THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 14

Structural Hazards

1. INTRODUCTION

In the next three labs, Labs 14-16, we discuss **pipeline hazards**. As explained by D. Patterson and J. Hennesy in Chapter 4, Section 5 of their recent RISC-V book (Computer Organization and Design RISC-V Edition, Patterson & Hennessy, © Morgan Kaufmann 2017) [PaHe]: Situations exist in pipelining when the next instruction cannot execute in the following clock cycle. These events are called hazards. Three different types of hazards occur: **structural hazards**, **data hazards** and **control hazards**.

As explained by Patterson and Hennessy in [PaHe], **structural hazards** occur when an instruction cannot execute because the hardware does not support the combination of instructions that are set to execute. In this lab, we analyse structural hazards in the SweRV EH1 processor.

In Lab 15 we analyse the second type of hazard, **data hazards**, in the SweRV EH1 processor. As explained by Hennessy and Patterson in the 6th edition of their book "Computer Architecture: A Quantitative Approach" [HePa]: Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor.

Finally, the third type of hazard is called a **control hazards**. As explained by S. Harris and D. Harris in Section 7.5.3 of their book "*Digital Design and Computer Architecture: RISC-V Edition*" (which we call DDCARV), a *control hazard* occurs when the decision of what instruction to fetch next has not been made by the time the fetch takes place. In Lab 16 we analyse control hazards in the SweRV EH1 processor.

2. STRUCTURAL HAZARDS IN SweRV EH1

In this section, we illustrate two cases of structural hazards that can occur in the SweRV EH1 processor. Each is resolved in a different way, thus resulting in a different performance-cost trade-off.

To illustrate the first situation we create an example in Section 2.A based on the integer multiply instruction (`mul`). At the same time, we describe the execution of this instruction in SweRV EH1, which we have not yet analysed in previous labs. Recall from Sections 3 and 4 of the GSG that this instruction belongs to the RISC-V M Extension (Standard Extension for Integer Multiplication and Division), which is supported in SweRV EH1. For executing this instruction, the SweRV EH1 processor implements a pipelined multi-cycle multiply unit (i.e. a multiplier that is pipelined and needs more than one cycle for computing the result) in the Multiply pipe (see Figure 4 of Lab 11), divided in three stages: M1, M2 and M3.

Specifically, in this example two `mul` instructions arrive at the Decode stage in the same cycle. Because SweRV EH1 only has one multiply unit, a structural hazard occurs, as the processor "does not support the combination of instructions that are set to execute" [PaHe]. In a processor using a non-pipelined multi-cycle multiplier, the second `mul` instruction would have to wait until the first one finished its execution, which would have an important impact in performance. However, as stated above, the multiplier used in SweRV EH1 is pipelined, thus the second `mul` instruction is only delayed by one cycle and it starts executing as soon as the first multiply instruction finishes the first stage of the multiplication (M1) and proceeds to the second stage (M2). This solution has a moderate impact in hardware cost (a pipelined structure is more expensive than a non-pipelined one), but it resolves the structural hazard with low impact on performance (only one cycle).

In the second example (Section 2.B), three instructions arrive at the Writeback stage in the same cycle, one of them being a non-blocking load executed several cycles earlier. In principle, because SweRV EH1 is a 2-way superscalar core, it would not be possible to complete three instructions in the same cycle; however, as we showed in Lab 11, SweRV EH1's register file has a third write port, which avoids the structural hazard in this situation. This solution has a high impact in hardware cost due to the extra register file port, but it resolves this structural hazard with no performance loss.

APPENDIX A – Two simultaneous `div` instructions in the Decode stage: In addition to these two examples, in the appendix at the end of this lab we illustrate another example based on divide instructions. Even though this example does not strictly illustrate a structural hazard, it is still very interesting and we recommend you to analyse it too.

A. Two simultaneous `mul` instructions in the Decode stage

The RISC-V M Extension includes, among others, the `mul` instruction. This instruction performs the multiplication of `rs1` by `rs2` and places the lower bits in the destination register (`rd`). The machine language instruction for `mul` is the following (see Appendix B of [DDCARV]):

```
0000001 | rs2 | rs1 | 000 | rd | 0110011
```

TASK: You can perform a similar study for the `mul` instruction as the one performed in Lab 12 for arithmetic-logic instructions: view the flow of the instruction through the pipeline stages, analyse the control bits (remember from Section 4 of SweRVref that there is a specific structure type for the `mul` instruction called `mul_pkt_t`, and there is a signal defined in module **dec_decode_ctl** called `mul_p`), etc.

The multiply unit is implemented in module **exu_mul_ctl** (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu_mul_ctl.sv*). As mentioned before, this unit is pipelined and it requires 3 cycles for computing the result. Using a pipelined – as opposed to non-pipelined – multiplier reduces the performance loss due to structural hazards.

TASK: Inspect the Verilog code from **exu_mul_ctl** and see how the multiplication is computed. Remember that RISC-V includes 4 multiply instructions (`mul`, `mulh`, `mulhsu` and `mulhu`), and all of them must be supported by the hardware.

As an optional exercise, you can replace the multiply unit with your own unit or one from the Internet.

The example from Figure 1 executes two `mul` instructions contained within a loop that repeats for 0xFFFF iterations (i.e. 65,535 in decimal). The `mul` instructions are highlighted in red in the figure. In this case, the `mul` instructions are surrounded by several `nop` instructions for isolating each iteration from each other. As usual, the program does nothing useful and is only intended to illustrate *structural hazards* due to `mul` instructions.

```
.globl Test_Assembly
Test_Assembly:

li t2, 0xFFFF

li t3, 0x3
li t4, 0x2
li t5, 0x2
li t6, 0x2

REPEAT:
   beq t2, zero, OUT      # Stay in the loop?
   INSERT_NOPS_9
   mul t0, t3, t4         # t0 = t3 * t4
   mul t1, t5, t6         # t1 = t5 * t6
   INSERT_NOPS_9
   add t2, t2, -1
   add t0, zero, zero
   add t1, zero, zero
   j REPEAT
OUT:

.end
```

Figure 1. Example with two consecutive `mul` instructions

Folder *[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instruction* provides the PlatformIO project so that you can analyse, simulate, and modify the program as desired. The structure of the project is based on the one provided in Lab 11 for using the performance counters: it contains a *.c* file that initializes, stops, and prints the value of the desired counters and a *.S* file that contains the assembly program that we want to test (in this case, the loop with the two `mul` instructions) and which is invoked from the *.c* file.

