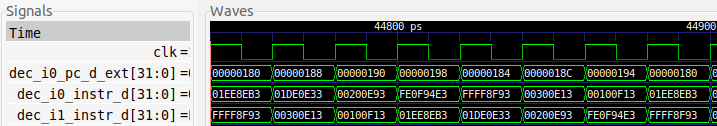
# TASKS

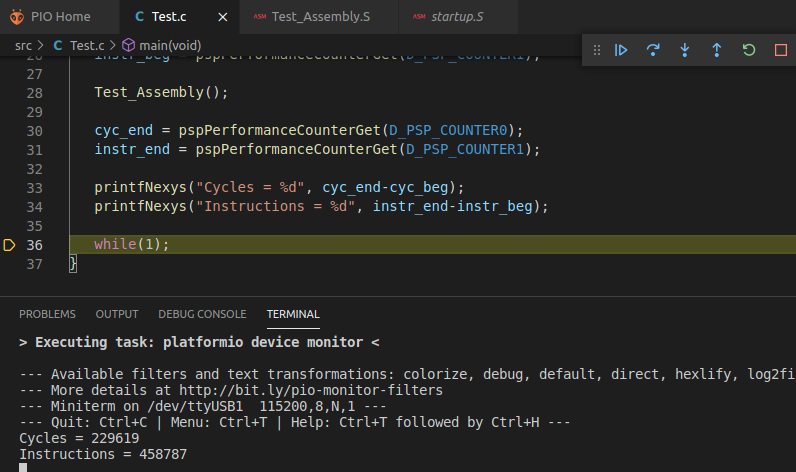
**TASK:** Replicate the simulation from Figure 5 on your own computer. You can use the *.tcl* file provided in: *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_AL-AL/test\_Basic.tcl*.

Solution provided in the main document of Lab 15.

**TASK:** Remove all nop instructions in the example from Figure 2. Draw a figure similar to Figure 3 for two consecutive iterations of the loop, then analyse and confirm that the figure is correct by comparing it to a Verilator simulation, and finally compute the IPC by using the Performance Counters while executing the program on the board.



Each iteration takes 3.5 cycles to execute and there are no stalls.



The IPC is the ideal: IPC = 458 / 229 = 2.

**TASK:** In the example from Figure 2, remove all nop instructions and move the add t6,t6,-1 instruction after the add t3,t3,t4 instruction, and then re-examine the program both in simulation and on the board. In this reordered program, the two dependent add instructions (add t4,t4,t5 and add t3,t3,t4) arrive at the Decode stage in the same cycle, and this has an impact in performance. Explain the impact of these changes, using both simulation and execution on the board.

Test similar situations where you replace the dependent add instruction for other dependent instructions, such as:

- add t4,t4,t5

**mul** t3,t3,t4

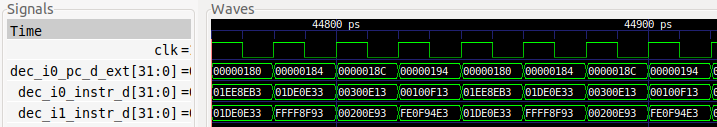
- add t4,t4,t5

**div** t3,t3,t4

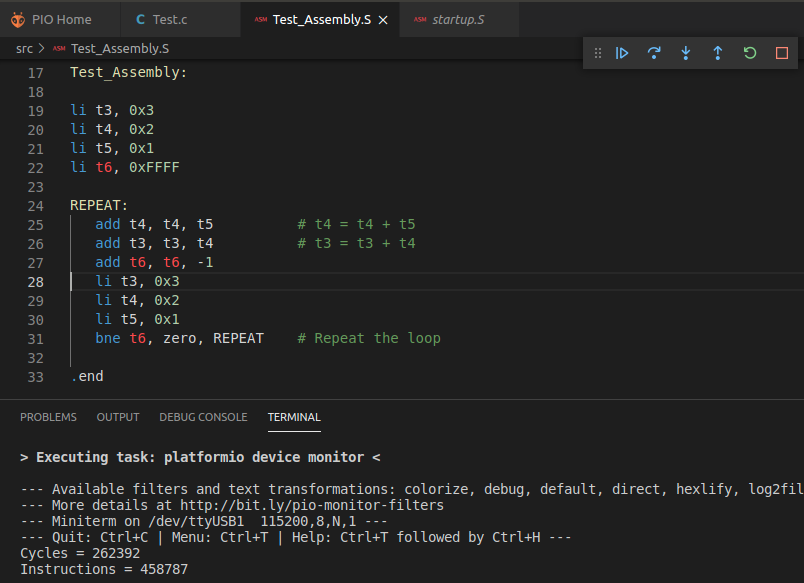
- add t4,t4,t5

**lw** t3, 0(t4)

* **Two add instructions:**

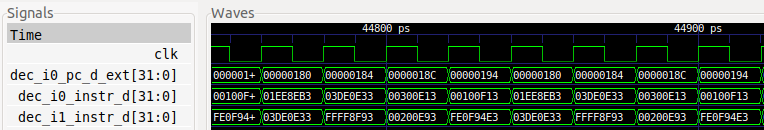


Now each iteration takes 4 cycles to execute as the dependent add instruction must stall for 1 cycle, since one of its input operands is not available until the first add executes in EX1 and forwards the result.



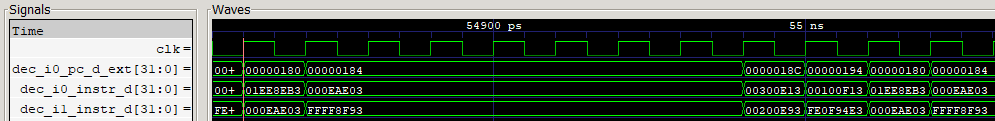
Now the IPC is not the ideal: IPC = 458 / 262 = 1.75

* **add instruction followed by mul instruction:**



Like before, the dependent mul instruction must stall for 1 cycle, since one of its input operands is not available until the first add executes in EX1 and forwards the result.

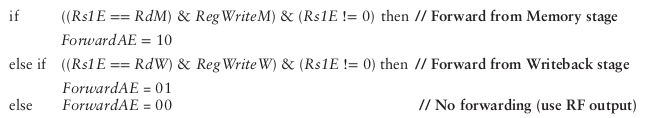
* **add instruction followed by lw instruction:**



Like before, the dependent lw instruction must stall for 1 cycle, since one of its input operands is not available until the first add executes in EX1 and forwards the result.

