# 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)**

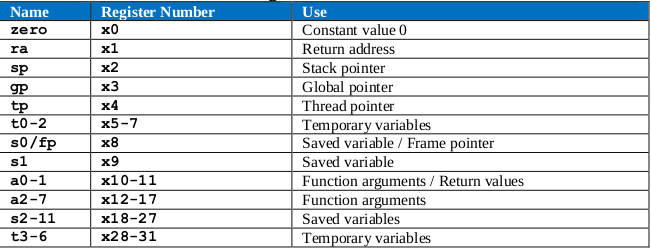
**op = 0110011**

From Appendix B of DDCARV:









**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:
  + Instantiation in line 525 of module **dec**.
  + 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:
  + Instantiation in line 401 of module **exu**.
  + 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:

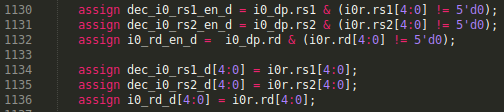


In module **dec\_decode\_ctl**:

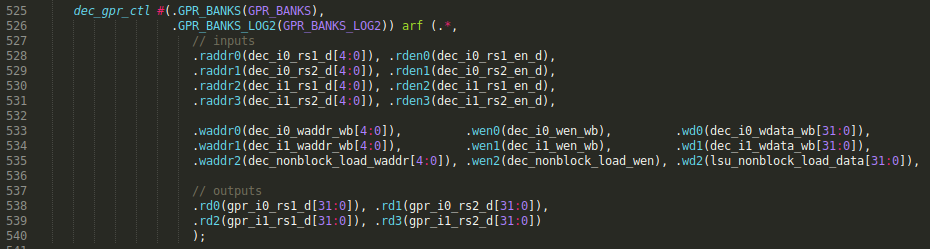


* The register identifiers and read enable signals are assigned to dec\_i0\_rs1\_d/dec\_i0\_rs2\_d and dec\_i0\_rs1\_en\_d/ dec\_i0\_rs2\_en\_d.

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

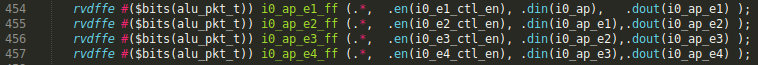


* The register identifiers and read enable signals are provided to the Register File, which is instantiated in module **dec**. In module **dec**:



**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**:



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**:











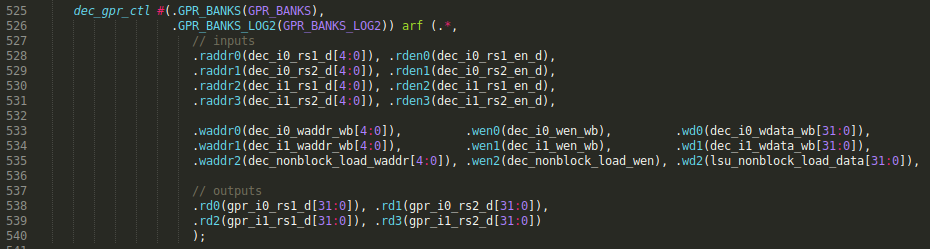
Note that the output of each register is slightly modified (and thus renamed) before going into the next register. You can look at the Verilog code if you want to check the details.

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



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:





**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.



These 3:1 muxes receive 3 inputs:

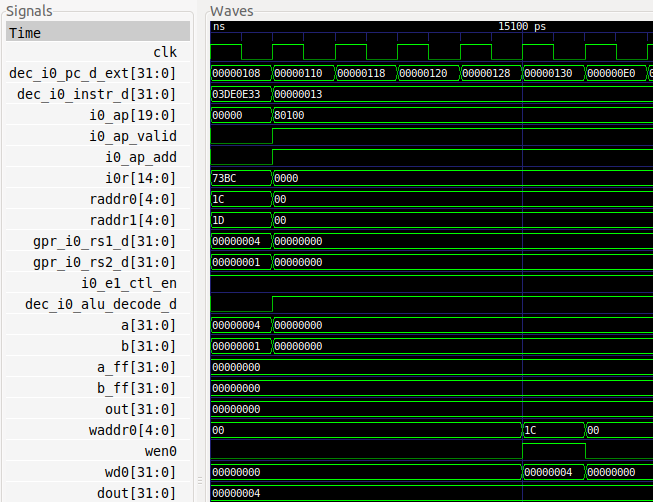
* One from the register file (gpr\_i0\_rs2\_d)
* One from the 32-bit instruction register, which constitutes the immediate (dec\_i0\_immed\_d)
* One from the bypass logic, that we analyse in Lab 15 (i0\_rs2\_bypass\_data\_d)

