

## 1. TASKS

**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.


Solution not provided.

**TASK:** Draw a figure similar to Figure 2 for the ICCM implemented in the previous task.

Solution not provided.

**TASK:** Replicate the simulation from Figure 4 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 4. 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 43,900 ps.

Solution provided in the main document of Lab 20.

**TASK:** Explain how signals `rden_bank`, `wren_bank`, and `addr_bank` are obtained in lines 103, 104, and 105 of module `lsu_dccm_mem`.

```

101 // 8 Banks, 16KB each (2048 x 72)
102 for (genvar i=0; i<DCCM_NUM_BANKS; i++) begin: mem_bank
103     assign wren_bank[i] = dccm_wren & (dccm_wr_addr[2+:DCCM_BANK_BITS] == i);
104     assign rden_bank[i] = dccm_rden & ((dccm_rd_addr_hi[2+:DCCM_BANK_BITS] == i) | (dccm_rd_addr_lo[2+:DCCM_BANK_BITS] == i));
105     assign addr_bank[i][(DCCM_BANK_BITS+DCCM_WIDTH_BITS)+:DCCM_INDEX_BITS] = wren_bank[i] ? dccm_wr_addr[(DCCM_BANK_BITS+DCCM_WIDTH_BITS)+:DCCM_INDEX_BITS] :
106                                     (((dccm_rd_addr_hi[2+:DCCM_BANK_BITS] == i) & rd_unaligned) ?
107                                     dccm_rd_addr_hi[(DCCM_BANK_BITS+DCCM_WIDTH_BITS)+:DCCM_INDEX_BITS] :
108                                     dccm_rd_addr_lo[(DCCM_BANK_BITS+DCCM_WIDTH_BITS)+:DCCM_INDEX_BITS]);

```

Signal `wren_bank`

- Signal `wren_bank[7:0]` contains 8 bits, one per bank. Writing bank *i* is enabled when `wren_bank[i]==1`.
- If the LSU sets signal `dccm_wren` (we analysed this signal in Lab 13), one bank is written, as determined by field `Bank` of the address provided in: `dccm_wr_addr`.

Signal `rden_bank`

- Signal `rden_bank[7:0]` contains 8 bits, one per bank. Reading of bank *i* is enabled when `rden_bank[i]==1`.
- If the LSU sets signal `dccm_rden` (we analysed this signal in Lab 13), one or two banks are read (depending on the access being aligned or unaligned), as determined by field `Bank` of the addresses provided in: `dccm_rd_addr_lo` and `dccm_rd_addr_hi`.

