ECE 4750 Computer Architecture, Fall 2020
Lab 4: Multicore Processor

School of Electrical and Computer Engineering
Cornell University

revision: 2020-11-13-00-04

"Let us open the Pandora's box of parallel programming."

In this lab, you will be composing the processor, cache, and network you have designed throughout the semester to create the baseline design (a single-core processor with its own instruction and data cache) and the alternative design (a simple multicore processor with private instruction caches and a shared, banked data cache). You will also have a chance to write a single-threaded and multi-threaded sorting microbenchmark in C, explore the compiled and assembled binary, and run these programs on the baseline and alternative designs. 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. **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.**

Some unique features of this lab assignment include its emphasis on structural composition to incrementally create a relatively complex system based on thoroughly unit-tested subsystems, and its emphasis on software/hardware co-design such that students must understand the software application, hardware/software interface, and hardware microarchitecture. You will be using various incremental compositions to: (1) test composing the network we provide you with, with special adapters to build memory networks suitable for sending memory requests and receiving memory responses; (2) test composing the memory networks and the cache from Lab 3 to build a shared, banked data cache; and (3) test the entire single core and multicore systems. You will be using the standard RISC-V cross-compiler to compile C programs down to a RISC-V binary suitable for execution on our RISC-V simulators. You will be writing your own single-threaded quicksort microbenchmark, as well as a multi-threaded sorting microbenchmark that uses an algorithm of your choice. You will have an opportunity to improve the performance of your final multi-threaded sorting microbenchmark through either software or hardware improvements.

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;
- software/hardware co-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 and the lab assessment rubric. To get started, you should access the ECE computing resources and you have should have used the `ece4750-lab-admin` script to create or join a GitHub group. If you have not do so already, 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.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}/ece4750/lab-groupXX
% git pull --rebase
% mkdir -p sim/build
% cd sim/build
% py.test ../lab5_mcore
```

Since we provide you the single core composition, the memory network compositions, and the shared, banked data cache composition, many of the tests may pass if the subsystems from your previous labs are completely functional. For this lab, you will be working in the `lab5_mcore` subproject which includes the following files:

• `SingleCorePRTL.py`	PyMTL single-core composition
• `SingleCoreRTL.py`	Verilog single-core composition
• `SingleCoreVRTL.v`	Wrapper to choose which RTL language
• `MsgAdapters.py`	PyMTL adapters to convert between MemMsg and NetMsg
• `MsgAdapters.v`	Verilog adapters to convert between MemMsg and NetMsg
• `CacheNetPRTL.py`	PyMTL CacheNet module
• `CacheNetVRTL.v`	Verilog CacheNet module
• `CacheNetRTL.py`	Wrapper to choose which RTL language
• `MemNetPRTL.py`	PyMTL MemNet module
• `MemNetVRTL.v`	Verilog MemNet module
• `MemNetRTL.py`	Wrapper to choose which RTL language
• `McoreDataCachePRTL.py`	PyMTL data cache system composition
• `McoreDataCacheVRTL.v`	Verilog data cache system composition
• `McoreDataCacheRTL.py`	Wrapper to choose which RTL language
• `MultiCorePRTL.py`	PyMTL multi-core composition
• `MultiCoreVRTL.v`	Verilog multi-core composition
• `MultiCoreRTL.py`	Wrapper to choose which RTL language
• `elf.py`	Utils for reading/writing ELF binary files
• `isa-sim`	ISA simulator for testing programs
• `score-sim`	Single-core simulator for evaluation
• `mcore-sim`	Multicore simulator for evaluation
• `__init__.py`	Package setup
• `test/TestNetCacheSink.py`	Yet another special test sink
• `test/CacheNetRTL_test.py`	Test for CacheNet
• `test/MemNetRTL_test.py`	Test for MemNet
• `test/McoreDataCacheRTL_test.py`	Test for data cache system

• `test/harnesses.py`	Test and simulation harnesses
• `test/inst_csr.py`	CSR tests for multi-core
• `test/inst_jal.py`	Staff `jal` tests
• `test/inst_mul.py`	Staff `mul` tests
• `test/inst_sw.py`	Simple `sw` tests for multi-core
• `test/SingleCoreRTL_test.py`	Test for single-core composition
• `test/MultiCoreRTL_test.py`	Test for multi-core composition
• `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× 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 with its own instruction and data cache, and a quad-core system with private instruction caches and a shared, banked data cache.

We will reuse the assembly tests for a representative subset of instructions (since the simulation time is increased due to more complex models) from Lab 2 to test the single-core system. To test the multi-core composition, we introduce an extension to the assembly test methodology from Lab 2 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: `CacheNet`, `MemNet`, and `McoreDataCache`. We unit test them, and reuse them to build larger components.

Unlike previous labs, this lab has significant portion of software. We organize files as follows: `lab-groupXX` is the top level lab repo you have been using in this course, `lab-groupXX/sim` contains PyMTL or Verilog designs and unit tests. `lab-groupXX/app` is the microbenchmark directory you will be working in for the software side of this lab. We will be using the 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. Here is how we build applications:

```
% cd ${HOME}/ece4750/lab-groupXX
% mkdir app/build
% cd app/build
% ../configure --host=riscv32-unknown-elf
% make
```

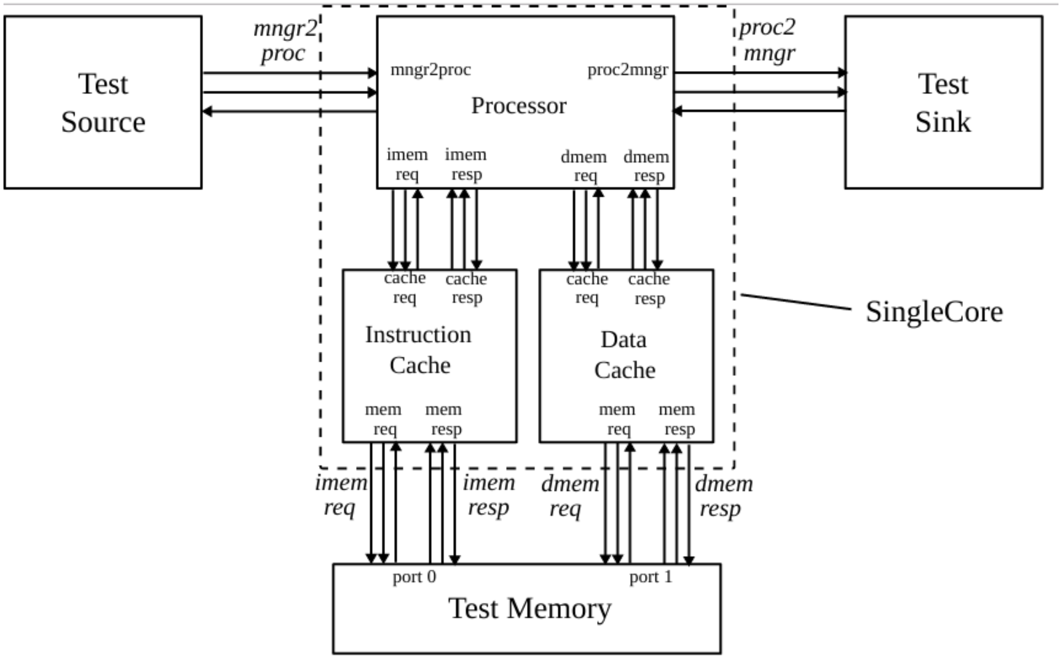

Figure 1: SingleCore – The SingleCore module as a whole, is hooked up to the test source, test sink, and test memory for testing and evaluation. Each bundle of the three arrows is a msg/val/rdy port bundle. The ports with inclined names are the top-level ports that SingleCore expose.

2. Baseline Design

For the baseline design, we are going to compose the processor from Lab 2 and two caches from Lab 3 (one as the instruction cache, and the other as the data cache). This will give you a simple yet realistic single-core system as the hardware part of the baseline. The software counterpart is a scalar quicksort program.

We have provided the composition in SingleCorePRTL.py/SingleCoreVRTL.v. Note that this design does not need a network. Although we give this file to you, you need to look into and understand the composition and the way we count cache miss/access statistics. We set num_banks=0 for both caches since we are only in a single-core configuration. We connect 0 to core_id port and we set num_cores=1 for the processor. These two parameters will be very important in your multicore processor, and are explained in depth in the next section.

The composition and the connections of the single core system will look like the 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/sp ports as shown. Note that the data bitwidth from processor to caches is 32 bits, while the data bitwidth from the caches to the test memory is the full cache line, which is 128 bits in this design.

As the software part of the baseline, you are required to implement scalar quicksort algorithm in C in the quicksort_scalar() function of app/ubmark/ubmark-quicksort.c. Currently, this function only has a template which copies the source array to destination, and your task is to sort the src array using the quicksort algorithm and write the result to destination. This file also contains verification logic to ensure the correctness of your algorithm implementation. You are welcome to consult

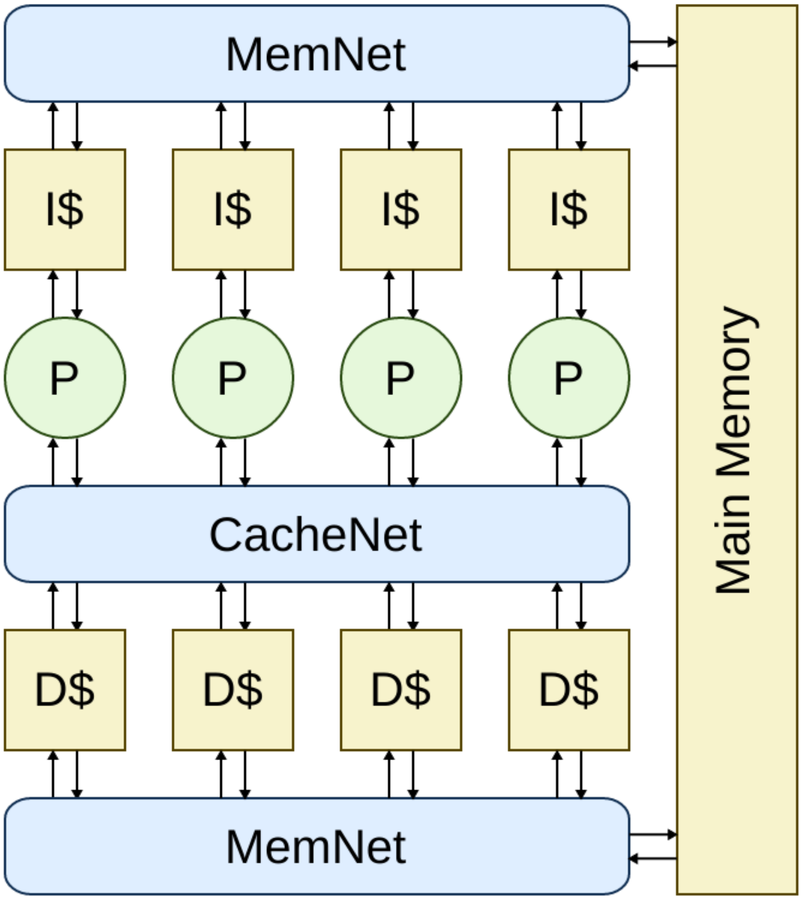

Figure 2: Alternative design block diagram – The alternative design consists of four processors, four private I-caches, a four-banked shared D-cache, and several networks to route dmem request/response from the processors and the D-cache, and refilling both the I-cache and the D-cache. Note that each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

textbooks or online resources to learn more about quicksort, but you should cite these resources. Copying code is not allowed; you should write it on your own.

3. Alternative Design

The alternative design is to implement the multicore system and run a parallel sorting implementation on it. The multicore system consists of four processors, four instruction caches (one private instruction cache per processor), four data cache banks that are shared across all four cores, and four-port networks to interconnect the subsystems. You are free to choose which design in the previous labs to use; while there is always a trade-off between the baseline design and the alternative design, your goal should be to achieve the highest performance possible. Choosing an appropriate network for the cache net is particularly important.

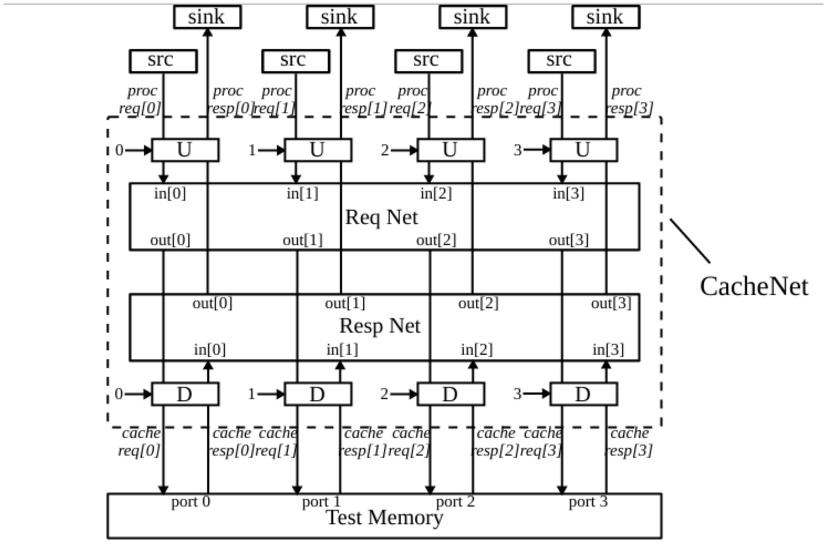

Figure 3: CacheNet – The CacheNet module as a whole, is hooked up to the test sources, test sinks, and four-port test memory for unit testing. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. Each "U" and "D" is an adapter which takes care of both the request port and the response port, and has its own adapter id. The ports with inclined names are the top-level ports that CacheNet exposes.

Figure 2 illustrates the system structure at a high level. Keep in mind that we will take an incremental design approach to build the whole system. We have provided you with the following sub-compositions with unit tests.

- `CacheNet`: the request/response network pair connecting four processors and four cache banks.
- `MemNet`: the request/response network pair connecting four caches and one main memory port.
- `McoreDataCache`: `CacheNet` + four cache banks + `MemNet` = the whole data cache subsystem

3.1. CacheNet: The Network Between Processors and Data Cache Banks

