THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 19

Instruction Cache

1. INTRODUCTION

In this and the next lab, we focus on the memory system of the RVfpga System. Remember from Figure 25 in the RVfpga Getting Started Guide (that we replicate in Figure 1 for the sake of convenience), that the RVfpga System has an External DDR 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).

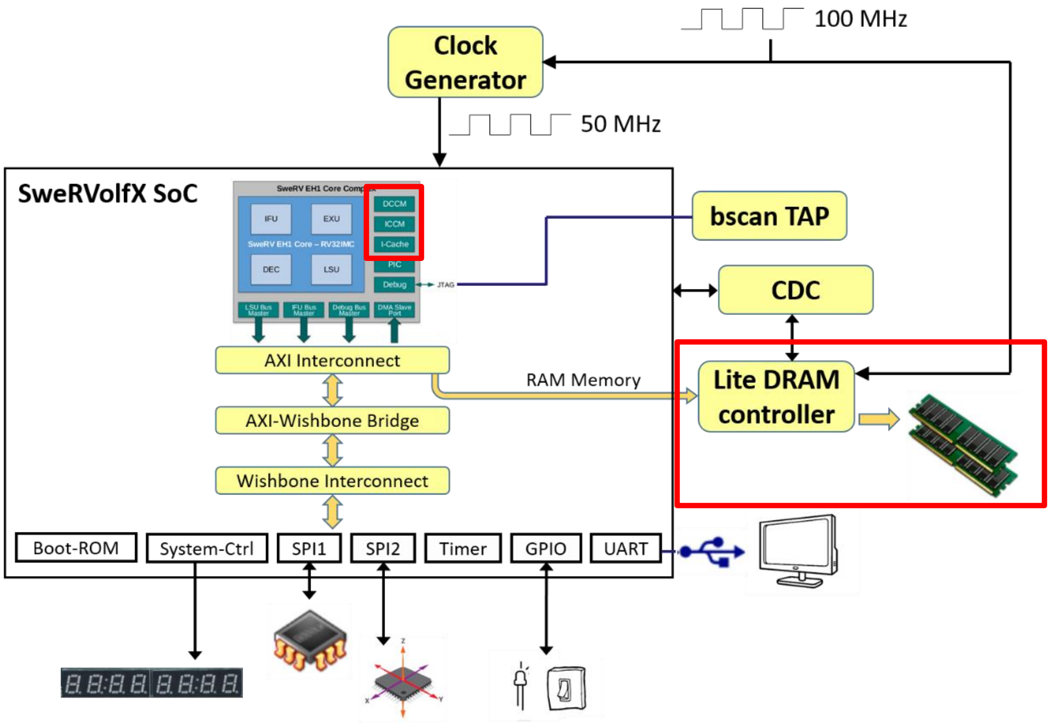


Figure 1. RVfpgaNexys: the RVfpga Memory System is highlighted by red boxes

NOTE: Before starting working on Lab 19, 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 shown in Figure 1, the RVfpga System does not include a data cache (D$). Thus, we cannot study a cache using the typical approaches where program data memory accesses are analysed. However, the RVfpga System does include an I$, which we use in this lab to demonstrate the main concepts of a 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 the DDR External Memory (Section 2), and then we delve into the operation and management of the I$ available in the RVfpga System (Section 3).

2. DDR EXTERNAL MEMORY DATA ACCESSES

Even though we cannot use a D$ in this lab to explain the cache, we use data accesses to describe the RVfpga System's overall memory system. In Labs 13 and 14, we showed how loads and stores use both the DDR External Memory and the DCCM. As explained in those labs, whenever the core needs to access data, the address is computed in DC1 and then that data is read or written from/to Main Memory during the remaining stages using the AXI Bus. The pipeline must be stalled for a few cycles when accessing the DDR 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 the DDR External Memory. Folder *[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory* provides the PlatformIO project so that you can analyse, simulate, and modify the program as desired. The program, shown in Figure 2, traverses a 10,000-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) in the same component.

```
.data
D: .space 40000

.text
Test_Assembly:

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

la t4, D
li t5, 50
li t0, 40000
la t6, D
add t6, t6, t0

REPEAT:
   lw t3, (t4)
   add t3, t3, t5
   sw t3, (t4)
   add t4, t4, 4
   bne  t4, t6, REPEAT     # Repeat the loop
```

Figure 2. Example program

Open the project in PlatformIO, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory/.pio/build/swervolf_nexys/firmware.dis*). Notice that the `lw` instruction (0x000eae03) and the `sw` instruction (0x01cea023) are at addresses 0x00000194 and 0x0000019c, respectively.

```
0x00000194:      000eae03          lw    t3,0(t4)
…
0x0000019c:      01cea023          sw    t3,0(t4)
```

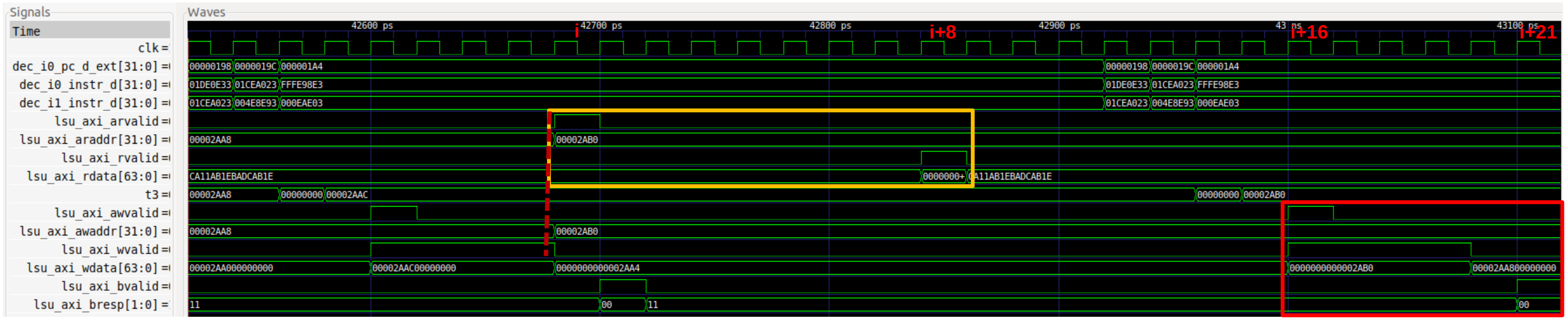


Figure 3. Simulation of a random iteration of the program from Figure 2

Figure 3 shows the simulation of a random iteration of the loop from Figure 2.

<u>TASK</u>: Replicate the simulation from Figure 3 on your own computer. To do so, follow the next steps (as described in detail in Section 7 of the GSG):
- If necessary, generate the simulation binary (*Vrvfpgasim*).
- In PlatformIO, open the project provided at: *[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory*.
- Establish the correct path to the RVfpga simulation binary (*Vrvfpgasim*) in file *platformio.ini*.
- Generate the simulation trace using Verilator (Generate Trace).
- Open the trace on GTKWave.
- Use file *test_Blocking_Extended.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory*) for opening the same signals as the ones shown in Figure 6. For that purpose, on GTKWave, click on *File → Read Tcl Script File* and select the *test_Blocking_Extended.tcl* file.
- Click on *Zoom In* ( ) several times and analyse the region starting at 42500 ps.

