Санкт-Петербургский политехнический университет Петра Великого

Высшая школа интеллектуальных систем и суперкомпьютерных технологий

**Отчёт по лабораторной работе №4**

**Дисциплина**: Низкоуровневое программирование

**Тема**: Раздельная компиляция

Выполнил студент гр. 3530901/90003 А.А. Ундольский

(подпись)

Преподаватель А. О. Алексюк

(подпись)

“ ” 2021 г.

Санкт-Петербург

2021

Оглавление

[1. Техническое задание 3](#_Toc69584074)

[2. Метод решения 3](#_Toc69584075)

[3. Решение 3](#_Toc69584076)

[3.1. Анализ выхода препроцессора: 5](#_Toc69584077)

[3.2. Анализ выхода компилятора: 6](#_Toc69584078)

[3.3. Анализ состава и содержимого секций, таблицы символов, таблицы перемещений и отладочной информации, содержащейся в объектных файлах и исполняемом файле: 8](#_Toc69584079)

[3.4. Содержимое таблицы перемещений: 12](#_Toc69584086)

[3.5. Результат компоновки 14](#_Toc69584087)

[3.6. Анализ отладочной информации 16](#_Toc69584158)

[3.7. Выделение разработанной функции в статическую библиотеку 17](#_Toc69584160)

[3.8. Создание и использование полученной статической библиотеки 17](#_Toc69584166)

[4. Результаты 19](#_Toc69584180)

# Техническое задание

1. На языке С разработать функцию, реализующую нахождение НОД массива. Поместить определение функции в отдельный исходный файл, оформить заголовочный файл. Разработать тестовую программу на языке С.
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 make-файлы для сборки библиотеки и использующей ее тестовой программы. Проанализировать ход сборки библиотеки и программы, созданные файлы зависимостей.

# Метод решения

Нахождение НОД осуществляется по алгоритму Евклида, последовательно применяя его к НОД предыдущей пары чисел и новому числу в массиве.

# Решение

Напишем программу на языке СИ:

gcd.h

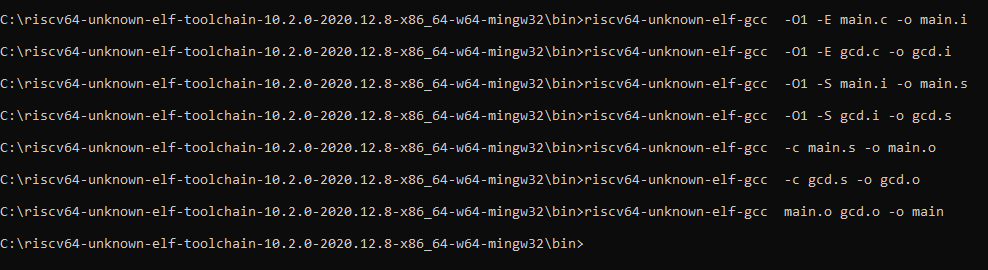
|  |
| --- |
| #ifndef LOWLEVEL\_GCD\_H  #define LOWLEVEL\_GCD\_H  // Функция возвращает НОД двух целых чисел  int getGCD(int val1, int val2);  // Функция возвращает НОД массива целых чисел  int getGCD\_Array(int array[], int n); // array - массив, n - размер массива  #endif // LOWLEVEL\_GCD\_H |

gcd.c

|  |
| --- |
| #include "gcd.h"  // Функция возвращает НОД двух целых чисел (по методу Евклида)  int getGCD(int val1, int val2) {  while (val1 != val2 ) {  if (val1 > val2)  val1 -= val2;  else  val2 -= val1;  }  return val1;  }  // Функция возвращает НОД массива целых чисел  int getGCD\_Array(int array[], int n) { // n - размер массива  int gcd = array[0];  for (int i=1; i<n; i++) {  gcd = getGCD(gcd,array[i]);  }  return gcd;  } |

main.c

|  |
| --- |
| #include <stdio.h>  #include "gcd.h"  int main() {  int array[] = {9, 18, 30, 60, 120, 36, 33, 24}; // Исходный массив  int length = sizeof(array) / sizeof(int); // Размер массива  // Контрольный вывод массива  printf("Input array: [");  for(int i = 0; i < length; i++) {  printf("%d", array[i]);  if (i < length-1 ) printf(", ");  else printf("]\n");  }  // Подсчет и вывод НОД  printf("Great common divisor of array = %d\n", getGCD\_Array(array, length));  } |



**Рис.1.** Сборка программы по этапам

## Анализ выхода препроцессора:

Директивы, прописанные в заголовочном файле, определяют вставку стандартной библиотеки ввода-вывода языка Си. Пользовательская часть кода практически не меняется:

Часть файла main.i

|  |
| --- |
| # 5 "gcd.h"  int getGCD(int val1, int val2);  int getGCD\_Array(int array[], int n);  # 3 "main.c" 2  int main() {  int array[] = {9, 18, 30, 60, 120, 36, 33, 24};  int length = sizeof(array) / sizeof(int);  printf("Input array: [");  for(int i = 0; i < length; i++) {  printf("%d", array[i]);  if (i < length-1 ) printf(", ");  else printf("]\n");  }  printf("Great common divisor of array = %d\n", getGCD\_Array(array, length));  } |

Аналогично происходит препроцессирование функции:

Часть файла gcd.i:

|  |
| --- |
| # 1 "gcd.c"  # 1 "gcd.h" 1  int getGCD(int val1, int val2);  int getGCD\_Array(int array[], int n);  # 2 "gcd.c" 2  int getGCD(int val1, int val2) {  while (val1 != val2 ) {  if (val1 > val2)  val1 -= val2;  else  val2 -= val1;  }  return val1;  }  int getGCD\_Array(int array[], int n) {  int gcd = array[0];  for (int i=1; i<n; i++) {  gcd = getGCD(gcd,array[i]);  }  return gcd;  } |

