



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 SweRV EH1 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 execute *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 DDR main memory available on the Nexys A7 board.

Figure 1 illustrates a high-level view of the microarchitecture of the SweRV EH1 processor. The figure highlights the stages that are relevant in this lab: Decode, DC1-3 (Data Access stages 1-3), Commit, and Writeback.

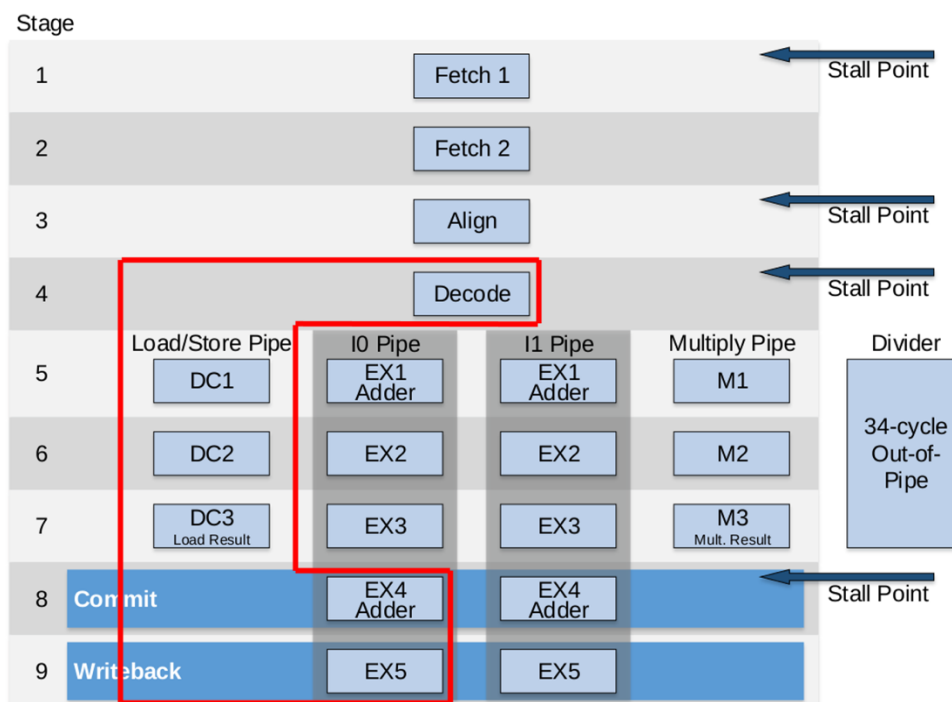


Figure 1 SweRV EH1 core microarchitecture

(figure from https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_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 as explained in the SweRVref document.

```
.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 `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM` provides the PlatformIO project so that you can analyse, simulate, and change the program. Open the project in PlatformIO, build it, and open the disassembly file (located at `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis`). In that file, locate the second `lw` instruction, which is at address `0x0000014c`. Notice the machine code for the instruction (`0x0042a303`):

```
0x0000014c:    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, in the Getting Started Guide (GSG) and in previous labs, we have been using the DDR memory available on the Nexys A7 board for storing both the instructions and the data from our program. However, accessing that external 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) available in SweRV EH1 for storing the program data.

The DCCM is a local memory tightly coupled to the core. It provides low-latency access and SECDED ECC protection¹. Its size is set as an argument at build time of the core, ranging from 4 KiB to 512 KiB (64 KiB is the default). In Lab 20 we will analyse the DCCM and ICCM in more detail; in this lab, we simply use it to simplify analysis of the load/store instructions. Note that, this way, everything happens inside the SweRV EH1 Core Complex (Figure 3), where both the SweRV EH1 pipeline and the DCCM are placed (highlighted in red).

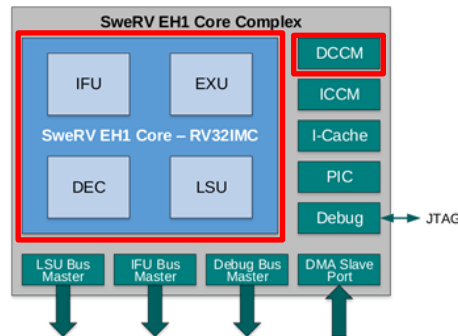


Figure 3 SweRV EH1 Core Complex

The code in Figure 2 defines an *ad-hoc* section called `.midccm` to allocate space in the DCCM. By default, the address space of the DCCM starts at 0xF0040000 in our default RVfpga System. The linker script provided with this project (available at: `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/ld/link.lds`) will take care of the proper address assignments. This linker script is used by including the following command in file `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/platformio.ini`:

```
board_build.ldscript = ld/link.lds
```

A. Basic analysis of the `lw` instruction

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: `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/scriptLoad.tcl`. Note that all iterations are the same: the first load reads the DCCM's first data word (2) into `t1` (x6); the second load reads the DCCM's second data word (8) into the same register (`t1`).

¹ See the *RISC-V SweRV™ EH1 Programmer's Reference Manual* for more details.

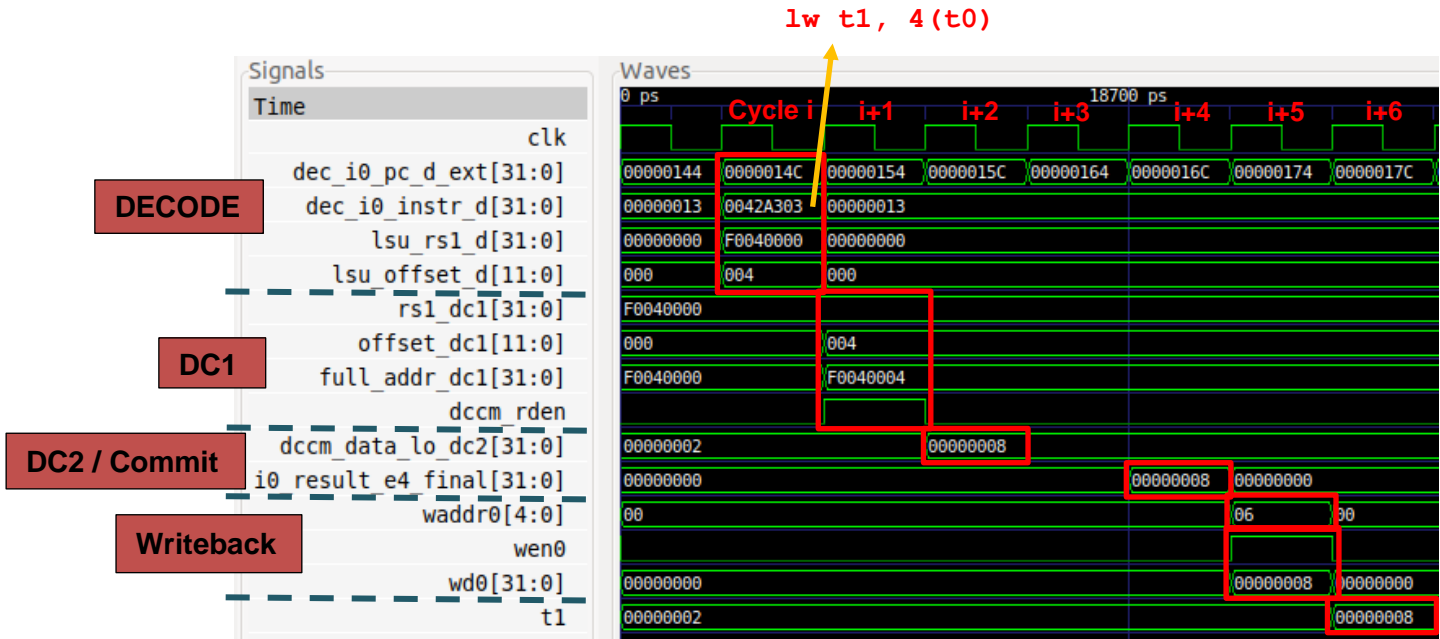


Figure 4. Verilator simulation for example program in Figure 2

Figure 5 shows a high-level view of the SweRV EH1 pipeline during the execution of the second `lw` instruction. Note that the figure merges the state of the processor in different cycles:

- **Cycle i:** The instruction is decoded and the register file is read.
- **Cycle i+1:** The effective address is computed using the adder.
- **Cycle i+2:** The DDCM is read using the address computed in the previous stage.
- **Cycle i+5:** The value read from memory is written to the Register File.

