ECE 4750 Computer Architecture, Fall 2022
Lab 4: Single-Core and Multi-Core Systems

School of Electrical and Computer Engineering
Cornell University

revision: 2022-12-05-01-26

In this lab, you will be composing the multiplier, processor, and cache you have designed throughout the semester to implement a single-core system with its own instruction and data cache. You will also implement a simple ring network and compose the multiplier, processor, cache, and network to implement a simple multicore system with private instruction caches and a shared, banked data cache. In addition to unit testing, you will be using integration testing to verify various incremental compositions: (1) the ring network with special adapters to build memory networks suitable for sending memory requests and receiving memory responses; (2) the memory networks and the cache to build a shared, banked data cache; and (3) the entire single-core and multi-core systems. Your baseline design will be a combination of the single-core system hardware and single-core system software in the form of a single-threaded sorting microbenchmark. Your alternative design will be a combination of the multi-core system hardware and multi-core system software in the form of a multi-threaded sorting microbenchmark. You will be using a standard RISC-V cross-compiler to compile C programs down to a RISC-V binary suitable for execution on our RISC-V simulators. You are free to choose any single-threaded or multi-threaded sorting algorithm you like. You are required to implement the baseline and alternative designs, verify the designs using an effective testing strategy, and perform an evaluation comparing the two implementations. Advanced students will have an opportunity to improve the performance of their sorting microbenchmarks through either hardware or software improvements. **The milestone for this lab is to complete the baseline design and corresponding single-threaded sorting microbenchmark. You should consult the course lab logistics document 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:

- composition of simpler subsystems to create more complex systems;
- programming single- and multi-threaded C programs;
- interaction between hardware and software design;
- design principles including modularity, hierarchy, encapsulation, and regularity;
- design patterns including latency insensitive message interfaces;
- agile design methodologies including incremental development and test-driven development;
- computer architecture evaluation methodologies based on architecture-level statistics;

This handout assumes that you have read and understand the course tutorials. You should have already used the `ece4750-lab-admin` script to create or join a GitHub group. To get started, login to an `ecelinux` server, source the setup script, and clone your lab group's remote repository from GitHub:

```
% source setup-ece4750.sh
% mkdir -p ${HOME}/ece4750
% cd ${HOME}/ece4750
% git clone git@github.com:cornell-ece4750/lab-groupXX
```

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 hardware and software tests in the lab like this:

```
% cd ${HOME}/ece4750/lab-groupXX
% git pull --rebase

% mkdir -p sim/build
% cd sim/build
% pytest ../lab4_sys

% mkdir -p app/build-native
% cd app/build-native
% ../configure
% make check

% mkdir -p app/build
% cd app/build
% ../configure --host=riscv32-unknown-elf
% make check
```

Some of the tests will pass but many will fail since you have not implemented the single-core composition, the ring network, nor the sorting algorithms. For this lab, you will be working in the `lab4_sys` subproject which includes the following files (note many of the Verilog RTL files also include corresponding Python wrappers that are not listed here):

- `SingleCoreSysFL.py` – FL single-core system
- `SingleCoreSys.v` – Verilog RTL for single-core system
- `NetMsg.py` – Network message/packet type
- `NetRouterRouteUnit.v` – Verilog RTL for route unit
- `NetRouterSwitchUnit.v` – Verilog RTL for switch unit
- `NetRouter.v` – Verilog RTL for network router
- `Net.v` – Verilog RTL for ring network
- `MsgAdapters.v` – Verilog RTL adapters to convert MemMsg/NetMsg
- `CacheNet.v` – Verilog RTL for cache req/resp network
- `MemNet.v` – Verilog RTL for memory req/resp network
- `MultiCoreDataCache.v` – Verilog RTL for multi-core banked data cache
- `MultiCoreSys.v` – Verilog RTL for multi-core system
- `sys-sim` – System simulator
- `__init__.py` – Package setup
- `test/harness.py` – Test harness
- `test/simple_score_test.py` – Very simple test for single-core sys
- `test/simple_mcore_test.py` – Very simple test for multi-core sys
- `test/SingleCoreSysFL_test.py` – Single-core system FL tests
- `test/SingleCoreSys_test.py` – Single-core system tests
- `test/NetRouterRouteUnit_test.py` – Network route unit tests

- `test/NetRouterSwitchUnit_test.py` – Network switch unit tests
- `test/NetRouter_test.py` – Network router unit tests
- `test/Net_test.py` – Network tests
- `test/CacheNet_test.py` – Cache req/resp network tests
- `test/MemNet_test.py` – Memory req/resp network tests
- `test/MultiCoreDataCache_test.py` – Multi-core banked data cache tests
- `test/MultiCoreSysFL_test.py` – Multi-core system FL tests
- `test/MultiCoreSys_test.py` – Multi-core system tests
- `test/mem_sim_test.py` – System simulator test
- `test/__init__.py` – Package setup

1. Introduction

Ever since transistor scaling has stopped delivering exponential growth in performance within similar area and power constraints, computer architects have tried different ways to scale the processor performance, beyond simply exploiting more instruction-level parallelism (ILP). Two promising approaches have been to exploit data-level parallelism (DLP) and/or thread-level parallelism (TLP). Multicore processors (a.k.a., chip multiprocessors) exploit TLP by allocating different threads within a single program to the different cores within the chip. Ideally, fully parallelizing an application on P processors leads to about P$\times$ speedup.

However, multicore designs also have their drawbacks. First of all, it is not possible to fully parallelize most applications, and serial portions of the code can quickly dominate the execution time. In this course we are using a simplified multicore system to ensure correct execution, but a *realistic* multicore system must manage coherence, consistency, and synchronization. Finally, the multicore design definitely has higher hardware complexity and more hardware overall. In this lab, you will get the chance to explore some of these trade offs using a single-core system with its own instruction and data cache, and a multi-core system with four cores each with private instruction caches and a shared, banked data cache.

We provide functional-level models of both the single-core and multi-core systems to enable verifying your tests before applying them to your RTL implementations. You can reuse assembly tests for just a representative subset of instructions (since the simulation time is increased due to more complex models) to test the single-core system. To test the multi-core composition, we introduce an extension to our assembly test methodology that enables initializing multiple test sources and sinks (one per core). In the spirit of incremental design, we will work with these sub-compositions first: a cache network, a memory network, and the multi-core data cache before testing the multi-core system.

