THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2 Lab 12

Arithmetic/Logic Instructions: The `add` Instruction

1. Introduction

In this lab, we analyse the flow of arithmetic and logical instructions through the stages of the VeeR EL2 pipeline. Figure 1 shows a high-level view of EL2's microarchitecture, with the stages that we analyse in this lab highlighted in red: D Stage, X Stage and R Stage.

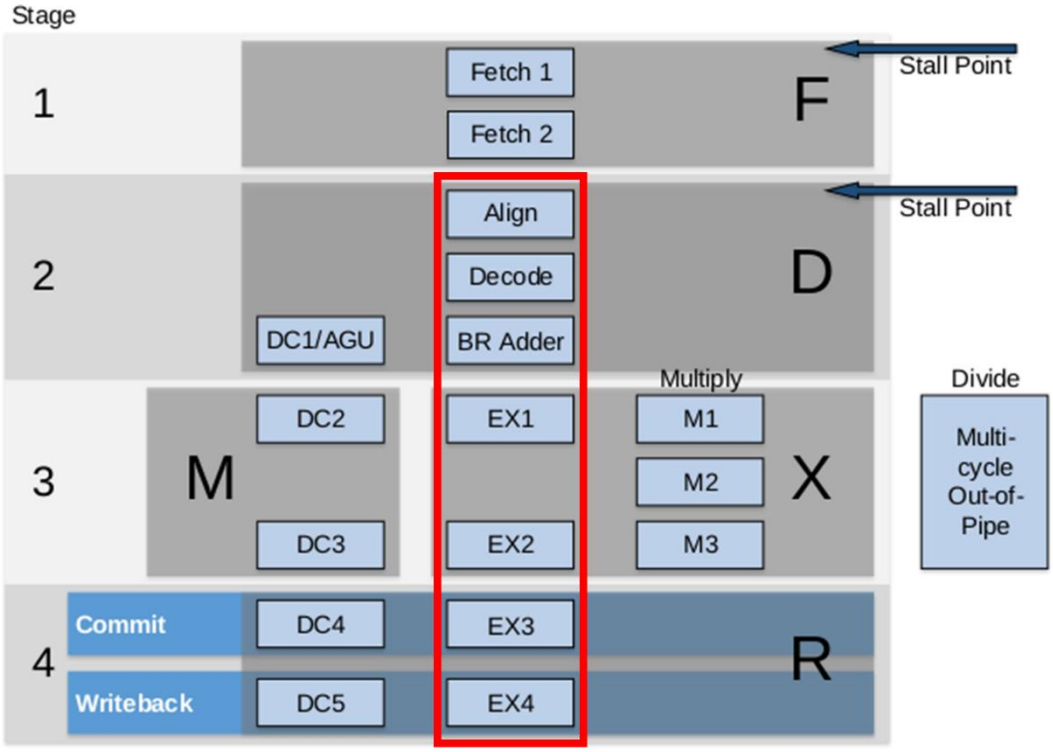


Figure 1. VeeR EL2 pipeline; Path of an `add` instruction highlighted

In Section 2 we analyse an `add` instruction from the D to R stages, when it writes the result to the Register File. During the explanations, we interleave a simulation of an `add` instruction that you should replicate on your own computer. In Section 3, we provide exercises for analysing other Arithmetic-Logic instructions following a similar procedure as the one described for the `add` instruction.

2. Analysis of the VeeR EL2 Core for an `add` instruction

Throughout this section, we will work with the example shown in Figure 2, which executes an `add` instruction contained within a loop that repeats forever. Folder *[RVfpgaBooleanPath]/Labs/Lab12/ADD_Instruction* provides the Catapult project so that you can analyse, simulate, and change the program as desired. In this project we disable the use of compressed instructions for simplicity, and as explained in Section 2 of the VeeRref document. Moreover, for convenience, we insert the `add` instruction in an infinite loop, which allows us to inspect it with no Instruction Cache (I$) misses if we avoid the first iteration for our analysis. This also makes it easy to find the region of interest in the simulation. Finally, the `add` instruction (highlighted in red in Figure 2) is surrounded by several `nop` (no-operation) instructions in order to isolate it from preceding/subsequent `add` instructions that belong to other iterations of the loop.

```
.globl main
main:

li t3, 0x4                  # t3 = 4
li t4, 0x1                  # t4 = 1

REPEAT:
   INSERT_NOPS_10
   add t3, t3, t4           # t3 = t3 + t4
   INSERT_NOPS_10
   beq  zero, zero, REPEAT # Repeat the loop

.end
```

Figure 2. Example program with an `add` instruction

If you open the project in Catapult, build it, and open the disassembly file, you will find the `add` instruction (0x01de0e33) as follows:

```
0x000002f0:      01de0e33                add   t3,t3,t4
```

TASK: Verify that these 32 bits (0x01de0e33) correspond to instruction `add t3,t3,t4` in the RISC-V architecture.

A. Basic analysis of the `add` instruction

Figure 3 shows the RVfpgaEL2-Trace simulation of the program from Figure 2 for the execution of the `add` instruction in the fourth iteration of the loop. The figure includes some signals associated with EL2's D, X and R stages. The values highlighted in red correspond to the `add` instruction as it traverses these stages through the I Pipe.

