

## TASKS

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

0x01de0e33 → 0000000 11101 11100 000 11100 0110011

funct7 = 0000000

rs2 = 11101 = x29 (t4)

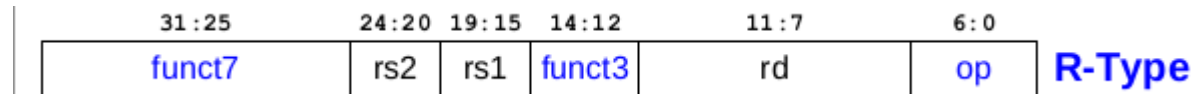
rs1 = 11100 = x28 (t3)

funct3 = 000

rd = 11100 = x28 (t3)

op = 0110011


From Appendix B of DDCARV:



op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2

Name	Register Number	Use
<b>zero</b>	<b>x0</b>	Constant value 0
<b>ra</b>	<b>x1</b>	Return address
<b>sp</b>	<b>x2</b>	Stack pointer
<b>gp</b>	<b>x3</b>	Global pointer
<b>tp</b>	<b>x4</b>	Thread pointer
<b>t0-2</b>	<b>x5-7</b>	Temporary variables
<b>s0/fp</b>	<b>x8</b>	Saved variable / Frame pointer
<b>s1</b>	<b>x9</b>	Saved variable
<b>a0-1</b>	<b>x10-11</b>	Function arguments / Return values
<b>a2-7</b>	<b>x12-17</b>	Function arguments
<b>s2-11</b>	<b>x18-27</b>	Saved variables
<b>t3-6</b>	<b>x28-31</b>	Temporary variables

**TASK:** 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.

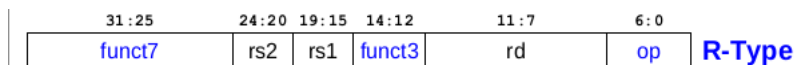
Solution provided in the main document of Lab 12.

**TASK:** Locate the main structures and signals from Figure 6 in the Verilog files of the SweRV EH1 processor.

- Control Unit in module **dec\_decode\_ctl**
- Register file:
  - o Instantiation in line 525 of module **dec**.
  - o Implementation in module **dec\_gpr\_ctl**.
- 3:1 muxes in Decode stage: Line 279 of module **exu**.
- Pipeline Registers for Control Signals: Distributed in several modules.
- Registers aff and bff: Lines 90 and 92 of module **exu\_alu\_ctl**.
- I0 ALU at EX1:
  - o Instantiation in line 401 of module **exu**.
  - o Implementation in module **exu\_alu\_ctl**.
- Pipeline registers with the result of the operation (i0e2resultff, i0e3resultff, i0e4resultff, i0wbresultff): Lines 2260-2283 of module **dec\_decode\_ctl**.
- 3:1 mux in EX3 stage: Line 2268 of module **dec\_decode\_ctl**.
- 3:1 mux in EX4 stage: Line 2277 of module **dec\_decode\_ctl**.
- 2:1 mux in Writeback stage: Line 2286 of module **dec\_decode\_ctl**.

**TASK:** Find in the Verilog code (module **dec\_decode\_ctl**) how the **i0r** control signal is used for reading the Register File.

- The register identifiers are obtained from the 32-bit instruction in Way-0: signal `i0[31:0] = dec_i0_instr_d[31:0]`.  
In an R-Type instruction they are located in the following fields:



In module **dec\_decode\_ctl**:

```
1121 assign i0r.rs1[4:0] = i0[19:15];
1122 assign i0r.rs2[4:0] = i0[24:20];
1123 assign i0r.rd[4:0] = i0[11:7];
```

- The register identifiers and read enable signals are assigned to `dec_i0_rs1_d/dec_i0_rs2_d` and `dec_i0_rs1_en_d/dec_i0_rs2_en_d`.

These signals are sent from module **dec** to module **dec\_decode\_ctl**. In module **dec\_decode\_ctl**:

```
1130 assign dec_i0_rs1_en_d = i0_dp.rs1 & (i0r.rs1[4:0] != 5'd0);
1131 assign dec_i0_rs2_en_d = i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
1132 assign i0_rd_en_d = i0_dp.rd & (i0r.rd[4:0] != 5'd0);
1133
1134 assign dec_i0_rs1_d[4:0] = i0r.rs1[4:0];
1135 assign dec_i0_rs2_d[4:0] = i0r.rs2[4:0];
1136 assign i0_rd_d[4:0] = i0r.rd[4:0];
```

- The register identifiers and read enable signals are provided to the Register File, which is instantiated in module **dec**. In module **dec**:

```
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               );
```

**TASK:** Find in the Verilog code (module **exu**) how the `i0_ap` and the `dd` control signals are propagated from the Decode Stage to the Execution Stage. Also, find how the `dd` control signal is used by the Register File at the Write-Back Stage, after traversing all the stages from Decode to Writeback.

Signal `i0_ap` is obtained in module **dec\_decode\_ctl**. It is provided to module `exu`, where it is propagated to EX1, EX2, EX3 and Commit (EX4). In module **exu**:

```
454   rvdffe #($bits(alu_pkt_t)) i0_ap_e1_ff (.*, .en(i0_e1_ctl_en), .din(i0_ap), .dout(i0_ap_e1) );
455   rvdffe #($bits(alu_pkt_t)) i0_ap_e2_ff (.*, .en(i0_e2_ctl_en), .din(i0_ap_e1), .dout(i0_ap_e2) );
456   rvdffe #($bits(alu_pkt_t)) i0_ap_e3_ff (.*, .en(i0_e3_ctl_en), .din(i0_ap_e2), .dout(i0_ap_e3) );
457   rvdffe #($bits(alu_pkt_t)) i0_ap_e4_ff (.*, .en(i0_e4_ctl_en), .din(i0_ap_e3), .dout(i0_ap_e4) );
```

Signal `dd` is obtained in module **dec\_decode\_ctl** and propagated to EX1, EX2, EX3, Commit (EX4) and WB (EX5). In module **dec\_decode\_ctl**:

```
2139   rvdffe #($bits(dest_pkt_t) ) e1ff (.*, .en(i0_e1_ctl_en), .din(dd), .dout(e1d));
2155   rvdffe #($bits(dest_pkt_t) ) e2ff (.*, .en(i0_e2_ctl_en), .din(e1d_in), .dout(e2d));
2168   rvdffe #($bits(dest_pkt_t) ) e3ff (.*, .en(i0_e3_ctl_en), .din(e2d_in), .dout(e3d));
2193   rvdffe #($bits(dest_pkt_t) ) e4ff (.*, .en(i0_e4_ctl_en), .din(e3d_in), .dout(e4d));
2219   rvdffe #($bits(dest_pkt_t) ) wbff (.*, .en(i0_wb_ctl_en | exu_div_finish | div_wen_wb), .din(e4d_in), .dout(wbd));
```

Note that the output of each register is slightly modified (and thus renamed) before going into the next register. You can look at the Verilog code if you want to check the details.