Open the project in PlatformIO, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instruction/.pio/build/swervolf_nexys/firmware.dis*). Notice that the `mul` instructions are placed at addresses 0x000001e8 and 0x000001ec.

```
0x000001e8:      03de02b3           mul   t0,t3,t4
0x000001ec:      03ff0333           mul   t1,t5,t6
```

TASK: Verify that this pair of 32 bits (0x03de02b3 and 0x03ff0333) correspond to instructions `mul t0,t3,t4` and `mul t1,t5,t6` in the RISC-V architecture.

Figure 2 shows the simulation of the program from Figure 1 at the second iteration of the loop.

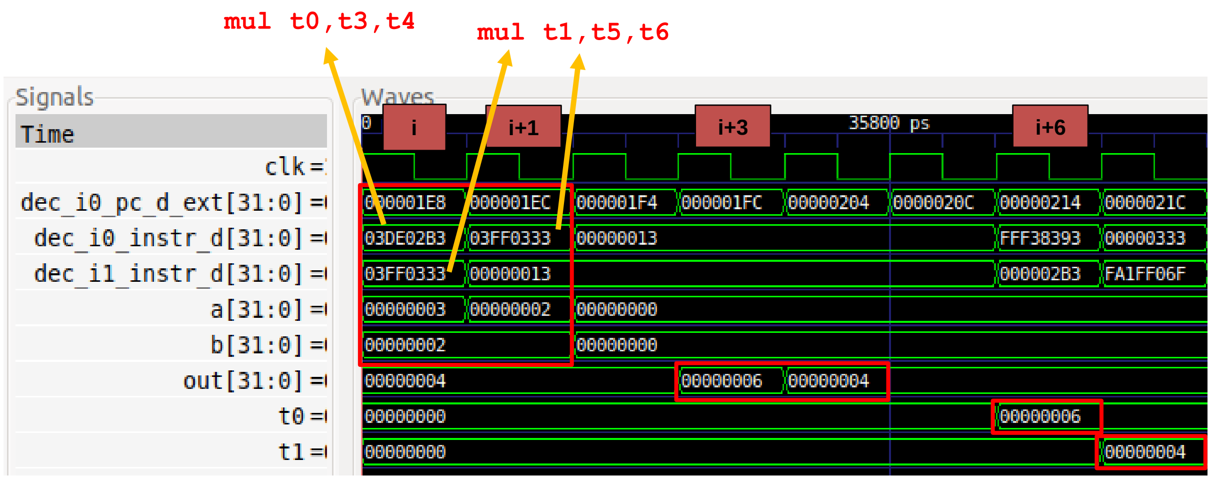


Figure 2. Verilator simulation of the example from Figure 1

TASK: Replicate the simulation from Figure 2 on your own computer and analyse it more closely. You can use the *.tcl* file *[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instruction/test.tcl*

Analyse the waveform from Figure 2. The values highlighted in red correspond to different signals related with the `mul` instructions as they traverse the pipeline.

- **Cycle i**: The two `mul` instructions arrive in the same cycle to the Decode stage. A Structural Hazard prevents the second `mul` instruction (`dec_i1_instr_d` = 0x03ff0333) to advance to the next stage, given that the first `mul` instruction (`dec_i0_instr_d` = 0x03de02b3) is scheduled to that unit.

- **Cycle i+1**: The first `mul` instruction executes in the first stage of the pipelined multiplier (M1), while the second `mul` instruction waits in the Decode stage.

- **Cycle i+2**: The first `mul` instruction executes in the second stage of the pipelined multiplier (M2) and the second `mul` executes in the first stage (M1).

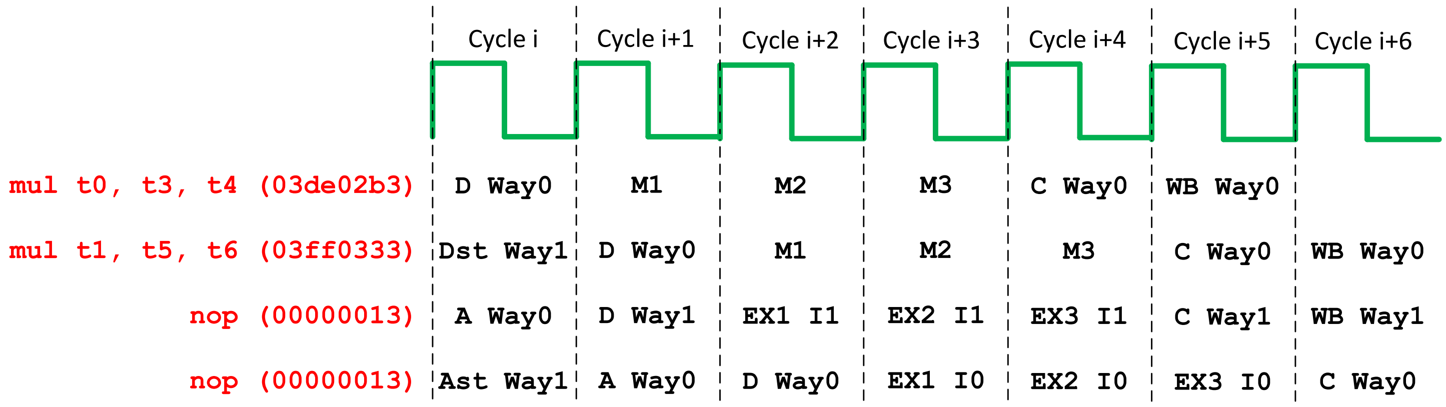
- **Cycle i+3**: The first `mul` instruction reaches EX3, when the result of the multiplication is produced (`out` = 0x6 for the first `mul` instruction).

- **Cycle i+4**: The second `mul` instruction reaches EX3, when the result of the multiplication is produced (`out` = 0x4 for the second `mul` instruction).

- **Cycle i+6**: The register file is updated with the result of the first `mul` instruction (`t0` = 0x6).