The most confusion often comes from this question: how do we actually compose the cache with the network? Figure 3 shows the diagram of the `CacheNet` 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 `UpstreamMsgAdapter`, where "D" corresponds to `DownstreamMsgAdapter`. The reason why we call them "upstream" and "downstream" is the following. Intuitively, the processor is the upstream, and the main memory (here it's cache) is the downstream which passively accepts memory request and emits memory response. As a result, the `UpstreamMsgAdapter` is meant to be put at the processor side of the network, and vice versa.

In general, the `UpstreamMsgAdapter` has two functions, one for the request network and the other for the response network:

- Accept a memory request and turn it into a network message whose payload is the request.
- Turn a network message back to a memory response by just throwing away the header.

In contrast, the `DownstreamMsgAdapter` has the following two functions:

31 10	9 6	5 4	3 0
tag	index	bank	offset

Figure 4: 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.

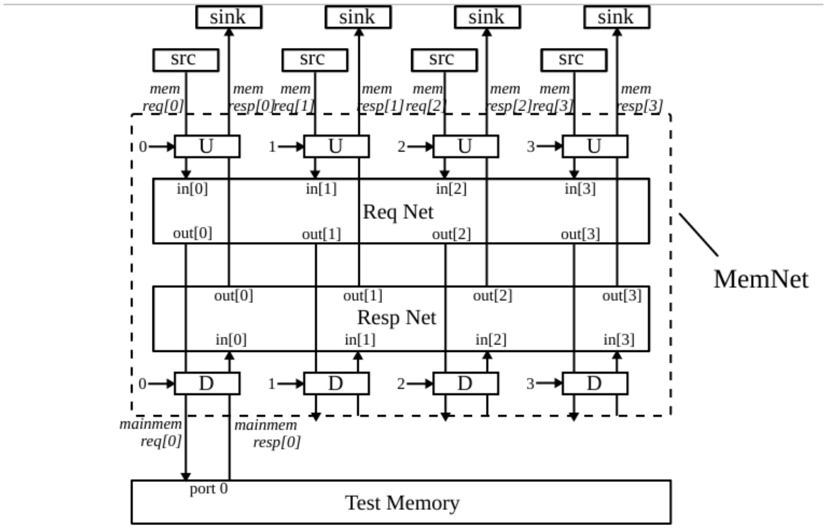

Figure 5: MemNet – The `MemNet` module as a whole, is hooked up to the test sources, test sinks, and single-port test memory for unit testing. Each arrow is actually a msg/val/rdy port bundle whose direction is the msg/val direction. Each "U" and "D" is an adapter which takes care of both the request port and the response port, and has its own adapter id. The ports with inclined names are the top-level ports that `MemNet` exposes.

- Turn a network message back to a memory request by throwing away the header.
- Accept a memory response and turn it into a network message whose payload is the response.

You may ask where the source and the destination in the header are from when `UpstreamMsgAdapter` turns a memory request into a network message. Recall from Lab 3 that in a multi-banked cache design, we use a set of "bank bits" so that cache lines are interleaved across different cache banks. Figure 4 illustrates which bits are used for the four-banked data cache. In `CacheNet`, the `UpstreamMsgAdapter` extracts this bank index from the address of the memory message from processor and uses it as the destination field in the header (the source field is basically the processor/adapter id). Also, don't forget that the corresponding cache response generated by a cache bank needs to be sent back to the correct processor. To determine who had sent the cache request originally, we use some high bits of the opaque field in the memory message to store the processor/adapter id. The cache and the memory system are required to preserve the opaque field throughout the memory system. When the `DownstreamMsgAdapter` gets a cache response, it will extract the destination adapter/processor id from the opaque field, possibly restore the previous opaque field value, and then send the cache response into the response network.

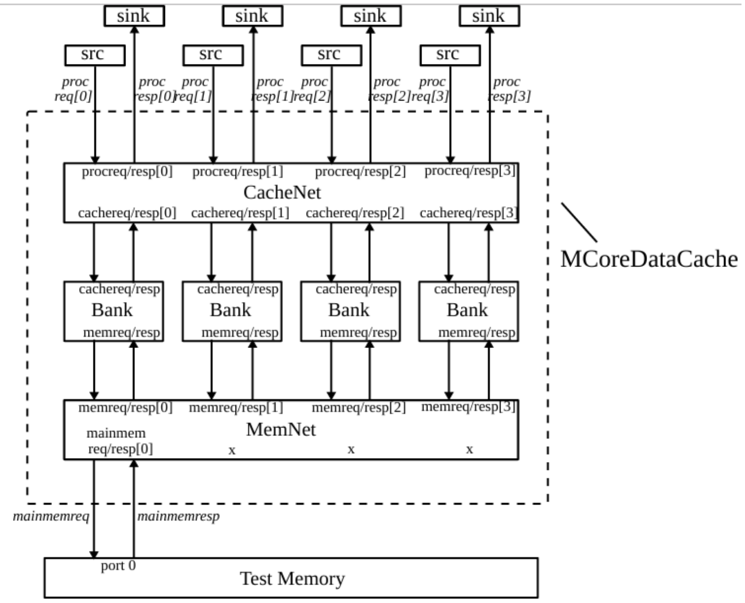

Figure 6: MCoreDataCache – The `McoreDataCache` module as a whole, is hooked up to the test sources, test sinks, and single-port test memory for unit testing. Each arrow is actually a msg/-val/rdy port bundle whose direction is the msg/val direction. We omit the details inside `MemNet` and `CacheNet` but show the port names to match the previous diagrams. The ports with inclined names are the top-level ports that this module exposes.

3.2. MemNet: The Network Between Caches and Main Memory

Unlike the `CacheNet`, the `MemNet` only uses one output terminal, but has four input terminals. Figure 5 shows the diagram of the `MemNet`. In the `MemNet`, we use a different mode for the `UpstreamMsgAdapter` which always inserts `0` into the destination field. The rest of the module works in the same way as `CacheNet`. We can use the `MemNet` as the refill networks for the instruction caches and the data-cache banks, since four caches/banks need to send requests to one main memory port.

3.3. McoreDataCache: The Data Cache Subsystem

We compose the `CacheNet`, `MemNet`, and four cache banks to create the complete shared, banked data-cache subsystem. We set `num_banks=4` for each data cache to ensure they correctly handle the bank bits. Figure 6 shows the diagram of `McoreDataCache` module.

3.4. MultiCore: The Multicore Processor System