The register identifier for the output operand is assigned at Decode Stage:

```
2070   assign dd.i0rd[4:0] = i0r.rd[4:0];
```

Signal `dd` is propagated from Decode to Writeback as shown above: `dd` → `e1d` → `e2d` → `e3d` → `e4d` → `wbd`. Then the destination register is provided to the Register File at the Writeback stage:

```
2221   assign dec_i0_waddr_wb[4:0] = wbd.i0rd[4:0];
```

```

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             );

```

**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`.

Solution not provided.

**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.

```

286     assign i0_rs2_d[31:0] = ({32{~dec_i0_rs2_bypass_en_d}} & gpr_i0_rs2_d[31:0]) |
287                             ({32{~dec_i0_rs2_bypass_en_d}} & dec_i0_immed_d[31:0]) |
288                             ({32{ dec_i0_rs2_bypass_en_d}} & i0_rs2_bypass_data_d[31:0]);

```

These 3:1 muxes receive 3 inputs:

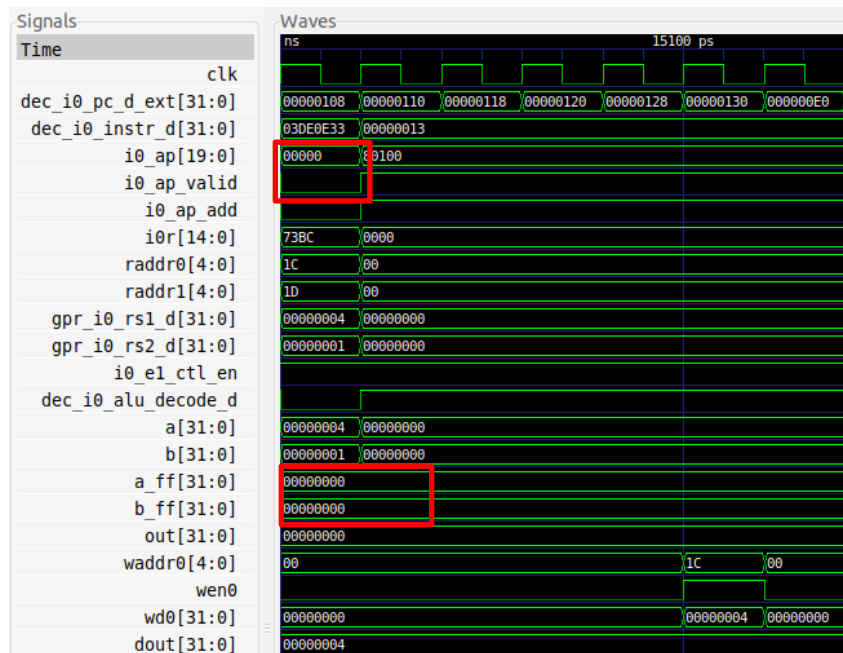
- One from the register file (`gpr_i0_rs2_d`)
- One from the 32-bit instruction register, which constitutes the immediate (`dec_i0_immed_d`)
- One from the bypass logic, that we analyse in Lab 15 (`i0_rs2_bypass_data_d`)

**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.

Solution provided in the main document of Lab 12.

**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.

For example, the simulation of `mul t3, t3, t4 (0x03de0e33)` provides the following results:

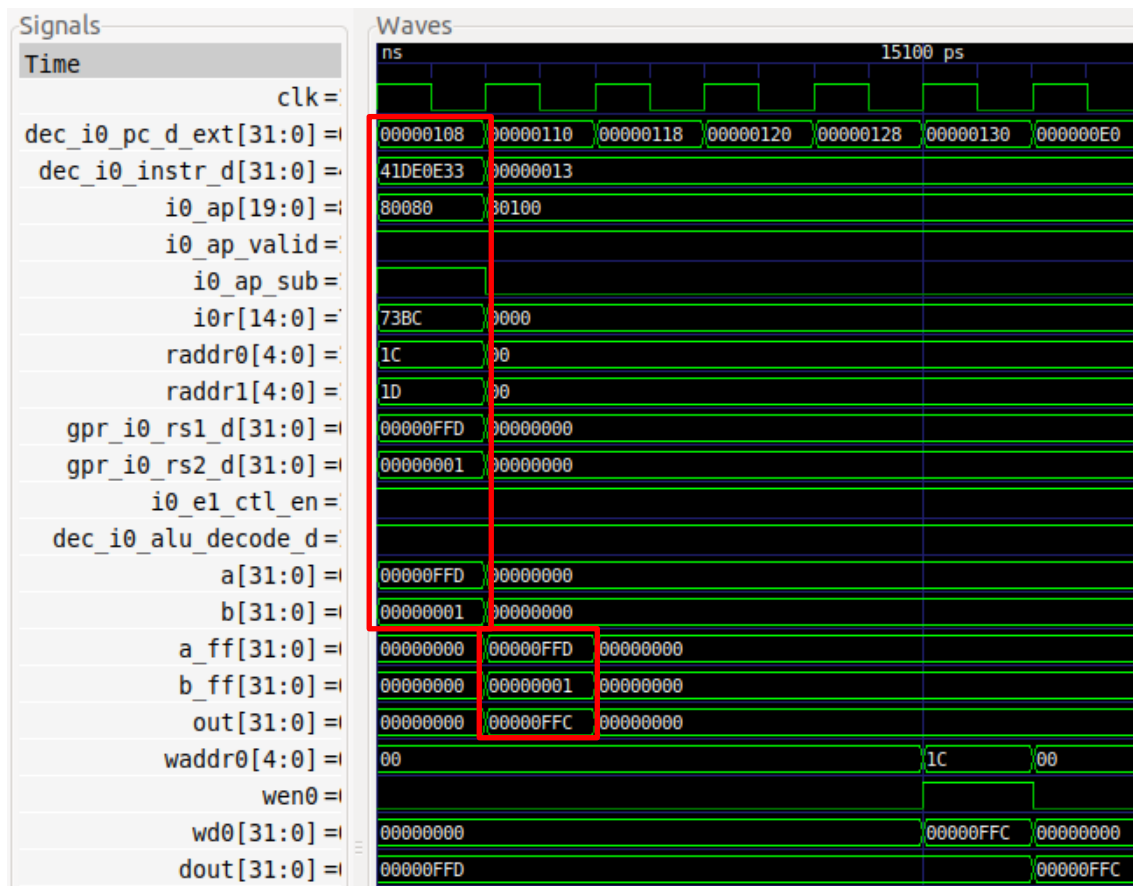


**TASK:** Include the new signals analysed in this section in the simulation from Figure 7.

Solution not provided.

