THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2 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 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 VeeR EL2 processor.

In Lab 15 we analyse the second type of hazard, **data hazards**, in the VeeR EL2 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 VeeR EL2 processor.

2. Structural Hazards in VeeR EL2

In this lab, we illustrate two cases of structural hazards that can occur in the VeeR EL2 processor. Each is resolved in a different way, thus resulting in a different performance-cost trade-off.

In the first situation (Section 2.A), hardware cost is sacrificed for performance. To illustrate this scenario, we create an example where two instructions arrive at the R Stage and try to write the Register File in the same cycle. In principle, because VeeR EL2 is a single-issue core, it would not be possible to complete two instructions in the same cycle; however, as we showed in previous labs, VeeR EL2's register file has more than one write port, which avoids the structural hazard in this situation. This solution has a high impact in hardware cost due to the extra port in the Register File, but it resolves this structural hazard with no performance loss.

In the second situation (Section 2.B), performance is sacrificed for hardware cost. To illustrate this scenario, we create an example where two divide (`div`) instructions are executed sequentially. Given that there is only one multicycle divisor in the core, the second instruction cannot execute until the first one completes due to the structural hazard. In this case, the solution for resolving the structural hazard is to stall the pipeline, thus resulting in performance loss but no hardware cost.

A. Two simultaneous instructions in the R Stage

VeeR EL2 is a single-issue processor; this means that only one instruction can execute per cycle. In a situation where two 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 VeeR EL2; however, there is a specific case when this can happen:

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

- Given that 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 another instruction arrives at the Writeback stage in cycle ***i*** and also wants to write to the register file (for example, an `add` instruction).

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

One simple solution would be to stall the pipeline for one cycle and complete the two writes sequentially. A more efficient solution performance-wise is to design the Register File with more than one write port, thus being able to perform the two writes in parallel, resolving the structural hazard with no stalls and no performance loss. This is the solution used in VeeR EL2, as we analyse in this lab.

The example from Figure 1 illustrates this situation. It executes a non-blocking `lw` instruction followed by several `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 `add` instruction that arrives in the Retire (R) Stage in the same cycle as the `lw` instruction (cycle ***i***) is also highlighted. In this case, `nop` instructions are not included as they are not necessary. As usual, the program does nothing useful and is only intended to illustrate the example of this lab.

```
REPEAT:
   lw x28, (x29)
   add x30, x30, -1
   add a1, a1, 1
   add a2, a2, 1
   add a3, a3, 1
   add a4, a4, 1
   add a5, a5, 1
   bne x30, zero, REPEAT # Repeat the loop
```

Figure 1. Example for a non-blocking `lw` instruction followed by 36 A-L instructions

Folder *[RVfpgaBasysPath]/Labs/Lab14/Lw_Instruction* provides the Catapult 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. Note however that the code that configures

and reads the performance counters is commented in the provided program, thus, when you want to use them, you must uncomment the corresponding code.

As shown in Section 4 of Lab 13, the 32-bit data obtained in the **`el2_lsu_bus_intf`** module (Bus Interface) is provided to the register file through signal `lsu_nonblock_load_data[31:0]`. Also, the control signal that tells the register file where to write that data, which was generated in the D (Decode) Stage and propagated through the Pipeline Registers, is provided to the register file through signal `dec_nonblock_load_waddr[4:0]`. These two signals go into the register file through the second write port available in this structure (`waddr1` and `wd1`).

Figure 2 shows the RVfpgaEL2-Trace simulation for the program from Figure 1 and a diagram that illustrates the execution of this program for a random iteration of the loop.

