MIPSfpga by Imagination

Lab 20

A first glance into caches

These materials produced in association with Imagination.
Join our University community for more resources.
community.imgtec.com/university

Lab 20
A first glance into caches

1. Overview

This is the first of a series of labs analyzing microAptiv's memory system. Before beginning these labs, it is recommended to study chapters 4 and 8 of document [1], chapter 8 of [2], and the appendix B of [3]. Figure 1 illustrates the top-level block diagram for the microAptiv UP processor core design, highlighting the modules that we analyze along this series of labs: the Data and Instruction Caches (**m14k_dc** and **m14k_ic**), the Data and Instruction Cache Controllers (**m14k_dcc** and **m14k_icc**), and the ISPRAM and DSPRAM Interfaces.

In this initial lab (Lab 20) we perform several experiments on the FPGA board where we examine different cache configuration parameters by propagating signals related with the cache to the board peripherals (LEDs and 7-segment displays), similarly to what we did in Lab 18.

The other labs of this series perform more precise analysis. Specifically, in Lab 21 we analyze the different arrays that make up the cache (Data Array, Tag Array and Way-Select Array), create new cache configurations and test them via the performance counters. Then, three labs delve into the cache controller. As such, in Lab 22 we analyze the cache hit and miss management. Then, in Lab 23 we analyze the different cache content management policies available in MIPSfpga, such as the replacement policy, the write policy, etc., and experiment with them. Finally, in Lab 24 we analyze store instructions and fill operations. The last lab of this series (Lab 25) explains the operation of a scratch pad memory and asks you to implement a simple scratch pad and test its behavior.

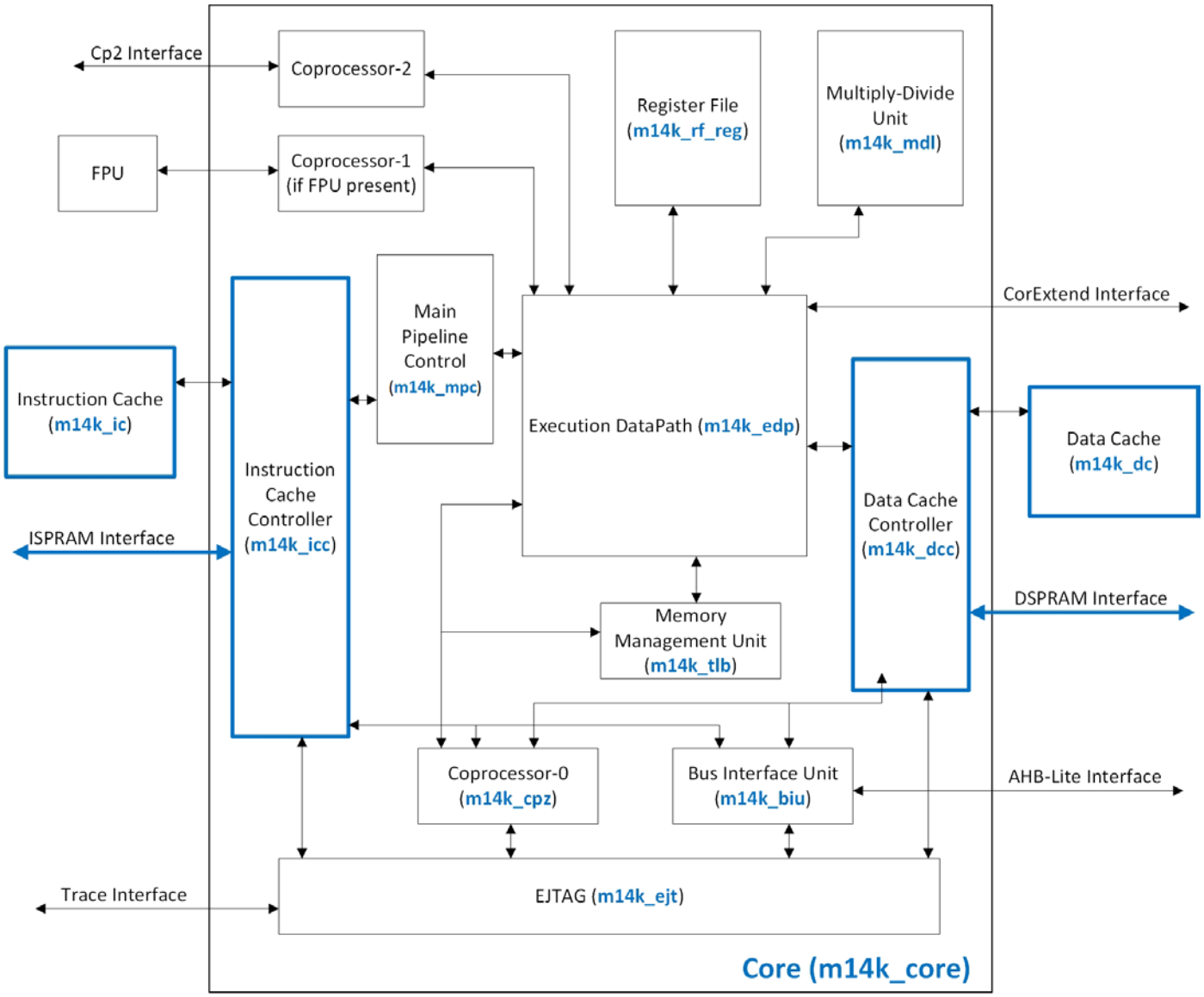

Figure 1. MicroAptiv UP CPU-level Block Diagram.

