THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2 Lab 20
ICCM, DCCM, and Benchmarking

1. Introduction

In this lab, we analyse the scratchpad memories (ICCM and DCCM) available in the VeeR EL2 processor, and then we provide several benchmarking examples and exercises to demonstrate some of the concepts from previous labs.

Recall that the RVfpgaEL2 System includes two scratchpad memories (also called closely-coupled memories: CCM):one for data (DCCM) and one for instructions (ICCM).

> **NOTE:** Before starting to work on this lab, we recommend reading Sections 1 and 3 of the paper by Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau. "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems". ACM Trans. Design Autom. Electr. Syst. 5(3): 682-704 (2000) (available at: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.2430&rep=rep1&type=pdf). This paper presents a good introduction to the use of scratchpad memories in embedded processors.

The next figures show an illustration of the address space occupied by the Instruction Memory (Figure 1) and by the Data Memory (Figure 2) available in the RVfpgaEL2 System.
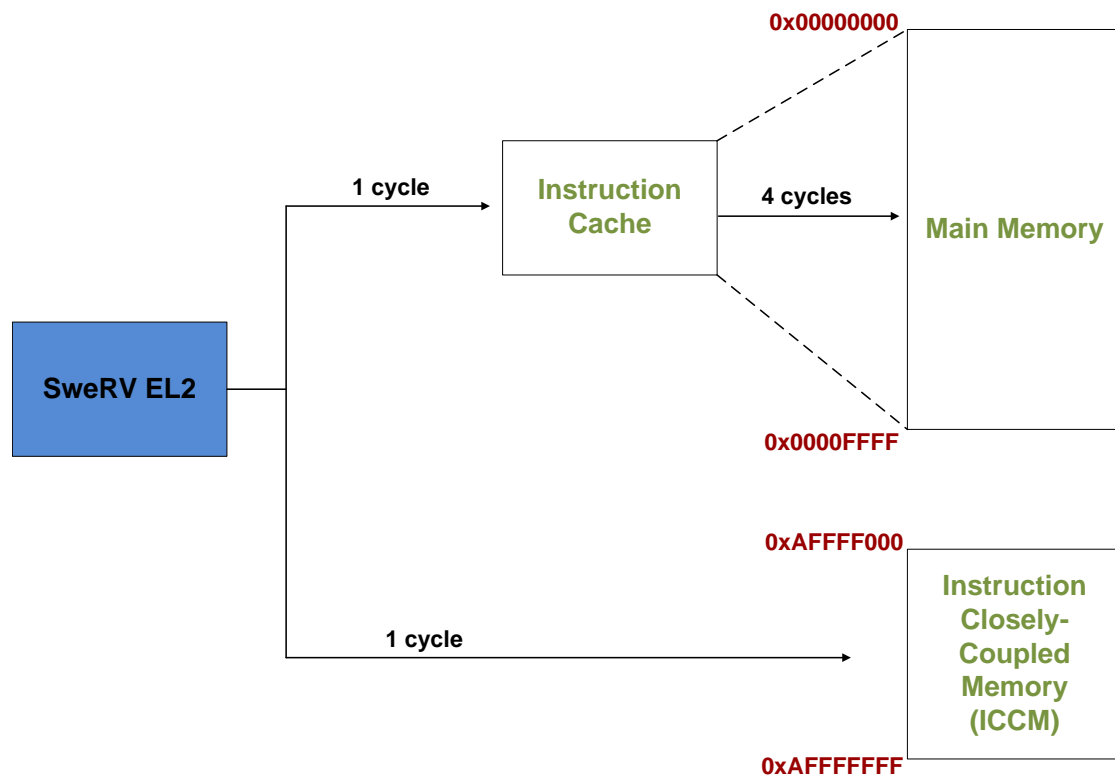

