ECE 5740 Computer Architecture, Fall 2023
Position Paper Logistics

School of Electrical and Computer Engineering
Cornell University

revision: 2023-08-20-10-18

Graduate students enrolled in the ECE 5740 co-meet must submit three position papers throughout the semester. The goal of these position papers is to give students a chance for independent reading and analysis on processors, memories, and networks. Students might need to do additional reading beyond what is suggested here to prepare a compelling position paper. Please cite any relevant works in your position paper.

All position papers should include a title and the name(s) and NetID(s) of the student(s) who worked on the assignment at the top of the first page. Do not put this information on a separate title page. The report should be written using a serif font (e.g., Times, Palatino), be single spaced, use margins in the range of 0.5–1 in, use a 10 pt font size. All figures must be legible. Avoid scanning hand-written figures and do not use a digital camera to capture a hand-written figure. Do not just use a screen capture of the code. Definitely do not include screen captures that have white text on a black background. Your paper should not look like an outline. It should have paragraphs and prose. **Although there is no page limit, most high-quality position papers will probably require 2–3 pages.** Note you should interleave your figures, plots, and tables in the main body text where appropriate and not place all of your figures, plots, and tables at the end. **Submit your position paper in PDF format on Canvas.**

1. Position Paper 1: Processors

In this position paper, students will compare and contrast a classic CISC instruction set architecture (Intel x86) with an emerging RISC instruction set architecture (RISC-V RV64GC).

Start with the following readings:

- K. Asanović and D. Patterson, "Instruction Sets Should Be Free: The Case for RISC-V," UC Berkeley Technical Report No. UCB/EECS-2014-146, Aug. 2014.
  `http://people.eecs.berkeley.edu/~krste/papers/EECS-2014-146.pdf`
- A. Waterman, "Design of the RISC-V Instruction Set Architecture," UC Berkeley Ph.D. Thesis, Jan. 2016. (just read Chapter 5)
  `http://people.eecs.berkeley.edu/~krste/papers/EECS-2016-1.pdf`

Show the assembly code for implementing our element-wise vector-vector addition function using both the Intel x86 instruction set and the RISC-V RV64GC instruction set. Try to make sure your implementation is reasonably optimized. **Briefly explain how this assembly code implements the elementwise vector-vector addition function.** Note that you should use the full RISC-V RV64GC instruction set (which is the standard for all RISC-V Linux distributions) and not the much simpler TinyRV2 instruction set we primarily use elsewhere in the course. Also note that your RV64GC will almost certainly want to take advantage of the RISC-V compressed instructions to reduce the code size. **Clearly indicate which RISC-V instructions are four bytes and which are two bytes, and estimate the overall static code size in bytes for both implementations.** You are free to write the element-wise vector-vector addition function in C, compile this function for both the Intel x86 and

RISC-V RV64GC instruction set, and then include the resulting disassembly. Just be sure to explain how this assembly code works. **Create a table using the following template and include your own qualitative analysis for each instruction set architecture considering each metric. This table should not report quantitative results for your vector-vector addition function but should instead enable a high-level qualitative comparison.**

	Intel x86	**RISC-V RV64GC**
High-Performance Commercial Implementations?		
Low-Power Commercial Implementations?		
Licensing Restrictions		
Extensible		
Static Code Size		
Microarchitectural Complexity		
Dynamic Instructions / Program		
Cycles / Instruction		
Time / Cycle		

Use your readings and the above analysis to make a compelling argument for which of these two instructions sets will be more widely adopted in commercial products in 5–10 years. If you feel it depends on some additional factors, then be very specific on what are these additional factors.

2. Position Paper 2: Memories

In this position paper, students will compare a simple blocking cache with a more sophisticated non-blocking cache. In lecture, we have studied an FSM blocking cache microarchitecture. A *blocking cache* is a cache that cannot execute any other memory requests when handling a miss. Even the pipelined cache microarchitecture we studied is blocking since it uses an FSM to handle misses. A *non-blocking cache* is a cache that can execute other memory requests while servicing a miss. Start by learning more about non-blocking caches. **First, watch this Coursera lecture by Prof. Wentzlaff from Princeton University:**

- `https://www.coursera.org/lecture/comparch/non-blocking-caches-yHmtB`

Then consider the following readings:

- S. Belayneh and D. Kaeli, "A Discussion on Non-Blocking/Lockup-Free Caches," ACM SIGARCH Computer Architecture News, 24(3):18–25, Jun. 1996.
  `https://dl.acm.org/doi/10.1145/381718.381727`

