THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 13

Memory Instructions: `lw` and `sw` Instructions

1. Introduction

In the previous labs we introduced the basic concepts of pipelining and its use in the VeeR EL2 processor, and we analysed how Arithmetic-Logic instructions are executed in this processor. In this lab, we continue with the analysis of basic instructions; specifically, we analyse memory reads and writes.

The memory system is one of the most critical performance bottlenecks in modern computers. Memory latencies are usually much higher than the core clock cycle, so the processor may have to stall while waiting for data from memory.

In this lab, we first examine the *Load/Store pipe* (the set of pipeline stages devoted to executing *load/store* operations) when reading a low-latency memory location – that is, one that does not stall the processor. We then examine store instruction execution. Finally, we repeat our analysis ignoring the low-latency memory and directly interfacing with the main memory used on the board.

Figure 1 illustrates a high-level view of the microarchitecture of the VeeR EL2 processor. The figure highlights the stages that are relevant in this lab.

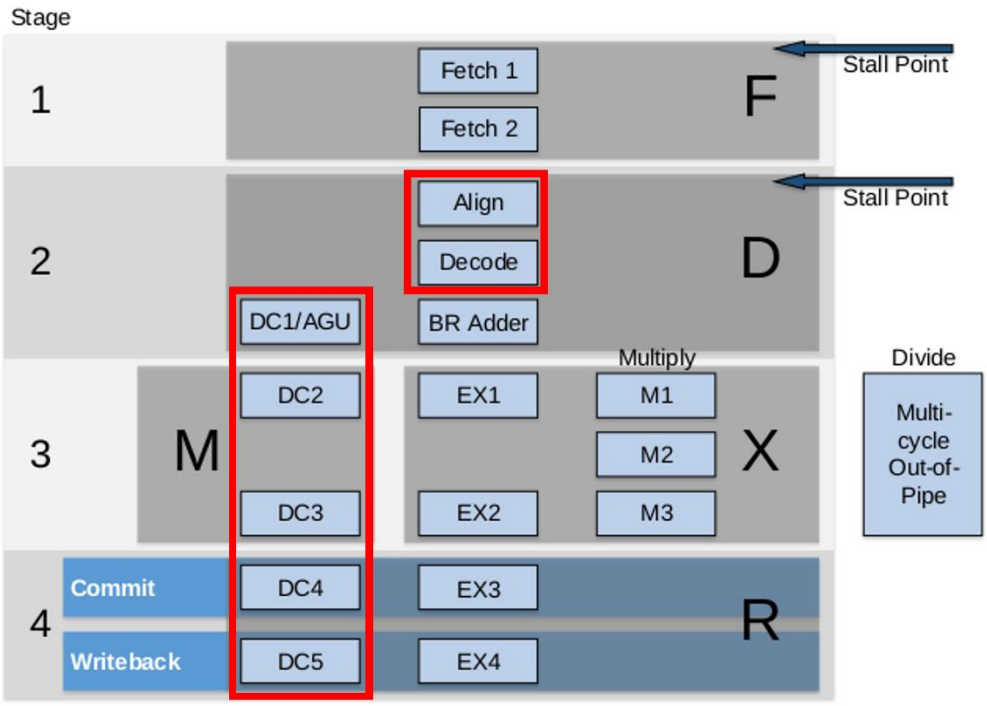


Figure 1 VeeR EL2 core microarchitecture
(figure from https://github.com/chipsalliance/Cores-VeeR-EL2/blob/main/docs/RISC-V_VeeR_EL2_PRM.pdf)

2. The `lw` Instruction Accessing a Low-Latency Memory

In this section we use the simple code in Figure 2 to illustrate the most relevant events of the execution of a load instruction. The example program consists of a loop that contains two `lw` (*load word*) operations (highlighted in red), each reading a 32-bit word from consecutive word-aligned memory addresses. All iterations access the same data and do nothing with them.

As in Lab 12, the `lw` instructions (highlighted in red in the figure) are surrounded by several `nop` (no-operation) instructions in order to isolate them from preceding and subsequent instructions. For the sake of simplicity, in this lab we also disable the use of compressed instructions.

```
.globl main

.section .midccm
A: .space 8

.text

main:

# Register t3 = x28 (register 28)
la  t0, A                              # t0 = addr(A)
li  t1, 0x2                            # t1 = 2
sw  t1, (t0)                           # A[0] = 2
add t1, t1, 6                          # t1 = 8
sw  t1, 4(t0)                          # A[1] = 8
INSERT_NOPS_9

REPEAT:
       INSERT_NOPS_1
       lw t1, (t0)
       INSERT_NOPS_9
       INSERT_NOPS_4
       lw t1, 4(t0)
       INSERT_NOPS_10
       INSERT_NOPS_4
       beq  zero, zero, REPEAT    # Repeat the loop

.end
```

Figure 2 Example program with two `lw` instructions

Folder *[RVfpgaBasysPath]/Labs/Lab13/LW_Instruction_DCCM* provides the Catapult project so that you can analyse, simulate, and change the program. Open the project in Catapult, build it, and open the disassembly file. In that file, locate the second `lw` instruction, which is at address 0x00000334. Notice the machine code for the instruction (0x0042a303):

```
0x00000334:      0042a303          lw  t1,4(t0)
```

TASK: Verify that these 32 bits (0x0042a303) correspond to instruction `lw t1,4(t0)` in the RISC-V architecture.

So far, we have been using main memory for storing both the instructions and the data from our program. However, accessing that memory requires several cycles and makes it difficult to analyse the stages of a load/store instruction; thus, in this section, we use the low-latency DCCM (data closely-coupled memory) for storing program data.