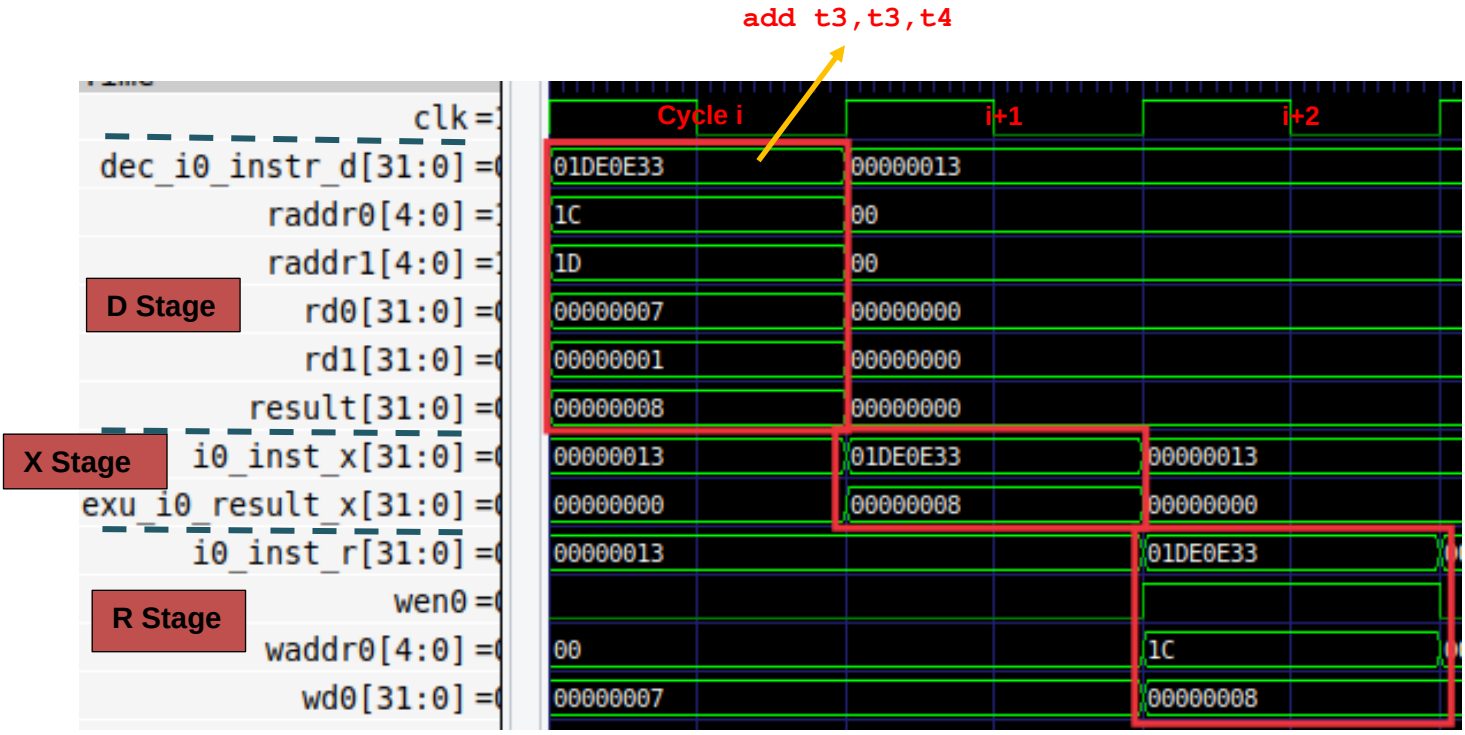


Figure 3. RVfpgaEL2-Trace simulation for the example program in Figure 2

Figure 4 shows a simplified diagram of the VeeR EL2 pipeline executing the `add` instruction during the fourth iteration of the loop (see program in Figure 2) through the I Pipe. Note that the figure merges the state of the processor in different cycles:

- **Cycle i - D Stage:** The instruction is decoded, the Register File is read and the addition is completed in the ALU.
- **Cycle i+1 – X Stage:** The result is selected and propagated to the next stage.
- **Cycle i+2 – R Stage:** The result of the addition is written to the Register File using write port 0.

