

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]);
end
```

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`

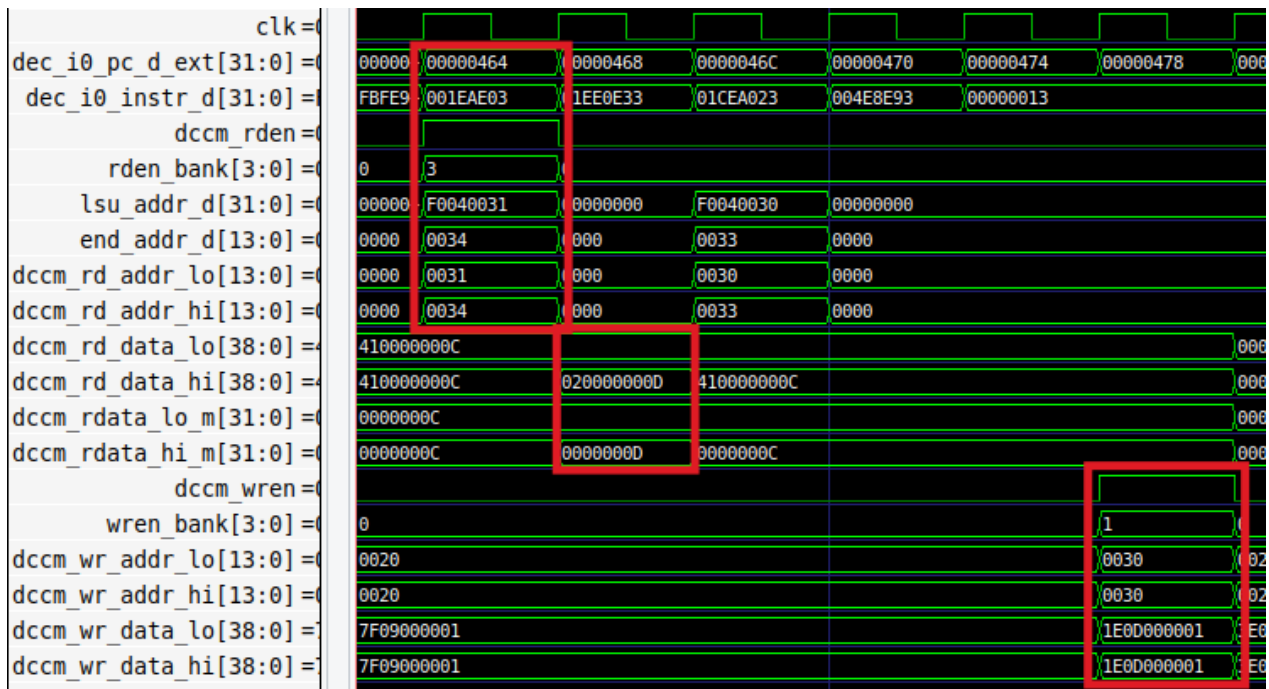
- In our case, `DCCM_NUM_BANKS=4`, thus signal `rden_bank[3:0]` contains 4 bits, one per bank. Reading of bank *i* is enabled when `rden_bank[i]==1`.
- If the LSU sets signal `dccm_rden` (we analysed this signal in Lab 13), one or two banks are read (depending on the access being aligned or unaligned), 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.
 - o 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).
 - o 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 (`[RVfpgaBasysPath]/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:
 - o `dccm_rd_data_lo` = 0x41000000C
 - o `dccm_rd_data_hi` = 0x02000000D
- 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` = 0x1E0D00001

TASK: Simulate a DCCM bank conflict by modifying the program from `[RVfpgaBasysPath]/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
```

