

Learning RISC-V with embedded targets and QEMU

Loh Siu Yin
Beyond Broadcast LLP
siuyin@beyondbroadcast.com

1. Why this article?

The chips I use are ARM Cortex M0/M0+, MSP430G2, PIC16 and STM8.

I then came across RISC-V RV32EC in the form of the CH32V003. The 32 means 32-bit E for the embedded capability and C for compressed instructions to help reduce code size.

Each chip had a steep learning curve and RISC-V was similar. The amount of resources was overwhelming. Where do I start? How do I create bare-metal code to blink an LED?

My starting point was: <https://riscv.org/>.

2. Running Linux on QEMU

<https://risc-v-getting-started-guide.readthedocs.io/en/latest/linux-qemu.html> shows you how to *almost* get a QEMU environment up the hard way. You compile QEMU, Linux and BusyBox for RISC-V. I later found I could also use the QEMU packaged the linux distribution I used. For Ubuntu 22.04 it was the `qemu-system-misc` package.

When I got to the running stage at <https://risc-v-getting-started-guide.readthedocs.io/en/latest/linux-qemu.html#running>, it crashed with:

```
[ 1.200880] [<ffffffff80825ac4>] kernel_init+0x1e/0x10a
[ 1.201294] [<ffffffff80003762>] ret_from_fork+0xa/0x1c
[ 1.202683] ---[ end Kernel panic - not syncing: VFS: Unable to mount root fs on unknown
QEMU: Terminated
```

After searching the internet, I learned I needed to provide an initial ram filesystem. The invocation that finally worked for me was:

`runqemu.sh:`

```
qemu-system-riscv64 -nographic -machine virt \
-kernel linux/arch/riscv/boot/Image -append "root=/dev/vda ro console=ttyS0" \
-drive file=busybox/busybox,format=raw,id=hd0 \
-device virtio-blk-device,drive=hd0 \
-initrd initramfs.cpio.gz
```

And the output was a root linux prompt in QEMU:

```
[ 1.368036] clk: Disabling unused clocks
[ 1.448878] Freeing unused kernel image (initmem) memory: 2184K
[ 1.450971] Run /init as init process
```

Boot took 1.71 seconds

Ctrl a x to exit QEMU
~ #

But how do you get that `initramfs.cpio.gz` file?

3. Using Busybox as the root filesystem in RISC-V QEMU

After more searching on the internet, I found:

<https://gist.github.com/chrisdone/02e165a0004be33734ac2334f215380e>

That lead me to write:

mkinitramfs.sh:

```
rm -rf initramfs
mkdir -p initramfs
cp init initramfs
chmod +x init

cd initramfs
mkdir -p bin sbin etc proc sys usr/bin usr/sbin
cp -a ../busybox/_install/* .

cp ../hello .

find . -print0 | cpio --null -ov --format=newc \
    | gzip -9 > ../initramfs.cpio.gz
```

And my init:

```
#!/bin/sh

mount -t proc none /proc
mount -t sysfs none /sys
mknod -m 666 /dev/ttyS0 c 4 64

echo -e "\nBoot took $(cut -d' ' -f1 /proc/uptime) seconds\n"
echo "Ctrl a x to exit QEMU"

setsid cttyhack sh
exec /bin/sh
```

For reference here is a snapshot of my project root layout:

```
siuyin@ln03:~/riscv64-linux$ ls -F
about.article  busybox/  go.sum  hello.o  initramfs/  linux/  README.md
build.sh*      go.mod    hello*  init*    initramfs.cpio.gz  mkinitramfs.sh*  runqemu.sh*
```

Ignore the go.sum and go.mod files as there are there only to support this about.article I am writing.

4. Running a RISC-V binary in QEMU runing RISC-V Linux

I chose to learn RISC-V assembly first as that was closest to the bare metal. Below is my hello world in assembly.

