита на «разделы», называемые обычно секциями. Следующая команда обеспечивает отображение заголовков секций файлов “main.o” и “adder.o”:



Заголовки секций файлов main.o и adder.o:



В файлах “main.o” и “adder.o” имеются следующие секции:

.text – секция кода, в которой содержатся коды инструкций (название секции обусловлено историческими причинами);

.data – секция инициализированных данных;

.bss – секция неинициализированных статических переменных (название секции также обусловлено историческими причинами);

.rodata – аналог .data для неизменяемых данных

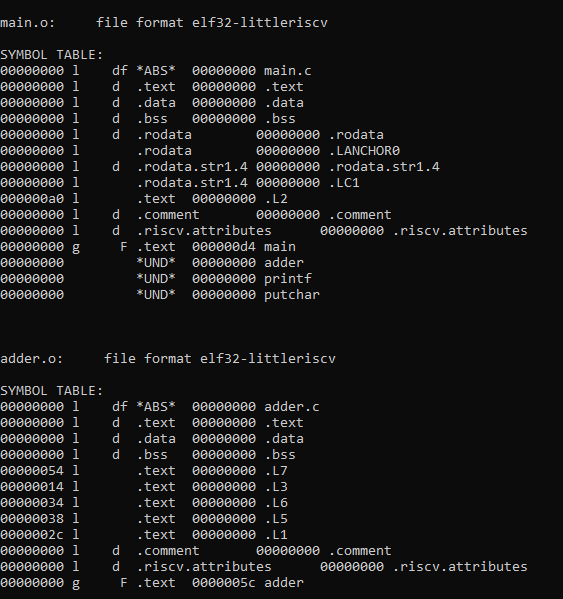
.comment – секция данных о версиях размером 12 байт

.riscv.attributes – информация про RISC-V

Изучим содержимое таблиц символов объектных файлов “main.o” и “adder.o”:



Таблица символов файла main.o и adder.o:

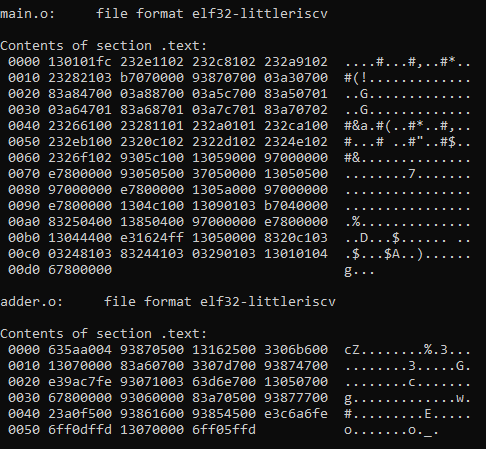


В таблице символов main.o имеется запись: символ “adder” типа \*UND\*. Эта запись означает, что символ “adder” использовался в ассемблерном коде, из которого был получен данный объектный файл, но не был определен; ассемблер сделал вывод о том, что символ должен быть определен где-то еще, и отразил это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

Изучим содержимое секции “.text” объектных файлов “main.o” и “adder.o”:



Вывод утилиты:

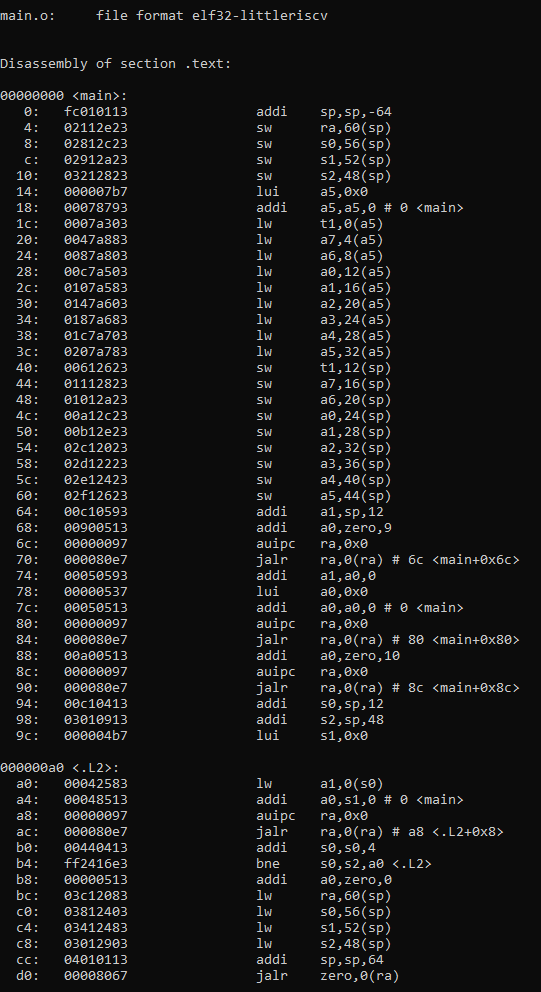
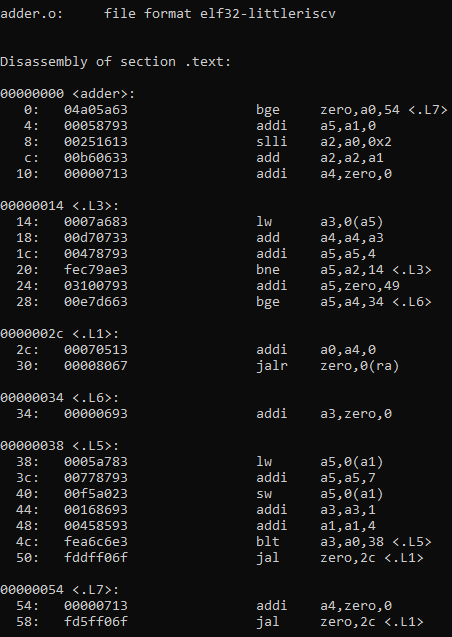