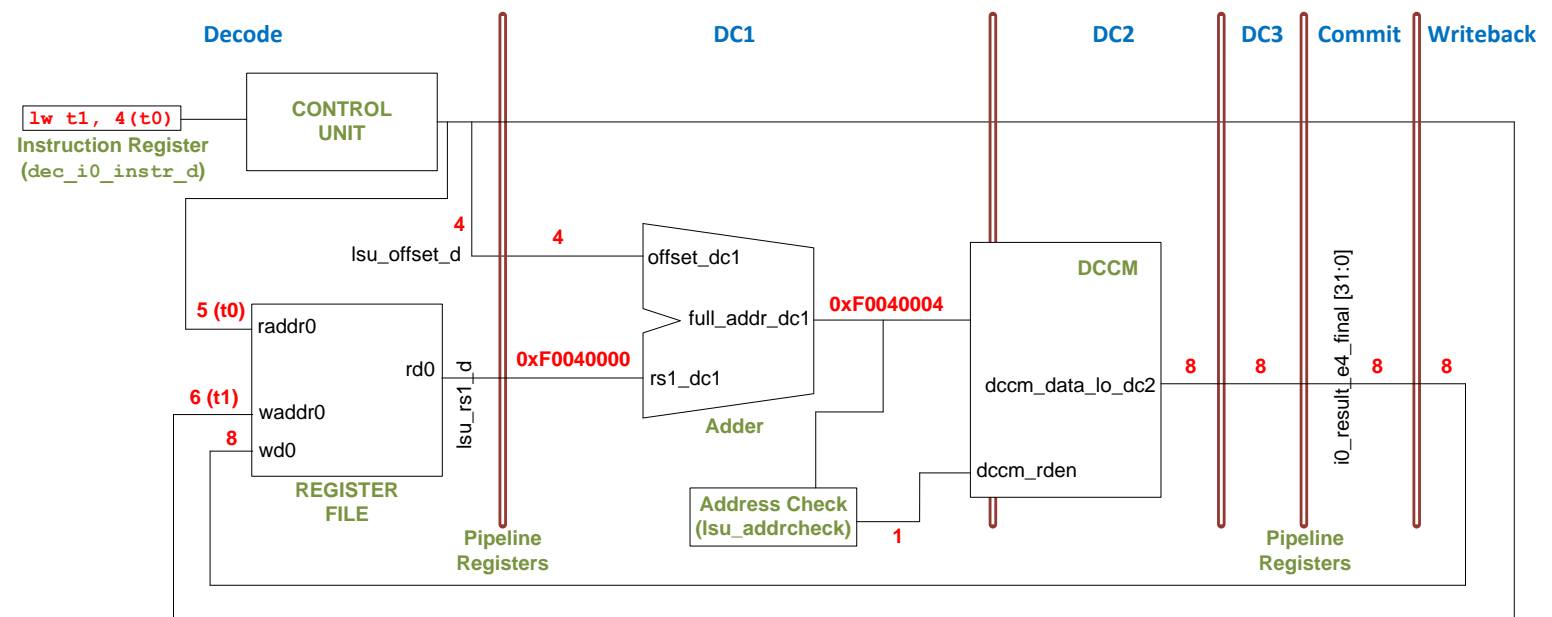



Figure 5. High-level view of the `lw` instruction executing in the SweRV EH1 pipeline

TASK: Replicate the simulation from Figure 4 on your own computer. 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/Lab13/LW_Instruction_DCCM.
- Correct the path to the RVfpga simulation binary (*Vrvfpgasim*) in file *platformio.ini*.
- Generate the simulation trace with Verilator (Generate Trace).
- Open the trace using GTKWave.
- Use file *scriptLoad.tcl* (provided at
[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM) to open the same signals as the ones shown in Figure 4. For that purpose, on GTKWave, click on *File → Read Tcl Script File* and select the *scriptLoad.tcl* file.
- Click on *Zoom In* () several times and move to 18600ps.

Analyse the waveform from Figure 4 and the diagram from Figure 5 at the same time. The figures include some signals associated with the Decode, DC1-3, Commit, and Writeback stages. The values highlighted in red correspond to the second *lw* instruction as it traverses these stages.

- **Cycle i: Decode:** *dec_i0_pc_d_ext* holds the address of the *lw* instruction (0x0000014C) and signal *dec_i0_instr_d* contains the 32 bits of the *lw* machine instruction (0x0042a303).

During this stage, the **control signals are generated**. Moreover, the **operands for computing the load effective address are obtained**: signal *lsu_rsl_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).

- **Cycle i+1: DC1:** The address is computed using an adder located inside module **lsu_lsc_ctl**. The address is the base address (*rs1_dc1* = 0xF0040000) plus the sign-extended offset (*offset_dc1* = 0x00000004); the final address is *full_addr_dc1* = 0xF0040004. This address is checked (Address Check) to determine the memory region of the access (DCCM, PIC, or external 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_dc1*) and the enable signal (*dccm_rden*) are provided to the DCCM, which is read in the next cycle.
- **Cycle i+2: DC2:** the DCCM is read and the data is placed in *dccm_data_lo_dc2* = 0x8, which is propagated to the next stage.
- **Cycle i+3: DC3:** the data read from the DCCM is propagated to the next stage.
- **Cycle i+4: Commit:** the data read from the DCCM (signal *i0_result_e4_final* = 0x8) is propagated to the next stage.
- **Cycle i+5: Writeback:** Finally, the value read from memory is **written back** to the register file through signal *wd0* = 0x8. Given that *wen0* = 1, the value is written at the end of that cycle into register x6 (*waddr0* = 0x6). You can observe that, in the following cycle (last cycle shown in Figure 4), register x6 (also called *t1*) contains the new value

($t1 = 0x8$). Note that the signal $t1$ shown in the waveform is an alias defined in the *.tcl* script for signal *dout*.

B. Advanced analysis of the *lw* instruction

In this section, we analyse the stages traversed by the *lw* instruction in more detail. Figure 6 shows a diagram of the main elements that the load instruction from our example traverses during its execution along the Load/Store pipe (DC1, DC2, and DC3 stages). You may need to zoom into the figure to be able to see the details. The black blocks labelled *LOGIC* in the figure contain various blocks, such as multiplexers and logic gates. For the sake of simplicity, only some of the block's interface signals are included in the figure.

The Decode and Writeback stages are identical as the ones shown for A-L instructions (see Figure 6 in Lab 12). However, we point out a few details of the Decode stage. Recall that in the Decode stage, control signals are generated and instructions and operands are scheduled to the proper pipes:

- The load's immediate offset is in signal *lsu_offset_d*.
- The load's base address is in signal *exu_lsu_rsl_d*. (This signal is produced from a 4:1 multiplexer (shown in Figure 4 of Lab 11), and propagated to the DC1 Stage after traversing some logic.)
- The signals for load/store instructions are in *lsu_p*, a new control signal packet shown in Figure 6.

Similarly to Decode, the Commit Stage was also analysed in Lab 12, but we now include the input to the final 3:1 multiplexer related to load instructions (*lsu_result_corr_dc4*), which was omitted in Lab 12 for the sake of simplicity. Remember that the output of this 3:1 multiplexer is *i0_result_e4_final[31:0]*, as shown in Figure 6. Moreover, we only focus on Way 0 in this lab, but a load/store can execute through either way of the two-way superscalar processor. Note, however, that there is only one L/S (Load/Store) Pipe. Thus, Way 1 also has a 3:1 multiplexer (whose output is *i1_result_e4_final[31:0]*) and one of its inputs is *lsu_result_corr_dc4*, as shown in Figure 4 of Lab 11.

