Санкт-Петербургский политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**Отчёт по лабораторной работе № 4**

Дисциплина: Низкоуровневое программирование

Тема: раздельная компиляция

Вариант: 2

Выполнил студент гр. 3530901/10003 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А.С. Гаранин

(подпись)

Принял преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Коренев Д.А.

(подпись)

«\_\_\_\_»\_\_\_\_\_\_\_\_\_\_\_\_\_\_2022 г.

Санкт-Петербург

2022

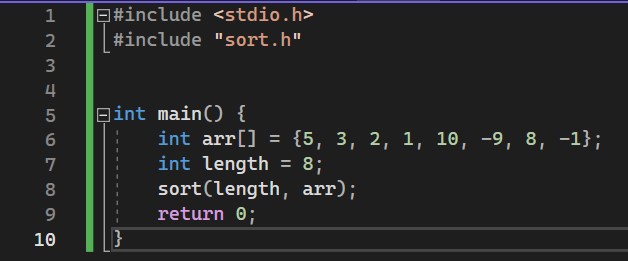
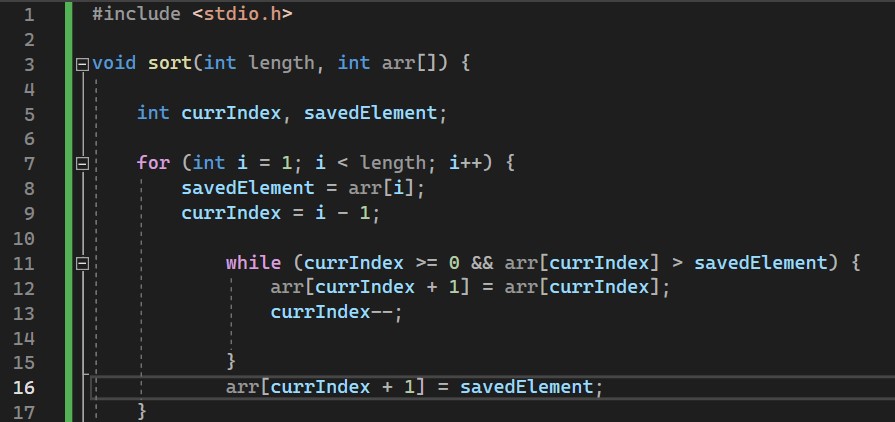
**Оглавление**

1. ТЗ………………………………………………………………………стр. 3
2. Программа на языке С……………………………………….……….стр. 4
3. Препроцессирование……..………………………………….……….стр. 5
4. Компиляция……..…………………………………………………….стр. 7
5. Ассемблирование………………………………………………….…стр. 9

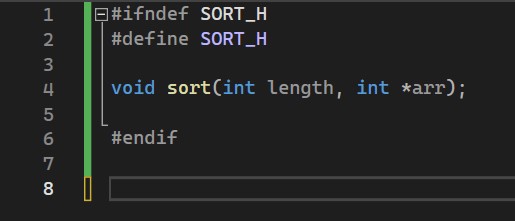
Работа программы Orders1…………………………………………стр. 4

**ТЗ**

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 сортировку массива вставками. 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 исполняемом 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

******Main.c**

**Sort.c**

****

**Заголовочный файл**

**Сборка программы по шагам**

**1.Препроцессирование**

Для запуска препроцессирования файлов main.c и sort.c и создания файлов main.i и sort.i (единицы трансляции) воспользуемся 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 sort.c -o sort.i

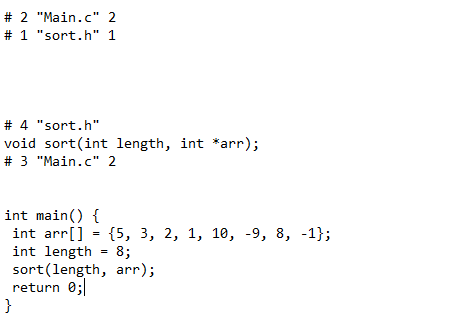
Используем следующие параметры:

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

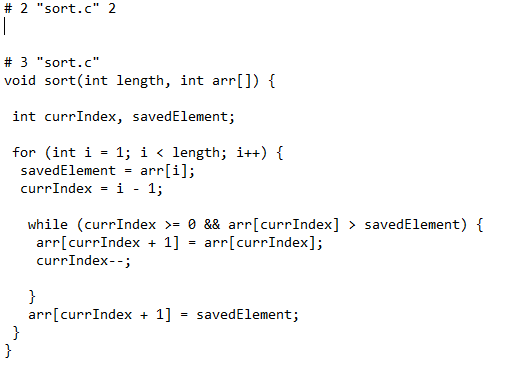
-O1 - выполнять простую оптимизацию генерируемого кода