2. Introduction

MicroAptiv UP contains a single cache level, made up of Harvard-style caches with a separate instruction cache (I$), implemented within module **m14k_ic** (Figure 1), and a load/store data cache (D$), implemented within module **m14k_dc** (Figure 1). The use of separate caches allows instruction and data references to proceed simultaneously. The caches are managed via a cache controller: the instruction cache controller, implemented within module **m14k_icc** (Figure 1), manages the I$ and the data cache controller, implemented within module **m14k_dcc** (Figure 1), manages the D$.

Tables 8.1 and 8.2 in [1] show the different cache configurations supported by microAptiv. The I$ and D$ are independently configurable in terms of size, associativity, management policies, etc. As shown in those tables, the caches can range in size from 0 to 64 KB and can range in associativity from 1 (direct-mapped cache) to 4 ways. The cache line/block size is 16 Bytes.

The two caches are virtually indexed and physically tagged, allowing cache access to occur in parallel with virtual-to-physical address translation. Moreover, the caches are accessed in a single processor cycle. Thus, in case of a hit, the pipeline can maintain its optimal throughput. In case of a miss, the missed line is requested to main memory. The missed line is returned from main memory during a 4-beat burst transaction. The critical word is always returned first. The caches are blocking until the critical word is returned, but the pipeline may proceed while the other 3 beats of the burst are still active on the bus.

Section 3 explains how to display Core signals related with the D$ on the LEDs, so that you can study them. The process is similar to the one used in Lab 18, but we explain it again. Then, the lab proposes some exercises, where you analyze the access pattern of different programs and infer cache parameters such as the line size or the cache associativity.

3. Display signals through the board peripherals

In this section we illustrate how to propagate Core signals to the board peripherals. We first reconfigure MIPSfpga for supporting the slow clock option (explained in detail in Lab 10) and then we reconfigure it to output some signals related with the D$. Specifically, we output through the LEDs the following signals:

- LED5: *clock signal*
- LED4: *mpc_run_ie*
- LED3: *mpc_run_m*
- LED2: *mpc_run_w*
- LED1: *dcc_dmiss_m*
- LED0: *dcc_pm_dhit_m*

In folder *Part4_Memory\Lab20_BasicCaching\Verilog* we provide both a *bitfile* and a set of Verilog files that you can use for creating the *bitfile* yourself. To do the latter, use the files provided in the *Part4_Memory\Lab20_BasicCaching\Verilog* folder along with the original MIPSfpga rtl_up files to create a Vivado project and a bitfile (refer to Lab 1 for a refresher of how to do this). Notice that some of the files replace the original files. These Verilog files were created according to the following steps:

Step 1. Modify MIPSfpga RTL files to support the slow clock option, as explained in detail in Lab 10

Step 2. Modify MIPSfpga Core RTL files to propagate signals to the top of the module hierarchy

Step 3. Modify MIPSfpga system RTL files to propagate signals to the LEDs

Step 1. Modify MIPSfpga RTL files to support the slow clock option, as explained in detail in Lab 10

Starting with the default MIPSfpga system provided in *MIPSfpga_GSG\rtl_up*, include the necessary hardware to support the slow clock option, as explained in Lab 10.

Step 2. Modify MIPSfpga Core RTL files to propagate signals to the top of the module hierarchy

Starting with the MIPSfpga system implemented in Step 1, modify the core RTL files in order to propagate some signals to the top of the module hierarchy. We want to be able to analyze signals *mpc_run_ie*, *mpc_run_m* and *mpc_run_w*, which determine if the pipeline stages progress or not, signal *dcc_dmiss_m*, which is 1 when a miss is detected at the D$, and signal *dcc_pm_dhit_m*, which is 1 when a hit is detected at the D$. Thus, we need to propagate those signals through three levels of hierarchy: module **m14k_core**, module **m14k_cpu** and module **m14k_top**. Follow the process explained in Step 2 - Lab 18, in that case for propagating signals related with the hazard unit. Use in this case a new define `MFP_DEMO_CACHE_HITS_MISSES`.

Step 3. Modify MIPSfpga system RTL files to propagate signals to the LEDs

As explained before, we want to output through the LEDs the clock signal (LD5) and signals *mpc_run_ie* (LD4), *mpc_run_m* (LD3), *mpc_run_w* (LD2), *dcc_dmiss_m* (LD1), and *dcc_pm_dhit_m* (LD0). For that purpose, modify file **system/mpf_sys.v** as explained in Lab 18 and define `MFP_DEMO_CACHE_HITS_MISSES` in file **system/mfp_config.vh**.

