Средства разработки

В примерах будут использованы средства разработки "SiFive GNU Embedded Toolchain", доступные для загрузки (для Windows, macOS или Linux) по следующей ссылке: https://www.sifive.com/products/tools/.

Следует отметить, что каждый «пакет средств разработки» (используются термины **toolchain**, **development kit** и др.) имеет свои особенности, однако, в общем и целом, процесс сборки (препроцессирования, компиляции, ассемблирования и компоновки) программ одинаков во всех пакетах.

1. Сборка простейшей программы

Текст программы

В данном примере рассматривается следующая простейшая программа на языке C, которая «ничего не делает» - результатом ее выполнения является формирование кода завершения 0:

```
// Файл main.c

int main( void ) {
    return 0;
}
```

Сборка программы

Для сборки (**building**) программы выполним следующую команду:

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

Программа `riscv64-unknown-elf-gcc` является драйвером компилятора gcc (**compiler driver**), в данном случае она запускается со следующими параметрами командной строки (**command line arguments**)[1]:

`--save-temps` – сохранять промежуточные (**intermediate**, **temporary**) 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» (**output redirection**):

`>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);

[1] См. https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html.

`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:

`main.i` – текстовый файл, содержащий результат препроцессирования файла “`main.c`” – единица трансляции (**translation unit**) в терминах стандарта языка C;

`main.s` – текстовый файл, содержащий код на языке ассемблера, сгенерированный компилятором в результате обработки файла “`main.i`”;

`main.o` – объектный файл (**object file**), сгенерированный ассемблером в результате обработки файла “`main.s`”;

`a.out` – исполняемый файл (**executable file**), сгенерированный компоновщиком в результате обработки файла “`main.o`”, а также других объектных файлов и файлов библиотек, входящих в состав пакета средств разработки.

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

Процесс сборки, реализуемый драйвером компилятора

Прежде всего, изучим содержимое файла “`log`”:

```
Using built-in specs.
COLLECT_GCC=C:\riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32\bin\riscv64-unknown-elf-gcc.EXE
COLLECT_LTO_WRAPPER=c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../libexec/gcc/riscv64-unknown-elf/7.2.0/lto-
     wrapper.exe
Target: riscv64-unknown-elf
Configured with: /scratch/palmer/work/20170105-
     toolchain_release/riscv-binary-tools/obj/x86_64-w64-
     mingw32/build/riscv-gnu-toolchain/riscv-gcc/configure --
     target=riscv64-unknown-elf --host=x86_64-w64-mingw32 --
     prefix=/scratch/palmer/work/20170105-toolchain_release/riscv-
     binary-tools/obj/x86_64-w64-mingw32/install/riscv64-unknown-elf-
     gcc-20171231-x86_64-w64-mingw32 --disable-shared --disable-
     threads --enable-languages=c,c++ --without-system-zlib --enable-
     tls --with-newlib --with-sysroot=/scratch/palmer/work/20170105-
     toolchain_release/riscv-binary-tools/obj/x86_64-w64-
     mingw32/install/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/riscv64-unknown-elf --with-native-system-header-
     dir=/include --disable-libmudflap --disable-libssp --disable-
     libquadmath --disable-libgomp --disable-nls --src=.././riscv-gcc
     --enable-checking=yes --enable-multilib --with-abi=lp64d --with-
     arch=rv32imafdc 'CFLAGS_FOR_TARGET=-Os  -mcmodel=medany'
Thread model: single
gcc version 7.2.0 (GCC)
COLLECT_GCC_OPTIONS='-save-temps' '-O1' '-v' '-march=rv32i'
     '-mabi=ilp32'
 c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../libexec/gcc/riscv64-unknown-
```

```
      elf/7.2.0/cc1.exe -E -quiet -v -imultilib rv32i/ilp32 -iprefix
      C:\riscv64-unknown-elf-gcc-20171231-x86_64-w64-
      mingw32\bin\../lib/gcc/riscv64-unknown-elf/7.2.0/ -isysroot
      C:\riscv64-unknown-elf-gcc-20171231-x86_64-w64-
      mingw32\bin\../riscv64-unknown-elf
      main.c -march=rv32i -mabi=ilp32 -O1 -fpch-preprocess
      -o main.i
ignoring duplicate directory "c:/riscv64-unknown-elf-gcc-20171231-
      x86_64-w64-mingw32/lib/gcc/../../lib/gcc/riscv64-unknown-
      elf/7.2.0/include"
ignoring nonexistent directory "C:\riscv64-unknown-elf-gcc-20171231-
      x86_64-w64-mingw32\bin\../riscv64-unknown-
      elf/scratch/palmer/work/20170105-toolchain_release/riscv-binary-
      tools/obj/x86_64-w64-mingw32/install/riscv64-unknown-elf-gcc-
      20171231-x86_64-w64-mingw32/lib/gcc/riscv64-unknown-
      elf/7.2.0/../../../../include"
ignoring duplicate directory "c:/riscv64-unknown-elf-gcc-20171231-
      x86_64-w64-mingw32/lib/gcc/../../lib/gcc/riscv64-unknown-
      elf/7.2.0/include-fixed"
ignoring duplicate directory "c:/riscv64-unknown-elf-gcc-20171231-
      x86_64-w64-mingw32/lib/gcc/../../lib/gcc/riscv64-unknown-
      elf/7.2.0/../../../../riscv64-unknown-elf/include"
ignoring duplicate directory "C:\riscv64-unknown-elf-gcc-20171231-
      x86_64-w64-mingw32\bin\../riscv64-unknown-elf/include"
#include "..." search starts here:
#include <...> search starts here:
 C:\riscv64-unknown-elf-gcc-20171231-x86_64-w64-
      mingw32\bin\../lib/gcc/riscv64-unknown-elf/7.2.0/include
 C:\riscv64-unknown-elf-gcc-20171231-x86_64-w64-
      mingw32\bin\../lib/gcc/riscv64-unknown-elf/7.2.0/include-fixed
 C:\riscv64-unknown-elf-gcc-20171231-x86_64-w64-
      mingw32\bin\../lib/gcc/riscv64-unknown-
      elf/7.2.0/../../../../riscv64-unknown-elf/include
End of search list.
COLLECT_GCC_OPTIONS='-save-temps' '-O1' '-v' '-march=rv64i'
      '-mabi=ilp32'
 c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
      mingw32/bin/../libexec/gcc/riscv64-unknown-
      elf/7.2.0/cc1.exe -fpreprocessed main.i -quiet -dumpbase
      main.c -march=rv32i -mabi=ilp32 -auxbase main -O1 -version -o
      main.s
GNU C11 (GCC) version 7.2.0 (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-0.16.1-GMP

GGC heuristics: --param ggc-min-expand=30 --param ggc-min-
      heapsize=4096
GNU C11 (GCC) version 7.2.0 (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-0.16.1-GMP
```

```
GGC heuristics: --param ggc-min-expand=30 --param ggc-min-
     heapsize=4096
Compiler executable checksum: 79e4b0fb68538700b5f2dcae65ffda99
COLLECT_GCC_OPTIONS='-save-temps' '-O1' '-v' '-march=rv32i'
     '-mabi=ilp32'
 c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../lib/gcc/riscv64-unknown-
     elf/7.2.0/../../../../riscv64-unknown-
     elf/bin/as.exe -v -traditional-format -march=rv32i
     -mabi=ilp32 -o main.o main.s
GNU assembler version 2.29 (riscv64-unknown-elf) using BFD version
     (GNU Binutils) 2.29
COMPILER_PATH=c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../libexec/gcc/riscv64-unknown-elf/7.2.0/;c:/riscv64-
     unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../libexec/gcc/;c:/riscv64-unknown-elf-gcc-20171231-
     x86_64-w64-mingw32/bin/../lib/gcc/riscv64-unknown-
     elf/7.2.0/../../../../riscv64-unknown-elf/bin/
LIBRARY_PATH=c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../lib/gcc/riscv64-unknown-
     elf/7.2.0/rv32i/ilp32/;c:/riscv64-unknown-elf-gcc-20171231-
     x86_64-w64-mingw32/bin/../lib/gcc/riscv64-unknown-
     elf/7.2.0/../../../../riscv64-unknown-
     elf/lib/rv32i/ilp32/;c:/riscv64-unknown-elf-gcc-20171231-x86_64-
     w64-mingw32/bin/../riscv64-unknown-
     elf/lib/rv32i/ilp32/;c:/riscv64-unknown-elf-gcc-20171231-x86_64-
     w64-mingw32/bin/../lib/gcc/riscv64-unknown-elf/7.2.0/;c:/riscv64-
     unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../lib/gcc/;c:/riscv64-unknown-elf-gcc-20171231-
     x86_64-w64-mingw32/bin/../lib/gcc/riscv64-unknown-
     elf/7.2.0/../../../../riscv64-unknown-elf/lib/;c:/riscv64-
     unknown-elf-gcc-20171231-x86_64-w64-mingw32/bin/../riscv64-
     unknown-elf/lib/
COLLECT_GCC_OPTIONS='-save-temps' '-O1' '-v' '-march=rv32i'
     '-mabi=ilp32'
 c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../libexec/gcc/riscv64-unknown-elf/7.2.0/collect2.exe
     -plugin c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../libexec/gcc/riscv64-unknown-
     elf/7.2.0/liblto_plugin-0.dll -plugin-opt=c:/riscv64-unknown-elf-
     gcc-20171231-x86_64-w64-mingw32/bin/../libexec/gcc/riscv64-
     unknown-elf/7.2.0/lto-wrapper.exe -plugin-opt=-
     fresolution=main.res -plugin-opt=-pass-through=-lgcc -plugin-
     opt=-pass-through=-lc -plugin-opt=-pass-through=-lgloss -plugin-
     opt=-pass-through=-lgcc --sysroot=C:\riscv64-unknown-elf-gcc-
     20171231-x86_64-w64-mingw32\bin\../riscv64-unknown-elf -
     melf32lriscv c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
     mingw32/bin/../lib/gcc/riscv64-unknown-
     elf/7.2.0/../../../../riscv64-unknown-elf/lib/rv32i/ilp32/crt0.o
```

```
    c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
    mingw32/bin/../lib/gcc/riscv64-unknown-
    elf/7.2.0/rv32i/ilp32/crtbegin.o -Lc:/riscv64-unknown-elf-gcc-
    20171231-x86_64-w64-mingw32/bin/../lib/gcc/riscv64-unknown-
    elf/7.2.0/rv32i/ilp32 -Lc:/riscv64-unknown-elf-gcc-20171231-
    x86_64-w64-mingw32/bin/../lib/gcc/riscv64-unknown-
    elf/7.2.0/../../../../riscv64-unknown-elf/lib/rv32i/ilp32 -
    Lc:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
    mingw32/bin/../riscv64-unknown-elf/lib/rv32i/ilp32 -Lc:/riscv64-
    unknown-elf-gcc-20171231-x86_64-w64-
    mingw32/bin/../lib/gcc/riscv64-unknown-elf/7.2.0 -Lc:/riscv64-
    unknown-elf-gcc-20171231-x86_64-w64-mingw32/bin/../lib/gcc -
    Lc:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
    mingw32/bin/../lib/gcc/riscv64-unknown-
    elf/7.2.0/../../../../riscv64-unknown-elf/lib -Lc:/riscv64-
    unknown-elf-gcc-20171231-x86_64-w64-mingw32/bin/../riscv64-
    unknown-elf/lib main.o -lgcc --start-group -lc -lgloss --end-
    group -lgcc c:/riscv64-unknown-elf-gcc-20171231-x86_64-w64-
    mingw32/bin/../lib/gcc/riscv64-unknown-
    elf/7.2.0/rv32i/ilp32/crtend.o
COLLECT_GCC_OPTIONS='-save-temps' '-O1' '-v' '-march=rv32i'
    '-mabi=ilp32'
```

