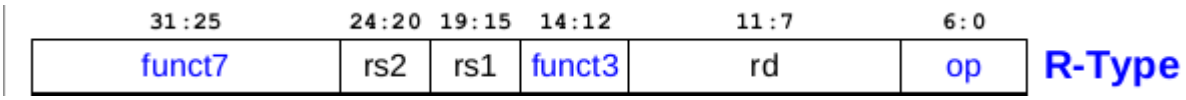
TASK: Verify that these 32 bits (0x01de0e33) correspond to instruction `add t3,t3,t4` in the RISC-V architecture.

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

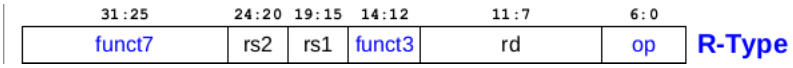
TASK: Locate the main structures and signals from **Error! Reference source not found.** in the Verilog files of the VeeR EL2 processor.

- Register q0ff, q1ff and q2ff in lines 252-254 in module **el2_ifu_aln_ctl**
- Aligner in module **el2_ifu_aln_ctl** and instruction buffer in module **el2_dec_ib_ctl**
- Control Unit in module **el2_dec_decode_ctl**
- Register file:
    - Instantiation in line 461 of module **el2_dec**.
    - Implementation in module **el2_dec_gpr_ctl**.
- 4:1 and 3:1 muxes in Decode stage: Line 246-253 of module **el2_exu**.

- Pipeline Registers for Control Signals: Distributed in several modules.
- Register i_result_ff in line 218 of module **el2_exu_alu_ctl**.
- ALU:
    - Instantiation in line 279 of module **el2_exu**.
    - Implementation in module **el2_exu_alu_ctl**.
- 2:1 mux in X stage in line 320 of module **el2_exu**.
- Register i0_result_r_ff in line 1443 of module **el2_dec_decode_ctl**.

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

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



In module **el2_dec_decode_ctl**:

```
assign i0r.rs1[4:0] = i0[19:15];
assign i0r.rs2[4:0] = i0[24:20];
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 **el2_dec** to module **el2_dec_decode_ctl**. In module **el2_dec_decode_ctl**:

```
assign dec_i0_rs1_en_d   =  i0_dp.rs1 & (i0r.rs1[4:0] != 5'd0);
assign dec_i0_rs2_en_d   =  i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
assign i0_rd_en_d        =  i0_dp.rd  & (i0r.rd[4:0]  != 5'd0);

assign dec_i0_rs1_d[4:0] =  i0r.rs1[4:0];
assign dec_i0_rs2_d[4:0] =  i0r.rs2[4:0];
```

- The register identifiers are provided to the Register File, which is instantiated in module **el2_dec**. In module **el2_ec**:

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

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

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

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

In module el2_exu, the following register propagates this signal from the D Stage to the X Stage (signal *mul_valid_x*):