You may find this quick reference card handy:

<https://www.cl.cam.ac.uk/teaching/1617/ECAD+Arch/files/docs/RISCVGreenCardv8-20151013.pdf>

src/linux/hello.s:

```
# Simple RISC-V hello world.

.global _start

.text
_start:
    addi    a0, x0, 1          # addi is add immediate. x0 is the zero-value register, 1+0 ->
    la      a1, helloworld     # la is load address
    addi    a2, x0, 13

    addi    a7, x0, 64          # 64 is the syscall to write
    ecall

_end:
    addi    a0, x0, 0          # return code is 0
```

```

    addi    a7, x0, 93      # 93 is the syscall to terminate
    ecall

```

```

.data
helloworld:
    .ascii  "Hello World.\n"

```

If you looked closely at my `mkinitramfs.sh` earlier, you will see that I copied the hello binary to the `initramfs.cpio.gz`.

This is how I built the hello binary:

`build.sh`:

```

#!/bin/bash

#riscv64-linux-gnu-gcc -o hello hello.s -nostdlib -static

riscv64-linux-gnu-as -march=rv64imac -o hello.o src/linux/hello.s
riscv64-linux-gnu-ld -o hello hello.o

```

The `-march=rv64imac` reads: assemble for RISC-V 64bit with integer, multiply (and divide), atomic instructions and compressed instructions.

Below is the output of a run in QEMU:

```

[ 1.418066] Freeing unused kernel image (initmem) memory: 2184K
[ 1.420248] Run /init as init process

Boot took 1.67 seconds

Ctrl a x to exit QEMU
~ # ls -F
bin/      etc/      init*     proc/     sbin/     usr/
dev/      hello*    linuxrc@ root/     sys/
~ # ./hello
Hello World.
~ #

```

5. Running RISC-V on QEMU 'bare-metal'

In the previous example I syscall'ed into Linux to print a string and to exit. Microcontrollers can be very small and will usually not run Linux. They may not run any operating system at all.

Thus my next RISC-V assembly program targets a `sifive_e` board that is emulated by QEMU.

```

siuyin@ln03:~/riscv64-linux$ qemu-system-riscv32 -machine help
Supported machines are:
none                empty machine
opentitan           RISC-V Board compatible with OpenTitan
sifive_e            RISC-V Board compatible with SiFive E SDK
sifive_u            RISC-V Board compatible with SiFive U SDK
spike               RISC-V Spike board (default)
virt                RISC-V VirtIO board

```

https://github.com/qemu/qemu/blob/792f77f376adef944f9a03e601f6ad90c2f891b2/hw/riscv/sifive_e.c#L15 gives more details about the emulated board.

5.1. Assembly code for the emulated red-v board

The `sifive_e` board emulates a serial port (UART) that allows strings to be printed onto the QEMU console.

`puts` in the assembly code below writes to the serial port.

src/embed/embedhello.s:

```
.align 2
.equ UART_REG_TXFIFO, 0
.equ UART_BASE, 0x10013000

.section .text
.globl _start

_start:
    # load the mhartid (machine hardware thread ID) control and status register into t0
    csrr    t0, mhartid
    # branch to halt if the hartid is not zero -- all other threads are halted
    bnez    t0, halt

    # load address stack_top (from linker script) into stack pointer
    la      sp, stack_top
    # load address of msg into a0 (argument 0) register, t0 above is temporary 0
    la      a0, msg
    jal     puts          # jump and link: call puts

    la      a0, msg2      # call puts again but now with msg2
    jal     puts

    j       halt          # end the program

puts:
    li      t0, UART_BASE

.puts_loop: lbu      t1, (a0)
    beqz    t1, .puts_leave

.puts_wait: lw      t2, UART_REG_TXFIFO(t0)
    bltz    t2, .puts_wait

    sw      t1, UART_REG_TXFIFO(t0)
    add     a0, a0, 1

    j       .puts_loop

.puts_leave:
    ret

halt:      j       halt

.section .rodata
msg:
    .string "Hello risc-v!\n"

msg2:
    .string "This is my second string.\n"
```

5.2. RISC-V privileged mode and multiple hardware threads

The `csrr` instruction surprised me. Normally embedded processes are single core with a single thread of execution. With some implementations, you may have multiple cores. This code targets only the core with thread ID of zero.

```
# load the mhartid (machine hardware thread ID) control and status register into t0
csrr    t0, mhartid
# branch to halt if the hartid is not zero -- all other threads are halted
bnez    t0, halt
```

See the extract from the RISC-V manual volume 2 below:

0x1003_6000	0x1FFF_FFFF		Reserved	
0x2000_0000	0x3FFF_FFFF	R XC	QSPI 0 Flash (512 MiB)	Off-Chip Non-Volatile Memory
0x4000_0000	0x7FFF_FFFF		Reserved	
0x8000_0000	0x8000_3FFF	RWX A	E31 DTIM (16 KiB)	On-Chip Volatile Memory
0x8000_4000	0xFFFF_FFFF		Reserved	

Table 4: FE310-G002 Memory Map. Memory Attributes: **R** - Read, **W** - Write, **X** - Execute, **C** - Cacheable, **A** - Atomics

<https://sourceware.org/binutils/docs/ld/Scripts.html> is the GNU Binutils LD linker script reference.