## Анализ выхода компилятора:

main.s

|  |
| --- |
| .file "main.c"  .option nopic  .attribute arch, "rv64i2p0\_m2p0\_a2p0\_f2p0\_d2p0\_c2p0"  .attribute unaligned\_access, 0  .attribute stack\_align, 16  .text  .section .rodata.str1.8,"aMS",@progbits,1  .align 3  .LC1:  .string "Input array: ["  .align 3  .LC2:  .string "%d"  .align 3  .LC3:  .string ", "  .align 3  .LC4:  .string "]"  .align 3  .LC5:  .string "Great common divisor of array = %d\n"  .text  .align 1  .globl main  .type main, @function  main:  addi sp,sp,-80  sd ra,72(sp)  sd s0,64(sp)  sd s1,56(sp)  sd s2,48(sp)  sd s3,40(sp)  lui a5,%hi(.LANCHOR0)  addi a5,a5,%lo(.LANCHOR0)  ld a2,0(a5)  ld a3,8(a5)  ld a4,16(a5)  ld a5,24(a5)  sd a2,0(sp)  sd a3,8(sp)  sd a4,16(sp)  sd a5,24(sp)  lui a0,%hi(.LC1)  addi a0,a0,%lo(.LC1)  call printf  lw a1,0(sp)  lui a0,%hi(.LC2)  addi a0,a0,%lo(.LC2)  call printf  mv s0,sp  addi s3,sp,28  lui s2,%hi(.LC3)  lui s1,%hi(.LC2)  .L2:  addi a0,s2,%lo(.LC3)  call printf  lw a1,4(s0)  addi a0,s1,%lo(.LC2)  call printf  addi s0,s0,4  bne s3,s0,.L2  lui a0,%hi(.LC4)  addi a0,a0,%lo(.LC4)  call puts  li a1,8  mv a0,sp  call getGCD\_Array  mv a1,a0  lui a0,%hi(.LC5)  addi a0,a0,%lo(.LC5)  call printf  li a0,0  ld ra,72(sp)  ld s0,64(sp)  ld s1,56(sp)  ld s2,48(sp)  ld s3,40(sp)  addi sp,sp,80  jr ra  .size main, .-main  .section .rodata  .align 3  .set .LANCHOR0,. + 0  .LC0:  .word 9  .word 18  .word 30  .word 60  .word 120  .word 36  .word 33  .word 24  .ident "GCC: (SiFive GCC-Metal 10.2.0-2020.12.8) 10.2.0" |

gcd.s

|  |
| --- |
| .file "gcd.c"  .option nopic  .attribute arch, "rv64i2p0\_m2p0\_a2p0\_f2p0\_d2p0\_c2p0"  .attribute unaligned\_access, 0  .attribute stack\_align, 16  .text  .align 1  .globl getGCD  .type getGCD, @function  getGCD:  mv a5,a0  mv a0,a1  bne a5,a1,.L5  .L2:  ret  .L3:  subw a0,a0,a5  .L4:  beq a5,a0,.L2  .L5:  ble a5,a0,.L3  subw a5,a5,a0  j .L4  .size getGCD, .-getGCD  .align 1  .globl getGCD\_Array  .type getGCD\_Array, @function  getGCD\_Array:  mv a5,a0  lw a0,0(a0)  li a4,1  ble a1,a4,.L12  addi sp,sp,-32  sd ra,24(sp)  sd s0,16(sp)  sd s1,8(sp)  addi s0,a5,4  addiw s1,a1,-2  slli a4,s1,32  srli s1,a4,30  addi a5,a5,8  add s1,s1,a5  .L9:  lw a1,0(s0)  call getGCD  addi s0,s0,4  bne s0,s1,.L9  ld ra,24(sp)  ld s0,16(sp)  ld s1,8(sp)  addi sp,sp,32  jr ra  .L12:  ret  .size getGCD\_Array, .-getGCD\_Array  .ident "GCC: (SiFive GCC-Metal 10.2.0-2020.12.8) 10.2.0" |

В программе main выполняется обращение к подпрограмме gcd (значение регистра ra, содержащее адрес возврата из main, сохраняется на время вызова в стеке). Следует отметить, что символ getGCD\_Array используется в файле main.s, но никак не определяется.

## Анализ состава и содержимого секций, таблицы символов, таблицы перемещений и отладочной информации, содержащейся в объектных файлах и исполняемом файле:

Сформированный ассемблером объектный файл main.o и gcd.o должны содержать 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:

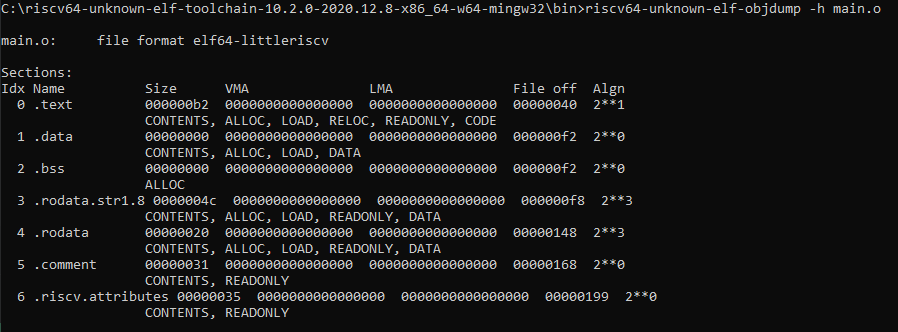


Рис.2. Содержимое заголовков секций main.o

В файле имеются следующие секции:

•.text – секция кода;

•.data – секция инициализированных данных;

•.bss – секция данных, инициализированных нулями;

•.rodata – секция неизменяемых данных;

•.rodata.str1.4 –подсекция неизменяемых данных, используется компилятором для хранения дополнительной информации (например, о типе данных) для компоновщика;

•.comment – секция данных о версиях;

•.riscv.attributes – атрибуты для указания определенных свойств функции (в помощь компилятору для проверок и оптимизации кода).

Значения в столбце size приведены в 16-ричной системе счисления.

