MIPSfpga

by Imagination

Lab 21

The Memory System – Cache Structure

Lab 21
The Memory System – Cache Structure

1. Introduction

In this lab, we analyze the different arrays that make up the cache, we create new cache configurations and we test them with the performance counters. Both caches in microAptiv (I$ and D$) are made up by 3 arrays: the Data Array, which stores the data/instructions, the Tag Array, which stores part of the physical address of each block stored in the Data Array, and the Way-Select Array, which stores additional information associated with each block.

In this lab we focus on the D$ as it is more complex and sophisticated than the I$. This is mainly due to the fact that the D$ must handle not only reads but also writes from the processor to the cache, as opposed to the I$, which is never written from the core. Once you understand the D$, you won´t have any problems analyzing the I$ on your own.

2. D$ Structure: Data, Tag and WS Arrays

In this section we analyze the arrays that make up the default D$ (2-Way Set-Associative D$ with 2KB per Way). This cache is implemented in module **m14k_dc** and is divided in three arrays: the Data Array, which stores the data and is implemented in the wrapper module **dataram_2k2way_xilinx**, the Tag Array, which stores the physical addresses of the data stored in the Data Array and is implemented in the wrapper module **tagram_2k2way_xilinx**, and the Way Select (WS) Array, which stores information about dirtiness and replacement policy state of the cache blocks (sometimes referred to as cache lines) and is implemented in the wrapper module **d_wsram_2k2way_xilinx**. Given that our processor is implemented in an FPGA, these three wrapper modules (**dataram_2k2way_xilinx**, **tagram_2k2way_xilinx** and **d_wsram_2k2way_xilinx**) instantiate generic Xilinx RAM models which allow the cache arrays to be mapped to the block RAM of the FPGA.

The D$ (**m14k_dc**) receives signals from the Data Cache Controller (**m14k_dcc**), such as the virtual address to read from or to write to (*data_addr[13:2]*, *tag_addr[13:4]*, *ws_addr[13:4]*), or the data to write (*wr_data[D_BITS-1:0]*, *tag_wr_data[T_BITS-1:0]*, *ws_wr_data[13:0]*), and sends back other signals to the Data Cache Controller, such as the data read (*rd_data[(D_BITS*`M14K_MAX_DC_ASSOC-1):0]*, *tag_rd_data[(T_BITS*`M14K_MAX_DC_ASSOC-1):0]*, *ws_rd_data[13:0]*). These signals are distributed within the D$ module among the different arrays, as explained in the following subsections.

a. Data Array

In this section we explain the interface and implementation of the D$ Data Array, which stores the data. The following fragment, extracted from module **m14k_dc**, shows the instantiation of the Data Array (**dataram_2k2way_xilinx**).

```
`M14K_DC_DATARAM dataram (
        .clk( gclk ),
        .line_idx( data_addr[(2 + DATA_DEPTH -1):2] ),
        .rd_mask({ASSOC{1'b1}}),
        .wr_mask( wr_mask[(4*ASSOC)-1:0] ),
        .rd_str( data_rd_str ),
        .wr_str( data_wr_str ),
        .wr_data(wr_data),
        .rd_data( data_rd_data ),
        .early_ce(early_data_ce),
        .bist_to( data_bist_to ),
        .bist_from( data_bist_from )
);
```

