The Not So Surprising Similarity Between MIPS and RISC-V

So it is the holidays and what does every hacker do at that time of the year? Hang around and write assembly code and poke at instruction sets right…of course.

Over the years I have poked at many instruction sets both professionally and as a free hacker: x86, PPC, Arc, OpenRisc, MIPs, Arm, EZChip…. Blah blah blah. A veritable list of joy. My current interests both professionally and hackerly have led me to poke at the latest craze in instruction sets RISC-V.

RISC-V has recently gained some traction and notoriety due to the fact that 1) it has some fairly strong supporters in the market place, 2) like OpenRisc before it and similar to Linux in the software world, it is open source, 3) as we shall see, some of its core developers are industry vets and it shares some commonality with other famous instruction sets, and 4) it has some extensible features.

As has been well documented elsewhere the "roll your own" technology path comes with both pluses and minuses and you as a programmer, designer, or system architect have to make the choice(s) that best suit your needs. This paper is a small peek at the programming environment and not a commentary on larger design choices.

Being in the industry and having been a vet of OpenRisc https://opencores.org/ and seeing my old friends in the industry like Embecosm https://www.embecosm.com/ delving into RISC-V gave me inspiration to return to my old addiction at least in some spare time during the holidays in a pandemic.

All different?

Typically instruction sets which for most software developers are represented as assembly language (or even at a higher level of abstraction by compilers) are typically quite different between core architectures. And you can't, for example, use an out-of-the-box x86 compiler to compile code for an Arm target (you must use a cross compiler to target a different architecture and the output is the instruction set of the target not the host so it won't run on the host). The instruction sets represent the core of the architecture and as such they form part of the Application Binary Interface or ABI.

And, of course, for anyone who wants to code at the machine/compiler level the 'must have' at a minimum is a good understanding of the instructions and registers that make up the programming interface.

Here are a few snips of what different assembly languages look like performing a similar operation (print a string with write system call):

x86 example

```
movl    $len,%edx       # third argument: message length
movl    $msg,%ecx       # second argument: pointer to message to write
movl    $1,%ebx         # first argument: file handle (stdout)
movl    $4,%eax         # system call number (sys_write)
int      $0x80          # call kernel
```

arm example

```
mov  r0, #1             @;place filehandle 1 in r0
ldr  r1, = msg          @;place address of myStr in r1
ldr  r2, =len           @;place  len in r2
mov  r7, #4             @;put write syscall 4 write in r7 per EABI
swi  #0                 @;call kernel
```