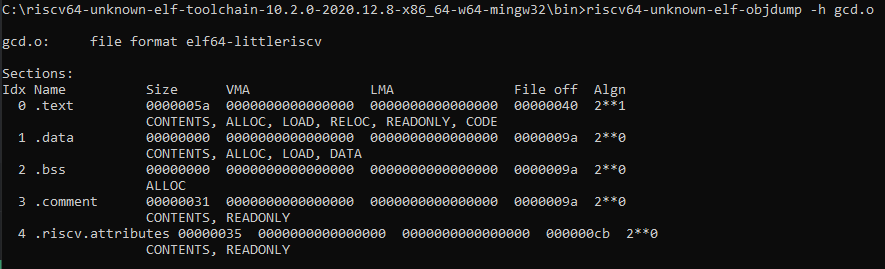


Рис.3. Содержимое заголовков секций gcd.o

Tаблицы символов объектных файлов main.o и gcd.o:

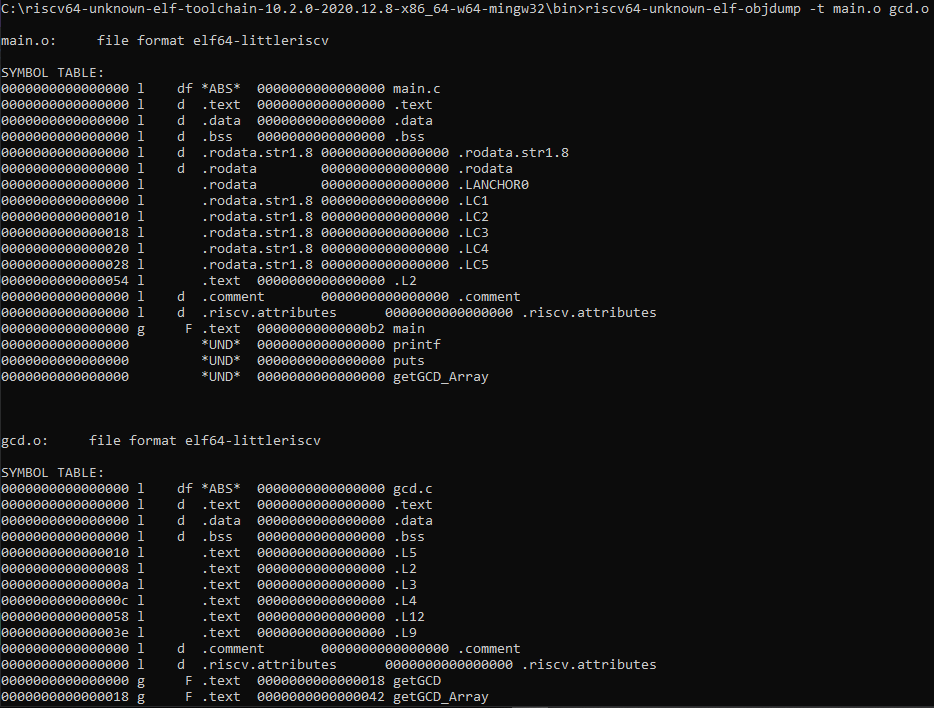


Рис.4. Содержимое таблиц символов объектных файлов

Как и ожидалось таблица содержит 3 глобальные (флаг g) функции (флаг F) – main, getGCD и getGCD\_Array, а также три неопределенных (UND) символа.

getGCD – внутренняя функция, поэтому в условиях данной работы есть смысл пристально рассматривать только getGCD\_Array.

UND означает, что символы printf, puts и getGCD\_Array использовались в ассемблерном коде, из которого был получен данный объектный файл, но не были определены; 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 gcd.o:

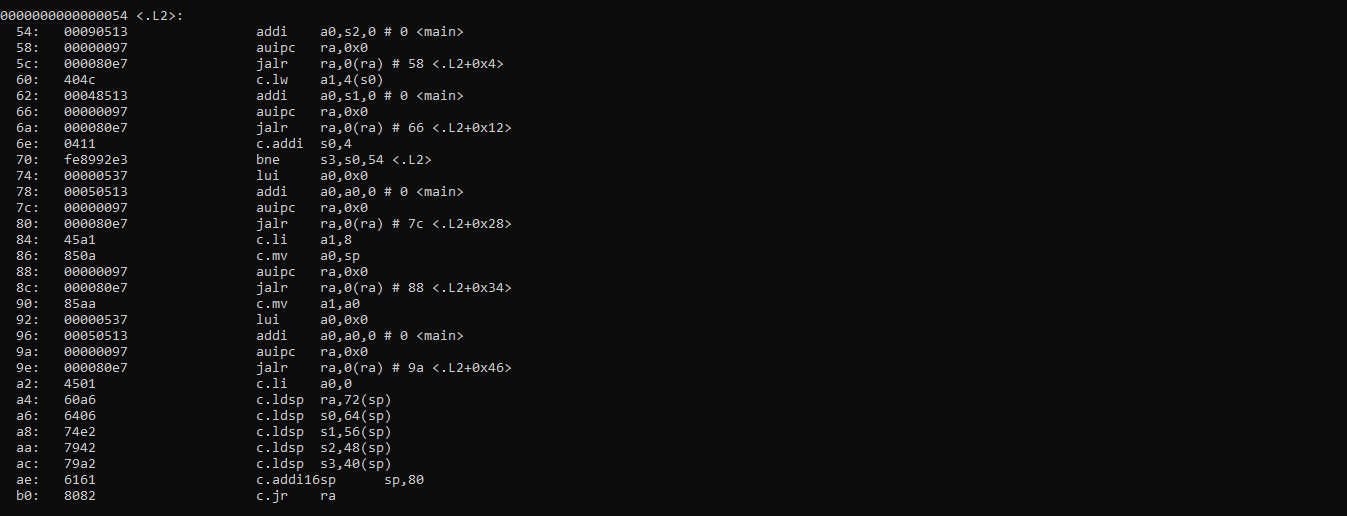
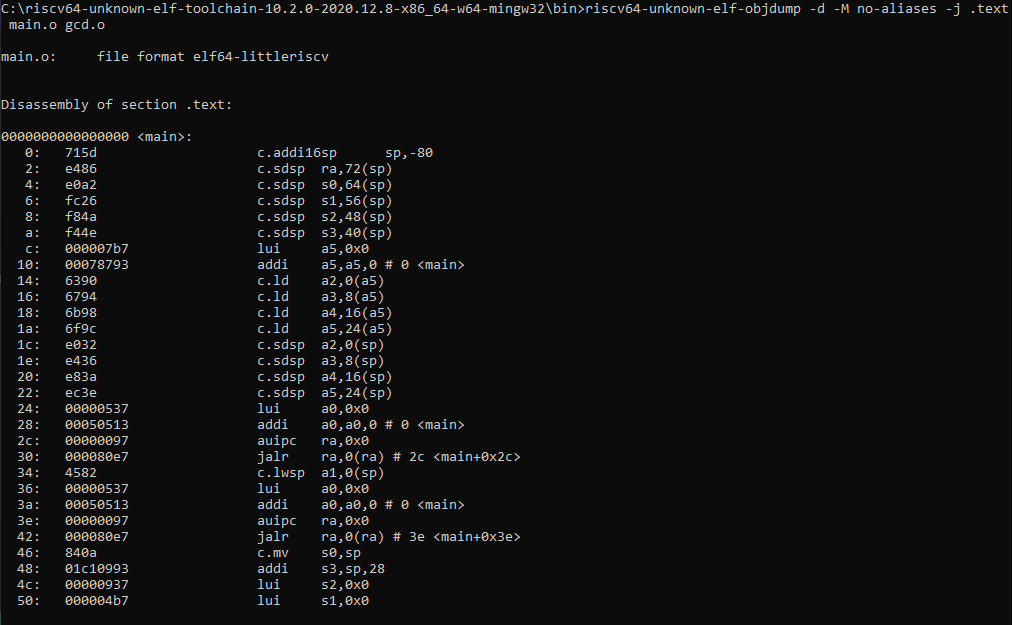