**TASK:** 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.

For example, the simulation of `sub t3, t3, t4 (0x41de0e33)` provides the following results:





**TASK:** Analyse the Verilog implementation of the adder/subtractor implemented in module `exu_alu_ctl`. Figure 8 gives you some help by showing the logic directly related with addition and subtraction operations.

```
90    rvdffe #(32) aff (.*, .en(enable & valid), .din(a[31:0]), .dout(a_ff[31:0]));
91
92    rvdffe #(32) bff (.*, .en(enable & valid), .din(b[31:0]), .dout(b_ff[31:0]));
```

The input operands are propagated from the Decode Stage (a and b) to the Execution Stage (a\_ff and b\_ff).

```
135    assign bm[31:0] = ( ap.sub ) ? ~b_ff[31:0] : b_ff[31:0];
136
137
138    assign {cout, aout[31:0]} = {1'b0, a_ff[31:0]} + {1'b0, bm[31:0]} + {32'b0, ap.sub};
```

This is the adder/subtractor.

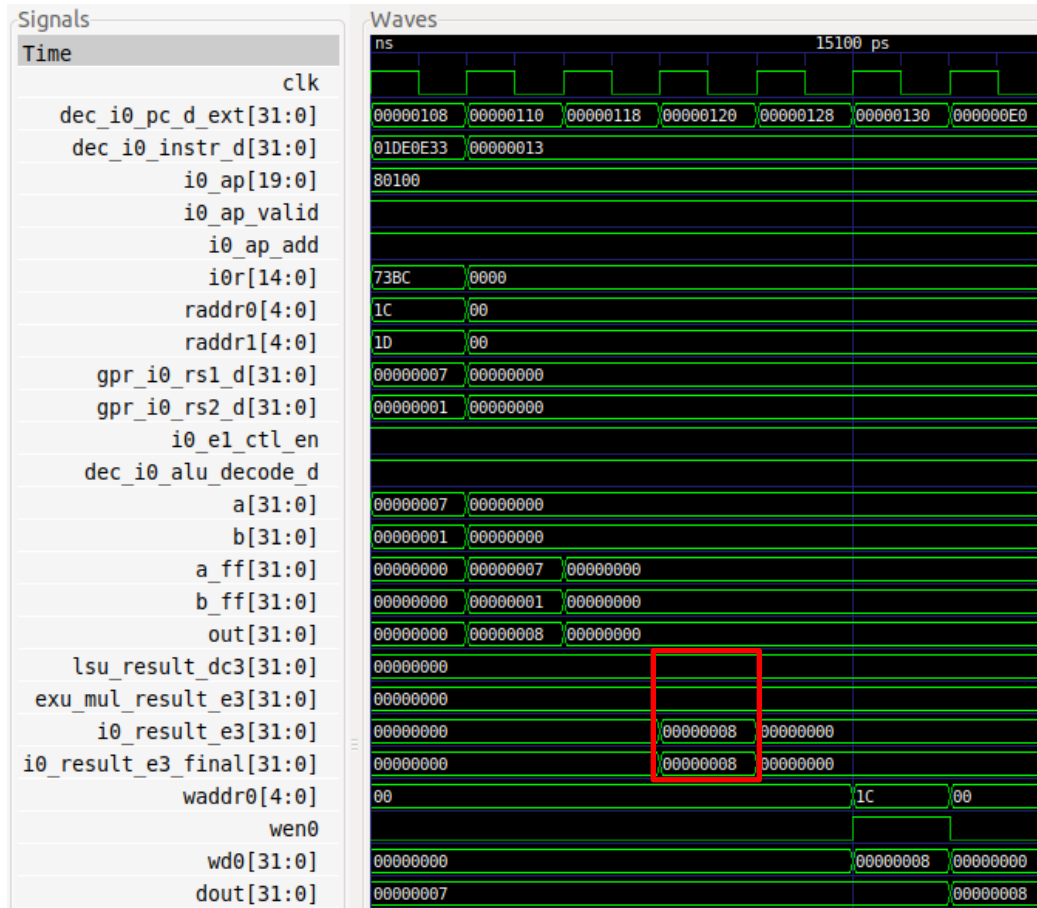
- If the instruction is an addition, aout = a\_ff + b\_ff
- If the instruction is a subtraction, b\_ff is first two's complemented and then a\_out is computed.

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

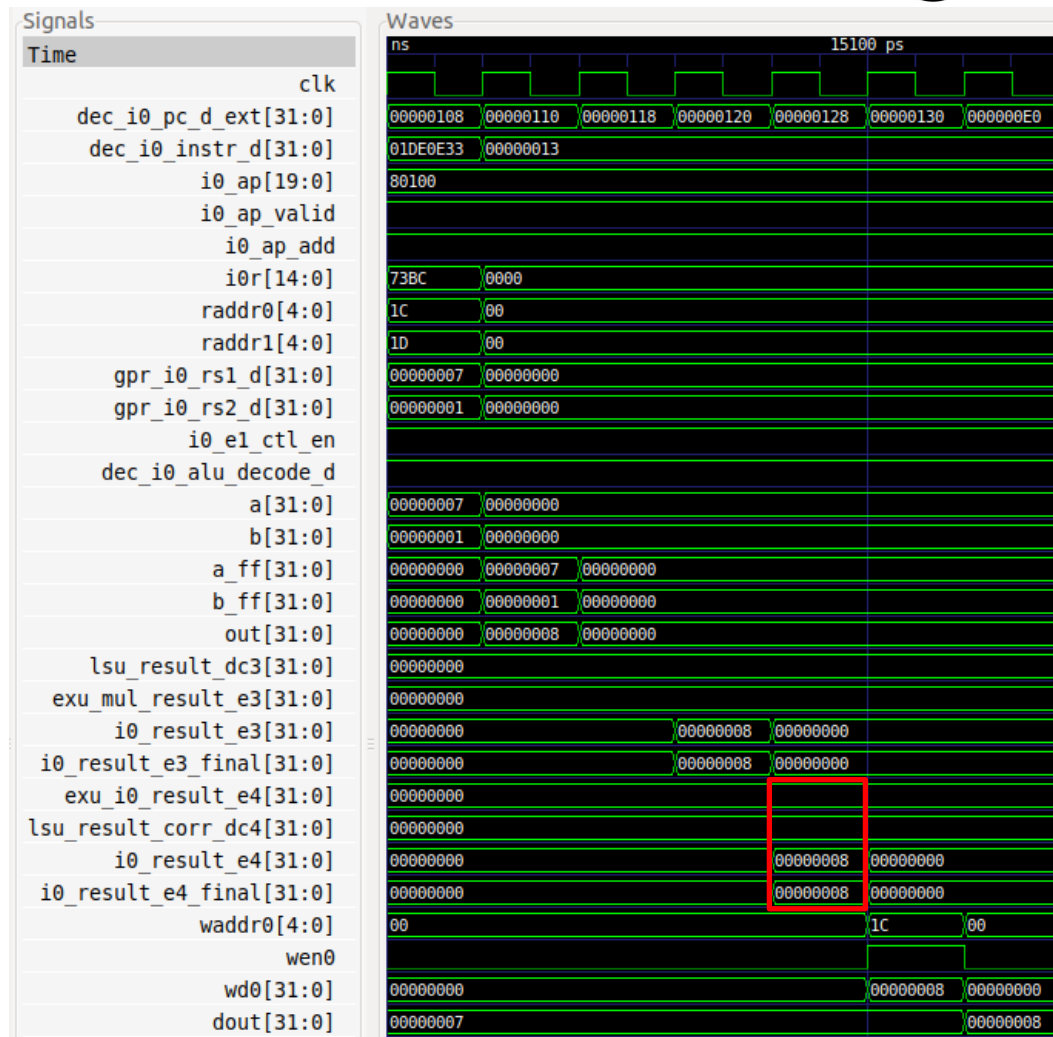
185    assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                      ({32{sel_shift}} & sout[31:0]) |
187                      ({32{sel_adder}} & aout[31:0]) |
188                      ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1], 1'b0}) |
189                      ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                      ({31'b0, slt_one});
```

If the instruction is an addition or a subtraction, then out = aout.

**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.



**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.



**TASK:** In the Verilog code, analyse how signals `wen0` and `waddr0` are generated in the Decode stage and propagated to the Writeback

stage.

