Understanding RISC-V Calling Convention

Nick Riasanovsky

Abstract

In these notes I will attempt to explain RISC-V calling convention and try and give some understanding for why we use this convention. Hopefully understanding these notes will give you better intuition for understanding RISC-V programs in general, making debugging easier.

1 RISC-V Convention

As you are probably now aware, assembly level programming is very different from the programming you have done so far in higher level languages like Java and C. One key detail is that assembly programming doesn't have anything like argument checks and everything is the result of "convention." When you think of "convention" you might think of things like how you name your functions, how long should each line be, etc, basically a bunch of choices to make your code consistent but not essential to functionality. However assembly is entirely based on convention, so if you do not strictly follow convention you will not have working code (unless you write literally all of the assembly you ever use...yuck). Understanding convention in RISC-V consists of 3 important parts: registers, function calls, and entering/exiting a function (prologue/epilogue).

1.1 Registers

In RISC-V each of the 32 registers has a different name which indicates its intended usage. We will not cover the exact meaning of every register in these notes but here are the important ones for this course:

Register Name(s)	Usage
x0/zero	Always holds 0
ra	Holds the return **address**
sp	Holds the address of the boundary of the stack
t0-t6	Holds temporary values that **do not** persist after function calls
s0-s11	Holds values that persist after function calls
a0-a1	Holds the first two arguments to the function or the return values
a2-a7	Holds any remaining arguments

Now let's cover a few of these in a bit more detail. **ra** holds the return address. This is a memory value in the code region. In particular it is useful for function calls. Let's imagine what this means by looking at some pseudocode:

```
def foo ():
    x = 1
    bar ()
    z = 2

def bar ():
    y = 7
```

Imagine we are at the line **y = 7** in bar. When bar finishes we want to resume execution inside of foo and complete the next instruction, **z = 2**. We do this by storing the address of the instruction where the code should return. In this case **ra** would hold the address of **z = 2** so execution resumes as expected after a function call.

The **sp** register holds the current base of the stack. In the C memory management section of the course we discussed how the stack grows downward with each function call. In RISC-V when we need to add more space onto the stack we will decrement **sp** (because the stack grows downward) which gives us more addresses at which to store data. Then when we exit a function we will increment **sp** to restore the stack back to the state when entering the function. One detail that may be unclear is where we store local variables. In the memory management portion we said they were stored on the stack, but trips to and from memory are very expensive, so if we don't need to use the address or store a value directly in memory we will avoid doing so.

The **t** and **s** registers serve a very similar purpose but crucially behave differently when interacting with functions. The **t** register values are not guaranteed to exist after calling a function, in fact it is an error to assume they do. In contrast **s** registers should be used for values needed after a function call. In the prologue/epilogue section we will explain how we achieve this convention.

Finally **a** registers are used to values between function calls. There can be up to two return values passed through registers and eight arguments (if you need more you use the stack but we will not cover how in this class).

1.2 Function Calls

We make a function call using a **jal** instruction to a label or a **jalr** instruction to a register rd. In particular this instructions should be **jal ra label** or **jalr ra rd imm** but we will sometimes abbreviate it with the pseudo-instruction **jal label** or **jalr rd** (when imm is 0). What this **jal** does is store $PC + 4$ in **ra**, which is the address of the next instruction to run after the function call and increment the PC by the offset to the label. **jalr** is similar except it sets the PC value to $rd + imm$.

This is different but similar from a standard jump used in a loop. Jumping to a label without making a function call is done with the instructions **jal x0 label** and **jalr x0 rd imm**, sometimes abbreviated with the pseudo-instructions **j label** and **jr rd** (when imm is 0). To do this we are exploiting the always 0 property of x0. The instruction will attempt to store $PC + 4$ in x0, but because x0 is always 0 nothing will be stored. In this way we can jump somewhere in

the code without providing a location to return to, which is what separates this from a function call.

