THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 15

Data Hazards

1. INTRODUCTION

In this lab we deal with **data hazards**. As explained by Hennessy and Patterson in their 6th edition of "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. Assume instruction *i* is followed by instruction *j* in the program and both instructions use register *x.* Three types of data hazards can occur between *i* and *j*:

- ***Read After Write* (RAW) data hazard:** This is the most common type of hazard. It occurs when instruction *j* reads register *x* before instruction *i* writes register *x*. Thus, instruction *j* would use the wrong value of *x*.

- ***Write After Read* (WAR) data hazard**: WAR hazards occur when instruction *j* writes *x* and instruction *i* reads *x*, and instruction *j* is reordered to occur before *i*. Thus, instruction *i* reads the incorrect value of *x*. This hazard only occurs when instructions are reordered, which only rarely happens in SweRV EH1; specifically, WAR hazards never happen in SweRV EH1.

- ***Write After Write* (WAW) data hazard:** WAW hazards occur when instructions are reordered and instruction *j* writes *x* before instruction *i* writes *x*. This hazard only occurs when instructions are reordered, which only rarely happens in SweRV EH1; however, in the case of non-blocking loads, a WAW hazard could occur as we will analyse later in this lab.

In the following sections we analyse how RAW data hazards are resolved in the SweRV EH1 processor, and then we describe tasks and exercises related to RAW hazards. We also describe an exercise analysing a situation when a WAW hazard takes place.

NOTE: Before analysing the SweRV EH1 data hazard logic, we recommend reading Section 7.5 in DDCARV about how hazards are resolved in the pipelined processor. Data hazards, specifically, are analysed in Section 7.5.3. Although the pipelined processor shown in the book is simpler than SweRV EH1, data hazards are resolved similarly in both processors.

2. SOLVING DATA HAZARDS WITH FORWARDING AT THE DECODE STAGE

As explained in Section 7.5.3 of DDCARV, some RAW data hazards can be solved by forwarding (also called bypassing) a result from an instruction executing in an advanced pipeline stage to a dependent instruction executing in an earlier pipeline stage. This requires adding multiplexers in front of the Functional Units (ALUs, Multiplier, Adder that computes the Effective Address in DC1, etc.) to select their operands from either the Register File or from subsequent stages.

Figure 1 extends the Decode stage shown in Figure 4 of Lab 11 with the bypass values. The Forwarding Logic produces bypass (i.e., forwarded) for each of the two source operands in each of the Ways:

- **Way-0:**
  - First input operand: `i0_rs1_bypass_data_d[31:0]`
  - Second input operand: `i0_rs2_bypass_data_d[31:0]`

- **Way-1:**
  - First input operand: `i1_rs1_bypass_data_d[31:0]`
  - Second input operand: `i1_rs2_bypass_data_d[31:0]`

These four inputs are distributed to the 3:1 and 4:1 multiplexers that determine the input operands for each of the Execution stages pipeline paths. For the sake of clarity in Figure 1, signals are connected by name. The inputs to the Forwarding Logic are the results produced by previous program instructions that are more advanced in the pipeline, as we will see below.

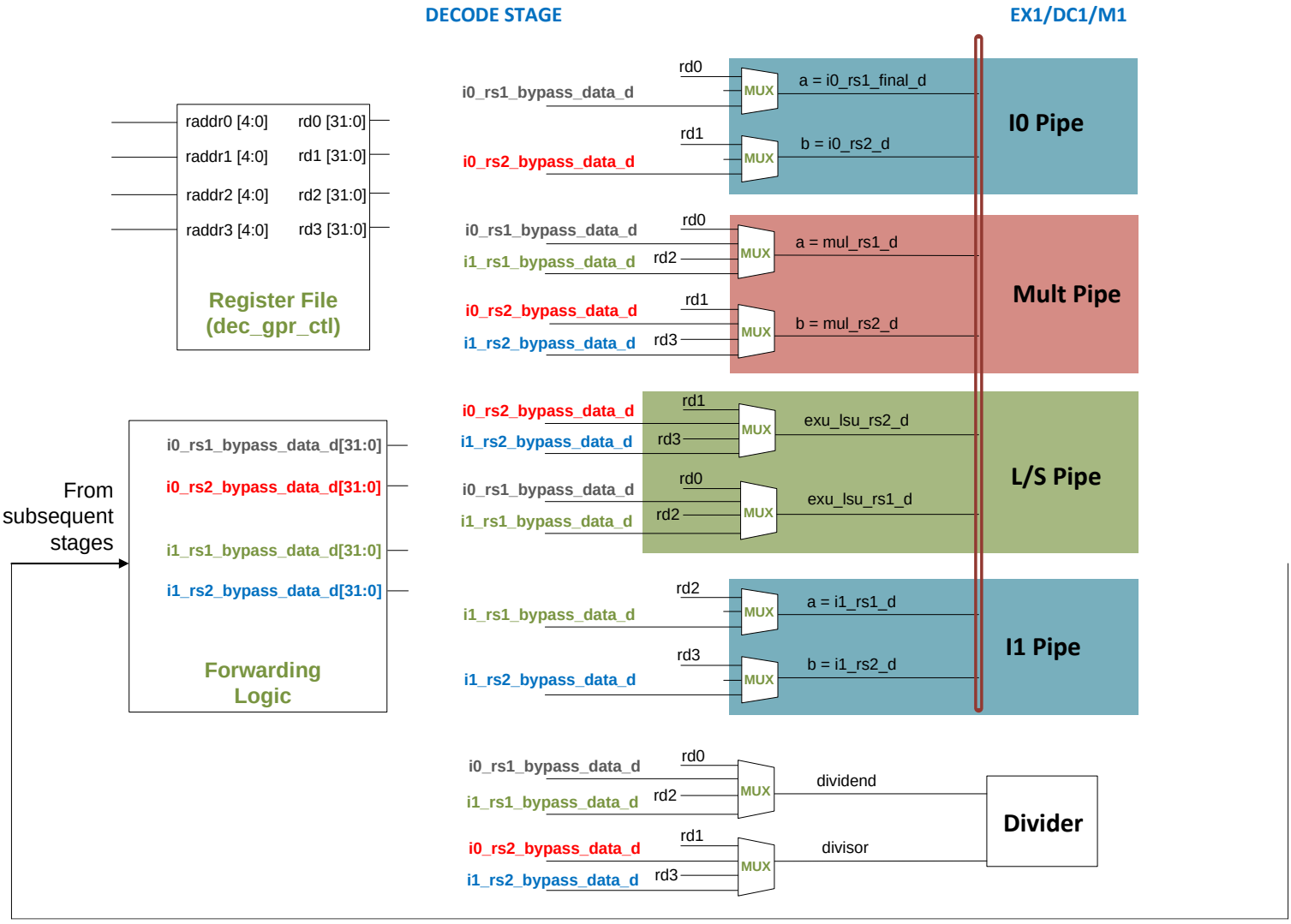


Figure 1. Bypass inputs to the Functional Units.

Many forwarding paths exist in the SweRV EH1 processor – in this section we focus on a specific path and analyse it in detail. Then, in the tasks and exercises, you will inspect other cases. We analyse the situation of two dependent A-L instructions executing simultaneously and how RAW data hazards are resolved. As we did in Labs 12 and 13, we start with a basic study (Section 2.A) and then proceed to an advanced analysis (Section 2.B). You may choose to complete the basic section only or to complete both sections.

We will work with the example shown in Figure 2, that executes two `add` instructions contained within a loop that repeats for 0xFFFF iterations. The first `add` instruction writes a value to `t4` and the second `add` instruction uses `t4` as its second input operand. An independent `add` instruction (`add t6, t6, -1`), which is the instruction that updates the loop index) is inserted in between the two `add` instructions to force the dependent `add` instructions use the same way of the processor.

```
.globl Test_Assembly

.text
Test_Assembly:

li t3, 0x3
li t4, 0x2
li t5, 0x1
li t6, 0xFFFF

REPEAT:
   INSERT_NOPS_8
   add t4, t4, t5          # t4 = t4 + t5 (t4 = 2 + 1)
   add t6, t6, -1
   add t3, t3, t4          # t3 = t3 + t4 (t3 = 3 + 3)
   INSERT_NOPS_9
   li t3, 0x3
   li t4, 0x2
   li t5, 0x1
   bne t6, zero, REPEAT    # Repeat the loop

.end
```

Figure 2. RAW data hazard between two `add` instructions

Folder *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL* provides the PlatformIO project so that you can analyse, simulate, and modify the program as desired. Open the project in PlatformIO, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL/.pio/build/swervolf_nexys/firmware.dis*) you will see that the two `add` instructions that we are analysing are placed at addresses 0x000001A0 and 0x000001A8:

```
0x000001a0:      01ee8eb3                add   t4,t4,t5
0x000001a4:      ffff8f93                addi  t6,t6,-1
0x000001a8:      01de0e33                add   t3,t3,t4
```

A. Basic analysis of a RAW data hazard between A-L instructions

In the example that we are analysing, the second `add` instruction (`add t3,t3,t4`) needs to use the result of the first `add` instruction (`add t4,t4,t5`) as its second input operand. This result is available at the EX1 stage, from where it can be bypassed to the Decode stage and used by the second `add` instruction. In our example (Figure 2), all iterations are equal and `t4` is 2 initially and 3 after the first addition. This last value (3) is the one that the second addition must use as its second input operand, and not the value read from the Register File (which is 2 until the first `add` instruction reaches the Writeback stage and updates it).

Figure 3 illustrates the flow of the instructions of the example from Figure 2 through the SweRV EH1 pipeline for a random iteration of the loop. In cycle *i*, the value computed at the

EX1 stage of the I0 Pipe must be forwarded to the instruction which is at the Decode stage of Way-0, due to the RAW data hazard between the two `add` instructions under analysis.