Коды инструкций программы, сформированные ассемблером и размещенные в секции “.text”, легко декодировать, пользуясь сводными таблицами, приведенными в конце спецификации “The RISC-V Instruction Set Manual Volume I: User-Level ISA”.

Процедура декодирования кодов инструкций является «механической» (иначе как бы ее реализовывало техническое устройство – процессор), следовательно, разумно поручить ее выполнение ЭВМ:



Опция “-d” инициирует процесс дизассемблирования (disassemble), опция “-M no-aliases” требует использовать в выводе только инструкции системы команд (но не псевдоинструкции ассемблера).

Сравнивая результаты дизассемблирования “main.o” с “main.s”, легко понять, что псевдоинструкция вызова подпрограммы “adder”, транслировалась ассемблером в следующую пару инструкций:

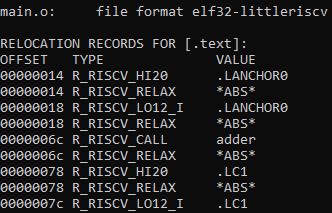


Результатом выполнения этой пары инструкций станет переход на адрес 6c, то есть зацикливанием. Ассемблер не имел возможности определить целевой адрес перехода (кроме того, что этот адрес обозначен символом “adder”), поэтому не мог сформировать корректную инструкцию (пару инструкций) передачи управления. В результате была сформирована пара инструкций с некорректными (нулевыми) значениями непосредственных операндов. Для получения исполняемого кода эта пара инструкций должна быть исправлена компоновщиком.

Информация обо всех «неоконченных» инструкциях передается ассемблером компоновщику посредством таблицы перемещений:

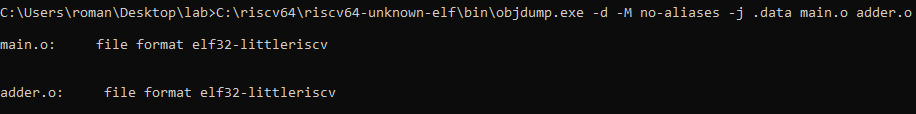


Вывод утилиты (фрагмент):

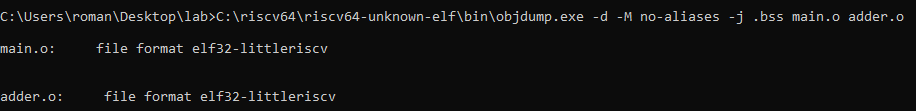


В файле “main.o” имеется две записи, относящиеся к адресу 14 (как мы видели выше, по этому адресу в “main.o” находится первая инструкция пары auipc+jalr).

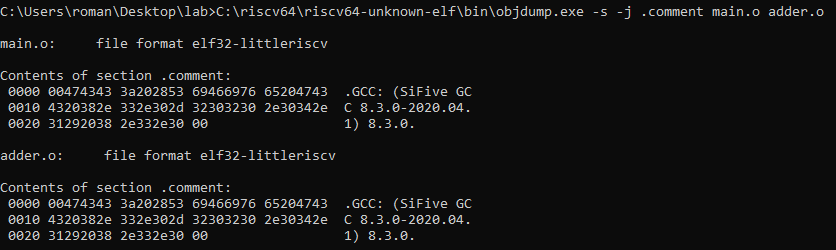
Секции .data объектных файлов – секции инициализированных данных – не содержат данных, размер секций, как было выведено выше, равен нулю:



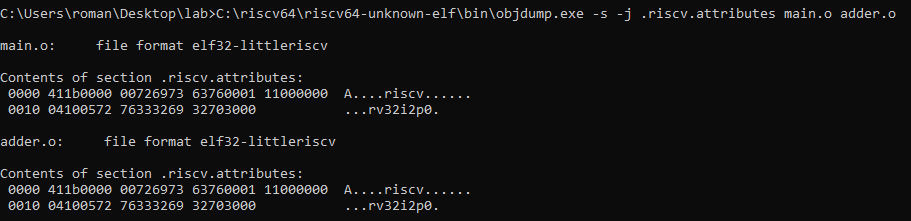
Секции .bss объектных файлов – секции данных, инициализированных нулями – таким же образом пусты:



Секция .comment – секция данных о версиях – и для одного и для другого файла содержит одни и те же значения – сведения о GCC:



Секция .riscv.attributes обоих объектных файлов содержит одну и ту же информацию об используемой архитектуре команд RV32I:



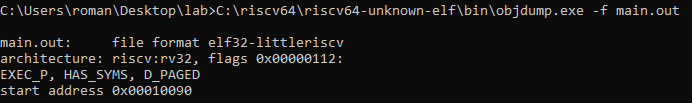
**Компоновка**

Компоновка программы выполняется по следующей команде:



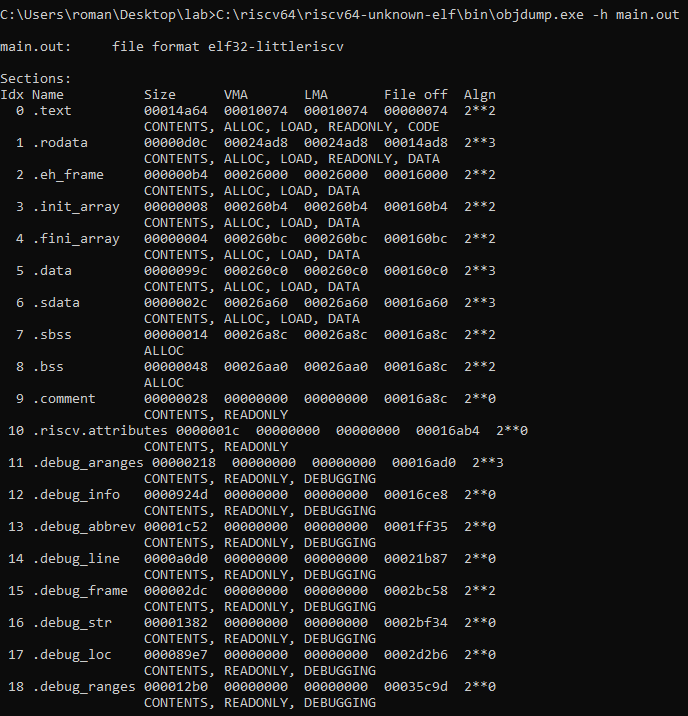
Результатом является исполняемый файл “main.out” (имя выходного файла указано явно опцией “-o”, если бы она отсутствовала, использовалось имя файла по умолчанию “a.out”).

Сформированный компоновщиком файл “main.out”, также является «бинарным»:



Флаг EXEC\_P указывает, что файл действительно является исполняемым, после загрузки его выполнение должно начаться с адреса 0x00010090.

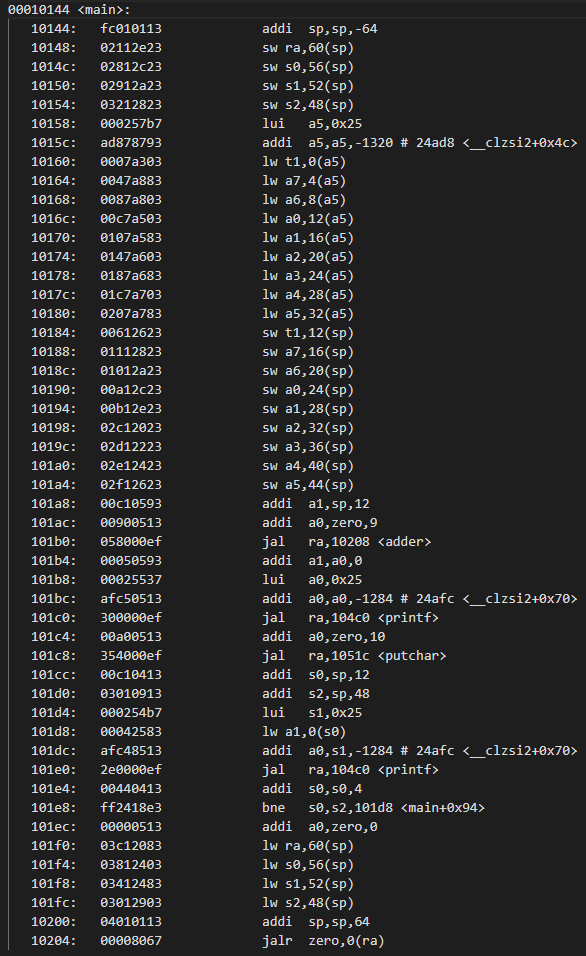
Перечислим секции исполняемого файла:



В исполняемом файле производится слияние содержания секций обоих объектных файлов, а также значительное расширение списка секций новыми блоками.