Рис.5. Cодержимое секции .text объектного файла main.o

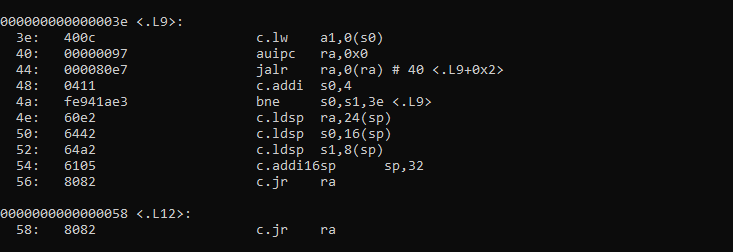
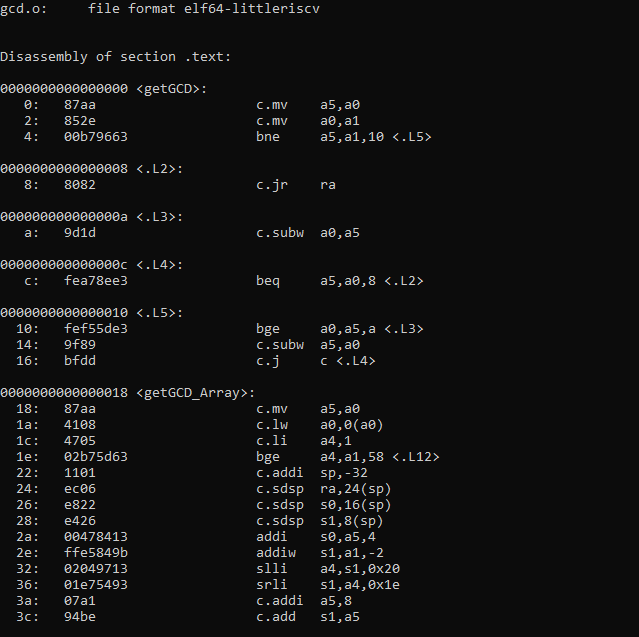


Рис.6. Cодержимое секции .text объектного файла gcd.o

Результат дизассемблирования gcd.o, как и результат дизассемблирования main.o представляют определённый интерес: сравнивая их соответственно с gcd.s и main.s, можно понять, что псевдоинструкции вызова подпрограмм getGCD и getGCD\_Array, транслировались ассемблером соответственно в следующие пары инструкций:





Результатом выполнения первой пары инструкций станет переход на адрес .L2+0x2 (3e+2=40) - произойдет зацикливание. Аналогично, результатом выполнения второй пары инструкций станет переход на адрес .L2+0x34 (54+34=88) - также произойдет зацикливание. Это показано в выводе дизассемблера. Загадочное поведение ассемблера объясняется очень просто: ассемблер не имел возможности определить целевой адрес перехода (кроме того, что этот адрес обозначен соответственно символами getGCD и getGCD\_Array), 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.



## Содержимое таблицы перемещений:

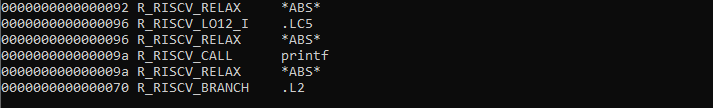
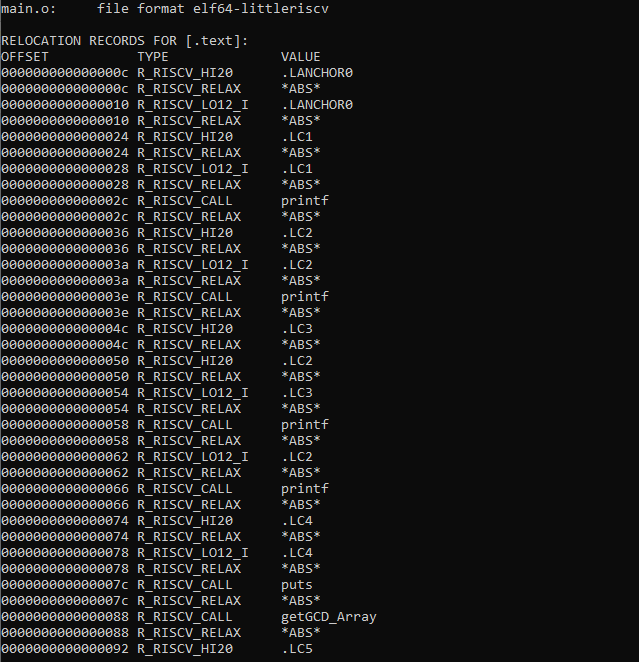