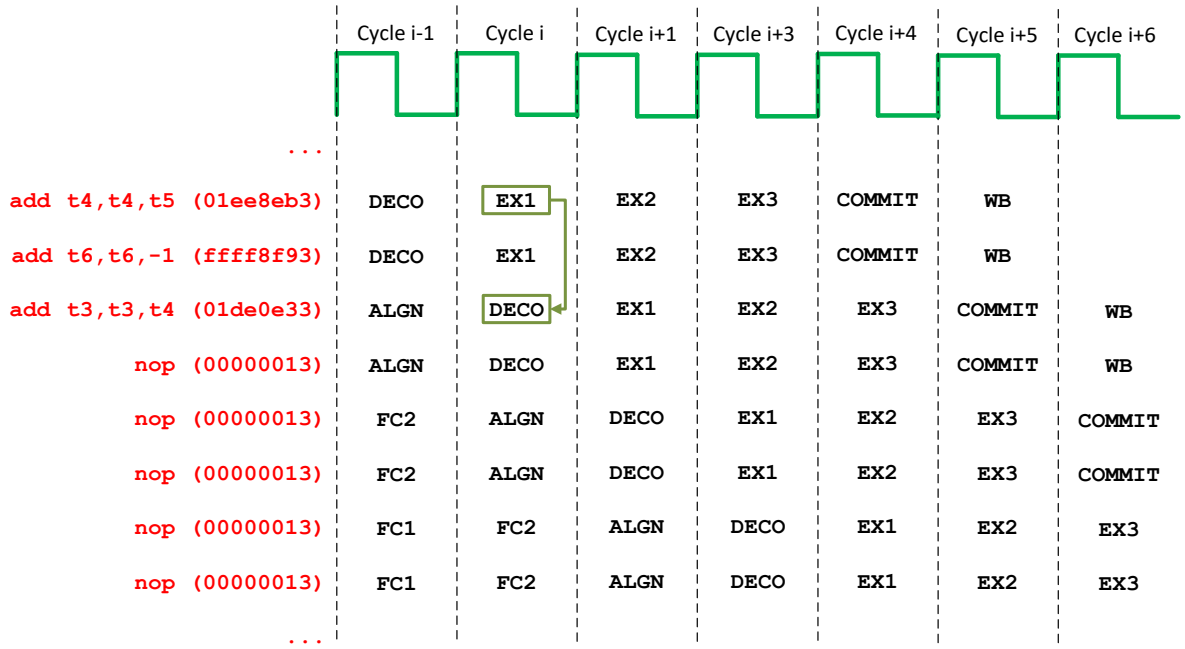


Figure 3. Execution of Figure 2 example code. Forwarding is performed in cycle *i*.

Figure 4 illustrates the SweRV EH1 Way-0 Decode and EX1 stages during cycle *i* of Figure 3. In this cycle, the first `add` instruction (`add t4,t4,t5`) is in the EX1 stage and the second `add` instruction (`add t3,t3,t4`) is in the Decode stage. As shown in the figure, the result of the first `add` instruction is bypassed to the Decode stage, it is selected by the Forwarding Logic (as we will analyse in detail in the following section), and it is used as the second input operand for the second `add` instruction.

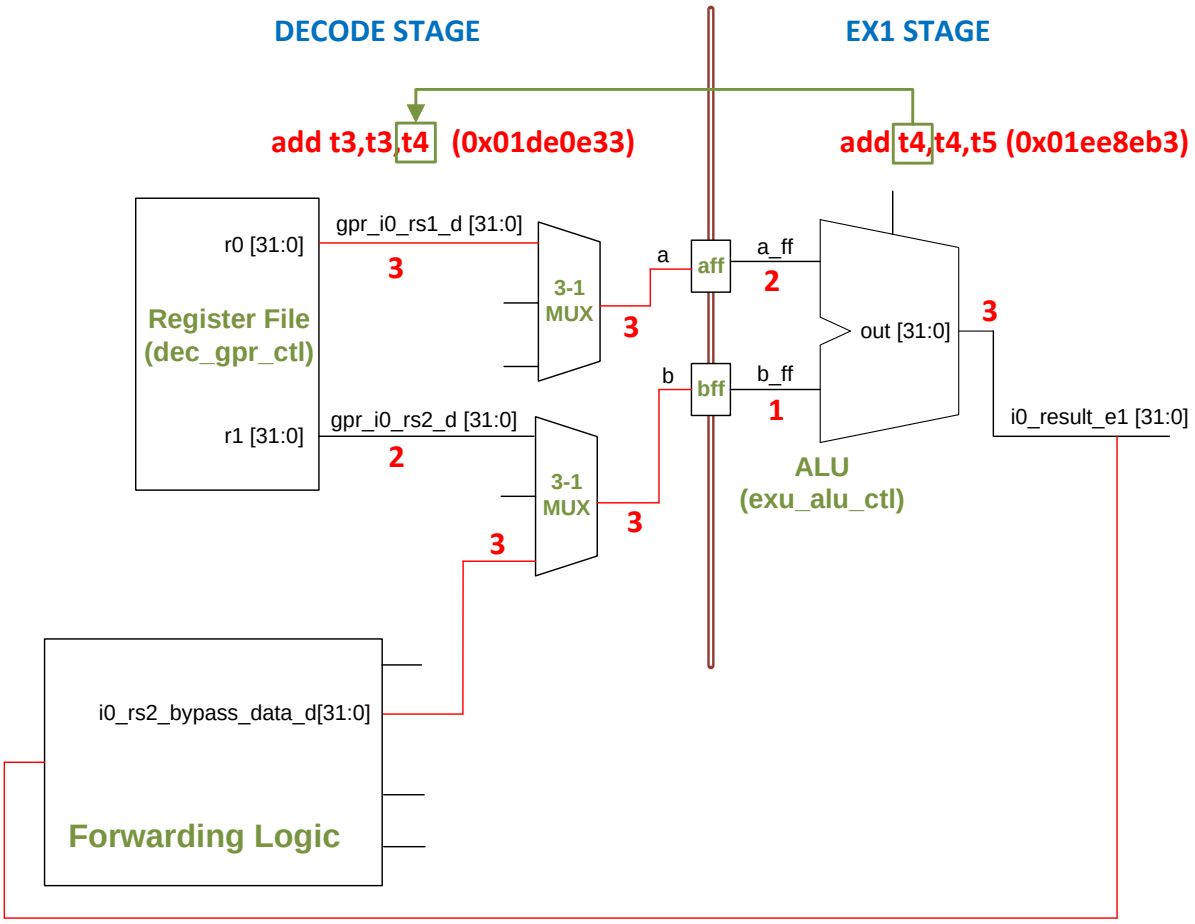


Figure 4. Result forwarded from EX1 to Decode (second operand) of Way 0

Finally, Figure 5 shows the simulation of the program from Figure 2 during cycles *i* and *i+1* of Figure 3.

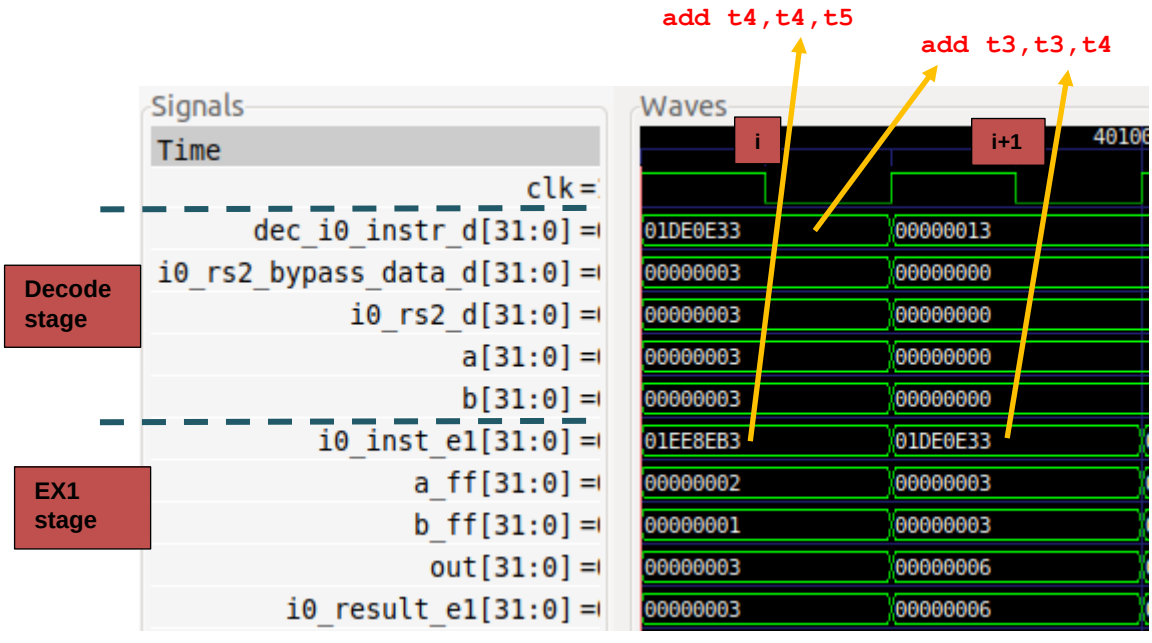


Figure 5. Simulation of Figure 2 example code

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*.

Analyse the simulation from Figure 5 and the diagram from Figure 4 at the same time.

- Instruction `add t4,t4,t5` (0x01ee8eb3):
  - In cycle *i*, this instruction is in the EX1 stage of the I0 Pipe (`i0_inst_e1` = 0x01ee8eb3). It computes the following addition in the ALU:
    `a_ff` (2) + `b_ff` (1) = `out` (3)
    The result of the addition is provided as an input to the Forwarding Logic at the Decode stage, as shown in Figure 4.

- Instruction `add t3,t3,t4` (0x01de0e33):
  - In cycle *i*, this instruction is in the Decode stage of Way-0 (`dec_i0_instr_d` = 0x01de0e33). The Forwarding Logic connects `i0_result_e1` with `i0_rs2_bypass_data_d`. The two 3:1 multiplexers select the input operands for the addition that will be calculated in the next cycle (cycle *i+1*) in the EX1 stage of the I0 Pipe; specifically:
    `a` = 3 (from the Register File)
    `b` = 3 (from the ALU output in the EX1 stage of the I0 Pipe, through the Forwarding Logic, signal `i0_rs2_bypass_data_d`)
  - In cycle *i+1*, this instruction is in the EX1 stage of the I0 Pipe (`i0_inst_e1` = 0x01de0e33). It computes the following addition in the ALU:
    `a_ff` (3) + `b_ff` (3) = `out` (6)

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.