TASK: Extend the simulation from Figure 4 to include the signals shown in Figure 6, which are explained below. A *.tcl* file that you can use is provided at: *[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/scriptLoadExtended.tcl*

TASK: Locate the modules and signals from Figure 6 in the Verilog files of the SweRV EH1 processor.

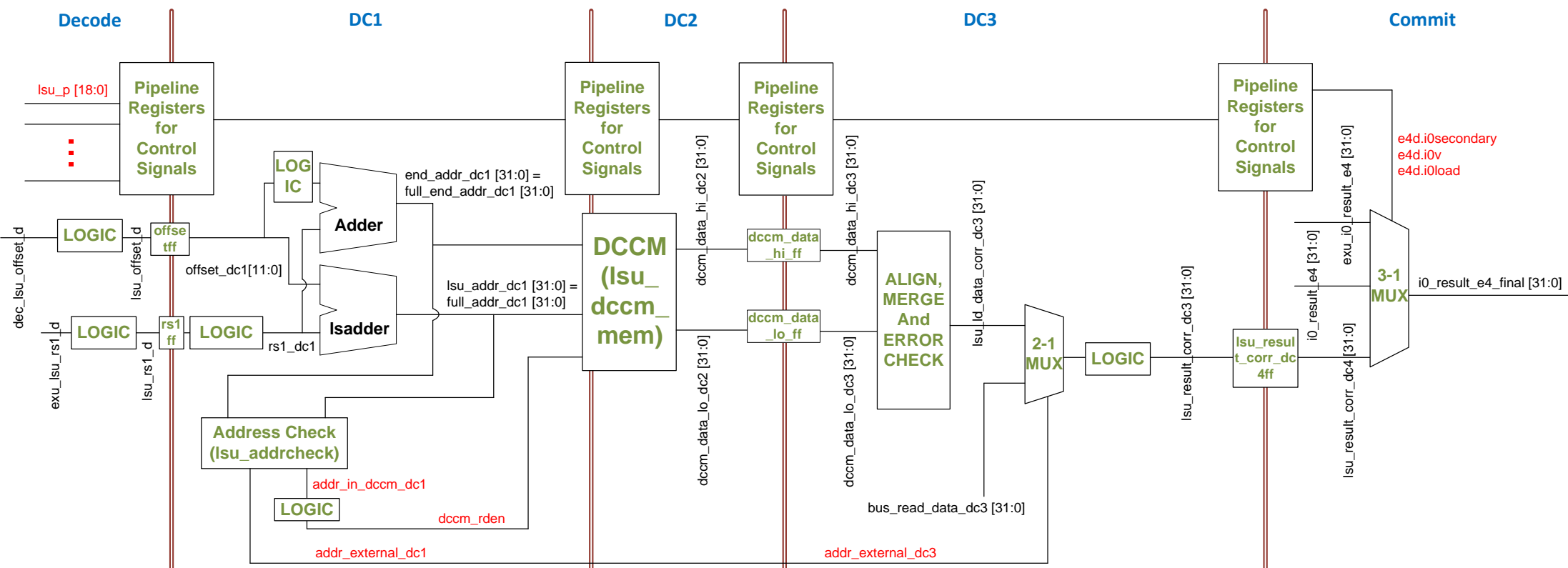


Figure 6 Main elements traversed by load instructions in the Load/Store Pipe

i. Decode Stage

General details of the Decode Stage were already analysed in Labs 11 and 12. Remember that the Decode Stage is responsible for two main tasks:

- **Decode** the instructions and generate **control signals**.
- **Distribute** the instructions to the appropriate pipes and provide the **input operands**.

Decode the instructions and generate control signals:

In addition to other control signals structures already analysed in Labs 11 and 12, an additional structure, `lsu_pkt_t`, contains load/store instruction signals. As usual, this structure is defined in file `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv`. Signal `lsu_p` is an example of a signal of this type that is propagated from the Decode Stage through the Load/Store Pipe stages.

This signal encapsulates some relevant information for memory read/write:

- o **Bit 0** (`valid`) is set to 1 if the operation is valid.
- o **Bit 12** (`unsign`) is set to 1 when the data to be read/written is unsigned.
- o **Bit 13** (`store`) is set to 1 if the operation is a store (`sb`, `sh`, `sw`...).
- o **Bit 14** (`load`) is set to 1 if the operation is a load (`lb`, `lh`, `lw`...).
- o **Bits 15-18** codify the size of the access (byte, halfword, word, doubleword).

TASK: Include signal `lsu_p` in the simulation from Figure 4 and analyse its bits according to this description.

Distribute the instructions to the appropriate pipes and provide the input operands:

As explained in Lab 11, the SweRV EH1 processor includes several pipes for executing instructions. In the Decode Stage the instructions, once decoded, must be scheduled through the appropriate pipes. In the program that we are analysing in this lab (Figure 2), the `lw` instruction is sent for execution to the LSU Pipe (stages DC1-3). Specifically, `exu_lsu_rs1_d` is the value held in the base register. Signal `dec_lsu_offset_d` is the 12-bit signed immediate offset, which is extracted from the instruction and sent to the DC1 stage.

TASK: Analyse in the Verilog code the path followed by the two inputs to the LSU (`exu_lsu_rs1_d` and `dec_lsu_offset_d`) from the sources where they are obtained. Several modules are involved in this process: **dec**, **exu**, **lsu**.

ii. DC1 Stage

In the DC1 Stage, `rs1_dc1` (base address, propagated from Decode) and `offset_dc1` (offset, propagated from Decode) are added in module **lsadder** to compute the *main effective address* (signal `full_addr_dc1[31:0]`, which is assigned to `lsu_addr_dc1[31:0]`). This is the memory address to be read.

In addition to the address to be read, the *end address* (`end_addr_dc1[31:0]`) is also computed in another adder (we should highlight that this second adder was not shown in Figure 5 nor in Figure 4 of Lab 11 for the sake of simplicity). This is the address of the last byte that must be read from memory. This address is used to handle unaligned accesses and sub-word (byte, half-word) accesses.

TASK: Analyse the implementation of the two adders from the DC1 stage, which are instantiated in module `lsu_lsc_ctl`. We provide guidance in Figure 7 below by showing the implementation of these adders.

```

185 // generate the ls address
186 // need to refine this is memory is only 128KB
187 rvlsadder lsadder (.rs1(rs1_dc1[31:0]),
188                  .offset(offset_dc1[11:0]),
189                  .dout(full_addr_dc1[31:0])
190                  );

```

```

199 // Calculate start/end address for load/store
200 assign addr_offset_dc1[2:0] = ((3{lsu_pkt_dc1.half}) & 3'b01) | ((3{lsu_pkt_dc1.word}) & 3'b11) | ((3{lsu_pkt_dc1.dword}) & 3'b111);
201 assign end_addr_offset_dc1[12:0] = {offset_dc1[11], offset_dc1[11:0]} + {9'b0, addr_offset_dc1[2:0]};
202 assign full_end_addr_dc1[31:0] = rs1_dc1[31:0] + {{19{end_addr_offset_dc1[12]}}, end_addr_offset_dc1[12:0]};
203 assign end_addr_dc1[31:0] = full_end_addr_dc1[31:0];

```

Figure 7. Verilog for adders in DC1 stage from file `lsu_lsc_ctl.sv`.

For example, a load word (`lw`) to the address starting at `0xF0040003` would have: `full_addr_dc1=0xF0040003` and `end_addr_dc1=0xF0040006` (see Figure 8). This way, the LSU Pipe can extract the word from the read bundle, which consists of two words that begin at addresses that are a multiple of four (in this case `0xF0040000` and `0xF0040004`).