**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:

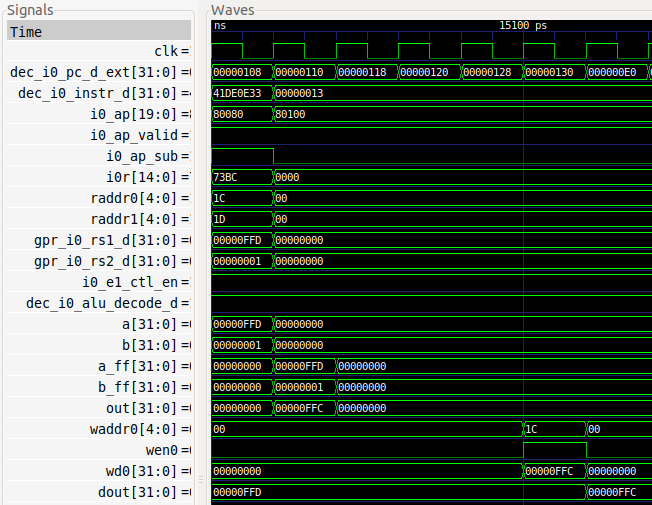


**TASK:** Include the new signals analysed in this section in the simulation from Figure 7.

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.

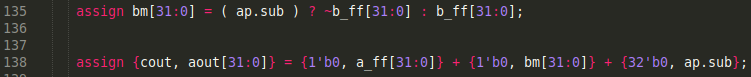
For example, the simulation of sub t3, t3, t4 (0x41de0e33) provides the following results:



**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.



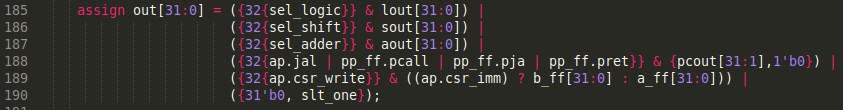
The input operands are propagated from the Decode Stage (a and b) to the Execution Stage (a\_ff and b\_ff).



This is the adder/subtractor.

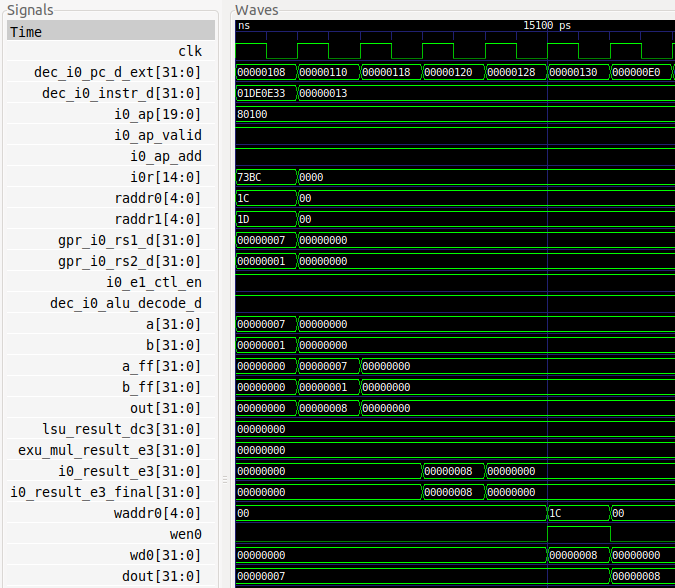
* If the instruction is an addition, aout = a\_ff + b\_ff
* If the instruction is a subtraction, b\_ff is first two’s complemented and then a\_out is computed.