IMPORTANT NOTE: The Basys 3 FPGA is quite small so that including both an I$ and a DCCM is not possible in our RVfpgaEL2 System. Thus, most of the examples included in this lab can only be done in simulation. The default configuration of the simulators provided at *[RVfpgaBasysPath]/Simulators* is the same as the default configuration of the provided bitstream, so you must modify the configuration used by the simulators as explained in Lab 11 in order to add a DCCM to the SoC.

For the sake of simplicity, in this lab we provide the new simulators at *[RVfpgaBasysPath]/Labs/Lab13/ExtendedSoC*, using the following configuration (note that the differences with respect to the default configuration are highlighted in red):

```
./veer.config -unset=assert_on -set=reset_vec=0x80000000 -
set=ret_stack_size=2 -set=btb_enable=1 -set=btb_size=8 -set=bht_size=32
-set=dccm_enable=1 -set=dccm_size=16 -set=dma_buf_depth=2 -
set=iccm_enable=0 -set=icache_enable=1 -set=icache_ecc=0 -
set=icache_size=8 -set=icache_2banks=0 -set=icache_num_ways=2 -
set=pic_size=32 --set=bitmanip_zba=0 -set=bitmanip_zbb=0 -
set=bitmanip_zbc=0 -set=bitmanip_zbe=0 -set=bitmanip_zbf=0 -
set=bitmanip_zbp=0 -set=bitmanip_zbr=0 -set=bitmanip_zbs=0 -
set=fast_interrupt_redirect=0
```

The DCCM is a local memory tightly coupled to the core. It provides low-latency access and SECDED ECC protection. Its size is set as an argument at build time of the core, ranging from 4 KiB to 512 KiB. Note that, this way, everything happens inside the VeeR EL2 Core Complex (Figure 3), where both the VeeR EL2 pipeline and the DCCM are placed (highlighted in red).

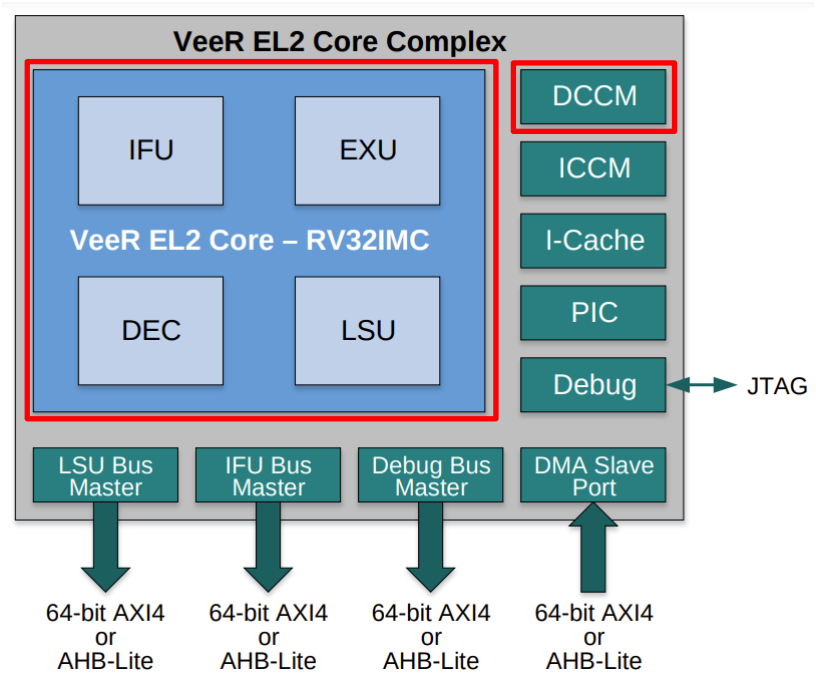


Figure 3 VeeR EL2 Core Complex

The code in Figure 2 defines an *ad-hoc* section called `.midccm` to allocate space in the DCCM. The address space of the DCCM starts at 0xF0040000 in our default RVfpga System. The linker script provided with this project (available at: *[RVfpgaBasysPath]/Labs/Lab13/LW_Instruction_DCCM/libraries/Ldscript.ld*) will take care of the proper address assignments.

Figure 4 shows the execution of the second `lw` instruction for an intermediate iteration of the loop from Figure 2. The signals shown are the ones specified in file: *[RVfpgaBasysPath]/Labs/Lab13/LW_Instruction_DCCM/commandLine/scriptLoad.tcl*. Note

that all iterations are the same: the first load reads the DDCM's first data word (2) into *t1* (*x6*); the second load reads the DDCM's second data word (8) into the same register (*t1*).