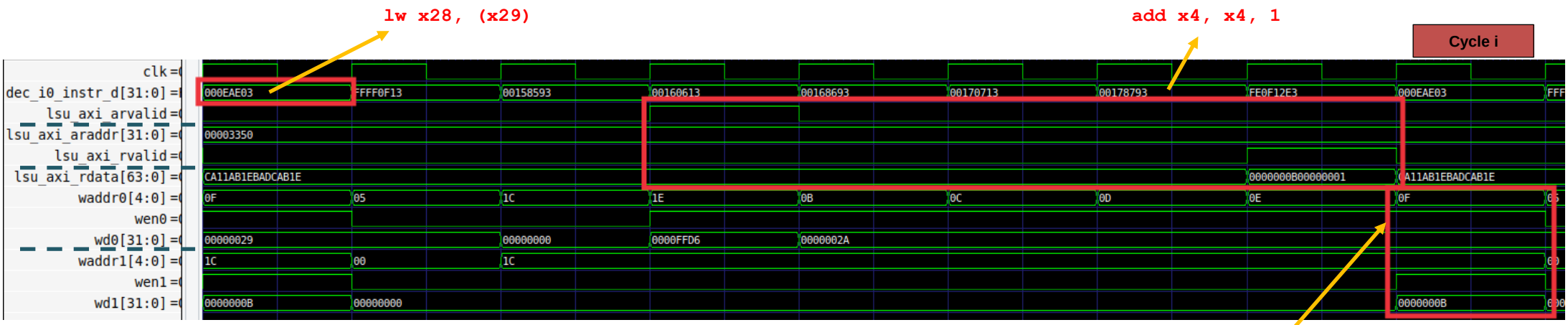


Figure 2. RVfpgaEL2-Trace simulation for the example from Figure 1

Two simultaneous writes to the Register File:

- **`lw` writes register x28 (0x1C)**
- **`add` writes register x15 (0xF)**

TASK: Replicate the simulation from Figure 2 on your own computer. Use file *test.tcl* (provided at *[RVfpgaBasysPath]/Labs/Lab14/Lw_Instruction/commandLine*).

Analyse the waveform from Figure 2.

- **Cycle i-8**: The `lw` instruction is at the D Stage.

- **Cycle i-5**: The effective memory address is sent to the Main Memory through the AXI Bus. The latency of Main Memory forces the load instruction to wait for the data to arrive to the core.

- **Cycle i-2**: The conflicting `add` instruction is decoded.

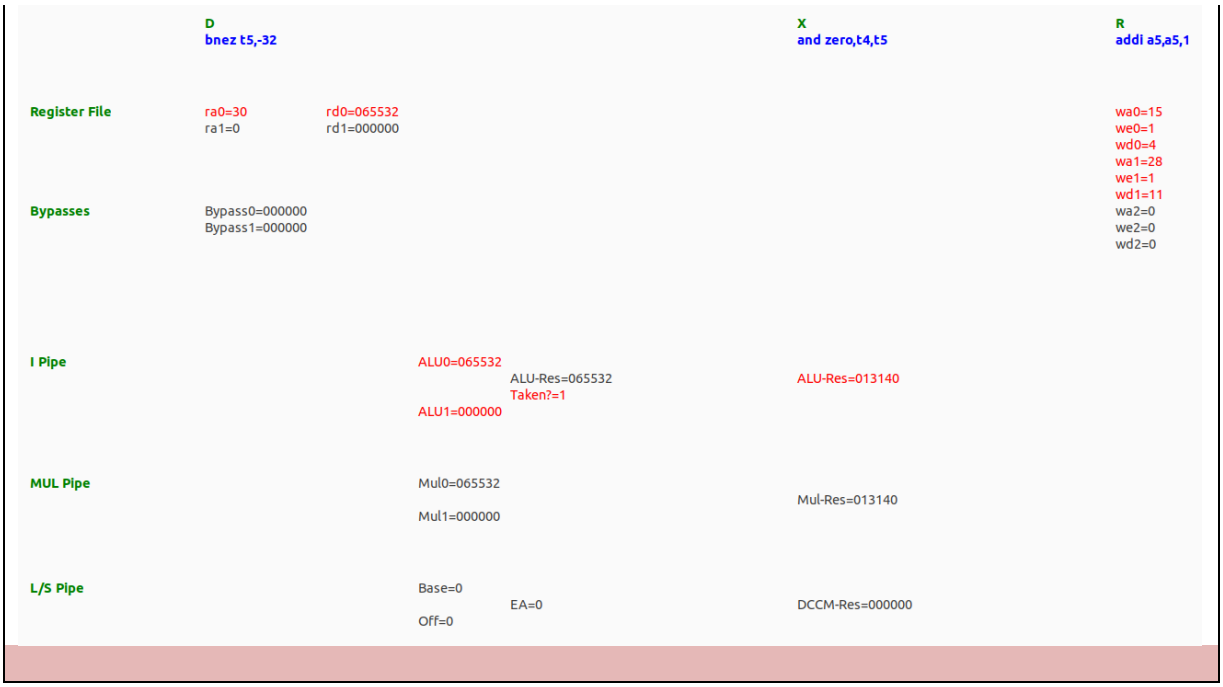
- **Cycle i-1**: The requested data for the `lw` arrives from Main Memory through the AXI bus. The `add` instruction computes its result.

