ECE 5745 Complex Digital ASIC Design, Spring 2020

Lab 2: Sorting Accelerator

School of Electrical and Computer Engineering
Cornell University

revision: 2020-02-28-00-13

In this lab, you will explore a medium-grain hardware accelerator for sorting an array of integer values of unknown length. The baseline design is a pure-software sorting microbenchmark running on a pipelined processor with its own instruction and data cache. The alternative design augments the baseline design with a hardware accelerator and includes the necessary software to configure and potentially assist the accelerator. You will use a standard-cell-based ASIC toolflow to quantitatively analyze the area, energy, and performance of both the baseline and alternative designs. You are required to implement the alternative design, verify the design using an effective testing strategy, push all designs through the ASIC toolflow, and perform an evaluation comparing the various implementations. You are welcome to implement a CL model of your accelerator if you think this will accelerate your design-space exploration, but this is not required. **As with all lab assignments, the majority of your grade will be determined by the lab report. You should consult the course lab assignment assessment rubric for more information about the expectations for all lab assignments and how they will be assessed.**

This lab is designed to give you experience with:

- application-specific accelerator design;
- software/hardware co-design;
- state-of-the-art standard-cell-based ASIC toolflow for quantitatively analyzing the area, energy, and timing of a design.

This handout assumes that you have read and understand the course tutorials and the lab assessment rubric. To get started, login to an `ecelinux` machine, source the setup script, and clone your lab group's remote repository from GitHub:

```
% source setup-ece5745.sh
% mkdir -p ${HOME}/ece5745
% cd ${HOME}/ece5745
% git clone git@github.com:cornell-ece5745/lab2-groupXX.git
```

where `XX` is your group number. **You should never fork your lab group's remote repository! If you need to work in isolation then use a branch within your lab group's remote repository.** If you have already cloned your lab group's remote repository, then use `git pull` to ensure you have any recent updates before running all of the tests. You can run all of the tests in the lab like this:

```
% cd ${HOME}/ece5745/lab2-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% py.test ../lab2_xcel
```

All of the tests should pass except for the tests related to the sorting accelerator you will be implementing in this lab. For this lab, you will be making use of the following subprojects:

• `lab1_imul`	– Staff solution to lab 1, used in `proc`
• `lab2_xcel`	– Your solution to lab 2
• `sram`	– SRAMs used in `cache` and potentially your accelerator
• `proc`	– Pipelined, five-stage, TinyRV2 processor
• `cache`	– Blocking, two-way, set-associative cache
• `pmx`	– Processor, cache, accelerator composition

You will be mostly be working in the `lab2_xcel` subproject which includes the following files:

• `SortXcelFL.py`	– FL sorting accelerator
• `SortXcelCL.py`	– CL model of sorting accelerator (optional)
• `SortXcelPRTL.py`	– PyMTL RTL model of sorting accelerator
• `SortXcelVRTL.v`	– Verilog RTL model of sorting accelerator
• `SortXcelRTL.py`	– Wrapper to choose which RTL language
• `sort-xcel-sim`	– Accelerator simulator for isolated eval
• `__init__.py`	– Package setup
• `test/SortXcelFL_test.py`	– FL sorting accelerator unit tests
• `test/SortXcelCL_test.py`	– CL sorting accelerator unit tests
• `test/SortXcelRTL_test.py`	– RTL sorting accelerator unit tests
• `test/__init__.py`	– Test package setup

If you do not plan to implement a CL model, then you should remove the corresponding file.

1. Introduction

In ECE 4750, you gained experience designing, implementing, testing, and evaluating general-purpose processors, memories, and networks. In this lab, you will gain experience with a similar process for an application-specific medium-grain accelerator. Fine-grain accelerators are tightly integrated within the processor pipeline (e.g., a specialized functional unit for bit-reversed addressing useful in implementing an FFT), while coarse-grain accelerators are loosely integrated with a processor through the memory hierarchy (e.g., a graphics rendering accelerator sharing the last-level cache with a general-purpose processor). Medium-grain accelerators are often integrated as co-processors: the processor can directly send/receive messages to/from the accelerator with special instructions, but the co-processor is relatively decoupled from the main processor pipeline and can also independently interact with memory.

We will be accelerating a simple sorting application. The application is given a source array, a destination array, and a size. The application should sort the input array of unsigned integers in increasing numerical order and place the result in the given destination array. The application is allowed to modify the source array. The application must be able to handle both very small arrays (e.g., four elements) and large arrays (e.g., thousands of elements). You can assume the data in the source array will be uniformly distributed 32-bit random integers.