Note that we provide you the `CacheNet`, `MemNet`, and `McoreDataCache` compositions. You are responsible for creating the final composition with four processors, four private instruction caches, and the shared, banked data-cache subsystem to create the complete multicore system (see Figure 7). We instantiate four processors with `num_cores=4` and connect 0, 1, 2, 3 respectively to the `core_id` port of each processor. The details of how we use these two parameters is discussed in the next subsection. We set `num_banks=0` for each of the instruction cache since they are private caches and not banked.

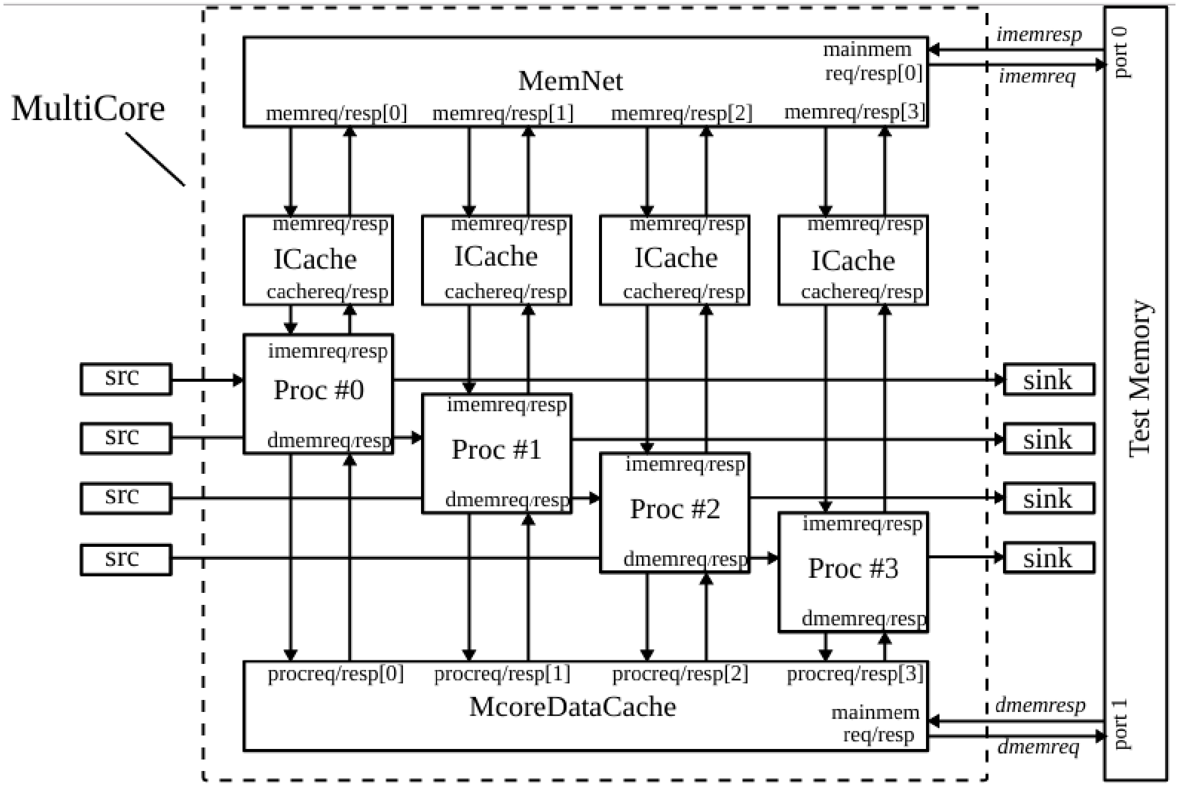

Figure 7: MultiCore – The `MultiCore` module as a whole, is hooked up to a dual-port test memory, four test sources and four test sinks. Note that to not to make the diagram too crowded, we omit the *proc2mngr* and *mngr2proc* port names both at the boundary of `Multicore` as well as in the processor. 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.

3.5. Software Side of Things

"Here Be Dragons!"

In addition to building the multicore composition, you also need to write a parallel sorting program. Before we jump into the software part, let's demystify the `csrr` and `csrw` instructions whose implementations have already been provided to you in Lab 2. Here we explain the `num_cores` and `core_id` control/status registers (CSRs) which are very important for this section, and in Section 5 we will explain the `stats_en` CSR. See the TinyRV ISA manual for more details `http://www.csl.cornell.edu/courses/ece4750/handouts/ece4750-tinyrv-isa.txt`.

- By executing `csrr x1, 0xF14`, the processor puts the core id into register `x1`, and then one core can differentiate itself from the others. For example, when we want processor 0 to process `a[0]`, processor 1 to process `a[1]`, processor 2 to process `a[2]`, and processor 3 to process `a[3]`, we can just write `a[core_id]` in the program after we retrieve `core_id`. Similarly, to ensure only one of the processors is executing a serial part of the code, in the program we can compare the core id to 0, and only then execute the given code block.
- By executing `csrr x1, 0xFC1`, the processor puts the number of cores into register `x1`. This is very useful when we want to distribute the work evenly across multiple cores. For instance,

to parallelize work on an array by dividing and assigning a part of this array to each core, the program can then divide the total number of elements by the number of cores, to get the block size for each core.

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. It can be quite challenging to parallelize quicksort directly. We recommend you implement a bottom-up hybrid mergesort/quicksort algorithm. Divide the input array into four blocks. Each core can use the serial quicksort algorithm 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.

We provide you a very light-weight threading library, called `bthread`. Note that we use thread and core interchangeably in this lab since each core has only one thread. We call core 0 the master core and the other cores the worker cores. Figure 8 illustrates how to use the `bthread` library to implement a parallel vector-vector add microbenchmark that makes use of two cores. The `bthread_init` function sets up the library and must be called at the beginning of the main function. All four cores enter the `bthread_init` function; core 0 will exit the function while the remaining three worker cores will iterate in a worker loop waiting for work. The master core can spawn work onto a worker core by using the `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 `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/common/bthread.cc`

A simple way to use the `bthread` library to implement the hybrid mergesort/quicksort mentioned above is to divide the input array into four blocks, create four argument structures, use a `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.

Besides, we provide you with some additional features to facilitate debugging. Similar to what normal `printf`, `putchar` and `puts` in C does, you can use `wprint` and `wprintf` function as the following examples to print char, string, or number to the terminal. The reason why we have to put "L" before the string literal is because our TinyRV2 processor only support full-word `lw/sw`. As a result we have to use wchar_t which brings the "L" prefix. **Note that the simulator only looks at Core 0's wprint(f) messages.**