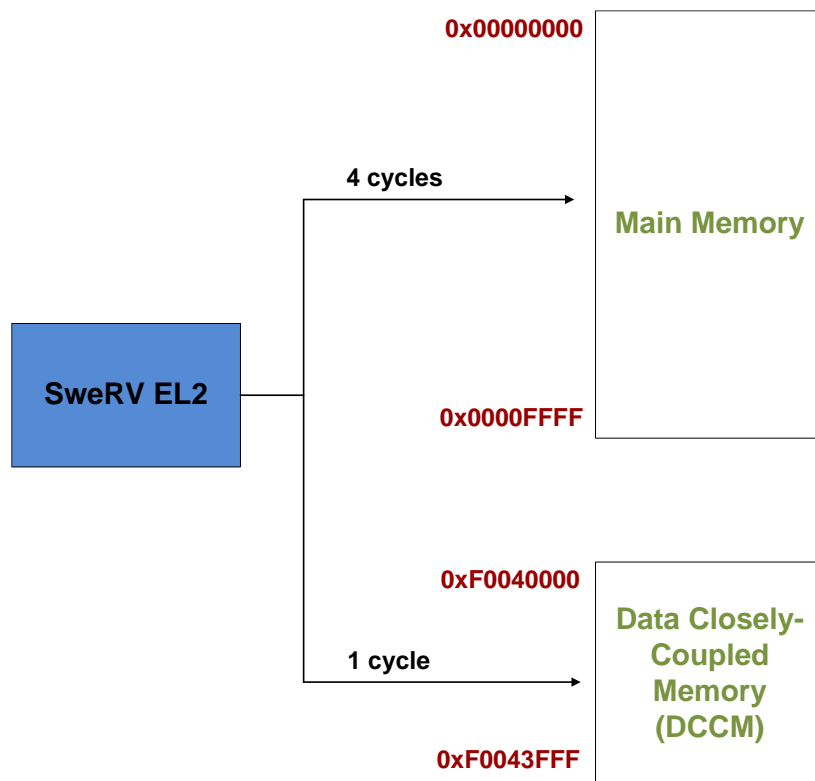**Figure 1. Instruction Memory address space: I$, ICCM, and Main Memory**

Figure 2. Data Memory address space: DCCM and DDR External Memory

In this lab, we focus on the configuration and operation of the DCCM and ICCM (Sections 2.A and 2.B, respectively) and then introduce several benchmarking examples and exercises (Section 3) where we use both ad hoc toy programs that illustrate specific situations and real applications.

2. DCCM and ICCM

In this section, we analyse the Data Closely-Coupled Memory (DCCM) and the Instruction Closely-Coupled Memory (ICCM) available in the RVfpgaEL2 System. We first describe how these two structures can be configured (Section 3.A) and then we illustrate how an access to the DCCM is performed (Section 3.B).

A.   DCCM and ICCM configuration in the RVfpgaEL2 System

The RVfpgaEL2 System's DCCM and ICCM (as well as the whole RVfpgaEL2 System) are highly configurable as explained in detail in Lab 11. The default RVfpgaEL2 System has the following parameters for these two structures (see file *[RVfpgaBooleanPath]/src/VeeRwolf/VeeR_EL2CoreComplex/include/el2_param.vh*):

DCCM:
```
DCCM_BANK_BITS          : 7'h02          ,
DCCM_BITS               : 9'h00E         ,
DCCM_BYTE_WIDTH         : 7'h04          ,
DCCM_DATA_WIDTH         : 10'h020         ,
DCCM_ECC_WIDTH          : 7'h07          ,
```

```
DCCM_ENABLE              : 5'h01           ,
DCCM_FDATA_WIDTH         : 10'h027         ,
DCCM_INDEX_BITS          : 8'h0A           ,
DCCM_NUM_BANKS           : 9'h004          ,
DCCM_REGION              : 8'h0F           ,
DCCM_SADR                : 36'h0F0040000   ,
DCCM_SIZE                : 14'h0010        ,
DCCM_WIDTH_BITS          : 6'h02           ,


ICCM:
ICCM_BANK_BITS           : 7'h02           ,
ICCM_BANK_HI             : 9'h003          ,
ICCM_BANK_INDEX_LO       : 9'h004          ,
ICCM_BITS                : 9'h00C          ,
ICCM_ENABLE              : 5'h01           ,
ICCM_ICACHE              : 5'h01           ,
ICCM_INDEX_BITS          : 8'h08           ,
ICCM_NUM_BANKS           : 9'h004          ,
ICCM_ONLY                : 5'h00           ,
ICCM_REGION              : 8'h0A           ,
ICCM_SADR                : 36'h0AFFFF000   ,
ICCM_SIZE                : 14'h0004        ,
```

Note that both the DCCM and the ICCM are enabled in our baseline system (DCCM_ENABLE = ICCM_ENABLE = 1). Table 1 summarizes the ICCM and DCCM configurations in the RVfpgaEL2 System.

