THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2 Lab 15

Data Hazards

© Copyright Imagination Technologies

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*.

- ***Write After Write* (WAW) data hazard:** WAW hazards occur when instructions are reordered and instruction *j* writes *x* before instruction *i* writes *x*.

In the following sections we analyse how RAW data hazards are resolved in the VeeR EL2 processor, and then we describe tasks and exercises related to these hazards.

NOTE: Before analysing the VeeR EL2 data hazard logic, we recommend reading Section 7.5 in DDCARV about how hazards are resolved in a pipelined processor. Data hazards, specifically, are analysed in Section 7.5.3. Although the pipelined processor shown in the book is not exactly the same as VeeR EL2, data hazards are resolved very similarly in both processors.

2. Solving RAW Data Hazards in VeeR EL2

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

Figure 1 extends the D (Decode) Stage shown in Lab 11 with the bypass values. The Forwarding Logic includes two 4:1 multiplexers that choose a value from subsequent stages as the bypass value for each of the two source operands.

The following inputs can be selected by these multiplexers:
- Result in the R Stage (`dec_i0_result_r`).
- Value read by a load instruction from the DCCM in the M Stage (`lsu_result_m`).
- Result in the X Stage (`exu_i0_result_x`).
- Value read by a load instruction from Main Memory (`lsu_nonblock_load_data`).

The control signals are 4-bit signals called `dec_i0_rs1_bypass_en_d[3:0]` and `dec_i0_rs2_bypass_en_d[3:0]` for operands 1 and 2 respectively.

© Copyright Imagination Technologies

The outputs of these multiplexers are provided in signals `i0_rs1_bypass_data_d` and `i0_rs2_bypass_data_d` for operands 1 and 2 respectively.

These two outputs are distributed to the 2:1, 3:1, and 4:1 multiplexers that select the input operands for each of the datapaths. For the sake of clarity in Figure 1, signals are connected by name. These multiplexers choose the value from the bypass path when any of the 4 bits in signals `dec_i0_rs1_bypass_en_d[3:0]` and `dec_i0_rs2_bypass_en_d[3:0]` are 1.