- **Cycle i+7**: The register file is updated with the result of the second `mul` instruction (`t1` = 0x4).

Figure 3 illustrates the flow of the instructions of the example from Figure 1 through the SweRV EH1 pipeline. **D** stands for the Decode stage, **A** for the Align stage, **C** for the Commit stage and **WB** for the Writeback stage. When the first `mul` instruction is decoded (cycle i), most subsequent instructions stall at their current stage (marked in the figure with the *st* suffix) and bubbles are inserted. In the next cycle (i+1) the instructions are resumed (note that the second `mul` instruction has been moved from Way 1 to Way 0, and Way 1 contains the next instruction, which is a `nop`). In cycle i+2 the first `mul` instruction is in the second stage of execution (M2) and the second `mul` instruction is in the first stage of execution (M1). In cycles i+5 and i+6, the two `mul` instructions write their result back to the register file, which can be seen updated in Figure 2 in cycles i+6 and i+7.



	Cycle i	Cycle i+1	Cycle i+2	Cycle i+3	Cycle i+4	Cycle i+5	Cycle i+6
mul t0, t3, t4 (03de02b3)	D Way0	M1	M2	M3	C Way0	WB Way0	
mul t1, t5, t6 (03ff0333)	Dst Way1	D Way0	M1	M2	M3	C Way0	WB Way0
nop (00000013)	A Way0	D Way1	EX1 I1	EX2 I1	EX3 I1	C Way1	WB Way1
nop (00000013)	Ast Way1	A Way0	D Way0	EX1 I0	EX2 I0	EX3 I0	C Way0

Figure 3. Execution of Figure 1 example code

TASK: Compare the illustration from Figure 3 with the simulation from Figure 2 focusing on the two `mul` instructions. Specifically, analyse how the two instructions are assigned to the two ways in the Align and Decode stages and how they progress through the pipeline.
- In module **ifu_aln_ctl** (Align stage) the two instructions are assigned to the following signals whenever possible:
  - Way 0: `ifu_i0_instr`
  - Way 1: `ifu_i1_instr`
- In module **dec_ib_ctl** the two instructions are buffered from Align to Decode. Note that in some cases instructions can be stalled in these buffers and reassigned to a different way:
  - Way 0: `ifu_i0_instr` → `dec_i0_instr_d`
  - Way 1: `ifu_i1_instr` → `dec_i1_instr_d`
- In module **dec_decode_ctl** (Decode stage) the two instructions are scheduled to the corresponding pipes whenever possible. Once they are sent, they continue through the three execution stages, the Commit stage and the Writeback stage:
  - Way 0: `i0_inst_e1` → `i0_inst_e2` → `i0_inst_e3` → `i0_inst_e4` → `i0_inst_wb`
  - Way 1: `i1_inst_e1` → `i1_inst_e2` → `i1_inst_e3` → `i1_inst_e4` → `i1_inst_wb`

We provide a *.tcl* file called *[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instruction/test_AssignmentWays.tcl* that includes all these signals.

TASK: Remove the `nop` instructions included within the loop from Figure 1 and measure different events (cycles, instructions/multiplies committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11. Is the number of cycles as expected after analysing the simulation from Figure 2? Justify your answer.
Now reorder the code within the loop trying to reach the ideal throughput. Justify the results obtained in the original code and in the reordered one.

TASK: Folder *[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instr_Accumul_C-Lang* provides the

PlatformIO project of a C program that accumulates the subtraction of two multiplications within a loop.

- Analyse the C program.

- Perform a simulation and inspect a random iteration of the loop. Note that the C program is compiled without optimizations.

- Measure different events (cycles, instructions/multiplications committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11.
Is the number of cycles as expected after analysing the simulation from Figure 2? Justify your answer.

- Create an analogous program in RISC-V assembly and compare it with the C version. Reorder the instructions trying to obtain the best possible IPC.

- Disable the **M** RISC-V extension in the C program and compare the results with the original program. To do so, modify the following line in file *platformio.ini* from:
```
    build_flags = -Wa,-march=rv32ima -march=rv32ima
```
To:
```
    build_flags = -Wa,-march=rv32ia -march=rv32ia
```
This avoids the use of the instructions from the M RISC-V extension and emulates them using other instructions instead.

TASK: Modify the program from Figure 1, replacing the two `mul` instructions for two `lw` instructions to the DCCM. You should observe a structural hazard analogous to the one analysed in this section and resolved in a similar way.

B. Three simultaneous instructions executing in the Writeback stage

SweRV EH1 is a 2-Way superscalar processor (we have briefly discussed this feature in the GSG and in previous labs, and we will analyse it in more detail in Lab 17). This means that two instructions can execute in this processor per cycle. In a situation where three instructions arrived at the same stage in the same cycle, a structural hazard would potentially occur. It might look like such a situation is not possible given the structure of SweRV EH1, however, there is a specific case when this can happen:

- The External DDR2 Memory has a moderate latency that forces load instructions to stall. When the load eventually receives its data from memory it proceeds to the Writeback stage, where it writes the read value to the register file (let's assume that this Writeback happens in cycle ***i***).

- If loads are non-blocking (i.e. while the load is waiting for the data to arrive from memory, the processor continues executing instructions that do not depend on that data), it may happen that two other instructions arrive at the Writeback stage in cycle ***i*** and also need to write to the register file (for example, two `add` instructions).

- In this situation, three instructions would be trying to write to the register file in the same cycle (cycle ***i***).

If the register file only had two write ports, a structural hazard would occur and one of the three instructions trying to write would have to wait for the register file to be free. However, in SweRV EH1, as we showed in Lab 11, a third write port is implemented, which allows this structural hazard to be resolved with no stalls and, thus, no performance loss.

