THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2 Lab 19

Instruction Cache

1. Introduction

In this and the next lab, we focus on the memory system of the RVfpgaEL2 System. Remember that the RVfpgaEL2 System includes a Main Memory, a Cache for instructions (I$) and two Scratchpad memories (also called closely-coupled memories), one for data (DCCM) and one for instructions (ICCM).

NOTE: Before starting to work on this lab, we recommend reading Sections 8.1-8.3 of the book by S. Harris and D. Harris, "*Digital Design and Computer Architecture: RISC-V Edition*", Morgan Kaufmann [DDCARV].

In this lab, we focus on the operation of the Cache. Unfortunately, as mentioned before, the RVfpgaEL2 System does not include a data cache (D$). Thus, we cannot study a cache using the typical approaches where program data memory accesses are analysed. However, the RVfpgaEL2 System does include an I$, which we use in this lab to demonstrate the main concepts of Cache Memory. Most of the concepts explained in Section 8.3 of [DDCARV] are also applicable to an I$ and thus they are useful for our purposes.

We first describe how data are read from and written to Main Memory (Section 2), and then we delve into the operation and management of the I$ available in the RVfpgaEL2 System (Section 3).

2. Main Memory Data Accesses

Even though we cannot use a D$ in this lab to explain the cache, we use data accesses to describe the RVfpgaEL2 System's overall memory system. In Labs 13 and 14, we showed how loads and stores use both Main Memory and the DCCM. As explained in those labs, whenever the core needs to access data, the address is computed in the D Stage and then that data is read or written from/to Main Memory using the AXI Bus. The pipeline must be stalled for a few cycles when accessing Main Memory, but it does not stall when accessing the DCCM.

The next example illustrates a program that includes a load instruction followed by a store instruction, focusing on the read/write of Main Memory. Folder *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab19/LW-SW_Instruction_MainMemory* provides the Catapult project so that you can analyse, simulate, and modify the program as desired. The program, shown in Figure 1, traverses a 1,000-element array (uninitialized and used only for illustrative purposes), reading each element (`lw` instruction, highlighted in red), adding a constant to it, and storing the element (`sw` instruction, highlighted in red).

```
.globl Test_Assembly

.data
D: .space 4000

.text
Test_Assembly:

li t2, 0x000
csrrs t1, 0x7F9, t2

la t4, D
```

```
li t0, 4000
la t6, D
add t6, t6, t0

REPEAT:
   lw t3, (t4)
   add t3, t3, t4
   sw t3, (t4)
   add t4, t4, 4
   bne  t4, t6, REPEAT    # Repeat the loop

   INSERT_NOPS_4

   ret

.end
```

Figure 1. Example program

Open the project, build it, and open the disassembly file. Look for the `lw` instruction (0x000eae03) and the `sw` instruction (0x01cea023):

```
0x00000470:      000eae03             lw    t3,0(t4)
…
0x00000478:      01cea023             sw    t3,0(t4)
```

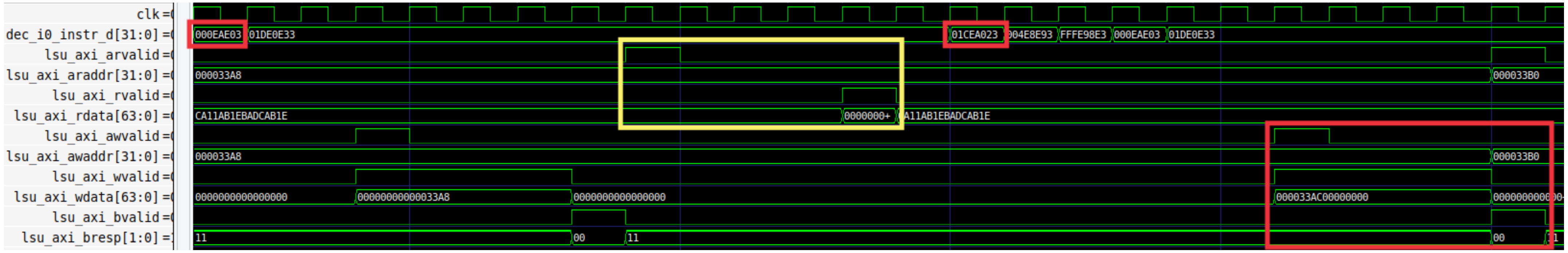


Figure 2. Simulation of a random iteration of the program from Figure 1

Figure 2 shows the simulation of a random iteration of the loop from Figure 1.

TASK: Replicate the simulation from Figure 2 on your own computer.

We next describe how memory reads and writes occur using Main Memory via the AXI bus.

- The processor reads data from Main Memory (yellow square in Figure 2) into `t3`. The reading starts in cycle *i* (the first cycle in the yellow square), when the write from the previous iteration has completed on the bus (i.e., when `lsu_axi_wvalid` goes from 1 to 0):
    - **Cycle *i*:** the effective address is sent to Main Memory through the AXI bus:
        - `lsu_axi_arvalid` = 1
        - `lsu_axi_araddr` = 0x000033A8
    - **Cycle *i*+4:** the read value is received through the AXI bus from Main Memory:
        - `lsu_axi_rvalid` = 1
        - `lsu_axi_rdata` (memory is not initialized in this program)
- The processor computes the addition (`add t3, t3, t5`) and writes the result to the Register File.
- The processor writes the value of `t3` to Main Memory (red square):
    - **Cycle *i*+12:** the effective address and the data are sent to Main Memory through the AXI bus:
        - `lsu_axi_awvalid` = 1
        - `lsu_axi_awaddr` = 0x000033A8
        - `lsu_axi_wvalid` = 1
        - `lsu_axi_wdata` = 0x000033AC00000000
    - **Cycle *i*+16:** Main Memory notifies through the AXI bus that the write has been correctly carried out:
        - `lsu_axi_bvalid` = 1
        - `lsu_axi_bresp` = 00 (defined as: everything has worked alright)

TASK: Using the hardware (HW) Counters, measure the number of cycles, instructions, loads and stores in the program from Figure 1. How much time in total (both for reading and writing) does it take to access Main Memory? You can compare the execution when using Main Memory as in Figure 3 and when using the DCCM. Another Catapult project is provided at *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab19/LW-SW_Instruction_DCCM/*, which contains the same program prepared for reading from / writing to the DCCM. Remember that you can use either RVfpgaEL2-NexysA7 on the physical board or RVfpgaEL2-ViDBo on the virtual board.

