MIPSfpga
by Imagination

Lab 25

The Memory System – Instruction Scratchpad

Lab 25
The Memory System – Instruction Scratchpad

1. Introduction

In this lab you incorporate an instruction scratchpad memory (ISPRAM) into MIPSfpga and analyze its performance using performance counters. A scratchpad memory is usually aimed at storing critical blocks of code that need to be retrieved with a small and predictable latency. As opposed to a cache memory, which is completely managed by hardware and thus totally transparent to the programmer, a scratchpad memory is directly managed by the programmer or the compiler. For you to understand the main concepts of scratchpad memories you can read [1], whereas to understand the scratchpad memory interface implemented in microAptiv, you can read Chapter 7 of [2].

2. ISPRAM Implementation

In this section we guide you through the process of including an instruction scratchpad memory (ISPRAM) in MIPSfpga. Our goal is to implement a 4KB *virtually-accessed physically-tagged* ISPRAM, mapped to the address range from 0x00040000 to 0x00040FFC. Along this lab we assume a 4KB 2-Way Set-Associative I$ (i.e. 2KB per way). Figure 1 illustrates the new physical memory map of our system.

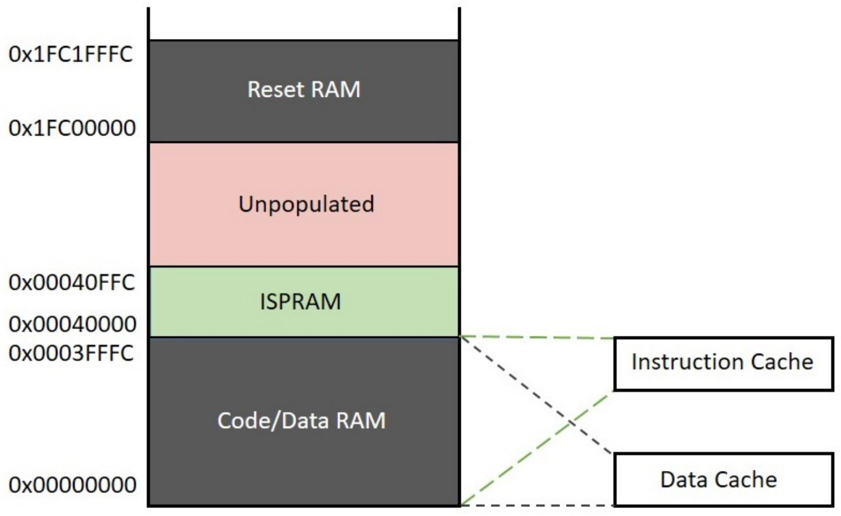

Figure 1. Physical memory map including the ISPRAM.

a. Basic configuration

Follow the next steps to configure MIPSfpga adequately:

1. Copy the original soft-core folder (**rtl_up**) into a new folder named **rtl_up_ISPRAM**.
2. In the new folder, expand the capability of the MIPSfpga system so that it can write to the 7-segment displays on the Nexys4 DDR board, as explained in Lab 5.
3. Assuming that our processor uses a 4KB 2-Way Set-Associative I$ (i.e. 2KB per way), you must set the following parameters in file **config.vh**:

   ```
   `define M14K_ICACHE_ASSOC 2
   `define M14K_ICACHE_WAYSIZE 2
   `define M14K_MAX_IC_ASSOC 3
   ```

   Note that the I$ associativity (M14K_ICACHE_ASSOC) must be configured as the number of ways in the I$, whereas the maximum I$ associativity (M14K_MAX_IC_ASSOC) must be configured as the number of ways in the I$ plus 1, where you also take into account the ISRPAM.
4. Module **m14k_icc_spstub**, instantiated at **m14k_icc**, will be used for communicating the ISPRAM banks with the instruction cache controller (i.e. we can consider it as the ISPRAM cache controller). This is a stubbed module in MIPSfpga.
   Copy file **m14k_icc_spstub.v** into a new file named **m14k_icc_sp.v** and rename the module **m14k_icc_spstub** as **m14k_icc_sp**.
   Redefine the following parameter, in file **config.vh**, as follows:

   ```
   `define M14K_ISPRAM_MODULE m14k_icc_sp
   ```

5. Module **m14k_ispram_ext_stub**, instantiated at module **m14k_spram_top** which is instantiated at module **m14k_top**, will be used for implementing the ISPRAM memory banks. This is a stubbed module in MIPSfpga.
   Copy file **m14k_ispram_ext_stub.v** into a new file named **m14k_ispram_ext.v** and rename module **m14k_ispram_ext_stub** to **m14k_ispram_ext**.
   Redefine the following parameter in file **config.vh** as follows:

   ```
   `define M14K_ISPRAM_EXT_MODULE m14k_ispram_ext
   ```

b. ISPRAM Controller (module m14k_icc_sp)

Although MIPSfpga does not include scratchpad memories by default, the instruction/data cache controller incorporates the support required to include a scratchpad memory. Lab 22-A explains how the *A* instructions (being *A* the I$ associativity) read from the I$ Data Array (instantiated within module **m14k_ic**) are sent to the instruction cache controller (module **m14k_icc**) through signal *ic_datain[63:0]*. This signal is communicated within the instruction cache controller to the **m14k_icc_sp** module. Within that module, this signal (*ic_datain[63:0]*) must be concatenated with the instruction read from the instruction scratchpad memory (**m14k_ispram_ext**), provided in signal *ISP_DataRdValue[31:0]*, into signal *spram_cache_data[95:0]*, as shown in Figure 2. Thus, modify module **m14k_icc_sp** as follows:

```
        assign spram_cache_data [D_BITS*`M14K_MAX_IC_ASSOC-1:0] = {ISP_DataRdValue,
ic_datain[D_BITS*`M14K_ICACHE_WAYSIZE-1:0]};
```

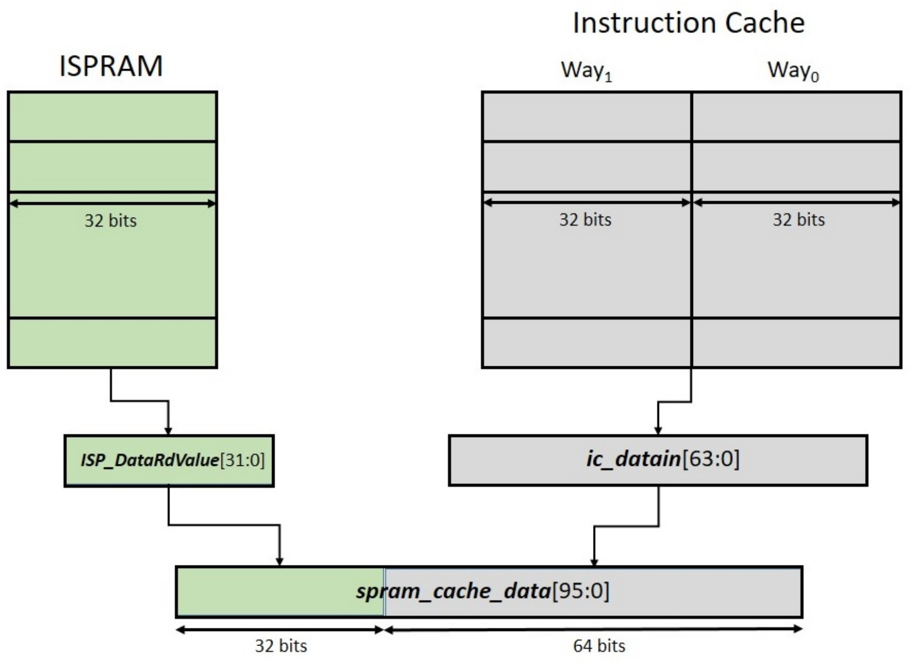

Figure 2. Concatenation of instructions from 2-way I$ and ISPRAM into *spram_cache_data*.