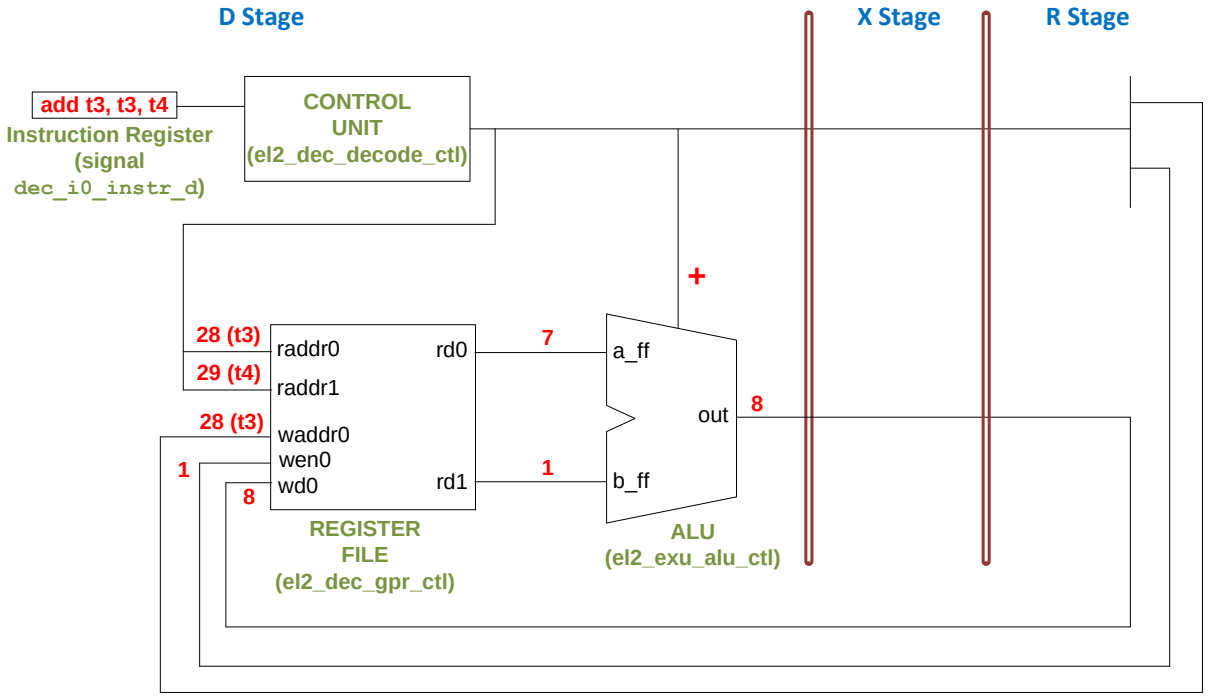


Figure 4. VeeR EL2 pipeline executing an `add` instruction

TASK: Replicate the simulation from Figure 3 on your own computer. To do so, follow the steps described in detail in the GSG.

We follow the `add` instruction through the pipeline by analysing the waveform from Figure 3 and the diagram from Figure 4 at the same time, and as described below.

- **Cycle i – D Stage:** Signal `dec_i0_instr_d` contains the 32-bit machine instruction 0x01DE0E33. In RISC-V, the opcode for the `add` instruction is (see Appendix B of [DDCARV]):

  ```
  00      | rs1 | 000 | rd | 0110011
  ```

  You can easily verify that 0x01DE0E33 corresponds to: `add t3, t3, t4` (remember that `t3` = x28 and `t4` = x29).
  In this stage:
  - The **control signals** are generated in the Control Unit. Among other things, the address of the registers is determined (`raddr0` = 0x1C, `raddr1` = 0x1D) and the ALU operation is chosen (in this case as addition).
  - The **Register File is read**, in this case providing the values `rd0` = 7 and `rd1` = 1.
  - The I pipe computes the addition in the ALU, in this case `result` = 8.

- **Cycle i+1 – X Stage:** The result of the addition is selected in several multiplexers (not shown in the figure) and propagated to the next stage (`exu_i0_result_x` = 8).

- **Cycle i+2 – R Stage:** Finally, the result of the addition is **written-back** to the Register File through signal `wd0` = 8, which contains the data to write. Given that `wen0` = 1 (write enable) in this cycle, the result of the addition is written at the end of the cycle into

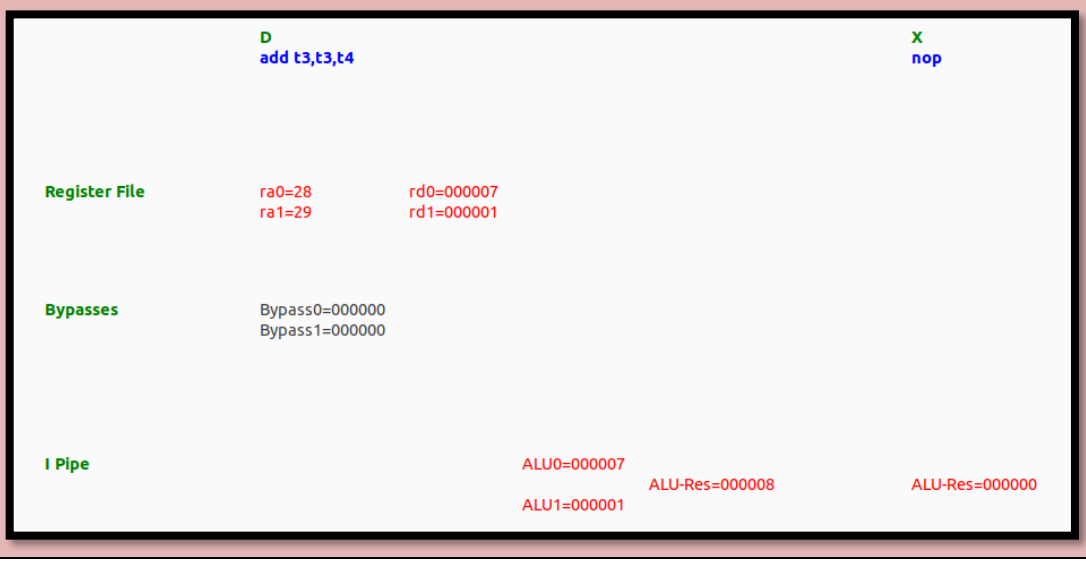
register `x28` (shown in hexadecimal, `waddr0` = 0x1C).