TASK: Use the example from *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab19/LW_Instruction_MainMemory* to estimate the Main Memory read latency using the HW Counters. As in the previous task, you can use the example from *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab19/LW_Instruction_DCCM* to compare with a program with no stalls due to the memory accesses. Remember that you can use either RVfpgaEL2-NexysA7 on the physical board or RVfpgaEL2-ViDBo on the virtual board.

3. Instruction Fetch from the Instruction Cache (I$)

In this section we analyse the operation of the Instruction Cache (I$) available in the RVfpgaEL2 System. We first describe how the I$ can be configured (Section 3.A) and then investigate how cache misses and hits are processed (Sections 3.B and 3.C), and finally we analyse the I$ Replacement Policy used in VeeR EL2 (Section 3.D).

A. I$ Configuration

The RVfpgaEL2 System’s I$ is highly configurable based on a set of parameters defined in file *[RVfpgaEL2NexysA7NoDDRPath]/src/VeeRwolf/VeeR_EL2CoreComplex/include/el2_param.vh*. The default RVfpgaEL2 System has the following I$ parameters:

```
ICACHE_2BANKS           : 5'h00         ,
ICACHE_BANK_BITS        : 7'h01         ,
ICACHE_BANK_HI          : 7'h03         ,
ICACHE_BANK_LO          : 6'h03         ,
ICACHE_BANK_WIDTH       : 8'h08         ,
ICACHE_BANKS_WAY        : 7'h02         ,
ICACHE_BEAT_ADDR_HI     : 8'h05         ,
ICACHE_BEAT_BITS        : 8'h03         ,
ICACHE_BYPASS_ENABLE    : 5'h01         ,
ICACHE_DATA_DEPTH       : 18'h00100      ,
ICACHE_DATA_INDEX_LO    : 7'h04         ,
ICACHE_DATA_WIDTH       : 11'h040        ,
ICACHE_ECC              : 5'h00         ,
ICACHE_ENABLE           : 5'h01         ,
ICACHE_FDATA_WIDTH      : 11'h044        ,
ICACHE_INDEX_HI         : 9'h00B        ,
ICACHE_LN_SZ            : 11'h040        ,
ICACHE_NUM_BEATS        : 8'h08         ,
ICACHE_NUM_BYPASS       : 8'h02         ,
ICACHE_NUM_BYPASS_WIDTH : 8'h02          ,
ICACHE_NUM_WAYS         : 7'h02         ,
ICACHE_ONLY             : 5'h00         ,
ICACHE_SCND_LAST        : 8'h06         ,
ICACHE_SIZE             : 13'h0008       ,
ICACHE_STATUS_BITS      : 7'h01         ,
ICACHE_TAG_BYPASS_ENABLE : 5'h01         ,
ICACHE_TAG_DEPTH        : 17'h00040      ,
ICACHE_TAG_INDEX_LO     : 7'h06         ,
ICACHE_TAG_LO           : 9'h00C        ,
ICACHE_TAG_NUM_BYPASS   : 8'h02         ,
ICACHE_TAG_NUM_BYPASS_WIDTH : 8'h02          ,
      ICACHE_WAYPACK          : 5'h01          ,
```

The I$ in the default RVfpgaEL2 System has 8KiB (`ICACHE_SIZE:8`), 2 ways (`ICACHE_NUM_WAYS:2`) and a block size of 64B. You will analyse the remaining parameters in the next task. A simplified diagram of the I$ used in the default RVfpgaEL2 System is shown in Figure 3.

