EECS 150 Spring 2012 Checkpoint 3: Caches

Prof. John Wawrzynek
TAs: James Parker, Daiwei Li, Shaoyi Cheng
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Revision 1

1 Introduction

At this point the processor is able to run C programs and communicate over serial; however, the programs are constrained by memory capacity (recall that there is only about 5Mb of block RAM available on the FPGA). In this checkpoint, you will hook up your CPU to caches backed by DDR2, which has a capacity of 256MB.

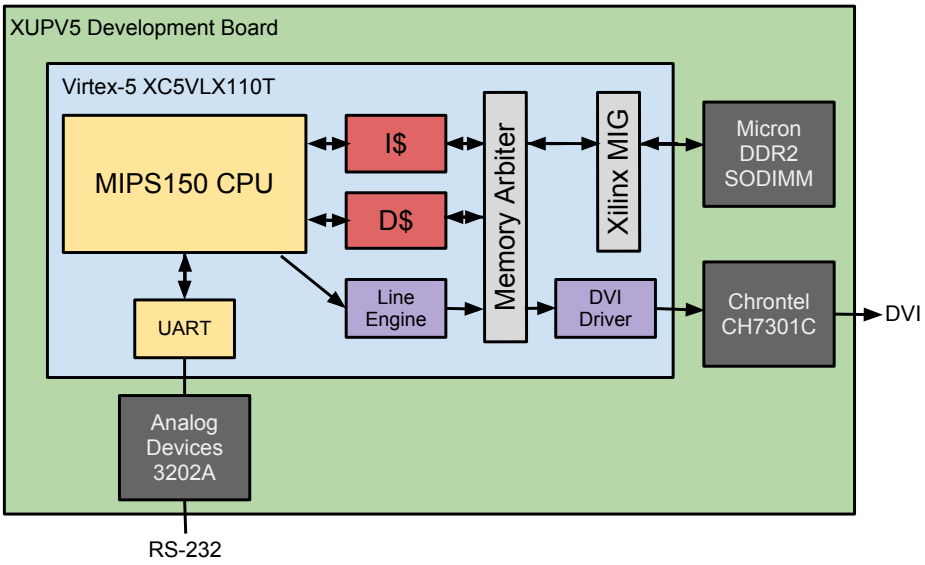

Figure 1: A high-level overview of the final system

Recall Figure 1 from the first checkpoint. The yellow blocks are now complete; in this checkpoint, the focus is on the data and instruction caches, shown in red. The cache, memory arbiter and Xilinx MIG (Memory Interface Generator) are provided in the skeleton files. Your task will be to integrate the cache and memory system with your CPU and then modify the cache to improve its performance.

2 New Files

This checkpoint will add a large number of new modules to your project. You should only need to modify the cache and the processor, but you'll need to be familiar with the provided code:

- `Cache.v`: This is the cache module. The skeleton files provide a 8KB, direct-mapped write-through, write-no-allocate implementation. You will need to change this to a set-associative, write-back, write-allocate cache of the same capacity.
- `cache_data_blk_ram`: Block RAM used in the provided cache to hold the data contents.
- `cache_tag_blk_ram`: Block RAM used in the provided cache to hold the tag contents.
- `Memory150.v`: This module contains instantiations of the caches, memory arbiter and the Xilinx DDR2 interface.
- `MemArbiter.v`: This controls which cache has access to the DDR2 and interleaves read operations.
- `mig_{a,rd,wd}f`: These are clock-crossing FIFOs. The CPU (currently) runs at 50 MHz, but the DDR2 controller runs at 200 MHz. These FIFOs are used to transfer data across the clock domains. `mig_af` is the address and command FIFO, `mig_wdf` is the write data FIFO, and `mig_rdf` is the read data FIFO.
- `mig_v3_61`: This is Xilinx's DDR2 memory interface, generated in Coregen.
- `request_fifo`: This is used by the memory arbiter to keep track of who sent requests. Overkill now, but necessary when graphics (i.e. additional memory accessors) are added.
- `bios_mem`: Dual-port read-only block RAM that will be initialized with `bios150v3` to bootstrap the processor. More on this in Section 2.