Remember that GTKwave allows you to easily change the data format of a signal. To do so, place the cursor on the signal, click on the right button of the mouse, and select the desired "Data Format". For example, it may be more convenient to see `waddr0` in decimal format (28) instead of hexadecimal (0x1C), as shown in Figure 5.

Figure 5. Signal `waddr0` shown in decimal format.

TASK: Simulate the previous program on RVfpgaEL2-Pipeline in order to perform a similar analysis to the one just carried out in the previous text and in Figure 3. 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 in *cycle i* shown in Figure 3.



B. Advanced analysis of the `add` instruction

In this section we analyse the stages traversed by the `add` instruction, from D to R, in more detail than in Section A and we progressively add more signals to the simulation from Figure 3.

Figure 6 shows a detailed (but still simplified) diagram of the main elements that an `add` instruction traverses during its execution through the I Pipe. This was already illustrated in Figure 4 of Lab 11 (we recommend comparing both figures), but we now focus only on the I pipe and provide details related to the `add` instruction. You may need to zoom into the figure to be able to see the details. The names of the control signals are shown in red whereas the names of the data signals are shown in black. These names are the actual names used in the VeeR EL2 Verilog modules. Equal symbols (`=`) represent signal assignments in the Verilog code.

TASK: Locate the main structures and signals from Figure 6 in the Verilog files of the VeeR EL2 processor.

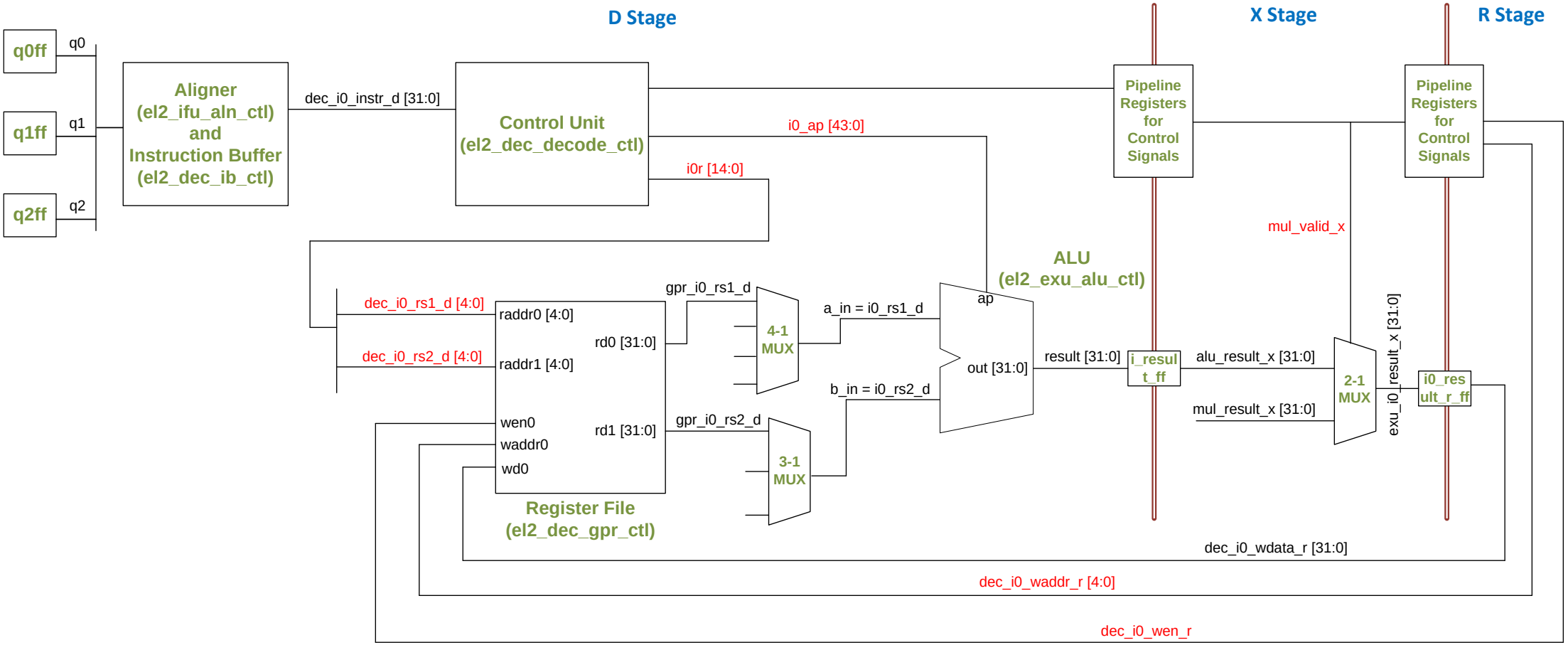


Figure 6. Main units used by Arithmetic-Logic instructions flowing through the I pipe

Figure 7 extends the RVfpgaEL2-Trace simulation from Figure 3 by adding the signals introduced in Figure 6. In the following sections we analyse each stage in detail, looking at the newly introduced signals.