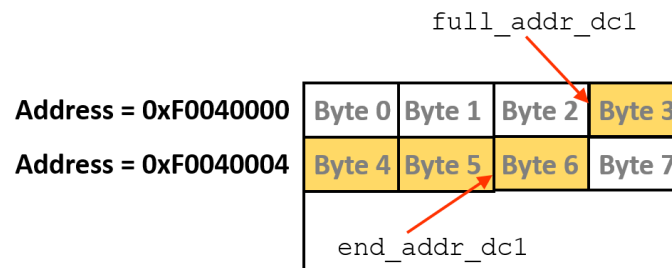


Figure 8. Example of a `lw` instruction to address `0xF0040003`

The two addresses (`lsu_addr_dc1[31:0]` and `end_addr_dc1[31:0]`) are sent to the Data Memory (in our example, the DCCM), which will be accessed in the next cycle.

TASK: In the program from Figure 2, try different access sizes (byte, half-word) and unaligned accesses. To do so, change the offset or the access type from `lw` to `lb` (*load byte*) or `lh` (*load half-word*). For example, if you change the offset from 4 to 3, the load word instruction performs an unaligned access to the 32-bits starting at address `0xF0040003`, as shown in Figure 8. Analyse the value of signals `lsu_addr_dc1[31:0]` (or `full_addr_dc1[31:0]`) and `end_addr_dc1[31:0]` under these different situations.

In Lab 20 we analyse this situation from the internals of the DCCM.

In addition to the address calculation, the DC1 stage performs an address range check in module **lsu_addrcheck** (see Figure 9) to determine the target memory for the access, which is the DCCM in our example.

```
// Module to generate the memory map of the address
lsu_addrcheck addrcheck (
    .start_addr_dc1(full_addr_dc1[31:0]),
    .end_addr_dc1(full_end_addr_dc1[31:0]),
    .*
);
```

Figure 9. Check range and location of memory address

As a result of the address check, it is determined which memory must be accessed: the DCCM, the PIC, or the External DDR Memory (see Figure 10).

```
43 output logic    addr_in_dccm_dc1,          // address in dccm
44 output logic    addr_in_pic_dc1,          // address in pic
45 output logic    addr_external_dc1,        // address in external
```

Figure 10. Addresses for each memory unit

In our example, the DCCM read enable signal goes high (`addr_in_dccm_dc1 = 1`). This signal traverses some logic and then it is provided to the DCCM (signal `dccm_rden`) to enable/disable the access (in our example, to enable it). Signal `addr_external_dc1`, which is 1 when the External DDR Memory must be enabled and 0 otherwise, is propagated and used by the DC3 Stage, as shown in Figure 6.

iii. DC2 Stage

If DCCM reading is enabled (`dccm_rden = 1`) the data is read during this stage. Note that two 32-bit values are read (`dccm_data_lo_dc2[31:0]` and `dccm_data_hi_dc2[31:0]`), as the data access could be unaligned and thus spread across two words (such as in the example from Figure 8).

TASK: In the program from Figure 2, compare the value of signals `dccm_data_lo_dc2[31:0]` and `dccm_data_hi_dc2[31:0]` when doing a `lw` to address `0xF0040004` and to address `0xF0040003`.

iv. DC3 Stage

The two 32-bit data values from the DCCM are propagated from DC2 (signals `dccm_data_lo_dc2[31:0]` and `dccm_data_hi_dc2[31:0]`) to DC3 (signals `dccm_data_lo_dc3[31:0]` and `dccm_data_hi_dc3[31:0]`). For aligned accesses, such as the one from our example, both signals are equal and only `dccm_data_lo_dc3[31:0]` is used.

In the DC3 stage, the two words read in the previous cycle (`dccm_data_lo_dc3[31:0]` and `dccm_data_hi_dc3[31:0]`) go through logic that performs several tasks:

- **Error Checking:** Data is checked for errors using ECC.
- **Handling Load/Store Hazards:** If a store instruction to the same address is still executing, data is forwarded from the store instruction to the load instruction instead of being read from memory. We will analyse this situation in Lab 15.
- **Alignment:** The requested data is aligned.

As a result of all these processes, the final data is provided in signal `lsu_ld_data_corr_dc3[31:0]`.

TASK: Analyse the Align, Merge, and Error Check logic used in the Verilog code in modules `lsu_dccm_ctl` and `lsu_ecc`.

TASK: In the program from Figure 2, compare the value of signal `lsu_result_corr_dc3[31:0]` when doing a `lw` to address `0xF0040004` and to address `0xF0040003`.

After this logic that performs error-checking, load/store hazard handling, and alignment, a 2:1 multiplexer selects between data from the DCCM (`lsu_ld_data_corr_dc3[31:0]`) or DDR memory (`bus_read_data_dc3[31:0]`). This multiplexer is controlled by signal `addr_external_dc3`, which was generated in module `lsu_addrcheck` in the DC1 stage (signal `addr_external_dc1`).

TASK: Analyse in the Verilog code how signal `addr_external_dc1` was computed in the DC1 stage in module `lsu_addrcheck`.

The output of this 2:1 multiplexer (`lsu_result_corr_dc3[31:0]`) is propagated to the Commit Stage.

v. Commit Stage

In the Commit stage, a 3:1 multiplexer selects the read data (`i0_result_e4_final[31:0]`) to be sent to the Writeback Stage (see Figure 6). This 3:1 multiplexer could also select the output of the ALU, as was already explained in Labs 11 and 12, and the output of the Secondary ALU, as we will analyse in Lab 15.

vi. Writeback Stage

This stage was already explained in Labs 11 and 12 and thus it is not shown in Figure 6, where the result of the addition was written to the destination register. In this case, this stage writes the DCCM data to the destination register.

3. THE `sw` INSTRUCTION ACCESSING A LOW LATENCY MEMORY

In this section we use the code shown in Figure 11 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 (0xF0040000 – 0xF004FFFF). Each `sw` is followed by a `lw` that checks that the correct value was stored. As usual, `nops` 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 as explained in the SweRVref document. Also, as in the example from the previous section, we use the DCCM for storing and loading data.

```
.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_2

REPEAT:
    sw t1, (t0)
    INSERT_NOPS_10
    INSERT_NOPS_4
    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 11 Example code with `sw` instruction

Folder `[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM` provides the PlatformIO project so that you can analyse, simulate and change the program. Open the project in PlatformIO, build it, and open the disassembly file (available at `[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis`). You will see that the `sw` instruction is placed at address 0x00000110, and you can also see the machine code for the instruction (0x0062a023):

```
0x00000110:      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 12 shows the execution of the `sw` instruction during the fourth iteration of the loop from Figure 11. 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.

