Note 8: RISC-V Functions

Kevin Moy

1 PseudoInstructions

Remember that assembly language is architecture specific: RISC-V can only be run on RISC-V supported CPU chips and vice versa. However, (higher-level) code can be compiled into various assembly languages, each fitting a different architecture.

Pseudoinstructions are instructions that aren't hardware-supported but are instead *translated* into a series of real instructions. For example, `mv dst reg1` translates into `addi dst reg1 0`. The point is that the names of these pseudoinstructions are more human-readable.

Some other pseudoinstructions include:

- `li dst imm`- load immediate `imm` into `dst`
- `la dst label`- load address of `label` into `dst`
- `nop` - do nothing

Finally, the `j` (jump) instruction is also a pseudoinstruction.

2 Converting C to RISC-V

Let's learn the complex job of the compiler: converting C into assembly.

Take the string copy example in C below, where we copy a string `p` to string `q`:

```
char *p, *q;
while ((*q++ = *p++) != '\0');
```

The key is writing out *every single step* of C code as an instruction, as detailed as possible. It helps to write out pseudocode, as below:

```
# copy string p to q
# p->s0, q->s1 (char* pointers)
lb t0 0(s0)                # store *p in t0
sb t0 0(s1)                   # Set *q to t0 = *p
                # Increment p
                # Increment q
                # if *p = t0 == 0, goto Exit
j Loop          # else goto Loop.
Exit: ...
```

3 RISC-V Function Calling

There are six steps to calling a RISC-V function:

1. Place arguments in correct (argument) register
2. Transfer control to function
3. Acquire local storage (stack frame)
4. Execute function
5. Place return value in appropriate place, ”clean up” stuff allocated on stack
6. Return control to caller

We now need to organize our 32 registers such that we can use them without worry that we will lose an important value they hold.

3.1 Argument Registers

Function parameter arguments should be placed in the eight `a0-a7` registers. Return values should be placed in `a0` or `a1`. Order of arguments do matter! If you have 3 arguments, for example, place them in `a0,a1,a2` for simplicity. We overwrite `a0` or `a1` upon returning (we don't need the func arguments anymore). If you run out of memory/need more registers, use the stack- DO NOT use the other registers.

Let's look at an example where we call `add(a,b)`. Examine the simple code below:

```
int main(){
    a = 3;
    b = a+1;
    a = add(a,b);
}
int add(a,b){
    return a+b;
}
```

We want to convert this to RISC-V. Let's do it.

```
main:
    addi a0 x0 3
    addi a1 a0 1
    jal ra, add #Set ra (RETURN ADDRESS) to be this instruction's address

add:
    add a0 a0 a1 #Put result in implicit return register a0
    jr ra #Jump back to ra
```

The `sp` register holds the memory address at the bottom of the stack - the "most recent" stack address used. Only edit `sp` if we're opening/closing stack frames.

3.2 Jump instructions

The `jal ra Label` (jump-and-link) instruction is doing two things: jumping to `Label` but also *linking* `ra` to the current instruction address PLUS 4 (we don't want infinite recursion). Every instruction is a word (4 bytes) long.

The `jr src` (jump register) instruction will jump to the address pointed to by `src`. The vast majority of these instructions will be function return statements, i.e. `jr ra`.

The `jalr dst src imm` jumps to the address at `src`, offsets that address by `imm` bytes, and links PC + 4 to `dst`.

`ret` is another pseudocode instruction used to return from a function (label). This basically just sets the PC to `ra`. This is actually pseudocode for `jalr x0 ra 0`, or `jr ra` (which is also pseudocode).

3.3 Local Variable Storage

The stack pointer starts at the highest possible address. **Decrementing** sp is like allocating stack memory. For example, to store a word on the stack, we'd have:

```
addi sp sp -4
sw t0 0(sp)
```

We decrement the stack pointer by 4 (bytes), then store a word. We can obviously decrement by different amounts for different amounts of storage (byte, word). "Cleaning up" stack frames is just re-incrementing the stack pointer.

4 RISC-V Calling Convention

Calling convention is all about knowing which registers are safe to change in a function call- "safe" meaning we don't lose important data. This is just to make your life a lot easier, so follow this convention religiously!!

Let's see an example where this can get tricky:

```
int f(x,y){
    return g(x,x) + x;
}
int g(x,y){
    return x+y;
}
```

First of all, we have another function call inside a function call- so `ra` will take two different values in calling `f`. We need a way to *save* the initial `ra`.

5 RISC-V Caller-Callee Convention

The convention establishes which registers will be unchanged after a function call, and which registers could possibly have been changed.

The **caller** is the calling function. The **callee** is the function being called. So in our above example, `f` would be the caller of `g`, and `g` would be the callee.

The **saved registers**, which include `s0-s11` and `sp`, are expected to be unchanged before and after a function call. **If a function needs to use a saved register, the function must restore the original value before returning**. To do this, we will save the old values into memory (on the stack) then reload them back when finished. We call these registers **callee-saved**, since they are responsible for maintaining the registers' original values.

There are also **volatile registers**, registers that can be freely changed by the callee function. These include registers `t0-t6` (temp regs), `a0-a7`(arg regs), and `ra` (return address). If the caller needs to mutate these registers, it must save values (to stack memory) before making a function call. We call these registers **caller-saved**.

Let's implement the above code in RISC-V, displayed again for convenience:

```
int f(x,y){
    return g(x,x) + y;
}
int g(x,y){
    return x*y;
}
```

Here it is in RISC-V:

```
f:
    addi sp sp -8 #space on stack
    sw ra, 4(sp)  #save ra to stack (caller saved)
    sw a1 0(sp)   #save y to stack (caller saved)
    add a1 a0 x0  #set a1 to the same as a0
    jal ra g      #call g
    lw a1 0(sp) #RESTORE a1 (caller-saved)
    add a0 a0 a1 #Return our value a0 = return value of g(x,x) + a1 = y
    lw ra, 4(sp) #restore return address, since calling g changed it
    addi sp sp 8 #Clean up stack since we finna return
    jr ra       #ret
g:
```

Immediately, as the caller the first thing we do is note our caller-saved registers we need- that is, `a1` and `ra`, and immediately save their values onto the stack. Note that if an argument **position** changes throughout all your function calls, you'll need to save the corresponding argument register. So we see that `a0=x` won't really change but `a1 = y` certainly will, in the second argument call of `g`! So we must save that old value of `a1` to the stack first. Note whenever we enter a subroutine we also must save `ra`.

5.1 RISC-V Function Structure

A function generally goes like this:

```
#PROLOGUE
func_label:
    addi sp sp (x)
    sw ra (x-4)(sp)
    sw other CALLEE-SAVED registers
    sw other registers (if this function is a caller)
#BODY
    ... (call other functions)
#EPILOGUE
    #Restore other registers
    # Restore callee-saved registers
    lw ra (x-4)sp
    addi sp sp x
    jr ra
```

It is generally good practice to code the function body logic first, THEN save registers on the stack as necessary, and finally decrement `sp` as needed.

6 Summary

It's extremely important to know these rules. One more time:

- Callers must save **volatile registers** before making a call
- Callers can trust **saved registers** to be unchanged before and after a call (by the callee).
- If a callee wants to overwrite a saved register, it must save and restore them in the prologue and epilogue!