ECE 411: Computer Organization and Design

MP2: The RV32I Processor with a Unified 2-Way Set-Associative Cache

Version
1.3.0

Table of Contents

1. Introduction

MP1 left you with a working machine that implements the RV32I Instruction Set. Now that we have a machine that we can fully program and operate, we can start to look at performance. As you may have noticed, the memory used in MP1 included a delay before responding. In the real world, main memory access takes a long time relative to the length of the processor clock; and it takes even more cycles in high-end microprocessors. In fact, the relative speed difference between processors and memory has been growing over time and developing techniques for overcoming this difference is very important. Memory caches are therefore vital for achieving high performance. To help you gain a more detailed understanding of caches, how they work, and their performance, you will be augmenting the RV32I design from MP1 with a simple one-level cache. Specifications of the cache are in section 2 of this document.

Once again, this handout is only intended to help you get started. You are responsible for implementing the additions and modifications to the MP1 design on your own. We will cover cache design in lecture. In addition, textbooks may have helpful information on caches. We have put together some information to help you get started in section 4. **Only start this MP2 when you have a correct implementation of MP1**. Refer to the RISC-V specification for ISA concerns.

You are strongly encouraged to write your own test code to test your implementation and not wait until we release ours. This includes verifying not only instruction implementation, but cache correctness as well.

Grades will be mainly based on the correctness of design and thoroughness of verification. Some points may be allocated to your ability to execute the handin requirements (highlighting, etc.).

2. Cache Specifications

You will need to design a **one-level, unified, 2-way set-associative cache** with the following specifications.

- 8 lines per way with 8 words per line (i.e. total of 8 sets with 2 ways per set)
- Write-back with a write allocate policy
- LRU replacement policy
- Read/Write hits must take exactly two clock cycles to complete
- No more than 5 control states in cache controller (Note: 4 states are recommended)

Previously, the CPU datapath was interacting with the main memory directly. Now, you will need to modify the interface to implement the memory hierarchy. That is, you will need to insert a cache between the CPU datapath and the main memory. **You may NOT add additional signals between the cache and the CPU datapath.** Your cache must work with the same signals that MP1 main memory communicated with CPU datapath. The datapath must have no knowledge of your memory hierarchy. Signals used are found in section 3. MP2 main memory code will be provided as *physical_memory.sv*. This memory module has a slightly increased delay and is no longer byte addressable..

The memory bandwidth is now 256-bits so that a single load will fill a whole cache line. The new memory will allow only read and write of cache-line sized data (8 words or 256-bits). The cache will be constructed using only the following components:

- Control unit (you must create a state diagram for this)
- Decoders
- Comparators
- Muxes
- 4 byte to 32 byte bus adapter
- 2 Data store arrays
- Metadata arrays
  - 2 Tag store arrays
  - 2 Valid bit arrays
  - 2 Dirty bit arrays
  - LRU bit array
- Logic gates
- Registers (module provided in MP1)

You must use the provided modules bus adapter, data array, and metadata array modules.

Read/Write hits MUST take exactly two clock cycles to complete in this cache. Other operations may take multiple cycles, if necessary. A good way to test for a 2-cycle hit is look at the IF2 state for an instruction that should be in the cache. IF2 in this case should be exactly twice as long as IF1.

3. Signal Specifications

3.1. Memory Hierarchy Signals

These signals define the interface between the CPU Datapath and the Memory Hierarchy. Each of these signals must be present, and no additional signals are allowed. The main memory has its own set of signals, which are also detailed below.

