Note 7: RISC-V

Kevin Moy

1 Introduction

Now that we've reached a solid base with understanding how C works, we now delve a level even lower: RISC-V, where *every* instruction is implemented by hardware!

2 Assembly Languages

RISC-V is an **assembly language**, an (even lower level) programming language where instructions are very specific to a computer architecture, or *processor*. It is usually the result of compiling higher-level language code like C. In this process, we break source code down into single assembly instructions that each do one thing for the overall program.

There are many assembly languages, each specific to an architecture. Thus, assembly languages, unlike C, are NOT portable to other architectures. Today, the most common architectures are Intel's x86, ARM, and, of course, RISC-V.

2.1 Assembly Instructions

Basic instructions that will definitely have to be supported are:

- Add/Subtract
- Bit shift
- Reading/Writing from/to memory

What about more complex stuff like conditionals? Simultaneous execution of expressions?

3 A Brief History

Early on, people thought more instructions meant being able to make more complex programs- formally, a *Complex Instruction Set Computing* (CISC). Here's the thing though: nobody wants to learn that bullshit. The only plus that really came from more instructions was that the compiler generally did less work.

So someone came up with *Reduced Instruction Set Computing* (RISC) architecture. This focused on simpler, smaller instruction sets, which made it a lot easier to build fast(er) hardware. So now, people could actually write instructions with minimal expletives! (Actually, that's false, but it was at least less than with CISC). The idea is to let the software do the work with more complicated ops- they *compose* the simpler instructions detailed by RISC. RISC-V has developed since to become an industry standard.

Everything we need to know about RISC-V for 61c is included in the RISC-V Green Card.

4 Registers

Unfortunately, assembly languages do NOT have variables, but instead **registers** to store values. There are 32 of them in total for RISC-V: each register is a 32-bit container (holds a *word*: in fact, a word is actually defined as the size of a CPU's registers, which will be 4 bytes in this class) that is readable and writable. The main benefit of registers is that register access is extremely quick and low-power.

Thus it makes sense to store the most frequently used variables in these registers, and move all the others to memory (a process called *spilling to memory*). Register access is far faster than memory access, on the magnitude of 100. Recall the memory hierarchy:**there is a tradeoff between storage size and access speed**. Registers would be at the very top, right below CPU in speed.

The 32 registers in RISC-V are numbered `x0` to `x31`, each with more special names based on what types of values they ideally should hold:

- `s0` and `s1` refer to `x8` and `x9`
- `s2` to `s11` refer to `x18` and `x27`
- `t0` to `t2` refer to `x5` and `x7`
- `t3` to `t6` refer to `x28` and `x31`

Registers do not have types. The contents of the register are completely static and its use is entirely determined by the operation.

4.1 The Zero Register

Since zero is so common in programming, it got its own register `x0` (name `zero`), which will always contain value 0 and cannot be changed. Basically, if you have an instruction that writes some register but you don't want it to "show", writing to `x0` won't do anything.

4.2 Registers: A Summary

In ISAs (instruction set architectures), we have a fixed number of registers to hold operand values. They are directly built in hardware and extremely fast to read to/write from. However, we are limited to this fixed number of registers, lest our access speed decrease more and more.

Registers form a part of the datapath segment of the processor in a computer. In general, *any* computer can be divided into these five parts:

- Control
- Datapath
- Memory
- Input
- Output

As a side note, **networking buses** interconnect these components, but they won't be important for this class.

5 Assembly Language

Now let's cover the basics of assembly language. Just like in higher-level languages, there's a basic structure for assembly instructions:

```
op dst, src1, src2
```

For any given instruction is always one operator and three operand registers. Sometimes registers are implicit (not included in the instruction) and sometimes they are immediates (numbers) instead, but this basic format is universal in RISC-V.

- `op` = operator
- `dst` = destination register
- `src1` = operand register 1 (”source” register 1)
- `src2` = operand register 2 (”source” register 2)

The vast majority of RISC-V instructions only allow one operator and one instruction per line.