**TASK:** Compare the previous equations with the ones explained for the pipelined processor from DDCARV.

Equations for the pipelined processor from DDCARV:



**TASK:** Analyse the Verilog code to explain how the computation of the previous equation is performed. You must inspect the following lines of module **dec\_decode\_ctl**.

Solution not provided.

**TASK:** Write equations (similar to the one above) for other control bits of i0\_rs2bypass[9:0], i0\_rs1bypass[9:0], i1\_rs2bypass[9:0], and i1\_rs1bypass[9:0].

You can obtain the equations from the following lines in module **dec\_decode\_ctl**:

* 2372 – 2417
* 1721 – 1767
* 1497 – 1544
* 1130 – 1131 and 1255 – 1256

**TASK:** Replicate the simulation from Figure 8 on your own computer. You can use the *.tcl* file provided in: *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_AL-AL/test\_Advanced.tcl*.

Solution provided in the main document of Lab 15.

**TASK:** For the program from Figure 2, perform the same analysis as in Figure 8 for situations where the two dependent instructions are placed at different distances one from each other. You can control the distance by changing the number of nops between the two dependent add instructions.

Also, create other examples where the first input operand is the one that receives the forwarding data.

You can also create other examples where the two add instructions are executing through the I1 Pipe and confirm that the behaviour is the same.

Finally, substitute the dependent add instruction (add t3,t3,t4) for other dependent instructions executing though other pipes and analyse the results of the simulation. For example, instead of the second add instruction, you could include one of the following instructions:

- lw t3, (t4) (force the read value to come from the DCCM as explained in Lab 13)

- mul t3, t3, t4

- div t3, t3, t4

Example new simulated program: Bypass from the EX2 stage to the Decode stage:

1a0: 01ee8eb3 **add t4,t4,t5**

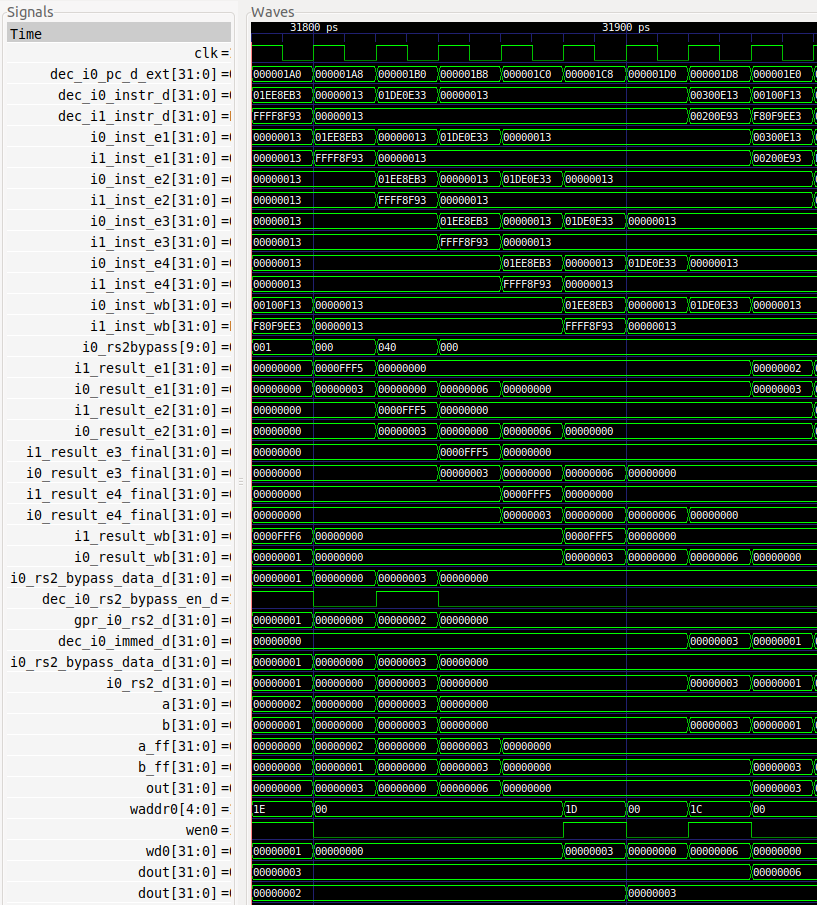
1a4: ffff8f93 addi t6,t6,-1

1a8: 00000013 nop

1ac: 00000013 nop

1b0: 01de0e33 **add t3,t3,t4**

1b4: 00000013 nop

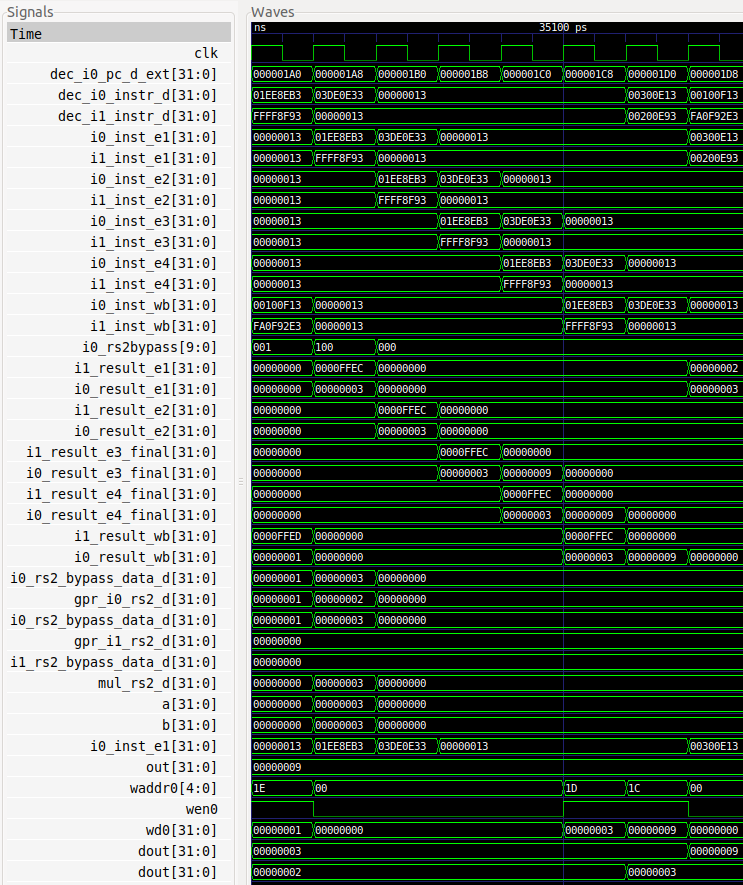


Example new simulated program: Execution of mul instead of the second add:

1a0: 01ee8eb3 **add t4,t4,t5**

1a4: ffff8f93 addi t6,t6,-1

1a8: 03de0e33 **mul t3,t3,t4**



**TASK:** Add logic to Figure 10 to produce the first input operand (a) of the Secondary ALU in the I0 Pipe.

Solution not provided.

