THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga 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 SweRV EH1 pipeline. Figure 1 shows a high-level view of EH1's microarchitecture, with the stages that we analyze in this lab highlighted in red: Decode, EX1, EX2, EX3, Commit (sometimes called EX4), and Writeback of the I0 Pipe. (The I1 Pipe is almost identical to the I0 Pipe, but we delay its deep analysis to Lab 17, when we study superscalar processing. We also analyse the Fetch and Align stages in Lab 11 and Lab 16.)

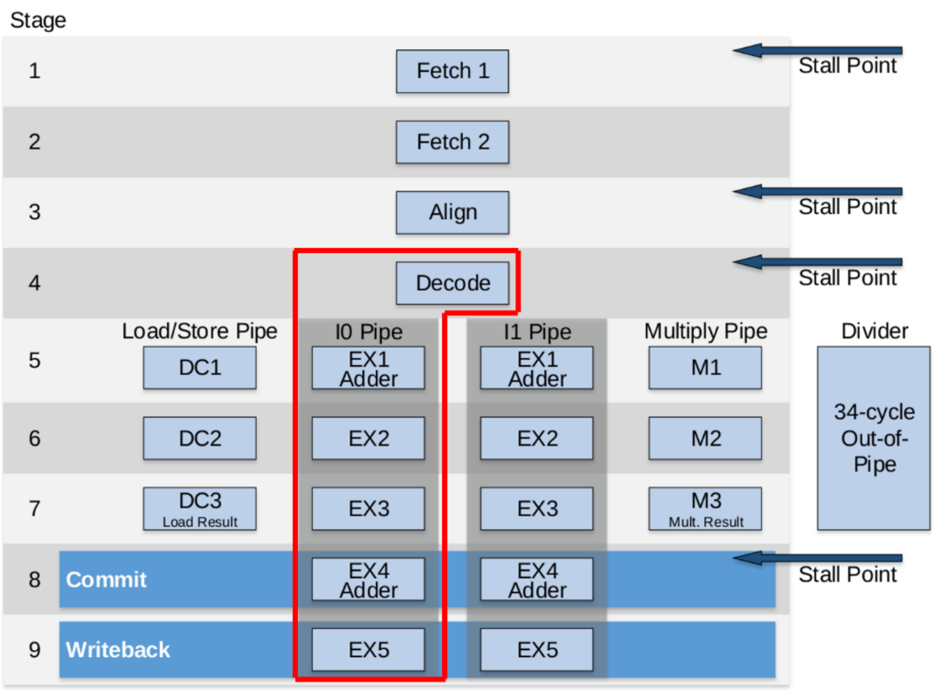


Figure 1. SweRV EH1 pipeline: stages 4-9 of an `add` instruction highlighted

In Section 2 we analyse an `add` instruction from the Decode to Writeback stage, 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 SweRV EH1 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 *[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction* provides the PlatformIO project so that you can analyse, simulate and change the program as desired. As explained in Section 2 of the SweRVref document, for the sake of simplicity, in this project we disable the use of compressed instructions. 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, as we also did in the example included in that Lab, 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 for an `add` instruction

If you open the project in PlatformIO, build it, and open the disassembly file (available at *[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/.pio/build/swervolf_nexys/firmware.dis*) you will see that the `add` instruction (0x01de0e33) is placed at address 0x00000108 in this program.

```
0x00000108:   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 Verilator 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 the Decode, EX1 and Writeback (WB) stages. The values highlighted in red correspond to the `add` instruction as it traverses these three stages through the I0 Pipe. Note that the signals shown in the figure correspond to the I0 Pipe.

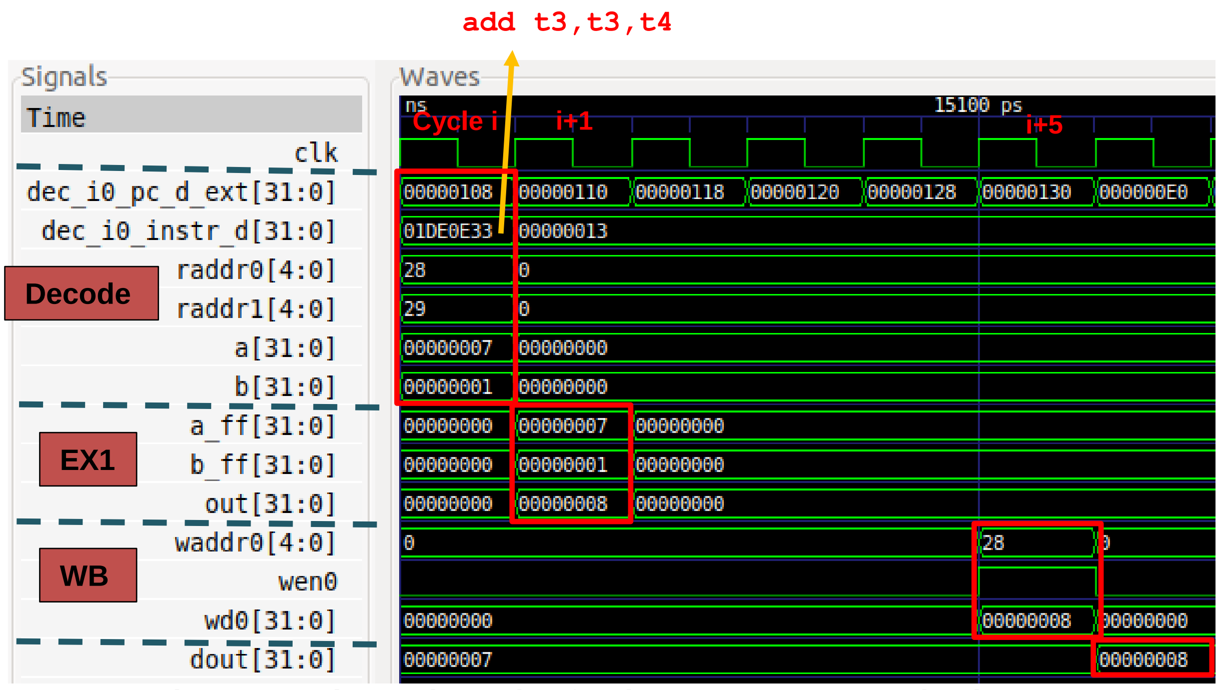


Figure 3. Verilator simulation for the example program in Figure 2

Figure 4 shows a simplified diagram of the SweRV EH1 pipeline executing the `add` instruction during the fourth iteration of the loop (see program in Figure 2) through the I0 Pipe. Note that the figure merges the state of the processor in different cycles:

- **Cycle i:** **Decode:** The instruction is decoded and the Register File is read. The `add` instruction is sent through the I0 Pipe.
