Lab 8,9 - Design a Level 1 Cache Hierarchy

For Labs 8 and 9, you are required to perform the following activities:

1. Integrate an instruction cache that we provide into your pipelined cpu. The icache has 16 sets, is direct mapped, and has one word per block.
2. Create a 256B two-way set associative data cache with two words per block. A set will have four words in total. The dcache is write back and write allocate. You must implement an LRU replacement policy.
3. Provide a block diagram for the icache and dcache. Hand drawn or photocopied/scanned diagrams will not be accepted. No Exceptions.

NOTE

Any evaluation turned in without block diagram(s) of the design will receive a grade of 0.

4. Create and run test benches for each cache.
5. Verify that your cache synthesizes correctly.
6. Hook your cache hierarchy up to your pipeline.
7. Verify correct behavior of your pipeline with caches.
8. Demonstrate that your pipeline is working with the caches in hardware.

1 Background

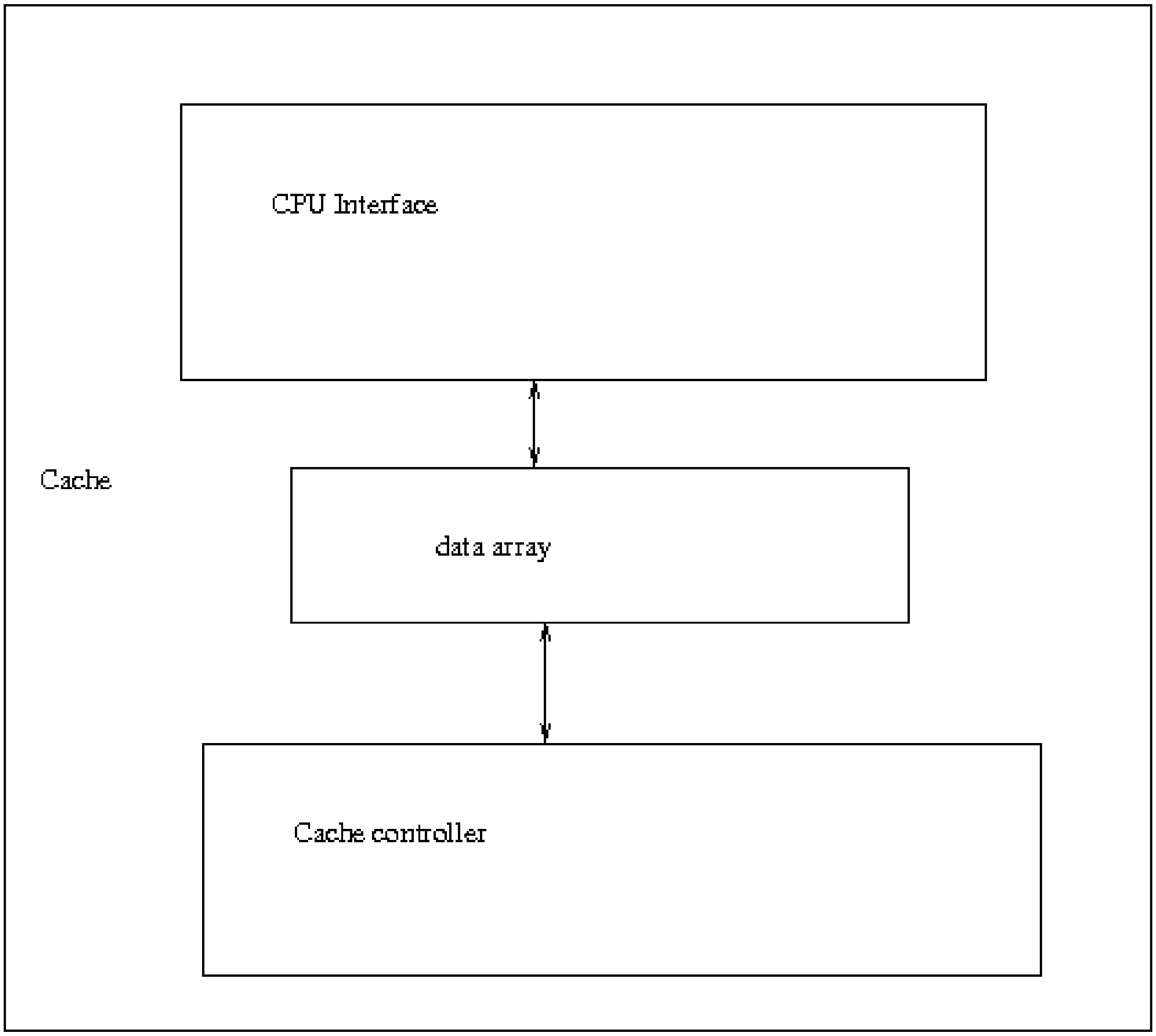
The caches will start a new project. You will want to create a `project3` directory in the `ece437` directory. You will need to set up this directory correctly. Re-read the Lab 1 handout for instructions on how to use dirset.

The cache may be designed independently of your processor. After you design and debug your cache you must integrate it into your processor.

2 Cache Design

Figure 1 is a diagram of a basic cache structure. The main part of any cache is the data array. The data array is used to store the data in the cache.

