



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 20

ICCM, DCCM, and Benchmarking

1. INTRODUCTION

In this lab, we analyse the scratchpad memories (ICCM and DCCM) available in the SweRV EH1 processor, and then we provide several benchmarking examples and exercises to demonstrate some of the concepts from Labs 11 to 20.

Recall from Figure 25 of the RVfpga Getting Started Guide (that we repeat below in Figure 1 for the sake of convenience), that the RVfpga System includes two scratchpad memories (highlighted in red in the figure): one for data, called Data Closely-Coupled Memory (DCCM), and one for instructions, called Instruction Closely-Coupled Memory (ICCM).

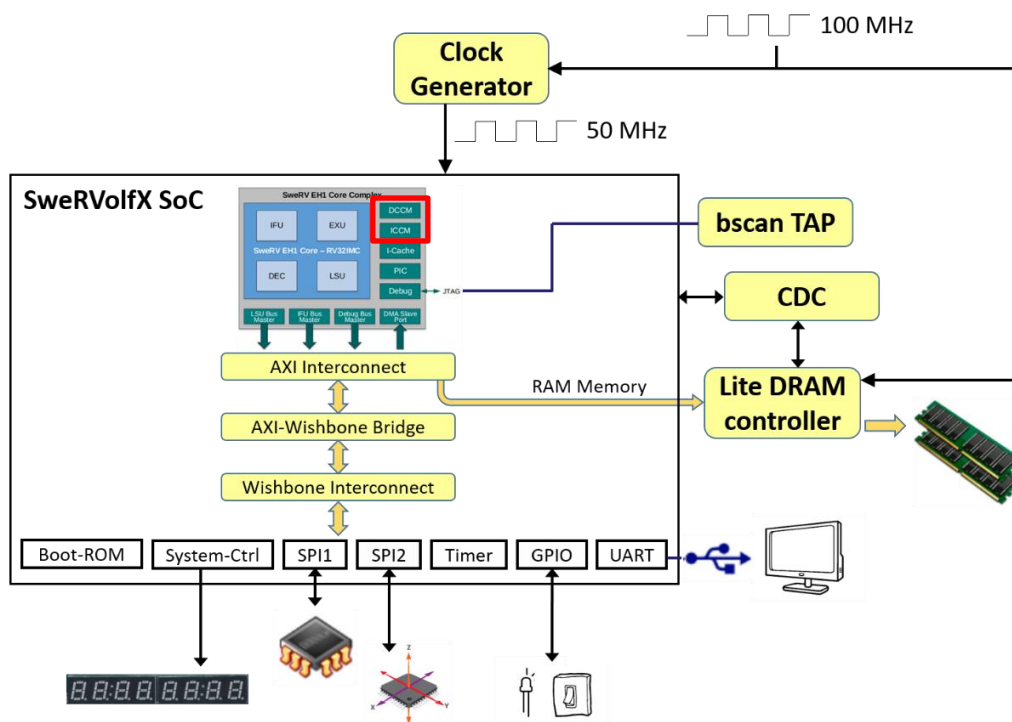
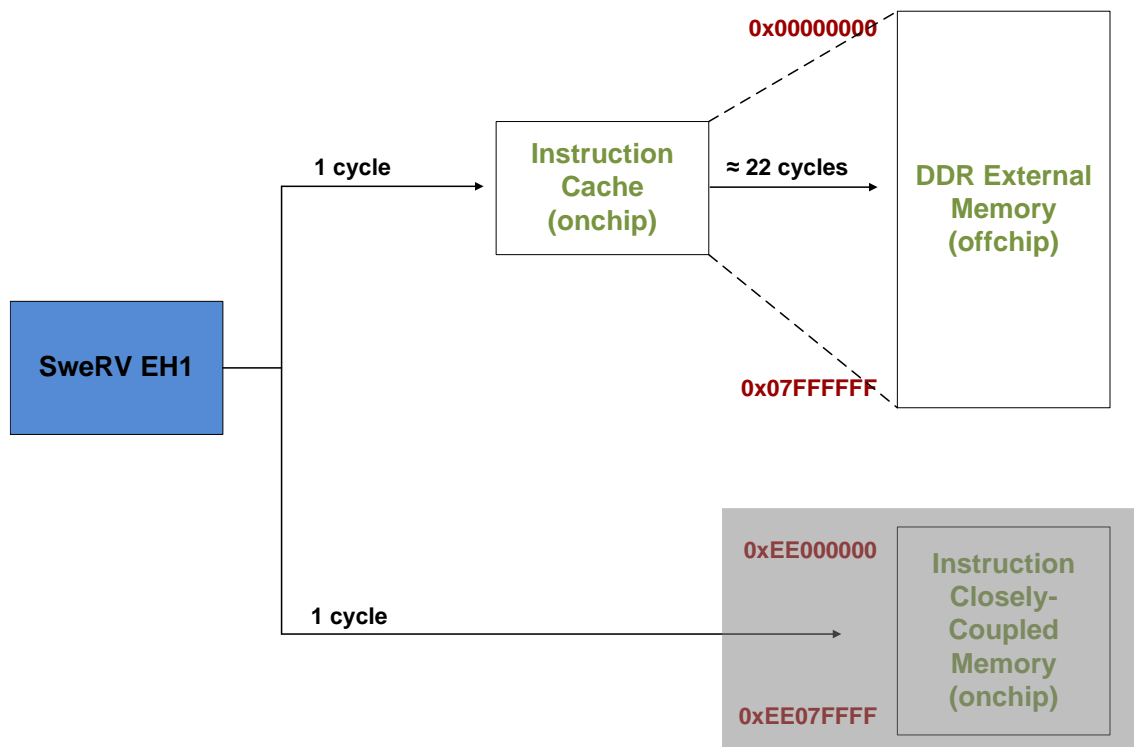