We next describe how memory reads and writes occur using the DDR External Memory via the AXI bus. Refer to Section 4.B.iii of the Getting Started Guide for more details about how the bus operates.

- The processor reads data from the DDR External Memory (yellow square) into `t3`. The reading starts in cycle i, 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 the External Memory through the AXI bus:
        - `lsu_axi_arvalid` = 1
        - `lsu_axi_araddr` = 0x00002AB0
    - **Cycle i+8:** (Note that the simulated memory is not equal to the actual DDR memory on the Nexys A7 board, and thus the latency seen in the simulation is not the same as the latency on the board, which we will analyse later), the read value is received through the AXI bus from the External Memory:
        - `lsu_axi_rvalid` = 1
        - `lsu_axi_rdata` = 0x0
- The processor computes the addition (`add t3, t3, t5`) in the Secondary ALU and writes it to the Register File, as explained in Lab 15. (This is not shown in the figure, but you can analyse it in your own simulation.)
    - **Cycle i+15:** The processor writes the result to `t3`: `t3` = 0x2AB0.
- Finally, the processor writes the value of `t3` to the DDR External Memory (red square):
    - **Cycle i+16:** the effective address and the data are sent to the External Memory through the AXI bus:
        - `lsu_axi_awvalid` = 1

        - `lsu_axi_awaddr` = 0x00002AB0
        - `lsu_axi_wvalid` = 1
        - `lsu_axi_wdata` = 0x0000000000002AB0

    - **Cycle i+21:** (again, this latency is different in simulation and on the board), the External 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 HW Counters, measure the number of cycles, instructions, loads and stores in the program from Figure 2. How much time in total (both for reading and writing) does it take to access the DDR External Memory? You can compare the execution when using the DDR memory as in Figure 3 and when using the DCCM (another PlatformIO project is provided at *[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_DCCM/*, which contains the same program prepared for reading from / writing to the DCCM). Remember that the simulated memory is not the same as the actual DDR memory on the Nexys A7 board, thus the read/write latency observed in the simulation and in the execution on the board differs.

TASK: Use the example from *[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_ExtMem* to estimate the DDR External Memory read latency using the HW Counters. As in the previous task, you can use the example from *[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_DCCM* to compare with a program with no stalls due to the memory accesses. Remember that the simulated memory is not the same as the actual DDR memory on the Nexys A7 board, thus the read latency observed in the simulation and in the execution on the board differs.

TASK: A quite complex but very interesting exercise is to analyse the Memory Controller used in the RVfpga System. Remember that you can find the modules that make up this controller in folder *[RVfpgaPath]/RVfpga/src/LiteDRAM*, and that the top module is implemented in file *litedram_top.v* inside that folder. You can start with the simulation from Figure 3 and add and analyse some signals from the LiteDRAM controller.

3. INSTRUCTION FETCH FROM THE INSTRUCTION CACHE

In this section we analyse the operation of the Instruction Cache (I$) available in the RVfpga System. We first describe how the I$ 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 SweRV EH1 (Section 3.D).

A. Instruction Cache Configuration

The RVfpga System's I$ is highly configurable based on a set of parameters defined in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh*. The default RVfpga System has the following I$ parameters:

```
`define RV_ICACHE_SIZE 16
`define RV_ICACHE_DATA_CELL ram_256x34
`define RV_ICACHE_IC_INDEX 8
`define RV_ICACHE_TAG_CELL ram_64x21
`define RV_ICACHE_ENABLE 1
`define RV_ICACHE_IC_ROWS 256
`define RV_ICACHE_TAG_DEPTH 64
`define RV_ICACHE_TAG_HIGH 12
`define RV_ICACHE_TAG_LOW 6
`define RV_ICACHE_IC_DEPTH 8
`define RV_ICACHE_TADDR_HIGH 5
```

However, some of the above parameters are overridden in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/global.h*:

```
localparam ICACHE_TAG_HIGH  = `RV_ICACHE_TAG_HIGH;
localparam ICACHE_TAG_LOW   = `RV_ICACHE_TAG_LOW;
localparam ICACHE_IC_DEPTH  = `RV_ICACHE_IC_DEPTH;
localparam ICACHE_TAG_DEPTH = `RV_ICACHE_TAG_DEPTH;
```

Thus, the I$ has the following configuration:

Characteristic	Value
I$ Size	
Data Array (without parity information) **Parity information** for data:	16 Kibytes 1 Kibyte (4 Bytes per block)
Tag Array (without parity information) **Parity information** for tags	640 Bytes 32 Bytes (1 bit per tag)
LRU State	24 Bytes (3 bits per set)
Valid Bit	32 Bytes (1 valid bit per tag)
Associativity (not configurable)	4 ways
Block Size	64 Bytes
Number of blocks (Size/Block Size=16Ki/64)	256 blocks
Number of blocks per way (# blocks/Assoc.=256/4)	64 blocks

According to this configuration, the I$ used in the RVfpga System is shown in Figure 4.