Рис.7. Таблица перемещений main.o

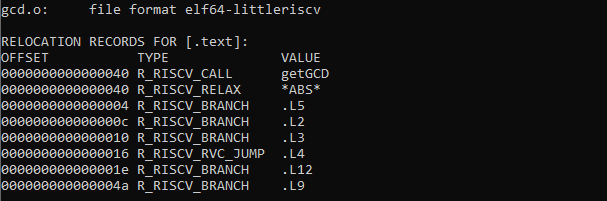
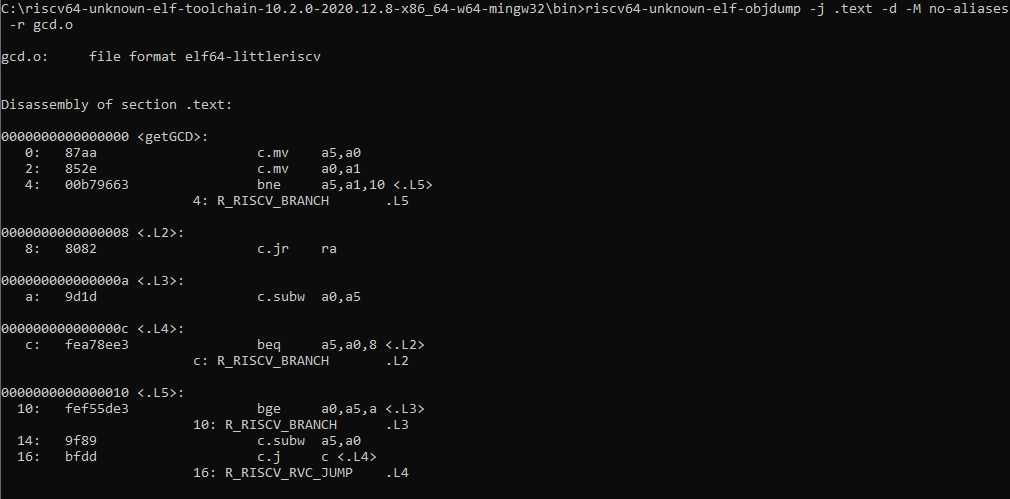
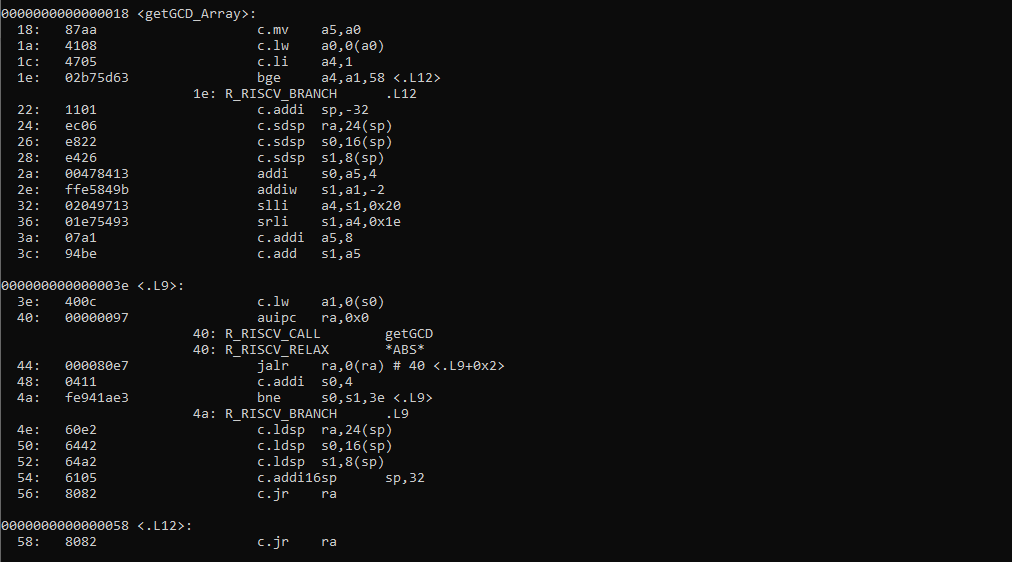
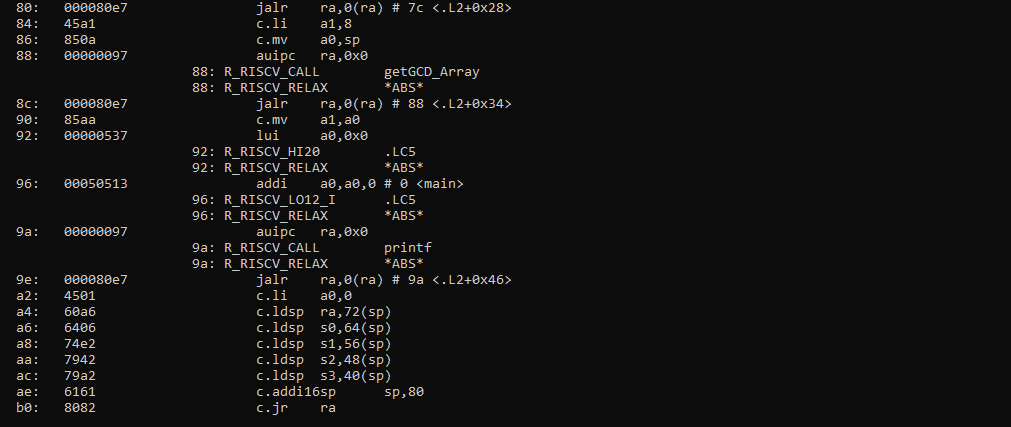
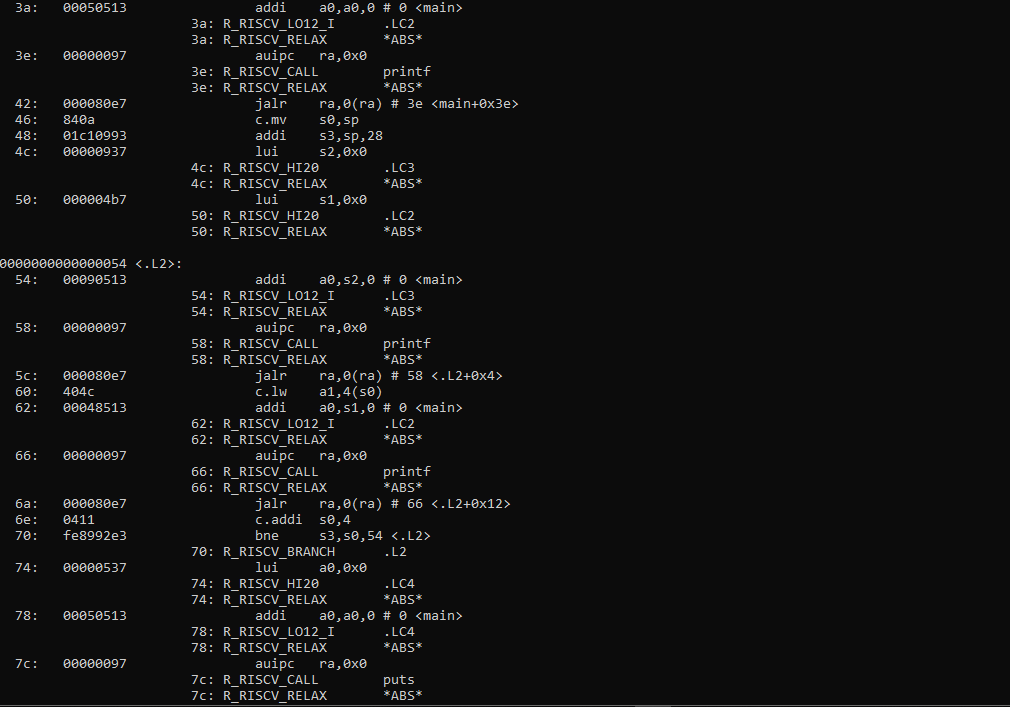
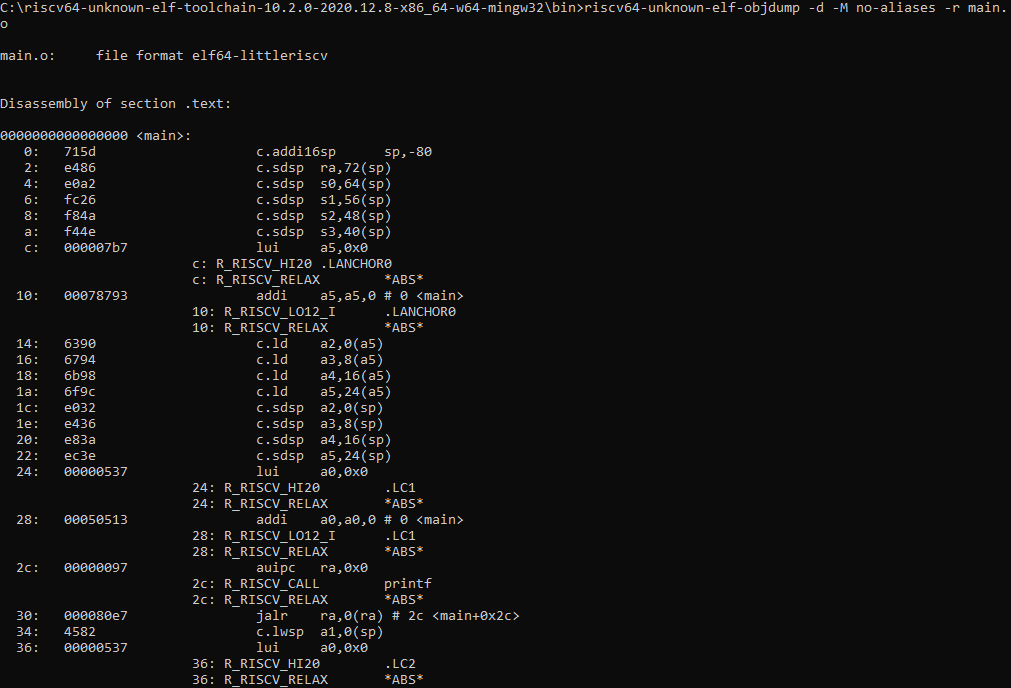