**TASK:** Replicate the simulation from Figure 12 on your own computer. You can use the *.tcl* file provided at: *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_Close-LW-AL/scriptLoad.tcl*

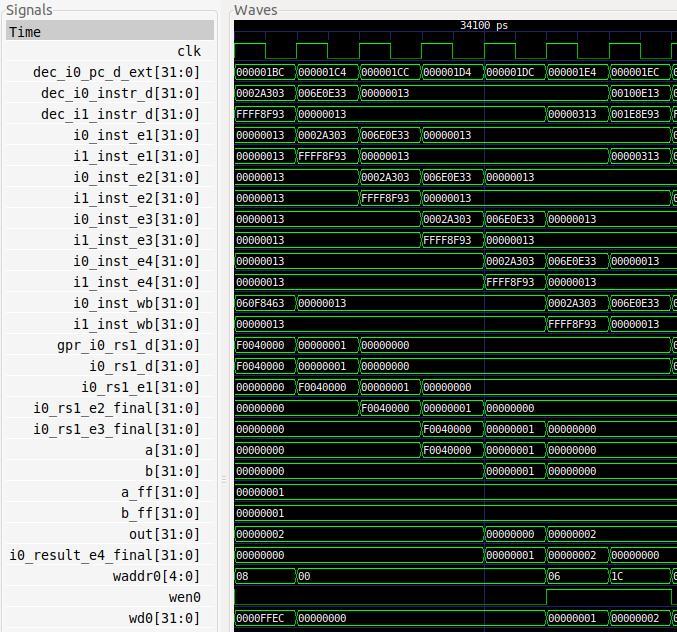
Solution provided in the main document of Lab 15.

**TASK:** Draw a figure similar to Figure 3 for the example from Figure 11.

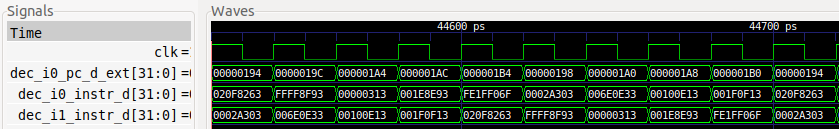
Solution not provided.

**TASK:** In the previous example, analyse how the first operand for the add t3, t3, t1 instruction (t3) is obtained. You can use the *.tcl* file provided at: *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards\_Close-LW-AL/scriptLoad\_FirstOperand.tcl*

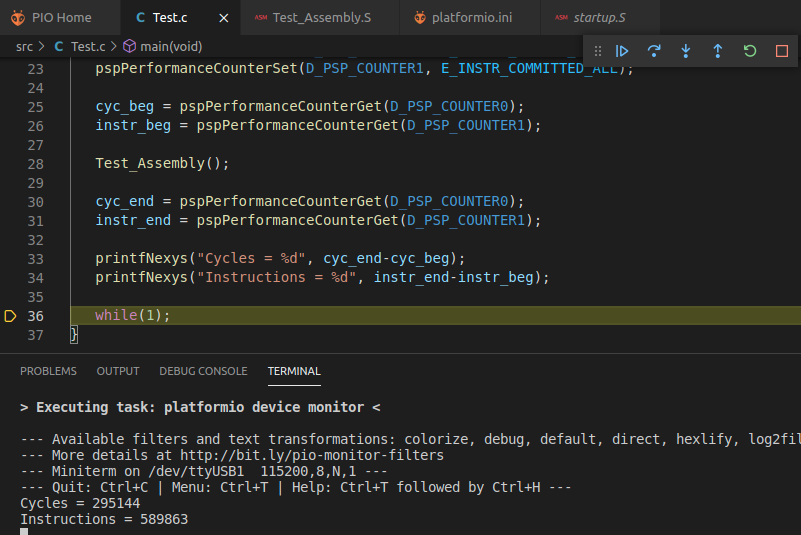
The first operand does not depend on previous instructions, thus it is obtained directly from the Register File.



**TASK:** Remove the nop instructions in the example from Figure 11 and obtain the IPC using the HW Counters.

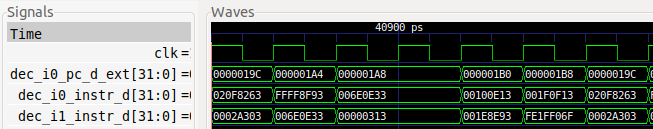


2 iterations in 9 cycles. Each iteration contains 9 instructions. Thus: IPC = 18 / 9 = 2

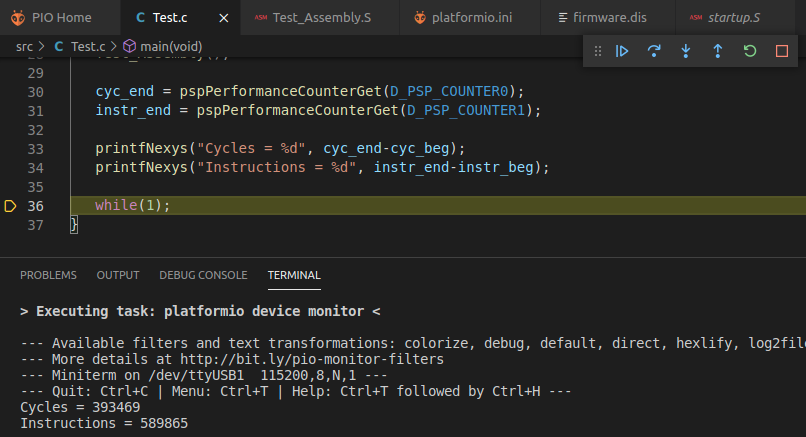


Thanks to the forwarding logic and the Secondary ALU, there are no stalls and the IPC is the ideal: IPC = 5898 / 2951 = 1.998

**TASK:** Disable the Secondary ALU as explained in Lab 11 and analyse the example from Figure 11 both with a Verilator simulation and with an execution on the board.



After the lw (0x0002a303), the dependent add instruction (0x006e0e33) is stalled for some cycles. 1 iteration takes 6 cycles.