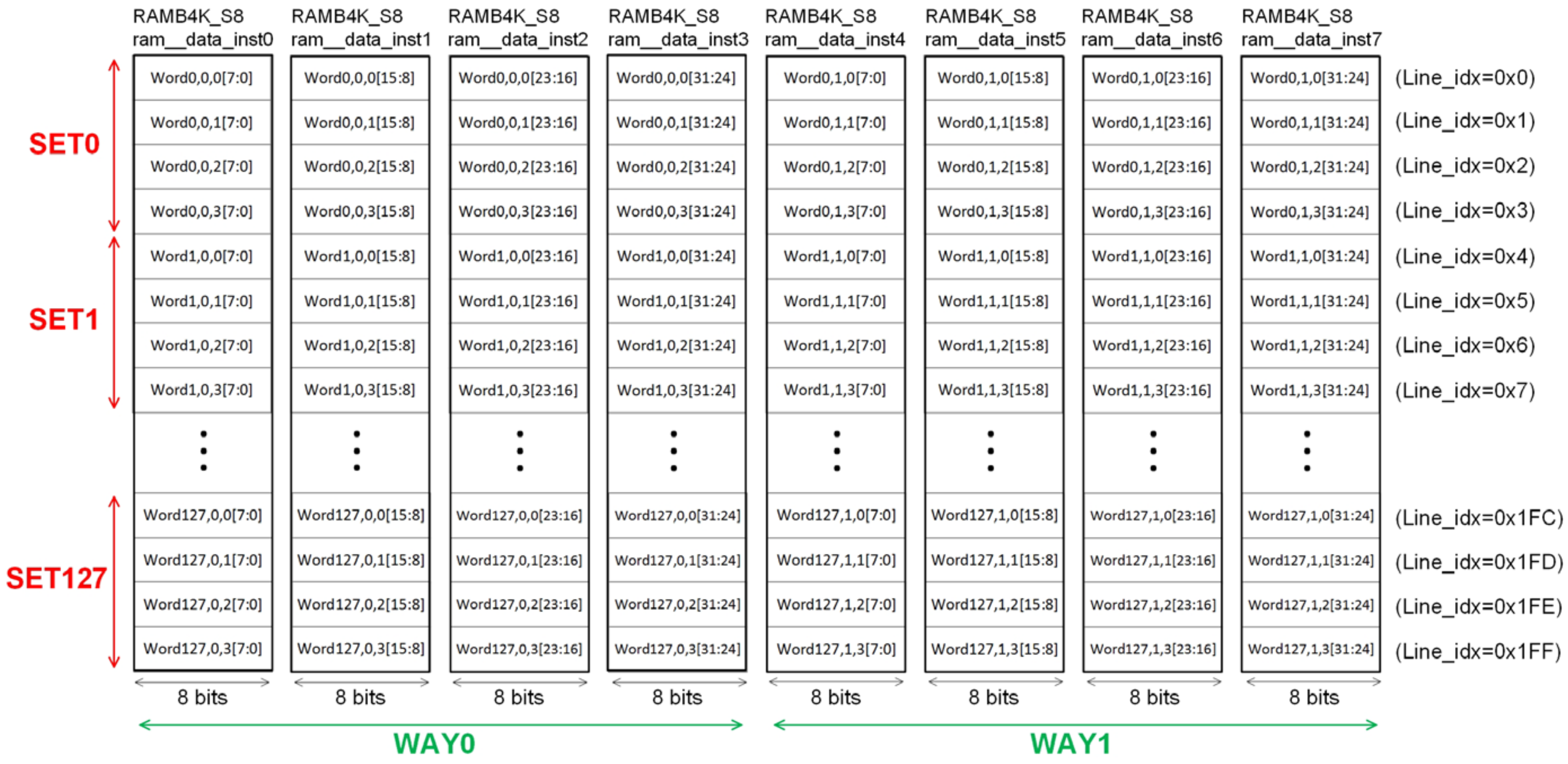
Table 1 explains the main signals communicated from the core to the D$ Data Array (inputs) and from the D$ Data Array to the core (outputs).

Table 1. Main signals in the D$ Data Array interface with the Core

Direction	Signal name	Description
Inputs	*data_addr[(2+DATA_DEPTH-1):2]*	✓ Used for indexing one word per way in the Data Array ✓ It is obtained from a subset of the Effective Address computed by the core at the E-Stage, as explained in Lab 16 (the caches in microAptiv are virtually indexed) ✓ For the default 4KB 2-Way D$ configuration, *DATA_DEPTH* is equal to 9. Thus, only bits [10:2] of *data_addr* are used for indexing the Data Array. Observe that the least 2 significant bits of the Effective Address are not used, as the access to the Data Array is word-aligned
	*rd_mask[(ASSOC)-1:0] / wr_mask[(4*ASSOC)-1:0]*	✓ *rd_mask* determines which way must be read. Given that all ways are always read, this signal is fixed to 1s ✓ The write mask determines the specific byte/s to update in case of a write operation
	data_rd_str / data_wr_str	✓ These signals enable (1) or disable (0) read / write operations
	wr_data[31:0]	✓ Provides the bits to write to the Data Array in case of a store instruction from the core or a fill operation from main memory
Output	*data_rd_data[63:0]*	✓ Contains the bits read from each way of the 2-way D$

The wrapper module **dataram_2k2way_xilinx** (defined as M14K_DC_DATARAM) implements the Data Array of a 4KB 2-way D$. For efficiency reasons, the Data Array is subdivided in several generic Xilinx memory banks that can be read in parallel. Specifically, it is formed by eight 512B banks (eight instantiations of the generic Xilinx RAM model RAMB4K_S8), for a total of 4KB memory space (2KB per way). Figure 1 illustrates the Data Array organization:

- Each element in Figure 1 is identified by: set, way, position within the cache block, and bit range within the word. For example: *Word1,0,2[15:8]* refers to the bits 8 to 15 of the third word of the block stored in set 1, way 0.
- Each column represents one independent RAMB4K_S8 bank. The first 4 banks (*ram__data_inst0* to *ram__data_inst3*) store the first way (Way_0) and the second 4 banks (*ram__data_inst4* to *ram__data_inst7*) store the second way (Way_1).
- Each row represents 2 words, each one belonging to a different way. Observe that there are 512 rows in Figure 1, as there are 512 words per way. Note that each word is distributed across the 4 banks of the way it belongs to.
- 4 rows represent one set. There are 128 sets in the D$. Recall that, in the default D$, each set is formed by 2 cache lines, each one containing 4 words.



