1. **Формулировка задачи**

1. 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.

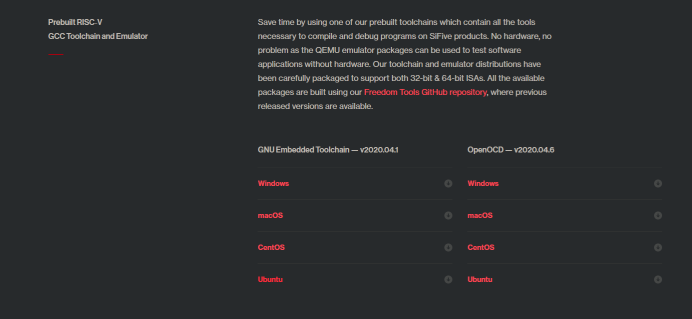
2. Собрать программу «по шагам». Проанализировать выход препроцессора и компилятора. Проанализировать состав и содержимое секций, таблицы символов, таблицы перемещений и отладочную информацию, содержащуюся в объектных файлах и исполняеммом файле.

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

1. **Реализация. Подготовка к проведению работы.**

A. Загрузка пакета “SiFive GNU Embedded Toolchain” для RISC-V.

Переходим на сайт, и в нашем случае в секции Download загружаем архив под нашу операционную систему, т.е. для Windows.



Распаковываем архив в любое место. В данной работе архив был распакован в C:\riscv64\

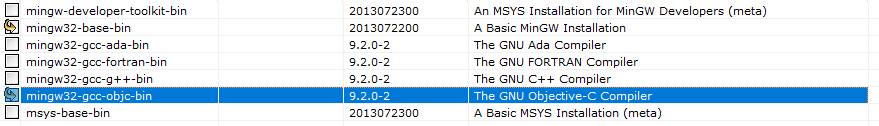
Б. Загрузка пакет “Mingw32” для использования утилиты make и GCC.

Переходим на сайт, и в нашем случае загружаем Installer для Windows.

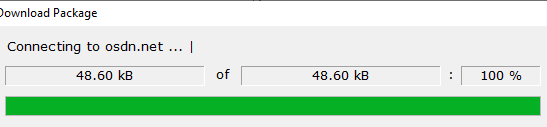
Открываем Installer, и отмечаем необходимые пакеты, в нашем случае это

Mingw32-base-bin

Mingw32-gcc-objc-bin



Устанавливаем пакеты.



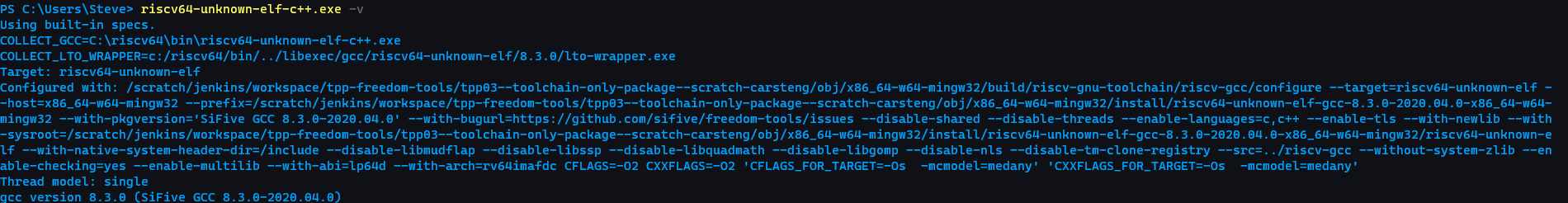
В. Добавить исполняемые файлы в Path.

Открываем утилиту переменные среды, и в системной переменной Path добавляем:

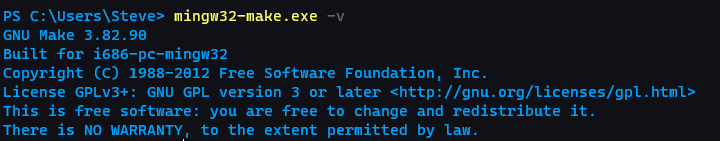


D. Проверяем работу програм:

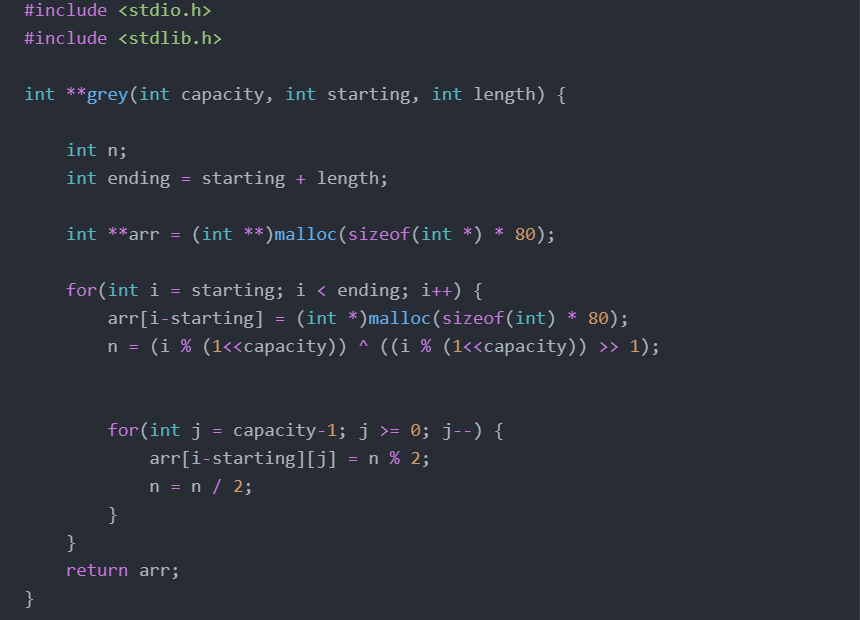
Riscv64

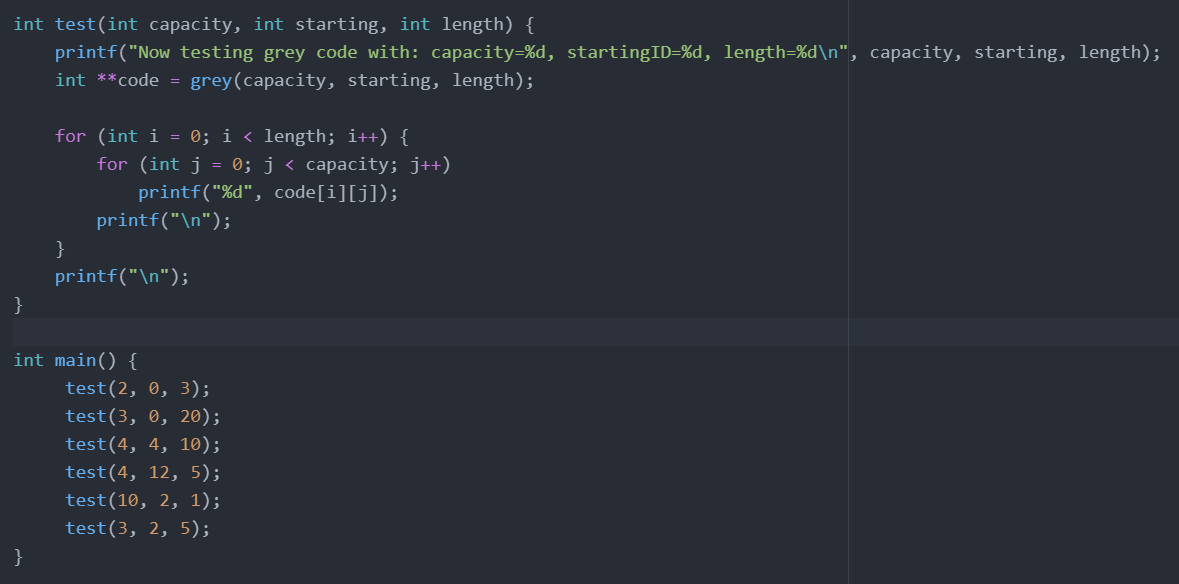


Mingw32



1. Сборка простейшей программы
   1. Текст программы





* 1. Сборка программы

Для сборки программы выполним следующую команду:



Программа *riscv64-unknown-elf-gcc* является драйвером компилятора gcc, в данном случае она запускается со следующими параметрами командной строки:

*--save-temps* – сохранять промежуточные файлы, создаваемые в процессе сборки;

*-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);

*-v* – печатать (в стандартный поток ошибок) выполняемые драйвером команды, а также дополнительную информацию.