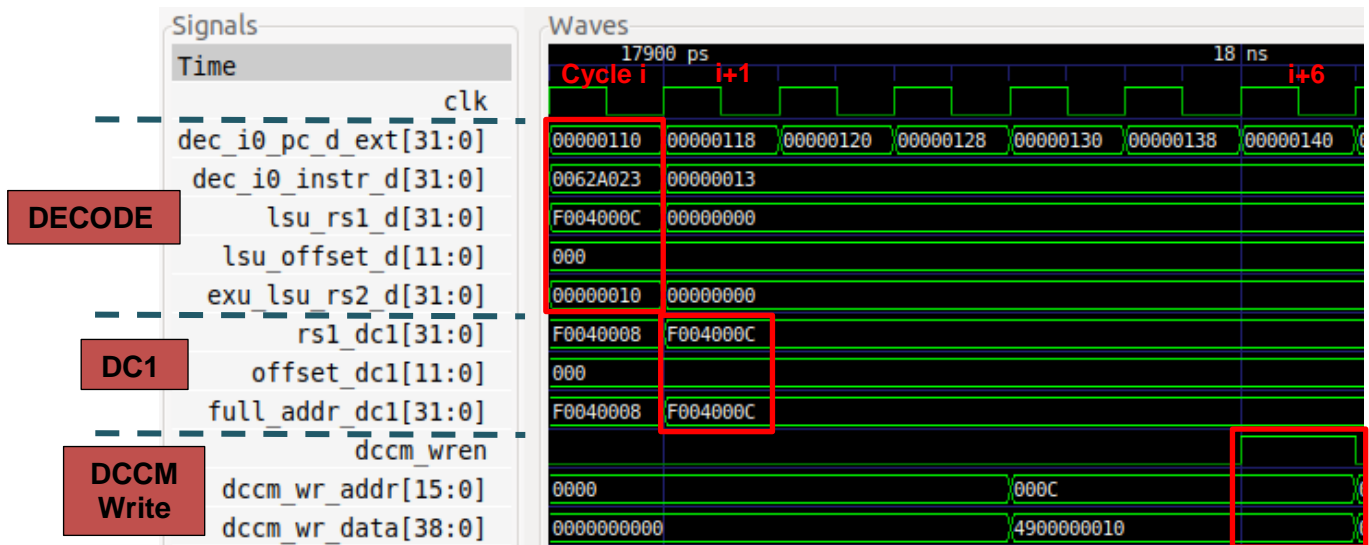


Figure 12 Verilator simulation for the example of Figure 11

Figure 13 shows a high-level view of the SweRV EH1 pipeline while executing the `sw` instruction during the fourth iteration of the loop from Figure 11. Register `t1` (which holds the value to write to memory) is `0x10` and `t0` (which holds the base address) is `0xF004000C`. Thus, `sw` writes the value `0x10` to the DCCM address `0xF004000C`. The figure shows the real names used in the Verilog modules of the SweRV EH1 processor. Note that the figure merges the state of the processor in different cycles:

- **Cycle i:** The `store` instruction is decoded at the Decode stage, it is assigned to the LSU Pipe and the operands are provided, in this case from the instruction's immediate field and from the Register File, which is read in this cycle.
- **Cycle i+1:** The effective address is computed at the Adder Unit as explained for the `load`. Note that only the `lsadder` shown in Figure 6 is included in the figure for the sake of simplicity.
- **Cycle i+6:** The second operand (read from register `t1`) is stored in the DCCM, after traversing the Store Buffer, which we explain in the Appendix.

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 13), 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. We analyse this situation in Lab 15.

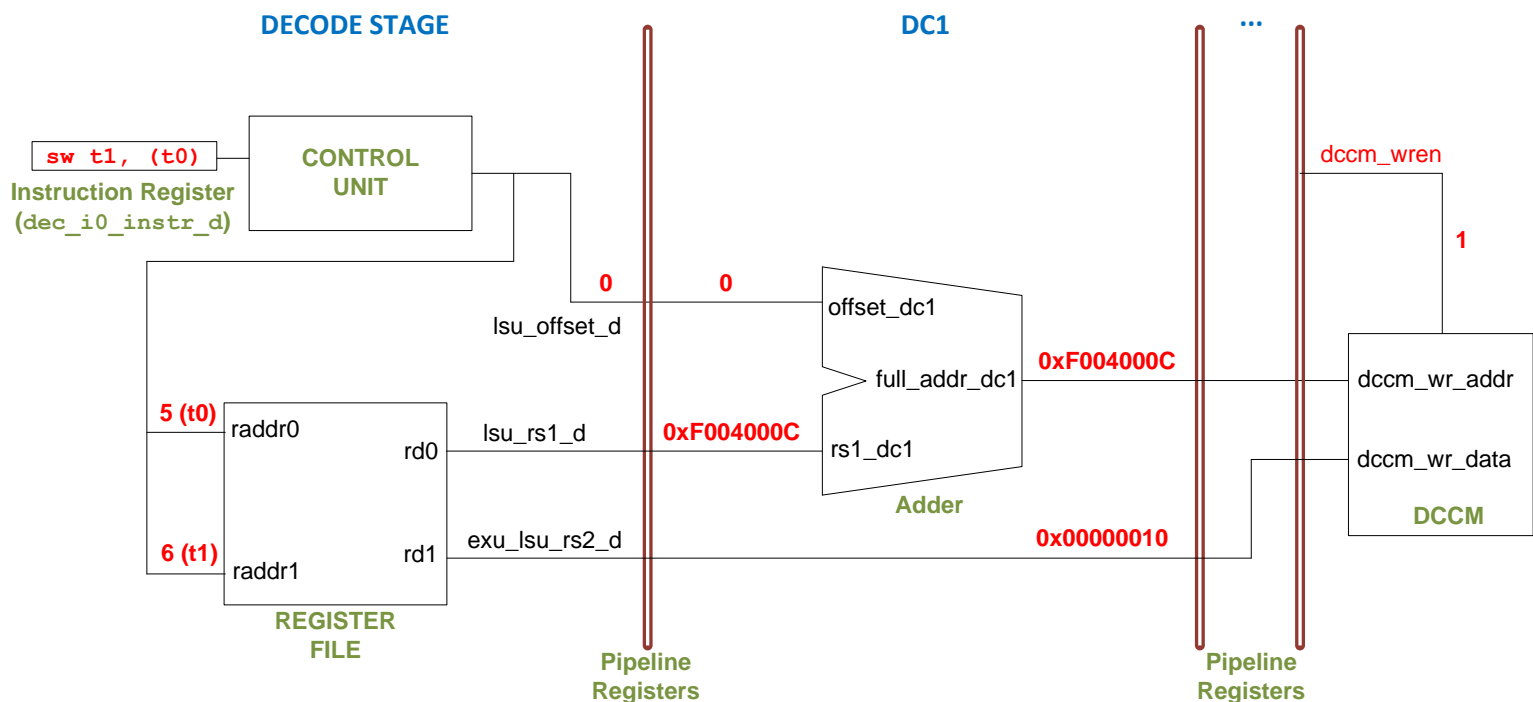



Figure 13 High-level view of the execution of the `sw` instruction in SweRV EH1

TASK: Replicate the simulation from Figure 12 on your own computer. Follow the next steps (as described in detail in Section 7 of the GSG):

- If necessary, generate the simulation binary (*Vrvfpgasim*).
- Open in PlatformIO the project provided at:
[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM.
- Update the path to the RVfpga simulation binary (*Vrvfpgasim*) in file *platformio.ini*.
- Generate the simulation trace with Verilator (Generate Trace).
- Open the trace on GTKWave.
- Use file *scriptStore.tcl* (provided at
[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM/) to display 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 *scriptStore.tcl* file.
- Click on *Zoom In* () several times and move to 17900ps.

Analyse the waveform from Figure 12 and the diagram from Figure 13 at the same time. The figure includes some signals associated with the Decode and DC1 stages, as well as some signals related with the DCCM write, which happens several cycles later. The values highlighted in red correspond to the `sw` instruction as it traverses these stages.

- **Cycle i: Decode:** As explained for the load instruction, signal `dec_i0_pc_d_ext` contains the address of the `sw` instruction (0x00000110) and 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 0xF004000C, 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 (`exu_lsu_rs2_d = 0x10`). Thus, it must be propagated to the subsequent stages.

- **Cycle i+1: DC1:** As explained for loads, during this stage the address is computed (`full_addr_dc1 = rs1_dc1 + offset_dc1 = 0xF004000C`).
- **Cycle i+6: DCCM Write:** After five cycles, the DCCM receives the write data and address (`dccm_wr_addr=0x000C` and `dccm_wr_data=0x4900000010`) from the *Store Buffer*. Note that the DCCM only receives the last 16 bits of the address (`0x000C`), because its actual size is 64 KiB in our configuration (see file `common_defines.vh`) and 16 bits are enough for addressing 2^{16} bytes. The data has been prepended with some ECC bits (`0x49`). When signal `dccm_wren` asserts (in cycle i+6 in Figure 12) the write to the DCCM completes.