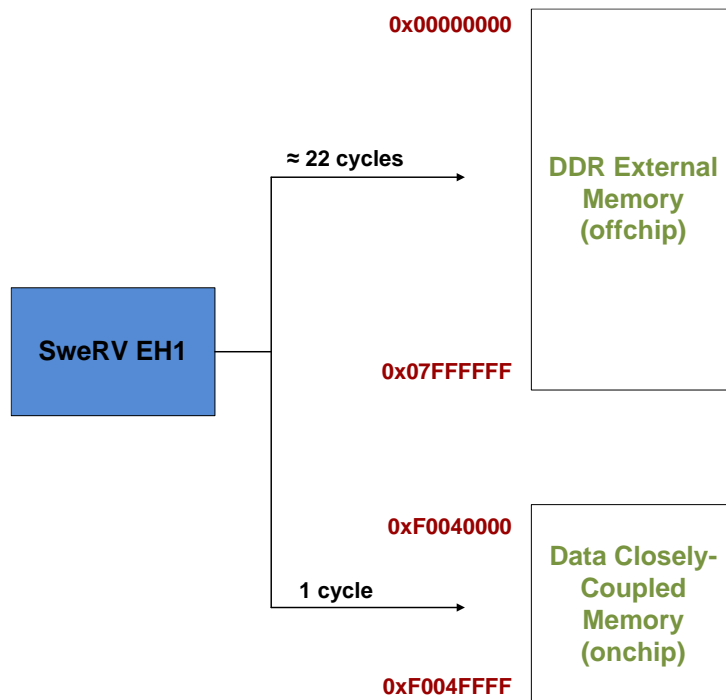
Figure 1. RVfpgaNexys System

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 Scratch-Pad memories in embedded processors.

The RVfpga System memory map was described in Section 4.B of the Getting Started Guide. The next figure complements that description with an illustration of the address space occupied by the Instruction Memory (Figure 2a) and by the Data Memory (Figure 2b) available in the RVfpga System.



(a) Address space of Instruction Memory, consisting of an instruction cache (I\$) and DDR External Memory. The ICCM is disabled in the default system.



(b) Address space of Data Memory, consisting of a DCCM and DDR External Memory.

Figure 2. RVfpga System address space for Instruction and Data Memories

In this lab, we focus on the configuration and operation of the Data/Instruction Closely-Coupled Memories (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. DATA/INSTRUCTION CLOSELY-COUPLED MEMORIES (DCCM AND ICCM)

In this section, we analyse the Data Closely-Coupled Memory (DCCM) and the Instruction Closely-Coupled Memory (ICCM) available in the RVfpga 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 RVfpga System

The RVfpga System's DCCM and ICCM are 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 parameters for these two structures:

DCCM:

```
`define RV_DCCM_EADR 32'hf004ffff
`define RV_DCCM_FDATA_WIDTH 39
`define RV_LSU_SB_BITS 16
`define RV_DCCM_SIZE 64
`define RV_DCCM_ECC_WIDTH 7
`define RV_DCCM_SADR 32'hf0040000
`define RV_DCCM_BYTE_WIDTH 4
`define RV_DCCM_NUM_BANKS 8
`define RV_DCCM_SIZE_64
`define RV_DCCM_NUM_BANKS_8
`define RV_DCCM_OFFSET 28'h40000
`define RV_DCCM_WIDTH_BITS 2
`define RV_DCCM_ENABLE 1
`define RV_DCCM_DATA_CELL ram_2048x39
`define RV_DCCM_RESERVED 'h1000
`define RV_DCCM_ROWS 2048
`define RV_DCCM_BANK_BITS 3
`define RV_DCCM_DATA_WIDTH 32
`define RV_DCCM_INDEX_BITS 11
`define RV_DCCM_BITS 16
`define RV_DCCM_REGION 4'hf
```

ICCM:

```
`define RV_ICCM_DATA_CELL ram_16384x39
`define RV_ICCM_BITS 19
`define RV_ICCM_ROWS 16384
`define RV_ICCM_INDEX_BITS 14
`define RV_ICCM_NUM_BANKS 8
`define RV_ICCM_NUM_BANKS_8
`define RV_ICCM_BANK_BITS 3
`define RV_ICCM_SIZE_512
`define RV_ICCM_RESERVED 'h1000
```

```

`define RV_ICCM_SIZE 512
`define RV_ICCM_REGION 4'he
`define RV_ICCM_OFFSET 10'he000000
`define RV_ICCM_SADR 32'hee000000
`define RV_ICCM_EADR 32'hee07ffff