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)
```

B. Advanced analysis of a RAW data hazard between A-L instructions

i. Theoretical explanation

Figure 6 extends the diagrams from Figure 1 and Figure 4 by adding a 10:1 multiplexer (surrounded by a blue square in Figure 6) that produces signal `i0_rs2_bypass_data_d`, which in our example from Figure 2 provides the second input operand for the second `add` instruction (`add t3,t3,t4`). This 10:1 multiplexer is implemented inside the Forwarding Logic box shown in Figure 1 and Figure 4.

The figure also shows the input connections of this 10:1 multiplexer. The bypassed value can come from an instruction executing through Way 0 or Way 1. Thus five forwarding paths are necessary per way. Specifically, the inputs to the 10:1 multiplexer can come from any of the subsequent stages (EX1, EX2, EX3, Commit, and Writeback) of Way 0 or Way 1. For the sake of simplicity, we connect the five inputs coming from Way 0 using wires, whereas the 5 inputs coming from Way 1 are connected by name.

Three additional 10:1 multiplexers inside the Forwarding Logic compute the three other source operands: signals `i0_rs1_bypass_data_d`, `i1_rs1_bypass_data_d` and `i1_rs2_bypass_data_d`. However, we do not show them in the figure as they are not used in the example that we analyse in this section (Figure 2). All four multiplexers can be found in lines 2429-2473 of module **dec_decode_ctl**.

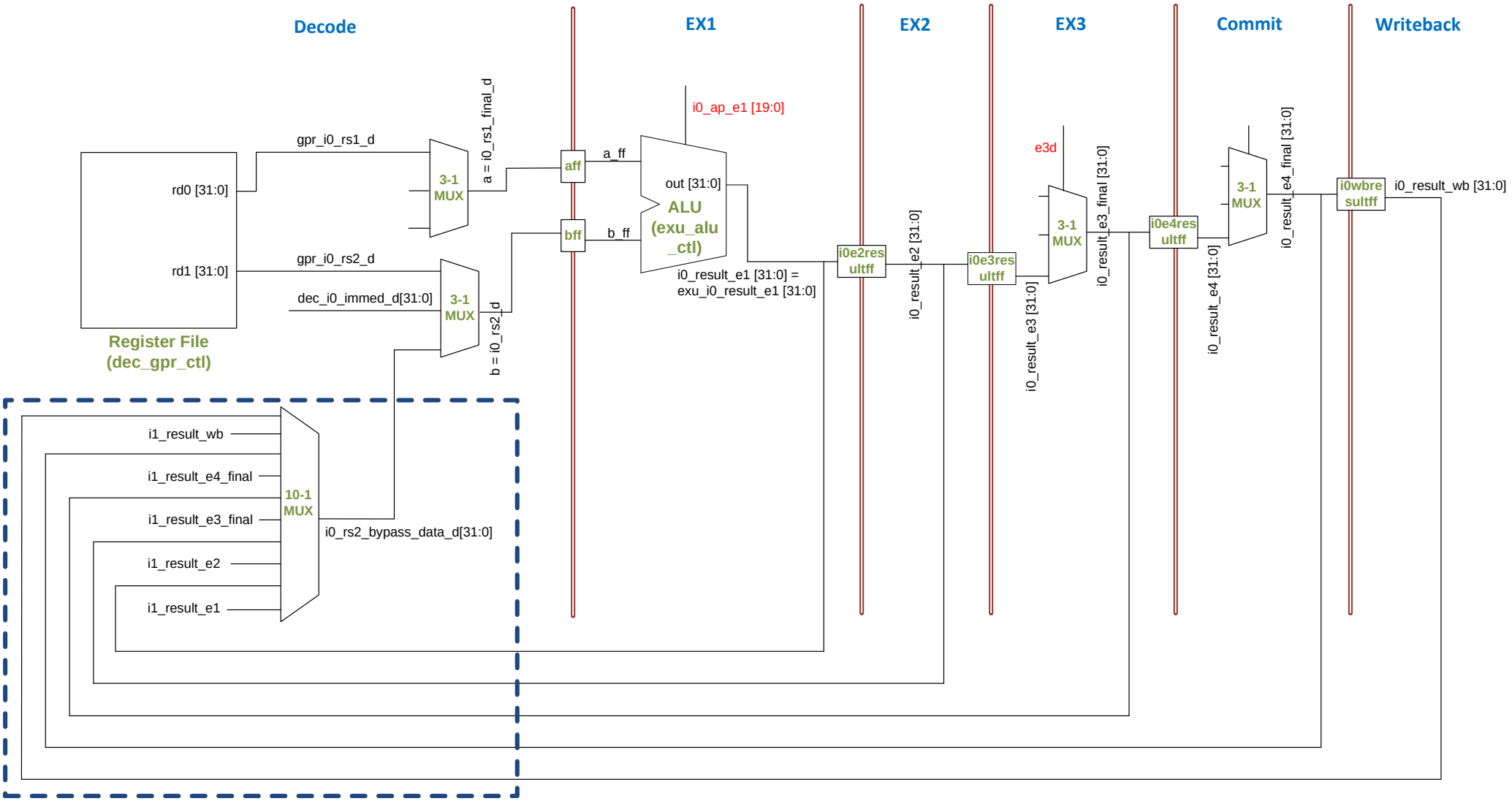
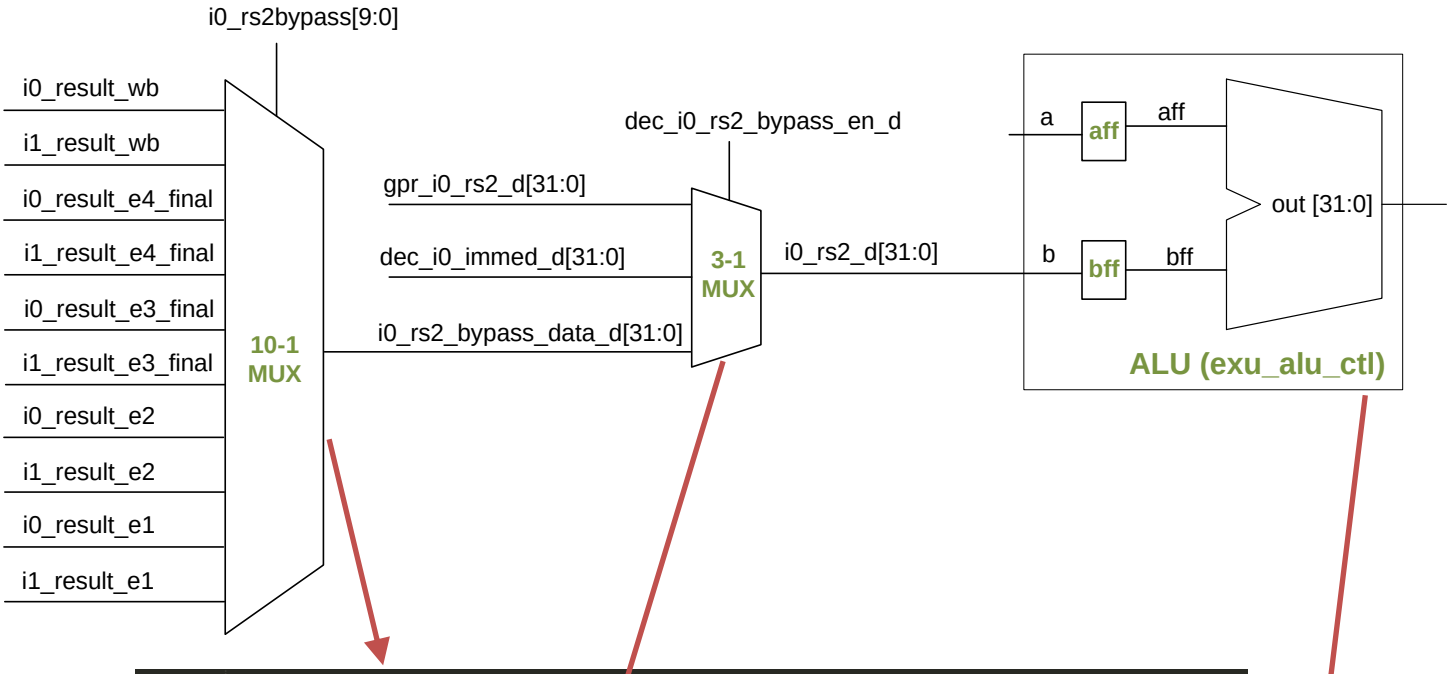


Figure 6. I0 Pipe including the Forwarding Logic used for the second input source of the ALU




```
2441     assign i0_rs2_bypass_data_d[31:0] = ({32{i0_rs2bypass[9]}} & i1_result_e1[31:0]) |
2442                                         ({32{i0_rs2bypass[8]}} & i0_result_e1[31:0]) |
2443                                         ({32{i0_rs2bypass[7]}} & i1_result_e2[31:0]) |
2444                                         ({32{i0_rs2bypass[6]}} & i0_result_e2[31:0]) |
2445                                         ({32{i0_rs2bypass[5]}} & i1_result_e3_final[31:0]) |
2446                                         ({32{i0_rs2bypass[4]}} & i0_result_e3_final[31:0]) |
2447                                         ({32{i0_rs2bypass[3]}} & i1_result_e4_final[31:0]) |
2448                                         ({32{i0_rs2bypass[2]}} & i0_result_e4_final[31:0]) |
2449                                         ({32{i0_rs2bypass[1]}} & i1_result_wb[31:0]) |
2450                                         ({32{i0_rs2bypass[0]}} & i0_result_wb[31:0]);
```

(Module **dec_decode_ctl**)

