Optimizing RISC-V Software for Code Density

Version 1.0

© SiFive, Inc.

Optimizing RISC-V Software for Code Density

Proprietary Notice

Copyright © 2019, SiFive Inc. All rights reserved.

Information in this document is provided “as is,” with all faults.

SiFive expressly disclaims all warranties, representations, and conditions of any kind, whether express or implied, including, but not limited to, the implied warranties or conditions of merchantability, fitness for a particular purpose and non-infringement.

SiFive does not assume any liability rising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation indirect, incidental, special, exemplary, or consequential damages.

SiFive reserves the right to make changes without further notice to any products herein.

Release Information

Version	Date	Changes
V1.0	June 19, 2019	• Initial release

Contents

Chapter 1

Optimizing RISC-V Software for Code Density

1.1 Introduction

Code density is a very important metric for many embedded applications. Application code size is highly dependent on how the compiler and linker are used. This Application Note describes how to properly measure code density, as well as common compiler and linker flags to help optimize a RISC-V application for code density.

1.2 How To Check Code Size

The proper way to check an application's code size is to use the toolchain's size tool.

```
riscv64-unknown-elf-size -t libc.a
   text    data     bss     dec     hex filename
 215414    2626    1246  219286   35896 (TOTALS)

riscv64-unknown-elf-size dhrystone.elf
   text    data     bss     dec     hex filename
  69242    4544   10408   84194   148e2 dhrystone
```

The output of the size command tells you about different aspects of your application's code size:

- **text** - The size of the application code, specifically the size of instructions. This segment will need to be saved into non-volatile memory, such as flash.
- **data** - The size of initialized data such as constants or variables which have an initial value other than 0. The initial value of this data must be stored in non-volatile memory, though the actual variables will likely be located in RAM and also consume space there.
- **bss** - Uninitialized data that does not consume non-volatile memory. It doesn't require an initialized value, and is therefore initialized to 0. Bss does consume RAM.
- **dec** - The sum of *text* + *data* + *bss* in decimal.

- **hex** - The sum of *test* + *data* + *bss* in hex.

The total size of code to be stored in non-volatile memory is *text* + *data*. The total size of data to be allocated in RAM is *data* + *bss*.

Binary file size might be much larger than expected, especially when compared to other architectures. This is normal and expected. If you do not use the *riscv64-unknown-elf-size* tool to check code size (TEXT/DATA/BSS), you will see much larger binary file sizes for RISC-V object files due to RISC-V having more relocation information for linker relaxation, and more debug information in generated object files. After fully linking the application, you will see smaller file sizes. For more about linker relaxation, see:

https://www.sifive.com/blog/all-aboard-part-3-linker-relaxation-in-riscv-toolchain

Stripping is another technique used for discarding all the symbols from a binary file. After stripping, you will see a much smaller file size, but the real code size will be unaffected.

```
riscv64-unknown-elf-strip dhrystone -o dhrystone.strip

ls -l
-rwxrwxr-x 1 user user 320220 May 14 00:55 dhrystone
-rwxrwxr-x 1 user user  89388 May 14 00:56 dhrystone.strip

riscv64-unknown-elf-size dhrystone dhrystone.strip
   text    data     bss     dec     hex filename
  85764    2676   10336   98776   181d8 dhrystone
  85764    2676   10336   98776   181d8 dhrystone.strip
```

In the above example, you can see the file size is reduced from 320220 bytes to 89388 bytes after stripping, but the text/data/bss section sizes are unaffected.

The real code size is text size + data size = 85764 + 2676 = 88440 bytes. Note that bss size is not counted toward code size (storage space), but it does consume memory space during runtime.

1.3 Compiler Options That Improve Code Density

The following list of options are helpful in reducing the size of RISC-V applications. Applying compiler optimizations targeting code density can affect performance. It is always recommended to check if the code size vs performance trade-off is acceptable for your application.

Compiler and linker options:

- Enable Link Time Optimizations by applying ***-flto-***
    - This optimization is application-dependent and may increase the code size of some applications.
- Section garbage collection: ***-ffunction-sections*** and ***-fdata-sections***

    - The linker depends on information provided by the compiler to remove unused code
        - ***ffunction-sections*** - instructs gcc to put each function into its own section in the object file
        - ***fdata-sections*** - instructs gcc to put data items into its own section in the object file
        - ***Wl,--gc-sections*** - instructs the linker to remove all unused sections. Requires ***ffunction-sections***, ***fdata-sections***.
- Use global pointer and linker relaxation with ***-mrelax*** (enabled by default)
    - See https://www.sifive.com/blog/all-aboard-part-3-linker-relaxation-in-riscv-toolchain
- Enable code size optimization by ***-Os***, which also enable ***-finline-functions***, ***-fthread-jumps***
    - See https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
- Use the correct code model ***-mcmodel=medlow*** or **mcmodel=*medany***
    - ***mcmodel=medany*** is bigger than ***mcmodel=medlow***.
    - For RV32 implementations with less than 2GB of memory ***-mcmodel=medlow*** is suggested
    - See https://www.sifive.com/blog/all-aboard-part-4-risc-v-code-models
- Try ***-mexplicit-relocs***
    - See https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html#RISC-V-Options
- Try ***-msave-store***
    - See https://gcc.gnu.org/onlinedocs/gcc/RISC-V-Options.html#RISC-V-Options
- Linking smaller libraries
    - Use newlib-nano ***--specs=nano.specs***

Note that when applying compiler optimizations, order matters when multiple settings of the same flag are used, as the final option specified will determine the compiler operation performed. For example, consider the following option settings: "***-Os -fxxx -fyyy…. -O3***". The ***-O3*** will supercede ***-Os***, resulting in a binary that will not be optimized for code size.