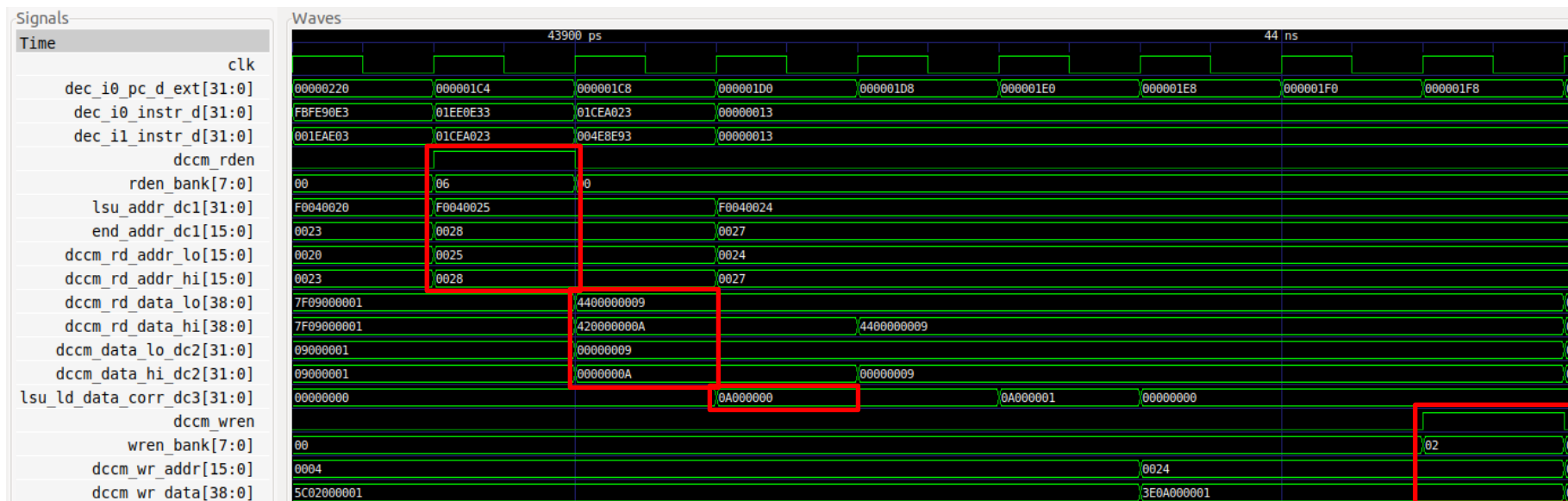
Signal `addr_bank`

- Signal `wren_bank[7:0][11]` contains 8 11-bit addresses, one per bank.

- In case of a write, the address is obtained from signal `dccm_wr_addr`.
- In case of a read, the address is either in signal `dccm_rd_addr_lo` (upon an aligned read), or signals `dccm_rd_addr_lo` and `dccm_rd_addr_hi` (upon an unaligned read).

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

```
lw t3, (t4) → lw t3, 1(t4)
```



- Signal `dccm_rden` = 0x06, thus two banks are enabled for reading.
- Two values are provided to the core:
  - `dccm_data_lo_dc2` = 0x9
  - `dccm_data_hi_dc2` = 0xA

- The core aligns the value as explained in Lab 13: `lsu_ld_data_corr_dc3 = 0x0A000000`
- 5 cycles later the value, plus one, is written in the DCCM: `dccm_wr_data = 0x3E0A000001`

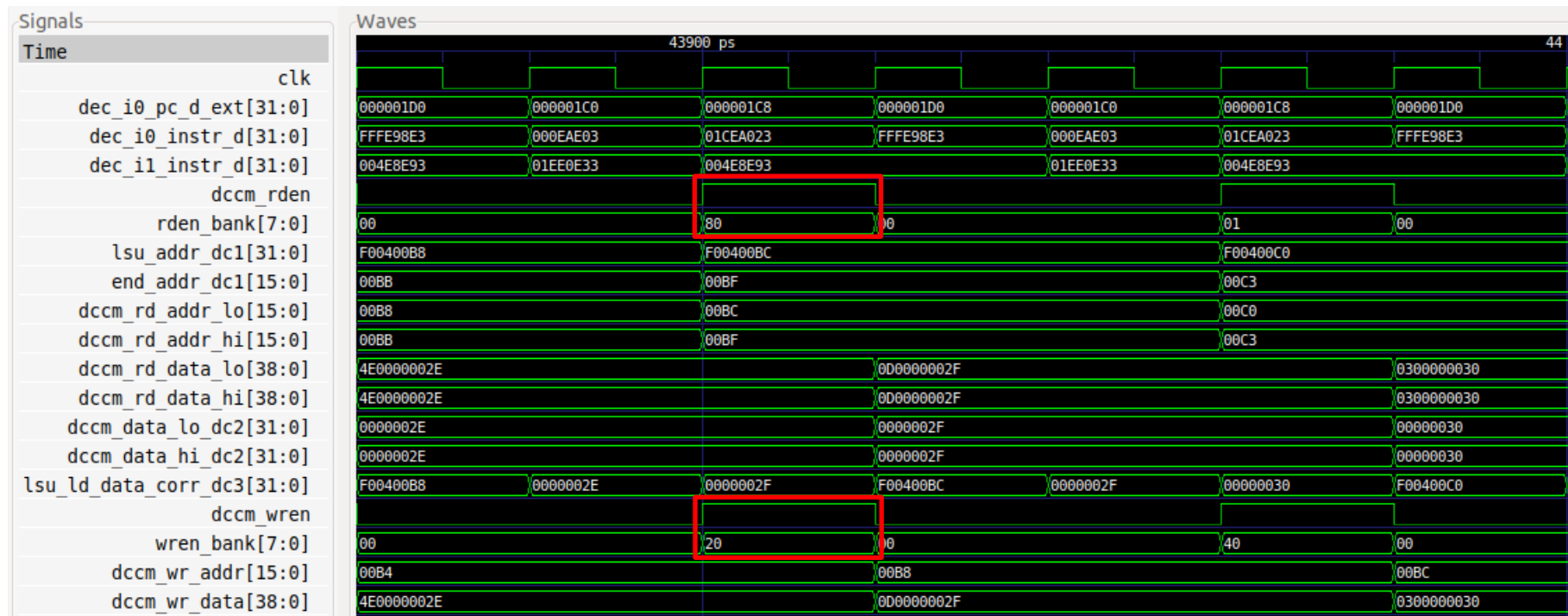
**TASK:** Simulate a DCCM bank conflict by modifying the program from Figure 4 (*[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW\_Instruction\_DCCM*).