RAMB4K_S8 ram__data_inst0	RAMB4K_S8 ram__data_inst1	RAMB4K_S8 ram__data_inst2	RAMB4K_S8 ram__data_inst3	RAMB4K_S8 ram__data_inst4	RAMB4K_S8 ram__data_inst5	RAMB4K_S8 ram__data_inst6	RAMB4K_S8 ram__data_inst7	
Word0,0,0[7:0]	Word0,0,0[15:8]	Word0,0,0[23:16]	Word0,0,0[31:24]	Word0,1,0[7:0]	Word0,1,0[15:8]	Word0,1,0[23:16]	Word0,1,0[31:24]	(Line_idx=0x0)
Word0,0,1[7:0]	Word0,0,1[15:8]	Word0,0,1[23:16]	Word0,0,1[31:24]	Word0,1,1[7:0]	Word0,1,1[15:8]	Word0,1,1[23:16]	Word0,1,1[31:24]	(Line_idx=0x1)
Word0,0,2[7:0]	Word0,0,2[15:8]	Word0,0,2[23:16]	Word0,0,2[31:24]	Word0,1,2[7:0]	Word0,1,2[15:8]	Word0,1,2[23:16]	Word0,1,2[31:24]	(Line_idx=0x2)
Word0,0,3[7:0]	Word0,0,3[15:8]	Word0,0,3[23:16]	Word0,0,3[31:24]	Word0,1,3[7:0]	Word0,1,3[15:8]	Word0,1,3[23:16]	Word0,1,3[31:24]	(Line_idx=0x3)
Word1,0,0[7:0]	Word1,0,0[15:8]	Word1,0,0[23:16]	Word1,0,0[31:24]	Word1,1,0[7:0]	Word1,1,0[15:8]	Word1,1,0[23:16]	Word1,1,0[31:24]	(Line_idx=0x4)
Word1,0,1[7:0]	Word1,0,1[15:8]	Word1,0,1[23:16]	Word1,0,1[31:24]	Word1,1,1[7:0]	Word1,1,1[15:8]	Word1,1,1[23:16]	Word1,1,1[31:24]	(Line_idx=0x5)
Word1,0,2[7:0]	Word1,0,2[15:8]	Word1,0,2[23:16]	Word1,0,2[31:24]	Word1,1,2[7:0]	Word1,1,2[15:8]	Word1,1,2[23:16]	Word1,1,2[31:24]	(Line_idx=0x6)
Word1,0,3[7:0]	Word1,0,3[15:8]	Word1,0,3[23:16]	Word1,0,3[31:24]	Word1,1,3[7:0]	Word1,1,3[15:8]	Word1,1,3[23:16]	Word1,1,3[31:24]	(Line_idx=0x7)
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	
Word127,0,0[7:0]	Word127,0,0[15:8]	Word127,0,0[23:16]	Word127,0,0[31:24]	Word127,1,0[7:0]	Word127,1,0[15:8]	Word127,1,0[23:16]	Word127,1,0[31:24]	(Line_idx=0x1FC)
Word127,0,1[7:0]	Word127,0,1[15:8]	Word127,0,1[23:16]	Word127,0,1[31:24]	Word127,1,1[7:0]	Word127,1,1[15:8]	Word127,1,1[23:16]	Word127,1,1[31:24]	(Line_idx=0x1FD)
Word127,0,2[7:0]	Word127,0,2[15:8]	Word127,0,2[23:16]	Word127,0,2[31:24]	Word127,1,2[7:0]	Word127,1,2[15:8]	Word127,1,2[23:16]	Word127,1,2[31:24]	(Line_idx=0x1FE)
Word127,0,3[7:0]	Word127,0,3[15:8]	Word127,0,3[23:16]	Word127,0,3[31:24]	Word127,1,3[7:0]	Word127,1,3[15:8]	Word127,1,3[23:16]	Word127,1,3[31:24]	(Line_idx=0x1FF)
8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	8 bits	

Figure 1. Data Array organization for the default D$.