Here's an example of some RISC-V code, that actually prints the Fibonacci sequence.

```
main: add  t0, x0, x0
      addi t1, x0, 1
      la   t3, n
      lw   t3, 0(t3)
fib:  beq  t3, x0, finish
      add  t2, t1, t0
      mv   t0, t1
      mv   t1, t2
      addi t3, t3, -1
      j    fib
finish: addi a0, x0, 1
        addi a1, t0, 0
        ecall # print integer ecall
        addi a0, x0, 10
        ecall # terminate ecall
```

Each line holds a single assembly instruction. Comments use `#` and are **incredibly** necessary here to maintain sanity and understand the instruction(s). Then we have **labels** (`main, fib, finish`) that just mark a section of code, sort of like defining a function but with important differences we shall get into later.

RISC-V has many instructions, and are documented in the green sheet.

6 Basic Arithmetic in RISC-V

Addition and subtraction in RISC-V is pretty straightforward: `add s1 s2 s3` is basically the same as `s1 = s2 + s3`, and `sub s1 s2 s3` is equivalent to `s1 = s2 - s3`. Note: commas in instructions are optional but might help for some.

For more complicated instructions that need intermediate values calculated, use temporary registers to hold those values.

On the green sheet, there are 4 columns for instructions: MNEMONIC, FMT, NAME, and DESCRIPTION. "Mnemonic" just refers to the operation name, and FMT refers to the instruction format- how the instruction will translate into machine code. NAME refers to the human name for the instruction. Finally, the description (in Verilog) gives a sort of (Verilog) code-like equivalent for the instruction. Consider `R` as an array of all our registers, and `M` as the memory (array of memory cells).

7 Immediate Instructions

Remember that constants are called **immediates** in RISC-V. The basic structure for immediate instructions is very similar to basic arithmetic instructions (in fact, most are just normal operations with an `i` appended, like `addi`), but a source register is replaced with an immediate instead:

```
opi dst, src, imm
```

We'd perform the operation specified by `opi` on `src` and `imm` and then set its result to `dst`. `imm` can be up to 12 bits, or $2^{12} - 1 = 4095$. Immediates can be negatives, hence why there is no `subi` instruction.

8 Data Transfer Instructions

In the case where we need (lots of) memory space instead of just registers, data transfer instructions come in handy. This might be for cases like arrays or particularly large structs. However, remember that RISC-V instructions *only* operate on registers!

Data transfer instructions concern the movement of data between registers and memory. **Store instructions** store information from registers TO memory, while **load instructions** load information from memory TO registers.

The basic syntax for data transfer operations is:

```
memop reg, off(bAddr)
```

Here, `reg` would be the register to load into or store from, `bAddr` is a register containing the base address in memory, and `off` is an offset immediate in bytes that indicates the offset from `bAddr` to load from/store into. In short, memory is addressed at `bAddr+off`.

Memory addresses are *byte-addressed.* not word-addressed. Pointer arithmetic must be done manually in assembly, and you must account for this alignment in doing so.

The load word instruction (`lw`) loads a word of memory starting at `bAddr+off` and loads it into `reg`. On the other hand, the store word instruction takes the (word of) data in `reg` and stores it into memory at memory address `bAddr+off`.

Consider arrays: If an int array `arr` is stored at the address held in `s3`, then `arr[i]` can be loaded via `lw t0 4*i(s3)`. The same offset is used for storing into an array in memory as well.

8.1 Assembler Directives- .data and .text

The `.data` directive denotes data storage- in particular, the static memory segment. Following in data storage is the `.word` directive, which stores its subsequent argument(s) in memory.

The `.text` directive denotes that everything below is part of code memory.

8.2 Review: Endianness and Sign Extension

Sometimes we don't want to load/store a word- maybe just a byte.

Recall big-endian computers have, for a multiple-byte value, its MSB at the lowest address (as memory addresses increase, byte significance decreases). Little-endian computers have LSB at lowest address (memory addresses increase, byte significance increase). **All RISC-V computers are little-endian.** These make a difference where `bAddr` starts at: for our purposes, `0(bAddr)` will start at the LSB.

Also recall sign extension involves taking the MS bit and left-extending it all the way. This will preserve a number's value in 2's complement. **All base RISC-V instructions sign-extend (when needed)**.