- **Cycle i+1:** **EX1:** The addition is computed by the ALU.
- **Cycle i+5:** **Writeback:** The result of the addition is written to the Register File using write port 0.

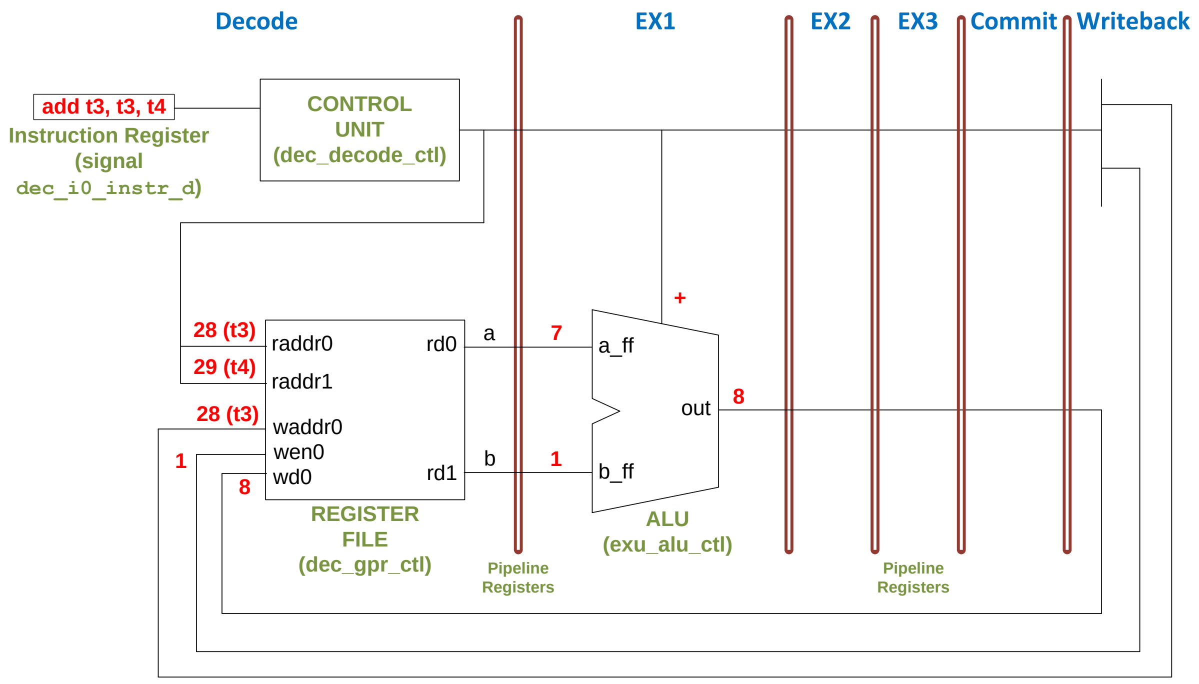


Figure 4. SweRV EH1 pipeline executing an `add` instruction

<u>TASK</u>: Replicate the simulation from Figure 3 on your own computer. To do so, follow the next 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/Lab12/ADD_Instruction*.
- 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 on GTKWave.
- Use file *test_1.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/*) for opening the same signals as the ones shown in Figure 3. For that purpose, on GTKWave, click on *File – Read Tcl Script File* and select the *test_1.tcl* file.
- Click on *Zoom In* (⊕) several times and move to 15000ps.

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**: **Decode:** 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).

  During this stage, **control signals** are generated and the **Register File is read**. In the next stage (EX1), the operands will be sent to the ALU in the I0 pipe. Signals `raddr0` and `raddr1` (shown in decimal in the figures) contain the two source register numbers of the `add` instruction, and signals `a` and `b` contain the values that will be sent to the ALU in the next (EX1) stage. In this case, `a` and `b` are the values read from the Register File. For other instructions, `a` and `b` may be different values; for example, `b` could be an immediate. We will analyse other instructions in later labs.

- Cycle i+1: **EX1:** The `add` instruction is **executed**. Signals `a_ff` and `b_ff` contain the inputs to the ALU (in this case, 7 and 1, respectively), whereas signal `out` contains the result of the addition (8).

