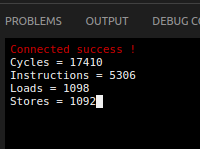
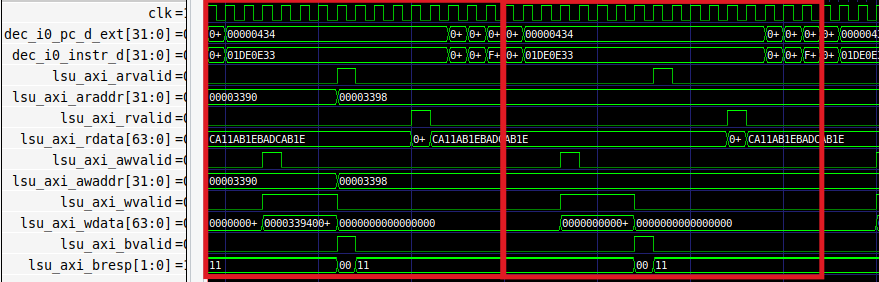
**TASK:** Usingthe HW Counters, measure the number of cycles, instructions, loads and stores in the program from Figure 2. How much time in total (both for reading and writing) does it take to access Main Memory? You can compare the execution when using the DDR memory as in Figure 3 and when using the DCCM (another PlatformIO project is provided at *[RVfpgaBooleanPath]/Labs/Lab19/LW-SW\_Instruction\_DCCM/*, which contains the same program prepared for reading from / writing to the DCCM).

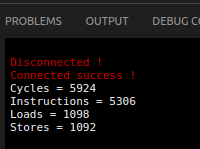


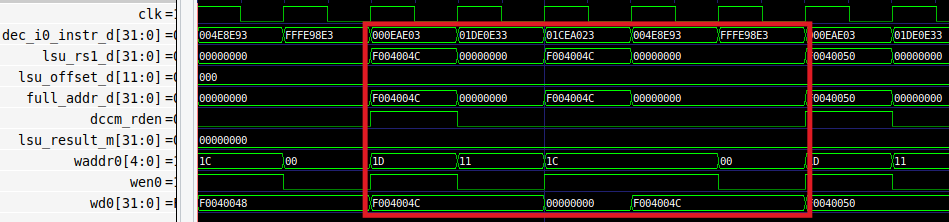
* The loop contains 5 instructions.
* Ideally, IPC could be up to 1.
* However, we miss 11.5 cycles per iteration due to the read/write latency to Main Memory.
* This is coherent with the RVfpga-Trace simulation shown in the lab and repeated next for 2 consecutive iterations:



* The first iteration shown in the previous figure takes 16 cycles (5 + 11) and the second iteration takes 17 cycles (5 + 12).
* This pattern is repeated, thus, on average, we need 11.5 cycles per iteration for reading/writing Main Memory.

If we now execute the program that uses the DCCM, we obtain:

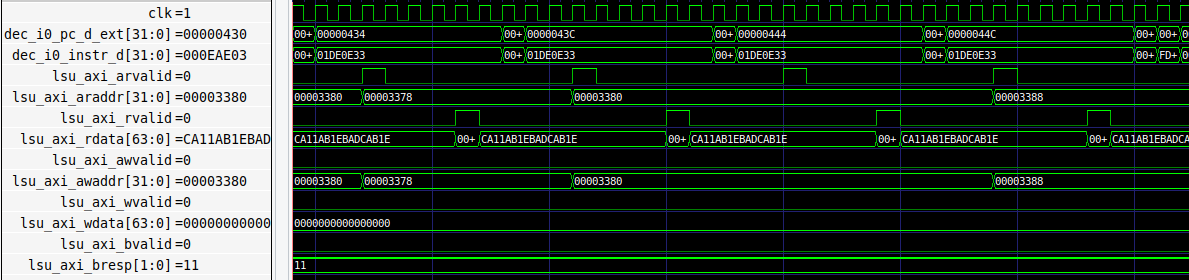




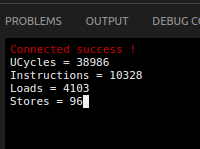
* In this case, we achieve the ideal IPC of 1, as the DCCM has 1 cycle read latency.

**TASK:** Use the example from *[RVfpgaBooleanPath]/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 *[RVfpgaBooleanPath]/Labs/Lab19/LW\_Instruction\_DCCM* to compare with a program with no stalls due to the memory accesses.

**Execution in Main Memory:**

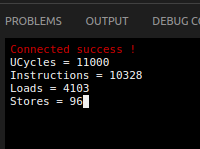


* It takes 38 cycles to execute 1 iteration.
* Each iteration has 10 instructions, being 4 of them lw instructions.
* Thus: 38-10 = 28 and 28/4 = 7.
* It takes 7 cycles to carry out the read from Main Memory.



* The same results are obtained when executing on the board.

**Execution in DCCM:**

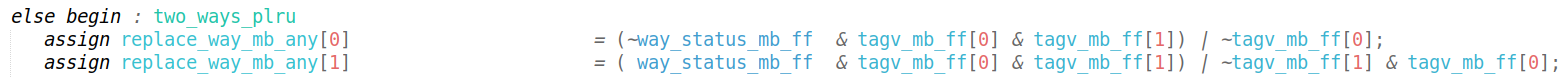


* In this case, we achieve the ideal IPC of 1, as the DCCM has 1 cycle read latency.

