香港公開大學 科技學院
THE OPEN UNIVERSITY OF HONG KONG
SCHOOL OF SCIENCE AND TECHNOLOGY

ELEC S411F 2020/21

Electronic and Computer Engineering Project

Final Report

Implementation of a New Executable and Linkable Format for RISC-V

Project Number: A01
Student Name: Law Tsz Wang Leslie (12419115)
Supervisor: Dr. Angus Wong
Submission Date: 13 June 2021

Declaration of Originality

I, Law Tsz Wang Leslie, declare that this report and the work reported herein was composed by and originated entirely from me. This report has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given in the reference section.

13 June 2021

Abstract

Since RISC-V is becoming a revolutionary instruction set architecture (ISA), many technology companies have developed a strong interest in this ISA, and then invest resources for development. RISC-V will change the ecology of the entire computer architecture industry.

This project is to design a lightweight and dynamic executable and linkable format (ELF) file in order to have a more efficient file linking and execution. Then, implement the modified ELF file to RISC-V environment, to better support the future RISC-V CPU and OS development.

The GCC file is chosen to be optimized for testing that modify an ELF file can get improvement when building a RISC-V operating environment. The result showed that the optimized GNU C Compiler (GCC) file can compile almost 10% faster than the original one. It also means that the ELF file act as a main character between operating system and ISA.

Table of Content

1 Introduction ... 6

1.1 Project Objectives ... 6

1.2 Organization of the Report ... 6

2 Literature Review ... 7

2.1 Compete the x86 and ARM ... 7

2.2 RISC-V Emulator Development ... 7

2.3 Low Performance Emulator ... 8

2.4 GCC Compiler ... 8

3 Methodology ... 9

3.1 Read and Write ELF file ... 9

3.2 GNU Compiler Toolchain ... 9

3.3 64-bit RISC-V Linux on QEMU ... 10

4 Implementation ... 11

4.1 Read ELF on Terminal ... 11

4.2 Read and Write ELF Program ... 12

4.3 Section Header in ELF Files ... 14

4.4 Building RISC-V Environment ... 15

4.5 Compile GCC ... 16

5 Results and Discussion ... 17

5.1 GCC Compiling Result ... 17

6 Conclusion ... 18

7 References ... 19

8 Appendix ... 20

List of Figures

Figure 4.1	Result on “readelf” of GCC	11
Figure 4.2	Java program and result – Read ELF	12
Figure 4.3	Java program and result – Write ELF	13
Figure 4.4	Cover page of “64-bit ELF Object File Specification”	13
Figure 4.5	First 5 section headers of GCC	14
Figure 4.6	Help instruction and update process of Homebrew	15
Figure 4.7	Content of the GNU Compiler Toolchain	15
Figure 4.8	Screenshot when compiling GCC	16

List of Tables

Table 4.1	ELF-64 Data Types	13
Table 4.2	ELF-64 Header Structure	14
Table 5.1	Compile Time Result	17

1 Introduction

As the technology nowadays becomes more complex and more connected, the interoperability between inventors and consumers has become more important. Global standards can make the interoperability easier and more convenient, thereby driving innovation at the basic platform level.

RISC-V is a free and open instruction set architecture (ISA) which still under development. RISC-V means the fifth edition of reduced instruction set computer (RISC) architecture. It is driven through open standard collaboration of global developers, and aims to achieve freedom of design in all domains and industries, and consolidate the strategic foundation of semiconductors.

Executable and Linkable Format (ELF) is an standard, flexible and cross-platform object files participate in program linking and execution which is a binary file format commonly found in in Unix-like systems. Since the bit "zero-one" in computer system are like the DNA molecules of the organism, ELF should be the general structure of the cell. ELF supports different endianness and address-sizes, and not requires specific CPU or ISA. Therefore, implementing an optimized ELF may improve the development for RISC-V.

1.1 Project Objectives

This project is aims to modify ELF files so as to have a more efficient file linking and execution in RISC-V environment, and hopefully can participate in a very small little part of the RISC-V development.

1.2 Organization of the Report

The report is begin with section 2, the background observation of RISC-V and ELF. Some literature will be reviewed for reference on the further work on RISC-V environment. After understanding the development of RISC-V, section 3 is the methodology is listing in detail to clearly describe what will be done. Section 4 is the working process and some screenshot of the work. The testing and work result of section 4 and the further discussion is showed in section 5. The section 6 is the conclusion of the project.

2 Literature Review

RISC-V is an ISA which not optimised for any particular target, so it is suitable for all computing purposes. It is also a simple load-store architecture which can be divided into two parts, base ISA and optional extensions which means RISC-V is restricted to contain minimal instructions set but also support extensive customisation.