Signal *spram_cache_data[95:0]* is then passed to the instruction cache controller (**m14k_icc**), where it is assigned to *cache_recode[191:48]*, after traversing other signals. This signal is used as input to the multiplexer *datamux* as explained in Lab 22. However, this multiplexer can now select the fetched instruction from the I$ or from the ISPRAM, as illustrated in Figure 3. Note that all this functionality is already available in the instruction cache controller, so no changes are needed.

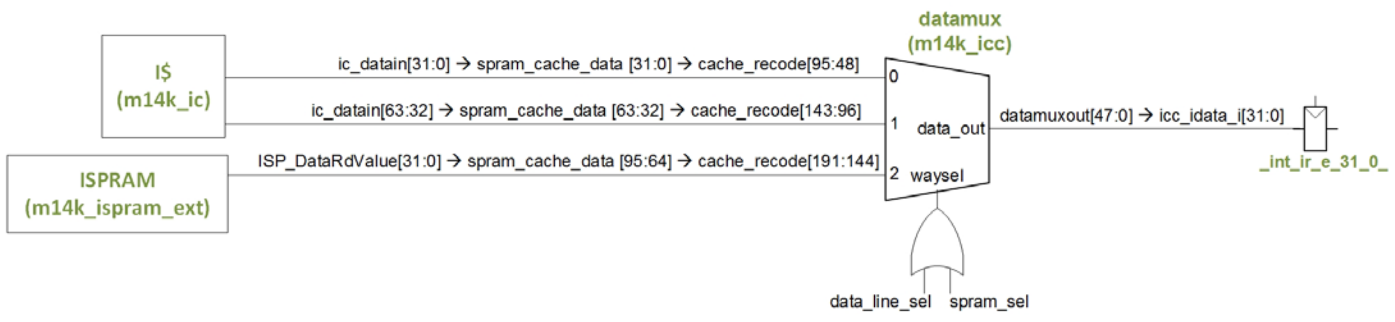
datamux
(m14k_icc)
I$
(m14k_ic)
ic_datain[31:0] → spram_cache_data [31:0] → cache_recode[95:48]
ic_datain[63:32] → spram_cache_data [63:32] → cache_recode[143:96]
ISPRAM
(m14k_ispram_ext)
ISP_DataRdValue[31:0] → spram_cache_data [95:64] → cache_recode[191:144]
0
1
2
data_out
waysel
datamuxout[47:0] → icc_idata_i[31:0]
_int_ir_e_31_0_
data_line_sel
spram_sel

Figure 3. Selection of instruction from the I$ or from the ISPRAM.

For the sake of consistency, you should do the same concatenation for the tags. Thus, within module **m14k_icc_sp**, signal *ic_tagin[47:0]* must be concatenated with the tag read from the instruction scratchpad memory (**m14k_ispram_ext**), provided in signal *ISP_TagRdValue[23:0]*, into *spram_cache_tag[71:0]*. Thus, modify module **m14k_icc_sp** as follows:

```
        assign spram_cache_tag [T_BITS*`M14K_MAX_IC_ASSOC-1:0] = {ISP_TagRdValue,
ic_tagin[T_BITS*`M14K_ICACHE_WAYSIZE-1:0]};
```

As for the control signal used in the *datamux* multiplexer shown in Figure 3, observe that it is computed as the *OR* operation between signal *data_line_sel[3:0]*, computed in the **m14k_cache_cmp** module as explained in Lab 22-A, and signal *spram_sel[3:0]*, assigned from signal *spram_sel_raw[3:0]*, which you must compute in module **m14k_icc_sp**. Signal *spram_sel_raw[3:0]* must be *4'b0100* if the processor requests an instruction within the ISPRAM address region, and *4'b0* otherwise. You can detect if the processor is accessing the ISPRAM by comparing signals *ISP_TagRdValue[23:4]*, which stores the *tag* contained in the ISPRAM, and *icc_tagcmpdata[31:12]*, which stores the *tag* requested by the processor. Thus, modify module **m14k_icc_sp** as follows:

```
   assign spram_sel_raw[3:0] = (icc_tagcmpdata[31:12]==ISP_TagRdValue[23:4])? 4'b0100 : 4'b0;
```

