Lab 4: RISC-V Calling Convention

Deadline: Monday, February 27, 11:59:59 PM PT

Setup

Starter

> You must complete this lab on your local machine. See Lab 0 if you need to set up your local machine again.

In your `labs` directory on your local machine, pull any changes you may have made in past labs:

```
git pull origin main
```

Still in your `labs` directory on your local machine, pull the files for this lab with:

```
git pull starter main
```

If you get an error like the following:

```
fatal: 'starter' does not appear to be a git repository
fatal: Could not read from remote repository.
```

make sure to set the starter remote as follows:

```
git remote add starter https://github.com/61c-teach/sp23-
```

and run the original command again.

Still in your `labs` directory, run the following command to download the newest version of some tools we may need:

```
bash tools/download_tools.sh
```

Venus

Like lab 3, we will be using the Venus RISC-V simulator . Also, please refer to the Venus reference on our course website when you need a refresher on any of the Venus features.

Mount the lab 4 files as you did with Lab 3.

Calling Convention

We have an appendix about calling convention. We recommend reading over it if you are not feeling comfortable, since correct calling convention is crucial for RISC-V programming.

We understand these examples are long, but please try to read through them to get an idea of how to follow calling convention when writing RISC-V.

Example: Putting It Into Practice

Let's say we have the following assembly code structure for a function named `func1`. Let's say `func1` is a function being called by another function (let's say the `main` function). From the code, you will also see that `func1` will also call another function named `func2`. When `func1` was called by the `main` function, `func1` was the callee. However, when `func1` calls `func2`, `func1` becomes the caller. Therefore, `func1` will have to assume the responsibilities of being a callee and a caller when following the calling convention.

```
func1: # modifies a0, t0, and s0. ra points to the `main` f
    # Checkpoint 1: What do you need to do before you start

    # Some block of code using a0, t0, and s0
    # Checkpoint 2: What do you need to do before you call

    # input argument at a0, return value at a0
    jal ra, func2  # call func2
    # Checkpoint 3: What do you need to do after a functior

    # Some block of code using a0, t0, and s0
    # Checkpoint 4: What do you need to do before this func

    jr ra   # function return
```

Saving values into the stack is done using the `sw` instruction to the appropriate stack pointer (`sp`). Retrieving values from the stack is done using the `lw` instruction on the corresponding stack pointer. Whenever you want to save values into the stack, you need to adjust the stack pointer (by decreasing the stack pointer). Whenever you want to retrieve values from the stack, you need to adjust the stack pointer back in reverse (by increasing the stack pointer). Doing these procedures will ensure that the stack pointer will end up being the same at the start and end of the function call. That is how functions utilize the stack memory as temporary storage.

Let's try to fill in the code for every checkpoint in the code.

For Checkpoint 1, since `func1` is the callee (called my `main`), it has to save the callee saved registers that it will be modifying. The code says that we will be modifying `s0` which is a saved register. Moreover, since this `func1` will also be calling another function `func2`, it will also need to save the return address (`ra`). While the `ra` register is labelled as a caller saved register, saving `ra` into the stack is typically done at this step. Since we will be saving 2 registers, we will have to adjust the `sp` register by 2 words (8 bytes). Accordingly, we have the following code:

```
    # Checkpoint 1: What do you need to do before you start
    addi sp sp -8   # Push the stack pointer down by 2 word
    sw ra 0(sp)     # Save the return address register (ra)
    sw s0 4(sp)     # Save the saved register (s0)
```

For Checkpoint 2, we see that `a0`, `t0`, `s0` have been modified before, and we are now about to call `func2` (`func1` now becomes the caller). We see that `func2` takes in the input argument at `a0` and sets the return value also at the same register. Moreover, we see that later on in the code (after Checkpoint 3), we will be using `t0` and `s0` registers again. Thus, in the perspective of `func1`, both of these registers should be unchanged before and after the function call to `func2`. We know that if `func2` is following the calling convention, it should be saving the `s0` register into the stack and retrieving it back to the original value before it returns. That is something that we want. However, `t0` is not saved by `func2` and will have the freedom to change it within the function. Thus, as the caller to `func2`, `func1` is now responsible in saving `t0` into the stack as well. Accordingly, we have the following code:

```
    # Checkpoint 2: What do you need to do before you call
    addi sp sp -4   # Push the stack pointer down by 1 word
    sw t0 0(sp)     # Save the temporary register (t0)
```