It's important to note that when we do recursion this is exactly the same as calling any other function call so we will use a **jal label**. Technically if we are a little clever we can implement some functions to be tail recursive using a jump instruction (if you remember this from 61a). We will not cover how to do this but is a good exercise to test if you understand calling convention and why we do each step.

When calling a function we will pass in the arguments to the **a** registers. Then when we return we will look for return values is **a0-1**. Crucially this implies that the **a** registers are not preserved across function calls (or else how would we return values).

1.3 Prologue/Epilogue

The final crucial step to achieving our calling convention is to introduce the prologue and epilogue. This is where we meet our guarantees. Namely these are:

- The **sp** will have the same value when exiting the function that it did entering (unless we store return values on the stack).
- All **s** registers will have the same value exiting the function that they did entering.
- The function will return to the value stored in **ra**, assuming no abnormal execution.

To achieve this we add a section before our function called the prologue and a section after called the epilogue. The prologue looks like this in general:

```
def prologue():
    decrement sp by num s registers + local var space
    Store any saved registers used
    Store ra if a function call is made
```

The epilogue looks like this in general:

```
def epilogue():
    Reload any saved registers used
    Reload ra (if necessary)
    Increment sp back to previous value
    Jump back to return address
```

Following this general procedure we can always meet our guarantees and write code that properly interacts with other people's procedures.

Let's look at an example for a function $sum\_squares(n)$ which sums the values of calling a function square on every value from 1 to n.

```
sum_squares:

prologue:
    addi sp sp -16
    sw s0 0(sp)
    sw s1 4(sp)
    sw s2 8(sp)
    sw ra 12(sp)

    li s0 1
    mv s1 a0
    mv s2 0

loop_start:
    bge s0 s1 loop_end
    mv a0 s0
    jal square
    add s2 s2 a0
    addi s0 s0 1
    j loop_start

loop_end:
    mv a0 s2

epilogue:
    lw s0 0(sp)
    lw s1 4(sp)
    lw s2 8(sp)
    lw ra 12(sp)
    addi sp sp 16
    jr ra
```

Notice that we store values in the **s** registers because we need those values for after the function call. We store enough stack space for each of those **s** registers to be restored and **ra** because we call a function.

2 Why choose this convention?

A decent question to ask when confronted with convention is why do we this. We use the concept of **sp** and **ra** because these are general programming assumptions that need to be met. **x0** is for efficiency because it can be very useful to always have a source of 0. In general we want to avoid the model where we have to save every register we use. Doing so is both burdensome and possibly wasteful because we could end up saving registers that are never changed. Additionally it would be wasteful to always save values so instead we specify exactly what values to save using **s** registers. This implies that we shouldn't always save **a** registers (after all we may not always need the arguments later). This is crucially why we use the **a** registers for return values as well. We cannot ever assume **a** registers persist so we can always use them, making any other dedicated return registers wasteful.

3 Violating Convention

Don't. Just Don't. Sometimes you may find that you can get your code to work correctly even if you don't follow convention. This of course can happen but provides no guarantees. In particular many students will look in functions they call for t registers it does not use and use different t registers as though they are s registers. However this directly violates our abstraction barrier and prevents us from later modifying this function. So I repeat 1 more time for emphasis, **don't**.

4 Using Convention to Debug

Now that we know this convention how can we use it to debug. Assembly code can get complicated and lengthy. As a result while you can step through every single instruction it may not be desirable to do so. Here are some tips to help you debug.

- Check that you stored **ra** properly. For recursion make sure you link **ra** for each recursive call. You can test this by putting a break point at the end of the epilogue and seeing where you return.
- Check that you don't use any **t** registers after a function call.
- Check that **sp** enters and exits with the same value.
- Check the number of times you enter the prologue equals the number of times you enter the epilogue.
- Make sure you restore every register you modified.

These checks won't help solve every debugging problem but can find difficult looking cases that result from violating the assumptions with which we program.