2.1 Compete the x86 and ARM

Comparing to X86 and ARM, the world's most famous ISA for PC and mobile. RISC-V are free and no need to pay for the IP license fee [3]. It is simpler and smaller than other ISAs, and support modularisation with multiple standard extensions. The high stability due to fixed base ISA and first extension let the developers only need to update or change the extensions. And, due to high extensibility, specific functions can be added by extensions. So, RISC-V may have a good future to compete the x86 and ARM.

In the 1980s, chip size and processor design complexity were the main limiting factors, while desktop computers and servers completely dominated the computing industry [1]. Today, energy efficiency are the main design constraints, and the computing industry is very different: the growth of smartphones or tablets running ARM exceeds that of desktops or laptops running x86. In addition, the traditional low-power ARM is entering the high-performance computer market, while the traditional high-performance x86 is entering the low-power mobile device market. Therefore, the ISA performance and energy efficiency is becoming more and more important.

2.2 RISC-V Emulator Development

RISC-V is an open and free ISA, originally developed by the University of California, and now maintained by the RISC-V Foundation, with a few companies supporting its development [4]. It is a small RISC-based architecture that divided into multiple modules supporting floating-point calculations, vector operations and atomic operations. Each module focuses on different future computing goals, such as IoT embedded devices and cloud servers.

The rapid growth and adoption of RISC-V is attracting worldwide attention. So far, Linux kernel, GCC, Clang all support it. However, the performance of all current RISC-V simulators is very poor. Having a high-performance RISC-V emulator suitable for common architectures, x86 and ARM, in order to simplify software deployment, not only promotes its adoption and testing, but also showcases it as a useful virtual architecture. A way to implement a high-performance simulator is to use dynamic binary conversion (DBT), a technique for dynamically selecting and converting code regions during simulation [4].

This technology has been used to implement fast virtual machines (VM), simulators, debuggers, and high-level language VMs [4]. The DBT engine usually interprets the code first, and then after translating all the hot areas, most of the time is spent executing the translation area.

2.3 Low Performance Emulator

In ISA design, RISC-V has not reached a mature and stable state until now. Physically, there are several open source RISC-V CPU designs available. Although open source design can enable mass production, this type of design is still under research and experimentation, so there are currently few platforms that implement RISC-V architecture [4]. It usually takes some time to realize that the hardware of the new ISA is widely available. Until then, simulation plays a vital role, it can use the new ISA without a physical CPU available.

The main job of the ISA emulator is to convert client instructions into host instructions. The goal is to make the host execute the client's instructions. Although there are already some RISC-V simulators available, such as Spike and QEMU, they cannot achieve close to native performance, which limits their performance, so they are usually used in situations where performance requirements are not high.

2.4 GCC Compiler

GNU Compiler Collection (GCC) is the traditional compiler for most embedded systems because it supports many different ISAs on the back end. It supports many embedded processors and microcontrollers, and its status as the official GNU/Linux compiler is due to the open source model and support from the GNU/Linux community. Also for RISC-V, it is the first default compiler available [5].

3 Methodology

This project is using Java in programming for creating and modify the ELF file. In system programing, a Java SE Development Kit 16 is used for implementation of the Java platform function, integrated development environment for Java and other related function in Shell script. When emulating RISC-V environment, the GNU Compiler Toolchain recommended by the RISC-V Foundation is the main character.

3.1 Read and Write ELF file

ELF file can be found in any OS. Command line on Windows or Terminal on macOS provide an easiest way to read the ELF file information of different system or complier file. The command line program will show the ELF header, section header and program header of the ELF file. Those information will tells the system how the ELF file creating a process image.

In order to show the above information in the binary form, a simple Java program is needed. The program can read the byte inside the ELF file, then print out in hexadecimal form which is easier to read and analysis. This program involves the conversion between the byte of the file and the binary code, as well as the conversion between binary and hexadecimal string.

As we can read the ELF in customise form, a separate program can also write a whole new ELF file by converting the information string to hexadecimal code, then convert it to binary code and write into a newly created file.

3.2 GNU Compiler Toolchain

The RISC-V foundation recommended the RISC-V GNU Compiler Toolchain which is avalible on Github. This toolchain is the RISC-V C and C++ cross-compiler. It supports two build modes: a generic ELF/Newlib toolchain and a more sophisticated Linux-ELF/glibc toolchain.

The original GNU C Compiler (GCC) is developed from the GNU project to create a complete Unix-like operating system as free software, to promote freedom and cooperation among computer users and programmers.

GCC has grown over times to support many different programing languages such as C (gcc), C++ (g++), Java (gcj) which is now referred to as "GNU Compiler Collection". The GNU Compiler Toolchain is for developing applications and writing operating systems which include GCC, GNU Make, GNU Binutils, GNU Debugger (GDB), GNU Autotools and GNU Bison [2].