3.1.1 Signals between CPU Datapath and Cache

- `mem_address[31:0]`
  - The memory system is accessed using this 32-bit signal. It specifies the address that is to be read or written
- `mem_rdata[31:0]`
  - 32-bit data bus for receiving data *from* the memory system
- `mem_wdata[31:0]`
  - 32-bit data bus for sending data *to* the memory system
- `mem_read`
  - Active high signal that tells the memory system that the address is valid and that the processor is trying to perform a memory read
- `mem_write`
  - Active high signal that tells the memory system that the address is valid and that the processor is trying to perform a memory write
- `mem_byte_enable[3:0]`
  - A mask describing which byte(s) of memory should be written on a memory write. The behavior of this signal is summarized in the following table:

`mem_byte_enable`	Behavior
4’b0000	Don’t write to memory even if `mem_write` becomes active
4’b????	Write only bytes specified in the mask (by a 1) when `mem_write` becomes active
4’b1111	Write all bytes of a word when `mem_write` becomes active

- `mem_resp`
  - Active high signal generated by the memory system indicating that the memory has finished the requested operation

3.1.2 Signals between Cache and Main Memory

- `pmem_address[31:0]`
    - Physical memory is accessed using this 32-bit signal. It specifies the physical memory address that is to be read or written.
- `pmem_rdata[255:0]`
    - 256-bit data bus for receiving data *from* physical memory
- `pmem_wdata[255:0]`
    - 256-bit data bus for sending data *to* physical memory
- `pmem_read`
    - Active high signal that tells physical memory that the address is valid and that the cache is trying to perform a physical memory read
- `pmem_write`
    - Active high signal that tells physical memory that the address is valid and that the cache is trying to perform a physical memory write
- `pmem_resp`
    - Active high signal generated by physical memory indicating that it has finished the requested operation

3.2. Memory Interaction

The main memory takes multiple cycles to respond to requests. When a response is ready, the memory will assert the `mem_resp` signal. Once a memory request is asserted, the input signals to memory should be held constant until a response is received. You may assume in your design that the memory response will always occur so the processor never has an infinite wait. As before, make sure that you never attempt a read and a write to memory at the same time.

4. Getting Started

Since MP2 is an extension of the work done in MP1, you should copy your completed MP1 design into a new folder for MP2. The steps for copying and beginning MP2 are below.

4.1. Get new MP2 files

- Merge the provided MP2 files into your repository:

```
$ git fetch release
$ git merge --allow-unrelated-histories release/mp2 -m “Merging MP2”
```

- Copy your MP1 design into your MP2 directory. Note that we have given you fresh copies of the provided CPU files. Do not overwrite these as the autograder will use its own copy and your design may not work

```
$ cp -pn mp1/hdl/* mp2/hdl/cpu
$ cp -pn mp1/testcode/* mp2/testcode/   # optional
```

- In the new directory, change all references to mp1 to reference mp2 instead (e.g., *mp2_tb* instead of *mp1_tb, mp2.sv* instead of *mp1.sv*). You can do this manually or try to use the *rereference.sh* script

```
$ cd mp2
$ ./bin/rereference.sh . mp1 mp2
```

- Ensure the `DEFAULT_TARGET` variable in the *bin/rv_load_memory.sh* script is correct so that the memory initialization file is written to the MP2 simulation directory. Furthermore, change the addressability of memory to correspond correctly to the physical memory (256 bits == 32 bytes)

```
# Settings
ECE411DIR=$HOME/ece411
DEFAULT_TARGET=$ECE411DIR/mp2/simulation/modelsim/memory.lst
ASSEMBLER=/class/ece411/software/riscv-tools/bin/riscv32-unknown-elf-gcc
ADDRESSABILITY=32
```

4.2. Description of Given Files

The following files are given (in `hdl/cache/` and `hvl/`):

- *array.sv*
  - Array module to be used for tag arrays, LRU array, etc.
- *data_array.sv*

    - Array module for the data arrays
- *bus_adapter.sv*
    - Module which will help your CPU (which likes to deal with 4 bytes at a time) talk to your cache (which likes to deal with 32 bytes at a time)
- *cache_control.sv, cache_datapath.sv, cache.sv*
    - Some blank modules to help get you started
- *mp2_tb.sv*
    - Testbench to simulate the MP2 design. Your design must match the register file, IR, data array, tag array, CPU, cache, CPU datapath, cache datapath, MP2 top level module names, as well as the identifier hierarchy imposed by the monitor module in order to work with the autograder
- *rvfimon.v*
    - RVFI verification monitor
- *physical_memory.sv*
    - Main memory with delay which will be connected to the cache (32-byte addressable)
- *shadow_memory.sv*
    - This module is similar to the RVFI verification monitor, it will help detect errors in your cache. The RVFI verification monitor aims to be synthesizable, which means it is impossible to keep track of memory state. This module does not aim to be synthesizable so it is able to maintain a copy of memory which updates every time the CPU performs a write. Refer to this file to see how the testbench and autograder expect memory to be formatted coming out of your cache