The example from Figure 4 illustrates this situation. It executes a non-blocking `lw` instruction followed by 36 `add` instructions contained within a loop that repeats for 0xFFFF iterations (i.e. 65,535). The `lw` instruction is highlighted in red in the figure. The two `add` instructions, which arrive at the Writeback stage in the same cycle as the `lw` instruction (cycle ***i***), are also highlighted. In this case, `nop` instructions are not included. As usual, the program does nothing useful and is only intended to illustrate the example of this section.

```
REPEAT:
   lw x28, (x29)
   add x30, x30, -1
   add x1, x1, 1
   add x31, x31, 1
   add x3, x3, 1
   add x4, x4, 1
   add x5, x5, 1
   add x6, x6, 1
   add x7, x7, 1
   add x8, x8, 1
   add x9, x9, 1
   add x10, x10, 1
   add x11, x11, 1
   add x12, x12, 1
   add x13, x13, 1
   add x14, x14, 1
   add x15, x15, 1
   add x16, x16, 1
   add x17, x17, 1
   add x18, x18, 1
   add x19, x19, 1
   add x20, x20, 1
   add x21, x21, 1
   add x22, x22, 1
   add x23, x23, 1
   add x24, x24, 1
   add x25, x25, 1
   add x26, x26, 1
   add x27, x27, 1
   add x31, x31, 1
   add x3, x3, 1
   add x4, x4, 1
   add x5, x5, 1
   add x6, x6, 1
   add x25, x25, 1
   add x26, x26, 1
   add x27, x27, 1
   bne x30, zero, REPEAT    # Repeat the loop
```

Figure 4. Example for a non-blocking `lw` instruction followed by 36 A-L instructions

Folder *[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory* provides the PlatformIO project so that you can analyse, simulate, and modify the program as desired. The structure of the project is based on the one provided in Lab 11 for using the performance counters: it contains a *.c* file that initializes, stops, and prints the value of the desired counters and a *.S* file that contains the assembly program that we want to test (in this case, the loop with the non-blocking `lw` instruction) and which is invoked from the *.c* file.

As shown in Figure 5, the 32-bit data obtained in the **lsu_bus_intf** module (Bus Interface) is provided to the register file through signal `lsu_nonblock_load_data[31:0]`. Also, the control signals that tell the register file where to write that data and when to write it, which were generated in the Decode stage and propagated through the Pipeline Registers, are provided to the register file through signals `dec_nonblock_load_waddr[4:0]` and `dec_nonblock_load_wen` respectively. These three signals go into the register file through the third write port available in this structure (`waddr2`, `wen2` and `wd2`), as illustrated in the figure. Remember that in Figure 6 of Lab 11 we illustrated the register file in detail.

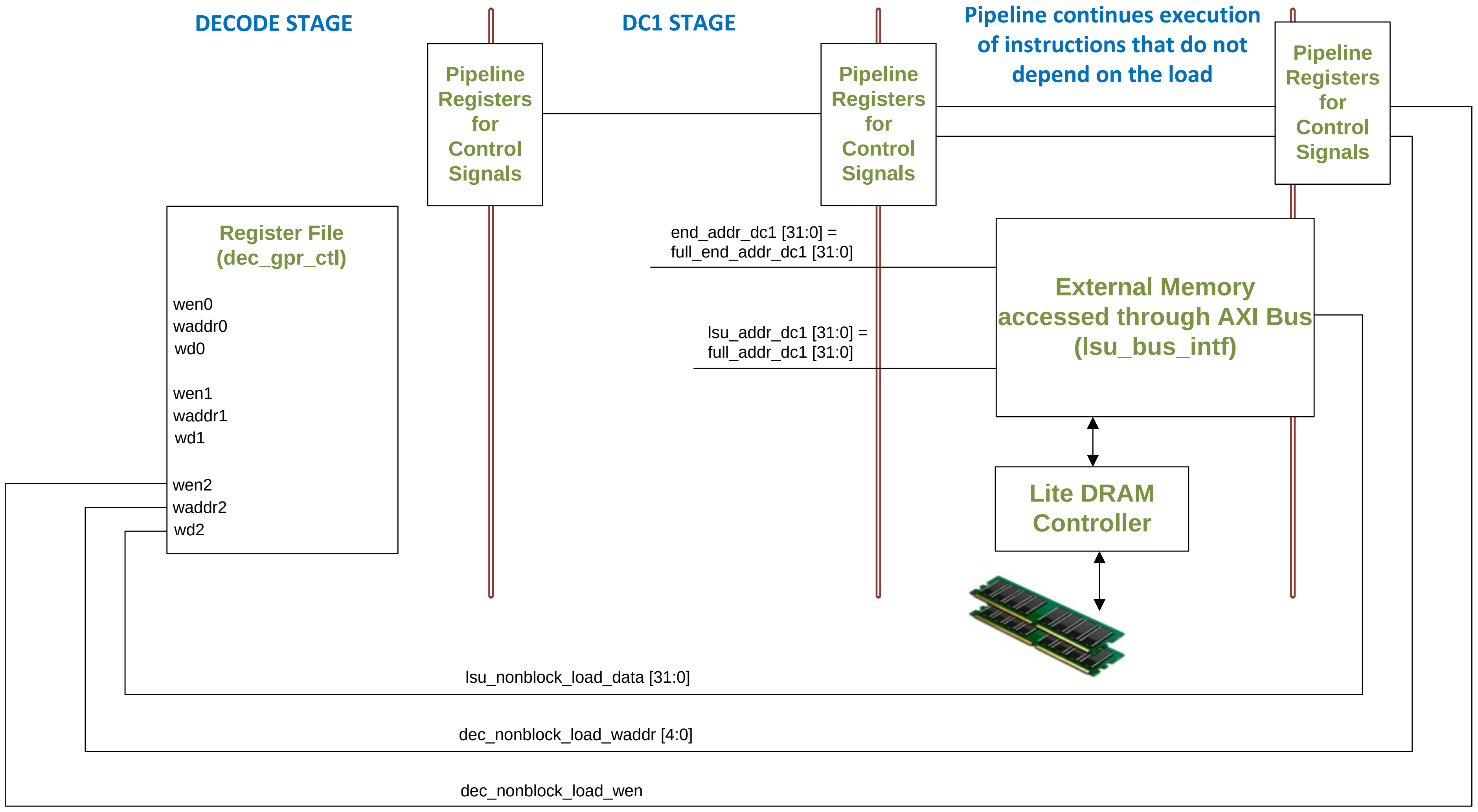

DECODE STAGE
DC1 STAGE
Pipeline continues execution of instructions that do not depend on the load
Pipeline Registers for Control Signals
Pipeline Registers for Control Signals
Pipeline Registers for Control Signals
Register File (dec_gpr_ctl)
wen0
waddr0
wd0
wen1
waddr1
wd1
wen2
waddr2
wd2
end_addr_dc1 [31:0] = full_end_addr_dc1 [31:0]
lsu_addr_dc1 [31:0] = full_addr_dc1 [31:0]
External Memory accessed through AXI Bus (lsu_bus_intf)
Lite DRAM Controller
lsu_nonblock_load_data [31:0]
dec_nonblock_load_waddr [4:0]
dec_nonblock_load_wen