Как можно видеть, процесс сборки простейшей программы состоит из следующих шагов:

1) Запуск программы `cc1` с параметром "`-E`". Исполняемая команда в упрощенном виде:
   `cc1.exe -E -v main.c -march=rv32i -mabi=ilp32 -O1 -o main.i`
   На данном шаге выполняется обработка файла исходного текста "`main.c`" *только* препроцессором (опция "`-E`"), результат сохраняется в файле "`main.i`" (параметр "`-o`").
2) Запуск программы `cc1` с параметром "`-fpreprocessed`". Исполняемая команда в упрощенном виде:
   ```
   cc1.exe -fpreprocessed main.i -march=rv32i -mabi=ilp32 -O1
     -o main.s
   ```
   На данном шаге выполняется компиляция файла "`main.i`", уже обработанного препроцессором (опция "`-fpreprocessed`"), результат работы компилятора – код на языке ассемблера – сохраняется в файле "`main.s`".
3) Запуск программы `as`. Исполняемая команда в упрощенном виде:
   `as.exe -v -march=rv32i -mabi=ilp32 -o main.o main.s`
   На данном шаге выполняется ассемблирование файла "`main.s`", результат работы ассемблера – объектный код – сохраняется в файле "`main.o`".
4) Запуск программы `collect2`. Исполняемая команда в упрощенном виде:
   ```
   collect2.exe lib/rv32i/ilp32/crt0.o rv32i/ilp32/crtbegin.o main.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 "`main.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”.

Выход препроцессора

Поскольку в данном примере не использовались директивы препроцессора (начинающиеся в языке С с символа “#”), результат работы препроцессора (файл “main.i”) мало отличается от исходного файла “main.c”:

```
# 1 "main.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "main.c"


int main( void ) {
    return 0;
}
```

Появившиеся *нестандартные* директивы, начинающиеся с символа “#”, используются для передачи информации об исходном тексте из препроцессора в компилятор; например, последняя директива «# 1 "main.c"» информирует компилятор о том, что следующая строка является результатом обработки строки 1 исходного файла “main.c”. Заметим, что в выходе препроцессора отсутствует комментарий, присутствовавший в исходном файле.

Выход компилятора

Сгенерированный компилятором код на языке ассемблера в дополнительных комментариях не нуждается:

```
      .file "main.c"
      .option nopic
      .text
      .align     2
      .globl     main
      .type main, @function
main:
      li    a0,0
      ret
      .size main, .-main
      .ident     "GCC: (GNU) 7.2.0"
```

Выход ассемблера – объектный файл

Общая информация

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
```

Вывод утилиты:

```
main.o:     file format elf32-littleriscv
architecture: riscv:rv32, flags 0x00000010:
HAS_SYMS
start address 0x00000000
```

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**). Следующая команда обеспечивает отображение заголовков секций файла “`main.o`”:

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

Вывод утилиты:

```
main.o:     file format elf32-littleriscv

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000008  00000000  00000000  00000034  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .data         00000000  00000000  00000000  0000003c  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  0000003c  2**0
                  ALLOC
  3 .comment      00000012  00000000  00000000  0000003c  2**0
                  CONTENTS, READONLY
```

В файле “`main.o`” имеются следующие секции:

- `.text` – секция кода, в которой содержатся коды инструкций (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 12 байт.

Как можно видеть из колонки “`Size`” (значения в ней приведены в 16-й системе счисления), размер секции “.text” – 8 байт, размер секции “.comment” – 18 байт.