In addition, we have provided some components of the MP1 design in hdl/cpu/, as well as auxiliary files such as `/hdl/rv32i_mux_types.sv` and `hdl/*`. If you have additional supporting files from MP1, be sure to copy them over.

4.3. Beginning the New Design

To organize your MP2 design, we recommend that you organize your component files in the following manner:

- *hdl/mp2.sv*
    - the MP2 design. It contains the CPU and cache
- *hdl/cache/cache.sv*
    - the cache design. It contains the cache controller, cache datapath, and bus adapter
- *hdl/cache/cache_control.sv*
    - the cache controller. It is a state machine that controls the behavior of the cache
- *hdl/cache/cache_datapath.sv*

    - the cache datapath. It contains the data array, valid array, dirty array, tag array, LRU array, comparators, muxes, logic gates, and other logic
- *hdl/cpu/cpu.sv*
    - the CPU design. It contains the CPU controller and CPU datapath
- *hdl/cpu/cpu_control.sv*
    - the CPU controller. It is a state machine that controls the CPU datapath. This is the same as the control component from MP1
- *hdl/cpu/cpu_datapath.sv*
    - the CPU datapath. This is the same as the datapath component from MP1
- *hvl/mp2_tb.sv*
    - the testbench. It contains MP2 design and physical memory

These files are the upper hierarchy of the design, and you will be creating more files for lower-level components. You can define your own interface. But you need to make sure it is easily understood by others.

Once you have set up the interface correctly, you can start to work on the implementation.

The last thing you must do is confirm the target FPGA for the project. The FPGA you should use for this is the **Stratix IV EP4SGX530NF45C4ES**.

4.4 A Note About Alignment

In MP1 your design had to work with a memory module that only allowed aligned accesses. As in MP1, we will only test word accesses (`lw/sw`) that are word aligned and half word accesses (`lh/lhu/sh`) that are half word aligned (but possibly not word aligned) since otherwise these could span across two cache lines. Byte accesses (`lb/lbu/sb`) will never span across two cache lines so we could test any alignment for those. The RVFI monitor will enforce proper word aligned memory accesses, which requires you to ensure the bottom two bits of mem_address between the CPU and cache are zero'd, and your mem_byte_enable is correctly set.

5. Design Limitations

5.1 Things You Must Not Do:

- Start working on MP2 without being sure that MP1 works. The autograder for MP1 will continue running until MP2 CP1 is due to help you debug your design
- Make any changes to the CPU datapath or CPU controller (except to fix bugs from MP1)
- Model the cache as a single SystemVerilog component (i.e. making a single component and then writing SystemVerilog code to model the cache behaviorally) (This is a perfectly fine way of writing a testbench monitor, though) (Refer to section 5.2)
- Modify certain given files. The following files will be overwritten by the autograder: `mp2.qsf`, `mp2.qpf`, `hvl/*.sv`, `hdl/cpu/alu.sv`, `hdl/cpu/ir.sv`, `hdl/cpu/pc_reg.sv`, `hdl/cpu/register.sv`, `hdl/cpu/regfile.sv`, `hdl/rv32i_mux_types.sv`, `hdl/rv32i_types.sv`, `hdl/cache/array.sv`, `hdl/cache/bus_adapter.sv`, `hdl/cache/data_array.sv`. Please watch Piazza, as the set of replaced files may be changed, and if so, an announcement will be made