```

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

DCCM:

```

localparam DCCM_BITS          = `RV_DCCM_BITS;
localparam DCCM_BANK_BITS     = `RV_DCCM_BANK_BITS;
localparam DCCM_NUM_BANKS     = `RV_DCCM_NUM_BANKS;
localparam DCCM_DATA_WIDTH    = `RV_DCCM_DATA_WIDTH;
localparam DCCM_FDATA_WIDTH   = `RV_DCCM_FDATA_WIDTH;
localparam DCCM_BYTE_WIDTH    = `RV_DCCM_BYTE_WIDTH;
localparam DCCM_ECC_WIDTH     = `RV_DCCM_ECC_WIDTH;

```

ICCM:

```

localparam ICCM_SIZE          = `RV_ICCM_SIZE;
localparam ICCM_BITS          = `RV_ICCM_BITS;
localparam ICCM_NUM_BANKS     = `RV_ICCM_NUM_BANKS;
localparam ICCM_BANK_BITS     = `RV_ICCM_BANK_BITS;
localparam ICCM_INDEX_BITS    = `RV_ICCM_INDEX_BITS;
localparam ICCM_BANK_HI       = 4 + (`RV_ICCM_BANK_BITS/4);

```

Note that, as shown in Figure 2, the DCCM is enabled in our baseline system ($RV_DCCM_ENABLE = 1$) but the ICCM is disabled (RV_ICCM_ENABLE not defined), so no ICCM is included in the SoC used in the previous labs.

Table 1 summarizes the ICCM and DCCM configurations in the RVfpga System.

Table 1. DCCM and ICCM Configurations

Characteristic	Value
DCCM	
Enable	1
Address space	0xF0040000 – 0xF004FFFF
Size	64 KiB
Number of banks	8
Bank size	2048x39 bits (7 bits for parity)
ICCM	
Enable	0

Figure 3 shows a block diagram of RVfpga's DCCM configuration. The input signals to the DCCM (lsu_addr_dc1 , end_addr_dc1 , $stbuf_addr_any$, $stbuf_ecc_any$ and $stbuf_data_any$) and the output signals from the DDCM ($dccm_data_lo_dc2$ and $dccm_data_hi_dc2$) are provided from/to the Load Store Unit (lsu), as explained in Lab 13 (see Figures 6 and 13 in Lab 13).