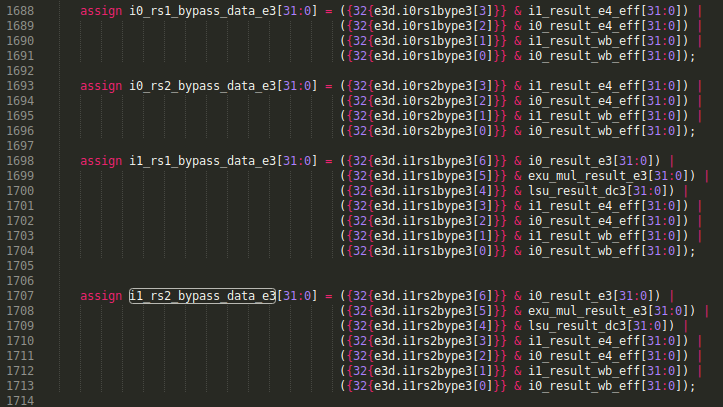


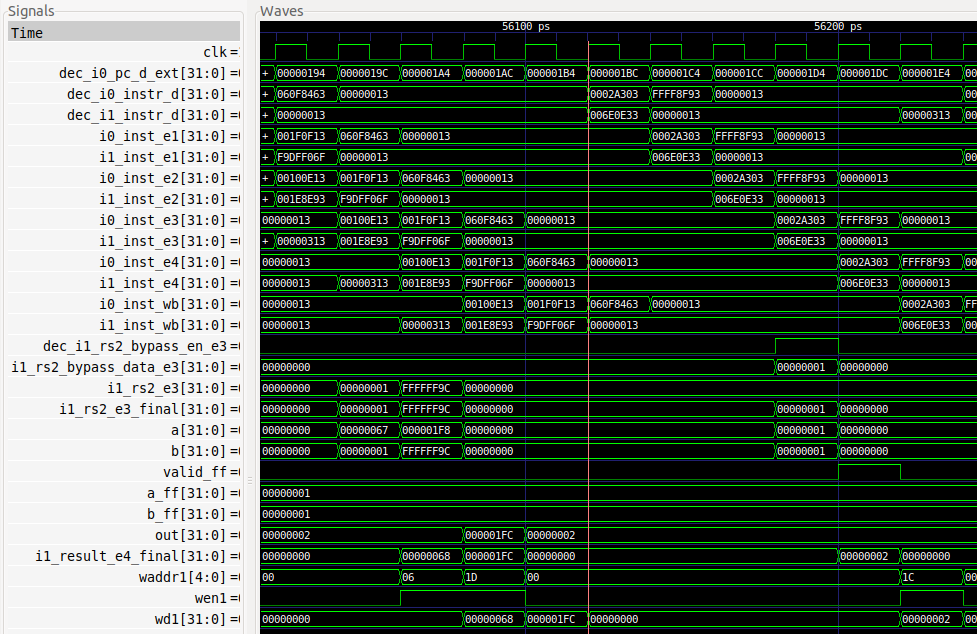
The IPC now is far from the ideal now, due to the stall created by the lw-add data hazard: IPC = 5898 / 3934 = 1.499

**TASK:** In the example from Figure 11, move the add t6,t6,-1 instruction after the add t3,t3,t1 instruction and re-examine the program both in simulation and on the board.

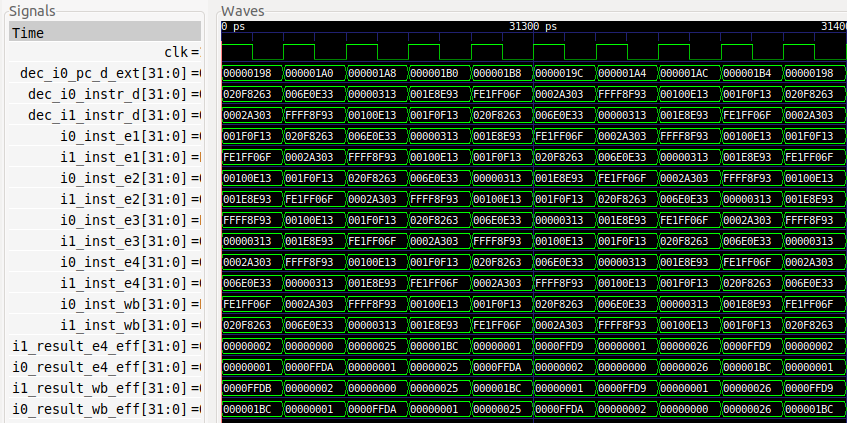
You can use the program provided at: *[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\_Solutions/Lab15/DataHazards\_SameCycle-LW-AL*

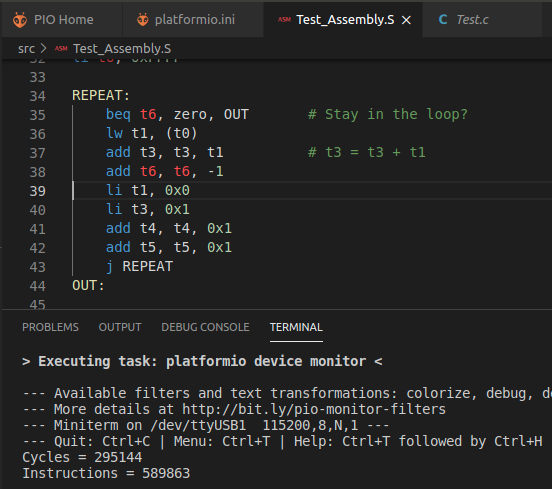
The add t3,t3,t1 instruction depends on the load and they both execute in parallel through the two ways. The add reexecutes in the Secondary ALU in this case. Note that Way-1 can receive the input operand from the other pipes so that no cycles are lost in this situation.





If we remove the nop instructions and perform the simulation and execute on the board, we obtain:





Thanks to the forwarding logic and the Secondary ALU, there are no stalls and the IPC is the ideal: IPC = 5898 / 2951 = 1.998

# EXERCISES

1. Modify the program used in Section 3 by adding an extra arithmetic-logic instruction that depends on the result of the add instruction. For example, you can replace the loop from Figure 11 with the following code, where a new AND instruction has been included (**and t3, t4, t3**), and where we have slightly reordered the code by moving forward instruction **add t5, t5, 0x1**:

REPEAT:

beq t6, zero, OUT

INSERT\_NOPS\_9

lw t1, (t0)

add t6, t6, -1

add t3, t3, t1

**add t5, t5, 0x1**

**and t3, t4, t3**

INSERT\_NOPS\_8

li t1, 0x0

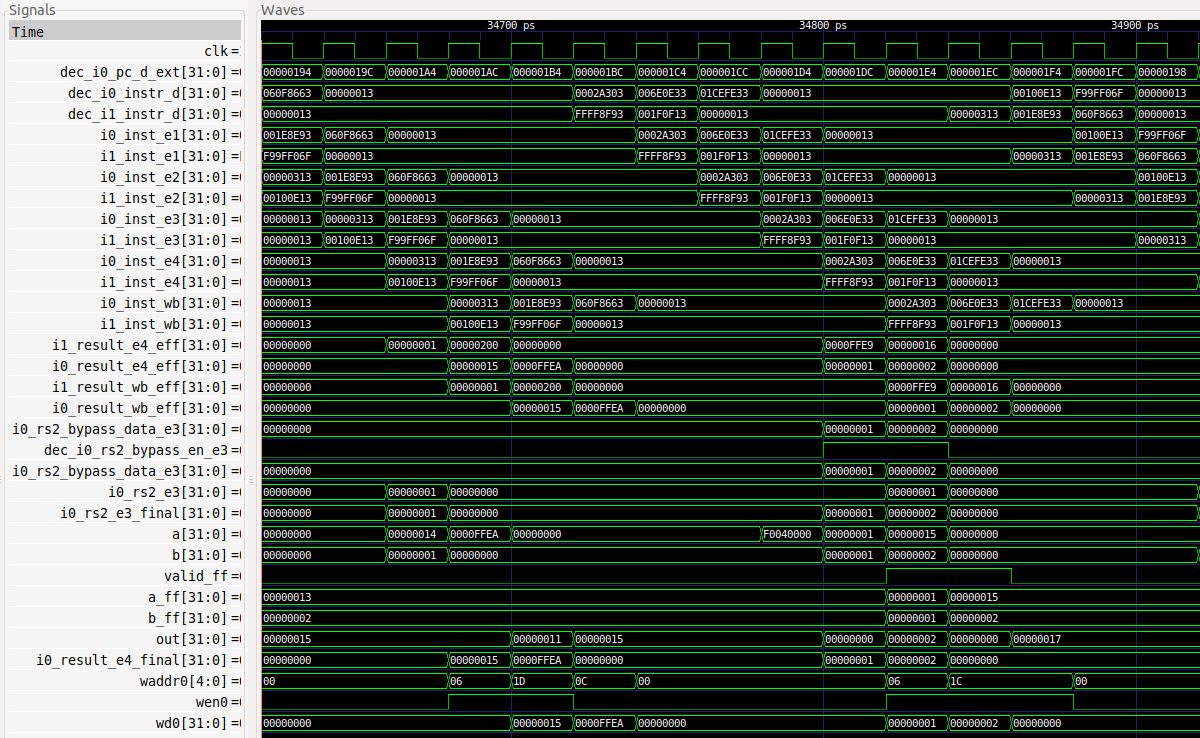
li t3, 0x1

add t4, t4, 0x1

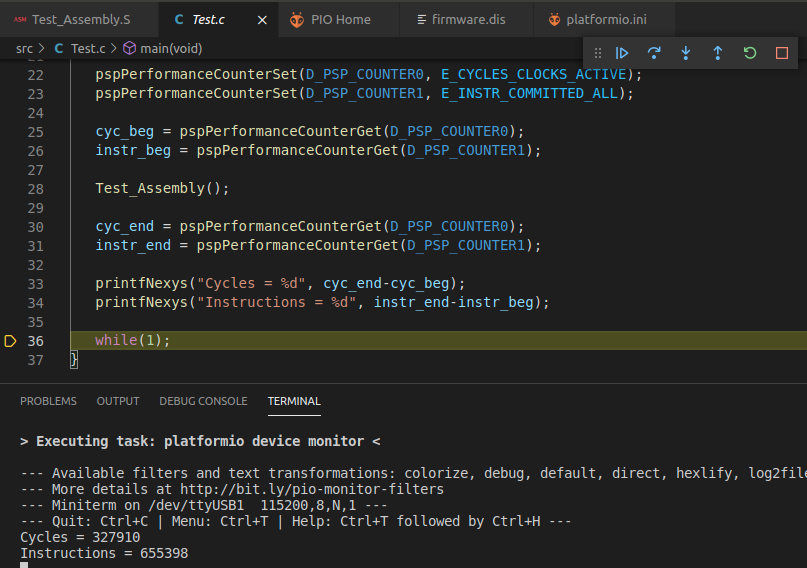
j REPEAT

OUT:

Analyse the Verilator simulation and explain how data hazards are handled for the new A-L instruction. Then remove all nop instructions and analyse the results provided by the HW counters.



Both the dependent add and and instructions use the secondary ALU for recalculating the result. Note that the second input operand for the and instruction is bypassed at EX3.



IPC = 6553 / 3279 = 1.998

Instructions executed per iteration: 655398 / 65535 = 10

Number of cycles per iteration: 327910 / 65535 = 5

Thanks to the forwardings and the secondary ALU the ideal IPC is achieved in this program.

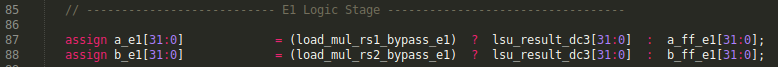
1. Analyse the same situation as the one described in Section 2.C for a mul instruction followed by an add instruction that uses the result of the multiplication. In the program from Figure 11 you can simply substitute the lw for a mul that writes to register t1.

Solution not provided.

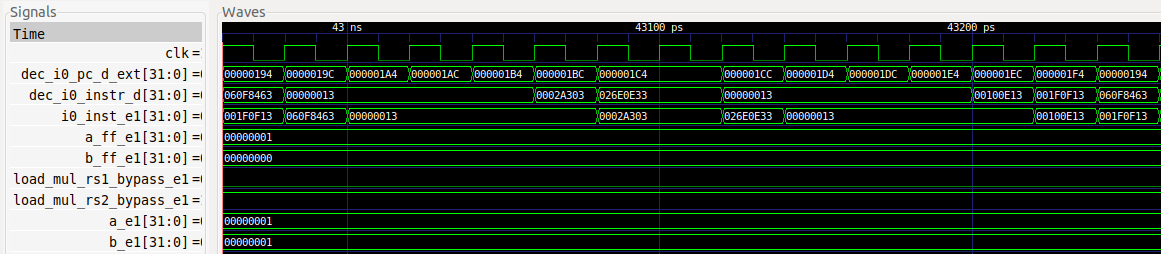
1. Analyse a situation with a lw instruction followed by a mul instruction that depends on the value read by the load. In the program from Figure 11 you can simply substitute the dependent add instruction for a mul instruction.