For Checkpoint 3, `func2` has now returned with the return value stored at `a0`. `func1` is now about to do some operations using this return value (`a0`) and the `t0` and `s0` registers. Again, if `func2` followed the calling convention, the value of `s0` register should've been unchanged in the perspective of `func1`. However, `t0` might have been modified. Good thing is that we have already saved it in the stack before calling `func2`. We just have to retrieve it now. Accordingly, we have the following code:

```
    # Checkpoint 3: What do you need to do after a function
    lw t0 0(sp)     # Retrieve the saved temporary register
    addi sp sp 4    # Return the stack pointer up by 1 word
```

For Checkpoint 4, we are now at the point where `func1` has finished doing its operations and will be returning back to `main`. However, before it returns, it has to make sure it has accomplished its callee responsibilities as well. Earlier in Checkpoint 1, we saved the `ra` and `s0` registers into the stack. It is now time to retrieve them. During the operation of `func1`, it has modified the `s0` register and the `ra` register (due to the function call to `func2`). As the callee, it has the responsibility to bring them back to their original values such that in the perspective of the `main` function, these registers were unchanged. Accordingly, we have the following code:

```
    # Checkpoint 4: What do you need to do before this func
    lw s0 4(sp)     # Retrieve the original saved register
    lw ra 0(sp)     # Retrieve the original return address
    addi sp sp 8    # Return the stack pointer up by 2 word
```

And with this, we have now satisfied the RISC-V calling convention for `func1`. Here is the complete code:

```
func1: # modifies a0, t0, and s0. ra points to the `main` f
    # Checkpoint 1: What do you need to do before you start
    addi sp sp -8   # Push the stack pointer down by 2 word
    sw ra 0(sp)     # Save the return address register (ra)
    sw s0 4(sp)     # Save the saved register (s0)

    # Some block of code using a0, t0, and s0
    # Checkpoint 2: What do you need to do before you call
    addi sp sp -4   # Push the stack pointer down by 1 word
    sw t0 0(sp)     # Save the temporary register (t0)

    # input argument at a0, return value at a0
    jal ra, func2  # call func2
    # Checkpoint 3: What do you need to do after a function
    lw t0 0(sp)     # Retrieve the saved temporary register
    addi sp sp 4    # Return the stack pointer up by 1 word
```

```
    # Some block of code using a0, t0, and s0
    # Checkpoint 4: What do you need to do before this func
    lw s0 4(sp)     # Retrieve the original saved register
    lw ra 0(sp)     # Retrieve the original return address
    addi sp sp 8    # Return the stack pointer up by 2 word

    jr ra   # function return
```

Let's say register `sp` starts at 0x7FFFFFF0 at the start of `func1` call.

▼ At what memory address is `s0` saved at?

The stack pointer was adjusted -8, but `s0` is stored at 4 + `sp`. Thus, 0x7FFFFFF0 - 8 + 4 = 0x7FFFFFEC

▼ At what memory address is `t0` saved at?

The stack pointer was adjusted -8 earlier, then before saving t0, we adjust it again by -4. Thus, 0x7FFFFFF0 - 8 - 4 = 0x7FFFFFE4

▼ At the end of `func1` (at the line jr ra), where is the stack pointer pointing to?

It should be back at the original value since we have retrieved the saved data and returned the stack pointer afterwards. Thus, 0x7FFFFFF0.

▼ If we didn't implement the code in Checkpoint 2 and 3, what can go wrong?

After Checkpoint 3, it was described that `func1` will be using `t0` again. Since `t0` is a caller saved register, the callee (which is `func2`) has the freedom to change `t0` and does not need to return the original value back. If `t0` was indeed modified by `func2`, then the calculations done by `func1` will be wrong since the function call changed one of its temporary variables.

▼ If we didn't save `ra` at all, what can go wrong?