5.2. Things You Must Do:

- The right way to model the cache is to implement small components that do simple work, and connect them to form the complete design (just like how the datapath was implemented in MP1). As stated in the list of section 2, you will need to create low-level components (e.g. decoders, logic blocks etc.) and connect them in upper component like in cache.sv
- Follow certain naming conventions: for the autograder to work properly, you should rename your top level mp1 module from mp1 to cpu. You should maintain all other names you currently have working with the autograder. Please check the distributed test bench files for proper naming conventions for your cache modules and datapath.
- You **MUST** initialize your cache LRU values to zero
- You **MUST** ensure that your module hierarchy and signal identifiers adhere to those assumed by both the shadow_memory and riscv_formal_monitor_rv32i modules in the MP2 testbench
- You **MUST** ensure that you are testing your design with the rvfi monitor enabled, as grading will fail on any monitor errors

6. Checkpoints

For MP2, there will be two checkpoints prior to the final handin.

6.1. First Checkpoint

For the first checkpoint, you will need to submit a **paper design of your cache datapath and controller** that shows that you have made significant progress on your design. What does significant progress mean? Your paper design should be detailed enough for TAs to trace the execution of cache reads and writes (with a similar level of detail as the given MP1 spec). It should show at least:

- How data is read from the data arrays on a read hit
- How data is loaded into the data arrays from main memory on a read/write miss
- How data is written to the data arrays on a write hit
- How data is written from the data arrays to main memory on an eviction
- How the LRU determines which way to use
- The cache controller with states, state descriptions, transition conditions, and output signals as a function of state (Moore machine), or as a function of state and input (Mealy machine)

In addition to the paper design, you should start planning out how you will test your design. Having completed MP0 and successfully tested MP1, you should have some understanding of how to verify a moderately complex design. In this section, you will reflect on some lessons learned from the earlier MPs, as well as plan out how to test MP2. You will, in no more than a single page:

- Describe what worked well or didn’t from your MP1 testing. Please ensure answer the following questions (please answer candidly):
    - Was randomized testing part of your testing methodology, and if so, did it reveal any design flaws?
    - Did autograding runs alert you to the presence of design bugs?
    - Did autograding runs help you identify design bugs?
    - Did autograding runs impact your testing methodology?
- Analyze your cache design to identify two edge cases you will deliberately test
- Provide a short snippet of test code or sequence of cache input stimuli which tests one of your identified corner cases
- Briefly describe how you will unit test your cache as the DUT itself, rather than as part of your processor

Upload this document in PDF format to Compass before the deadline.

6.2. Second Checkpoint

For the second checkpoint, you will be required to have **cache reads** working. The autograder will only assign full credit for this portion, so if you want partial credit, you will need to meet with a TA during office hours to verify that your reads are functioning correctly.

Timing analysis will run the day before the checkpoint is due and the final checkpoint run, however you will not be graded on your ability to meet the timing requirements.

7. Final Handin

7.1 Implementation

For the final handin, you should have implemented the memory hierarchy including single-level, 2-way set associative cache with full ISA.

7.2. Testing:

We will provide you with a basic suite of test code, but you are responsible for the correctness of your design. Passing the provided test codes doesn't necessarily mean that your design is working in all cases. You need to write your own test code to cover more corner cases.

7.3. What to hand in:

You must commit ***AND PUSH*** your relevant files to Github before the deadline. **You must include the timing constraints file with the filename mp2.out.sdc.** The autograder will use the given version of any given files (from this or previous MPs), so your design should not rely on any changes you make to those files. You should not upload any .sv files which are not part of your project as the autograder will assume these are meant to be compiled which could generate grading errors.

8. Grading Rubrics

Total: 120 points

- **Checkpoints: 44 points (37%)**
  - **Checkpoint 1 - Paper Design: 16 points (13%)**
  - **Checkpoint 1 - Testing Strategy: 8 points (7%)**
  - **Checkpoint 2 - Cache Reads: 20 points (17%)**
- **Final Hand-in: 76 points (63%)**
  - **Target Tests 46 points (38%)**
  - **Longer Test 24 points (20%)**
  - **Timing Analysis Report: 6 points (5%)**

Note: For the cache way contents, the definition of "correct" is complex. In order to make the autograder happy, you must have your LRU array initialized to all zeros (the default array module behavior) and this should mean that array 0 is always filled first, array 1 should only be filled when there is valid data at that index in array 0.