```
rvdffie #(pt.BHT_GHR_SIZE+2,1)      i_misc_ff          (.*, .clk(clk),                    .din ({ghr_d_ns[pt.BHT_GHR_SIZE-1:0], mul_p.valid, dec_i0_branch_d}),
                                                                                          .dout({ghr_d[pt.BHT_GHR_SIZE-1:0]   , mul_valid_x, i0_branch_x}) );
```

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

Solution not provided for this exercise.

TASK: Find in the Verilog code (module **el2_exu**) the 3:1 multiplexer on the bottom of **Error! Reference source not found.** (second input operand) and try to find the origin of its inputs (in **Error! Reference source not found.** only the input coming from the Register File is shown).

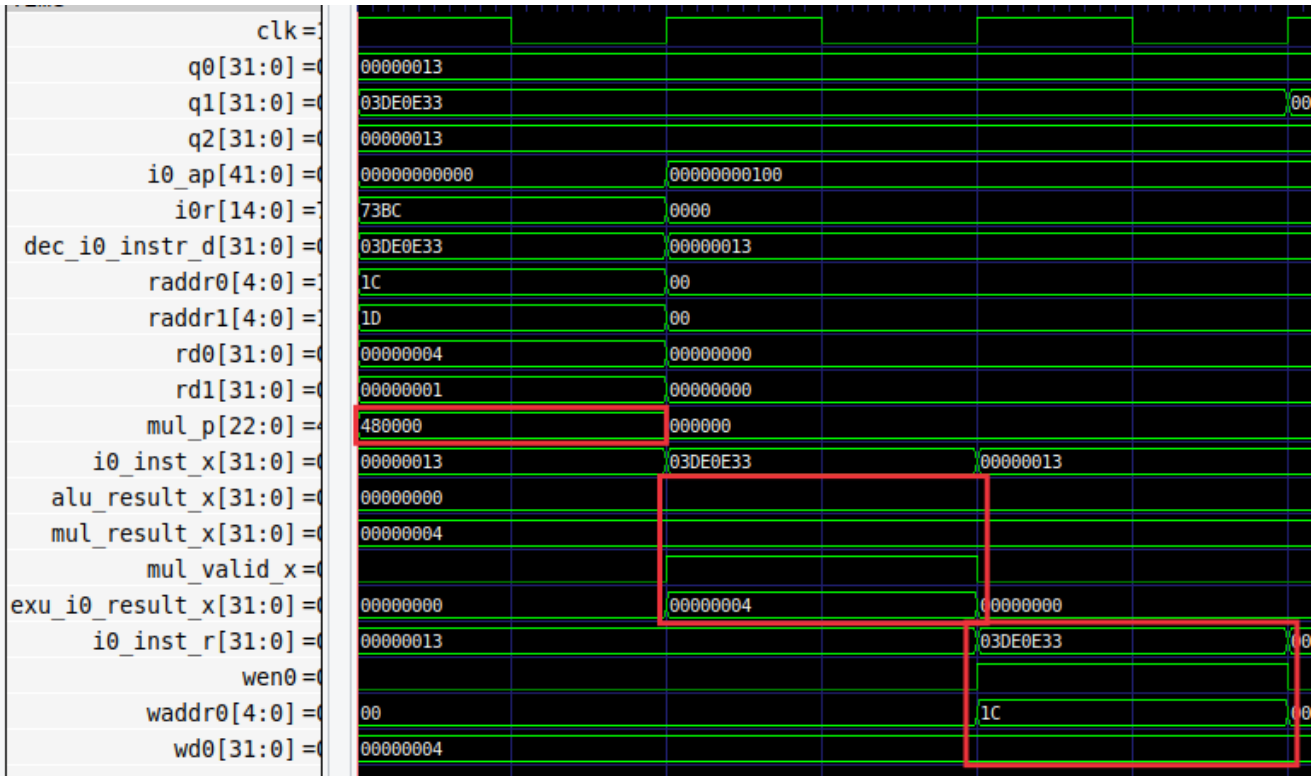
```
assign i0_rs2_d[31:0]          = ({32{~i0_rs2_bypass_en_d & dec_i0_rs2_en_d}}
                                  ({32{~i0_rs2_bypass_en_d                 }}
                                  ({32{ i0_rs2_bypass_en_d                 }}
                                                                                & gpr_i0_rs2_d[31:0]        ) |
                                                                                & dec_i0_immed_d[31:0]      ) |
                                                                                & 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: In the example from **Error! Reference source not found.**, replace the `add` instruction with a non A-L instruction (such as a `mul` instruction) and analyse the control signals.

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



Structure used for mul instructions:

```
typedef struct packed {
                        logic valid;
                        logic rs1_sign;
                        logic rs2_sign;
                        logic low;
                        logic bcompress;
                        logic bdecompress;
                        logic clmul;
                        logic clmulh;
                        logic clmulr;
                        logic grev;
                        logic gorc;
                        logic shfl;
                        logic unshfl;
                        logic crc32_b;
                        logic crc32_h;
                        logic crc32_w;
                        logic crc32c_b;
                        logic crc32c_h;
                        logic crc32c_w;
                        logic bfp;
                        logic xperm_n;
                        logic xperm_b;
                        logic xperm_h;
                        } el2_mul_pkt_t;
```

Analysis:

- D Stage:
  - The two operands are read from the RF: rd0=4 and rd1=1
  - mul_p=0x480000, thus valid=1 (there is a mul instruction) and low=1 (thus only the 32 least significant bits are provided in the result).
- X Stage:

- Signal mul_valid_x=1, thus the result of the multiplier is selected.