В конце команды используется т.н. «перенаправление вывода»:

*>log* - вместо печати в консоли (обычно, это означает «на экране») вывод программы направляется в файл с именем “log” (если файл не существует, он создается; если файл существует, его содержимое будет утеряно);

*2>&1* – поток вывода ошибок (2 – стандартный «номер» этого потока) «связывается» с поток вывода («номер» 1), т.е. сообщения об ошибках (и информация, вывод которой вызван использованием флага “-v”, см.выше) также выводятся в файл “log”.

В результате исполнения приведенной команды в текущем каталоге будут созданы следующие файлы:

*a.out* – исполняемый файл, сгенерированный компоновщиком в результате сборки.

*log* – текстовый файл, содержащий сообщения компилятора, ассемблера и компоновщика, а также выполняемые команды и дополнительную информацию;

* 1. Процесс сборки, реализуемый драйвером компилятора

log

|  |
| --- |
| COLLECT\_GCC=C:\riscv64\bin\riscv64-unknown-elf-gcc-8.3.0.exe  COLLECT\_LTO\_WRAPPER=c:/riscv64/bin/../libexec/gcc/riscv64-unknown-elf/8.3.0/lto-wrapper.exe  Target: riscv64-unknown-elf  Configured with: /scratch/jenkins/workspace/tpp-freedom-tools/tpp03--toolchain-only-package--scratch-carsteng/obj/x86\_64-w64-mingw32/build/riscv-gnu-toolchain/riscv-gcc/configure --target=riscv64-unknown-elf  --host=x86\_64-w64-mingw32 --prefix=/scratch/jenkins/workspace/tpp-freedom-tools/tpp03--toolchain-only-package--scratch-carsteng/obj/x86\_64-w64-mingw32/install/riscv64-unknown-elf-gcc-8.3.0-2020.04.0-x86\_64-w6  4-mingw32 --with-pkgversion='SiFive GCC 8.3.0-2020.04.0' --with-bugurl=https://github.com/sifive/freedom-tools/issues --disable-shared --disable-threads --enable-languages=c,c++ --enable-tls --with-newlib --w  ith-sysroot=/scratch/jenkins/workspace/tpp-freedom-tools/tpp03--toolchain-only-package--scratch-carsteng/obj/x86\_64-w64-mingw32/install/riscv64-unknown-elf-gcc-8.3.0-2020.04.0-x86\_64-w64-mingw32/riscv64-unkno  wn-elf --with-native-system-header-dir=/include --disable-libmudflap --disable-libssp --disable-libquadmath --disable-libgomp --disable-nls --disable-tm-clone-registry --src=../riscv-gcc --without-system-zlib  --enable-checking=yes --enable-multilib --with-abi=lp64d --with-arch=rv64imafdc CFLAGS=-O2 CXXFLAGS=-O2 'CFLAGS\_FOR\_TARGET=-Os -mcmodel=medany''CXXFLAGS\_FOR\_TARGET=-Os -mcmodel=medany'  Thread model: single  gcc version 8.3.0 (SiFive GCC 8.3.0-2020.04.0)  COLLECT\_GCC\_OPTIONS='-save-temps''-march=rv32i''-mabi=ilp32''-O1''-v'  c:/riscv64/bin/../libexec/gcc/riscv64-unknown-elf/8.3.0/cc1.exe -E -quiet -v -imultilib rv32i/ilp32 -iprefix c:\riscv64\bin\../lib/gcc/riscv64-unknown-elf/8.3.0/ -isysroot c:\riscv64\bin\../riscv64-unknown-e  lf .\gray.c -march=rv32i -mabi=ilp32 -O1 -fpch-preprocess -o gray.i  ignoring duplicate directory "c:/riscv64/lib/gcc/../../lib/gcc/riscv64-unknown-elf/8.3.0/include"  ignoring nonexistent directory "c:\riscv64\bin\../riscv64-unknown-elf/scratch/jenkins/workspace/tpp-freedom-tools/tpp03--toolchain-only-package--scratch-carsteng/obj/x86\_64-w64-mingw32/install/riscv64-unknown  -elf-gcc-8.3.0-2020.04.0-x86\_64-w64-mingw32/lib/gcc/riscv64-unknown-elf/8.3.0/../../../../include"  ignoring duplicate directory "c:/riscv64/lib/gcc/../../lib/gcc/riscv64-unknown-elf/8.3.0/include-fixed"  ignoring duplicate directory "c:/riscv64/lib/gcc/../../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../riscv64-unknown-elf/include"  ignoring duplicate directory "c:\riscv64\bin\../riscv64-unknown-elf/include"  #include "..." search starts here:  #include <...> search starts here:  c:\riscv64\bin\../lib/gcc/riscv64-unknown-elf/8.3.0/include  c:\riscv64\bin\../lib/gcc/riscv64-unknown-elf/8.3.0/include-fixed  c:\riscv64\bin\../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../riscv64-unknown-elf/include  End of search list.  COLLECT\_GCC\_OPTIONS='-save-temps''-march=rv32i''-mabi=ilp32''-O1''-v'  c:/riscv64/bin/../libexec/gcc/riscv64-unknown-elf/8.3.0/cc1.exe -fpreprocessed gray.i -quiet -dumpbase gray.c -march=rv32i -mabi=ilp32 -auxbase gray -O1 -version -o gray.s  GNU C17 (SiFive GCC 8.3.0-2020.04.0) version 8.3.0 (riscv64-unknown-elf)  compiled by GNU C version 5.3.1 20160211, GMP version 6.1.0, MPFR version 3.1.4, MPC version 1.0.3, isl version isl-0.18-GMP  GGC heuristics: --param ggc-min-expand=30 --param ggc-min-heapsize=4096  GNU C17 (SiFive GCC 8.3.0-2020.04.0) version 8.3.0 (riscv64-unknown-elf)  compiled by GNU C version 5.3.1 20160211, GMP version 6.1.0, MPFR version 3.1.4, MPC version 1.0.3, isl version isl-0.18-GMP  GGC heuristics: --param ggc-min-expand=30 --param ggc-min-heapsize=4096  Compiler executable checksum: 1b317a4192a81762ab84201f3c177447  COLLECT\_GCC\_OPTIONS='-save-temps''-march=rv32i''-mabi=ilp32''-O1''-v'  c:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../riscv64-unknown-elf/bin/as.exe -v --traditional-format -march=rv32i -mabi=ilp32 -o gray.o gray.s  GNU assembler version 2.32 (riscv64-unknown-elf) using BFD version (SiFive Binutils 2.32.0-2020.04.0) 2.32  COMPILER\_PATH=c:/riscv64/bin/../libexec/gcc/riscv64-unknown-elf/8.3.0/;c:/riscv64/bin/../libexec/gcc/;c:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../riscv64-unknown-elf/bin/  LIBRARY\_PATH=c:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/rv32i/ilp32/;c:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../riscv64-unknown-elf/lib/rv32i/ilp32/;c:/riscv64/bin/../riscv64-unk  nown-elf/lib/rv32i/ilp32/;c:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/;c:/riscv64/bin/../lib/gcc/;c:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../riscv64-unknown-elf/lib/;c:/riscv64/bi  n/../riscv64-unknown-elf/lib/  COLLECT\_GCC\_OPTIONS='-save-temps''-march=rv32i''-mabi=ilp32''-O1''-v'  c:/riscv64/bin/../libexec/gcc/riscv64-unknown-elf/8.3.0/collect2.exe -plugin c:/riscv64/bin/../libexec/gcc/riscv64-unknown-elf/8.3.0/liblto\_plugin-0.dll -plugin-opt=c:/riscv64/bin/../libexec/gcc/riscv64-unkn  own-elf/8.3.0/lto-wrapper.exe -plugin-opt=-fresolution=gray.res -plugin-opt=-pass-through=-lgcc -plugin-opt=-pass-through=-lc -plugin-opt=-pass-through=-lgloss -plugin-opt=-pass-through=-lgcc --sysroot=c:\ris  cv64\bin\../riscv64-unknown-elf -melf32lriscv c:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../riscv64-unknown-elf/lib/rv32i/ilp32/crt0.o c:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/rv3  2i/ilp32/crtbegin.o -Lc:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/rv32i/ilp32 -Lc:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../riscv64-unknown-elf/lib/rv32i/ilp32 -Lc:/riscv64/bin/../  riscv64-unknown-elf/lib/rv32i/ilp32 -Lc:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0 -Lc:/riscv64/bin/../lib/gcc -Lc:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/../../../../riscv64-unknown-elf/lib  -Lc:/riscv64/bin/../riscv64-unknown-elf/lib gray.o -lgcc --start-group -lc -lgloss --end-group -lgcc c:/riscv64/bin/../lib/gcc/riscv64-unknown-elf/8.3.0/rv32i/ilp32/crtend.o  COLLECT\_GCC\_OPTIONS='-save-temps''-march=rv32i''-mabi=ilp32''-O1''-v' |