-Е - остановить процесс сборки после препроцессирования

-о – выходной файл



**Фрагмент файла Main.i**

****

**Фрагмент файла Sort.c**

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

**2. Компиляция**

Для компиляции программы на языке ассемблера и создания файлов Main.s и sort.s выполним следующие команды:

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 sort.i -o sort.s

Для останова процесса сборки после компиляции (без запуска ассемблера) используется опция “-S” драйвера компилятора

Получаем файлы содержащие код программы на языке ассемблера:

**Main.s – тестовая программа**

.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,-48

sw ra,44(sp)

lui a5,%hi(.LANCHOR0)

addi a5,a5,%lo(.LANCHOR0)

lw a7,0(a5)

lw a6,4(a5)

lw a0,8(a5)

lw a1,12(a5)

lw a2,16(a5)

lw a3,20(a5)

lw a4,24(a5)

lw a5,28(a5)

sw a7,0(sp)

sw a6,4(sp)

sw a0,8(sp)

sw a1,12(sp)

sw a2,16(sp)

sw a3,20(sp)

sw a4,24(sp)

sw a5,28(sp)

mv a1,sp

li a0,8

call sort

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 5

.word 3

.word 2

.word 1

.word 10

.word -9

.word 8

.word -1

.ident "GCC: (SiFive GCC 8.3.0-2020.04.1) 8.3.0"

**Sort.s – подпрограмма выполняющая сортировку**

.file "sort.c"

.option nopic

.attribute arch, "rv32i2p0"

.attribute unaligned\_access, 0

.attribute stack\_align, 16

.text

.align 2

.globl sort

.type sort, @function

sort:

li a5,1

ble a0,a5,.L1

addi a7,a1,4

addi t1,a0,-1

li a6,0

li a0,-1

j .L5

.L3:

addi a5,a5,1

slli a5,a5,2

add a5,a1,a5

sw a2,0(a5)

addi a6,a6,1

addi a7,a7,4

beq a6,t1,.L1

.L5:

lw a2,0(a7)

mv a5,a6

blt a6,zero,.L3

lw a3,-4(a7)

bge a2,a3,.L3

mv a4,a7

.L4:

sw a3,0(a4)

addi a5,a5,-1

beq a5,a0,.L3

addi a4,a4,-4

lw a3,-4(a4)

bgt a3,a2,.L4

j .L3

.L1:

ret

.size sort, .-sort

.ident "GCC: (SiFive GCC 8.3.0-2020.04.1) 8.3.0"

**3. Ассемблирование**

Ассемблирование файлов main.s и sort.s выполняется по следующим командам:

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -c

Main.s -o Main.o

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -c

sort.s -o sort.o

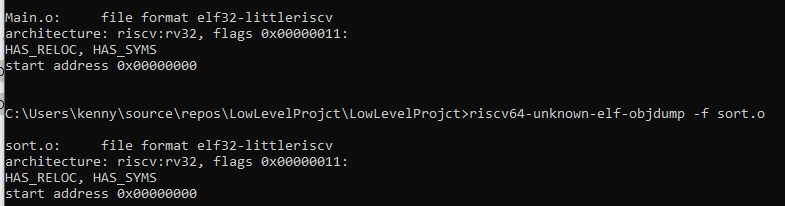
Опция “-c” останавливает процесс сборки после ассемблирования.

На выходе получаем два объектных файла.

Сформированный ассемблером объектный файл “main.o” должен содержать коды инструкций, таблицу символов и таблицу перемещений (relocations). 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:

riscv64-unknown-elf-objdump -f Main.o

riscv64-unknown-elf-objdump -f sort.o

Вывод утилиты:

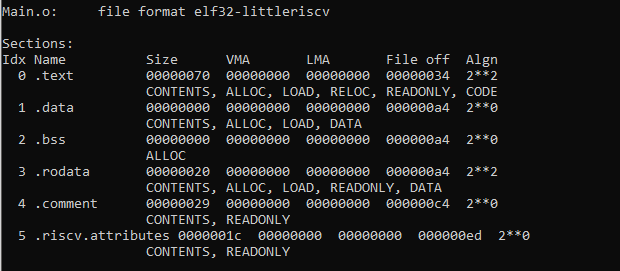
Файл имеет формат ELF (этого следовало ожидать, учитывая название драйвера компилятора), является объектным файлом 32-разрядной архитектуры RISC-V, содержит символы (флаг HAS\_SYMS), но не содержит таблицу перемещений (в списке флагов нет флага HAS\_RELOC).