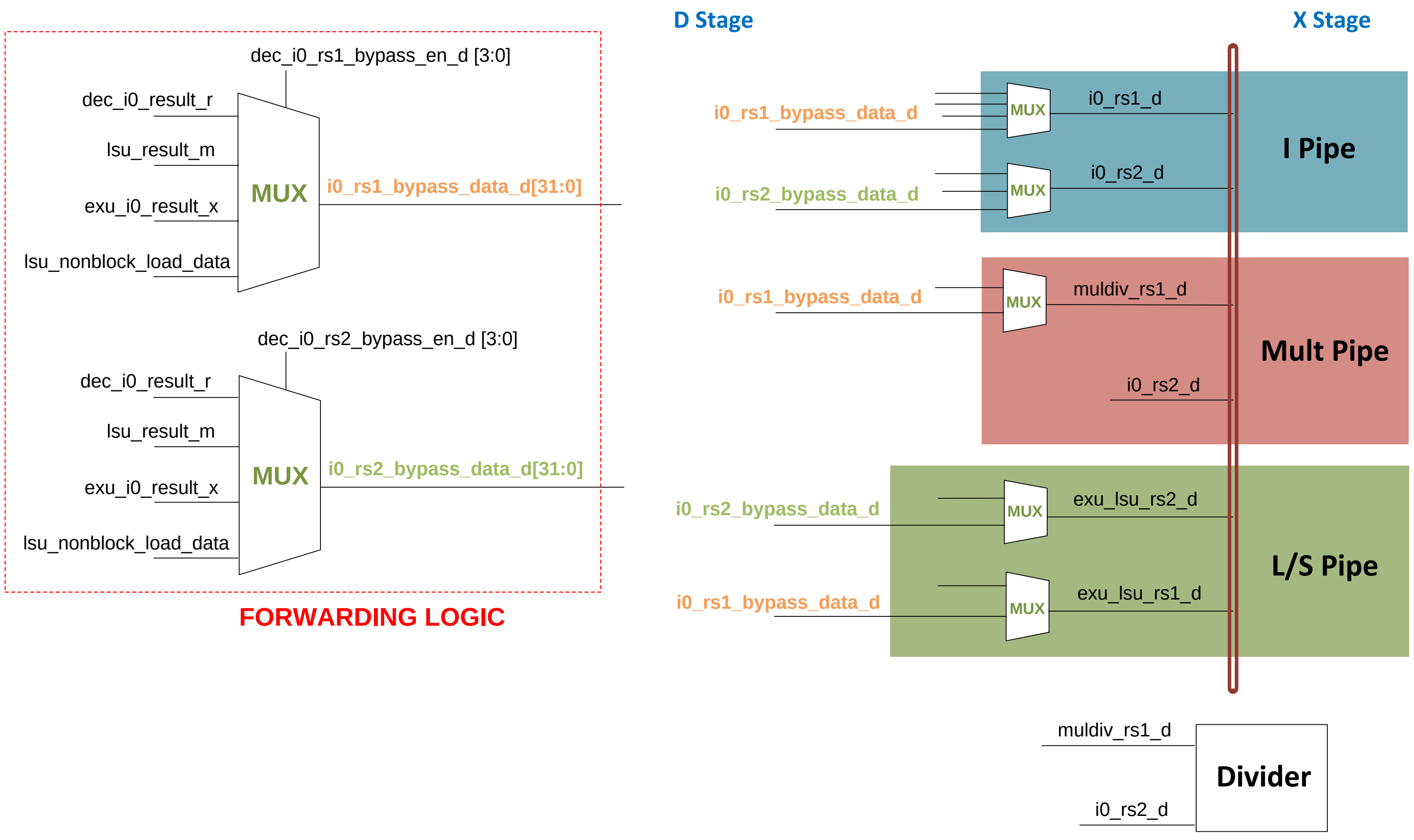

Figure 1. Bypass inputs to the Functional Units

Many forwarding paths exist in the VeeR EL2 processor. In this section we focus on a specific path and analyse it in detail. We analyse the situation of two dependent A-L instructions executing and how RAW data hazards are resolved.

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.

```
.globl Test_Assembly

.text
Test_Assembly:

li t3, 0x3
li t4, 0x2
```

© Copyright Imagination Technologies

```
li t5, 0x1
li t6, 0xFFFF

REPEAT:
   INSERT_NOPS_2
   add t4, t4, t5          # t4 = t4 + t5
   add t3, t3, t4          # t3 = t3 + t4
   INSERT_NOPS_4
   add t6, t6, -1
   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 *[RVfpgaBooleanPath]/Labs/Lab15/DataHazards_AL-AL* provides the Catapult project so that you can analyse, simulate, and modify the program as desired. Open the project, build it, and open the disassembly file. You will see that the two `add` instructions that we are analysing are placed at addresses 0x00000428 and 0x0000042c:

```
0x00000428:     01ee8eb3          add   t4,t4,t5
0x0000042c:     01de0e33          add   t3,t3,t4
```

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 X Stage, from where it can be bypassed to the D 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 VeeR EL2 D and X stages during cycle *i+1* of Figure 4. In this cycle, the first `add` instruction (`add t4,t4,t5`) is in the X Stage and the second `add` instruction (`add t3,t3,t4`) is in the D Stage. As shown in the figure, the result of the first `add` instruction is bypassed to the D stage, it is selected by the Forwarding Logic, and it is used as the second input operand for the second `add` instruction.

© Copyright Imagination Technologies

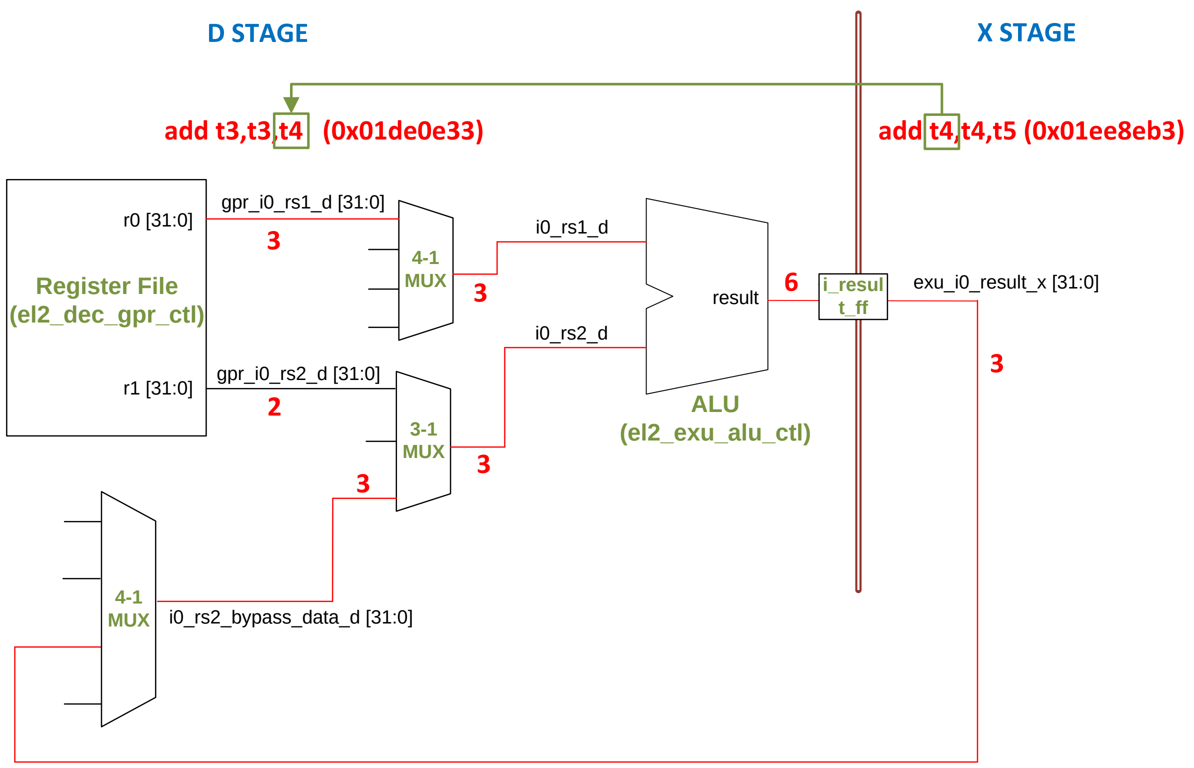

Figure 3. Result forwarded from X to D (second operand) in cycle i+1

Figure 4 shows the RVfpgaEL2-Trace simulation of the program from Figure 2.

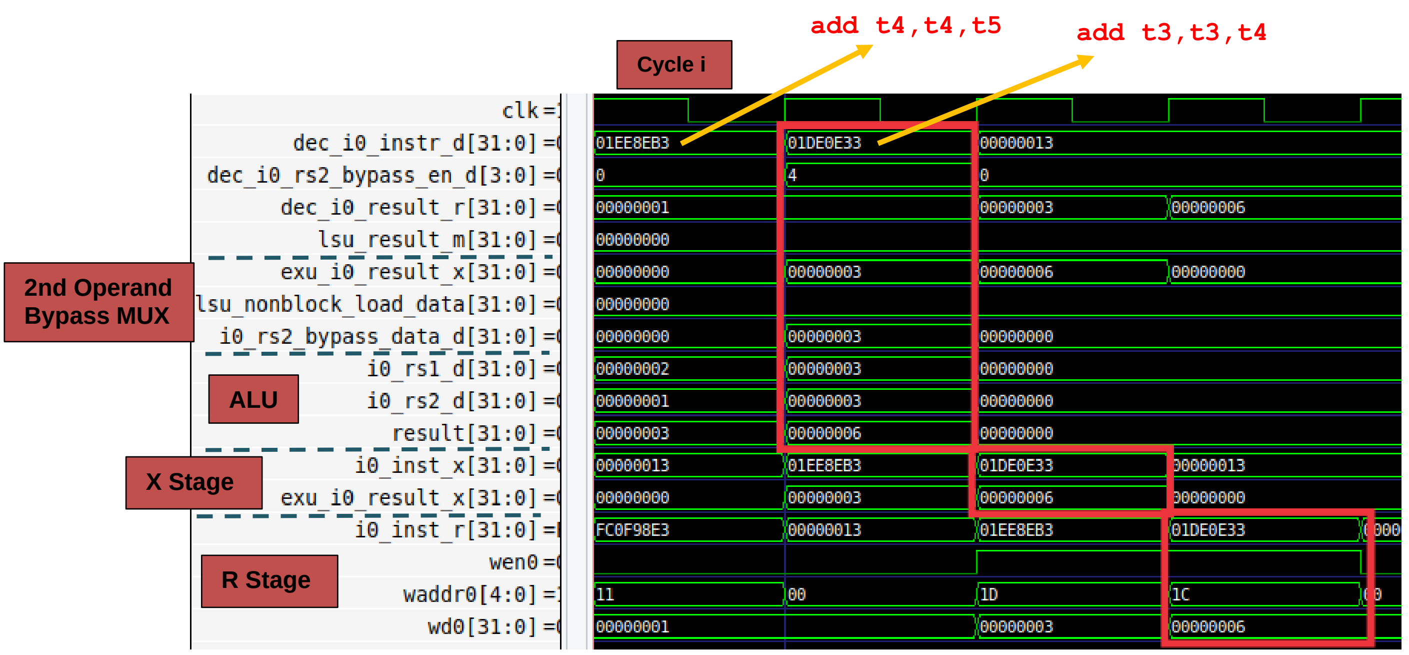
add t4,t4,t5
add t3,t3,t4
Cycle i
2nd Operand Bypass MUX
ALU
X Stage
R Stage
clk
dec_i0_instr_d[31:0] 01EE8EB3 01DE0E33 00000013
dec_i0_rs2_bypass_en_d[3:0] 0 4 0
dec_i0_result_r[31:0] 00000001 00000003 00000006
lsu_result_m[31:0] 00000000
exu_i0_result_x[31:0] 00000000 00000003 00000006 00000000
lsu_nonblock_load_data[31:0] 00000000
i0_rs2_bypass_data_d[31:0] 00000000 00000003 00000000
i0_rs1_d[31:0] 00000002 00000003 00000000
i0_rs2_d[31:0] 00000001 00000003 00000000
result[31:0] 00000003 00000006 00000000
i0_inst_x[31:0] 00000013 01EE8EB3 01DE0E33 00000013
exu_i0_result_x[31:0] 00000000 00000003 00000006 00000000
i0_inst_r[31:0] FC0F98E3 00000013 01EE8EB3 01DE0E33
wen0
waddr0[4:0] 11 00 1D 1C 0
wd0[31:0] 00000001 00000003 00000006

Figure 4. Simulation of Figure 2 example code

