CS 61C Fall 2024 — RISC-V Calling Convention

Discussion 4

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 After calling a function and having that function return, the `t` registers may have been changed during the execution of the function, while `a` registers cannot.

False. `a0` and `a1` registers are often used to store the return value from a function, so the function can set their values to the its return values before returning.

1.2 In order to use the saved registers (`s0-s11`) in a function, we must store their values before using them and restore their values before returning.

True. The saved registers are callee-saved, so we must save and restore them at the beginning and end of functions. This is frequently done in organized blocks of code called the "function prologue" and "function epilogue".

1.3 The stack should only be manipulated at the beginning and end of functions, where the callee saved registers are temporarily saved.

False. While it is a good idea to create a separate 'prologue' and 'epilogue' to save callee registers onto the stack, the stack is mutable anywhere in the function. A good example is if you want to preserve the current value of a temporary register, you can decrement the `sp` to save the register onto the stack right before a function call.

2 Calling Convention

Let's review what special meaning we assign to each type of register in RISC-V.

Register	Convention	Saver
`x0`	Stores **zero**	N/A
`sp`	Stores the **stack pointer**	Callee
`ra`	Stores the **return address**	Caller
`a0 - a7`	Stores **arguments** and **return values**	Caller
`t0 - t6`	Stores **temporary** values that *do not persist* after function calls	Caller
`s0 - s11`	Stores **saved** values that *persist* after function calls	Callee

To save and recall values in registers, we use the `sw` and `lw` instructions to save and load words to and from memory, and we typically organize our functions as follows:

```
1  # Prologue
```

```
addi sp sp -8 # Room for two registers. (Why?)
sw s0 0(sp) # Save s0 (or any saved register)
sw s1 4(sp) # Save s1 (or any saved register)

# Code omitted

# Epilogue

lw s0 0(sp) #Load s0 (or any saved register)
lw s1 4(sp) #Load s1 (or any saved register)
addi sp sp 8 #Restore the stack pointer
```

Now, let's see what happens if we ignore calling convention.

2.1 Consider the following blocks of code:

```
main:
# Prologue
# Saves ra

# Code omitted
addi s0 x0 5
# Breakpoint 1
jal ra foo
# Breakpoint 3
mul a0 a0 s0
# Code omitted

# Epilogue
# Restores ra
j exit
```

```
foo:
# Preamble
# Saves s0

# Code omitted
addi s0 x0 4
# Breakpoint 2

# Epilogue
# Restores s0
jr ra
```

(a) Does `main` always behave as expected, as long as `foo` follows calling convention?

Yes, since `foo` saves the saved registers, and `main` saves the return address

(b) What does `s0` store at breakpoint 1? Breakpoint 2? Breakpoint 3?

5, then 4, then 5

(c) Now suppose that `foo` didn't have a prologue or epilogue. What would `s0` store at each of the breakpoints? Would this cause errors in our code?

5, then 4, then still 4. This would cause errors, since we use the value of `s0` in our calculations.

In part (c) above, we saw one way how not following calling convention could make our code misbehave. Other things to watch out for are: assuming that `a` or `t` registers

will be the same after calling a function, and forgetting to save `ra` before calling a function.

Function `myfunc` takes in two arguments: a0, a1. The return value is stored in a0. In `myfunc`, `generate_random` is called. It takes in 0 arguments and stores its return value in `a0`.

```
1  myfunc:
2      # Prologue (omitted)
3
4      addi t0 x0 1
5      slli t1 t0 2
6      add t1 a0 t1
7      add s0 a1 x0
8
9      jal generate_random
10
11     add t1 t1 a0
12     add a0 t1 s0
13
14     # Epilogue (omitted)
15     ret
```

2.2 Which registers, if any, need to be saved on the stack in the prologue?

`s0`, `ra`. We must save all s-registers we modify. In addition, if a function contains a function call, register ra will be overwritten when the function is called (i.e. jal ra label). ra must be saved before a function call. It is conventional to store ra in the prologue (rather than just before calling a function) when the function contains a function call. myfunc contains the function call generate_random.