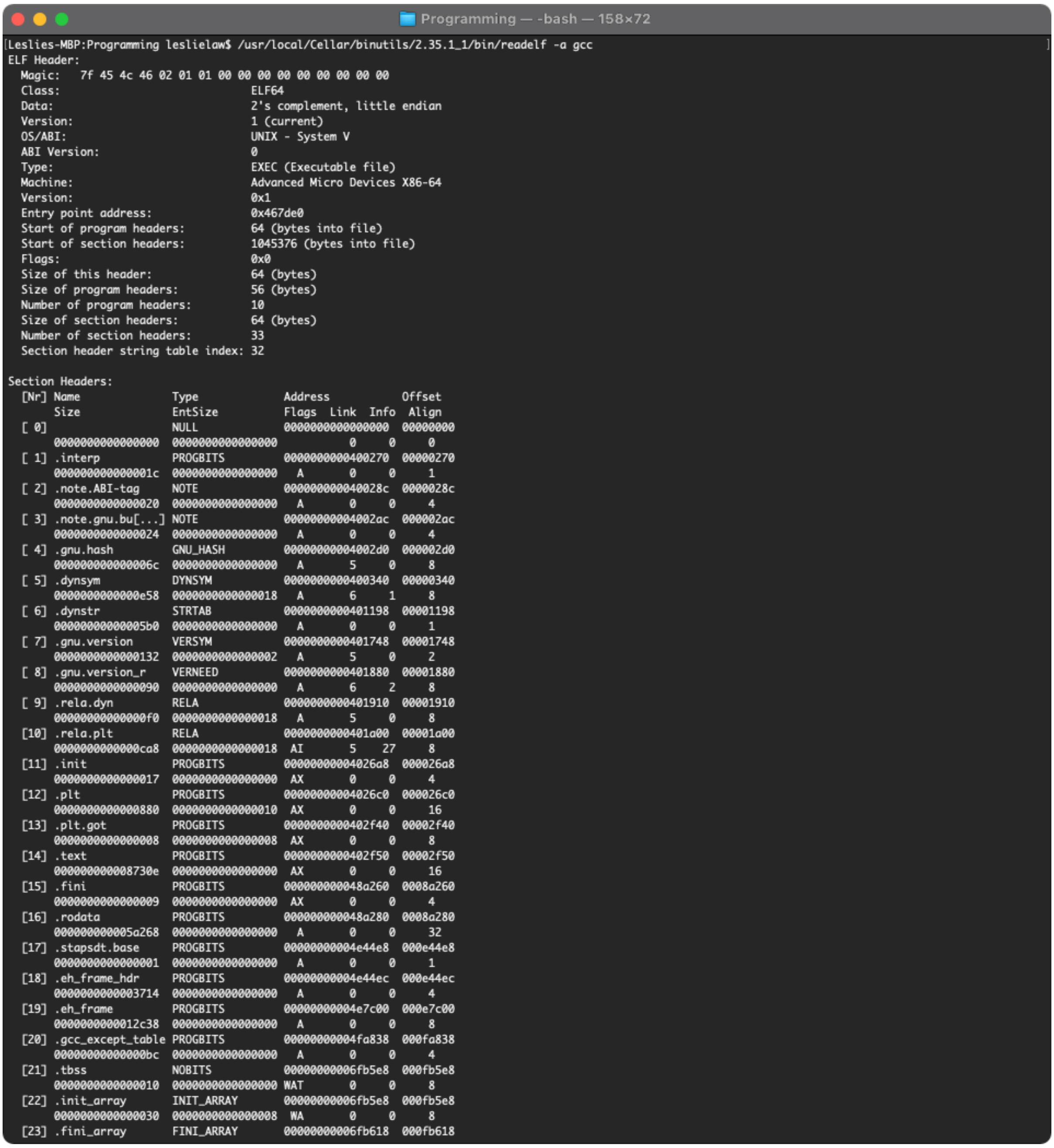
3.3 64-bit RISC-V Linux on QEMU

As the project is worked on macOS environment, a QEMU, a generic and open-source machine emulator, is needed to emulate the RISC-V environment. Booting a Linux on RISC-V QEMU can create an environment to test the customized ELF file since ELF is widely used in Linux and other Linux-like system, and Linux is suitable to run on RISC-V ISA.

4 Implementation

The implementation is doing on a Macbook Pro with macOS 11.4. The test file used is the GCC file from GNU Compiler Toolchain.