Исходя из вывода лог-файла, процесс сборки простейшей программы состоит из следующих шагов:

1. Запускается первая фаза сборки, т.е. программа cc1.exe с параметрами:

-E -quiet -v -imultilib rv32i/ilp32 -iprefix

Полныйсинтаксвызова:

cc1.exe -E -quiet -v -imultilib rv32i/ilp32 -iprefix

1. Запуск программы *cc1* с параметром *“-fpreprocessed”*. Исполняемая команда в упрощенном виде:

*cc1.exe-fpreprocessed gray.i -march=rv32i -mabi=ilp32 -O1 -o gray.s*

На данном шаге выполняется компиляция файла *“gray.i”*, уже обработанного препроцессором (опция “-fpreprocessed”), результат работы компилятора – код на языке ассемблера – сохраняется в файле *”\*.s”*.

1. Запуск программы *as*. Исполняемая команда в упрощенном виде:

*as.exe -v -march=rv32i -mabi=ilp32 -o gray.o gray.s*

На данном шаге выполняется ассемблирование файла *“gray.s”*, результат работы ассемблера – объектный код – сохраняется в файле *”gray.o”*.

1. Запуск программы collect2. Исполняемая команда в упрощенном виде:

*collect2.exe lib/rv32i/ilp32/crt0.o rv32i/ilp32/crtbegin.o gray.o -lgcc -lc -lgloss -lgcc rv32i/ilp32/crtend.o*

Программа collect2 является утилитой *gcc*, запускающей компоновщик. На данном шаге выполняется компоновка – формирование исполнимого файла из ранее созданных объектных файлов. Как можно видеть из команды, осуществляется компоновка объектных файлов *“crt0.o“*, *“crtbegin.o“*, *“crtend.o“*, относящихся к реализации среды времени выполнения (C runtime) и созданного на предыдущем шаге объектного файла “gray.o”. Кроме того, в компоновке могут участвовать объектные файлы из библиотек “libgcc”, “libc”, “libgloss” (опции “-l…”). Имя выходного файла не указано, и по умолчанию результат работы компоновщика записывается в файл *“a.out”.*

* 1. Выход препроцессора

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

В программе используется директива #include<stdio.h>, поэтому в самом файле gray.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.

|  |
| --- |
| # 1 ".\\gray.c"  # 1 "<built-in>"  # 1 "<command-line>"  # 1 ".\\gray.c"  # 1 "c:\\riscv64\\riscv64-unknown-elf\\include\\stdio.h" 1 3  # 29 "c:\\riscv64\\riscv64-unknown-elf\\include\\stdio.h" 3  # 1 "c:\\riscv64\\riscv64-unknown-elf\\include\\\_ansi.h" 1 3  # 10 "c:\\riscv64\\riscv64-unknown-elf\\include\\\_ansi.h" 3  # 1 "c:\\riscv64\\riscv64-unknown-elf\\include\\newlib.h" 1 3  # 14 "c:\\riscv64\\riscv64-unknown-elf\\include\\newlib.h" 3  # 1 "c:\\riscv64\\riscv64-unknown-elf\\include\\\_newlib\_version.h" 1 3  # 15 "c:\\riscv64\\riscv64-unknown-elf\\include\\newlib.h" 2 3  # 11 "c:\\riscv64\\riscv64-unknown-elf\\include\\\_ansi.h" 2 3  # 1 "c:\\riscv64\\riscv64-unknown-elf\\include\\sys\\config.h" 1 3  …  void \* aligned\_alloc(size\_t, size\_t) \_\_attribute\_\_((\_\_malloc\_\_)) \_\_attribute\_\_((\_\_alloc\_align\_\_(1)))  \_\_attribute\_\_((\_\_alloc\_size\_\_(2))) \_\_attribute\_\_((\_\_warn\_unused\_result\_\_));  int at\_quick\_exit(void (\*)(void));  \_Noreturn void  quick\_exit(int);  # 3 ".\\gray.c" 2  # 5 ".\\gray.c"  int \*\*grey(int capacity, int starting, int length) {  int n;  int ending = starting + length;  int \*\*arr = (int \*\*)malloc(sizeof(int \*) \* 80);  for(int i = starting; i < ending; i++) {  arr[i-starting] = (int \*)malloc(sizeof(int) \* 80);  n = (i % (1<<capacity)) ^ ((i % (1<<capacity)) >> 1);  for(int j = capacity-1; j >= 0; j--) {  arr[i-starting][j] = n % 2;  n = n / 2;  }  }  return arr;  }  int test(int capacity, int starting, int length) {  printf("Now testing grey code with: capacity=%d, startingID=%d, length=%d\n", capacity, starting, length);  int \*\*code = grey(capacity, starting, length);  for (int i = 0; i < length; i++) {  for (int j = 0; j < capacity; j++)  printf("%d", code[i][j]);  printf("\n");  }  printf("\n");  }  int main() {  test(2, 0, 3);  test(3, 0, 20);  test(4, 4, 10);  test(4, 12, 5);  test(10, 2, 1);  test(3, 2, 5);  } |

* 1. Выход компилятора

Компилятором сгенерирован код на языке низкого уровня – Assembly.

