This project has 3 parts:

- Write 3 sort routines (quicksort, bubble sort, merge sort) in RISC-V assembly, and use them to sort a random 4 kB file of 32-bit unsigned integers
  - These unsigned integers will never have their high bit set (bit 31 will always be 0).
- Analyze the instruction trace from your runs
- Using the memory trace from your runs, design, implement, and analyze a 1 kB data cache that would accelerate running your three sort routines

Every place in this assignment that we use the term "kB", we mean "kibibytes" (a power of two) not "kilobytes" (a power of ten). 4 kB, for instance, is 4096 bytes.

## WHAT YOU CAN START DOING IMMEDIATELY

- Write your three sort routines
- Write the code that analyzes the instruction memory trace
- Write the code that simulates your cache



## WRITE THREE SORT ROUTINES

You will write three sort functions, each of which will take three arguments:



- The first argument is the memory address of where you will write your 4 kB output (the sorted list). You are allowed to change the contents of this memory region if you would like.
- The second argument is the memory address of the 4 kB unsorted input. You can assume the input is exactly 4 kB (1024 32-bit unsigned integers).
- The third argument is the memory address of a 16 kB uninitialized memory region if you find that useful in writing your routines

Use these scaffolds, main.asm and file_read.asm, the data-set data.txt.

What does *main.asm* do?

- Calls *file_read.asm* and sort routines.
- Saves required environment(registers) and clears all registers except for those used in function calls. This avoids unexpected results.
- Provides necessary addresses for the sort routines.

What does *file_read.asm* do?

- Reads the data-set *data.txt* into a buffer.
- Since the data can only be read as strings, it has a routine which coverts them to unsigned integers.
- Returns start and end address of unsorted data.

How to use these scaffolds?

- Open *main.asm*, *file_read.asm* and your sort routines into RARS.
- If you don't want to call a routine, comment the respective call macro in *main.asm*. for example, if you don't want to assemble merge and bubble sort, just comment *func_call(merge)* and *func_call(bubble)* in *main.asm*.
- Do not use the label 'main' in your sort routines.
- Before you assemble, go to setting select '*Assemble all files currently open*' and '*Initialize program counter to global 'main' if defined*'.
- The data-set '*data.txt*' should be in the same directory as '*rars.jar*'.
- Follow RISCV calling conventions, save the return address at the beginning of your sort routine, and retrieve it back at the end.



The three algorithms you will implement are:

- https://en.wikipedia.org/wiki/Bubble_sort (Links to an external site.)
- https://en.wikipedia.org/wiki/Merge_sort (Links to an external site.)
- https://en.wikipedia.org/wiki/Quicksort (Links to an external site.) (Links to an external site.)





Submit your sort routines as:

- Bubble sort : filename (bubble.asm), function name (bubble).
- Merge sort : filename (merge.asm), function name (merge).
- Quick sort : filename (quick.asm), function name (quick).

## GENERATE AN INSTRUCTION/MEMORY TRACE

Your instructor wrote this in Java. Source code is in https://github.com/jowens/rars (Links to an external site.); you can download a compiled jar from here or (on ECE work stations) at /software/classtools/EEC170/rars.jar. You are welcome to compile it yourself. When you run a program in (this version of) RARS, you can generate a trace by doing the following:

1. Open your source file in RARS.
2. Tools menu, Instruction/Memory Dump.
3. Change filename to a filename of your choice.
4. Click button: Connect to Program

5. Run, Assemble.
6. Run, Go.
7. Go back to Instruction/Memory Dump window: click "Dump Log". This saves the dump to the file you specified in step 3.

These steps are pretty brittle (i.e., do them in this exact order) because your instructor doesn't know how to use Swing very well. If you have suggestions, the instructional staff is happy to hear them.

The file you generate has one line per datum. The four kinds of data you will see in the trace are:

- 'I": The address of an access into instruction memory
- 'i': A 32-bit RISC-V instruction (the trace first dumps the address then the instruction)
- 'L': The address of a memory load into data memory
- 'S': The address of a memory store into data memory (we don't think you need the contents of the memory load/store for this project, so they aren't in the trace)



Your instructor put a C++ file that you can use as a scaffold for your analysis, as well as a sample log file, in the Project 2 directory in Canvas.



## ANALYZE THE INSTRUCTION TRACE



For this part you only need to look at the 'I' and 'i' entries in your trace. In your report, answer the following questions. It seems easiest to the staff that you report separately for each of your three programs but it's acceptable if you combine all three together.

- Compare the performance, in terms of instructions executed, of the three sorts.

- What is the instruction mix that you actually run over your three programs: what is the fraction of instructions that are loads, stores, branches, arithmetic operations, etc.? Use your judgment on how best to report this mix.
- What fraction of memory loads incur a load-use hazard?
- Given our 5-stage RISC-V pipeline, what fraction of instructions executed have at least one register argument that must be forwarded?
- How effective is static branch prediction (what percentage of the time will a static branch prediction algorithm—forward branches are not taken, backward branches are taken—make the right prediction)?

## DESIGN A CACHE

You will design a 1 kB data cache, implement it (in C), and analyze its performance using the traces you produce. The most straightforward way for you to do this is to allocate a data structure in your trace analyzer that has the exact same data as the pictures of cache that we showed in class (for example, slide 11 of the lecture memory-2.pdf, “Example: Intrinsity FastMath”; for this example, you will have to keep track of a valid bit and a tag for each entry in the cache; you do NOT have to keep track of data because we don’t need to do so for this assignment, and you should think through why we say this in the assignment so you understand). You are only focusing on data accesses (load/store), not on instruction accesses.

In your simulator, every time you see a data memory access, you will check the cache to see if it is a hit or miss, record whatever data you want to record for your analysis, and update the cache. The size of your cache is fixed (1 kB). You will certainly want to explore varying the block size and associativity of your cache. You may also want to vary replacement policy and write policy (write-through vs. write-back).

Your goal is to maximize system performance. In your report, quantify how much your cache helps system performance vs. having no cache. Assume that your workload is equally split between the three sort routines (you will call each sort routine the same number of times in your workload). Maximizing system performance will primarily mean maximizing hit rate. Justify your design and your conclusions in your report succinctly but thoroughly. To measure system performance, assume the following:



- Cache hits complete in a single cycle.
- Cache misses cost an additional 50 cycles (to go to memory).
- If you implement an associative cache, it affects the clock cycle of the machine. Any set associative cache has a 10% clock cycle penalty when compared to a direct-mapped cache, and each additional doubling of set associativity adds an additional 2%. So a 2-way s-a cache has a 10% penalty, a 4-way s-a cache has a 12% penalty, 8-way is 14%, 16-way is 16%, etc.

## WHAT YOU SUBMIT

- A PDF writeup (technical report) of no more than 2 pages that summarizes your findings. MUST BE PDF. DO NOT USE ANOTHER FILE FORMAT.
- Your source code for your three sort routines