Creating A Boot Program in RISC-V

Ask Question

Asked 4 years, 8 months ago Modified 3 years, 9 months ago Viewed 8k times

7

I am trying to create a boot program for RISC-V based boards. I am following this guide, and adapting it for riscv. osdev

The problem I'm having is translating this instruction. `times 510 -( $ - $$ ) db 0`

The best I could think of is to just fill 63ish lines of this `.8byte 0` But this doesn't seem very feasible.

Here is the Full code.

```
###################################
########### Boot Init ############
###################################

.section .text

start:                          # begins the program
    nop                         # the do nothing instruction
    j start                     # loops back to start

# Todo:: figure out the size of the np and j instruction
# The intent of this portion is to fill the remaning bin file with 0's up until t

.section .rodata
    .4byte 0                    # size of start instructions + this

    .8byte 0                    # begins the zero's, currently 510 byte
    .8byte 0
     # repeat 60ish times

    .8byte 0x0000000000aa55     # fills the last two bytes with the universal
                                # 55aa to indicate boot program
```

EDIT

I am using the gcc toolchain for risc. Found here. I am using the `.rept` directive.

> Here is the updated code.

```
###################################
########### Boot Init ############
###################################

start:                          # begins the program
    nop                         # the do nothing instruction
```

```
    j start                     # loops back to start

# Todo:: figure out the size of the np and j instruction
# The intent of this portion is to fill the file with 0's up until the last few b

    .rept 63
    .8byte 0
    .endr

    .4byte 0                    # size of start instructions + this

    .8byte 0                    # begins the zero's, currently 510 byte
    .8byte 0

    .8byte 0×0000000000aa55     # fills the last two bytes with the universal
                                # 55aa to indicate boot program
```

The hex dump is below:

```
00000000  01 00 fd bf 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000010  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000210  55 aa 00 00 00 00 00 00                           |U.......|
00000218
```

Here is where I can see that I messed up the endianness of the code obviously. However, I have a new question. What the heck is going on with the left column of the hex dump?? I understand that the * means its filled to 0. However the line goes 0 to 10 then from 210 to 218 why did it increase by 10 first, then 8 at the end? why do i have a blank line (218)?

EDIT No need to tell me about the line numbering, I realized now its hex. So the last question remains. How do I indicate to this board that the program is a boot program. Is there a magic number? I could not find any indication on any of their documentation.

assembly riscv

Share Follow

edited Oct 27, 2019 at 18:13

asked Oct 26, 2019 at 21:36

christopher clark

2,053 ● 5 ● 31 ● 50

First, find out that the signature thing is even applicable to your platform. Second, make sure your toolchain even supports multiple sections for a boot sector. If it does, you can probably just make a separate one for the signature and tell the linker to place it at the proper offset. Alternatively, consult your assembler's documentation about fill or repeat directives. GNU assembler for example supports `.org 510` – Jester Oct 26, 2019 at 21:44

I couldn't seem to find where the makers of the board discuss the details of the signature (here sifive.cdn.prismic.io/…). I'm also not sure what you meant by `Second, make sure your`

`toolchain even supports multiple sections for a boot sector` . – christopher clark Oct 26, 2019 at 21:57

This is the actual board I have. Apologies. sifive.cdn.prismic.io/… – christopher clark Oct 26, 2019 at 22:03

You used `.section .text` and `.section .rodata` ... verify that your toolchain simply puts them directly one after another (unlikely). – Jester Oct 26, 2019 at 22:05

2 Yeah, doesn't sound like it searches for a `55 AA` signature at the end of the first 512-byte sector of a disk / block-device at all. You just flash your code into the right address and execution jumps there. – Peter Cordes Oct 27, 2019 at 18:58

Show **4** more comments

2 Answers

Sorted by: Highest score (default)

17

I have an original hifive1 board. For the original board the getting started guide says this:

The HiFive1 Board is shipped with a modifiable boot loader at the beginning of SPI Flash (0x20000000). At the end of this program's execution the core jumps to the main user portion of code at 0x20400000.