clk=											
dec_i0_pc_d_ext[31:0]=	0000+	00000464	00000468	0000046C	00000470	00000474	00000478	0000047C	00000480	00000464	00000468
dec_i0_instr_d[31:0]=	FFFE+	000EAE03	01EE0E33	01CEA023	01CEA223	01CEA423	01CEA623	004E8E93	FFFE92E3	000EAE03	01EE0E33
dccm_rden=											
rden_bank[3:0]=	0	1	0						2	0	
lsu_addr_d[31:0]=	0000+	F00400C0	00000000	F00400C0	F00400C4	F00400C8	F00400CC	00000000		F00400C4	00000000
end_addr_d[13:0]=	0000	00C3	0000	00C3	00C7	00CB	00CF	0000		00C7	0000
dccm_rd_addr_lo[13:0]=	0000	00C0	0000	00C0	00C4	00C8	00CC	0000		00C4	0000
dccm_rd_addr_hi[13:0]=	0000	00C3	0000	00C3	00C7	00CB	00CF	0000		00C7	0000
dccm_rd_data_lo[38:0]=	0000000000	0300000030			0000000000					4000000031	
dccm_rd_data_hi[38:0]=	0000000000	0300000030			0000000000					4000000031	
dccm_rdata_lo_m[31:0]=	00000000	00000030			00000000					00000000	
dccm_rdata_hi_m[31:0]=	00000000	00000030			00000000					00000000	
dccm_wren=											
wren_bank[3:0]=	2	4	0			1		2	4	8	0
dccm_wr_addr_lo[13:0]=	00C4	00C8	00BC			00C0		00C4	00C8	00CC	00C0
dccm_wr_addr_hi[13:0]=	00C4	00C8	00BC			00C0		00C4	00C8	00CC	00C0
dccm_wr_data_lo[38:0]=	0300000030					4000000031					
dccm_wr_data_hi[38:0]=	0300000030					4000000031					

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
```

Imagination
university programme

clk=												
dec_i0_pc_d_ext[31:0]=	0000+	00000464	00000468	0000046C	00000470	00000474	00000478	0000047C	00000480	00000464	00000468	0000046C
dec_i0_instr_d[31:0]=	FFFE+	000EAE03	01EE0E33	01CEA423	01CEA623	01CEA023	01CEA223	004E8E93	FFFE92E3	000EAE03	01EE0E33	01CEA423
dccm_rden=												
rden_bank[3:0]=	0	1	0							2	0	
lsu_addr_d[31:0]=	0000+	F00400B0	00000000	F00400B8	F00400BC	F00400B0	F00400B4	00000000		F00400B4	00000000	F00400B8
end_addr_d[13:0]=	0000	00B3	0000	00BB	00BF	00B3	00B7	0000		00B7	0000	00BB
dccm_rd_addr_lo[13:0]=	0000	00B0	0000	00B8	00BC	00B0	00B4	0000		00B4	0000	00B8
dccm_rd_addr_hi[13:0]=	0000	00B3	0000	00BB	00BF	00B3	00B7	0000		00B7	0000	00BB
dccm_rd_data_lo[38:0]=	0000000000	4B0000002B	0000000000							0B0000002C	0000000000	
dccm_rd_data_hi[38:0]=	0000000000	4B0000002B	0000000000							0B0000002C	0000000000	
dccm_rdata_lo_m[31:0]=	00000000	0000002B	00000000							0000002C	00000000	
dccm_rdata_hi_m[31:0]=	00000000	0000002B	00000000							0000002C	00000000	
dccm_wren=												
wren_bank[3:0]=	8	0	1	0		4	8	1	0		2	0
dccm_wr_addr_lo[13:0]=	00AC	00B0		00B4		00B8	00BC	00B0		00B4		00B8
dccm_wr_addr_hi[13:0]=	00AC	00B0		00B4		00B8	00BC	00B0		00B4		00B8
dccm_wr_data_lo[38:0]=	0B0000002C					480000002D						
dccm_wr_data_hi[38:0]=	0B0000002C					480000002D						

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.

Execution using -O3:

```
User time 0
Measured time too small to obtain meaningful results
Please increase number of runs

Cycles = 6077034
Instructions = 2620081
Data Bus Transactions = 1470765
Inst Bus Transactions = 432
```

Disconnect Clear UART output

7 SEGMENT DISPLAYS: 0 0 0 0



- 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. 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.