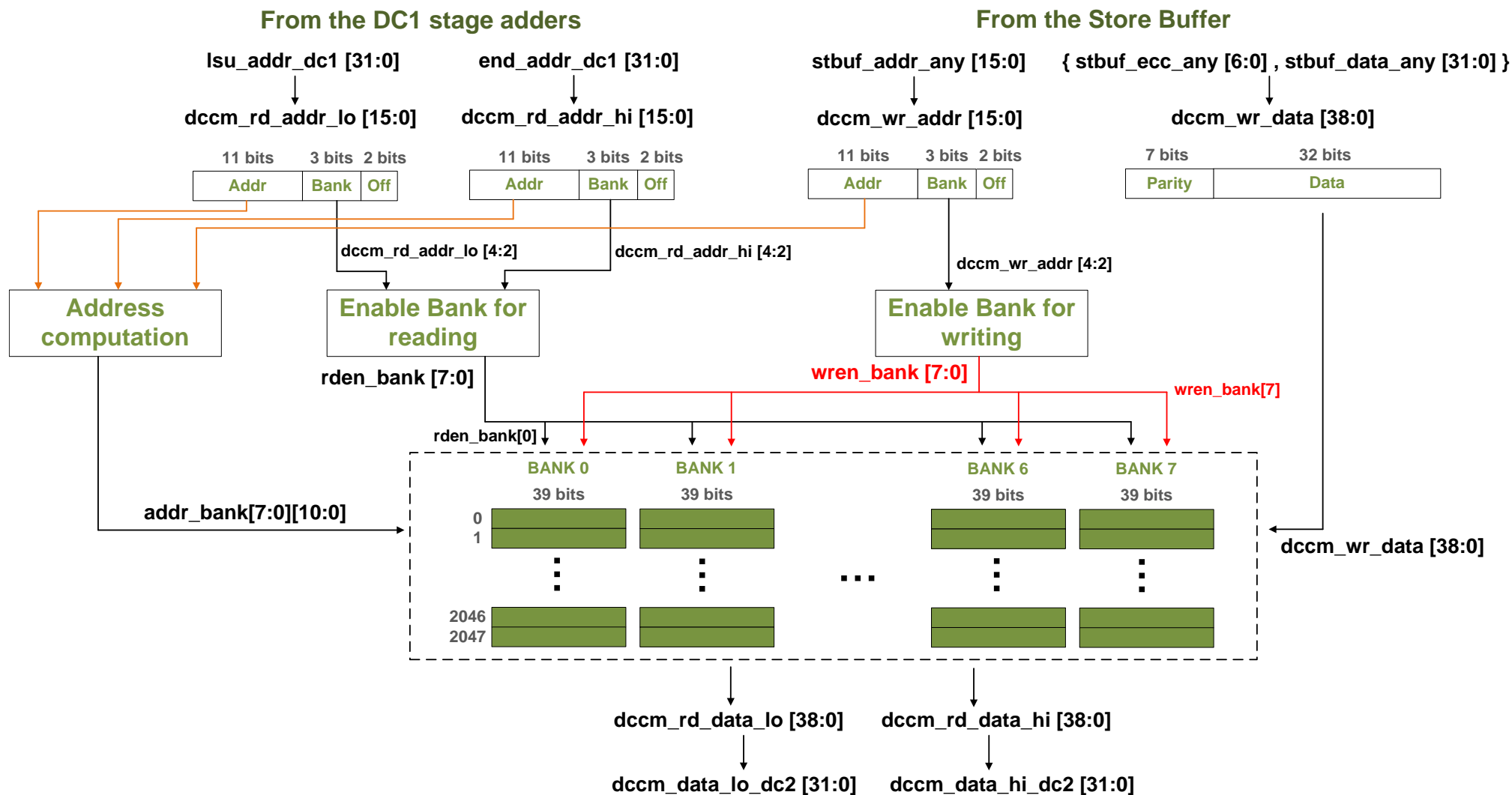


Figure 3. DCCM internal design.

The RVfpga System's DCCM is implemented in module **lsu_dccm_mem**, included in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/lsu/lsu_dccm_mem.sv*. As shown in Figure 3, the DCCM is divided into 8 banks. Two read addresses are provided for supporting unaligned accesses: `dccm_rd_addr_lo[15:0] = lsu_addr_dc1[15:0]` and `dccm_rd_addr_hi[15:0] = end_addr_dc1[15:0]`. 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.
- Note that 7 parity bits are added to each 32-bit word.

As also explained in Lab 13 and as it can be seen in Figure 3, one write address is provided in signal `dccm_wr_addr[15:0]` by the Store Buffer (see the appendix from Lab 13 for further descriptions of the Store Buffer operation). The write address is divided as the read addresses (see the previous item). Based on the 3-bit **Bank** field of these addresses (plus other signals not specified in the figure that you will analyse in a task below), 8 read/write enable bits are obtained in `rden_bank[7:0]` and `wren_bank[7:0]`, respectively. Each bit determines if the corresponding bank must be enabled or disabled for reading and writing.

Based on the 11-bit **Addr** field of these addresses (and other signals not specified in the figure that you will analyse in a task below), eight 11-bit addresses are obtained in `addr_bank[7:0][10:0]`, one 11-bit address per bank.

Each of the 8 banks can be accessed independently, as you will analyse in a task below. Thus, for example, in the most extreme situation, it would be possible to perform two reads and one write in the same cycle, as long as the three accesses are to three different banks:

- In an unaligned read, banks *j* and *k* can be read in the same cycle by providing the 11-bit addresses in signals `addr_bank[j]` (which is obtained from the 11-bit **Addr** field of signal `dccm_rd_addr_lo`) and `addr_bank[k]` (which is obtained from the 11-bit **Addr** field of signal `dccm_rd_addr_hi`), and by setting the corresponding enable signals: `rden_bank[j] = rden_bank[k] = 1`.
- At the same time, it is also possible to write to bank *i*, by providing the 11-bit address in signal `addr_bank[i]` (obtained from the 11-bit **Addr** field of signal `dccm_wr_addr`), and by setting the corresponding enable signal: `wren_bank[i] = 1`.

TASK: Using the instructions provided in Lab 1, implement a new RVfpga System that includes a 64 KiB ICCM.

Remember that the ICCM is disabled in our default system. Thus, as explained in Section 2.A of the SweRVref document, in order to enable the ICCM you must include the following line in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh*:

```
`define RV_ICCM_ENABLE 1
```

In addition, the parameters provided in the default RVfpga System are for a 512 KiB ICCM. Thus, in order to implement a 64 KiB ICCM, you must modify the following lines of the

same file (file *common_defines.vh*):

```
RV_ICCM_DATA_CELL ram_16384x39 → RV_ICCM_DATA_CELL ram_2048x39
RV_ICCM_BITS 19 → RV_ICCM_BITS 16
RV_ICCM_ROWS 16384 → RV_ICCM_ROWS 2048
RV_ICCM_INDEX_BITS 14 → RV_ICCM_INDEX_BITS 11
RV_ICCM_SIZE_512 → RV_ICCM_SIZE_64
RV_ICCM_SIZE 512 → RV_ICCM_SIZE 64
RV_ICCM_EADR 32'hee07ffff → RV_ICCM_EADR 32'hee00ffff
```

As explained in Section 2.A of the SweRVref document, instead of manually modifying file *common_defines.vh*, you can also modify the configuration of the SweRV EH1 processor using the *swerv.config* script.

TASK: Draw a figure similar to Figure 3 for the ICCM implemented in the previous task.

B. Accessing the DCCM

Similar to the I\$ that we analysed in Lab 19, the ICCM and the DCCM have a low access latency – that is, that allows data to be read or written in a single cycle (see Figure 2). 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 *[RVfpgaPath]/RVfpga/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 PlatformIO 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 in PlatformIO, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis*). Notice that the *lw*

instruction (0x000eae03) and the `sw` instruction (0x01cea023) are placed at addresses 0x000001c0 and 0x000001c8, respectively.

0x000001c0:	000eae03	lw	t3,0(t4)
...			
0x000001c8:	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.