The following code fragment shows the instantiation of the first RAMB4K_S8 bank (*ram__data_inst0*) at module **dataram_2k2way_xilinx**. All interface signals are common to all banks (for example, *line_idx*) except for *wr_mask[i]*, *wr_data[j:k]* and *rd_data[j:k]*.

```
// 512 x 8
RAMB4K_S8 ram__data_inst0 (
        .WE     (wr_str && wr_mask[0]),
        .EN     (en[0]),
        .RST    (1'b0),
        .CLK    (clk),
        .ADDR   (line_idx),
        .DI     (wr_data[7:0]),
        .DO     (rd_data[7:0])
);
```

Finally, the bank implementation is shown in the next code fragment (extracted from module RAMB4K_S8). All banks used in the D$ (Data Array, Tag Array and WS Array) are of size 512B, but they are organized differently. For example, the RAMB4K_S8 bank implements 512 8-bit-wide entries. Thus, as illustrated in Figure 1, each bank stores ¼ of a word.

```
module RAMB4K_S8(WE, EN, RST, CLK, ADDR, DI, DO);
    input              WE;
    input              EN;
    input              RST;
    input              CLK;
    input           [8:0]  ADDR;
    input            [7:0] DI;
    output           [7:0] DO;

    reg              [7:0] mem[0:511];
    reg              [7:0] DO;

    always @(posedge CLK) begin
      DO <= #1 mem[ADDR];
      if (EN)
        begin
              if (WE)
               mem[ADDR] <=#1  DI;
        end
    end
endmodule
```

b. Tag Array

In this section we explain the D$ Tag Array, which stores part of the physical address of each block stored in the Data Array. The following fragment, extracted from module **m14k_dc**, shows the instantiation of the Tag Array (**tagram_2k2way_xilinx**).

```
`M14K_DC_TAGRAM tagram (
        .clk( gclk ),
        .greset( greset),
        .line_idx( tag_addr[(4+TAG_DEPTH)-1:4] ),
        .wr_mask( tag_wr_en[(ASSOC)-1:0] ),
        .rd_str( tag_rd_str ),
        .wr_str( tag_wr_str ),
        .wr_data( tag_wr_data ),
        .rd_data( tag_rd_int[T_BITS*ASSOC-1:0] ),
        .early_ce(early_tag_ce),
        .hci( hci),
        .bist_to( tag_bist_to ),
        .bist_from( tag_bist_from )
);
```

Table 2 explains the main signals communicated from the core to the D$ Tag Array (inputs) and from the D$ Tag Array to the core (outputs).

Table 2. Main signals in the D$ Tag Array interface with the Core

Direction	Signal name	Description
Inputs	*tag_addr[(4+TAG_DEPTH-1):4]*	✓ Used for indexing one tag per way in the Tag Array ✓ It is obtained from a subset of the Effective Address computed by the core at the E-Stage, as explained in Lab 16 ✓ For the default 4KB 2-Way D$ configuration, *TAG_DEPTH* is equal to 7. Thus, only bits [10:4] of *tag_addr* are used for indexing the Tag Array. Observe that the least 4 significant bits of the Effective Address are not used in the index, as the access to the Tag Array is block-aligned
	tag_wr_en[(ASSOC)-1:0]	✓ The write mask determines the tag to update by a write operation
	tag_rd_str / *tag_wr_str*	✓ These signals enable (1) or disable (0) read / write operations
	tag_wr_data[23:0]	✓ Provides the 24 bits to write to the Tag Array o PA: 22 bits to store part of the Physical Address of the block o Lock bit: 1 bit to indicate to the Cache Controller if the block should be considered for eviction (unlocked) or should be always maintained in the cache (locked) o Valid bit: 1 bit to indicate to the Cache Controller if the block is valid or not
Output	*tag_rd_int[48-1:0]*	✓ Contains the 24 bits read from each way of the 2-way D$

The wrapper module **tagram_2k2way_xilinx** (defined as M14K_DC_TAGRAM) implements the Tag Array of a 4KB 2-way D$. As the Data Array, the Tag Array is subdivided in several generic Xilinx memory banks which can be read in parallel. Specifically, it is formed by four 512B banks (four instantiations of the generic Xilinx RAM model RAMB4K_S16), for a total of 2KB memory space (1KB per way). This bank size is used because it is the most convenient one among the different bank configurations provided by Xilinx. However, as we will see below, more than half of the bits are wasted. Figure 2 illustrates the Tag Array organization:

- Each element in Figure 2 is identified by: set, way, bit range within the tag. Note that comparing with the Data Array (Figure 1), there is not a *position within the cache block* field here, as there is only one tag for each block. Thus, for example: *Tag1,0[15:8]* refers to the bits 8 to 15 of the tag stored in set 1, way 0.
- Each column represents one independent RAMB4K_S16 bank. The first 2 banks (*ram__tag_inst0* and *ram__tag_inst1*) store the tags in the first way (Way$_0$) and the second 2 banks (*ram__tag_inst2* to *ram__tag_inst3*) store the tags in the second way (Way$_1$).
- Each row represents the two tags within one set. Observe that there are 128 rows in Figure 2, as there are 128 tags per way, or 128 sets. Note that each tag is distributed across the 2 banks of the way it belongs to.