2.3 Which registers do we need to save on the stack before calling `generate_random`?

`t1`.

Under calling conventions, all the t-registers and a-registers may be changed by generate_random, so we must store all of these which we need to know the value of after the call. A total of 2 t-registers are used before calling generate_random, t0 and t1, but only t1's value is referenced again after the function call.

2.4 Which registers need to be recovered in the epilogue before returning?

`s0`, `ra`. This mirrors what we saved in the prologue.

3 Recursive Calling Convention

Write a function `sum_square` in RISC-V that, when given an integer `n`, returns the summation below. If `n` is not positive, then the function returns 0.

$$n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$$

To implement this, we will use a tail-recursive algorithm that uses the `a1` register to help with recursion. More specifically, you will be writing the following function:

`sum_squares_recursive`: Return the value $m + n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$		
Arguments	`a0`	A 32-bit number. n. You may assume $n \leq 10000$.
	`a1`	A 32-bit number. m.
Return value	`a0`	$m + n^2 + (n-1)^2 + (n-2)^2 + \ldots + 1^2$. If $n \leq 0$, return m.

When the above function is called with `a1` set to 0, we will get the behavior that we expect. For this problem, you are given a RISC-V function called `square` that takes in a single integer and returns its square.

`square`: Square a number		
Arguments	`a0`	n.
Return value	`a0`	n^2

3.1 Since this is a recursive function, let's start with the base case of our recursion.

```
sum_squares:
    bge x0 a0 zero_case

    # To be implemented in the next question.

zero_case:
    mv a0 a1
    jr ra
```

3.2 Next, implement the recursive logic. *Hint: If you let $m' = m + n^2$, then*

$$m + n^2 + (n-1)^2 + \ldots + 1^2 = m' + (n-1)^2 + \ldots + 1^2$$

```
sum_squares:
    # Handle zero case (previous question)
    ______________ zero_case

    mv t0 a0
    jal ra square

    add a1 t0 a1
```

```
    addi a0 t0 -1

    jal ra sum_squares
    jr ra
zero_case:
    # Handle zero case (previous question)
    _____________
    jr ra
```

3.3 Now, think about calling convention from the caller perspective. After the call to `square`, what is in `a0`? `a1`? Which one of the registers will cause a calling convention violation?

`a0` will contain n^2, and `a1` will contain garbage data, causing a calling convention violation. The register `t0` will also hold garbage, which would also cause a CC violation.

3.4 What about the recursive call? What will be in `a0` after the call to `sum_squares`? `a1`?

`a0` will contain $m + n^2 + \cdots + 1^2$, and `a1` will contain garbage data. However, since `a0` now contains the expected return value, we no longer care about the value in `a1`, and can directly return — it is the job of whichever function called `sum_squares` to deal with saving caller-saved registers if they are still needed.

3.5 Now, go back and fix the calling convention issues you identified. Note that not all blank lines may be used. There may also be another caller saved register that you need to save as well!

```
sum_squares:
    # Handle zero case (previous question)

    # Save caller saved register on the stack
    mv t0 a0

    addi sp sp -12
    sw a1 0(sp)
    sw t0 4(sp)
    sw ra 8(sp)
    jal ra square
    # Restore register and stack
    lw a1 0(sp)
    lw t0 4(sp)
    lw ra 8(sp)
    addi sp sp 12

    add a1 a0 a1
    addi a0 t0 -1
```

```
    addi sp sp -4
    sw ra 0(sp)
    jal ra sum_squares
    lw ra 0(sp)
    addi sp sp 4

    jr ra
zero_case:
    # Handle zero case (previous question)
    jr ra
```

3.6 Now, from a callee perspective, do we have to save any registers in the prologue and epilogue? If yes, what registers do we have to save, and where do we place the prologue and epilogue? If no, briefly explain why.

No, we do not have to take callee saved registers into account because we do not use any callee saved registers. However, since we call two functions, it is possible to save `ra` in the prologue and restore it in an epilogue immediately before the `jr ra` before the `zero_case` label.