



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 11

SweRV EH1 Configuration, Organization, and Performance Monitoring

1. INTRODUCTION

In the first 10 RVfpga labs (Labs 1–10) we introduced the RISC-V architecture and how to communicate the SweRV EH1 Core using various peripherals. In the next ten labs (Labs 11–20), we will dive down to the microarchitectural level and analyse how the SweRV EH1 processor operates internally and how the cache/memory hierarchy works.

SIGASI STUDIO: In these labs we are going to deal with an extensive Verilog project: the SweRV EH1 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, there are more suitable and specific alternatives, such as **Sigasi Studio** (<https://www.sigasi.com/>), which we highly recommend. A supplementary document, **RVfpga_SweRVref.docx**, shows, among other things, how to install and use Sigasi Studio (Section 1 of the RVfpga_SweRVref document).

As explained in the RVfpga Getting Started Guide (GSG), SweRV EH1 is a 32-bit 2-way superscalar 9-stage pipelined in-order processor. Figure 1 shows a high-level view of the SweRV EH1 microarchitecture. SweRV EH1 supports RISC-V's integer (I), compressed instruction (C), and integer multiplication and division (M) extensions. Its impressively high performance per MHz (4.9 CM/MHz) is accomplished thanks to the inclusion of several microarchitectural techniques, from the most basic and common ones, such as pipelining and an instruction cache, to other more specific and advanced techniques, such as superscalar execution, non-blocking loads and divisions, two secondary ALUs that allow Arithmetic-Logic instructions being repeated when necessary due to data hazards (see Lab 15 for details), unaligned loads and stores, scratch pad memories for both instruction and data, and advanced branch prediction. All these techniques will be extensively analysed in these labs.

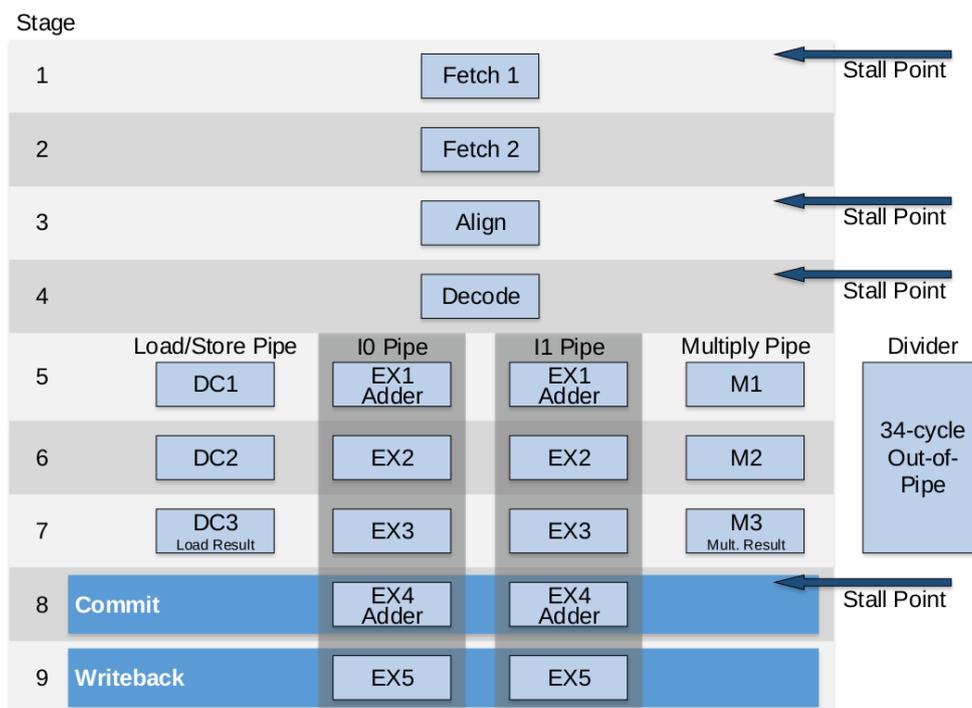


Figure 1. SweRV EH1 core microarchitecture

(figure from https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_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.

Most of the labs are divided into two parts: a fundamentals section followed by an advanced section. Moreover, given the high complexity of some parts of a real processor such as the SweRV EH1, some details are moved to a given lab's appendix. This way, users can choose to only complete the fundamental section, to complete both the fundamental and advanced sections, or even delve into the appendices and understand the more complex parts of the processor.

Labs 11-20 begin with a theoretical explanation of the concepts and then illustrate the concepts using figures and a Verilator simulation 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, 14, 15 and first part of 16 (branch instructions)
- **Memory:** Labs 11, 13 and 19
- **Advanced microarchitecture and memory hierarchy:** Labs 17, 18, 20 and second part of 16 (branch predictor)

In this lab (Lab 11), we begin to analyse the SweRV EH1 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.

The supplementary document (**RVfpga_SweRVref.docx**) describes:

- **Section 1:** Use of Sigasi Studio.
- **Section 2:** Configuration of the SweRV EH1 processor.
- **Section 3:** RVfpga System hierarchy of modules and their most relevant signals
- **Section 4:** Structures and types for grouping control bits
- **Section 5:** RISC-V compressed instructions
- **Section 6:** Real benchmarks

After this initial approach, we extend this analysis in Labs 12-20 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 focusing on the 3-cycle pipelined multiplication instruction and on a specific case related with non-blocking loads. The lab also analyses the 34-cycle non-pipelined division instruction in an appendix.
- **Lab 15** analyses **data hazards** by describing the processor's bypass paths.

- **Lab 16** describes **control hazards**, branch instructions, and the branch predictor, for which we will focus on the Fetch 1 and Fetch 2 stages of the SweRV EH1 processor.
- While in the previous labs only one way of the processor is used in most cases, **Lab 17** describes 2-way superscalar processors, such as SweRV EH1.
- **Lab 18** is a practical lab where you will add new instructions and hardware counters to the SweRV EH1 core.
- **Labs 19** and **20** focus on the various low-latency memories available in the processor: the instruction cache (I\$) and the closely-coupled instruction and data memories (ICCM and DCCM).

2. AN INITIAL APPROXIMATION TO THE SweRV EH1 MICROARCHITECTURE

The processor described in DDCARV has 5 pipeline stages, which are called the *Fetch*, *Decode*, *Execute*, *Memory* and *Writeback* stages. In contrast, the SweRV EH1 pipeline is divided into 9 stages (Figure 1): the *Fetch1*, *Fetch2*, *Align*, *Decode*, *EX1/DC1/M1*, *EX2/DC2/M2*, *EX3/DC3/M3*, *Commit*, and *Writeback* stages. When comparing the two processors, some stages are equivalent, such as the Decode and Writeback stages. But SweRV EH1 adds parallel paths (load/store vs. integer vs. multiply pipes), splits some stages into multiple stages (*Fetch* is 2 stages and *Execute* is 3 stages), and adds stages (the *Commit* and *Align* stages).

The remainder of this section describes the Verilog RTL organization and details of each pipeline stage. Section A describes the hierarchy of SweRV EH1's Verilog modules. Sections B and C discuss the microarchitecture of SweRV EH1 stage-by-stage. Finally, Section D provides a practical example of the theoretical explanations given in Sections B and C.

CONFIGURATION OF THE SWERV EH1 PROCESSOR: Many of the structures and features of the SweRV EH1 processor can be configured or enabled/disabled. The supplementary document, RVfpga_SweRVref.docx, explains these different options in Section 2, which you will frequently use in Labs 12-20.