```
wprint( L'a'); wprint( L'b'); wprint( L'c');
wprint( L"abc" );
wprintf( L"number=%d, char=%C, string=%S", 42, L'a', L"abc" ) # capital C and S
```

4. Testing strategy

Since this lab has an emphasis on both composition on the hardware side and parallel programming on the software side, we have provided you with basic tests and random tests to test all of the incremental compositions in the alternative design, and instruction tests for both the single- and multi-core systems.

```c
typedef struct {
  int* dest;  // pointer to dest array
  int* src0;  // pointer to src0 array
  int* src1;  // pointer to src1 array
  int  begin; // first element this core should process
  int  end;   // (one past) last element this core should process
} arg_t;

void vvadd_mt( 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* dest  = arg_ptr->dest;
  int* src0  = arg_ptr->src0;
  int* src1  = arg_ptr->src1;
  int  begin = arg_ptr->begin;
  int  end   = arg_ptr->end;

  // Do the actual work.
  for ( int i = begin; i < end; i++ )
    dest[i] = src0[i] + src1[i];
}

int main( int argc, char* argv[] )
{
  // Initialize bare threads (bthread).
  bthread_init();

  // This array will be where the results are stored.
  int dest[size];

  // Start counting stats.
  test_stats_on();

  // Create two argument structures that include the array pointers and
  // what elements each core should process.
  arg_t arg0 = { dest, src0, src1, 0,      size/2 };
  arg_t arg1 = { dest, src0, src1, size/2, size   };

  // Spawn work onto core 1
  bthread_spawn( 1, &vvadd_mt, &arg1 );

  // Have core 0 also do some work.
  vvadd_mt( &arg0 );

  // Wait for core 1 to finish.
  bthread_join(1);