```

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             .waddr0(dec_i0_waddr_wb[4:0]), .wen0(dec_i0_wen_wb), .wd0(dec_i0_wdata_wb[31:0]),
533             .waddr1(dec_i1_waddr_wb[4:0]), .wen1(dec_i1_wen_wb), .wd1(dec_i1_wdata_wb[31:0]),
534             .waddr2(dec_nonblock_load_waddr[4:0]), .wen2(dec_nonblock_load_wen), .wd2(lsu_nonblock_load_data[31:0]),
535             // outputs
536             .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0]),
537             .rd2(gpr_i1_rs1_d[31:0]), .rd3(gpr_i1_rs2_d[31:0])
538             );
539
540
541

```

```

2221 assign dec_i0_waddr_wb[4:0] = wbd.i0rd[4:0];

```

```

2224 assign i0_wen_wb = wbd.i0v & ~(~dec_tlu_i0_kill_writeb_wb & ~i0_load_kill_wen & wbd.i0v & wbd.ilv & (wbd.i0rd[4:0] == wbd.ilrd[4:0])) & ~dec_tlu_i0_kill_writeb_wb;
2225 assign dec_i0_wen_wb = i0_wen_wb & ~i0_load_kill_wen; // don't write a nonblock load 1st time down the pipe
2226

```

```

2070 assign dd.i0rd[4:0] = i0r.rd[4:0];
2071 assign dd.i0v = i0_rd_en_d & i0_legal_decode_d;

```

## EXERCISES

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

The following example, provided at [\[RVfpgaPath\]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\\_Solutions/Lab12/AND\\_Instruction](#), illustrates the execution of an `and` instruction contained within a loop that repeats forever. As in the example for the `add` instruction, the `and` instruction (highlighted in red) is surrounded by several `nop` instructions. Two instructions are included at the end of the loop for modifying the values stored in `t3` and `t4`.

```
#define INSERT_NOPS_1      nop;
#define INSERT_NOPS_2      nop; INSERT_NOPS_1
#define INSERT_NOPS_3      nop; INSERT_NOPS_2
#define INSERT_NOPS_4      nop; INSERT_NOPS_3
#define INSERT_NOPS_5      nop; INSERT_NOPS_4
#define INSERT_NOPS_6      nop; INSERT_NOPS_5
#define INSERT_NOPS_7      nop; INSERT_NOPS_6
#define INSERT_NOPS_8      nop; INSERT_NOPS_7
#define INSERT_NOPS_9      nop; INSERT_NOPS_8
#define INSERT_NOPS_10     nop; INSERT_NOPS_9

.globl main
main:

li t3, 0xFC              # t3 = 0xFC
li t4, 0x7                # t4 = 0x7

REPEAT:
    INSERT_NOPS_10
    and t3, t3, t4        # t3 = t3 & t4
    INSERT_NOPS_10
    li t3, 0xFC           # t3 = 0xFC
    add t4, t4, t4
    beq zero, zero, REPEAT # Repeat the loop

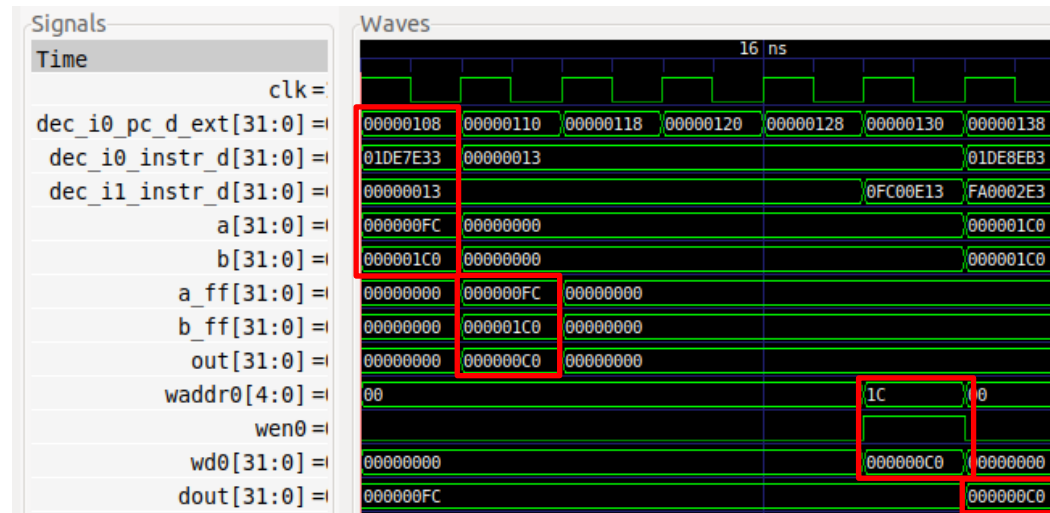
.end
```

If you open the project in PlatformIO, build it, and open the disassembly file (available at [\[RVfpgaPath\]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\\_Solutions/Lab12/AND\\_Instruction/.pio/build/swervolf\\_nexys/firmware.dis](#)) you

will see that the `and` instruction is placed at address `0x00000108`, and you can also see the machine code for the instruction (`0x01de7e33`):

```
0x00000108:      01de7e33      and    t3, t3, t4
```

We next simulate the program in Verilator and then open the trace file generated by the simulator on GTKWave. Move to the any iteration of the loop, except the first one.



Analyse the waveform (the values highlighted in red correspond to the `and` instruction). In this lab we skip the fetch and align stages, which will be explained in a forthcoming lab.

- **Decode** stage: Signal `dec_i0_pc_d_ext` contains the address of the instruction (in the textbooks, this is usually called the Program Counter), which for the `and` is `0x00000108`, and signal `dec_i0_instr_d` contains the 32-bit machine instruction `0x01DE7E33` (in the textbooks, this is usually called the Instruction Register).

In RISC-V, the opcode for the `and` instruction is (see Appendix B of [Harris&Harris]):