A. Hierarchy of SweRV EH1'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 SweRV EH1 processor. This figure expands Figure 29 of the GSG, where we showed the hierarchy of the Verilog modules that make up the RVfpga System. These modules are located in files with the same name in: `[RVfpgaPath]/RVfpga/src/SweRVofSoC/SweRVEh1CoreComplex` directory.

The **mem** module instantiates the structures that make up the memory hierarchy of the SweRV EH1 processor: ICCM, DCCM and I\$. The **swerv** module is the overall CPU; it instantiates the modules that make up the the SweRV EH1 processor: Instruction Fetch Unit (**ifu**), Decode Unit (**dec**), Execution Unit (**exu**), Load/Store Unit (**Isu**)...

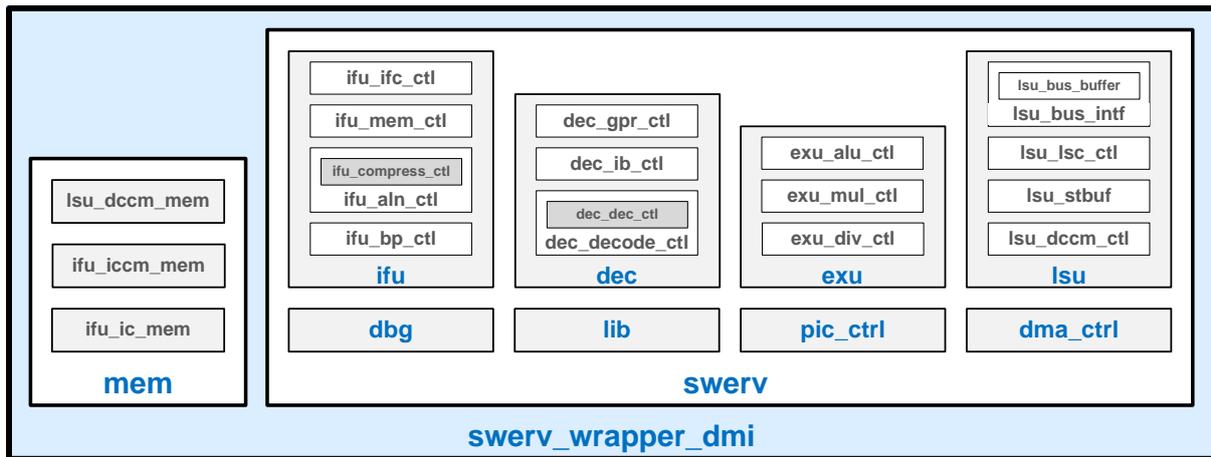


Figure 2. Swerv EH1 main modules

MAIN SIGNALS OF THE Swerv EH1 CORE: The supplementary document, RVfpga_Swervref.docx, provides, in Section 3, the main input/output signals to/from the modules of the Swerv EH1 processor. You may use it as a reference while completing Labs 11-20.

B. Fetch (FC1 and FC2) and Align stages

In this section we analyse the first three stages of the pipeline: the two Fetch stages (FC1 and FC2) and the Align stage of the Swerv EH1 pipeline. Figure 3 illustrates a very simplified view of these stages.

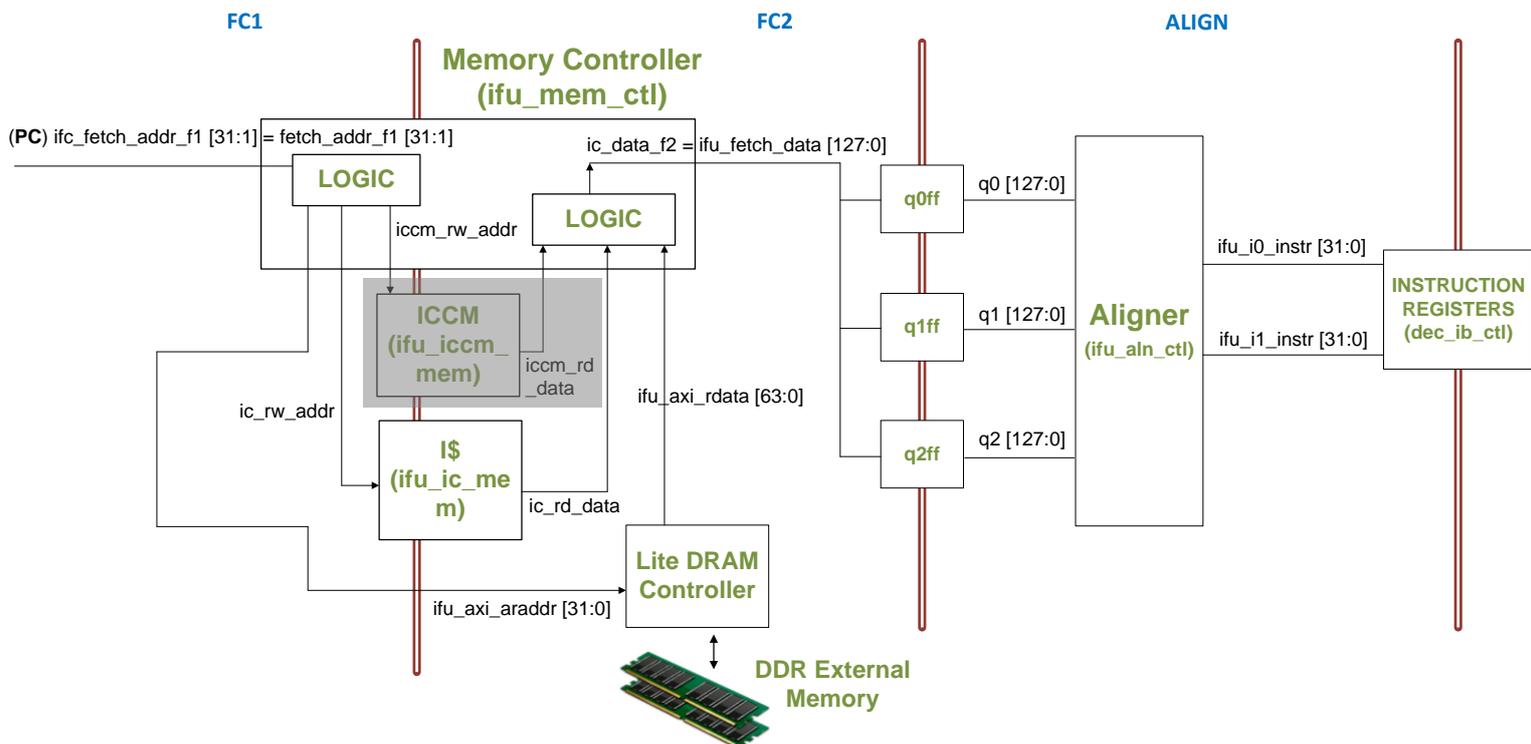


Figure 3. Simplified view of the FC1, FC2 and Align stages. Note that the ICCM is shadowed, indicating that it is disabled in our RVfpga System.

i. Fetch Stages (FC1 and FC2)

In each cycle, the Fetch stage is responsible for **reading the instructions from the Instruction Memory**. In our configuration, the Instruction Memory is made up by an ICCM (implemented in module `ifu_iccm_mem`), an Instruction Cache (I\$, implemented in module `ifu_ic_mem`) and the DDR External Memory. Both the I\$ and the ICCM are controlled from a unified memory controller (`ifu_mem_ctl`), whereas the External Memory is controlled from the Lite DRAM Controller. In our default RVfpga System the ICCM is disabled, but you can easily include it as explained in Lab 20.