APPENDIX A – OPERATION OF THE STORE BUFFER: Appendix A explains the *Store Buffer*, which is an important structure that temporarily keeps the value and address that must be written into memory by the store instruction.

TASK: Analyse in the simulation the load instruction that follows the store to verify that the value has been correctly written to the DCCM. You will need to add some of the signals from Figure 4 and Figure 6 to analyse the load.

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 EXTERNAL MEMORY

In Sections 2 and 3 we used the DCCM for storing and loading the data. In this section, we analyse load instructions that access the External Memory available on the Nexys A7. Note that in this case (see Figure 14), as opposed to the scenario analysed in Section 2 (see Figure 3), the SweRV EH1 Core must communicate with the External Memory through the AXI bus in order to obtain the data requested by the load instruction.

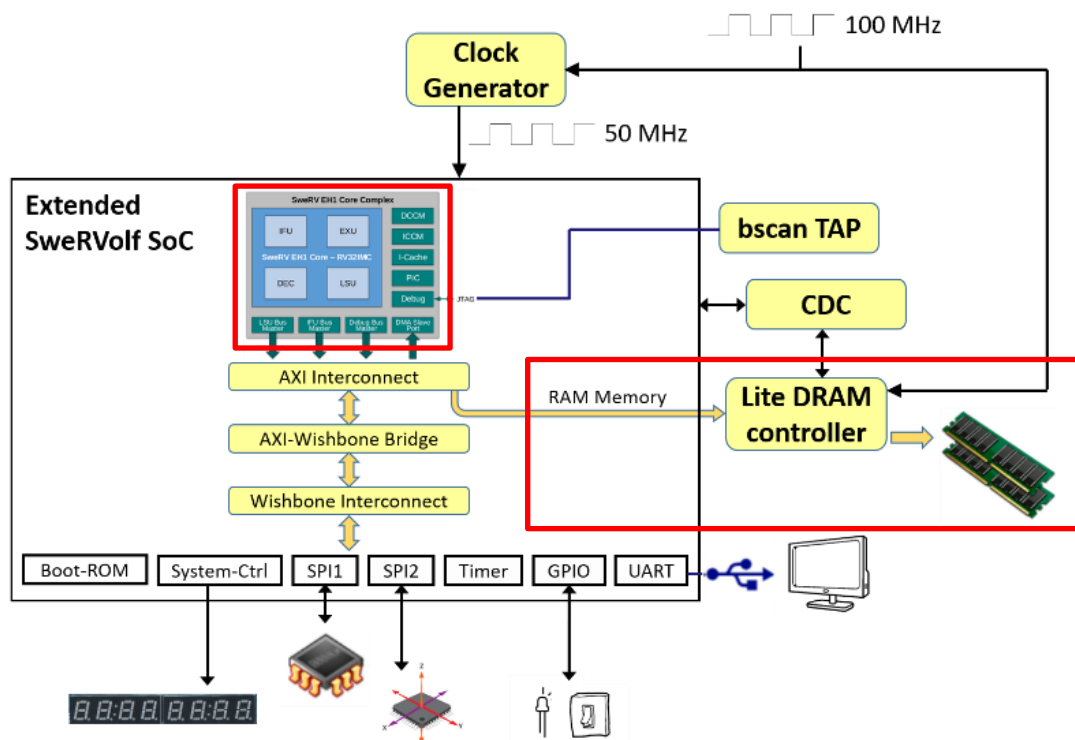


Figure 14. RVfpgaNexys

We analyse both blocking and non-blocking accesses. 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. In contrast, 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 these two scenarios, the data read from memory follows different paths through the pipeline; in this lab we analyse the first one (blocking loads) and in the next lab (Lab 14) we analyse the second case (non-blocking loads) in the context of structural hazards.

The code in Figure 15 depicts a simple example to illustrate the execution of a `lw` instruction reading the external DDR 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.

Vector D contains 12 words and it is placed in Main Memory. For that aim, the array is declared within section `.data` and the usual linker script is employed for the project (available at `~/platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/link.lds`). This way the data defined in the `.data` section are placed in the external memory and not in the DCCM as was done in the program from Figure 2.

By default, load instructions are non-blocking in SweRV EH1. If we want load instructions to be blocking, we must include the next two instructions at the beginning of the assembly code that we are going to analyse, shown in Figure 15 (refer to Section 2.C of the SweRVref document for more explanations on core features enabling/disabling):

```
li t2, 0x020
csrrs t1, 0x7F9, t2
```

```
.globl main

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

.text
main:

    li t2, 0x020
    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 15 Example of blocking lw instruction

Folder `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory` provides the PlatformIO project so that you can analyse, simulate, and change the program. If you open the project in PlatformIO, build it, and open the disassembly file (available at `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory/.pio/build/swervolf_nexys/firmware.dis`), you will see that the `lw` instruction is placed at address `0x000000f4`, and you can also see the machine code for the instruction (`0x000eae03`):