`ra` points to the address of the calling function. At the start of `func1`, it is pointing to somewhere in the `main` function. Whenever you run the `jal` instruction, it modifies `ra` such that it points to the next line after that instruction. This is done so that when `jr ra` is run, it will continue execution to where the `ra` is pointing at. `func1` has a function call to `func2`. This changes `ra` such that it will point to the instruction after that (`lw t0 0(sp)`). This is now pointing somewhere in `func1` itself. If `ra` was not saved at all, then when `func1` executes the `jr ra` instruction at the end, it will jump back to that instruction (`lw t0 0(sp)`), which is not the intended execution flow.

Example: Converting from C to RISC-V with Calling Convention

In this exercise, you will be guided through how to translate the below C program into RISC-V. If you are looking for an additional challenge, you can translate the code first before looking at the solution.

```
int source[] = {3, 1, 4, 1, 5, 9, 0};
int dest[10];

int fun(int x) {
        return -x * (x + 1);
}

int main() {
    int k;
    int sum = 0;
    for (k = 0; source[k] != 0; k++) {
        dest[k] = fun(source[k]);
        sum += dest[k];
    }
    printf("sum: %d\n", sum);
}
```

Let's start with initializing the `source` and `dest` arrays. Just like we did

in Lab 3, we need to declare our arrays in the `.data` section as seen below:

```
.data
source:
    .word   3
    .word   1
    .word   4
    .word   1
    .word   5
    .word   9
    .word   0
dest:
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
    .word   0
```

Next, let's write `fun`.

```
int fun(int x) {
        return -x * (x + 1);
}
```

Calling convention states that

- We can find `x` in register `a0`.
- We must put our return value in register `a0`

The rest of the code is explained in the comments below.

```
.text
fun:
    addi t0, a0, 1 # t0 = x + 1
```

```
    sub t1, x0, a0 # t1 = -x
    mul a0, t0, t1 # a0 = (x + 1) * (-x)
    jr ra # return
```

▼ Ask yourself: why did we not save `t1` before using it?

`t1` is a caller saved register. Calling convention does not guarantee that caller saved registers will remain unchanged after a function call. Therefore, `fun` can change `t1` without saving its old value. If the function that called `fun` had a value stored in `t1` that it wants to use after `fun` returns, it would need to save `t1` before calling `fun`.

Let's move onto main. (We are going to ignore calling convention for a minute).

```
int main() {
    int k;
    int sum = 0;
```

The above code becomes the following:

```
main:
    addi t0, x0, 0 # t0 = k = 0
    addi s0, x0, 0 # s0 = sum = 0
```

We have to initialize `k` to 0 because there is no way to declare a variable in RISC-V and not set it.

Next, let's load in the address of the two arrays.

```
    la s1, source
    la s2, dest
```

Remember that `la` loads the address of a label. This is the only way that we can access the address of `source` and `dest`. `s1` is now a pointer to the source array and `s2` is now a pointer to the `dest` array.

Let's move on to the loop.

```
for (k = 0; source[k] != 0; k++) {
    dest[k] = fun(source[k]);
    sum += dest[k];
}
```

First, we'll construct the outer body of the loop.

```
loop:
#1  slli s3, t0, 2
#2  add t1, s1, s3
#3  lw t2, 0(t1)
#4  beq t2, x0, exit
    ...
#5  addi t0, t0, 1
#6  jal x0, loop
exit:
```

1. Lines 1-3 are needed to access `source[k]`. First we want to compute the byte offset of the element. We are dealing with `int` arrays, so the size of each element is `4 bytes`. This means that we need to multiply `t0` (k) by 4 to compute the byte offset. To multiply a value by 4, we can just shift it left by 2.
2. Next, we need to add the offset to the array pointer to compute the address of `source[k]`.
3. Now that we have the address, we can load the value in from memory.
4. Then, we check to see if `source[k]` is 0. If it is, we jump to the exit.
5. At the end of the loop, we increment `k` by 1
6. Finally, we loop back the to beginning

Now, Let's fill in the rest of the loop (ignoring calling convention at first)

```
loop:
    slli s3, t0, 2
    add t1, s1, s3
```

```
    lw t2, 0(t1)
    beq t2, x0, exit
#1  add a0, x0, t2 # 1
    ...
#2  jal fun # 2
    ...
#3  add t3, s2, s3 # 4
#4  sw a0, 0(t3) # 5
#5  add s0, s0, a0 # 6
    addi t0, t0, 1
    jal x0, loop
exit:
```