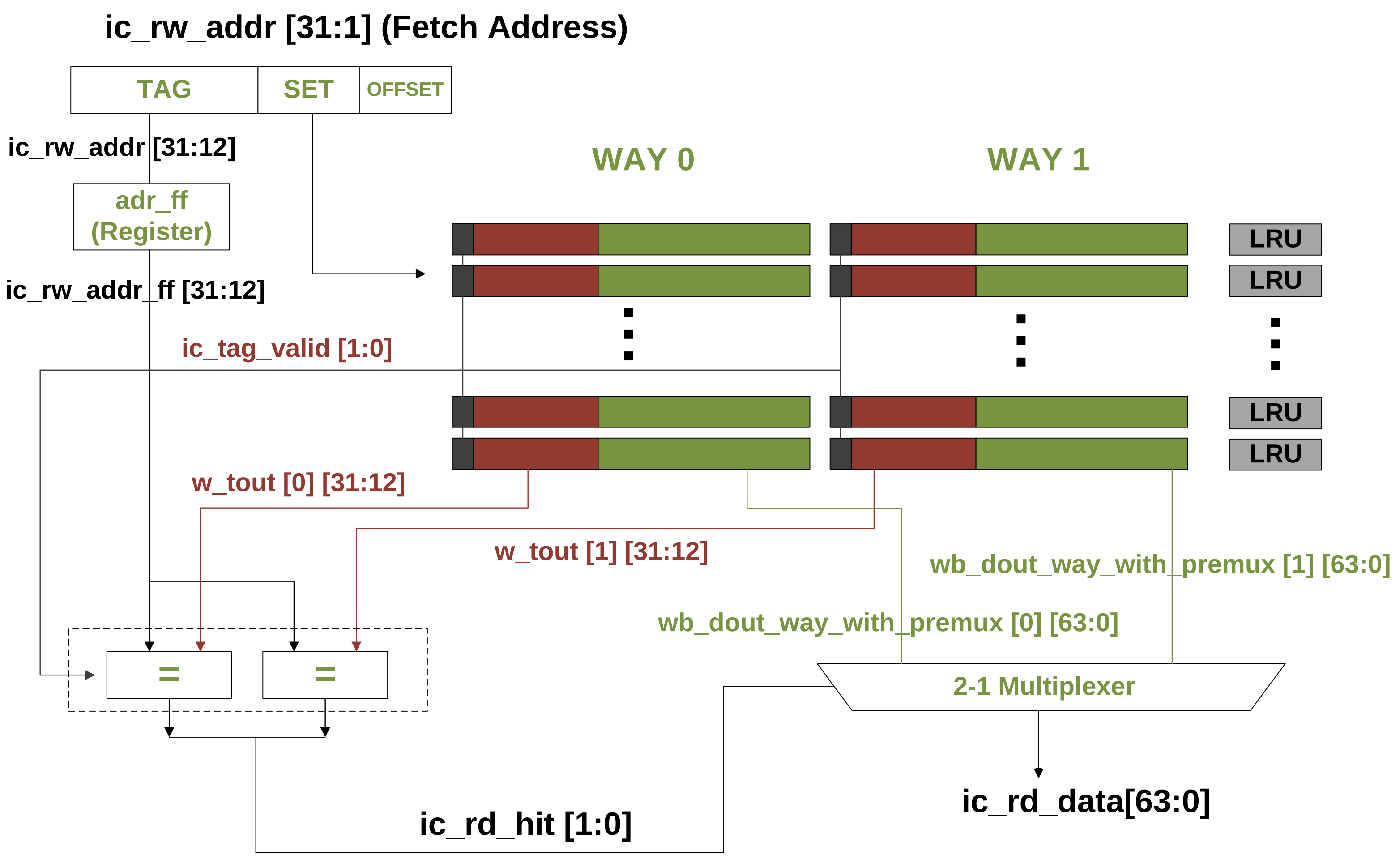
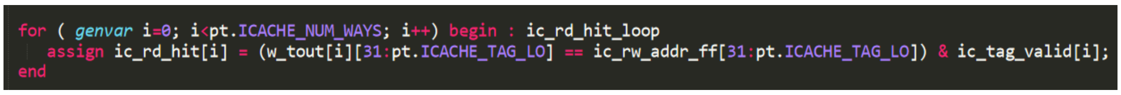


Figure 3. I$ internal design. The input signal to the I$ (`ic_rw_addr`) and the output signal from the I$ (`ic_rd_data`) is provided from/to the Cache Controller (module `el2_ifu_mem_ctl`).

The RVfpgaEL2 System’s I$ is implemented in module **el2_ifu_ic_mem**, included in file *[RVfpgaEL2NexysA7NoDDRPath]/src/VeeRwolf/VeeR_EL2CoreComplex/ifu/el2_ifu_ic_mem.sv*. This module instantiates two other modules:

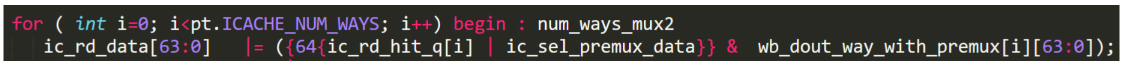
- **EL2_IC_TAG**: This module includes the Tag Array (the red boxes shown in Figure 3) and the logic to compute the hit signal, `ic_rd_hit`. The module receives, among other signals, the Address, `ic_rw_addr`. It outputs, among other signals, signal `ic_rd_hit`, which is used by the Data Array to select the cache way where the hit takes place.

  The tags read when an access to the I$ is performed are provided in signal `w_tout[1:0][31:12]`, as shown in Figure 3. The tags are compared with the most significant bits of the Fetch Address and the valid signal is checked in order to determine if a hit takes place. This is the Verilog code for the comparator:

  ```
  for ( genvar i=0; i<pt.ICACHE_NUM_WAYS; i++) begin : ic_rd_hit_loop
     assign ic_rd_hit[i] = (w_tout[i][31:pt.ICACHE_TAG_LO] == ic_rw_addr_ff[31:pt.ICACHE_TAG_LO]) & ic_tag_valid[i];
  end
  ```

- **EL2_IC_DATA**: This module is the Data Array, which includes the green boxes shown in Figure 3, as well as the 2:1 multiplexer that selects the data from the way where a hit takes place. The module receives, among other signals, the Fetch Address (`ic_rw_addr`) and the hit signal from the IC_TAG module (`ic_rd_hit`). Based on these signals, the module reads a whole set and, if there is a hit, selects the 64-bit instruction bundle (`ic_rd_data[63:0]`) that must be sent to the VeeR EL2 processor. This is the Verilog code for the multiplexer that selects the 64-bit instruction bundle:

  ```
  for ( int i=0; i<pt.ICACHE_NUM_WAYS; i++) begin : num_ways_mux2
     ic_rd_data[63:0]    |= ({64{ic_rd_hit_q[i] | ic_sel_premux_data}} &  wb_dout_way_with_premux[i][63:0]);
  ```

<u>TASK</u>: Analyse module **ifu_ic_mem** and the parameters of file *el2_param.vh* to understand how the elements in Figure 3 are implemented.

