**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)**

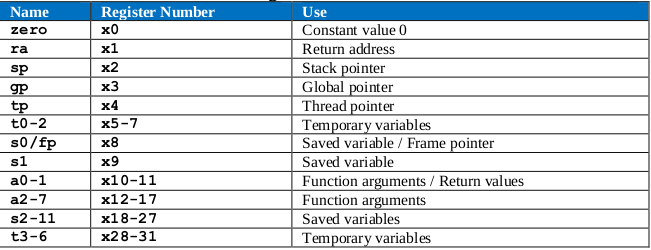
**op = 0110011**

From Appendix B of DDCARV:









**TASK:** Locate the main structures and signals from Figure 6 in the Verilog files of the VeeR EL2 processor.

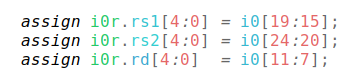
* Register q0ff, q1ff and q2ff in lines 252-254 in module **el2\_ifu\_aln\_ctl**
* Aligner in module **el2\_ifu\_aln\_ctl** and instruction buffer in module **el2\_dec\_ib\_ctl**
* Control Unit in module **el2\_dec\_decode\_ctl**
* Register file:
  + Instantiation in line 461 of module **el2\_dec**.
  + 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.

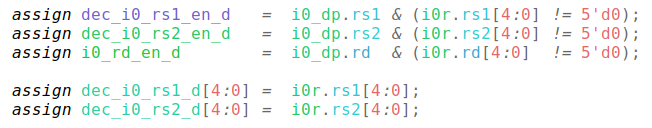
* The register identifiers are obtained from the 32-bit instruction: signal i0[31:0] = dec\_i0\_instr\_d[31:0]. In an R-Type instruction they are located in the following fields:



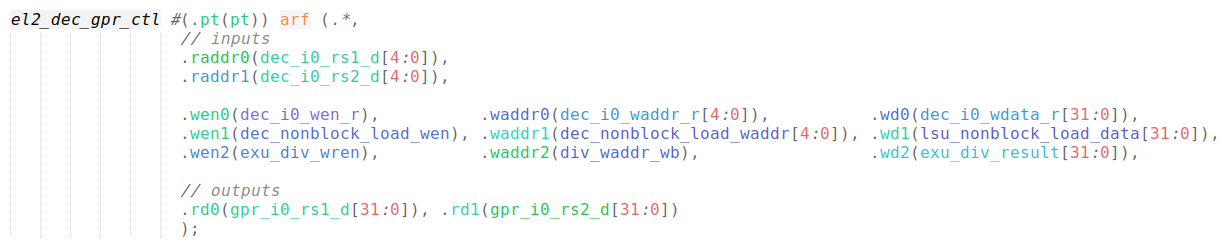
In module **el2\_dec\_decode\_ctl**:



* 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**:



* The register identifiers are provided to the Register File, which is instantiated in module **el2\_dec**. In module **el2\_ec**:



**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*):



**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 Figure 6 (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).

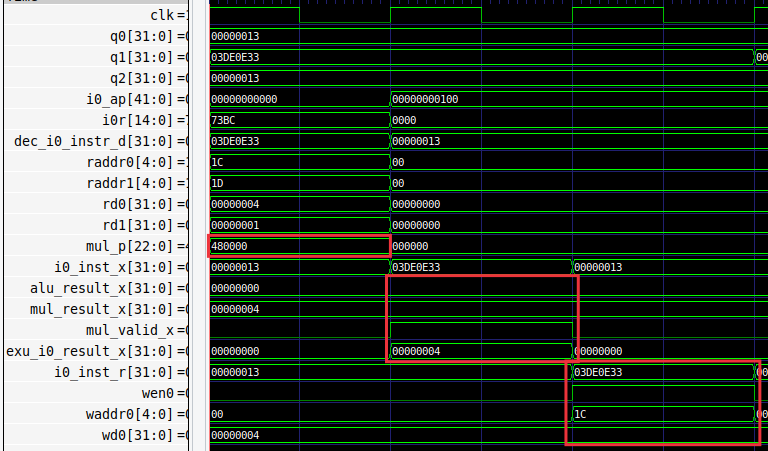


These 3:1 muxes receive 3 inputs:

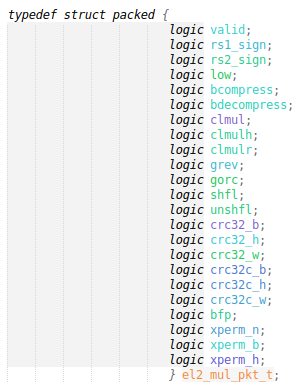
* One from the register file (gpr\_i0\_rs2\_d)
* One from the 32-bit instruction register, which constitutes the immediate (dec\_i0\_immed\_d)
* One from the bypass logic, that we analyse in Lab 15 (i0\_rs2\_bypass\_data\_d)

**TASK:** In the example from Figure 2, 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:



Analysis:

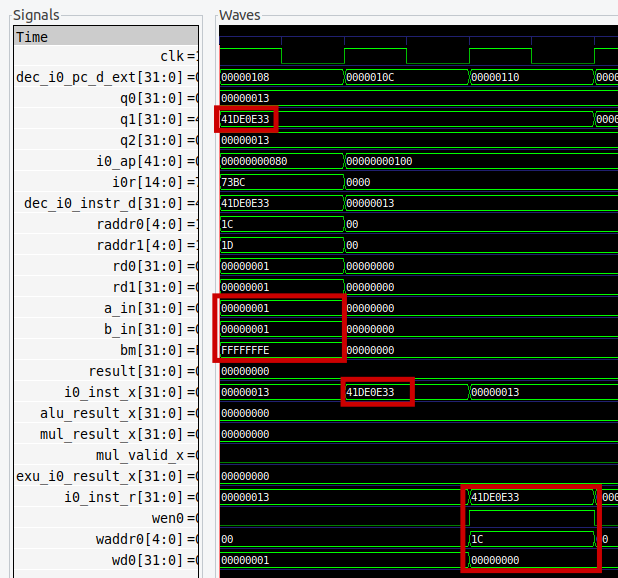
* D Stage:
  + The two operands are read from the RF: rd0=4 and rd1=1
  + mul\_p=0x480000, thus valid=1 (there is a mul instruction) and low=1 (thus only the 32 least significant bits are provided in the result).
* X Stage:
  + Signal mul\_valid\_x=1, thus the result of the multiplier is selected.
* R Stage:
  + The result is written into the RF.

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

Solution not provided for this exercise.

**TASK:** Perform a simulation of a sub instruction similar to the one from Figure 7. 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:



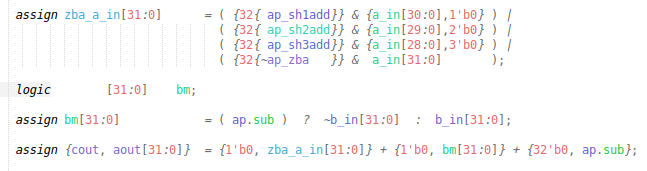
Region of code in the ALU module related with the sub instruction, which uses the 2’s complement of the second input operand:



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**. Figure 9 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.

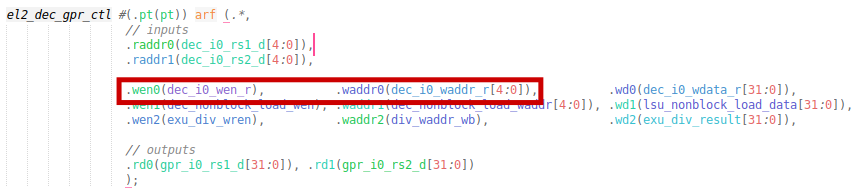


* The inputs are prepared into signals zba\_a\_in and bm:
  + Signal zba\_a\_in: For addition/subtraction the input is left untouched.
  + Signal bm:
    - For addition, the input is left untouched
    - For subtration, it is first 1’s complemented (~b\_in[31:0]) and then 2’s complemented by adding 1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



* 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.