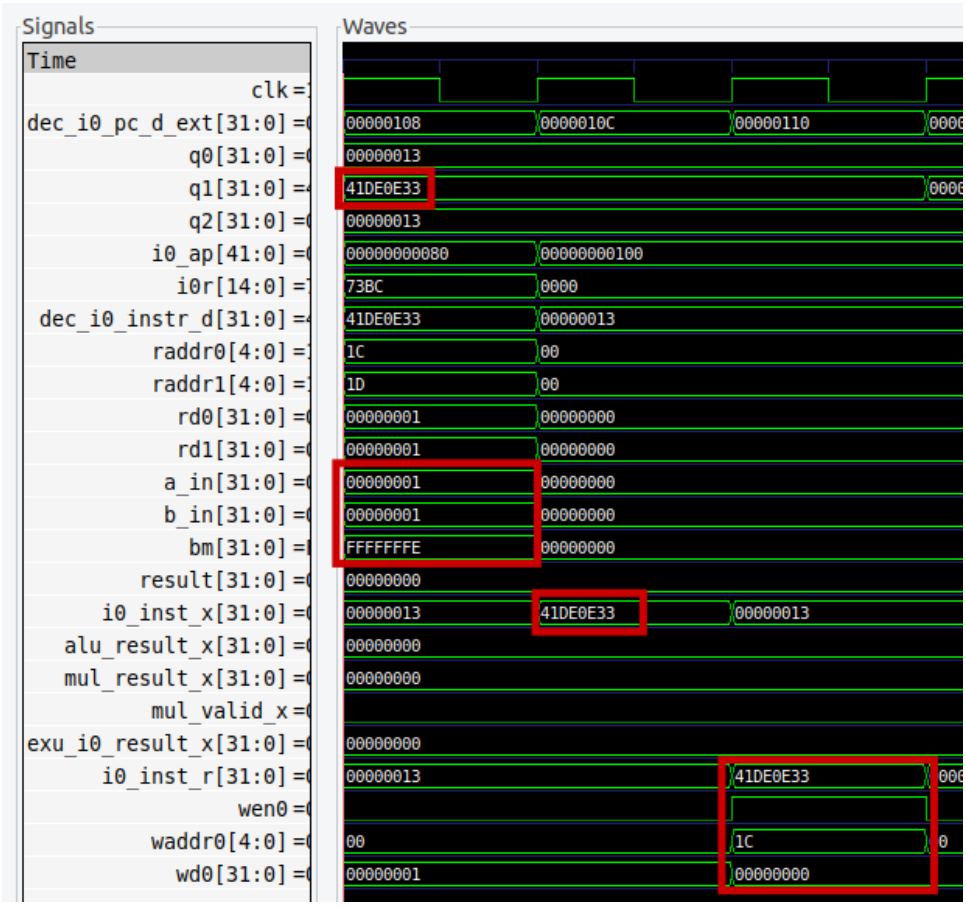
- R Stage:
  - The result is written into the RF.

TASK: Include the new signals analysed in this section in the simulation from **Error! Reference source not found.**.

Solution not provided for this exercise.

TASK: Perform a simulation of a `sub` instruction similar to the one from **Error! Reference source not found.**. You can include new signals in the simulation if it is convenient for your analysis.

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



Structure used for A-L instructions:

```
typedef struct packed {
                    logic clz;
                    logic ctz;
                    logic cpop;
                    logic sext_b;
                    logic sext_h;
                    logic min;
                    logic max;
                    logic pack;
                    logic packu;
                    logic packh;
                    logic rol;
                    logic ror;
                    logic grev;
                    logic gorc;
                    logic zbb;
                    logic bset;
                    logic bclr;
                    logic binv;
                    logic bext;
                    logic sh1add;
                    logic sh2add;
                    logic sh3add;
                    logic zba;
                    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;
                    } el2_alu_pkt_t;
```

Region of code in the ALU module related with the sub instruction, which uses the 2's complement of the second input operand:

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

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

Analysis:

- D Stage:
  - The two operands are read from the RF: rd0=1 and rd1=1
  - The 1's complement is computed for the second operand: bm=0xfffffffe
  - The subtraction is computed in aout as follows: aout=zba_a_in+bm+1=0. Note that a value of 1 is added in order to use the 2's complement of the second operand.
- X Stage:
  - The result of the subtraction is selected and propagated.
- R Stage:
  - The result of the subtraction is written into the RF.

TASK: Analyse the Verilog implementation of the adder/subtractor implemented in module **el2_exu_alu_ctl**. Error! Reference source not found. gives you some help by showing the

logic directly related with addition and subtraction operations. You can use an RVfpga-Trace simulation as a help.

