Appendix: 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 have the ability to 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 overwr
mv s1, a1          # save return value 2, before moving on, in case a1 is used in the fut

# 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 to not change any of the preserved registers, from the perspective of the caller.

This is another common calling convention bug: **a function cannot noticably change any preserved registers**. In other words, change whatever you'd like, callee, just make sure you restore the preserved register values before returning back 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 that being said, 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.

Randomizing non-preserved registers

Remember that immediately after you call a function, you must assume that the non-preserved registers contain garbage. (This is because the function could have changed any of the non-preserved registers.)

Sometimes your code is violating calling convention by using the garbage values in non-preserved registers after calling a function. However, your code might just happen to work correctly because you got lucky with the garbage values.

One way to comprehensively check that your code follows calling convention is by putting totally random garbage values in non-preserved registers, and checking that your code still works correctly (doesn't rely on those garbage values).

In `utils.s`, we provided helper functions `randomizeCallerSavedRegs` and `randomizeCallerSavedRegsBesidesA0` that put random garbage values in all the non-preserved registers for you.

One example of how you might use this is by putting garbage in all the non-preserved registers after the function returns:

```
addi t0, x0, 5                      # t0 contains 5
jal ra, func                        # call a function
jal ra, randomizeCallerSavedRegs    # put garbage in non-preserved registers
# t0 will not equal 5 at this point anymore, even if 'func' didn't modify it.

# everything after this should still work!
```

If the function stores a return value in a0 and you don't want to put garbage in `a0`, you could use `randomizeCallerSavedRegsBesidesA0` there instead.

Another example of how you might use this is by putting garbage in all the non-preserved registers just before calling a function and setting its arguments:

```
addi t0, x0, 5                      # t0 contains 5
jal ra, randomizeCallerSavedRegs    # put garbage in non-preserved registers
# t0 will not equal 5 at this point anymore. 'func' shouldn't rely on t0 being equal to 5

# set arguments to the function using the a* registers here if needed

jal ra, func                        # this function call should still work!
```

The function should not rely on any values in the non-preserved registers, except for the argument registers relevant to the function itself. Therefore, your code should still work correctly even with garbage values in the non-preserved registers.

Randomizing preserved registers

We can also put random values in registers to check that preserved registers stay unchanged. At the start of your function, you can put a constant, well-known set of values in the preserved registers. At the end of your function, you can check that those values didn't change from your function.

In `utils.s`, we provided helper functions: `randomizeCalleeSavedRegs` puts random values in the preserved registers. `checkCalleeSavedRegs` checks that the random values are still in the preserved registers, and returns with error code 100 if any saved register is modified.

An example of how you might use this at the start and end of your function:

```
jal ra, randomizeCalleeSavedRegs     # put values in saved registers

# your function call

jal ra, checkCalleeSavedRegs         # check the saved registers values didn't change
```