B. I$ Miss Management

In this section, we show how instruction misses are managed in the processor. The example in Figure 4 illustrates a program that includes 16 sequential, uncompressed `add` instructions (which occupy 4*16 = 64 bytes), shown in red in the figure, within a loop with 0x10000 iterations. Several `nop` instructions are placed before the 16 `add` instructions to force the 16 add instructions to be aligned on a block boundary. Folder *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab19/InstructionMemory_Example* provides the Catapult project so that you can analyse, simulate, and modify the program as desired.

```
Test_Assembly:

   INSERT_NOPS_3
   INSERT_NOPS_8
```

```
   INSERT_NOPS_8

   li t6, 0x10000

REPEAT:
   add t6, t6, -1

   add t0, t0, t0
   add t1, t1, t1
   add t2, t2, t2
   add t3, t3, t3
   add t4, t4, t4
   add t5, t5, t5
   add t6, t6, t6
   add a7, a7, a7
   add t0, t0, t0
   add t2, t2, t2
   add t1, t1, t1
   add t3, t3, t3
   add t4, t4, t4
   add t6, t6, t6
   add t5, t5, t5
   add a7, a7, a7

   INSERT_NOPS_8
   INSERT_NOPS_8

   INSERT_NOPS_8
   INSERT_NOPS_8
   INSERT_NOPS_8
   INSERT_NOPS_8

   bne t6, zero, REPEAT    # Repeat the loop

ret
```

Figure 4. Example program

Open the project, build it, and open the disassembly file. Notice that the first `add` instruction (0x005282b3) is at address 0x00000440 (which is aligned on a block boundary) and the sixteenth `add` (0x011888b3) is at address 0x0000047c.

```
0x00000440:     005282b3          add   t0,t0,t0
    …              …                      …
0x0000047c:     011888b3          add   a7,a7,a7
```

Figure 5 shows the simulation of the region around the first execution of the 16 `add` instructions, where the instructions are not in the I$ and a miss takes place.

TASK: Replicate the simulation from Figure 5 on your own computer. You can also analyse some things in more detail, such as the write to the I$ or the bypass of the initial instructions.

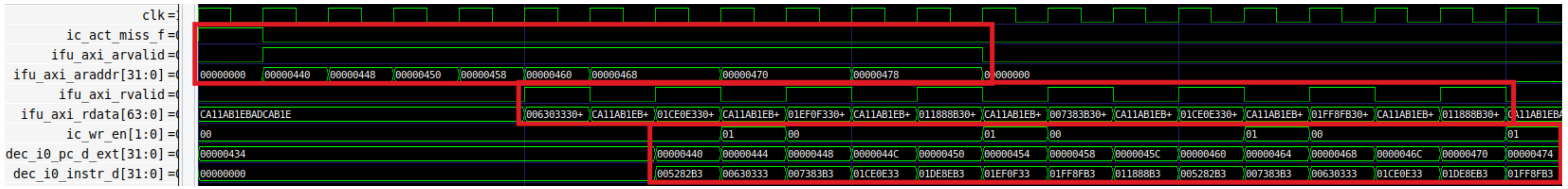

clk
ic_act_miss_f
ifu_axi_arvalid
ifu_axi_araddr[31:0]
00000000 00000440 00000448 00000450 00000458 00000460 00000468 00000470 00000478 00000000
ifu_axi_rvalid
ifu_axi_rdata[63:0]
CA11AB1EBADCAB1E 006303330+ CA11AB1EB+ 01CE0E330+ CA11AB1EB+ 01EF0F330+ CA11AB1EB+ 011888B30+ CA11AB1EB+ 007383B30+ CA11AB1EB+ 01CE0E330+ CA11AB1EB+ 01FF8FB30+ CA11AB1EB+ 011888B30+
ic_wr_en[1:0]
00 01 00 01 00 01 00 01
dec_i0_pc_d_ext[31:0]
00000434 00000440 00000444 00000448 0000044C 00000450 00000454 00000458 0000045C 00000460 00000464 00000468 0000046C 00000470 00000474
dec_i0_instr_d[31:0]
00000000 005282B3 00630333 007383B3 01CE0E33 01DE8EB3 01EF0F33 01FF8FB3 011888B3 005282B3 007383B3 00630333 01CE0E33 01DE8EB3 01FF8FB3

Figure 5. Simulation of the program from Figure 4 showing an I$ miss

This example illustrates how an I$ miss is handled in VeeR EL2. It shows the fetch of the 16 `add` instructions the first time they are executed. When these instructions are not in the I$ yet and they must be copied from Main Memory into the I$.

- You can see that an I$ miss is signalled `ic_act_miss_f2` = 1 in the first cycle shown in the figure. This triggers the request of the block through the AXI bus (`ifu_axi_arvalid` = 1).

- Then, the target block is requested in consecutive 64-bit chunks through the AXI bus.
  - Signal `ifu_axi_arvalid` goes high for several cycles. This signal indicates that the channel is signalling valid read address and control information.
  - During these cycles where `ifu_axi_arvalid` = 1 the initial addresses of the eight 64-bit chunks that make up the requested block are provided sequentially through the AXI bus using signal `ifu_axi_araddr`:
    - `ifu_axi_araddr` = 0x00000440
    - `ifu_axi_araddr` = 0x00000448
    - `ifu_axi_araddr` = 0x00000450
    - `ifu_axi_araddr` = 0x00000458
    - `ifu_axi_araddr` = 0x00000460
    - `ifu_axi_araddr` = 0x00000468
    - `ifu_axi_araddr` = 0x00000470
    - `ifu_axi_araddr` = 0x00000478