2. Baseline Design

The baseline design for this lab assignment is a pure-software sorting microbenchmark running on a pipelined processor with its own instruction and data cache. Figure 1 illustrates the overall system we will be using in this lab assignment. The processor includes eight latency insensitive val/rdy

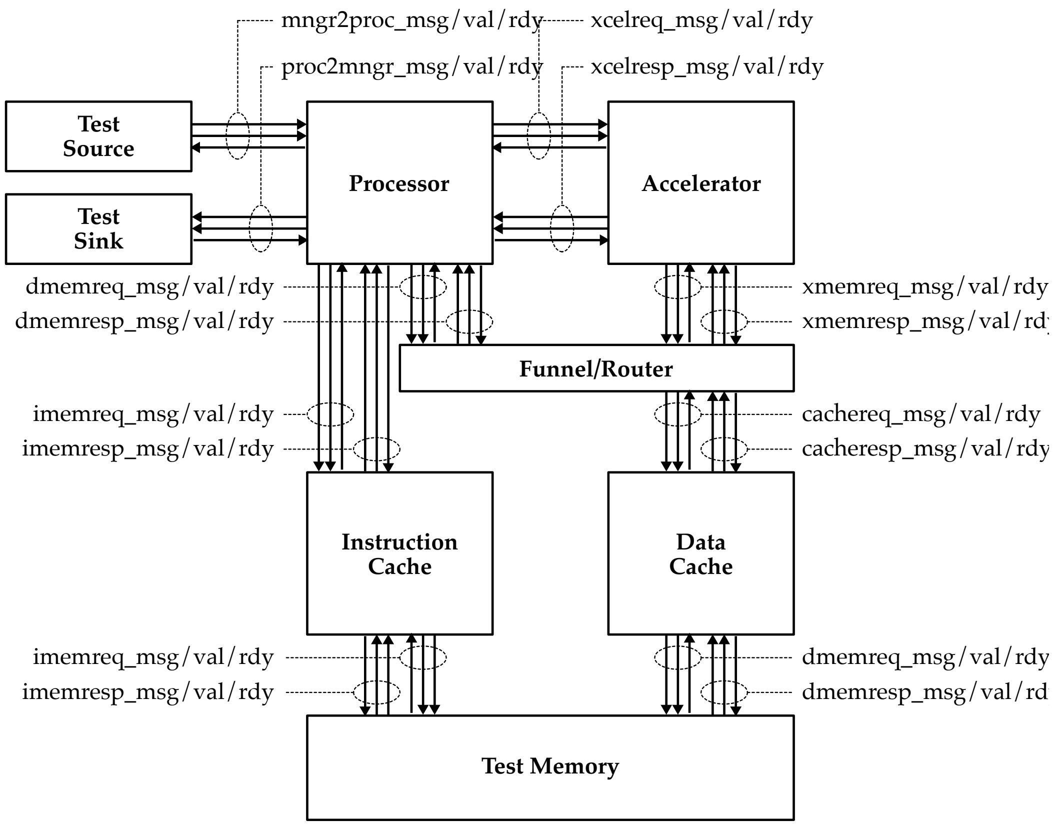

Figure 1: Processor, Cache, Accelerator Composition

interfaces. The mngr2proc/proc2mngr interfaces are used for the test harness to send data to the processor and for the processor to send data back to the test harness. The imemreq/imemresp interfaces are used for instruction fetch, and the dmemreq/dmemresp interfaces are used for implementing load/store instructions. The system includes both instruction and data caches. The xcelreq/xcelresp interfaces are used for the processor to send messages to the accelerator. The mngr2proc/proc2mngr and memreq/memresp interfaces were all introduced in ECE 4750. For the baseline design, we provide a simple null accelerator which you can largely ignore.

We provide two implementations of the TinyRV2 processor. The FL model in `sim/proc/ProcFL.py` is essentially an instruction-set-architecture (ISA) simulator; it simulates only the instruction semantics and makes no attempt to model any timing behavior. The RTL model in `sim/proc/ProcPRTL.py` is similar to the alternative design for lab 2 in ECE 4750. It is a five-stage pipelined processor that implements the TinyRV2 instruction set and includes full bypassing/forwarding to resolve data hazards. There are two important differences from the alternative design for lab 2 of ECE 4750. First, the new processor design uses a single-cycle integer multiplier. We can push the design through the flow and verify that the single-cycle integer multiplier does not adversely impact the overall processor cycle time. Second, the new processor design includes the ability to handle new CSRs for interacting with medium-grain accelerators. The datapath diagram for the processor is shown in Figure 2.