4.1 Read ELF on Terminal



```
Programming — -bash — 158×72
Leslies-MBP:Programming leslielaw$ /usr/local/Cellar/binutils/2.35.1_1/bin/readelf -a gcc
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x467de0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          1045376 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         10
  Size of section headers:           64 (bytes)
  Number of section headers:         33
  Section header string table index: 32

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400270  00000270
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             000000000040028c  0000028c
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.bu[...] NOTE             00000000004002ac  000002ac
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         00000000004002d0  000002d0
       000000000000006c  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           0000000000400340  00000340
       0000000000000e58  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000401198  00001198
       00000000000005b0  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           0000000000401748  00001748
       0000000000000132  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000401880  00001880
       0000000000000090  0000000000000000   A       6     2     8
  [ 9] .rela.dyn         RELA             0000000000401910  00001910
       00000000000000f0  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000401a00  00001a00
       0000000000000ca8  0000000000000018  AI       5    27     8
  [11] .init             PROGBITS         00000000004026a8  000026a8
       0000000000000017  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         00000000004026c0  000026c0
       0000000000000880  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000402f40  00002f40
       0000000000000008  0000000000000008  AX       0     0     8
  [14] .text             PROGBITS         0000000000402f50  00002f50
       000000000008730e  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         000000000048a260  0008a260
       0000000000000009  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         000000000048a280  0008a280
       000000000005a268  0000000000000000   A       0     0     32
  [17] .stapsdt.base     PROGBITS         00000000004e44e8  000e44e8
       0000000000000001  0000000000000000   A       0     0     1
  [18] .eh_frame_hdr     PROGBITS         00000000004e44ec  000e44ec
       0000000000003714  0000000000000000   A       0     0     4
  [19] .eh_frame         PROGBITS         00000000004e7c00  000e7c00
       0000000000012c38  0000000000000000   A       0     0     8
  [20] .gcc_except_table PROGBITS         00000000004fa838  000fa838
       00000000000000bc  0000000000000000   A       0     0     4
  [21] .tbss             NOBITS           00000000006fb5e8  000fb5e8
       0000000000000010  0000000000000000 WAT       0     0     8
  [22] .init_array       INIT_ARRAY       00000000006fb5e8  000fb5e8
       0000000000000030  0000000000000008  WA       0     0     8
  [23] .fini_array       FINI_ARRAY       00000000006fb618  000fb618
```

Figure 4.1

Figure 4.1 showing that the function readelf can list out all the information of an ELF file, include ELF header, section header and program header. In the ELF header, the first line showed the magic byte "7f 45 4c 46" which mean "\x7fELF". It also showed the class and version of the ELF format, and the size of the section and program header.

4.2 Read and Write ELF Program

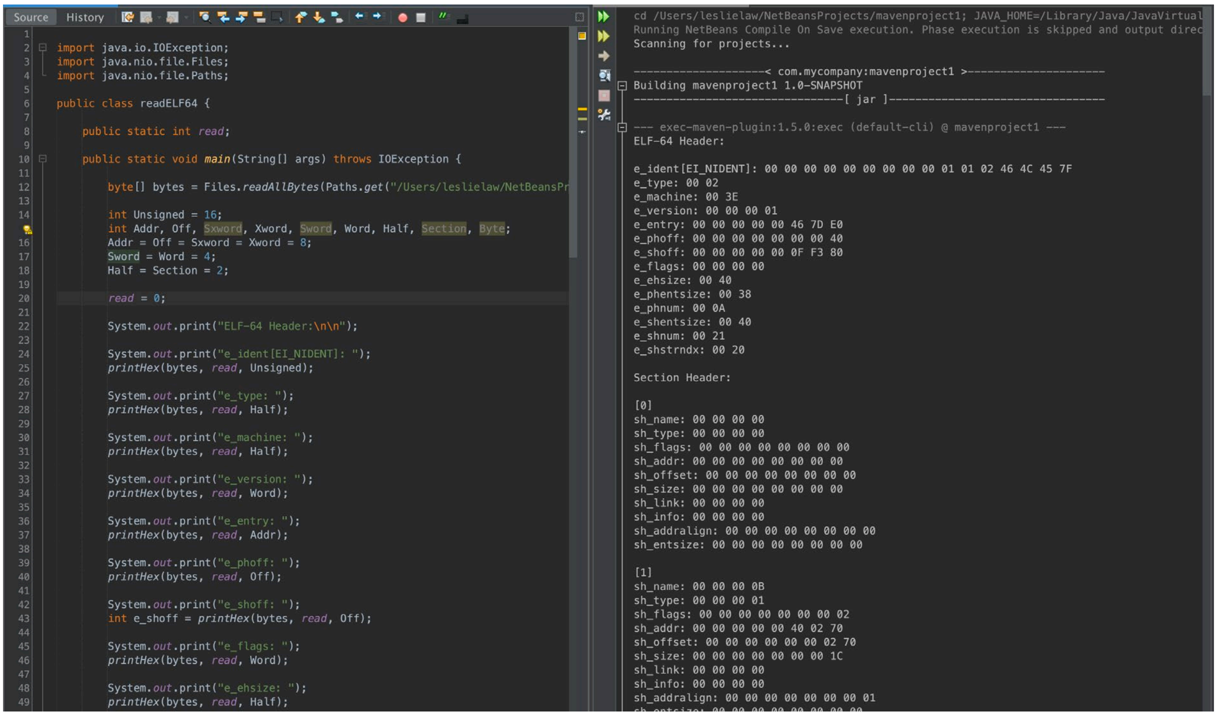
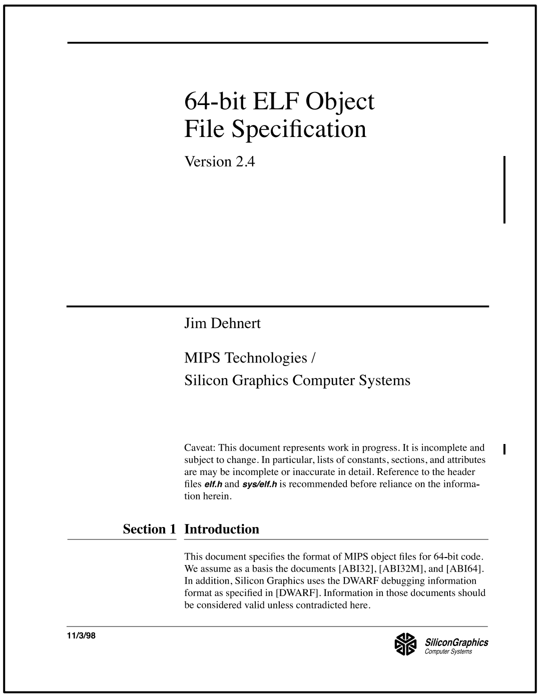