For the rev b board it says this:

The HiFive1 Rev B Board is shipped with a modifiable boot loader at the beginning of SPI Flash (0x20000000). At the end of this program's execution the core jumps to the main user portion of code at 0x20010000.

Both chips show 0x80000000 for ram and 0x20000000 for (external) flash. Assume that is the interface where they put the flash on the rev B board.

First program.

novectors.s

```
.globl _start
_start:
    lui x2,0x80004
    jal notmain
    sbreak
    j .

.globl dummy
dummy:
    ret
```

notmain.c

```
void  dummy ( unsigned int );
int notmain ( void )
```

```
{
    unsigned int ra;

    for(ra=0;;ra++) dummy(ra);
    return(0);
}
```

memmap

```
MEMORY
{
    ram : ORIGIN = 0x80000000, LENGTH = 0x4000
}
SECTIONS
{
    .text : { *(.text*) } > ram
    .rodata : { *(.rodata*) } > ram
    .bss : { *(.bss*) } > ram
}
```

build

```
riscv32-none-elf-as -march=rv32i -mabi=ilp32 novectors.s -o novectors.o
riscv32-none-elf-gcc -march=rv32i -mabi=ilp32 -Wall -O2 -nostdlib -nostartfiles -
riscv32-none-elf-ld novectors.o notmain.o -T memmap -o notmain.elf
riscv32-none-elf-objdump -D notmain.elf > notmain.list
riscv32-none-elf-objcopy notmain.elf -O binary notmain.bin
```

In theory you can use riscv32-whatever-whatever (riscv32-unknown-elf, etc). As this code is generic enough. Also note I am using the minimal rv32i, you can probably use rv32imac.

Check the disassembly:

```
Disassembly of section .text:

80000000 <_start>:
80000000:   80004137            lui x2,0x80004
80000004:   010000ef            jal x1,80000014 <notmain>
80000008:   00100073            ebreak
8000000c:   0000006f            j   8000000c <_start+0xc>

80000010 <dummy>:
80000010:   00008067            ret

80000014 <notmain>:
80000014:   ff010113            addi    x2,x2,-16 # 80003ff0 <notmain+0x3fdc>
80000018:   00812423            sw  x8,8(x2)
8000001c:   00112623            sw  x1,12(x2)
80000020:   00000413            li  x8,0
80000024:   00040513            mv  x10,x8
80000028:   fe9ff0ef            jal x1,80000010 <dummy>
```

```
8000002c:   00140413            addi    x8,x8,1
80000030:   ff5ff06f            j   80000024 <notmain+0x10>
```

Being rv32i it is all 32 bit instructions and that is fine. This program is intended to be loaded into ram and run there with a debugger, I use openocd and telnet in.

```
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Open On-Chip Debugger
```

Then

```
halt
load_image notmain.elf
resume 0x80000000
```

in the telnet window.

Then you can halt again.

```
80000024:   00040513            mv  x10,x8
80000028:   fe9ff0ef            jal x1,80000010 <dummy>
8000002c:   00140413            addi    x8,x8,1
80000030:   ff5ff06f            j   80000024 <notmain+0x10>
```

You can examine either x8 or x10 to see that it counted:

```
resume
halt
```

and examine the registers again they should have incremented. First program running, moving on.

Second program use this linker script instead:

memmap

```
MEMORY
{
    rom : ORIGIN = 0x20010000, LENGTH = 0x4000
    ram : ORIGIN = 0x80000000, LENGTH = 0x4000
}

SECTIONS
{
    .text : { *(.text*) } > rom
    .rodata : { *(.rodata*) } > rom
```

```
    .bss : { *(.bss*) } > ram
}
```

examine disassembly.

```
Disassembly of section .text:

20010000 <_start>:
20010000:   80004137            lui x2,0×80004
20010004:   010000ef            jal x1,20010014 <notmain>
20010008:   00100073            ebreak
2001000c:   0000006f            j   2001000c <_start+0×c>

20010010 <dummy>:
20010010:   00008067            ret

20010014 <notmain>:
20010014:   ff010113            addi    x2,x2,-16 # 80003ff0 <notmain+0×5fff3fdc>
20010018:   00812423            sw  x8,8(x2)
2001001c:   00112623            sw  x1,12(x2)
20010020:   00000413            li  x8,0
20010024:   00040513            mv  x10,x8
20010028:   fe9ff0ef            jal x1,20010010 <dummy>
2001002c:   00140413            addi    x8,x8,1
20010030:   ff5ff06f            j   20010024 <notmain+0×10>
```

It appears to be position independent so it should have just worked as was with the other linker script but best to use the correct addresses.