- Cycle i+5: **Writeback**: Finally, 4 cycles later, the result of the addition is **written-back** to the Register File through signal `wd0` = 0x8, 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 decimal, `waddr0 = 28`). You can observe that, in the following cycle (last cycle shown in the figure), register x28 has been updated with the new value (`dout` = 8).

  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.

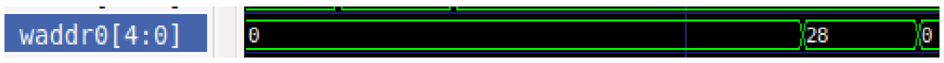


Figure 5. Signal `waddr0` shown in decimal format.

B. Advanced analysis of the `add` instruction

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

Figure 6 shows a detailed diagram of the main elements that an `add` instruction traverses during its execution through the I0 Pipe. This was already illustrated in Figure 4 of Lab 11 (we recommend comparing both figures), but we now focus only on the I0 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 SweRV EH1 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 SweRV EH1 processor.

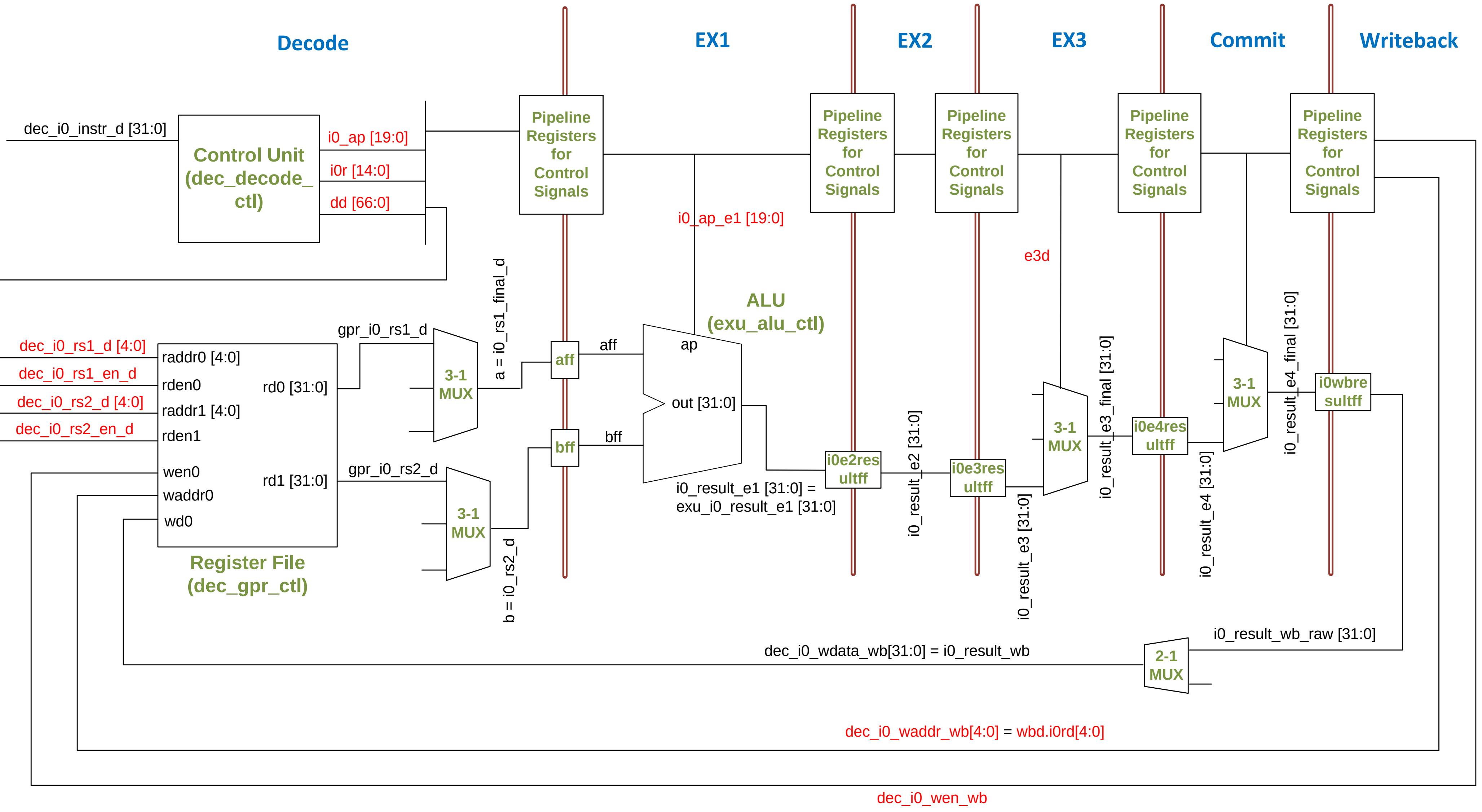


Figure 6. Main units used by Arithmetic-Logic instructions flowing through the I0 pipe

i. Decode Stage

As explained in Lab 11, the Decode Stage is responsible for two main tasks:

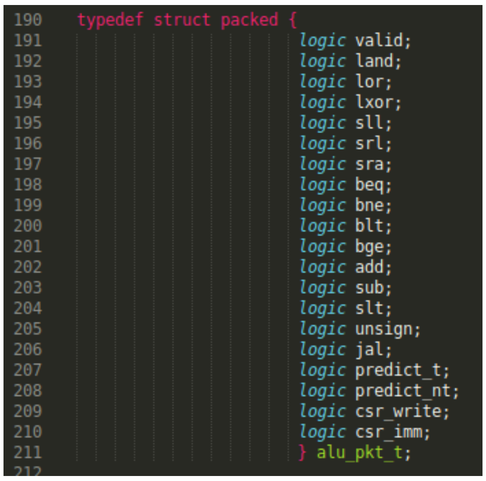
- **Decode the instructions and generate control signals**.