Объектный файл не должен содержать адрес точки входа (адрес, с которого начинается исполнение программы), однако поскольку соответствующее поле присутствует в заголовке файла формата ELF (т.к. этот же формат используется также для исполняемых файлов, на что указывает буква “E” в его названии), оно заполняется 0.

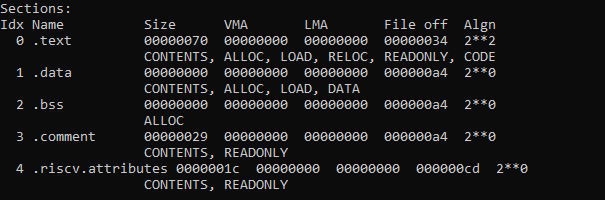
Как известно, содержательная часть объектного файла разбита на «разделы», называемые обычно секциями (section). Следующие команды обеспечивают отображение заголовков секций файлов:

riscv64-unknown-elf-objdump -h Main.o

riscv64-unknown-elf-objdump -h sort.o



**Заголовки секций файла Main.o**



**Заголовки секций файла Sort.o**

В файлах имеются следующие секции:

.text – секция кода, в которой содержатся коды инструкций весом 112 байт (название секции

обусловлено историческими причинами);

.data – секция инициализированных данных;

.bss – секция данных, инициализированных нулями (название секции также обусловлено

историческими причинами);

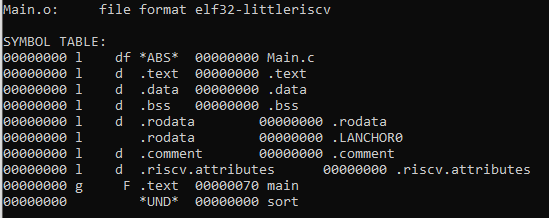
.comment – секция данных о версиях размером 41 байт

.rodata – секция неизменяемых данных

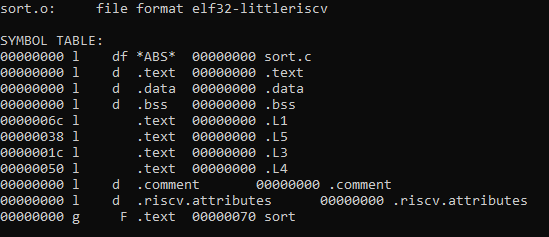
**Теперь изучим таблицы символов файлов:**

riscv64-unknown-elf-objdump –t Main.o

riscv64-unknown-elf-objdump –t sort.o



**Таблица символов Main.o**



**Таблица символов sort.o**

Таблицы символов содержат один глобальный флаг типа функция(F) для sort и main.

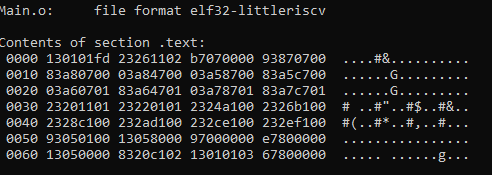
В таблице символов файла Main.o содержится запись о том, что sort – undefined. Это значит, что символ sort использовался в ассемблерном коде, но не был определен, следовательно он должен быть определен где-то еще.

**Инструкции программы – секция .text**

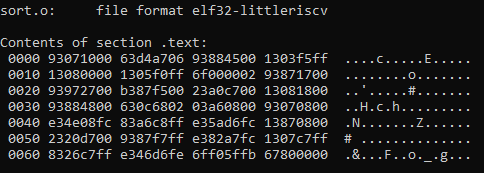
Изучим содержимое секции “.text” :

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

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



**Main.o**



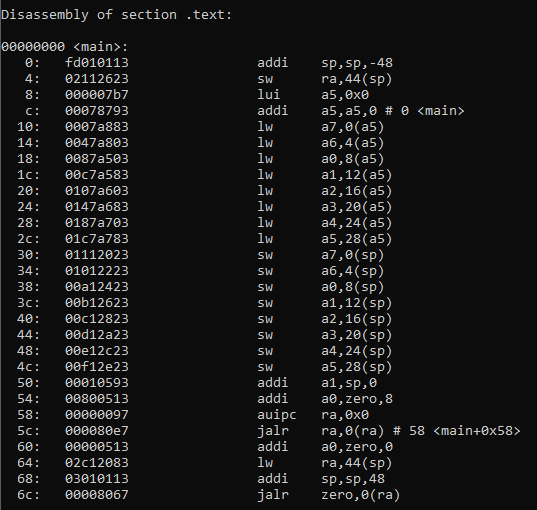
**Sort.o**

Файлы содержат по 28 инструкций размером по 4 байта.

