THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 11

VeeR EL2 Configuration, Organization, and Performance Monitoring

1. Introduction

In the first nine RVfpga labs (Labs 1–9), we introduced the RISC-V architecture and how to communicate with the VeeR EL2 Core using various peripherals. In the next eight labs (Labs 11–16 and 18–19), we will dive down to the microarchitectural level and analyse how the VeeR EL2 processor operates internally and how the cache/memory hierarchy works. Note that the EL2 version of the RVfpga labs do not include Lab 17, which discusses superscalar execution in the VeeR EH1 version of RVfpga, because EL2 is a scalar processor. Moreover, we do not include Lab 20 in the Basys 3 version, as the small FPGA size makes the use of a DCCM difficult if not impossible in our system (if you are interested in doing this lab, you can still do it in simulation using any of the other boards).

SIGASI STUDIO: In these labs we are going to deal with an extensive Verilog project: the VeeR EL2 Core RTL. One way of analysing the various modules and signals is to use a typical editor such as Sublime Text (https://www.sublimetext.com/), which offers interesting functionalities for navigating through a project, inspecting the files, looking for strings, etc. However, more suitable and specific alternatives are available, such as **Sigasi Studio** (https://www.sigasi.com/), which we highly recommend.

As explained in the RVfpgaEL2 Getting Started Guide (GSG), VeeR EL2 (see Figure 1) is a scalar 4-stage core with one execution pipeline, one load/store pipeline, one multiplier pipeline, and one out-of-pipeline divider. Two stall points can occur in the pipeline: 'Fetch' and 'Decode'. The figure also shows how VeeR EL2's logic stages have been shifted up with respect to VeeR EH1 and merged into 4 stages named Fetch (F), Decode (D), Execute/Memory (X/M), and Retire (R). Also shown is additional logic such as a new branch adder in the D stage, not included in VeeR EH1. The branch mispredict penalty is either 1 or 2 cycles in VeeR EL2. The merged F stage performs the program counter calculation and the I-cache/ICCM memory access in parallel. The load pipeline has been moved up so that the DC1 memory address generation (AGU) logic is now combined with align and decode logic to enable a DCCM memory access to start at the beginning of the M stage. The design supports a load-to-use of 1 cycle for smaller memories and a load-to-use of 2 cycles for larger memories. For 1-cycle load-to-use, the memory is accessed and the load data aligned and formatted for the register file and forwarding paths, all in the single-cycle M stage. For 2-cycle load-to-use, almost the entire M stage is allocated to the memory access, and the DC3/DC4 logic combined into the R stage is used to perform the load align and formatting for the register file and forwarding paths. EX3 and EX4/WB are combined into the R stage and primarily used for commit and writeback to update the architectural registers.

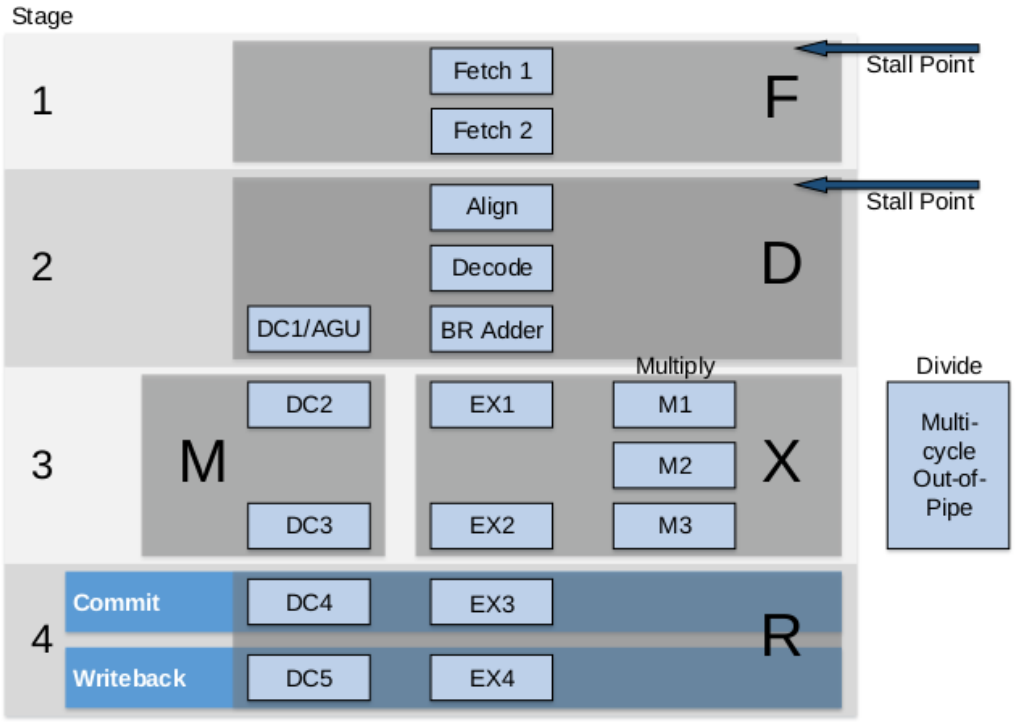


Figure 1. VeeR EL2 core microarchitecture
(figure from https://github.com/chipsalliance/Cores-VeeR-EL2/blob/main/docs/RISC-V_VeeR_EL2_PRM.pdf)

NOTE: Before starting this set of labs, we recommend that you carefully read chapters 7 and 8 of the textbook *Digital Design and Computer Architecture: RISC-V Edition* by S. Harris and D. Harris (Morgan Kaufmann © 2021). Some of the contents of these labs are inspired by that book. We will refer to the book as DDCARV.

Labs 11-16 and 18-19 begin with a theoretical explanation of the concepts and then illustrate the concepts using figures and Verilator simulations of an example program. These are toy programs that are only intended to illustrate the concept. We also provide exercises to deepen understanding of and experience with the described concepts.

One may complete only a subset of the labs, depending on the aim and depth of the course. The concepts of pipelining, memory organization, and advanced microarchitecture/memory hierarchy are covered in the following labs:

- **Pipelining:** Labs 11, 12, 13, 14, 15 and 16
- **Memory:** Labs 11, 13, and 19
- **Advanced microarchitecture:** Lab 18

In this lab (Lab 11), we begin to analyse the VeeR EL2 processor. Specifically:

- **Section 2** describes the Verilog RTL organization and details of each pipeline stage.
- **Section 3** shows how to use performance counters to analyse processor performance.
- **Section 4** describes how to configure the VeeR EL2 core.

After this initial approach, we extend this analysis to various processor units. Specifically:

- **Lab 12** focuses on **arithmetic-logic** instructions by diving deeper into the Decode, EX1/EX2/EX3, and Writeback stages.
- **Lab 13** describes **memory instructions** (loads and stores) by focusing on the DC1/DC2/DC3 stages.

- **Lab 14** discusses **structural hazards** by analyzing two situations where two different structural hazards are resolved in a different way.
- **Lab 15** analyses **data hazards** by describing the processor's bypass paths.
- **Lab 16** describes **control hazards**, and branch instructions, for which we will focus on the Fetch stage of the VeeR EL2 processor.
- **Lab 17** does not exist in the EL2 version of the RVfpga labs because it focuses on superscalar execution, which is only available in VeeR EH1, not EL2.
- **Lab 18** is a practical lab where you will add new instructions to the VeeR EL2 core.
- **Lab 19** analyzes the instruction cache (I$) included in VeeR EL2.

2. An Initial Overview of the VeeR EL2 Microarchitecture

The processor described in DDCARV has 5 pipeline stages, which are called the *Fetch, Decode, Execute, Memory,* and *Writeback* stages. The VeeR EL2 pipeline is very similar but it merges the Execute and Memory stages into a single stage, resulting in one less stage, and it slightly reorganizes some tasks between the stages, as we analyse in this lab in depth.

The remainder of this section describes the Verilog RTL organization and details of each pipeline stage. Section A describes the hierarchy of VeeR EL2's Verilog modules. Section B discusses the microarchitecture of VeeR EL2 stage-by-stage. Finally, Section C provides a practical example of the theoretical explanations given in the other sections.

A. Hierarchy of VeeR EL2's Verilog Modules

Figure 2 shows the hierarchy of the main Verilog modules (some modules are not included in the figure) that make up the VeeR EL2 processor. These modules are located in files with the same name in: *[RVfpgaBasysPath]/src/VeeRwolf/VeeR_EL2CoreComplex* directory.