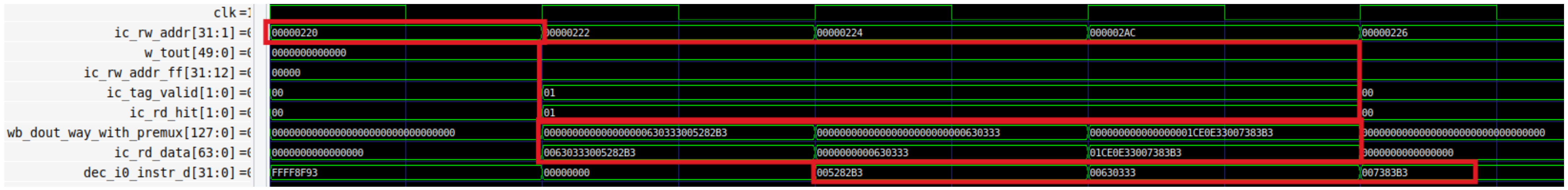
- The eight 64-bit chunks arrive sequentially to the processor through the AXI bus in signal `ifu_axi_rdata`.
  - Signal `ifu_axi_rvalid`, which indicates that the channel is receiving the required read data, goes high every two cycles.
  - Each of the eight 64-bit chunks (each containing two uncompressed `add` instructions) is provided in signal `ifu_axi_rdata` (this cannot be seen in Figure 5 – you can replicate the simulation on your computer to verify it):
    - `ifu_axi_rdata` = 0x00630333005282b3
    - `ifu_axi_rdata` = 0x01ce0e33007383b3
    - `ifu_axi_rdata` = 0x01ef0f3301de8eb3
    - `ifu_axi_rdata` = 0x011888b301ff8fb3
    - `ifu_axi_rdata` = 0x007383b3005282b3
    - `ifu_axi_rdata` = 0x01ce0e3300630333
    - `ifu_axi_rdata` = 0x01ff8fb301de8eb3
    - `ifu_axi_rdata` = 0x011888b301ef0f33

- The chunks are written sequentially to the I$ by enabling the `ic_wr_en` signal. Moreover, the instructions are bypassed from the I$ controller to the pipeline so that it can restart execution as soon as possible after the I$ miss. The two signals on the bottom (`dec_i0_pc_d_ext` and `dec_i0_instr_d`) show the instructions at the D Stage executing sequentially.

C. I$ Hit Management

In this section we work with the same example from Section 3.B (Figure 4), but we now focus on analysing I$ hits. Figure 6 shows the second iteration of the loop when executing the program in Figure 4 (any iteration would be valid except the first one, which, as we analysed in Figure 5, experiences misses in the I$).

<u>TASK</u>: Replicate the simulation from Figure 6 on your own computer. Use file *test1_Hit.tcl* (provided at *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab19/InstructionMemory_Example*).



```
clk=1
ic_rw_addr[31:1] =0           00000220          00000222                          00000224                          000002AC                          00000226
w_tout[49:0] =0               0000000000000
ic_rw_addr_ff[31:12] =0       00000
ic_tag_valid[1:0] =0          00                01                                                                                                      00
ic_rd_hit[1:0] =0             00                01                                                                                                      00
wb_dout_way_with_premux[127:0] =0  00000000000000000000000000000000  000000000000000000630333005282B3  00000000000000000000000000630333  000000000000000001CE0E33007383B3  00000000000000000000000000000000
ic_rd_data[63:0] =0           0000000000000000  00630333005282B3                  0000000000630333                  01CE0E33007383B3                  0000000000000000
dec_i0_instr_d[31:0] =0       FFFF8F93          00000000                          005282B3                          00630333                          007383B3
```

Figure 6. Simulation waveform for the program in Figure 4 illustrating an I$ hit

Analyse the simulation waveform in Figure 6. The iI$ hit occurs as follows:

- **Cycle 1:** The address of the first add instruction (`add t0,t0,t0`) in the program from Figure 4 is given in signal `ic_rw_addr` (`ic_rw_addr[31:1]` = 0x00000100. Note that the least significant bit is not provided as instructions must be 2-byte (16-bit) aligned).

  The Tag Array and the Data Array are accessed using a subset of the Fetch Address, as shown in Figure 3. The result of the access will be available in the next cycle.

- **Cycle 2:** The two tags (`w_tout`), one per cache way, are compared to the TAG field of the Fetch Address (`ic_rw_addr_ff`). In this case, all tags are the same as the TAG field, however only one way (Way 0) is valid (`ic_tag_valid` = 01), thus a hit is signalled in Way 0: `ic_rd_hit` = 01.

  Also, two 64-bit bundles are in signal `wb_dout_way_with_premux`. The 2:1 multiplexer from Figure 3 selects the data provided by Way 0. Thus:
  `ic_rd_data` = 0x00630333005282b3

  Cycle 3: The 64 bits selected by the I$ in cycle 2 are propagated to the D Stage, where the 32-bit instruction is selected into `dec_i0_instr_d` = 0x005282b3. Note that these 32 bits correspond to the first `add` instruction.

  Also, in this cycle another I$ hit happens and a new 64-bit bundle is selected into `ic_rd_data` = 0x0000000000630333

- **Cycle 4:** The 64 bits selected by the I$ in cycle 3 are propagated to the D Stage, where the 32-bit instruction is selected into `dec_i0_instr_d` = 0x00630333. Note that these 32 bits correspond to the second `add` instruction.

  Also, in this cycle another I$ hit happens and a new 64-bit bundle is selected into `ic_rd_data` = 0x01ce0e33007383b3

- **Cycle 5:** The 64 bits selected by the I$ in cycle 4 are propagated to the D Stage, where the 32-bit instruction is selected into `dec_i0_instr_d` = 0x007383b3. Note that these 32 bits correspond to the third `add` instruction.

D. I$ Replacement Policy

This section describes the RVfpgaEL2 System's cache replacement policy. As explained by Harris & Harris in Section 8.3.3 of [DDCARV], in set associative caches (also called simply associative caches), the cache must choose which block to evict when a cache set is full. The principle of temporal locality suggests that the best choice is to evict the least recently used block because it is least likely to be used again soon. Hence, most associative caches have a least recently used (LRU) replacement policy. However, tracking the least recently used way becomes complicated, thus approximate LRU policies (usually called Pseudo LRU) are often used and good enough in practice. Specifically, VeeR EL2 uses an approximate policy called **Binary Tree Pseudo LRU**.