Теперь изучим таблицу символов файла:

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

Вывод утилиты:

```
main.o:     file format elf32-littleriscv

SYMBOL TABLE:
00000000 l    df *ABS*     00000000 main.c
00000000 l    d  .text     00000000 .text
00000000 l    d  .data     00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .comment  00000000 .comment
00000000 g     F .text     00000008 main
```

Как и следовало ожидать, таблица содержит один глобальный (флаг “g”) символ типа «функция» (“F”) – символ “`main`”.

Инструкции программы – секция .text

Изучим содержимое секции “`.text`” :

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

Вывод утилиты:

```
main.o:     file format elf32-littleriscv

Contents of section .text:
 0000 13050000 67800000                    ....g...
```

Как мы уже видели, секция содержит 8 байт. Учитывая, что целевой архитектурой является RV32I (без каких-либо расширений), в которой инструкция всегда имеет размер 32 разряда – 4 байта, это соответствует 2 инструкциям. Обращаясь к приведенному выше коду на языке ассемблера, сформированному компилятором, можно видеть, что подпрограмма main реализуется двумя *псевдоинструкциями* ассемблера – “`li`” (загрузка константы, в данном случае константы 0) и “`ret`” (возврат из подпрограммы). Известно, что псевдоинструкция “`ret`” соответствует одной инструкции “`jalr`” процессора (см. презентацию «RISC-V Assembler and ABI Basics»). Можно также ожидать, что загрузка константы 0 в регистр может быть реализована в архитектуре RISC-V одной инструкцией. Таким образом, можно констатировать, что размер секции “.text” соответствует нашим ожиданиям.

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” (убедитесь, что Вы можете сделать это самостоятельно!). Учитывая, что RISC-V является **little-endian** архитектурой (и вывод objdump нам об этом постоянно напоминает), разрядные сетки инструкций имеют следующий вид:

31 24	23 16	15 8	7 0
00_{16}	00_{16}	05_{16}	13_{16}
00_{16}	00_{16}	80_{16}	67_{16}

Прежде всего, определим значение поля `opcode`:

31 24	23 16	7	6 0
–	–	–	opcode
00_{16}	00_{16}	0	0010011
00_{16}	00_{16}	0	1100111

Сверившись со сводной таблицей инструкций RISC-V, определяем, что первая инструкция является одной из вычислительных инструкций[2] формата “I-type”, а вторая – инструкцией JALR, которая также использует формат “I-type”. Установив формат инструкции, легко завершить ее декодирование:

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
000000000000	00000	000	01010	0010011
000000000000	00001	000	00000	1100111

Теперь мы можем записать инструкции программы на языке ассемблера:

```
addi x10, x0, 0
jalr x0, 0(x1)
```

Перепишем инструкции, используя обозначения регистров, принятые в ABI (см. презентацию «RISC-V Assembler and ABI Basics»)

```
addi a0, zero, 0  ; a0 = 0 – возвращаемое значение функции main
jalr zero, 0(ra)  ; возврат из подпрограммы – переход адрес,
                  ; записанный вызывающей подпрограммой в регистр ra
```

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

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

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

```
main.o:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <main>:
   0: 00000513              addi  a0,zero,0
   4: 00008067              jalr  zero,0(ra)
```

Секция .comment

Изучим содержимое секции “`.comment`” :

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

Вывод утилиты:

```
main.o:     file format elf32-littleriscv
```

[2] В колонке `opcode` сводной таблицы “RV32I Base Instruction Set” значение “0010011” встречается в следующих строках:ADDI, SLTI(U),XORI, ORI, ANDI, SLLI, SRLI, SRAI.

```
Contents of section .comment:
 0000 00474343 3a202847 4e552920 372e322e  .GCC: (GNU) 7.2.
 0010 3000                                 0.
```

Можно видеть, что секция содержит строку, указанную в директиве “.ident” ассемблера (см. “main.s” выше).

Выход компоновщика – исполняемый файл

Общая информация

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
```

Вывод утилиты:

```
a.out:     file format elf32-littleriscv
architecture: riscv:rv32, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00010074
```

Можно видеть, что файл имеет формат ELF и действительно является исполняемым (флаг “EXEC_P”), после загрузки его выполнение должно начаться с адреса 0x10074 (**entry point**).

Далее изучим секции файла:

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

Вывод утилиты:

```
a.out:     file format elf32-littleriscv

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000b38  00010074  00010074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .eh_frame     00000004  00010bac  00010bac  00000bac  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .init_array   00000004  00011000  00011000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .fini_array   00000004  00011004  00011004  00001004  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .data         00000428  00011008  00011008  00001008  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  5 .sdata        0000000c  00011430  00011430  00001430  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .sbss         0000000c  0001143c  0001143c  0000143c  2**2
                  ALLOC
  7 .bss          0000001c  00011448  00011448  0000143c  2**2
                  ALLOC
  8 .comment      0000001a  00000000  00000000  0000143c  2**0
                  CONTENTS, READONLY
```

```
  9 .debug_aranges 00000020  00000000  00000000  00001458  2**3
                   CONTENTS, READONLY, DEBUGGING
 10 .debug_info   00000026  00000000  00000000  00001478  2**0
                   CONTENTS, READONLY, DEBUGGING
 11 .debug_abbrev 00000014  00000000  00000000  0000149e  2**0
                   CONTENTS, READONLY, DEBUGGING
 12 .debug_line   000000bb  00000000  00000000  000014b2  2**0
                   CONTENTS, READONLY, DEBUGGING
 13 .debug_str    000000a8  00000000  00000000  0000156d  2**0
                   CONTENTS, READONLY, DEBUGGING
```

Сразу можно заметить, что состав секций “`a.out`” значительно расширен по сравнению с “`main.o`”. Следует также обратить внимание на то, что размеры секций “`.text`”, “`.data`”, “`.bss`” (и “`.comment`”) увеличились. Откуда в исполняемом файле «появился» дополнительный код и данные? (*Можете ли Вы ответить на этот вопрос?*) Ответ очевиден – они могли «появиться» только из других объектных файлов, переданных на вход компоновщика (см. упрощенную команду запуска компоновщика выше).

Таблица символов после компоновки также значительно расширилась:

```
riscv64-unknown-elf-objdump –t a.out
```

Вывод утилиты:

```
a.out:     file format elf32-littleriscv

SYMBOL TABLE:
00010074 l    d  .text     00000000 .text
000105e0 l    d  .eh_frame 00000000 .eh_frame
000115e4 l    d  .init_array    00000000 .init_array
000115e8 l    d  .fini_array    00000000 .fini_array
000115f0 l    d  .data     00000000 .data
00011a18 l    d  .sdata    00000000 .sdata
00011a24 l    d  .bss 00000000 .bss
00000000 l    d  .comment  00000000 .comment
00000000 l    d  .debug_aranges 00000000 .debug_aranges
00000000 l    d  .debug_info    00000000 .debug_info
00000000 l    d  .debug_abbrev  00000000 .debug_abbrev
00000000 l    d  .debug_line    00000000 .debug_line
00000000 l    d  .debug_str     00000000 .debug_str
00000000 l    df *ABS*     00000000 crtstuff.c
000105e0 l     O .eh_frame 00000000 __EH_FRAME_BEGIN__
000100b8 l     F .text     00000000 deregister_tm_clones
000100e0 l     F .text     00000000 register_tm_clones
0001011c l     F .text     00000000 __do_global_dtors_aux
00011a24 l     O .bss 00000001 completed.5176
000115e8 l     O .fini_array    00000000
__do_global_dtors_aux_fini_array_entry
00010164 l     F .text     00000000 frame_dummy
00011a28 l     O .bss 00000018 object.5181
```