The **`mem`** module instantiates the structures that make up the memory hierarchy of the VeeR EL2 processor: ICCM, DCCM and I$. The **`VeeR`** module is the overall CPU; it instantiates the modules that make up the VeeR EL2 processor: the Instruction Fetch Unit (**`ifu`**), Decode Unit (**`dec`**), Execution Unit (**`exu`**), Load/Store Unit (**`lsu`**), etc.

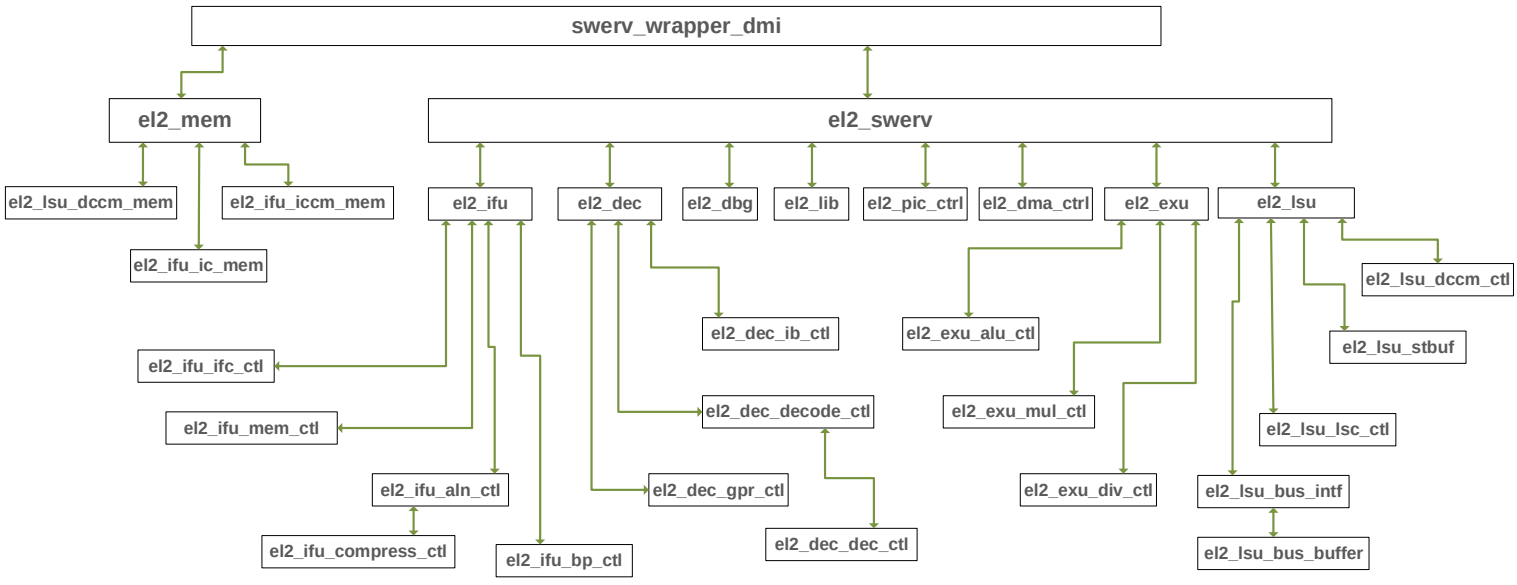


Figure 2. VeeR EL2 main modules

B. Analysis of the pipeline stages of the VeeR EL2 core

Figure 3 illustrates the VeeR EL2 pipeline, which includes four stages called F (Fetch), D (Decode), X/M (Execute/Memory), and R (Retire). The figure includes many details and it may be difficult to view it. The Visio source is provided in the package and in the following subsections, where we analyse each stage in detail. We replicate the piece of the figure that corresponds to the analysed stage.

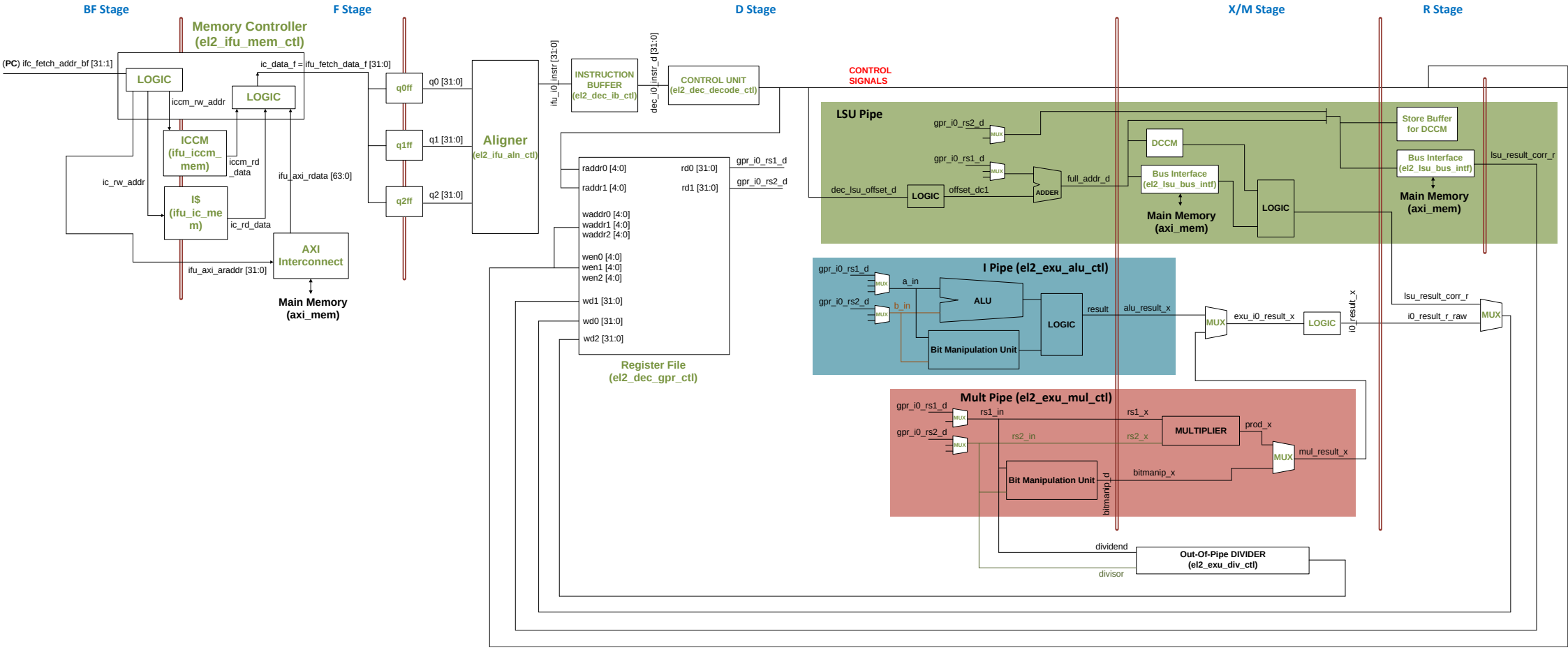


Figure 3. Simplified view of the VeeR EL2 pipeline

i. F Stage

In this section, we analyse the first stage of the pipeline, the F (Fetch) Stage. Figure 4 illustrates a very simplified view of this initial pipeline stage.