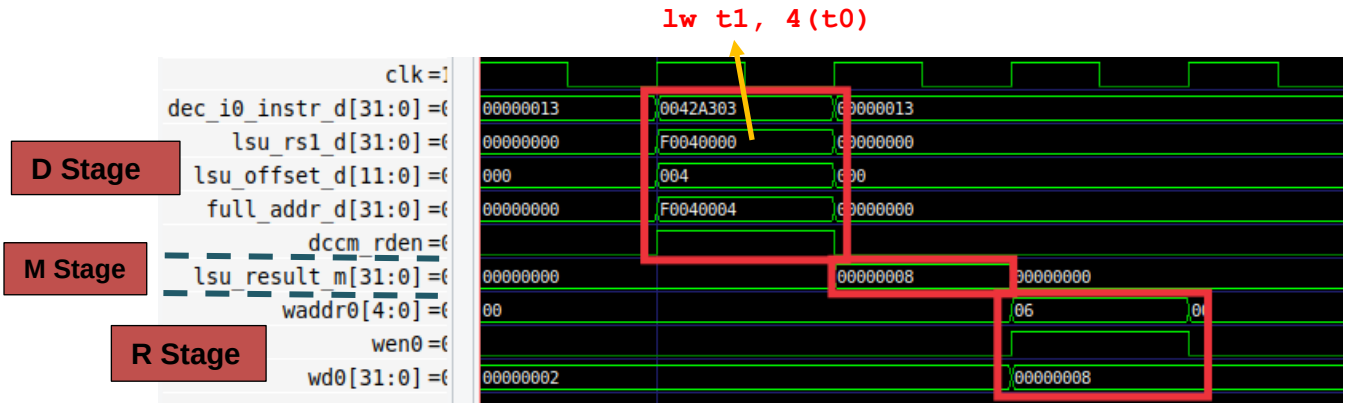


Figure 4. Verilator simulation for example program in Figure 2

Figure 5 shows a high-level view of the VeeR EL2 pipeline during the execution of the second `lw` instruction. Note that the figure merges the state of the processor in different cycles:

- **Cycle i:** **D Stage:** The instruction is decoded, the register file is read and the effective address is computed using the adder.
- **Cycle i+1:** **M Stage:** The DDCM is read using the address computed in the previous stage.
- **Cycle i+2:** **R Stage:** The value read from memory is written to the Register File.

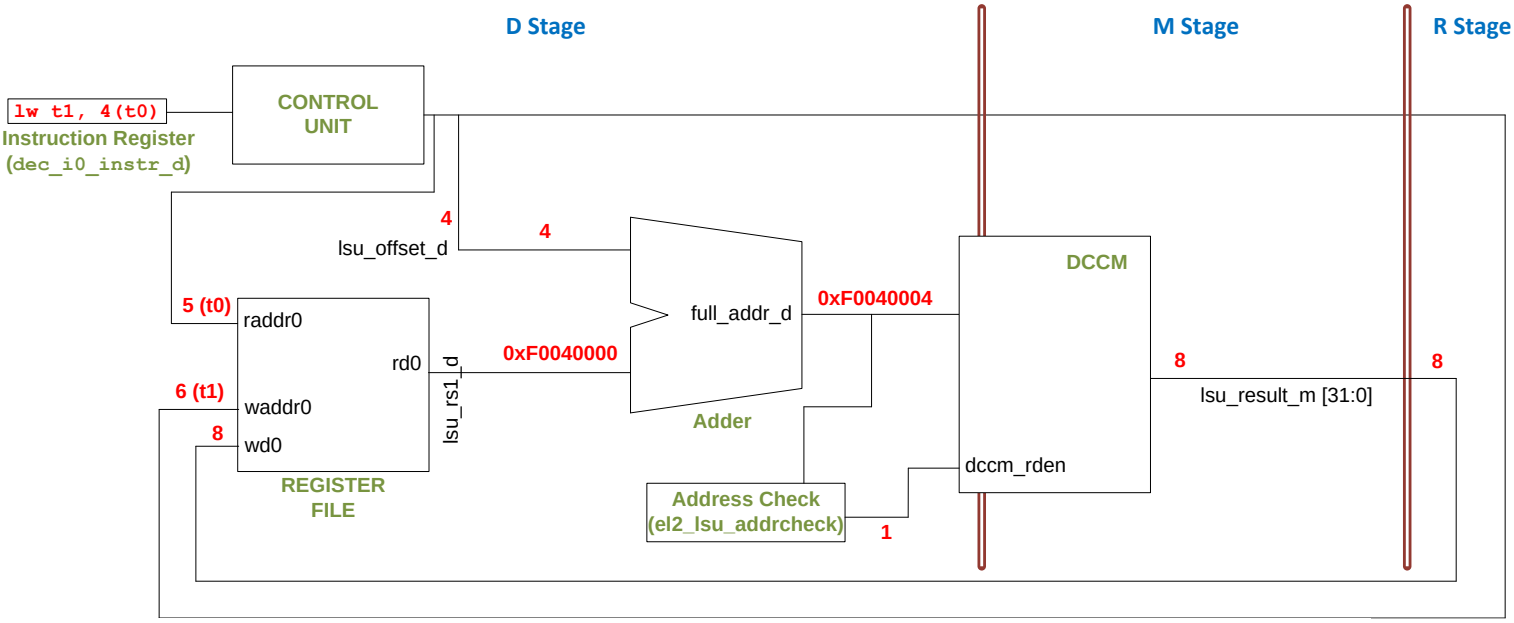


Figure 5. High-level view of the `lw` instruction executing in the VeeR EL2 pipeline

TASK: Replicate the simulation from Figure 4 on your own computer. Follow the steps described in detail in the GSG.

Analyse the waveform from Figure 4 and the diagram from Figure 5 at the same time. The figures include some signals associated with the D, M, and R stages. The values highlighted in red correspond to the second `lw` instruction as it traverses these stages.

- **Cycle i: D Stage:** signal `dec_i0_instr_d` contains the 32 bits of the `lw` machine instruction (0x0042a303).

  During this stage, the **control signals are generated** and the **operands for computing the load effective address are obtained**: signal `lsu_rs1_d` contains the base address of the `lw` operation (which in this example is held in register `t0` and is equal to 0xF0040000), and signal `lsu_offset_d` contains the 12-bit signed immediate extracted from the instruction (0x004 in this example).