As shown in Figure 3, the instruction address (called the fetch address, `ifu_fetch_addr_f1`) is computed in the first Fetch stage (**FC1**) as will be discussed further in Lab 16. This address is provided to the Instruction Memory Controller (implemented in module `ifu_mem_ctl`): `fetch_addr_f1 = ifu_fetch_addr_f1`.

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, “f1” indicates the FC1 stage.

The instruction is read during the second Fetch stage (**FC2**) from either Main Memory (i.e., DDR External Memory) or the ICCM. 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 External Memory through the AXI bus, which takes several cycles. If the instruction address is within the ICCM address range, the instruction is provided with low latency from the ICCM through a multiplexer implemented inside the `ifu_ic_mem` module.

The RVfpga System’s Instruction Memory is configured as follows (this configuration can be modified, as we will show in future labs):

- 16 KiB Instruction Cache
- 512 KiB ICCM (disabled): address range: 0xEE000000 – 0xEE07FFFF
- 128 MiB External Memory: address range: 0x00000000 – 0x07FFFFFF

If the program has no stalls (i.e. no control, data, or structural hazards, no I\$ misses, etc.), four 32-bit instructions (128 bits total) are read every two cycles: see signal `ifu_fetch_data[127:0]`. This is enough to keep the 2-way superscalar pipeline working at its maximum throughput of 2 instructions per cycle. Three buffers (`q0ff`, `q1ff` and `q2ff`) can store up to three of these 128-bit bundles.

ii. Align Stage

The Align stage, which follows the two Fetch stages (see Figure 3), is implemented in module `ifu_aln_ctl`. The Align stage is responsible for performing two main tasks:

- **Provide two 32-bit instructions per cycle to the Decode stage:** The Align stage extracts two instructions per cycle from the 128-bit bundles provided by the Instruction Memory and which are temporarily stored in buffers `q0ff`, `q1ff` and `q2ff`. These two instructions are assigned to each of the two ways available in SweRV EH1 through

signals `ifu_i0_instr[31:0]` (Way 0) and `ifu_i1_instr[31:0]` (Way 1), and are then stored in the two Instruction Registers (IR) implemented in module `dec_ib_ctl`.

- **Uncompress instructions:** 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 Align stage uncompresses these 16-bit instructions, when necessary, before passing them to the Decode stage, which only decodes 32-bit instructions. This is performed by the `ifu_compress_ctl` module, which is instantiated inside the aligner (module `ifu_aln_ctl`).

COMPRESSED INSTRUCTIONS: The supplementary document, `RVfpga_SweRVref.docx`, explains, in Section 5, the execution of compressed instructions in SweRV EH1 and proposes a few new tasks.

C. Decode, Execution, Commit and Writeback Stages

In this section we analyse the Decode, Execution, Commit and Writeback stages of the SweRV EH1 pipeline. Figure 4 illustrates a simplified view of these stages, which we will extend in future labs.

i. Decode Stage

The Verilog modules for this stage are in folder `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec`. In each cycle, the Decode stage is responsible for two main tasks:

- **Decode the instructions and generate the control signals:** The control signals are organized in several types, as defined in file `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv`. Each structure/type is related to a given unit: ALU (`alu_pkt_t`), Multiply Unit (`mul_pkt_t`), Divide Unit (`div_pkt_t`), Registers (`reg_pkt_t`), etc.

STRUCTURES USED FOR THE CONTROL BITS: The supplementary document, `RVfpga_SweRVref.docx`, extends, in Section 4, the description of the main structures/types used in the SweRV EH1 processor for grouping the control signals and proposes a few new tasks. In later labs, we will focus on the types related to the discussed unit.

The Control Unit, implemented in module `dec_decode_ctl`, receives the two 32-bit instructions fetched, uncompressed, aligned and assigned to each way in the previous stages (signals `dec_i0_instr_d[31:0]` for Way 0 and `dec_i1_instr_d[31:0]` for Way 1) and decodes them, generating the control signals for each instruction. Figure 5 shows a high-level view of the Control Unit (module `dec_decode_ctl`), which generates control signals in two stages: The first two modules (`i0_dec` and `i1_dec`) use the instructions (`i0` and `i1`) to produce overall control signals (`i0_dp` and `i1_dp`, both of them of type `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`, `i1_ap`, `lsu_p`, `mul_p`, etc.).

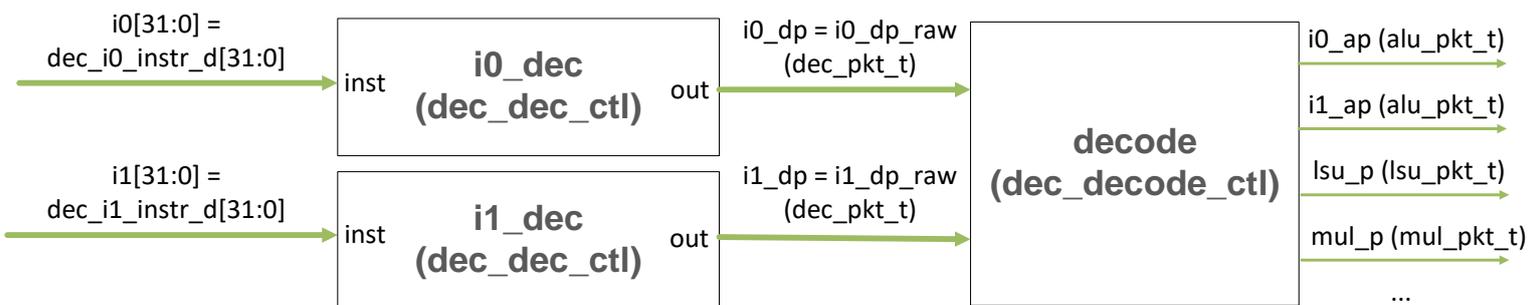


Figure 5. Control Unit

The Control Unit propagates these control signals to later pipeline stages using pipeline registers (labelled **Control Pipeline Registers** in Figure 4), which are placed between each pipeline stage.

- **Distribute the instructions to the appropriate pipes and provide the operands:** As shown in Figure 4, SweRV EH1 includes two Integer pipes (I0 and I1), one Multiply pipe, and one Load/Store pipe (L/S). In addition, it includes a 34-cycle Divider which is outside of the pipeline. Once each instruction is decoded, the processor sends it to one of four separate pipelines:

- Arithmetic-Logic and branch instructions are executed in the I0/I1 pipe.
- Loads and stores are executed in the L/S pipe.
- Multiplication instructions are executed through the Multiply pipe.
- Divide instructions executed through to the Divider pipe.

Given that up to two instructions are decoded every cycle, one in Way 0 and the other one in Way 1, both are scheduled for execution whenever possible. For example, some possible combinations are:

- Two independent Arithmetic-Logic instructions are sent to the I0 and I1 pipes.
- An Arithmetic-Logic instruction and a multiply (`mul`) instruction are sent to the I0 (or I1) and Multiply pipes, respectively.
- A memory (load or store) instruction executes in the L/S pipe, and a multiply instruction executes in the Multiply pipe.

Unfortunately, some situations exist (such as hazards, which we analyse in Labs 14-16) when one or the two instructions must be stalled. These situations are also determined at the Decode stage. For example:

- If two `mul` instructions are decoded in the same cycle, the structural hazard is resolved by delaying the second `mul` instruction in one cycle (this will be analysed in detail in Lab 14).
- If two dependent Arithmetic-Logic (A-L) instructions are decoded in the same cycle, the RAW data hazard is resolved by delaying the second A-L instruction by one cycle (this will be further analysed in Lab 15).