Figure 1 Cache Components



On one side of the data array we have the processor interface. This interface is generally created with asynchronous logic. When the processor accesses the cache this logic will check for a hit or miss. When the cache hits the data is routed out of the data array into the processor, and on a cache miss the logic alerts the cache controller.

The cache controller is the brains of the cache. It services the cache on misses by loading data from the memory. It is usually implemented with a state machine. The state machine stays in an idle state during cache hits. When there is a miss, the processor interface signals the controller to start loading a new block into the cache. When the loading of the block into the cache is complete, the processor interface recognizes a hit and signals a hit to the processor.

WARNING

Always load a block from memory to the cache upon a miss (read miss and write miss). Just overwriting the data upon write miss without fetching the block from memory is NOT considered a proper cache design.

Data is transferred between memory and caches in block size granularity. A block is the number of words per set per way. There is a valid and dirty bit per block, not per word.

Tag data is usually stored separately from the data portion of the cache in a real design. In your design you may structure this information however you choose.

Don't forget about halts. If your processor executes a halt instruction, make sure your cache controller isn't in the middle of any operations. If you assert your halt signal and you are in the middle of a memory transaction you may get an incorrect memout.hex.

Some hints about caches:

1. The processor does not write to the I-Cache.

2. There will be cache miss/hit not only for a read operation but also for write operations.

3. The Dcache has 2-words per block but the memory port bandwidth is only 1-word long. Because of this, each data transfer operation between the Dcache and memory will take more than 1 cycle.

4. Upon a miss, think about how the CPU is to know when data fetched from memory is ready to use. (use miss/hit signal, valid bits)

5. Write back means only write back from the cache to memory when a dirty block gets replaced. For clean blocks, the CPU simply overwrites the block in cache.

6. Due to write back, the memory may not have the latest values of data, even at the end of execution.

IMPORTANT

You need to dump the dirty cache contents to memory before the testbench dumps memory out at halt. Hold your halt until all dirty blocks in the Dcache are transferred to memory. This is IMPORTANT! If this is not done, your memout.hex will be wrong even if your entire execution process operates correctly!

7. For LRU use a bit per set to identify which way was accessed last. Only one bit is needed because there are only two ways.

8. For ease of grading, please name your cache top level file dcache.vhd

3 Generating RAM for the Cache

To create a block of RAM for your cache, you will have to write the ram module. The ram module should be asynchronous read and synchronous write. It will resemble your register file with one read port and one write port. There is a sample file posted on the materials page of the website called data16x126.vhd. You should structure your data array in this format.

NOTE

Your design must continue to work with the variable latency RAM.

4 LRU Replacement and the Hit Counter

As mentioned above, you are required to implement LRU replacement in your Dcache. The success of LRU depends on temporal locality as we expect the block that was accessed least recently to be a good choice for replacement. This can be implemented with one bit per set. This bit should be updated every time a set is accessed (reads and writes).

Caches are simulated in `sim` with the exact same configuration you will be implementing in hardware. You can run `sim -c`, where the -c flag tells sim you want it to operate with caches. In addition to the normal stats, it will also give you hit rates for the Icache and Dcache. You can also see cache hits and misses in the trace.

One major difference in this lab is the existence of a hit counter. Hardware counters are used in modern processors to keep track of different metrics. You will be implementing a counter that keeps track of hits in the Dcache. An example file, hitcounter.vhd, is on the materials page for you to base your counter off of.

We will be using this counter to verify that your cache is operating correctly. You must keep track of cache hits and write that value to memory upon halt. The specific memory location can be found in the hitcounter.vhd file. Your hit count must match the simulator or your memout.hex will not match memdump.hex.

IMPORTANT

For Lab 8 you will run `sim` as you have in the past (only the Icache is required and there is no counter value written for it). For Lab 9, you must run `sim -c`. Failure to match the simulator with caches in Lab 9 will cause you to fail the grading script.

Make the integration of the hardware counter modular. You will not be using it in the multicore labs.

5 Cache Testing

Create a test bench that executes a variety of cache hits and misses. This should test limited functionality of your Dcache. Remember that you should test your cache as much as possible so that when you hook your cache up to your CPU it is as straight forward as possible.

In addition, it would be a good idea to write your own asm files that test corner cases such as dirty writebacks.

6 Demonstrations

In order to get points for your lab you must integrate your caches in your pipeline and run fib, mult, and search. Compare cycle counts for a system with caches and a system without caches. You should see a big improvement for longer, memory-intense programs.

7 Turning In

Again, to make sure your `Makefile` works, type the following from your project3 directory:

```
~/ece437/project3> make clean
~/ece437/project3> make lab9
```

Now invoke vsim, load your design and run the simulation for all the benchmarks. Re-Diff the `.hex` files to make sure you are submitting the working version of your design.

Also, please make sure your tb_cpu.vhd and cpu.vhd are the same as given (do not comment any signals out), and make sure your very top level file is called mycpu.vhd, and your cache top level file is called dcache.vhd before you do your submission.

Lastly, from your `project3` directory, type:

```
~/ece437/project3> submit -p lab9
```