```
00000000 | rs2 | rs1 | 111 | rd | 0110011
```

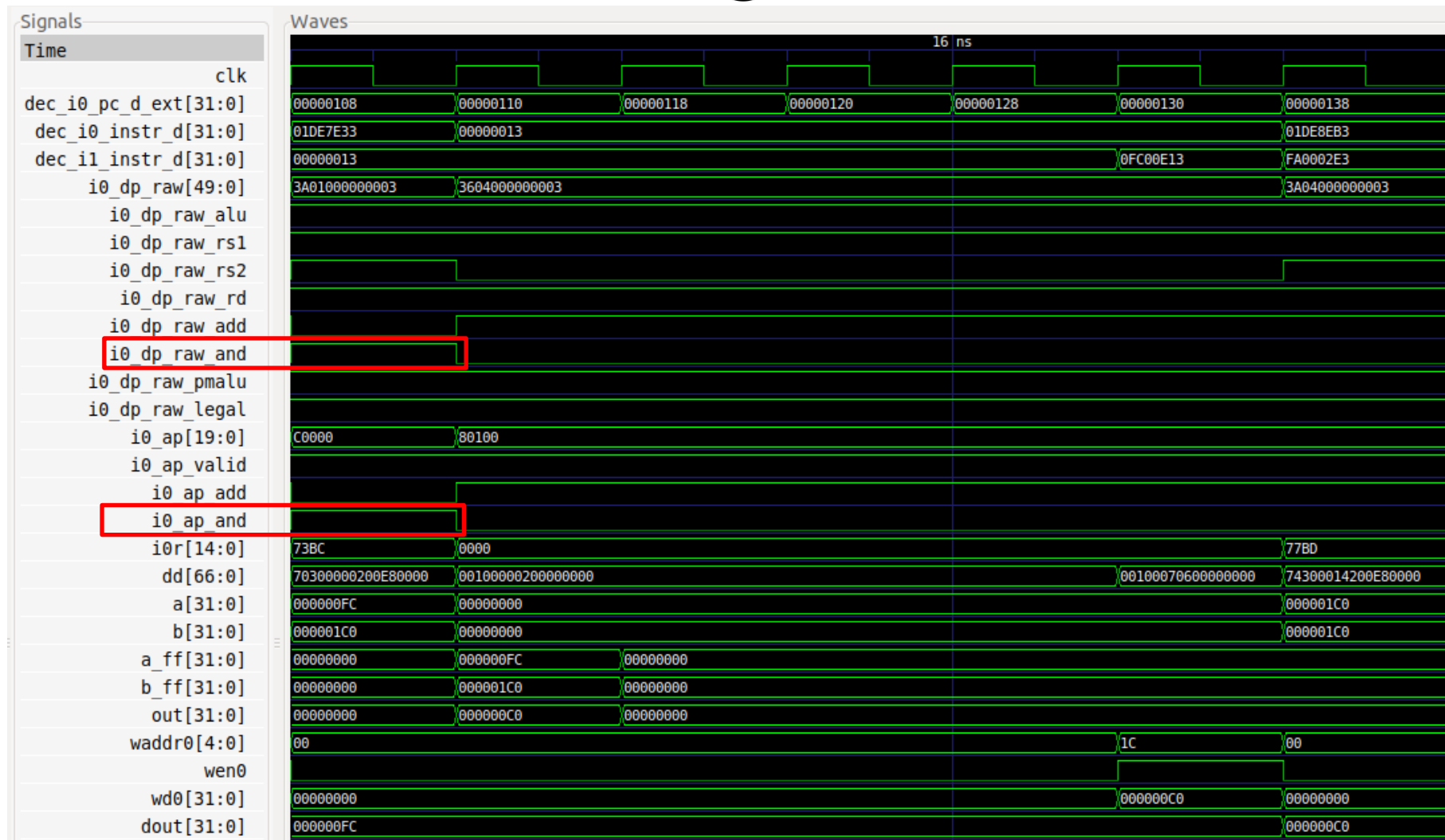
so you can easily verify that `0x01DEFE33` corresponds to: `and t3, t3, t4` (remember that `t3=x28` and `t4=x29`).

During this stage the **pipeline control signals are generated** (we will show some details in the next section). Moreover, the

**Register File is read** in this stage. Signals `a` and `b` contain the inputs to the ALU, which in this case coincide with the values read from the Register File (in other cases that we will analyse in forthcoming labs, this will not be the case).

- **EX1 Stage:** In the next cycle, the `and` instruction is **executed**. Signals `a_ff` and `b_ff` contain the inputs to the ALU (0xFC and 0x1C0 respectively), whereas `out` contains the result of the addition (0xC0).
- **EX5 Stage, also called Writeback:** Finally, 4 cycles later, the result of the addition is **written-back** to the Register File through signal `wd0=0xC0`, which contains the data to write. Given that `wen0=1` (write enable), the result of the `and` operation is written at the end of that cycle into register x28 (the register index, `waddr0=0x1C`). You can observe that, in the following cycle (last cycle shown in the figure), register x28 contains the new value (`dout=0xC0`).

We next add the control signals to the previous simulation:



You can see that the control bit for the `and` instruction is 1 in the first cycle.



The following Verilog fragments show the Logical Unit of SweRV EH1.

```

90      rvdffe #(32) aff (.*, .en(enable & valid), .din(a[31:0]), .dout(a_ff[31:0]));
91
92      rvdffe #(32) bff (.*, .en(enable & valid), .din(b[31:0]), .dout(b_ff[31:0]));

149      assign logic_sel[3] = ap.land | ap.lor;
150      assign logic_sel[2] = ap.lor | ap.lxor;
151      assign logic_sel[1] = ap.lor | ap.lxor;
152
153
154
155      assign lout[31:0] = ( a_ff[31:0] & b_ff[31:0] & {32{logic_sel[3]}} ) |
156                        ( a_ff[31:0] & ~b_ff[31:0] & {32{logic_sel[2]}} ) |
157                        ( ~a_ff[31:0] & b_ff[31:0] & {32{logic_sel[1]}} );

168      assign sel_logic = |{ap.land,ap.lor,ap.lxor};