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

logic        [31:0]    bm;

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

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

- The inputs are prepared into signals zba_a_in and bm:
  - Signal zba_a_in: For addition/subtraction the input is left untouched.
  - Signal bm:
    - For addition, the input is left untouched
    - For subtration, it is first 1's complemented (~b_in[31:0]) and then 2's complemented by adding 1 (+ {32'b0, ap.sub})

```
assign sel_shift           =  ap.sll  | ap.srl | ap.sra | ap_rol | ap_ror;
assign sel_adder           = (ap.add  | ap.sub | ap_zba) & ~ap.slt & ~ap_min & ~ap_max;
assign sel_pc              =  ap.jal  | pp_in.pcall | pp_in.pja | pp_in.pret;
assign csr_write_data[31:0]= (ap.csr_imm)  ?  b_in[31:0]  :  a_in[31:0];

assign slt_one             =  ap.slt & lt;


assign result[31:0]        =                          lout[31:0]             |
                            ({32{sel_shift}}     &  sout[31:0]             ) |
                            ({32{sel_adder}}     &  aout[31:0]             ) |
                            ({32{sel_pc}}        & {pcout[31:1],1'b0}       ) |
                            ({32{ap.csr_write}} &  csr_write_data[31:0] ) |
                                                  {31'b0, slt_one}          |
                            ({32{ap_bext}}       & {31'b0, sout[0]}         ) |
                                                  {26'b0, bitmanip_clz_ctz_result[5:0]} |
                                                  {26'b0, bitmanip_cpop_result[5:0]}    |
                                                   bitmanip_sext_result[31:0]    |
                                                   bitmanip_minmax_result[31:0]  |
                                                   bitmanip_pack_result[31:0]    |
                                                   bitmanip_packu_result[31:0]   |
                                                   bitmanip_packh_result[31:0]   |
                                                   bitmanip_rev8_result[31:0]    |
                                                   bitmanip_orc_b_result[31:0]   |
                                                   bitmanip_sb_data[31:0];
```

- Signal sel_adder=1 for add (ap.add) and sub (ap.sub) instructions.
- In that case, aout is selected as the result of the ALU.

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

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

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

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

```
assign dec_i0_waddr_r[4:0]       =  r_d_in.i0rd[4:0];

assign dec_i0_wen_r              =  i0_wen_r    & ~r_d_in.i0div & ~i0_load_kill_wen_r;
```

1. EXERCISES

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

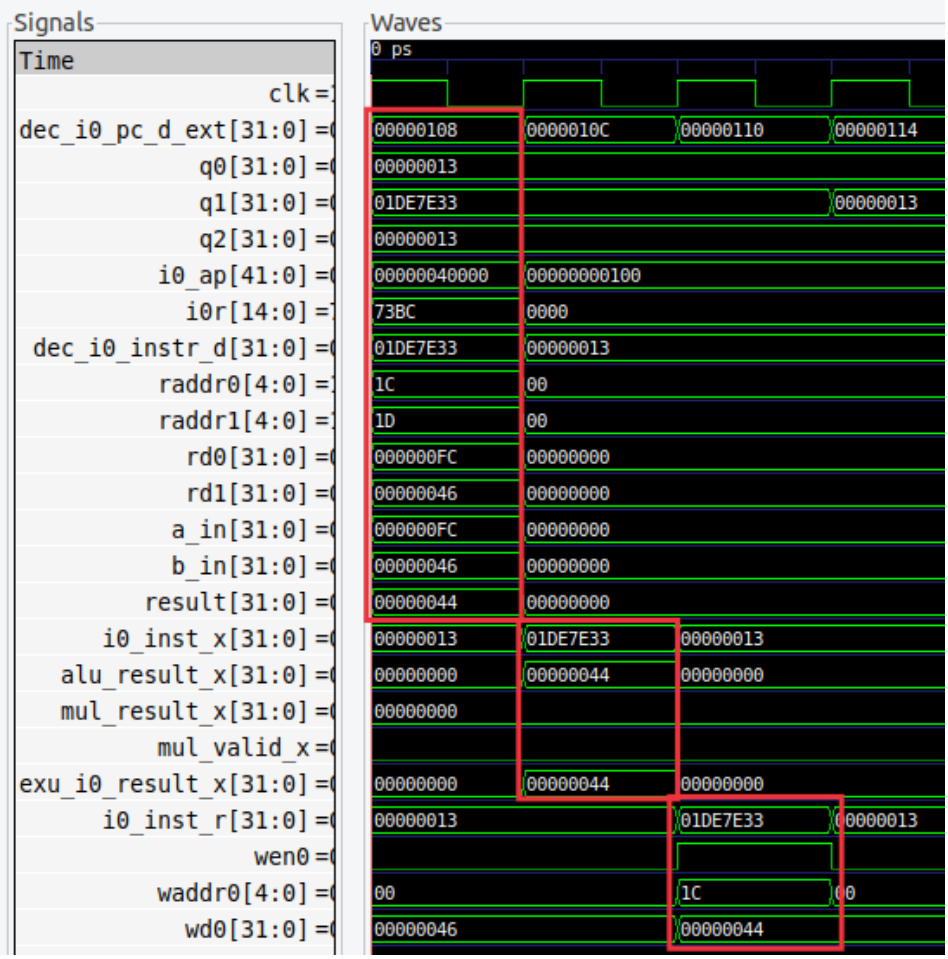
The following example, provided at *Labs/RVfpgaLabsSolutions/Lab12/AND_Instruction/*, illustrates the execution of an `and` instruction contained within a loop that repeats forever. As in the example for the `add` instruction, the `and` instruction (highlighted in red) is surrounded by several `nop` 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, 0x7
   beq  zero, zero, REPEAT # Repeat the loop