gray.s

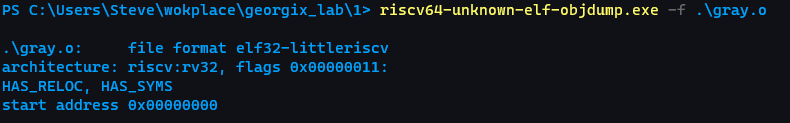
|  |
| --- |
| .file "gray.c"  .option nopic  .attribute arch, "rv32i2p0"  .attribute unaligned\_access, 0  .attribute stack\_align, 16  .text  .globl \_\_modsi3  .align 2  .globl grey  .type grey, @function  grey:  addi sp,sp,-48  sw ra,44(sp)  sw s0,40(sp)  sw s1,36(sp)  sw s2,32(sp)  sw s3,28(sp)  sw s4,24(sp)  sw s5,20(sp)  sw s6,16(sp)  sw s7,12(sp)  mv s6,a0  mv s2,a1  add s5,a1,a2  li a0,320  call malloc  mv s7,a0  bge s2,s5,.L1  li s4,1  sll s4,s4,s6  mv s3,a0  li s1,-1  j .L5  .L3:  addi s2,s2,1  addi s3,s3,4  beq s5,s2,.L1  .L5:  li a0,320  call malloc  mv s0,s3  sw a0,0(s3)  mv a1,s4  mv a0,s2  call \_\_modsi3  srai a5,a0,1  xor a0,a5,a0  addi a4,s6,-1  blt a4,zero,.L3  .L4:  lw a3,0(s0)  slli a5,a4,2  add a3,a3,a5  srli a2,a0,31  add a5,a0,a2  andi a5,a5,1  sub a5,a5,a2  sw a5,0(a3)  add a0,a2,a0  srai a0,a0,1  addi a4,a4,-1  bne a4,s1,.L4  j .L3  .L1:  mv a0,s7  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)  addi sp,sp,48  jr ra  .size grey, .-grey  .align 2  .globl test  .type test, @function  test:  addi sp,sp,-32  sw ra,28(sp)  sw s0,24(sp)  sw s1,20(sp)  sw s2,16(sp)  sw s3,12(sp)  sw s4,8(sp)  sw s5,4(sp)  mv s2,a0  mv s0,a1  mv s5,a2  mv a3,a2  mv a2,a1  mv a1,a0  lui a0,%hi(.LC0)  addi a0,a0,%lo(.LC0)  call printf  mv a2,s5  mv a1,s0  mv a0,s2  call grey  ble s5,zero,.L10  mv s1,a0  li s4,0  lui s3,%hi(.LC1)  j .L11  .L13:  li a0,10  call putchar  addi s4,s4,1  addi s1,s1,4  beq s5,s4,.L10  .L11:  li s0,0  ble s2,zero,.L13  .L12:  lw a5,0(s1)  slli a4,s0,2  add a5,a5,a4  lw a1,0(a5)  addi a0,s3,%lo(.LC1)  call printf  addi s0,s0,1  bne s2,s0,.L12  j .L13  .L10:  li a0,10  call putchar  lw ra,28(sp)  lw s0,24(sp)  lw s1,20(sp)  lw s2,16(sp)  lw s3,12(sp)  lw s4,8(sp)  lw s5,4(sp)  addi sp,sp,32  jr ra  .size test, .-test  .align 2  .globl main  .type main, @function  main:  addi sp,sp,-16  sw ra,12(sp)  li a2,3  li a1,0  li a0,2  call test  li a2,20  li a1,0  li a0,3  call test  li a2,10  li a1,4  li a0,4  call test  li a2,5  li a1,12  li a0,4  call test  li a2,1  li a1,2  li a0,10  call test  li a2,5  li a1,2  li a0,3  call test  li a0,0  lw ra,12(sp)  addi sp,sp,16  jr ra  .size main, .-main  .section .rodata.str1.4,"aMS",@progbits,1  .align 2  .LC0:  .string "Now testing grey code with: capacity=%d, startingID=%d, length=%d\n"  .zero 1  .LC1:  .string "%d"  .ident "GCC: (SiFive GCC 8.3.0-2020.04.0) 8.3.0" |

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

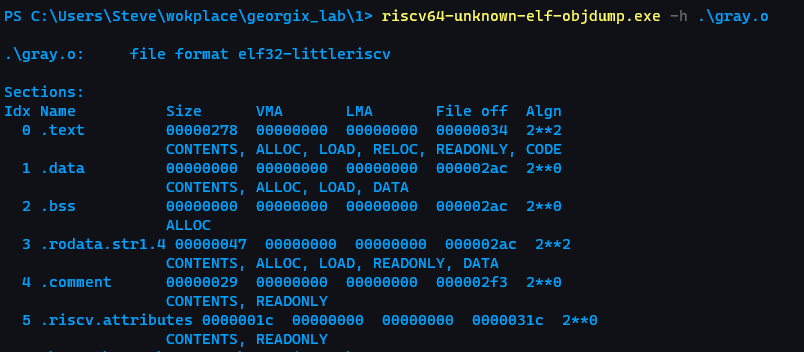
Сформированный объектный файл “gray.o” должен содержать коды инструкций, таблицу символов и таблицу перемещений. В отличие от ранее рассмотренных файлов, объектный фал не является текстовым, и не может быть напрямую выведен на экран в читаемом формате. Для изучения его содержимого используем утилиту objdump, отображающую содержимое бинарных файлов в читаемом формате.

Вывод утилиты:

Файл имеет формат ELF, является объектным файлом 32-разрядной архитектуры RISC-V, содержит символы (флаг HAS\_SYMS), содержит таблицу перемещений (флаг HAS\_RELOC). Объектный файл не должен содержать адрес точки входа (адрес, с которого начинается исполнение программы), однако поскольку соответствующее поле приутствует 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. Следующая команда обеспечивает отображение заголовков секций файла ”gray.o”:



В файле “gray*.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 таблицу символов файла:

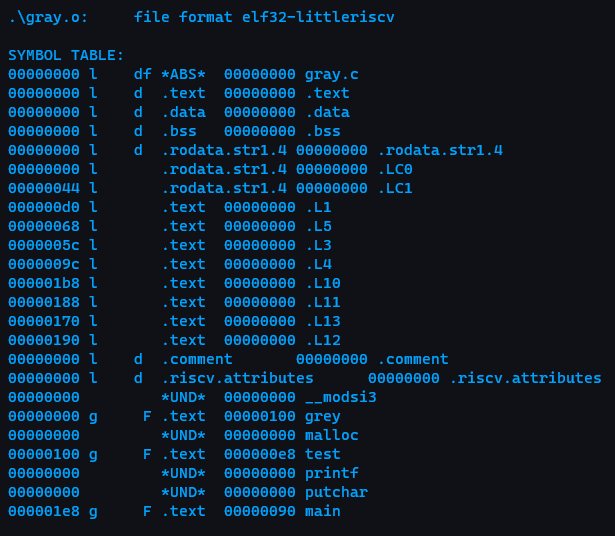


Таблица содержит 3 глобальные (флаг g) функции (флаг F) – main,grey,test, а также четыре неопределённых символа (UND).

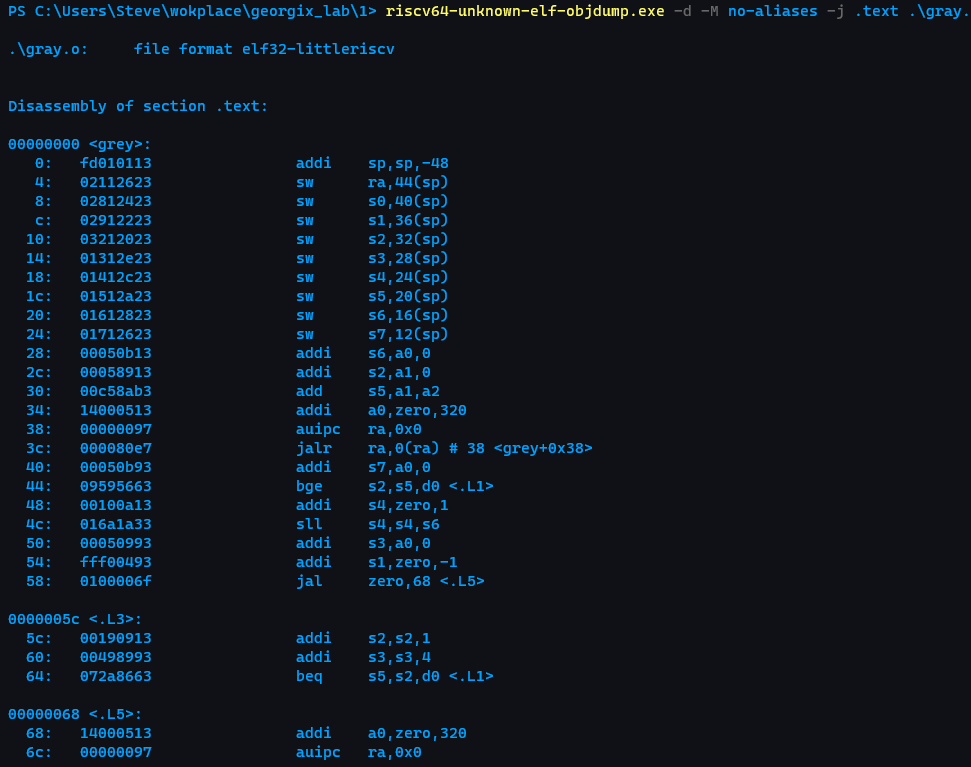
UND означает, что символы printf и т.д. использовались в ассемблерном коде, из которого был получен данный объектный фал, но не был определён;

