Digital Design Verification

Lab Manual # 16 – Calling Convention in RISC-V Assembly

Release: 1.0

Date: 05-March-2024

NUST Chip Design Centre (NCDC), Islamabad, Pakistan

Revision History

Revision Number	Revision Date	Revision By	Nature of Revision	Approved By
1.0	05/03/2024	Hira Sohail, M. Bilal	Complete manual	-
1.1	25/03/2024	Hira Sohail	Revision	-

Contents

Objective

The objective of this lab is to:

- Understand and practice Calling Convention in RISC-V Assembly
- Learn finding bugs using calling convention checker in Venus

Tools

- Venus

Overview of Calling Convention:

What is calling convention?

According to calling convention, when we call a function, that function promises to leave some, but not all, registers unchanged.

(Reminder: The **caller** is the function making the call, and the **callee** is the function that is being called. ALL functions could be a caller and a callee; we as programmers should never assume a function isn't something because we have no knowledge about where a function may or may not be called from. Functions that call other functions are always a caller, even if they're by default a callee.**)**

The registers that a function promises to leave unchanged are the **callee-saved registers** (preserved registers). s0 through s11 (saved registers) and sp are preserved registers.

The registers that a function does not promise to leave unchanged are the **caller-saved registers** (non-preserved registers). a0 through a7 (argument registers), t0 through t6 (temporary registers), and ra (return address) are non-preserved registers.

The caller perspective

When we call a function, that function promises to not modify any of the preserved registers, from the perspective of the caller. This means that when the function returns, we can be sure that the preserved registers have not changed. This does **not** however, guarantee that function called will not modify preserved register values across the function call; it just guarantees that from the perspective of the caller, the preserved register values appear unchanged.

```
addi s0, x0, 5    # s0 contains 5
jal ra, func      # call a function
addi s0, s0, 0    # s0 still contains 5 here!
```

However, that function is allowed to freely modify any of the non-preserved registers. This means that as soon as you call a function and the function returns, every non-preserved register now contains *garbage*. You do not know what values are in the non-preserved registers anymore.

NOTE:

Garbage refers to unknown values, even if the values in non-preserved registers remain unchanged across a function call. This is because our guarantees don't say that non-preserved registers are untouched, so even if they aren't changed, we have to assume they are.

```
addi t0, x0, 5     # t0 contains 5
jal ra, func       # call a function
addi t0, t0, 0     # t0 contains garbage!
```

This is a common calling convention bug**:** when a function returns, you cannot rely on the values in any non-preserved register.

One way to avoid this bug is to save the values in the non-preserved registers on the stack before calling the function, then restore the values in the non-preserved registers after the function returns. For example:

```
addi t0, x0, 5     # t0 contains 5
addi a0, t0, 10    # a0 (argument to func) contains 15

# Prologue
addi sp, sp, -8    # decrement stack
sw t0, 0(sp)       # store t0 value on the stack
sw a0, 4(sp)       # store a0 value on the stack

# Function call
jal ra, func       # call a function
mv s0, a0          # save return value 1, before restoring a0's old value to avoid overwriting the return value
mv s1, a1          # save return value 2, before moving on, in case a1 is used in the future, to avoid potentially overwriting the return value

# Epilogue
lw t0, 0(sp)       # restore t0 value from the stack
lw a0, 4(sp)       # restore a0 value from stack
addi sp, sp, 8     # increment stack

addi t0, t0, 0     # t0 contains 5 here because you saved it before the function call!
xori t1, a0, t0    # t0^a0 safely here
```

This is why the non-preserved registers are often called caller-saved registers, because the caller must save them when calling a function.

The callee perspective

When we write a function, we are allowed to freely change any of the non-preserved registers. However, we must promise not to change any of the preserved registers, from the perspective of the caller.

This is another common calling convention bug**:** a function cannot noticeably change any preserved registers. In other words, change whatever you'd like, callee, just make sure you restore the preserved register values before returning to the caller, and don't worry about the non-preserved registers.

If we want to use one of the preserved registers during the function, we need to save the register values on the stack at the start of the function, then restore the values at the end of the function. For example:

```
# Prologue
addi sp, sp, -12    # decrement stack
sw ra, 0(sp)       # store ra value on the stack
sw s0, 4(sp)       # store s0 value on the stack
sw s1, 8(sp)       # store s1 value on the stack

# do stuff in the function

# Epilogue
lw ra, 0(sp)       # restore ra value from the stack
lw s0, 4(sp)       # restore s0 value from the stack
lw s1, 8(sp)       # restore s1 value from the stack
addi sp, sp, 12    # increment stack

# finish up any loose ends

ret              # return from function
```

Notice that we also saved the value of ra on the stack. Remember that the ra register stores the address that we'll jump back to after this function finishes executing. Saving the value of ra on the stack is necessary if we decide to call another function inside this function. If we make a function call at the "do stuff" comment, then that function call will overwrite ra, and we'll lose the address that we were supposed to jump back to.