185      assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                        ({32{sel_shift}} & sout[31:0]) |
187                        ({32{sel_adder}} & aout[31:0]) |
188                        ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1],1'b0}) |
189                        ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                        ({31'b0, slt_one});

```

When the and control bit is 1, the result of the and operation is selected:

$\text{logic\_sel}[3]=1$  and  $\text{logic\_sel}[2]=\text{logic\_sel}[1]=0 \rightarrow \text{lout} = \text{a\_ff} \& \text{b\_ff}$

2) (*The following exercise is based on exercise 4.1 from the book “Computer Organization and Design – RISC-V Edition”, by Patterson & Hennessy ([HePa]).*)

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?

Solution not provided.

3) Analyse, both 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`.

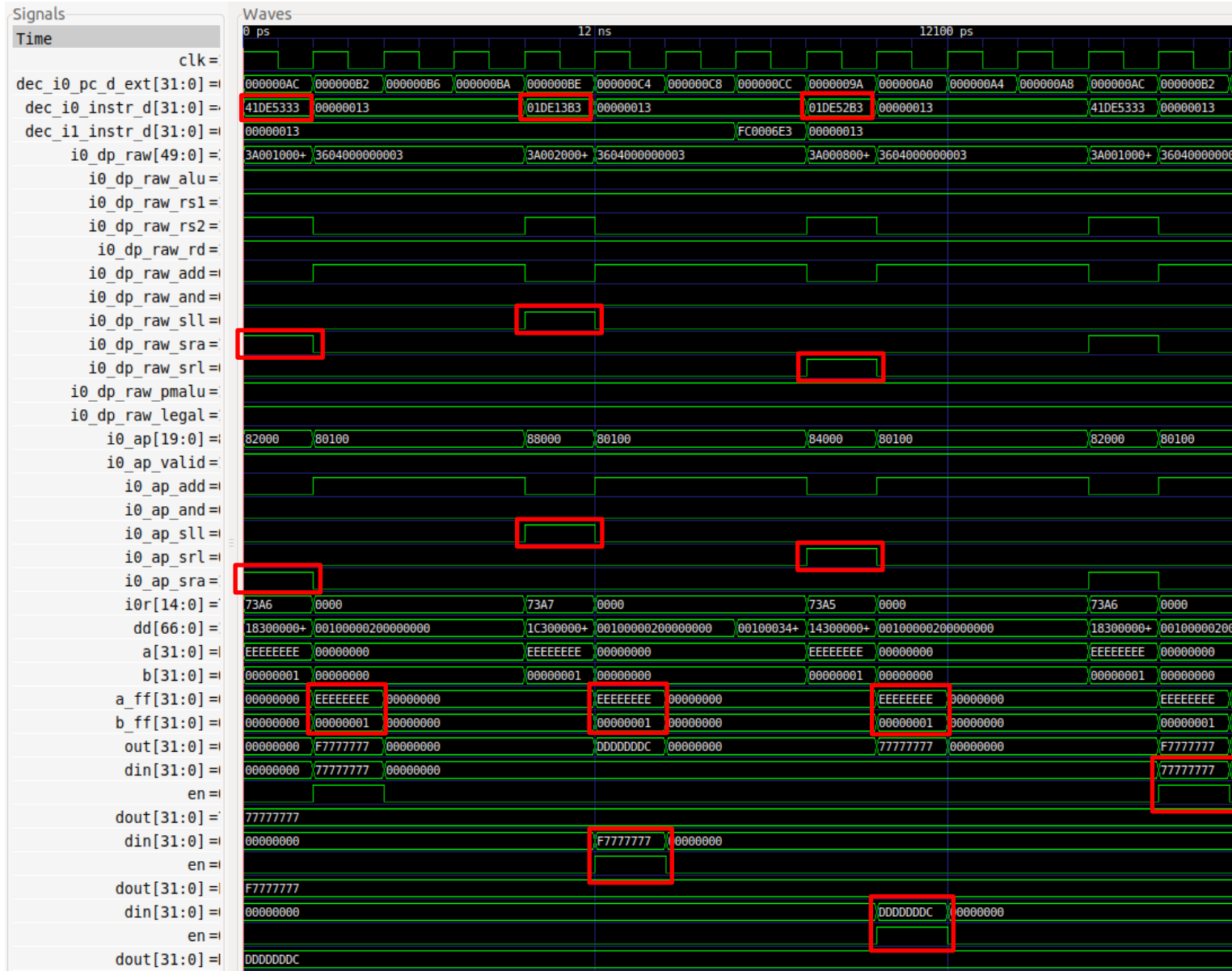
```
#define INSERT_NOPS_0
#define INSERT_NOPS_1      nop; INSERT_NOPS_0
#define INSERT_NOPS_2      nop; INSERT_NOPS_1
#define INSERT_NOPS_3      nop; INSERT_NOPS_2
#define INSERT_NOPS_4      nop; INSERT_NOPS_3
#define INSERT_NOPS_5      nop; INSERT_NOPS_4
#define INSERT_NOPS_6      nop; INSERT_NOPS_5
#define INSERT_NOPS_7      nop; INSERT_NOPS_6
#define INSERT_NOPS_8      nop; INSERT_NOPS_7
#define INSERT_NOPS_9      nop; INSERT_NOPS_8
#define INSERT_NOPS_10     nop; INSERT_NOPS_9

.globl main
main:

li t3, 0xEEEEEEEEE
li t4, 0x1

REPEAT:
    srl t0, t3, t4
    INSERT_NOPS_7
    sra t1, t3, t4
    INSERT_NOPS_7
    sll t2, t3, t4
    INSERT_NOPS_6
    beq zero, zero, REPEAT # Repeat the loop

.end
```



The following Verilog fragments show the Shift Unit of SweRV EH1.

```
161    assign ashift[31:0] = a_ff >>> b_ff[4:0];
```

```
163    assign sout[31:0] = ( {32{ap.sll}} & (a_ff[31:0] << b_ff[4:0]) ) |
164                        ( {32{ap.srl}} & (a_ff[31:0] >> b_ff[4:0]) ) |
165                        ( {32{ap.sra}} & ashift[31:0] );
```