.end
```

If you open the project in Catapult, build it, and open the disassembly file, 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).

- **1st cycle – D 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]):

  `0000000 | 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**.

- Signal `i0_ap` contains a single 1 in bit *land* of the structure.
- Signal `i0r` contains the source and destination registers of the operation.

The **Register File is read** in this stage. Signals `a_in` and `b_in` 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).

The `and` instruction is also **executed** in this stage. Signal `result` contains the result of the and operation: 11111100 (0xFC) & 01000101 (0x46) = 01000100 (0x44).

- **2nd cycle – X Stage**: The result of the and operation is propagated to this stage. It is selected in the 2:1 multiplexer and propagated to the final stage.

- **3rd cycle – R Stage**: The result of the and 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 see the implementation of the Logic Unit, and specifically the AND operation, in file *el2_exu_alu_ctl.sv*:

```
assign lout[31:0]          =  ( {32{csr_ren_in       }} &  csr_rddata_in[31:0]       ) |
                              ( {32{ap.land & ~ap_zbb}} &  a_in[31:0] &  b_in[31:0]   ) |
                              ( {32{ap.lor  & ~ap_zbb}} & (a_in[31:0] |  b_in[31:0]) ) |
                              ( {32{ap.lxor & ~ap_zbb}} & (a_in[31:0] ^  b_in[31:0]) ) |
                              ( {32{ap.land &  ap_zbb}} &  a_in[31:0] & ~b_in[31:0]   ) |
                              ( {32{ap.lor  &  ap_zbb}} & (a_in[31:0] | ~b_in[31:0]) ) |
                              ( {32{ap.lxor &  ap_zbb}} & (a_in[31:0] ^ ~b_in[31:0]) );
```

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

Solution not provided.

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

The following example, provided at *Labs/RVfpgaLabsSolutions/Lab12/SrlSraSll_Instruction/*, illustrates the execution of the three shift instructions.

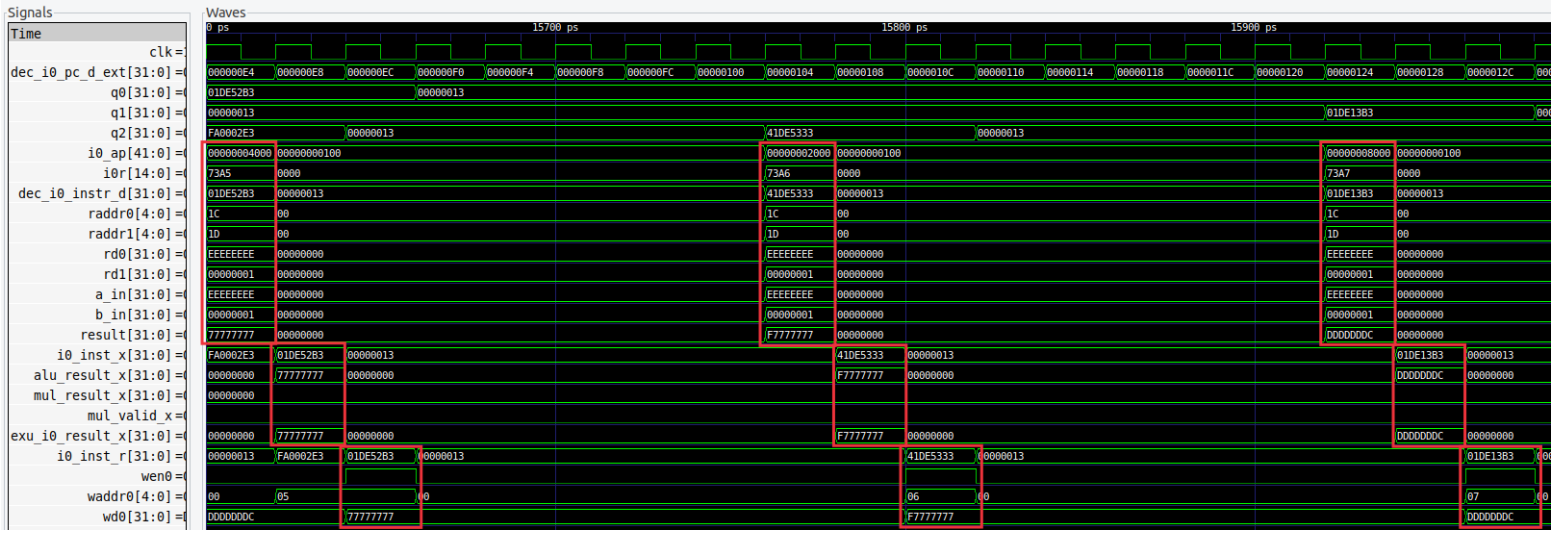
```
#define INSERT_NOPS_1    nop;
#define INSERT_NOPS_2    nop; INSERT_NOPS_1
#define INSERT_NOPS_3    nop; INSERT_NOPS_2
#define INSERT_NOPS_4    nop; INSERT_NOPS_3
#define INSERT_NOPS_5    nop; INSERT_NOPS_4
```