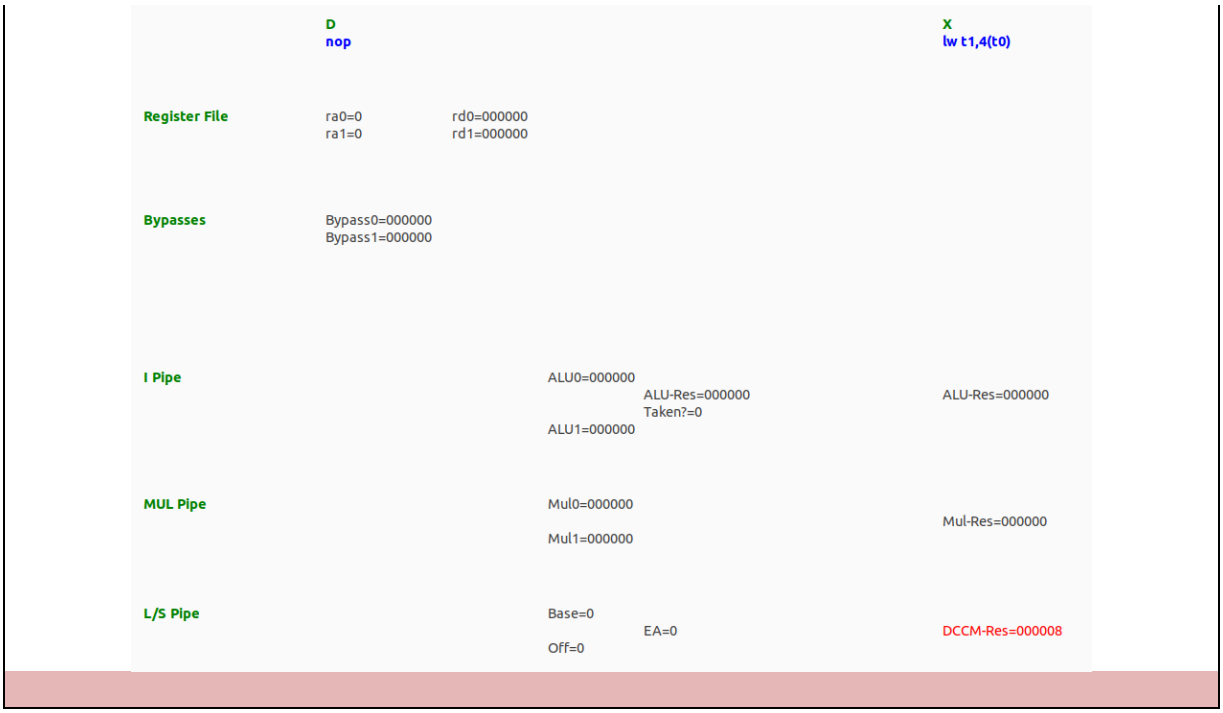
  The address is computed using an adder called *lsadder* and located inside module **`el2_lsu_lsc_ctl`**. The address is the base address (`lsu_rs1_d` = 0xF0040000) plus the sign-extended offset (`lsu_offset_d` = 0x00000004); the final address is `full_addr_d` = 0xF0040004. This address is checked (Address Check) to determine the memory region of the access (DCCM, PIC, or Main Memory). In this example, given that the final address belongs to the DCCM range (0xF0040004), `dccm_rden` asserts to enable the read of the corresponding DCCM bank. The final address (`full_addr_d`) and the enable signal (`dccm_rden`) are provided to the DCCM, which is read in the next cycle.

- **Cycle i+1: M Stage**: the DCCM is read and the data is placed in `lsu_result_m` = 0x8, which is propagated to the next stage.

- **Cycle i+2: R Stage**: Finally, the value read from memory is **written back** to the register file using signal `wd0` = 0x8. Given that `wen0` = 1, the value is written at the end of that cycle into register x6 (`waddr0` = 0x6).

TASK: Simulate the previous program on RVfpgaEL2-Pipeline in order to perform a similar analysis to the one just carried out. Remember that you must include the control instruction at the point where you want the simulator to stop execution (`and zero, t4, t5`).

For example, this is the VeeR EL2 pipeline when the load is at the M Stage when the DCCM is read (`DCCM-Res=8`). In the previous cycle (load at the D Stage) you will see the Effective Address computation and in the next one (load at the R Stage you will see the Register File writing).



TASK: Perform a more detailed analysis of the `lw` instruction, similar to the one performed in Section 2.B of Lab 12 for an `add` instruction, or to the one performed in Section 2.B of Lab 13 of the RVfpga package for VeeR EL2.

3. The `sw` Instruction Accessing a Low-Latency Memory

In this section we use the code shown in Figure 6 to illustrate the most relevant events of the execution of a store instruction. The code contains a loop with 1000 iterations that writes to consecutive addresses of memory. Vector A contains 1000 words and is placed at the DCCM (remember that the DCCM contains the address range: 0xF0040000 – 0xF004FFFF). Each `sw` is followed by a `lw` that checks that the correct value was stored. As usual, `nop`s are inserted to isolate the instructions and, in this case, also to ensure that the data is actually written to and read from memory and not just forwarded from the `sw` instruction to the `lw`. As usual, we disable the use of compressed instructions.

```
.globl main

.section .midccm
A: .space 4000

.text

main:

la t0, A                    # t0 = addr(A)
li t1, 0x2                  # t1 = 2
li t2, 1000                 # t2 = 1000

INSERT_NOPS_10

REPEAT:
    sw t1, (t0)
```

```
    INSERT_NOPS_10
    lw t1, (t0)
    INSERT_NOPS_10
    add t1,t1,t1
    add t0,t0,0x04
    add t2,t2,-1
    INSERT_NOPS_10
    bne  t2, zero, REPEAT   # Repeat the loop
    nop
    nop

.end
```

Figure 6 Example code with `sw` instruction

Folder *[RVfpgaBasysPath]/Labs/Lab13/SW_Instruction_DCCM* provides the Catapult project so that you can analyse, simulate, and modify the program. Open the project, build it, and open the disassembly file. You will see that the `sw` instruction is placed at address 0x000002f8, and you can also see the machine code for the instruction (0x0062a023):

```
0x000002f8:      0062a023            sw  t1,0(t0)
```

TASK: Verify that these 32 bits (0x0062a023) correspond to instruction `sw t1,0(t0)` in the RISC-V architecture.