Ассемблер сделал вывод о том, что символ должен быть определён где-то ещё, и отразил это в таблице символов.

1. Секция .text

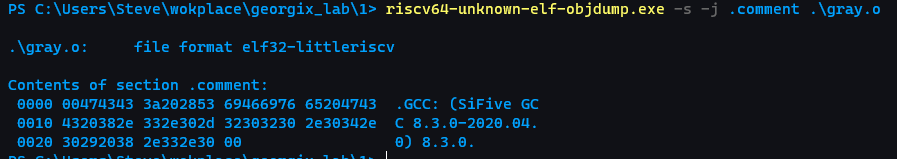
Изучим содержимое секции “.text”:

Разумеется, 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, опция *“-M no-aliases”* требует использовать в выводе только инструкции системы команд (но не псевдоинструкции ассемблера). Вывод утилиты:

1. Секция .comment

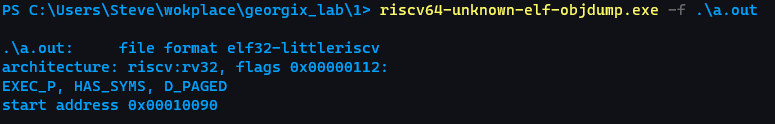
Изучим содержимое секции “.comment”:



Как и ожидалось, в секции .comment, записаны инициалы компилятора, которым проводилась процедура.

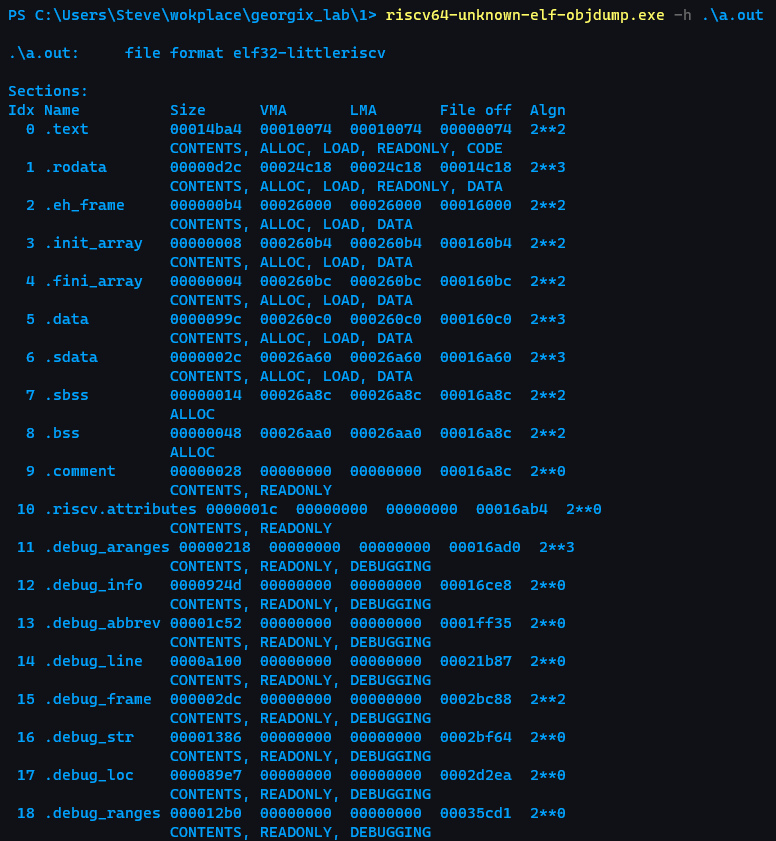
1. Вывод компоновщика – исполняемый файл
2. Общая информация

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



Видим, что формат файла – ELF. Также видим наличие флага EXEC\_P – он говорит о том, что файл исполняемый. После загрузки файла его выполнение начинается с адреса 0x00010090

Изучим секции файла:



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

1. Инструкции програмы

Изучим содержимое секции “.text”:

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

В результате выполнения получили файл *“a.ds”*. Изучим его.

|  |
| --- |
| *00010090 <\_start>:*  *10090: 00017197 auipc gp,0x17*  *10094: 83018193 addi gp,gp,-2000 # 268c0 <\_\_global\_pointer$>*  *10098: 1cc18513 addi a0,gp,460 # 26a8c <\_edata>*  *1009c: 22818613 addi a2,gp,552 # 26ae8 <\_\_BSS\_END\_\_>*  *100a0: 40a60633 sub a2,a2,a0*  *100a4: 00000593 addi a1,zero,0*  *100a8: 4b9000ef jal ra,10d60 <memset>*  *100ac: 00000517 auipc a0,0x0*  *100b0: 3d450513 addi a0,a0,980 # 10480 <\_\_libc\_fini\_array>*  *100b4: 384000ef jal ra,10438 <atexit>*  *100b8: 428000ef jal ra,104e0 <\_\_libc\_init\_array>*  *100bc: 00012503 lw a0,0(sp)*  *100c0: 00410593 addi a1,sp,4*  *100c4: 00000613 addi a2,zero,0*  *100c8: 244000ef jal ra,1030c <main>*  *100cc: 3800006f jal zero,1044c <exit>* |

Строки 16-32.

“\_start” – “точка входа” в нашу программу. Код, начинающийся с метки “\_start” обеспечивает инициализацию памяти, регистров процессора и среды времени выполнения, после чего передаёт управление определённой нами функции main.

*100c8: 244000ef jal ra,1030c <main>*

Как мы знаем, в результате выполнения этой инструкции адрес возврата (значение pc+4) заносится в регистр ra (x1), после чего управление передается на адрес 1030с16. В угловых скобках показано, что этот адрес соответствует символу *“main”*, что также видно из таблицы символов. Таким образом, действительно, данная инструкция вызывает функцию *main*.

|  |
| --- |
| 0001044c <exit>:  1044c: ff010113 addi sp,sp,-16  10450: 00000593 addi a1,zero,0  10454: 00812423 sw s0,8(sp)  10458: 00112623 sw ra,12(sp)  1045c: 00050413 addi s0,a0,0  10460: 389030ef jal ra,13fe8 <\_\_call\_exitprocs>  10464: 1b818793 addi a5,gp,440 # 26a78 <\_global\_impure\_ptr>  10468: 0007a503 lw a0,0(a5)  1046c: 03c52783 lw a5,60(a0)  10470: 00078463 beq a5,zero,10478 <exit+0x2c>  10474: 000780e7 jalr ra,0(a5)  10478: 00040513 addi a0,s0,0  1047c: 6800f0ef jal ra,1fafc <\_exit> |

Строки 117-130.

Можно видеть, что в конце *“exit”* управление передается на символ *“\_exit”*.