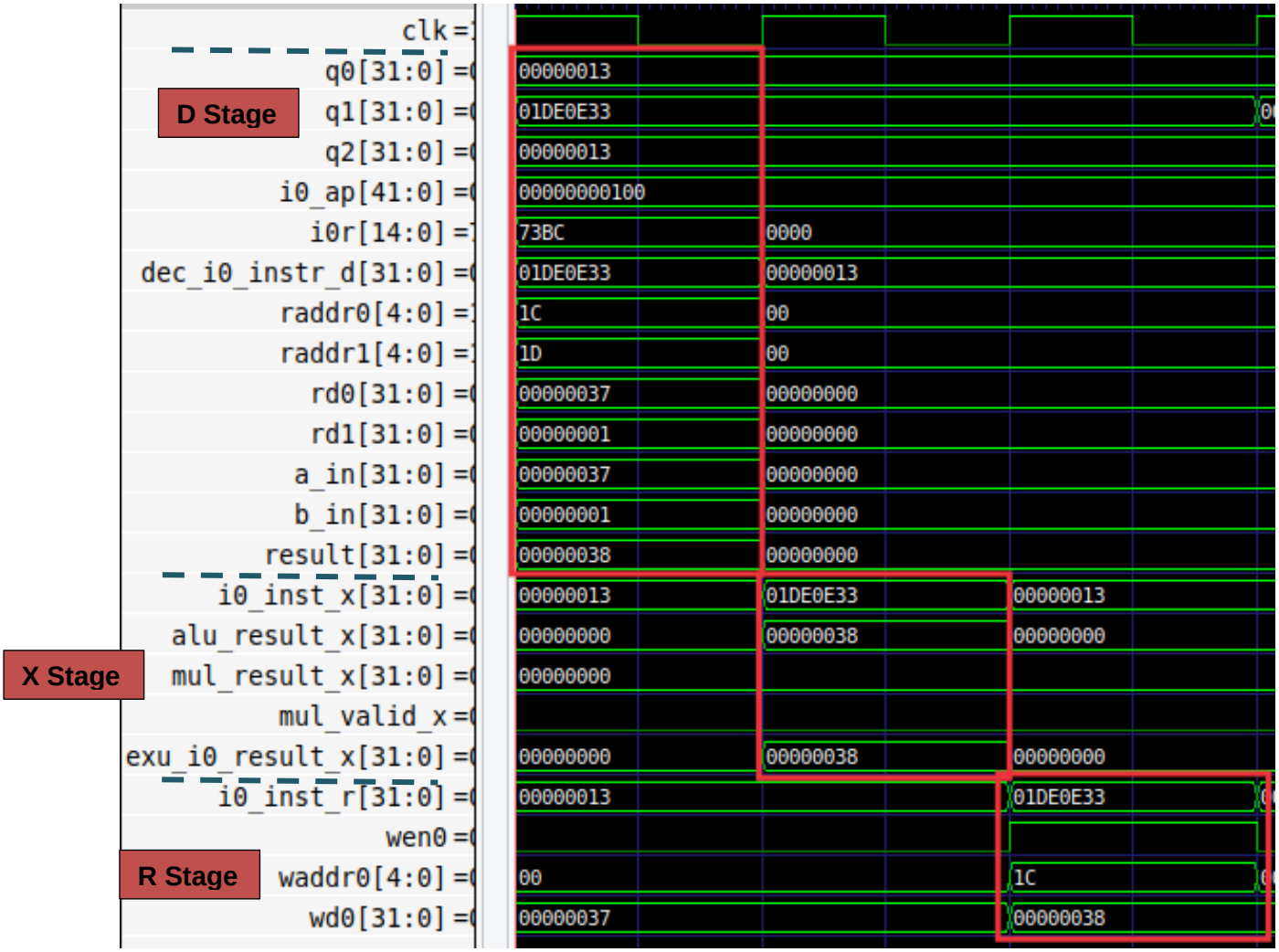


Figure 7. RVfpgaEL2-Trace simulation of the example program of Figure 2, including control signals and Register File read ports.

i. D Stage

As explained in Lab 11, the D Stage is responsible for many of the tasks that Arithmetic-Logic instructions have to perform:

- **Align the instruction.**
- **Decode the instruction and generate control signals.**
- **Read or assemble the source operands and send the instruction to the appropriate pipe.**

- **Perform the operation in the ALU**

We next analyse each of these tasks for the `add` instruction by analysing the waveform from Figure 7, the diagram from Figure 6 and some parts of the SoC Verilog code.

Align the instruction:

As shown in Figure 6, the three buffers store different instructions that might be executed soon. They store both compressed (16-bit) and uncompressed (32-bit) instructions. The Aligner selects one instruction for execution and uncompresses it if necessary (remember that in our example from this lab we have disabled the use of compressed instructions).

Specifically, in the waveform from Figure 7 we can see the three instructions stored in the buffers and the instruction selected by the Aligner when the `add` instruction is in the D Stage:

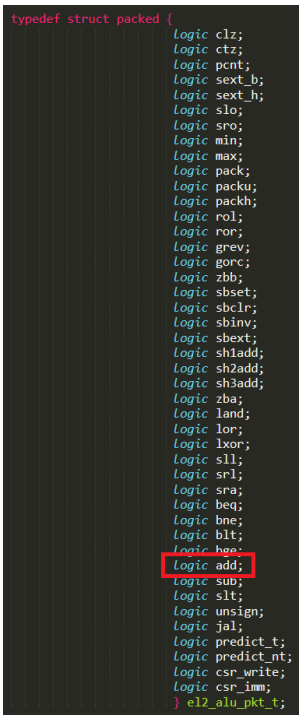
- `q0` = **0x00000013** (`nop` instruction)
- `q1` = **0x01DE0E33** (`add` instruction)
- `q2` = **0x00000013** (`nop` instruction)
- `dec_i0_instr_d` = **0x01DE0E33** (`add` instruction)

If you are interested in knowing more details about the Aligner, you can inspect module **`el2_ifu_aln_ctl`** (file *[RVfpgaBooleanPath]/src/VeeRwolf/VeeR_EL2CoreComplex/ifu/el2_ifu_aln_ctl.sv*).