NOTE: If you haven't done so already, read Section 8.3.3 of [DDCARV]. Also, we recommend reading Section 4 of the Master Thesis by Gille Damien, "Study of Different Cache Line Replacement Algorithms in Embedded Systems" (8 March 2007), which you can find online at: https://people.kth.se/~ingo/MasterThesis/ThesisDamienGille2007.pdf. We refer to that document as [GiDa].

i. Implementation of the Binary Tree Pseudo LRU policy in VeeR EL2

As explained in [GiDa], a Binary Tree LRU policy, which is an approximation of an LRU policy, requires *N*-1 bits per set (which we call LRU State) for an *N*-way associative cache. Thus, in the case of our default RVfpgaEL2 System, where a 2-way Instruction Cache is used, only 1 bit is required per set to track the access history to the different ways. Note that in this case the Binary Tree LRU coincides with the LRU policy.

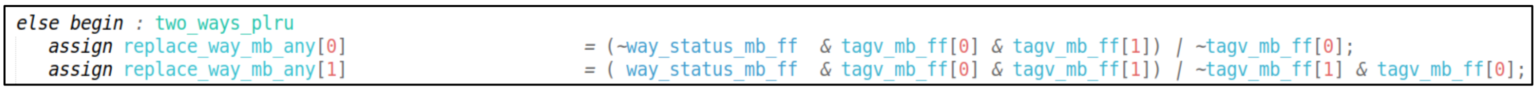
As explained in Section 3.B, when an I$ miss occurs, the block must be requested from Main Memory. When Main Memory supplies the cache block, it must be written into the I$. The SET field of the Fetch Address determines the I$ set where the new block must be written (see Figure 3). Two things can happen:

- The set is not full, meaning that one or more blocks are non-valid. In this case, the new block is written in the lowest way that contains a non-valid block.

- The set is full, meaning that all blocks are valid. In our processor, the Binary Tree LRU Replacement Policy determines which block must be evicted. This policy determines the way to replace based on the 1-bit LRU State of the Set, according to the following table:

LRU State	Way to replace
0	Way 0
1	Way 1

The following Verilog snippet (Figure 7), extracted from module **`el2_ifu_mem_ctl`**, implements the logic for the selection of the way that must be used for storing the new I$ block, according to the previous explanation.



```
else begin : two_ways_plru
   assign replace_way_mb_any[0]                        = (~way_status_mb_ff  & tagv_mb_ff[0] & tagv_mb_ff[1]) | ~tagv_mb_ff[0];
   assign replace_way_mb_any[1]                        = ( way_status_mb_ff  & tagv_mb_ff[0] & tagv_mb_ff[1]) | ~tagv_mb_ff[1] & tagv_mb_ff[0];
```

Figure 7. Verilog code for selecting which way must be replaced

The signals used in the Verilog snippet from Figure 7 are the following:

- **`replace_way_mb_any`** (2 bits): holds a one-hot value that is 1 for the way that must be replaced.

- **`way_status_mb_ff`** (1 bit): holds the LRU state of the selected set.

- **`tagv_mb_ff`** (2 bits): holds the valid bit of each way of the selected set. Ways that are valid have a valid bit of 1, whereas invalid ways have a valid bit of 0.

TASK: Analyse the Verilog code from Figure 7 and explain how it operates based on the above explanations.

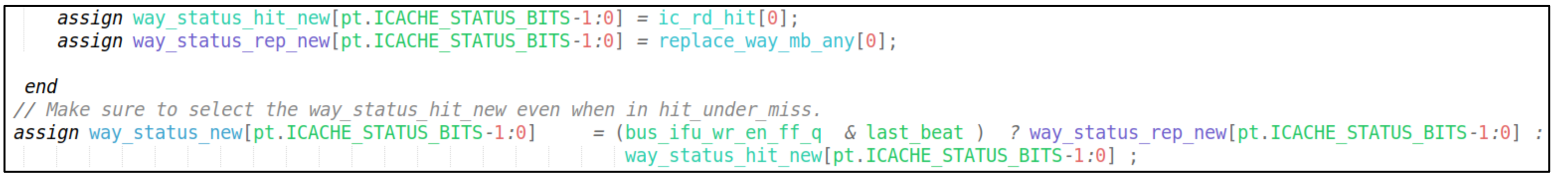
TASK: Analyse the Verilog code that performs the same functionality on a 4-way I$.

When a hit or a miss happens in the I$, the LRU state of the set must be updated according to the following table:

Written Way	**Next LRU state**
Way 0	1
Way 1	0

If you analyse this table, you will see that, as explained by [GiDa], upon a hit or miss, the bits on the path towards the hit/inserted line are inversed to indicate the opposite part of the tree as pseudo LRU. The idea behind it is to protect the last accessed data from eviction by inversing the nodes to point away from it. In the case of a 2-way cache, this simply means that the next way to select is the one that was not written last.

The following Verilog snippet (
Figure 8), extracted from module **ifu_mem_ctl**, implements the logic for this update of the LRU state.



```
    assign way_status_hit_new[pt.ICACHE_STATUS_BITS-1:0] = ic_rd_hit[0];
    assign way_status_rep_new[pt.ICACHE_STATUS_BITS-1:0] = replace_way_mb_any[0];

 end
// Make sure to select the way_status_hit_new even when in hit_under_miss.
assign way_status_new[pt.ICACHE_STATUS_BITS-1:0]     = (bus_ifu_wr_en_ff_q  & last_beat )  ? way_status_rep_new[pt.ICACHE_STATUS_BITS-1:0] :
                                                       way_status_hit_new[pt.ICACHE_STATUS_BITS-1:0] ;
```

Figure 8. Verilog snippet for updating the LRU state

The signals used in the Verilog snippet from
Figure 8 are the following:

- **`ic_rd_hit`** (1 bit): holds the way where a hit has taken place.

- **`replace_way_mb_any`** (1 bit): holds a one-hot value that is 1 for the way that must be replaced. This signal was also explained below Figure 7.