My notes say:

```
flash protect 0 64 last off
program notmain.elf verify
resume 0×20010000
```

And now you should be able to reset or power cycle the board, connect with openocd in a way that doesn't reset (or does if you wish) and then you don't need to load anything it should have run their bootloader that then launched your bootloader at that address (jumped to it as they mention). Examine r8 or r10 (r10 for this abi is the first parameter passed, so even if your gcc builds using something other than r8, r10 should still reflect the counter) resume, halt, reg, resume, halt, reg ...

Before overwriting their bootloader at 0x20000000 I would dump it and make sure you have a good copy of it, and or perhaps they have a copy on their website. Then you can change the linker script to 0x20000000. Before I would do that personally I would disassemble and examine their bootloader and find out what if anything it is doing, is it worth keeping, etc. Their text says "modifiable"

I cut my risc-v teeth on the hifive1 board, but have moved on to sim open source cores, the hifive boards are pretty expensive. I also made a minimal pcb and put down some sifive parts, was going to only run out of ram, etc, but my board was too minimal and I didn't go back and try again, little support on their forums for pcb work and their docs left something to be desired.

The point being there are a number of cores out there that you can sim with verilator or other and see everything going on, and you can't brick nor let smoke out because it is a sim.

Note rv32ic

```
riscv32-none-elf-as -march=rv32ic -mabi=ilp32 novectors.s -o novectors.o
riscv32-none-elf-gcc -march=rv32ic -mabi=ilp32 -Wall -O2 -nostdlib -nostartfiles
riscv32-none-elf-ld novectors.o notmain.o -T memmap -o notmain.elf
riscv32-none-elf-objdump -D notmain.elf > notmain.list
riscv32-none-elf-objcopy notmain.elf -O binary notmain.bin
```

and you can see it uses the compressed instructions where it can

```
20010000 <_start>:
20010000:   80004137            lui x2,0x80004
20010004:   00a000ef            jal x1,2001000e <notmain>
20010008:   9002                    ebreak
2001000a:   a001                    j   2001000a <_start+0xa>

2001000c <dummy>:
2001000c:   8082                    ret

2001000e <notmain>:
2001000e:   1141                    addi    x2,x2,-16
20010010:   c422                    sw  x8,8(x2)
20010012:   c606                    sw  x1,12(x2)
20010014:   4401                    li  x8,0
20010016:   8522                    mv  x10,x8
20010018:   3fd5                    jal 2001000c <dummy>
2001001a:   0405                    addi    x8,x8,1
2001001c:   bfed                    j   20010016 <notmain+0x8>
```

Also it is pretty easy to write your own emulator. Depends on how you want to stage learning this platform. How much is mastering the instruction set vs the toolchain vs a specific chip and its peripherals.

You definitely want the risc-v documents from riscv.org that match the version supported by the core, lots of internal core registers and stuff plus the instruction set(s). As well as the getting started and the chip doc for the chip in question, if you want to do your own thing. If you want to play in one of their sandboxes and use some third party libraries, then you need to learn their sandbox and play in their sandbox rather than do your own thing. Looks like you are wanting to do your own thing.

Note I am using the current version of gcc/binutils from gnu mainline sources, hand built.

```
riscv32-none-elf-gcc --version
riscv32-none-elf-gcc (GCC) 9.2.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

riscv32-none-elf-as --version
GNU assembler (GNU Binutils) 2.32
Copyright (C) 2019 Free Software Foundation, Inc.
This program is free software; you may redistribute it under the terms of
the GNU General Public License version 3 or later.
This program has absolutely no warranty.
This assembler was configured for a target of `riscv32-none-elf'.
```

The above code worked fine years ago against the original hifive1 and this style tends to work for major revs of gnu and I have used this toolchain against other riscv cores, so even if yours is older it should still work. The most important thing is matching the arch (-march) to the instruction sets supported by the core, or at least a subset rv32i should be supported by all cores, compressed and multiply and such are not always supported.