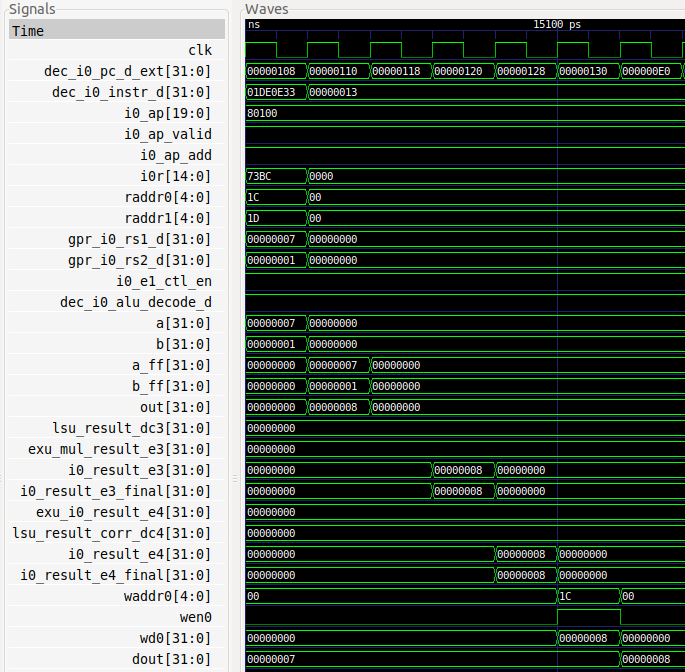


If the instruction is an addition or a subtraction, then out = aout.

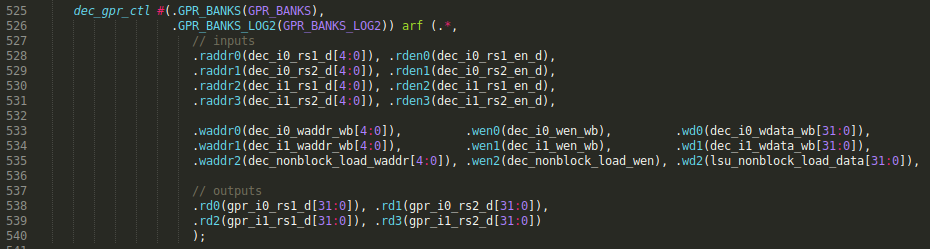
**TASK:** Verify in the simulation that this multiplexer selects the result from the expected Pipe for the add instruction, for the example from Figure 2.



**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.









# EXERCISES

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

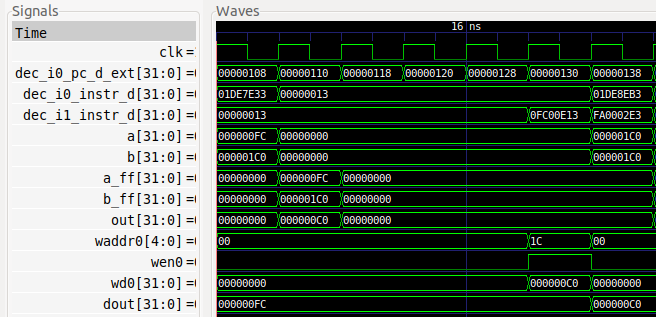
The following example, provided at *[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\_Solutions/Lab12/AND\_Instruction*, illustrates the execution of an and instruction contained within a loop that repeats forever. As in the example for the add instruction, the and instruction (highlighted in red) 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]):

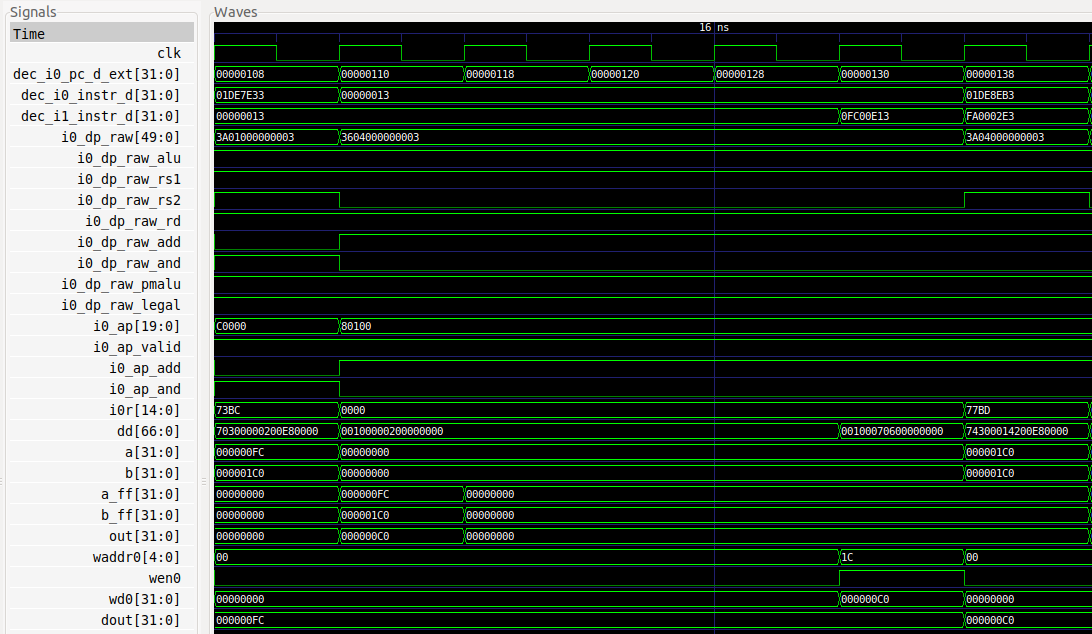
0000000 | rs2 | rs1 | 111 | rd | 0110011