Figure 7 shows the execution of the `sw` instruction during a random iteration of the loop from Figure 6. Any iteration except the first one could be analysed. As usual, the first execution of an instruction should not be used in order to avoid instruction cache (I$) misses.

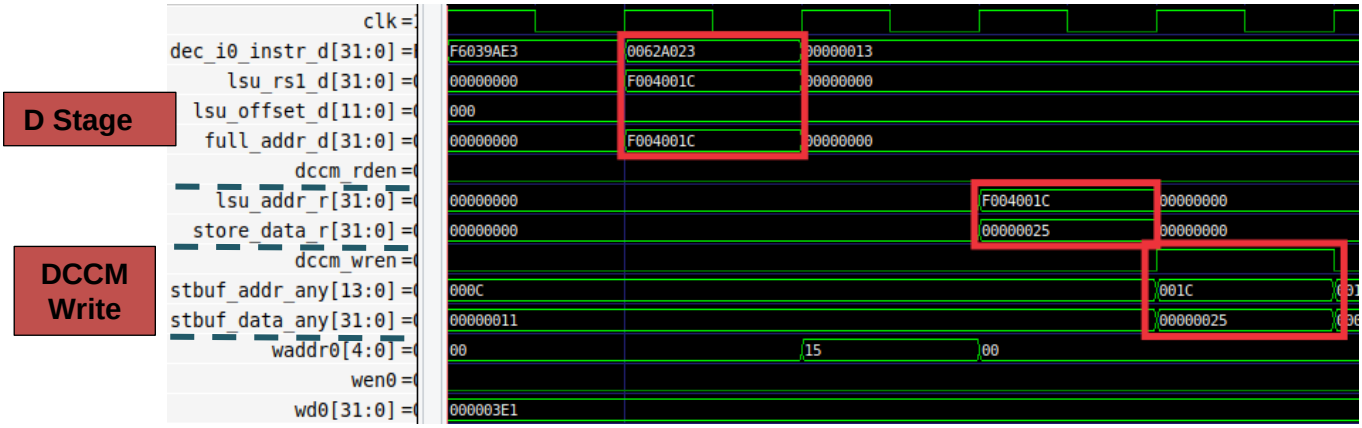


Figure 7 Verilator simulation for the example of Figure 6

Figure 8 shows a high-level view of the VeeR EL2 pipeline while executing the `sw` instruction during the same iteration of the loop. Register `t1` (which holds the value to write to memory) is 0x25 and `t0` (which holds the base address) is 0xF004001C. Thus, `sw` writes the value 0x25 to the DCCM address 0xF004001C. The figure shows the real names used in the Verilog modules of the VeeR EL2 processor. Note that the figure merges the state of the processor in different cycles:

- **Cycle i:** **D Stage:** The store (`sw`) instruction is decoded in the D (Decode) stage, it is assigned to the LSU Pipe and the operands are provided from the instruction's immediate field and from the Register File, which is read in this cycle. Moreover, the effective address is computed at the Adder Unit as before for the load (`lw`).

- **Cycle i+3:** **M Stage:** The second operand (read from register `t1`) is stored in the DCCM, after traversing the Store Buffer.