```
170    assign sel_shift = |{ap.sll,ap.srl,ap.sra};
```

```
185    assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                      ({32{sel_shift}} & sout[31:0]) |
187                      ({32{sel_adder}} & aout[31:0]) |
188                      ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1],1'b0}) |
189                      ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                      ({31'b0, slt_one});
```

- 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`.

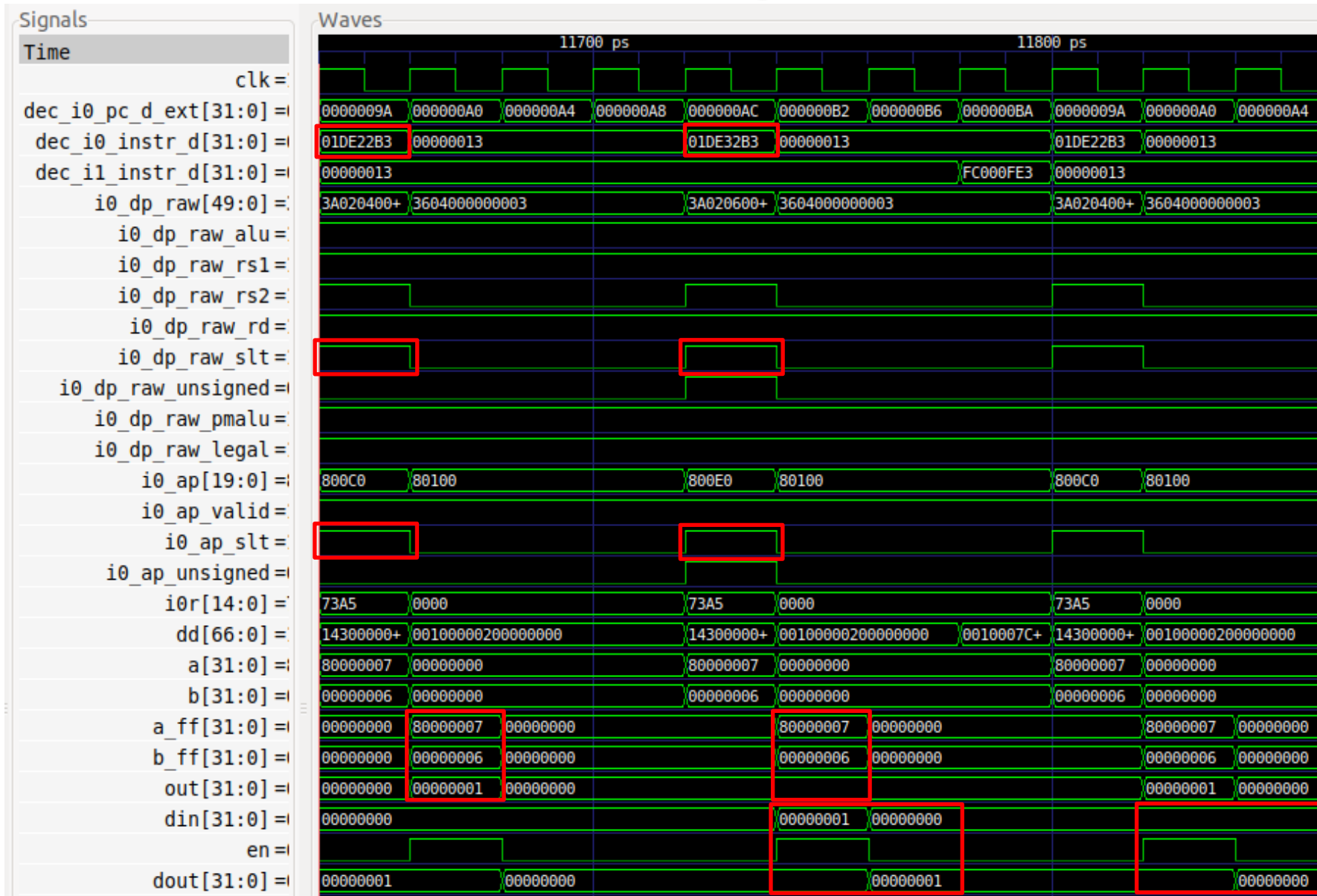
```
#define INSERT_NOPS_0
#define INSERT_NOPS_1    nop; INSERT_NOPS_0
#define INSERT_NOPS_2    nop; INSERT_NOPS_1
#define INSERT_NOPS_3    nop; INSERT_NOPS_2
#define INSERT_NOPS_4    nop; INSERT_NOPS_3
#define INSERT_NOPS_5    nop; INSERT_NOPS_4
#define INSERT_NOPS_6    nop; INSERT_NOPS_5
#define INSERT_NOPS_7    nop; INSERT_NOPS_6
#define INSERT_NOPS_8    nop; INSERT_NOPS_7
#define INSERT_NOPS_9    nop; INSERT_NOPS_8
#define INSERT_NOPS_10   nop; INSERT_NOPS_9

.globl main
main:

li t3, 0x80000007
li t4, 0x6

REPEAT:
    slt  t0, t3, t4
    INSERT_NOPS_7
    sltu t0, t3, t4
    INSERT_NOPS_6
    beq  zero, zero, REPEAT # Repeat the loop

.end
```



The following Verilog fragments show the logic that executes these operations in SweRV EH1.

```

135   assign bm[31:0] = ( ap.sub ) ? ~b_ff[31:0] : b_ff[31:0];
136
137
138   assign {cout, aout[31:0]} = {1'b0, a_ff[31:0]} + {1'b0, bm[31:0]} + {32'b0, ap.sub};
139
140   assign ov = (~a_ff[31] & ~bm[31] & aout[31]) |
141             ( a_ff[31] & bm[31] & ~aout[31] );
142
143   assign neg = aout[31];
144

```

```

177   assign lt = (~ap.unsign & (neg ^ ov)) |
178             ( ap.unsign & ~cout);
179
180   assign ge = ~lt;
181
182
183   assign slt_one = (ap.slt & lt);
184
185   assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                     ({32{sel_shift}} & sout[31:0]) |
187                     ({32{sel_adder}} & aout[31:0]) |
188                     ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1],1'b0}) |
189                     ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                     ({31'b0, slt_one});
191

```



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`, `srl`, `srai`, `slli`, `slti` and `sltiu`.

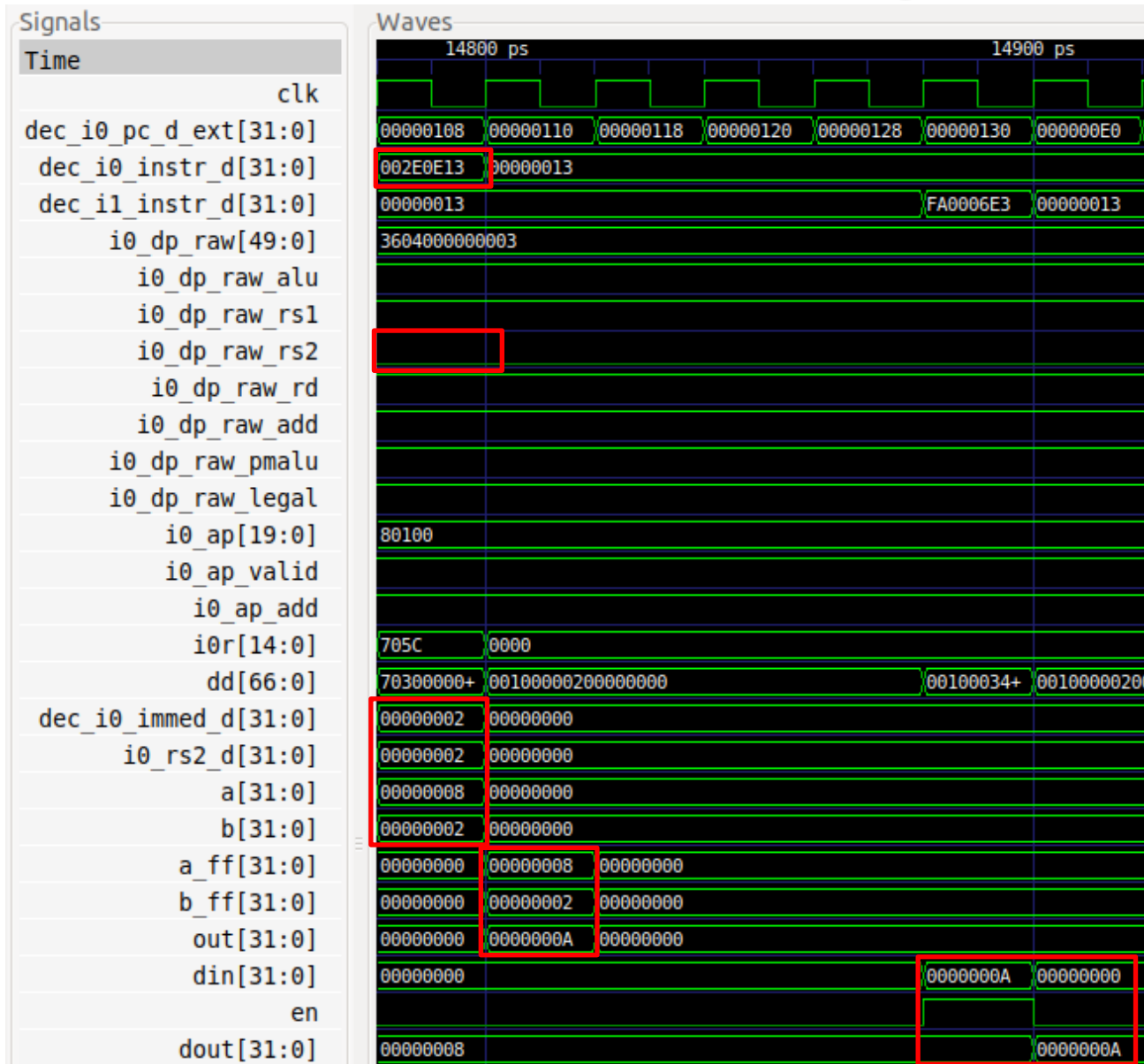
```
#define INSERT_NOPS_0
#define INSERT_NOPS_1      nop; INSERT_NOPS_0
#define INSERT_NOPS_2      nop; INSERT_NOPS_1
#define INSERT_NOPS_3      nop; INSERT_NOPS_2
#define INSERT_NOPS_4      nop; INSERT_NOPS_3
#define INSERT_NOPS_5      nop; INSERT_NOPS_4
#define INSERT_NOPS_6      nop; INSERT_NOPS_5
#define INSERT_NOPS_7      nop; INSERT_NOPS_6
#define INSERT_NOPS_8      nop; INSERT_NOPS_7
#define INSERT_NOPS_9      nop; INSERT_NOPS_8
#define INSERT_NOPS_10     nop; INSERT_NOPS_9