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** (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).

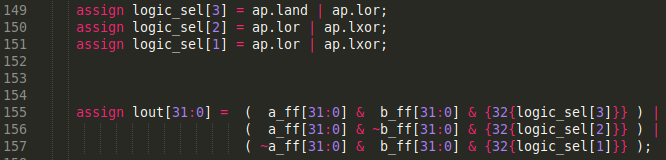
We next add the control signals to the previous simulation:



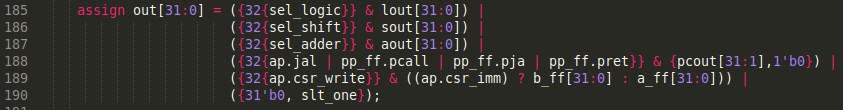
You can see that the control bit for the and instruction is 1 in the first cycle.

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









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

logic\_sel[3]=1 and logic\_sel[2]=logic\_sel[1]=0 🡪 lout = a\_ff & b\_ff

1. (*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

* 1. What are the values of control signals generated by SweRV EH1 for this instruction?
  2. Which resources (blocks) perform a useful function for this instruction?
  3. Which resources (blocks) produce no output for this instruction? Which resources produce output that is not used?

Solution not provided.

1. 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, 0xEEEEEEEE

li t4, 0x1

REPEAT:

srl t0, t3, t4

INSERT\_NOPS\_7

sra t1, t3, t4

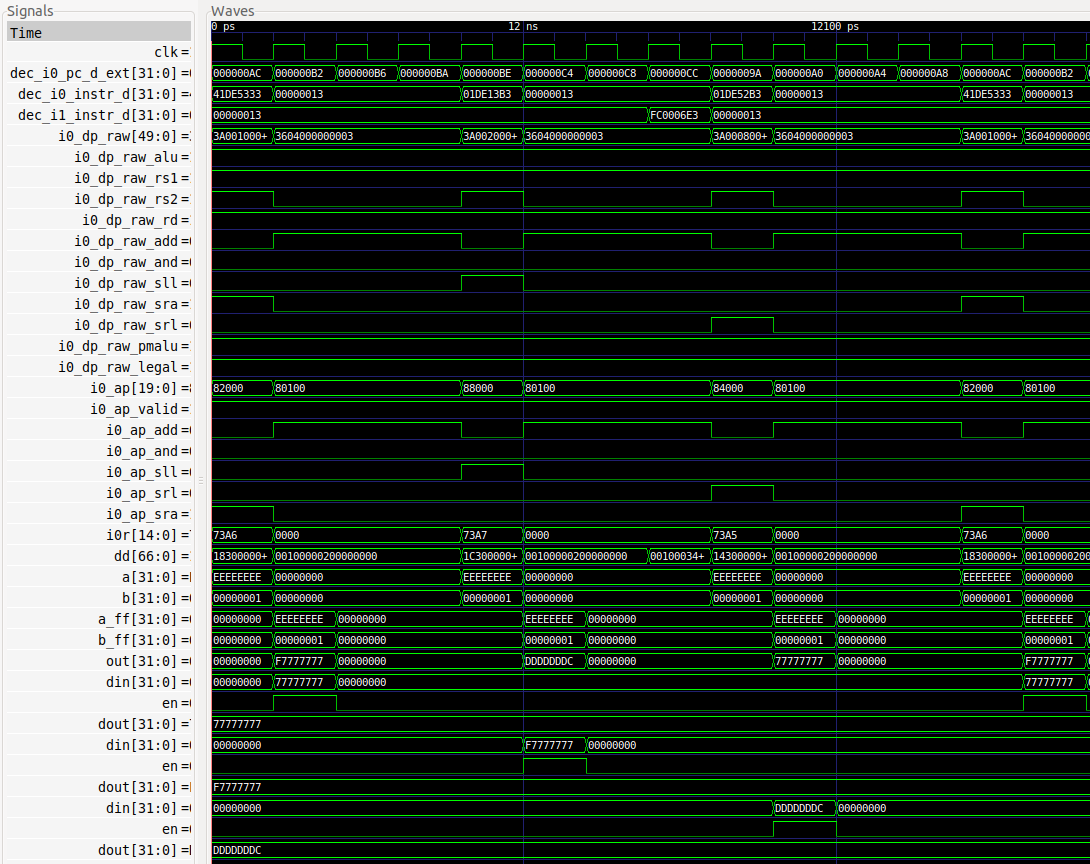
INSERT\_NOPS\_7

sll t2, t3, t4

INSERT\_NOPS\_6

beq zero, zero, REPEAT # Repeat the loop

.end

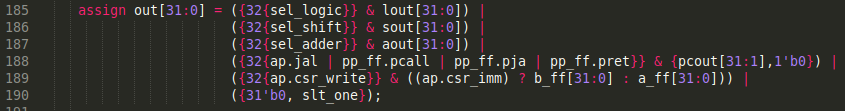


The following Verilog fragments show the Shift Unit of SweRV EH1.









1. 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

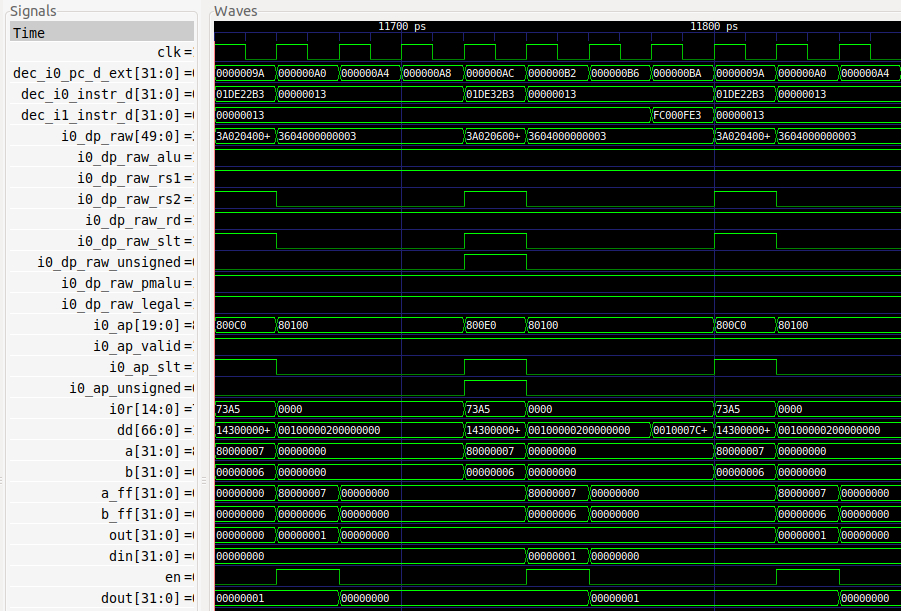
INSERT\_NOPS\_7

sltu t0, t3, t4

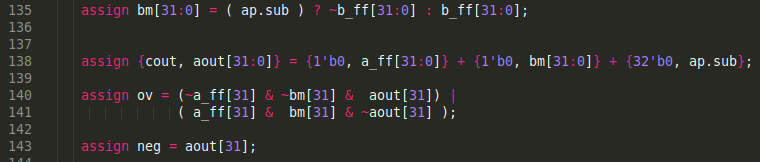
INSERT\_NOPS\_6

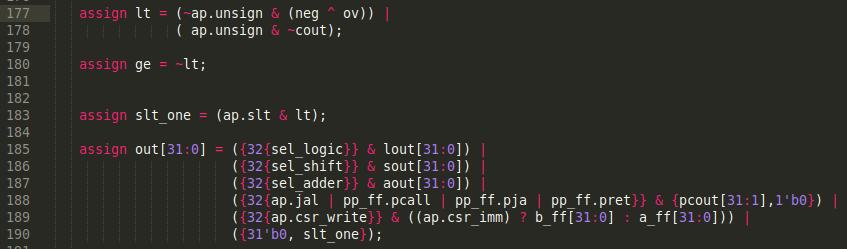
beq zero, zero, REPEAT # Repeat the loop

.end



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





1. 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.