Unlike previous labs, this lab has significant portion of software design. We organize files as follows: `sim` contains Verilog RTL designs and unit tests; and `app` is the microbenchmark directory you will be working in for the software side of this lab. We will be using a standard RISC-V cross-compiler which compiles C programs to the full RISC-V ISA and a modular C++ build system to simplify running the compiler.

2. Baseline Design

The baseline design will include both the single-core system hardware and the single-core system software to create a complete single-core system capable of sorting arrays of integers.

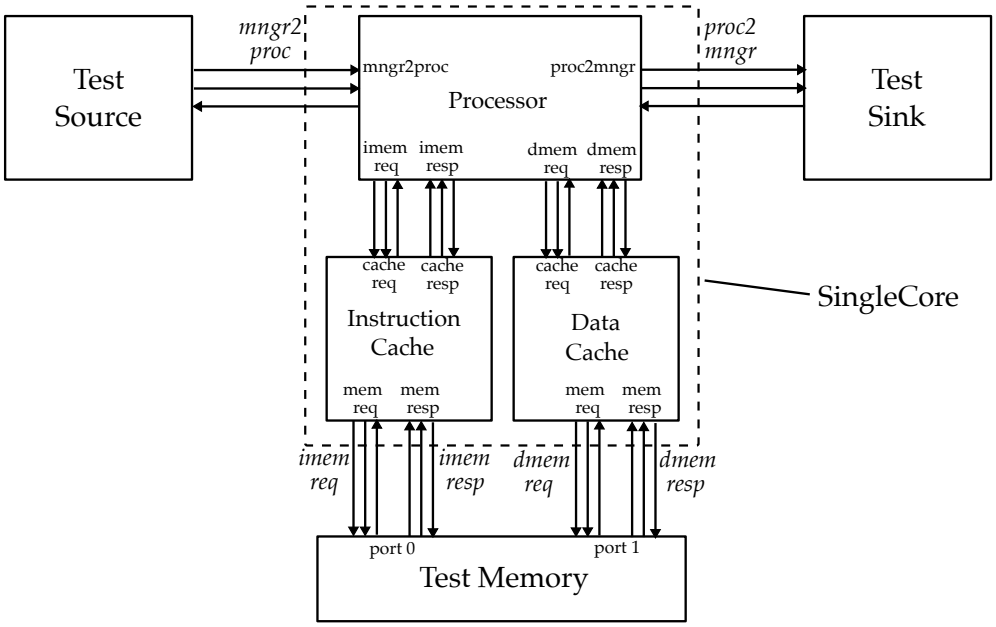

Figure 1: SingleCore – The `SingleCore` module as a whole, is hooked up to the test source, test sink, and test memory for testing and evaluation. Each bundle of the three arrows is a msg/val/rdy port bundle. The ports with inclined names are the top-level ports that `SingleCore` expose.

2.1. Single-Core System Hardware

The single-core system hardware will compose the processor from Lab 2 (which includes the multiplier from Lab 1) and two instances of the cache from Lab 3 (one as the instruction cache, and the other as the data cache). The composition and the connections for the single-core system are shown Figure 1. The cache request and response ports of the caches connect to the respective i/dcache ports of the processor, while the memory request and response ports connect to outside facing i/dmemreq and i/dmemresp ports as shown. Note that the data bitwidth from the processor to caches is one word (i.e., 32 bits), while the data bitwidth from the caches to main memory is the full cache line (i.e., 128 bits). Note that you should make sure to set the `core_id` for the processor to zero and the number of banks for both the instruction and data cache to one. You will implement the single-core composition in `SingleCoreSys.v`.

2.2. Single-Core System Software

The single-core system software will be a single-threaded sorting algorithm which should be implemented in C. You can use any algorithm you like, although keep in mind that your goal is to make a fast sorting algorithm. You are welcome to consult textbooks or online resources to learn more about sorting algorithms, but you should cite these resources. Copying code is not allowed; you should write it on your own. You will implement the sorting algorithm in `app/ubmark/ubmark-sort.c`.

3. Alternative Design

The alternative design will include both the multi-core system hardware and the multi-core system software to create a completely multi-core system capable of using parallel execution to sort an array of integers.

3.1. Multi-Core System Hardware

The multi-core system hardware will compose four instances of the processor from Lab 2 (which includes the multiplier from Lab 1) and eight instances of the cache from Lab 3 (four instruction caches and four data caches). You will implement a new ring network which will be used to interconnect the processors to the data caches and to interconnect the caches to the main memory interfaces.

The ring network is shown in Figure 3. The network has four input stream interfaces (also called input terminals) and four output stream interfaces (also called output terminals). Figure 4 shows the network message format which includes a two-bit source field, two-bit destination field, eight-bit opaque field, and payload. The network is implemented using four network routers each with three input stream interfaces and three output stream interfaces. Network messages are injected into the network on one of four input terminals as indicated by the source field, then traverse the routers in the ring, and eventually are ejected on one of four output terminals as indicated by the destination field. The network router is implemented using three four-entry normal input queues, three route units, and three switch units as shown in Figure 5. The route units implement the routing algorithm by using the network message's destination field and the router id to decide which switch unit to send the network message. The switch unit implements the arbitration algorithm which determines which message can go out the corresponding output port.