3 Integrating Caches in the Processor

There are two steps to this checkpoint: integrating the provided direct-mapped cache into your processor and making performance improvements to the cache. The best approach is to make changes step-by-step and run regression tests after each change.

3.1 Memory Map

The first part of this checkpoint requires integrating the provided cache with the processor. However, the contents of the DDR2 can't be initialized[1], so a read-only 'BIOS' memory will be added to bootstrap the CPU. The memory map from checkpoint 1 is now:

[1]The caches use block RAM, so they can be initialized, but that solution requires organizing the program in memory based on the cache parameters.

Table 1: Updated Memory Address Partitions

Address[31:28]	Address Type	Device	Access	Notes
4'b00x1	Data	Data Cache	Read/Write	
4'b0001	PC	Instruction Cache	Read-only	
4'b001x	Data	Instruction Cache	Write-Only	Only if PC[30]
4'b0100	PC	BIOS memory	Read-only	
4'b0100	Data	BIOS memory	Read-only	
4'b1000	Data	I/O	Read/Write	

This memory map is designed to allow a few important features:

- **Initialization**: The top nibble of the PC should now start at 0x4. This allows the bios memory to be initialized with a `.coe` file, in the same manner as the instruction and data memories in the previous checkpoints. Remember to change the addresses in the `.ld` and `start.s` files.
- **Reprogrammable**: When running from the bios, the instruction cache can be written to. This allows reprogramming in the same manner as the first checkpoint. When programming the CPU, store the new program to an address beginning with 0x3 for coherence between the caches.
- **Unified instruction and data memories**: Both the instruction and data caches are backed by the same DDR2 memory. Programs need to read from the data section, so in this scheme the program is only stored once in memory (because the caches will evict it to the same location in DDR2).

3.2 Cache Interface

An 8KB direct-mapped write-through, write-no-allocate cache has been provided for you in the skeleton files. This is used for both the instruction and data caches, which are instantiated in `Memory150`. The CPU interface to the caches has been wired into `MIPS150`:

- `{i,d}cache_addr`: 32-bit address, used for both reads and writes.
- `{i,d}cache_we`: 4-bit write mask, identical to the write mask used for the block RAMs. When at least one bit of the mask is high, the cache attempts to write din to the provided address.
- `{i,d}cache_re`: Read enable signal.
- `{i,d}cache_din`: 32-bit Data into the cache.
- `{i,d}cache_dout`: 32-bit Data out from the cache.
- `stall`: Same meaning as in checkpoint 2. This is set while either cache services a miss.

3.3 Recommended Steps

Rather than try to implement everything at once, do it in a few steps and verify that everything is working between steps. When you are not using a cache, simply assign the enable outputs in `MIPS150` to 0.

1. Add the bios memory to your CPU and make modifications to your datapath to accommodate the new address scheme. Treat the instruction memory and data memory as if they were the instruction and data caches (with regards to which top nibble addresses should access them).

2. Replace the data memory with the data cache. Make to re-build the `bios150v3` program with the base address (in `bios150v3.ld`) set to 0x40000000. Loads from the data section of the program now come from the new bios memory, and loads/stores on the stack now go to the data cache.

3. Once the data cache is working, add the instruction cache. You should then be able to follow the procedure used for checkpoint 1 checkoff to reprogram the CPU.

3.4 Benchmarking

The next part of this checkpoint involves making changes to the cache to improve performance. To determine CPI (cycles per instruction), the I/O memory map is expanded to include counters:

Table 2: I/O Memory Map

Address	**Function**	**Access**	**Data Encoding**
32'h80000000	UART transmitter control	Read	{31'b0, DataInReady}
32'h80000004	UART receiver control	Read	{31'b0, DataOutValid}
32'h80000008	UART transmitter data	Write	{24'b0, DataIn}
32'h8000000c	UART receiver data	Read	{24'b0, DataOut}
32'h80000010	Cycle counter	Read	Total number of cycles
32'h80000014	Stall counter	Read	Number of cycles stalled
32'h80000018	Reset counters to 0	Write	N/A

The cycle counter should be incremented every cycle, and the stall counter should be incremented every cycle the processor is high. From these counts, the CPI of the processor can be determined for a given benchmark.

