# EECS 150 Fall 2012 Checkpoint 2: Cache Integration

Prof. John Wawrzynek
TAs: Vincent Lee, Shaoyi Cheng
Department of Electrical Engineering and Computer Sciences
College of Engineering, University of California, Berkeley

Revision 1, Due Wednesday April 3rd, 2013 @ 2:00PM

## 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.

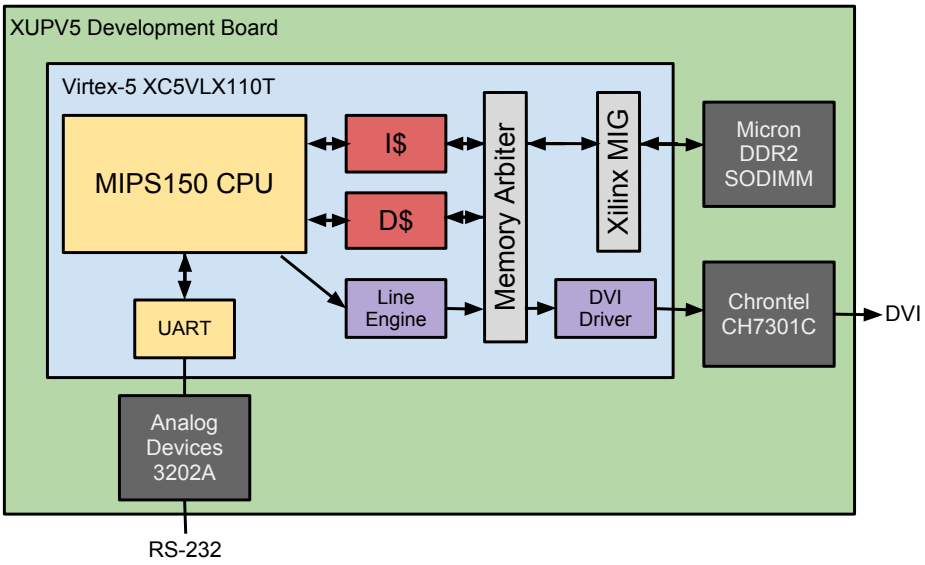


Figure 1: A high-level overview of the final system

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 add a couple counters for benchmarking.

## 2 New Files

This checkpoint will add a large number of new modules to your project. You should only need to modify the processor, but you may want to familiarize yourself 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.
- `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, request controller, and the Xilinx DDR2 interface.
- `RequestController.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.
- `bios_mem`: Dual-port read-only block RAM that will be initialized with `bios150v3` to bootstrap the processor. More on this in Section 3.

You will need to generate all of the new cores for simulation or synthesis to work. To assist with this, there is a build script that will build all of the necessary modules. Simply run `./build_memories` from the `/src` directory. Don't forget to rebuild the bios with the new addresses (see section 3) and copy the `.coe` file into the `bios_mem` directory before running the script. Each of the directories also has its own build script if you need to rebuild just one, for whatever reason.

## 3 Integrating Caches in the Processor

### 3.1 Memory Map

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 address in the `.ld` file to begin with 0x4, recompile bios150v3, and copy the `.coe` file into the `bios_mem` directory.
- **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 1. 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 sure 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.

We have also provided a few testbenches (`EchoTestbenchCaches`, `CacheTestBench`, and `Memory150Testbench`) that show how to write tests for the caches. This semester's cache checkpoint is significantly easier than previous semesters, so you may not actually need all of these, but they are included just in case you may find them useful.

### 3.4 Benchmarking

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 | Instruction counter | Read | Number of instructions executed |
| 32'h80000018 | Reset counters to 0 | Write | N/A |

The cycle counter should be incremented every cycle, and the instruction counter should be incremented every cycle the processor is high (not stalling). From these counts, the CPI of the processor can be determined for a given benchmark.

Once you've integrated the provided caches with your processor, 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 and the counters discussed above. The checksum is 0001f800. If you do not get this, there is likely a problem in your CPU with one of the instructions that is used by the bios but not mmult.

# 4 Checkoff

Checkoff will consist of demonstrating that mmult runs with the correct checksum. The correct value of the checksum is 0x001f800. This checkpoint is due **Wednesday, April 3rd 2013 at 2:00 pm**.

# 5 How to Survive This Checkpoint

In the past many groups have had bugs that were masked in checkpoint 1 manifest themselves in checkpoint 2. Several of the most commonly suppressed bugs involve the bit width of the PC, faulty JALR implementations, and bad or missing reset logic. The elements in this checkpoint are relatively straightfoward and easy but be open to the fact that the BIOS tests ALMOST everything and there may exists some bugs that get unmasked in this checkpoint. Another pitfall in this checkpoint is clobbering the address space and getting the .ld and stack pointers right. When you run the code for this checkpoint, make sure that your $sp and .ld pointers correspond to the correct location. Finally, some students try and be clever and gate the clock in order to implement the stalling for this checkpoint. Don't do this. It is more complicated than you may think and you will spend lots of time banging your head against the table if you do this.

# A Provided Cache

The configuration of the cache is 8KB, direct-mapped, write-through, and write-no-allocate. The next section provides documentation on the DDR2 memory interface. You do not need to modify any of the provided code. You may want to take a look at the cache to understand the basics of the timing but you do not need to modify it. The information in this section is merely to help you understand what the cache is actually doing if you are curious. Note that this section will be helpful later when we do graphics processor DMA.

## A.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 request controller:

- `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 request controller:

- `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.