In addition to scheduling the instructions, the pipes must be provided with the corresponding operands. For that purpose, several 3:1 and 4:1 multiplexers (see Figure 4) select among the possible operands and propagate them to the next stages using pipeline registers. These multiplexers are implemented in lines 279-328 of module `exu` (even though the multiplexers are inside the `exu` module, they operate in the Decode stage). Their input operands can come from several places:

- **Bypass Logic:** Most data dependencies are resolved at the Decode stage by means of bypassing, as we will analyse in Lab 15. The inputs coming from the Bypass Logic are not labelled in the 3:1 and 4:1 multiplexers from Figure 4 for the sake of simplicity – only blank wires are shown.
- **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 3:1 and 4:1 multiplexers from Figure 4 – only a blank input wire is shown).
- **Register File:** The Register File available in SweRV EH1 processor (Figure 6) has 4 read ports and 3 write ports (note that the third write port is ignored in the Register File included in Figure 4 as it is only used for specific situations that we will analyse in future labs). These read/write ports allow the execution of two instructions per cycle. The inputs coming from the Register File are shown in the 3:1 and 4:1 multiplexers from Figure 4 using only the names of the signals. The connections with the Register File are not shown for the sake of simplicity.

Each read/write port has a 5-bit address (`raddr0 ... raddr3, waddr0 ... waddr2`), as well as a 1-bit enable signal (`rden0 ... rden3, wen0 ... wen2`) not

shown in Figure 4. Write ports also have a 32-bit write data input (`wd0 ... wd2`), and read ports have a 32-bit read data output (`rd0 ... rd3`). The Register File contains 32 32-bit registers, called `x0-x31`, with `x0` hardwired to 0.

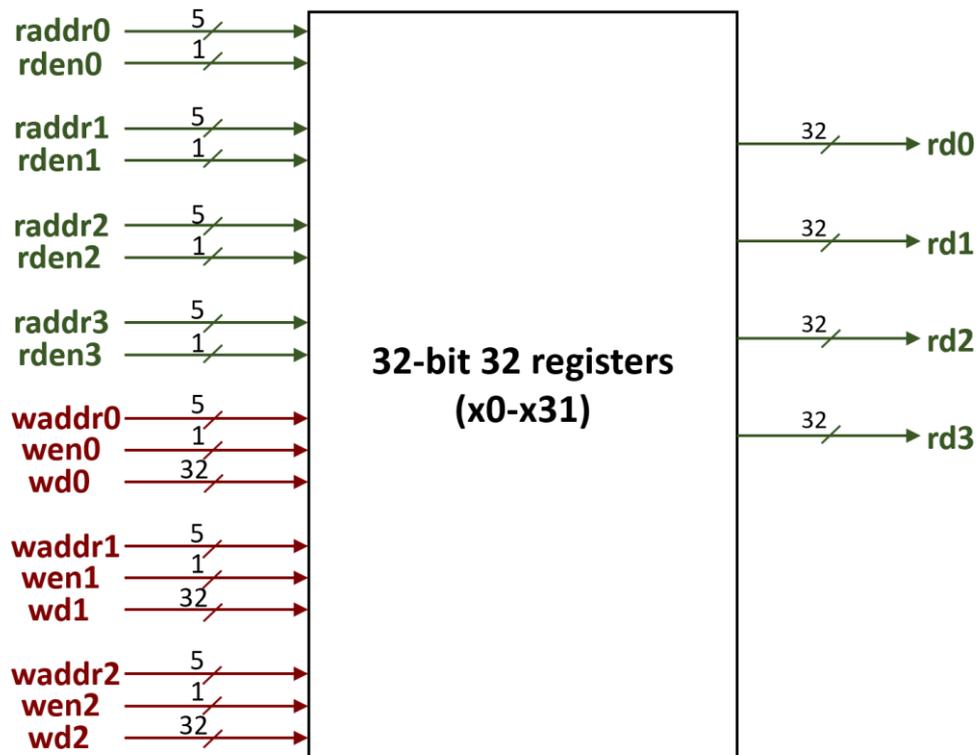


Figure 6. Register File available in SweRV EH1

TASK: The Register File is implemented in module `dec_gpr_ctl` and it is instantiated in module `dec` (see Figure 7). Analyse both the Verilog code and the simulation of the main signals of module `dec_gpr_ctl` (available in file `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_gpr_ctl.sv`), in order to understand how it works. Note that the SweRV EH1 processor allows the inclusion of several Register Files, but the configuration used in the RVfpga System only uses one Register File (see line 402 of file `dec.sv`: `localparam GPR_BANKS = 1;`).

```

525     dec_gpr_ctl #(.GPR_BANKS(GPR_BANKS),
526                 .GPR_BANKS_LOG2(GPR_BANKS_LOG2)) arf (.*,
527                 // inputs
528                 .raddr0(dec_i0_rs1_d[4:0]), .rden0(dec_i0_rs1_en_d),
529                 .raddr1(dec_i0_rs2_d[4:0]), .rden1(dec_i0_rs2_en_d),
530                 .raddr2(dec_i1_rs1_d[4:0]), .rden2(dec_i1_rs1_en_d),
531                 .raddr3(dec_i1_rs2_d[4:0]), .rden3(dec_i1_rs2_en_d),
532
533                 .waddr0(dec_i0_waddr_wb[4:0]), .wen0(dec_i0_wen_wb), .wd0(dec_i0_wdata_wb[31:0]),
534                 .waddr1(dec_i1_waddr_wb[4:0]), .wen1(dec_i1_wen_wb), .wd1(dec_i1_wdata_wb[31:0]),
535                 .waddr2(dec_nonblock_load_waddr[4:0]), .wen2(dec_nonblock_load_wen), .wd2(lsu_nonblock_load_data[31:0]),
536
537                 // outputs
538                 .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0]),
539                 .rd2(gpr_i1_rs1_d[31:0]), .rd3(gpr_i1_rs2_d[31:0])
540                 );

```

Figure 7. Register File instantiation inside module dec

ii. Execution Stages

In this subsection we analyse simplified versions of the pipes available in SweRV EH1: two **Integer Pipes (I0 Pipe and I1 Pipe)**, a **Multiply Pipe**, a **Load/Store Pipe**, and a non-pipelined 34-cycle **Divider**.

I0/I1 Pipes: The two integer pipes are shown in blue in Figure 4. They are divided in three stages called EX1, EX2 and EX3. Each of these two pipes includes a 1-cycle latency ALU in **EX1**, which is capable of performing arithmetic operations such as *addition* or *subtraction*, as well as logical operations such as *and* or *or*. Stages **EX2** and **EX3** perform few tasks but they are necessary to synchronize the A-L instructions with the other instruction types (such as loads, stores, multiplications, etc.) that require three cycles for computing their operations. In Lab 12 we will analyse the I0/I1 pipes in further detail.

Multiply Pipe: The multiply pipe is shown in red in Figure 4. It is divided into three stages: **M1**, **M2**, and **M3**. This pipe includes a 3-cycle multiplier capable of performing integer multiplication. In Lab 14 we will analyse the Multiply pipe in more detail.