  // Stop counting stats
  test_stats_off();

  // Core 0 will verify the results.
  if ( bthread_get_core_id() == 0 )
    verify_results( dest, ref, size );

  return 0;
}
```

Figure 8: Simple Parallel Vector-Vector Add Microbenchmark for Two Cores

To test the single-core system with one instruction cache and one data cache, we will reuse the assembly tests for a representative subset of the instructions in Lab 2. We have already done extensive unit testing in Lab 2, so our goal in Lab 4 is to focus on those tests that might expose bugs in the single- and multi-core compositions. We test the following instructions.

- CSR	: `csrr, csrw`
- Reg-Reg	: `add, mul`
- Reg-Imm	: `addi`
- Memory	: `lw, sw`
- Jump	: `jal`
- Branch	: `bne`

As explained in Section 3, we incrementally build compositions, unit test them, and reuse them to build larger components such as the `McoreDataCache` and the `MultiCore`.

We reuse the test cases in Lab 3 to test the `CacheNet` and the `McoreDataCache` since the input messages of these two modules are basically cache requests. The difference is that now we have four sources and four sinks which means that we will put requests/responses in each of the test source/sink pair, and we have to use `TestNetCacheSink` to allow both out-of-order delivery and ignoring the cache test bits. The `MemNet` tests reuse the messages of `TestMemory` tests since the input of `MemNet` is essentially memory requests.

We also test the multicore composition using the same subset of instructions, except for `sw` and `csr`. In `sw` test cases there might be data races when different cores are trying to store/load data to the same address. Depending on the each core's progress, a core might read the data that has already been overwritten. For `csr` tests, all four cores have different `core_ids`. Thus, we introduce an extension to our assembly test syntax which enables initializing multiple test sources and sinks. Here's an example of using the extended syntax.

```
# Example of using a list of values
csrr x2, mngr2proc < {1,3,5,7}
csrw proc2mngr, x2 > {1,3,5,7}

# Check numcores/coreid
csrr x2, numcores
csrw proc2mngr, x2 > 4
csrr x2, coreid
csrw proc2mngr, x2 > {0,1,2,3}
```

Each core has its own test source/sink, so the first line initializes all four tests sinks with four different values. Here are the commands to run the tests.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% py.test ../lab5_mcore/
% py.test ../lab5_mcore/test/SingleCoreRTL_test.py
% py.test ../lab5_mcore/test/CacheNetRTL_test.py
% py.test ../lab5_mcore/test/MemNetRTL_test.py
% py.test ../lab5_mcore/test/McoreDataCacheRTL_test.py
% py.test ../lab5_mcore/test/MultiCoreRTL_test.py
```

For the testing strategy, unlike the earlier labs, you are not required to come up with additional test cases. However, if you find that the provided test cases fail to capture a bug, you are encouraged to design additional test cases, put them in the test files, and discuss them in your report. You can look into these test cases from previous labs and the PyMTL repo for inspiration.

Since this lab has an emphasis on both software and hardware, you *definitely need to develop a testing strategy for your single- and multi-threaded sorting microbenchmarks!*. We recommend testing that your single and multi-core sorting microbenchmarks work on arrays with different values and different sizes.

Your report should briefly summarize the testing strategy used to test each subsystem in the first four labs, and also summarize the testing strategy that we provide you in this lab. You should go into some nice detail on how we do the incremental testing, the single-core testing, and the multicore testing in this lab. Make sure you include at least a paragraph about your software testing strategy for the single- and multi-threaded sorting microbenchmarks. Discuss how all of this comes together to ensure we have a very sophisticated working design that can execute multi-threaded programs with realistic caches and networks.

5. Evaluation

We provide you with four single-threaded micro-benchmarks: *vvadd*, *mfilt*, *cmult*, and *bsearch*, whose assembly sequences have been studied in Lab 2. All single-threaded microbenchmarks are in the `app/ubmark` directory with their C source code and the dataset files. Take a look at all of them and familiarize yourself with what each one does.

In addition to the single-threaded benchmarks, we have provided multi-threaded benchmarks in the `app/mtbmark` directory, which are the parallel version of the same microbenchmarks: *vvadd*, *mfilt*, *cmult*, and *bsearch*. All of these multi-threaded microbenchmarks statically partition the input for the number of cores (threads) that are available in the system.

5.1. Compile a Program and Investigate the Assembly Code

As the first step, we recommend you to compile microbenchmarks natively on x86/x64, instead of debugging them on your processor directly. We leverage a modular C++ build system to provide one-stop compilation solution in this lab, and hence compiling your program natively is very similar to compiling it for RISC-V. The only difference is the target host of the configure command, which by default is x86/x64. Thus we don't need to specify it for native compilation.

Below are the commands to build the programs natively. Note that we *always* create a temporary build directory to avoid cluttering up the source directories. We create one build directory for native builds and a different build directory for RISC-V builds. Also note that You can also type `make ubmark-vvadd` to compile a specific benchmark.

```
% cd ${HOME}/ece4750/lab-groupXX/app
% mkdir build-native
% cd build-native
% ../configure
% make
% make ubmark-vvadd
```

When you run your binaries natively, it will tell you if it passed or failed the verification. If you need further debugging, you can add `printf` statements, or use a debugger such as `gdb`. When you get your app working, make sure you remove any `printf` statements because this is not supported on our TinyRV2 architecture.