Figure 4.2

Figure 4.2 is the Java program and result of reading ELF files. To read the byte of an ELF file and print in a right way, the program needs to know the length of each field, for example the magic byte is an unsigned data type which has a length of 16 bytes. Those figures can be found on the document called ELF Object File Specification (figure 4.3) by Silicon Graphics. The document showed the data type, header structure and the detail of each field of information (table 4.1).



64-bit ELF Object File Specification

Version 2.4

Jim Dehnert

MIPS Technologies /
Silicon Graphics Computer Systems

Caveat: This document represents work in progress. It is incomplete and subject to change. In particular, lists of constants, sections, and attributes are may be incomplete or inaccurate in detail. Reference to the header files ***elf.h*** and ***sys/elf.h*** is recommended before reliance on the information herein.

Section 1 Introduction

This document specifies the format of MIPS object files for 64-bit code. We assume as a basis the documents [ABI32], [ABI32M], and [ABI64]. In addition, Silicon Graphics uses the DWARF debugging information format as specified in [DWARF]. Information in those documents should be considered valid unless contradicted here.

11/3/98 SiliconGraphics Computer Systems

Figure 4.3

ELF-64 Data Types

Name	Size	Alignment	Purpose
Elf64_Addr	8	8	Unsigned program address
Elf64_Half	2	2	Unsigned small integer
Elf64_Off	8	8	Unsigned file offset
Elf64_Sword	4	4	Signed medium integer
Elf64_Sxword	8	8	Signed large integer

Name	Size	Alignment	Purpose
Elf64_Word	4	4	Unsigned medium integer
Elf64_Xword	8	8	Unsigned large integer
Elf64_Byte	1	1	Unsigned tiny integer
Elf64_Section	2	2	Section index (unsigned)

