THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 18

Adding New Features: Instructions and Counters

1. INTRODUCTION

In this lab, you will apply the knowledge acquired in previous labs to modify the SweRV EH1 processor to add the following new features:

- **Add A-L instructions:** Add Arithmetic-Logic instructions from the new bit manipulation extension available in the RISC-V architecture.
- **Add floating-point instructions:** Add three floating point instructions: add, multiply, and divide. Then use them to compute the bisection algorithm.
- **Add counter:** Add a new hardware counter that counts the number of I-Type instructions executed.

In some of these exercises we guide you through the process of modifying the core, and in others you will figure out on your own what needs to be done.

2. EXERCISES

1) The bit-manipulation (*bitmanip*) extension is comprised of several component extensions to the base RISC-V architecture that are intended to provide some combination of code size reduction, performance improvement, and energy reduction. You can find the complete specification at https://github.com/riscv/riscv-bitmanip. File https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf describes in detail all the instructions that belong to this extension.

In this exercise, you will include a new instruction from the *bitmanip* extension in the SweRV EH1 processor. Specifically, you will add the **`minu`** instruction, which places the smaller of the two unsigned integers in `rs1` and `rs2` into `rd`. The format used for this instruction is shown in the following illustration.

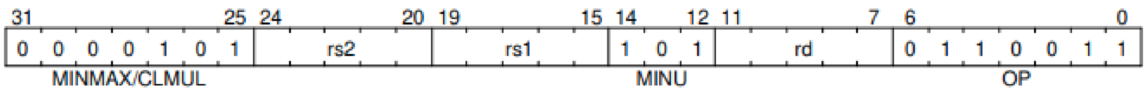


Figure 1. Format used for the `minu` instruction (figure obtained from https://github.com/riscv/riscv-bitmanip/releases/download/1.0.0/bitmanip-1.0.0.pdf**)**.

In order to include a new Arithmetic-Logic instruction, you must modify two main parts of the processor: the **Control Unit** and the **Execution Unit**. Figure 2 highlights in red the specific structures within these two units that you must modify for including the `minu` instruction (remember that this figure was first included in Lab 11 as Figure 4).