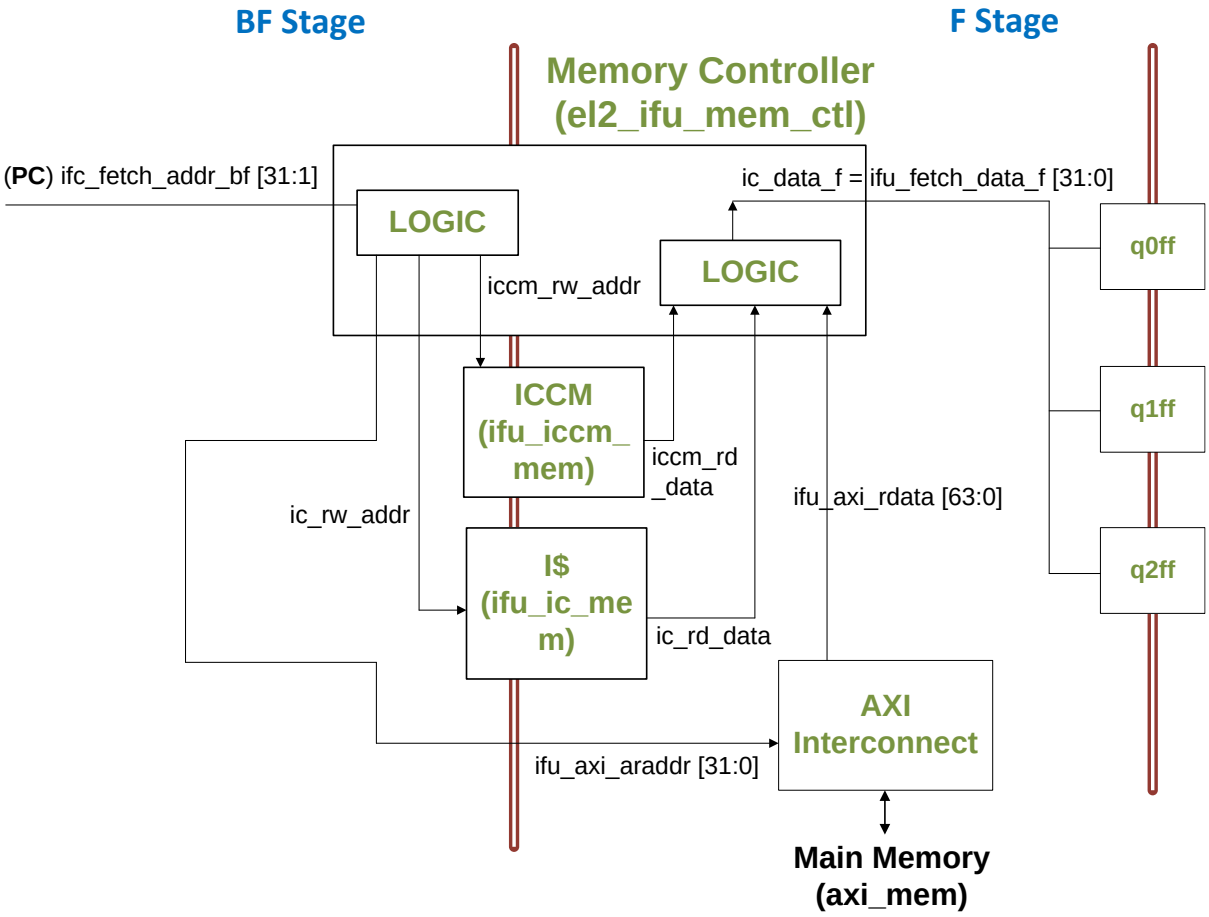


Figure 4. Simplified view of the F (Fetch) Stage

In each cycle, the F Stage is responsible for **reading the instructions from the Instruction Memory**. In our configuration, the Instruction Memory is made up by an instruction closely-coupled memory (ICCM, implemented in module **el2_ifu_iccm_mem**), an Instruction Cache (I$, implemented in module **el2_ifu_ic_mem**), and the Main Memory. Both the I$ and the ICCM are controlled by a unified memory controller (**el2_ifu_mem_ctl**), whereas the Main Memory is accessed through the AXI bus.

The default RVfpgaEL2 System's Instruction Memory is configured as follows (this configuration can be modified, as we show at the end of this lab):

- I$: 8 KiB
- Main Memory: 64 KiB Address range: `0x00000000 – 0x00010000`
- ICCM: Disabled

If the instruction address is within the Main Memory address range, the I$ provides the instruction. Upon an I$ miss, the pipeline must stall until the instruction is provided by Main Memory through the AXI bus, which takes several cycles.

Signals typically have a prefix corresponding to the unit they are a part of. For example, "`ifu`" stands for Instruction Fetch Unit. Signals append the stage they are associated with. For example, "`f`" indicates the F Stage.

As shown in Figure 4, and as we will further discuss in Lab 16, the instruction address (called the fetch address, `ifc_fetch_addr_bf`) is computed in a previous stage to the F Stage, called BF Stage, where it is provided to the Instruction Memory Controller (implemented in module **el2_ifu_mem_ctl**).

If the program has no stalls (i.e. no control, data, or structural hazards, no I$ misses, etc.), one 32-bit instruction is read every cycle in signal `ifu_fetch_data_f[31:0]`. This is enough to keep the pipeline working at its maximum throughput of 1 instruction per cycle. Three buffers (`q0ff`, `q1ff` and `q2ff`) can store up to three 32-bit instructions.

ii. D Stage

In this section we analyse the D (Decode) Stage of the VeeR EL2 pipeline. Figure 5 illustrates a simplified view of this stage, which we will extend in future labs.

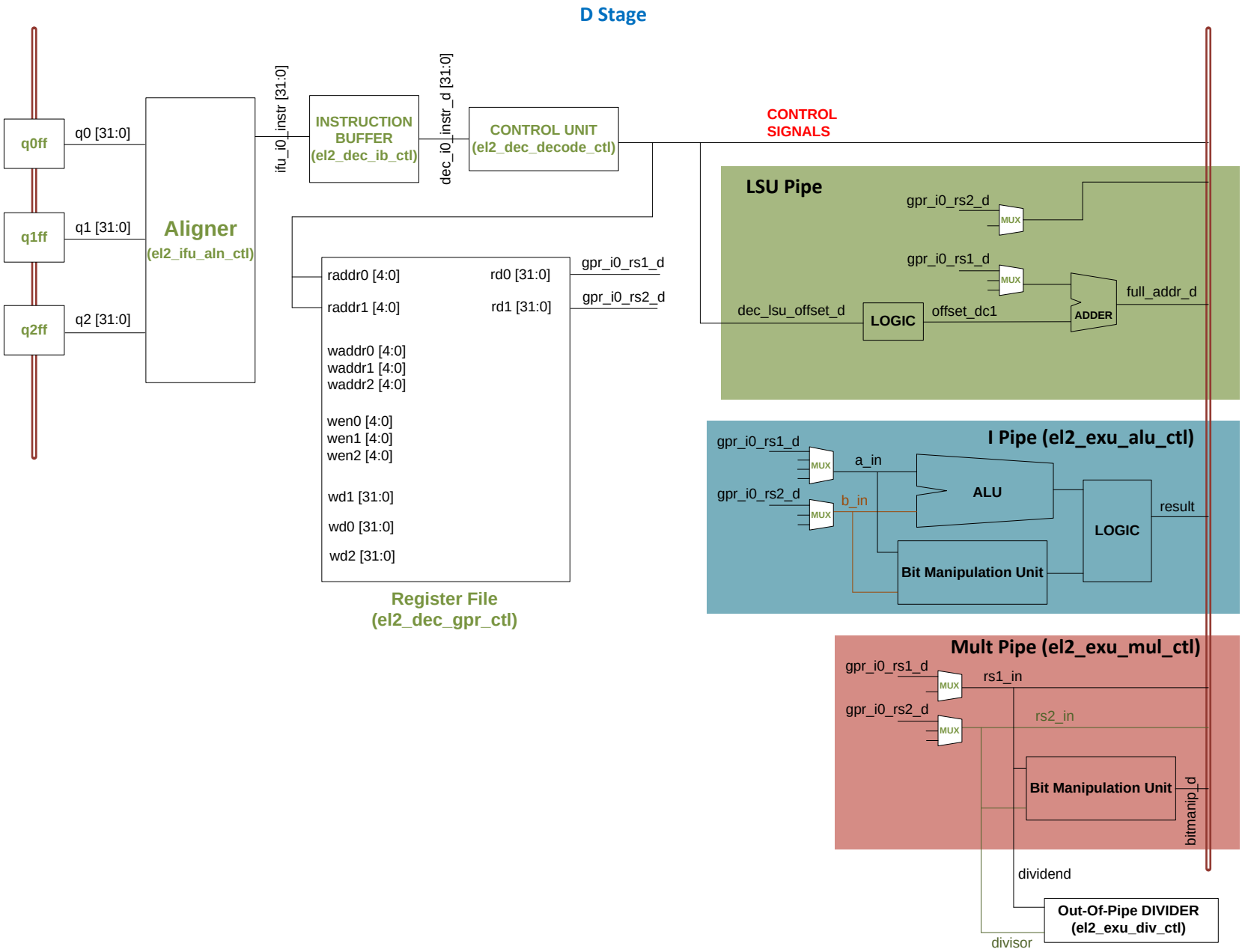


Figure 5. Simplified view of the D (Decode) Stage

Aligner

The aligner is implemented in module **`el2_ifu_aln_ctl`**. This module is responsible for performing two main tasks:

- **Select one 32-bit instruction per cycle**: The aligner selects an instruction from the three buffers `q0ff`, `q1ff` and `q2ff` and provides it to the Instruction Buffer, which then passes the instruction to the Decoder (Control Unit).