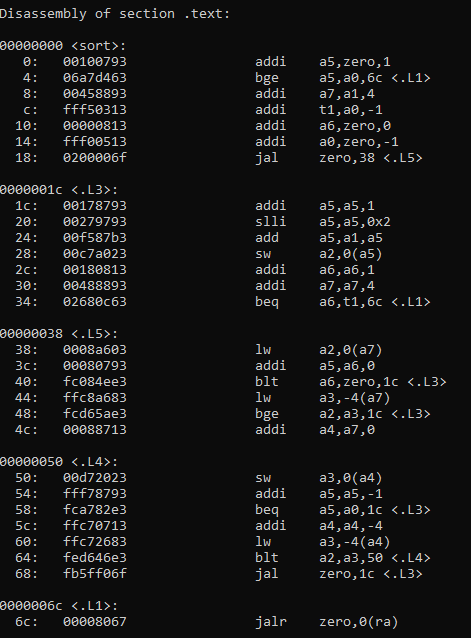
Проведем дизассемблирование секции .text с помощью команд:

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

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



**Инструкции файла Main.s**



**Инструкции sort.s**

Видно, что псевдоинструкция call sort транслируется в пару инструкций:

58: 00000097 auipc ra, 0x0

5c: 000080e7 jalr ra,0(ra) # 58 <main+0x58>

Результатом выполнения этой пары инструкций станет переход на адрес 58, т.е. зацикливание.

Это объясняется тем, что ассемблер не имел возможности

определить *целевой адрес* перехода (кроме того, что этот адрес обозначен символом sort), поэтому не мог сформировать корректную инструкцию. В

результате была сформирована пара инструкций с некорректными (нулевыми) значениями

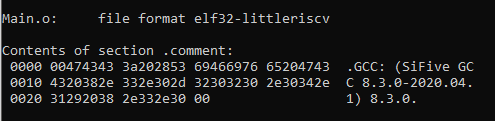
непосредственных операндов. Для получения исполняемого кода эта пара инструкций должна

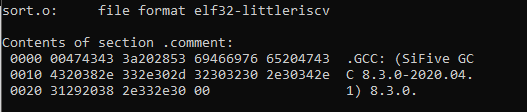
быть исправлена компоновщиком.

**Изучим содержимое секции “.comment” :**

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

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

****

****

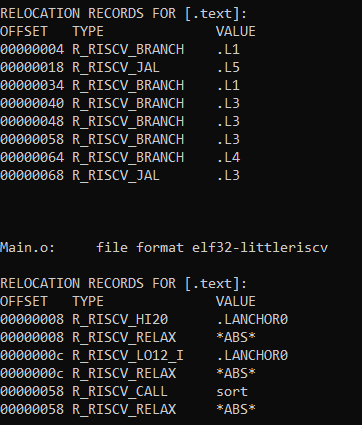
Секции содержат информацию о драйвере компилятора.

**Изучим содержимое таблицы перемещений:**

riscv64-unknown-elf-objdump -d -M no-aliases -r sort.o Main.o

Информация обо всех «неоконченных» инструкциях передается ассемблером компоновщику

посредством таблицы перемещений.

****

Для того чтобы внести необходимые исправления, требуется знать, что исправить, как исправить и какой символ следует использовать, именно эта информация и содержится в записях о перемещениях. Мы видим запись R\_RISCV\_CALL sort, которая сообщает нам о том, что компоновщику нужно исправить инструкцию/инструкции находящиеся по адресу 58, для того, чтобы передача управления осуществлялась корректно. R\_RISCV\_RELAX сообщает компоновщику что инструкции могут быть оптимизированы.