My openocd config file for the first board

```
adapter_khz     10000

interface ftdi
ftdi_device_desc "Dual RS232-HS"
ftdi_vid_pid 0x0403 0x6010

ftdi_layout_init 0x0008 0x001b
ftdi_layout_signal nSRST -oe 0x0020 -data 0x0020

set _CHIPNAME riscv
jtag newtap $_CHIPNAME cpu -irlen 5 -expected-id 0x10e31913

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position $_TARGETNAME
$_TARGETNAME configure -work-area-phys 0x80000000 -work-area-size 10000 -work-are

flash bank onboard_spi_flash fespi 0x20000000 0 0 0 $_TARGETNAME
init
```

openocd -f riscv.cfg in one terminal/window then telnet localhost 4444 in another.

Now as far as the gnu assembler nuances you are asking about see the gnu assembler, or even better use as little assembler/toolchain specific stuff as you can as it may change and/or you may change tools some day. YMMV

The gnu tools don't know this board from a hole in the wall. you tell the gnu tools about the processor core architecture and in the linker script the memory map. Your code, directly or

indirectly (if you use someone elses bootstrap and linker script) must match the boot properties of the processor core be it a risc-v from sifive or some arm core or mips or x86, etc.

Vector table or not, execute at some address, etc. In the above case their bootloader jumps to 0x20010000 so you need to put the first instruction at 0x20010000 which is done by having that instruction be the first one in the bootstrap source, and if not specified in the linker script by having that object first on the ld command line, and by examining the disassembly to confirm it worked before ever attempting to run it on the hardware.

The riscv cores I have used don't have a vector table, for reset they simply start execution at some address. So you would use the same approach if you didn't have a pre-bootloader jump to you. For other architectures not risc-v the construction of the program for the board/platform would vary if it is a jump to an address thing vs a vector table thing.

Now saying that, if you are using their sandbox then this is a sandbox question not a gnu toolchain question.

It is in their documentation the board documentation and/or website indicates that the rev b board uses the FE310-G002 chip in the FE310-G002 documentation you find the memory map. It also indicates this is a risc-v architecture and from that you go to the riscv.org foundation and get the documentation for that architecture which tells you how it boots. And back in the FE310-G002 it tells you the boot process from the MSEL pins. Which you would need to examine the schematics. So the reality is their documentation does tell you how to indicate that this is a bootloader program, by providing the information you need to give to gnu.

Saying that...some experimenting is desired/required. It is possible/easy to write a simple position infinite loop, build for 0x00000000 but load at 0x20010000 based on their documentation and come in with openocd to examine the program counter to see if it really is 0x20010000 based. From that you can assume that ultimately as shipped the board works its way through their bootloader into yours through whatever MSEL selection.

Hmmm:

> On power-on, the core’s reset vector is 0x1004 .

And it goes further to indicate the different first instruction addresses for each of the MSEL strap options. So if you were to take over their bootloader and replace it with your own based on the documentation you would link for 0x20000000 and have the entry point there.

Edit

Just got my rev b board.

You can look at the getting started guide to see how to specify the board using their sandbox. But that is not required, if you have a (gnu) toolchain that supports rv32i or more than that

rv32imac you can build programs with no other outside dependencies.

The toolchain itself doesn't know one board from another, one chip from another.

The sifive documentation says:

The HiFive1 Rev B Board is shipped with a modifiable boot loader at the begnning of SPI Flash (0x20000000). At the end of this program’s execution the core jumps to the main user portion of code at 0x20010000.