Summarize how a non-blocking cache works in a single paragraph. Then draw three pipeline diagrams that illustrate how two iterations of our classic vector-vector add loop would execute on three different microarchitectures. All three microarchitectures are based on the canonical five-stage fully bypassed pipelined TinyRV1 processor. Assume a perfect instruction cache with a single-cycle hit latency and no misses. Assume a fully associative L1 data cache with four 16B cache lines that uses a write-through, no write-allocate write policy. The L1 data cache has a single-cycle read and write hit latency and a four-cycle miss penalty. Here are the three microarchitectures:

- Baseline processor with a *blocking* cache

- Stall-on-use processor with a *non-blocking* cache that supports only hit-under-miss
- Stall-on-use processor with a *non-blocking* cache that also supports miss-under-miss

The baseline processor will stall in the M stage on a miss. A *stall-on-use* processor marks the destination register of a load as "pending" and does not stall on a miss. The stall-on-use processor can keep executing later instructions and only stalls if a later instruction tries to read a pending register. Usually, a stall-on-use processor will have a second register file write port which is only used by loads to avoid structural hazards.

You should use your best judgment to figure out an effective way to illustrate how non-blocking caches work in your pipeline diagram. You likely will need to introduce new notation or symbols, which you should clearly explain.

Create a table using the following template and include your own qualitative analysis for each cache microarchitecture. This table should not report quantitative results for your vector-vector add loop but should instead enable a high-level qualitative comparison.

	Blocking Caches	**Non-Blocking Caches**
Microarchitectural Complexity		
Hit Latency		
Miss Rate		
Miss Penalty		
Time / Cycle		

Use your readings and the above analysis to make a compelling argument on whether we should use: (1) a baseline processor with a blocking cache; (2) (2) a stall-on-use processor with a *non-blocking* cache that supports only hit-under-miss; or (3) a stall-on-use processor with a *non-blocking* cache that supports supports both hit-under-miss and miss-under-miss. If you feel it depends on some additional factors then be very specific on what are these additional factors.

3. Position Paper 3: Networks

In this position paper, students will compare two on-chip network topologies, a ring-based topology vs. a mesh-based topology that can be used to interconnect multicore processors.

Start with the following readings:

- W.J. Dally and B. Towles. Principles and Practices of Interconnection Networks. Morgan Kaufmann, 2004. (Chapters 1–3,5)
  `https://canvas.cornell.edu/files/6330888/download`
- T.W. Ainsworth and T.M. Pinkston. "Characterizing the CELL EIB On-Chip Network." IEEE Micro, 27(5):6–14, Sep/Oct 2007.
  `http://dx.doi.org/10.1109/MM.2007.4378779`
- D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J.F. Brown, and A. Agarwal. "On-Chip Interconnection Architecture of the Tile Processor." IEEE Micro, 27(5):15–31, Sep/Oct 2007.
  `http://dx.doi.org/10.1109/MM.2007.4378780`

Assume we wish to implement an on-chip network for a multicore processor with 16 cores, and we are considering two different topologies. The first topology is a 1D torus (i.e., bi-directional ring) somewhat similar to the on-chip network used in the IBM Cell processor. The second topology is a 2D mesh (i.e., 4×4) somewhat similar to the on-chip network used in the Tile processor.

Create a table using the following template and calculate the zero-load latency and ideal throughput assuming uniform random traffic for each topology with 16 cores. Justify your results. Assume that each router hop requires one cycle and each channel hop requires one cycle. Assume the bandwidth of each channel in the ring is 256 bits/cycle and the bandwidth of each channel in the mesh is 128 bits/cycle.

	1D Torus	**2D Mesh**
Terminals	16	16
Channel Bandwidth	256 b/cycle	128 b/cycle
Zero-Load Latency		
Bisection Channel Count		
Bisection Bandwidth		
Ideal Throughput		

It is possible to use even higher-dimension torus topologies. **Include a paragraph explaining how the zero-load latency varies with dimension under a bisection bandwidth constraint.** *Hint: Chapter 5 in the Dally & Towles book should be of help!*

Use your readings and above analysis to make a compelling argument on whether we should use: (1) a 1D torus or (2) a 2D mesh in a multicore with 16 cores. How would your conclusions change for a multicore with 256 cores? If you feel it depends on some additional factors, then be very specific on what are these additional factors.