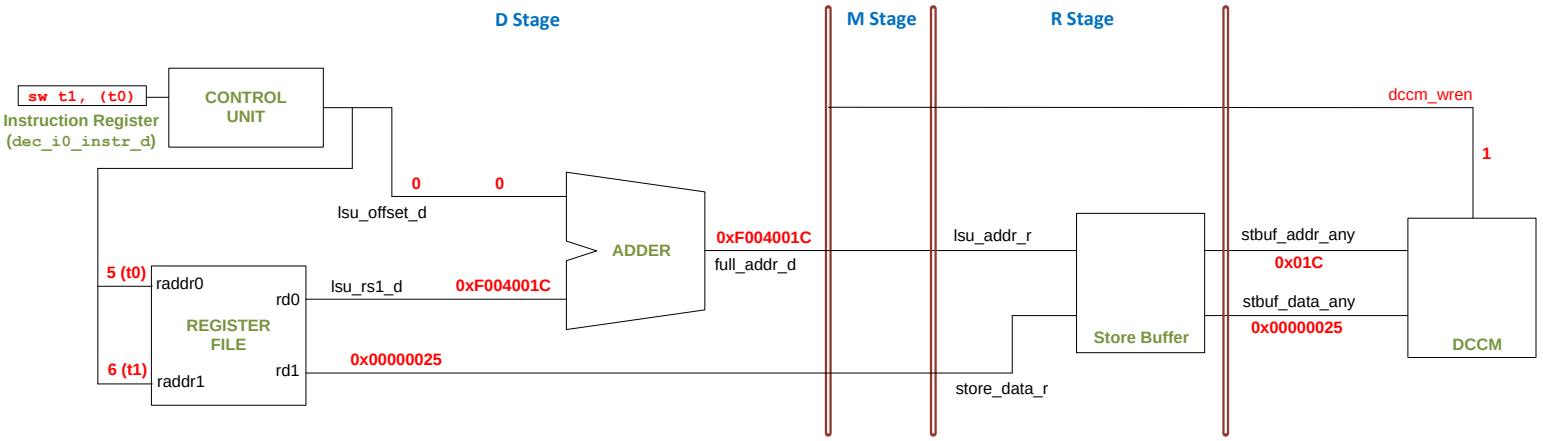


Figure 8 High-level view of the execution of the `sw` instruction in VeeR EL2

Note that the store is not a critical operation in terms of program execution time, so it can be delayed several cycles without impacting performance. In contrast, load instructions can be critical, as they often read a value needed by a subsequent instruction, thus, as mentioned in the previous section, a store-load forwarding path is implemented (not shown in Figure 8), which saves memory accesses and avoids pipeline stalls in case of a data hazard between a store and a subsequent load to the same memory address.

TASK: Replicate the simulation from Figure 7 on your own computer. Follow the steps described in detail in the GSG.

Analyse the waveform from Figure 7 and the diagram from Figure 8 at the same time. The figure includes some signals associated with the pipeline stages, as well as some signals related with the Store Buffer and DCCM read/write. The values highlighted in red correspond to the `sw` instruction as it traverses these stages.

- **Cycle i:** **D Stage:** As explained for the load instruction, signal `dec_i0_instr_d` contains the 32-bit `sw` instruction (0x0062a023). Signal `lsu_rs1_d` contains the base address of the `sw` operation (which in this example is 0xF004001C, as provided by register `t0`), and signal `lsu_offset_d` contains the 12-bit immediate (0x000 in this example) that was extracted from the instruction and subsequently added to the base address. For store instructions, the value read from the second register (in this case `t1`) will eventually be written to memory. Thus, it must be propagated to the subsequent stages. As explained for loads, during this stage the address is computed (`full_addr_d` = `lsu_rs1_d` + `lsu_offset_d` = 0xF004001C).

- **Cycle i+2:** **R Stage – Store Buffer Write:** After two cycles, at the R Stage, the Store Buffer receives the write data and address (`lsu_addr_r`=0xF004001C and `store_data_r`=0x00000025).

- **Cycle i+3:** **R Stage – DCCM Write:** In the following cycle, the DCCM receives the write data and address (`stbuf_addr_any` = 0x01C and `stbuf_data_any` = 0x00000025) from the *Store Buffer*. Note that the Store Buffer only stores the last 12

bits of the address (0x01C), which are enough for writing the DCCM. In this cycle, signal `dccm_wren` asserts, thus the write to the DCCM is performed.

TASK: In simulation, analyse the load instruction that follows the store to verify that the value has been correctly written to the DCCM.

TASK: Extend the basic analysis performed in this section for the `sw` instruction in a similar way as the advanced analysis performed for the `lw` instruction in Section 2.B.

TASK: Analyse unaligned stores to the DCCM, as well as sub-word stores: store byte (`sb`) or *store half-word* (`sh`).

4. Accessing Main Memory

In Sections 2 and 3, we used the DCCM for storing and loading the data. In this section, we analyse non-blocking load instructions that access the Main Memory used on the board. Note that in this case, as opposed to the scenario analysed in Section 2, the VeeR EL2 Core must communicate with the Main Memory through the AXI bus in order to obtain the data requested by the load instruction. Non-blocking loads allow program execution to continue as long as the instructions do not depend on the data read by the load; execution only stops when an instruction is executed that depends on the load. In contrast, blocking loads completely stop the processor until they receive the data read from memory. This means that no other instruction progresses until the load receives its data, even if no dependency exists.

The code in Figure 9 depicts a simple example to illustrate the execution of a `lw` instruction reading Main Memory. The code contains a loop that reads a 12-element array (`lw t3, (t4)`) and accumulates the sum of its elements in register `t6` (`add t6, t3, t6`). As usual, several `nop` operations are inserted to isolate the instructions and make them easier to analyse, and compressed instructions are disabled.

Array D contains 12 words and it is placed in Main Memory by declaring it within section *.data*. The data defined in the *.data* section are placed in Main Memory and not in the DCCM as was done in the program from Figure 2.

```
.globl main

.data
D: .word 3,5,6,8,7,10,12,2,1,4,11,9

.text
main:

li t2, 0x008 # Disable Branch Predictor
csrrs t1, 0x7F9, t2

la t4, D
li t5, 12
li t6, 0x0
INSERT_NOPS_1
```

```
REPEAT:
   lw t3, (t4)
   add t5, t5, -1
   INSERT_NOPS_10
   add t6, t3, t6
   add t4, t4, 4
   INSERT_NOPS_9
   bne  t5, zero, REPEAT    # Repeat the loop

INSERT_NOPS_4

.end
```

Figure 9 Example of blocking `lw` instruction

Folder *[RVfpgaBasysPath]/Labs/Lab13/LW_Instruction_ExtMemory* provides the Catapult project so that you can analyse, simulate, and change the program. If you open the project, build it, and open the disassembly file, you will see that the `lw` instruction is placed at address 0x000002d8, and you can also see the machine code for the instruction (0x000eae03):

```
0x000002d8:      000eae03          lw  t3,0(t4)
```

Loads that access Main Memory follow the same path as the one explained in Section 2 for loads accessing the DDCM in the D Stage. However, the path followed in subsequent stages is quite different, as we also show in that figure. The module that controls Main Memory access through the AXI bus is called **`el2_lsu_bus_intf`** in VeeR EL2. It is responsible for providing the address to Main Memory and, some cycles later, receiving and aligning the requested data and inserting it into the core. Note that an AXI bus is used for communicating with Memory. Also, note that port 1 of the Register File (and not port 0) is used for writing the data.