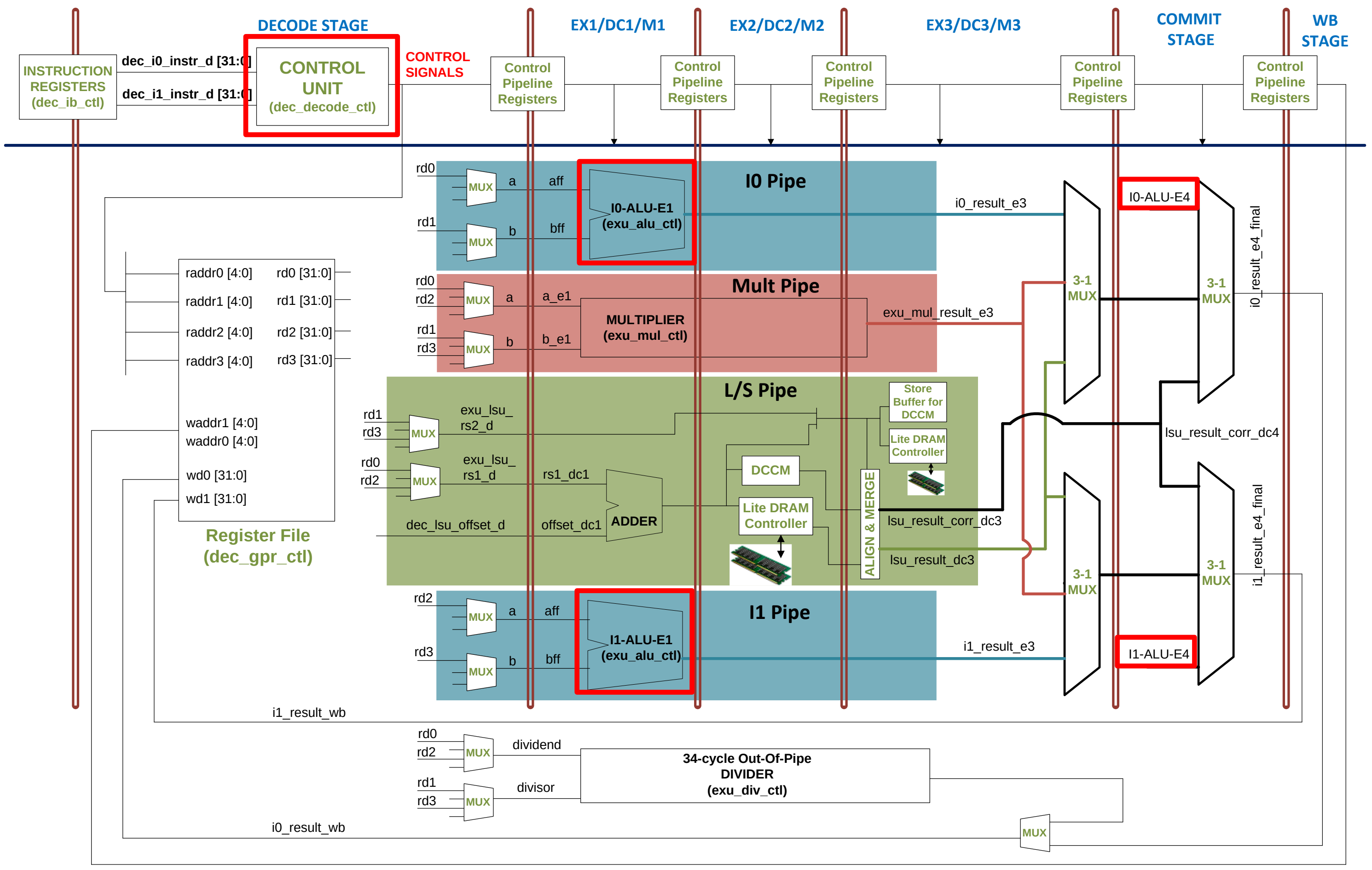


Figure 2. SweRV EH1's Decode, Execution, Commit and Writeback stages

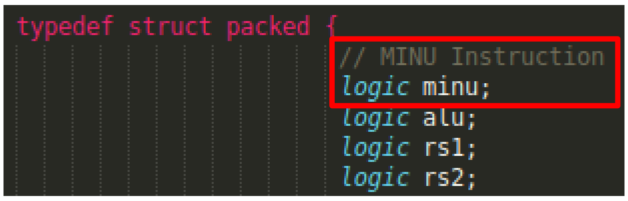
In this exercise, we give step-by-step instructions on how to add a new instruction, in this case `minu`. Then, in Exercise 2, you will follow a similar procedure to add other *bitmanip* instructions.

Control Unit Modfications:

NOTE: We recommend reviewing Section 2.C.i of Lab 11 and Section 4 of the SweRVref.docx before completing the next steps.

Now we will modify/create new control signals necessary to support the new instruction.

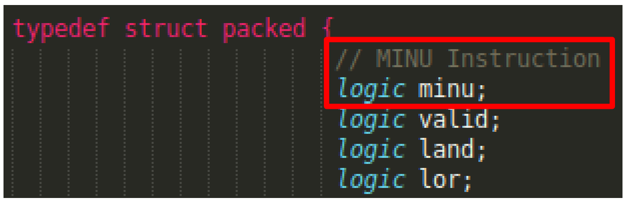
- Create two new bits in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv*. These two bits inform the processor if a `minu` instruction is executing.
    - Create a new bit, called *minu*, as part of the structure type `dec_pkt_t` (Figure 3). Remember that this is the main structure type used in the Control Unit.



```
typedef struct packed {
    // MINU Instruction
    logic minu;
    logic alu;
    logic rs1;
    logic rs2;
```