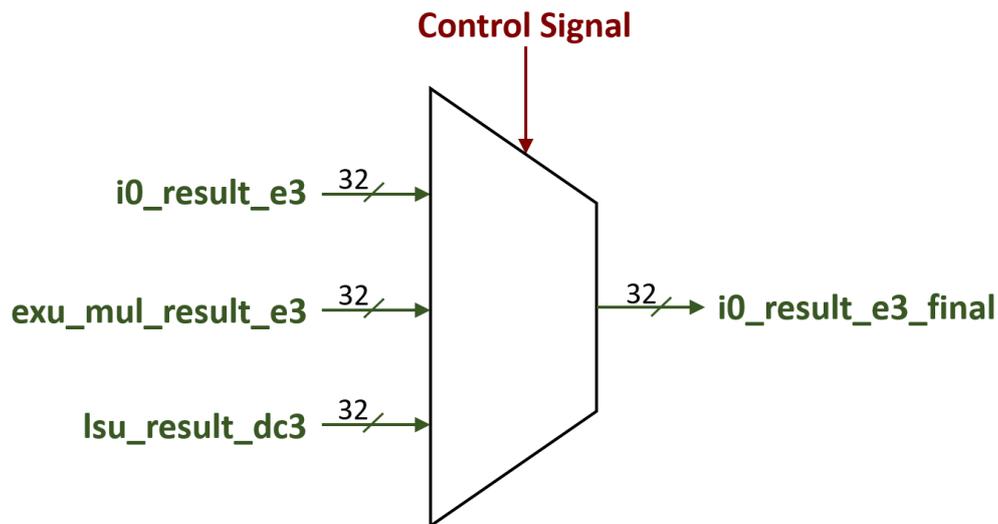
Load/Store (L/S) Pipe: The L/S pipe is shown in green in Figure 4. In Lab 13 we explore this pipeline path in depth. Both load and store instructions are executed through the L/S pipe. It includes 3 stages:

- **DC1:** In the first stage, the Adder Unit calculates the address by adding the register base address and the immediate offset.
- **DC2:** In the second stage, load instructions read memory using the address computed in DC1. If the address maps to the DCCM, the access latency is only 1 cycle and the pipeline continues with no stalls. However, if the access is mapped to the Main Memory, the pipeline may need to be stalled for several cycles, depending on the use of blocking/non-blocking loads and the existence of dependencies, as we will analyse in future labs.
- **DC3:** In the third stage, data is aligned and merged (for example, if a previous store to the same address is still executing, the data from that store may need to be forwarded to the load). In this stage, store instructions start writing memory, which will continue for several cycles. If the write is mapped to the DCCM, both the data and address are buffered in the Store Buffer before being sent to the DCCM, as we analyse in Lab 13; if the write is mapped to Main Memory, both the data and the address are sent to the External Memory through the AXI bus (the Lite DRAM controller manages the accesses to this memory).

Divider: The divider is shown in white in Figure 4. It is a non-pipelined unit that requires up to 34 cycles to compute its result. Lab 14 analyses the Divider in more detail.

Two 3:1 multiplexers: At the end of the third execution stage (EX3/DC3/M3), as illustrated in Figure 4, the result of the instructions is selected from the proper pipe (I0/I1, MUL, or L/S) using two 3:1 multiplexers, one for each way. These multiplexers are located in the **dec_decode_ctl** module. The upper multiplexer, associated with Way 0, is shown in Figure 8. The three inputs to this multiplexer are:

1. **I0 pipe result:** `i0_result_e3`. Lab 12 analyses this path.
2. **L/S pipe result:** `lsu_result_dc3`. Lab 13 analyses this path.
3. **Multiply pipe result:** `exu_mul_result_e3`. Lab 14 analyses this path.



```
2268 assign i0_result_e3_final[31:0] = (e3d.i0v & e3d.i0load) ? lsu_result_dc3[31:0] : (e3d.i0v & e3d.i0mul) ? exu_mul_result_e3[31:0] : i0_result_e3[31:0];
```

Figure 8. 3:1 multiplexer to select EX3 result: diagram and Verilog

TASK: Analyse the control bits of the multiplexer from Figure 8. Note that the control bits are in signal `e3d`, which was pipelined from signal `dd`, which was generated in the Decode stage by the Control Unit (see `RVfpga_SweRVref.docx` for descriptions of the control bits).

iii. Commit Stage

In the Commit stage, two 3:1 multiplexers, one per way, select the result to write back to the register file (see Figure 4). The upper multiplexer, associated with Way 0, is shown in Figure 9. It has three inputs:

1. **EX3 result:** `i0_result_e4`. (The output from the 3:1 multiplexer of EX3).
2. **Corrected read data:** `lsu_result_corr_dc4`. Lab 13 analyses this path.
3. **Secondary ALU result:** `exu_i0_result_e4`. These ALUs are not shown in Figure 4 for the sake of simplicity. As we mentioned above, they allow arithmetic-logic instructions being repeated when necessary due to data hazards (see Lab 15 for details).

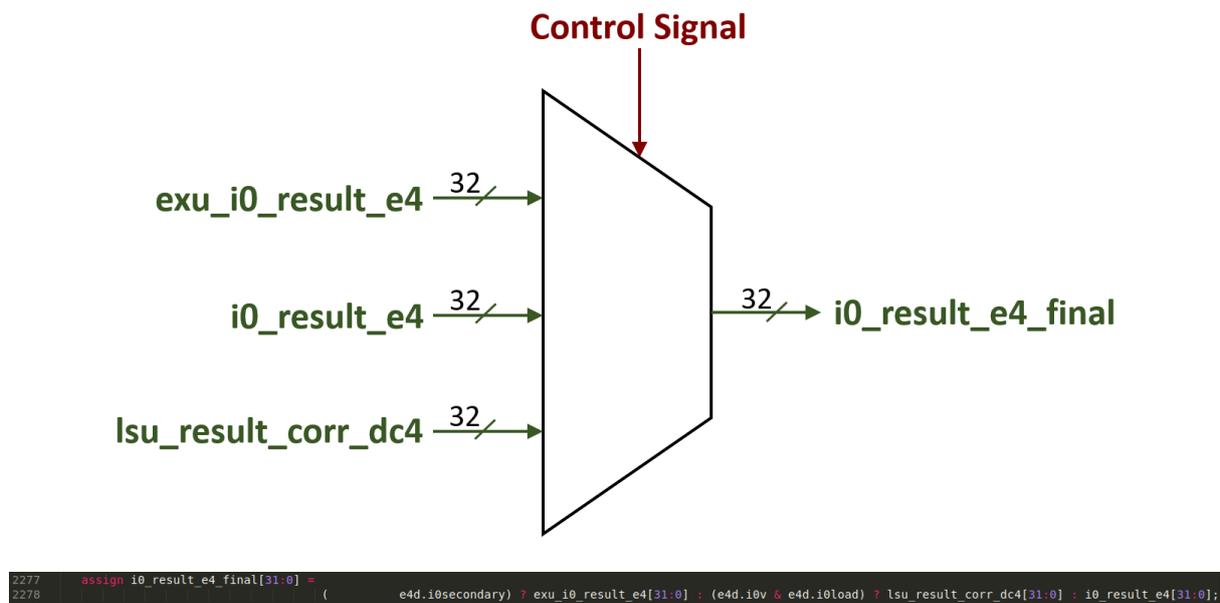


Figure 9. 3:1 multiplexer to select final result: diagram and Verilog

TASK: Analyse the control bits of the multiplexer from Figure 9, which you can find in module `dec_decode_ctl`.

iv. Writeback Stage

The final stage, the Writeback stage, writes the results to the Register File using the two first write ports (0 and 1) illustrated in Figure 6 (in lab 14 we will see when the third write port – 2 – is used). Not all cycles will write two results: some instructions do not write a register (i.e., branch instructions, store instructions...), and not all cycles execute two instructions. The register identifiers and the enable signals were generated in the Decode stage and are provided by the Control Pipeline Registers.