TASK: Replicate the RVfpgaEL2-Trace simulation from Figure 4 on your own computer. You can use the *.tcl* file provided in: *[RVfpgaBooleanPath]/Labs/Lab15/DataHazards_AL-AL/test_Basic.tcl*.

© Copyright Imagination Technologies

Finally, Figure 5 shows the RVfpgaEL2-Pipeline simulation in cycle i+1. Note that the second operand for the second `add` instruction is forwarded from the X Stage. You can detect this because the *Bypass1* signal (highlighted in red) contains the same value as the *ALU-Res* signal (*Bypass1* = *ALU-Res* = 3). In contrast, the *ra1/rd1* signals are black, to indicate that they are not being used during this cycle.

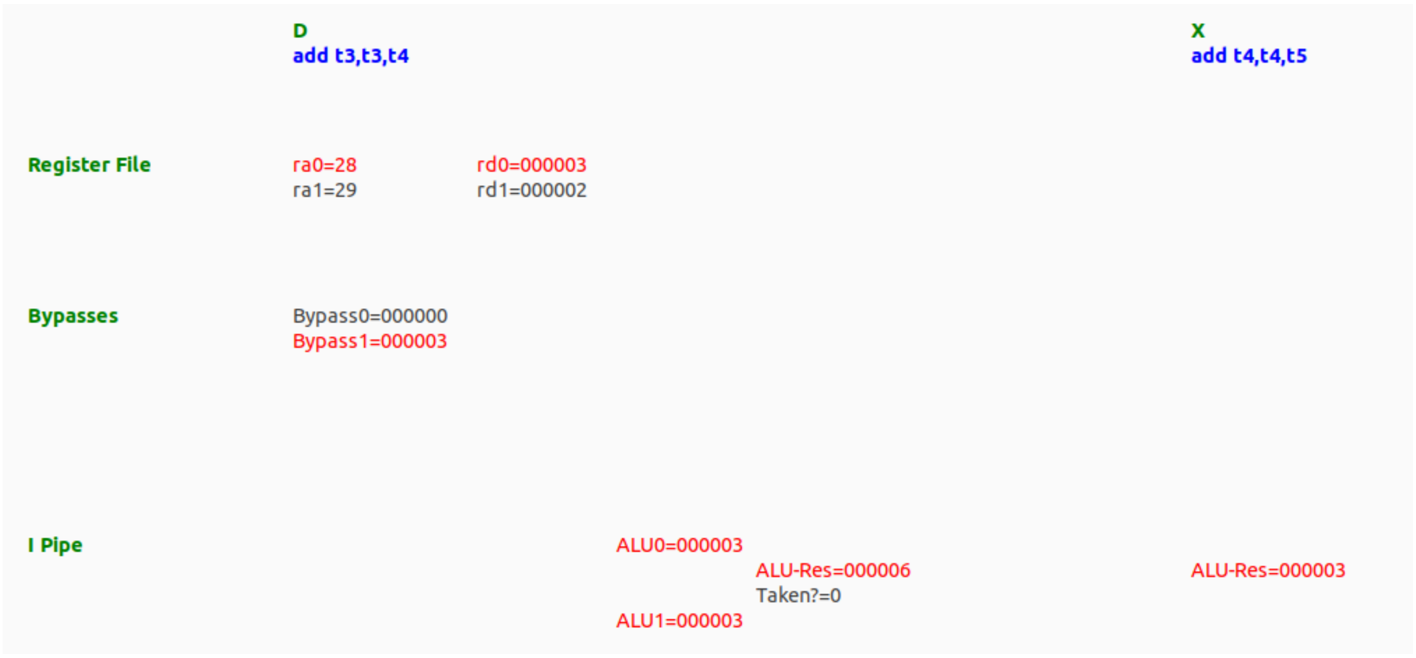

Figure 5. Result forwarded from X to D (second operand) in cycle *i*+1

<u>TASK</u>: Replicate the RVfpgaEL2-Pipeline simulation from Figure 5 on your own computer. Remember that you must include the control instruction for this simulator to work correctly.

Analyse the simulations from Figure 4 and Figure 5 and the diagram from Figure 3 at the same time.

- Instruction `add t4,t4,t5` (0x01ee8eb3):
    - In cycle *i*, this instruction is in the D Stage (`dec_i0_instr_d` = 0x01ee8eb3).
      It computes the following addition in the ALU:
      `i0_rs1_d` (2) + `i0_rs2_d` (1) = `result` (3)
    - In cycle *i+1*, this instruction is in the X Stage (`i0_inst_x` = 0x01ee8eb3). The result of the addition (`exu_i0_result_x` = 3) is provided as an input to the Forwarding Logic in the D Stage.

- Instruction `add t3,t3,t4` (0x01de0e33):
    - In cycle *i+1*, this instruction is in the D Stage (`dec_i0_instr_d` = 0x01de0e33).
      The Forwarding Logic connects `exu_i0_result_x` with `i0_rs2_bypass_data_d`.
      The ALU receives the output of the 4:1 and 3:1 multiplexers:
      `i0_rs1_d` = 3 (from the Register File)