- **Uncompress instruction**: RISC-V's compressed instruction extension (RVC) reduces the size of common integer and floating-point instructions to 16 bits by reducing the sizes of the control, immediate, and register fields and by taking advantage of redundant or implied registers. This reduced instruction size decreases cost, power, and required memory (see Section 6.6.5 of DDCARV). The aligner uncompresses these 16-bit instructions, when necessary, before passing them to the decoder, which only decodes 32-bit instructions. This is performed by the **`el2_ifu_compress_ctl`** module, which is instantiated inside the aligner (module **`el2_ifu_aln_ctl`**).

Decoder

The Verilog modules for this stage are in folder *[RVfpgaBasysPath]/src/VeeRwolf/VeeR_EL2CoreComplex/dec*. In each cycle, the D Stage is responsible for two main tasks:

- **Decode the instruction and generate the control signals**: The control signals are organized in several types, as defined in file *[RVfpgaBasysPath]/src/VeeRwolf/VeeR_EL2CoreComplex/include/el2_def.sv*. Each structure/type is related to a given unit: ALU (`el2_alu_pkt_t`), Multiply Unit (`el2_mul_pkt_t`), Divide Unit (`el2_div_pkt_t`), Registers (`el2_reg_pkt_t`), etc.

  The Control Unit, implemented in module **`el2_dec_decode_ctl`**, receives the 32-bit instruction fetched, uncompressed, and aligned in the previous steps in signal `dec_i0_instr_d[31:0]` and decodes it, generating the control signals for the instruction.

  Figure 6 shows a high-level view of the Control Unit (module **`el2_dec_decode_ctl`**), which generates control signals in two stages: The first module (**`i0_dec`**) uses the instruction (`i0`) to produce overall control signals (`i0_dp_raw`, of type `el2_dec_pkt_t`), and then the second unit (**`decode`**) uses those signals to generate control signals for each pipeline path, also referred to as "pipes" (`i0_ap`, `lsu_p`, `mul_p`, etc.).

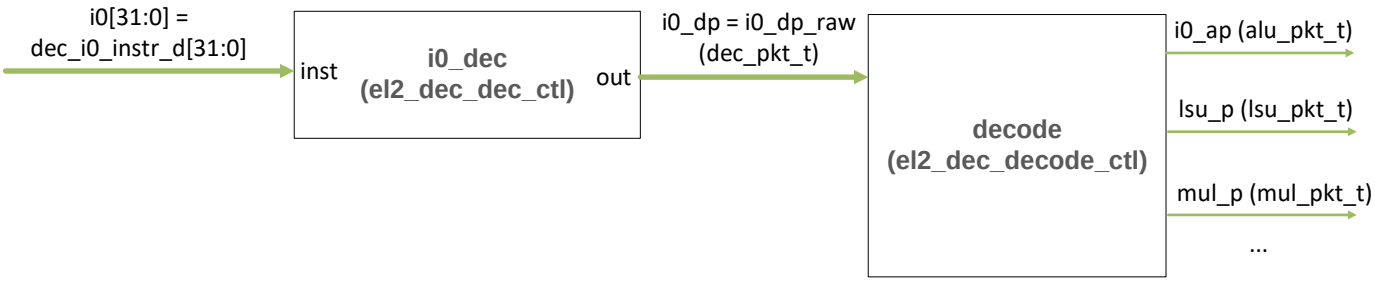


Figure 6. Control Unit

The Control Unit propagates these control signals to later pipeline stages using pipeline registers, which are placed between each pipeline stage (see Figure 5).

- **Distribute the instructions to the appropriate datapaths and provide the operands**: As shown in Figure 5, VeeR EL2 includes an Integer pipe (I), one Multiply pipe, and one Load/Store pipe (L/S). In addition, it includes a Divider which is outside the pipeline. Once the instruction is decoded, the processor sends it to one of four separate pipelines:

    - Arithmetic-Logic, some bit manipulation, and branch instructions are executed in the I pipe.
    - Loads and stores are executed in the L/S pipe.
    - Multiplication and some bit manipulation instructions are executed in the Multiply pipe.
    - Divide instructions executed in the Divider.

    As we will analyse in later labs, some situations exist when the instruction must be stalled.

    In addition to scheduling the instructions, the pipes must be provided with the corresponding operands. For that purpose, several 2:1, 3:1, and 4:1 multiplexers (see Figure 5) select among the possible operands and propagate them to the next stages using pipeline registers. These multiplexers are implemented in lines 232-265 of module **`exu`.** (Even though the multiplexers are inside the **exu** module, they operate in the D Stage). Their input operands can come from several places:

    - **Bypass Logic**: Most data dependencies are resolved in the D Stage by means of bypassing, as we will analyse in Lab 14. The inputs coming from the Bypass Logic are not labelled in the 2:1, 3:1, and 4:1 multiplexers from Figure 5 for the sake of simplicity.

    - **Immediate**: Some RISC-V instructions use Immediate Addressing Mode, in which the operand is provided directly from the instruction bits. The inputs coming from the Immediate are not shown in the 2:1, 3:1, and 4:1 multiplexers from Figure 5.

    - **Register File**: The Register File available in the VeeR EL2 processor has 2 read ports and 3 write ports. The inputs coming from the Register File are shown in the 2:1, 3:1, and 4:1 multiplexers from Figure 5 using only the names of the signals. The connections to the Register File are not shown for the sake of simplicity.

      Each read/write port has a 5-bit address (`raddr0` …, `waddr0` …), as well as a 1-bit enable signal for the write ports (`wen0` …), not shown in Figure 5. Write ports also have a 32-bit write data input (`wd0` …), and read ports have a 32-bit read data output (`rd0` …). The Register File contains 32 32-bit registers, called x0-x31, with x0 hardwired to 0.

      TASK: The Register File is implemented in module **`el2_dec_gpr_ctl`** and it is instantiated in module **`el2_dec`** (see Figure 7). Analyse both the Verilog code and the simulation of the main signals of module **`el2_dec_gpr_ctl`** (available in file *[RVfpgaBasysPath]/src/VeeRwolf/VeeR_EL2CoreComplex/dec/el2_dec_gpr_ctl.sv*), in order to understand how it works.

```
el2_dec_gpr_ctl #(.pt(pt)) arf (.*,
                // inputs
                .raddr0(dec_i0_rs1_d[4:0]),
                .raddr1(dec_i0_rs2_d[4:0]),

                .wen0(dec_i0_wen_r),           .waddr0(dec_i0_waddr_r[4:0]),          .wd0(dec_i0_wdata_r[31:0]),
                .wen1(dec_nonblock_load_wen), .waddr1(dec_nonblock_load_waddr[4:0]), .wd1(lsu_nonblock_load_data[31:0]),
                .wen2(exu_div_wren),           .waddr2(div_waddr_wb),                 .wd2(exu_div_result[31:0]),

                // outputs
                .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0])
                );
```

Figure 7. Register File instantiation inside module el2_dec

Execution Pipes

In this subsection we analyse simplified versions of the pipes available in VeeR EL2: an **Integer pipe** (**I Pipe**), a **Multiply pipe**, a **Load/Store pipe**, and a non-pipelined **Divider**.

I Pipe: The integer pipe is shown in blue in Figure 5. It includes a 1-cycle latency Arithmetic-Logic Unit (ALU) and a 1-cycle latency Bit Manipulation Unit (BMU). The ALU is capable of performing arithmetic operations such as *addition* or *subtraction*, as well as logical operations such as *and* or *or*. The BMU supports bit manipulation for Zba, Zbb, Zbc, Zbe, Zbf, Zbp, Zbr, Zbs extensions. In Lab 12 we will analyse the I pipe in further detail.

Multiply Pipe: The multiply pipe is shown in red in Figure 5. It includes a multiplier, that operates in the X Stage, and a second BMU, which operates in the D Stage and which performs bit manipulation instructions that are not performed in the I Pipe's BMU.

Load/Store (L/S) Pipe: The L/S pipe is shown in green in Figure 5. In Lab 13 we explore this pipeline path in depth. Both load and store instructions are executed through the L/S pipe. It includes several stages; during the D Stage, the Adder Unit calculates the address by adding the register base address and the immediate offset.