**1<sup>st</sup> modification:** Remove the 20 `nop` instructions, regenerate the simulation, and analyse the `lw` and `sw` in a random iteration of the loop.

**2<sup>nd</sup> modification:** Modify the immediate of the `sw` instruction to make the `lw` and `sw` try to access the same bank in the same cycle:  
`sw t3, (t4) → sw t3, 8(t4)`

### 1<sup>st</sup> modification:

```
// 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
    bne t4, t6, REPEAT_Access    # Repeat the loop
```



In this case, the DCCM read and DCCM write occur in the same cycle. However, because they are to different banks, they can be performed in the same cycle.

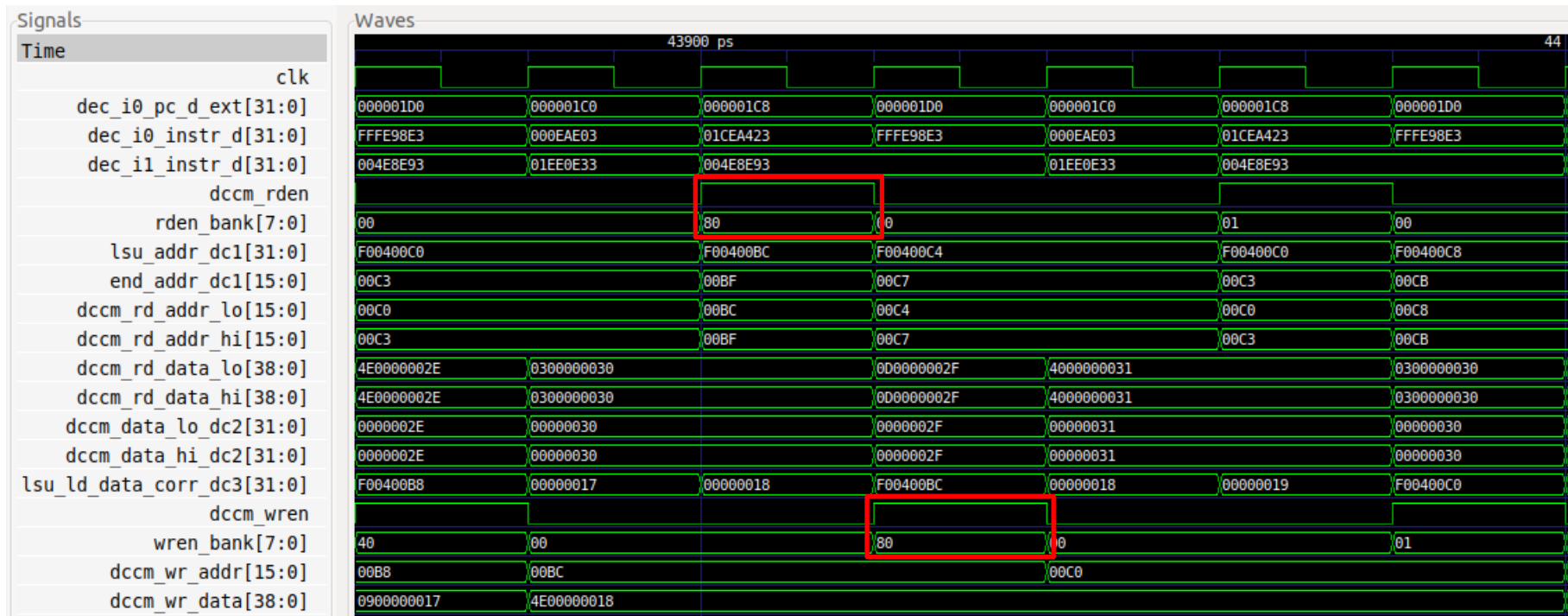
## 2<sup>nd</sup> modification:

```
// 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, 8(t4)
    add t4, t4, 4
    bne t4, t6, REPEAT_Access    # Repeat the loop