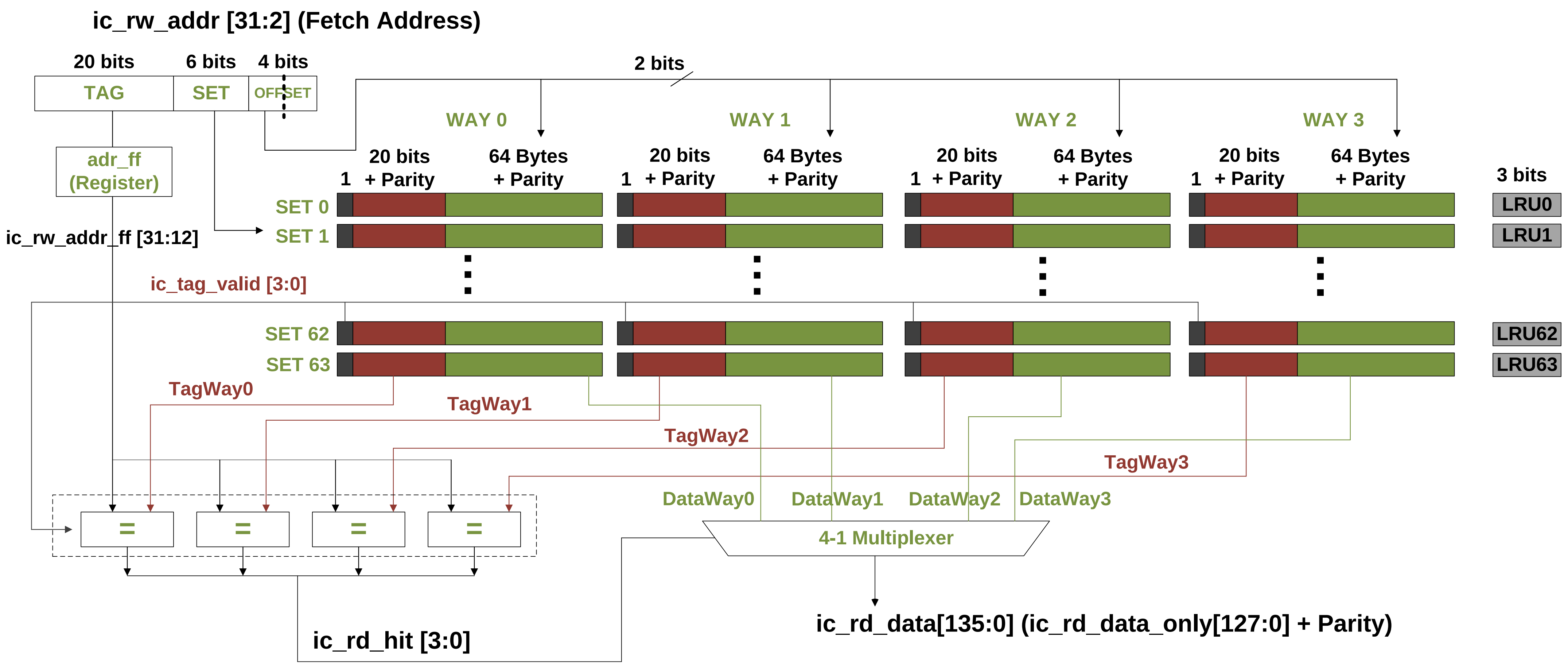


Figure 4. I$ internal design. The input signal to the I$ (`ic_rw_addr`) and the output signal from the I$ (`ic_rd_data`) is provided from/to the Cache Controller (module *ifu_mem_ctl*), as we explained in Figure 3 of Lab 11 (repeated below as Figure 8).

The RVfpga System's I$ is implemented in module **ifu_ic_mem**, included in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/ifu/ifu_ic_mem.sv*. This module instantiates two other modules:

- **IC_TAG**: This module includes the Tag Array (the red boxes shown in Figure 4) 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 signals `TagWay0`–`TagWay3`, as shown in Figure 4 and in simulation (shown later). Note that the processor uses a signal called `w_tout` to read the tags. Signals `TagWay0`–`TagWay3` are extracted from `w_tout` in lines 583-590 of file *ifu_ic_mem.sv*.

- **IC_DATA**: This module is the Data Array, which includes the green boxes shown in Figure 4, as well as the 4:1 multiplexer that selects the data from the way where a hit takes place. Each way is physically split into 4 banks (not shown in the figure). 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 the 6-bit SET field and on 2 bits of the OFFSET field of the Fetch Address, this module selects and outputs, in signal `ic_rd_data`, the 128-bit instruction bundle plus some parity bits that must be sent to the SweRV EH1 processor. Note that signal `ic_rd_data_only` is the same as signal `ic_rd_data` without the parity information, thus we use it in our simulations below.

  The data read from the I$ is in signals `DataWay0`–`DataWay3`. Note that these signals are used both in the figure and in simulation, but they not used by the processor. The actual processor signal is called `wb_dout_way_with_premux`, from which signals `DataWay0`–`DataWay3` are obtained in lines 313-320 of file *ifu_ic_mem.sv*.

<u>TASK</u>: Analyse module **ifu_ic_mem** to understand how the elements in Figure 4 are implemented.

B. Instruction Cache Miss Management

In this section, we show how instruction misses are managed in the processor. The example in Figure 5 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 map into a single I$ block. Recall that the I$ block size is 64 bytes. Thus the first `add` instruction must be aligned on 64-byte boundary. Folder *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example* provides the PlatformIO 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 5. Example program

Open the project in PlatformIO, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example/.pio/build/swervolf_nexys/firmware.dis*). Notice that the first `add` instruction (0x005282b3) is placed at address 0x000001c0 (which is aligned on a 64-byte boundary) and the sixteenth one (0x011888b3) is placed at address 0x000001fc (the last word in the block).

```
0x000001c0:      005282b3            add   t0,t0,t0
     …               …                      …
0x000001fc:      011888b3            add   a7,a7,a7
```

Figure 6 shows the simulation of the region around the 16 `add` instructions (from 28900 ps to 30220 ps). The figure in the middle (main one) shows the execution of the region of interest for our analysis. The figures on the top and the two on the bottom zoom into specific regions of the main figure.

TASK: Replicate the simulation from Figure 6 on your own computer. To do so, follow the next steps (as described in detail in Section 7 of the GSG):
- If necessary, generate the simulation binary (*Vrvfpgasim*).

- In PlatformIO, open the project provided at: *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example*.
- Update the path to the RVfpga simulation binary (*Vrvfpgasim*) in file *platformio.ini*.
- Generate the simulation trace with Verilator (Generate Trace).
- Open the trace on GTKWave.
- Use file *test1_Miss.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example*) for opening the same signals as the ones shown in Figure 6. For that purpose, on GTKWave, click on *File → Read Tcl Script File* and select the *test1_Miss.tcl* file.
- Click on *Zoom In* (⊕) several times and analyse the region from 28900 ps to 30220 ps.

You can also analyse some things in more detail, such as the write to the I$ or the bypass of the initial instructions.