We provide an RTL model in `sim/cache/BlockingCachePRTL.py` which is very similar to the alternative design for lab 3 of ECE 4750. It is a two-way set-associative cache with 16B cache lines and a write-back/write-allocate write policy and LRU replacement policy. There are three impor-

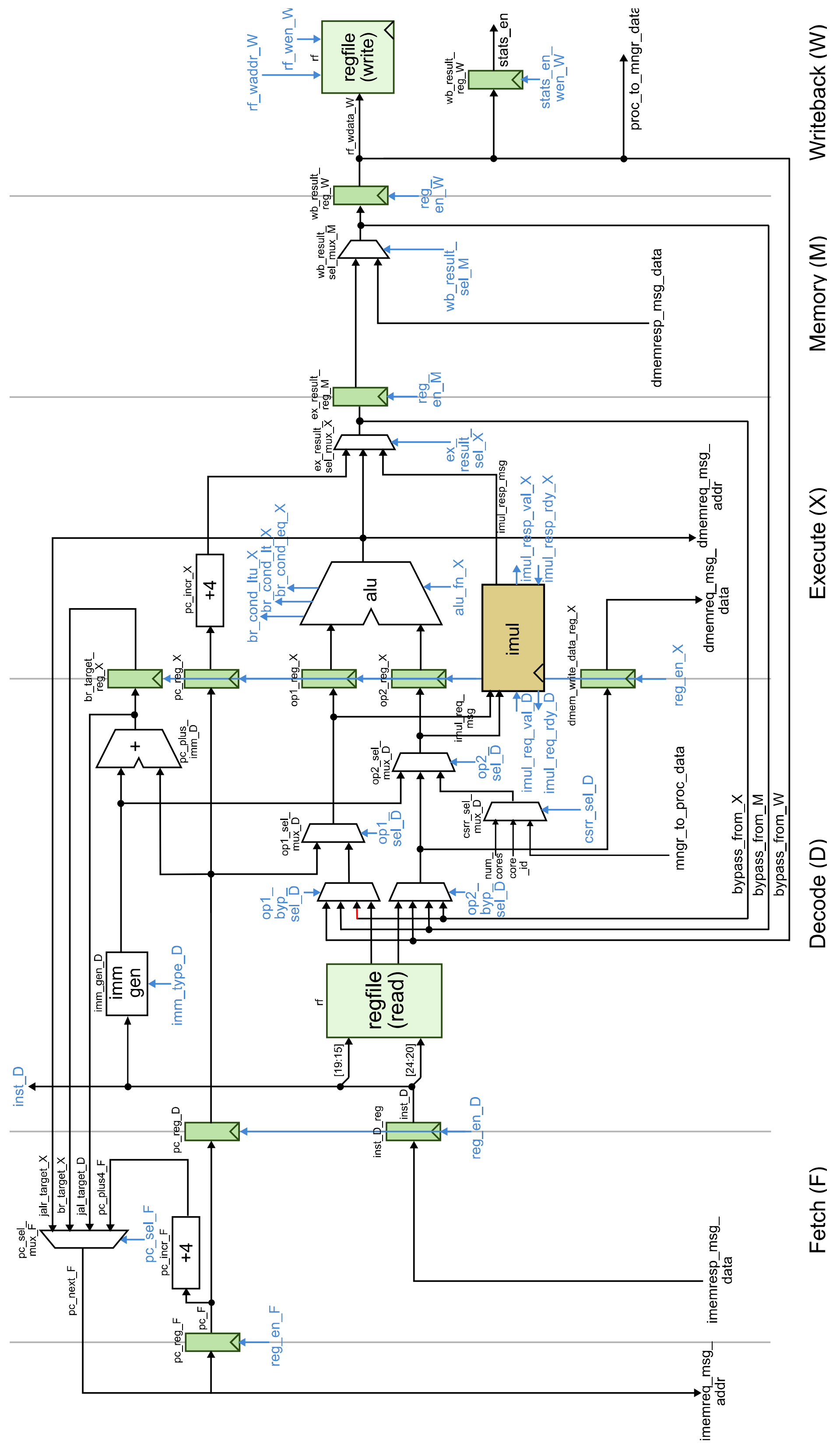

Figure 2: Baseline Processor Datapath

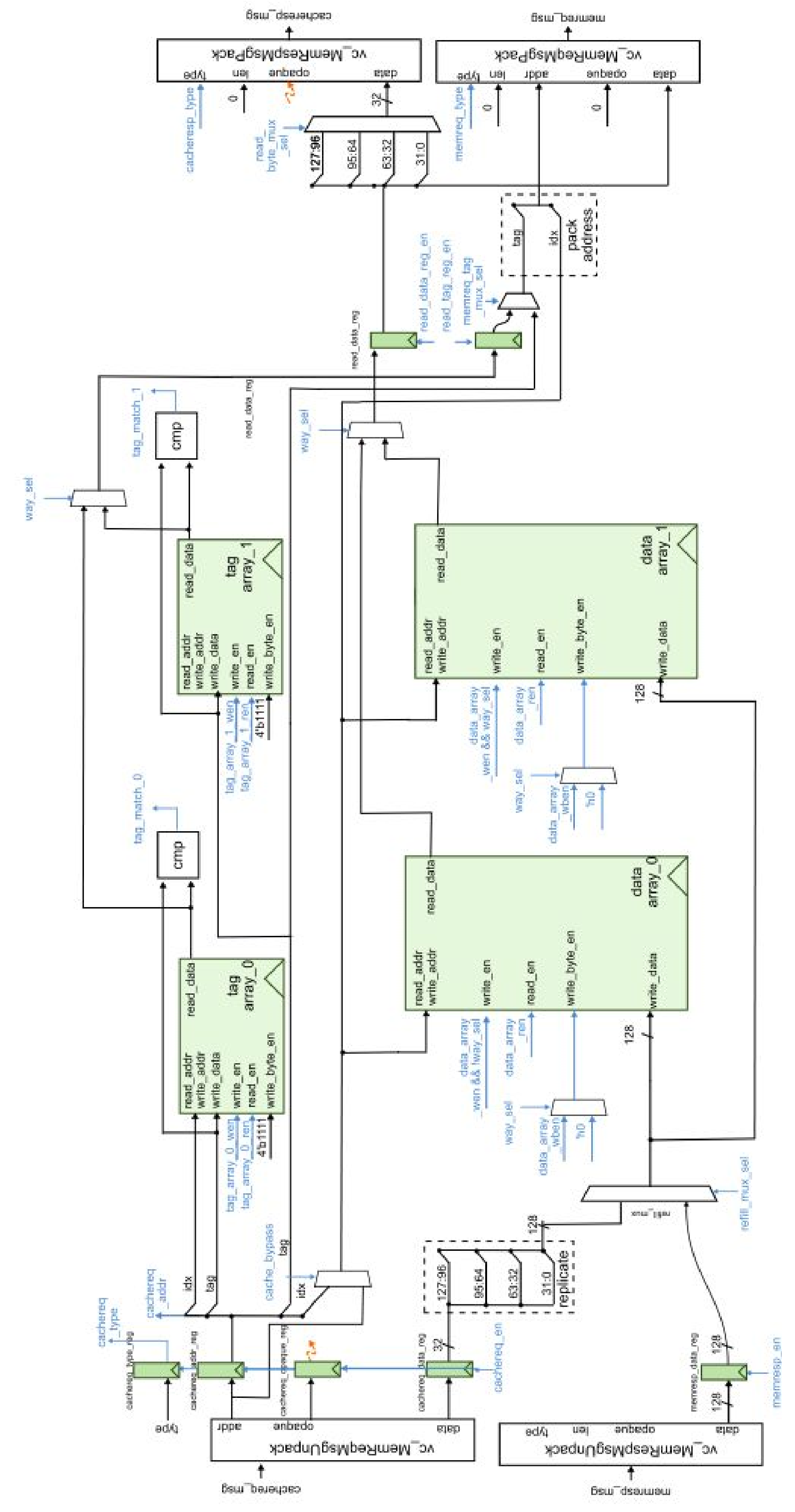

Figure 3: Baseline Cache Datapath

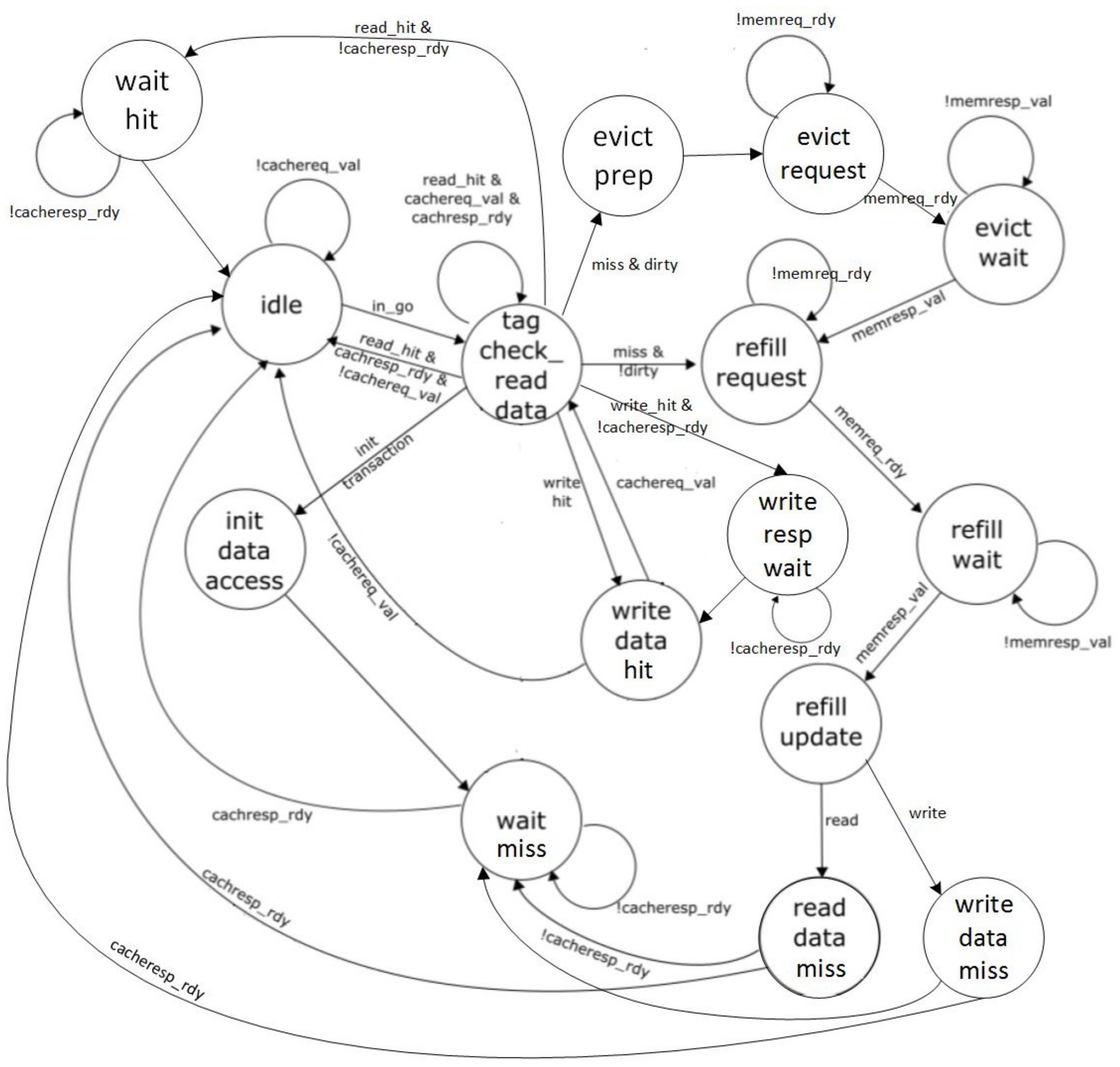

Figure 4: Baseline Cache FSM

tant differences from the alternative design for lab 3 of ECE 4750. First, the new cache design is larger with a total capacity of 8 KB. Second, the new cache design carefully merges states to enable a single-cycle hit latency for both reads and writes. Note that writes have a two cycle occupancy (i.e., back-to-back writes will only be able to be serviced at half throughput). Third, the previous cache design used combinational-read SRAMs, while the new cache design uses synchronous-read SRAMs. Combinational-read SRAMs mean the read data is valid on the same cycle we set the read address. Synchronous-read SRAMs mean the read data is valid on the cycle *after* we set the read address. Combinational SRAMs simplify the design, but are not realistic. Almost all real SRAM memory generators used in ASIC toolflows produce synchronous-read SRAMs, and indeed the CACTI memory compiler discussed in the previous tutorial also produces synchronous-read SRAMs. Using synchronous-read SRAMs requires non-trivial changes to both the datapath and the control logic. The cache FSM must make sure all control signals for the SRAM are ready the cycle before we need the read data. The datapath and FSM diagrams for the new cache are shown in Figures 3 and 4.