- **Read or assemble the source operands and send the instructions to the appropriate pipes**.

We next analyse each of these tasks for the `add` instruction and add some related signals to the simulation.

<u>Decode the instructions and generate control signals</u>:

As explained in Section 2.C.i of Lab 11, several structures are defined at *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv* for grouping the control bits. Three of these structures are directly related to Arithmetic-Logic (A-L) instructions:

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

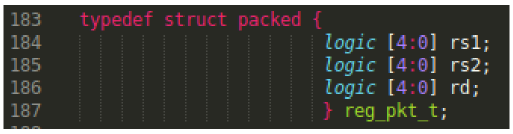


```
typedef struct packed {
                    logic valid;
                    logic land;
                    logic lor;
                    logic lxor;
                    logic sll;
                    logic srl;
                    logic sra;
                    logic beq;
                    logic bne;
                    logic blt;
                    logic bge;
                    logic add;
                    logic sub;
                    logic slt;
                    logic unsign;
                    logic jal;
                    logic predict_t;
                    logic predict_nt;
                    logic csr_write;
                    logic csr_imm;
                    } alu_pkt_t;
```

  Two signals of this type, called `i0_ap` (for Way-0) and `i1_ap` (for Way-1) are defined and assigned inside module **dec_decode_ctl** at the Decode Stage, and propagated through the subsequent execute (EX1-4) stages (signals `i0_ap_e1`, `i0_ap_e2`, `i0_ap_e3` and `i0_ap_e4` for Way-0, and signals `i1_ap_e1`, `i1_ap_e2`, `i1_ap_e3` and `i1_ap_e4` for Way-1). They contain the control signals for informing the ALU of the operation that it must perform. When an `add` instruction is executed, all bits of `i0_ap`/`i1_ap` are set to 0 except for the following:
  - o `valid`: indicating that this is a valid ALU instruction
  - o `add`: indicating that this is an `add` instruction

  When the instruction in Way-0/1 is not an A-L instruction, all bits of signal `i0_ap/i1_ap` are 0 (and specifically `valid` = 0), which makes the I0/I1 ALU not work at all.

- **`reg_pkt_t`**: Two signals of this type, called `i0r` (for Way-0) and `i1r` (for Way-1) are defined, assigned and used inside module **dec_decode_ctl**. They contain the numbers of the two source registers (fields `rs1` and `rs2`) and the destination register (field `rd`):



```
typedef struct packed {
                     logic [4:0] rs1;
                     logic [4:0] rs2;
                     logic [4:0] rd;
                     } reg_pkt_t;
```

- **`dest_pkt_t`**: One signal of this type, called `dd`, is defined and assigned inside module **dec_decode_ctl** during the Decode stage. The signal is propagated through all of the remaining stages (signals `e1d`, `e2d`, `e3d`, `e4d`, and `wbd`). It contains several fields, such as the destination register of the instructions in Way-0 and Way-1: `i0rd[4:0]` and `i1rd[4:0]` respectively.

Some of these signals are used in the Decode stage and are not propagated through the Control Pipeline Registers to later stages. This is the case for `i0r.rs1`/`i1r.rs1` and `i0r.rs2`/`i1r.rs2` which are directly provided to the Register File during the Decode stage for reading the two input operands (signals `raddr0`, `raddr1`, `raddr2`, `raddr3`).

<u>TASK</u>: Find in the Verilog code (module **dec_decode_ctl**) how the `i0r` control signal is used for reading the Register File during the Decode stage.

However, other control signals must be propagated to later stages. This is the case for `i0_ap`/`i1_ap,` which are used by the ALU in order to know the operation that it must perform (in our case, an addition), or signal `dd`, which is used, among other things, by the Register File for writing the two results.

<u>TASK</u>: Find in the Verilog code (module **exu**) how the `i0_ap` and the `dd` control signals are propagated from the Decode stage to the Execute (EX1) stage. Also, find how the `dd` control signal is used by the Register File during the Write-Back stage, after traversing all the stages from Decode to Writeback.

<u>Read or assemble the source operands and send the instructions to the appropriate pipes</u>:

As explained in Lab 11, the SweRV EH1 processor includes several pipes for executing the instructions. In the Decode stage, the instructions, once decoded, must be scheduled through the appropriate pipe. Specifically, if an A-L instruction is at Way 0 it must be sent, if possible, to the I0 pipe; similarly, if an A-L instruction is at Way 1 it must be sent, if possible, to the I1 pipe. In the program that we are analysing in this lab (Figure 2), once the processor has decoded at Way 0 the `add` instruction (i.e. it “knows” that it is an A-L instruction and thus it must send it to the I0 pipe), it must check if all the conditions for execution through the I0 pipe are met: Valid decoding?, 2 input operands available?, Pipeline not blocked?… In our case, the result of this check is sent to the I0 Pipe through two status signals that are computed in the **dec_decode_ctl** module and that are used by the ALU in the **exu** module (in the next subsection we will explain the ALU in more detail). These two status signals are:

- `i0_e1_ctl_en` (renamed `enable` inside the ALU): This signal depends on `dec_i0_ctl_en[4:1]`, which establishes, at decode time, if each of the Execution stages (EX1-3) and the Commit stage of Way 0 must be enabled (1) or not (0). Note that the instruction could be illegal, or the pipeline might be blocked, flushed, etc., due to different circumstances (wrong branch prediction, division computation, etc.), that would disable the pipeline.

