RISC-V Large Code Model Software Workaround

Version 1.0

RISC-V Large Code Model Software Workaround

Proprietary Notice

Release Information

Version	Date	Changes
V1.0	September 9, 2019	• Initial release

Contents

Chapter 1

RISC-V Large Code Model Software Workaround

1.1 Introduction

RISC-V code models are used when building a software program, and they define a method to generate instruction combinations to access global symbols. There are two types of code models currently used on RISC-V architectures: `-mcmodel=medlow` and `-mcmodel=medany`. Both code models restrict the generated **code** to be 2GiB or less. Additionally, both code models require that **global symbols** reside within +/- 2GiB (32-bit signed offset) from the generated code. The below example shows how this range works on 32-bit architectures, using x0-relative addressing.

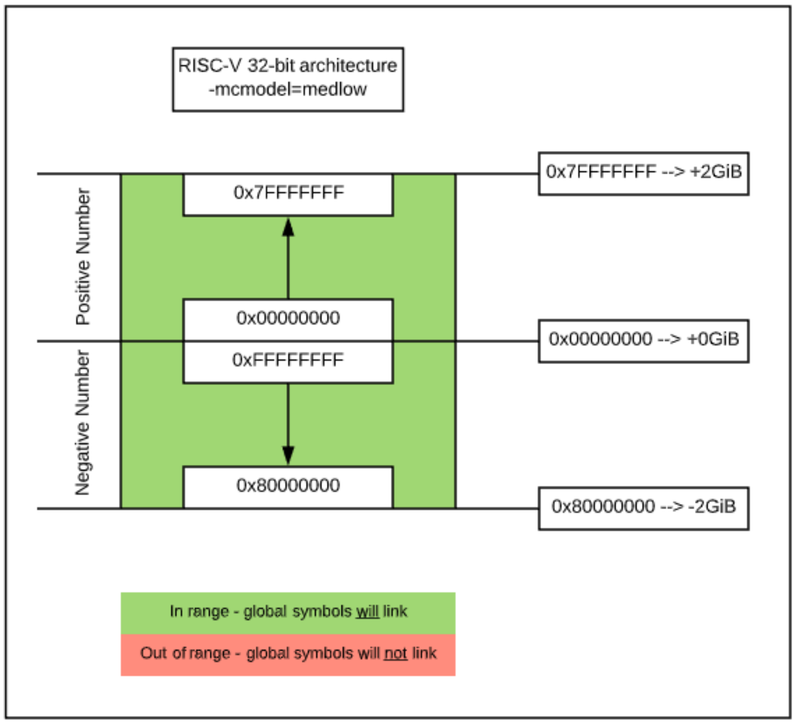

Figure 1: 32-bit architecture addressing using -mcmodel=medlow

For 32-bit architectures using `-mcmodel=medlow`, the full 32-bit address range is accessible for compiling and linking your program.

For 64-bit architectures which use `-mcmodel=medany`, **code** can be linked at any base address, but linked global symbols follow a similar +/- 2GiB range restriction. Depending on the code base address, there are scenarios where the negative range is generally not used.

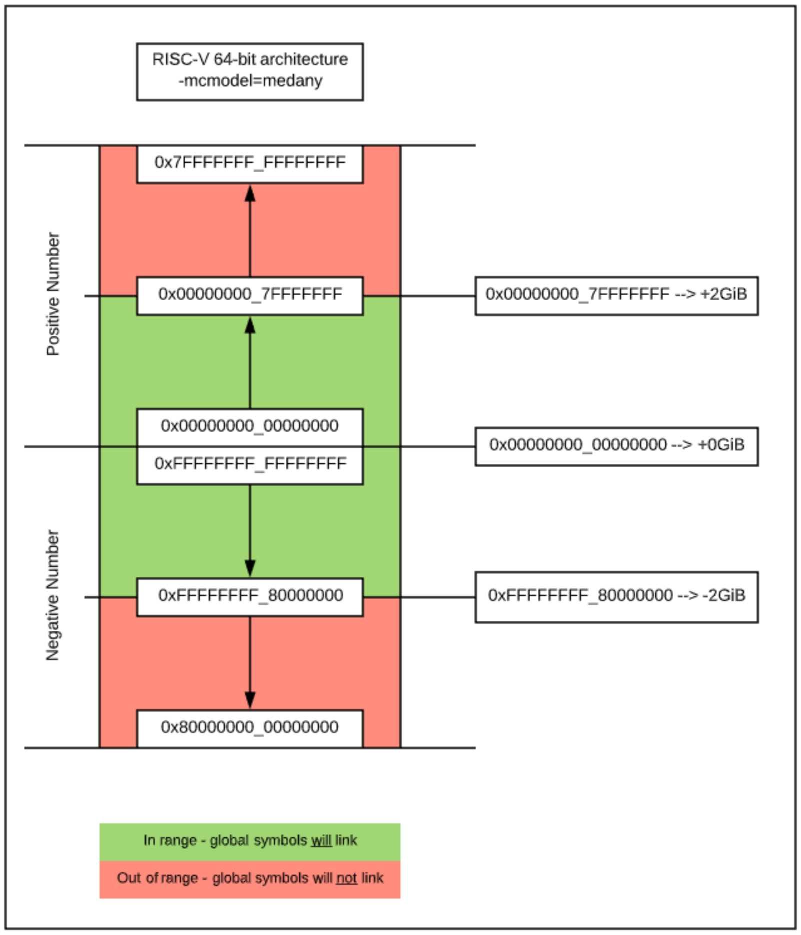
RISC-V 64-bit architecture
-mcmodel=medany
0x7FFFFFFF_FFFFFFFF
0x00000000_7FFFFFFF
0x00000000_00000000
0xFFFFFFFF_FFFFFFFF
0xFFFFFFFF_80000000
0x80000000_00000000
Positive Number
Negative Number
0x00000000_7FFFFFFF --> +2GiB
0x00000000_00000000 --> +0GiB
0xFFFFFFFF_80000000 --> -2GiB
In range - global symbols will link
Out of range - global symbols will not link