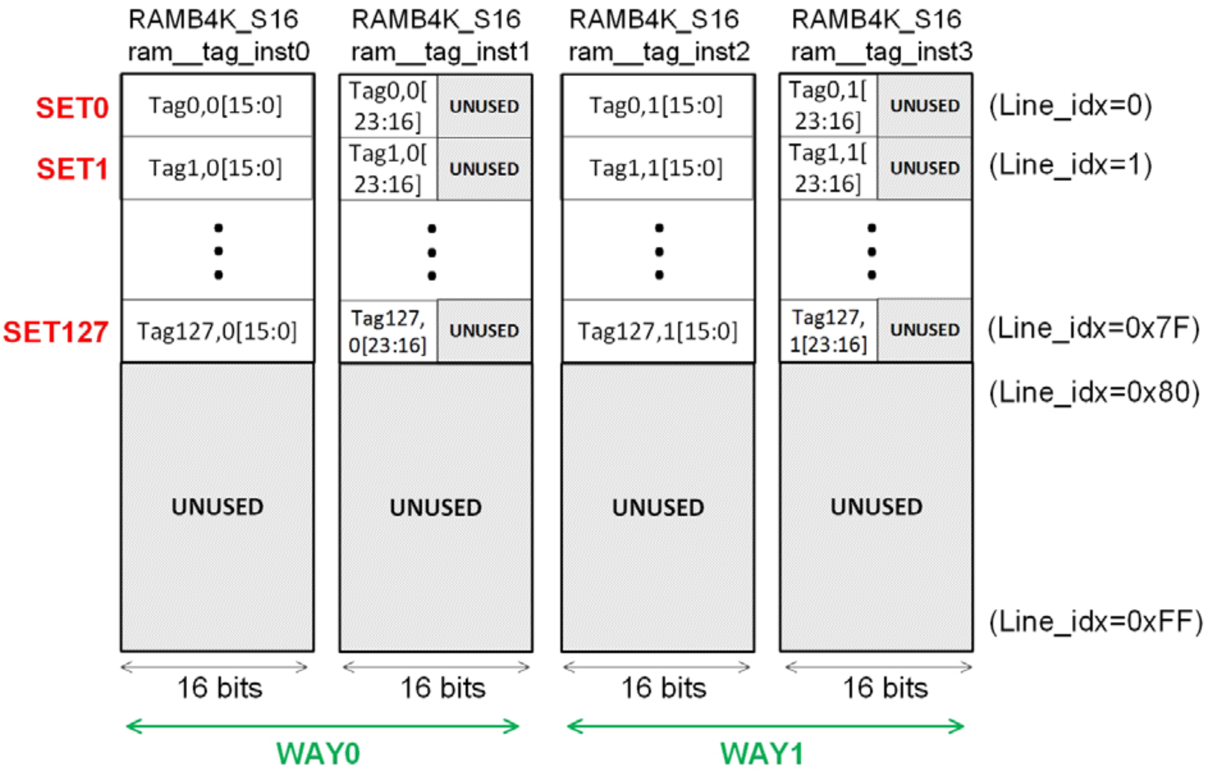


Figure 2. Tag Array organization for the default D$.

The following code fragment shows the instantiation of the first RAMB4K_S16 bank (*ram__tag_inst0*) at module **tagram_2k2way_xilinx**. All interface signals are common to all banks (for example, *line_idx*) except for *wr_mask[i]*, *wr_data[j:k]* and *rd_data[j:k]*. Observe that a 0 is added to the left-most bit of the Tag Array index ({1'b0,*line_idx*}), thus we are always accessing the upper half of it.

```
// 256 x 16 (We only need 128 x 16 but this is what Xilinx got)
RAMB4K_S16 ram__tag_inst0 (
        .WE     (wr_str && wr_mask[0]),
        .EN     (en[0]),
        .RST    (1'b0),
        .CLK    (clk),
        .ADDR   ({1'b0,line_idx}),
        .DI     (wide_wr_data[15:0]),
        .DO     (wide_rd_data[15:0])
);
```

Finally, the bank implementation is shown in the next code fragment (extracted from module RAMB4K_S16). Each RAMB4K_S16 bank implements 256 16-bit-wide memory entries, each one storing ½ of a tag.

```
module RAMB4K_S16(WE, EN, RST, CLK, ADDR, DI, DO);
    input                  WE;
    input                  EN;
    input                  RST;
    input                  CLK;
    input           [7:0]  ADDR;
    input            [15:0]       DI;
    output           [15:0]       DO;

    reg              [15:0]       mem[0:255];
    reg              [15:0]       DO;
    always @(posedge CLK) begin
      DO <= #1 mem[ADDR];
      if (EN)
        begin
              if (WE)
               mem[ADDR] <=#1  DI;
        end
    end
endmodule
```

