ECE 6745 Complex Digital ASIC Design, Spring 2025

Lab 2: ASIC Sorting Accelerator

School of Electrical and Computer Engineering
Cornell University

revision: 2025-03-19-22-18

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 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. This lab is designed to give you experience with: (1) application-specific accelerator design; (2) software/hardware co-design; (3) state-of-the-art standard-cell 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 assignment logistics document. To get started, login to an `ecelinux` machine, source the setup script, and clone your lab group's remote repository from GitHub:

```
% source setup-ece6745.sh
% mkdir -p ${HOME}/ece6745
% cd ${HOME}/ece6745
% git clone git@github.com:cornell-ece6745/lab2-groupXX.git
% TOPDIR=$PWD
```

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}/ece6745/lab2-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% pytest ../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
- `SortXcel.v` – Verilog RTL model of sorting accelerator
- `SortXcel.py` – Pyton wrapper for sorting accelerator
- `sort-xcel-sim` – Accelerator simulator for isolated eval
- `__init__.py` – Package setup
- `test/SortXcelFL_test.py` – FL sorting accelerator unit tests
- `test/SortXcel_test.py` – RTL sorting accelerator unit tests
- `test/__init__.py` – Test package setup

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 an array and a size. The application should sort the input array of unsigned integers in increasing numerical order. 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 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/Proc.v` 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