Divider: The divider is shown in white in Figure 5. It is a non-pipelined unit that requires several cycles to compute its result.

iii. X/M Stage and R Stage

As in the D Stage, in the last two stages of the VeeR EL2 pipeline (X/M and R stages) several things are carried out. Figure 8 illustrates a reduced version of these two stages.

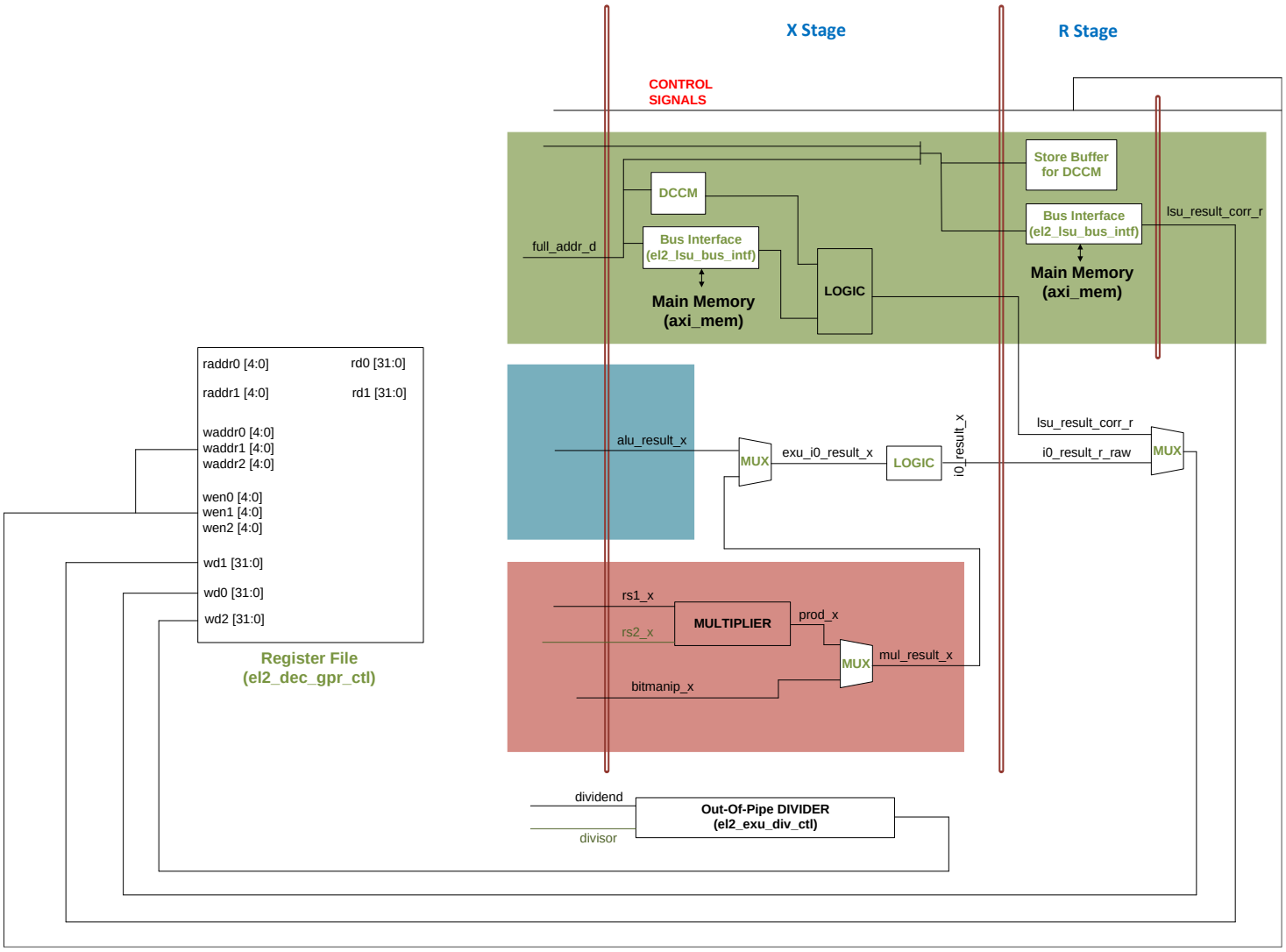


Figure 8. X/M and R Stages

Load and store instructions access the DCCM in the X Stage. In addition, they start some operations with the bus to access the Main Memory. Multiplications compute their result in the X Stage.

In the R Stage, communication with the Main Memory through the bus continues, and in fact the pipeline may need to be stalled for several cycles, depending on several factors, such as the latency of the Main Memory, the use of blocking/non-blocking loads, the existence of dependencies, etc., as we will analyse in future labs. Besides, either the result of the Arithmetic-Logic operations or the result of the Multiplier is selected and written to the Register file through write port 0 (`wd0`, `wen0`, `waddr0`).

C. Example Simulation in Verilator

In this section, we illustrate the simulation of two instructions executing in the VeeR EL2 pipeline, showing the signals introduced in the previous sections. Future labs will also use Verilator simulations to visualize the processor's internal signals and to illustrate the theoretical explanations.

We next execute the example code shown in Figure 9 (the top figure shows the source program, and the bottom figure shows the disassembly program), focusing on the `mul` and `add` instructions (highlighted in red), which are part of the infinite loop. Folder *[RVfpgaBasysPath]/Labs/Lab11/ExampleProgram* provides the Catapult project so that you can analyse, simulate, and change the program as desired. Open the project in Catapult and build it. Then, from the command line, generate the disassembly file.

The disassembly file shows the addresses and machine code. Notice that the two instructions are at addresses 0x000000F0 and 0x000000F4:

```
0x000002e0:     03de8e33                mul   t3,t4,t4
0x000002e4:     01ff0f33                add   t5,t5,t6
```

These two instructions are surrounded by several `nop` (no-operation) instructions in order to isolate them from other instructions and be able to analyse them better. The `nop` instruction does not change the state of the system. In RISC-V, `nop` is translated into `addi x0,x0,0`, which is encoded as a 32-bit machine instruction with the value of 0x00000013. In this code, we define several macros for inserting a number of `nop` instructions (from 1 to 10) in our code (for simplicity, the macros definitions are not included in Figure 9 but they can be seen in the Catapult project).

For clarity, we disable the Branch Predictor and compressed instructions, following the procedure that we explain in Section 4 below.

```
li t2, 0x008                    # Disable Branch Predictor
csrrs t1, 0x7F9, t2

li x28, 0x1
li x29, 0x2
li x30, 0x4
li x31, 0x1

REPEAT:
   mul x28, x29, x29
   add x30, x30, x31
   INSERT_NOPS_4
   add x29, x29, 1
   INSERT_NOPS_4
   beq  zero, zero, REPEAT # Repeat the loop
```

(a)

```
000002c0 <main>:
2c0:   00800393            li     t2,8
2c4:   7f93a373            csrrs  t1,0x7f9,t2
2c8:   00100e13            li     t3,1
2cc:   00200e93            li     t4,2
2d0:   00400f13            li     t5,4
2d4:   00100f93            li     t6,1

000002d8 <REPEAT>:
2d8:   00000013            nop
2dc:   00000013            nop
2e0:   03de8e33            mul    t3,t4,t4
2e4:   01ff0f33            add    t5,t5,t6
2e8:   00000013            nop
2ec:   00000013            nop
2f0:   00000013            nop
```

```
2f4:   00000013           nop
2f8:   001e8e93           add    t4,t4,1
2fc:   00000013           nop
300:   00000013           nop
304:   00000013           nop
308:   00000013           nop
30c:   fc0006e3           beqz   zero,2d8 <REPEAT>
```

(b)

Figure 9. Example program: `mul` and `add` in a loop: (a) assembly program (b) disassembled program

Figure 10 shows RVfpgaEL2-Trace waveforms of the processor signals while executing the program in Figure 9. The following signals are included in the figure to trace the instructions as they progress through the pipeline.