Figure 5. Non-Blocking load instruction accessing External Memory

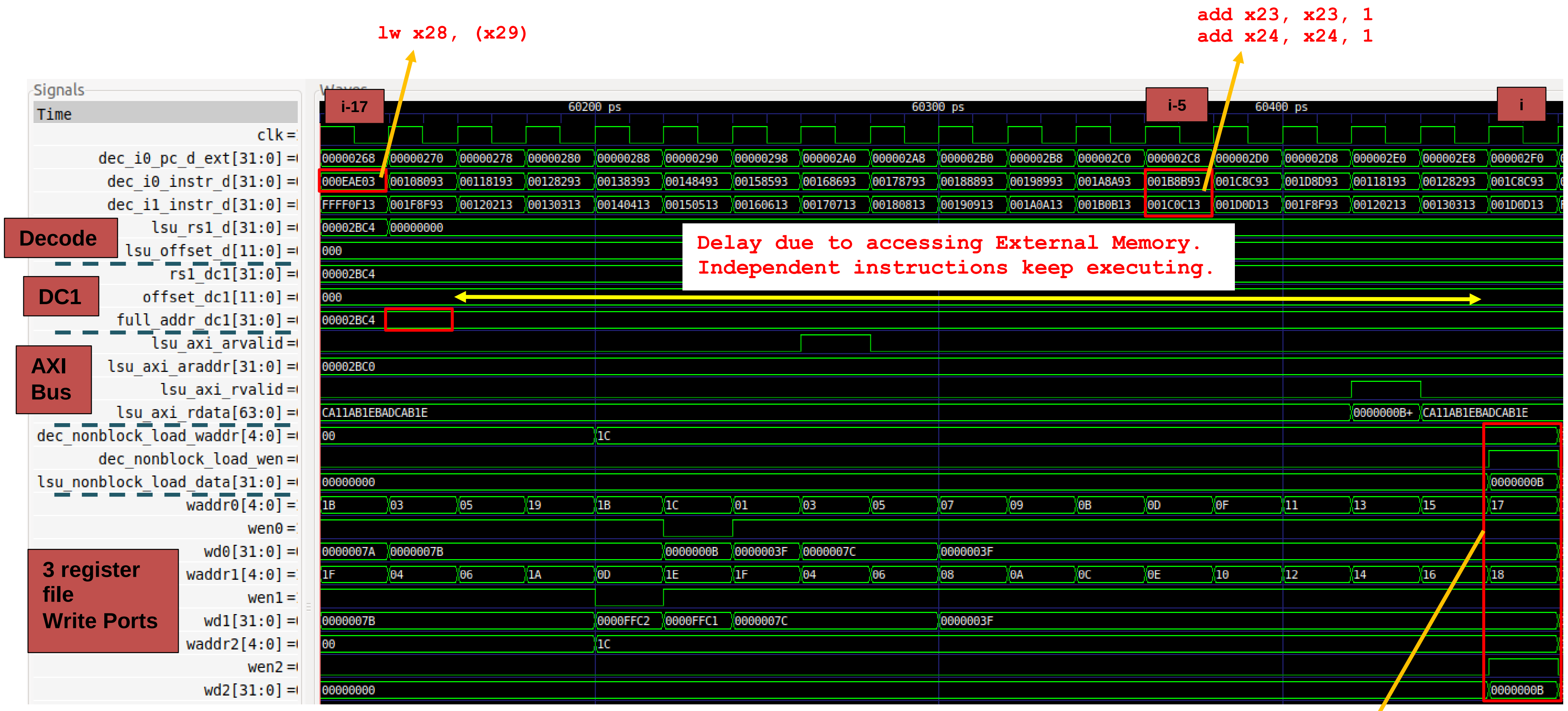


Figure 6. Verilator simulation for the example from Figure 4

Three simultaneous writes to the Register File:
- lw writes register x28 (0x1C)
- add writes register x23 (0x17)
- add writes register x24 (0x18)