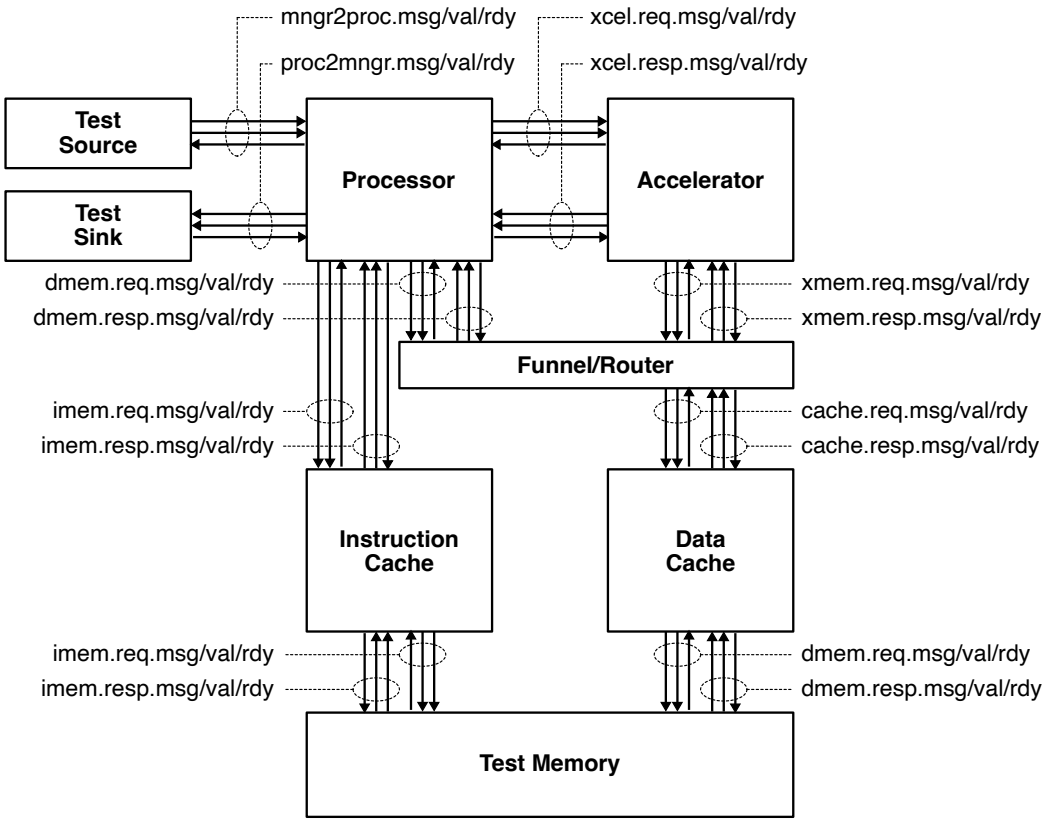

Figure 1: Processor, Cache, Accelerator Composition

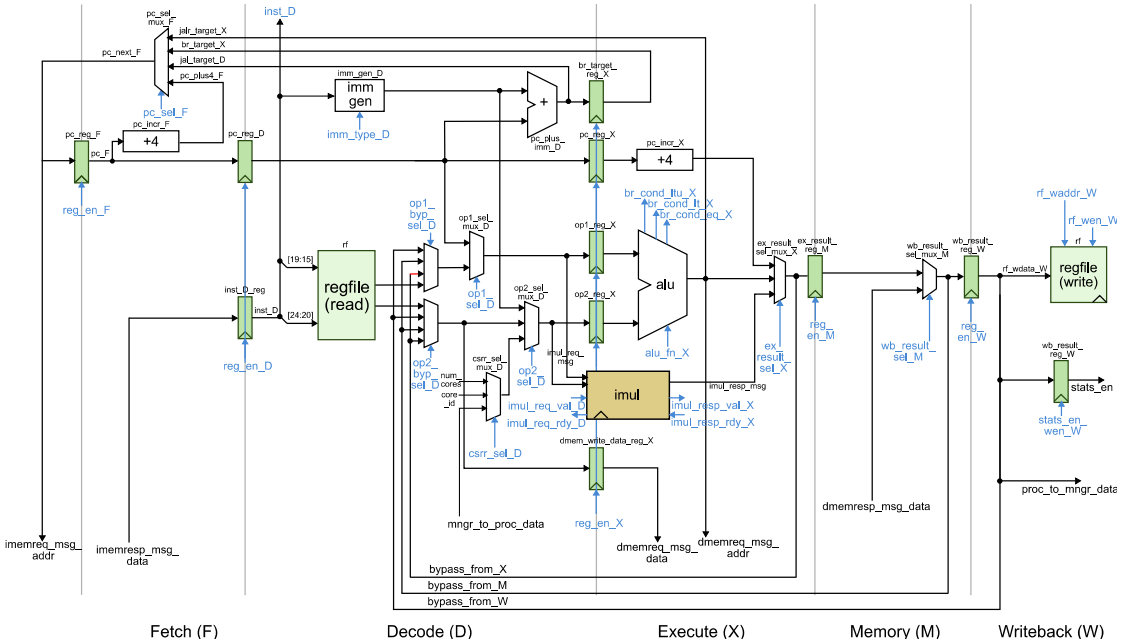

Figure 2: Baseline Processor Datapath

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/Cache.v` 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 important 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`.