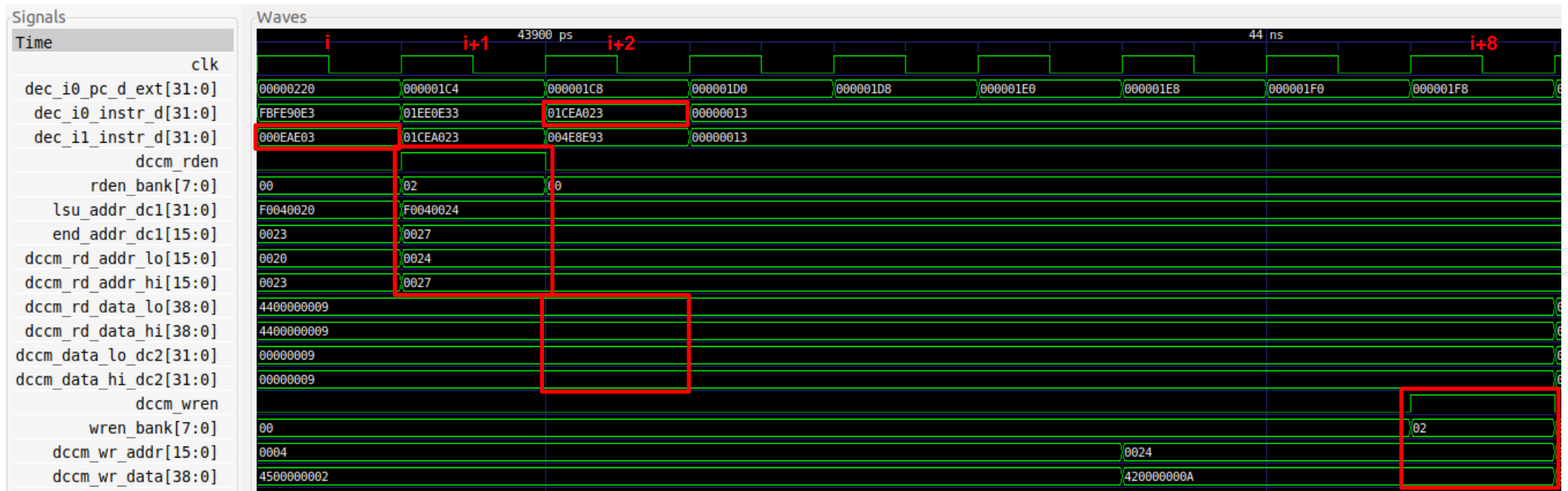



Figure 5. Simulation of a random iteration of the program from Figure 4

TASK: Replicate the simulation from Figure 5 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/Lab20/LW-SW_Instruction_DCCM*.
- 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 *scriptLoadStore.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM*) for opening the same signals as the ones shown in Figure 5. For that purpose, in GTKWave, click on *File → Read Tcl Script File* and select the *scriptLoadStore.tcl* file.
- Click on *Zoom In* () several times and analyse the region starting at 43900 ps.

Memory reads and writes using the DCCM occur as follows:

- **Cycle i:** The *lw* instruction is decoded in Way 1: *dec_i1_instr_d* = 0x000eae03.
- **Cycle i+1:** The address is generated in the DC1 stage, as described in Lab 13 (see Figure 6 of that lab), and provided to the DCCM:
 - *lsu_addr_dc1*[31:0] = 0xF0040024 → *dccm_rd_addr_lo*[15:0] = 0x0024
 - *end_addr_dc1*[15:0] = 0x0027 → *dccm_rd_addr_hi*[15:0] = 0x0027

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 3-bit *Bank* field of the address, determines the bank that must be read. In this case, only the second bank of the access needs to be read as the access is word-aligned: *rden_bank* = 0x02 (in binary 00000010).

- **Cycle i+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_data_lo_dc2* is effectively used by the core (again, this was explained in Lab 13):
 - *dccm_rd_data_lo* = 0x4400000009 → *dccm_data_lo_dc2* = 0x00000009
 - *dccm_rd_data_hi* = 0x4400000009 → *dccm_data_hi_dc2* = 0x00000009
- **Cycle i+8:** After adding 1 (the immediate) to the read value (0x00000009 + 1 = 0x0000000A) and traversing the Store Buffer, as explained in the appendix of Lab 13, 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* = 0x02 (in binary 00000010; i.e, the second bank)
 - *dccm_wr_addr* = 0x0024
 - *dccm_wr_data* = 0x420000000A

TASK: Explain how signals *rden_bank*, *wren_bank*, and *addr_bank* are obtained in lines 103, 104, and 105 of 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 (`[RVfpgaPath]/RVfpga/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 (`[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM`).

1st modification: Remove the 20 `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) is 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 `[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks`. We describe these benchmarks, along with the **Image Processing** program from Lab 5, next.

Folder `[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters` contains a PlatformIO project of the CoreMark benchmark targeted to the RVfpga System. We have adapted CoreMark to the RVfpga System using the sources provided by Chips Alliance at <https://github.com/chipsalliance/Cores-SweRV>. 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 Nexys A7 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_HwCounters` project in PlatformIO.
- Open file `src/Test.c` (see Figure 6), which includes the `main` function of our program:
 - o The `main` function first configures the HW Counters for measuring four events: number of cycles, I-bus transactions (instructions) and D-bus transactions (`ld/st` instructions). For this purpose, function `pspPerformanceCounterSet()` is

used.