- `dec_i0_alu_decode_d` (renamed as `valid` inside the ALU): This signal is 1 if the instruction at Way 0 has been legally decoded, it is an Arithmetic/Logic instruction and it does not use the Secondary ALU (we will explain this structure in Lab 15).

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 (`i0_e1_ctl_en` and `dec_i0_alu_decode_d`) is quite a complex process that we do not explain here in detail but that you can further analyse on your own in modules **dec_decode_ctl** and **exu**.

As also explained in Lab 11, the input operands are provided to the I0 pipe (`i0_rs1_final_d` and `i0_rs2_d`) through two 3:1 multiplexers implemented in the Decode 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_final_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 **exu**) the 3:1 multiplexer on the bottom (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). You do not need to look into the inputs too closely, as they will be analysed in the exercises proposed in Section 3 and in future labs.

Figure 7 extends the Verilator simulation from Figure 3 by adding the signals introduced above:

- `i0_ap[19:0]`
- `i0_ap.valid` (named in the figure as `i0_ap_valid` by means of an alias in the *.tcl* script described below and available at *[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/test_2.tcl*)
- `i0_ap.add` (named in the figure as `i0_ap_add` by means of an alias in the *.tcl* script described below)
- `i0r[14:0]`
- `raddr0`
- `raddr1`
- `i0_e1_ctl_en` (renamed as `enable` in the ALU)
- `dec_i0_alu_decode_d` (renamed as `valid` in the ALU)
- `gpr_i0_rs1_d[31:0]`
- `gpr_i0_rs2_d[31:0]`

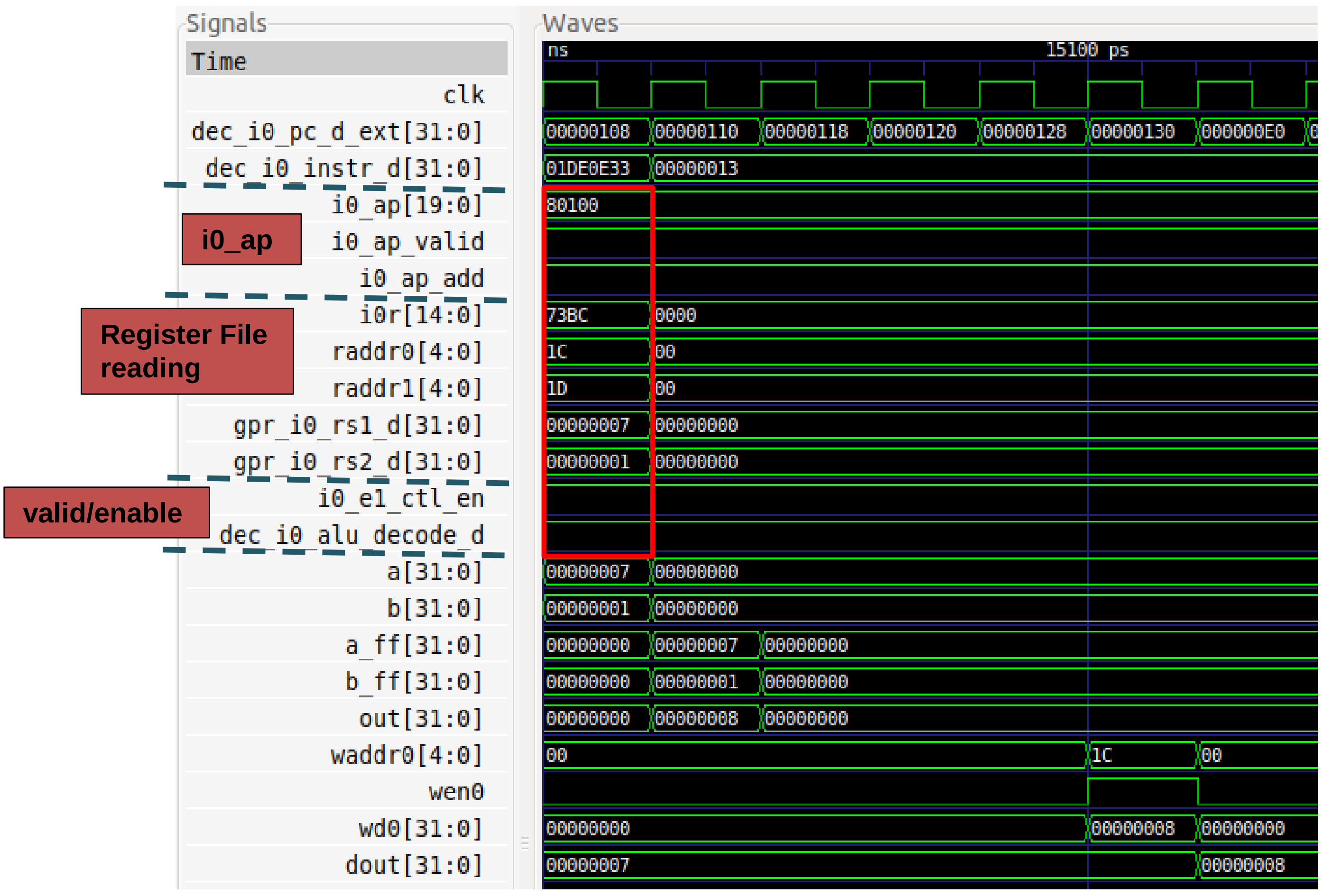


Figure 7. Verilator simulation of the example program of Figure 2, including control signals and Register File read ports

TASK: Replicate the simulation from Figure 7 on your own computer. You can use the *.tcl* script provided at: *[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/test_2.tcl*. Note that aliases are used in this *.tcl* file for some of the control bits.