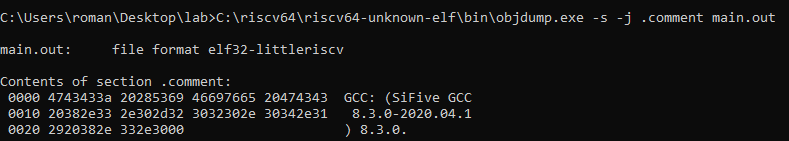
Изучим содержимое секции “.text” полученного в результате компоновки программы исполняемого файла:

 Нас интересует только небольшой фрагмент результирующего файла “main.ds”:

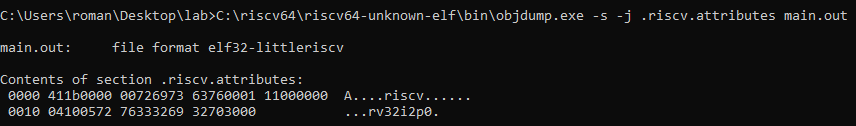


Компоновщик все переходы auipc+jalr, заменил на одну инструкцию jal и корректным адресом перехода.

Секция .comment хранит сведения о GCC версии 8.3.0:



Секция .riscv.attributes содержит информацию об используемой архитектуре команд RV32I:



Проанализируем таблицу перемещений исполняемого файла:



Таблица перемещений оказывается пуста, все необходимые релокации, оптимизации и замены инструкций были успешно проведены компоновщиком.

Итогом сборки программ на языке C по шагам является исполняемый на процессорах архитектуры RISC-V файл, решающий задачу суммирования элементов массива и, в том случае, если сумма меньше 50, увеличение элементов массива на 7.

1. **Создание статической библиотеки**

Статическая библиотека является, по сути, архивом объектных файлов.

Поместим adder.o в статическую библиотеку sumlib.a:

 Параметры:

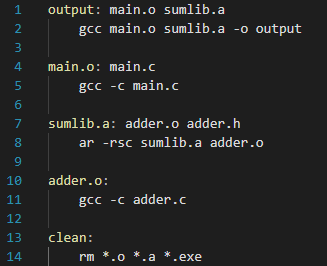
-r – заменить старые файлы с такими названиями (adder.o), если они уже есть в архиве;

-s – записать «index» в архив. Index – это список всех символов, объявленных во включенных в архив объектных файлах, и его присутствие ускоряет линковку;

-с – создать архив, если его еще не было.

Используем статическую библиотеку для сборки программ, для этого напишем make-файл. Makefile – это набор инструкций для программы make, которая помогает собирать программу из файлов в один вызов make.

Makefile:



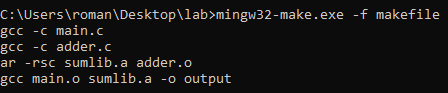
Что происходит в Makefile:

1. Создаём объектный файл main.o из исходного main.c

2. Создаём объектный файл adder.o из исходного adder.c

3. Архивируем объектный файл adder.o (создаём статическую библиотеку lib.a)

4. Компонуем статическую библиотеку lib.a с объектным файлом main.o и получаем исполняемый файл output



1. С помощью команды make собираем программу: make обращается к Makefile и выполняет инструкции в нём (как видим, порядок инструкций соответствует алгоритму выше)



2. Получаем исполняемый файл output. Проверяем его работу. Он работает верно – в программу подается массив [1, 2, 3, 4, 5, 6, 7, 8, 9]. Его сумма = 45, что < 50, поэтому каждый элемент массива был увеличен на 7.

1. **Вывод**

В ходе выполнения данной лабораторной работы была разработана функция на языке C, реализующая заданную вариантом задания функциональность (Разработать программу для нахождения суммы всех элементов массива. Если сумма меньше 50 – увеличиваем значения всех элементов на 7.). Определение функции было помещено в отдельный исходный файл, также был оформлен заголовочный файл. Была разработана тестовая программа на языке C.

Были изучены особенности каждого этапа пошаговой сборки набора программ, а также инструменты, позволяющие выделить разработанные программы в статическую библиотеку и автоматизировать сборку этой библиотеки.

Проанализированы ход сборки библиотеки и программы, созданные файлы зависимостей.