D. Example Simulation in Verilator

In this section, we illustrate the simulation of two instructions executing in parallel in the SweRV EH1 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 10, focusing on the `mul` and `add` instructions (highlighted in red), which are part of the infinite loop. Folder `[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram` provides the PlatformIO project so that you can analyse, simulate and change the program as desired. Open the project in PlatformIO and build it (remember from the Getting Started Guide that you can build the project by clicking on button , located at the bottom part of VSCode). The disassembly file (available at `[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram/.pio/build/swervolf_nexys/firmware.dis`)

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

```
0x000000f0:    03de8e33          mul   t3,t4,t4
0x000000f4:    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 10 but they can be seen in the PlatformIO project).

For clarity, we disable the Branch Predictor and compressed instructions, following the procedure that we explain in Section 2 of the RVfpga_SweRVref document.

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

REPEAT:
    mul x28, x29, x29    # x28 = 2 * 2 = 4 (later iterations: 3*3=9, 4*4=16, ...)
    add x30, x30, x31    # x30 = 4 + 1 = 5 (later iterations: 5+1=6, 6+1=7, ...)
    INSERT_NOPS_10
    add x29, x29, 1      # x29 = x29 + 1
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repeat the loop
```

Figure 10. Example program containing a `mul` and `add` instructions within a loop

Figure 11 and 12 show Verilator waveforms of the processor signals while executing the program from Figure 10. Figure 11 shows the signals from the first three pipeline stages (FC1, FC2, and Align – see Figure 3). Figure 12 shows the signals from the remaining stages (see Figure 4). We split the results in two figures for the sake of consistency with Figure 3 and Figure 4, but remember that these two instructions go from the Align stage (on the right of Figure 11) to the Decode stage (on the left of Figure 12).

The following signals are included in the figures to trace the instructions as they progress through the pipeline (`ifu` for the instructions at the Align stage, `dec` for the instructions at the Decode stage, `eX` for the instructions at the X (X = first, second, third) Execution stage, `e4` for the instructions at the Commit stage, and `wb` for the instructions at the Writeback stage) and to know to which way they are assigned (`i0` for Way 0 and `i1` for Way 1).

- `ifu_i0_instr` and `ifu_i1_instr` → instructions in the Align stage
- `dec_i0_instr_d` and `dec_i1_instr_d` → instructions in the Decode stage
- `i0_inst_e1` and `i1_inst_e1` → instructions in the EX1 stage
- `i0_inst_e2` and `i1_inst_e2` → instructions in the EX2 stage
- `i0_inst_e3` and `i1_inst_e3` → instructions in the EX3 stage
- `i0_inst_e4` and `i1_inst_e4` → instructions in the Commit stage
- `i0_inst_wb` and `i1_inst_wb` → instructions in the Writeback stage

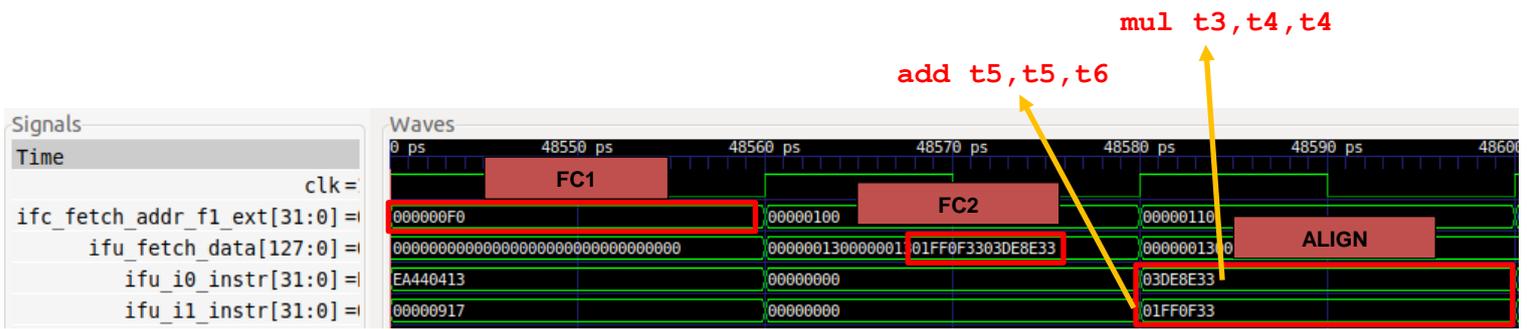


Figure 11. Simulation of the first three pipeline stages: FC1, FC2 and Align

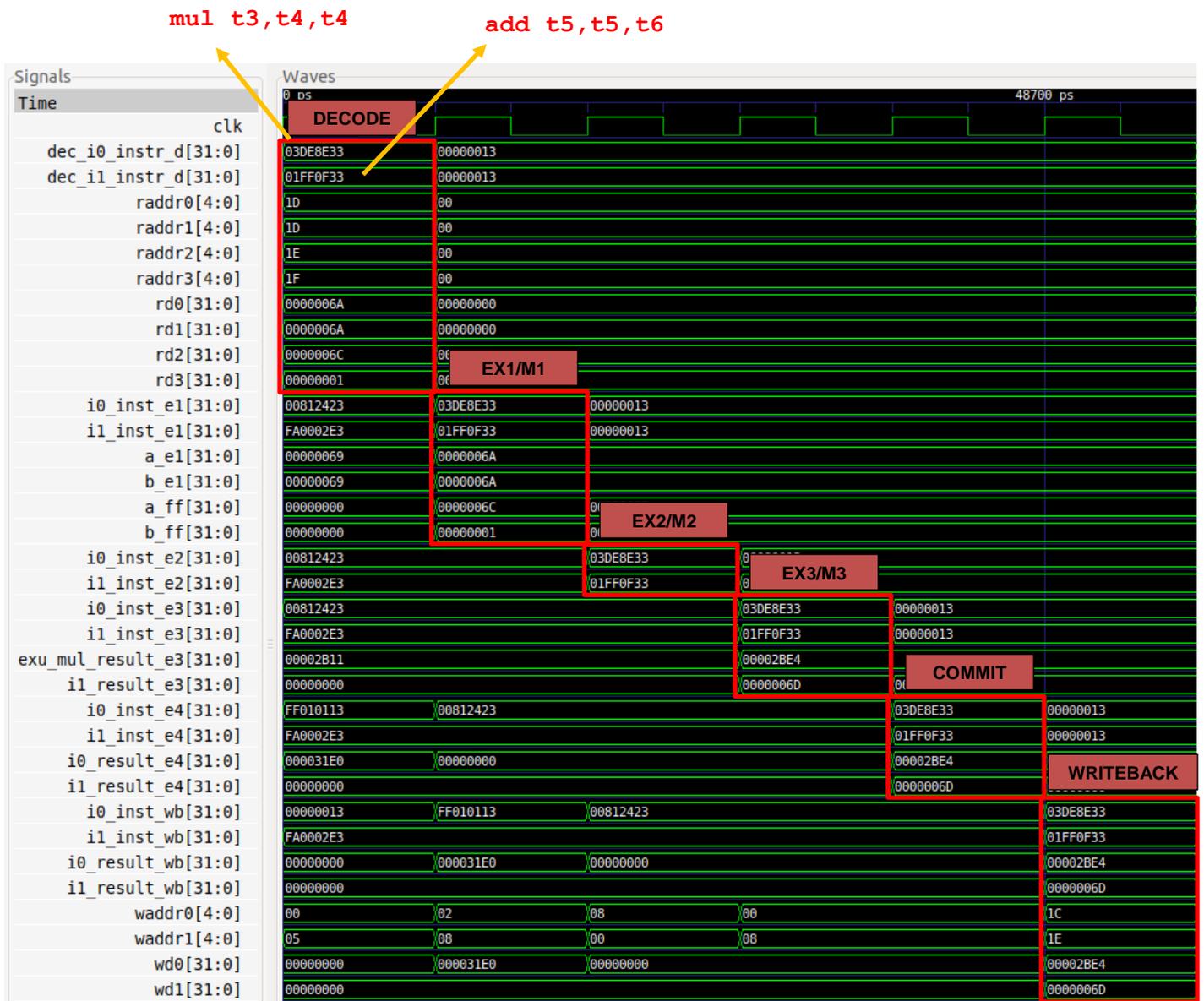


Figure 12. Simulation of the final stages: Decode, EX1/M1, EX2/M2, EX3/M3, Commit, Writeback