```
#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, 0xEEEEEEEE
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
```

If you open the project in Catapult, build it, and open the disassembly file you will see the three instructions:

```
0x000000e4:      01de52b3              srl   t0,t3,t4
…
0x00000104:      41de5333              sra   t1,t3,t4
…
0x00000124:      01de13b3              sll   t2,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.



This are the Verilog regions of file *el2_exu_alu_ctl.sv* where the shift operations are performed:

```
// * * * * * * * * * * * * * * * * * * * *  BitManip  :  ROL,ROR      * * * * * * * * * * * * * * * * * * * *
// * * * * * * * * * * * * * * * * * * * *  BitManip  :  ZBEXT        * * * * * * * * * * * * * * * * * * * *

logic        [5:0]     shift_amount;
logic        [31:0]    shift_mask;
logic        [62:0]    shift_extend;
logic        [62:0]    shift_long;


assign shift_amount[5:0]            = ( { 6{ap.sll}}   & (6'd32 - {1'b0,b_in[4:0]}) ) |   // [5] unused
                                      ( { 6{ap.srl}}   &          {1'b0,b_in[4:0]}  ) |
                                      ( { 6{ap.sra}}   &          {1'b0,b_in[4:0]}  ) |
                                      ( { 6{ap_rol}}   & (6'd32 - {1'b0,b_in[4:0]}) ) |
                                      ( { 6{ap_ror}}   &          {1'b0,b_in[4:0]}  ) |
                                      ( { 6{ap_bext}}  &          {1'b0,b_in[4:0]}  );


assign shift_mask[31:0]             = ( 32'hffffffff << ({5{ap.sll}} & b_in[4:0]) );


assign shift_extend[31:0]           =  a_in[31:0];

assign shift_extend[62:32]          = ( {31{ap.sra}} & {31{a_in[31]}} ) |
                                      ( {31{ap.sll}} &     a_in[30:0] ) |
                                      ( {31{ap_rol}} &     a_in[30:0] ) |
                                      ( {31{ap_ror}} &     a_in[30:0] );


assign shift_long[62:0]    = ( shift_extend[62:0] >> shift_amount[4:0] );   // 62-32 unused

assign sout[31:0]          =   shift_long[31:0] & shift_mask[31:0];
```

```
assign sel_shift           =  ap.sll  | ap.srl | ap.sra | ap_rol | ap_ror;
assign sel_adder           = (ap.add  | ap.sub | ap_zba) & ~ap.slt & ~ap_min & ~ap_max;
assign sel_pc              =  ap.jal  | pp_in.pcall | pp_in.pja | pp_in.pret;
assign csr_write_data[31:0]= (ap.csr_imm)  ?  b_in[31:0]  :  a_in[31:0];

assign slt_one             =  ap.slt & lt;
```

```
assign result[31:0]        =                      lout[31:0]           |
                            ({32{sel_shift}}    &  sout[31:0]         ) |
                            ({32{sel_adder}}    &  aout[31:0]         ) |
                            ({32{sel_pc}}       & {pcout[31:1],1'b0}  ) |
                            ({32{ap.csr_write}} &  csr_write_data[31:0] ) |
                                                  {31'b0, slt_one}      |
                            ({32{ap_bext}}      & {31'b0, sout[0]}     ) |
                                                  {26'b0, bitmanip_clz_ctz_result[5:0]} |
                                                  {26'b0, bitmanip_cpop_result[5:0]}    |
                                                   bitmanip_sext_result[31:0]    |
                                                   bitmanip_minmax_result[31:0]  |
                                                   bitmanip_pack_result[31:0]    |
                                                   bitmanip_packu_result[31:0]   |
                                                   bitmanip_packh_result[31:0]   |
                                                   bitmanip_rev8_result[31:0]    |
                                                   bitmanip_orc_b_result[31:0]   |
                                                   bitmanip_sb_data[31:0];
```