After the native build succeeds with no error and hopefully passes the verification, you can compile the microbenchmarks for the TinyRV2 processor. The only difference is to specify the target to RISC-V by setting host to riscv32-unknown-elf. We usually term this compilation process as "cross-compilation".

```
  % cd ${HOME}/ece4750/lab-groupXX/app
  % mkdir -p build
  % cd build
  % ../configure --host=riscv32-unknown-elf
  % make
  % make ubmark-vvadd
```

After the cross-compilation finishes, you can use `objdump` in the gcc toolchain to see the compiled assembly. Specifically, we have to use `riscv32-unknown-elf-objdump` which is part of the RISC-V gcc toolchain. We have provided an alias for you to avoid typing these objdump options every time. You are also encouraged to change options and search online to learn about `objdump`. Here we use *vvadd* as an example, and we assume we have already built the microbenchmarks using the above commands.

```
 % riscv32-unknown-elf-objdump -dC --no-show-raw-insn -Mno-aliases -Mnumeric ubmark-vvadd
...
00000248 <vvadd_scalar(int*, int*, int*, int)>:
 248:   bge     x0,x13,274 <vvadd_scalar(int*, int*, int*, int)+0x2c>
 24c:   slli    x13,x13,0x2
 250:   add     x13,x11,x13
 254:   lw      x15,0(x11)
 258:   lw      x14,0(x12)
 25c:   addi    x11,x11,4
 260:   addi    x12,x12,4
 264:   add     x15,x15,x14
 268:   sw      x15,0(x10)
 26c:   addi    x10,x10,4
 270:   bne     x11,x13,254 <vvadd_scalar(int*, int*, int*, int)+0xc>}
 274:   jalr    x0,x1,0
...
 % riscv32-objdump ubmark-vvadd  # this command is an alias of the above command
```

Look familiar? Yes, the code sequence from pc=0x254 to pc=0x270 is almost identical to the vector-vector add code snippet we have seen throughout the course.

5.2. Run a Cross-Compiled Program

After making sure your application works natively, the second step is to run your program on the single-core or multi-core FL model. An ISA simulator, called `isa-sim`, supports both single- and multi-core mode, and it will serve as the FL model in this lab. This simulator simply instantiates one or four `ProcFL`s from Lab 2 based on the command line options.

It is important to debug your application on the ISA simulator before moving onto RTL. By default the ISA simulator is single-core. The `--mcore` command line option makes the ISA simulator simulate a quad core. Here are the commands to run apps on the ISA simulator. Note that you can add `--trace` option to turn on line tracing.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/
% mkdir -p build
% cd build
% ../lab5_mcore/isa-sim ../../app/ubmark-vvadd

    [ passed ]

  In stats_en region:
    total_committed_inst   = 811

% ../lab5_mcore/isa-sim --mcore --trace ../../app/mtbmark-vvadd
  ...

    [ passed ]

  In stats_en region:
    total_committed_inst   = 1400

    core0_committed_inst   = 341
    core1_committed_inst   = 353
    core2_committed_inst   = 353
    core3_committed_inst   = 353
```

Here is one more recipe of `isa-sim`. Recall that in the iron law of processor performance, there is a term called instruction per program. Although the ISA simulator doesn't know the cycle count, it can provide you with the number of committed instructions.

The most exciting step is to execute the native-tested and isa-simulated program on our `SingleCore` or `MultiCore` system. We have provided you with two simulators, `score-sim` and `mcore-sim` to simulate these two systems respectively. You can run the *vvadd* microbenchmark using the following two simulation commands:

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% ../lab5_mcore/score-sim --stats ../../app/build/ubmark-vvadd
% ../lab5_mcore/mcore-sim --stats ../../app/build/mtbmark-vvadd
```

The `--stats` option dumps all statistics during the `stats_en` region (explained in the next subsection). The `--trace` displays the line tracing. The `--dump-vcd` enables `.vcd` waveform generation.

When you run a simulation, the simulator will also report whether the verification passed or failed. Make sure all of the verifications pass. If not, please go back to our systematic debugging strategy to isolate and resolve the error using test cases instead of debugging directly on simulator.

5.3. The Real Evaluation: Data and Statistics

Before we talk about statistics, let's go back and think critically about what the performance means for our system with respect to a real binary executable file. We have done evaluations for Lab 1-4 separately, so we might ask the following questions.

What is the difference between a compiled binary, and the assembly sequences we used in Lab 2?

A typical compiled program includes a lot of code to bootstrap, manage the stack, various library calls, etc. The code that we are interested in might be masked out by the uninteresting parts of this boilerplate code, so we only enable stats collection when we are actually in the piece of code of interest. To do so, we need to know about one more CSR.

- By executing `addi x1, x0, {1 or 0}; csrr 0x7C1, x1`, the processor set a special "stats" bit to high or low, to tell the simulator if the program has entered the "statistics enabled" region, which we usually abbreviate as the "`stats_en` region".

From the above example of using `objdump`, we can see that the `vvadd-unopt` assembly sequence in Lab 2 actually corresponds to a subset of the `stats_en` region.

OK, we agree to focus on the `stats_en` region only, but what is our goal?

In this lab, we want to finish the `stats_en` region as soon as possible which means that the cycle count is the ultimate metric we want to optimize for. Notice that when we have multiple cores, CPI might not be suitable because we need the number of actually useful instructions to do fair comparison, which is very hard under most parallel programming frameworks involving busy-waiting, including our `bthread` library.

Fine, what else should we look at besides cycle count?

We have evaluated each component we use in the multicore composition separately using different metrics: CPI for processor, average memory access latency and miss rate for caches, and network latency vs. offered bandwidth for networks. We have already provided you some statistics in the simulators. The simulators will report the correct cycle count. `score` sim will report the number of committed instructions, CPI, and the number of instruction/data cache misses/accesses/missrate.