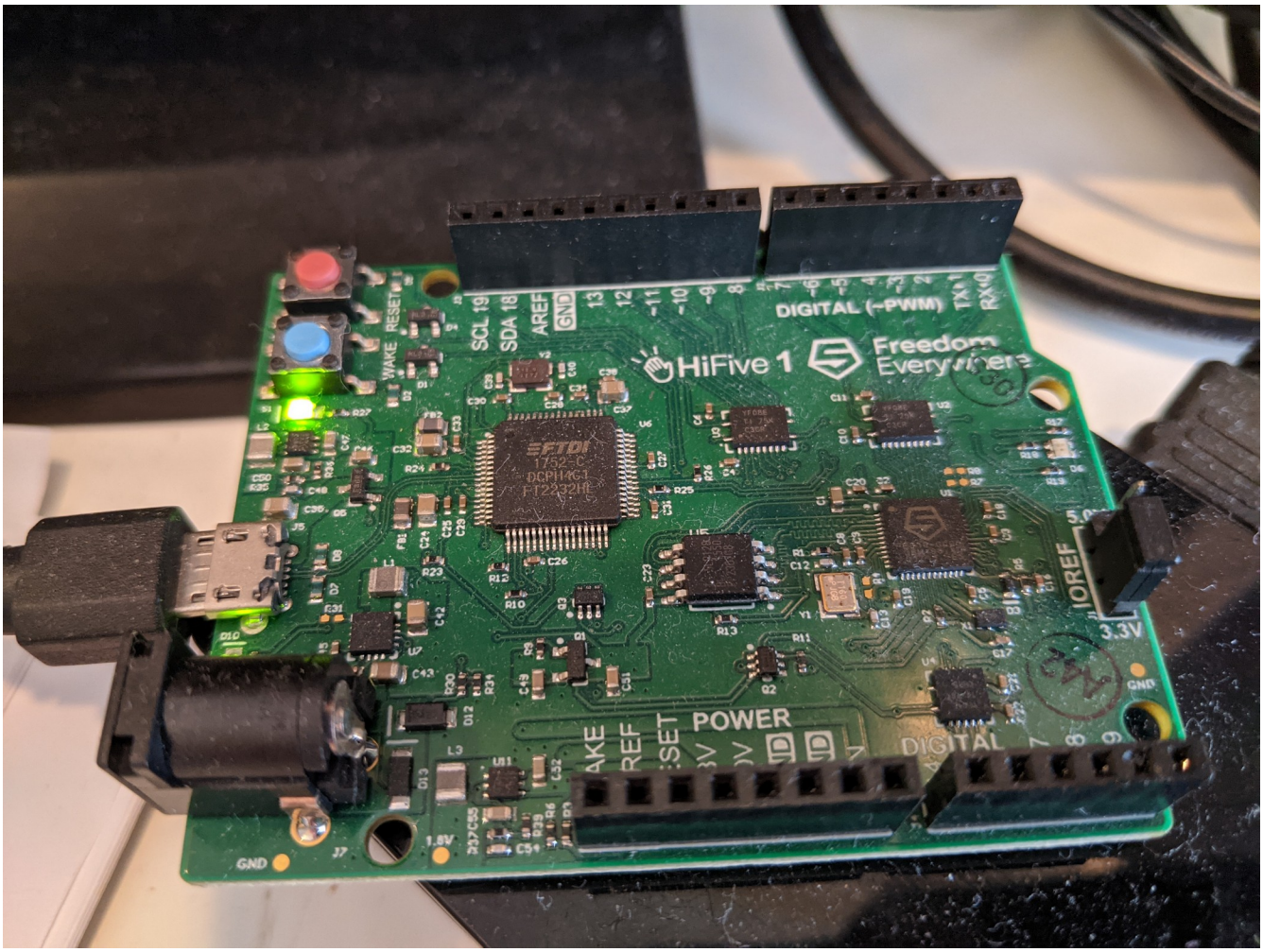
openrisc example

```
 l.addi      r3,r0,0x1       #filehandle 1 into r3 arg1
l.movhi      r4, hi(msg)     #move upper addr into r4 upper
l.ori        r4,r4,lo(msg)   #move lower adrr into r4
l.addi       r5,r0,len       #len of the string
l.addi       r11,r0,0x4      #system call 4 (write)
l.sys        0x1             #call kernel
l.nop        0x0             #fill delay slot
```

All three of these pieces of code do essentially the same thing, load some key registers and call the write system call and are usually used in printing the first helloworld string --the first exercise to tackle on a new architecture.

As can be seen there are different registers, naming conventions, calling conventions, and ways of moving and storing data (There are other major differences as well that I am skipping here).

So back to the point, recently I have been writing some code for RISC-V, initially I started out just using a simulator (Spike) and then eventually picked up a HiFive 1 physical board to mess around with.