c. WS Array

In this section we explain the D$ Way Select (WS) Array, which stores additional information associated with each block. The following fragment, extracted from module **m14k_dc**, shows the instantiation of the WS Array (**d_wsram_2k2way_xilinx**).

```
`M14K_DC_WSRAM wsram (
        .clk( gclk ),
        .greset( greset),
        .line_idx( ws_addr[(4+TAG_DEPTH)-1:4] ),
        .wr_mask( ws_wr_mask_short ),
        .rd_str( ws_rd_str ),
        .wr_str( ws_wr_str ),
        .wr_data( ws_wr_data_short ),
        .rd_data( ws_rd_data_short ),
        .early_ce(early_ws_ce),
        .bist_to( ws_bist_to ),
        .bist_from( ws_bist_from )
```

```
    );
```

Table 3 explains the main signals communicated from the core to the D$ WS Array (inputs) and from the D$ WS Array to the core (outputs).

Table 3. Main signals in the D$ WS Array interface with the Core

Direction	Signal name	Description
Inputs	*ws_addr[(4+TAG_DEPTH-1):4]*	✓ Used for indexing one entry per set in the WS Array ✓ It is obtained from a subset of the Effective Address computed by the core at the E-Stage, as explained in Lab 16 ✓ For the default 4KB 2-Way D$ configuration, *TAG_DEPTH* is equal to 7. Thus, only bits [10:4] of *tag_addr* are used for indexing the Tag Array. Observe that the least 4 significant bits of the Effective Address are not used in the index, as the access to the Tag Array is block-aligned
	ws_wr_mask_short	✓ The write mask determines the bits to update by a write operation
	ws_rd_str / *ws_wr_str*	✓ These signals enable (1) or disable (0) read / write operations
	ws_wr_data_short[(WS_WIDTH-1):0]	✓ Provides the 3 bits to write to the WS Array o LRU state: 1 bit to indicate to the cache controller the LRU state of the set o Dirty state: 2 bits to indicate to the cache controller the dirtiness state of each block within a set
Output	*ws_rd_data_short[(WS_WIDTH-1):0]*	✓ Contains the 3 bits read from the WS Array

The wrapper module **d_wsram_2k2way_xilinx** (defined as M14K_DC_WSRAM) implements the WS Array of a 4KB 2-way D$. As the Data and Tag Arrays, the WS Array is subdivided in several generic Xilinx memory banks which can be read in parallel. Specifically, it is formed by three 512B banks (three instantiations of the generic Xilinx RAM model RAMB4K_S2), for a total of 1.5KB memory space. This bank size is used because it is the most convenient one among the different bank configurations provided by Xilinx. However, it would be enough using a much smaller size (128*3 bits, as we only need one 3-bits entry per set). Figure 3 illustrates the WS Array organization:

- Each pair of 1-bit columns represents one independent RAMB4K_S2 bank. The first bank (*d_wsram___inst0*) stores the LRU state of the set, and the second and third banks (*d_wsram___inst1* and *d_wsram___inst2*) store the dirtiness state of each block in the set.
- Each row represents the information relative to a set (there are 128 sets in the D$).

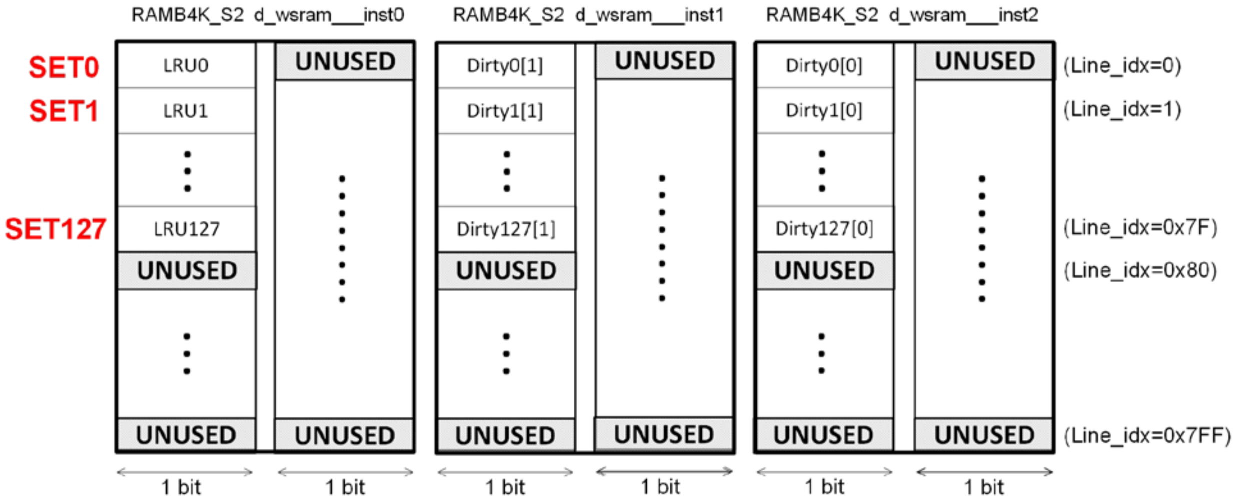


Figure 3. WS Array organization for the default D$.