8.3 Variable Data Transfer

There are times where we don't want to load/store a word- maybe just a byte. `lb` and `sb` do this, **utilizing the least significant byte of the register** (since little endian). `lb` sign-extends the upper 3 bytes. So if `s0` held memory address `0x00000180`, and we executed `lb s2, 0(s0)`, the LSB (`80`) is loaded and sign-extended into `s2` giving `0xFFFFFF80`.

Another set of instructions for data transfer is `lh` and `sh`: these load and store a half-word (2 bytes) respectively. They work the exact same way as above: sign extension when loading and none when storing.

There are also *unsigned* variations of these load instructions `lhu` and `lbu`: this simply means instead of sign-extending zero pad, treating the loaded data as unsigned (instead of 2's complement).

9 Control Flow in RISC-V

Unlike most normal programming languages, assembly doesn't really have code blocks- just **labels**, which refer to the instruction immediately below. We can **jump** to these labels as a sort of "fake" control flow.

Two main types: conditional and unconditional jumps. Conditional jumps, or **branches**, jump only if a condition is fulfilled. Unconditional jumps just jump where specified.

9.1 (Un)Conditional Jumps

`beq reg1 reg2 label` tests conditional `reg1=reg2`: if true, go to `label`, otherwise go to next instruction (immediately below `beq`). `bne` does the same except jumps to `label` iff `reg1 != reg2`. There is also `blt` and `bge`

Unconditonal jumps are just specified by `j label`.

9.2 Implementing Loops

While, do...while, and for loops are all interchangeable. We can simulate a loop via conditional branches in RISC-V.

Before that, though, we must understand the Program Counter (PC). It is a special register that points to the current instruction being executed. Normally, it is NOT accessible as the other standard 32 registers, but certain special instructions can edit it.

These instructions are, of course, jump statements (other statements like `auipc` do as well).

Assembly instructions are all in the code segment of memory and each have an associated memory address. Labels indicate (to the PC) the address of the next instruction to execute.

Whenever we execute a conditional jump, it's PC-relative. Since our instruction addresses are half-aligned (must always be even addresses), the immediate (offset) added to the PC must be an even number- so a 0 bit is prepended.

10 Shifting Instructions

Recall the basics of bit shifting: left shifts by x is equivalent to multiplying by 2^x, while right shifts by y is the same as FLOOR dividing by 2^y.

There are also different kinds of shifts. **Logical shifts** add zeros as you shift. **Arithmetic shifts** sign-extend as we shift- which only really makes sense if we're right-shifting, in order to preserve the sign of the number. A register/immediate will tell us how many bits to shift by.

Interestingly, these shifting operations will allow us to implement `lbu` and `lhu` with `lw` by simply cutting off the extra bits loaded, and sign-extending/padding as necessary. Let's consider an example: `lbu s1, 1(s0)`, which we want to load byte 1 from `s0` into `s1`. We can first load in the entire word with `lw s1, 0(s0)`. We then can load a byte-1 bitmask via `li t0, 0x0000FF00` and then AND it with `s1` to get only the first byte: `and s1 s1 t0`. Finally, in order to get the byte into byte 0 as desired (at byte 1 right now), we logical right shift by 8, or `srli s1 s1 8`.

We could also do the same with implementing `sb` via `sw`. Let's say we want to implement `sb s1 3(s0)`, i.e. storing a byte from address `s0 + 3` to `s1`. We only have `sw`, so we'd have to manipulate the bits perfectly such that `sw` at `0(s0)` stores the single byte in the right spot (index 3).

11 Other Useful Instructions

`mul, mulh, div, rem` are RISC-V *extensions*: most, but not all RISC-V CPUs will support them. `mul` gets the lower 32 bits of the 32-bit multiplication while `mulh` gets the upper 32 bits.

There are also bitwise operations: `and, or, xor` as well as their immediate versions.

Next up are compare instructions: `slt` and `slti`, with format

`slt(i) dist reg1 reg2`: This compares `reg1` to `reg2` in some way (in this case if `reg1 < reg2`), and if TRUE, then `dst` is set to 1.

Finally, we have `ecall`. This is how we interact with our OS: it takes the values in the `a` registers and performs special functions such as:

- Printing values
- Exiting program
- Allocating memory (heap).