Start by implementing the route unit in `NetRouterRouteUnit.v` and the switch unit in `NetRouterSwitchUnit.v`. An initial route unit might use a simple routing algorithm which always routes all network message clockwise around the ring network. A more optimized routing algorithm might choose the shortest path to the desired output terminal. An initial switch unit might use fixed priority arbitration by always giving priority to messages in the ring network over new messages arriving from an input terminal. A more optimized arbitration algorithm might use oblivious rotational arbitration or round-robin arbitration to make a fairer decision. Once you have implemented the route unit and switch unit, compose them with three normal queues to implement the network router. Once you

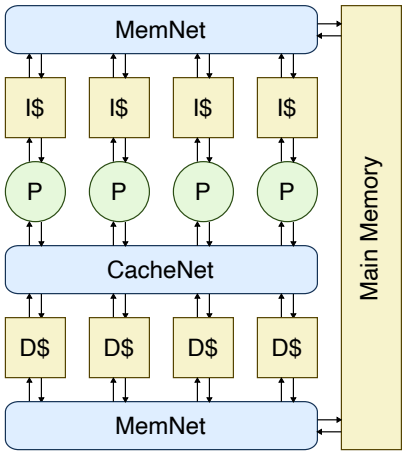

Figure 2: Alternative design block diagram – The alternative design consists of four processors, four private I-caches, a four-banked shared D-cache, and several networks to route dmem request/response from the processors and the D-cache, and refilling both the I-cache and the D-cache. Note that each network shown in the diagram is actually two networks: one for request and the other for response.

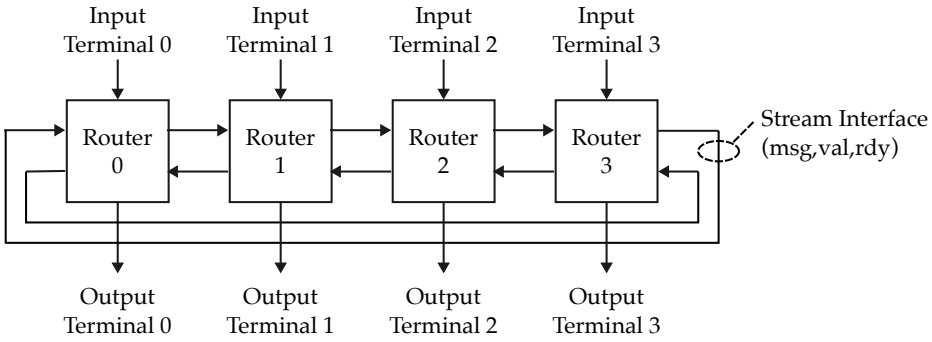

Figure 3: Ring Network – Ring network has four input streams (terminals) and four output streams (terminals) and is implemented using four routers. All arrows are stream interfaces with corresponding msg, val, rdy ports.

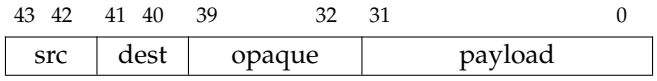

Figure 4: Network Message Format – Network messages are sent from a network input terminal to a network output terminal. Example message is shown with a payload of 32 bits.

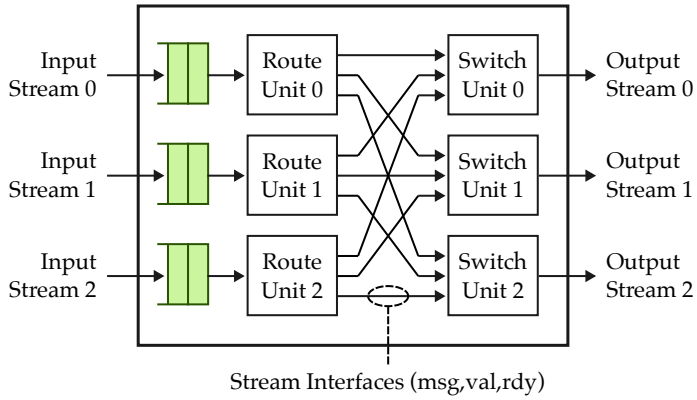

Figure 5: Ring Network Router – Ring network router has three input streams and three output streams and is implemented using three input queues, three route units, and three switch units. All arrows are stream interfaces with corresponding msg, val, rdy ports.

have implemented the network router, you can compose four routers to create the complete ring network.

Once you have the ring network implemented, we can now put together the complete multi-core system as shown in Figure 2. We will take an incremental design approach to build the whole system. **Note that we have provided you with the following sub-compositions with unit tests!**

- **Cache Network**: the request/response network pair for interconnecting four processors and four cache banks
- **Memory Network**: the request/response network pair for interconnecting four caches and one main memory port
- **Multi-Core Data Cache**: the cache network, four cache banks, and the memory network

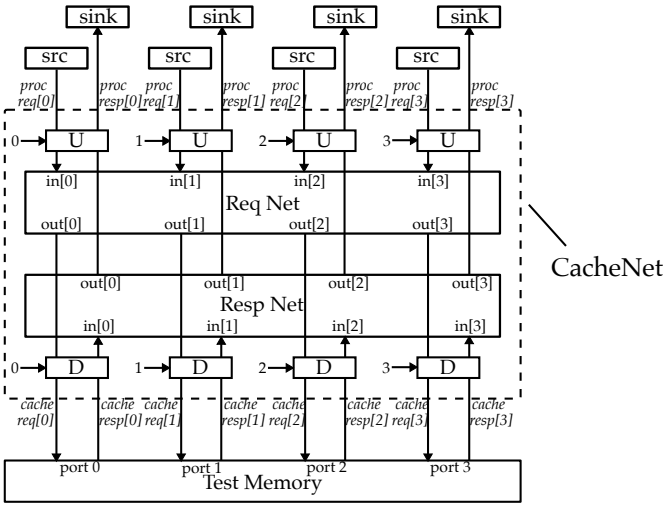

Figure 6: Cache Network – The cache network as a whole, is hooked up to the test sources, test sinks, and a mock up for unit testing. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. Each "U" and "D" is an adapter pair that takes care of both the request port and the response ports.

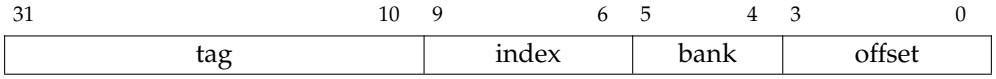

Figure 7: Memory Address Formats With Banking – Addresses for the data cache include two bank bits which are used to choose which bank to send the memory request; in other words, the bank bits are used as the destination field when converting a memory request message into a network message.