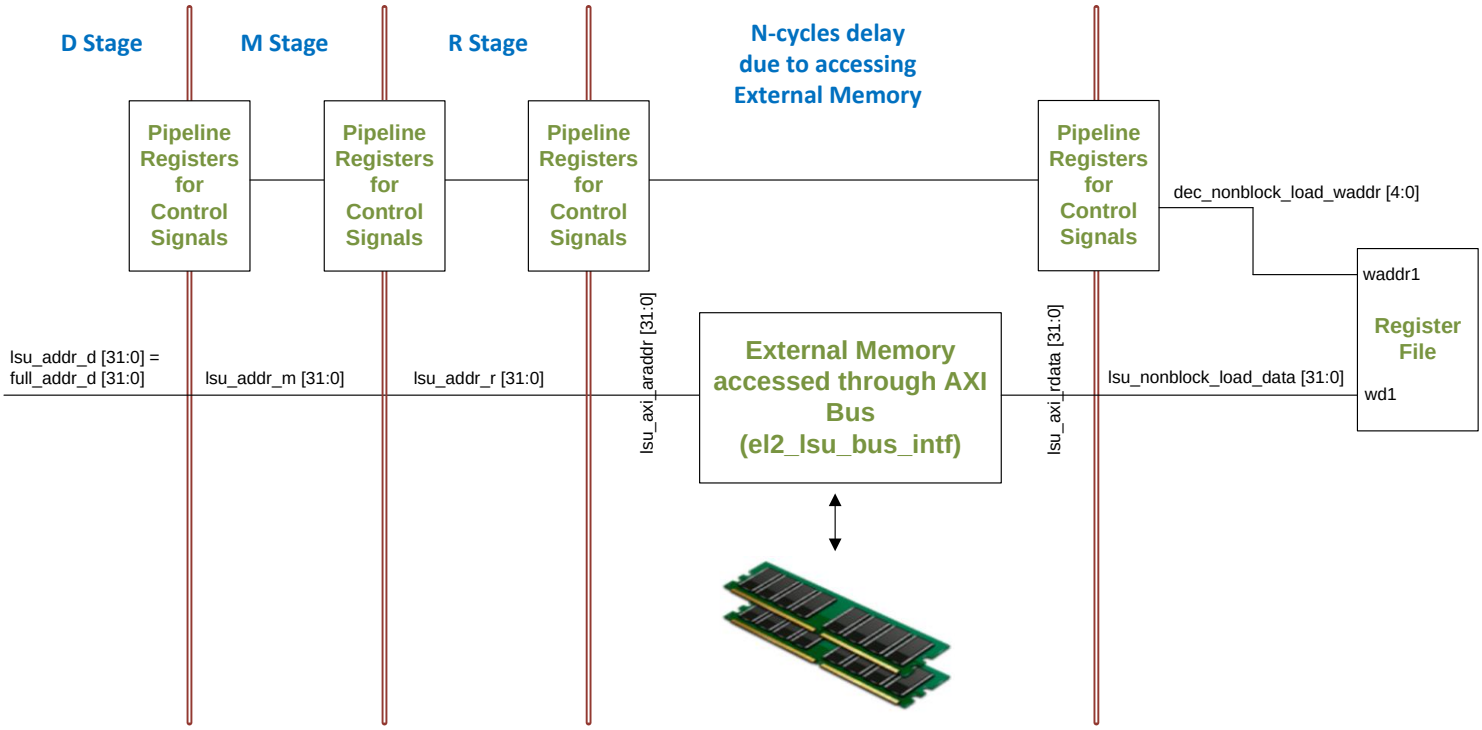


Figure 10. Blocking `lw` instruction accessing Main Memory

Figure 11 shows the execution of the `lw` in a random iteration of the loop of Figure 9, where it reads the value stored in address 0x000007c8 into register `t3`.

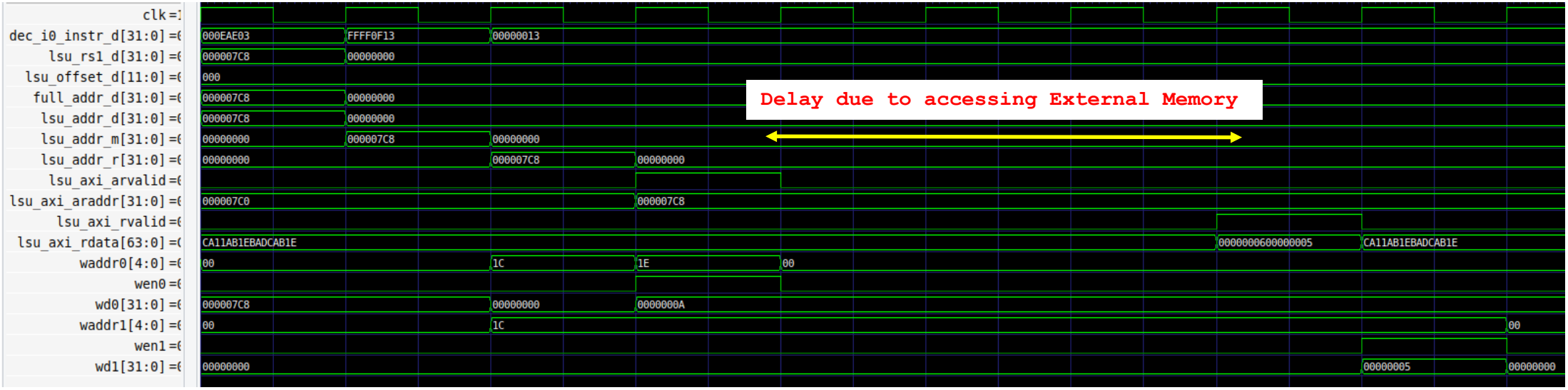


Figure 11. Verilator simulation of the example from Figure 9

TASK: Replicate the simulation from Figure 11 on your own computer.