2 Сборка простейшей программы «по шагам»

Для выполнения отдельных шагов мы будем по-прежнему запускать драйвер компилятора, и контролировать его действия, используя флаг “-v”.

А. Препроцессирование

Первым шагом является препроцессирование файла исходного текста (файла “gray.c”), результат записывается в файл “gray.i”:

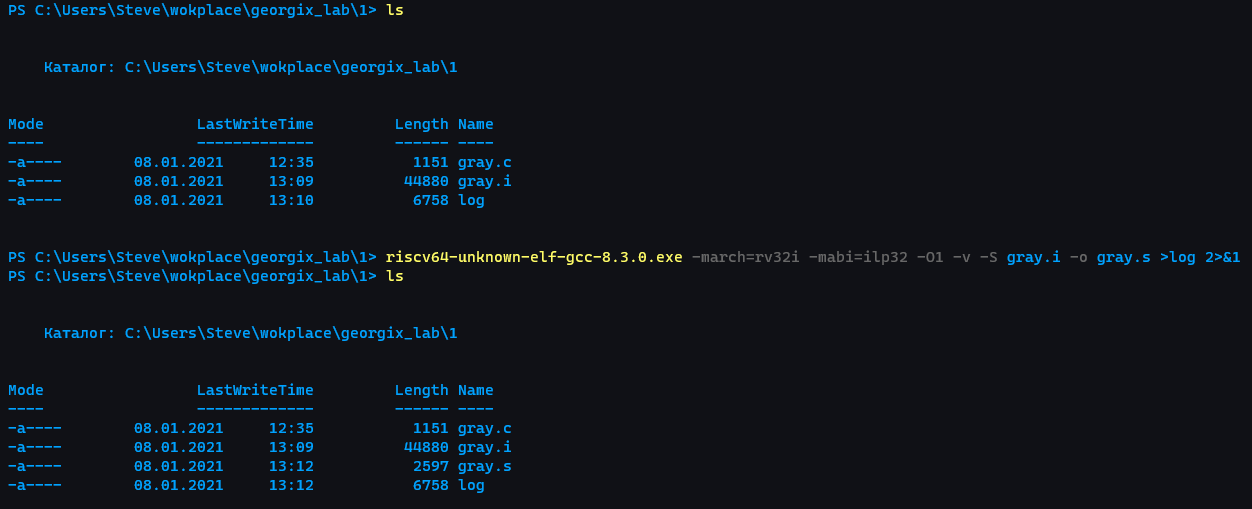


Параметры:

* *-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)
* *-v* – печатать (в стандартный поток ошибок) выполняемые драйвером команды, а также дополнительную информацию
* *-o* – путь к выходному файлу
* *-E* – остановить процесс сборки после препроцессирования

B. Компиляция

Далее необходимо выполнить компиляцию файла “gray.i”, сохранив результат – сгенерированный код на языке ассемблера – в файл “gray.s”.



Параметры:

* *-S* – остановить процесс сборки после компиляции, не запуская ассемблер
* *-fpreprocessed* – выполняется компиляция файла, уже обработанного препроцессором
* все остальные параметры из прошлого пункта означают тоже самое

C. Ассемблирование

Ассемблирование файла “gray.s” выполняется по следующей команде:



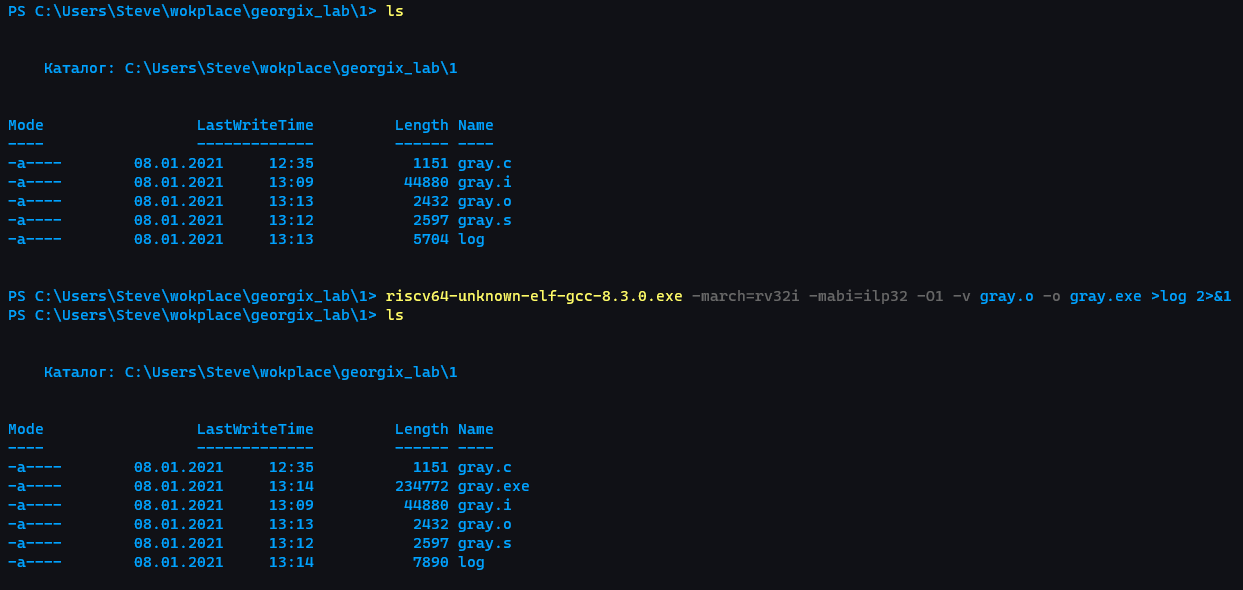
Параметры:

* *-c* – остановить процесс сборки после ассемблирования

На выходе получили объектный файл *gray.o*

D. Компоновка

Компоновка программы выполняется по следующей команде:



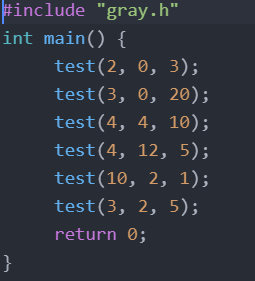
Результатом является исполняемый файл *“gray”* (имя выходного файла указано явно опцией “-o”, если бы она отсутствовала, использовалось имя файла по умолчанию *“a.out”*).

3. Раздельная компиляция

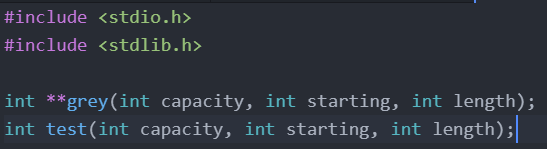
a. Текст программы

Разобьём исходную программу на 3 файла.