Notice in the FSM how we are able to stay in the `TAG_CHECK_READ_DATA` state if another request is ready; this is what produces the single-cycle hit latency.

Finally, we provide a reasonably optimized pure-software sorting microbenchmark which uses quicksort in `app/ubmark/ubmark-sort.c`. Note that we also provide more optimized versions of quicksort in (i.e., `-v2`, `-v3`). However, `-v1` is the one currently used in `ubmark-sort.c`, and this is the one you should use as your baseline design.

3. Alternative Design

The alternative design is to implement a medium-grain sorting accelerator and to develop a corresponding software microbenchmark that takes advantage of this new accelerator. This lab is completely open-ended. There are many ways to design such an accelerator. You can design an simple FSM-based accelerator that iteratively loads one or two values from memory does a comparison and writes one or two values back to memory algorithm to implement bubble sort, selection sort, merge sort, quicksort, or some other comparison sort. You might consider a non-comparison sort such as radix sort. You can also implement a more sophisticated accelerator that streams a block of elements into a temporary buffer in the accelerator, sort these values inside the accelerator, and then streams this block of elements back out to the memory system. You would need to merge the sorted blocks in either software or hardware. You can implement multiple "sub-accelerators" which map to different accelerator registers. You are free to use SRAMs.

We provide you an FL model of a sorting accelerator which uses the following accelerator registers:

- `xr0`: go/done
- `xr1`: base address of array
- `xr2`: number of elements in array

Using the accelerator protocol involves the following steps:

- 1. Write the base address of array via `xr1`
- 2. Write the number of elements in array via `xr2`
- 3. Tell accelerator to go by writing `xr0`
- 4. Wait for accelerator to finish by reading `xr0`, result will be 1

You can see an example of how to use this accelerator from software in `app/ubmark/ubmark-sort-xcel.c`. You are free to modify this software. You are free to use a completely different accelerator protocol (i.e., the number and semantics of the accelerator registers). If you modify the accelerator protocol you will also need to modify the FL model of the accelerator. You are free to implement part of your sorting algorithm in software and part of your sorting algorithm in hardware. You are free to overlap work on the processor with work in the accelerator.

The only restrictions are that you cannot change the processor or cache implementation. This means your accelerator is restricted to a peak memory bandwidth of one 4B memory request/response per cycle and you must use the provided accelerator message interface; you can change the accelerator protocol but you must use the basic xcelreq/xcelresp interface. We also ask you not to flatten your design, nor make significant modifications to the scripts used in the automated ASIC flow. We ask you to keep the clock constraint at a specific value which will be provided by the instructor. If your accelerator significantly impacts the cycle time compared to the baseline design, then you should optimize your accelerator so that it is no longer on the critical path.

Recall that the sorting application must be able to handle both very small arrays (e.g., four elements) and large arrays (e.g., thousands of elements). It is fine for your accelerator to only handle a fixed array size, but you will need to ensure that your software can use this accelerator to sort an arbitrary array size.

You should not feel limited to implementing only one alternative design. Feel free to implement multiple alternative designs, or to experiment with different parameters in order to lay the foundation for a rich and compelling design-space exploration in your lab report.

4. Testing Strategy

We have provided you with some basic unit tests that test the sorting accelerator in isolation. These tests are defined in `sim/lab2_xcel/test/SortXcelFL_test.py` and they are reused in the test scripts for the CL and RTL models. You will need to modify these tests if you change the sorting accelerator protocol. For example, if your sorting accelerator can only sort a fixed array size, you will need to modify the unit tests appropriately. You will almost certainly want to add more tests for specific corner cases related to your sorting accelerator implementation. The following commands illustrate how to run all of the tests for the entire project, how to run just the tests for this lab, and how to run just the tests for the FL and RTL models.

```
% cd ${HOME}/ece5745/lab2-groupXX/sim/build
% py.test ..
% py.test ../lab2_xcel
% py.test ../lab2_xcel/test/SortXcelFL_test.py  --verbose
% py.test ../lab2_xcel/test/SortXcelRTL_test.py --verbose
```