```
000115e4 l     O .init_array    00000000
__frame_dummy_init_array_entry
00000000 l    df *ABS*     00000000 main.c
00000000 l    df *ABS*     00000000 atexit.c
00000000 l    df *ABS*     00000000 exit.c
00000000 l    df *ABS*     00000000 fini.c
00000000 l    df *ABS*     00000000 impure.c
000115f0 l     O .data     00000428 impure_data
00000000 l    df *ABS*     00000000 init.c
00000000 l    df *ABS*     00000000 __atexit.c
00000000 l    df *ABS*     00000000 __call_atexit.c
00000000 l    df *ABS*     00000000 sys_exit.c
00000000 l    df *ABS*     00000000 errno.c
00000000 l    df *ABS*     00000000 crtstuff.c
000105e0 l     O .eh_frame 00000000 __FRAME_END__
00000000 l    df *ABS*     00000000
000115ec l       .fini_array    00000000 __fini_array_end
000115e8 l       .fini_array    00000000 __fini_array_start
000115e8 l       .init_array    00000000 __init_array_end
000115e4 l       .init_array    00000000 __preinit_array_end
000115e4 l       .init_array    00000000 __init_array_start
000115e4 l       .init_array    00000000 __preinit_array_start
00012218 g       .sdata    00000000 __global_pointer$
000105b0 g     F .text     0000000c __errno
00011a18 g     O .sdata    00000000 .hidden __TMC_END__
00011a18 g     O .sdata    00000000 .hidden __dso_handle
00011a1c g     O .sdata    00000004 _global_impure_ptr
00010260 g     F .text     00000098 __libc_init_array
000100b4 g     F .text     00000000 _init
000101f0 g     F .text     00000070 __libc_fini_array
00010458 g     F .text     00000114 __call_exitprocs
00010074 g     F .text     00000040 _start
000103d4 g     F .text     00000084 __register_exitproc
000105bc g       .text     00000000 .hidden __mulsi3
00011a24 g       .bss 00000000 __bss_start
000102f8 g     F .text     000000dc memset
0001019c g     F .text     00000008 main
000100b4 g     F .text     00000000 _fini
000101a4 g     F .text     00000014 atexit
00011a20 g     O .sdata    00000004 _impure_ptr
00011a24 g       .sdata    00000000 _edata
00011a40 g       .bss 00000000 _end
000101b8 g     F .text     00000038 exit
0001056c g     F .text     00000044 _exit
```

Инструкции программы

Изучим содержимое секции “.text”:

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

Результирующий файл “a.ds” весьма велик (387 строк), поэтому здесь мы рассмотрим только его фрагменты.

```
a.out:     file format elf32-littleriscv


Disassembly of section .text:

00010074 <_start>:
   10074:  00002197        auipc gp,0x2
   10078:  1a418193        addi  gp,gp,420 # 12218 <__global_pointer$>
   1007c:  80c18513        addi  a0,gp,-2036 # 11a24 <_edata>
   10080:  82818613        addi  a2,gp,-2008 # 11a40 <_end>
   10084:  40a60633        sub   a2,a2,a0
   10088:  00000593        addi  a1,zero,0
   1008c:  26c000ef        jal   ra,102f8 <memset>
   10090:  00000517        auipc a0,0x0
   10094:  16050513        addi  a0,a0,352 # 101f0 <__libc_fini_array>
   10098:  10c000ef        jal   ra,101a4 <atexit>
   1009c:  1c4000ef        jal   ra,10260 <__libc_init_array>
   100a0:  00012503        lw    a0,0(sp)
   100a4:  00410593        addi  a1,sp,4
   100a8:  00000613        addi  a2,zero,0
   100ac:  0f0000ef        jal   ra,1019c <main>
   100b0:  1080006f        jal   zero,101b8 <exit>
   …
```

Заметим, что адрес символа “_start” мы уже встречали ранее – этот адрес указан в качестве «точки входа» в заголовке файла “a.out” (см. выше). Код, начинающийся с метки “_start” обеспечивает инициализацию памяти, регистров процессора и среды времени выполнения, после чего передает управление определенной нами функции main:

```
   100ac:  0f0000ef        jal   ra,1019c <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 $1019C_{16}$. В угловых скобках показано, что этот адрес соответствует символу “main”, что также видно из таблицы символов. Таким образом, действительно, данная инструкция вызывает функцию main.

```
…
0001019c <main>:
   1019c:  00000513        addi  a0,zero,0
   101a0:  00008067        jalr  zero,0(ra)
…
```

В этом фрагменте нет ничего неожиданного: инструкции, составляющие тело функции “main” попали в секцию “.text” исполняемого файла (выхода компоновщика) из одноименной секции объектного файла “main.o” (одного из входных файлов компоновщика).

Как мы уже видели,

…

```
   100ac:  0f0000ef        jal   ra,1019c <main>
   100b0:  1080006f        jal   zero,101b8 <exit>
   …
```

после возврата из функции `main` управление будет передано на адрес $101b8_{16}$, соответствующий символу “`exit`”.

```
…
000101b8 <exit>:
   101b8:  ff010113    addi  sp,sp,-16
   101bc:  00000593    addi  a1,zero,0
   101c0:  00812423    sw    s0,8(sp)
   101c4:  00112623    sw    ra,12(sp)
   101c8:  00050413    addi  s0,a0,0
   …
   101e8:  00040513    addi  a0,s0,0
   101ec:  380000ef    jal   ra,1056c <_exit>
…
```

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

```
0001056c <_exit>:
   1056c:  00000593            addi  a1,zero,0
   10570:  00000613            addi  a2,zero,0
   10574:  00000693            addi  a3,zero,0
   10578:  00000713            addi  a4,zero,0
   1057c:  00000793            addi  a5,zero,0
   10580:  05d00893            addi  a7,zero,93
   10584:  00000073            ecall
…
```

Инструкции `addi` используются для формирования значений регистров a1-a7. Инструкция `ecall` реализует «системный вызов» - обращение к операционной системе (мы не рассматриваем детали реализации этой инструкции, в качестве первого приближения можно думать о ней, как о вызове подпрограммы).

Вспомним, что в RISC-V ABI аргументы функции передаются в регистрах a0-a7. В порте ОС Linux для архитектуры RISC-V аргументы системного вызова передаются в регистрах a0-a5 (системный вызов может принимать до 6 аргументов), а номер системного вызова - число, идентифицирующее требуемую операцию – в регистре a7. Номер системного вызова 93 соответствует операции завершения текущего процесса (**exit system call**)[3], первый аргумент (a0) определяет код завершения процесса, остальные аргументы не используются и, в обеспечение совместимости с будущими версиями (**forward compatibility**), устанавливаются равными 0. (*Вопрос: где устанавливается значение регистра a0?*)

Как видно из этого примера, «ничего не делающая» программа может иметь довольно много инструкций.

[3] (*) https://github.com/riscv/riscv-newlib/, файл libgloss/riscv/machine/syscall.h

2. Сборка простейшей программы «по шагам»

В настоящем разделе выполним сборку простейшей программы по шагам. Для выполнения отдельных шагов мы будем по-прежнему запускать драйвер компилятора (а не обращаться, скажем, к ассемблеру или компоновщику напрямую), и контролировать его действия, используя флаг “-v”.

Препроцессирование

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

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -v -E
    main.c -o main.i
    >log.pp 2>&1
```

Как мы видели ранее, в пакете средств разработки “gcc” препроцессор не реализован, как отдельная программа, а является частью компилятора (в некоторых других пакетах препроцессор - отдельная программа, как это было исторически), и опция “-E” (мы уже видели опцию “-E” программы “cc1”, теперь же речь идет об опции драйвера компилятора “gcc”) приводит к останову процесса сборки после препроцессирования.

Возможность запуска отдельно препроцессора языка C иногда используется для обработки файлов исходных текстов на других языках, не имеющих встроенных макросредств (например, Java, некоторые ассемблеры). В этом случае может быть полезной также опция “-P”, подавляющая генерацию маркеров строк (директив вида “`# <lineno> <filename>`”, см. “main.i” выше)

Компиляция

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

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -v
    -S -fpreprocessed main.i -o main.s
    >log.cc 2>&1
```

Для останова процесса сборки после компиляции (без запуска ассемблера) используется опция “-S” драйвера компилятора. Заметим, что сам компилятор – программа “cc1” – такой опции, разумеется, не имеет, т.к. никогда не запускает ассемблер.