Figure 6 shows the diagram of the cache network that we are going to use to interconnect the four processors and the four data cache banks. You can see from the diagram that we instantiate two networks (a request network and a response network). There are several important adapters to convert between memory messages and network messages at each network terminal. "U" corresponds to an upstream adapaters and the "D" corresponds to a downstream adapters. The upstream adapters convert from cache requests to network messages, and the downstream adapters covert network messages back into the corresponding cache request. The upstream adapters look at the bank bits in the address (see Figure 7) in the cache request to determine the appropriate destination output terminal for the request network. The upstream adapters also set the source field in the network message appropriately. The downstream adapters will store these source and destination fields in the opaque field of the cache request. Eventually, the cache response will come back from the cache and the downstream adapters also convert cache responses into network messages, and the upstream adapters convert network messages back into the corresopnding cache response. The downstream adapters look at the opaque field in the cache response to determine the appropriate destination output terminal for the response network. The downstream adapters also set the source field in the network message appropriately.

Unlike the cache network, the memory network only uses one output terminal, but has four input terminals (see Figure 8). In the memory network, the upstream adapters always inserts zero as the

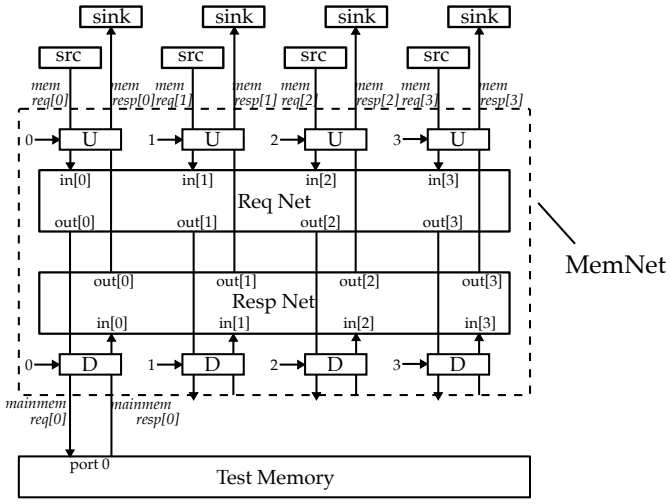

Figure 8: Memory Network – The memory network as a whole, is hooked up to the test sources, test sinks, and a mock up for unit testing. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. Each "U" and "D" is an adapter pair that takes care of both the request port and the response port.

destination output terminal. The rest of the network is identical to the cache network except of course the messages are much wider since this network is to handle cache refills and evictions.

We compose the cache network, memory network, and four cache banks to create the complete shared, banked data-cache subsystem. We set `p_num_banks=4` for each data cache to ensure they correctly handle the bank bits. Figure 9 shows the diagram of multi-ore data cache.

Finally, we can compose the cache network, processors, and the multi-core data cache to create the complete multi-core system shown in Figure 10. We instantiate four processors with `p_num_cores=4` and connect 0, 1, 2, 3 respectively to the `core_id` port of each processor. We must make sure `p_num_banks=0` for each of the instruction cache since they are private caches and not banked.

3.2. Multi-Core System Software

The multi-core system software will be a multi-threaded sorting algorithm which should be implemented in C using the provided threading library. You are free to implement any parallel sorting algorithm. This is a non-trivial task since there are many trade-offs in terms of amount of parallelism, locality, and load balancing. A simple approach is to implement a bottom-up hybrid sorting algorithm. Divide the input array into four blocks. Each core can use the single-threaded sorting algorithm you used in the baseline to sort its block. Then have core 0 merge the four blocks using mergesort. You are welcome to consult textbooks or online resources to learn more about different sorting algorithms, but you should cite these resources. Copying code is not allowed; you should write it on your own. You will implement the sorting algorithm in `app/mtbmark/mtbmark-sort.c`.

All four cores always start executing the exact same program at the same reset vector. There is always one-to-one mapping between cores and threads (i.e., exactly one thread runs on each core), so we use the terms thread and core interchangeably in this lab. Cores can use two control/status registers to enable them to execute different code. CSR 0xfc1 contains the total number of cores in the system,

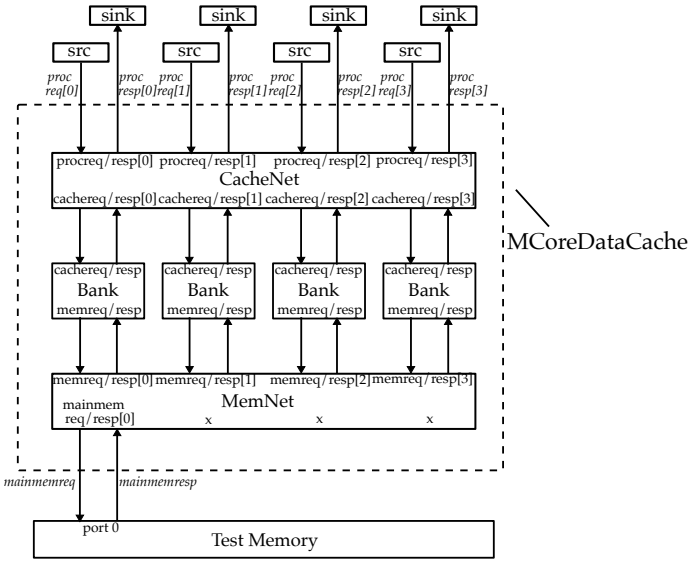

Figure 9: MCoreDataCache – The `McoreDataCache` module as a whole, is hooked up to the test sources, test sinks, and single-port test memory for unit testing. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. We omit the details inside `MemNet` and `CacheNet` but show the port names to match the previous diagrams. The ports with inclined names are the top-level ports that this module exposes.

and CSR 0xf14 contains the current core's unique ID. If there are four cores in the system, then the core IDs will be 0, 1, 2, and 3.

We provide you a very light-weight threading library, called `bthread`. We call core 0 the master core and the other cores the worker cores. Figure 11 illustrates how to use the `bthread` library to implement a parallel accumulation microbenchmark that makes use of two cores. Before main, each core calls `ece4750_bthread_init`; core 0 will perform some initialization and then return from this function, while cores 1–3 stay in this function executing a worker loop. The master core can spawn work onto a worker core by using the `ece4750_bthread_spawn` function, where the first argument is the ID of the core (thread) we want spawn to, the second argument is the function pointer we want the worker core to execute, and the final argument is a pointer to a structure that holds the arguments the worker core should use to do the work. The `ece4750_bthread_join` function will cause a core to wait for given worker core. We provide you with four multi-threaded microbenchmarks to show how to use bthread library to parallelize single-threaded microbenchmarks. You may also want to look at the implementation of each `bthread` library in `app/ece4750/ece4750-bthread.h` and `app/ece4750/ece4750-bthread.c`. Note that only core 0 can use `ece4750_wprintf`.