# 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

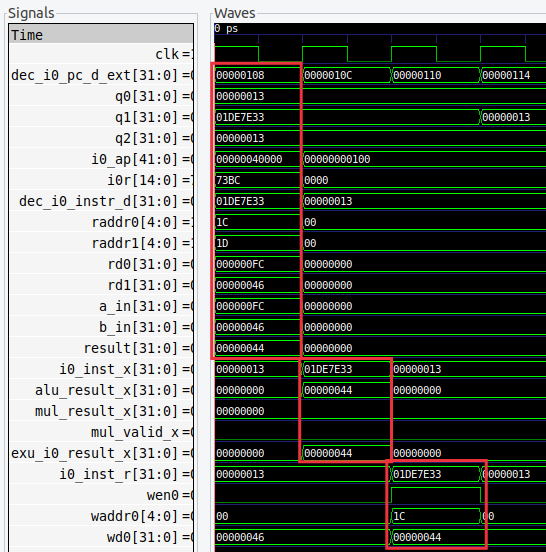
beq zero, zero, REPEAT # Repeat the loop

.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**.

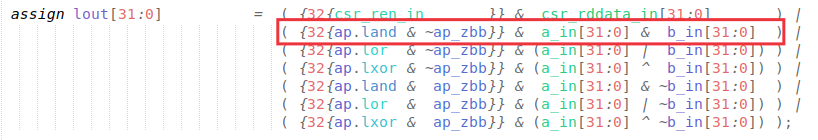
* + Signal i0\_ap contains a single 1 in bit *land* of the structure.
  + Signal i0r contains the source and destination registers of the operation.

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*:



1. (*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

* 1. What are the values of control signals generated by VeeR EL2 for this instruction?
  2. Which resources (blocks) perform a useful function for this instruction?
  3. Which resources (blocks) produce no output or output that is not used for this instruction?

Solution not provided.