Table 1. DCCM and ICCM configurations

Characteristic	Value
DCCM	
Enable	1
Address space	0xF0040000 – 0xF0043FFF
Size	16 KiB
ICCM	
Enable	1
Address space	0xAFFFF000 – 0xAFFFFFFF
Size	4 KiB

Figure 3 shows a block diagram of RVfpgaEL2's DCCM configuration. The input signals to the DCCM and the output signals from the DDCM are provided from/to the Load Store Unit (LSU), as explained in Lab 13 (see Figures 6 and 13 in Lab 13).
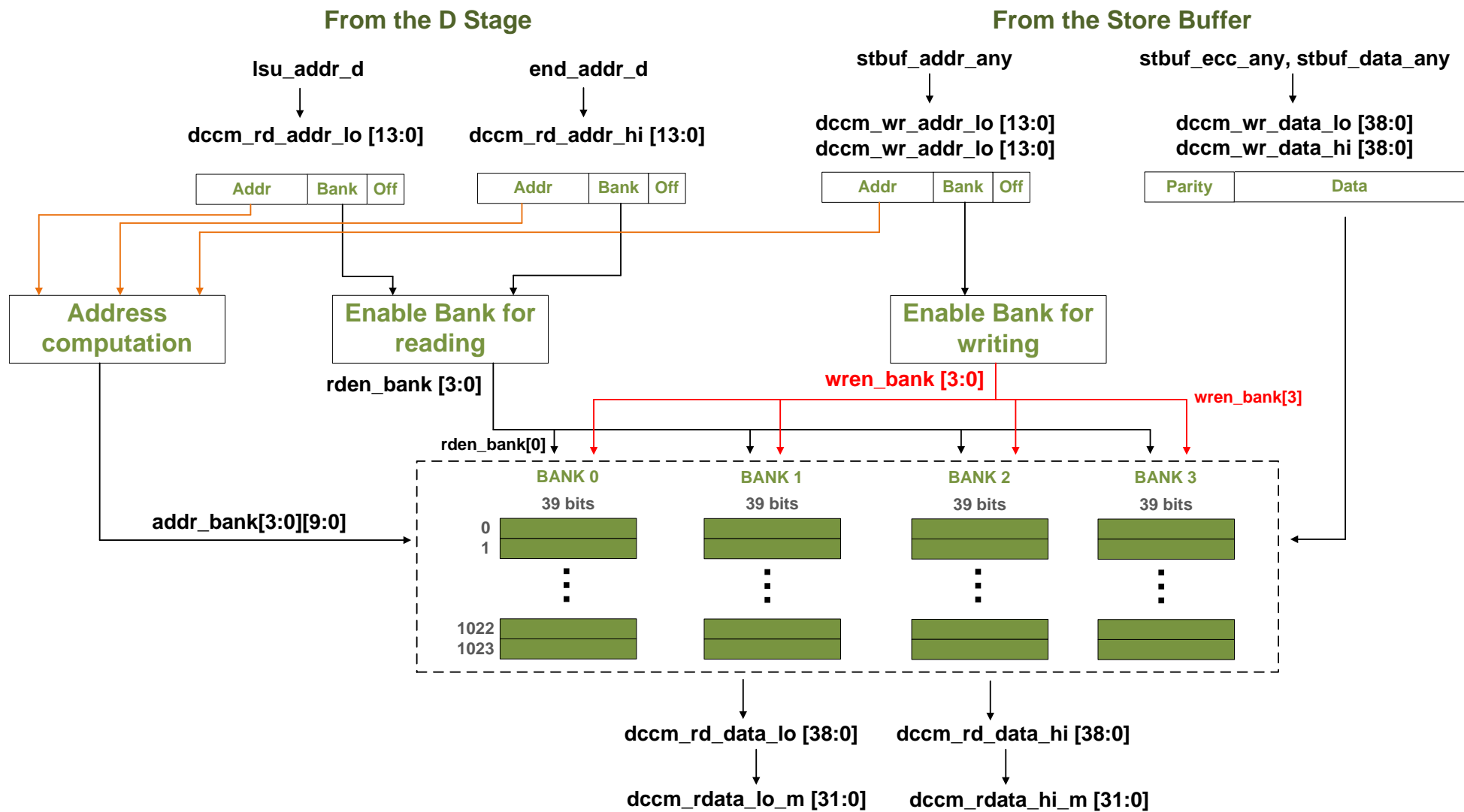
Figure 3. DCCM internal design

The RVfpgaEL2 System's DCCM is implemented in module `el2_lsu_dccm_mem`, included in file
[RVfpgaBooleanPath]/src/VeeRwolf/VeeR_EL2CoreComplex/lsu/el2_lsu_dccm_mem.sv. As shown in Figure 3, the DCCM is divided into 4 banks. Two read addresses are provided for supporting unaligned accesses: `dccm_rd_addr_lo[13:0] = lsu_addr_d` and `dccm_rd_addr_hi[13:0] = end_addr_d`. These addresses are logically divided into 3 fields:
- **Bank**: Bank selected.
- **Addr**: Address of the 32-bit word read within the bank.
- **Off**: Byte read within the 32-bit word.