- **`way_status_new`** (1 bit): holds the new LRU state for the set just referenced on a hit or a miss.

TASK: Analyse the Verilog code from
Figure 8 and explain how it operates based on the above explanations.

TASK: Analyse the Verilog code that performs the same functionality on a 4-way I$.

ii. Example demonstrating the operation of the Binary Tree LRU policy

For analysing the replacement policy of VeeR EL2, we provide a new example in folder *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab19/InstructionMemory_LRU_Example*. In this example (Figure 9), three different I$ blocks are accessed inside an infinite loop and all five of these blocks map to the same I$ set. For that purpose, we create an infinite loop that contains three `j` (jump) instructions, where each pair of `j` instructions is separated by 1023 nops. Notice that the `j` instruction plus the nops occupy 4KiB (1024 * 4Bytes/Instruction), which is equal to the size of each Way in the I$.

```
Block1: j Block2 # This j instruction is stored at address 0x00000200
        INSERT_NOPS_1023

Block2: j Block3 # This j instruction is stored at address 0x00001200
        INSERT_NOPS_1023

Block3: j Block1 # This j instruction is stored at address 0x00002200
```

Figure 9. Example program showing `j` instructions that map to the same set

Open the project, build it, and open the disassembly file. Notice the following:

- The first `j` instruction (`j Block2`) is at address 0x00000200. According to the address division shown in Figure 3 for accessing the I$:
  I$ Address in binary = 00000000000000000000001000000000
  TAG = 0x0
  SET-OFFSET = 0x200

- The second `j` instruction (`j Block3`) is at address 0x00001200. According to the address division shown in Figure 3 for accessing the I$:
  I$ Address in binary = 00000000000000000001001000000000
  TAG = 0x1
  SET-OFFSET = 0x200
  OFFSET = 0x0

- The third `j` instruction (`j Block1`) is at address 0x00002200. According to the address division shown in Figure 3 for accessing the I$:
  I$ Address in binary = 00000000000000000010001000000000
  TAG = 0x2
  SET-OFFSET = 0x200

In this program (Figure 9), when the first iteration is executed, the set is initially empty. Figure 10 shows the theoretical changes to the set in the I$ while executing the first iteration. Later, we show several Verilator simulations that confirm these theoretical explanations.

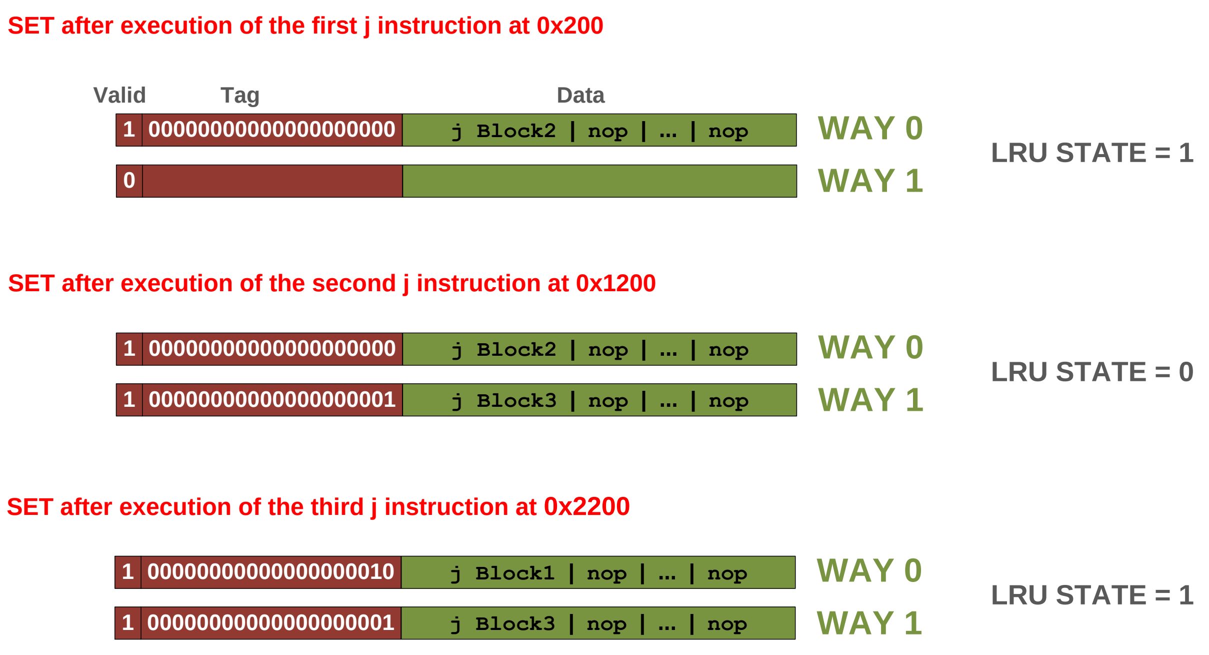
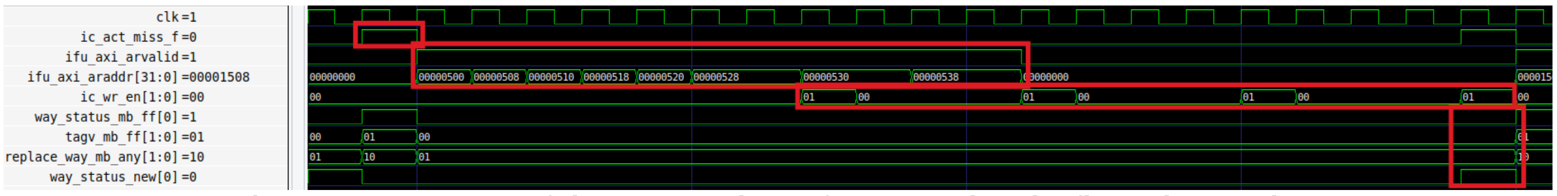

SET after execution of the first j instruction at 0x200
Valid
Tag
Data
1 00000000000000000000 j Block2 | nop | … | nop WAY 0
0 WAY 1
LRU STATE = 1
SET after execution of the second j instruction at 0x1200
1 00000000000000000000 j Block2 | nop | … | nop WAY 0
1 00000000000000000001 j Block3 | nop | … | nop WAY 1
LRU STATE = 0
SET after execution of the third j instruction at 0x2200
1 00000000000000000010 j Block1 | nop | … | nop WAY 0
1 00000000000000000001 j Block3 | nop | … | nop WAY 1
LRU STATE = 1

Figure 10. Set 8 of the I$ during execution of the first loop iteration in Figure 9

The following RVfpgaEL2-Trace simulations show the cache signals during the first iteration of the loop, and they confirm the analysis shown in Figure 10.