Рис.8. Таблица перемещений gcd.o

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

Содержимое gcd.o, как и содержимое main.o требует модификации, потому что содержит в себе запись о перемещениях (relocation entries), которая относится к адресу 40 (как мы видели выше, по этому адресу в gcd.o находится инструкция пары auipc+jalr). В файле же main.o имеется 7 записей, среди которых есть запись относящаяся к адресу 88 (аналогично, по этому адресу в main.o находится инструкция пары auipc+jalr). Дизассемблирование и вывод таблицы перемещений можно совместить. 

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. Так, в первой записи таблице перемещений для gcd.o указано, что по адресу 40 следует исправить пару инструкций (тип перемещения “R\_RISCV\_CALL”) так, чтобы результат соответствовал вызову подпрограммы getGCD. Типы перемещений специфичны для каждой архитектуры системы команд и обычно определены в ABI (Application Binary Interface). \*



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. Так, в первой записи таблицы перемещений для main.o указано, что по адресу 88 следует исправить пару инструкций (тип перемещения “R\_RISCV\_CALL”) так, чтобы результат соответствовал вызову подпрограммы getGCD\_Array. Типы перемещений специфичны для каждой архитектуры системы команд и обычно определены в ABI (Application Binary Interface). \*

* Вторая запись таблицы перемещений специфична для средств разработки RISC-V. Записи типа “R\_RISCV\_RELAX” заносятся в таблицу перемещений в дополнение к записям типа “R\_RISCV\_CALL” (и некоторым другим) и сообщают компоновщику, что пара инструкций, обеспечивающих вызов подпрограммы, может быть оптимизирована.

