Sergio Garcia Tapia
Digital Design and Computer Architecture: RISC-V, by Sarah and David Harris
Chapter 6: Architecture
April 3, 2024

Exercise 6.1. Give three examples from the RISC-V architecture of each of the following design principles: (1) regularity supports simplicity; (2) make the common case fast; (3) smaller is faster; and (4) good design demands good compromises. Explain how each of your examples exhibits the design principle.

Solution:

(1) In Section 6.2.1, the `add` and `sub` instructions were introduced, and they both have the same format: `mnemonic source1 source1 destination`. The format is consistent with nearly all RISC-V binary operations, making them predictable and their binary encoding consistent and simple. For example, the **func7** bits are all 0 for all binary operations with an `op` code of 51; they differ only in their `funct3` control bits. Another example is that operands all 32 bits in RISC-V 32, in contrast with x86-32 where operands can be 8-bit, 16-bit, or 32-bit. Also instructions in x86-64 vary in size, whereas in RISC-V they are uniformly 16 bits, as mentioned in Section 6.8.5.

(2) As discussed in 6.2.1, RISC-V has a relatively small set of simple instructions that are commonly used, in contrast with CISC architectures that provide complex operations and therefore add overhead to simple instructions. In Section 6.8.6, the author mentions that x86 contains string operations that are usually slower than performing the equivalent operation with a series of simple instructions. By supporting simple byte operations that are fast, RISC-V can reap these benefits.

(3) RISC-V32 has 32 registers, which are sufficient to perform several operations without having to add local variables to the stack in the case of small functions. Since the stack relies on larger memory, accessing is slower, so avoiding this is best.

(4) In Section 6.4, the author mentions compromise regarding the length of the instruction encoding; some instructions do not require 32 bits, but they are encoded as such anyway to support simplicity. Moreover, instead of a single instruction format, there are four to provide enough flexibility. Another example is the bit-swizzling of the immediate encodings discussed in 6.4.5. Though complicated, it ensures that other instruction fields are consistent and simple.

Exercise 6.2. The RISC-V architecture has a register set that consists of 32 32-bit registers. Is it possible to design a computer architecture without a register set? If so, briefly describe the architecture, including the instruction set. What are the advantages and disadvantages of this architecture over the RISC-V architecture?

Solution: According to Section 6.2.2, registers exist to support access to operands quickly so that they can run fast; they do this by providing a small address space. Therefore, it is possible to not use a register set and use memory. One way would be to reserve the first 32

address locations of memory for the same functions as the corresponding registers. According to Section 5.5.4, memory is commonly implemented with DRAM, whereas a register set is implemented with SRAM. DRAM latency is longer than SRAM because its bitline is not actively driven by a transistor; it fundamentally has lower throughput because data must be refreshed periodically and after a read. Moreover, latency and throughput depend on memory size, an therefore since DRAM is typically very large in comparison to a small register set, this means relying on just memory would be too slow. One advantage is that it generally takes less transistors to build DRAM in comparison to SRAM, making them cheaper and less power hungry.

Exercise 6.3. Write the following strings using ASCII encoding. Write your final answers in hexadecimal.

(a) hello there

(b) bag o'chips

(c) To the rescue!

Solution:

(a) Referring to an ASCII take encoding, we see that the encoding is:

HEX	68	65	6c	6c	6f	20	74	68	65	72	65
String	h	e	l	l	o		t	h	e	r	e

(b)

HEX	62	61	67	20	27	6f	63	68	69	70	73
String	b	a	g		o	'	c	h	i	p	s

(c)

HEX	54	6f	20	74	68	65	20	72	65	73	63	75	65	21
String	T	o		t	h	e		r	e	s	c	u	e	!

Exercise 6.4. Repeat Exercise 6.3 for the following strings.

1. Cool
2. RISC-V
3. boo!

Solution:

(a)

HEX	43	6f	6f	6c
String	C	o	o	l

(b)

HEX	52	49	53	43	2d	56
String	R	I	S	C	-	V

(c)

HEX	62	6f	6f	21
String	b	o	o	!

Exercise 6.5. Show how the strings in Exercise 6.3 are stored in byte-addressable memory starting at memory address `0x004F05BC`. The first character of the string is stored at the lowest byte address (in this case, `0x004F05BC`). Clearly indicate the memory address of each byte.

Solution: I am using word addresses that increase upwards. The last character is therefore at the top.

(a)

hello there	
Byte	Word Address
65	0x004F05C6
72	0x004F05C5
65	0x004F05C4
68	0x004F05C3
74	0x004F05C2
20	0x004F05C1
6f	0x004F05C0
6c	0x004F05BF
6c	0x004F05BE
65	0x004F05BD
68	0x004F05BC

(b)

bag o'chips

Byte	Word Address
73	0x004F05C6
70	0x004F05C5
69	0x004F05C4
68	0x004F05C3
63	0x004F05C2
6f	0x004F05C1
27	0x004F05C0
20	0x004F05BF
67	0x004F05BE
61	0x004F05BD
62	0x004F05BC

(c)

To the rescue!