Once you've integrated the provided caches with your process, load the mmult program into the instruction cache and run it (in `software/mmult/`). This program computes $S = AB$, where A and B are 64x64 matrices. The program will print a checksum[2]and the counters discussed above. Record these numbers for later comparison against the modified cache.

[2]The checksum is 0001f800. If you do not get this, there is likely a problem with cache eviction.

4 Improving Cache Performance

After the caches are integrated in the CPU, you will need to modify the cache module for performance improvements. The starting configuration of the cache is 8KB, direct-mapped, write-through, and write-no-allocate. You will modify the cache to be 8KB, 2-way set-associative, write-back, and write-allocate. Please refer to Appendix A if you are confused about any of the terms in this paragraph.

The next section provides documentation on the DDR2 memory interface. You will need to understand this interface and how the provided cache uses it before you attempt to make modifications to it.

4.1 DDR2 Interface

The XUPV5 development board has a 256MB DDR2 SODIMM (small outline dual inline memory module) mounted on the underside. The interface to this module is provided through Xilinx's MIG (memory interface generator), which in turn is connected via clock-crossing FIFOs to the memory arbiter. The arbiter sits between the caches and the MIG and provides the illusion that each cache has exclusive access to the FIFOs.

Inputs to the cache from the arbiter:

- `rdf_dout`: 128 bits of data out from the read data FIFO.
- `rdf_valid`: Indicates the data from the read data FIFO is valid.
- `af_full`: Indicates the address FIFO is full. You can think of this as a ready signal.
- `wdf_full`: Indicates the write data FIFO is full. You can think of this as a ready signal.

Outputs from the cache to the arbiter:

- `rdf_rd_en`: Read-enable signal for the read data FIFO. You can think of this a ready signal.
- `af_cmd_din`: 3-bit command to the DDR2 controller.
- `af_addr_din`: Address (in the DDR2 domain).
- `af_wr_en`: Write-enable signal for the address FIFO (which also writes the command). You can think of this as a valid signal.
- `wdf_din`: 128 bits of data to the write data FIFO.
- `wdf_mask_din`: 16-bit active-low byte write mask.
- `wdf_wr_en`: Write-enable signal for the write data fifo (includes the mask). You can think of this as a valid signal.

The memory on the board is configured for a burst length of 4 and each address maps to 64 bits of data. DDR stands for 'Double Data Rate', which means that data is transferred on both edges of the controller's clock. The controller therefore has a port width of 128 bits that is connected to the clock crossing FIFOs. Finally, to exploit the efficiency of reading in bursts, a natural block size for the cache is 256 bits.

For the cache, this means the following steps need to be performed for a write:

1. Supply a 31-bit address to `af_addr_din`, of which the low 25-bits matter, while the upper 6 should be zero.
2. Set `af_cmd_din` to 3'b000.
3. Supply 128-bits worth of data to `wdf_din`.
4. Supply 16-bits worth of byte mask to `wdf_mask_din`.
5. Assert `wdf_wr_en` and `af_wr_en`when `!af_full && !wdf_full`.
6. Supply the next 128 bits of data and assert `wdf_wr_en` when `!wdf_full`.

Then, to read:

1. Supply a 31-bit address to `af_addr_din`, of which the low 25-bits matter, while the upper 6 should be zero.
2. Set `af_cmd_din` to 3'b001.
3. Assert `af_wr_en` when `!af_full`.
4. Assert `rdf_rd_en` to indicate waiting for data.
5. Wait for `rdf_data_valid` to be asserted and store the first half of the block.
6. Wait for `rdf_data_valid` to be asserted again and store the second half of the block, and set `rdf_rd_en` low again.

4.2 Recommended Steps

Once you understand the cache implementation provided in `Cache.v`, begin working on the required modifications. We recommend that you do this in three steps, testing your design after each one:

1. Change the write policy from write-through to write-back. This will require modification of the portion of the FSM which handles write hits.
2. Change the write allocate policy from write-no-allocate to write-allocate. This will require further modification of the FSM in the cache.
3. Make the cache 2-way set associative. You may need to generate your own block RAMs to do this; instructions are in appendix B.

4.3 Restrictions

There must be only one control FSM for your cache (e.g. to make it set-associative, you are not permitted to tie together two direct-mapped caches).