Следует отметить, что результатом работы компилятора языка высокого уровня вовсе не обязательно является код на языке ассемблера – компилятор может сразу формировать объектный код, например, так обычно работают компиляторы «языков Вирта» (Паскаль, Модула, Оберон), напротив, компиляторы C обычно транслируют программу на C в ассемблер.

Ассемблирование

Ассемблирование файла “`main.s`” выполняется по следующей команде:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -v
    -c main.s -o main.o
    >log.as 2>&1
```

Опция “-c” останавливает процесс сборки после ассемблирования. Заметьте, что здесь не используется опция “-O1”, т.к. ассемблер (*обычно*) не выполняет оптимизацию.

Как драйвер компилятора «понимает», что во входном файле “`main.s`” находится код на языке ассемблера (а не программа на языке C)? (Драйвер должен это понимать, чтобы запускать ассемблер, а не компилятор.) Для этого анализируется имя входного файла: если оно оканчивается на “.c”, предполагается, что файл содержит программу на C, на “.i” – программу на C, уже обработанную препроцессором (так что опцию “-fpreprocessed” выше можно было не указывать), на “.s” – код на языке ассемблера и т.д.[4] В случае необходимости, тип содержимого (входной язык) можно задать явно опцией “-x”. Напротив, компилятор и ассемблер обычно не пытаются «угадать» содержимое файла, вся необходимая информация должна передаваться явно аргументами командной строки.

Компоновка

Компоновка программы выполняется по следующей команде:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -v
    main.o -o main
    >log.ld 2>&1
```

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

3. Раздельная компиляция

Текст программы

В данном примере рассматривается тривиальная программа на языке C, исходный код которой разбит на 2 файла. Наша цель состоит в том, чтобы лучше разораться в процессе компоновки.

```
// Файл main.c

extern int zero( void );

int main( void ) {
    return zero();
}
```

```
// Файл zero.c

int zero( void ) {
    return 0;
}
```

Сборка программы

Благодаря драйверу компилятора, сборка программы, состоящей из двух файлов, нисколько не сложнее, чем сборка программы, весь текст которой находится в одном файле:

[4] https://gcc.gnu.org/onlinedocs/gcc/Overall-Options.html

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

В результате исполнения приведенной команды в текущем каталоге будут созданы следующие файлы, содержание которых не требует пояснений:

- `main.i`, `zero.i`;
- `main.s`, `zero.s`;
- `main.o`, `zero.o`;
- `a.out`;
- `log`.

Изучив файл “`log`“ легко видеть, что процесс сборки состоит из препроцессирования отдельно файлов “main.c” и “zero.c”, компиляции отдельно файлов “main.i” и “zero.i” (всего 4 запуска программы “`cc1`”), ассемблирования отдельно файлов “main.s” и “zero.s” (2 запуска программы “as”) и компоновки программы (один (!) запуск утилиты collect2, в командной строке которой теперь указан не только “main.o”, но и “zero.o”).

Также несложно провести сборку программы по шагам:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -E
    main.c -o main.i
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -S
    main.i -o main.s
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -c
    main.s -o main.o

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -E
    zero.c -o zero.i
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -S
    zero.i -o zero.s
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -c
    zero.s -o zero.o

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32
    main.o zero.o -o main
```

Выход компилятора

Изучим файлы “`main.s`” и “`zero.s`", сформированные компилятором:

```
      .file "zero.c"
      .option nopic
      .text
      .align 2
      .globl zero
      .type  zero, @function
zero:
      li    a0,0
      ret
      .size  zero, .-zero
      .ident "GCC: (GNU) 7.2.0"
```

```
      .file "main.c"
      .option nopic
      .text
      .align 2
      .globl main
      .type  main, @function
main:
      addi  sp,sp,-16
      sw    ra,12(sp)
      call  zero
      lw    ra,12(sp)
      addi  sp,sp,16
      jr    ra
      .size  main, .-main
      .ident "GCC: (GNU) 7.2.0"
```

Содержимое файла “zero.s” не нуждается в комментариях. В подпрограмме “`main`”, как несложно видеть, выполняется обращение к подпрограмме “`zero`” (значение регистра ra, содержащее адрес возврата из “`main`”, сохраняется на время вызова в стеке). Следует отметить, что символ “`zero`” используется в файле “`main.s`”, *но никак не определяется*.

Таблицы символов

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

```
riscv64-unknown-elf-objdump -t zero.o main.o
```

Вывод утилиты:

```
zero.o:     file format elf32-littleriscv

SYMBOL TABLE:
00000000 l    df *ABS*     00000000 zero.c
00000000 l    d  .text     00000000 .text
00000000 l    d  .data     00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .comment  00000000 .comment
00000000 g     F .text     00000008 zero


main.o:     file format elf32-littleriscv

SYMBOL TABLE:
00000000 l    df *ABS*     00000000 main.c
00000000 l    d  .text     00000000 .text
00000000 l    d  .data     00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .comment  00000000 .comment
00000000 g     F .text     0000001c main
00000000         *UND*     00000000 zero
```

Содержимое таблицы символов “zero.o” абсолютно предсказуемо. В таблице же символов “main.o” имеется интересная запись: символ “zero” типа “*UND*” (undefined – не определен). Эта запись означает, что символ “zero” 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.

Инструкции программы

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

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

Вывод утилиты:

```
zero.o:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <zero>:
   0: 00000513              addi  a0,zero,0
   4: 00008067              jalr  zero,0(ra)

main.o:     file format elf32-littleriscv


Disassembly of section .text:

00000000 <main>:
   0: ff010113              addi  sp,sp,-16
   4: 00112623              sw    ra,12(sp)
   8: 00000097              auipc ra,0x0
   c: 000080e7              jalr  ra,0(ra) # 8 <main+0x8>
  10: 00c12083              lw    ra,12(sp)
  14: 01010113              addi  sp,sp,16
  18: 00008067              jalr  zero,0(ra)
```

Результат дизассемблирования “zero.o” интереса не представляет, в отличие от результата дизассемблирования “main.o”: сравнивая его с “main.s”, легко понять, что псевдоинструкция вызова подпрограммы “zero”, транслировалась ассемблером в следующую пару инструкций:

```
   8: 00000097              auipc ra,0x0
   c: 000080e7              jalr  ra,0(ra) # 8 <main+0x8>
```

Результатом выполнения этой пары инструкций станет переход на адрес 8 (это показано в выводе дизассемблера; убедитесь, что Вы можете обосновать это утверждение по материалам презентации «RISC-V ISA Basics» или спецификации “The RISC-V Instruction Set Manual Volume I: User-Level ISA”), т.е. зацикливание!

Загадочное поведение ассемблера объясняется очень просто: ассемблер не имел возможности определить *целевой адрес* перехода (кроме того, что этот адрес обозначен символом “zero”),

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. Но как компоновщик узнает о том, что и каким образом нужно исправить? (*Можете ли Вы ответить на этот вопрос?*)

Таблица перемещений

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

```
riscv64-unknown-elf-objdump -r zero.o main.o
```

Вывод утилиты:

```
zero.o:     file format elf32-littleriscv


main.o:     file format elf32-littleriscv

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_RISCV_CALL      zero
00000008 R_RISCV_RELAX     *ABS*
```

Содержимое "`zero.o`" не требует модификации, поэтому не содержит записей о перемещениях (**relocation entries**). В файле же "`main.o`" имеется две записи, относящиеся к адресу 8 (как мы видели выше, по этому адресу в "`main.o`" находится первая инструкция пары `auipc`+`jalr`). Дизассемблирование и вывод таблицы перемещений можно совместить:

```
riscv64-unknown-elf-objdump -d -M no-aliases -r main.o
```

Вывод утилиты:

```
main.o:     file format elf32-littleriscv

Disassembly of section .text:

00000000 <main>:
   0: ff010113              addi  sp,sp,-16
   4: 00112623              sw    ra,12(sp)
   8: 00000097              auipc ra,0x0
                8: R_RISCV_CALL zero
                8: R_RISCV_RELAX *ABS*
   c: 000080e7              jalr  ra,0(ra) # 8 <main+0x8>
  10: 00c12083              lw    ra,12(sp)
  14: 01010113              addi  sp,sp,16
  18: 00008067              jalr  zero,0(ra)
```

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 указано, что по адресу 8 следует исправить *пару инструкций* (тип перемещения "`R_RISCV_CALL`") так, чтобы результат

соответствовал вызову подпрограммы “zero”. Типы перемещений специфичны для каждой архитектуры системы команд и обычно определены в ABI (**Application Binary Interface**)[5].

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. К этому вопросу мы еще вернемся, а сейчас, для удобства изложения, запретим компоновщику выполнять такую оптимизацию:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 –Wl,--no-relax
    main.o zero.o -o main
```

Опция “-Wl” драйвера “gcc” позволяет передавать дополнительные аргументы компоновщику. Здесь мы используем опцию компоновщика “--no-relax”, отключающую оптимизацию, о которой шла речь выше.

Результат компоновки

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

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

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

```
main:     file format elf32-littleriscv

Disassembly of section .text:
…
000101c8 <main>:
   101c8:  ff010113            addi  sp,sp,-16
   101cc:  00112623            sw    ra,12(sp)
   101d0:  00000097            auipc ra,0x0
   101d4:  014080e7            jalr  ra,20(ra) # 101e4 <zero>
   101d8:  00c12083            lw    ra,12(sp)
   101dc:  01010113            addi  sp,sp,16
   101e0:  00008067            jalr  zero,0(ra)

000101e4 <zero>:
   101e4:  00000513            addi  a0,zero,0
   101e8:  00008067            jalr  zero,0(ra)
…
```

Прежде всего можно видеть, что в результат компоновки попало содержимое обоих объектных файлов – “main.o” и “zero.o”. Инструкции подпрограммы “zero” начинаются с адреса $101E4_{16}$, и пара инструкций auipc+jalr, вызывающих подпрограмму “zero” соответствующим образом откорректированы: по команде auipc в регистр ra будет загружено значение pc + *signext*(0<<12), равное $101D0_{16}$ (*почему?*). Результатом jalr станет переход на адрес ra + 20 = $101D0_{16}$ + 14_{16} = $101E4_{16}$ – адрес первой инструкции подпрограммы “zero”.

[5] ELF ABI RISC-V находится по адресу https://github.com/riscv/riscv-elf-psabi-doc/blob/master/riscv-elf.md

Оптимизация при компоновке

В рассматриваемом нами примере точка вызова подпрограммы “zero” и сама подпрограмма находятся очень близко – их разделяет всего 12 байт. Учитывая это, в данном случае нет никакой необходимости использовать пару инструкций `auipc+jalr`, достаточно одной инструкции “`jal`”, 20-разрядный непосредственный операнд которой позволяет задавать переход в пределах ±1 МиБ относительно адреса инструкции (значения `pc` в момент ее выполнения).

В общем случае целевой адрес перехода может отличаться от адреса инструкции перехода более чем на 1МиБ, и поскольку ассемблер не имеет информации о целевом адресе перехода, при использовании псевдоинструкции `call` формируется пара инструкций `auipc+jalr`, использование которой позволяет выполнять переход в пределах ±2 ГиБ (в любую точку, в случае 32-разрядного адресного пространства).

В отличие от ассемблера, компоновщик имеет всю необходимую информацию об адресах, и может принять решение о возможности использования инструкции `jal` вместо пары `auipc+jalr`. Такая оптимизация реализуется компоновщиком для архитектуры RISC-V и называется “**linker relaxation**”[6]. Каждая пара инструкций, которая *может* быть оптимизирована, помечается ассемблером записью о перемещении типа “`R_RISCV_RELAX`”, пример которой мы уже видели. Рассмотрим результат такой оптимизации:

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

Соответствующий фрагмент “`a.ds`”:

```
a.out:     file format elf32-littleriscv

Disassembly of section .text:
…
0001019c <main>:
   1019c:  ff010113            addi  sp,sp,-16
   101a0:  00112623            sw    ra,12(sp)
   101a4:  010000ef            jal   ra,101b4 <zero>
   101a8:  00c12083            lw    ra,12(sp)
   101ac:  01010113            addi  sp,sp,16
   101b0:  00008067            jalr  zero,0(ra)

000101b4 <zero>:
   101b4:  00000513            addi  a0,zero,0
   101b8:  00008067            jalr  zero,0(ra)
…
```

Можно видеть, что подпрограмма main имеет на одну инструкцию меньше: пара инструкций `auipc+jalr` заменена компоновщиком одной инструкцией `jal`. (В выводе дизассемблера показан целевой адрес перехода - $101B4_{16}$; фактически же, разумеется, в коде инструкции `jal` целевой адрес задается *смещением* относительно текущего значения `pc`, это легко видеть из кода инструкции).

[6] https://www.sifive.com/blog/2017/08/28/all-aboard-part-3-linker-relaxation-in-riscv-toolchain/

Порядок компоновки

Изменим порядок, в котором указываются объектные файлы в команде запуска компоновщика:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32
    zero.o main.o -o main
riscv64-unknown-elf-objdump -j .text -d -M no-aliases main >main.ds
```

Фрагмент “main.ds”:

```
main:     file format elf32-littleriscv

Disassembly of section .text:
…
0001019c <zero>:
   1019c:  00000513             addi  a0,zero,0
   101a0:  00008067             jalr  zero,0(ra)

000101a4 <main>:
   101a4:  ff010113             addi  sp,sp,-16
   101a8:  00112623             sw    ra,12(sp)
   101ac:  ff1ff0ef             jal   ra,1019c <zero>
   101b0:  00c12083             lw    ra,12(sp)
   101b4:  01010113             addi  sp,sp,16
   101b8:  00008067             jalr  zero,0(ra)
…
```

(*) Оптимизация размера исполняемого файла

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

```
riscv64-unknown-elf-objdump –f –h a.out
```

Вывод утилиты:

```
a.out:     file format elf32-littleriscv
architecture: riscv:rv32, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00010074

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000584  00010074  00010074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .eh_frame     00000004  000105f8  000105f8  000005f8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .init_array   00000004  000115fc  000115fc  000005fc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .fini_array   00000004  00011600  00011600  00000600  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .data         00000428  00011608  00011608  00000608  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  5 .sdata        0000000c  00011a30  00011a30  00000a30  2**2
```

```
                  CONTENTS, ALLOC, LOAD, DATA
  6 .bss          0000001c  00011a3c  00011a3c  00000a3c  2**2
                  ALLOC
  7 .comment      00000011  00000000  00000000  00000a3c  2**0
                  CONTENTS, READONLY
  8 .debug_aranges 00000020  00000000  00000000  00000a50  2**3
                  CONTENTS, READONLY, DEBUGGING
  9 .debug_info   00000026  00000000  00000000  00000a70  2**0
                  CONTENTS, READONLY, DEBUGGING
 10 .debug_abbrev 00000014  00000000  00000000  00000a96  2**0
                  CONTENTS, READONLY, DEBUGGING
 11 .debug_line   00000098  00000000  00000000  00000aaa  2**0
                  CONTENTS, READONLY, DEBUGGING
 12 .debug_str    000000f7  00000000  00000000  00000b42  2**0
                  CONTENTS, READONLY, DEBUGGING
```

В реальных программах объем этой информации может быть весьма большим. Поскольку, разумеется, она абсолютно не требуется для загрузки и исполнения программы, можно предположить, что должен быть способ удалить ее из исполняемого файла. Это действительно так:

```
riscv64-unknown-elf-strip –R .comment –o a.img a.out

riscv64-unknown-elf-objdump –f –h a.img
```

Вывод утилиты:

```
a.img:     file format elf32-littleriscv

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000584  00010074  00010074  00000074  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .eh_frame     00000004  000105f8  000105f8  000005f8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .init_array   00000004  000115fc  000115fc  000005fc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  3 .fini_array   00000004  00011600  00011600  00000600  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  4 .data         00000428  00011608  00011608  00000608  2**3
                  CONTENTS, ALLOC, LOAD, DATA
  5 .sdata        0000000c  00011a30  00011a30  00000a30  2**2
                  CONTENTS, ALLOC, LOAD, DATA
  6 .bss          0000001c  00011a3c  00011a3c  00000a3c  2**2
                  ALLOC
```

Опция “`-R .comment`” указывает утилите “strip”, что секцию “.comment” также следует удалить из файла (по умолчанию секция “.comment” не удаляется). Утилита “strip” позволяет удалять отдельные символы, секции, записи о перемещениях и пр.

(**) Следует отметить, что удаление отладочной информации из исполняемого файла не означает, что его отладка невозможна: требуемую информацию отладчик может найти в «исходном» - до обработки утилитой “strip” - исполняемом файле.

4. Использование статических библиотек

Тексты программ

В данном примере рассматривается 2 простые программы на языке C, исходный код которых разбит на 6 файлов. Наша цель состоит в том, чтобы разораться в вопросах создания и использования статических библиотек.

```
// Файл main1.c

extern int do_something( void );
extern int do_something_else( void );

int main( void ) {
    return do_something();
}


// Файл main2.c

extern int do_something( void );
extern int do_something_else( void );

int main( void ) {
    return do_something_else();
}


// Файл ds.c

extern int do_something_internal( void );

int do_something( void ) {
    return do_something_internal();
}

extern int do_something_internal2( void );

int do_something2( void ) {
    return do_something_internal2();
}


// Файл dsi.c

int do_something_internal( void ) {
```

```
    return 0;
}


// Файл dsi2.c

int do_something_internal2( void ) {
    return 0;
}


// Файл dse.c

int do_something_else( void ) {
    return 0;
}
```

Первая программа будет собираться из файлов “main1.c”, “ds.c”, “dsi.c”, “dsi2.c”, “dse.c”, вторая - из файлов “main2.c”, “ds.c”, “dsi.c”, “dsi2.c”, “dse.c”. (*Какие файлы можно исключить из сборки каждой программы?*)

Сборка программ

Сборка программ осуществляется следующими командами:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -v
  main1.c ds.c dsi.c dsi2.c dse.c  -o main1  >log1 2>&1

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 -v
  main2.c ds.c dsi.c dsi2.c dse.c  -o main2  >log2 2>&1
```

Анализируя содержимое файлов “log1” и “log2”, можно убедиться в том, что процесс сборки неэффективен: файлы “ds.c”, “dsi.c”, “dsi2.c”, “dse.c” обрабатываются (препроцессором, компилятором, ассемблером) дважды. Для наших программ, разумеется, это не является проблемой, однако в случае больших программ время сборки может серьезно влиять на производительность *труда* программиста. Заметив этот недостаток, легко его исправить:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1
  -c ds.c -o ds.o
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1
  -c dsi.c -o dsi.o
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1
  -c dsi2.c -o dsi2.o
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1
  -c dse.c -o dse.o

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1
  main1.c ds.o dsi.o dsi2.o dse.o  -o main1
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1
  main2.c ds.o dsi.o dsi2.o dse.o  -o main2
```

В первых командах мы используем опцию “-c” *драйвера* компилятора, что, как мы уже видели, приводит к останову процесса сборки после ассемблирования, т.е. после формирования

объектного файла. 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.

В двух последних командах следует обратить внимание на то, что в командной строке драйвера компилятора перечисляются исходные файлы на языке C и объектные файлы. Драйвер компилятора обеспечивает при этом препроцессирование, компиляцию и ассемблирование исходного кода с последующей компоновкой программы из всех объектных файлов. (*Как драйвер компилятора «понимает», как обрабатывать каждый входной файл?*)

Оптимизация состава программ

Изучим таблицы символов полученных исполняемых файлов:

```
riscv64-unknown-elf-objdump –t main1 main2 >symtab
```

Фрагменты файла “symtab”:

```
main1:     file format elf32-littleriscv

SYMBOL TABLE:
…
000101e4 g     F .text     00000015 do_something2
…
00010210 g     F .text     00000008 do_something_else
00010208 g     F .text     00000008 do_something_internal2
…
000101ac g     F .text     00000015 main
…
00010200 g     F .text     00000008 do_something_internal
…
000101c8 g     F .text     00000019 do_something
…

main2:     file format elf32-littleriscv

SYMBOL TABLE:
…
000101e8 g     F .text     00000015 do_something2
…
00010214 g     F .text     00000008 do_something_else
0001020c g     F .text     00000008 do_something_internal2
…
000101ac g     F .text     00000019 main
…
00010204 g     F .text     00000008 do_something_internal
…
000101cc g     F .text     00000019 do_something
…
```

Как и следовало ожидать, в состав исполняемого файла вошло содержимое всех объектных файлов, указанных в команде сборки. В то же время, анализируя тексты программ, легко видеть,

что состав исходных файлов может быть оптимизирован (уменьшен). Так, исполнение первой программы начинается с функции “main”, определенной в файле “main1.c”, эта функция вызывает функцию “do_something”, определенную в “ds.c”, которая, в свою очередь, вызывает “do_something_internal”, определенную в “dsi.c”. Таким образом, «полезный» код программы находится в файлах “main1.c”, “ds.c” и “dsi.c”. (*В каких файлах содержится «полезный» код второй программы?*)

Тем не менее, попытка сборки первой программы только из «полезных» файлов:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1
  main1.c ds.c dsi.c –o main1
```

завершается ошибкой:

```
C:\Users\u\AppData\Local\Temp\cc1Em8Yo.o: In function `do_something':
ds.c:(.text+0x20): undefined reference to `do_something_internal2'
collect2.exe: error: ld returned 1 exit status
```

Внимательно прочитав сообщение об ошибке, можно понять, что проблема возникла на этапе компоновки (не препроцессирования, не компиляции, не ассемблирования). (*Можете ли Вы объяснить, чем вызвана ошибка, и как ее исправить?*) Причину ошибки несложно понять, а поняв – исправить.

Прежде всего, объясним «загадочное» имя файла “`C:\Users\u\AppData\Local\Temp\cc1Em8Yo.o`”, в тексте сообщения об ошибке: в процессе сборке программы необходимо сформировать объектный файл из исходного файла “`ds.c`” (из «известных нам» имен только это фигурирует в тексте сообщения), причем сохранять сформированный объектный файл не требуется (в команде не указана опция “`--save-temps`”), поэтому *драйвер компилятора* использует временный (**temporary**, обратите внимание на компонент “\Temp\” пути) файл, расположение и способ генерации имени которого зависит от используемой операционной системы.