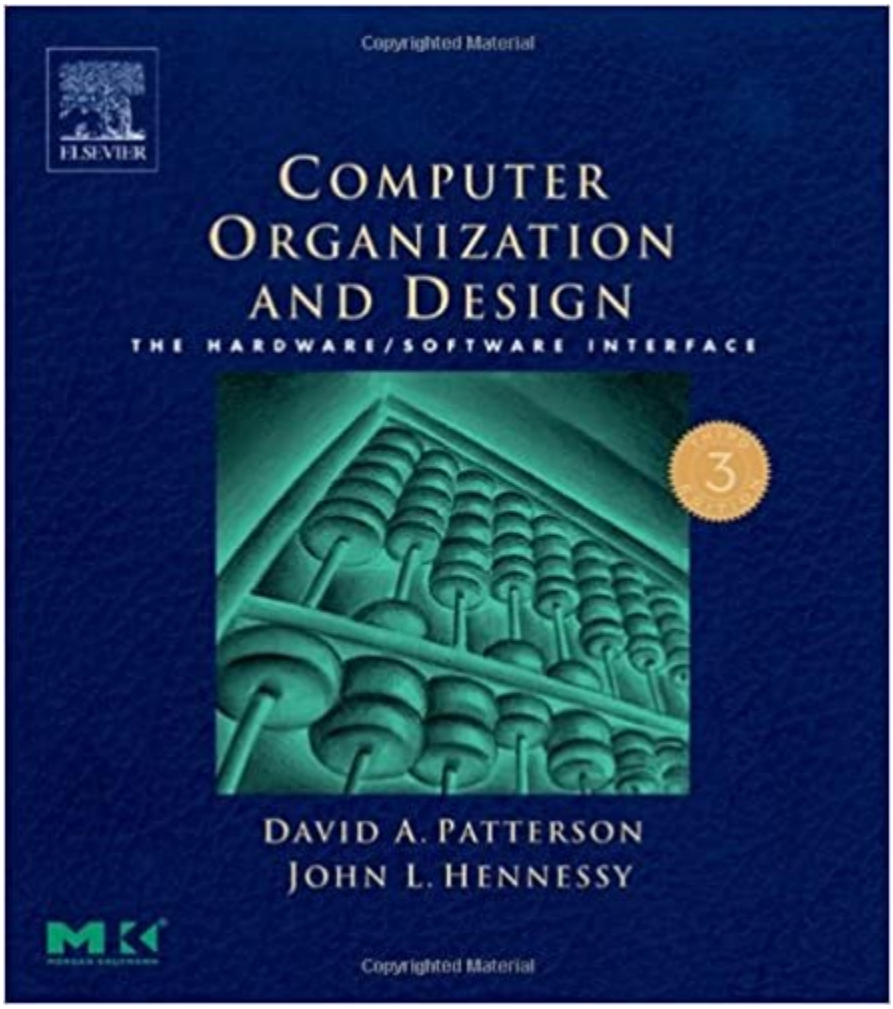


As usual the first few programs were bumbling around trying to get the tools and the environment setup and do a basic helloworld. I try to keep my setup as simple as possible avoiding IDKs and special tools which can tend to hide the nuances of an architecture.

Yes tools are great for production and workplaces but to really obtain an understanding of how the machine works it is sometimes best to deal with the nuances head on. The proper toolchain, fav basic editor, and Make and you should be good to go. Right. Anyhow you can go to my github page if you want to see those meanderings https://github.com/rgdaddio/riscv_stuff.

But once I got past the initial hurdles and starting writing more code there was something oddly familiar to me about the RISC-V registers and the instructions. At first I attributed it to age and experience or possibly senility. Yet having done this before is somewhat different than coding deja vu…

As I continued to code the magic started to wear off and I knew I had seen similar constructs before and it dawned on me that one of the key architects of RISC-V and the author of my absolute favorite CS book was David Patterson. Hmmm. And I have relied on the same old edition for a long time:



“Computer Organization And Design: The Hardware/Software Interface” AKA “The Blue Book” or AKA CO&D. Apparently more recently the work has been updated to be architecture specific but no doubt contains the same awesome material. Pretty much everything the aspiring software developer who works at the HW/SW interface needs to know. It is like the bible for embedded folks.

Of course, my trusty copy was sitting nearby faithfully waiting to rescue me for some stupid idea or another and as I began to peruse it found out where my deja vu was coming from. So I went and took a look at some of my old favorites ones I really like are the swap and sort routines in chapter 2. They are simple yet revealing.

Really?

Here is the MIPS swap routine copied right out of the CO&D:

```
swap:
        sll     $t1, $a1, 2
        add     $t1, $a0, $t1

        lw      $t0, 0($t1)
        lw      $t2, 4($t1)

        sw      $t2, 0($t1)
        sw      $t0, 4($t1)

        jr      $ra
```

If you go to your text editor and simply remove the ‘$’ from the MIPS swap code and create a small program like this:

```
.global _start
.type _start,@function
_start:
        jal swap



.global swap
.type swap,@function
swap:                ##MIPS code with just the ‘$’ removed from reg names
        sll t1, a1, 2
        add t1, a0, t1

        lw  t0, 0(t1)
        lw  t2, 4(t1)

        sw  t2, 0(t1)
        sw  t0, 4(t1)

        jr  ra
```

And then assemble and link this same exact swap code with the ‘$’ removed using a RISC-V tool chain with something like:

```
riscv64-unknown-elf-as testswap.s -o testswap
riscv64-unknown-elf-ld -o testswap testswap.o
```

You will find this assembles and links without errors, which though only 7 lines of code is somewhat remarkable in and of itself. Much of the register naming, instruction naming, conventions, and even mnemonics seem to be somewhat similar.