Analyse the waveform from Figure 7. As explained above, all bits of `i0_ap` are 0 except the `valid` and `add` bits. As also explained, signal `i0r` contains the identifiers of the two source and one destination register of the `add` instruction. Using `i0r.rs1` and `i0r.rs2`, the Register File is accessed (see signals `raddr0` and `raddr1`) and the values read are provided to the I0 pipe: `gpr_i0_rs1_d` = `a` = 0x7 and `gpr_i0_rs2_d` = `b` = 0x1. Finally, you can see that both the `valid` and `enable` signals are 1, thus the I0 pipe ALU will be used in the next cycle.

TASK: In the example from Figure 2, replace the `add` instruction with a non A-L instruction (such as a `mul` instruction). Verify that the `i0_ap` signal has all its fields equal to 0 and that this makes the I0 ALU not work (you will see that signals `a_ff` and `b_ff` for the I0 Pipe at the EX1 Stage remain stable for this instruction). You can use the same *test_2.tcl* file used in the example from Figure 7.

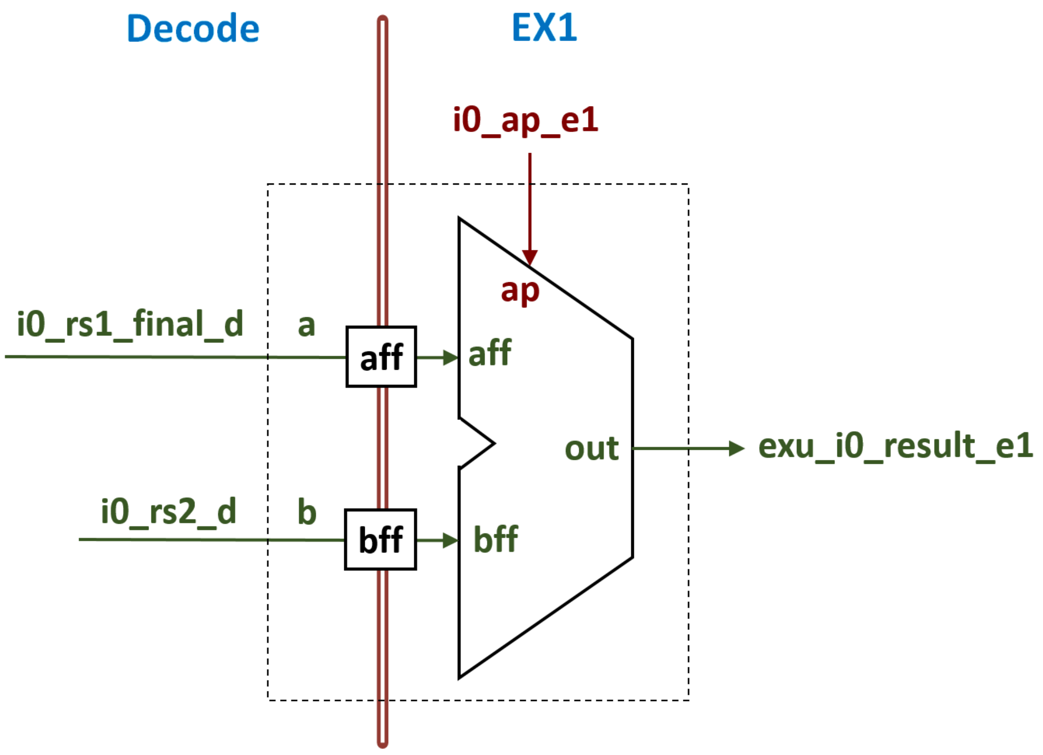
ii. Execution Stage

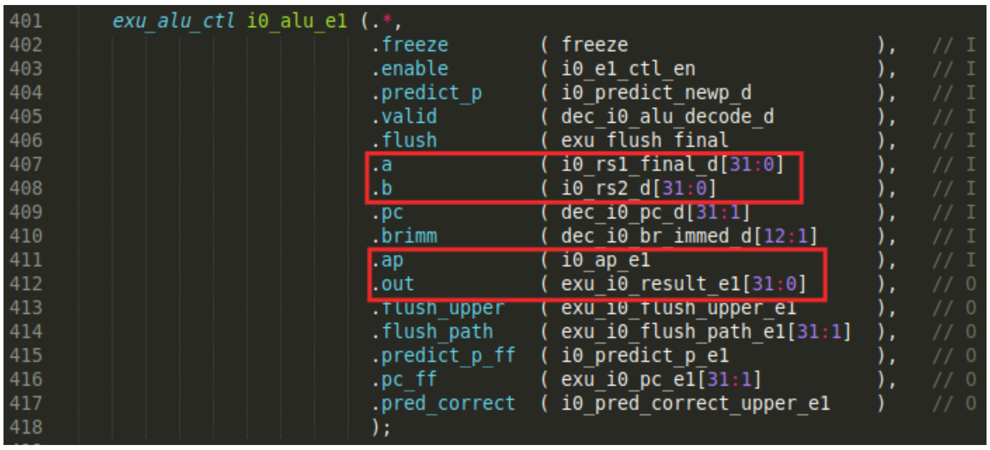
As explained in Lab 11, SweRV EH1 includes four execution pipes (see Figure 4 in Lab 11): I0/I1, Multiply, and L/S pipes. Moreover, it contains a non-pipelined Divider. Each of the pipes is divided in 3 stages: **EX1/EX2/EX3** (I0/I1 Pipes), **M1/M2/M3** (Multiply Pipe), **DC1/DC2/DC3** (L/S Pipe). In this lab we focus on the I0 Pipe, where the `add` instruction is executed. The main task of the I0 pipe for an `add` instruction is to compute the addition in the ALU and propagate it to the Commit stage.