В “`ds.c`” помимо «полезной» нам функции “do_something” определена функция “do_something2”, вызывающая функцию “do_something_internal2”, в результате, в объектном файле, генерируемом из “ds.c”, имеется соответствующая запись о перемещении:

```
riscv64-unknown-elf-objdump -r ds.o
```

Вывод утилиты:

```
ds.o:     file format elf32-littleriscv

RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000008 R_RISCV_CALL      do_something_internal
00000008 R_RISCV_RELAX     *ABS*
00000024 R_RISCV_CALL      do_something_internal2
00000024 R_RISCV_RELAX     *ABS*
```

Для обработки этой записи компоновщику требуется «разрешить» (**resolve**) – т.е. определить адрес - символ “do_something_internal2”, а этот символ, очевидно, соответствует функции

“do_something_internal2”, определенной в “dsi2.c”. Следовательно, “dsi2.c” необходимо включить в состав исходных кодов программы:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1
  main1.c ds.c dsi.c dsi2.c -o main1
```

Аналогично, при сборке программы с использованием подготовленных ранее объектных файлов, надо не забыть указать *все необходимые* файлы:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1
  main1.c ds.o dsi.o dsi2.o –o main1
```

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.

В нашем примере в библиотеку объединим объектные файлы “`ds.o`”, “`dsi.o`”, “`dsi2.o`” и “`dse.o`”:

```
riscv64-unknown-elf-ar  –rsc libds.a  ds.o dsi.o dsi2.o dse.o
```

Результирующим файлом является “libds.a” (“.a” – от “**archive**”). Проверим его содержимое:

```
riscv64-unknown-elf-ar  –t libds.a
```

Вывод утилиты:

```
ds.o
dsi.o
dsi2.o
dse.o
```

Используем статическую библиотеку для сборки программ:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 --save-temps
  main1.c libds.a  -o main1

riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -O1 --save-temps
  main2.c libds.a  -o main2
```

Изучим таблицы символов полученных исполняемых файлов:

```
riscv64-unknown-elf-objdump –t main1 main2 >symtabl
```

Фрагменты файла “`symtabl`”:

```
main1:     file format elf32-littleriscv

SYMBOL TABLE:
…
000101e4 g     F .text     00000015 do_something2
…
```

```
00010204 g     F .text     00000008 do_something_internal2
…
000101ac g     F .text     00000019 main
…
000101fc g     F .text     00000008 do_something_internal
…
000101cc g     F .text     00000019 do_something
…

main2:     file format elf32-littleriscv

SYMBOL TABLE:
…
000101cc g     F .text     00000008 do_something_else
…
000101ac g     F .text     00000019 main
…
```

Легко видеть, что в состав программы “main1” не вошло содержимое объектного файла “dse.o”, а в состав “main2” – объектных файлов “ds.o”, “dsi.o”, “dsi2.o”. 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). В частности, при сборке программы “main1” компоновщик сам определил, что в состав исполняемого файла должно быть включено содержимое “ds.o”, *а значит*, и “dsi.o” и “dsi2.o”.

Каким образом компоновщик определяет, какие файлы следует использовать? В соответствии с порядком указания «входных» файлов в команде сборки, первым в компоновке будет участвовать объектный файл, полученный из “main1.c” (на самом деле, как мы видели ранее, *первым* в компоновке будет участвовать файл “crt0.o”). Его включение приведет к необходимости разрешения символа “do_something”:

```
riscv64-unknown-elf-objdump -t main1.o
```

Вывод утилиты:

```
main1.o:     file format elf32-littleriscv

SYMBOL TABLE:
00000000 l    df *ABS*     00000000 main1.c
00000000 l    d  .text     00000000 .text
00000000 l    d  .data     00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .comment  00000000 .comment
00000003 g     F .text     0000001c main
00000000         *UND*     00000000 do_something
```

(*Откуда можно видеть, что “do_something” требует разрешения?*)

Символ “do_something” определен (*только*) в объектном файле “ds.o”, входящем в библиотеку “libds.a”:

```
riscv64-unknown-elf-objdump -t ds.o
```

Вывод утилиты:

```
ds.o:     file format elf32-littleriscv

SYMBOL TABLE:
00000000 l    df *ABS*     00000000 ds.c
00000000 l    d  .text     00000000 .text
00000000 l    d  .data     00000000 .data
00000000 l    d  .bss 00000000 .bss
00000000 l    d  .comment  00000000 .comment
00000000 g     F .text     0000001c do_something
00000000         *UND*     00000000 do_something_internal
0000001c g     F .text     0000001c do_something2
00000000         *UND*     00000000 do_something_internal2
```

(*Откуда можно видеть, что символ "do_something" определен в "ds.o"?*)

Символ do_something теперь разрешен, но к списку неразрешенных символов добавились "do_something_internal" и "do_something_internal2". Компоновщик *продолжает* поиск объектных файлов, содержащих хотя бы один из оставшихся неразрешенными символов, в библиотеке "`libds.a`". В ходе этого процесса в состав исполняемого файла включается содержимое объектного файла "`dsi.o`" (разрешающего символ "do_something_internal", и не добавляющего новых символов к списку требующих разрешения) и объектного файла "`dsi2.o`" (аналогично).

Список символов библиотеки

Следующая команда выводит список символов библиотеки "`libds.a`":

```
riscv64-unknown-elf-nm libds.a
```

Вывод утилиты:

```
ds.o:
00000000 T do_something
         U do_something_internal
         U do_something_internal2
0000001c T do_something2

dsi.o:
00000000 T do_something_internal

dsi2.o:
00000000 T do_something_internal2

dse.o:
00000000 T do_something_else
```

Несложно догадаться, что в выводе утилиты "nm" кодом "T" обозначаются символы, определенные в соответствующем объектном файле, кодом "U" - внешние символы.

Как обычно, утилита “nm” имеет целый ряд опций, позволяющих управлять составом и форматом вывода символов. Кроме того, утилита “nm” может применяться не только к файлам библиотек, но и к объектным и исполняемым файлам.

Порядок перечисления входных файлов компоновщика

Попробуем выполнить сборку программы “main1” следующей командой:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32
  libds.a main1.o  -o main1
```

(*Можете ли Вы предсказать результат?*)

```
main1.o: In function `main':
main1.c:(.text+0x8): undefined reference to `do_something'
collect2.exe: error: ld returned 1 exit status
```

(*Можете ли Вы объяснить результат?*)

Поскольку компоновщик (обычно) обрабатывает входные файлы в порядке их указания в командной строке, библиотека “`libds.a`” обрабатывается (просматривается) до того, как в списке неразрешенных символов появляются те символы, которые определены во входящих в ее состав объектных файлах. В результате, компоновщик не включает ни один из объектных файлов библиотеки в состав исполняемого файла и переходит к следующему входному файлу. Следующим входным файлом является объектный файл “`main1.o`”, содержимое которого *безусловно* включается в состав исполняемого файла. При этом в списке требующих разрешения появляется символ “do_something”, однако библиотека “libds.a” к этому моменту уже просмотрена, так что символ остается неразрешенным, что приводит к ошибке компоновки.

Учитывая сказанное, можно сформулировать следующие рекомендации по упорядочиванию входных файлов компоновщика: сначала должны указываться объектные файлы «основной» программы, потом библиотеки; библиотеки должны указываться в порядке от «частного» («специализированного») к «общему» («стандартному»). (*Вопрос: В каком порядке следует указывать стандартную библиотеку математических функций и библиотеку поддержки компилятора, в которой реализованы, функции умножения, деления и пр.?*)

Поиск библиотек

Компоновщик может осуществлять поиск файлов библиотек по имени в «стандартных», для данной операционной системы, и явно указанных в командной строке каталогах. Так, сборка программы “main1” может быть осуществлена следующей командой:

```
riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32
  main1.o -L. -lds -o main1
```

Здесь опция “`-L.`” включает текущий каталог (“.”) в пути поиска библиотек; опция “`-lds`” указывает на необходимость использования (просмотра) библиотеки “ds”. Следует заметить, что в опции “`-l`” указывается только «содержательная» часть имени библиотеки (только “ds” вместо “libds.a”). Если бы файл “libds.a” находился в одном из «стандартных» каталогов, содержащих библиотеки, указывать опцию “`-L`” не потребовалось.