1. 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**

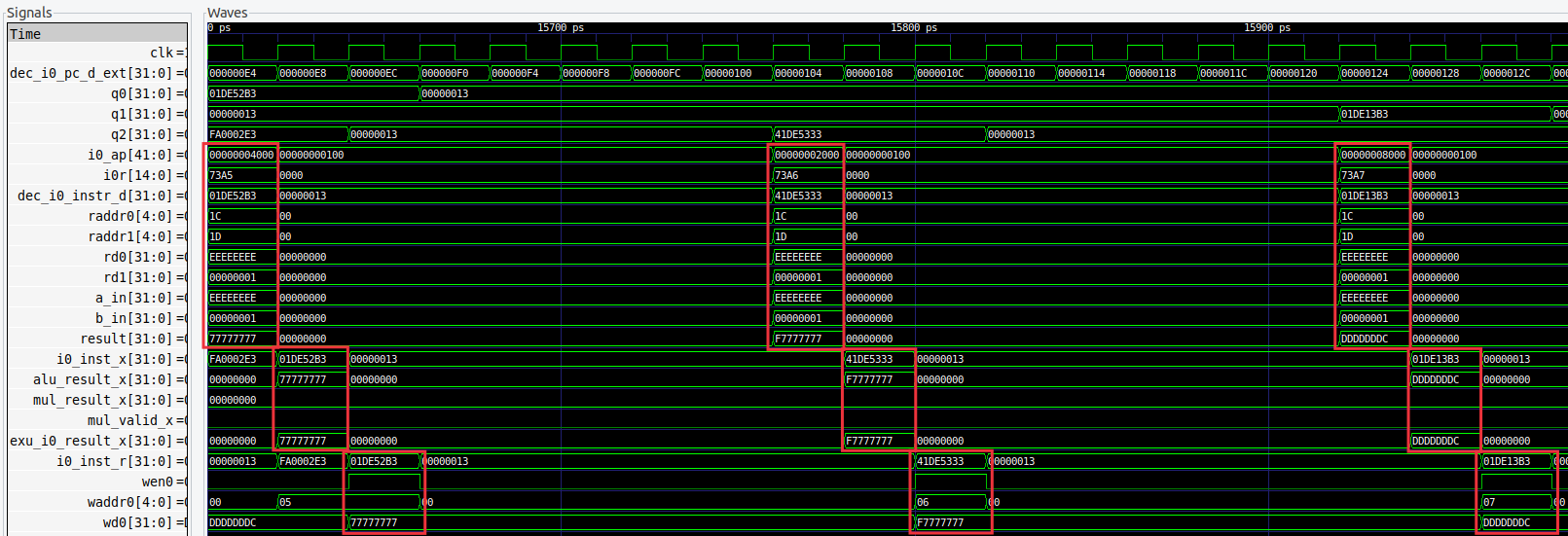
**…**

**0x00000104:** **41de5333**  **sra** **t1,t3,t4**

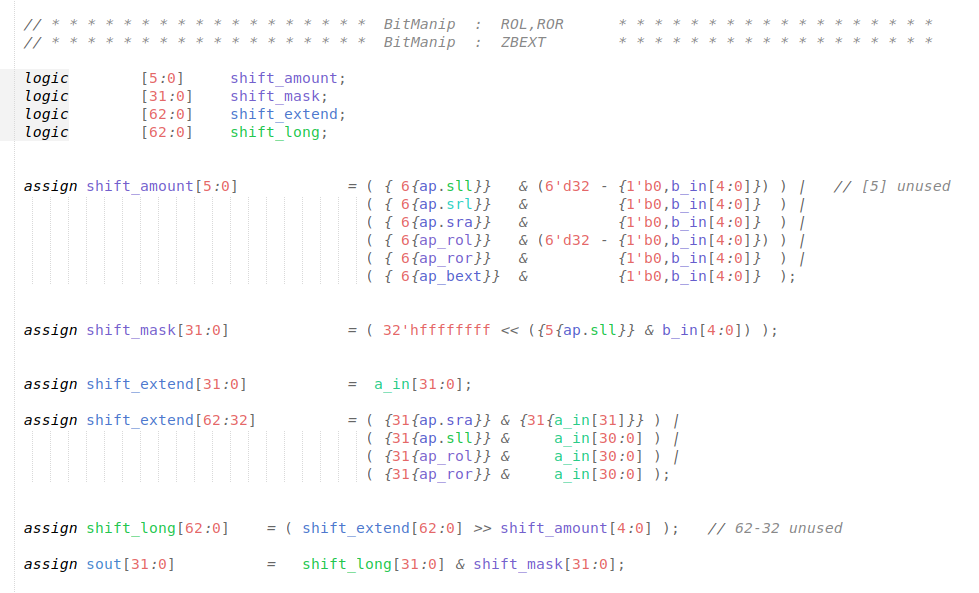
**…**

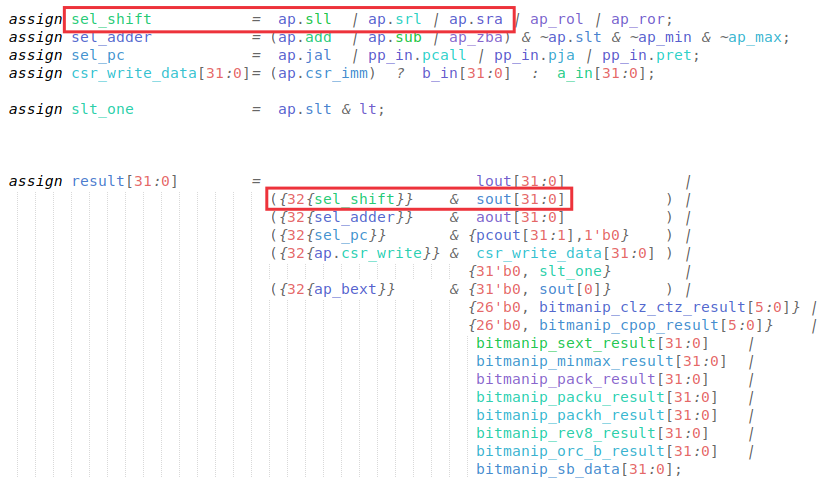
**0x00000124:** **01de13b3**  **sll** **t2,t3,t4**

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:





1. 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.