You can use the program provided at: *[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\_Solutions/Lab15/DataHazards\_Close-LW-MUL*

The mul instruction cannot be executed by the Secondary ALU. A new bypass path is implemented inside the multiplier (module **exu\_mul\_ctl**) that forwards the value read by a load to the M1 Stage.



This way only 1 cycle is lost due to this RAW dependency.



The second operand is bypassed from the value read by the load. This way, only 1 cycle is lost due to the dependency.

1. (*The following exercise is based on exercises 4.18*, *4.19*, *4.20* and *4.26 of [HePa].*)

Suppose you executed the code below on a version of the SweRV EH1 processor that does not handle data hazards (i.e., the programmer is responsible for addressing data hazards by inserting nop instructions where necessary). Add nop instructions to the code so that it will run correctly.

addi x11, x12, 5

add x13, x11, x12

addi x14, x11, 15

add x15, x13, x12

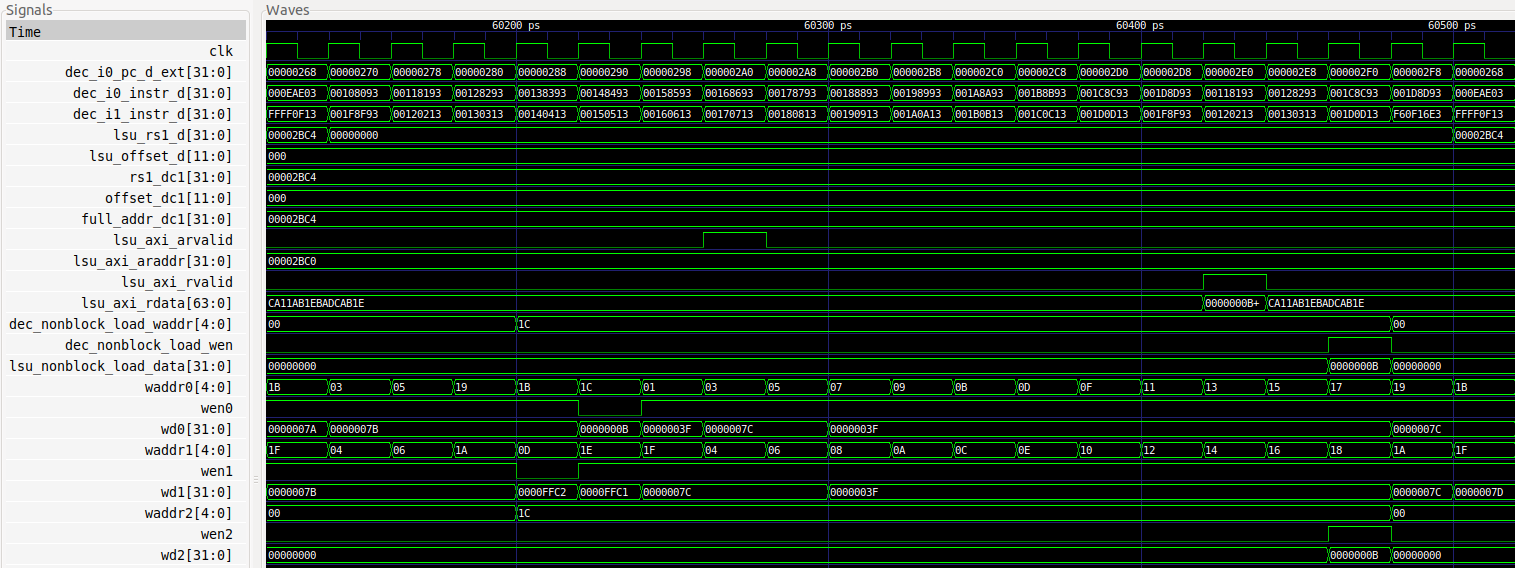
Then make up sequences of at least three assembly code snippets that exhibit different types of RAW data hazards. The type of RAW data dependence is identified by the stage that produces the result and the next instruction that consumes the result.

For each sequence, how many nops would need to be inserted and where, to allow your code to run correctly on a SweRV EH1 processor with no forwarding or hazard detection? What is the CPI if we use the forwarding available in SweRV EH1 and don’t insert nops?

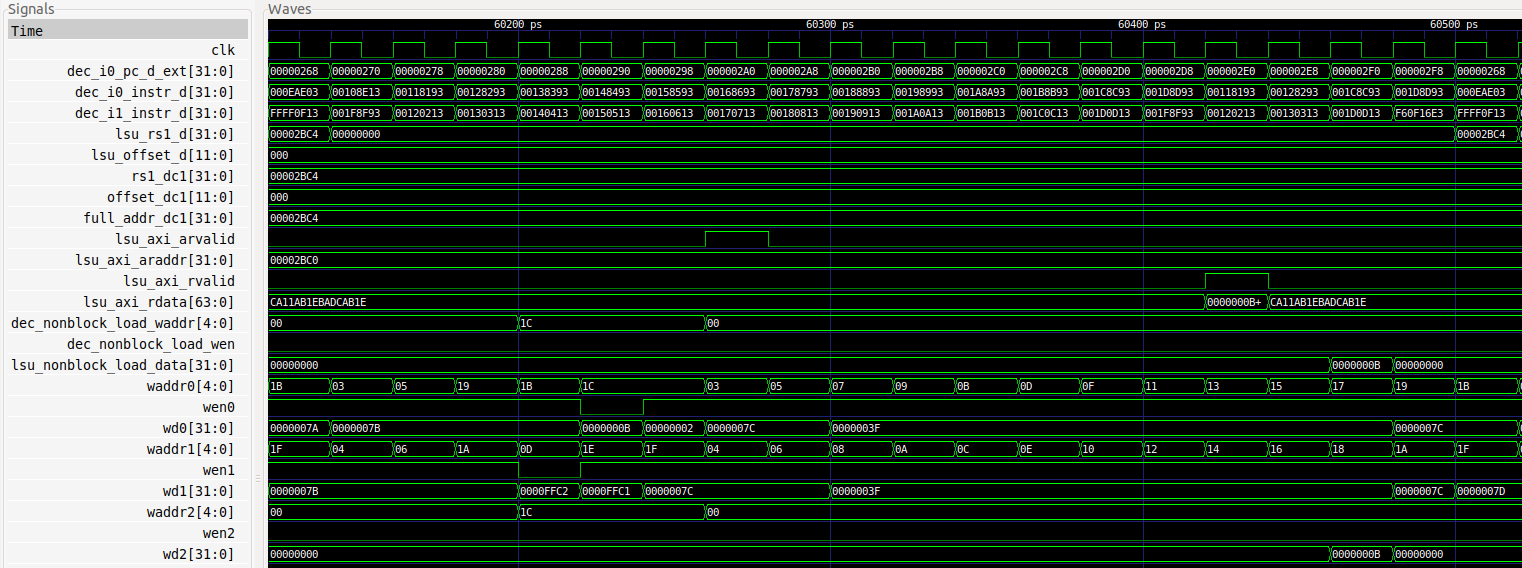
Solution not provided.