Figure 2: 64-bit architecture addressing using -mcmodel=medany

In Summary:

- `-mcmodel=medlow` — Used for 32-bit architectures, this code model requires that the program and its statically defined symbols must lie between the absolute addresses -2 GiB and +2 GiB, which covers the full 32-bit address range. Code is generally linked around address 0x00000000, and instruction pairs `lui` and `ld` are used to generate addresses for global symbols.
- `-mcmodel=medany` — Used for 64-bit architectures, this code model also generates a 32-bit signed offset to refer to global symbols. Linked **code** can reside at any address, and instruction pairs `auipc` and `ld` are used to generate global symbol addresses in a +/- 2GiB window from the **code** area.

Both code models limit the address range where linked symbols can reside, but this is not an issue for 32-bit architectures. The only limit this introduces is on 64-bit architectures using `-mcmodel=medany`, where there is a requirement for linked symbols to be outside of +/- 2GiB from the **code** area.

Note

As an optimization, if an address is out of range of `auipc`, but within range of zero-base addressing, then the linker will convert the auipc to an lui. This is primarily to make undefined weak references work, as these will always be zero at the end of linking, which is generally not in range of `auipc` when `-mcmodel=medany` is used.

There is a lot more detail, and a precise description of RISC-V code models in this blog post: https://www.sifive.com/blog/all-aboard-part-4-risc-v-code-models.

1.2 Linker File Options

Before we dig into an example it is worthwhile to note the different linker combinations that are available for SiFive's cores in the freedom-e-sdk repository. The options are selected by using the LINK_TARGET option, described in the repository README file.

- **metal.default.lds** — This linker option maps code and data to Flash, with the exception of uninitialized RAM section .bss, and stack and heap, which always reside in RAM
- **metal.ramrodata.lds** — This linker option maps read-only data sections .rodata and .rdata to RAM, in addition to .bss, stack, and heap, placing executable code in flash section
- **metal.scratchpad.lds** — This linker option maps everything to RAM, flash is not used

1.2.1 Standard Linker Option

The linker file example below represents an example memory map configuration for some of SiFive's 32-bit and 64-bit standard cores.

```
MEMORY
{
    ram (wxa!ri) : ORIGIN = 0x80000000, LENGTH = 0x4000
    flash (rxai!w) : ORIGIN = 0x20400000, LENGTH = 0x1fc00000
}
```

It should be noted that the MEMORY section layout is the same for most linker file options listed previously. The difference in how a compiled program is organized, and whether the defined regions are actually used, is controlled by the SECTIONS and PHDRS areas of the linker file.

```
PHDRS
{
        flash PT_LOAD;
        ram_init PT_LOAD;
        itim_init PT_LOAD;
        ram PT_NULL;
        itim PT_NULL;
}
```

```
SECTIONS
{
        __stack_size = DEFINED(__stack_size) ? __stack_size : 0x400;
        PROVIDE(__stack_size = __stack_size);
        __heap_size = DEFINED(__heap_size) ? __heap_size : 0x400;
        PROVIDE(__metal_boot_hart = 0);
        PROVIDE(__metal_chicken_bit = 0);


        .init                 :
        {
                KEEP (*(.text.metal.init.enter))
                KEEP (*(SORT_NONE(.init)))
                KEEP (*(.text.libgloss.start))
        } >flash AT>flash :flash


        .text                 :
        {
                *(.text.unlikely .text.unlikely.*)
                *(.text.startup .text.startup.*)
                *(.text .text.*)
                *(.itim .itim.*)
                *(.gnu.linkonce.t.*)
        } >flash AT>flash :flash

        ...
```

The above example shows a small piece of metal.default.lds, and does not load any executable code into `ram` or `itim` since it uses `PT_NULL`.