TASK: Replicate the simulation from Figure 11 and Figure 12 on your own computer by following these steps (as described in detail in Section 7 of the GSG):

- If necessary, generate the simulation binary (*Vrvfpgasim*).
- In PlatformIO, open the project provided at:
`[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram`.
- Establish the correct path to the RVfpga simulation binary (*Vrvfpgasim*) in file *platformio.ini*.
- Generate the simulation trace with Verilator (Generate Trace).
- Open the trace using GTKWave.
- Use files *test_1.tcl* and *test_2.tcl* (provided at `[RVfpgaPath]/RVfpga/Labs/Lab11/ExampleProgram`) for opening the same signals as the ones shown in Figure 11 and Figure 12. For that purpose, on GTKWave, click on *File* → *Read Tcl Script File* and select the *test_1.tcl* or *test_2.tcl* file.
- Click on *Zoom In* () several times and move to 48600ps (or any other iteration of the loop, except the first one).

Analyse the waveform from Figure 11 and Figure 12 and the diagrams from Figure 3 and Figure 4 at the same time. 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.

- **FC1:** In the first cycle of Figure 11, signal *ifc_fetch_addr_f1_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_f1_ext* = 0x000000F0).
- **FC2:** In the second cycle of Figure 11, the Instruction Memory provides a new 128-bit signal that includes the two instructions that we are analysing in the example (*mul* is shown in green and *add* is shown in red):

```
ifu_fetch_data = 0x000000130000001301FF0F3303DE8E33
```

- **Align:** In the final cycle of Figure 11, the two instructions are extracted from the new 128-bit signal and distributed to the two ways that SweRV EH1 includes.

```
ifu_i0_instr = 0x03DE8E33 (Way 0)
ifu_i1_instr = 0x01FF0F33 (Way 1)
```

- **Decode:** In the first cycle of Figure 12, the two instructions are decoded – that is, the instructions' register values are read from the Register File, and the control bits are generated (not shown in the figure, but you can add some of them as described in *RVfpga_SweRVref.docx*). The operands (register values) are placed in *rd0*, *rd1*, *rd2*, and *rd3*.

```
rd0 = 0x0000006A
rd1 = 0x0000006A
rd2 = 0x0000006C
rd3 = 0x00000001
```

- **EX1/M1, EX2/M2, EX3/M3** and **Commit:** In the next three cycles of Figure 12, the addition and the multiplication are carried out. At the end of EX3/M3 the results are

selected using the two 3:1 multiplexers and then propagated to the Commit stage.

$$\begin{aligned} i0_result_e4 &= exu_mul_result_e3 &= 0x6A * 0x6A = 0x2BE4 \\ i1_result_e4 &= i1_result_e3 &= 0x6C + 0x01 = 0x6D \end{aligned}$$

- **Writeback:** In the final cycle of Figure 12, the results are written back to the Register File.

$$\begin{aligned} waddr0 &= 0x1C & wd0 &= 0x2BE4 \\ waddr1 &= 0x1E & wd1 &= 0x6D \end{aligned}$$

3. HARDWARE COUNTERS IN SweRV EH1

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 Labs 12-20 we will regularly use the Performance Counters available in SweRV EH1 for measuring and comparing the different magnitudes.

REAL BENCHMARKS: In folder `[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks` we provide three real applications (CoreMark, Dhrystone and Image Processing) that you will use in Lab 20 for testing the different features of our SweRV EH1 processor. The supplementary document, `RVfpga_SweRVref.docx`, briefly describes these applications in Section 6, and Lab 20 extends these descriptions and proposes several tasks.

A. Performance Counters in SweRV EH1

The RISC-V SweRV EH1 Programmer's Reference Manual (https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_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 SweRV EH1, 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 using bit 0 of the *mgpmpc* register (0 = disable, 1 = enable).

Chapter 7 of the SweRV EH1 Programmer’s Reference Manual

(https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf) describes in detail the features and operation of the four performance counters available in SweRV EH1:

- 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 7-2 in that document lists the 50 countable events available in SweRV EH1, which are summarized in Table 1.

Table 1. List of Countable Events in SweRV EH1

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

B. Use of the Performance Counters by means of Western Digital’s Processor Support Package (PSP)

Using the performance monitoring system at a register-level would be a bit complex; fortunately, WD’s PSP (<https://github.com/westerndigitalcorporation/riscv-fw-infrastructure>) includes several functions that provide a much simpler approach to performance monitoring. If you have installed PlatformIO following the instructions in the GSG, you should find the following two files in your Ubuntu system:

- `~/.platformio/packages/framework-wd-riscv-sdk/psp/psp_performance_monitor_eh1.c`
- `~/.platformio/packages/framework-wd-riscv-sdk/psp/api_inc/psp_performance_monitor_eh1.h`

Windows: The `.platformio` folder is located inside your user folder (`C:\Users\<USER>`). Note that you may need to enable the system for viewing hidden files/folders.

macOS: Like in Linux, the `.platformio` folder is located inside your home folder (`~/.platformio`).

The `.c` file (`psp_performance_monitor_eh1.c`) implements functions that allow you to do things such as enabling/disabling the group performance monitor (`pspEnableAllPerformanceMonitor`), pairing a counter to an event

(`pspPerformanceCounterSet`) or getting the counter value (`pspPerformanceCounterGet`).

The .h file (`psp_performance_monitor_eh1.h`) provides names for each of the events from Table 1 in: `typedef enum pspPerformanceMonitorEvents`.

The following example (Figure 13), provided at `[RVfpgaPath]/RVfpga/Labs/Lab11/HwCounters_Example`, illustrates the use of the four hardware counters available in SweRV EH1 to measure: *cycles*, *instructions*, and *branches committed* and *mispredicted*. The main function:

- Initializes the UART (`uartInit()`)
- Enables the hardware counters (`pspEnableAllPerformanceMonitor(1)`)
- Assigns the events that are to be measured (*cycles*, *instructions* and *branches committed* and *mispredicted*) to each counter (`D_PSP_COUNTER0` - `D_PSP_COUNTER3`)
- Reads the counters (`pspPerformanceCounterGet(D_PSP_COUNTER0)`)
- Calls a simple assembly program (`Test_Assembly()`) and reads the counters again
- Prints the value of each counter using function `printfNexys`.

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. The disassembly file is also shown at the end of Figure 13 so that you know the value of the 32-bit machine instructions that make up the loop body.