```
0x000000f4:    000eae03        lw    t3,0(t4)
```

Blocking loads accessing the External DDR2 Memory follow almost the same path explained in Section 2 for loads accessing the DDCM, as we will show in Figure 16. However, there is an important difference: during some cycles, the processor is stalled waiting for the data provided by the External Memory; then, when the requested data is received, instructions can continue executing.

The module that controls External Memory access through the AXI bus is called **lsu_bus_intf** in SweRV EH1. It is responsible for providing the address to the Lite DRAM controller and, some cycles later, receive and align the requested data and insert it into the core in the DC3 stage. Note that an AXI bus is used for communicating with the DDR2 External Memory. In this example (Figure 15), a 2:1 multiplexer at the DC3 stage, which was also included in Figure 6, selects the input coming from the External Memory (i.e. `lsu_result_corr_dc3 = bus_read_data_dc3`), instead of the input coming from the DDCM (`lsu_ld_data_corr_dc3`) that was selected in the example from Figure 2. As for the 3:1 multiplexer at the Commit stage, it selects the same input as in the example from Figure 2 (i.e. `i0_result_e4_final = lsu_result_corr_dc4`).

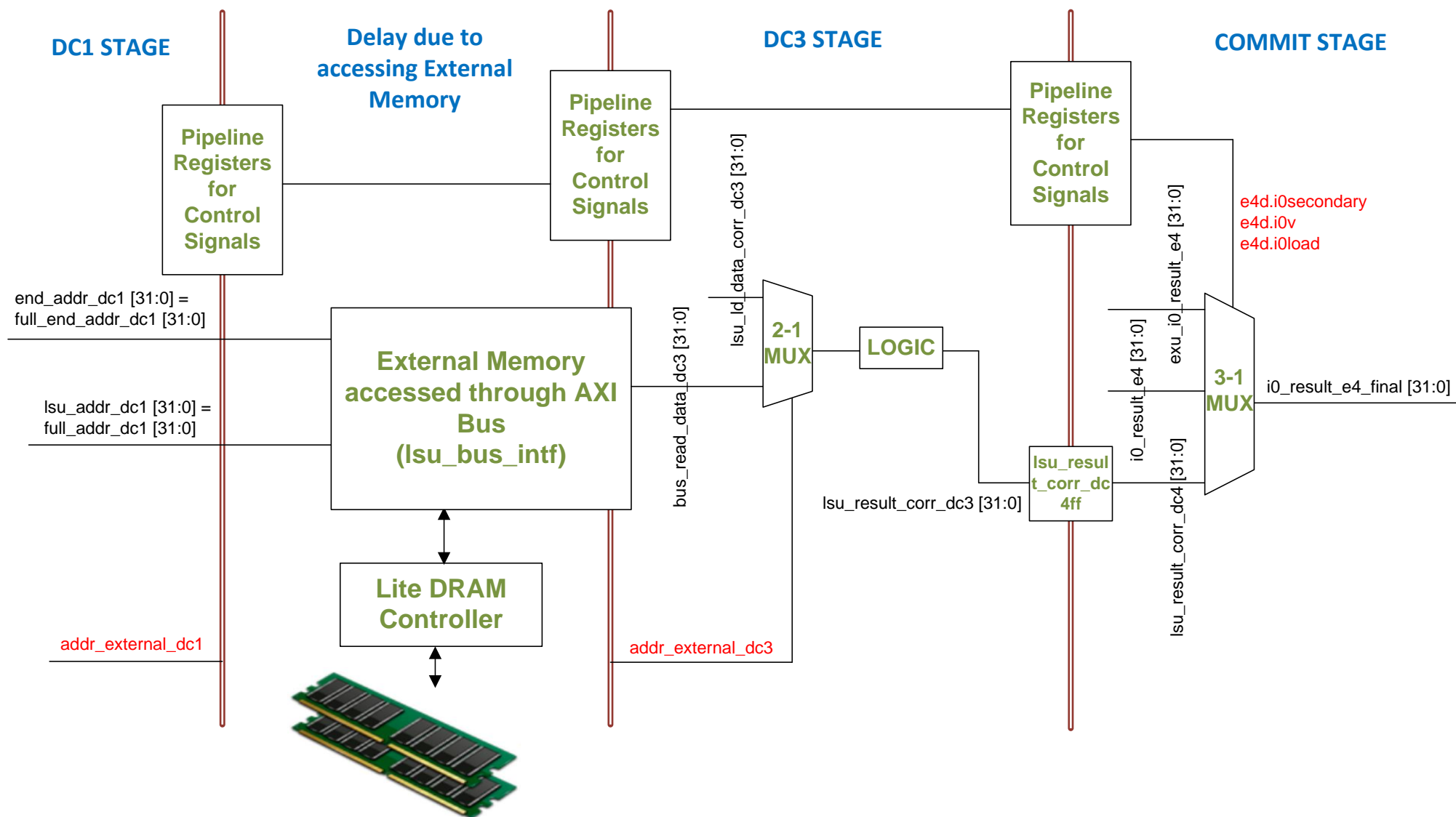


Figure 16. Blocking 1w instruction accessing External Memory

Figure 17 shows the execution of the `lw` in the fourth iteration of the loop of Figure 15, where it reads the value stored in address `0x00002204` into register `t3`. Note that for this program the D array starts at address `0x000021F8`.

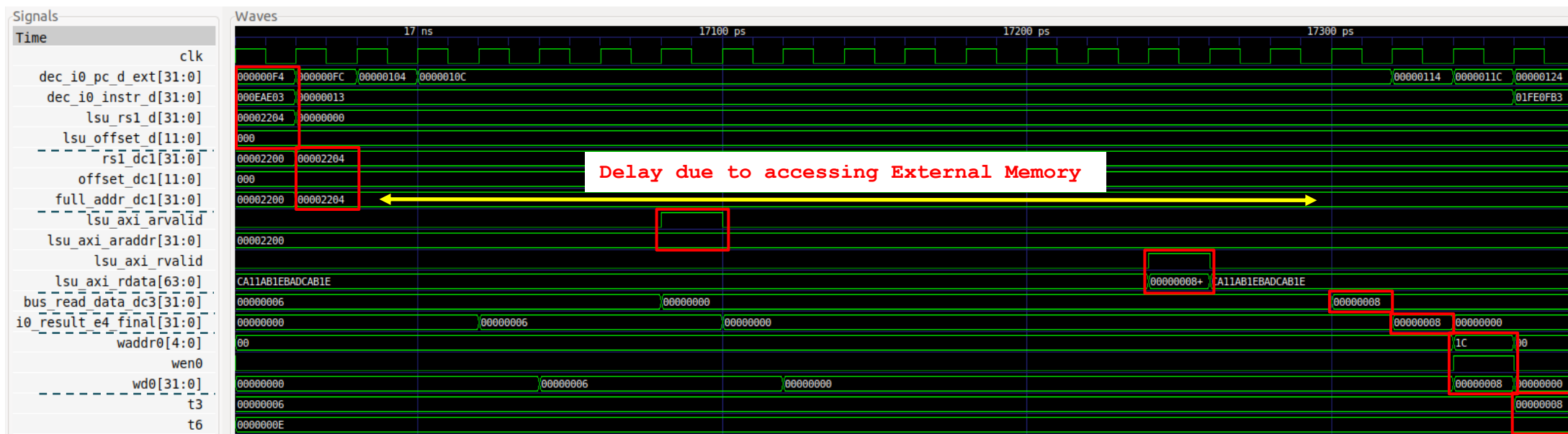



Figure 17. Verilog simulation of the example from Figure 15

TASK: Replicate the simulation from Figure 17 on your own computer. Use file `test_Blocking.tcl` (provided at

`[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory`). Zoom In () several times and move to 16940ps.

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_dc1`) and the set of signals on the bottom (`i0_result_e4_final` through `wd0`) 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 Decode stage, as explained in Section 2. Signal `full_addr_dc1[31:0]` contains the *address*, which in the fourth iteration of our example (the one shown in Figure 17) is 0x00002204. Signal `end_addr_dc1[31:0]` (not shown in the figure) is also computed as explained in Section 2 and contains the address of the last byte to be accessed.
- Some cycles later, the address is sent to the external memory through the AXI bus using the following signals: `lsu_axi_arvalid = 1` and `lsu_axi_araddr = 0x00002200`. Note that the address sent is double-word aligned as 64 bits are read from memory per access. Data is read into signal `lsu_axi_rdata` (in Labs 19 and 20 we will analyse the memory hierarchy in detail). 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.

TASK: Modify the program from Figure 15 in order to analyse an unaligned load access that needs to send two addresses to the External Memory through the AXI Bus.

- Some cycles later, the external memory returns a 64-bit data read through the AXI Bus (`lsu_axi_rdata = 0x0000000800000006` and `lsu_axi_rvalid = 1`). This data is buffered within the LSU (module `lsu_bus_buffer`).
- The requested 32-bit data is extracted from the 64-bit data read from memory and inserted in the main pipeline path: `bus_read_data_dc3 = 0x00000008`.
- Then, this data is written into the Register File following the same path as in the example from Section 2: `i0_result_e4_final` → `wd0`.

TASK: Add to the simulation the signals that control the multiplexers (in the DC3 and Commit stages in Figure 16) that select the data provided by the DDR External Memory. You can find these multiplexers at the following lines of the Verilog code:

- 2:1 Multiplexer: Line 264 of module `lsu_lsc_ctl`.
- 3:1 Multiplexer: Line 2277 of module `dec_decode_ctl`.

A `.tcl` file that you can use is provided at:

`[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory/test_Blocking_Extended.tcl`

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