Decode the instruction and generate control signals:

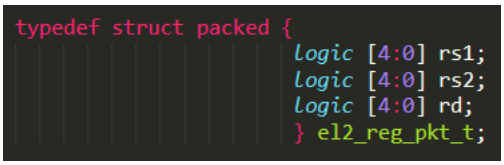
As explained in Section 2.C.ii of Lab 11, several structures are defined at *[RVfpgaBooleanPath]/src/VeeRwolf/VeeR_EL2CoreComplex/include/el2_def.sv* for grouping the control bits. Some of these structures are directly related to Arithmetic-Logic (A-L) instructions.

- **`el2_alu_pkt_t`**: This is the main structure for A-L instructions:



One signal of this type, called `i0_ap`, is defined and assigned inside module **`el2_dec_decode_ctl`** at the D Stage, and used by the ALU in this same stage. Specifically, when an `add` instruction is executed, as can be seen in Figure 7, `i0_ap` sets bit `add` to 1 and all remaining bits to 0.

- **`el2_reg_pkt_t`**: This signal contains the numbers of the two source registers (fields `rs1` and `rs2`) and the destination register (field `rd`):



A signal of this type, called `i0r`, is defined, assigned and used inside module **`el2_dec_decode_ctl`**. Specifically, for the `add` instruction of our example program, as can be seen in Figure 7, `i0r` = 0x73BC = 111 0011 1011 1100, thus `rs1` = 28, `rs2` = 29 and `rd` = 28.

Some of these control signals are used in the D stage and are not propagated through the Control Pipeline Registers to later stages. This is the case for `i0r.rs1` and `i0r.rs2` which are no longer needed after the D stage, because they are used by the Register File during the D Stage to read the two input operands (signals `raddr0`, `raddr1`).

TASK: Find in the Verilog code (module **`el2_dec_decode_ctl`**) how the `i0r` control signal is used for reading the Register File during the D Stage.

However, other control signals must be propagated to later stages. This is the case, for example, for signal `mul_p.valid`, which is used by the multiplexer in the X Stage (`mul_valid_x`) shown in Figure 6.

TASK: Find in the Verilog code (module **el2_exu**) how the `mul_p.valid` signal is propagated from the D Stage to the X Stage.

Read or assemble the source operands and send the instruction to the appropriate pipe:

As explained in Lab 11, the VeeR EL2 processor includes several pipes for executing the instructions. In the D Stage, the instructions, once decoded, must be assigned to the appropriate pipe. Specifically, if an A-L instruction has been decoded it must be sent, if possible, to the I Pipe. In the program that we are analysing in this lab (Figure 2), once the processor has decoded the `add` instruction (i.e., it "knows" that it is an A-L instruction that, thus, executes in the I Pipe), it must check if all the conditions for execution through the I pipe are met: Valid decoding?, 2 input operands available?, Pipeline not blocked? Etc. In our case, the result of this check is sent to the I pipe through two status signals that are computed in the **el2_dec_decode_ctl** module and that are used by the ALU in the **el2_exu** module (in the next subsection we will explain the ALU in more detail). These two status signals are:

- `x_data_en` (renamed `enable` inside the ALU): This signal depends on `dec_data_en[1:0]`, which establishes, at decode time, if the stages must be enabled (1) or not (0). Note that the pipeline could be blocked due to different circumstances, such as a wrong branch prediction, a division computation, etc.

- `dec_i0_alu_decode_d` (renamed as `valid_in` inside the ALU): This signal is 1 if the instruction has been legally decoded and it is an Arithmetic/Logic instruction.

Both signals must be 1 for the ALU to perform the `add` operation in the next stage (EX1 Stage).

TASK: The generation of these two signals is quite a complex process that we do not explain here in detail but that you can further analyse on your own in modules **el2_dec_decode_ctl** and **el2_exu**.

As also explained in Lab 11, the input operands are provided to the I pipe (`i0_rs1_d` and `i0_rs2_d`) through two multiplexers implemented in the D Stage (see Figure 6). In the `add` instruction from our example, both input operands are obtained directly from the Register File:

- First input operand: `i0_rs1_d[31:0] = gpr_i0_rs1_d[31:0]`
- Second input operand: `i0_rs2_d[31:0] = gpr_i0_rs2_d[31:0]`

TASK: Find in the Verilog code (module **el2_exu**) the 3:1 multiplexer on the bottom of Figure 6 (second input operand) and try to find the origin of its inputs (in Figure 6 only the input coming from the Register File is shown).

TASK: Replicate and analyse the simulation from Figure 7 on your own computer. You can use the *.tcl* script provided at: *[RVfpgaBooleanPath]/Labs/Lab12/ADD_Instruction/test1_Extended.tcl*. Add to the simulation from Figure 7 the signals explained above (such as `x_data_en` or

`dec_i0_alu_decode_d`) and confirm that they have the correct value.