Coding advice

When following calling convention, it's better to be safe than sorry! It doesn't hurt to save an extra register you didn't need to save, but it's very bad to forget to save a register that you were supposed to save. With, don't save every register because it takes up more stack space than necessary and also muddles your code, because eventually it's unclear as to what registers are being actively used and which ones are just chilling.

- Always save the value of ra at the start of a function and restore it at the end of a function. Even though it looks like it's being saved in the callee-prologue and restored in the callee-epilogue as a caller-saved register, it's actually preemptively saving ra in case at any point a function is called and guarantees the return address we want to return to is preserved, before doing anything else.
- Save the values of all the preserved registers at the start of a function and restore them at the end of the function in the prologue and epilogue. If you choose not to save the value in one of the save registers, be absolutely sure that you aren't using that register in the function.
- Save the values of the non-preserved registers that we rely on after a function is being called; in other words, only non-preserved registers values we rely on across a function call should be saved. Otherwise, they can be discarded. To do this, save them before calling a function and restore them after calling a function. You can do this by moving the temporary values by moving them to a free saved register, if any are available, or by saving them on the stack. If you choose not to save the value in one of the non-preserved registers, be absolutely sure that you do not rely on the value in that register after the function returns.

Debugging advice

Calling convention can be very hard to debug! If you don't follow calling convention, your code behavior is undefined; sometimes it works, and sometimes it won't.

To check that you're following calling conventions for preserved registers:

- **Identify every preserved register** you use in a function. For every preserved register you modify, make sure its value was saved at the start of the function and restored at the end of the function.
- Check your prologue and epilogue for typos. It's very easy to mistype a register or number!

To check that you're following calling convention for non-preserved registers:

- Find all the function calls in your code. For each function call, **identify every non-preserved register** you use after that function returns, and make sure its value was saved before the call and restored after the call.

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. 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
```

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

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.

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

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 integer
    addi a1, s0, 0 # argument to ecall, the value to be printed
    ecall # print integer ecall
    addi a0, x0, 10 # argument to ecall, 10 = terminate program
    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.

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

Q: Which callee saved registers do we need to save?

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 integer
    addi a1, s0, 0 # argument to ecall, the value to be printed
    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 program
ecall # terminate program
```

- You can find the entire program in **example_c_to_riscv.s**.

LAB TASK- 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 lab task, 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 like the errors below.

```
[CC Violation]: (PC=0x0000004C) Setting of a saved register (s0) which has not been saved! ex1.s:54
li s0, 1
[CC Violation]: (PC=0x00000054) Setting of a saved register (s0) which has not been saved! ex1.s:57
mul s0, s0, a0
[CC Violation]: (PC=0x00000054) Setting of a saved register (s0) which has not been saved! ex1.s:57
mul s0, s0, a0
[CC Violation]: (PC=0x00000054) Setting of a saved register (s0) which has not been saved! ex1.s:57
mul s0, s0, a0
[CC Violation]: (PC=0x00000054) Setting of a saved register (s0) which has not been saved! ex1.s:57
mul s0, s0, a0
[CC Violation]: (PC=0x00000054) Setting of a saved register (s0) which has not been saved! ex1.s:57
mul s0, s0, a0
[CC Violation]: (PC=0x00000054) Setting of a saved register (s0) which has not been saved! ex1.s:57
mul s0, s0, a0
```

```
[CC Violation]: (PC=0x00000054) Setting of a saved register (s0) which has not been saved! ex1.s:57 mul s0, s0, a0
[CC Violation]: (PC=0x00000064) Save register s0 not correctly restored before return! Expected 0x00000000, Actual 0x00000080. ex1.s:65 jr ra
[CC Violation]: (PC=0x00000070) Setting of a saved register (s0) which has not been saved! ex1.s:79 mv s0, a0 # Copy start of array to saved register
[CC Violation]: (PC=0x00000074) Setting of a saved register (s1) which has not been saved! ex1.s:80 mv s1, a1 # Copy length of array to saved register
[CC Violation]: (PC=0x000000A4) Setting of a saved register (s0) which has not been saved! ex1.s:117 addi s0, t1, 1
Found 12 warnings!
--------------------
[ERROR] An error has occurred!

Error:
`StoreError: You are attempting to edit the text of the program though the program is set to immutable at address 0x00000006!
```

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

Exercise 1

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.

Questions:

1. Is next_test a function?
2. What caused the errors in pow, and inc_arr that were reported by the Venus CC checker?
3. 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?
4. Why do we need to store ra in the prologue for inc_arr, but not in any other function?
5. Why wasn't the calling convention error in helper_fn reported by the CC checker? (Hint: it's mentioned above in the exercise instructions.)

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. 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. 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 program never stops (due to an infinite loop).
2. **Fix** the calling convention errors. The error is within the ex3 function.

- Submit the code along with the answers to the above questions in your submission report.

Reference

Lab 4 | CS 61C Fall 2023 (berkeley.edu)

Appendix: Calling Convention | CS 61C Fall 2023 (berkeley.edu)