5 Testing

Just as in checkpoint 1, testing will be key to getting your design working quickly. A testing framework has been provided for you in `hardware/src/CacheTestTasks.vh`. This header file contains several tasks that will be useful for writing tests for your caches.

In addition, several tests have been written already for you in `CacheTestBench.v` and `Memory150TestBench.v`. Note that the provided tests are not complete and were written for the write-through, write-no-allocate cache. You will need to modify the existing tests as well as add additional tests to ensure you cover all the possible cache interactions. The possible cache interactions are as follows:

Table 3: Cache interactions

Instruction Cache	**Data Cache**
Idle	Read/write miss, read/write hit
Read hit	Read/write miss, read/write hit
Read miss	Read/write miss, read/write hit
Write hit	Write miss, write hit (same address)
Write miss	Write miss, write hit (same address)

Before modifying the testbench and the cache, you may want to run the testbenches as provided to get an idea of what kind of output you should expect.

6 Checkoff

Checkoff will consist of demonstrating mmult with the improved cache and reporting the performance changes. This checkpoint is due **Friday, April 6**.

A Cache Policies and Properties

The relevant cache properties for this project can be found in the slides here:
`http://inst.eecs.berkeley.edu/~cs150/sp12/agenda/lec/sram2-proj2.pdf`.

As a quick reference, they are explained here:

A.1 Write policy

1. Write through - You always write the data to main memory, even if there is a copy of it in the cache already. Note that in the provided cache, the FSM transitions to the write state on a write enable.
2. Write back - You only write back the data to main memory when a cache line is evicted. One optimization you can and should make is to use a dirty bit to keep track of whether the line in the cache had been modified. If it hasn't been modified, then you don't have to write back the cache line to main memory when it is evicted.

A.2 Write miss policy

1. Write-allocate - On a write miss, you pull in the missing line from main memory and then update the line with the data that you were trying to write. This is usually associated with a write-back write policy.

2. Write-no-allocate - On a write miss, you don't pull in the missing line from main memory. This means that a write to some address followed by a read to the same address will cause a read miss. However, main memory will have the updated data that you had tried to write to the cache, so it will still be correct. This policy is usually associated with a write-through write policy. Note that in the provided cache, the FSM does not transition to the first read state (FETCH) on a write miss, and instead transitions directly to the write state.

A.3 Set associativity

You can think of a set associative cache as having multiple copies of direct mapped caches side by side. To understand the difference, let us consider what happens in the four possible interactions (this is written assuming that you have a write-back, write-allocate cache):

1. Read hit - On the first cycle of a read, the tags in all the ways (sets) of the cache are compared. If the tags match and the valid bit for the tag is set, then you should use a mux to select the data coming out of that cache.

2. Read miss - If none of the tags are a hit, then we want to fetch the requested data from main memory. However, we must decide which way we will write the data into once we get it back. A simple implementation would be to pick the way randomly. However, this may lead to unnecessary write backs to main memory. A more sophisticated method would be to find the first way that isn't valid to write back into, or if all the ways are valid, find the first way that isn't dirty, and finally, if all the ways are dirty, then pick one at random. An even more sophiscated implementation would keep track of the least recently used way with additional metadata.

3. Write hit - This is analogous to a read hit. If you get a tag hit in one of the ways, you simply write the data to that way on the next cycle.

4. Write miss - This is analogous to a read miss. You must implement some sort of eviction policy as described, and make sure to update the correct way after fetching data from main memory.

B Block RAM generation

You may need to create RAMs of various geometries for your cache. These can be generated using coregen:

1. Open coregen (type `coregen` at the terminal).

2. Create a new project. The part family is `Virtex-5`, the device is `xc5v1x110t`, and the design entry should be Verilog. The other options can be left at defaults.

3. In the pane on the left, find the Block Memory Generator and select it. Click 'Customize and Generate' in the right pane. There are 5 pages of configuration information; if you are unsure about any of the options look at the `.xco` files used for the instruction and data memories or of the provided cache memories.

4. After the part is generated, copy the `build` and `clean` scripts into the output directory. Edit the part name in the `build` script to match the generated part.