.globl main
main:

li t3, 0x4                # t3 = 4
INSERT_NOPS_1

REPEAT:
    INSERT_NOPS_10
    addi t3, t3, 2          # t3 = t3 + 2
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repeat the loop

.end
```



At module **dec\_decode\_ctl** the 32-bit immediate is computed.

```

1231 // read the csr value through rs2 immid port
1232 assign dec_i0_immed_d[31:0] = ({32{i0_dp.csr_read}} & dec_csr_rddata_d[31:0]) |
1233                               ({32{~i0_dp.csr_read}} & i0_immed_d[31:0]);
1234
1235 // end csr stuff
1236
1237 assign i0_immed_d[31:0] = ({32{i0_dp.imm12}} & { {20{i0[31]}}, i0[31:20] }) | // jalr
1238                               ({32{i0_dp.shimm5}} & {27'b0, i0[24:20]}) |
1239                               ({32{i0_jalimm20}} & { {12{i0[31]}}, i0[19:12], i0[20], i0[30:21], 1'b0 }) |
1240                               ({32{i0_uiimm20}} & {i0[31:12], 12'b0 }) |
1241                               ({32{i0_csr_write_only_d & i0_dp.csr_imm}} & {27'b0, i0[19:15]}); // for csr's that only write csr, dont read csr
1242

```

At module **exu** the proper *rs2* source is selected. In this case, we use *dec\_i0\_immed\_d*.

```

286 assign i0_rs2_d[31:0] = ({32{~dec_i0_rs2_bypass_en_d}} & gpr_i0_rs2_d[31:0]) |
287                               ({32{~dec_i0_rs2_bypass_en_d}} & dec_i0_immed_d[31:0]) |
288                               ({32{ dec_i0_rs2_bypass_en_d}} & i0_rs2_bypass_data_d[31:0]);

```

At module **dec\_gpr\_ctl** the enable signal *rden1* determines if the register file is accessed for the second operand or not. If an instruction uses an immediate operand:  $i0\_dp.rs2=0 \rightarrow rden1=0 \rightarrow rd1[31:0]=0x00000000 \rightarrow gpr\_i0\_rs2\_d[31:0]=0x00000000$ .

```

90 rd1[31:0] |= ({32{rden1 & (raddr1[4:0]== 5'(j)) & (gpr_bank_id[GPR_BANKS_LOG2-1:0] == 1'(i))}} & gpr_out[i][j][31:0]);

```

6) (The following exercise is based on exercise 4.6 of [HePa].)

Figure 5 does not discuss I-type instructions like *addi* or *andi*.

- 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 5 and explain their purpose.
- List the values of the signals generated by the control unit for *addi*.

One of the inputs to the two 3-1 multiplexers at the Decode Stage comes from the immediate in signal `dec_i0_immed_d[31:0]`. The immediate is a 32-bit signal that is computed differently depending on the I-Type instruction that is executed. It is a subset of 32 bits that make up the instruction, which are selected and sign extended as follows:

```

1231 // read the csr value through rs2 immed port
1232 assign dec_i0_immed_d[31:0] = ({32{ i0_dp.csr_read}} & dec_csr_rddata_d[31:0]) |
1233                               ({32{~i0_dp.csr_read}} & i0_immed_d[31:0]);
1234
1235 // end csr stuff
1236
1237 assign i0_immed_d[31:0] = ({32{i0_dp.imm12}} & { {20{i0[31]}},i0[31:20] }) | // jalr
1238                               ({32{i0_dp.shimm5}} & {27'b0, i0[24:20]}) |
1239                               ({32{i0_jalimm20}} & { {12{i0[31]}},i0[19:12],i0[20],i0[30:21],1'b0}) |
1240                               ({32{i0_uiimm20}} & {i0[31:12],12'b0 }) |
1241                               ({32{i0_csr_write_only_d & i0_dp.csr_imm}} & {27'b0,i0[19:15]}); // for csr's that only write csr, dont read csr
1242

```

The values of the control signals for the `addi` can be seen in the simulation from Exercise 5.

7) (The following exercise is based on exercise 4.4 of [HePa] 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.

The structure type is defined in file `swerv_types.sv`:

```

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;
```

- Signal `valid` stuck-at-0: It would not be possible to execute any A-L instruction, as any A-L instruction would be considered invalid.
- Signals `land`, `lor`, `lxor`, `sll`, `srl`, `sra`, `beq`, `bne`, `blt`, `bge`, `add`, `sub`, `slt` and `jal` stuck-at-0: For each of these bits, it would not be possible to execute the corresponding A-L instruction; for example, if `land` is stuck-at-0, it would not be possible to execute an `and` instruction.
- Signal `unsign` stuck-at-0: It would not be possible to communicate to the processor that the operation must be unsigned.
- Signals `predict_t` and `predict_nt`: It would not be possible to communicate to the processor that a branch is predicted taken or not-taken.
- Signals `csr_write` and `csr_imm`: It would not be possible to write or to operate with an immediate in the CSR Register.