5.4. C code for the emulated red-v board

Below is the startup assembly code to jump to the main C code.

src/c_embed/startup.s:

```
.align 2

.section .text
.globl _start

_start:
    csrr    t0, mhartid
    bnez    t0, halt

    la      sp, stack_top    # initialize stack

    j       main             # jump to main

halt:     j       halt
```

And below is the main C code.

src/c_embed/main.c:

```
typedef unsigned int size_t;
typedef unsigned char uint8_t;

#define UART_BASE ((size_t *)0x10013000)
#define TXDATA (UART_BASE + 0)

// Making putchar static optimizes space as it will not be 'exported'
// to be visible in other source files.
static void putchar(uint8_t ch) {
    while ((volatile int)*TXDATA < 0);
    *TXDATA = ch;
}

void puts(char *s) {
    // while char pointed to by s is non-zero, putchar and increment the pointer
    while (*s) putchar(*s++);
    putchar('\n');
}

void main() {
    puts("Hello RISC-V from C!");
    puts("bye");
}
```

Let's take a closer look at:

```
while ((volatile int)*TXDATA < 0);
```

We cast the `size_t` pointed to by `UART_BASE` to a `volatile int`.

`volatile` tells the compiler the value may be changed by the hardware and thus to not eliminate it when optimizing the code.

`int` makes it a signed integer. As the red-v manual extract below states, the most significant bit (MSB) will be set if the FIFO is full -- and thus not ready to take another byte. if the MSB is set, it is a negative number.

Thus the code reads:

wait for the TX FIFO to be not full then write `ch` to the memory pointed to by `TXDATA`.

```
while ((volatile int)*TXDATA < 0);
*TXDATA = ch;
```

18.4 Transmit Data Register (`txdata`)

Writing to the `txdata` register enqueues the character contained in the data field to the transmit FIFO if the FIFO is able to accept new entries. Reading from `txdata` returns the current value of the `full` flag and zero in the data field. The `full` flag indicates whether the transmit FIFO is able to accept new entries; when set, writes to data are ignored. A RISC-V `amoor.w` instruction can be used to both read the `full` status and attempt to enqueue data, with a non-zero return value indicating the character was not accepted.

Transmit Data Register (<code>txdata</code>)				
Register Offset		0x0		
Bits	Field Name	Attr.	Rst.	Description
[7:0]	data	RW	X	Transmit data
[30:8]	Reserved			
31	full	RO	X	Transmit FIFO full

Table 56: Transmit Data Register

5.5. Makefile for embedded C

`src/c_embed/Makefile:`

```
CC=riscv64-unknown-elf-gcc -O2 -g -nostdlib -Wno-builtin-declaration-mismatch -march=rv32ec
AS=riscv64-unknown-elf-as -march=rv32ec
LD=riscv64-unknown-elf-ld -nostdlib -melf32lriscv -Tred-qemu.ld
SIZE=riscv64-unknown-elf-size
OBJDUMP=riscv64-unknown-elf-objdump

OBS=c_embed startup.o main.o main.s

all: c_embed

c_embed: startup.o main.o
    $(LD) -o c_embed startup.o main.o
    $(SIZE) c_embed

%.o: %.s
    $(AS) -o $@ $<

main.s: main.c
    $(CC) -o main.s -S main.c

.PHONY: clean
clean:
    rm -f $(OBS)
```

```
.PHONY: dump
dump: c_embed
    $(OBJDUMP) -S --source-comment='#' -d c_embed
```

The line:

```
CC=riscv64-unknown-elf-gcc -O2 -g -nostdlib -Wno-builtin-declaration-mismatch -march=rv32ec
```

is needed because gcc already knows of a putchar function in libc and it is not a void function.