- `ifu_fetch_data_f` → instruction in the F Stage
- `dec_i0_instr_d` → instruction in the D Stage
- `i0_inst_x` → instruction in the X Stage
- `i0_inst_r` → instruction in the R stage

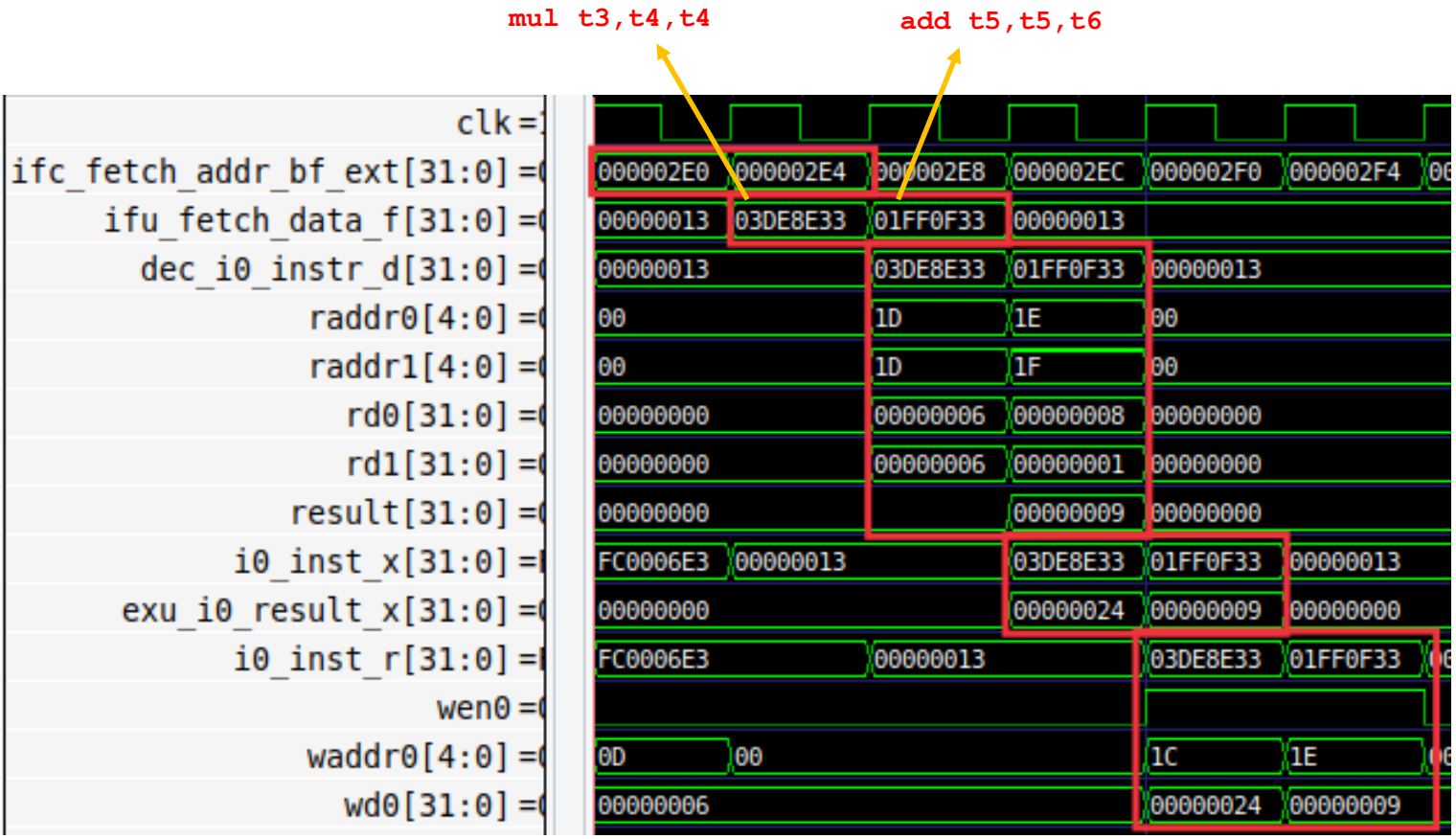


Figure 10. Simulation of the 4 stages of the VeeR EL2 core

TASK: Replicate the simulation from Figure 10 on your own computer by following the steps described in detail in the GSG.

Analyse the waveform from Figure 10 at the same time as the diagrams for the core shown in previous sections. The figures include some signals associated with each of the pipeline stages. The values highlighted in red correspond to the two instructions (`mul` and `add`) as they flow through the pipeline.

- **First cycle of Figure 10**:
  - Signal `ifc_fetch_addr_bf_ext[31:0]` (the Program Counter, which is provided to the Instruction Memory) contains the address of (i.e., *points* to) the `mul` instruction (`ifc_fetch_addr_bf_ext` = 0x000002E0).

- **Second cycle of Figure 10**:
  - Signal `ifc_fetch_addr_bf_ext[31:0]` contains the address of (i.e., *points* to) the `add` instruction (`ifc_fetch_addr_bf_ext` = 0x000002E4).
  - Instruction `mul` is at the F Stage (`ifu_fetch_data_f` = 0x03DE8E33): The Instruction Memory provides a 32-bit signal that includes the first instruction (`mul`) that we are analysing in the example.

- **Third cycle of Figure 10**:
  - Instruction `add` is in the F Stage (`ifu_fetch_data_f` = 0x01FF0F33): The

Instruction Memory provides a 32-bit signal that includes the second instruction (`add`) that we are analysing in the example.
    - Instruction `mul` is in the D Stage (`dec_i0_instr_d` = 0x03DE8E33): The `mul` instruction gets decoded, thus the pipeline generates its control signals and the Register File is read (`rd0=0x6` and `rd1=0x6`).

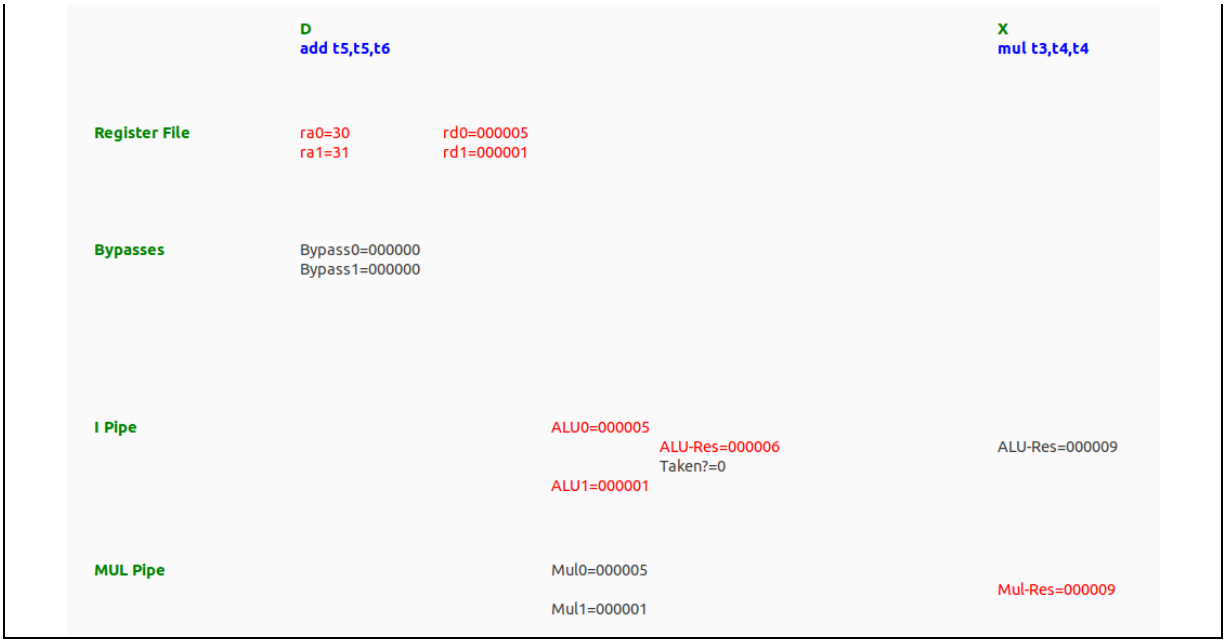
- **Fourth cycle of Figure 10**:
    - Instruction `add` is in the D Stage (`dec_i0_instr_d` = 0x01FF0F33): The `add` instruction gets decoded, thus the pipeline generates its control signals and the Register File is read (`rd0=0x8` and `rd1=0x1`). In this cycle, the ALU calculates the addition result: `result = 0x9`.
    - Instruction `mul` is in the X/M Stage (`i0_inst_x` = 0x03DE8E33): The `mul` instruction obtains its result: `exu_i0_result_x` = `0x24`.