**Компоновка программы**

Произведем компоновку программы:

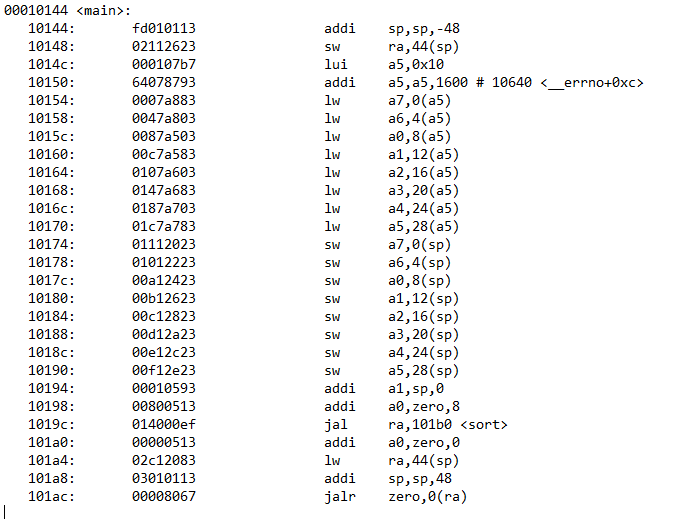
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -v Main.o sort.o -o main

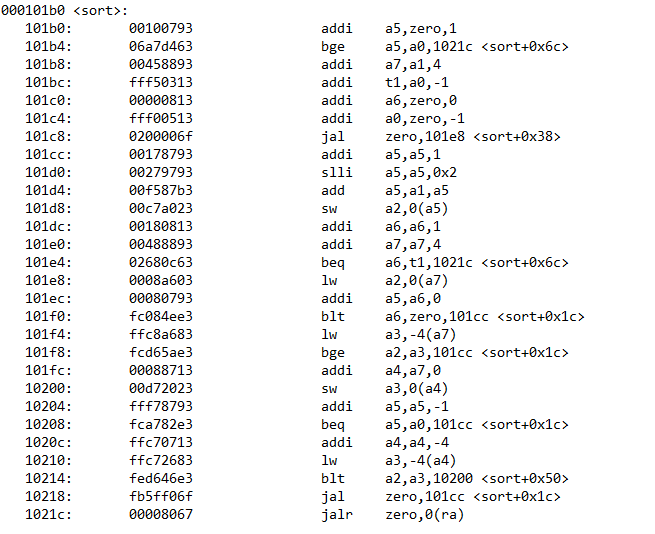
В результате получим исполняемый файл main, так как с помощью опции -о мы задали имя выходного файла main.

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

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

Рассмотрим фрагменты файла main.ds:





Как видно, в результат компоновки попало содержимое обоих объектных файлов – main.o и sort.o Инструкции подпрограммы sort начинаются с адреса 101b0. По адресу 1019с происходит безусловный переход на 101b0, т.е. управление передается подпрограмме корректно. Как видно, в сравнении с кодом из объектных файлов, компоновщик оптимизировал пару инструкций auipc+jalr и использовал вместо них одну инструкцию jal, 20-разрядный непосредственный операнд которой позволяет задавать переход в пределах

1 МиБ относительно адреса инструкции. Это происходит, потому что, в отличии от ассемблера, компоновщик имеет всю информацию об адресах, и ему не нужно по умолчанию использовать пару инструкций auipc+jalr, которая позволяет выполнять переход в пределах 2 ГиБ.

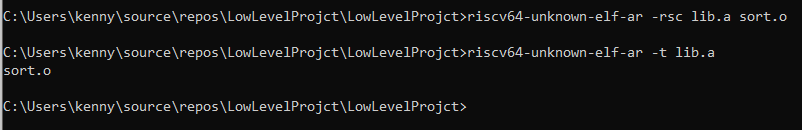
**Создание статической библиотеки**

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.

Выделим объектный файл с подпрограммой в статическую библиотеку:

riscv64-unknown-elf-ar –rsc lib.a sort.o

riscv64-unknown-elf-ar –t lib.a – проверим его содержимое



Используем статическую библиотеку для сборки программы:

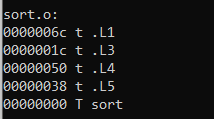
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 --save-temps

Main.c lib.a -o mainwithlib

Получили файл, аналогичный main.

Выведем список символов библиотеки:

riscv64-unknown-elf-nm lib.a



Символом Т обозначаются символы, определенные в соответствующем объектном файле.

Создадим make-файл для сборки программ с использованием статической библиотеки.

main.o:

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

sort.o:

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

lib.a: sort.o

riscv64-unknown-elf-ar -rsc lib.a sort.o

a.out: main.o lib.a

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 main.o lib.a -o a.out