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

```
for (genvar i=0; i<pt.DCCM_NUM_BANKS; i++) begin: mem_bank
   assign  wren_bank[i]        = dccm_wren & ((dccm_wr_addr_hi[2+:pt.DCCM_BANK_BITS] == i) | (dccm_wr_addr_lo[2+:pt.DCCM_BANK_BITS] == i));
   assign  rden_bank[i]        = dccm_rden & ((dccm_rd_addr_hi[2+:pt.DCCM_BANK_BITS] == i) | (dccm_rd_addr_lo[2+:pt.DCCM_BANK_BITS] == i));
   assign  addr_bank[i][(pt.DCCM_BANK_BITS+DCCM_WIDTH_BITS)+:DCCM_INDEX_BITS] = wren_bank[i] ? (((dccm_wr_addr_hi[2+:pt.DCCM_BANK_BITS] == i) & wr_unaligned) ?
                                                                          dccm_wr_addr_hi[(pt.DCCM_BANK_BITS+DCCM_WIDTH_BITS)+:DCCM_INDEX_BITS] :
                                                                          dccm_wr_addr_lo[(pt.DCCM_BANK_BITS+DCCM_WIDTH_BITS)+:DCCM_INDEX_BITS])  :
                                                                     (((dccm_rd_addr_hi[2+:pt.DCCM_BANK_BITS] == i) & rd_unaligned) ?
                                                                          dccm_rd_addr_hi[(pt.DCCM_BANK_BITS+DCCM_WIDTH_BITS)+:DCCM_INDEX_BITS] :
                                                                          dccm_rd_addr_lo[(pt.DCCM_BANK_BITS+DCCM_WIDTH_BITS)+:DCCM_INDEX_BITS]);
```

Signal `wren_bank`

- In our case, DCCM_NUM_BANKS=4, thus signal `wren_bank[3:0]` contains 4 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 or two banks are written (depending on the access being aligned or unaligned), as determined by field Bank of the address provided in: `dccm_wr_addr_lo` and `dccm_wr_addr_hi`.

Signal `rden_bank`

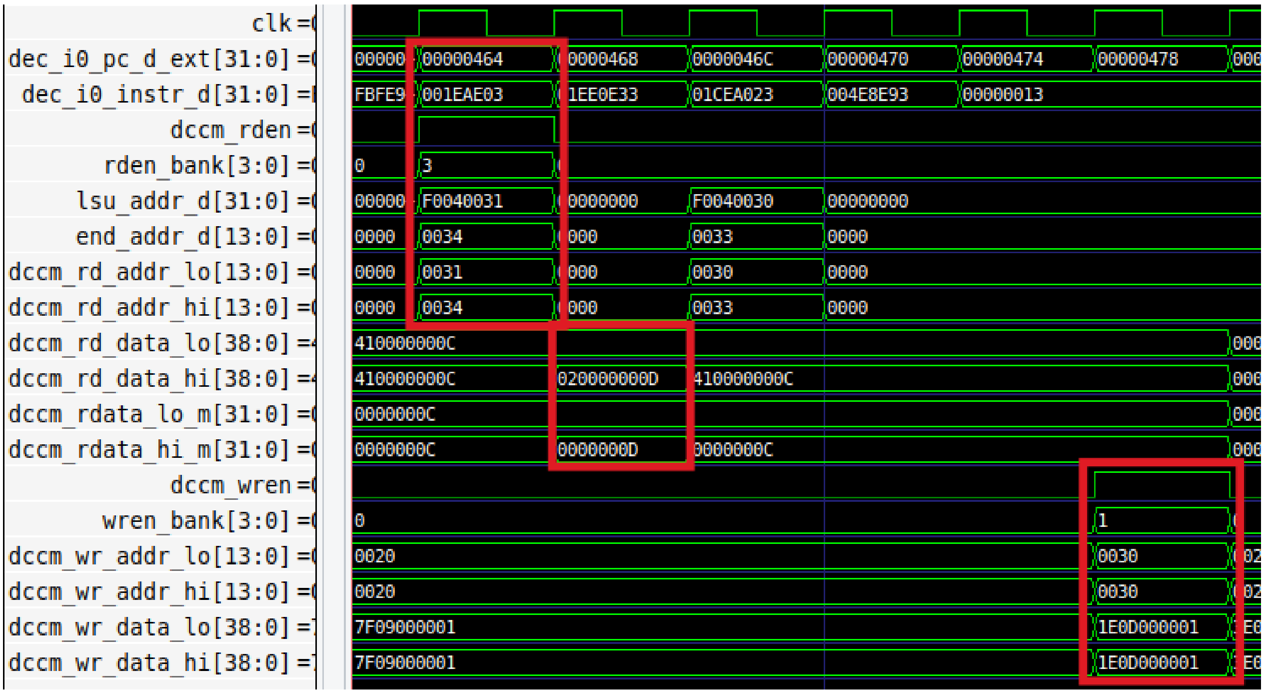
- In our case, DCCM_NUM_BANKS=4, thus signal `rden_bank[3:0]` contains 4 bits, one per bank. Reading of bank *i* is enabled when `rden_bank[i]`==1.
- If the LSU sets signal `dccm_rden` (we analysed this signal in Lab 13), one or two banks are read (depending on the access being aligned or unaligned), as determined by field Bank of the addresses provided in: `dccm_rd_addr_lo` and `dccm_rd_addr_hi`.