And that is the critical information we need, plus the address space for memory in the memory map for the part 0x80000000 0x4000 bytes of sram.

novectors.s

```
.globl _start
_start:
    lui x2,0x80004
    jal notmain
    j .

.globl dummy
dummy:
    ret

.globl PUT32
PUT32:
    sw x11,(x10)
    ret

.globl GET32
GET32:
    lw x10,(x10)
    ret
```

notmain.c

```
void PUT32( unsigned int, unsigned int);
unsigned int GET32 ( unsigned int );
void  dummy ( unsigned int );

#define GPIOBASE 0x10012000
#define GPIO_VALUE          (GPIOBASE+0x00)
#define GPIO_INPUT_EN       (GPIOBASE+0x04)
#define GPIO_OUTPUT_EN      (GPIOBASE+0x08)
#define GPIO_PORT           (GPIOBASE+0x0C)
#define GPIO_PUE            (GPIOBASE+0x10)
#define GPIO_OUT_XOR        (GPIOBASE+0x40)

int notmain ( void )
{
    unsigned int rx;

    PUT32(GPIO_OUTPUT_EN,(1<<19)|(1<<21)|(1<<22));
```

```
    PUT32(GPIO_PORT,(1<<19)|(1<<21)|(1<<22));
    PUT32(GPIO_OUT_XOR,0);
    while(1)
    {
        PUT32(GPIO_PORT,(1<<19)|(1<<21)|(1<<22));
        for(rx=0;rx<2000000;rx++) dummy(rx);
        PUT32(GPIO_PORT,0);
        for(rx=0;rx<2000000;rx++) dummy(rx);
    }

    return(0);
}
```

memmap

```
MEMORY
{
    rom : ORIGIN = 0x20010000, LENGTH = 0x1000
    ram : ORIGIN = 0x80000000, LENGTH = 0x4000
}
SECTIONS
{
    .text : { *(.text*) } > rom
    .rodata : { *(.rodata*) } > rom
    .bss : { *(.bss*) } > ram
}
```

build

```
riscv32-none-elf-as -march=rv32imac -mabi=ilp32 novectors.s -o novectors.o
riscv32-none-elf-gcc -march=rv32imac -mabi=ilp32 -Wall -O2 -nostdlib -nostartfile
riscv32-none-elf-ld novectors.o notmain.o -T memmap -o notmain.elf
riscv32-none-elf-objdump -D notmain.elf > notmain.list
riscv32-none-elf-objcopy notmain.elf -O ihex notmain.hex
riscv32-none-elf-objcopy notmain.elf -O binary notmain.bin
```

Now in theory you can use the riscv64-unknown-elf they talk about even though they want to build for rv32 not rv64. I can try that too.

notmain.list

```
Disassembly of section .text:

20010000 <_start>:
20010000:   80004137            lui x2,0x80004
20010004:   010000ef            jal x1,20010014 <notmain>
20010008:   a001                    j   20010008 <_start+0x8>

2001000a <dummy>:
2001000a:   8082                    ret

2001000c <PUT32>:
```

```
2001000c:   c10c                    sw  x11,0(x10)
2001000e:   8082                    ret

20010010 <GET32>:
20010010:   4108                    lw  x10,0(x10)
20010012:   8082                    ret

20010014 <notmain>:
20010014:   1141                    addi    x2,x2,-16
20010016:   c04a                    sw  x18,0(x2)
20010018:   10012937            lui x18,0x10012
2001001c:   00890513            addi    x10,x18,8 # 10012008 <_start-0xfffdff8>
20010020:   006805b7            lui x11,0x680
20010024:   c606                    sw  x1,12(x2)
20010026:   c226                    sw  x9,4(x2)
20010028:   c422                    sw  x8,8(x2)
2001002a:   37cd                    jal 2001000c <PUT32>
2001002c:   00c90513            addi    x10,x18,12
20010030:   006805b7            lui x11,0x680
20010034:   3fe1                    jal 2001000c <PUT32>
20010036:   04090513            addi    x10,x18,64
2001003a:   4581                    li  x11,0
```

Important to check before you try to load the program onto the device, our desired entry code, first instructions of novectors.s need to be at 0x20010000 for this board/chip as shipped (factory bootloader). And it is.