1. 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

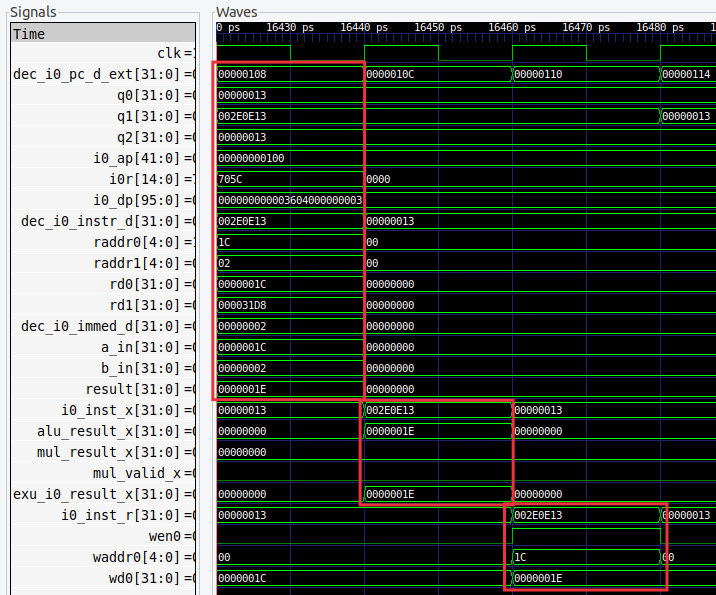
beq zero, zero, REPEAT # Repeat the loop

.end

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

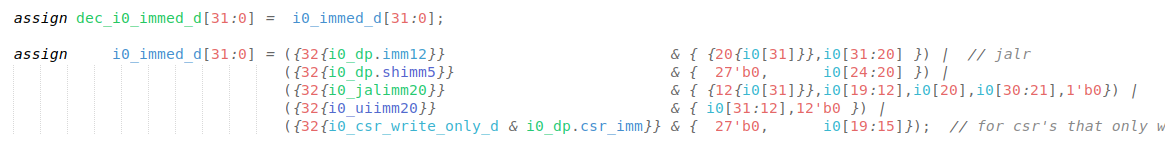
**0x00000108:** **002e0e13**  **addi** **t3,t3,2**

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.

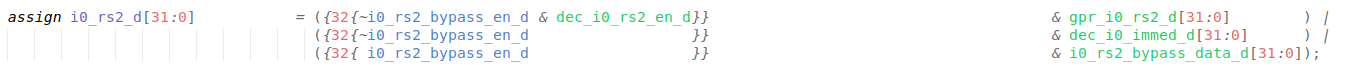


* You can see that the second operand is selected from the immediate (dec\_i0\_immed\_d) and not from the RF.
* In the control signal i0\_ap, the bit for add is set.
* In the control signal i0\_dp, the **imm12** bit is set and instead the **rs2** bit is not set.

The immediate is generated in file *el2\_dec\_decode\_ctl.sv*:



The multiplexer selects the second operand as follows:



Signal dec\_i0\_rs2\_en\_d is not set for immediate instructions:

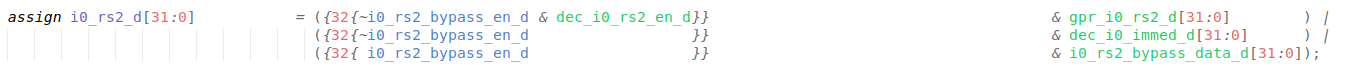


1. (*The following exercise is based on exercise 4.6 of [PaHe].*)

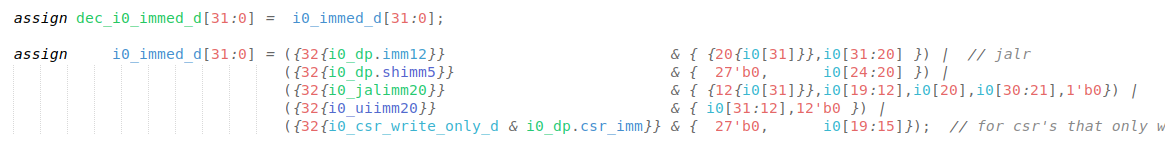
Figure 6 does not discuss I-type instructions like addi or andi.

* 1. What additional logic blocks, if any, are needed to support execution of I-type instructions in VeeR EL2? Add any necessary logic blocks to Figure 6 and explain their purpose.
  2. 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]:



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 [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.