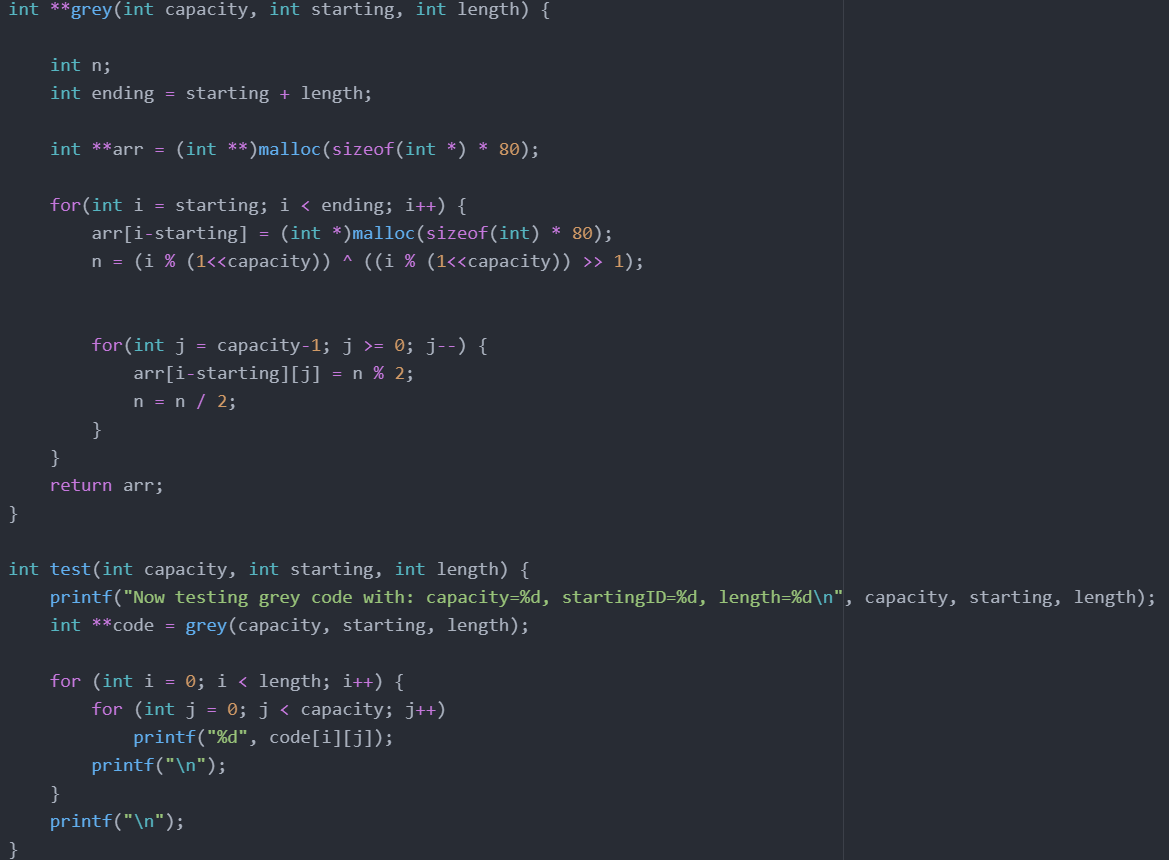
main.c



gray.h



gray.c



b. Сборка программы

Сборка программы осуществляется следующей командой:



Ранее препроцессирование, компиляция и ассемблирование выполнялось нами по шагам, но на практике это требуется редко, обычно необходимо выполнить все стадии обработки исходного файла, получив в результате объектный файл.



Параметры:

* *“-c”* - приводит к останову процесса сборки после ассемблирования, т.е. после формирования объектного файла

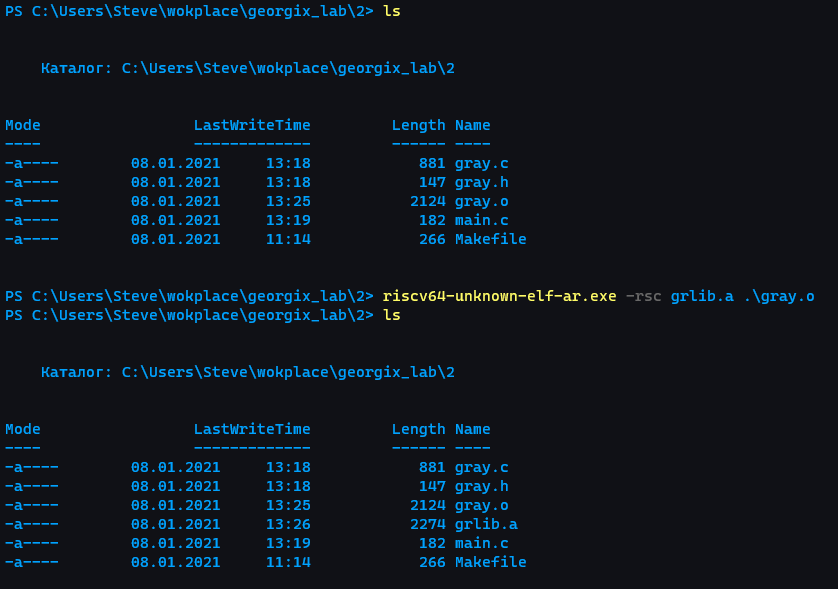
После создания объектного файла его нужно скомпоновать с нашей программой



*d. Создание и использование 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.

Поместим gray.o в такой архив:

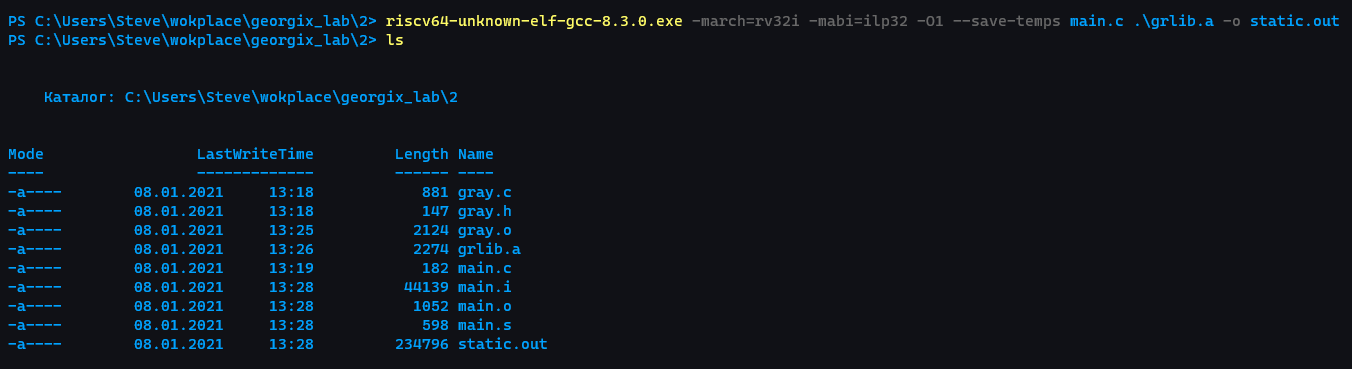


Параметры:

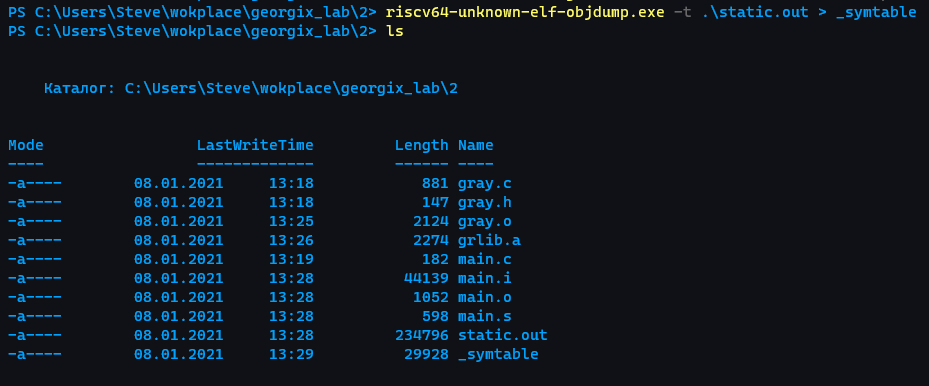
* -r – заменить старые файлы с такими названиями (*gray.o*), если они уже есть в архиве
* -s – записать «index» в архив. Index – это список всех символов, объявленных во включенных в архив объектных файлах, и его присутствие ускоряет линковку
* -с – создать архив, если его еще не было

Результирующим файлом является “grlib.a” (“.a” – от “archive”).