```
286     assign i0_rs2_d[31:0]       = ({32{~dec_i0_rs2_bypass_en_d}} & gpr_i0_rs2_d[31:0]) |
287                                   ({32{~dec_i0_rs2_bypass_en_d}} & dec_i0_immed_d[31:0]) |
288                                   ({32{ dec_i0_rs2_bypass_en_d}} & i0_rs2_bypass_data_d[31:0]);
```

(Module **exu**)

```
401     exu_alu_ctl i0_alu_e1 (.*,
402                            .freeze        ( freeze                    ),   // I
403                            .enable        ( i0_e1_ctl_en              ),   // I
404                            .predict_p     ( i0_predict_newp_d         ),   // I
405                            .valid         ( dec_i0_alu_decode_d       ),   // I
406                            .flush         ( exu_flush_final           ),   // I
407                            .a             ( i0_rs1_final_d[31:0]      ),   // I
408                            .b             ( i0_rs2_d[31:0]            ),   // I
409                            .pc            ( dec_i0_pc_d[31:1]         ),   // I
410                            .brimm         ( dec_i0_br_immed_d[12:1]   ),   // I
411                            .ap            ( i0_ap_e1                  ),   // I
412                            .out           ( exu_i0_result_e1[31:0]    ),   // O
413                            .flush_upper   ( exu_i0_flush_upper_e1     ),   // O
414                            .flush_path    ( exu_i0_flush_path_e1[31:1] ),   // O
415                            .predict_p_ff  ( i0_predict_p_e1           ),   // O
416                            .pc_ff         ( exu_i0_pc_e1[31:1]        ),   // O
417                            .pred_correct  ( i0_pred_correct_upper_e1   )    // O
418                            );
```

(Module **exu**)

Figure 7. 10:1 and 3:1 multiplexers highlighted in Figure 6

Figure 7 zooms into the two multiplexers (10:1 and 3:1 multiplexers) from Figure 6 that compute the second input operand for the I0 Pipe ALU (`b`). The figure shows both a block diagram and the Verilog code where these multiplexers are implemented in modules **dec_decode_ctl** and **exu**.

NOTE: The two multiplexers from Figure 7 also exist in the processor from DDCARV. Data is forwarded to the Execute stage in that processor and fewer forwarding paths exist because it is not superscalar and has a shorter pipeline. You can analyse the forwarding paths in Figure 7.55 of DDCARV.

We next analyse the inputs, outputs, and control signal of the two multiplexers shown in Figure 7.

10:1 Multiplexer:

Output: The output of the 10:1 multiplexer is `i0_rs2_bypass_data_d[31:0]`. This signal contains the value that must be forwarded (bypassed) to the instruction in the Decode stage.

Inputs: The inputs to the 10:1 multiplexer are the results of previous instructions in the program that are executing at later stages (EX1, EX2, EX3, Commit, or Writeback). Five of these signals come from the I0 Pipe (as shown in Figure 6) and the other five signals come from the I1 Pipe (not shown in Figure 6), as the instruction in the Decode stage could potentially depend on an instruction executing in any of the two ways.

Control signal: The control signal (`i0_rs2bypass[9:0]`) selects which input is connected with the output of the multiplexer. It is formed by 10 bits, with at most one of them being high at the same time (they can all be zero if there is no data hazard). The multiplexer operates as follows:

- If `i0_rs2bypass[9]` == 1 → `i0_rs2_bypass_data_d` = `i1_result_e1`
- If `i0_rs2bypass[8]` == 1 → `i0_rs2_bypass_data_d` = `i0_result_e1`
- If `i0_rs2bypass[7]` == 1 → `i0_rs2_bypass_data_d` = `i1_result_e2`
- If `i0_rs2bypass[6]` == 1 → `i0_rs2_bypass_data_d` = `i0_result_e2`
- If `i0_rs2bypass[5]` == 1 → `i0_rs2_bypass_data_d` = `i1_result_e3_final`
- If `i0_rs2bypass[4]` == 1 → `i0_rs2_bypass_data_d` = `i0_result_e3_final`
- If `i0_rs2bypass[3]` == 1 → `i0_rs2_bypass_data_d` = `i1_result_e4_final`
- If `i0_rs2bypass[2]` == 1 → `i0_rs2_bypass_data_d` = `i0_result_e4_final`
- If `i0_rs2bypass[1]` == 1 → `i0_rs2_bypass_data_d` = `i1_result_wb`
- If `i0_rs2bypass[0]` == 1 → `i0_rs2_bypass_data_d` = `i0_result_wb`

In order to understand how this 10-bit control signal is computed, we explain the computation of signal `i0_rs2bypass[8]`, which is the one which goes high in our example from Figure 2 for the `add-add` bypass.

- If `i0_rs2bypass[8]` is 1, the bypassed value selected is `i0_result_e1`, which is the result of the instruction executing in the EX1 stage of the I0 pipe (see Figure 6).

- For EX1 to forward data to the Decode stage (both in the I0 pipe), the following conditions must occur (see Section 4 of the SweRVref document to review the control signals):

    - The second input operand of the instruction in the Decode stage is read from the

Register File, and it is not read from register zero. In the SweRV EH1 Control Unit, this occurs when `dec_i0_rs2_en_d` is 1. The corresponding Verilog code is:

```
dec_i0_rs2_en_d = i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
```

- The instruction in the EX1 stage of the I0 Pipe is valid:
  `e1d.i0v` == 1
- The destination register of the instruction in the EX1 stage (of the I0 pipe) and the second source register of the instruction in the Decode stage (of Way 0) are the same:
  `e1d.i0rd[4:0]` == `i0r.rs2[4:0]`
- The instruction in the EX1 stage (of the I0 Pipe) is an ALU operation:
  `i0_rs2_class_d.alu` == 1

- Taking all this into account, we can conclude that:

```
i0_rs2bypass[8] =
      (i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0))   &
      e1d.i0v                                &
      (e1d.i0rd[4:0] == i0r.rs2[4:0])        &
      i0_rs2_class_d.alu                     ;
```

TASK: Compare the previous equations with the ones explained 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**.

```
2384     assign i0_rs2bypass[9:0] = {   i0_rs2_depth_d[3:0] == 4'd1  &  i0_rs2_class_d.alu,
2385                                     i0_rs2_depth_d[3:0] == 4'd2  &  i0_rs2_class_d.alu,
2386                                     i0_rs2_depth_d[3:0] == 4'd3  &  i0_rs2_class_d.alu,
```

```
1733     assign {i0_rs2_class_d, i0_rs2_depth_d[3:0]} =
1734                                                    (i0_rs2_depend_i1_e1) ? { i1_e1c, 4'd1 } :
1735                                                    (i0_rs2_depend_i0_e1) ? { i0_e1c, 4'd2 } :
1736                                                    (i0_rs2_depend_i1_e2) ? { i1_e2c, 4'd3 } :
```

```
1509     assign i0_rs2_depend_i0_e1 = dec_i0_rs2_en_d & e1d.i0v & (e1d.i0rd[4:0] == i0r.rs2[4:0]);
```

```
1131     assign dec_i0_rs2_en_d = i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
```

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]`.

3:1 Multiplexer:

Output: The output of the 3:1 multiplexer is `i0_rs2_d[31:0]`. This signal is sent to the second input (`b`) of the ALU in Way 0.

Inputs: The inputs to the 3:1 multiplexer are:

- The value read from the Register File (`gpr_i0_rs2_d`).
- The Immediate value (`dec_i0_immed_d`), obtained from the instruction.
- The value bypassed from later stages (`i0_rs2_bypass_data_d`) obtained from the 10:1 multiplexer described earlier.

Control signal: The control signal to the 3:1 multiplexer (`dec_i0_rs2_bypass_en_d`) selects either:

- The value bypassed from later stages (`i0_rs2_bypass_data_d`), if `dec_i0_rs2_bypass_en_d` ==1
- Or the value coming from the Register File or the Immediate (`gpr_i0_rs2_d` and `dec_i0_immed_d`, respectively), if `dec_i0_rs2_bypass_en_d` == 0. It may look strange that the same signal selects two inputs; however, the signal that must not be selected (either `gpr_i0_rs2_d` or `dec_i0_immed_d`) is forced to zero in the Verilog code.

The select signal of the 3:1 multiplexer (`dec_i0_rs2_bypass_en_d`) is simply computed as the logical OR of the 10-bit control signal of the 10:1 multiplexer:

```
assign dec_i0_rs2_bypass_en_d = |i0_rs2bypass[9:0];
```

Thus, whenever the second input operand of an instruction depends on the result of an earlier instruction that is still executing (i.e. any of the 10 bits of signal `i0_rs2bypass[9:0]` is 1), `dec_i0_rs2_bypass_en_d` == 1 and the operand is obtained through forwarding. Conversely, if it does not depend on any earlier instruction, `dec_i0_rs2_bypass_en_d` == 0 and the operand comes from either the Register File or the Immediate.

ii. Experiment

Figure 8 shows the simulation of the program from Figure 2 in a random iteration of the loop. Cycle *i* from Figure 3 is indicated at the top part of the figure.

The signals on the top (Trace Signals) are included to help trace the instructions as they progress through the pipeline. Note that these signals were already used in previous labs. The meaning of each signal in Way 0 is as follows (the same applies to Way 1 by just substituting `i0` for `i1` in the signal names):

- `dec_i0_instr_d` → instruction in the Decode stage
- `i0_inst_e1` → instruction in the EX1 stage
- `i0_inst_e2` → instruction in the EX2 stage

- `i0_inst_e3` → instruction in the EX3 stage
- `i0_inst_e4` → instruction in the Commit stage
- `i0_inst_wb` → instruction in the Writeback stage

Below the Trace Signals, the main signals of each multiplexer analysed above are shown. Each multiplexer is surrounded by two blue lines, whereas the control signal, inputs, and output of each multiplexer are separated by red lines.

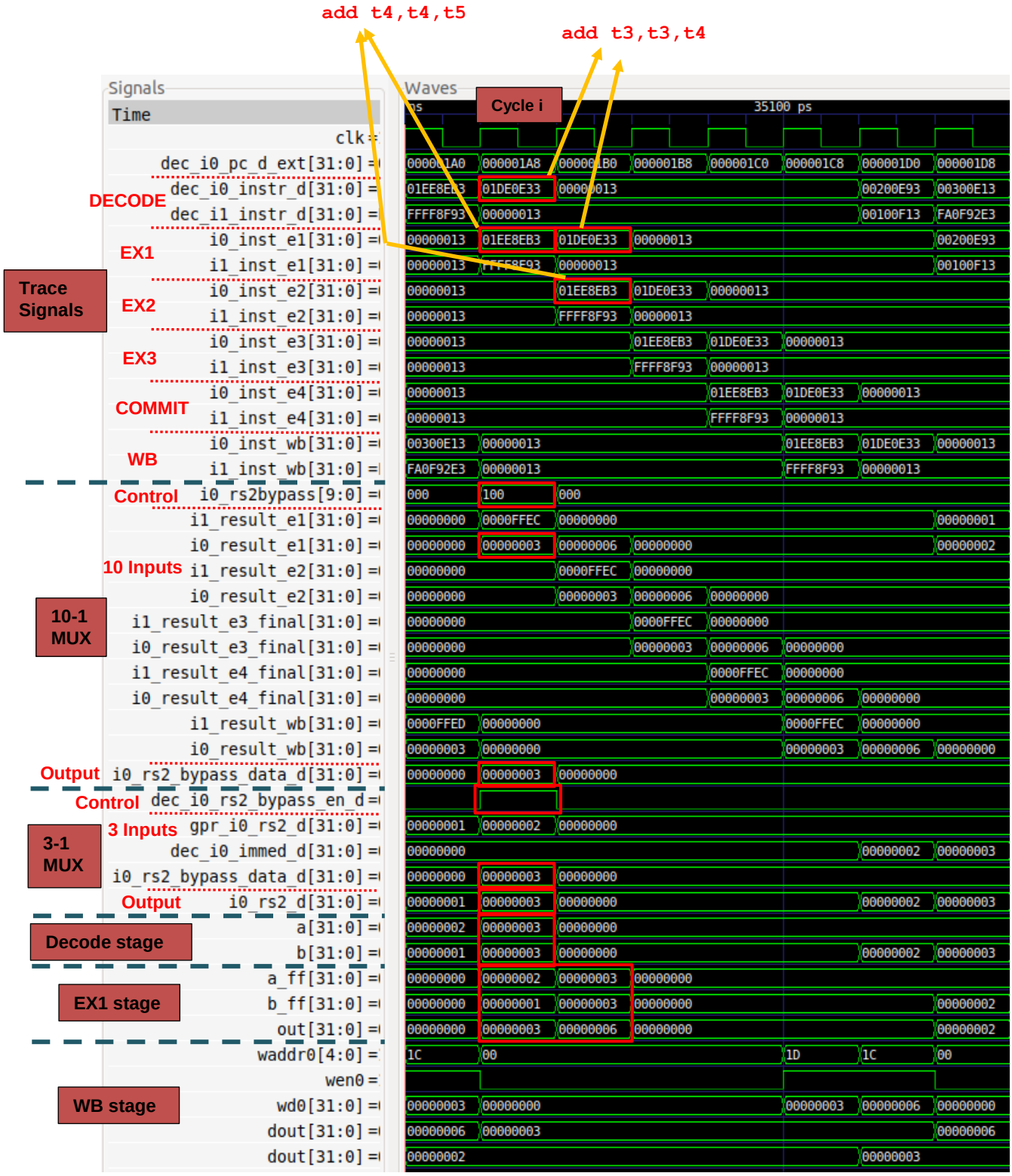


Figure 8. Simulation of the program from Figure 2 in a random iteration of the loop

Figure 9 shows the Decode and EX1 stages during the execution of the program from Figure 2 in Cycle *i* (as defined in Figure 8).

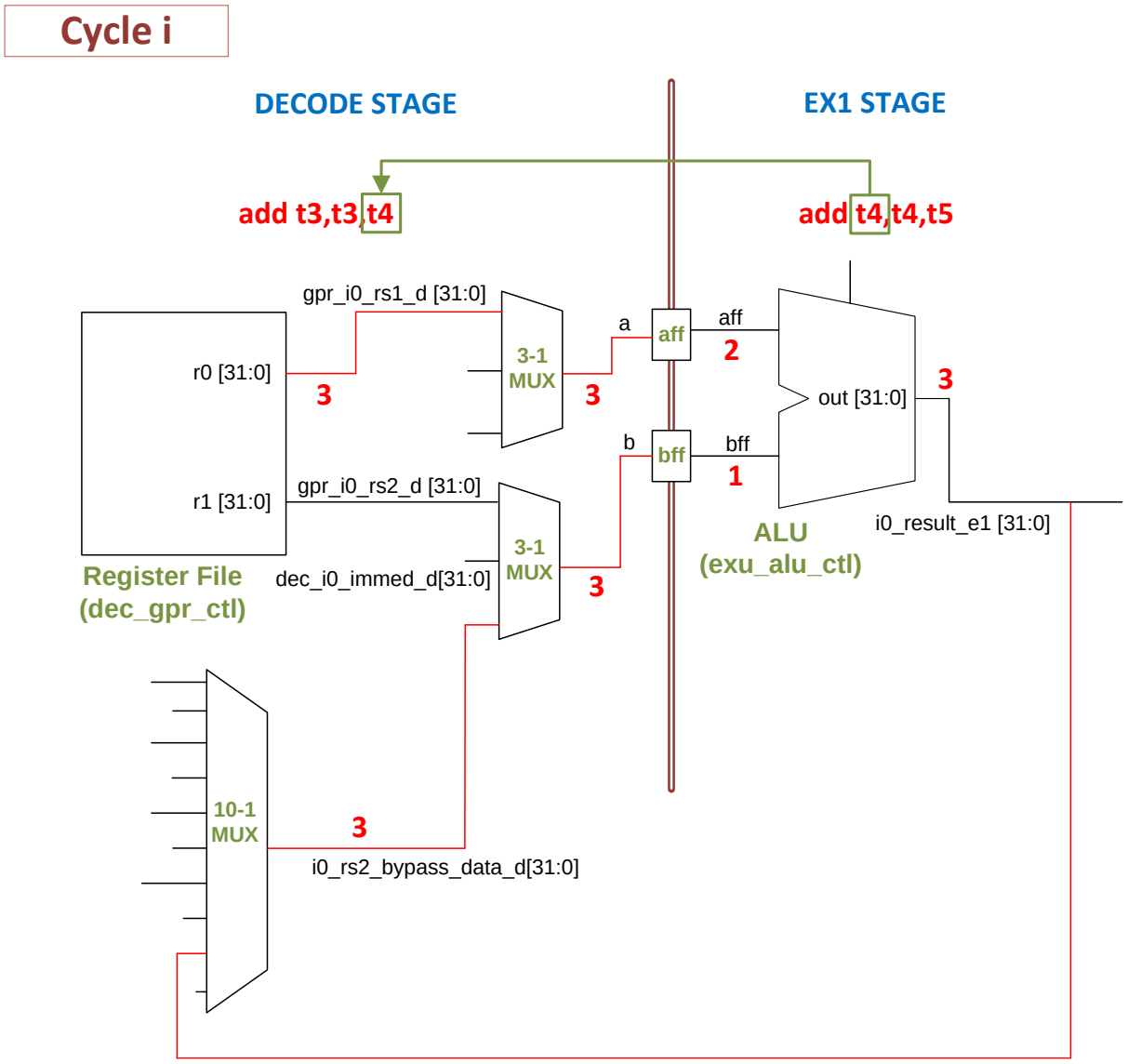


Figure 9. Decode and EX1 stages during execution of the Figure 2 example program in Cycle *i* (as defined in Figure 8)

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*.

Analyse the simulation from Figure 8 and the diagram from Figure 9 at the same time.

- **Trace Signals shown in Figure 8**:
    - In cycle *i*, the second `add` instruction is executing in the Decode stage of Way 0 (`dec_i0_instr_d` = 0x01DE0E33), and the first `add` instruction is executing in the EX1 stage of the I0 Pipe (`i0_inst_e1` = 0x01EE8EB3).
    - In cycle *i+1*, the second `add` progresses to the EX1 stage of the I0 Pipe (`i0_inst_e1` = 0x01DE0E33) and the first `add` progresses to the EX2 stage of the I0 Pipe (`i0_inst_e2` = 0x01EE8EB3).

- **10:1 Multiplexer**: In Cycle *i*, signal `i0_rs2bypass[9:0]` = 0x100 (i.e. `i0_rs2bypass[8]` = 1), thus the output is connected with the value coming from the EX1 stage of the I0 Pipe (see Figure 9):
  `i0_rs2_bypass_data_d` = `i0_result_e1` = 0x00000003

- **3:1 Multiplexer**: In Cycle *i*, signal `dec_i0_rs2_bypass_en_d` = 1, thus the output is connected with the value coming from the bypass logic (see Figure 9):
  `i0_rs2_d` = `i0_rs2_bypass_data_d` = 0x00000003

- **EX1 stage shown in Figure 8**:

  - In cycle *i*, the first `add` instruction computes the addition in the I0 Pipe ALU: `a_ff` (2) + `b_ff` (1) = `out` (3).

  - In cycle *i+1*, the second `add` instruction computes the addition in the I0 Pipe ALU: `a_ff` (3) + `b_ff` (3) = `out` (6).

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`

3. SOLVING DATA HAZARDS WITH FORWARDING AT THE COMMIT STAGE