Signal `addr_bank`

- Signal `addr_bank[3:0][9:0]` contains 8 10-bit addresses, one per bank.
  - In case of a write, the address is obtained in signal `dccm_wr_addr_lo` (upon an aligned write), or signals `dccm_wr_addr_lo` and `dccm_wr_addr_hi` (upon an unaligned write).
  - 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 used above (*[RVfpgaEL2NexysA7NoDDRPath]/Labs/Lab20/LW-SW_Instruction_DCCM/*) and simply substitute the load instruction as follows:

`lw t3, (t4)` → `lw t3, 1(t4)`



- Signal `dccm_rden` = 0x03, thus two banks are enabled for reading.

- Two values are provided to the core:
  - `dccm_rd_data_lo` = 0x410000000C
  - `dccm_rd_data_hi` = 0x020000000D

- The core aligns the value into signal `lsu_ld_data_m` = 0x0D000000

- A few cycles later, the value plus one is written in the DCCM: dccm_wr_data_lo = 0x1E0D000001

<u>TASK</u>: Simulate a DCCM bank conflict by modifying the program from *[RVfpgaEL2NexysA7NoDDRPath]/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.

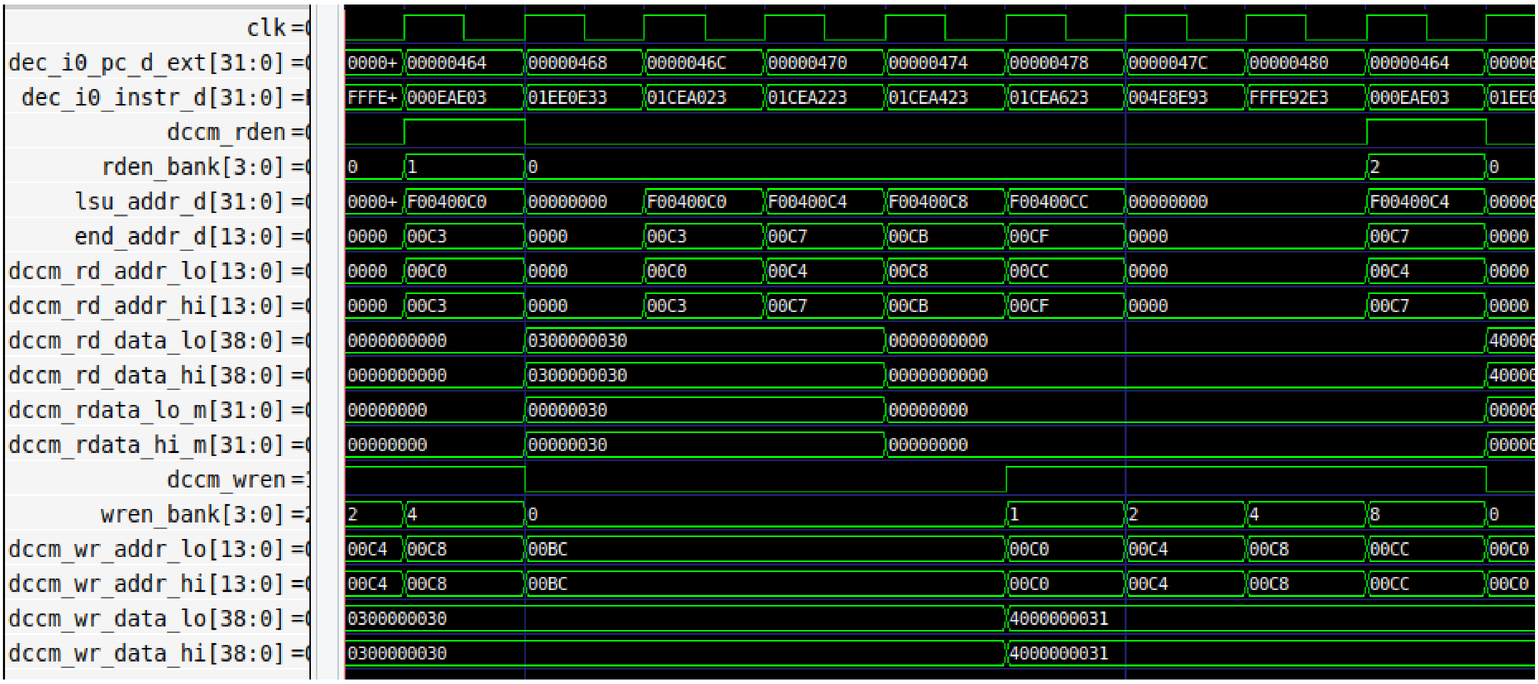
2nd modification: Replace the `sw` instruction for 4 consecutive `sw` instructions (each accessing a different bank), making the `lw` and `sw` try to access the same bank in the same cycle:

```
sw t3, (t4) →  sw t3, (t4)
               sw t3, 4(t4)
               sw t3, 8(t4)
               sw t3, 12(t4)
```

Test different offset combinations and compare a program with no conflicts and a program with bank conflicts.