`bthread` can run multi-threaded programs on a single-core system; `ece4750_bthread_spawn` directly executes the given function pointer instead of spawning the work onto a worker thread. `bthread` can also run single-threaded programs on a multi-core system by setting the number of workers using the `ece4750_bthread_set_num_workers` function.

A simple way to use the `bthread` library to implement the hybrid sorting algorithm mentioned above is to divide the input array into four blocks, create four argument structures, use a `ece4750_bthread_spawn` to spawn work onto cores 1–3, have core 0 also do some work, then core 0 can use a join to wait for cores 1–3 to finish, and core 0 can do the final merge.

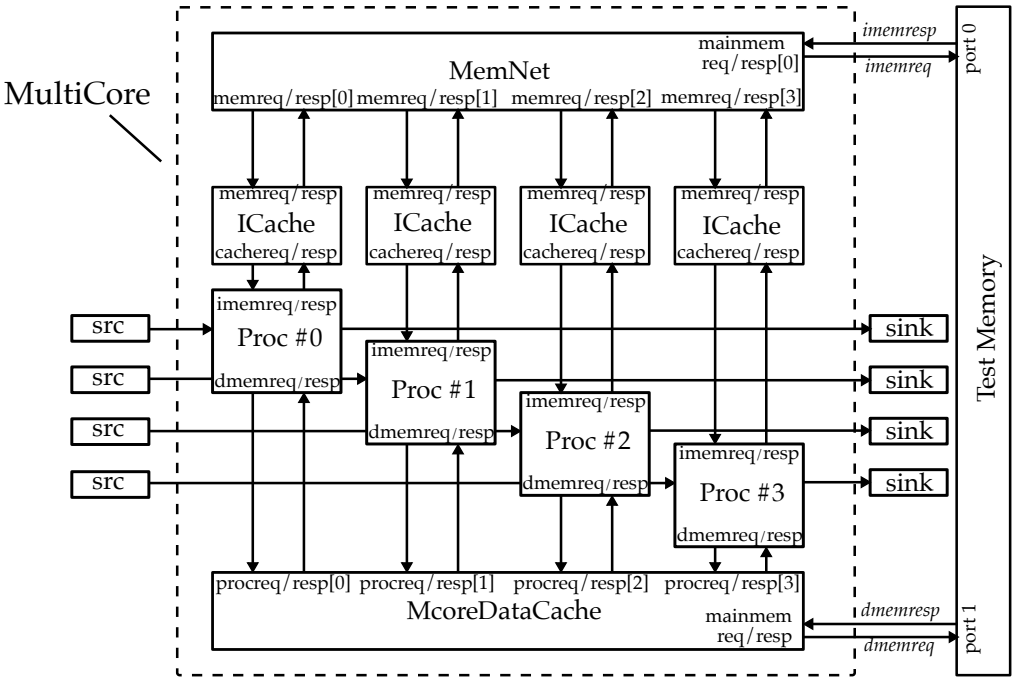

Figure 10: MultiCore – The `MultiCore` module as a whole, is hooked up to a dual-port test memory, four test sources and four test sinks. Note that to not to make the diagram too crowded, we omit the *proc2mngr* and *mngr2proc* port names both at the boundary of `Multicore` as well as in the processor. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. The ports with inclined names are the top-level ports that `MultiCore` exposes.

4. Testing Strategy

This lab requires an effective testing strategy for both hardware and software. On the hardware side, you will be using the same Python-based testing framework as in previous labs. On the software side, you will be using a simple C testing framework.

4.1. Single-Core System Hardware Testing

Start by making sure your multiplier from Lab 1, processor from Lab 2, and cache from Lab 3 are passing all tests. There is no sense in testing the single-core system if we are not sure the components are working!

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ../lab1_imul
% pytest ../lab2_proc
% pytest ../lab3_mem
```

To test the single-core system hardware, we will reuse the assembly tests for a representative subset of the instructions in Lab 2. We have already done extensive unit testing in the previous labs, so our goal in this lab is to focus on those tests that might expose bugs in the single-core system. We recommend testing the following instructions.

```
typedef struct
{
  int* sum_ptr; // accumulation result
  int* src;     // pointer to src array
  int  first;   // first element this core should process
  int  last;    // (one past) last element this core should process
}
arg_t;

void work( void* arg_vptr )
{
  // Cast void* to argument pointer.

  arg_t* arg_ptr = (arg_t*) arg_vptr;

  // Create local variables for each field of the argument structure.

  int* sum_ptr = arg_ptr->sum_ptr;
  int* src     = arg_ptr->src;
  int  first   = arg_ptr->first;
  int  last    = arg_ptr->last;

  // Do the actual work

  int sum = 0;
  for ( int i = first; i < last; i++ )
    sum += src[i];

  *sum_ptr = sum;
}

int mtbmark_accumulate( int* src, int size )
{
  int block_size = size/2;

  // Create four argument structures that include the array pointers and
  // what elements each core should process.

  int sum0;
  int sum1;

  arg_t arg0 = { &sum0, src, 0,      size/2 };
  arg_t arg1 = { &sum1, src, size/2, size   };

  // Spawn work onto core 1

  ece4750_bthread_spawn( 1, &work, &arg1 );

  // Have core 0 also do some work.

  work( &arg0 );

  // Wait for core 1

  ece4750_bthread_join(1);

  // Return final sum

  return sum0 + sum1;
}
```

Figure 11: Simple Parallel Accumulate Microbenchmark for Two Cores

- CSR : `csrr, csrw`
- Reg-Reg : `add, mul`
- Reg-Imm : `addi`
- Memory : `lw, sw`
- Branch : `bne`
- Jump : `jal`
- Any additional mixed instruction tests

You should update `SingleCoreSysFL_test.py` appropriately with your test cases. Feel free to copy tests from the Lab 2 staff tests if you like. You can then use `pytest` to run the tests on the FL model and RTL model.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ../lab4_sys/test/SingleCoreSysFL_test.py
% pytest ../lab4_sys/test/SingleCoreSys_test.py
```

4.2. Single-Core System Software Testing

To test the single-core system software, you will need to use the simple C unit testing framework provided for you as part of the `ece4750` standard C library. You can apply all of the same test-driven techniques we have used for hardware to test your software including ad-hoc vs. assertion testing, directed vs. random testing, and black-box vs. white-box testing.