A more delicate situation occurs when an instruction depends on a previous instruction that needs several cycles to obtain the result (i.e. a multi-cycle operation), such as a `lw` instruction, a `mul` instruction, a `div` instruction, etc. In this section we analyse a specific situation that can occur in the execution of a `lw` instruction and a dependent `add` instruction, and we leave as an exercise the analysis of other instructions and situations.

As explained in Lab 13, a `lw` instruction needs three cycles (stages DC1, DC2 and DC3) to obtain its result when the low-latency DCCM memory is used. This is the scenario used in this section. (As we also analysed in Labs 13 and 14, a larger delay is incurred when the

External DDR2 Memory is used – the effects of this larger memory latency on data hazards is left as an exercise.)

If the `lw` instruction executes three or more cycles ahead of the dependent `add` instruction, the hazard is resolved as explained in Section 2. In this case, the same 10:1 and 3:1 multiplexers described in that section are used for forwarding the data read by the load instruction to the subsequent instruction that depends on it.

APPENDIX A: The appendix at the end of the document includes an example of a `lw-add` RAW data hazard that is handled as explained in Section 2.

However, if the load instruction executes closer to the dependent `add` instruction, the hazard is resolved in a different way than described in Section 2. The problem now is that when the `add` instruction reaches the EX1 stage, the value read by the `lw` instruction is not yet available.

In the pipelined processor explained in DDCARV, bubbles are introduced in this case, which make the dependent instruction wait and only use the read value when it is available. This requires little added hardware, but it impacts performance. Thus, SweRV EH1 allows the dependent instruction to continue through the pipeline and then recalculate the operation in the Commit stage, if needed due to a data dependency.

Specifically, SweRV EH1 adds an extra ALU (the Secondary ALU) in the Commit stage of each way. This ALU recalculates the arithmetic-logic operation with the proper inputs when necessary. Thus, no cycles are lost due to stalling – but at the cost of adding two extra ALUs (one per way) as well as added control signals and logic. Figure 10 illustrates the implementation of this Secondary ALU in the Commit stage of Way 0 (the ALU is surrounded by a blue square) as well as the added forwarding logic in the EX3 stage for the second input operand (this logic is surrounded by a red square). (In Figure 4 of Lab 11 these two extra ALUs and the forwarding paths were not included for the sake of simplicity.)

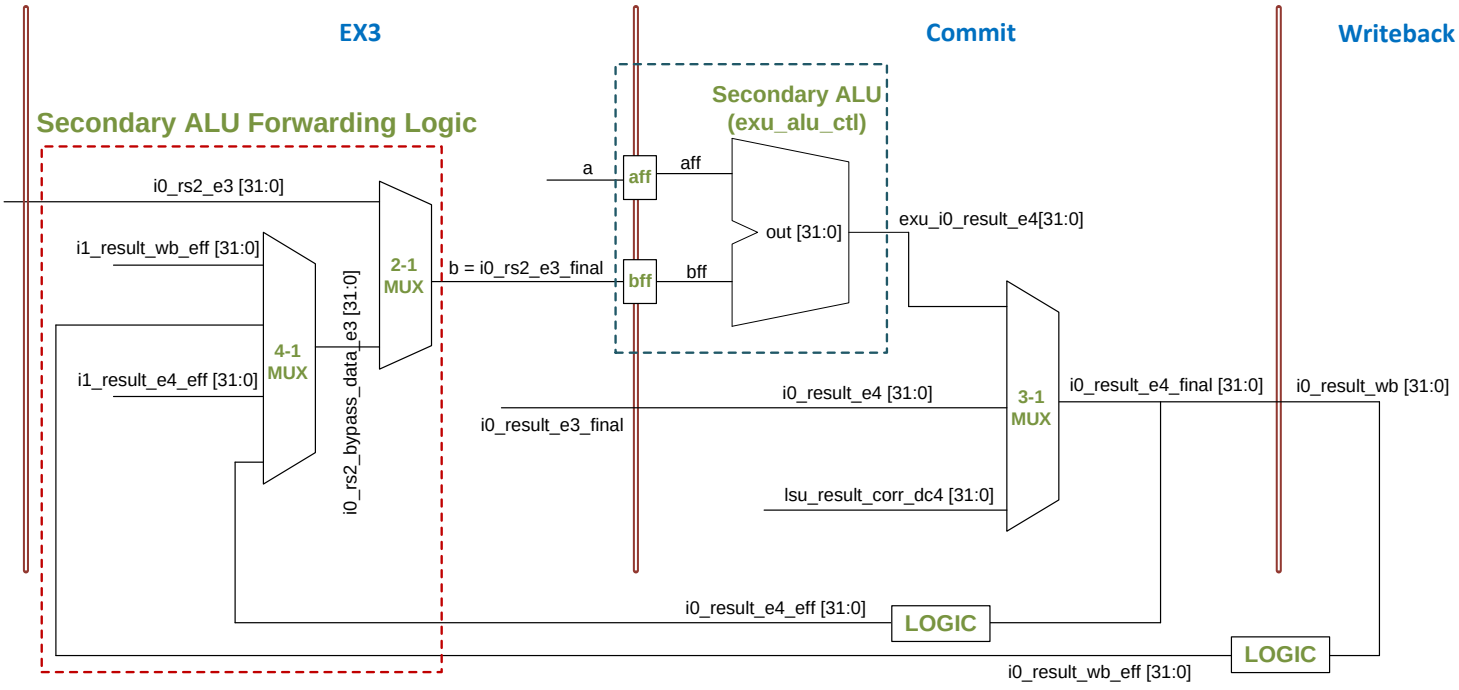


Figure 10. Secondary ALU in the Commit stage of Way 0

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

Figure 11 shows the example code used in this section. It executes a `lw` instruction followed immediately by an independent add instruction (`add t6, t6, -1`: that calculates the loop index) and then an `add` instruction that depends on the load. The independent `add` instruction is included to force both the `lw` instruction and the dependent `add` instruction to execute through Way 0. Thus, the only difference with respect to the program from the Appendix is that the `lw` and `add` instructions are closer now; however, this small difference in the program translates into a huge difference in the way it is executed, as we have just explained and will demonstrate next.