Finally, you must also compute some other signals as explained below:

- The index used to access the ISPRAM, signal *ISP_Addr[19:2]*, is provided from the instruction cache controller through signal *icc_dataaddr[13:2]*. Thus, assign *ISP_Addr[19:2]* as follows (note that the six most significant bits can be set to 0):
  ```
        assign ISP_Addr[19:2] = {6'b0,icc_dataaddr[13:2]};
  ```
- Signal *raw_isp_hit* must be set to 1 when there is a hit in the ISPRAM and 0 otherwise (you can use signal *spram_sel_raw*). For that purpose, compute signal *raw_isp_hit* as follows:
  ```
        assign raw_isp_hit = | spram_sel_raw;
  ```
- Signal *icc_sp_pres* determines the presence of the ISPRAM, and must be assigned from signal *ISP_Present*, provided from module **m14k_ispram_ext**, as follows:
  ```
        assign icc_sp_pres = ISP_Present;
  ```
- Signal *icc_sptrstb_int* provides the read strobe for the ISPRAM as determined in the instruction cache controller. It must be assigned to signal *ISP_RdStr* as follows:
  ```
        assign ISP_RdStr = icc_sptrstb;
  ```

c. ISPRAM Data and Tag Arrays (module m14k_ispram_ext)

In the **m14k_ispram_ext** module you must include the arrays constituting the scratchpad memory (analogously to the **m14k_ic** module for the I$). Specifically, signal *ISP_TagRdValue*, which constitutes the 'ISPRAM Tag Array', must hold the address range that is mapped to the ISPRAM. In our implementation (Figure 1) this value must be assigned as follows:

```
assign ISP_TagRdValue [23:0] = 24'h000400;
```

As for the ISPRAM Data Array, given that the size of our ISPRAM is 4KB, you must instantiate, at module **m14k_ispram_ext**, a new wrapper (**dataram_4k1way_xilinx.v**) and a new bank (**RAMB8K_S8.v**), which you can find at folder *Lab21_CacheStructure\NewCacheWrappers*. Copy those two files into folder *rtl_up_ISPRAM\core* and include the following instantiation in module **m14k_ispram_ext**:

```
dataram_4k1way_xilinx dataram (
      .clk( SI_ClkIn ),
      .line_idx( ISP_Addr[11:2] ),
      .rd_mask( 1'b1 ),
      .wr_mask( 4'b1111 ),
      .rd_str( ISP_RdStr ),
      .wr_str( ISP_DataWrStr  ),
      .early_ce( 1'b1 ),
      .wr_data( ISP_DataTagValue ),
      .rd_data( ISP_DataRdValue ),
      .bist_to( 1'b1 ),
      .bist_from(  )
);
```

Moreover, given that *ISP_DataTagValue* is assigned in the new wrapper module, you must delete or just comment the following assignment:

```
assign ISP_DataRdValue [31:0] = 32'h0;
```

Finally, set signal *ISP_Present* to 1 in this module, notifying to the Core that an ISPRAM is implemented.

```
assign ISP_Present = 1'b1;
```

d. Initialization of the ISPRAM

The ISPRAM Tag and the ISPRAM Data Array must be initialized before accessing them. In the previous subsection we explained how the tag is initialized in our system (`assign ISP_TagRdValue [23:0] = 24'h000400`). As for the Data Array, in this lab we initialize it by using the *cache* instruction. You can find the detailed description of the *cache* instruction in pages 324-329 of [3]; we do not go into details here but just provide the implementation. As such, follow the next steps:

- The *cache* instruction provides the value that must be written to the cache through signal *fill_data_raw*. This value must be assigned to signal *ISP_DataTagValue*. Thus, include the following assignment in module **m14k_icc_sp**:

```
assign ISP_DataTagValue[31:0] = fill_data_raw[31:0];
```