File Test.C

```
#if defined(D_NEXYS_A7)
#include <bsp_printf.h>
#include <bsp_mem_map.h>
#include <bsp_version.h>
#else
    PRE_COMPILED_MSG("no platform was defined")
#endif

#include <psp_api.h>

extern void Test_Assembly(void);

int main(void)
{
    int cyc_beg, cyc_end;
    int instr_beg, instr_end;
    int BrCom_beg, BrCom_end;
    int BrMis_beg, BrMis_end;

    /* Initialize Uart */

    uartInit();

    pspEnableAllPerformanceMonitor(1);

    pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
    pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
    pspPerformanceCounterSet(D_PSP_COUNTER2, E_BRANCHES_COMMITTED);
    pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);

    cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
    instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
    BrCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
    BrMis_beg = pspPerformanceCounterGet(D_PSP_COUNTER3);
```

```

Test_Assembly();

cyc_end   = pspPerformanceCounterGet(D_PSP_COUNTER0);
instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
BrCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
BrMis_end = pspPerformanceCounterGet(D_PSP_COUNTER3);

printfNexys("Cycles = %d", cyc_end-cyc_beg);
printfNexys("Instructions = %d", instr_end-instr_beg);
printfNexys("BrCom = %d", BrCom_end-BrCom_beg);
printfNexys("BrMis = %d", BrMis_end-BrMis_beg);

while(1);
}

```

File Test_Assembly.S

```

.globl Test_Assembly

.text

Test_Assembly:

li t1, 0x1
li t3, 0x3
li t4, 0x4
li t5, 0x5
li t6, 0x6
li a0, 0x0
lui a1, 0xF4
add a1, a1, 0x240
nop

REPEAT:
    add a0, a0, 1
    add t3, t3, t1
    sub t4, t4, t1
    or  t5, t5, t1
    xor t6, t6, t1
    bne a0, a1, REPEAT # Repeat the loop

.end

```

File firmware.dis

```

000001e4 <Test_Assembly>:
1e4: 00100313      li    t1,1
1e8: 00300e13      li    t3,3
1ec: 00400e93      li    t4,4
1f0: 00500f13      li    t5,5
1f4: 00600f93      li    t6,6
1f8: 00000513      li    a0,0
1fc: 000f45b7      lui   a1,0xf4
200: 24058593      addi  a1,a1,576 # f4240 <_sp+0xf0788>
204: 00000013      nop

00000208 <REPEAT>:
208: 00150513      addi  a0,a0,1
20c: 006e0e33      add   t3,t3,t1
210: 406e8eb3      sub   t4,t4,t1

```

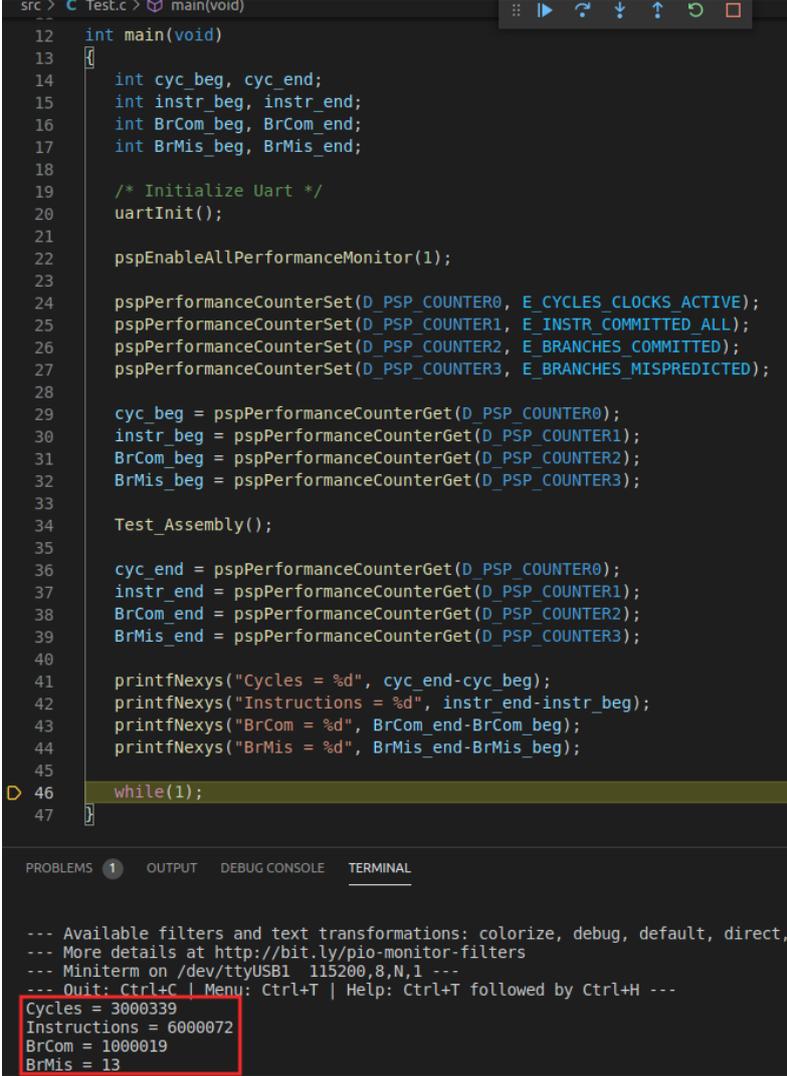
```

214: 006f6f33      or      t5,t5,t1
218: 006fcfb3      xor      t6,t6,t1
21c: feb516e3      bne     a0,a1,208 <REPEAT>

```

Figure 13. Test.C, Test_Assembly.S and firmware.dis

TASK: Execute the program from Figure 13 on the Nexys A7 board as explained in the GSG. You should obtain the results shown in Figure 14 for the four measured events. Explain and justify the results.



```

src > C Test.c > main(void)
12 int main(void)
13
14 int cyc_beg, cyc_end;
15 int instr_beg, instr_end;
16 int BrCom_beg, BrCom_end;
17 int BrMis_beg, BrMis_end;
18
19 /* Initialize Uart */
20 uartInit();
21
22 pspEnableAllPerformanceMonitor(1);
23
24 pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
25 pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
26 pspPerformanceCounterSet(D_PSP_COUNTER2, E_BRANCHES_COMMITTED);
27 pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);
28
29 cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
30 instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
31 BrCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
32 BrMis_beg = pspPerformanceCounterGet(D_PSP_COUNTER3);
33
34 Test_Assembly();
35
36 cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
37 instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
38 BrCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
39 BrMis_end = pspPerformanceCounterGet(D_PSP_COUNTER3);
40
41 printfNexys("Cycles = %d", cyc_end-cyc_beg);
42 printfNexys("Instructions = %d", instr_end-instr_beg);
43 printfNexys("BrCom = %d", BrCom_end-BrCom_beg);
44 printfNexys("BrMis = %d", BrMis_end-BrMis_beg);
45
46 while(1);
47

```

```

--- Available filters and text transformations: colorize, debug, default, direct,
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1 115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Cycles = 3000339
Instructions = 6000072
BrCom = 1000019
BrMis = 13

```

Figure 14. Execution of Test.C

TASK: Measure other events in the Hardware Counters for the program from Figure 13. For this purpose, you must change in file *Test.c* the configuration of the events to be measured with function `pspPerformanceCounterSet`. Note that the different events (shown in Table 1) can be configured using the macros defined in WD's PSP file: `.platformio/packages/framework-wd-riscv-sdk/psp/api_inc/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: pspPerformanceCounterSet(D_PSP_COUNTER3, E_BRANCHES_MISPREDICTED);  
for line: pspPerformanceCounterSet(D_PSP_COUNTER3, E_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 provoke pipeline stalls.