notmain.hex

```
:020000042001D9
:10000000374100800EF00000101A082800CC1828096
:1000100008418280411114AC0372901101305890027
:10002000B705680006C626C222C4CD371305C9002D
:10003000B7056800E13F1305090481458B7841E0038
:10004000F13731099384044 8B70568004A857D3F3C
:1000500001442285050455 3FE31D94FE81454A85F0
:10006000753701442285050 44D37E31D94FEE9BF31
:04000005200100 00D6
:00000001FF
```

Copy notmain.hex to the mounted HiFive media. Now this cost me an hour or two trying to figure out the hex file as I started, here, it didn't work. Downloaded their sdk dug through that found an elf2hex but that was a bad tangent that was for fpga work it appears. Figured it out and all they are doing is riscv...objcopy -O ihex just like I have, tried it one more time. And now it works. I was getting a fail.txt saying it couldn't connect to the cpu before. Don't know what I did or didn't do to make this work.

In theory you can cut and paste the hex file above and save it and copy it. Why does nobody have an example hex file, you gotta have the 75 special things installed right and run a build rather than also provide here is a complete example with intermediate files. I certainly will do this in my examples for this platform. Or at least the one above.

Instead of their rainbow led blinking pattern, the above will make it blink "white" on and off at a regular rate.

Note LEDs are on the same GPIO lines on the rev a board, the bootloader lands on a different address 0x20400000 than the rev b 0x20010000. So the same can be built for the rev a board with that one memmap change.

If you or the reader wants to go back to a rev a if they have one it is a modified openocd, which at the time of this writing is at github riscv user riscv-openocd project. the normal ./bootstrap, ./configure, make to get the tools and in the tcl dir there is the riscv openocd config file shown above

```
interface ftdi
ftdi_device_desc "Dual RS232-HS"
ftdi_vid_pid 0x0403 0x6010
```

was the key, the rev2 board lsusb:

```
Bus 001 Device 018: ID 1366:1051 SEGGER
```

and no hits on those pid/vid values in the openocd config files. Leading to reading more of the getting started manual.

Share Follow

edited Sep 14, 2020 at 23:54

halfer

20.3k ● 19 ● 105 ● 197

answered Oct 28, 2019 at 10:03

old_timer

70.6k ● 8 ● 96 ● 171

Wow. Thank you for the thorough answer, There is a lot to digest here, and I'm sure I'll have some follow up questions. But the HiFive1 Rev b is not badly priced and you can order it here. crowdsupply.com/sifive/hifive1-rev-b – christopher clark Oct 28, 2019 at 17:44

1 Okay they need to change their website from pre-order to order. Forgot they are going through crowdsupply. A board like that should be more like $10 - $20. I went ahead and ordered one... – old_timer Oct 28, 2019 at 18:41

There are some other boards with these parts that are in the right price range around $15 and now that I think about it I have one or two of those laying about somewhere. Should have the 002 part but not necessarily sifive firmware. – old_timer Oct 28, 2019 at 18:46

I can't quite figure out how to get openocd working on windows. and wsl with your config can't find the device. I'll have to come up with something – christopher clark Nov 1, 2019 at 3:03

get a live ubuntu cd/dvd/usb, apt-get install openocd...just a thought...there should be openocd ports to windows though – old_timer Nov 1, 2019 at 8:37

Show **9** more comments

`times` is not an instruction. It is an assembler directive. $ returns your current address and $$ denotes the beginning of your current sector. So, you're filling the remaining portions of your

2

512 byte sector with 0 (with 2 bytes being magic number). Yeah 64ish lines of code definitely won't work. Support for times command will depend on your assembler. So, if there is an assembler which support TIMES and RISC-V out there, you should be able to use times. NASM does support the times directive and there is a version of NASM for RISC-V https://github.com/riscv/riscv-nasm. So, you should check this out.

Share Follow

answered Oct 26, 2019 at 21:55

Moosa Mahsoom

34 • 4

Moosa, it seems that that repository is not being maintained. There is a gcc toolchain, I will try that. – christopher clark Oct 27, 2019 at 2:24

you can see my edit. the gnu toolchain seems to be the supported way. – christopher clark Oct 27, 2019 at 17:58

Add a comment

Not the answer you're looking for? Browse other questions tagged assembly riscv or ask your own question.