4. Exercises

Exercise 1. Sequential accesses – Infer the D$ line size

The following program performs a sequential accesses to an array (*test_data*). The array is aligned to a 16B boundary by using the `__attribute__((aligned(16)))`. Note that the program is totally useless; our only intention is to be able to infer the D$ line size. The source files for the execution on the board are provided in folder *Lab20_BasicCaching\Exercise1\ExecutionOnBoardSources_LineSize*. If you open the *makefile* for this exercise (see *Lab20_BasicCaching\Exercise1\ExecutionOnBoardSources_LineSize\makefile*), you can see that we have removed all optimization options.

```
lui $t6, 0x8000;
addiu $t6, $t6, test_data;
loop:
   lw  $t1, 0($t6);
   lw  $t1, 4($t6);
   lw  $t1, 8($t6);
   lw  $t1, 12($t6);
   lw  $t1, 16($t6);
   lw  $t1, 20($t6);
   lw  $t1, 24($t6);
   lw  $t1, 28($t6);
   addiu $t6, $t6, 32;
   b       loop;
   nop;
```

Explain the results of the execution on the board of the program shown above from the point of view of the data cache, by analyzing the core signals related with the D$ shown on the LEDs. If you are not able to explain all the details of the execution, try to give a rough explanation. Then, after you complete Lab 22, you can come back to this example and provide a more detailed explanation.

Follow the next steps:

1. Modify the soft-core as explained in Section 3.
2. Synthesize the new processor, as explained in Step 3 – Lab 1.
3. Program the FPGA board, as explained in Step 4 – Lab 1.
4. Make sure that switches 0 and 1 are low. Then, download the program on the board, as explained in Step 3 – Section 2 of Lab 2.
5. Once the program is running, turn on switch 0, which activates the slow clock. Observe and explain the results.

Exercise 2. Infer the D$ associativity

Our default processor includes a 2-way D$ with 2KB per Way. In this exercise we execute the following program, which contains an infinite loop with 3 `lw` instructions that access 3 different cache lines that map to the same set, thus generating conflict misses. Note that the program is totally useless; our only intention is to be able to infer the D$ associativity.

```
lui $t6, 0x8000;
addiu $t6, $t6, test_data;
loop:
   lw  $t1, 0($t6);
   nop;
   nop;
   lw  $t1, 2048($t6);
   nop;
```

```
nop;
lw  $t1, 4096($t6);
nop;
nop;
nop;
nop;
nop;
nop;
b       loop;
nop;
```

Explain the results of the execution on the board of the program shown above from the point of view of the data cache. Similarly to the previous exercise, if you do not understand some behaviors, give a rough explanation now and come back to this exercise after completing Labs 21 and 22. The sources for the execution on the board are provided in folder *Lab20_BasicCaching\Exercise2\ExecutionOnBoardSources_Associativity*. If you open the *makefile* for this exercise (see *Lab20_BasicCaching\Exercise2\ExecutionOnBoardSources_Associativity\makefile*), you can see that we have removed all optimization options. For executing the program on MIPSfpga, follow similar steps to Exercise 1.

After performing this test, remove the third `lw` instruction contained in the loop and explain the new results.

Exercise 3. Nested loop in C

Analyze the following C program.

```
while ((MFP_SWITCHES & 4) == 0);

for (i = 0; i < 8; i ++)
      for (j = 0; j < 8; j ++)
            a[i][j] = i + j;
```

Explain the results of the execution on the board of the program shown above from the point of view of the data cache. Like in previous exercises, if you are not able to explain all the details of the execution, try to give a rough explanation, and come back to this exercise after completing Labs 21 and 22. The sources for the execution on the board are provided in folder *Lab20_BasicCaching\Exercise3\ExecutionOnBoardSources_C-Program*. Note that, before performing the accesses to the matrix, the D$ is emptied by using *cache* instructions. Also, if you open the *makefile* for this exercise (see *Lab20_BasicCaching\Exercise3\ExecutionOnBoardSources_C-Program\makefile*), you can see that we are optimizing the code with *-O1* option.

For executing the program on MIPSfpga, follow similar steps to Exercise 1. There is one difference in this exercise: before downloading the program to the board, make sure that switches 0, 1 and **2** are low. Then, when the program starts executing, turn switch 0 on, and then turn switch 2 on, so that the program can enter the nested loop.

After performing this test, change line "`a[i][j]=i+j;`" for "`a[j][i]=i+j;`", execute the new program and explain the results.

5. References

[1] "MIPS32® microAptiv™ UP Processor Core Family Software User's Manual -- MD00942".

[2] "Digital Design and Computer Architecture", 2nd Edition. David Money Harris and Sarah L. Harris. Morgan Kaufmann, 2012.

[3] "Computer Organization and Design", 5th Edition. David A. Patterson and John L. Hennesy. Morgan Kaufmann, 2013.