You should always start by making sure your tests pass natively before testing your software on a FL or RTL simulator. You can compile and run the tests for your single-threaded sorting algorithm natively using the provided build system.

```
% cd ${HOME}/ece4750/app/build-native
% make ubmark-sort-test
% ./ubmark-sort-test
```

Once your tests pass natively, you can also cross-compile and run the tests for your single-threaded sorting algorithm on the single-core system FL simulator.

```
% cd ${HOME}/ece4750/app/build
% make ubmark-sort-test
% ../../sim/lab4_sys/sys-sim ./ubmark-sort-test
```

One your tests pass natively and on the FL simulator, then you can run the tests on the RTL implementation of the single-core system.

```
% cd ${HOME}/ece4750/app/build
% make ubmark-sort-test
% ../../sim/lab4_sys/sys-sim --impl base ./ubmark-sort-test
```

We provide you with a simple basic test for your sorting algorithm which is shown in Figure 12. A test case is simply a function. Every test case should start with a call to `ECE4750_CHECK` to set the test case name, and then a series of calls to various `ECE4750_CHECK` macros to verify properties of your program. The `main` function checks command line arguments and then calls every test case function. You will need to add many more test cases. You must use assertion testing as opposed to ad-hoc testing (i.e., you must use macros to verify properties, not just print out values and verify the results manually). You should primarily use directed testing with some limited random testing

```
#include "ece4750.h"
#include "ubmark-sort.h"

void test_case_1_sort_basic()
{
  ECE4750_CHECK( L"test_case_1_sort_basic" );

  int a[]     = { 4, 3, 6, 5, };
  int a_ref[] = { 3, 4, 5, 6, };

  ubmark_sort( a, 4 );

  for ( int i = 0; i < 4; i++ )
    ECE4750_CHECK_INT_EQ( a[i] , a_ref[i] );
}

int main( int argc, char** argv )
{
  __n = ( argc == 1 ) ? 0 : ece4750_atoi( argv[1] );

  if ( (__n <= 0) || (__n == 1 ) ) test_case_1_sort_basic();

  ece4750_wprintf( L"\n\n" );
  return ece4750_check_status;
}
```

Figure 12: Example C Test Program for Sorting Algorithm

(i.e., with calls to `ece4750_srand` and `ece4750_rand`. You should use both black-box testing (i.e., test the `ubmark_sort` function) along with white-box testing (i.e., test helper functions like a `partition` function for quick sort, a `merge` function for merge sort, or a small sorting function optimized for the base case in a recursive algorithm).

4.3. Multi-Core System Hardware Testing

You will need to start with unit testing before testing the multi-core system hardware. Write unit tests for your route unit and switch unit. Once you know these units are functionally correct, write unit tests for your router. **Make sure your test your router with different router IDs!** Once you know your router is functionally correct, write unit tests for the complete ring network. Use a combination of directed and random testing for your network. Since sometimes it can be hard (especially with random testing) to determine the exact order that messages will arrive at a stream sink, you may want to use the `ordered=False` flag for the sink. This flag means the sink will only check that a received message is present in the sink by searching all expected messages for a match. **Your routing and arbitration algorithms may not guarantee deadlock freedom, which means you may need to be careful with tests that rapidly inject many messages!**. It is fine for your network not to guarantee deadlock freedom, because when we use the network in the multi-core system it will only experience very light loads (i.e., each processor and cache can only have a single request/response in flight at a time). You can use `pytest` to run all of the network tests.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ../lab4_sys/test/Net*
```

```
csrr x2, mngr2proc < {1,2,3,4} # init each core's mngr2proc with different values
csrr x3, mngr2proc < {2,3,4,5} # init each core's mngr2proc with different values
add  x4, x2, x3
csrw proc2mngr, x4 > {3,5,7,9} # verify each core's proc2mngr with different values
```

Figure 13: Example of Extended Assembly Syntax for Multi-Core System Testing

Since we do not know the details of your routing and arbitration algorithms, we cannot apply our own staff tests to your route unit, switch unit, or router. However, we will apply our own staff tests to your ring network.

After completing the unit testing for the ring network, you can use the tests we provide you to test the cache network, memory network, and multi-core data cache. *It is critical for you to have robust directed and random testing of your network before testing the cache network, memory network, multi-core data cache, and the multicore system!*

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ../lab4_sys/test/CacheNet_test.py
% pytest ../lab4_sys/test/MemNet_test.py
% pytest ../lab4_sys/test/MultiCoreDataCache_test.py
```

Once you have passed all of these tests, you are ready to reuse assembly tests from the single-core testing to test the multi-core system. Again you should test a representative subset of instructions. When you reuse the single-core tests you will essentially be testing each core executing the same test in parallel. Note that if two cores load, modify, store different values to the same address there is no guarantee what the final correct result should be. This is called a "race condition". You may need to exclude some of your SW tests from Lab 2 if you cannot guarantee that a LW always produces the same value regardless of what the other cores are doing. You should update `MultiCoreSysFL_test.py` appropriately with your test cases. Feel free to copy tests from the Lab 2 staff tests if you like. You can then use `pytest` to run the tests on the FL model and RTL model.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ../lab4_sys/test/MultiCoreSysFL_test.py
% pytest ../lab4_sys/test/MultiCoreSys_test.py
```

Once you have these tests passing, you need to add some specific tests meant meant to specifically stress the multicore memory system. You will need to make use of a new feature in our assembly test syntax which enables initializing multiple test sources and sinks. See Figure 13 for an example of how to write a test where each core does an ADD instruction on different data producing four different results. In this example, core 0 will add 1 and 2 while core 1 will add 2 and 3. Focus on tests with different load/store access patterns, but keep in mind that different cores should not load/store the same word! Again, if two cores load, modify, store different values to the same address there is no guarantee what the final correct result should be. You *should* include tests where multiple cores are accessing different words on the same cache line. Random testing is also great to help stress the multicore memory system. You can then use `pytest` to run these new tests on the FL model and RTL model.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ../lab4_sys/test/MultiCoreSysFL_mem_test.py
% pytest ../lab4_sys/test/MultiCoreSys_mem_test.py
```