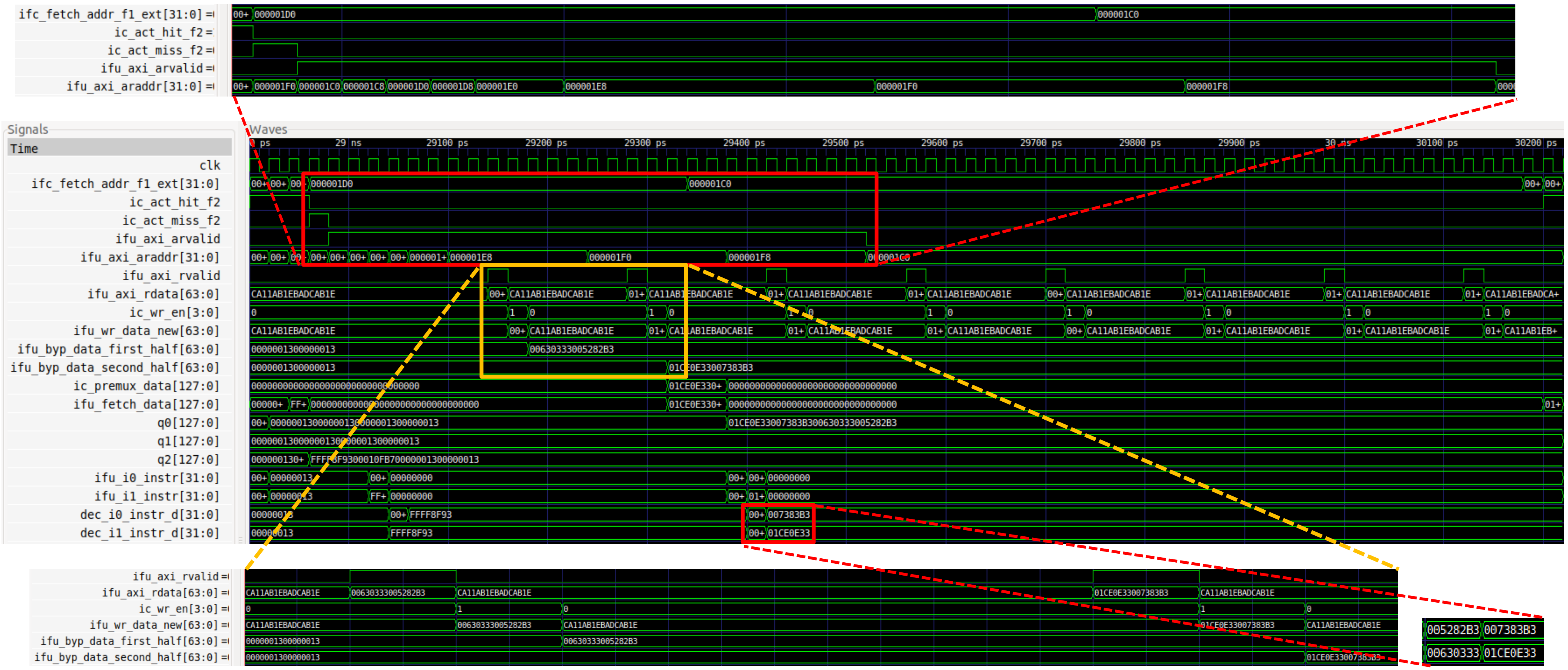


Figure 6. Simulation of the program from Figure 5 showing an I$ miss

This example illustrates how an I$ miss is handled in SweRV EH1. 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 the DDR External Memory into the I$.

- In the figure on the top you can see that an I$ miss is signalled around 29ns (`ic_act_miss_f2` = 1), which triggers the request of the block through the AXI bus (`ifu_axi_arvalid` = 1).

- Then, the eight 64-bit chunks that make up the target block are requested sequentially through the AXI bus.
    - Signal `ifu_axi_arvalid` goes high for 27 cycles. This signal indicates that the channel is signalling valid read address and control information.
    - During these 27 cycles where `ifu_axi_arvalid` = 1 the initial addresses of the eight 64-bit chunks are provided sequentially through the AXI bus using signal `ifu_axi_araddr`, which provides the 8 addresses that must be read from the DDR Memory:
        - `ifu_axi_araddr` = 0x000001c0
        - `ifu_axi_araddr` = 0x000001c8
        - `ifu_axi_araddr` = 0x000001d0
        - `ifu_axi_araddr` = 0x000001d8
        - `ifu_axi_araddr` = 0x000001e0
        - `ifu_axi_araddr` = 0x000001e8
        - `ifu_axi_araddr` = 0x000001f0
        - `ifu_axi_araddr` = 0x000001f8

- The middle figure shows the eight 64-bit chunks arriving sequentially to the processor through the AXI bus in signal `ifu_axi_rdata`.
    - Signal `ifu_axi_rvalid`, which indicates that the channel is signalling the required read data, goes high for one cycle every 7 cycles.
    - Each of the eight 64-bit chunks (each containing two instructions) is provided in signal `ifu_axi_rdata` (this cannot be seen in Figure 6 – 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 two bottom figures show that each of the eight 64-bit chunks is written into the I$ right after their arrival to the cache controller. For example, the first two 64-bit chunks are written as follows:
    - Signal `ic_wr_en` goes high and at the same time `ic_wr_data` = 0x00630333005282b3. Thus, the first and second `add` instructions are written into the I$.
    - Several cycles later, signal `ic_wr_en` goes high again and at the same time `ic_wr_data` = 0x01ce0e33007383b3. Thus, the third and fourth `add` instructions are written into the I$.

- Finally, you can see that the four instructions are bypassed from the I$ controller to the pipeline (signals `ifu_byp_data_first_half` and `ifu_byp_data_second_half`) so that it can restart execution as soon as possible after the I$ miss. Several cycles later, the four instructions arrive at the Decode Stage (see the figure on the bottom right corner that zooms into signals `dec_i0_instr_d` and `dec_i1_instr_d`).

C. Instruction Cache Hit Management

In this section we work with the same example from Section 3.B (Figure 5), but we now focus on analysing I$ hits. Figure 7 shows the second iteration of the loop when executing the program in Figure 5 (any iteration would be valid except the first one, which, as we analysed in Figure 6, experiences misses in the I$).

<u>TASK</u>: Replicate the simulation from Figure 7 on your own computer. Use file *test1_Hit.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example*). *Zoom In* ( ) several times and move to 34680ps.