As it can also be seen in Figure 3, the write address is provided in signals `dccm_wr_addr_lo[13:0]` and `dccm_wr_addr_hi[13:0]`. The write address is divided as the read addresses (see the previous item). Based on the 2-bit Bank field of these addresses (plus other signals not specified in the figure that you will analyse in a task below), 4 read/write enable bits are obtained in `rden_bank[3:0]` and `wren_bank[3:0]`, respectively. Each bit determines if the corresponding bank must be enabled or disabled for reading and writing.

Based on the Addr field of these addresses (and other signals not specified in the figure that you will analyse in a task below), four addresses are obtained in `addr_bank[3:0]`, one per bank.

Each of the 4 banks can be accessed independently. Thus, for example, it would be possible to perform one read and one write in the same cycle, as long as the two accesses are to different banks.

B.   Accessing the DCCM

Similar to the I$, the ICCM and the DCCM have a low access latency – that is, they allow data to be read or written in a single cycle. However, as opposed to the I$, the ICCM and DCCM are controlled by software.

In this section we illustrate and describe an access to the DCCM. We use the DCCM internal design shown in Figure 3 as a reference and execute a program similar to one already used in Lab 19. This program, shown in Figure 4, is provided in folder *[RVfpgaBooleanPath]/Labs/Lab20/LW-SW_Instruction_DCCM/.* It traverses a 250-element array, reading each element (`lw` instruction, highlighted in red), adding one to it and storing the element (`sw` instruction, highlighted in red) back to the same array element. The loop contains 20 `nop` instructions to isolate the iterations from each other. The array is initialized before accessing it (the initialization loop is not shown in Figure 4, but you can see the array initialization in the Catapult project).

```
// Access array
la t4, D
li t5, 50
li t0, 1000
la t6, D
add t6, t6, t0
li t5, 1
```

```
REPEAT_Access:
    lw t3, (t4)
    add t3, t3, t5
    sw t3, (t4)
    add t4, t4, 4
    INSERT_NOPS_10
    INSERT_NOPS_10
    bne  t4, t6, REPEAT_Access     # Repeat the loop
```

Figure 4. Example program

Open the project, build it, and open the disassembly file. Notice that the `lw` instruction (0x000eae03) and the `sw` instruction (0x01cea023) are at addresses 0x00000464 and 0x0000046c, respectively.

```
0x00000464:        000eae03                lw    t3,0(t4)
...
0x0000046c:        01cea023                sw    t3,0(t4)
```

Figure 5 shows the simulation of a random iteration of the loop from Figure 4. The figure includes some of the signals shown in Figure 3 as well as some of the LSU core signals that we described in Lab 13.

Figure 5. Simulation of a random iteration of the program from Figure 4

TASK: Replicate the simulation on your own computer.

Memory reads and writes using the DCCM occur as follows:

o **Cycle 1:** The `lw` instruction is decoded: `dec_i0_instr_d` = 0x000eae03. Also, the address is generated in the D Stage, as described in Lab 13, and provided to the DCCM:

  ▪ `lsu_addr_d[31:0]` = 0xF0040030 → `dccm_rd_addr_lo[13:0]` = 0x0030
  ▪ `end_addr_d[13:0]` = 0x0033   → `dccm_rd_addr_hi[13:0]` = 0x0033

As a result of the address check, reading the DCCM is enabled: `dccm_rden` = 1. This signal is provided to the DCCM and, along with the 2-bit *Bank* field of the address, determines the bank that must be read. In this case, only the first bank needs to be read:

`rden_bank = 0x1`.

- o **Cycle 2:** The read data is obtained from the DCCM and provided to the core. Given that it is an aligned access, the two read signals are equal and only `dccm_rd_data_lo` is effectively used by the core:
  - `dccm_rd_data_lo` = 0x**41**0000000C
  - `dccm_rd_data_hi` = 0x**41**0000000C
  - `dccm_rdata_lo_m` = 0x0000000C
  - `dccm_rdata_hi_m` = 0x0000000C

- o **Cycle 3:** The `sw` instruction is decoded: `dec_i0_instr_d` = 0x01CEA023.