APPENDIX A – OPERATION OF THE STORE BUFFER

The *Store Buffer* is an 8-entry circular queue, located inside the LSU, where every store (s_w) operation is tracked, annotating its target address and data. Generally a *store buffer* may be used to:

- Satisfy subsequent *load* operations with a previous *store* data if their target addresses match. This data forwarding solves Read After Write hazards and saves memory accesses.
- Let independent *load* operations fast-forward previous pending *store* operations, given that *load* operations are likely to be in the critical path.
- Merge compatible *store* operations into a single operation, thus solving Write After Write hazards and saving memory accesses.

Figure 18 shows some of the relevant signals of the *Store Buffer* when executing the code from Figure 11. These new signals are added to the ones shown in Figure 12. Like that figure, Figure 18 shows the execution of the s_w instruction in the fourth iteration of the loop.

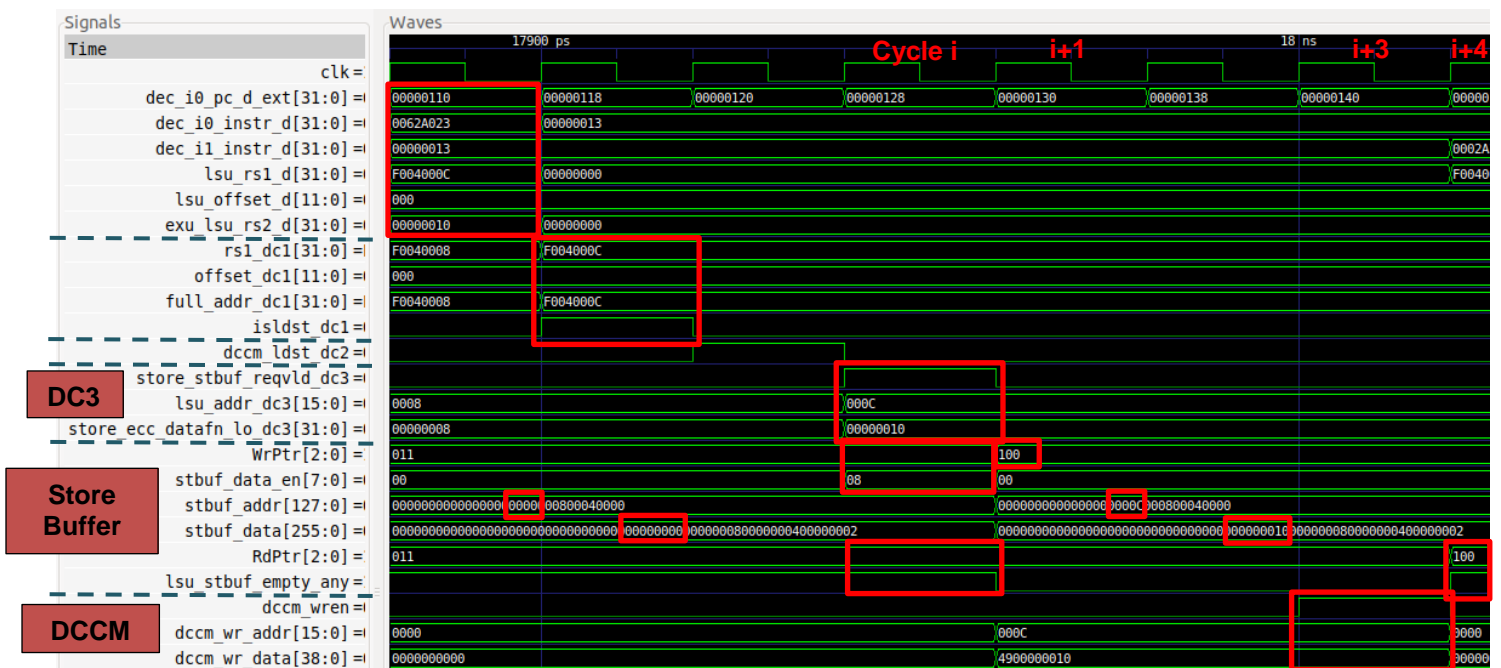



Figure 18. Verilator simulation for the example from Figure 11 illustrating the *Store Buffer* operation

TASK: Replicate the simulation from Figure 18 on your own computer. Use file *scriptStoreBuffer.tcl* (provided at

[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM). Zoom In  several times and move to 17900ps.

The top signals (from the Decode, DC1 and DC2 stages), were shown in Figure 12 and explained there, thus we do not explain them here again. In DC3 (cycle i in the figure), the write to the *Store Buffer* is prepared. The final address and the data to be written to memory by the store instruction are sent to the *Store Buffer* through signals $lsu_addr_dc3 =$

0x000C and `store_ecc_datafn_lo_dc3 = 0x00000010`. Signal `store_stbuf_reqvld_dc3` is asserted when a store operation is identified at the DC3 Stage, which triggers the store buffer operation.

The next set of signals corresponds to internal *Store Buffer* signals (you can find these in module `lsu_stbuf`). `WrPtr` codifies the entry of the *Store Buffer* where the next *sw* operation will place its address and data. In the example, `WrPtr` is 0b011 (i.e. the entry number three, which is the fourth entry since numbering starts in 0).

During the DC3 Stage (cycle *i*), the fourth entry of the store buffer is enabled by asserting the fourth bit of signal `stbuf_data_en` (note that 0x08 translates to 00001000 in one-hot coding, and the only '1' value is in the fourth bit position). Signal `lsu_stbuf_empty_any` goes low at the end of this cycle to indicate that the *Store Buffer* is not empty – that is, the *Store Buffer* holds data that is pending to be written to memory.

In the Commit stage (cycle *i*+1), the update of the fourth entry of the store buffer happens. Signal `stbuf_addr` and `stbuf_data` contain in their fourth entry: 0x000C (which corresponds to the DCCM address to write) and 0x00000010 (which corresponds to the data to be stored in the DCCM), respectively. `WrPtr` has been incremented to point to the next buffer entry (b100), and `RdPtr` tracks the oldest value in the buffer that has not yet been committed (b011).

One cycle after the Writeback stage (cycle *i*+3), the DCCM write enable signal (`dccm_wren`) is asserted, thus writing to memory and releasing the fourth entry of the buffer. Finally, at cycle *i*+4, `RdPtr` is updated (to b100) and the buffer is empty again so `lsu_stbuf_empty_any` goes high again.

Figure 19 shows how the 8-entry *Store Buffer* evolves in the example shown in Figure 18. In cycle *i*, the *Store Buffer* is empty, indicated by `WrPtr == RdPtr`. In cycle *i*+1 the *Store Buffer* contains one *address/data* pair (0x000C/0x00000010), which corresponds to the store analysed in Figure 18. Finally, in cycle *i*+4, the data is written to the DCCM and the *Store Buffer* becomes empty again (`WrPtr == RdPtr`).

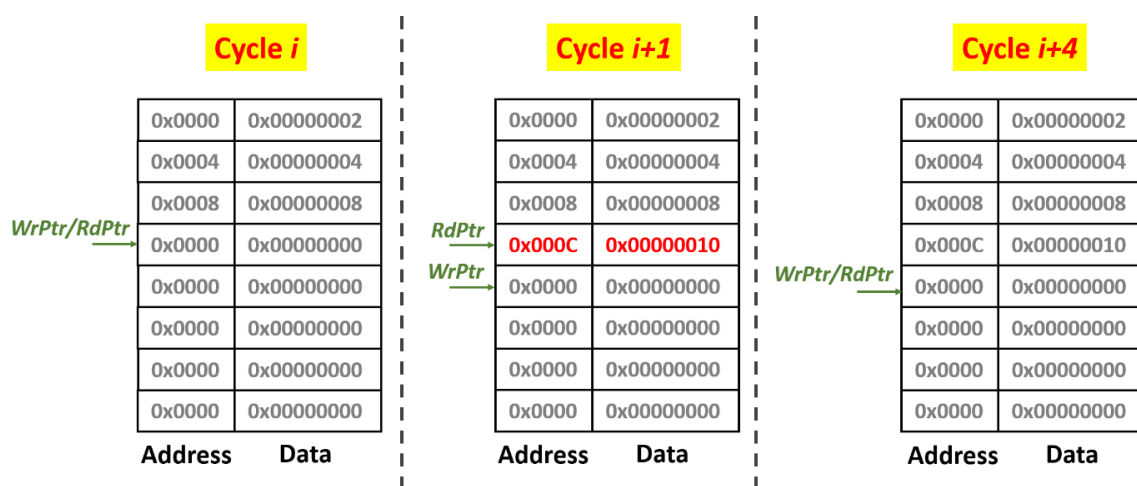


Figure 19. *Store Buffer* changes during the example from Figure 18

TASK: Modify the program from Figure 11 in order to have two outstanding stores and perform a similar analysis to the one from Figure 18.