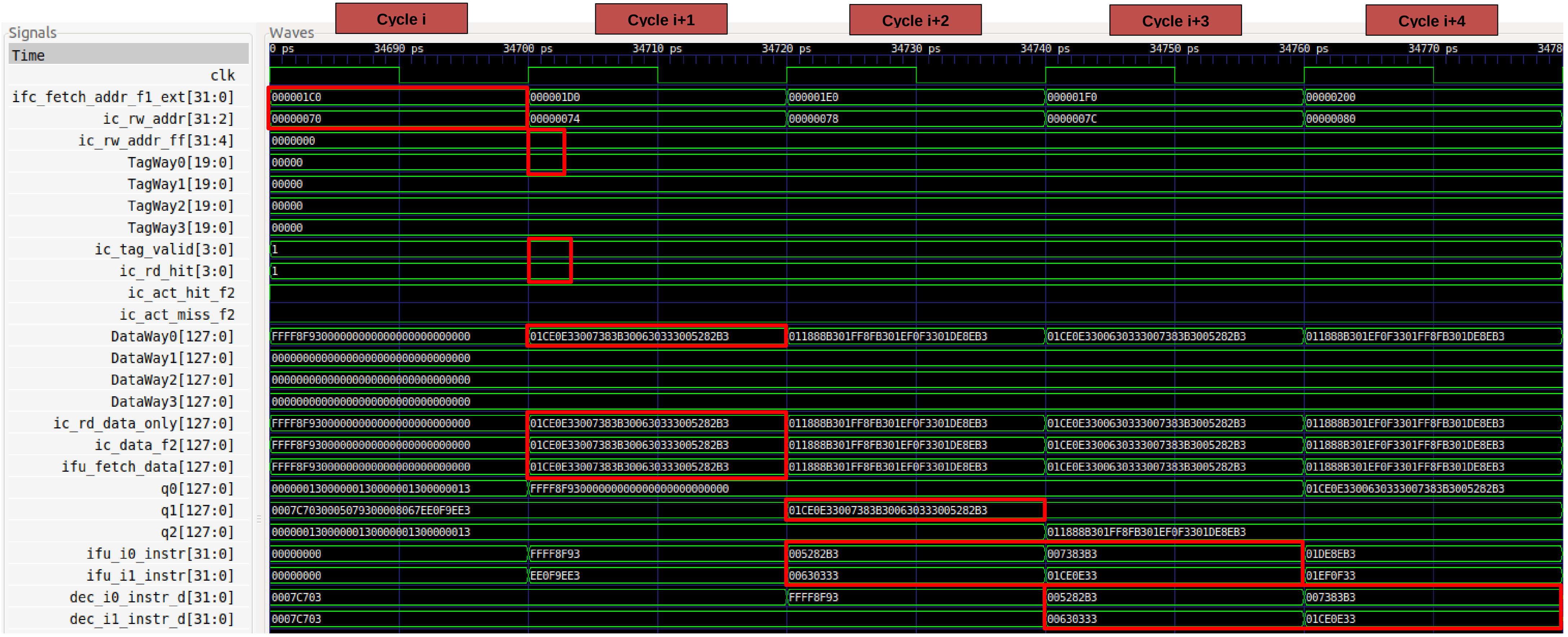


Figure 7. Simulation waveform for the program in Figure 5 illustrating an I$ hit

Analyse the simulation waveform in Figure 7, which includes some of the signals from Figure 3 of Lab 11 that explained the FC1 and FC2 stages. We repeat a diagram of those stages in Figure 8 below, for convenience. The simulation also includes some of the signals shown in Figure 4 of the current lab. Note that Figure 4 shows the I$ design, which is shown as a black box in Figure 8 below.

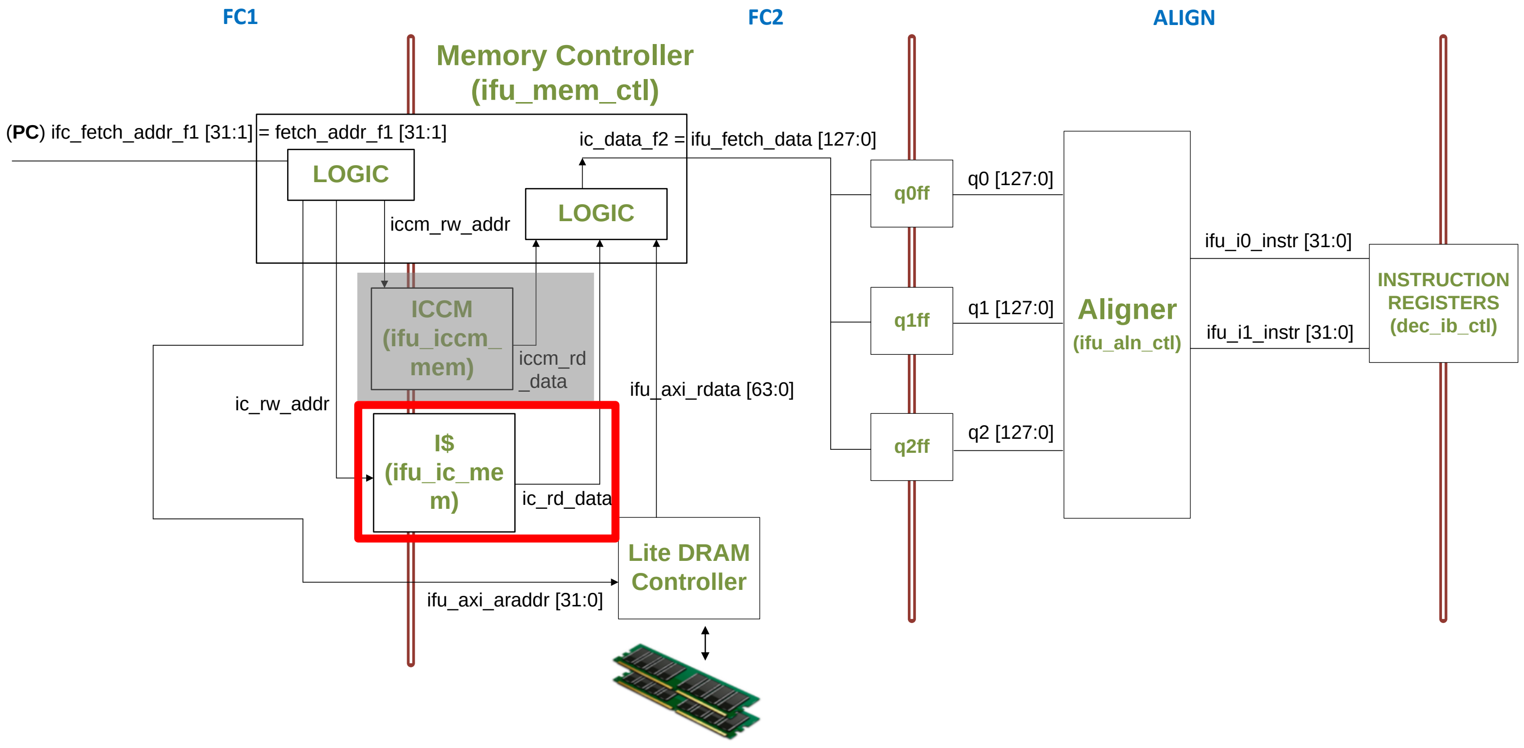


Figure 8. FC1, FC2 and Align stages

The i$ hit occurs as follows, as shown in Figure 7:

- **Cycle i:** The address of the first add instruction (`add t0,t0,t0`) in the program from Figure 5 is given in signal `ifc_fetch_addr_f1_ext` (`ifc_fetch_addr_f1_ext` = 0x000001c0). This signal is passed to the I$ except for its two least significant bits, which are needed because instructions are 4-byte (32-bit) aligned. Thus, `ic_rw_addr` = 0x0000070.

  The Tag Array and the Data Array are accessed using a subset of the Fetch Address, as shown in Figure 4. The result of the access will be available in the next cycle.

- **Cycle i+1:** The four tags, one per cache way, are in signals `TagWay0-TagWay3`. These are compared to the TAG field of the Fetch Address that was registered in `adr_ff` (output signal `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` = 0001), thus a hit is signalled in Way 0: `ic_rd_hit` = 0001.

  Also, four 128-bit bundles are in signals `DataWay0-DataWay3`. The 4:1 multiplexer from Figure 4 selects the data provided by Way 0 (i.e. `DataWay0`). Thus:

  `ic_rd_data_only` = 0x01ce0e33007383b300630333005282b3

Note that the signal that is shown in Figure 8 is `ic_rd_data`, which is the same as `ic_rd_data_only` plus the parity information.

These 128 bits are propagated to the Align stage as shown in Figure 8.

`ifu_fetch_data` = `ic_data_f2` = `ic_rd_data_only` = 0x01ce0e33007383b300630333005282b3

Note that these 128 bits correspond to the first four add instructions.