- o **Cycle 6:** After adding 1 (the immediate) to the read value (0x0000000C + 1 = 0x0000000D), the data and address are provided to the DCCM, and writing of the correct bank is enabled using the following signals:
  - `dccm_wren` = 1
  - `wren_bank` = 0x01
  - `dccm_wr_addr_lo` = 0x0030
  - `dccm_wr_data_lo` = 0x**02**0000000D

TASK: Explain how signals `rden_bank`, `wren_bank`, and `addr_bank` are obtained in module **lsu_dccm_mem**.

TASK: Simulate an unaligned read to the DCCM and analyse how it is handled inside the DCCM. You can use the program used above (*[RVfpgaBooleanPath]/Labs/Lab20/LW-SW_Instruction_DCCM/*) and simply substitute the load instruction as follows:
        `lw t3, (t4)` → `lw t3, 1(t4)`

TASK: Simulate a DCCM bank conflict by modifying the program from Figure 4 (*[RVfpgaBooleanPath]/Labs/Lab20/LW-SW_Instruction_DCCM/*).

   1st modification: Remove the `nop` instructions, regenerate the simulation, and analyse the `lw` and the `sw` in a random iteration of the loop.

   2nd modification: Modify the immediate of the `sw` instruction for making the `lw` and `sw` try to access the same bank in the same cycle:
        `sw t3, (t4)` → `sw t3, 8(t4)`

3. Benchmarking

To benchmark a processor, a program or set of programs are run and the processor performance is measured. We compare processors by running the same benchmarks (i.e., sets of programs) on those processors. We introduce two common benchmarks: **CoreMark** and **Dhrystone**. These benchmarks are in folder *[RVfpgaBooleanPath]/Labs/Lab20/RealBenchmarks*.

Folder *[RVfpgaBooleanPath]/Labs/Lab20/RealBenchmarks/CoreMark* contains a Catapult project of the CoreMark benchmark targeted to the RVfpgaEL2 System. We have adapted CoreMark to the RVfpgaEL2 System using the sources provided by CHIPS Alliance at https://github.com/chipsalliance/Cores-VeeR-EL2. For any benchmark, we use the hardware counters (HW Counters) to measure various processor events, such as numbers of instructions executed and number of processor cycles, as explained in Lab 11. In addition to modifying the benchmark to use the RISC-V HW Counters, we have added some support for using the DCCM/ICCM and for using compiler optimizations.

In the next section, we show how to run CoreMark on the Boolean board under various scenarios.

A.   Variation 1: No compiler optimizations or DCCM/ICCM

First, we show how to execute the CoreMark benchmark under the processor conditions used in previous labs: debug mode and no use of the DCCM/ICCM. To do so, follow the next steps:

- Open the *CoreMark* project in Catapult.

- Open file *src/Test.c* (see Figure 6), which includes the *main* function of our program:
  - The *main* function first configures the HW Counters for measuring four events: number of cycles, instructions executed, I-bus transactions (instructions) and D-bus transactions (ld/sw instructions).
  - It then configures the different features of the VeeR EL2 processor, using two assembly instructions (li and csrrs). In this case, all features are left to their default values.
  - The program then invokes function main_cmark(), which implements the CoreMark benchmark itself, which is implemented in file *src/cmark.c*.
  - It finally prints the four events using function printfNexys().

```
int main(void)
{

  /* Initialize Uart */
  config_uart();

  pspMachinePerfMonitorEnableAll();

  pspMachinePerfCounterSet(D_PSP_COUNTER0, D_CYCLES_CLOCKS_ACTIVE);
  pspMachinePerfCounterSet(D_PSP_COUNTER1, D_INSTR_COMMITTED_ALL);
  pspMachinePerfCounterSet(D_PSP_COUNTER2, D_D_BUD_TRANSACTIONS_LD_ST);
  pspMachinePerfCounterSet(D_PSP_COUNTER3, D_I_BUS_TRANSACTIONS_INSTR);

  /* Modify core features as desired */
  __asm("li t2, 0x000");
  __asm("csrrs t1, 0x7F9, t2");

  main_cmark();

  ee_printf("Cycles = %d\n", cyc_end-cyc_beg);
  ee_printf("Instructions = %d\n", instr_end-instr_beg);
  ee_printf("Data Bus Transactions = %d\n", LdSt_end-LdSt_beg);
  ee_printf("Inst Bus Transactions = %d\n", Inst_end-Inst_beg);


  while(1);
}
```