- It then configures the different features of the SweRV EH1 processor, using two assembly instructions (`li` and `csrrs`) as explained in Section 2.C of the SweRVref document. In this case, all features are left to their default values.
- The program then executes a loop that is only exited when any of the switches on the board is inverted. The purpose of this loop is to allow the user to open the serial monitor before the benchmark executes and outputs its results.
- 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()`.

```

25 int main(void)
26 {
27     /* Initialize Uart */
28     uartInit();
29
30     pspEnableAllPerformanceMonitor(1);
31
32     pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
33     pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
34     pspPerformanceCounterSet(D_PSP_COUNTER2, E_D_BUD_TRANSACTIONS_LD_ST);
35     pspPerformanceCounterSet(D_PSP_COUNTER3, E_I_BUS_TRANSACTIONS_INSTR);
36
37     /* Modify core features as desired */
38     __asm("li t2, 0x000");
39     __asm("csrrs t1, 0x7F9, t2");
40
41     /* Invert Switch to execute CoreMark*/
42     int switches_value, switches_init;
43     WRITE_GPIO(GPIO_INOUT, 0xFFFF);
44     switches_init = (READ_GPIO(GPIO_SWs) >> 16);
45     switches_value = switches_init;
46     while (switches_value==switches_init) {
47         switches_value = (READ_GPIO(GPIO_SWs) >> 16);
48         printfNexys("Invert any Switch to execute CoreMark");
49     }
50
51     main_cmark();
52
53     printfNexys("Cycles = %d", cyc_end-cyc_beg);
54     printfNexys("Instructions = %d", instr_end-instr_beg);
55     printfNexys("Data Bus Transactions = %d", LdSt_end-LdSt_beg);
56     printfNexys("Inst Bus Transactions = %d", Inst_end-Inst_beg);
57
58     while(1);
59 }

```

Figure 6. File `src/Test.c` in CoreMark PlatformIO 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 (lines 1109-1112 and 1130-1133), and that the benchmark itself is executed inbetween (lines 1114-1128).

```

1104      /* perform actual benchmark */
1105      start_time();
1106
1107      __asm("__perf_start:");
1108
1109      cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
1110      instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
1111      LdSt_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
1112      Inst_beg = pspPerformanceCounterGet(D_PSP_COUNTER3);
1113
1114      #if (MULTITHREAD>1)
1115          if (default_num_contexts>MULTITHREAD) {
1116              default_num_contexts=MULTITHREAD;
1117          }
1118          for (i=0 ; i<default_num_contexts; i++) {
1119              results[i].iterations=results[0].iterations;
1120              results[i].execs=results[0].execs;
1121              core_start_parallel(&results[i]);
1122          }
1123          for (i=0 ; i<default_num_contexts; i++) {
1124              core_stop_parallel(&results[i]);
1125          }
1126      #else
1127          iterate(&results[0]);
1128      #endif
1129
1130      cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
1131      instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
1132      LdSt_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
1133      Inst_end = pspPerformanceCounterGet(D_PSP_COUNTER3);
1134
1135      __asm("__perf_end:");
1136
1137      stop_time();
1138      total_time=get_time();

```

Figure 7. File `src/cmark.c` in CoreMark PlatformIO project

- Run the program on the board. Then open the serial monitor as explained in Section 6.F of the GSG.

After opening the serial monitor, you will first see a repeating message that asks you to invert a switch in the board for executing the CoreMark benchmark (see the upper red box in Figure 8). Once you invert a switch, the benchmark executes and outputs the results, as shown in 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 number of iterations it completes per second is called the *CoreMark score* (CM). The number of iterations per MHz is *CM/MHz*. The benchmark provides the CM/MHz – also called Iterat/Sec/MHz (iterations/second/MHz) – which is 0.47. You can also view the values provided by the hardware counters, which were used to calculate the CM/MHz.

The execution took ~2 million cycles and approximately half million instructions were processed, resulting in an IPC (instructions per cycle) ≈ 0.25 ; specifically, $\frac{1}{2}$ million instructions / 2 million cycles ≈ 0.25 . This performance is really poor: recall that the ideal IPC in the SweRV EH1 processor is 2 because it is two-way superscalar. However,

performance is graded because of the large number of data reads/writes and the slow DDR External Memory. The number of data transactions through the bus is about 133,000. The number of instruction transactions through the bus is only 392 because most instruction accesses hit in the I\$. Recall that the RVfpga System does not have a D\$ (data cache).

```

58 while(1);
59
60
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
2K performance run parameters for coremark.

CoreMark Size      : 666
Total ticks        : 2099661
Total time (secs): 2099
Iterat/Sec/MHz     : 0.47
Iterations         : 1
Compiler version   : GCC8.3.0
Compiler flags     : -O2
Memory location    : STATIC
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0xe714

Correct operation validated. See readme.txt for run and reporting rules.

Cycles = 2099422
Instructions = 496678
Data Bus Transactions = 133628
Inst Bus Transactions = 392

```

Figure 8. Execution results of the CoreMark benchmark

B. Variation 2: Using the DCCM

Now we enable the DCCM in the RVfpga System so that most data accesses use the DCCM (instead of the external DDR memory). As we will see, this change increases performance, as expected. Follow the next steps to run CoreMark on a version of the RVfpga system that uses the DCCM:

- The default linker script that we have used so far in most labs is available at

`.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/link.lds`. However, in order to use the DCCM to store some data of the program, we make use of a specific linker script that is provided as part of the PlatformIO project that you are using and which is available at:

`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM.ld`. Open this file and inspect it. Figure 9 shows some parts of this file, which we describe briefly.