```
.globl Test_Assembly

.section .midccm
#.data
A: .space 4

.text

Test_Assembly:

la t0, A                                # t0 = addr(A)
li t1, 0x1                              # t1 = 1
sw t1, (t0)                             # A[0] = 1
li t1, 0x0
li t3, 0x1
li t6, 0xFFFF

REPEAT:
   beq t6, zero, OUT       # Stay in the loop?
   INSERT_NOPS_9
   lw t1, (t0)
   add t6, t6, -1
   add t3, t3, t1          # t3 = t3 + t1
   INSERT_NOPS_8
   li t1, 0x0
   li t3, 0x1
   add t4, t4, 0x1
   add t5, t5, 0x1
   j REPEAT
OUT:

.end
```

Figure 11. Program that executes a `lw`, independent `add`, and dependent `add`

As usual, folder *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL* provides the PlatformIO project so that you can analyse, simulate, and modify the program as desired. Open the project in PlatformIO, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL/.pio/build/swervolf_nexys/firmware.dis*). Notice that the `lw` and `add` instructions are placed at addresses 0x000001bc and 0x000001c4.

```
0x000001bc:     0002a303            lw    t1,0(t0)
0x000001c0:     ffff8f93            addi  t6,t6,-1
0x000001c4:     006e0e33            add   t3,t3,t1
```

Figure 12 shows the simulation of the program from Figure 11 in a random iteration of the loop. Again, any iteration would be valid except for the first one, which you should try to avoid due to instruction cache misses. As in the example from the previous section, the signals on the top (Trace Signals) are included for help trace the instructions as they progress through the pipeline. Below the Trace Signals, the main signals of the 4:1 and 2:1 multiplexers and the new ALU from Figure 11 are shown.

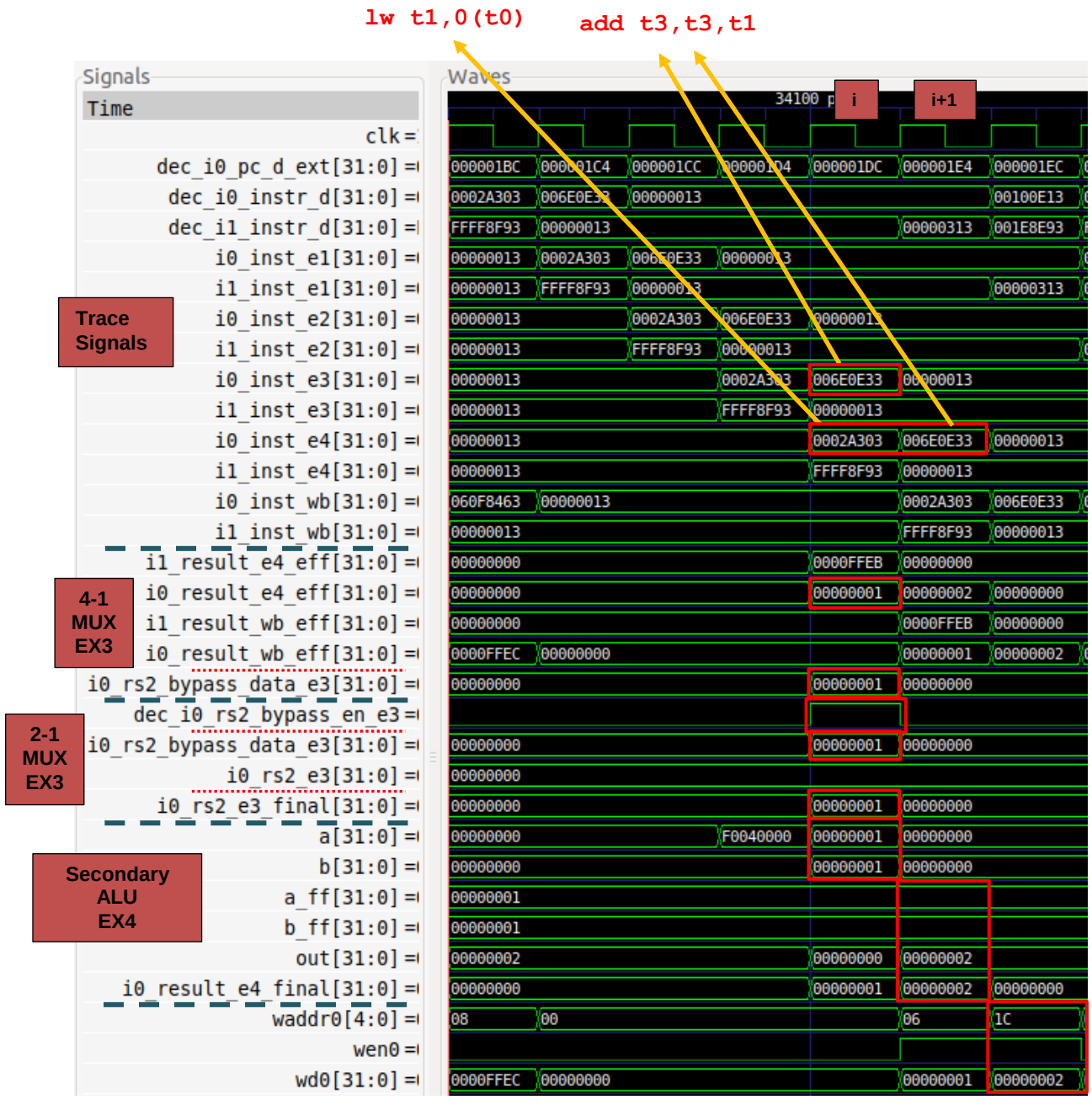
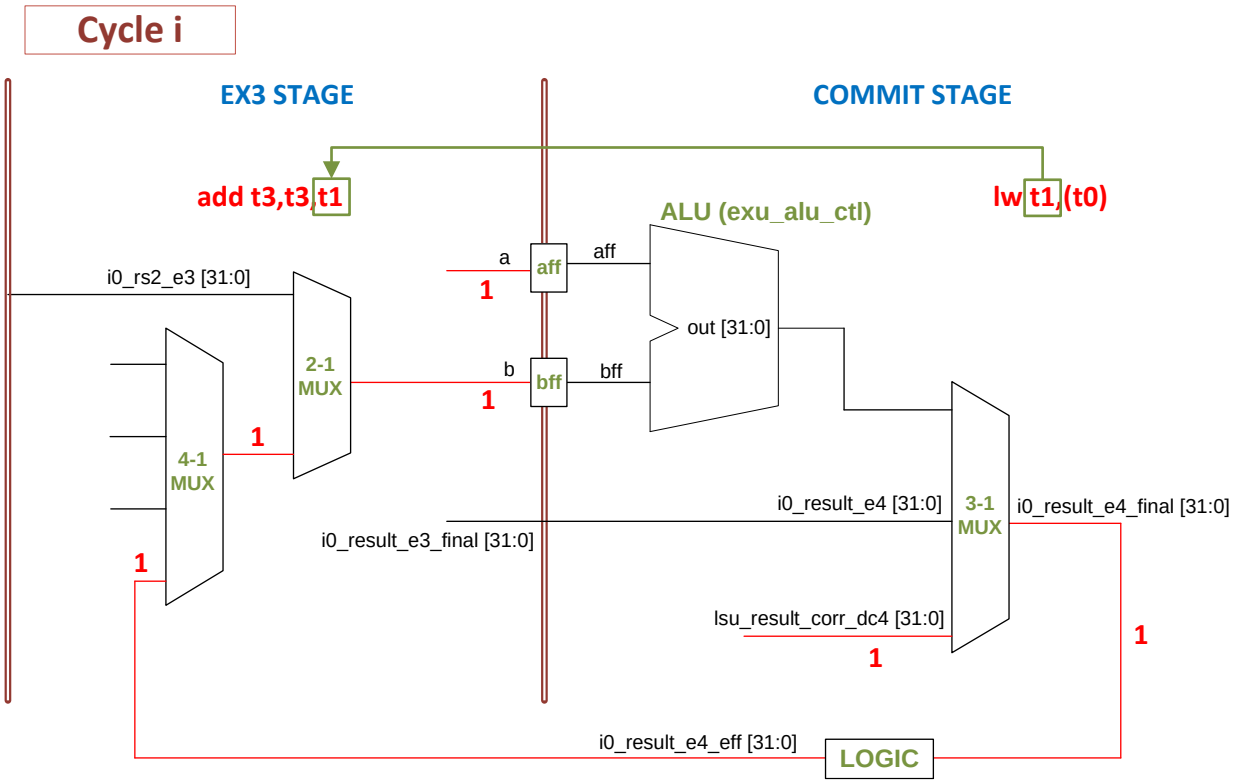


Figure 12. Simulation of the program from Figure 11 in the third iteration of the loop

Figure 13 shows a diagram of the execution of the program from Figure 11 in the seventh iteration of the loop and for cycle *i* shown in Figure 12, when the `add` instruction is in the EX3 stage and the `lw` instruction is in the Commit stage, and for cycle *i+1*, when the `add`

instruction is in the Commit (i.e., EX4) stage and recalculates the operation on the proper inputs.



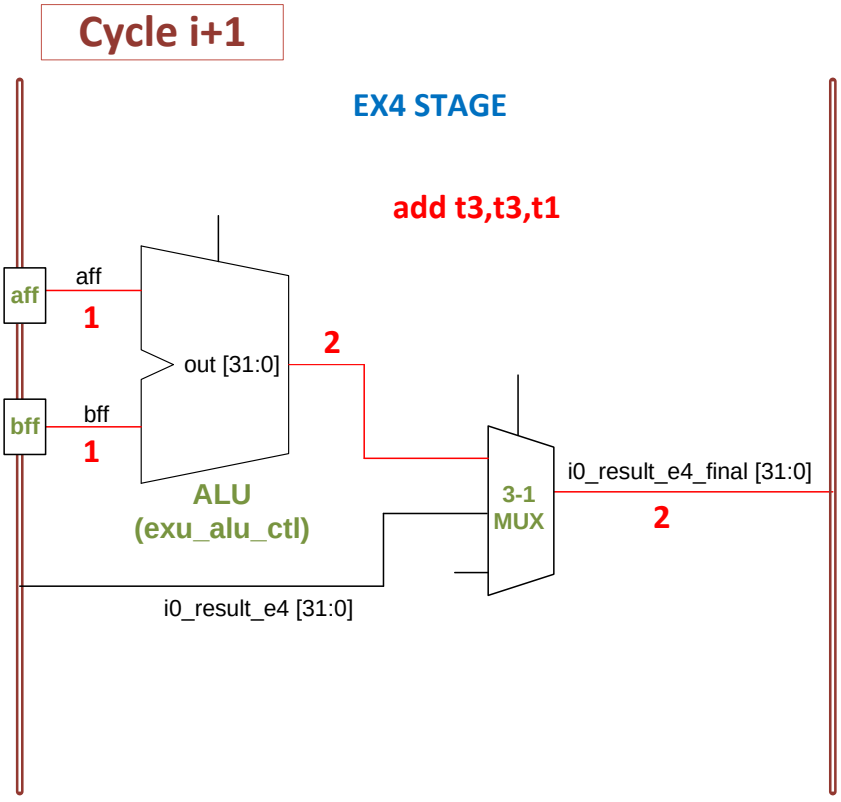


Figure 13. Diagram of the execution of the program from Figure 11 in the seventh iteration of the loop and for cycles *i* and *i+1* from Figure 12

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*

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

Analyse the waveform from Figure 12 and the diagram from Figure 13 at the same time.

- **Trace Signals shown in Figure 12:**
  - In cycle *i*, the `add` instruction is in the EX3 stage of Way 0 (`i0_inst_e3` = 0x006E0E33), and the `lw` instruction is in the Commit stage of the I0 Pipe (`i0_inst_e4` = 0x0002A303).
  - In cycle *i+1*, the `add` instruction is in the Commit stage of Way 0 (`i0_inst_e4` = 0x006E0E33).
- **4-1 Multiplexer**: In Cycle *i*, the value read by the load instruction, which in this cycle is in the Commit stage, is selected:
  `i0_rs2_bypass_data_e3` = `i0_result_e4_eff` = 0x00000001
- **2-1 Multiplexer**: In Cycle *i*, due to the dependency between the load and the addition, the bypassed value is selected (`dec_i0_rs2_bypass_en_e3` = 1). Thus:
  `i0_rs2_e3_final` = `i0_rs2_bypass_data_e3` = 0x00000001
- **Commit stage ALU**: In Cycle *i+1*, the addition is recomputed using the correct values:
  `out` = `a_ff` + `b_ff` = 0x00000001 + 0x00000001 = 0x00000002
  Then, in the 3:1 multiplexer, the ALU output is selected (`exu_i0_result_e4`). Note that, if no dependency exists, the value in signal `i0_result_e4` would be selected.

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*

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

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.

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.

4. 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.

2) Analyse the same situation as the one described in Section 3 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`.

3) 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.

4) (*The following exercise is based on exercises 4.18*, *4.19*, *4.20,* and *4.26 of [PaHe].*) 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 nops where necessary). Add nops 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?

5) In the program from 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**).

6) In the program from 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.

7) In the program from 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 to occur. Analyse in simulation how these two hazards are handled in SweRV EH1.

8) **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 store-load forwarding is carried out in SweRV EH1.

APPENDIX A

In this appendix we include an example of a `lw-add` RAW data hazard that is handled as explained in Section 2. Figure 14 shows the example code used in this appendix. It executes a `lw` instruction followed by 5 `nop` instructions and an `add` instruction that depends on the load. The intermediate `nop` instructions are included in order to separate the two dependent instructions.

```
.globl Test_Assembly

.section .midccm
#.data
A: .space 4

.text
Test_Assembly:

# Register t3 is also called register 28 (x28)
la t0, A                                # t0 = addr(A)
li t1, 0x1                              # t1 = 1
sw t1, (t0)                             # A[0] = 1
li t1, 0x0
li t3, 0x1
li t6, 0xFFFF

REPEAT:
   beq t6, zero, OUT       # Stay in the loop?
   INSERT_NOPS_8
   lw t1, (t0)
   INSERT_NOPS_5
   add t3, t3, t1          # t3 = t3 + t1
   INSERT_NOPS_8
   li t1, 0x0
   li t3, 0x1
   add t6, t6, -1
   j REPEAT
OUT:

.end
```

Figure 14. Program that executes `lw`, 5 nops, and a dependent `add` instruction

As usual, folder *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_FarAway-LW-AL* provides the PlatformIO project so that you can analyse, simulate, and modify the program as desired. Open the project, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_FarAway-LW-AL/.pio/build/swervolf_nexys/firmware.dis*). Notice that the `lw` and `add` instructions are placed at addresses 0x000001b0 and 0x000001c8.