Figure 11 shows the simulation of the program after executing the first `j` instruction (`j Block2`), which is mapped to address 0x500. An I$ miss is detected (`ic_act_miss_f` = 1), and the block is requested from Main Memory in 64-bit chunks (see signals `ifu_axi_arvalid` and `ifu_axi_araddr`) and then written into the I$ (see signal `ic_wr_en`).

The set is initially empty: `tagv_mb_ff` = 00. Thus, according to the Binary Tree LRU policy, the new block must be written in Way 0: `replace_way_mb_any` = 01. The LRU state of the set is updated as follows: `way_status_new` = 1.



```
clk=1
ic_act_miss_f=0
ifu_axi_arvalid=1
ifu_axi_araddr[31:0]=00001508
ic_wr_en[1:0]=00
way_status_mb_ff[0]=1
tagv_mb_ff[1:0]=01
replace_way_mb_any[1:0]=10
way_status_new[0]=0
```

Figure 11. LRU state of the accessed set after executing the first `j` instruction

Figure 12 shows the simulation of the program after executing the second `j` instruction (`j Block3`), which is mapped to address 0x1500. An I$ miss is detected (`ic_act_miss_f` = 1), and the block is requested from Main Memory in 64-bit chunks (see signals `ifu_axi_arvalid` and `ifu_axi_araddr`) and then written into the I$ (see signal `ic_wr_en`).

The set is not empty, but Way 1 is not valid: `tagv_mb_ff` = 01. Thus, according to the Binary Tree LRU policy, the new block must be written in Way 1: `replace_way_mb_any` = 10. The LRU state of the set is updated as follows: `way_status_new` = 0.

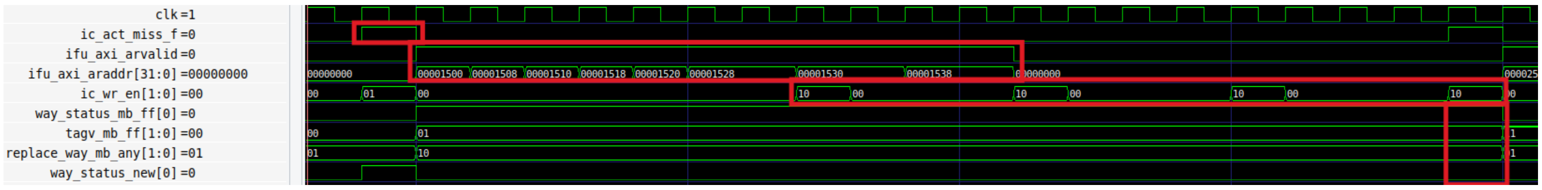


Figure 12. LRU state of the accessed set after executing the second `j` instruction

Figure 13 shows the simulation of the program after executing the third `j` instruction (`j Block1`), which is mapped to address 0x2500. An I$ miss is detected (`ic_act_miss_f` = 1), and the block is requested from Main Memory in 64-bit chunks (see signals `ifu_axi_arvalid` and `ifu_axi_araddr`) and then written into the I$ (see signal `ic_wr_en`).

The set is full: `tagv_mb_ff` = 11. Thus, according to the Binary Tree LRU policy, the new block must be written in Way 0: `replace_way_mb_any` = 01. The LRU state of the set is updated as follows: `way_status_new` = 1.

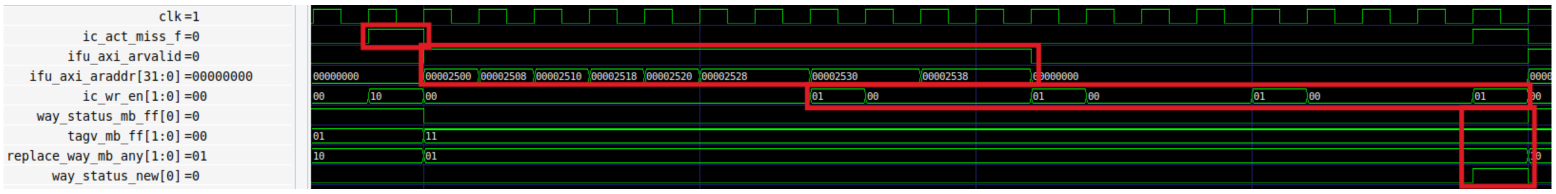


Figure 13. LRU state of the requested set after executing the third `j` instruction

TASK: Replicate the simulation from Figure 11-Figure 13 on your own computer.

4. Exercises

1) Transform the infinite loop from Figure 9 into a loop with 10,000 iterations, but keep the `j` instructions at the same addresses. Measure the number of cycles and I$ hits and misses. Then remove one of the `j` instructions and measure the same metrics. Compare and explain the results.

2) Extend Figure 5 (program provided at *[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab19/InstructionMemory_Example*) to analyse in detail how each 64-bit chunk is written in the I$.

3) Analyse in simulation and on the board other I$ configurations. For example, it can be very interesting to analyse a 4-way I$.

4) Analyse the logic that checks the correctness of the parity information.