- **Cycle i**: The `lw` instruction and the conflicting `add` instruction proceed to the R Stage, where they must write the register file. This is possible thanks to the three write ports available in VeeR EL2's register file. Note that the register numbers are shown in hexadecimal in the simulation. `x15` and `x28` (registers 0xF and 0x1c) are being written.

TASK: Measure different events (cycles, instructions/loads committed, etc.) using the Performance Counters available in VeeR EL2, as explained in Lab 11. Remember that you must uncomment the code that configures and uses the Performance Counters. Is the number of cycles as expected after analysing the RVfpgaEL2-Trace simulation from Figure 2? Justify your answer. Remember that you can test your program in any of the tools available in RVfpgaEL2: RVfpgaEL2-Basys3 (physical board), RVfpgaEL2-ViDBo (virtual board), RVfpgaEL2-Trace, etc.

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

For example, this is the VeeR EL2 pipeline when the two writes are performed in the Register File (`wa0 = 15, we0 = 1, wd0 = 4, wa1 = 28, we1 = 1, wd1 = 11`).



B. Two sequential `div` instructions

In this section we analyse a second scenario based on the `div` instruction. We analyse an example program where two sequential `div` instructions are included, which make the processor stall for some cycles due to a structural hazard.

The `div` instruction belongs to the RISC-V M Extension (Standard Extension for Integer Multiplication and Division), which is supported in VeeR EL2. This 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
```

For executing this instruction, the VeeR EL2 processor implements a non-pipelined multi-cycle divide unit in module **`el2_exu_div_ctl`** (*[RVfpgaBasysPath]/src/VeeRwolf/VeeR_EL2CoreComplex/exu/el2_exu_div_ctl.sv*). This unit needs several cycles to compute the result. The divide unit outputs, besides the result in signal `out`, signal `finish_dly` that indicates the status of the divide instruction.

TASK: Inspect the Verilog code from **`el2_exu_div_ctl`** to understand how the division is computed. Also analyse the effect of signal `finish_dly`.

The example from Figure 1 illustrates this situation. It executes two sequential divide (`div`) instructions contained within a loop that repeats for 0xFFFF iterations (i.e., 65,535). The `div` instructions are highlighted in red in the figure. As usual, the program does nothing useful and is only intended to illustrate the hazard example in this lab.