1. In the program Section 2.C of Lab 14 (available at *[RVfpgaPath]/RVfpga/Labs/Lab14/LW\_Instruction\_ExtMemory*), replace instruction add **x1**, x1, 1 with add **x28**, x1, 1. This introduces a WAW hazard between the modified add instruction and the non-blocking load at the beginning of the loop (lw x28, (x29)). Analyse in simulation how this hazard is handled in SweRV EH1, for which you can look at the value of signal wen2 in the Register File. Try to understand how this signal is computed in the Control Unit (module **dec**).

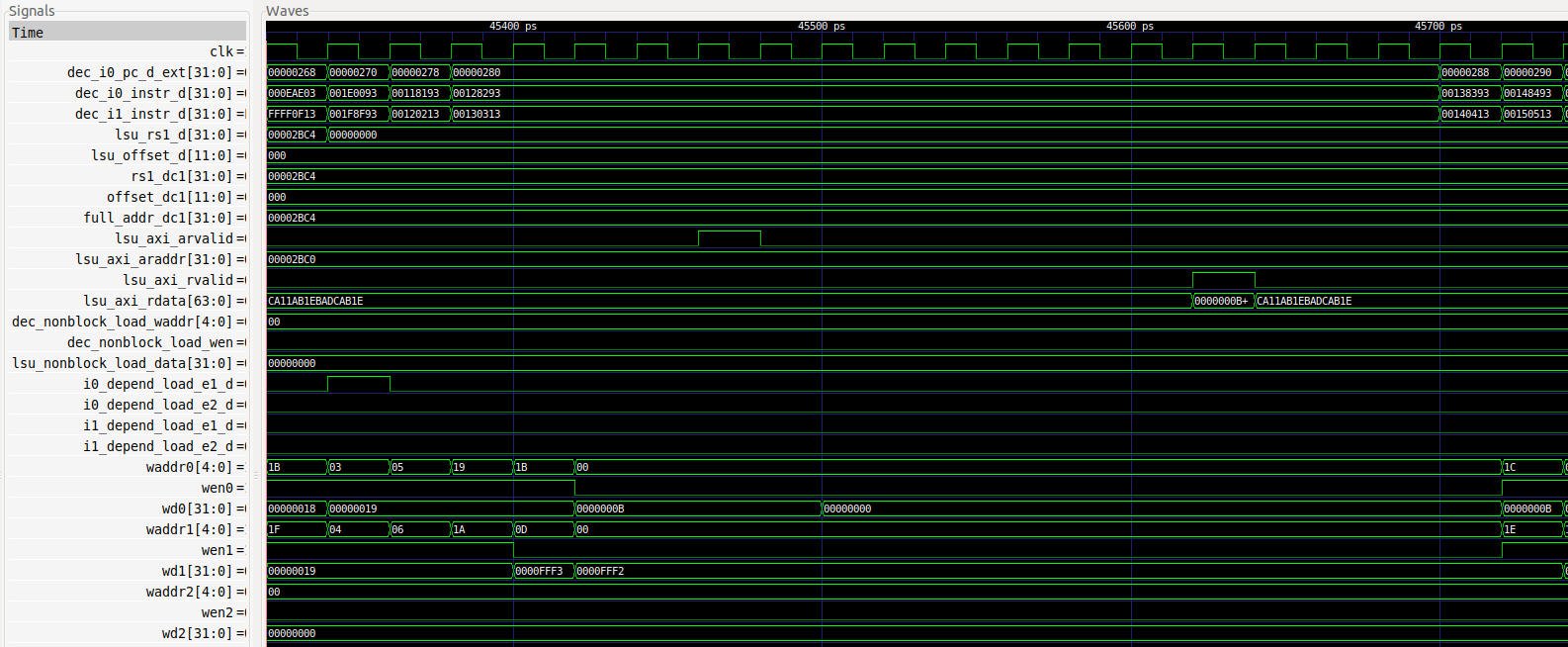
The simulation for the original program is the following. As we analysed in Lab 14, there is a cycle (highlighted in the figure) when 3 simultaneous writes are performed to the Register File (2 add instructions and a non-blocking load).



In the new program, shown below, where we replace instruction add **x1**, x1, 1 for instruction add **x28**, x1, 1, it is detected that a later instruction in program order modifies the same register, thus the write of the load is disabled (the wen2 signal does never go high), resolving the WAW data hazard.



1. In the program Section 2.C of Lab 14 (available at *[RVfpgaPath]/RVfpga/Labs/Lab14/LW\_Instruction\_ExtMemory*), replace instruction add x1, **x1**, 1 with add x1, **x28**, 1. This introduces a RAW hazard between the modified add instruction and the non-blocking load at the beginning of the loop (lw x28, (x29)). Analyse in simulation how this hazard is handled in SweRV EH1.



The RAW hazard is detected, the pipeline is stalled and the forwarding takes place as explained in the lab.

1. Finally, in the program Section 2.C of Lab 14 (available at *[RVfpgaPath]/RVfpga/Labs/Lab14/LW\_Instruction\_ExtMemory*), replace instruction add x1, **x1**, 1 with add x1, **x28**, 1, and instruction add **x7**, x7, 1 with add **x28**, x7, 1. This causes both a RAW and a WAW hazard. Analyse in simulation how these two hazards are handled in SweRV EH1.

Solution not provided.

1. **Store to Load Forwarding**

This is a very interesting situation that we have not analysed in this lab and that you will analyse in this exercise. When a store followed by a load access the same address, data can be forwarded from the store to the load within the core and DDR External Memory reading can be avoided, saving both time and power.