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

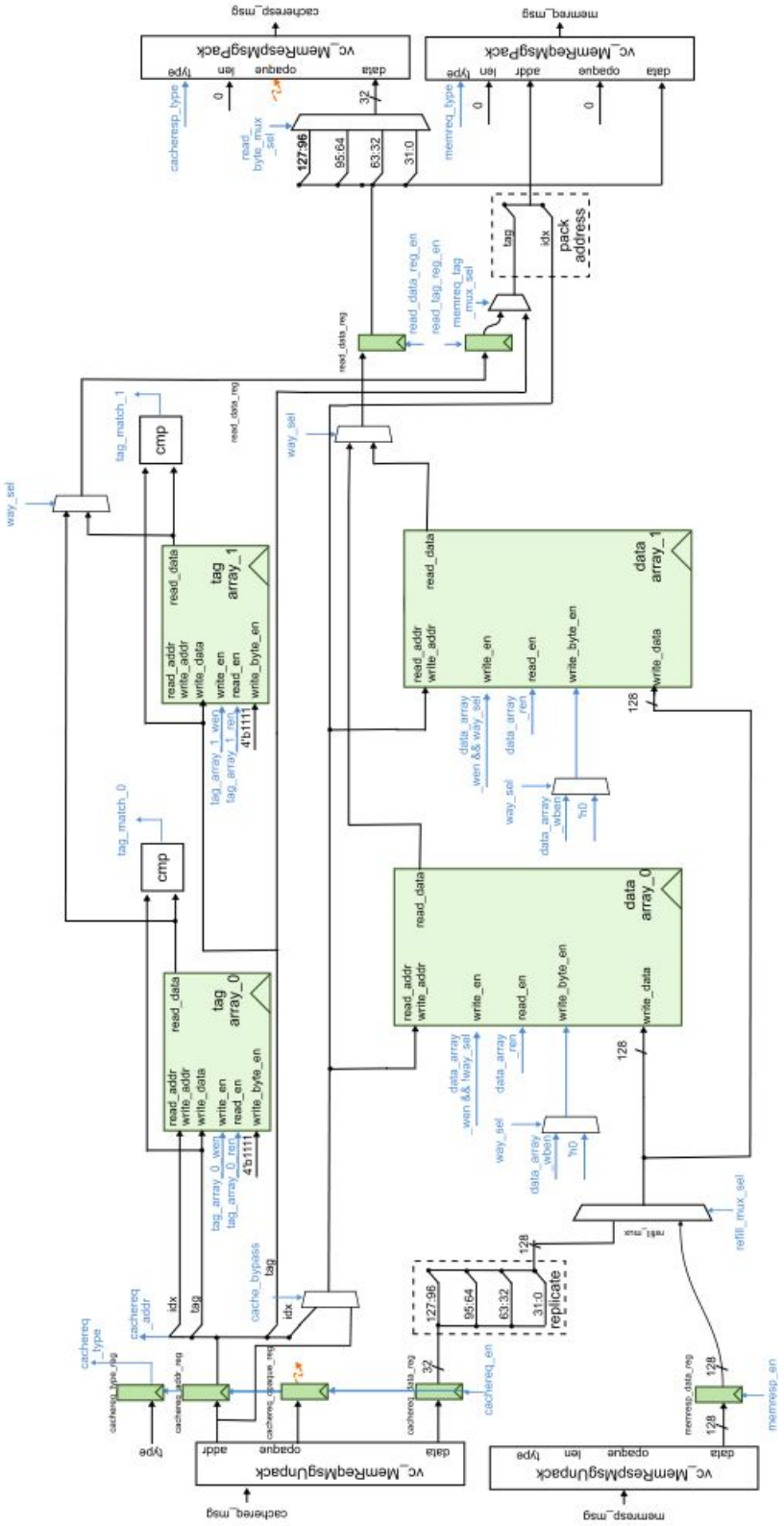

Figure 3: Baseline Cache Datapath

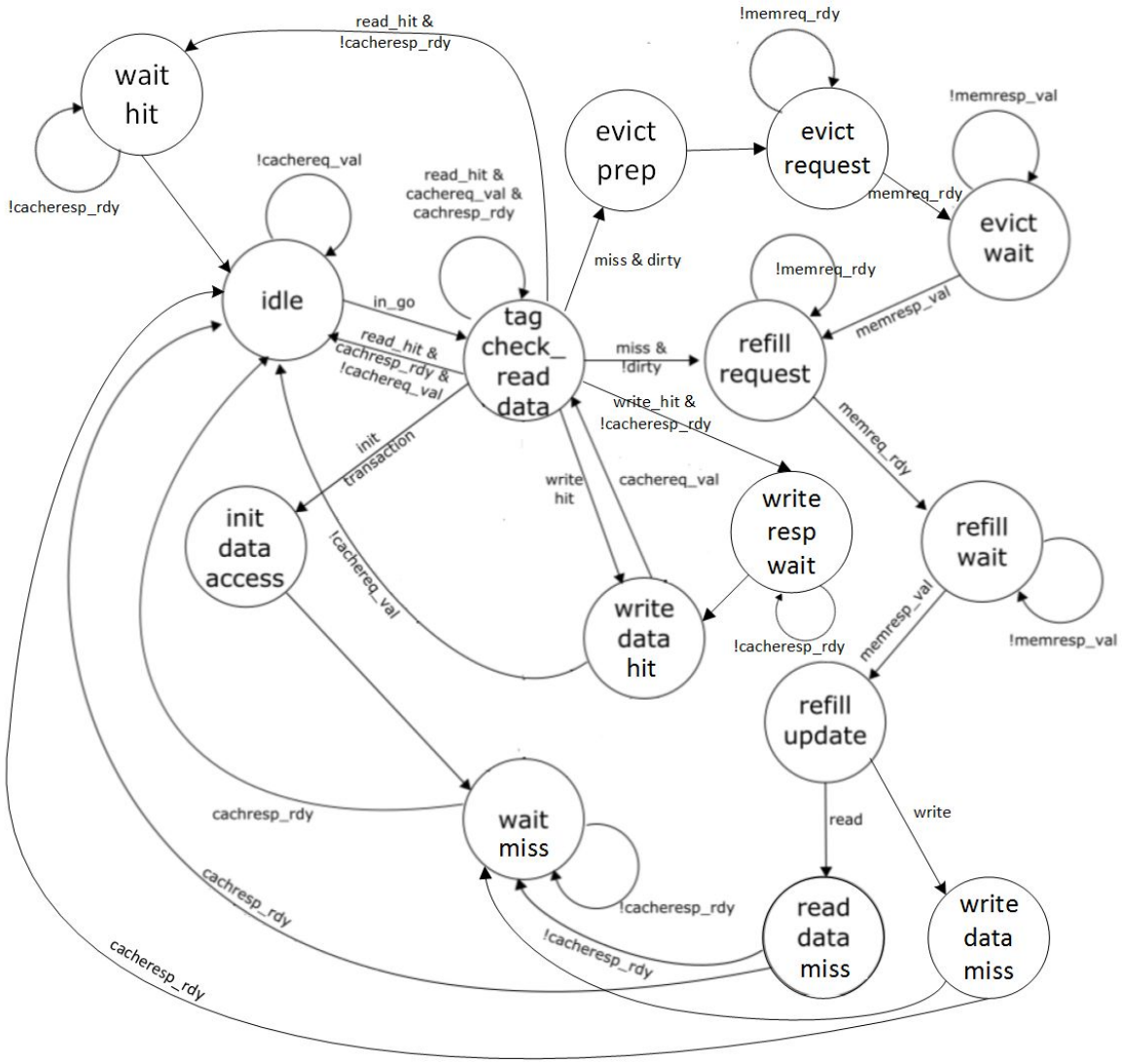

Figure 4: Baseline Cache FSM

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 on your software is that you cannot change the interface to the sorting function. The sorting function must take just an array and a size. You may not use any global state so if you need a temporary array you will need to dynamically allocate it on the heap in the implementation of your sorting function. The only restrictions on your hardware is 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 in-

terface. 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

The testing strategy has four parts: (1) testing of the accelerator in isolation; (2) testing of the processor and accelerator composition; (3) testing the accelerator using four-state RTL, fast-functional gate-level, and back-annotated gate-level simulation; and (4) testing the processor and accelerator composition using four-state RTL, fast-functional gate-level, and back-annotated gate-level simulation.

4.1. Testing Accelerator in Isolation

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 RTL model. You will need to modify the FL model and 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 ${TOPDIR}/sim/build
% pytest ..
% pytest ../lab2_xcel
% pytest ../lab2_xcel/test/SortXcelFL_test.py --verbose
% pytest ../lab2_xcel/test/SortXcel_test.py   --verbose
```

4.2. Testing Accelerator in Isolation Using ASIC Flow

Now that we know our accelerator works in isolation using two-state RTL simulation, we can use the ASIC flow to verify the accelerator works using four-state RTL, fast-functional gate-level, and back-annotated gate-level simulation. First, we generate the test benches for the accelerator in isolation.

```
% cd ${TOPDIR}/sim/build
% pytest ../lab2_xcel/test/SortXcel_test.py --test-verilog --dump-vtb
```

Now we can use pyhflow to push just the accelerator through the ASIC flow and also run the additional simulations. Note that the cycle time is fixed at 3ns since this is the target cycle time of the processor. Make sure every step is successful before moving on to the next step.