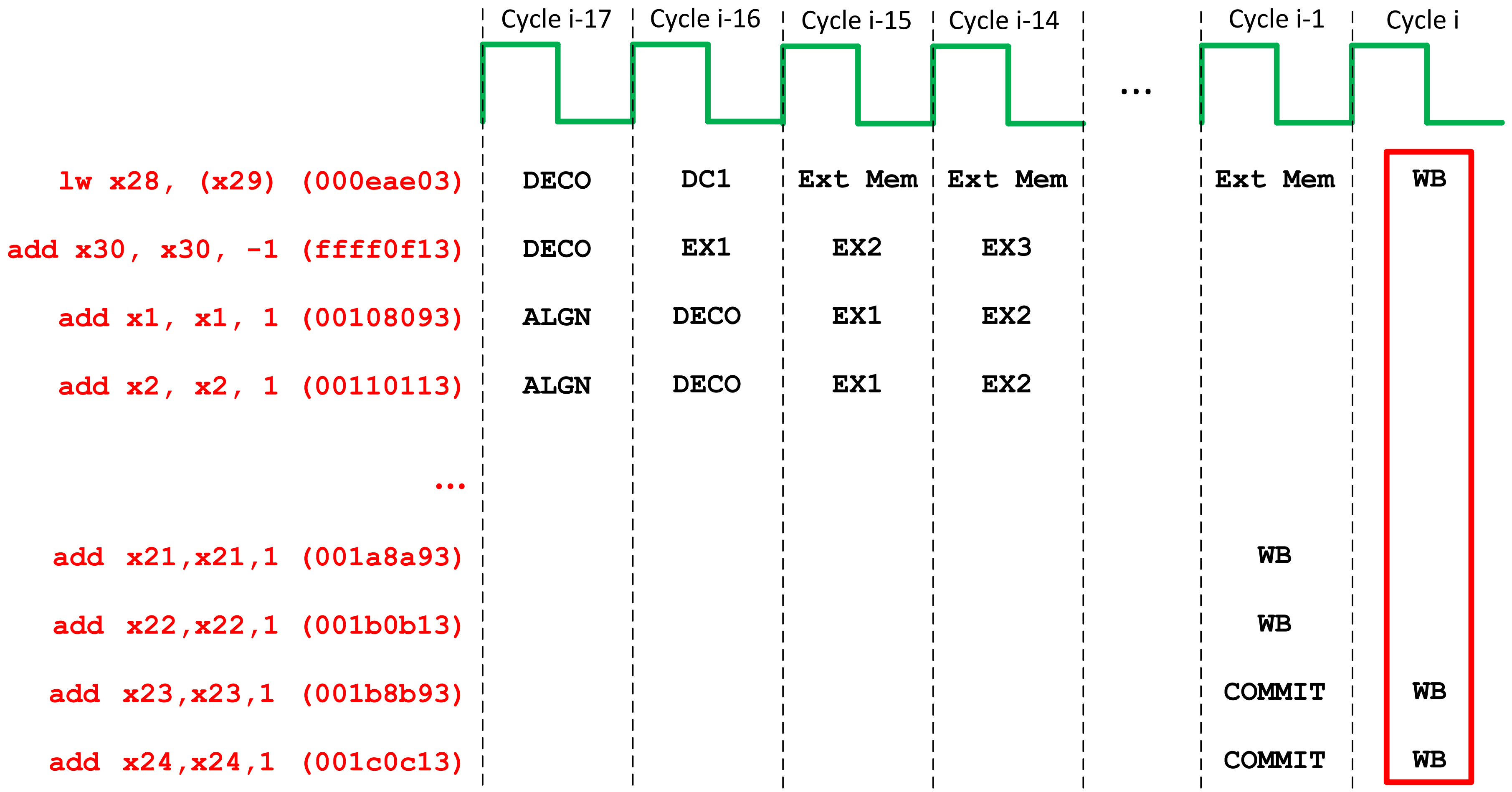


Figure 7. Execution of the example code from Figure 4

Figure 6 and Figure 7 show the Verilator simulation for the program from Figure 4 and a diagram that illustrates the execution of this program for a random iteration of the loop.

TASK: Replicate the simulation from Figure 6 on your own computer. Use file *test_NonBlocking.tcl* (provided at

[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory). *Zoom In* ( ) several times and move to 60120ps.

Analyse the waveform from Figure 6 and the diagram from Figure 7.

- **Cycle i-17**: The `lw` instruction is at the Decode stage.

- **Cycle i-16**: The effective memory address is computed and sent to the External Memory through the AXI Bus. The latency of the External Memory forces the load instruction to wait several cycles for the data to arrive to the core.

- **Cycle i-5**: The two conflicting `add` instructions are decoded.

- **Cycle i**: The `lw` instruction and the two conflicting `add` instructions proceed to the Writeback stage, where they all must write the register file. This is possible thanks to the three write ports available in SweRV EH1's register file. Note that the register numbers are shown in hexadecimal in the simulation. `x23`, `x24` and `x28` (registers 0x17, 0x18, and 0x1c) are being written.

TASK: Compare the simulation shown in Figure 6 (non-blocking load) with the simulation shown in Figure 14 of Lab 13 (blocking load). Add all of the signals needed for the comparison.

TASK: Compare the illustration from Figure 7 with the simulation from Figure 6 that you have replicated on your own computer. Add signals to extend the simulation and deepen understanding, as desired.

TASK: Measure different events (cycles, instructions/loads committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11. Is the number of cycles as expected after analysing the simulation from Figure 6? Justify your answer. Compare these results with those obtained when loads are configured as blocking loads.

3. EXERCISES

1. Analyse, both in simulation and on the board, the structural hazard that happens between two consecutive memory instructions (you can analyse any combination of two consecutive memory instructions such as loads and stores) that arrive at the L/S Pipe in the same cycle. You can use the PlatformIO project provided at: *[RVfpgaPath]/RVfpga/Labs/Lab14/TwoConsecutiveLW_Instructions*.

2. (*The following exercise is based on exercise 4.22 from the book "Computer Organization and Design – RISC-V Edition", by Patterson & Hennessy ([PaHe]).*)
   Consider the fragment of RISC-V assembly below:

```
sw x29, 12(x16)
lw x29, 8(x16)
sub x17, x15, x14
beqz x17, label
add x15, x11, x14
sub x15, x30, x14
```

   Suppose we modify the SweRV EH1 processor so that it has only one memory (that handles both instructions and data). In this case, there will be a structural hazard every time a program needs to fetch an instruction during the same cycle in which another instruction accesses data.
   a. Draw a pipeline diagram to show where the code above will stall in this imaginary version of the SweRV EH1 processor.
   b. In general, is it possible to reduce the number of stalls/nops by reordering code?
   c. Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding nops to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.

APPENDIX A – TWO SIMULTANEOUS DIV INSTRUCTIONS IN THE DECODE STAGE

This extra example is based on the integer division instruction (`div`). Like the `mul` instruction, the `div` instruction belongs to the RISC-V M Extension (Standard Extension for Integer Multiplication and Division), which is supported in SweRV EH1.

The `div` instruction performs the signed integer division of `rs1` by `rs2` and stores the result in `rd`. The machine language instruction for `div` is (see Appendix B of [DDCARV]):

```
0000001 | rs2 | rs1 | 100 | rd | 0110011
```

TASK: You can perform a similar study for the `div` instruction as the one performed in Lab 12 for arithmetic-logic instructions: view the flow of the instruction through the pipeline stages, analyse the control bits (remember from Section 4 of SweRVref that there is a specific structure type for the `div` instruction called `div_pkt_t`, and there is a signal defined in module **dec_decode_ctl** called `div_p`), etc.

For executing this instruction, the SweRV EH1 processor implements a non-pipelined blocking multi-cycle divide unit in module **exu_div_ctl** (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu_div_ctl.sv*). This unit needs up to 34 cycles to compute the result, however, depending on the inputs, it can be much smaller. The divide unit outputs several signals to the processor (`div_stall`, `finish_early`, `finish`) to indicate the status of a divide instruction.