**TASK:** Analyse module **ifu\_ic\_mem** and the parameters of file *el2\_param.vh* to understand how the elements in Figure 3 are implemented.

Solution not provided for this exercise.

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

  
If both ways are invalid (i.e. **tagv\_mb\_ff = 00**), way 0 must be replaced first:

* replace\_way\_mb\_any[0] = 1, as the second operand of the OR, which is ~tagv\_mb\_ff[0], is 1.
* replace\_way\_mb\_any[1] = 0, as the two operands of the OR are 0.

If there is one invalid way, this is the one replaced.

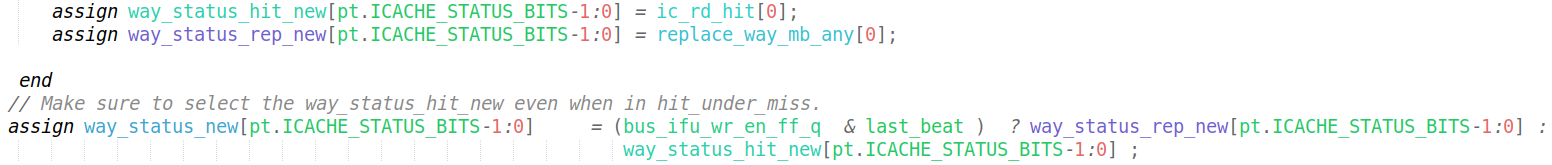
* If way 0 is invalid, the second operand of the OR, which is ~tagv\_mb\_ff[0], is 1.
* If way 1 is invalid and way 0 is valid, the second operand of the OR, which is ~tagv\_mb\_ff[1] & tagv\_mb\_ff[0], is 1.

If both ways are valid, signal **way\_status\_mb\_ff** holds the LRU state (thus, the least recently used way, or the way to replace first) of the selected set, determines the way to replace.

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

Solution not provided for this exercise.

**TASK:** Analyse the Verilog code from Figure 9 and explain how it operates based on the above explanations.

  
The new value of the LRU state is determined by signal **way\_status\_new**.

* If there was a hit, signal **ic\_rd\_hit** determines the new value, as it holds the way where the hit has taken place.
* If there was a replacement, signal **replace\_way\_mb\_any** determines the new value, as it holds the way that has been replaced.

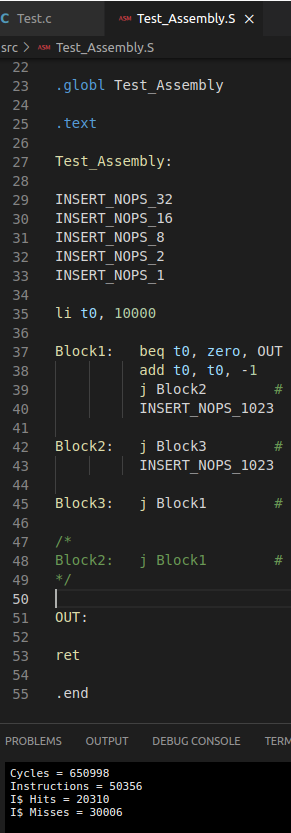
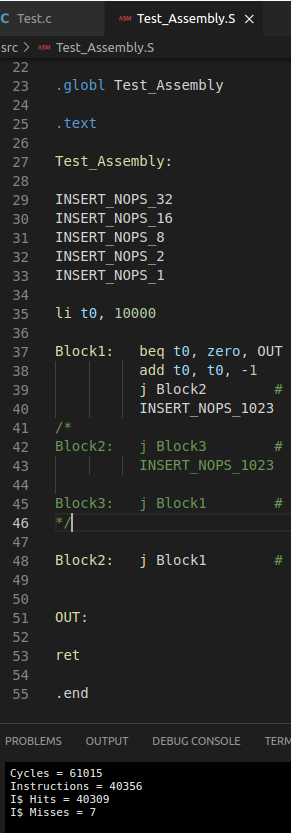
**TASK:** Analyse the Verilog code that performs the same functionality on a 4-way I$.

Solution not provided for this exercise.

# EXERCISES

1. Transform the infinite loop from Figure 10 into a loop with 10000 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.

A Catapult project is provided at: *[RVfpgaBooleanPath]/Labs/RVfpgaLabsSolutions/Lab19/InstructionMemory\_LRU\_Example\_FiniteLoop*

* The number of I$ misses in the code with 3 jump instructions is 3 per iteration (30000 / 10000 = 3).
* There are no I$ misses in the code with 2 jump instructions, except for the first iteration. This dramatically decreases the number of cycles.

1. Extend Figure 5 to analyse in detail how each 64-bit chunk is written in the I$.

Solution not provided for this exercise.

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

Solution not provided for this exercise.

You can find a useful study in RVfpga v2.2 (provided at: <https://university.imgtec.com/rvfpga-download-page-en/>), Lab 19, where a 4-way I$ is used in SweRV EH1.

1. Analyse the logic that checks the correctness of the parity information.

Solution not provided for this exercise.