## Результат компоновки

riscv64-unknown-elf-gcc -Wl,--no-relax main.o gcd.o -o main

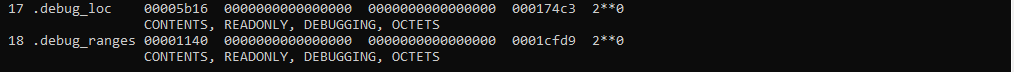
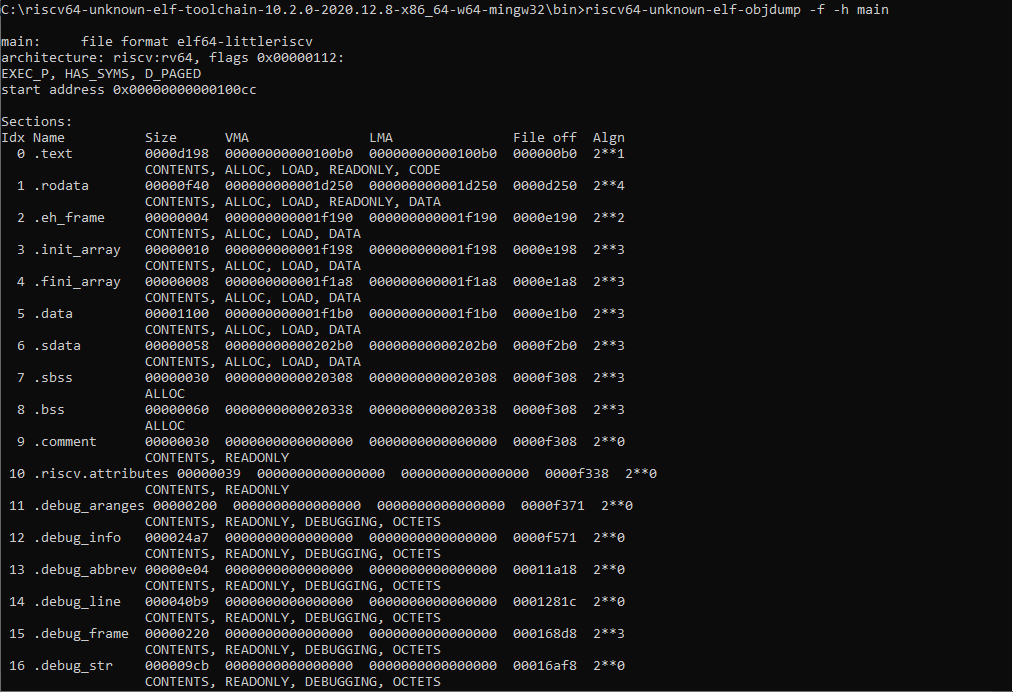
riscv64-unknown-elf-objdump -j .text -d -M no-aliases main >main.ds

main.ds (строки 79-176)

|  |
| --- |
| 0000000000010188 <main>:  10188: 715d c.addi16sp sp,-80  1018a: e486 c.sdsp ra,72(sp)  1018c: e0a2 c.sdsp s0,64(sp)  1018e: fc26 c.sdsp s1,56(sp)  10190: f84a c.sdsp s2,48(sp)  10192: f44e c.sdsp s3,40(sp)  10194: 0001d7b7 lui a5,0x1d  10198: 2a078793 addi a5,a5,672 # 1d2a0 <\_\_clzdi2+0x88>  1019c: 6390 c.ld a2,0(a5)  1019e: 6794 c.ld a3,8(a5)  101a0: 6b98 c.ld a4,16(a5)  101a2: 6f9c c.ld a5,24(a5)  101a4: e032 c.sdsp a2,0(sp)  101a6: e436 c.sdsp a3,8(sp)  101a8: e83a c.sdsp a4,16(sp)  101aa: ec3e c.sdsp a5,24(sp)  101ac: 0001d537 lui a0,0x1d  101b0: 25050513 addi a0,a0,592 # 1d250 <\_\_clzdi2+0x38>  101b4: 00000097 auipc ra,0x0  101b8: 244080e7 jalr ra,580(ra) # 103f8 <printf>  101bc: 4582 c.lwsp a1,0(sp)  101be: 0001d537 lui a0,0x1d  101c2: 26050513 addi a0,a0,608 # 1d260 <\_\_clzdi2+0x48>  101c6: 00000097 auipc ra,0x0  101ca: 232080e7 jalr ra,562(ra) # 103f8 <printf>  101ce: 840a c.mv s0,sp  101d0: 01c10993 addi s3,sp,28  101d4: 0001d937 lui s2,0x1d  101d8: 0001d4b7 lui s1,0x1d  101dc: 26890513 addi a0,s2,616 # 1d268 <\_\_clzdi2+0x50>  101e0: 00000097 auipc ra,0x0  101e4: 218080e7 jalr ra,536(ra) # 103f8 <printf>  101e8: 404c c.lw a1,4(s0)  101ea: 26048513 addi a0,s1,608 # 1d260 <\_\_clzdi2+0x48>  101ee: 00000097 auipc ra,0x0  101f2: 20a080e7 jalr ra,522(ra) # 103f8 <printf>  101f6: 0411 c.addi s0,4  101f8: fe8992e3 bne s3,s0,101dc <main+0x54>  101fc: 0001d537 lui a0,0x1d  10200: 27050513 addi a0,a0,624 # 1d270 <\_\_clzdi2+0x58>  10204: 00000097 auipc ra,0x0  10208: 2b4080e7 jalr ra,692(ra) # 104b8 <puts>  1020c: 45a1 c.li a1,8  1020e: 850a c.mv a0,sp  10210: 00000097 auipc ra,0x0  10214: 042080e7 jalr ra,66(ra) # 10252 <getGCD\_Array>  10218: 85aa c.mv a1,a0  1021a: 0001d537 lui a0,0x1d  1021e: 27850513 addi a0,a0,632 # 1d278 <\_\_clzdi2+0x60>  10222: 00000097 auipc ra,0x0  10226: 1d6080e7 jalr ra,470(ra) # 103f8 <printf>  1022a: 4501 c.li a0,0  1022c: 60a6 c.ldsp ra,72(sp)  1022e: 6406 c.ldsp s0,64(sp)  10230: 74e2 c.ldsp s1,56(sp)  10232: 7942 c.ldsp s2,48(sp)  10234: 79a2 c.ldsp s3,40(sp)  10236: 6161 c.addi16sp sp,80  10238: 8082 c.jr ra  000000000001023a <getGCD>:  1023a: 87aa c.mv a5,a0  1023c: 852e c.mv a0,a1  1023e: 00b79663 bne a5,a1,1024a <getGCD+0x10>  10242: 8082 c.jr ra  10244: 9d1d c.subw a0,a5  10246: fea78ee3 beq a5,a0,10242 <getGCD+0x8>  1024a: fef55de3 bge a0,a5,10244 <getGCD+0xa>  1024e: 9f89 c.subw a5,a0  10250: bfdd c.j 10246 <getGCD+0xc>  0000000000010252 <getGCD\_Array>:  10252: 87aa c.mv a5,a0  10254: 4108 c.lw a0,0(a0)  10256: 4705 c.li a4,1  10258: 02b75d63 bge a4,a1,10292 <getGCD\_Array+0x40>  1025c: 1101 c.addi sp,-32  1025e: ec06 c.sdsp ra,24(sp)  10260: e822 c.sdsp s0,16(sp)  10262: e426 c.sdsp s1,8(sp)  10264: 00478413 addi s0,a5,4  10268: ffe5849b addiw s1,a1,-2  1026c: 02049713 slli a4,s1,0x20  10270: 01e75493 srli s1,a4,0x1e  10274: 07a1 c.addi a5,8  10276: 94be c.add s1,a5  10278: 400c c.lw a1,0(s0)  1027a: 00000097 auipc ra,0x0  1027e: fc0080e7 jalr ra,-64(ra) # 1023a <getGCD>  10282: 0411 c.addi s0,4  10284: fe941ae3 bne s0,s1,10278 <getGCD\_Array+0x26>  10288: 60e2 c.ldsp ra,24(sp)  1028a: 6442 c.ldsp s0,16(sp)  1028c: 64a2 c.ldsp s1,8(sp)  1028e: 6105 c.addi16sp sp,32  10290: 8082 c.jr ra  10292: 8082 c.jr ra |