- **Cycle i+2:** The first and second `add` instructions are extracted in the Align stage from buffer `q1`:
  - `ifu_i0_instr` = 0x005282b3
  - `ifu_i1_instr` = 0x00630333

- **Cycle i+3:** The third and fourth `add` instructions are extracted in the Align stage and, at the same time, the first and second `add` instructions are decoded:
  - `ifu_i0_instr` = 0x007383b3
  - `ifu_i1_instr` = 0x01ce0e33
  - `dec_i0_instr_d` = 0x005282b3
  - `dec_i1_instr_d` = 0x00630333

- **Cycle i+4:** Finally, the third and fourth `add` instructions are decoded:
  - `dec_i0_instr_d` = 0x007383b3
  - `dec_i1_instr_d` = 0x01ce0e33

D. Instruction Cache Replacement Policy

This section describes the RVfpga System's cache replacement policy. As explained by Harris & Harris in Section 8.3.3 of [DDCARV], in set 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, SweRV EH1 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 SweRV EH1

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) in an N-way associative cache.

Thus, in the case of SweRV EH1, where a 4-way Instruction Cache is used, 3 bits are required per set to track the access history to the different ways.

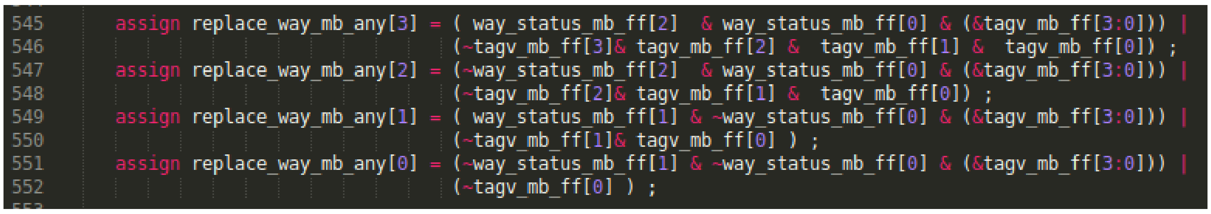
As explained in Section 3.B, when an I$ miss occurs, the block must be requested from the DDR External Memory. When the DDR External Memory supplies the cache block, it must be written into the I$. The SET field of the Fetch Address determines the I$ set where the new block must be written (see Figure 4). 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 the four 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 3-bit LRU State of the Set, according to the following table (where *x* means don't care):

LRU State	**Way to replace**
x00	Way 0
x10	Way 1
0x1	Way 2
1x1	Way 3

The following Verilog snippet (Figure 9), extracted from module **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.



```
    assign replace_way_mb_any[3] = ( way_status_mb_ff[2]  & way_status_mb_ff[0] & (&tagv_mb_ff[3:0])) |
                                   (~tagv_mb_ff[3]& tagv_mb_ff[2] &  tagv_mb_ff[1] &  tagv_mb_ff[0]) ;
    assign replace_way_mb_any[2] = (~way_status_mb_ff[2]  & way_status_mb_ff[0] & (&tagv_mb_ff[3:0])) |
                                   (~tagv_mb_ff[2]& tagv_mb_ff[1] &  tagv_mb_ff[0]) ;
    assign replace_way_mb_any[1] = ( way_status_mb_ff[1] & ~way_status_mb_ff[0] & (&tagv_mb_ff[3:0])) |
                                   (~tagv_mb_ff[1]& tagv_mb_ff[0] ) ;
    assign replace_way_mb_any[0] = (~way_status_mb_ff[1] & ~way_status_mb_ff[0] & (&tagv_mb_ff[3:0])) |
                                   (~tagv_mb_ff[0] ) ;
```

Figure 9. Verilog code for selecting which way must be replaced

The signals used in the Verilog snippet from Figure 9 are the following:

- **`replace_way_mb_any`** (4 bits): holds a one-hot value that is 1 for the way that must be replaced.

- **`way_status_mb_ff`** (3 bits): holds the LRU state of the new block's set.

- **`tagv_mb_ff`** (4 bits): holds the valid bits of the new block's set; ways that are valid have valid bits of 1, whereas invalid ways have valid bits that are 0.

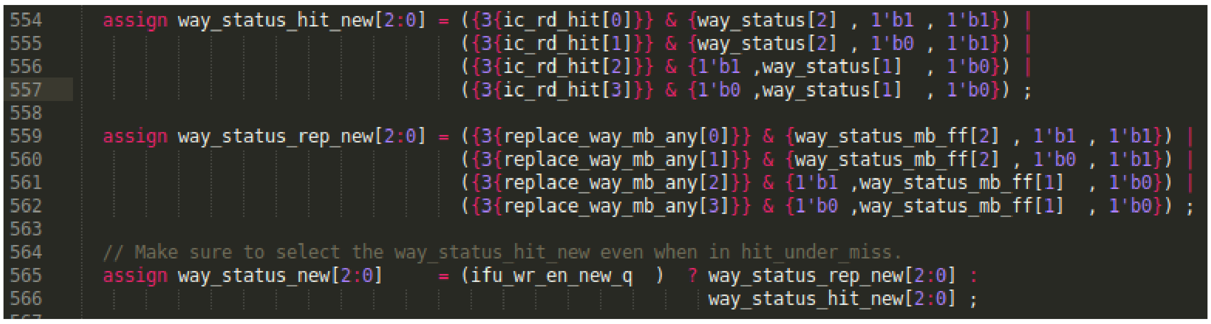
<u>TASK</u>: Analyse the Verilog code from Figure 9 and explain how it operates based on the above explanations.

When a hit or a miss happens in the I$, the LRU state of the set must be updated according to the following table (where '-' means that the bit remains unchanged):

Written Way	Next LRU state
Way 0	-11
Way 1	-01
Way 2	1-0
Way 3	0-0

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 is to protect the last accessed data from eviction by inversing the nodes towards it.

The following Verilog snippet (Figure 10), extracted from module **ifu_mem_ctl**, implements the logic for this update of the LRU state.



```
assign way_status_hit_new[2:0] = ({3{ic_rd_hit[0]}} & {way_status[2] , 1'b1 , 1'b1}) |
                                 ({3{ic_rd_hit[1]}} & {way_status[2] , 1'b0 , 1'b1}) |
                                 ({3{ic_rd_hit[2]}} & {1'b1 ,way_status[1]  , 1'b0}) |
                                 ({3{ic_rd_hit[3]}} & {1'b0 ,way_status[1]  , 1'b0}) ;

assign way_status_rep_new[2:0] = ({3{replace_way_mb_any[0]}} & {way_status_mb_ff[2] , 1'b1 , 1'b1}) |
                                 ({3{replace_way_mb_any[1]}} & {way_status_mb_ff[2] , 1'b0 , 1'b1}) |
                                 ({3{replace_way_mb_any[2]}} & {1'b1 ,way_status_mb_ff[1]  , 1'b0}) |
                                 ({3{replace_way_mb_any[3]}} & {1'b0 ,way_status_mb_ff[1]  , 1'b0}) ;

// Make sure to select the way_status_hit_new even when in hit_under_miss.
assign way_status_new[2:0]     = (ifu_wr_en_new_q  )  ? way_status_rep_new[2:0] :
                                                        way_status_hit_new[2:0] ;
```

Figure 10. Verilog snippet for updating the LRU state

The signals used in the Verilog snippet from Figure 10 are the following:

- **`ic_rd_hit`** (4 bits): holds the way where a hit has taken place.
- **`way_status`** and **`way_status_mb_ff`** (3 bits each): hold the previous LRU state of the set where the hit or replacement has taken place.
- **`ifu_wr_en_new_q`** (1 bit): is 1 if a replacement has occurred.
- **`way_status_new`** (3 bits): holds the new LRU state for the set just referenced on a hit or a miss.
- **`replace_way_mb_any`** (4 bits): holds a one-hot value that is 1 for the way that must be replaced. This signal was also explained below Figure 9.