Figure 6. File *src/Test.c* in CoreMark Catapult project

- Briefly analyse the functions from the CoreMark benchmark implemented in file *src/cmark.c*. Note that the HW Counters are started and stopped inside the `main_cmark()` function and that the benchmark itself is executed in between.

```
cyc_beg = pspMachinePerfCounterGet(D_PSP_COUNTER0);
instr_beg = pspMachinePerfCounterGet(D_PSP_COUNTER1);
LdSt_beg = pspMachinePerfCounterGet(D_PSP_COUNTER2);
Inst_beg = pspMachinePerfCounterGet(D_PSP_COUNTER3);

#if (MULTITHREAD>1)
        if (default_num_contexts>MULTITHREAD) {
                default_num_contexts=MULTITHREAD;
        }
        for (i=0 ; i<default_num_contexts; i++) {
                results[i].iterations=results[0].iterations;
                results[i].execs=results[0].execs;
                core_start_parallel(&results[i]);
        }
        for (i=0 ; i<default_num_contexts; i++) {
                core_stop_parallel(&results[i]);
        }
#else
        iterate(&results[0]);
#endif

cyc_end = pspMachinePerfCounterGet(D_PSP_COUNTER0);
instr_end = pspMachinePerfCounterGet(D_PSP_COUNTER1);
LdSt_end = pspMachinePerfCounterGet(D_PSP_COUNTER2);
Inst_end = pspMachinePerfCounterGet(D_PSP_COUNTER3);
```
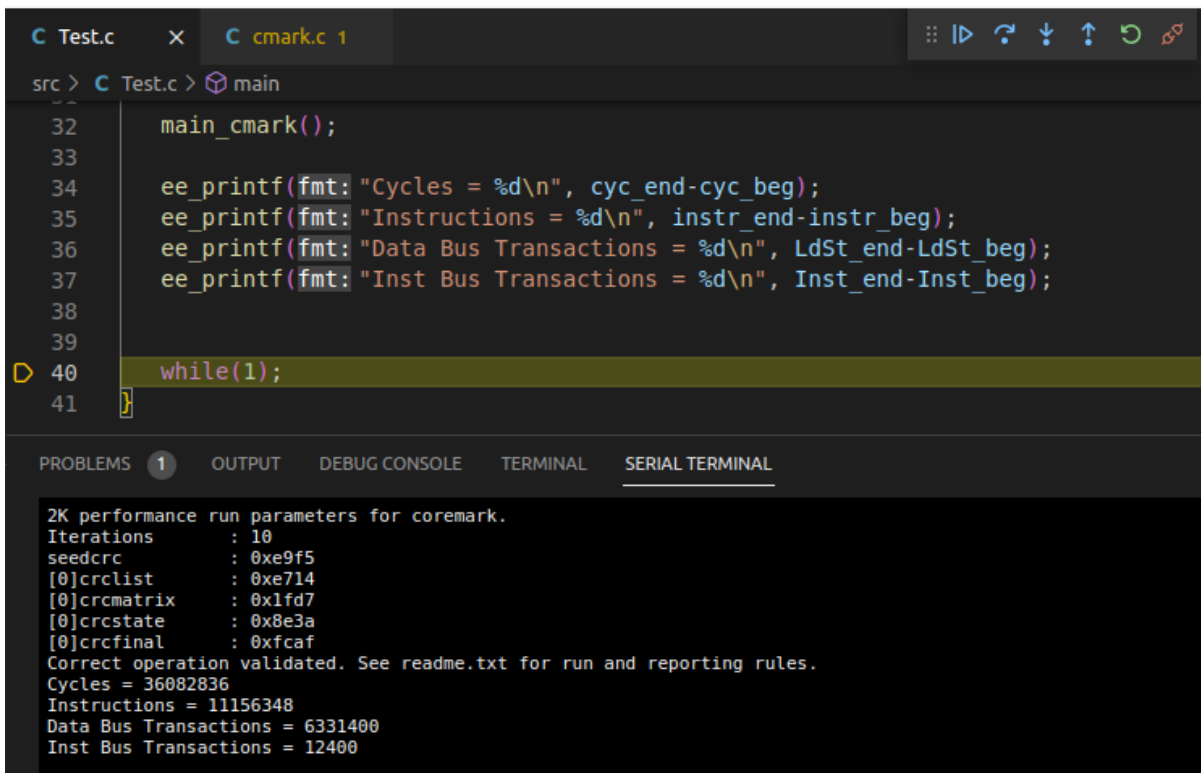
Figure 7. File *src/cmark.c* in CoreMark Catapult project