Прежде всего можно видеть, что в результат компоновки попало содержимое обоих объектных файлов – main.o и gcd.o. Инструкции подпрограммы getGCD\_Array начинаются с адреса 1025216, и пара инструкций auipc+jalr, вызывающих подпрограмму getGCD\_Array соответствующим образом откорректированы, аналогично и для getGCD.

## Анализ отладочной информации

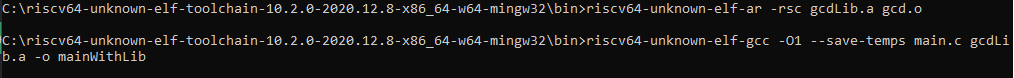


Сформированный исполняемый файл содержит информацию для отладки (в секциях ***.debug…***), полную таблицу символов и сведения о версиях средств разработки.

Встреченные разделы DWARF:

* ***.debug\_abbrev*** – сокращения , используемые в ***.debug\_info*** разделе;
* ***.debug\_aranges***– таблица поиска для сопоставления адресов с единицами компиляции;
* ***.debug\_frame*** – информация о кадре вызова;
* ***.debug\_info*** – раздел основной информации DWARF;
* ***.debug\_line*** – информация о номере строки;
* ***.debug\_loc*** – списки местоположений, используемые в атрибутах ***DW\_AT\_location***;
* ***.debug\_ranges*** – диапазоны адресов, используемые в атрибутах ***DW\_AT\_ranges***;
* ***.debug\_str*** – таблица строк, используемая в ***.debug\_info***.

## Выделение разработанной функции в статическую библиотеку

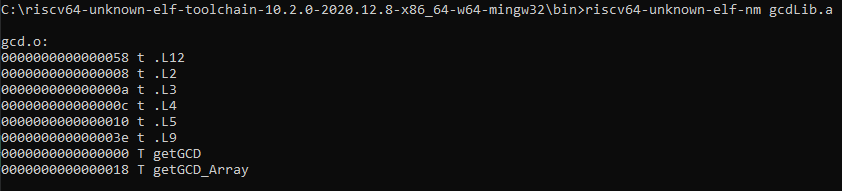


|  |
| --- |
| …  00000000000102bc g F .text 00000000000000aa memset  0000000000010156 g F .text 0000000000000088 main  …  00000000000101f6 g F .text 000000000000003e getGCD\_Array  … |

Таблица символов полученного исполняемого файла

Как и следовало ожидать, в состав исполняемого файла вошло содержимое всех объектных файлов, указанных в команде сборки.

## Создание и использование полученной статической библиотеки



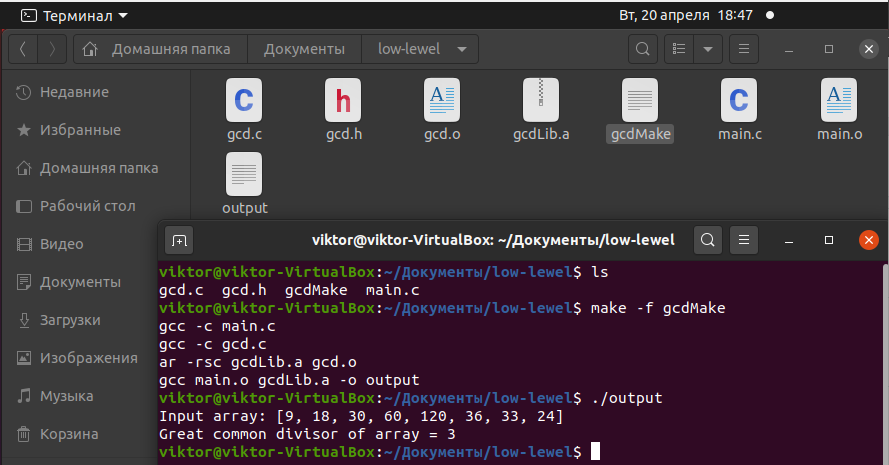
Список символов библиотеки

Создадим make-файл:

|  |
| --- |
| output: main.o gcdLib.a  gcc main.o gcdLib.a -o output  main.o: main.c  gcc -c main.c  gcdLib.a: gcd.o gcd.h  ar -rsc gcdLib.a gcd.o  gcd.o:  gcc -c gcd.c  clean:  rm \*.o \*.a output |

Что происходит в ***makefile***:

* Создаём объектный файл ***main.o*** из исходного ***main.c***;
* Создаём объектный файл ***gcd.o*** из исходного ***gcd.c***;
* Архивируем объектный файл ***gcd.o*** (создаём статическую библиотеку ***gcdLib.a***);
* Компонуем статическую библиотеку ***gcdLib.a*** с объектным файлом ***main.o***, получаем исполняемый файл ***output***.



# Результаты

В ходе работы исследован процесс сборки проекта на языке C.

Он состоит из:

• Препроцессирования исходного <filename>.c в <filename>.i;

• Компиляции полученного <filename>.i в файл ассемблера <filename>.s;

• Ассемблирования <filename>.s в объектный файл <filename>.o;

• Компоновки объектного файла <filename>.o в исполняемый файл.

Также были рассмотрены makefile’ы, которые существенно упрощают процесс сборки.

Вместо того, чтобы поочередно набирать команды в терминале, используется единственная команда make, которая по инструкциям в makefile’е собирает программу в автоматическом режиме.