TASK: Analyse the Verilog code from Figure 10 and explain how it operates based on the above explanations.

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

For analysing the replacement policy of SweRV EH1, we provide a new example in folder *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_LRU_Example*. In this example (Figure 11), five different I$ blocks are accessed inside an infinite loop and all five of these blocks map to the same I$ set: SET = 8. For that purpose, we create an infinite loop that contains five `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$ (see Section 3.A and Figure 4).

```
Set8_Block1:   j Set8_Block2        # This j instruction is at address 0x00000200
               INSERT_NOPS_1023

Set8_Block2:   j Set8_Block3        # This j instruction is at address 0x00001200
               INSERT_NOPS_1023

Set8_Block3:   j Set8_Block4        # This j instruction is at address 0x00002200
               INSERT_NOPS_1023

Set8_Block4:   j Set8_Block5        # This j instruction is at address 0x00003200
               INSERT_NOPS_1023

Set8_Block5:   j Set8_Block1        # This j instruction is at address 0x00004200
```

Figure 11. Example program showing `j` instructions that map to the same set

Open the project in PlatformIO, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_LRU_Example/.pio/build/swervolf_nexys/firmware.dis*). Notice the following:

- The first `j` instruction (`j Set8_Block2`) is at address 0x00000200. According to the address division shown in Figure 4 for accessing the I$:
  I$ Address in binary = 000000000000000000000010000000~~00~~
  TAG = 0x0
  SET = 0x8
  OFFSET = 0x0

- The second `j` instruction (`j Set8_Block3`) is at address 0x00001200. According to the address division shown in Figure 4 for accessing the I$:
  I$ Address in binary = 000000000000000000010010000000~~00~~
  TAG = 0x1
  SET = 0x8
  OFFSET = 0x0

- The third `j` instruction (`j Set8_Block4`) is at address 0x00002200. According to the address division shown in Figure 4 for accessing the I$:
  I$ Address in binary = 000000000000000000100010000000~~00~~
  TAG = 0x2
  SET = 0x8
  OFFSET = 0x0

- The fourth `j` instruction (`j Set8_Block5`) is at address 0x00003200. According to the address division shown in Figure 4 for accessing the I$:
  I$ Address in binary = 000000000000000000110010000000~~00~~
  TAG = 0x3
  SET = 0x8
  OFFSET = 0x0

- The fifth `j` instruction (`j Set8_Block1`) is at address 0x00004200. According to the address division shown in Figure 4 for accessing the I$:

I$ Address in binary = 00000000000000000100001000000000~~00~~
TAG = 0x4
SET = 0x8
OFFSET = 0x0

In this program (Figure 11), when the first iteration is executed, Set 8 is initially empty. Figure 12 shows the theoretical changes to Set 8 in the I$ while executing the first iteration. Later, we show several Verilator simulations that confirm these theoretical explanations.

SET 8 after execution of the first j instruction at 0x200

Valid	Tag	Data	
1	00000000000000000000	`j Set8_Block2 | nop | … | nop`	WAY 0
0			WAY 1
0			WAY 2
0			WAY 3

LRU STATE = 011

SET 8 after execution of the second j instruction at 0x1200

1	00000000000000000000	`j Set8_Block2 | nop | … | nop`	WAY 0
1	00000000000000000001	`j Set8_Block3 | nop | … | nop`	WAY 1
0			WAY 2
0			WAY 3

LRU STATE = 001

SET 8 after execution of the third j instruction at 0x2200

1	00000000000000000000	`j Set8_Block2 | nop | … | nop`	WAY 0
1	00000000000000000001	`j Set8_Block3 | nop | … | nop`	WAY 1
1	00000000000000000010	`j Set8_Block4 | nop | … | nop`	WAY 2
0			WAY 3

LRU STATE = 100

SET 8 after execution of the fourth j instruction at 0x3200

1	00000000000000000000	`j Set8_Block2 | nop | … | nop`	WAY 0
1	00000000000000000001	`j Set8_Block3 | nop | … | nop`	WAY 1
1	00000000000000000010	`j Set8_Block4 | nop | … | nop`	WAY 2
1	00000000000000000011	`j Set8_Block5 | nop | … | nop`	WAY 3

LRU STATE = 000

SET 8 after execution of the fifth j instruction at 0x4200

1	00000000000000000100	`j Set8_Block1 | nop | … | nop`	WAY 0
1	00000000000000000001	`j Set8_Block3 | nop | … | nop`	WAY 1
1	00000000000000000010	`j Set8_Block4 | nop | … | nop`	WAY 2
1	00000000000000000011	`j Set8_Block5 | nop | … | nop`	WAY 3

LRU STATE = 011

Figure 12. Set 8 of the I$ during execution of the first loop iteration in Figure 11

The following Verilator simulations show the cache signals during the first iteration of the loop, and they confirm the analysis shown in Figure 12. Figure 13 shows the Verilator simulation of the program after executing the first `j` instruction (`j Set8_Block2`). Again, this instruction address (0x200) maps to Set 8 of the I$. That set is initially empty: `tagv_mb_ff` = 0000. Thus, according to the Binary Tree LRU policy, the new block must be written in Way 0: `replace_way_mb_any` = `ic_wr_en` = 0001. The LRU state of Set 8 is updated as follows: `way_status_new` = 011.

Recall from Section 3.B that the block is read from the DDR Memory and written into the I$ in 64-bit chunks. Figure 13 illustrates the write of the tag and the two first instructions of the new block into SET 8:

`ic_rw_addr_q[11:4]` = 00100000 (SET 8)
`ic_tag_wr_data[19:0]` = 0x0 (the most significant bit is used for error correction and not included here)
`ic_wr_data1[31:0]` = 0x0000106F (`j Set8_Block2`)
`ic_wr_data2[31:0]` = 0x00000013 (`nop`)

(`ic_wr_data1` and `ic_wr_data2` are signals created for the sake of clarity, but the signal used in the I$ is called `ic_wr_data[67:0]`, which includes the two instructions plus some parity information).