- Run the program on the board and open the serial monitor (see Figure 8). CoreMark runs multiple iterations of a loop (you can easily modify the number of iterations by means of a parameter called `ITERATIONS` and defined in file *src/cmark.c*). The execution took ~36 million cycles and approximately ~11 million instructions were executed, resulting in an IPC (instructions per cycle) ≈ 0.3. This performance is really poor: recall that the ideal IPC in the VeeR EL2 processor is 1.

Figure 8. Execution results of the CoreMark benchmark

B. Variation 2: Using the DCCM

Now we will use the DCCM for the *.bss* section so that accesses to these data use the DCCM instead of Main Memory. As we will see, this change increases performance, as expected.

For that purpose, open file *[RVfpgaBooleanPath]/Labs/Lab20/RealBenchmarks/CoreMark/common_tuned/Ldscript.ld* and modify it as shown inFigure 9 so that the `.bss` data will be accessed in the fast DCCM instead of the slow Main Memory. Then recompile the application.
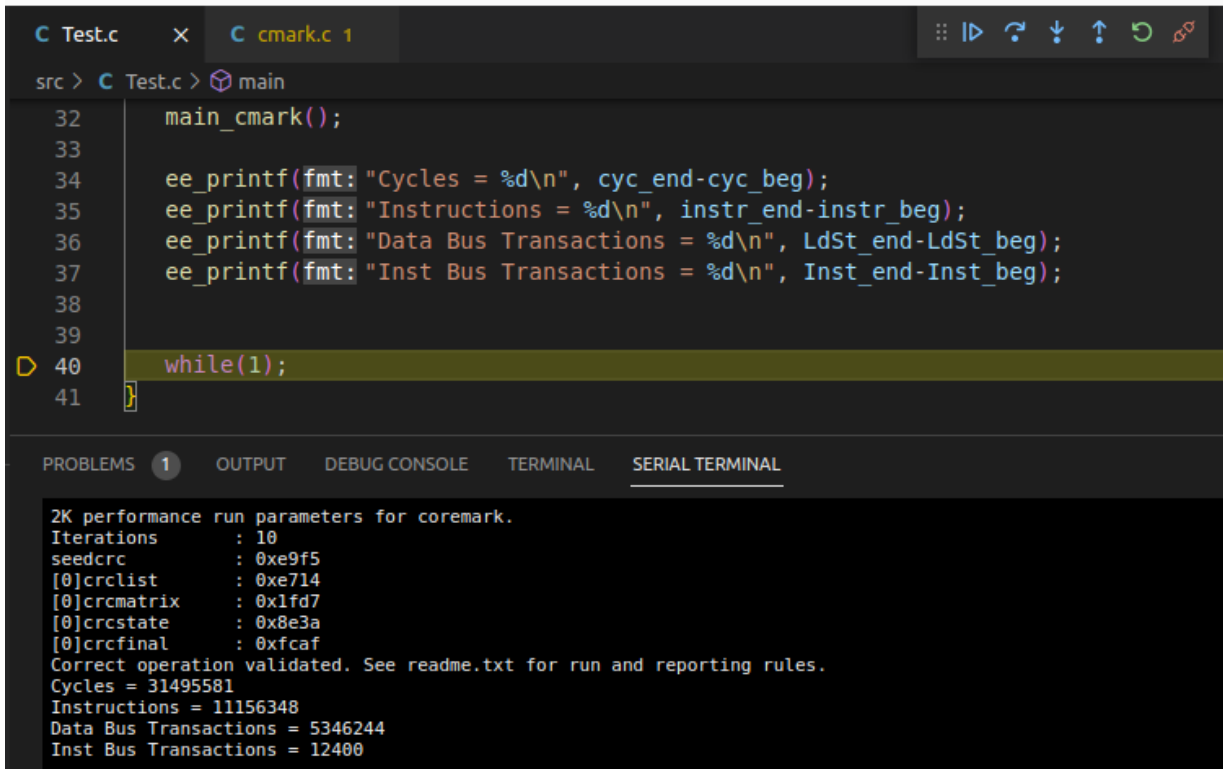
Figure 9. Modify the *Ldscript.ld*

Run the program on the board and open the serial monitor (see

Figure 10). In this case, the number of cycles has decreased to ~31 million cycles.