```
% mkdir -p ${TOPDIR}/asic/build-lab2-sort-xcel
% cd ${TOPDIR}/asic/build-lab2-sort-xcel
% pyhflow --one-test ../designs/lab2-sort-xcel.yml
% ./01-synopsys-vcs-rtlsim/run
% ./02-synopsys-dc-synth/run
% ./03-synopsys-vcs-ffglsim/run
% ./04-cadence-innovus-pnr/run
% ./05-synopsys-vcs-baglsim/run
```

If your design does not meet timing you will need to modify the design so it does meet timing.

4.3. Testing Processor with/without Accelerator

Once you know your accelerator works in isolation you should test the processor and accelerator composition by running using the simple C unit testing framework. First, let's build the tests for the pure-software baseline.

```
% mkdir -p ${TOPDIR}/app/build
% cd ${TOPDIR}/app/build
% ../configure --host=riscv32-unknown-elf
% make ubmark-sort-test
```

Then we can run this test on a FL model of the processor.

```
% cd ${TOPDIR}/app/build
% ../../sim/pmx/pmx-sim ./ubmark-sort-test
```

And finally we can run this test on the RTL model of the processor.

```
% cd ${TOPDIR}/app/build
% ../../sim/pmx/pmx-sim --proc-impl rtl ./ubmark-sort-test
```

Now let's build the test program that takes advantage of the sorting accelerator.

```
% cd ${TOPDIR}/app/build
% make ubmark-sort-xcel-test
```

Then we can make sure these tests pass on various compositions including: (1) FL processor and FL accelerator; (2) FL processor and RTL accelerator; and (3) RTL processor and RTL accelerator.

```
% cd ${TOPDIR}/app/build
% ../../sim/pmx/pmx-sim --proc-impl fl  --xcel-impl sort-fl  ./ubmark-sort-xcel-test
% ../../sim/pmx/pmx-sim --proc-impl fl  --xcel-impl sort-rtl ./ubmark-sort-xcel-test
% ../../sim/pmx/pmx-sim --proc-impl rtl --xcel-impl sort-rtl ./ubmark-sort-xcel-test
```

4.4. Testing Processor with/without Accelerator Using ASIC Flow

Now that we know our accelerator works in isolation and that the processor and accelerator composition works using two-state RTL simulation, we are finally ready to use the ASIC flow to verify the complete processor and accelerator composition also works using four-state RTL, fast-functional gate-level, and back-annotated gate-level simulation. First, we generate the test benches for the baseline processor.

```
% cd ${TOPDIR}/app/build
% ../../sim/pmx/pmx-sim --proc-impl rtl --xcel-impl null-rtl \
    --translate --dump-vtb ./ubmark-sort-test
```

Now we can use pyhflow to push the baseline processor through the ASIC flow and run the additional simulations. Note that the cycle time is fixed at 3ns since this is the target cycle time of the processor. Make sure every step is successful before moving on to the next step.

```
% mkdir -p ${TOPDIR}/asic/build-lab2-px-null
% cd ${TOPDIR}/asic/build-lab2-px-null
% pyhflow --one-test ../designs/lab2-px-null.yml
% ./01-synopsys-vcs-rtlsim/run
% ./02-synopsys-dc-synth/run
% ./03-synopsys-vcs-ffglsim/run
% ./04-cadence-innovus-pnr/run
% ./05-synopsys-vcs-baglsim/run
```

Now we can generate the test benches for the processor and accelerator composition.

```
% cd ${TOPDIR}/app/build
% ../../sim/pmx/pmx-sim --proc-impl rtl --xcel-impl sort-rtl \
    --translate --dump-vtb ./ubmark-sort-xcel-test
```

Finally, we can use pyhflow to push the processor and accelerator composition through the ASIC flow and also run the additional simulations. Make sure every step is successful before moving on to the next step.

```
% mkdir -p ${TOPDIR}/asic/build-lab2-px-sort
% cd ${TOPDIR}/asic/build-lab2-px-sort
% pyhflow --one-test ../designs/lab2-px-sort.yml
% ./01-synopsys-vcs-rtlsim/run
% ./02-synopsys-dc-synth/run
% ./03-synopsys-vcs-ffglsim/run
% ./04-cadence-innovus-pnr/run
% ./05-synopsys-vcs-baglsim/run
```

5. Evaluation

Once you have fully verified the functionality of both the accelerator in isolation as well as the processor and accelerator composition, you can use various interactive simulators to evaluate the cycle-level performance and the ASIC flow to evaluate the area, energy, and timing.

5.1. Evaluating Accelerator in Isolation

You can use the interactive sort accelerator simulator to evaluate the performance of your sorting accelerator in isolation. The simulator will display the number of cycles to execute the given input dataset.