1. Fun takes in the argument `x`. We must pass this argument through `a0` so that `fun` will know where to find it.
2. Call `fun`. `jal` automatically saves the return address in `ra`.
3. Next, we want so store this value in `dest`. First we need to compute the address of where we want to store the value in `dest`. Remember that we can reuse the `offset` that we computed earlier (this can be found in `s3`). `s2` is a pointer to the beginning of `dest`.
4. Store value at `dest[k]`. Remember that `fun` placed the return value in `a0`.
5. Increment `sum` by `dest[k]`

Now, let's add in the proper calling convention around `jal fun`. Before scrolling down, ask yourself what code we need to add to meet calling convention.

To meet calling convention (and therefore have our code behave as expected), we need to save any caller saved registers whose values we want to remain the same after calling `fun`. In this case, we can see that we use registers `t0`, `t1`, `t2`, and `t3` in `main`.

▼ Do we need to save and restore all of these registers?

No, we only need to save and restore `t0`. We use `t1` and `t2` before `fun`, but we don't reuse their after. We write to `t3` after `fun`, so we

don't care what its old value was.

`t0` is the only caller saved register we have whose value must stay the same before and after `fun`.

Let's add the proper calling convention code around `jal fun`.

```
addi sp, sp, -4
sw t0, 0(sp)
jal fun
lw t0, 0(sp)
addi sp, sp, 4
```

Next, let's move on to `exit` (excluding calling convention for the moment).

```
exit:
    addi a0, x0, 1 # argument to ecall, 1 = execute print i
    addi a1, s0, 0 # argument to ecall, the value to be pri
    ecall # print integer ecall
    addi a0, x0, 10 # argument to ecall, 10 = terminate prc
    ecall # terminate program
```

The final sum is stored in `s0`. To return this value, we need to store it in `a0`.

Now we have completed the logic of our program. Next we need to finish up calling convention for `main`.

▼ Think to yourself, which piece of the calling convention is missing?

We are overwriting callee saved registers without saving them! Remember that it is the callee's job to ensure that callee saved registers have the same value at the start and end of the function.

▼ Which callee saved registers do we need to save?

We need to save any callee saved registers that we used. We don't

know which of these registers are being used by the function that called us, so we have to save all of the ones that we overwrite. In this case, that would be registers `s0–s3` and `ra`.

It might be tricky understanding why we need to save `ra`. Remember that another function called `main`. When that function called `main`, it stored a return address in `ra` so that `main` would know where to return to when it finished executing. When `main` calls `fun`, it needs to store a return address in `ra` so that `fun` knows where to return to when it finishes executing. Therefore, `main` must save `ra` before it overwrites it.

Below, you can find the prologue and epilogue for `main`:

```
main:
    # BEGIN PROLOGUE
    addi sp, sp, -20
    sw s0, 0(sp)
    sw s1, 4(sp)
    sw s2, 8(sp)
    sw s3, 12(sp)
    sw ra, 16(sp)
    # END PROLOGUE
    ...
    ...
exit:
    addi a0, x0, 1 # argument to ecall, 1 = execute print i
    addi a1, s0, 0 # argument to ecall, the value to be pri
    ecall # print integer ecall
    # BEGIN EPILOGUE
    lw s0, 0(sp)
    lw s1, 4(sp)
    lw s2, 8(sp)
    lw s3, 12(sp)
    lw ra, 16(sp)
    addi sp, sp, 20
    # END EPILOGUE
    addi a0, x0, 10 # argument to ecall, 10 = terminate prc
    ecall # terminate program
```

You can find the entire program in `example_c_to_riscv.s`.

Exercise 1: Calling Convention Checker

Calling convention errors can cause bugs in your code that are difficult to find. The calling convention checker is used to detect calling convention violations in your code. However, it is **not** comprehensive. In this exercise, you will use the calling convention checker to fix some calling convention issues.

Note: Venus's calling convention checker will not report all calling convention bugs; it is intended to be used primarily as a basic check. Most importantly, **it will only look for bugs in functions that are exported with the `.globl` directive** - the meaning of `.globl` is explained in more detail in the Venus reference.