Figure 10. Execution results of the CoreMark benchmark using DCCM

C.   Variation: Using the DCCM and compiler optimizations

Now we add another way to improve performance: compiler optimizations. As in the previous section, we use the DCCM to store the *.bss* section – but now we also enable compiler optimizations. Up until this point, we have executed programs in debug mode with no compiler optimizations.

To enable compiler optimizations, open file *[RVfpgaBooleanPath]/Labs/Lab20/RealBenchmarks/CoreMark/common_tuned/Common.cmake* and modify it as shown in Figure 11 so that the compiler optimizes with the –O3 level. Then recompile the application.
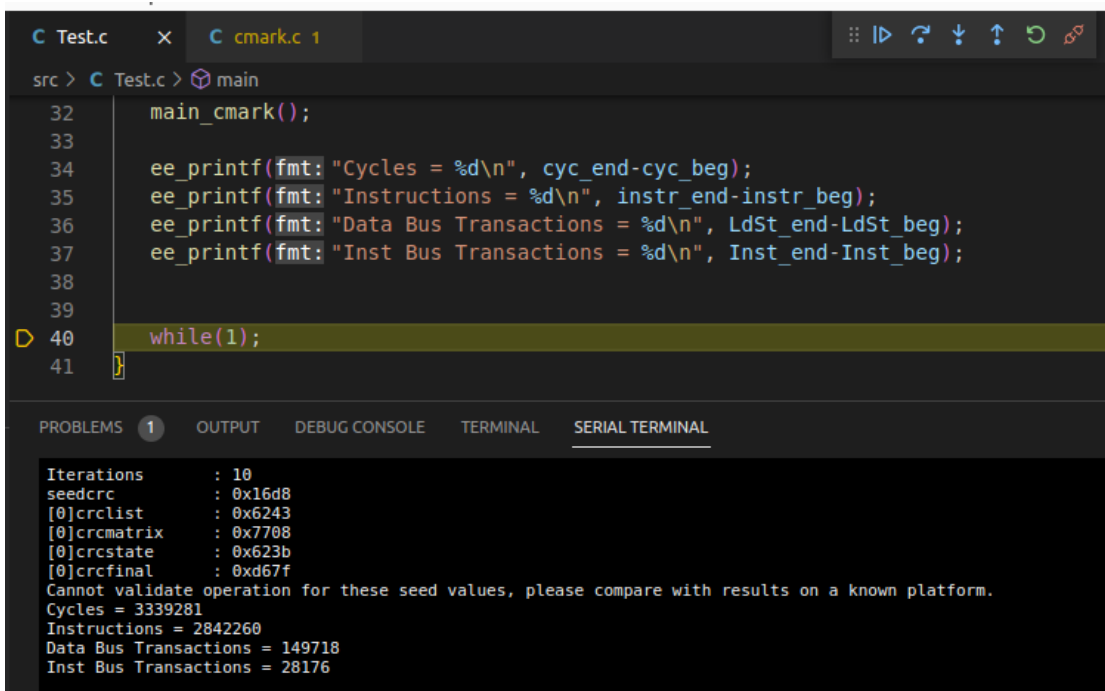
```
# Set all compiler and link options
if(CMAKE_BUILD_TYPE MATCHES "Debug")
    target_compile_options(${TARGET_NAME} PRIVATE -O3)
else()
    target_compile_options(${TARGET_NAME} PRIVATE -O3 -g)
endif()
```

Figure 11. File *Common.cmake*, option *build_flags*

Run the program and open the serial monitor. See Figure 12. The number of cycles has decreased to around 3 million, and the number of instructions is also close to 3 million. The IPC is now ≈ 1.

Figure 12. Execution results of CoreMark when using compiler optimizations

4. Exercises

1) Do the same analysis as was done for CoreMark but this time using the Dhrystone benchmark. A Catapult project that contains the Dhrystone benchmark is in: *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Dhrystone*.

2) Enable/disable various core features as described in Lab 11. Compare the performance results – that is, values of the HW Counters when executing the programs on these modified cores. Run all programs (CoreMark, Dhrystone) on these modified RVfpgaEL2 Systems on the Boolean Board. Variations include:
   a. Using different Branch Predictor configurations and implementations (such as always not-taken, GShare, and the bimodal predictor implemented in Lab 16).
   b. Using various I$/DCCM/ICCM configurations (such as different sizes or different I$ Replacement Policies).