Table 4.1

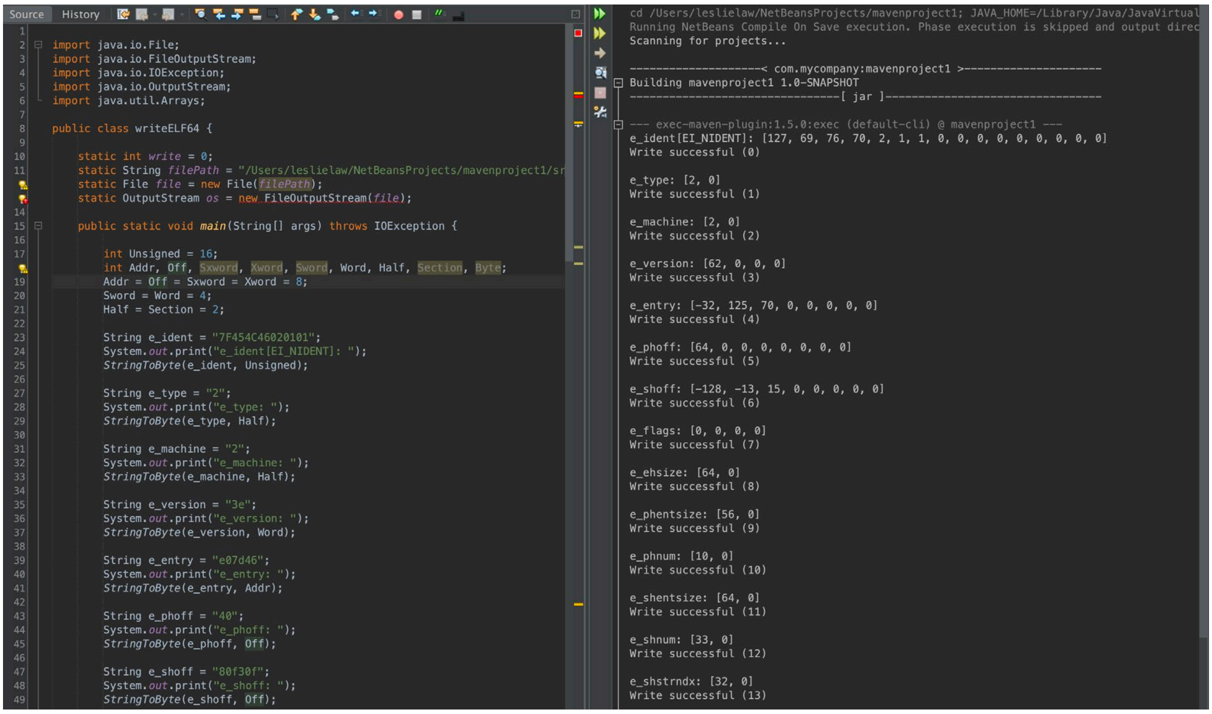


Figure 4.4

Figure 4.4 is the Java program and result of writing ELF files. It provides a way to create a customized ELF file from zero. Same as reading, the program can assign the byte into the correct position in each header which is following the specification sheet on table 4.2.

ELF-64 Header Structure

Field Name	Type	Comments
e_ident[EI_NIDENT]	**unsigned char**	See Table 5
e_type	**Elf64_Half**	See [ABI32]
e_machine	**Elf64_Half**	Machine (**EM_MIPS** = 8)
e_version	**Elf64_Word**	File format version
e_entry	**Elf64_Addr**	Process entry address
e_phoff	**Elf64_Off**	Program header table file offset
e_shoff	**Elf64_Off**	Section header table file offset
e_flags	**Elf64_Word**	Flags — see Table 6
e_ehsize	**Elf64_Half**	ELF header size (bytes)
e_phentsize	**Elf64_Half**	Program header entry size
e_phnum	**Elf64_Half**	Number of program headers
e_shentsize	**Elf64_Half**	Section header entry size
e_shnum	**Elf64_Half**	Number of section headers
e_shstrndx	**Elf64_Half**	Section name string table section header index

Table 4.2

4.3 Section Header in ELF Files

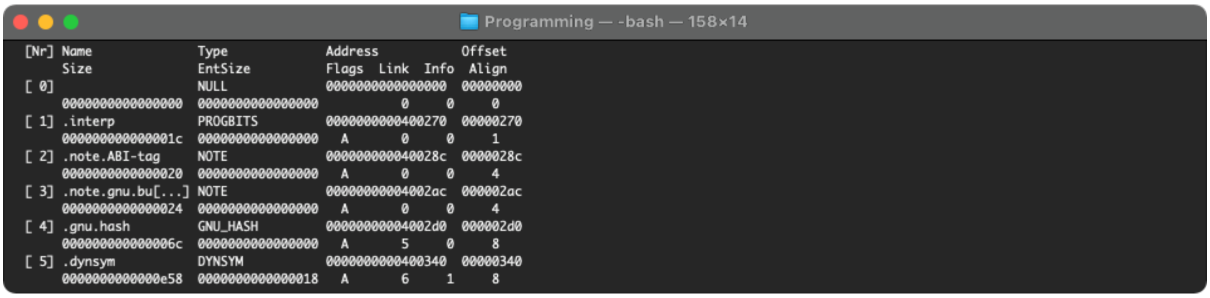


Figure 4.5

Figure 4.5 showing the first 5 fields in the section header. Each field is containing the content to be execute. Changing their order can make the data mapping in memory more efficient. If deleting the unused section, the ELF file can be more light-weighted. After analysis the section header of the GCC file, the writing ELF file program can be used to create the optimized ELF file. The new GCC file is ready for replacing the original one.

4.4 Building RISC-V Environment

Before building a RISC-V environment, Homebrew is needed which is a free and open-source software package management system that simplifies the installation of software on macOS as well as Linux. Figure 4.6 showing the help instruction and update process of Homebrew. Then, installing the RISC-V GNU Compiler Toolchain using Homebrew. Figure 4.7 showing the content of the toolchain.