1. To enable the calling convention checker, **click** on the Venus tab at the top of the page, and **click** "Enable" for the "Calling Convention" row under the "Settings" pane.

- You can also run the calling convention checker in your command line using the `-cc` flag. For example, `java -jar tools/venus.jar -cc lab04/ex1.s`.

2. **Open** `ex1.s` in the simulator tab.
3. **Run** the simulator, and you should see some errors similar to the errors below.

```
[CC Violation]: (PC=0x0000004C) Setting of a saved register
[CC Violation]: (PC=0x00000054) Setting of a saved register
[CC Violation]: (PC=0x00000054) Setting of a saved register
[CC Violation]: (PC=0x00000054) Setting of a saved register
[CC Violation]: (PC=0x00000054) Setting of a saved register
[CC Violation]: (PC=0x00000054) Setting of a saved register
```

```
[CC Violation]: (PC=0x00000054) Setting of a saved register
[CC Violation]: (PC=0x00000054) Setting of a saved register
[CC Violation]: (PC=0x00000064) Save register s0 not correc
[CC Violation]: (PC=0x00000070) Setting of a saved register
[CC Violation]: (PC=0x00000074) Setting of a saved register
[CC Violation]: (PC=0x000000A4) Setting of a saved register
Found 12 warnings!
--------------------
[ERROR] An error has occurred!

Error:
`SimulatorError: Attempting to access uninitialized memory
```

More information about these errors can be found in the Venus reference.

4. Using the printed errors, **resolve** all calling convention errors in `ex1.s`.

- The fixes for all of these errors (both the ones reported by the CC checker and the ones it can't find) should be added near the lines marked by the `FIXME` comments in the starter code.
- Your output should look similar to

```
Tests passed.
Found 0 warnings!
```

After you finish the exercise, be sure that you can answer the following questions.

▼ Is `next_test` a function?

No, it's just a label that is used to skip over the call to `failure`. If a label is a function, we will jump and link to it because we always want our functions to return back to us. In this case, we just jumped to `next_test`.

▼ What caused the errors in `pow`, and `inc_arr` that were reported

by the Venus CC checker?

- `pow`: Missing epilogue and prologue.
- `inc_arr`: Failure to save `s0`, `s1` in prologue/epilogue, and failure to save `t0` before calling `helper_fn`.

▼ In RISC-V, we call functions by jumping to them and storing the return address in the `ra` register. Does calling convention apply to the jumps to the `pow_loop` or `pow_end` labels?

No, since they're not functions, we don't need to return to the location the function was called.

▼ Why do we need to store `ra` in the prologue for `inc_arr`, but not in any other function?

`inc_arr` itself calls another function - `ra` holds the address of the instruction to continue executing after returning, which is overwritten when we call another function since we need to be able to return to the body of `inc_arr`.

▼ Why wasn't the calling convention error in `helper_fn` reported by the CC checker? (Hint: it's mentioned above in the exercise instructions.)

It's not declared `.globl`. The calling convention checker will not test functions that are not declared `.globl`. Note: The bug in helper_fn will initially be reported by the checker, but when the bugs in the function that calls it are fixed, this one is no longer reported.

Exercise 2: Fixing CC Errors: `s` Edition

In this exercise (and the next exercise), the starter code has one or

more common calling convention errors we've seen in past semesters. Your task will be fixing these errors, and hopefully avoiding these errors yourself when you are writing RISC-V.

> **Warning**: Please do not change any lines of the starter code. You may only add lines to the starter.

1. **Open** `ex2.s` and **run** it in Venus. **Note** that the program does not exit because of an infinite loop.
2. **Fix** the calling convention errors. The error is within the `ex2` function.

Exercise 3: Fixing CC Errors: `t` Edition

Similar to the last exercise, the starter code for this exercise has one or more common calling convention errors we've seen in past semesters. Your task will be fixing these errors, and hopefully avoiding these errors yourself when you are writing RISC-V.

> **Warning**: Please do not change any lines of the starter code. You may only add lines to the starter.

1. **Open** `ex3.s` and **run** it in Venus. **Note** that the output seems garbled.
2. **Fix** the calling convention errors. The error is within the `ex3` function.

Submission

Save, commit, and push your work, then submit to the **Lab 4** assignment on Gradescope.