- **Fifth cycle of Figure 10**:
    - Instruction `add` is in the X/M Stage (`i0_inst_x` = 0x01FF0F33): The `add` instruction propagates its result to the next stage: `exu_i0_result_x = 0x9`.
    - Instruction `mul` is in the R Stage (`i0_inst_r` = 0x03DE8E33): The `mul` instruction writes the result to the Register File and abandons the pipeline:
        - `wen0` = `1`
        - `waddr0` = `0x1C`
        - `wd0` = `0x24`

- **Sixth cycle of Figure 10**:
    - Instruction `add` is in the R Stage (`i0_inst_r` = 0x01FF0F33): The `add` instruction writes the result to the Register File and abandons the pipeline:
        - `wen0` = `1`
        - `waddr0` = `0x1E`
        - `wd0` = `0x9`

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 10. 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 at the fourth cycle shown in Figure 10. Note that the values are shown in decimal here (thus, the result of the multiplication is 36) whereas in Figure 10 they are shown in hexadecimal (thus, the result of the multiplication is 0x24).



3. Hardware Counters in VeeR EL2

We now show how to use performance counters to analyse processor performance. Hardware counters are a set of special-purpose registers included in most current processors to record a variety of metrics, such as the number of instructions executed, the number of cycles executed, the average clock cycles per instruction (CPI), the number of Instruction Cache hits/misses, the number of right/wrong predicted branches, etc.

In the following labs we will regularly use the Performance Counters available in VeeR EL2 for measuring and comparing various performance metrics.

A. Performance Counters in VeeR EL2

The RISC-V VeeR EL2 Programmer's Reference Manual (https://github.com/chipsalliance/Cores-VeeR-EL2/blob/main/docs/RISC-V_VeeR_EL2_PRM.pdf) describes basic hardware performance monitoring capabilities of a RISC-V processor. The following performance counters, which are also control and status registers (CSRs), must be implemented:

- *mcycle*: number of clock cycles the hart (hardware thread) has executed since some arbitrary time in the past.
- *minstret*: number of instructions the hart has retired since some arbitrary time in the past.
- *mhpmcounter3–mhpmcounter31*: 29 other event counters. The event selector CSRs, *mhpmevent3–mhpmevent31*, are WARL (write any value, read legal values) registers that control which event causes the corresponding counter to increment. The meaning of these events is defined by the platform, but event 0 is reserved to mean “no event”.

Not all counters need to be implemented. It is a legal implementation to hard-wire both the counter and its corresponding event selector to 0. Specifically, in VeeR EL2, only event counters 3 to 6 (*mhpmcounter3-mhpmcounter6*) and their corresponding event selectors (*mhpmevent3-mhpmevent6*) are functional, whereas event counters 7 to 31 (*mhpmcounter7-mhpmcounter31*) and their corresponding event selectors (*mhpmevent7-mhpmevent31*) are

hardwired to '0'. Enabling these counters is controlled by bit 0 of the *mgpmc* register (0 = disable, 1 = enable).

Chapter 7 of the VeeR EL2 Programmer's Reference Manual (https://github.com/chipsalliance/Cores-VeeR-EL2/blob/main/docs/RISC-V_VeeR_EL2_PRM.pdf) describes in detail the features and operation of the four performance counters available in VeeR EL2:

- Four standard 64-bit wide event counters
- Standard separate event selection for each counter
- Standard selective count enable/disable controllability
- Synchronized counter enable/disable controllability
- Standard cycle counter
- Standard retired instructions counter
- Support for standard SoC-based machine timer registers

Table 7-1 in that document lists the countable events in active state available in VeeR EL2, which are summarized in Table 1.

Table 1. List of Countable Events in VeeR EL2

1	Cycles clock active	18	CSR write rd==0	36	Cycles DMA ICCM transaction stalled
2	I-Cache hits	19	Ebreak	37	Exceptions taken
3	I-Cache misses	20	Ecall	38	Timer interrupts taken
4	Instrs commmited	21	Fence	39	Exteranal interrupts taken
5	Instrs commited 16-b	22	Fence.i	40	TLU flushes
6	Instrs commited 32-b	23	Mret	41	Branch error flushes
7	Instrs aligned	24	Branches commited	42	I-bus transactions – instr
8	Instrs decoded	25	Branches mispredicted	43	D-bus transactions – ld/st
9	Muls commited	26	Branches taken	44	D-bus transactions misaligned
10	Divs commited	27	Unpredictable branches	45	I-bus errors
11	Loads commited	28	Cycles fetch stalled	46	D-bus errors
12	Stores commited	29	Cycles aligner stalled	47	Cycles stalled due to I-bus busy
13	Misaligned loads	30	Cycles decode stalled	48	Cycles stalled due to D-bus busy
14	Misaligned stores	31 / 32	Cycles postsync/presync stalled	49	Cycles interrutps disabled
15	Alus commited	33	Cycles frozen	50	Cycles interrupts stalled while disabled
16	CSR read	34	Cycles SB/WB stalled	54	bitmanip committed
17	CSR read/write	35	Cycles DMA DCCM transaction stalled	55/ 56	D-bus loads/stores committed

B. Use of the Performance Counters by using the Processor Support Package (PSP)

The PSP (https://github.com/chipsalliance/riscv-fw-infrastructure) includes several functions that provide a simple approach to performance monitoring. You can analyse the following two files:

[RVfpgaBasysPath]/common/drivers/psp/psp_performance_monitor_el2.c
[RVfpgaBasysPath]/common/drivers/psp/api_inc/psp_performance_monitor_el2.h

The .c file (*psp_performance_monitor_el2.c*) implements functions that allow you to do things such as enabling/disabling the group performance monitor (`pspMachinePerfMonitorEnableAll`), pairing a counter to an event (`pspMachinePerfCounterSet`) or getting the counter value (`pspMachinePerfCounterGet`).

The .h file (*psp_performance_monitor_eh1.h*) provides names for each of the events from Table 1.

The example provided at *[RVfpgaBasysPath]\Labs\Lab11\HwCounters_Example*, illustrates the use of the four hardware counters available in VeeR EH1 to measure: *cycles*, *instructions*, and *branches commited* and *mispredicted*. The `main` function:

- Initializes the UART (`config_uart()`)
- Enables the hardware counters (`pspMachinePerfMonitorEnableAll()`)
- Assigns the events that are to be measured (*cycles*, *instructions* and *branches commited* and *mispredicted*) to each counter (`D_PSP_COUNTER0 - D_PSP_COUNTER3`)
- Reads the counters (`pspMachinePerfCounterGet(D_PSP_COUNTER0)`)
- Calls a simple assembly program (`Test_Assembly()`) and reads the counters again
- Prints the value of each counter using function `ee_printf`.

The `Test_Assembly()` function, after some register initializations, repeats a loop 1,000,000 times; the loop contains five arithmetic-logic (A-L) instructions and one conditional branch.

TASK: Execute the program on RVfpgaEL2-Basys (i.e. the physical Basys 3 board). Explain and justify the results.

TASK: Execute the program on the RVfpgaEL2-ViDBo simulator. You should obtain the results shown in Figure 11 for the four measured events. Note that the results should be the same as those obtained when the program is executed on the board.

