A0: RISC-V Assembly Programming

Computer Systems 2022
Department of Computer Science
University of Copenhagen

Troels Henriksen

Due: Sunday, 25th of September, 16:00
Version 1 (September 12, 2022)

This is the first assignment in the course Computer Systems 2022 at DIKU. We encourage pair programming, so please form groups of 2 or 3 students. Groups cannot be larger than 3, and we strongly recommend that you do not work alone.

For this assignment you will receive a mark out of 4 points; You must attain at least half of the possible points to be admitted to the exam. For details, see the *Course description* in the course page. This assignment does not belongs to a category. Resubmission is not possible.

---

1 Introduction

The art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.

— Edsger Dijkstra, EWD249

This assignment is meant to assess your knowledge of the following:

1. Accessing byte-addressed memory.
2. Expressing control flow using jumps.
3. Stack discipline.
4. The RISC-V calling convention.

Keep this in mind when working on the code and when choosing what to focus on for your report. Make sure to read the entire text before starting.

In all procedures you must obey the RISC-V calling convention:

1. Do not overwrite callee-saves registers without restoring them before returning from the procedure.
2. Do not assume caller-saves registers are preserved across procedure calls.
3. When a procedure returns, make sure it has restored the stack pointer (`sp`) to the position it had when the procedure was called.

You must use RARS to develop and test your solution.

2 Partitioning

2.1 Implementing partitioning (25%)

For this subtask you must implement a procedure in RISC-V assembly that *partitioning* a segment of an array, equivalent to the following C function.

```
int partition(int *array, int p, int r) {
  int pivot = array[p];
  int i=p-1;
  int j=r;

  while (1) {
    do { j--; } while (pivot < array[j]);
    do { i++; } while (array[i] < pivot);
    if (i < j) {
      int tmp = array[i];
      array[i] = array[j];
      array[j] = tmp;
    } else {
      return j+1;
    }
  }
}
```

The `array` argument is the whole array, while `p` and `r` identify start index (inclusive) and end index (exclusive) of the segment to partition. Partitioning involves selecting a *pivot* (in the above, always the first element) and reordering the segment such that all elements *less* than the pivot occur before it.

The arguments `array`, `p`, and `r` will be provided in registers `a0`, `a1`, and `a2` respectively. The return value must be returned in `a0`.

You must provide your implementation in the file `partition.s`, which is part of the code handout.

2.1.1 Hints

- A C `do-while` loop always executes one iteration of the loop before checking the condition.
- This is a *leaf procedure*, meaning you can use caller-saves registers such as `t0-t6` without worry.
- You are given a test program `test_partition.s` for testing partitioning. This program defines a small constant array in memory and passes it to the `partition` routine from `partition.s`. You should run this program in RARS to determine whether it correctly partitions the array. You can change the contents of the array. You can also change the size of the array, but if so make sure to change the size that is passed to `partition` as well. You can also change `test_partition.s` to accept input automatically.

2.2 Report (25%)

Your report must concisely answer the following questions:

- 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?
- Which registers do you use in your implementation, and for which purpose? Do you use any callee-save or caller-save registers, and why?
- Does your code access the stack? If yes, for what purpose?

3 Quicksort

3.1 Implementing Quicksort (25%)

For this subtask you must implement a procedure in RISC-V assembly for quicksort, equivalent to the following recursive C function.

```
void quicksort(int *array, int start, int end) {
  if (end-start < 2) {
    return;
  }

  int q = partition(a, start, end);
  quicksort(a, start, q);
  quicksort(a, q, end);
}
```

Reuse your definition of partion from the previous task.

The arguments `array`, `start`, and `and` will be provided in registers `a0`, `a1`, and `a2` respectively. There is no return value.

You should provide your implementation in the file `quicksort.s`, which is part of the code handout.

3.1.1 Hints

- When a recursive function returns, be careful to restore the stack and any callee-saves registers.
- The included `run_quicksort.s` file can be used to pass data from the outside world to your implementation of quicksort. It starts by reading a single number x from the console, after which it reads an array of x words from the console and stores it in memory at the `array` label. There is only space reserved for 1024 words, but this can be increased if desired. You can also write a testing program similar to `test_partition.s` if you wish.
- See section 3.3 below.

3.2 Report (25%)

Your report must concisely answer the following questions:

- 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?
- Which registers do you use in your implementation, and for which purpose? Do you use any callee-save or caller-save registers, and why?
- Does your code access the stack? If yes, for what purpose?

3.3 Testing script

We have included a script `test_quicksort.sh` for easily testing your implementation for correctness. It is run as follows:

```
$ sh test_quicksort.sh run_quicksort.s 1000_numbers
```

You will need to first modify `test_quicksort.sh` so that it can find `rars.jar`. See comments in the file.

This will run your RISC-V quicksort implementation on the given datafile[1], which should *not* include a line count. If the result is properly sorted, the script will report how many instructions your program spent. If the result is *not* properly sorted, the script will report an error, but will not provide further details. You should debug in RARS itself. The testing script is useful for detecting errors, but not for correcting them.

Optional: Gotta go fast! Once you get your implementation working, see how well you can optimise it. On `1000_numbers`, our non-optimised reference implementation requires 161523 instructions, while Troels' fastest implementation requires 108895 instructions. Surely a student can beat that!

4 Handout

Apart from this text, the handout consists of a directory `src` containing the following files:

`partition.s`: Template for implementing `partition`.

`test_partition.s`: Template for testing `partition`.

`quicksort.s`: Template for implementing `quicksort`.

`run_quicksort.s`: Template for testing `quicksort`.

`1000_numbers`: A file with one thousand lines, each containing an integer.

`io.s`: Helper procedures for reading and writing integers.

`test_quicksort.sh`: See section 3.3.

`Makefile`: : a Makefile that lets you run `make ../src.zip` to produce a zip archive with all relevant files.

[1] `1000_numbers` is included in the handout, but you can generate your own.

5 Handin

You should hand in a ZIP archive containing a `src` directory containing all *relevant* files (no ZIP bomb, no compiled objects, no auxiliary editor files, etc.).

Alongside a `src.zip` submit a `report.pdf`, and a `group.txt`. `group.txt` must list the KU-ids of your group members, one per line, and do so using *only* characters from the following set written using *standard alphabetical encoding*:

$$\{\texttt{0x0A}\} \cup \{\texttt{0x30}, \texttt{0x31}, \ldots, \texttt{0x39}\} \cup \{\texttt{0x61}, \texttt{0x62}, \ldots, \texttt{0x7A}\}$$

Please make sure that the file is UTF-8 encoded with UNIX line endings.