4) Analyse, both in an RVfpga-Trace simulation and directly in the Verilog code, the *set less than* instructions available in the RV32I Base Integer Instruction Set: `slt` and `sltu`.

Solution not provided.

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

The following example, provided at *Labs/RVfpgaLabsSolutions/Lab12/ADDI_Instruction/*, illustrates the execution of the add instruction.

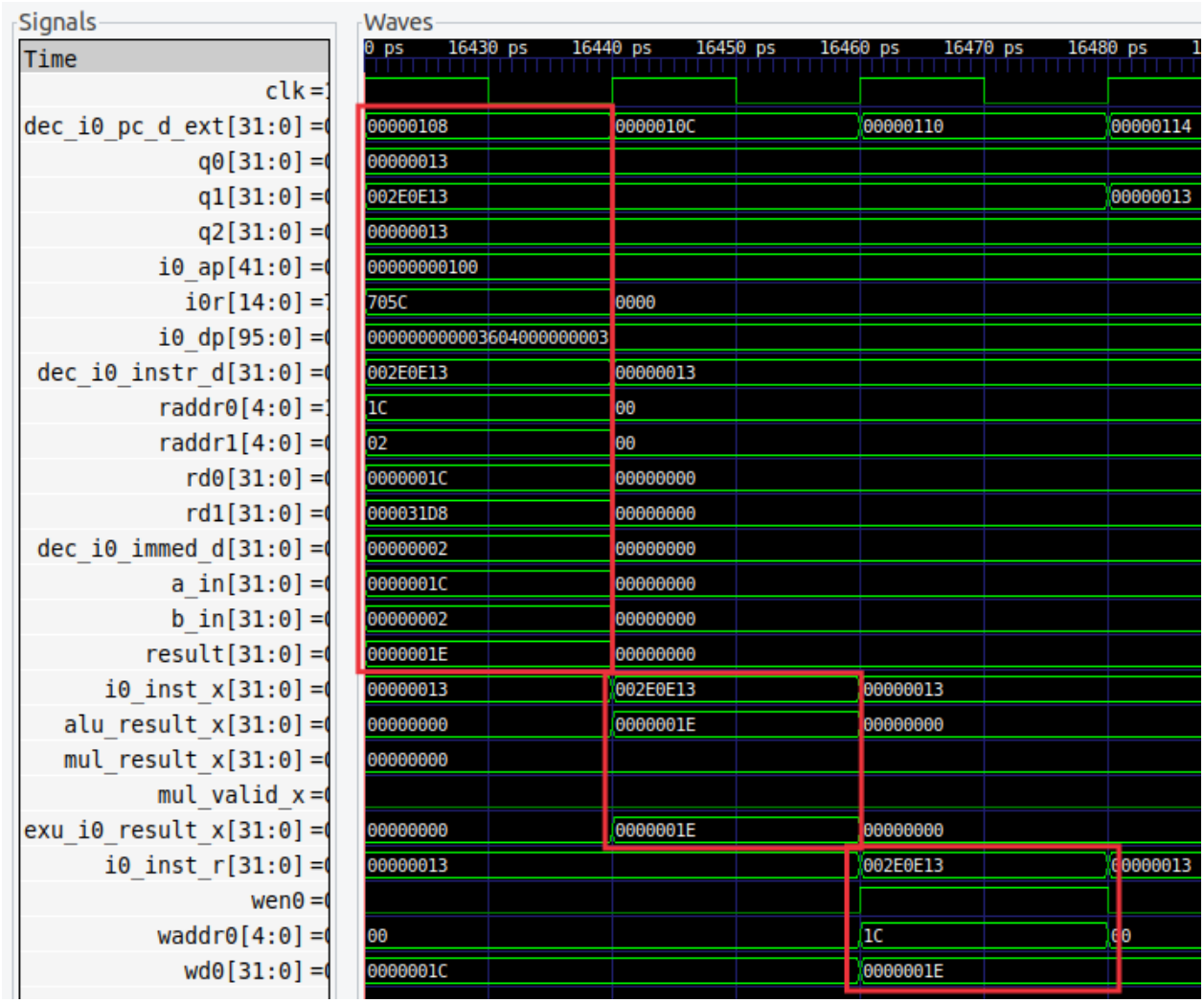
```
#define INSERT_NOPS_1    nop;
#define INSERT_NOPS_2    nop; INSERT_NOPS_1
```

```
#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, 0x4                 # t3 = 4
INSERT_NOPS_1

REPEAT:
   INSERT_NOPS_10
   addi t3, t3, 2          # t3 = t3 + t4
   INSERT_NOPS_10
   beq  zero, zero, REPEAT # Repeat the loop

.end
```