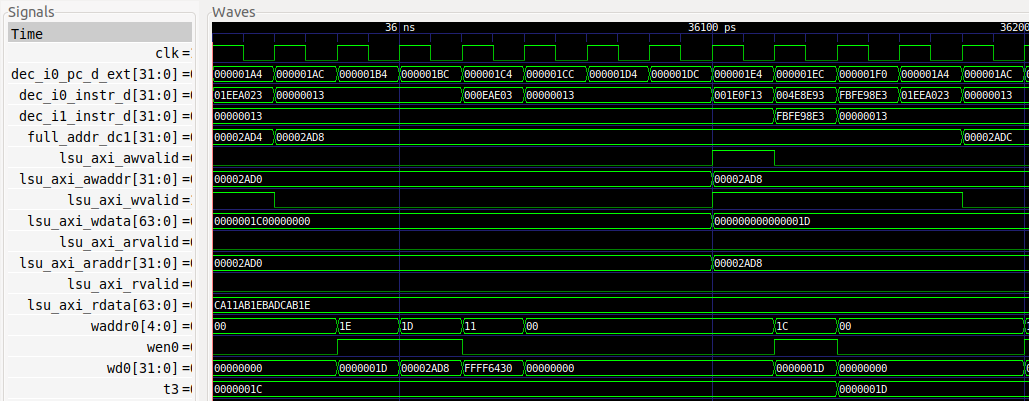
The logic that implements this forwarding is included in the LSU, and specifically in modules **lsu\_bus\_intf** and **lsu\_bus\_buffer**, which you must inspect in this exercise.

The PlatformIO project from *[RVfpgaPath]/RVfpga/Labs/Lab15/Sw-Lw-Forwarding* illustrates a store-load forwarding. A *.tcl* script is provided in that folder, which you can use for analysing a random iteration of the loop and understand how the forwarding is carried out.

**Verilator simulation:**

**lw t3, (t4)**

**sw t5, (t4)**



**1 2 3 4 5 6 7 8 9 10 11 12 13**

Analyse the simulation:

* **Cycle 1**: The sw instruction is decoded.
* **Cycle 5**: The lw instruction is decoded.
* **Cycles 2 to 11**: Signal full\_addr\_dc1 = 0x00002AD8 during the whole iteration. This happens because the address of the store and the address of the load are the same.
* **Cycle 9**: The store writes to the DDR External Memory through the write signals of the AXI bus.
  + lsu\_axi\_awvalid = 1
  + lsu\_axi\_awaddr = 0x00002AD8
  + lsu\_axi\_wvalid = 1
  + lsu\_axi\_wdata = 0x000000000000001D
* **Cycle 9, 10 and 11**: The load receives its data immediately through the bypass logic and writes it to the Register File. The read is never sent to the DDR memory (see the read enable signals of the AXI bus: lsu\_axi\_arvalid = lsu\_axi\_rvalid = 1):
  + waddr0 = 0x1C (which is register x28 = t3)
  + wen0 = 1
  + wd0 = 0x0000001D
  + t3 = 0x0000001D

**How is the forwarding performed inside the core?**

For the analysis of the store-load forwarding you must inspect two modules: **lsu\_bus\_intf** and **lsu\_bus\_buffer**.

1. In Section 4 of Lab 13 we analysed a read access to the DDR External Memory. We illustrated the SweRV EH1 structures involved in this access in Figure 16 of Lab 13:



Data is provided in signal: bus\_read\_data\_dc3.

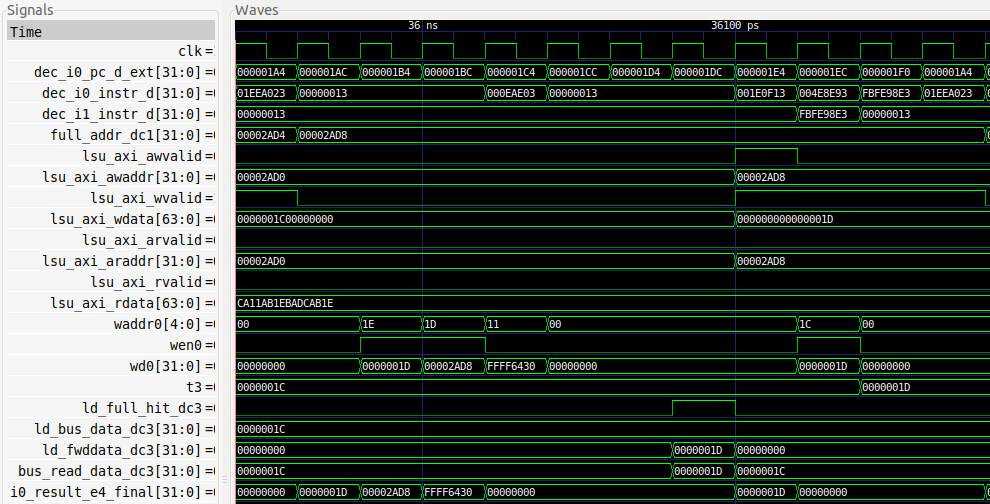
1. The value assigned to bus\_read\_data\_dc3 can come from the DDR memory or from the forwarding logic. For that purpose, module **lsu\_bus\_intf** includes a 2:1 multiplexer that selects between the value read from the DDR memory (ld\_bus\_data\_dc3) and the bypassed value (ld\_fwddata\_dc3).



We next include this 2:1 multiplexer in the previous figure:



1. If you add the inputs / output / control-signal of this multiplexer in the previous simulation, you will obtain:



You can see that the data provided to the load is bypassed from the store.

1. You can further analyse how the control signal (ld\_full\_hit\_dc3) and the bypassed value (ld\_fwddata\_dc3) are computed in modules **lsu\_bus\_intf** and **lsu\_bus\_buffer**.

# APPENDIX A

**TASK:** Replicate the simulation from Figure 15 on your own computer.

Solution provided in the main document of Lab 15.

**TASK:** Compare how the scenario above is handled in SweRV EH1 and in the pipelined processor from DDCARV.

Solution not provided.

**TASK:** If you compare carefully Figure 16 and Figure 6 of Lab 13, you will see that the value that the lw instruction reads into the Register File in Figure 6 of Lab 13 (signal lsu\_ld\_data\_corr\_dc3[31:0]) is different than the value forwarded by the lw in Figure 16 (signal lsu\_ld\_data\_dc3[31:0]). The difference between both values is that the former has been checked by the ECC logic in module **lsu\_ecc**, whereas the latter has not. Explain why it is not problematic that the value forwarded by the lw is not checked for errors.

If an error is detected in the data read by the load, the pipeline is stopped and flushed. Thus, both the load instruction and all subsequent instructions, some of which are the instructions that receive the incorrect forwarded value, are flushed and never commit.

**TASK:** In the example from Figure 14, remove all the nop instructions before the lw and after the add. Do not remove the 5 nops between the two dependent instructions. Analyse the simulation and then compute the IPC with the Performance Counters by executing the program on the board (it may seem awkward to keep nop instructions when measuring the IPC as they are useless instructions; however, the program itself is useless and our only aim here is to analyse data hazards and understand them).

Solution not provided.