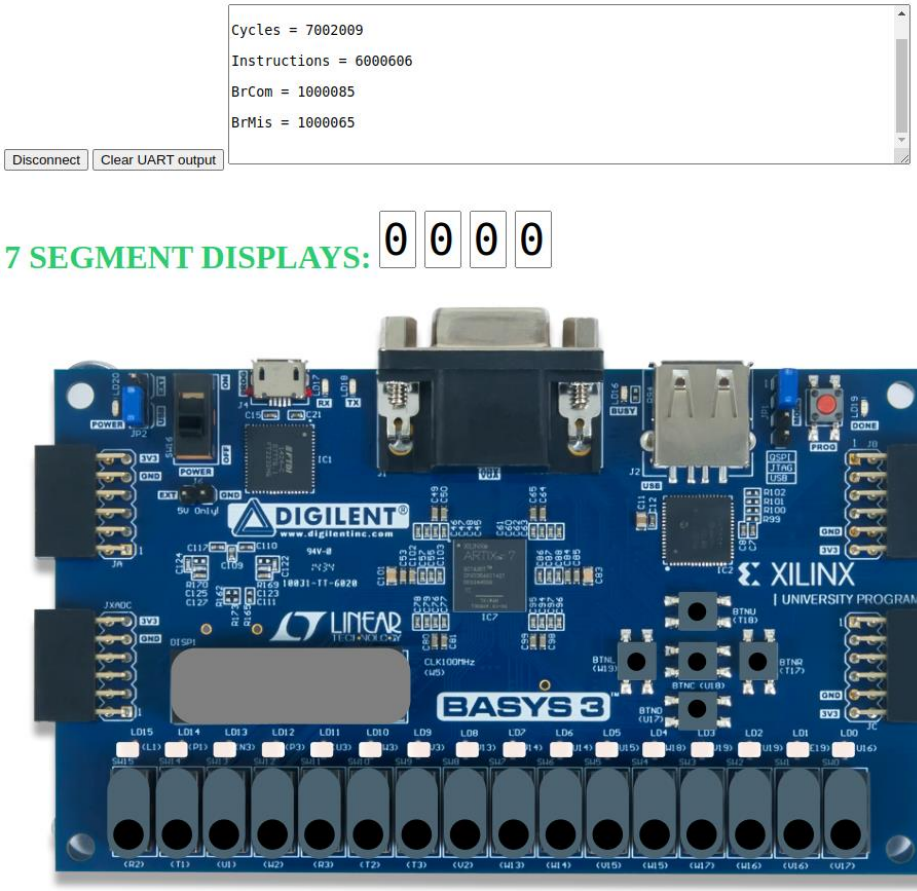


Figure 11. Execution of Test.C on RVfpgaEL2-ViDBo

TASK: Measure other events on the Hardware Counters for the same program. For this purpose, you must change in file *Test.c* the configuration of the events to be measured. Note that the different events (shown in Table 1) can be configured using the macros defined in the PSP file *psp_performance_monitor_eh1.h*. For example, if you want to measure the number of I$ misses instead of the number of branch misses, you must substitute in file *Test.c* line:

```
pspMachinePerfCounterSet(D_PSP_COUNTER3, D_BRANCHES_MISPREDICTED);
```

for line:

```
pspMachinePerfCounterSet(D_PSP_COUNTER3, D_I_CACHE_MISSES);
```

TASK: Propose other programs in the `Test_Assembly` function and check if the different events provide the expected results. You can try other instructions such as loads, stores, multiplications, divisions… as well as hazards that cause pipeline stalls.

4. Configuration of the VeeR EL2 Core

Many of the structures and features of the VeeR EL2 processor can be configured as described in this section. In addition, we show how to configure the compiler to use different RISC-V instruction sets, optimization levels, etc.

A. Enable/Disable Core Features

The VeeR EL2 Programmer's Reference Manual (Table 10-1 of the VeeR EL2 Programmer's Reference Manual available at: https://github.com/chipsalliance/Cores-VeeR-EL2/tree/main/docs) describes the `mfdc` register (*CSR* 0x7F9). This register hosts low-level core control bits to enable/disable specific features, such as pipelined execution, the Branch Predictor, etc.

The table below shows the core features that can be controlled by this register. Setting the proper bits of the register to 0 or 1, enables or disables each core feature. For example, you can include the following two assembly instructions in your assembly program for disabling the ICCM/DCCM ECC checking and the pipelined execution:

```
li t2, 0x101
csrrs t1, 0x7F9, t2
```

Table 2. Feature Disable Control Register (*mfdc*: CSR 0x7F9)

Field	Bits	Description	Access	Reset
Reserved	31:19	Reserved	R	0
dqc	18:16	DMA QoS control (see Section 2.14.3)	R/W	7
Reserved	15:13	Reserved	R	0
td	12	Trace disable: 0: enable trace 1: disable trace	R/W	0
elfd	11	External load-to-load forwarding disable: 0: enable external load-to-load forwarding 1: disable external load-to-load forwarding	R/W	0
Reserved	10:9	Reserved	R	0
cecd	8	Core ECC check disable: 0: ICCM/DCCM ECC checking enabled 1: ICCM/DCCM ECC checking disabled	R/W	0
Reserved	7	Reserved	R	0
sepd	6	Side effect pipelining disable: 0: side effect loads/stores are pipelined 1: side effect loads/stores block all subsequent bus transactions until load/store response with default value received **Note:** Reset value depends on selected bus core build argument	R/W	0 *(AHB-Lite)* 1 *(AXI4)*
Reserved	5:4	Reserved	R	0
bpd	3	Branch prediction disable: 0: enable branch prediction and return address stack 1: disable branch prediction and return address stack	R/W	0
wbcd	2	Write Buffer (WB) coalescing disable: 0: enable Write Buffer coalescing 1: disable Write Buffer coalescing	R/W	0
Reserved	1	Reserved	R	0
pd	0	Pipelining disable: 0: pipelined execution 1: single instruction execution	R/W	0

B. Configure the Core Structures

Folder *[RVfpgaBasysPath]/src/VeeRwolf/VeeR_EL2CoreComplex/include* contains several files used for the configuration of the RVfpgaEL2 System. In that folder, the initial comment of file *common_defines.vh* describes the configuration used in our default system:

// cmd: veer -unset=assert_on -set=reset_vec=0x80000000 -set=ret_stack_size=2 -set=btb_enable=0 -set=dccm_enable=0 -set=dma_buf_depth=2 -set=iccm_enable=0 -set=icache_enable=1 -set=icache_ecc=0 -set=icache_size=8 -set=icache_2banks=0 -set=icache_num_ways=2 -set=pic_size=32 --set=bitmanip_zba=0 -set=bitmanip_zbb=0 -set=bitmanip_zbc=0 -set=bitmanip_zbe=0 -set=bitmanip_zbf=0 -set=bitmanip_zbp=0 -set=bitmanip_zbr=0 -set=bitmanip_zbs=0 -set=fast_interrupt_redirect=0

You can easily change the configuration provided by default for the RVfpgaEL2 System, by means of the ***veer.config*** script provided with the VeeR EL2 package. In RVfpgaEL2 you can find this script at: *[RVfpgaBasysPath]/src/VeeRwolf/VeeR_EL2CoreComplex/include*. In order to create a new configuration, you must follow the next steps.

1. Make a copy of the src folder and name it "src_Default". This way you'll be able to return to the original configuration easily.

2. Go into folder *[RVfpgaBasysPath]/src/VeeRwolf/VeeR_EL2CoreComplex/include/* for creating the files for a new configuration of the SoC.

3. If necessary, give execution permissions to the script: `chmod +x veer.config`

4. If necessary, install json-perl library: `sudo apt-get install -y libjson-perl`

5. Create the new configuration files, which is the same as the default one except for the I$, which is disabled now, by running the ***veer.config*** script as follows:

```
./veer.config -unset=assert_on -set=reset_vec=0x80000000
-set=ret_stack_size=2 -set=btb_enable=0 -
set=dccm_enable=0 -set=dma_buf_depth=2 -set=iccm_enable=0
-set=icache_enable=0 -set=icache_ecc=0 -set=icache_size=8
-set=icache_2banks=0 -set=icache_num_ways=2 -
set=pic_size=32 --set=bitmanip_zba=0 -set=bitmanip_zbb=0
-set=bitmanip_zbc=0 -set=bitmanip_zbe=0 -
set=bitmanip_zbf=0 -set=bitmanip_zbp=0 -
set=bitmanip_zbr=0 -set=bitmanip_zbs=0 -
set=fast_interrupt_redirect=0
```

   You can view all configuration options by simply running: `./veer.config -h`

6. Replace the old configuration files with the new ones:

```
cp snapshots/default/common_defines.vh .
cp snapshots/default/el2_pdef.vh .
cp snapshots/default/el2_param.vh .
```

7. Remove the snapshots directory: `rm -rf snapshots`

8. Rebuild the simulators and the bitstream for the RVfpgaEL2 System to have the new configuration.

In Lab 19, you will use this configuration for comparing the two different I$ configurations (2-way vs. 4-way).

C. Modify the compiler flags

We can easily modify the compiler options established for the Basys 3 Platform in file *[RVfpgaBasysPath]/common/Common.cmake*.

Compressed Instructions: In the examples included in this chapter (and also in some of the examples from previous chapters) we disable the use of compressed instructions. You can easily use compressed instructions by making the following change in file *Common.cmake*, selecting the “PLATFORM MATCHES” that corresponds to the board that you use:

```
set(ARCH_FLAGS -march=rv32im_zicsr_zifencei -mabi=ilp32)
```

```
set(ARCH_FLAGS -march=rv32imc_zicsr_zifencei -mabi=ilp32)
```

Compiler optimizations: We can also modify the optimization level used by the compiler. You can easily use a -O2 optimization level by making the following change in file *Common.cmake*, selecting the “PLATFORM MATCHES” that corresponds to the board that you use:

```
set(ARCH_FLAGS -march=rv32im_zicsr_zifencei -mabi=ilp32)
```

```
set(ARCH_FLAGS -march=rv32im_zicsr_zifencei -mabi=ilp32 -O2)
```