Now you might think it is a small example but doing something as simple as this would be impossible in most other cases even with architectures like OpenRisc which shares some MIPS genetics.

And even larger examples such as sort from CO&D compile with the simple '$' removal and one minor mnemonic change 'move' to 'mv'.

This is the MIPS version of sort again copied directly from the book:

```
sort:
        addi $sp, $sp, -20
        sw   $ra, 16($sp)
        sw   $s3, 12($sp)
        sw   $s2, 8($sp)
        sw   $s1, 4($sp)
        sw   $s0, 0($sp)

        move $s2, $a0
        move  $s3, $a1

        move  $s0, $zero
for1tst:
        slt  $t0, s0, $s3
        beq  $t0, $zero, exit1
        addi $s1, s0, -1
for2tst:
        slti $t0, $s1, 0
        bne  $t0, $zero, exit2
        sll  $t1, $s1, 2
        add  $t2, $s2, $t1
        lw   $t3, 0($t2)
```

```
            lw   $t4, 4($t2)
            slt  $t0, $t4, $t3
            beq  $t0, $zero, exit2

            move  $a0, $s2
            move  $a1, $s1
            jal  swap

            addi $s1, $s1, -1
            j for2tst

exit2:
            addi $s0, $s0, 1
            j for1tst

exit1:

            lw   $s0, 0($sp)
            lw   $s1, 4($sp)
            lw   $s2, 8($sp)
            lw   $s3, 12($sp)
            lw   $ra, 16($sp)
            addi $sp, $sp, 0x20

            jr  $ra
```

And next here is the entire program for RISC-V with only the '$' removed and 'move' changed to 'mv' from the MIPS program then assembled and linked like before with the RISC-V GNU assembler and linker (this includes the earlier swap and is equivalent to the complete program listed in chapter 2 of CO&D). This is the RISC-V version with the miniscule modifications:

```
.global _start
.type _start,@function
_start:

        jal sort


.global swap
.type swap,@function
swap:

        sll t1, a1, 2
        add t1, a0, t1

        lw  t0, 0(t1)
        lw  t2, 4(t1)

        sw  t2, 0(t1)
        sw  t0, 4(t1)

        jr  ra


sort:
        addi sp, sp, -20
        sw   ra, 16(sp)
        sw   s3, 12(sp)
```

```
        sw   s2, 8(sp)
        sw   s1, 4(sp)
        sw   s0, 0(sp)

        mv s2, a0
        mv  s3, a1

        mv  s0, zero
for1tst:
        slt  t0, s0, s3
        beq  t0, zero, exit1
        addi s1, s0, -1
for2tst:
        slti t0, s1, 0
        bne  t0, zero, exit2
        sll  t1, s1, 2
        add  t2, s2, t1
        lw   t3, 0(t2)
        lw   t4, 4(t2)
        slt  t0, t4, t3
        beq  t0, zero, exit2

        mv  a0, s2
        mv  a1, s1
        jal  swap

        addi s1, s1, -1
        j for2tst
```

```
exit2:
        addi s0, s0, 1
j for1tst

exit1:
        lw   s0, 0(sp)
        lw   s1, 4(sp)
        lw   s2, 8(sp)
        lw   s3, 12(sp)
        lw   ra, 16(sp)
        addi sp, sp, 0x20

        jr  ra
```

```
riscv64-unknown-elf-as testswapandsort.s -o testswapandsort.o
riscv64-unknown-elf-ld -o testswapsort testswapandsort.o
```

Conclusion

It is remarkable that so little needs to be modified or changed to build this MIPS copy for RISC-V. Even mnemonics like ‘zero’ work.

I didn’t cover much of the code details in these examples, you can either pick up a copy of CO&D where the authors cover the code in great detail or you can visit my github page: https://github.com/rgdaddio/riscv_stuff where the focus is on RISC-V and there are working versions of swap and sort fully tested on the RISC-V HiFive 1 board.

Of course, just because the instruction sets are similar does not mean you can automatically assume the behavior is identical or that porting is just a matter of a few quick minor tweaks and done. The behavior is still the in the domain of the HW architects and system designers. And for that detail you must refer to the instruction set and other documentation. However having some background in or using some existing MIPS code for examples may be helpful in working with and developing low-level RISC-V code for embedded and bare metal applications.

RGD

12/2020