```
REPEAT:
   div a6, x28, x29
```

```
div a7, x29, x28
add x30, x30, -1
INSERT_NOPS_5
bne x30, zero, REPEAT # Repeat the loop
```

Figure 3. Example of back-to-back `div` instructions

Folder *[RVfpgaBasysPath]/Labs/Lab14/Sequential_Div_Instructions* provides the Catapult 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 `div` instructions) and which is invoked from the *.c* file. Note however that the code that configures and reads the performance counters is commented out in the provided program, thus, when you want to use them, you must uncomment the corresponding code.

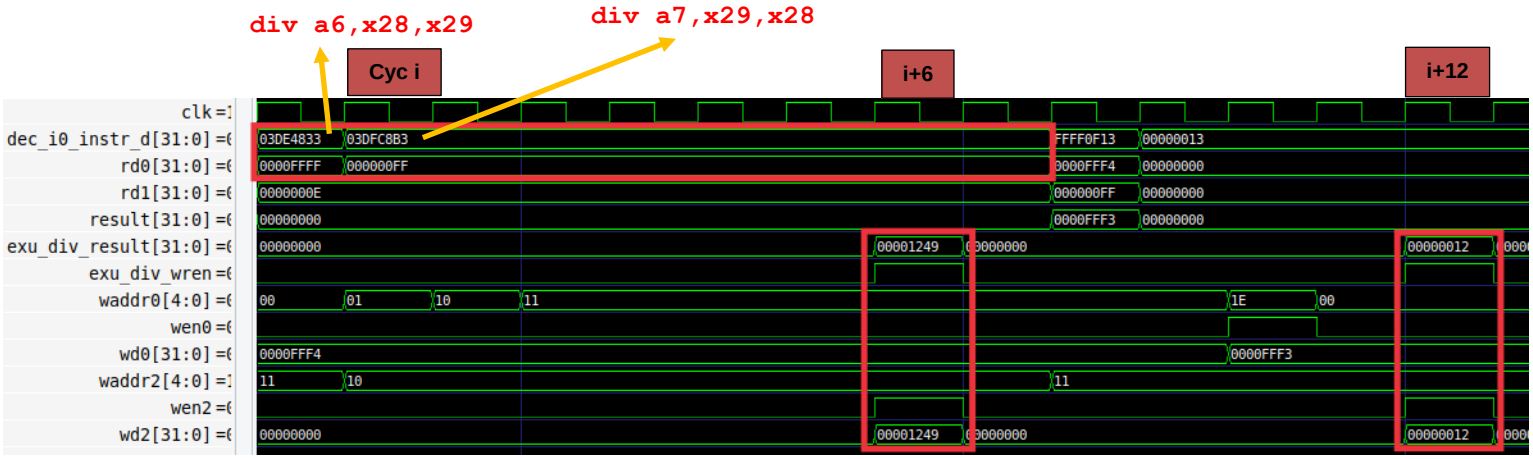


Figure 4. Example of back-to-back `div` instructions

Figure 4 shows the RVfpgaEL2-Trace simulation for the program from Figure 3 in a random iteration of the loop.

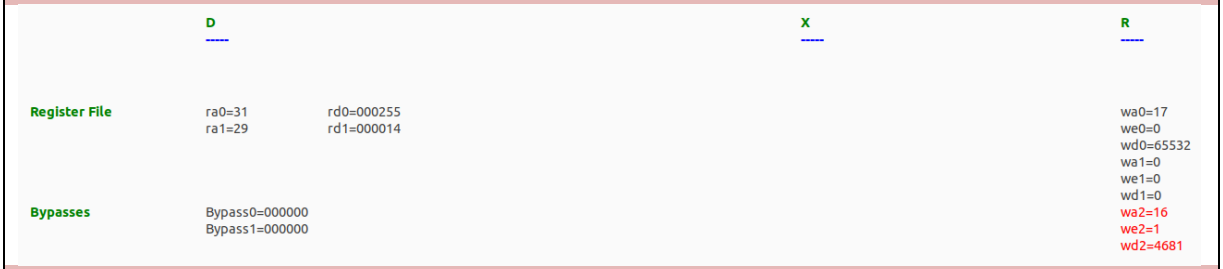
TASK: Replicate the simulation from Figure 2 on your own computer. Use file *test.tcl* (provided at *[RVfpgaBasysPath]/Labs/Lab14/Sequential_Div_Instructions/commandLine*).

Analyse the waveform from Figure 4. In this case, when the second `div` instruction arrives in the D Stage (*cycle i*), the divider unit is busy computing the first division and thus the pipeline must stall until the end of the operation (*cycle i+6*), when the divider is released and can start being used by the second `div` instruction (which obtains its result in *cycle i+12*). Thus, as in the previous section, there is a structural hazard; however, in this case, the hazard is resolved with no hardware cost but with some impact on performance.

TASK: Measure different events (cycles, instructions/loads committed, etc.) using the Performance Counters available in VeeR EL2, as explained in Lab 11. Remember that you must uncomment the code that configures and uses the Performance Counters. Are the number of cycles as expected after analysing the RVfpgaEL2-Trace simulation from Figure 4? Justify your answer. Remember that you can test your program in any of the tools available in RVfpgaEL2: RVfpgaEL2-Basys3 (physical board), RVfpgaEL2-ViDBo (virtual board), RVfpgaEL2-Trace, etc.

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

For example, this is the VeeR EL2 pipeline when the first division writes in the Register File (`wa2=16, we2=1, wd2=4681`) after several cycles where the pipeline has been stalled due to the structural hazard in the access to the Divider.



3. Exercises

1) Like loads, `div` instructions are non-blocking, thus independent instructions can continue executing while the `div` is being computed in the divisor. Also like loads, it is possible that when the division finishes and progresses to the R Stage, another instruction is also in this stage and needs to write its result to the Register File. The solution is again the same as the one used for load (`lw`) instructions: a third write port (port 2, see Lab 11) is included in the Register File so that the two writes can happen in the same cycle. Illustrate this situation by simulating the program provided in folder *[RVfpgaBasysPath]/Labs/Lab14/Div_Instruction*.

2) Create a program, similar to the one from Section 2.B, where two sequential independent load instructions are executed. How is this scenario handled in VeeR EL2? Is it equal to or different from the solution used for the two sequential divisions.

3) Analyse a scenario where three instructions arrive at the R Stage at the same time: `add`, `lw` and `div`. Is it necessary to stall the processor? Explain it theoretically and demonstrate it with an example program. Simulate the program on RVfpgaEL2-Pipeline.

4) 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, etc.

5) Analyse `mul` instructions in VeeR EL2, both theoretically and practically with example programs.

6) Replace the divider unit, implemented in module **`el2_exu_div_ctl`**, with your own unit or an open-source unit downloaded from the Internet. Compare the performance of this new divider with that of the original EL2 divider.