TASK: Inspect the Verilog code from **exu_div_ctl** to understand how the division is computed. Also analyse the effect of signals `div_stall`, `finish_early`, and `finish`. As an optional exercise, replace the divide unit with your own unit or one from the Internet.

The example from Figure 8 executes two `div` instructions contained within a loop that repeats for 0xFFFF iterations (i.e. 65,535 in decimal). The `div` instructions are highlighted in red in the figure. In this case, as opposed to many other examples, `nop` instructions are not necessary, as the `div` instructions are already isolated from any other instruction due to the high latency of the divide unit. As in previous toy programs we've used, the program does nothing useful.

```
.globl Test_Assembly
Test_Assembly:

li t2, 0xFFFF

li t3, 0x8000000
li t4, 0x2
li t5, 0x2000000
li t6, 0x2

REPEAT:
   div t0, t3, t4         # t0 = t3 / t4
   div t1, t5, t6         # t1 = t5 / t6
   add t2, t2, -1
   add t0, zero, zero
   add t1, zero, zero
   bne t2, zero, REPEAT   # repeat the loop

.end
```

Figure 8. Example for two consecutive `div` instructions

Folder *[RVfpgaPath]/RVfpga/Labs/Lab14/DIV_Instruction* provides the PlatformIO project so that you can analyse, simulate and change the program as desired. The structure of the project is like the one used for the `mul` instruction and is based on the one included in Lab 11 for using the Performance Counters.

If you open the project in PlatformIO, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab14/DIV_Instruction/.pio/build/swervolf_nexys/firmware.dis*) you will see that the `div` instructions are placed at addresses 0x000001c0 and 0x000001c4.

```
0x000001c0:      03de42b3              div   t0,t3,t4
0x000001c4:      03ff4333              div   t1,t5,t6
```

<u>TASK</u>: Verify that this pair of 32 bits (0x03de42b3 and 0x03ff4333) corresponds to instructions `div t0,t3,t4` and `div t1,t5,t6` in the RISC-V architecture.

Figure 9 shows the simulation of the program from Figure 8 at a random iteration of the loop.

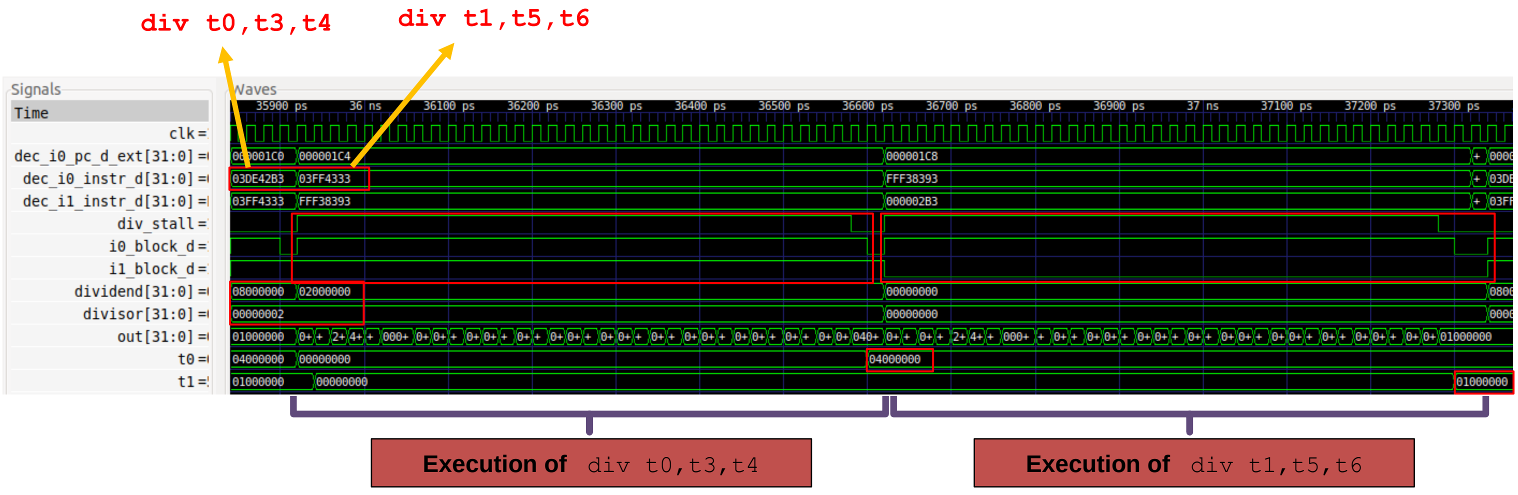


Figure 9. Verilator simulation for the example from Figure 8

TASK: Replicate the simulation from Figure 9 on your own computer and analyse it in detail.

Analyse the waveform from Figure 9. The values highlighted in red are signals related to the two `div` instructions as they traverse the pipeline.

- The two `div` instructions arrive at the Decode stage in the same cycle (`dec_i0_pc_d_ext` = 0x000001c0, which is the instruction address of the first `div`). The first `div` instruction (0x03de42b3) is scheduled to execute in the divide unit, so it sends the dividend and divisor (`dividend` = 0x08000000 and `divisor` = 0x00000002) to this unit. Note that we select high values for the dividends for making the division computation time close to the maximum (34 cycles).

  Given that the division is blocking in SweRV EH1, any other instruction after the `div` is stalled. Note however that, even if the division was non-blocking, a structural hazard due to having only one divisor would make the second `div` instruction to stall. As explained in Section 2, there would be other approaches for improving performance, such as pipelining the divider or including another one. However, given that division is not a frequent operation, hardware cost reduction prevails in this case.