Analyse the waveform. The figure includes some signals associated with the pipeline stages. Note that the set of signals on the top (`clk` through `full_addr_d`) are the same as those shown in Figure 4. The values highlighted in red correspond to the `lw` instruction as it traverses these stages.

- The address is computed in the D Stage, as explained in Section 2. Signal `full_addr_d[31:0]` contains the *address*, which in the analysed iteration of our example (the one shown in Figure 11) is 0x000007c8.

- This address is propagated through the pipeline stages (D to M, M to R) and then it is sent to Main Memory through the AXI bus using the following signals: `lsu_axi_arvalid` = 1 and `lsu_axi_araddr` = 0x000007c8. Note that the address sent is double-word aligned as 64 bits are read from memory per access. If more than one address is required for the access (due to an unaligned access), multiple addresses are sent and data are returned sequentially through the bus.

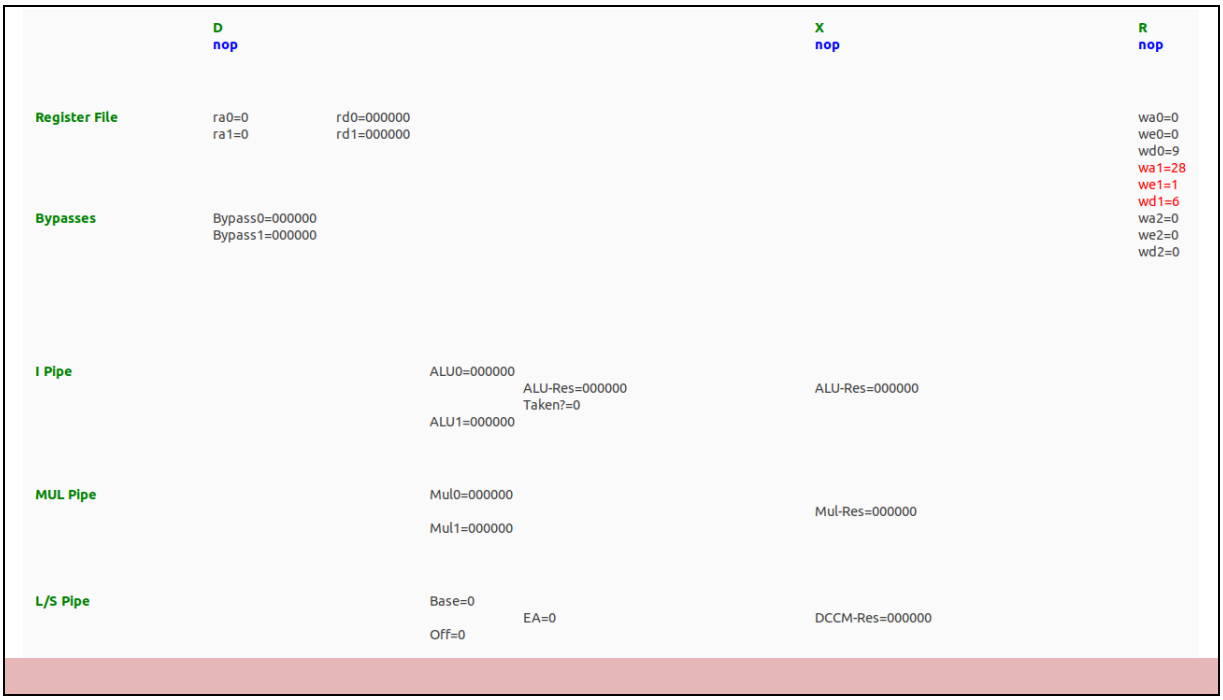
- Some cycles later, main memory returns a 64-bit data through the AXI Bus (`lsu_axi_rdata` = 0x 0000000600000005 and `lsu_axi_rvalid` = 1). This data is buffered within the LSU (module **`el2_lsu_bus_buffer`**).

  TASK: Modify the program from Figure 9 in order to analyse an unaligned load access that needs to send two addresses to the Main Memory through the AXI Bus and then combine the 64-bit data.

- The requested 32-bit data is extracted from the 64-bit data read from memory and inserted in the main pipeline path, and finally written into the Register File in what is called the WB Stage (a substage of the Retire Stage):
  - `wd1=0x00000005`
  - `waddr1=0x1C`

TASK: Simulate the previous program on RVfpgaEL2-Pipeline in order to perform a similar analysis to the one just carried out. Remember that you must insert the control instruction at the point where you want the simulator to stop execution (`and zero, t4, t5`).

For example, this is the VeeR EL2 pipeline when the load is in the WB Stage when the Register File is written using port 1 (`wa1=28, we1=1, wd1=6`). In the previous cycles, you will observe the load instruction flowing through the pipeline.



TASK: It can be interesting to analyse the AXI Bus implementation for accessing the DRAM Controller, for which you can inspect the **`lsu_bus_intf`** module.