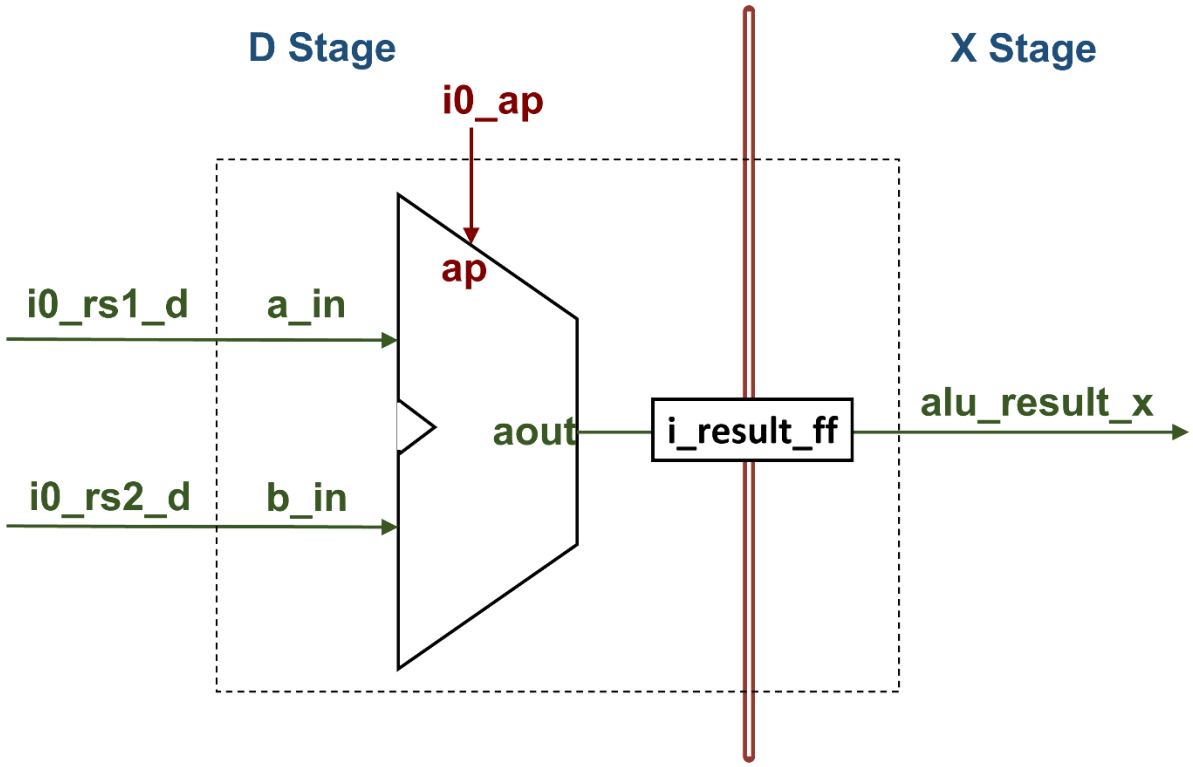
TASK: In the example from Figure 2, replace the `add` instruction with a non A-L instruction (such as a `mul` instruction) and analyse the control signals.

Perform the operation in the ALU:

As explained in Lab 11, VeeR EL2 includes three execution pipes: Integer Pipe, Multiply Pipe, and L/S Pipe. In this lab we focus on the I Pipe, where the `add` instruction is executed. The main task of the I pipe for an `add` instruction is to compute the addition in the ALU and propagate it to the R Stage.

During the D Stage, the ALU operation is performed – in this case, an addition. The Arithmetic-Logical Unit (ALU) of VeeR EL2 is implemented in module **`el2_exu_alu_ctl`** (which can be found in *[RVfpgaBooleanPath]/src/VeeRwolf/VeeR_EL2CoreComplex/exu/el2_exu_alu_ctl.sv*), and it is instantiated in module **`el2_exu`** (which can be found at *[RVfpgaBooleanPath]/src/VeeRwolf/VeeR_EL2CoreComplex/exu/el2_exu.sv*).

Figure 8 shows the instantiation and a simplified diagram of the ALU with some of its input/output ports. Note that most input/output signals are renamed inside the ALU.



```
el2_exu_alu_ctl #(.pt(pt)) i_alu  (.*,
                      .enable            ( x_data_en                  ),
                      .pp_in             ( i0_predict_newp_d          ),
                      .valid_in          ( dec_i0_alu_decode_d        ),
                      .flush_upper_x     ( i0_flush_upper_x           ),
                      .flush_lower_r     ( dec_tlu_flush_lower_r      ),
                      .a_in              ( i0_rs1_d[31:0]             ),
                      .b_in              ( i0_rs2_d[31:0]             ),
                      .pc_in             ( dec_i0_pc_d[31:1]          ),
                      .brimm_in          ( dec_i0_br_immed_d[12:1]    ),
                      .ap                ( i0_ap                      ),
                      .csr_ren_in        ( dec_csr_ren_d              ),
                      .csr_rddata_in     ( dec_csr_rddata_d[31:0]     ),
                      .result_ff         ( alu_result_x[31:0]         ),
                      .flush_upper_out   ( i0_flush_upper_d           ),
                      .flush_final_out   ( exu_flush_final            ),
                      .flush_path_out    ( i0_flush_path_d[31:1]      ),
                      .predict_p_out     ( i0_predict_p_d             ),
                      .pred_correct_out  ( i0_pred_correct_upper_d     ),
                      .pc_ff             ( exu_i0_pc_x[31:1]          ));
```

Figure 8. I's ALU (el2_exu_alu_ctl module): High-level diagram and Verilog code

ALU Inputs: The ALU inputs (`a_in` and `b_in`) are selected in the D Stage by the two multiplexers shown in Figure 6, as explained in the previous section. Inside the **`exu_alu_ctl`** module, the inputs are used by the ALU to perform the operation when both the `valid_in` and `enable` signals are 1.