As mentioned in Section 5.2, before you jump into the detailed statistics in `score-sim` and `mcore-sim`, you may need to first look at the committed instruction count in ISA simulator to get a feel for how good/bad your program is. For example, no matter how hard you optimize your system in this lab, it is impossible to finish one million instructions in 10,000 cycles.

```
% cd ${HOME}/ece4750/lab-groupXX/sim/build
% ../lab5_mcore/score-sim --stats ../../app/build/ubmark-vvadd

  [ passed ]

In stats_en region:

  num_cycles            = 4311
  total_committed_inst  = 811
  total_cpi             = 5.32

  total_icache_miss     = 5
  total_icache_access   = 912
  total_icache_miss_rate = 0.0055

  total_dcache_miss     = 76
  total_dcache_access   = 300
  total_dcache_miss_rate = 0.2533
```

We leave it up to you to add per cache access/miss/missrate for instruction caches and hook them up in `MultiCore`. You also need to hook up the statistics in `McoreDataCache` to ports in `MultiCore`. We already expose some output ports (hooks) to the simulator as shown below. A 4-bit port concatenates the four status bits from the four processors/caches/banks. For example, `icache_access[0]` indicates whether the first instruction cache has a handshake transaction. If you are interested in how a simulator count statistics, you can read the code snippet below and look for them in `score-sim` and `mcore-sim`. You may also need to look at how we count miss/access in `SingleCore`. How we count cache miss in `SingleCoreVRTL.v` is also shown in the below code snippet.

```
# from MultiCoreRTL.py

s.commit_inst   = OutPort( 4 )
s.icache_miss   = OutPort( 4 )
s.icache_access = OutPort( 4 )
s.dcache_miss   = OutPort( 4 )
s.dcache_access = OutPort( 4 )

# from mcore-sim

for i in xrange(4):
if model.model.icache_access[i]:
  icache_access[i] += 1
if model.model.icache_miss[i]:
  icache_miss[i] += 1

# from SingleCoreVRTL.v

assign icache_miss = icache_resp_val & icache_resp_val & ~icache_resp_msg.test[0];
```

If you add some logic to set the performance hooks correctly, `mcore-sim` will also report the accumulated statistics along with the per core/cache/bank breakdowns. It will report the number of committed instructions and CPI (per core and total), the number of cache access, misses, and miss rate (per cache and total), and the number of data cache bank accesses, misses, and miss rate (per bank and total). We recommend collecting this data for your parallel sorting implementation and reporting it in lab report as part of the evaluation section. Figure 9 shows the statistics from `mcore-sim` running the multithreaded using the staff solution for the `MultiCore` composition. Notice that the instruction cache miss rate is very low and the data cache miss rate is reasonable. Since vector-vector-add streams through the input and output arrays, there is significant spatial locality but little temporal locality. With four words per cache line we might expect a miss rate of around 0.25 since we will experience one compulsory miss every four words, which is similar to what we see in the statistics. Also noticed that the accesses are relatively balanced across the four data cache banks suggesting there are no hot spots.

You can also experiment with running single-threaded benchmarks on the multicore system. This allows you to see the impact of the extra hardware. Note that some of the benchmarks might not pass the verification because each core will try to execute the same exact code over the same exact data. Depending on whether the benchmark is written in an idempotent way or not, these additional cores might corrupt each others' data. There is a simple workaround to let other cores stop execution. Simply add the following code at the top of your single-threaded microbenchmarks.

```
num_cycles             = 2260      icache0_miss           = 26
total_committed_inst   = 1492      icache0_access         = 379
total_cpi              = 1.51      icache0_miss_rate      = 0.0686
                                   icache1_miss           = 9
total_icache_miss      = 55        icache1_access         = 424
total_icache_access    = 1646      icache1_miss_rate      = 0.0212
total_icache_miss_rate = 0.0334    icache2_miss           = 10
                                   icache2_access         = 424
total_dcache_miss      = 89        icache2_miss_rate      = 0.0236
total_dcache_access    = 408       icache3_miss           = 10
total_dcache_miss_rate = 0.2181    icache3_access         = 419
                                   icache3_miss_rate      = 0.0239
core0_committed_inst   = 341
core0_cpi              = 6.63      dcache_bank0_miss      = 23
core1_committed_inst   = 385       dcache_bank0_access    = 125
core1_cpi              = 5.87      dcache_bank0_miss_rate = 0.1840
core2_committed_inst   = 385       dcache_bank1_miss      = 22
core2_cpi              = 5.87      dcache_bank1_access    = 96
core3_committed_inst   = 381       dcache_bank1_miss_rate = 0.2292
core3_cpi              = 5.93      dcache_bank2_miss      = 21
                                   dcache_bank2_access    = 85
                                   dcache_bank2_miss_rate = 0.2471
                                   dcache_bank3_miss      = 23
                                   dcache_bank3_access    = 102
                                   dcache_bank3_miss_rate = 0.2255
```

Figure 9: Reference `mcore-sim` Statistics for Multithreaded *vvadd*

```
int main( int argc, char* argv[] )
{
  if (bthread_get_core_id() != 0) return 0;
  ...
```

In your lab report, discuss how each benchmark performed in each configuration. Did all of the benchmarks perform better on the multicore? Did they get the theoretical speedup of 4X over the single core? If you ran single-threaded benchmarks on the multicore system, is it slower than running on the single-core system? What is the slowdown due to? How do the two versions of sorting implementations perform on these two architectures? Can the parallel sort get a speedup over the serial version of sorting? Dive deeper into the result and discuss why you get the specific speedup.

6. Extensions

Advanced students might want to consider modifying the software and/or hardware to improve the performance of their parallel sorting microbenchmark. 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 128 elements. You can assume the data in these datasets will be uniformly distributed 32-bit random integers. 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 definitely want to experiment with the network. You will want to think critically about whether or not a bus or a ring makes the most sense for CacheNet or MemNet, 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 multicore system context. You might also want to use bus for some networks and and ring for other networks to get better performance. You will also want to experiment with different parallel sorting algorithms as algorithmic changes can often have the largest impact.

When you evaluate your SingleCore, you will quickly find that the performance of your baseline and even alternative multicore 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 used in this lab. 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. Reducing the read hit latency is the most critical since this would improve the performance of instruction fetch in your processor.

After you finish the cache hit latency, 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 designed in this lab 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.

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.