This standard memory map will build correctly for different combinations of code and data sections being mapped to either ram or flash. The code model description can be found in **settings.mk** which is located in the **/bsp** path for your SiFive project. For example, when represented as `RISCV_CMODEL=medlow` in **settings.mk**, this gets translated to `-mcmodel=medlow` through the build (Makefile) process.

1.2.2 Example of Invalid Configuration

Let's look at an example using **64-bit** architecture where there is a requirement to have a memory port at address 0x1_0000_0000, and we want linked symbols to reside there. If our code base remains at address 0x20400000, then the updated linker map will look like this:

```
MEMORY
{
    ram (wxa!ri) : ORIGIN = 0x100000000, LENGTH = 0x4000
    flash (rxai!w) : ORIGIN = 0x20400000, LENGTH = 0x1fc00000
}
```

This configuration will not build correctly since linked symbols will reside in the ram section, which is greater than 2 GiB from the flash section where code is mapped. If this memory map is used on a 64-bit architecture using `-mcmodel=medany`, it will not link successfully. Generated errors will contain the following phrase:

relocation truncated to fit:

The error message will contain a lot of other detail that we will omit for simplicity.

There is currently no method for a RISC-V software program to generate the instruction combinations to access global symbols outside of a 2GiB window. This applies to all current code models available. There is development underway to address this limitation, and support will be provided in what will be called a **large code model**. Until then, the workaround is in the form of software pointers.

1.2.3 Workarounds

Example #1

Even if global symbols cannot be linked with the toolchain, we can still access custom data sections using 64-bit software pointers. The following example creates a pointer to access memory at location 0x1_0000_0000:

```
// Create defines for new memory region
#define LARGE_DATA_SECTION_ADDRESS 0x100000000
#define LARGE_DATA_SECTION_SIZE_IN_BYTES 0x4000
#define DWORD_SIZE 8

int main(void)
{
    /****************************************************/
    /* Example #1 - use pointer to access 64-bit region */
    /****************************************************/
    uint32_t idx;
    uint64_t *data_array, addr;

    data_array = (uint64_t *)LARGE_DATA_SECTION_ADDRESS;
    for (addr = 0, idx = 0; \
            addr < LARGE_DATA_SECTION_SIZE_IN_BYTES; \
                    addr += DWORD_SIZE, idx++) {

            // Simply writing data to our region outside of 32-bit range
            data_array[idx] = addr;
    }
}
```

Example #2

Here we use an existing freedom-metal data structure to define a new region along with an available API to access attributes of the region.

```
// *************************************************************
// Create new structure to define a new memory region
// This method leverages existing freedom-metal struct format
// *************************************************************

#include <metal/memory.h>   // required for data struct

// Create defines for new memory region
#define LARGE_DATA_SECTION_ADDRESS 0x100000000
#define LARGE_DATA_SECTION_SIZE_IN_BYTES 0x4000
#define DWORD_SIZE 8

// Create our struct using existing metal_memory type in freedom-metal
const struct metal_memory large_data_mem_struct;
const struct metal_memory large_data_mem_struct = {
    ._base_address = LARGE_DATA_SECTION_ADDRESS,
    ._size = LARGE_DATA_SECTION_SIZE_IN_BYTES,
    ._attrs = {
        .R = 1,
        .W = 1,
        .X = 0,
        .C = 1,
        .A = 0},
};

int main(void)
{
    // Example #2 - Use structure which defines 64-bit addressable regions,
    // using existing structure type to define base addr,
    // size, and permissions

    size_t _large_data_size;
    uintptr_t _large_data_base_addr;
    int _atomics_enabled, _cachable_enabled;
    uint64_t *large_data_array;

    _large_data_base_addr = \
            metal_memory_get_base_address(&large_data_mem_struct);
    _large_data_size = metal_memory_get_size(&large_data_mem_struct);
    _atomics_enabled = metal_memory_supports_atomics(&large_data_mem_struct);
    _cachable_enabled = metal_memory_is_cachable(&large_data_mem_struct);

    large_data_array = (uint64_t *)_large_data_base_addr;

    // Access our new memory region
    // large_data_array[x] = 0x0;
    // ... add functional code ...
    return 0;
}
```

This example can be used if multiple data regions are required with different attributes. Once the base address is assigned within the required data struct, then pointers can be used to access memory, similar to Example #1 above. The existing struct and API format allows for multiple regions to be created easily.

1.3 Conclusion

We can simply generate pointers to create our own global memory regions, which don’t require dependence on the RISC-V toolchain. Additionally, SiFive’s freedom-metal API provides a method to define and manage the base address, size, and attributes for these custom defined global memory regions until a large code model is available for RISC-V.