© Copyright Imagination Technologies

`i0_rs2_d` = 3 (from the ALU output in the X stage of the I Pipe, through the Forwarding Logic, signal `i0_rs2_bypass_data_d`)

The ALU computes the addition: `result` = 6.

<u>TASK</u>: Remove all `nop` instructions in the example from Figure 2. Generate the trace with the RVfpgaEL2-Trace simulator, analyse the simulation on RVfpgaEL2-Pipeline, and then compute the IPC by using the Performance Counters while executing the program on the board (remember that you must uncomment all instructions in the main function, in file *Test.c*). You can obtain the values of the Performance Counters both in the physical board and in RVfpgaEL2-ViDBo.

3. Exercises

<u>Exercise 1</u>: In the example from Figure 2, analyse and explain similar situations where you replace the dependent `add` instruction with 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)
  ```

<u>Exercise 2</u>: Use the project called *DataHazards_LW-AL* to analyse a hazard between a load (`lw`) and an `add` instruction. Analyse the following two situations:

- **DCCM**: The load reads from the DCCM in a single cycle. You can use the tcl script called *test_DCCM.tcl*. For mapping the data to the DCCM, uncomment, in file Test_Assembly.S, line: `.section .midccm`, and comment line: `.section .ram`

- **Main Memory**: The load reads from Main Memory in several cycles. You can use the tcl script called *test_MainMemory.tcl*. For mapping the data to Main Memory, uncomment, in file Test_Assembly.S, line: `.section .ram`, and comment line: `.section .midccm`

Simulate the program both in RVfpgaEL2-Trace and RVfpgaEL2-Pipeline (do not forget to include the control instruction for the latter).

© Copyright Imagination Technologies