- For implementing the ISPRAM write strobe, which controls ISPRAM writing, we follow a simple approach. Specifically, we change the interface of the **m14k_icc_sp** module, by including signal *cpz_spram* as a new input. Thus, in the instantiation of the **m14k_icc_sp** module (line 1161 of the **m14k_icc** module), include signal *cpz_spram* as a new input, and declare this input in the **m14k_icc_sp** module itself.
  This signal determines if a *cache* instruction must access the I$ (in which case *cpz_spram*=0) or the ISPRAM (in which case *cpz_spram*=1). Based on this signal, the ISPRAM strobe (signal *ISP_DataWrStr*), is assigned as the *AND* operation between this signal (*cpz_spram*) and signal *icop_write* (which is 1 when the processor executes a *cache* write instruction). For that purpose, include the following assignment in module **m14k_icc_sp**:

```
assign ISP_DataWrStr = icop_write&cpz_spram;
```

3. Exercise

The aim of this exercise is to implement a 4KB *virtually-indexed physically tagged* instruction scratchpad memory in MIPSfpga and then compare the execution of the algorithm implemented in Exercise 6.35 of [4] in the processor including an ISPRAM and in the baseline processor. This algorithm performs floating point addition of two positive single precision floating point numbers, held in two temporal registers. If you need to review the IEEE754 single-precision format, you can go back to Lab 19, where this format is explained in Exercise 6. Recall that you can use the converter available in https://www.h-schmidt.net/FloatConverter/IEEE754.html. Follow the next steps:

1. Modify the soft-core as explained in Section 2. Besides, as in previous labs, it is essential to modify the MIPSfpga system (**rtl_up_ISPRAM**) as explained in Lab 5, in order to provide support for the 7-segment displays.
2. Create a new Vivado project (**project_ISPRAM**), following the instructions provided in Step 1 - Lab 1, using the source files in the new folder (**rtl_up_ISPRAM**).
3. Analyze in detail the following two folders, which include the implementation of the algorithm from Exercise 6.35 of [4], executing in the original processor and in the new ISPRAM-based processor:
   a. The source files provided in folder *Lab25_ScratchpadRAM\Simulations\SimulationSources\SimulationSources_WithoutISPRAM* execute the algorithm as usual, reading the instructions from the I$. Note that performance counters are initialized before executing the algorithm, measuring the number of cycles and instructions completed,

and they are read afterwards. After executing the algorithm, the result and the value of the two performance counters is shown on the 7-segment displays.

   b. The source files provided in folder *Lab25_ScratchpadRAM\Simulations\SimulationSources\SimulationSources_ISPRAM* first initialize the ISPRAM Data Array via *cache* instructions, using function *ISPRAM_Init()*. Once the ISPRAM has been initialized, the algorithm is executed from the ISPRAM, by jumping to the address range mapped to that memory (`jal 0x40000`). Note that, given that we are executing the program from the ISPRAM as if it was a subroutine, we must stick to the MIPS convention for calling functions, explained in Section 6.4.6 of [4]. Note also that performance counters are initialized before executing the `jal` instruction, measuring the number of cycles and instructions completed, and they are read afterwards. After executing the algorithm, the result and the value of the two performance counters is shown on the 7-segment displays.

   You can generate the executable files following the instructions from Section 7.2 of the Getting Started Guide, by using the *make* command.

4. Simulate both programs on Vivado's XSIM. You can use the waveform configuration file provided in folder *Lab25_ScratchpadRAM\Simulations*.
5. Execute both programs on the FPGA board. Follow the next steps:
   a. Step 1 – Synthesize the new processor, as explained in Step 3 – Lab 1.
   b. Step 2 – Program the FPGA board, as explained in Step 4 – Lab 1.
   c. Step 3 – Execute the two programs on the board, as explained in Step 3 – Section 2 of Lab 2, and explain the results.

4. References

[1] "Memory Systems: Cache, DRAM, Disk". Bruce Jacob, Spencer W. Ng and David T. Wang. Morgan Kaufmann", 2007.

[2] "MIPS32® microAptiv™ UP Processor Core Family Integrator's Guide -- MD00941".

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

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