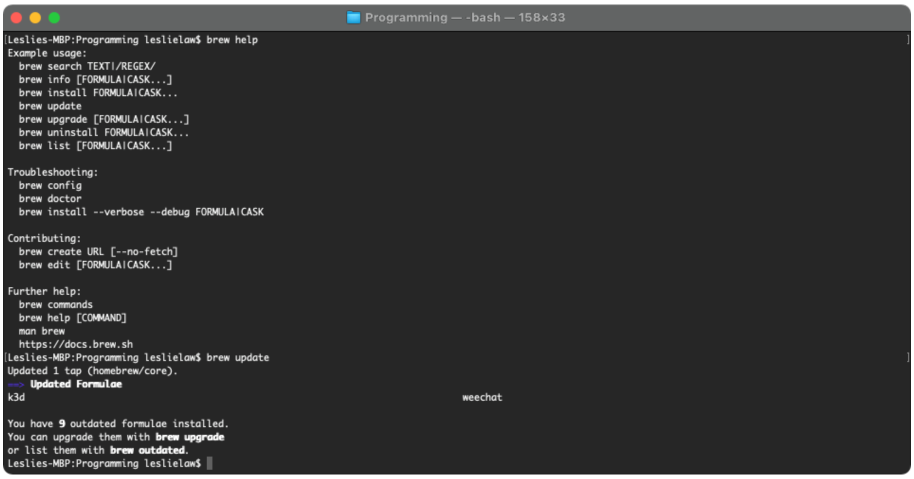


Figure 4.6

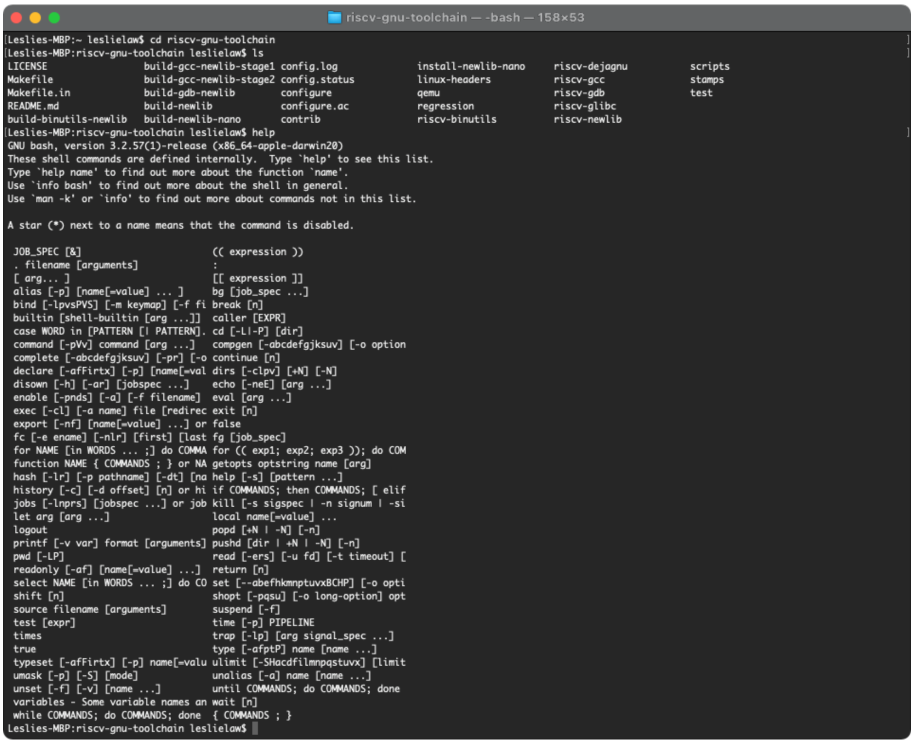


Figure 4.7

While the toolchain as known as the cross compiler is installed form Github, it is ready to build the RISC-V environment and compile the GCC. The time used to compile will be affected by the GCC file.

