Assignment 0 — COMPSYS

Schmidt, Victor Alexander, rqc908
Ibsen, Helga Rykov, mcv462
Barchager, Thomas Haulik, jxg170

Sunday 16:00, September 25th

1 Partitioning

1.1 How confident are you in the correctness of your implementation, and on what do you base this confidence? E.g. which tests have you performed?

Our implementation is correct. We have conducted a few tests, both with the original test in `test_partition.s` and with two additional test arrays. Our `partition` gets the first index, which is not less than the pivot, in almost all cases.

The only exception arises, when the pivot is the smallest element in the array. This is because `partition` returns 1, when it in reality should return 0 (the pivot's placement should be returned, as the first element which is not less than the pivot is the first element in the array). This, however, seems to be a problem with the given C example, which means our `partition` is correct even down to the example's faults.

1.2 Which registers do you use in your implementation, and for which purpose? Do you use any callee-save or caller-save registers, and why?

We have used caller-save registers a0 - a3 as function parameters and values to be used across while-loop calls. For saving the values within the if-statement we have used temporary caller-save registers t0 - t3: These registers were generated doing the while-loops, and saved to simplify the process which came after, inside of the if-statement.
We did not use any callee-save registers because the function is of a *leaf procedure*, i.e. a callee-type function that is later called by a caller function.

The reason we opted for caller-save registers was to avoid saving and loading on the stack/minimizing the amount of instructions, because callee-save registers have to be saved on the stack, whereas one can simply use caller-save registers since they can be freely overwritten.

1.3 Does your code access the stack? If yes, for what purpose?

Our code did not access the stack for this procedure.
To elaborate on why: We only used caller-save registers, which meant that as a leaf procedure, it was never necessary to save anything on the stack, as the registers we used would have been saved on the stack by the caller function. Further elaboration can be found in the previous question's answer.

2 Quick Sort

2.1 How confident are you in the correctness of your implementation, and on what do you base this confidence? E.g. which tests have you performed?

After having conducted the test provided to us and our own test, we can say with confidence that our implementation of `quicksort` is correct. We have already tested that `partition` works as intended, so all we needed to do was to make sure that our `quicksort` sorted our array correctly.

The provided test showed that the function was able to sort 1000 different numbers correctly. In our own test, we wanted to see what would happen if the array already had been sorted. In accord with our expectations, the function went through the array without changing it and returned a correctly sorted array.

2.2 Which registers do you use in your implementation, and for which purpose? Do you use any callee-save or caller-save registers, and why?

First of all, we use the caller-save registers a0-a2, which are our three arguments used throughout our implementation to call `partition` and `quicksort` itself recursively.

We have also used the caller-save registers t0-t1 to store values temporarily in the if-statement of `quicksort`. The callee-saved register ra was used for returning from the callee function to the caller function.

Lastly, we used the callee-saved registers sp to store four values, the return-address ra and our three arguments a0-a2 before calling other functions in order to be able to restore our values when returning from a callee function.

2.3 Does your code access the stack? If yes, for what purpose?

As mentioned above, we used the stack to store important values. The callee-save registers ra and caller-save a0-a2 were stored in the stack, so that we could easily restore them later on if needed.

We used the stack because we needed to save our arguments before calling `partition`. As intended, it would overwrite our array in a0 with its return value, which is why we cannot be sure that a1-a2 wouldn't also be overwritten.

In other words, we stored both callee-save and caller-save registers in the stack because the `quicksort` acts both as a caller and a callee due to being called recursively by itself.