The following code fragment shows the instantiation of the first RAMB4K_S2 bank (*d_wsram___inst0*) at module **d_wsram_2k2way_xilinx**. All interface signals are common to all banks (for example, *line_idx*) except for *wr_mask[i]*, *wr_data[j:k]* and *ws_rd_data[j:k]*. Observe that four 0s are added to the left-most bits of the WS Array index ({4'b0,*line_idx*}), thus we are always accessing the upper 1/16 of it. Moreover, note that the first bit of the write information is always set to 0 (unused):

```
// Need to create a 128 X 3 bit writeable array
// All xilinx block rams are 4kbit, so this is a huge waste of ram space
// 1 lru bit
RAMB4K_S2 d_wsram___inst0 (
        .WE     (wr_str && wr_mask[0]),
        .EN     (en),
        .RST    (1'b0),
        .CLK    (clk),
        .ADDR   ({4'b0,line_idx}),
        .DI     ({1'b0,wr_data[0]}),
        .DO     ({ws_rd_data[3],ws_rd_data[0]})
);
```

Finally, the bank implementation is shown in the next code fragment (extracted from module RAMB4K_S2). Each RAMB4K_S2 bank implements 2048 2-bit-wide memory entries.

```
module RAMB4K_S2(WE, EN, RST, CLK, ADDR, DI, DO);
    input              WE;
    input              EN;
    input              RST;
    input              CLK;
    input        [10:0] ADDR;
    input         [1:0] DI;
    output        [1:0] DO;
    reg           [1:0] mem[0:2047];
    reg           [1:0] DO;
    always @(posedge CLK) begin
      DO <= #1 mem[ADDR];
      if (EN)
        begin
               if (WE)
                mem[ADDR] <=#1  DI;
        end
    end
endmodule
```

3. Exercises

Exercise 1: Implement new D$ configurations

In this exercise you will implement and test new D$ configurations by modifying the size and associativity of the baseline cache (2-way D$ with 2KB per way). The supported sizes for the Data/Instruction Caches (D$/I$) in microAptiv can vary in the range from 1KB to 16KB per way, and the supported associativity can vary in the range from 1 to 4 ways. This means that the minimum cache size that can be used is 1KB, and the maximum cache size is 64KB (4-ways and 16KB per way).

To implement a processor with a new D$ configuration, you have to change the configuration file, create new wrappers and, in some cases, new memory banks. In folder *Lab21_CacheStructure\NewCacheWrappers* we provide the wrappers and memory banks for several D$ configurations. As for the configuration file, the following lines, extracted from the **m14k_config.vh** file, define a 2-way D$ with 2KB/Way.

```
`define M14K_DCACHE_ASSOC 2
`define M14K_DCACHE_WAYSIZE 2
`define M14K_MAX_DC_ASSOC 2

`define M14K_DC_TAGRAM tagram_2k2way_xilinx
`define M14K_DC_WSRAM d_wsram_2k2way_xilinx
`define M14K_DC_DATARAM dataram_2k2way_xilinx
```

These lines must be modified in order to support a new D$ configuration. Specifically, the new cache size, associativity and cache wrappers must be defined. For example, in order to implement a Direct-Mapped 2KB D$ you have to change the previous lines as follows.

```
`define M14K_DCACHE_ASSOC 1
`define M14K_DCACHE_WAYSIZE 2
`define M14K_MAX_DC_ASSOC 1

`define M14K_DC_TAGRAM tagram_2k1way_xilinx
`define M14K_DC_WSRAM d_wsram_2k1way_xilinx
`define M14K_DC_DATARAM dataram_2k1way_xilinx
```

In this exercise, you must implement processors with the following D$ configurations: a **Direct-Mapped 2KB D$**, a **2-way D$ with 1KB per Way** and a **Direct-Mapped 4KB D$**.

