CS61C Summer 2025

RISC-V Calling Convention

Discussion 4

1 Review: RISC-V Memory Access

Using the given instructions and the sample memory array, what will happen when the RISC-V code is executed? For load instructions (`lw`, `lb`, `lh`), write out what each register will store. For store instructions (`sw`, `sh`, `sb`), update the memory array accordingly. Recall that RISC-V is little-endian and byte addressable. For any unknown instructions, use the CS 61C reference card!

1.1

```
li t0 0x00FF0000
lw t1 0(t0)
addi t0 t0 4
lh t2 2(t0)
lw s0 0(t1)
lb s1 3(t2)
```

0xFFFFFFFF	
	...
0x00FF0004	0x000C561C
0x00FF0000	36
	...
0x00000036	0xFDFDFDFD
	...
0x00000024	0xDEADB33F
	...
0x0000000C	0xC5161C00
	...
0x00000000	

What value does each register hold after the code is executed?

`t0`: `0x00FF0004`. Line 3 adds 4 to the initial address.

`t1`: `36`. Line 2 loads the 4-byte word from address `0x00FF0000`.

`t2`: `0xC`. Line 4 loads two bytes starting at the address `0x00FF0004 + 2 = 0x00FF0006`. This returns `0x000C`

`s0`: `0xDEADB33F`. Line 5 loads the word starting at address `36 = 0x24` which is `0xDEADB33F`.

`s1`: `0xFFFFFFC5`. Line 6 loads the MSB starting of the 4-byte word at address `0xC`. The value is `0xC5` which is sign-extended to `0xFFFFFFC5`.

1.2 Update the memory array with its new values after the code is executed. Assume each byte in the memory array is initialized to zero.

```
li t0 0xABADCAF8
li t1 0xF9120504
li t2 0xBEEFDAB0
sw t0 0(t1)
addi t0 t0 4
sh t1 2(t0)
sh t2 0(t0)
lw t3 0(t1)
sb t1 1(t3)
sb t2 3(t3)
```

0xFFFFFFFF	0x00000000
	...
0xF9120504	0xABADCAF8
	...
0xBEEFDAB0	0x00000000
	...
0xABADCAFC	0x0504DAB0
0xABADCAF8	0xB0000400
	...
0x00000000	0x00000000

2 RISC-V 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:
  # Prologue
  # 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?

Breakpoint 1: 5, Breakpoint 2: 4, Breakpoint 3: 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?

Breakpoint 1: 5, Breakpoint 2: 4, Breakpoint 3: 4. This would cause errors because we rely on `s0` having the value of 5 in our calculations in `main`.

In part (c) above, we see 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.

2.2 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`.

```
myfunc:
# Prologue (omitted)

addi t0 x0 1
slli t1 t0 2
add t1 a0 t1
add s0 a1 x0

jal generate_random

add t1 t1 a0
add a0 t1 s0

# Epilogue (omitted)
ret
```

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

Answer: `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) if the function contains a function call. `myfunc` contains the function call `generate_random`.

b) Which registers, if any, need to be saved on the stack before calling `generate_random`?

Answer: `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.

c) Which registers, if any, restored from the stack in the epilogue before returning?

Answer: `s0, ra`

This mirrors what we did in the prologue.

3 Recursive Calling Convention

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

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

To implement this, we will use a tail-recursive algorithm that uses the `a1` register to help with recursion.

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

For this problem, you are given a RISC-V function called `square` that takes in a single integer and returns its square.

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

3.1 Since this a recursive function, let's implement 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*

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

```
sum_squares:
# Handle zero case (previous question)
bge x0 a0 zero_case

mv t0 a0
jal ra square

add  a1 a0 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` and `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 calling convention violation.

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

`a0` will contain $m + n^2 + ... + 1^2$, 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 needed in the future.

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)
mv t0 a0

# Save caller saved registers on the stack
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

# Save caller saved registers on the stack
# Note that we don't need to save a1 and t0
# because we do not need their values
# after the function call.
addi sp sp -4
sw ra 0(sp)
jal ra sum_squares
# Restore caller saved registers on the stack
lw 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.