You should now be able to pass all hardware tests for all labs.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% pytest ../lab1_imul
% pytest ../lab2_proc
% pytest ../lab3_mem
% pytest ../lab4_sys
```

4.4. Multi-Core System Software Testing

To test the multi-core system software, you should appply the same strategy you used for testing the single-core system software. As always, make sure your tests pass natively before testing your software on a FL or RTL simulator. You can compile and run the tests for your multi-threaded sorting algorithm natively using the provided build system.

```
% cd ${HOME}/ece4750/app/build-native
% make mtbmark-sort-test
% ./mtbmark-sort-test
```

Once your tests pass natively, you can also cross-compile and run the tests for your multi-threaded sorting algorithm on the single-core system FL simulator.

```
% cd ${HOME}/ece4750/app/build
% make mtbmark-sort-test
% ../../sim/lab4_sys/sys-sim ./mtbmark-sort-test
```

One your tests pass natively and on the single-core system FL simulator, then you can run the tests on multi-core system FL simulator.

```
% cd ${HOME}/ece4750/app/build
% make mtbmark-sort-test
% ../../sim/lab4_sys/sys-sim --impl mcore-fl ./mtbmark-sort-test
```

Finally, you can run the tests on the the RTL implementation of the multi-core system.

```
% cd ${HOME}/ece4750/app/build
% make mtbmark-sort-test
% ../../sim/lab4_sys/sys-sim --impl alt ./mtbmark-sort-test
```

5. Evaluation

We provide you with four single-threaded microbenchmarks and four multi-threaded bencmarks that illustrate different trade-offs in terms of arithmetic intensity, data access regularity, and control flow regularity. Your sorting algorithm makes a fifth microbenchmark.

5.1. Single-Core System Evaluation

We provide you with four single-threaded microbenchmarks: *vvadd*, *mfilt*, *cmult*, and *bsearch*, whose assembly sequences have been studied in Lab 2. Your sorting algorithm makes a fifth microbenchmark. All single-threaded microbenchmarks are in the `app/ubmark` directory with their C source code, test programs, evaluation programs, and dataset files. Figure 14 shows the evaluation program

```
#include "ece4750.h"
#include "ubmark-vvadd.h"
#include "ubmark-vvadd.dat"

int main( void )
{
  // Allocate destination array for results

  int* dest = ece4750_malloc( eval_size * (int)sizeof(int) );

  // Run the evaluation

  ece4750_stats_on();
  ubmark_vvadd( dest, eval_src0, eval_src1, eval_size );
  ece4750_stats_off();

  // Verify the results

  for ( int i = 0; i < eval_size; i++ ) {
    if ( dest[i] != eval_ref[i] ) {
      ece4750_wprintf( L"\n FAILED: dest[%d] != eval_ref[%d] (%d != %d)\n\n",
                       i, i, dest[i], eval_ref[i] );
      ece4750_exit(1);
    }
  }

  // Free destination array

  ece4750_free(dest);

  // Check for no memory leaks

  if ( ece4750_get_heap_usage() != 0 ) {
    ece4750_wprintf( L"\n FAILED: memory leak of %d bytes!\n\n",
                     ece4750_get_heap_usage() );
    ece4750_exit(1);
  }

  // Otherwise we passed

  ece4750_wprintf( L"\n **PASSED** \n\n" );

  return 0;
}
```

Figure 14: Example C Evaluation Program for Vector-Vector Add

for the single-threaded *vvadd* microbenchmark. We use calls to `ece4750_stats_on` and `ece4750_stats_off` to indicate the region of interest (ROI) where we want to collect statistics. Even though this is an evaluation program, we still verify that the results are correct.

You should always start by making sure your evaluation program works natively before running the evaluation program on a FL or RTL simulator. You can compile and run the evaluation for the single-threaded vector-vector add microbenchmark natively using the provided build system.

```
% cd ${HOME}/ece4750/app/build-native
```

```
% make ubmark-vvadd-eval
% ./ubmark-vvadd-eval
```

Once your evaluation works natively, you can also cross-compile and run the evaluation on the single-core system FL simulator.

```
% cd ${HOME}/ece4750/app/build
% make ubmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim ./ubmark-vvadd-eval
```

One your evaluation works natively and on the FL simulator, then you can run the evaluation on the RTL implementation of the single-core system.

```
% cd ${HOME}/ece4750/app/build
% make ubmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim --impl base ./ubmark-vvadd-eval
```

You can use the `-stats` option to print out the number of cycles, number of instructions, cycles/instruction, number of cache accesses, and cache miss rate. You can use the `-trace` to print out a line trace. Consider writing long line traces to a file to make them easier to analyze.

```
% cd ${HOME}/ece4750/app/build
% make ubmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim --stats --impl base ./ubmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim --trace --impl base ./ubmark-vvadd-eval > ./trace.txt
```

5.2. Multi-Core System Evaluation

We provide you with four multi-threaded microbenchmarks which are the parallel versions of the single-threaded microbenchmarks: *vvadd*, *mfilt*, *cmult*, and *bsearch*. Your sorting algorithm makes a fifth microbenchmark. All single-threaded microbenchmarks are in the `app/mtbmark` directory with their C source code, test programs, and evaluation programs. We use the same dataset files from the single-threaded microbenchmarks. All of these multi-threaded microbenchmarks statically partition the input for the number of cores (threads) that are available in the system. All of these multi-threaded microbenchmarks support running on a single-core system and also support using a single worker.

You should always start by making sure your evaluation program works natively before running the evaluation program on a FL or RTL simulator. You can compile and run the evaluation for multi-threaded vector-vector add natively using the provided build system.