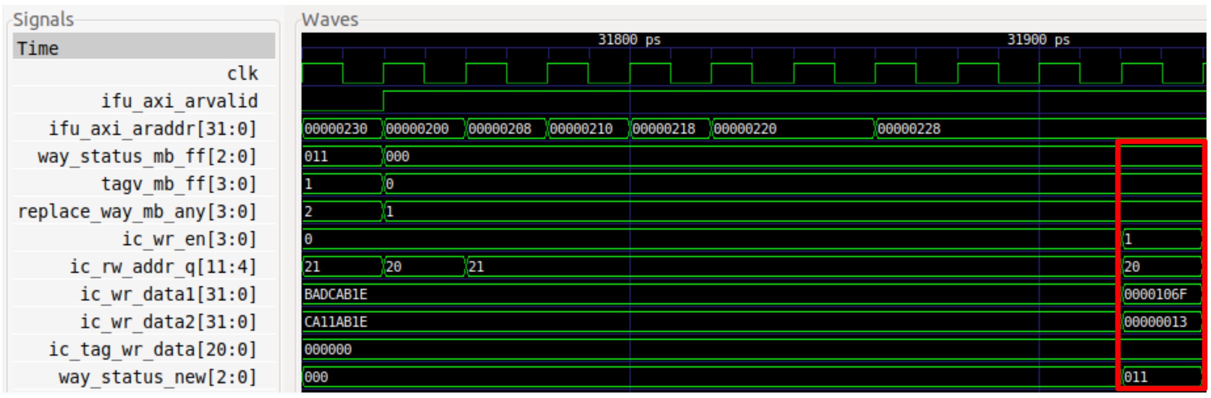


Figure 13. LRU state of set 8 after executing the first `j` instruction

Figure 14 illustrates the Verilator simulation after the execution of the second `j` instruction (`j Set8_Block3`). This instruction address (0x1200) also maps to Set 8 of the I$. Only way 0 is valid in that set: `tagv_mb_ff` = 0001. Thus, according to the Binary Tree LRU policy, the new block must be written in Way 1: `replace_way_mb_any` = `ic_wr_en` = 0010. The LRU state of Set 8 is updated as follows: `way_status_new` = 001.

As before, Figure 14 illustrates the write of the first two instructions of the new block into SET 8:

`ic_rw_addr_q[11:4]` = 00100000 (SET 8)
`ic_tag_wr_data[19:0]` = 0x1 (the most significant bit is used for error correction and not included here)
`ic_wr_data1[31:0]` = 0x0000106F (`j Set8_Block3`)
`ic_wr_data2[31:0]` = 0x00000013 (`nop`)

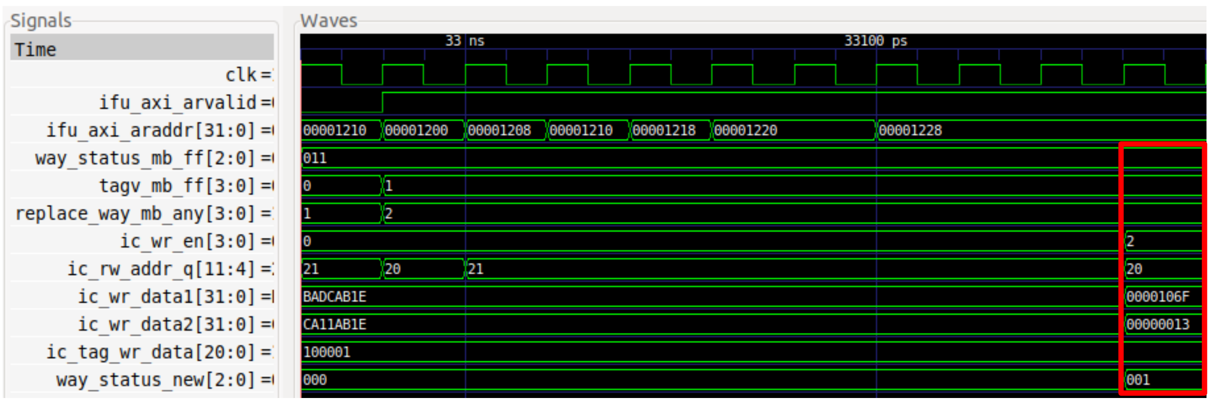


Figure 14. LRU state of set 8 after executing the second `j` instruction

Figure 15 illustrates the Verilator simulation after executing the fifth `j` instruction (`j Set8_Block1`). This instruction address (0x4200) also maps to Set 8 of the I$. However, as opposed to the previous situation, in this case the set is full: `tagv_mb_ff` = 1111. Thus, according to the Binary Tree LRU policy, the new block must be written to Way 1: `replace_way_mb_any` = `ic_wr_en` = 0001. The LRU state of Set 8 is updated as follows: `way_status_new` = 011.

As before, Figure 15 illustrates the write of the two first instructions of the new block into SET 8:

`ic_rw_addr_q[11:4]` = 00100000 (SET 8)
`ic_tag_wr_data[19:0]` = 0x4 (the most significant bit is used for error correction and not included here)
`ic_wr_data1[31:0]` = 0x800fc06f (`j Set8_Block1`)
`ic_wr_data2[31:0]` = 0x00008067 (`ret`)

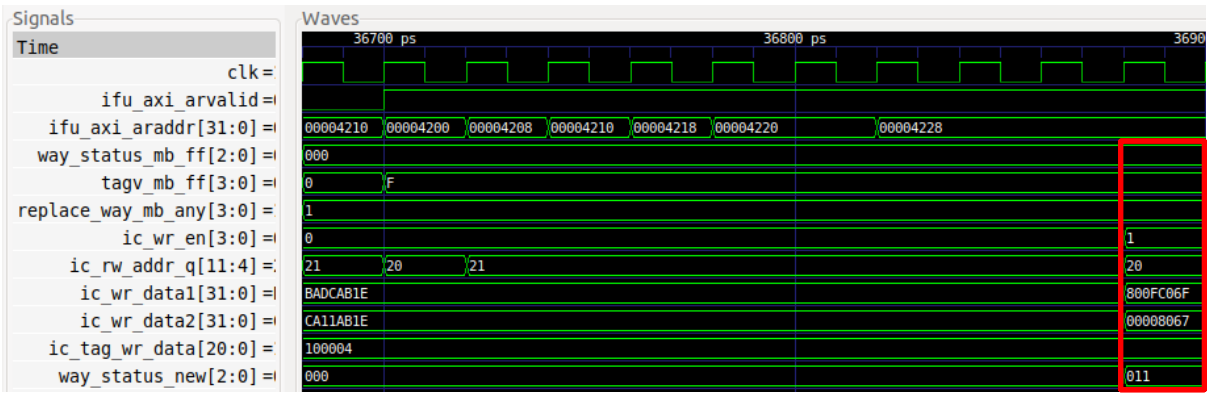


Figure 15. LRU state of set 8 after executing the fifth `j` instruction

TASK: Replicate the simulation from Figure 13-Figure 15 on your own computer.

4. EXERCISES

1) Transform the infinite loop from Figure 11 into a loop with 0x10000 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) Use the program from Figure 5 to analyse an I$ hit from the point of view of the I$ Replacement Policy.

3) Extend Figure 6 to analyse in detail how each 64-bit chunk is written in the I$.

4) Analyse in simulation and on the board other I$ configurations, such as an I$ with a different block size. Recall that the number of ways cannot be modified.

5) Analyse the logic that checks the correctness of the parity information from the Data Array and from the Tag Array.