4.5 Compile GCC

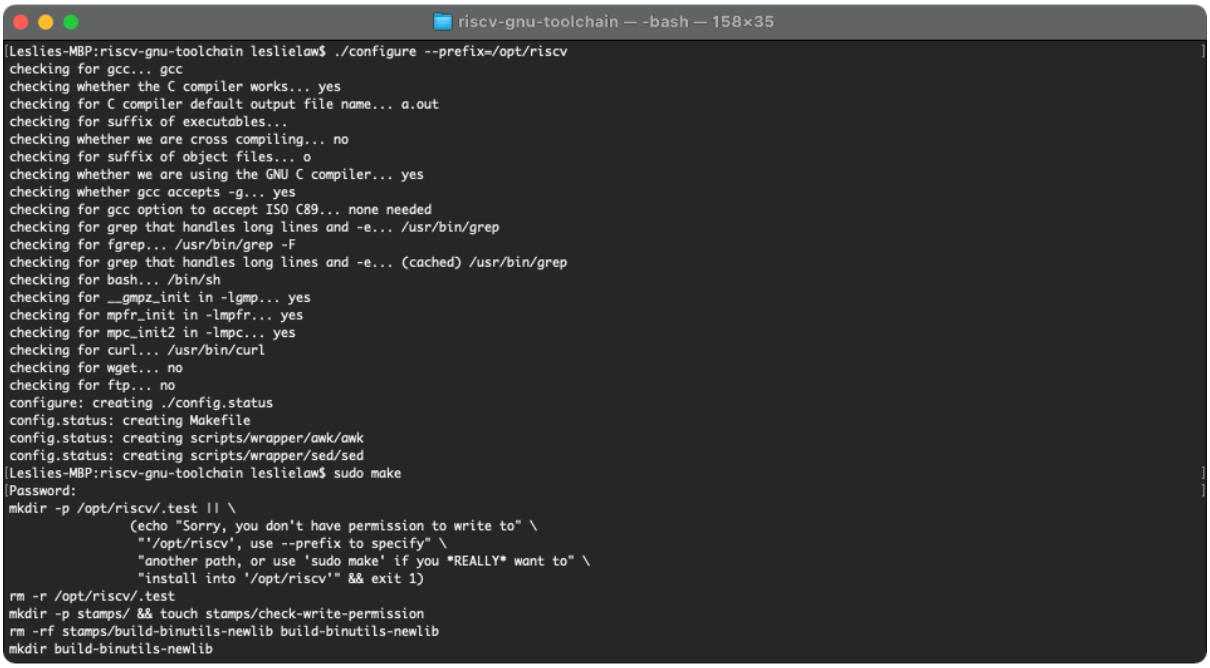


Figure 4.8

Figure 4.8 showing the compile process without editing the GCC file. Once the compiling done, the RISC-V operating environment should be ready to use. The compiling is a long process. After the first compiling done, the entire emulator file needs to be completely removed for another test.

5 Results and Discussion

5.1 GCC Compiling Result

RISC-V	32-bit	64-bit
Original GCC	32 mins 54 sec	44 mins 55 sec
Optimized GCC	30 mins 38 sec	41 mins 32 sec

Table 5.1

Table 5.1 showing the time used in compiling GCC in both 32-bit and 64-bit RISC-V setting with the original one and the optimized one. Since the compile time is very long, the time recorded is not 100% accurate, but the different is enough for comparison. For 32-bit RISC-V, the time used has decreased 6.68% which saved 2 minutes 16 second. For 64-bit RISC-V, the time used has decreased 7.57% which saved 3 minutes 23 second.

The data showed that the section header in an ELF file will affect the file execution order. The section order can be customized for different usage. The time different on a long-time consuming task will be more obvious. The lightweight GCC file make the compile time a bit shorter for building a RSIC-V environment.

6 Conclusion

The result on implementing a new optimized ELF file in RISC-V environment is as expected. The time used on compiling GCC to build RISC-V environment is decreased after using a lightweight ELF file. If doing the same thing to other ELF file for other situation in RISC-V environment, the noticeable improvement may help the future RISC-V development.

RISC-V is really a revolutionary ISA. Since it is still under development, everyone can have an opportunity to learn computer architecture from studying RISC-V. Design and make a CPU or OS is no longer exclusive for leading semiconductor and technology company.

7 References

[1] E. Blem, J. Menon, and K. Sankaralingam, “Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures,” 2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA), 2013.

[2] “GCC and Make,” GCC and Make - A Tutorial on how to compile, link and build C/C++ applications. [Online]. Available: https://www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html. [Accessed: 13-Jun-2021].

[3] J. Shepard, “RISC-V vs. ARM vs. x86 – What's the difference?,” Microcontroller Tips. [Online]. Available: https://www.microcontrollertips.com/risc-v-vs-arm-vs-x86-whats-the-difference/. [Accessed: 13-Jun-2021].

[4] L. Lupori, V. Rosario, and E. Borin, “Towards a High-Performance RISC-V Emulator,” 2018 Symposium on High Performance Computing Systems (WSCAD), 2018.

[5] M. Poorhosseini, W. Nebel, and K. Gruttner, “A Compiler Comparison in the RISC-V Ecosystem,” 2020 International Conference on Omni-layer Intelligent Systems (COINS), 2020.

8 Appendix

[1] RISC-V Toolchain for macOS (Homebrew)
URL: https://github.com/riscv/homebrew-riscv

[2] RISC-V GNU Compiler Toolchain
URL: https://github.com/riscv/riscv-gnu-toolchain

[3] 64-bit ELF Object File Specification (Draft Version 2.5)
URL: https://irix7.com/techpubs/007-4658-001.pdf

[4] Homebrew
URL: https://brew.sh