```
% cd ${HOME}/ece4750/app/build-native
% make mtbmark-vvadd-eval
% ./mtbmark-vvadd-eval
```

Once your evaluation works natively, you can also cross-compile and run the evaluation on the single-core system FL simulator.

```
% cd ${HOME}/ece4750/app/build
% make mtbmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim ./mtbmark-vvadd-eval
```

One your tests pass natively and on the single-core system FL simulator, then you can run the tests on multi-core system FL simulator.

```
% cd ${HOME}/ece4750/app/build
% make mtbmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim --impl mcore-fl ./mtbmark-vvadd-eval
```

One your evaluation works natively and on the FL simulator, then you can run the evaluation on the RTL implementation of the multi-core system.

```
% cd ${HOME}/ece4750/app/build
% make ubmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim --impl alt ./ubmark-vvadd-eval
```

Once all of this works you can use the `-stats` option to print out the number of cycles, number of instructions, cycles/instruction, number of cache accesses, and cache miss rate.

5.3. Comparative Evaluation

Consider using the following five experiments:

- Run the single-threaded sorting microbenchmark on the single-core system to measure the baseline design performance
- Run the single-threaded sorting microbenchmark on the multi-core system to measure the hardware overhead of the multi-core system
- Run the multi-threaded sorting microbenchmark on the single-core system to measure the software overhead of a multi-threaded program
- Run the multi-threaded sorting microbenchmark on the multi-core system using a single worker to measure the software and hardware overhead of a multi-threaded program
- Run the multi-threaded sorting microbenchmark on the multi-core system using all four workers to measure the alternative design performance

Here are the commands that implement the above five experiments.

```
% cd ${HOME}/ece4750/app/build
% make ubmark-vvadd-eval
% make mtbmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim --stats --impl base ./ubmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim --stats --impl alt  ./ubmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim --stats --impl base ./mtbmark-vvadd-eval
% ../../sim/lab4_sys/sys-sim --stats --impl alt  ./mtbmark-vvadd-eval 1
% ../../sim/lab4_sys/sys-sim --stats --impl alt  ./mtbmark-vvadd-eval
```

By default, multi-threaded evaluation programs use as many workers as cores in the system. To indicate we only want to use a single worker we pass in `1` as a command line argument to the multi-threaded evaluation program. Run these five experiments for all five microbenchmarks for a total of 25 experiments. For each experiment record the execution time in cycles and the total CPI. Create two bar plots; the first bar plot should plot the speedup for each experiment normalized to the baseline results; the second bar plot should plot the CPI for each experiment. Group the bar plots by microbenchmark. This data will enable quantifying the software and hardware overheads of the

multi-core system, and ultimately whether or not the complete alternative design is able to overcome these overheads to enable speedup over the baseline design.

6. Hardware and Software Optimizations

Advanced students might want to consider modifying the software and/or hardware to improve the performance of their sorting microbenchmarks. A *small* bonus will be given to the fastest few designs. We will run your sorting microbenchmark on your multicore system using one or more of our own private datasets. These datasets will be similar in size to the dataset we give you, but may not necessarily have exactly 100 elements. 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. The rest of this section includes some ideas on how you might want to improve your design.

You will quickly find that the performance of your single-core system and even the multi-core system is limited by the hit latency of the cache you designed in Lab 3. While we could move to a more aggressive pipelined cache microarchitecture, you can achieve much of the same benefit by simply merging states in the FSM control unit. Ideally, you would merge enough states to enable a single-cycle hit latency for reads (i.e., a single state for read hits) and a sustained throughput of one read hit per cycle. This requires performing tag check and data access in parallel, and carefully handling the val/rdy signals for the cache request and response interfaces. Writes can potentially use two states to do tag check and data access in sequence, although single-cycle hit latency for writes is still possible if the cache response is sent back in the first state. To enable full throughput, the cache should only go into the WAIT state if cache response interface is not ready, and if there is a new valid cache request then the cache should avoid going back into the IDLE state and instead go straight to the tag check state. performance. Reducing the read hit latency is the most critical since this would improve the performance of instruction fetch in your processor.

You might also notice that you are wasting some cycles due to control hazards. Our sorting microbenchmark will have one or more loops, and the pipelined processor will almost always mispredict the backwards branch used in these loops. Adding a simple branch target buffer (BTB) in the F stage could improve the performance of the provided microbenchmarks, and the sorting implementation, by effectively eliminating most squashes due to the backwards branch used in loops. A simple, yet effective approach would be to include a four entry BTB in the F stage. Each entry would include a valid bit, the PC of the branch, and the target address for the branch when it is taken. In the F stage, your processor would need to search the BTB for the current PC. If there is a hit, then the F stage can use the corresponding target address in the BTB. If there is a miss, then the F stage can simply use PC+4. If a branch is taken, then in the X stage we would need to write the BTB with the corresponding PC of the branch and target address. More complicated schemes are certainly possible.

The above optimizations will improve the performance of both the single-core and multi-core system. You may also want to experiment with the ring network. Consider using a more optimal network routing algorithm and/or using a more fair arbitration scheme. You will want to think critically about whether or not a bus or a ring makes the most sense for data cache network, and then run some experiments to see which achieves the best performance for a certain network composition. Some extra performance hooks may help you understand the injection rate/latency in a realistic multi-core system context. If you choose to use a bus, then should probably justify why a bus will not significantly impact the cycle time.

Finally, do not forget the importance of optimizing your software! Not all sorting algorithms in the same complexity class are equivalent. The constant factors and trailing terms can significantly impact real system performance. Experiment with different algorithms in the same complexity class. Use a hybrid sort to optimize the base case in a recursive algorithm. Try to optimize your C code to avoid unnecessary memory accesses or to manually unroll loops. You may also want to experiment with different ways to parallelize your sorting algorithm.

Acknowledgments

This lab was created by Shunning Jiang, Shuang Chen, Ian Thompson, Megan Leszczynski, Gaurab Bhattacharya, Moyang Wang, Christopher Torng, Berkin Ilbeyi, Christopher Batten, and Ji Kim as part of the course ECE 4750 Computer Architecture at Cornell University.