i. EX1 Stage

In this stage the ALU operation is performed – in this case, an addition. The Arithmetic-Logical Unit (ALU) of SweRV EH1 is implemented in module **exu_alu_ctl** (which can be found in *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu_alu_ctl.sv*), and it is instantiated in module **exu** (which can be found at *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu.sv*).

Figure 8 shows the instantiation of the ALU included in the EX1 stage of the I0 pipe, and a simplified diagram of the ALU with some of its input/output ports. Note that most input/output signals are renamed in the ALU.





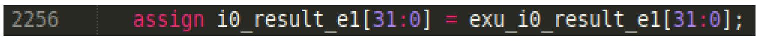
```
    exu_alu_ctl i0_alu_e1 (.*,
                           .freeze        ( freeze                    ),   // I
                           .enable        ( i0_e1_ctl_en              ),   // I
                           .predict_p     ( i0_predict_newp_d         ),   // I
                           .valid         ( dec_i0_alu_decode_d       ),   // I
                           .flush         ( exu_flush_final           ),   // I
                           .a             ( i0_rs1_final_d[31:0]      ),   // I
                           .b             ( i0_rs2_d[31:0]            ),   // I
                           .pc            ( dec_i0_pc_d[31:1]         ),   // I
                           .brimm         ( dec_i0_br_immed_d[12:1]   ),   // I
                           .ap            ( i0_ap_e1                  ),   // I
                           .out           ( exu_i0_result_e1[31:0]    ),   // O
                           .flush_upper   ( exu_i0_flush_upper_e1     ),   // O
                           .flush_path    ( exu_i0_flush_path_e1[31:1] ),  // O
                           .predict_p_ff  ( i0_predict_p_e1           ),   // O
                           .pc_ff         ( exu_i0_pc_e1[31:1]        ),   // O
                           .pred_correct  ( i0_pred_correct_upper_e1  )    // O
                           );
```

Figure 8. I0's ALU (exu_alu_ctl module): High-level diagram and Verilog code

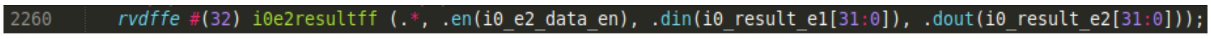
<u>ALU Inputs</u>: The ALU inputs (`a` and `b`) are selected in the Decode stage by the two 3:1 multiplexers shown in Figure 6, as explained in the previous section. Inside the **exu_alu_ctl** module, two registers (`aff` and `bff`) propagate the operands from the Decode stage to the EX1 stage when both the `valid` and `enable` signals are 1.

<u>ALU Control Signals</u>: The ALU is governed by the control bits generated in the Decode stage in signal `i0_ap` (remember that this is an `alu_pkt_t` type structure). This signal is propagated through the Pipeline Registers as explained in the previous section. In EX1, this signal is called `i0_ap_e1` and it is renamed as `ap` inside the ALU (see Figure 8).

<u>ALU Output</u>: The ALU output is obtained in signal `exu_i0_result_e1` (see Figure 8). This signal is propagated to EX2 using a new Pipeline Register (see Figure 6), that you can find in module **dec_decode_ctl** (the signal is first assigned to `i0_result_e1`):



```
   assign i0_result_e1[31:0] = exu_i0_result_e1[31:0];
```

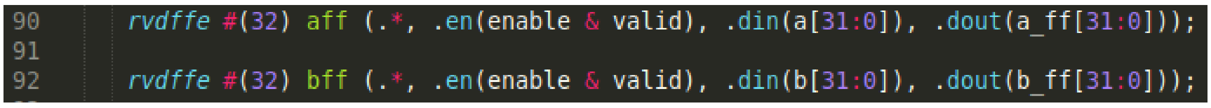


```
   rvdffe #(32) i0e2resultff (.*, .en(i0_e2_data_en), .din(i0_result_e1[31:0]), .dout(i0_result_e2[31:0]));
```

<u>TASK</u>: Include the new signals analysed in this section in the simulation from Figure 7.