If you open the project in Catapult, build it, and open the disassembly file you will see the three instructions:

```
0x00000108:     002e0e13              addi  t3,t3,2
```

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.



- You can see that the second operand is selected from the immediate (dec_i0_immed_d) and not from the RF.
- In the control signal i0_ap, the bit for add is set.
- In the control signal i0_dp, the **imm12** bit is set and instead the **rs2** bit is not set.

The immediate is generated in file *el2_dec_decode_ctl.sv*:

```
assign dec_i0_immed_d[31:0] =  i0_immed_d[31:0];

assign      i0_immed_d[31:0] = ({32{i0_dp.imm12}}                    & { {20{i0[31]}},i0[31:20] }) |  // jalr
                               ({32{i0_dp.shimm5}}                   & {  27'b0,      i0[24:20] }) |
                               ({32{i0_jalimm20}}                    & { {12{i0[31]}},i0[19:12],i0[20],i0[30:21],1'b0}) |
                               ({32{i0_uiimm20}}                     & { i0[31:12],12'b0 }) |
                               ({32{i0_csr_write_only_d & i0_dp.csr_imm}} & {  27'b0,      i0[19:15]});  // for csr's that only w
```

The multiplexer selects the second operand as follows:

```
assign i0_rs2_d[31:0]          = ({32{~i0_rs2_bypass_en_d & dec_i0_rs2_en_d}}
                                  ({32{~i0_rs2_bypass_en_d                }}
                                  ({32{ i0_rs2_bypass_en_d                }}
                                                                                & gpr_i0_rs2_d[31:0]       ) |
                                                                                & dec_i0_immed_d[31:0]     ) |
                                                                                & i0_rs2_bypass_data_d[31:0]);
```

Signal dec_i0_rs2_en_d is not set for immediate instructions:

```
assign dec_i0_rs2_en_d    =  i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
```

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

One of the inputs to the 3-1 multiplexer for the second input operand comes from the immediate in signal `dec_i0_immed_d[31:0]`:

```
assign i0_rs2_d[31:0]          = ({32{~i0_rs2_bypass_en_d & dec_i0_rs2_en_d}}
                                  ({32{~i0_rs2_bypass_en_d                 }}
                                  ({32{ i0_rs2_bypass_en_d                 }}
                                                                                & gpr_i0_rs2_d[31:0]        ) |
                                                                                & dec_i0_immed_d[31:0]      ) |
                                                                                & i0_rs2_bypass_data_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:

```
assign dec_i0_immed_d[31:0] =  i0_immed_d[31:0];

assign     i0_immed_d[31:0] = ({32{i0_dp.imm12}}                    & { {20{i0[31]}},i0[31:20] }) |  // jalr
                              ({32{i0_dp.shimm5}}                   & {  27'b0,      i0[24:20] }) |
                              ({32{i0_jalimm20}}                    & { {12{i0[31]}},i0[19:12],i0[20],i0[30:21],1'b0}) |
                              ({32{i0_uiimm20}}                     & { i0[31:12],12'b0 }) |
                              ({32{i0_csr_write_only_d & i0_dp.csr_imm}} & {  27'b0,      i0[19:15]});  // for csr's that only w
```

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

Solution not provided.