1. Copy the original soft-core folder (**rtl-up**) into three new folders (**rtl_up_2k1way**, **rtl_up_1k2way** and **rtl_up_4k1way**).
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. You will use this functionality in Exercise 2.
3. Copy the necessary wrappers and banks in each new folder. For example, in the case of a Direct-Mapped 4KB D$:
    a. Data Array:
        i. Copy **dataram_4k1way_xilinx.v** into folder **core\rtl_up_4k1way**
        ii. Copy the new bank for the Data Array: **RAMB8K_S8** into folder **core\rtl_up_4k1way**
    b. Tag Array:
        i. Copy **tagram_4k1way_xilinx** into folder **core\rtl_up_4k1way**
    c. WS Array:
        i. Copy **d_wsram_4k1way_xilinx** into folder **core\rtl_up_4k1way**
4. Modify file **core\m14k_config.vh** as explained above.
5. Create three new Vivado projects (**Project_2k1way**, **Project_1k2way** and **Project_4k1way**) following the instructions provided in Step 1 - Lab 1, using the files from the new folders (**rtl_up_2k1way**, **rtl_up_1k2way** and **rtl_up_4k1way** respectively). You probably will need to set *mfp_nexys4_ddr* as the top module.
6. Compile the 3 new Vivado projects, following the instructions provided in Step 3 of Section 6.4.1 of the Getting Started Guide. As such, click on the **Generate Bitstream** button at the top of the window. Now wait for synthesis, placement, routing, and bitstream generation to complete. This typically takes around 10-20 minutes or more, depending on your computer speed.

Exercise 2: Simple tests with the new D$ configurations

In this exercise you must test the different D$ configurations by executing simple tests on the board and measuring the results with the performance counters. Once the four *bitfiles* have been generated (i.e. **2k2way**, **2k1way**, **1k2way** and **4k1way**), you can program the board as explained in Step 4 of Section 6.4.1 of the Getting Started Guide. As such, click on **Open Hardware Manager** in the **Flow Navigator** window on the left. Make sure that the Nexys4 DDR FPGA board is turned on and connected to your computer, and click on **Open Target → Auto Connect**. Finally, click on **Program Device → xc7a100t_0**, select the *bitfile* if it is not selected yet, and click on **Program**. You can reprogram the board with a new D$ configuration as many times as necessary, by following these steps and selecting the *bitfile* corresponding to the desired D$ configuration.

1st test – Determine the D$ line size

Follow the next steps and discuss the results:

1. The source files, which are based on the skeleton provided in Lab 13, are provided in folder *Lab21_CacheStructure\SimpleTests\CacheLineSize*. Note however that in this case we are measuring *number of D$ accesses* and *D$ misses* (instead of *number of cycles* and *number of instructions completed* as we did in Lab 13). Note also that, as we did in Lab 20, the D$ is emptied by using *cache* instructions before performing our tests. Finally, note that, in this example, we are displaying not only the performance counter events, but also some parameters related with the D$ configuration; that way, you will be able to confirm the D$ configuration that you are using. Specifically:
    a. When switches == 2 → The D$ associativity is displayed through the 7-segment displays
    b. When switches == 3 → The D$ number of lines per way is displayed through the 7-segment displays
    c. When switches == 4 → The D$ line size (in Bytes) is displayed through the 7-segment displays
    d. When switches == 5 → The D$ write policy is displayed through the 7-segment displays, using the following encoding (as explained in Lab 23):
        i. 0 → *Write-through* and *no write allocate*
        ii. 1 → *Write-through* and *write allocate*
        iii. 3 → *Write-back* and *write allocate*
2. Compile the source files: open a shell (i.e., *cmd.exe* from the *Start* menu), go into the folder containing the source files, and type *make* in the shell. You can analyze the compiled program in file **program.dis**.
3. Using Vivado, program the FPGA board with MIPSfpga (note that in this first test you can use any D$ configuration, as they all have the same D$ line size).
4. Download the program into the board using the script **loadMIPSfpga.bat** as explained in Section 7.5 of the Getting Started Guide. The program executes and you will observe the results of the events measured by the performance counters on the 7-segment displays (use different switch combinations for selecting the different events).
5. In file *Lab21_CacheStructure\SimpleTests\CacheLineSize\main.c*, try different numbers of `lw` instructions (for example, from 4 loads to 9 loads), recompile and rerun the test for each number of loads. Discuss in detail the results obtained.

2nd test – Compare two D$ configurations with different associativity and same size

Follow the next steps:

1. Run the program provided in folder *Lab21_CacheStructure\SimpleTests\CacheAssociativity* on a processor with a **Direct-Mapped 2KB D$** and on a processor with a **2-way D$ with 1KB per Way**, and measure the number of *D$ accesses* and *D$ misses* for each configuration.
2. Discuss the results and determine what type of misses (compulsory, capacity or conflict) take place.

3rd test – Compare two D$ configurations with different size and same associativity

Follow the next steps:

1. Run the program provided in folder *Lab21_CacheStructure\SimpleTests\CacheSize* on a **Direct-Mapped 2KB D$** and on a **Direct-Mapped 4KB D$**, and measure the amount of *D$ accesses* and *D$ misses*.
2. Discuss the results and determine what type of misses (compulsory, capacity or conflict) take place.

4. 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.