Используем статическую библиотеку для сборки программ:



Изучим таблицы символов полученных исполняемых файлов:

\_symtable

|  |
| --- |
| …  0001d4b0 g F .text 000000c8 \_calloc\_r  00026a8c g .sbss 00000000 \_\_bss\_start  00010d60 g F .text 000000dc memset  00010144 g F .text 00000078 main  00026a8c g O .sbss 00000004 \_\_malloc\_max\_total\_mem  00013dd0 g F .text 00000014 \_\_swbuf  00019140 g F .text 00000008 \_\_sclose  …  00026a98 g O .sbss 00000004 \_PathLocale  00010438 g F .text 00000014 atexit  0001d450 g F .text 00000060 \_write\_r  000101bc g F .text 000000f4 grey  00017900 g F .text 00000014 setlocale  00026a80 g O .sdata 00000004 \_impure\_ptr  00014104 g F .text 0000025c \_\_sflush\_r… |

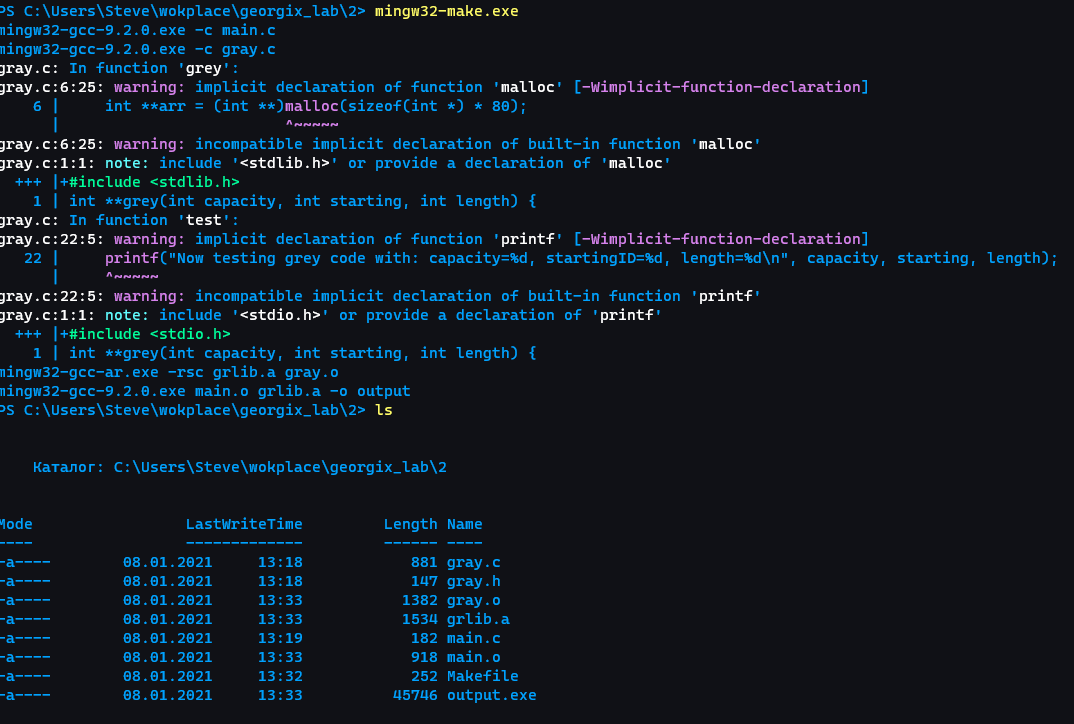
Видим, что все нужные символы вошли в исполняемый файл.

Несмотря на то, что в нашем случае компоновщик не используется, преимущества использования библиотеки очевидны: при компоновке были использованы необходимые объектные файлы и только они, причем задача выбора необходимых для сборки объектных файлов была возложена на компоновщик (а не нас).

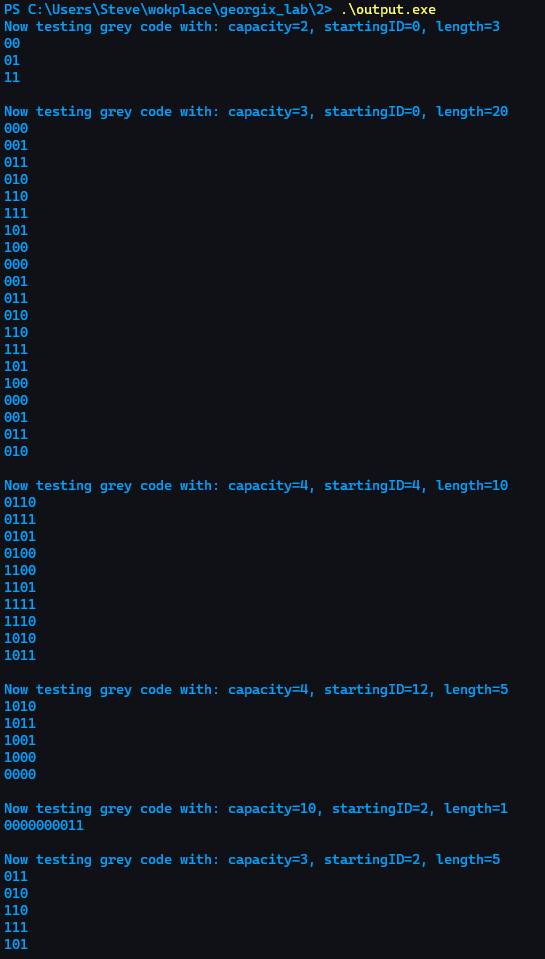
3. Makefile

Makefile - это набор инструкций для программы make, которая позволяет собирать проекты, состоящие из большого числа “\*.c” и “\*.h” файлов. Обычно эта программа используется в связке с системами сборки, например cmake, позволяя вести проекты модульно (т.е. проект с включенными подпроектами).

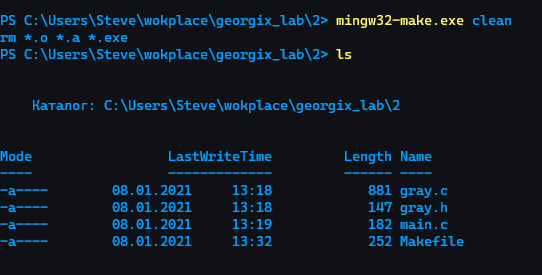
Сборка с помощью Makefile:



Демонстрация работы програмы:



Цель для Make – очистить:



*Makefile*

|  |
| --- |
| *output: main.o grlib.a*  *mingw32-gcc-9.2.0.exe main.o grlib.a -o output*  *main.o: main.c*  *mingw32-gcc-9.2.0.exe -c main.c*  *grlib.a: gray.o gray.h*  *mingw32-gcc-ar.exe -rsc grlib.a gray.o*  *gray.o:*  *mingw32-gcc-9.2.0.exe -c gray.c*  *clean:*  *rm \*.o \*.a \*.exe* |

Что происходит в Makefile:

1. Создаём объектный файл *main.o* из исходного *main.c*
2. Создаём объектный файл *gray.o* из исходного *gray.c*
3. Архивируем объектный файл gray.o (создаём статическую библиотеку *grlib.a*)
4. Компонуем статическую библиотеку *grlib.a* с объектным файлом *main.o* и получаем исполняемый файл *output*

4. Вывод

В данной лабораторной работе мы познакомились с процессом сборки проекта на языке C.

Он состоит из:

1. **Препроцессирования**: исходного *.c* файл препроцессируем в *.i* файл
2. **Компиляции**: полученный *.i*файл компилируется в файл ассемблера *.s*
3. **Ассемблирования**: файл *.s* асссемблируется в объектный файл *.o*
4. **Компоновки**: объектный файл *.o*компонуется в исполняемый файл

Также мы ознакомились в *makefile’ами,* которые упрощают процесс сборки.

Утилита Make позволяет собирать проекты, состоящие из большого количества файлов, вместо использования PS/SH скриптов, и прописывания файлов вручную.