In the upper screenshot of Figure 9 defines one memory section for the DCCM (called `dccm`), which corresponds to the address space defined in Figure 2(b) for this memory:

`dccm (wxa!ri) : ORIGIN = 0xf0040000, LENGTH = 64K`

The remaining screenshots map several code sections to the DCCM memory: `.rodata`, `.data`, `.sdata`, `.bss` and `.stack`.

```

26  MEMORY
27  {
28      ram (wxa!ri) : ORIGIN = 0x00000000, LENGTH = 64M
29      ram2 (wxa!ri) : ORIGIN = 0x04000000, LENGTH = 64M
30      dccm (wxa!ri) : ORIGIN = 0xf0040000, LENGTH = 64K
31      ovl          : ORIGIN = 0xE0000000, LENGTH = 8k
32  }

```

```

72  .rodata :
73  {
74      *(.rodata)
75      *(.rodata .rodata.*)
76      *(.gnu.linkonce.r.*)
77      KEEP(*(COMRV_RODATA_SEC))
78      . = ALIGN(4);
79  } > dccm : dccm_load

```

```

104  .data :
105  {
106      *(.data .data.*)
107      *(.gnu.linkonce.d.*)
108      . += 10; /* fix for linker false error message */
109      . = ALIGN(8);
110  } > dccm : dccm_load

```

```

122  .sdata :
123  {
124      . = ALIGN(8);
125      __global_pointer$ = . + 0x800;
126      *(.sdata .sdata.*)
127      *(.gnu.linkonce.s.*)
128      . = ALIGN(8);
129      *(.srodata .srodata.*)
130      . = ALIGN(8);
131  } > dccm : dccm_load

```

```

142  .bss :
143  {
144      *(.sbss .sbss.* .gnu.linkonce.sb.*)
145      *(.scommon)
146      *(.bss)
147      . = ALIGN(8);
148  } >dccm : dccm_load

```




```

152     .stack :
153     {
154         _heap_end = .;
155         . = . + __stack_size;
156         sp = .;
157     } > dccm : dccm load

```

Figure 9. File *ld/link_DCCM.ld* in the CoreMark PlatformIO project

- Open file *platformio.ini* and uncomment line 18 (see Figure 10) so that the program uses the linker script from Figure 9 instead the default linker script. This way, as explained above, most data will be accessed in the fast DCCM instead of the slow DDR memory.



```

11 [env:swervolf_nexys]
12 platform = chipsalliance
13 board = swervolf_nexys
14 framework = wd-riscv-sdk
15
16
17 # DCCM/ICCM link scripts
18 #board_build.ldscript = ld/link_DCCM.ld
19 #board_build.ldscript = ld/link_DCCM-ICCM.ld

```

```

11 [env:swervolf_nexys]
12 platform = chipsalliance
13 board = swervolf_nexys
14 framework = wd-riscv-sdk
15
16
17 # DCCM/ICCM link scripts
18 board_build.ldscript = ld/link_DCCM.ld
19 #board_build.ldscript = ld/link_DCCM-ICCM.ld

```

Figure 10. File *platformio.ini* in the CoreMark PlatformIO project

- Run the program on the board and open the serial monitor. Then invert a switch on the board. You will obtain the results shown in Figure 11.

In this case, the CM/MHz (i.e., the value of Iterat/Sec/MHz) is 1.88. The number of cycles has decreased to about a half million cycles. As in the previous version of the processor, about a half million instructions are processed; so we obtain an IPC of 1. By mapping sections of the program to the DCCM, performance has increased by a factor of four.

Finally, the number of data transactions through the bus is now 0, given that data are stored in the DCCM.

```

58 while(1);
59
60
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
2K performance run parameters for coremark.

CoreMark Size      : 666
Total ticks        : 530028
Total time (secs): 530
Iterat/Sec/MHz     : 1.88
Iterations         : 1
Compiler version   : GCC8.3.0
Compiler flags     : -O2
Memory location    : STATIC
seedcrc           : 0xe9f5
[0]crclist        : 0xe714
[0]crcmatrix      : 0x1fd7
[0]crcstate       : 0x8e3a
[0]crcfinal       : 0xe714

Correct operation validated. See readme.txt for run and reporting rules.

Cycles = 529897
Instructions = 496678
Data Bus Transactions = 0
Inst Bus Transactions = 392

```

Figure 11. Execution results of the CoreMark benchmark

TASK: In file *platformio.ini* (see Figure 10), comment out line 18 and uncomment line 19 so that the program uses the linker script provided at:
`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM-ICCM.ld`. Analyse this new linker script, which uses both the DCCM for storing most data and the ICCM for storing the instructions. Execute the CoreMark benchmark and compare the results with the ones obtained in this section. In this case, given that our default RVfpga System does not include an ICCM, use either the bitstream that you created in the first task of this lab or the bitstream we provide at:
`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Bitstreams/rvfpganexys_DCCM-ICCM.bit`.