```
0x000001b0:      0002a303              lw    t1,0(t0)
0x000001b4:      00000013              nop
0x000001b8:      00000013              nop
0x000001bc:      00000013              nop
0x000001c0:      00000013              nop
0x000001c4:      00000013              nop
0x000001c8:      006e0e33              add   t3,t3,t1
```

Figure 15 shows the simulation of the program from Figure 14 in the third iteration of the loop. Again, any iteration would be valid except for the first one, which you should try to avoid due to instruction cache misses. As in the examples from the main lab, the signals on the top (Trace Signals) help trace the instructions as they progress through the pipeline. Below the Trace Signals, the main signals of each multiplexer are shown. The signals from each multiplexer are surrounded by dashed blue lines. The control signal, inputs, and output of each multiplexer are illustrated, as was done in the main lab.

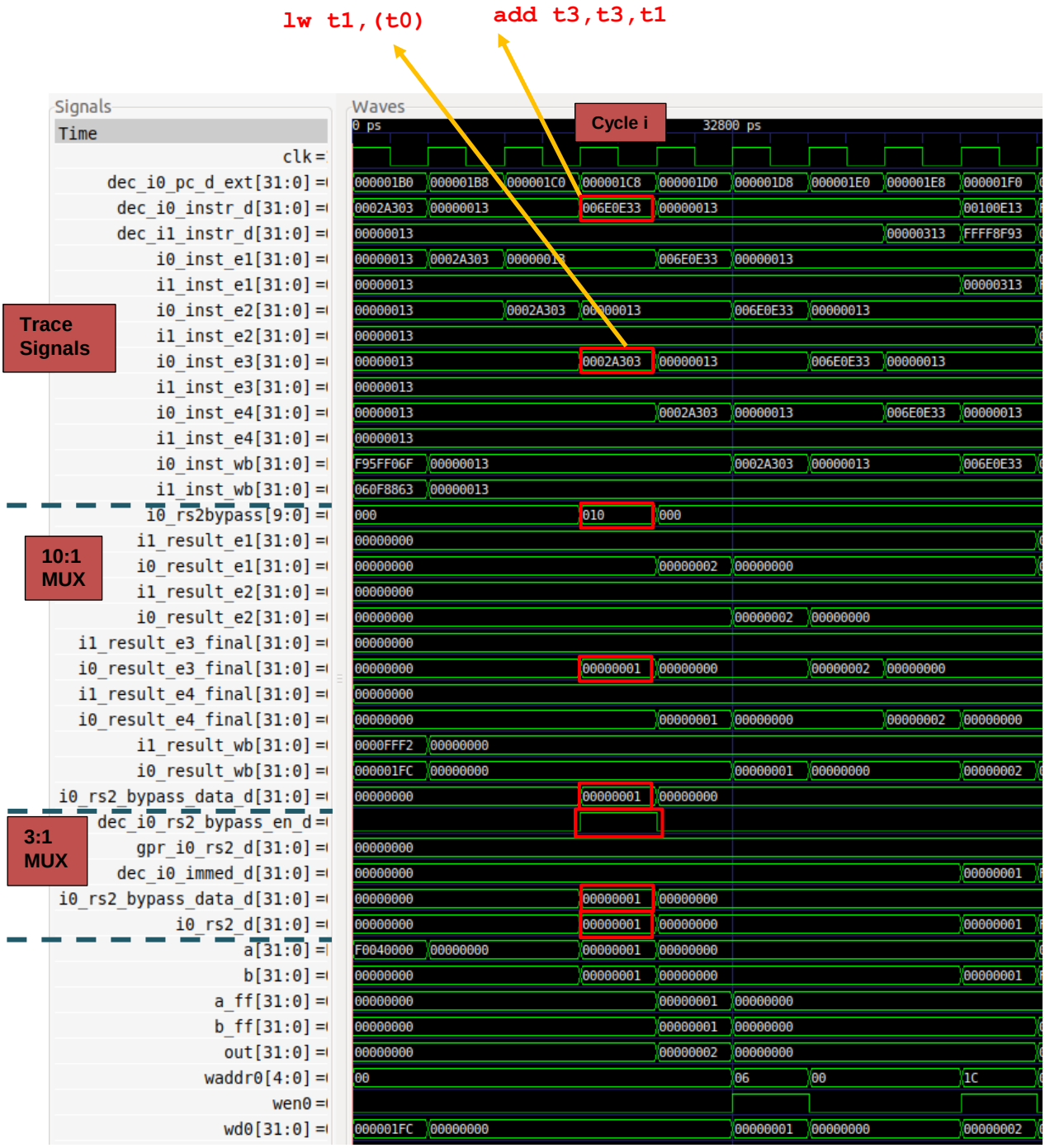


Figure 15. Simulation of the program from Figure 14 in the third iteration of the loop

Figure 16 shows a diagram of the execution of the program from Figure 14 in Cycle *i* (as defined in Figure 15), when the `add` instruction is in the Decode stage and the `lw` instruction is in DC3.

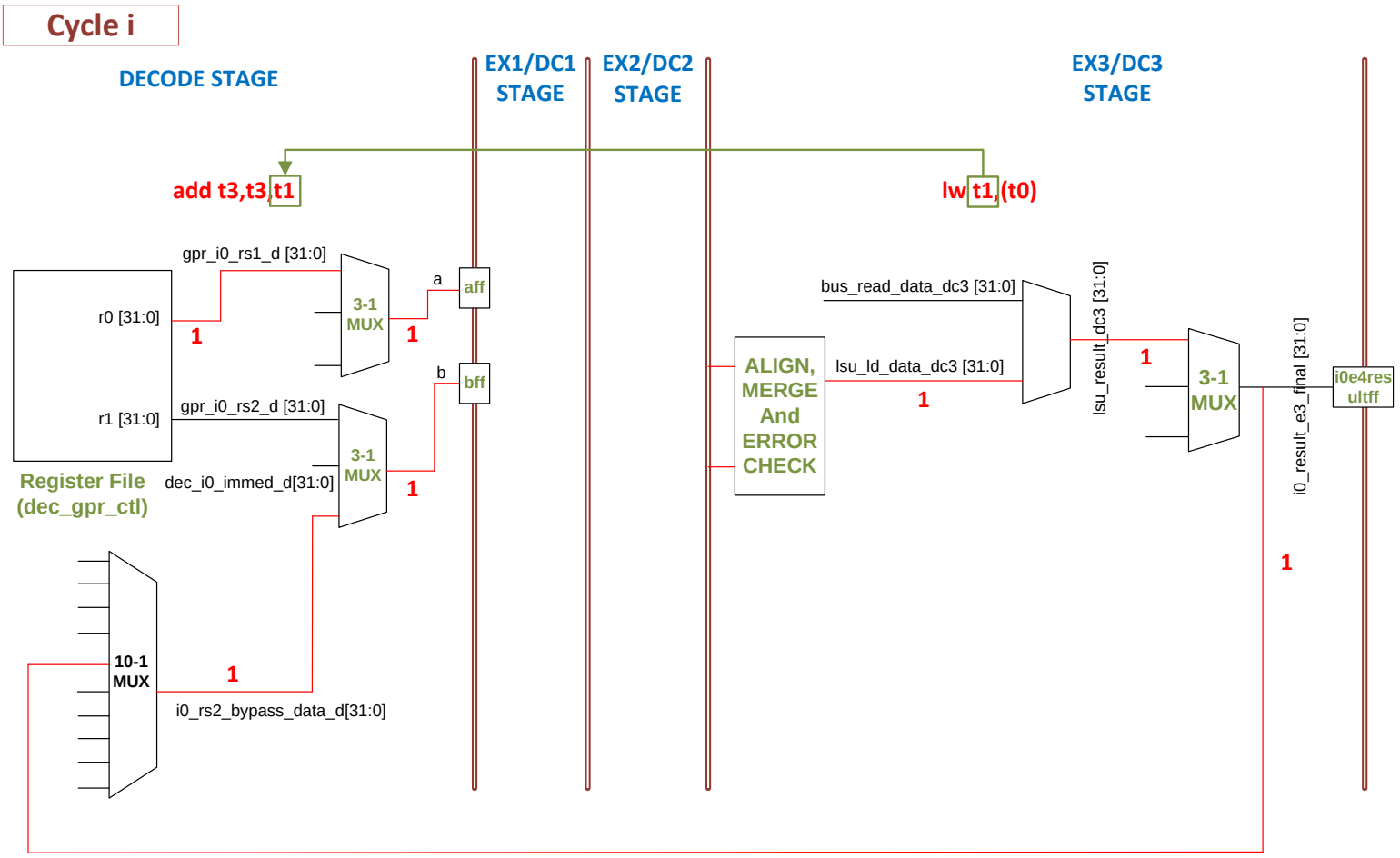


Figure 16. Hardware during execution of the program from Figure 14 in the third iteration of the loop and in the fourth cycle shown in Figure 15

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

Analyse the waveform from Figure 15 and the diagram from Figure 16 at the same time.

- **Trace Signals shown in Figure 15:**
    - In cycle *i*, the `add` instruction is in the Decode stage of Way 0 (`dec_i0_instr_d` = 0x006E0E33), and `lw` is in the DC3 stage of the I0 Pipe (`i0_inst_e3` = 0x0002A303).
- **10:1 Multiplexer**: In cycle *i*, signal `i0_rs2bypass[9:0]` = 0x010 (i.e. `i0_rs2bypass[4]` = 1), thus the output is connected to the value coming from the EX3/DC3 stage of the I0 Pipe (see Figure 16):
  `i0_rs2_bypass_data_d` = `i0_result_e3_final` = 0x00000001
- **3:1 Multiplexer**: In Cycle *i*, signal `dec_i0_rs2_bypass_en_d` = 1, thus the output is connected to the value coming from the bypass logic (see Figure 9):
  `i0_rs2_d` = `i0_rs2_bypass_data_d` = 0x00000001

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

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.

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).