Once your alternative design passes all of the provided tests, then you should verify that your RTL model can be successfully translated using the following command:

```
% cd ${HOME}/ece5745/lab2-groupXX/sim/build
% py.test ../lab2_xcel/test/SortXcelRTL_test.py --test-verilog
```

5. Evaluation

Once you have verified the functionality of your alternative design, you should then use the provided simulator to evaluate the cycle-level performance of both the baseline and alternative designs. You can build the microbenchmarks for the baseline and alternative designs like this:

```
% cd ${HOME}/ece5745/lab2-groupXX/app
% mkdir build
% cd build
% ../configure --host=riscv32-unknown-elf
% make ubmark-sort
% make ubmark-sort-xcel
```

You can run the simulator for the FL and RTL models of the baseline design like this:

```
% cd ${HOME}/ece5745/lab2-groupXX/sim/build
% ../pmx/pmx-sim ../../app/build/ubmark-sort
% ../pmx/pmx-sim --proc-impl rtl --cache-impl rtl \
    --stats ../../app/build/ubmark-sort
```

The next step is to test your sorting microbenchmark on an FL model of the processor and accelerator. This step verifies that your microbenchmark correctly uses the sorting accelerator protocol.

```
% cd ${HOME}/ece5745/lab2-groupXX/sim/build
% ../pmx/pmx-sim --xcel-impl sort-fl ../../app/build/ubmark-sort-xcel
```

Only after this works, should you try running your sorting microbencmark on the RTL models. You might want to try running your sorting microbenchmark with and without the cache.

```
% cd ${HOME}/ece5745/lab2-groupXX/sim/build
% ../pmx/pmx-sim --proc-impl rtl --xcel-impl sort-rtl \
    ../../app/build/ubmark-sort-xcel
% ../pmx/pmx-sim --proc-impl rtl --cache-impl rtl --xcel-impl sort-rtl \
    ../../app/build/ubmark-sort-xcel
```

Since we do not have any "real" integration tests of the processor, cache, and accelerator composition, you might want to consider running your sorting microbenchmark on the FL and RTL models as part of your testing strategy. Once the microbenchmark passes, we can use the `--stats` option to determine the total number of cycles to execute the specified input dataset.

```
% cd ${HOME}/ece5745/lab2-groupXX/sim/build
% ../pmx/pmx-sim --proc-impl rtl --cache-impl rtl --xcel-impl sort-rtl \
    --stats ../../app/build/ubmark-sort-xcel
```

You should study the line traces (with the `--trace` option) and possibly the waveforms (with the `--dump-vcd` option) to understand the reason why each design performs as it does on the various patterns.

Once you have explored the cycle-level performance of the baseline and alternative designs, you should push the RTL models through the standard-cell-based ASIC toolflow to quantify the area, energy, and timing of each design. You can generate the RTL (and VCD file for energy analysis) for both the baseline and alternative design like this:

```
% cd $TOPDIR/sim/build
% ../pmx/pmx-sim --proc-impl rtl --cache-impl rtl \
    --translate --dump-vcd ../../app/build/ubmark-sort
% ../pmx/pmx-sim --proc-impl rtl --cache-impl rtl --xcel-impl sort-rtl \
    --translate --dump-vcd ../../app/build/ubmark-sort-xcel
```

We will update this with more info on using the automated flow soon!

6. Pareto-Optimal Frontier Competition

We hope students will continue to modify their software and/or hardware to improve the performance of their sorting accelerator. To encourage such optimization, a small bonus will be given to those designs which lay on the pareto-optimal frontier in the area vs. performance space. We will run your sorting microbenchmark on your processor, cache, and accelerator composition using one or more of our own private datasets. These datasets will not be the same size as the dataset we give you. They may be smaller or they may be larger. Note that students should only attempt improving the software and/or hardware once everything is completely working and they have finished a draft of the lab report. We allow students to submit up to two different accelerator designs for the purposes of evaluating the pareto-optimal frontier.

More details on the exact commands we will be using for the pareto-optimal frontier competition will be included soon!

Acknowledgments

This lab was created by Christopher Batten, Khalid Al-Hawaj, Jason Setter, Shunning Jiang, and Moyang Wang as part of the course ECE 5745 Complex Digital ASIC Design at Cornell University.