#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:

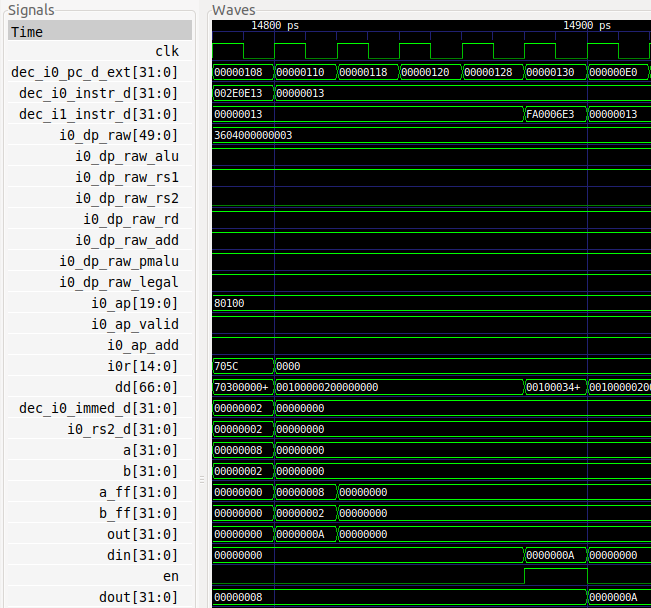
INSERT\_NOPS\_10

addi t3, t3, 2 # t3 = t3 + t4

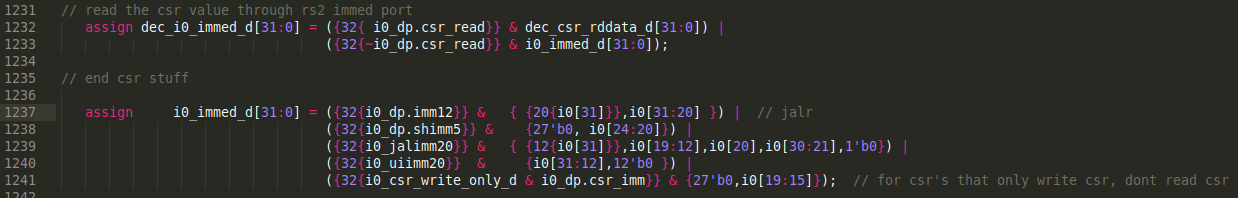
INSERT\_NOPS\_10

beq zero, zero, REPEAT # Repeat the loop

.end



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



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



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 🡪 rden1=0 🡪 rd1[31:0]=0x00000000 🡪 gpr\_i0\_rs2\_d[31:0]=0x00000000.

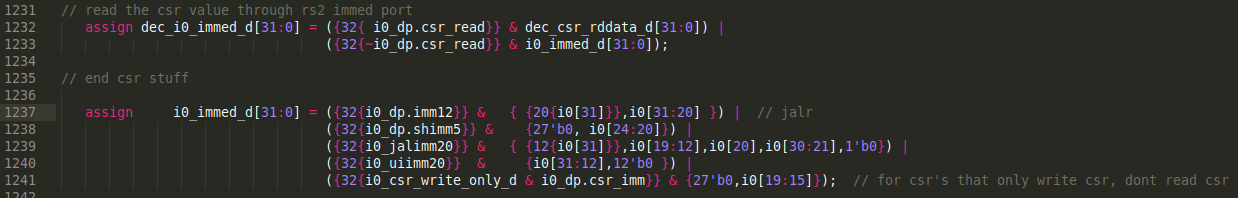


1. (*The following exercise is based on exercise 4.6 of [HePa].*)

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

* 1. 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.
  2. 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:



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

1. (*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.