ALU Control Signals: The ALU is governed by the control bits generated in the D stage and provided in signal `i0_ap`.

ALU Output: The ALU output is given in signal `result` and then it is propagated to the X Stage through signal `result_ff` and renamed as `alu_result_x` (see Figure 8):

```
rvdffe    #(32) i_result_ff  (.*, .clk(clk), .en(enable & valid_in),   .din(result[31:0]),   .dout(result_ff[31:0]));
```

TASK: Include the new signals analysed in this section in the simulation from Figure 7.

TASK: Perform a simulation of a `sub` instruction similar to the one from Figure 7. You can include new signals in the simulation as needed for your analysis.

TASK: Analyse the Verilog implementation of the adder/subtractor implemented in module **`el2_exu_alu_ctl`**. Figure 9 gives you some help by showing the logic directly related with addition and subtraction operations. You can use an RVfpgaEL2-Trace simulation as a starting point.

```
assign zba_a_in[31:0]      = ( {32{ ap_sh1add}} & {a_in[30:0],1'b0} ) |
                             ( {32{ ap_sh2add}} & {a_in[29:0],2'b0} ) |
                             ( {32{ ap_sh3add}} & {a_in[28:0],3'b0} ) |
                             ( {32{~ap_zba    }} &  a_in[31:0]      );

logic        [31:0]    bm;

assign bm[31:0]            = ( ap.sub )  ?  ~b_in[31:0]  :  b_in[31:0];

assign {cout, aout[31:0]}  = {1'b0, zba_a_in[31:0]} + {1'b0, bm[31:0]} + {32'b0, ap.sub};
```

Figure 9. Adder inside el2_exu_alu_ctl

ii. X Stage

This stage does little for an independent `add` instruction. In this example, input `alu_result_x` is selected by the 2:1 multiplexer available in this stage (see Figure 6). In our example from Figure 2, the value selected is again the result provided by the I pipe (`exu_i0_result_x` = `alu_result_x`).

iii. R Stage

In the last stage, the result of the `add` instruction is written to the Register File as shown in Figure 6. The 32-bit result (`dec_i0_wdata_r[31:0]`) was computed in the D Stage and was propagated to this stage. The register address (in Figure 7 signal `waddr0` is shown in hexadecimal, but it could be shown in decimal as explained before) and the write enable signals are provided through the Control Pipeline Registers.

TASK: In the Verilog code, analyse how signals `wen0` and `waddr0` are generated in the D Stage and propagated to the R Stage.

3. Exercises

1) Perform a similar analysis to the one presented in this lab for logical instructions: `and`, `or`, and `xor`.

2) (*The following exercise is based on exercise 4.1 from the book "Computer Organization and Design – RISC-V Edition", by Patterson & Hennessy ([PaHe]).*)
   Consider the following instruction: `and rd, rs1, rs2`
   a. What are the values of control signals generated by VeeR EL2 for this instruction?
   b. Which resources (modules or blocks of code) perform a useful function for this instruction?
   c. Which resources (modules or blocks of code) produce no output or output that is not used for this instruction?

3) Analyse in an RVfpgaEL2-Trace simulation and directly in the Verilog code, the *shift left/right* instructions available in the RV32I Base Integer Instruction Set: `srl`, `sra`, and `sll`.

4) Analyse, both in an RVfpgaEL2-Trace simulation and directly in the Verilog code, the

set less than instructions available in the RV32I Base Integer Instruction Set: `slt` and `sltu`.

5) Analyse, both in an RVfpgaEL2-Trace simulation and directly in the Verilog code, some of the *immediate* instructions available in the RV32I Base Integer Instruction Set: `addi`, `andi`, `ori`, `xori`, `srli`, `srai`, `slli`, `slti`, and `sltui`.

6) (*The following exercise is based on exercise 4.6 of [PaHe].*)
   Figure 6 does not discuss I-type instructions like `addi` or `andi`.
   a. What additional logic blocks, if any, are needed to support execution of I-type instructions in VeeR EL2? Add any necessary logic blocks to Figure 6 and explain their purpose.
   b. List the values of the signals generated by the control unit for `addi`.

7) (*The following exercise is based on exercise 4.4 of [PaHe] and exercise 1 of Chapter 7 of the textbook by S. Harris and D. Harris, "Digital Design and Computer Architecture: RISC-V Edition" [DDCARV].*)
   When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one signal wire to get "broken" and always register a logical 0. This is often called a "stuck-at-0" fault. Determine the effect of each of the control bits included in signal `i0_ap` (a signal of type `el2_alu_pkt_t`) being stuck at 0.