```



Again, the DCCM read and DCCM write occur in the same cycle. However, in contrast to the last example, the read and write are to the same bank, so the write is delayed one cycle.

**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/rvfpganexys\_DCCM-ICCM.bit*.

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

CoreMark Size : 666  
Total ticks : 515967  
Total time (secs): 515  
Iterat/Sec/MHz : 1.94  
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 = 515855  
Instructions = 496680  
Data Bus Transactions = 0  
Inst Bus Transactions = 0

In this case, the CM/MHz (i.e., the value of Iterat/Sec/MHz) is 1.94. When using the DCCM only, the CM/MHz is 1.88. This slight increase in performance is due to a small decrease in the number of cycles, when compared to using only the DCCM. The improvement is small because the SweRV EH1 processor has an I\$ and thus, using the ICCM makes only a small difference. Finally, you can observe that now both the data and instruction bus transactions are 0, given that both data and instructions are accessed from the DCCM and ICCM respectively.



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

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

CoreMark Size      : 666
Total ticks        : 268997
Total time (secs): 268
Iterat/Sec/MHz     : 3.73
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 = 268869
Instructions = 292421
Data Bus Transactions = 0
Inst Bus Transactions = 1760
```

In this case, the CM/MHz (i.e., value of Iterat/Sec/MHz) is 3.73. The number of instructions, and thus number of cycles, has decreased somewhat with respect to the execution where the DCCM and -O2 are used.

## 2. 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.

- No compiler optimizations, no DCCM, no ICCM

```
Cycles = 1838481
Instructions = 402057
Data Bus Transactions = 194011
Inst Bus Transactions = 232
```

- DCCM

```
Cycles = 475969
Instructions = 402057
Data Bus Transactions = 0
Inst Bus Transactions = 240
```

- DCCM and ICCM

```
Cycles = 475173
Instructions = 402057
Data Bus Transactions = 0
Inst Bus Transactions = 0
```

- Compiler optimizations (-O2) and DCCM

```
Cycles = 250481
Instructions = 274082
Data Bus Transactions = 0
Inst Bus Transactions = 176
```

- Compiler optimizations (-O3) and DCCM

```
Cycles = 236660
Instructions = 264082
Data Bus Transactions = 0
Inst Bus Transactions = 160
```

- 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`.

- No compiler optimizations, no DCCM, no ICCM

```
Created Grey Image
Cycles = 3218631
Instructions = 951907
Data Bus Transactions = 233659
Inst Bus Transactions = 64
```

- DCCM

```
Created Grey Image
Cycles = 735838
Instructions = 952263
Data Bus Transactions = 4119
Inst Bus Transactions = 64
```

- Compiler optimizations (-O2) and DCCM

```
Created Grey Image  
Cycles = 358716  
Instructions = 456133  
Data Bus Transactions = 4132  
Inst Bus Transactions = 40
```

- Compiler optimizations (-O3) and DCCM

```
Created Grey Image  
Cycles = 285475  
Instructions = 346027  
Data Bus Transactions = 4134  
Inst Bus Transactions = 48
```

- 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).

Solution not provided.