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 most of the data sections of the application – 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, follow the next steps:

- Use the `[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM.ld` linker script again. To do so, open file `platformio.ini` and uncomment line 18 (see Figure 10) and comment out line 19.
- **Using a different procedure than previously used**, run the program on the board: Upload the usual bitstream but then use option “Upload and Monitor” available in the Project Tasks of PlatformIO (see Figure 12).

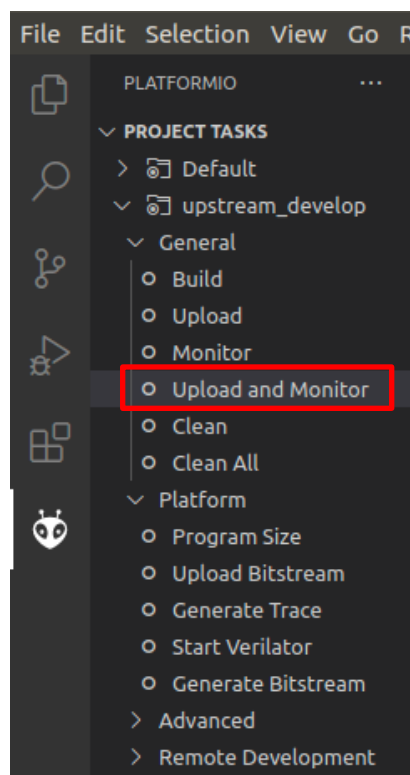


Figure 12. Upload and Monitor

This option will compile the program, execute it on the board and open the serial monitor. This option compiles using the optimization flags determined by the `build_flags` option in `platformio.ini`, in this case `-O2` (see Figure 13).

```
25 build_unflags = -Wa,-march=rv32imac -march=rv32imac -Os
26 build_flags = -Wa,-march=rv32ima -march=rv32ima -O2
27 extra_scripts = extra_script.py
```

Figure 13. File `platformio.ini`, option `build_flags`

Once the program starts executing, as usual, invert a switch on the board. You will obtain the results shown in Figure 14.

The CM/MHz (Iterat/Sec/MHz) is now 3.47. The number of cycles has decreased to around 288,000, and the number of instructions is now around 309,000. Even though the $IPC \approx 1$, the performance (CM/MHz and thus, execution time) is now much better than in the scenario analysed in Section B, as both the number of cycles and instructions have decreased significantly. This improvement is due to enabling compiler optimizations. The number of data bus transactions is still 0 given that data is stored in the DCCM.

```
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
2K performance run parameters for coremark.

CoreMark Size      : 666
Total ticks        : 288490
Total time (secs): 288
Iterat/Sec/MHz     : 3.47
Iterations         : 1
Compiler version   : GCC8.3.0
Compiler flags     : -O2
Memory location    : STATIC
seedcrc           : 0xe9f5
[0]crclist         : 0xe714
[0]crcmatrix       : 0x1fd7
[0]crcstate        : 0x8e3a
[0]crcfinal        : 0xe714

Correct operation validated. See readme.txt for run and reporting rules.

Cycles = 288337
Instructions = 309637
Data Bus Transactions = 0
Inst Bus Transactions = 504
```

Figure 14. Execution results of CoreMark when using compiler optimizations

TASK: Modify the compilation optimization to -O3 and explain the results.

4. EXERCISES

- 1) Do the same analysis as was done for CoreMark but this time using the Dhrystone benchmark. A PlatformIO project that contains the Dhrystone benchmark is in: `[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Dhrystone_HwCounters`. As required by all benchmarks, this Dhrystone benchmark has been adapted to the specific system, in this case the RVfpga System, using the sources provided at <https://github.com/chipsalliance/Cores-SweRV>. File `Test.c` is similar to the one from

CoreMark (Figure 6) but it invokes function `main_dhry()`, which includes the Dhrystone benchmark itself.

- 2) Do the same analysis as was done for CoreMark but this time for the ImageProcessing application. A PlatformIO project that contains the ImageProcessing application is in:
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/ImageProcessing_HwCounters. These are the applications we used in Lab 5 for transforming an RGB image into grayscale. File *Test.c* is similar to the one from CoreMark (Figure 6) but it invokes function `ImageTransformation()`, which includes the Image Transformation benchmark that we analysed in Lab 5. The DCCM of the default RVfpga System is not big enough to store the image, so instead use the RVfpga System (bitstream) that has a 128 KiB DCCM, which is at:
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Bitstreams/rvfpganexys_DCCM-128.bit.
- 3) Enable/disable various core features as described in Section 2.C of this lab. Compare the performance results – that is, values of the HW Counters when executing the programs on these modified cores. Run all three programs (CoreMark, Dhrystone, and ImageProcessing) on these modified RVfpga Systems on the Nexys A7 board. Variations include:
 - Using different Branch Predictor configurations and implementations (such as always not-taken, Gshare, and the bimodal predictor implemented in Exercise 1 of Lab 16).
 - Enabling/disabling the dual-issue feature.
 - Using various I\$/DCCM/ICCM configurations (such as different sizes or different I\$ Replacement Policies).