Byte	Word Address
21	0x004F05C9
65	0x004F05C8
75	0x004F05C7
63	0x004F05CD
73	0x004F05C5
65	0x004F05C4
72	0x004F05C3
20	0x004F05C2
65	0x004F05C1
68	0x004F05C0
74	0x004F05BF
20	0x004F05BE
6f	0x004F05BD
54	0x004F05BC

Exercise 6.5. Repeat Exercise 6.5 for the strings in Exercise 6.4.

Solution: As in Exercise 6.5, I am using word addresses that increase upwards. The last character is therefore at the top.

(a)

Cool

Byte	Word Address
6c	0x004F05BF
6f	0x004F05BE
6f	0x004F05BD
43	0x004F05BC

(b)

RISC-V	
Byte	Word Address
56	0x004F05C1
2d	0x004F05C0
43	0x004F05BF
53	0x004F05BE
49	0x004F05BD
52	0x004F05BC

(c)

boo!	
Byte	Word Address
21	0x004F05C3
6f	0x004F05C2
6f	0x004F05C1
62	0x004F05C0

Exercise 6.7. The `nor` instruction is not a part of the RISC-V instruction set because the same fucntionality can be implemented using existing instructions. Write a short assembly code snippet that has the same functionality: `s3 = s4 NOR s5`. Use as few instructions as possible.

Solution: Recall that NOR is short for NOT OR, so A NOR B means $\overline{A + B}$, where $+$ is the OR operator and the overline is the NOT operator. The truth table is below:

A	B	$\overline{A + B}$
0	0	1
0	1	0
1	0	0
1	1	0

Recalling that NOT can be implemented with `xori` with `-1` as the immediate, do precisely an OR followed by a NOT:

```
or s6, s4, s5
xori s3, s6, -1
```

Exercise 6.8. The `nand` instruction is not a part of the RISC-V instruction set because the same fucntionality can be implemented using existing instructions. Write a short assembly code snippet that has the same functionality: `s3 = s4 NAND s5`. Use as few instructions as possible.

Solution: Recall that NAND is short for NOT AND, so A NAND B means $\overline{A \cdot B}$, where $\cdot$ is the AND operator and the overline is the NOT operator. The truth table is below:

A	B	$\overline{A \cdot B}$
0	0	1
0	1	1
1	0	1
1	1	0

Recalling that NOT can be implemented with `xori` with `-1` as the immediate, do precisely an AND followed by a NOT:

```
and s6, s4, s5
xori s3, s6, -1
```

Exercise 6.9. Convert the following high-level code snippets into RISC-V assembly language. Assume that the (signed) integer variables `g` and `h` are in register `a0` and `a1`, respectively. Clearly comment your code:

(a)

```
if (g > h)
    g = g + 1;
else
    h = h - 1;
```

(b)

```
if (g <= h)
    g = 0;
else
    h = 0;
```

Solution:

(a) We can use a conditional jump, `bge`, to go to the `else` branch when `h >= g`. To ensure we do not execute the `else` branch if we take the `if` branch, we use an unconditional jump with the `j` instruction to the label after the last instruction in the `else` branch. To add the constant, we can use the `addi` command. The assembly follows:

```
    bge     a1, a0      // if h >= g goto else at .L1
    addi    a0, a0, 1   // g = g + 1
    j .L2               // Skip else branch
.L1:
    addi    a1, a1, -1, // h = h - 1
.L2:
```

(b) We can use `blt` to go to the `else` branch if `h > g` (equivalently, `g > h`).

```
    blt     a1, a0     // if h < h goto else at .L1
    addi    a0, zero, 0 // g = 0
    j .L2              // Skip else branch
.L1:
    addi    a1, zero, 0 // h = 0
.L2:
```

Exercise 6.10. Repeat Exercise 6.9 for the following code snippets:

(a)

```
if (g >= h)
    g = g + h;
else
    g = g - h;
```

(b)

```
if (g < h)
    h = h + 1;
else
    h = h * 2;
```

Solution:

(a)

```
        blt     a1, a0     // if h < g goto else at .L1
        add     a0, a0, a1 // g = g + h
        j .L2              // Skip else branch
.L1:
        sub     a0, a0, a1, // g = g - h
.L2:
```

(b) Instead of multiplying by 2 with the `mul` instruction, I used `sll` to shift left by 1 bit, which is equivalent.

```
        bge     a0, a1     // if g >= h goto else at .L1
        addi    a1, a1, 1  // h = h + 1
        j .L2              // Skip else branch
.L1:
        sll     a1, a1, 1  // h = h * 2
.L2:
```