```
% cd ${TOPDIR}/sim/build
% ../lab2_xcel/sort-xcel-sim --impl rtl --input random --stats --translate --dump-vtb
```

5.2. Evaluating Accelerator in Isolation using ASIC Flow

Now we can use the ASIC flow to determine the area, energy, and timing of the accelerator in isolation.

```
% cd ${TOPDIR}/asic/build-lab2-sort-xcel
% pyhflow ../designs/lab2-sort-xcel.yml
% ./run-flow
```

5.3. Evaluating Processor with/without Accelerator

We are now ready to evaluate the cycle-level performance of both baseline and alternative design using the provided interactive simulator. We need to build the corresponding programs.

```
% cd ${TOPDIR}/app/build
% make ubmark-sort-eval
% make ubmark-sort-xcel-eval
```

We can run the interactive simulator as follows.

```
% cd ${TOPDIR}/app/build
% ../../sim/pmx/pmx-sim --stats --proc-impl rtl --xcel-impl null-rtl \
    --translate --dump-vtb ./ubmark-sort-eval
% ../../sim/pmx/pmx-sim --stats --proc-impl rtl --xcel-impl sort-rtl \
    --translate --dump-vtb ./ubmark-sort-xcel-eval
```

The simulator will display the number of cycles for the baseline and alternative designs. 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.

5.4. Evaluating Processor with/without Accelerator using ASIC Flow

Finally, we can use pyhflow to push the baseline design through the flow and calculate the area, energy, and timing.

```
% cd ${TOPDIR}/asic/build-lab2-px-null
% pyhflow ../designs/lab2-px-null.yml
% ./run-flow
```

We can also use pyhflow to push the alternative design through the flow and calculate the area, energy, and timing.

```
% cd ${TOPDIR}/asic/build-lab2-px-sort
```

```
% pyhflow ../designs/lab2-px-sort.yml
% ./run-flow
```

Given all of these results, here is how you should analyze the execution time, cycle time, area, and energy.

- **Execution Time:** Use the number of cycles in the final report from the ASIC flow; this only counts cycles when stats are enabled. Use the line traces from `pmx-sim` to understand performance overheads.
- **Cycle Time:** Ensure your accelerator in isolation along with the processor/accelerator composition all meet timing at 3.0ns (333MHz). Use the timing reports from the accelerator in isolation to determine the critical path of your accelerator.
- **Area:** Use the area of your accelerator in isolation and compare it to the area of the baseline processor design. Use the hierarchical area reports from the accelerator in isolation to determine how much area each module in your design requires.
- **Energy:** Use the energy of running the baseline and alternative design. Use the hierarchical energy reports from the accelerator in isolation to determine how much energy each module in your design requires.

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.

Here are the steps we will use to determine the pareto-optimal frontier. We first clone your lab repo. Note that all of these steps must work from a clean clone, otherwise your design will not be considered for the pareto-optimal frontier.

```
% source setup-ece6745.sh
% mkdir -p $HOME/ece6745
% cd $HOME/ece6745
% git clone git@github.com:cornell-ece6745/lab2-groupXX
% cd lab2-groupXX
% TOPDIR=$PWD
```

We will then evaluate the area and cycle time of the accelerator in isolation.

```
% mkdir -p $TOPDIR/sim/build
% cd $TOPDIR/sim/build
% pytest ../lab2_xcel/test/SortXcel_test.py --test-verilog --dump-vtb
% ../lab2_xcel/sort-xcel-sim --impl rtl --input random --translate --dump-vtb

% mkdir -p $TOPDIR/asic/build-lab2-sort-xcel
```

```
% cd $TOPDIR/asic/build-lab2-sort-xcel
% pyhflow ../designs/lab2-sort-xcel.yml
% ./run-flow
```

We will use the area of your accelerator in isolation as one metric for determining the pareto-optimal frontier. The reason we use the area of your accelerator in isolation as opposed to the area of the processor and accelerator composition, is because we want to avoid any variation in the way the tools synthesize and place-and-route the processor from impacting our evaluation of the area of the accelerator. The cycle time of your accelerator in isolation must be less than the cycle time of the baseline design.

We will then evaluate the cycle time and the execution time of the processor and accelerator composition using your sorting microbenchmark.