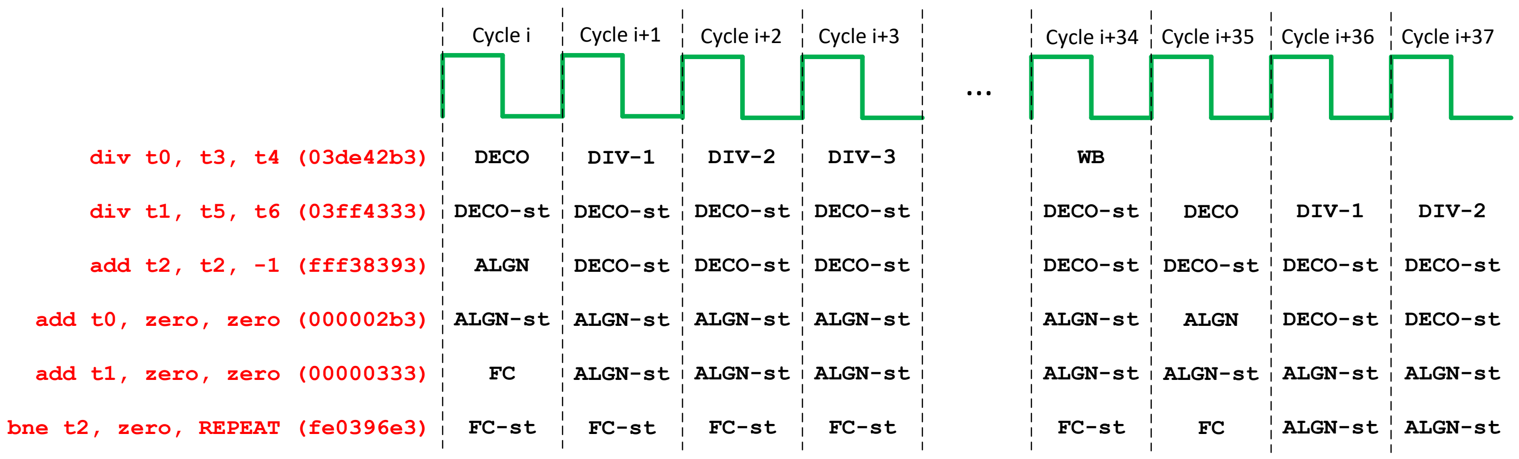
- The pipeline is stalled during the execution of the first `div` instruction (see signal `div_stall` = 1 during the first division computation). You can also see that both Way-0 and Way-1 are blocked with signals `i0_block_d` and `i1_block_d` being 1. Moreover, now `dec_i0_pc_d_ext` = 0x000001c4, which is the address of the second divide instruction, which is stalled in the Decode stage.

- Signal `out` of the divide unit provides the result after 34 cycles, which is written to the destination register (*t0* = 0x04000000). You can see how the output value changes every cycle as the divide operation successively becomes the final result.

- When the result is obtained, the divisor is released, the pipeline is allowed to continue (`div_stall` = 0) and the second `div` instruction is scheduled to the divide unit. Then, 34 cycles later, the result of the second `div` instruction is written to the register file (*t1* = 0x01000000).

As in the first `div` instruction, all the instructions after the second `div` must stall due to the blocking divisor. In this case, however, those instructions which do not depend on *t1* could continue, had it been a non-blocking divide.

Figure 10 illustrates the flow of the instructions in the example from Figure 8 through the SweRV EH1 pipeline. When the first `div` instruction is decoded (cycle i), the second `div` and subsequent instructions stall at their current stage both due to the blocking division of SweRV EH1 and to the structural hazard at the divide unit. (A stalled instruction is marked in the figure with the -*st* suffix.) Then, 34 cycles later (cycle i+34), the first `div` instruction finishes execution and writes the result back to the register file through the 2:1 multiplexer that was shown in Figure 4 of Lab 11. In the following cycle (i+35), subsequent instructions resume. Then, in cycle 36 the second `div` instruction starts execution and subsequent instructions are stalled again due to the blocking division of SweRV EH1.



	Cycle i	Cycle i+1	Cycle i+2	Cycle i+3	…	Cycle i+34	Cycle i+35	Cycle i+36	Cycle i+37
div t0, t3, t4 (03de42b3)	DECO	DIV-1	DIV-2	DIV-3		WB			
div t1, t5, t6 (03ff4333)	DECO-st	DECO-st	DECO-st	DECO-st		DECO-st	DECO	DIV-1	DIV-2
add t2, t2, -1 (fff38393)	ALGN	DECO-st	DECO-st	DECO-st		DECO-st	DECO-st	DECO-st	DECO-st
add t0, zero, zero (000002b3)	ALGN-st	ALGN-st	ALGN-st	ALGN-st		ALGN-st	ALGN	DECO-st	DECO-st
add t1, zero, zero (00000333)	FC	ALGN-st	ALGN-st	ALGN-st		ALGN-st	ALGN-st	ALGN-st	ALGN-st
bne t2, zero, REPEAT (fe0396e3)	FC-st	FC-st	FC-st	FC-st		FC-st	FC	ALGN-st	ALGN-st

Figure 10. Execution of Figure 8 example code (–*st* suffix indicates a stalled instruction)

<u>TASK</u>: Compare the illustration from Figure 10 and the simulation from Figure 9 that you have replicated on your own computer. Add signals to extend the simulation and deepen understanding, as desired.

<u>TASK</u>: Measure different events (cycles, instructions/divisions committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11.
Is the number of cycles as expected after analysing the simulation from Figure 9? Justify your answer.

<u>TASK</u>: Try different dividends and divisors and see how the number of cycles for computing the result depends on their value. View the experiment both in simulation and with the HW Counters.

<u>TASK</u>: Folder *[RVfpgaPath]/RVfpga/Labs/Lab14/DIV_Instr_Accumul_C-Lang* provides the PlatformIO project of a C program that accumulates the subtraction of two divisions within a loop.

- Analyse the C program.

- Perform a simulation and inspect a random iteration of the loop. Note that the C program is compiled without optimizations.

- Measure different events (cycles, instructions/divisions committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11.
Is the number of cycles as expected after analysing the simulation from Figure 9? Justify your answer.

- Create an analogous program in RISC-V assembly and compare it with the C version.

- Disable the **M** RISC-V extension in the C program and compare the results with the original program. To do so, modify the following line in file *platformio.ini* from:

```
    build_flags = -Wa,-march=rv32ima -march=rv32ima
```

To:

```
    build_flags = -Wa,-march=rv32ia -march=rv32ia
```

This avoids the use of the instructions from the RISC-V M extension and emulates them using other instructions instead.

TASK: In SweRV EH1, `div` instructions are blocking. Modify the processor to allow non-blocking `div` instructions.

Then add a second divider to the SweRV EH1 processor, so that two `div` instructions of the example from Figure 8 are allowed to execute in parallel.