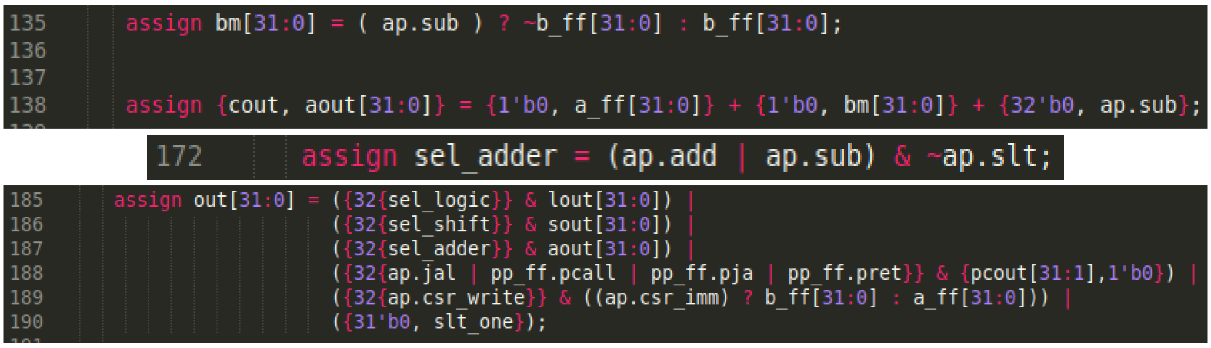
<u>TASK</u>: Perform a simulation of a `sub` instruction similar to the one from Figure 7. Remember that you can include new signals in the simulation through the *.tcl* file.

<u>TASK</u>: Analyse the Verilog implementation of the adder/subtractor implemented in module **exu_alu_ctl**. Figure 9 gives you some help by showing the logic directly related with addition and subtraction operations. You can use a Verilator simulation as a help.



```
   rvdffe #(32) aff (.*, .en(enable & valid), .din(a[31:0]), .dout(a_ff[31:0]));

   rvdffe #(32) bff (.*, .en(enable & valid), .din(b[31:0]), .dout(b_ff[31:0]));
```



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


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

```
assign sel_adder = (ap.add | ap.sub) & ~ap.slt;
```

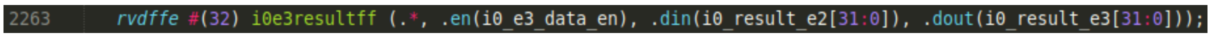
```
assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
                   ({32{sel_shift}} & sout[31:0]) |
                   ({32{sel_adder}} & aout[31:0]) |
                   ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1],1'b0}) |
                   ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
                   ({31'b0, slt_one});
```

Figure 9. Adder inside exu_alu_ctl

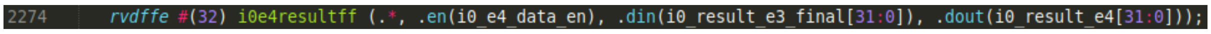
ii. EX2 and EX3 Stages

These stages do few tasks in arithmetic-logic instructions; however, they are necessary in order to synchronize these instructions with other instruction types (such as loads, stores, multiplications, etc.) that do require three cycles to compute their operations. Remember that in a multi-cycle design each instruction can have a different number of stages, but in an in-order pipelined processor such as SweRV EH1, all the instructions must traverse the same number of stages.

In our example, the result of the addition is propagated through new pipeline registers, which can be found in module **dec_decode_ctl**. In the 3:1 multiplexer from EX3, input `i0_result_e3` is selected (see Figure 6). This 3:1 multiplexer was also shown in Figure 4 and Figure 8 of Lab 11. As explained in that lab, it selects the result from the proper pipe, which in our example from Figure 2 is the result provided by the I0 pipe: (`i0_result_e3_final` = `i0_result_e3`).



```
rvdffe #(32) i0e3resultff (.*, .en(i0_e3_data_en), .din(i0_result_e2[31:0]), .dout(i0_result_e3[31:0]));
```



```
rvdffe #(32) i0e4resultff (.*, .en(i0_e4_data_en), .din(i0_result_e3_final[31:0]), .dout(i0_result_e4[31:0]));
```

TASK: Verify in the simulation that this multiplexer selects the result from the expected pipe for the `add` instruction, for the example from Figure 2.

iii. Commit Stage

Similar to EX2 and EX3, this stage does few little for an independent `add` instruction (in Lab 15 we will analyse a scenario where an `add` instruction that depends on a previous instruction must recalculate the addition in the Secondary ALU, not shown in Figure 6). In this example, input `i0_result_e4` is selected by the 3:1 multiplexer available in this stage. This 3:1 multiplexer was also shown in Figure 4 and Figure 9 of Lab 11. In our example from Figure 2, the value selected is again the result provided by the I0 pipe (`i0_result_e4_final` = `i0_result_e4`).

TASK: Verify in the simulation that this multiplexer selects the result from the proper input source (`i0_result_e4`) for the `add` instruction of our example from Figure 2.

iv. Writeback 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 (`i0_result_wb_raw[31:0])` was computed in the EX1 stage and was propagated to this stage. It traverses a 2:1 multiplexer before being passed to the Register File (the other input of this multiplexer comes from the Divisor, as we will analyse in Lab 14). 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 Decode stage and propagated to the Writeback 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 SweRV EH1 for this instruction?
   b. Which resources (blocks) perform a useful function for this instruction?
   c. Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?

3) Analyse in a Verilator 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 a Verilator 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 a Verilator 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 SweRV EH1? 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 `alu_pkt_t`) being stuck at 0.