```
% mkdir -p $TOPDIR/app/build
% cd $TOPDIR/app/build
% ../configure --host=riscv32-unknown-elf
% make ubmark-sort-xcel-test
% make ubmark-sort-xcel-eval

% cd $TOPDIR/app/build
% ../../sim/pmx/pmx-sim --proc-impl rtl --xcel-impl sort-rtl \
    --translate --dump-vtb ./ubmark-sort-xcel-test
% ../../sim/pmx/pmx-sim --proc-impl rtl --xcel-impl sort-rtl \
    --stats --translate --dump-vtb ./ubmark-sort-xcel-eval

% mkdir -p $TOPDIR/asic/build-lab2-px-sort
% cd $TOPDIR/asic/build-lab2-px-sort
% pyhflow ../designs/lab2-px-sort.yml
% ./run-flow
```

For those groups that want to submit two different accelerator designs, we will also do the following steps. Note that you are responsible for: modifying `sort-xcel-sim` and `pmx-sim` so they can correctly instantiate both sorting accelerators; adding a new `lab2-sort2-xcel.yml` design YAML to the `designs` directory; and adding a new microbenchmark which uses your second sorting accelerator.

```
% mkdir -p $TOPDIR/sim/build
% cd $TOPDIR/sim/build
% pytest ../lab2_xcel/test/SortXcel2_test.py --test-verilog --dump-vtb
% ../lab2_xcel/sort-xcel-sim --impl rtl2 --input random --translate --dump-vtb

% mkdir -p $TOPDIR/asic/build-lab2-sort2-xcel
% cd $TOPDIR/asic/build-lab2-sort2-xcel
% phyflow ../designs/lab2-sort2-xcel.yml
% ./run-flow

% mkdir -p $TOPDIR/app/build
% cd $TOPDIR/app/build
% ../configure --host=riscv32-unknown-elf
% make ubmark-sort2-xcel-test
```

```
% make ubmark-sort2-xcel-eval

% cd $TOPDIR/app/build
% ../../sim/pmx/pmx-sim --proc-impl rtl --xcel-impl sort2-rtl \
    --translate --dump-vtb ./ubmark-sort2-xcel-test
% ../../sim/pmx/pmx-sim --proc-impl rtl --xcel-impl sort2-rtl \
    --stats --translate --dump-vtb ./ubmark-sort2-xcel-eval

% mkdir -p $TOPDIR/asic/build-lab2-px-sort2
% cd $TOPDIR/asic/build-lab2-px-sort2
% phyflow ../designs/lab2-px-sort2-xcel.yml
% ./run-flow
```

7. Report

The *Lab Assignment Logistics* document provides general details about the requirements for the report. You must actually read this document to ensure you know what goes in your report and how to format it.

Alternative Design Section – In the alternative design section of your report you should discuss your sorting accelerator design in detail. Be sure to include one or more diagrams illustrating your design. If you also implemented more than one sorting accelerator, then you must describe all of these accelerators in the alternative design section.

Evaluation Section – In addition to tables and bar plots, you should include several energy vs. performance plots similar to what we use in lecture to compare the various designs. We recommend having two energy vs. performance plots; one plot for sorting a smaller array and one plot for sorting a larger array. Each plot should have two points: one for the baseline design and one for the alternative design. You could also have additional energy vs. performance plots for different datasets with the same input array size. These energy vs. performance plots should enable you to provide deep insight into the various trade-offs in the evaluation section of your report. If you also implemented more than one accelerator, then you must evaluate it in the alternative design section. **You must include at least two Klayout screenshots in your report: one of your accelerator in isolation and one of the processor and accelerator composition!**

Acknowledgments

This lab was created by Christopher Batten, Khalid Al-Hawaj, Jason Setter, Shunning Jiang, Moyang Wang, and Jack Brzozowski as part of the course ECE 6745 Complex Digital ASIC Design at Cornell University.