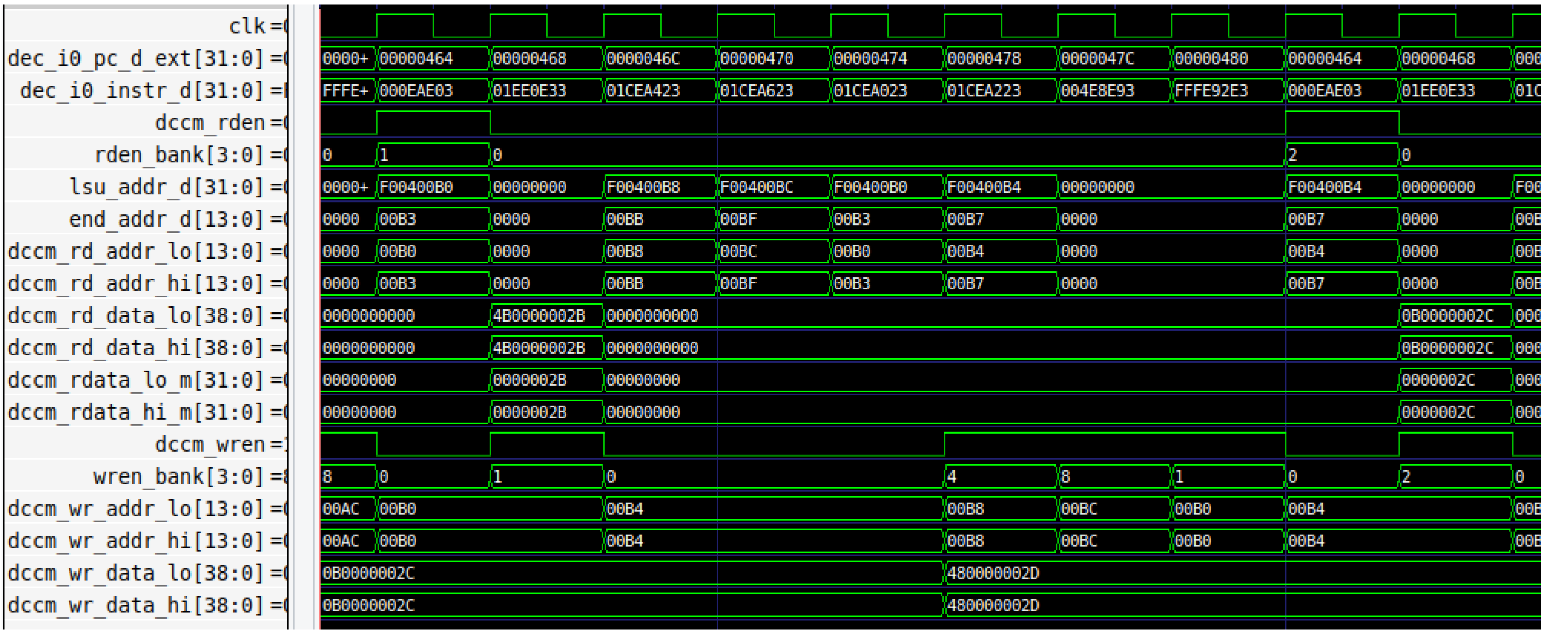
```
REPEAT_Access:
      lw t3, (t4)
```

```
        add t3, t3, t5
        sw t3, (t4)
        sw t3, 4(t4)
        sw t3, 8(t4)
        sw t3, 12(t4)
        add t4, t4, 4
        bne t4, t6, REPEAT_Access # Repeat the loop
```



In this case, the load to bank 2 and the store to bank 8 (last cycle shown in the figure) can happen in the same cycle.

```
REPEAT_Access:
        lw t3, (t4)
        add t3, t3, t5
        sw t3, 8(t4)
        sw t3, 12(t4)
        sw t3, (t4)
        sw t3, 4(t4)
        add t4, t4, 4
        bne t4, t6, REPEAT_Access # Repeat the loop
```



In this case, the store to bank 2 has to be delayed 1 cycle as it conflicts with the load to the same bank (last two cycles shown in the figure).

1. 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*. As required by all benchmarks, this Dhrystone benchmark has been adapted to the specific system, in this case the RVfpgaEL2 System. File *Test.c* is similar the one used in CoreMark but it invokes function `main_dhry()`, which includes the Dhrystone benchmark itself.

Solution not provided.

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 RVfpga Systems on the Nexys A7 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).

Solution not provided.