Figure 3. New bit in structure `dec_pkt_t`

- Create a new bit, called *minu*, as part of the structure type `alu_pkt_t` (Figure 4). Remember that this is the specific structure type used for Arithmetic-Logic instructions.



```
typedef struct packed {
    // MINU Instruction
    logic minu;
    logic valid;
    logic land;
    logic lor;
```

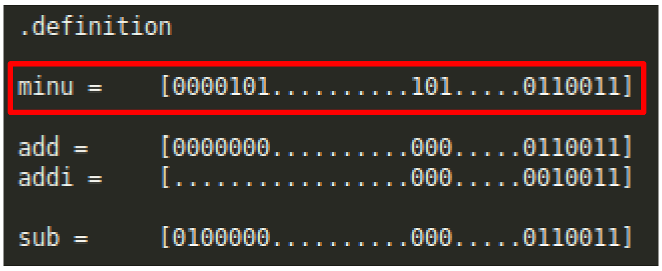
Figure 4. New bit in structure `alu_pkt_t`

- Assign a value to the new control signals in module **dec_decode_ctl** (implemented in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv*).

    - Assign the value to the new *minu* bit in the Decode stage, using signals `i0_dp_raw` and `i1_dp_raw`. To do so, you must modify the equations from module **dec_dec_ctl** (lines 2497-2672 of file *dec_decode_ctl.sv*), as explained next (note that these explanations are summarized in lines 2482-2495 of module **dec_decode_ctl**, from where we have obtained them and extended them a bit):

        1. File *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/decode* is a human readable file that has all of the instruction decodes defined in the SweRV EH1 processor, and that you must modify as explained next for including the `minu` instruction.

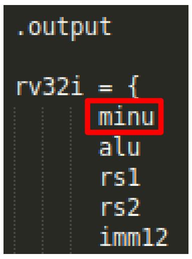
            - In section `.definition`, create a new line (Figure 5) for the new instruction according to its format, shown in Figure 1.



```
.definition

minu =    [0000101..........101.....0110011]

add =     [0000000..........000.....0110011]
addi =    [.................000.....0010011]

sub =     [0100000..........000.....0110011]
```

Figure 5. Modify *.definition* section

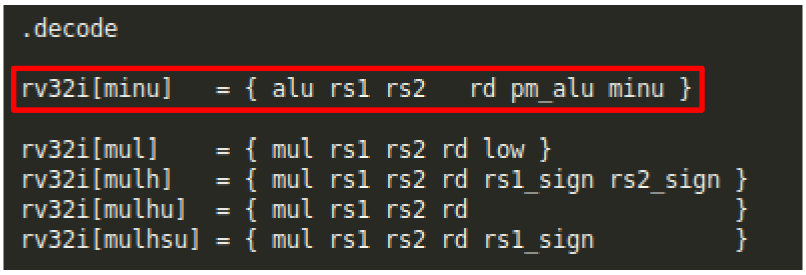
- In section `.output`, create a new bit called `minu` (Figure 6).



```
.output

rv32i = {
        minu
        alu
        rs1
        rs2
        imm12
```

Figure 6. Modify *.output* section

- In section `.decode`, create a new line for instruction `minu` (Figure 7). For the new instruction, the same bits as the ones enabled for an `add` instruction should be enabled, except the *add* bit. That is: *alu*, *rs1*, *rs2*, *rd*, *pm_alu*. Besides, the new *minu* bit should also be enabled.



```
.decode

rv32i[minu]   = { alu rs1 rs2   rd pm_alu minu }

rv32i[mul]    = { mul rs1 rs2 rd low }
rv32i[mulh]   = { mul rs1 rs2 rd rs1_sign rs2_sign }
rv32i[mulhu]  = { mul rs1 rs2 rd                   }
rv32i[mulhsu] = { mul rs1 rs2 rd rs1_sign          }
```

Figure 7. Modify *.decode* section

2. In the same folder (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/*), generate the *general equations*, which, after the modification of the *decode* file, include the instructions supported by SweRV EH1 plus the `minu` instruction.

```
./coredecode -in decode > coredecode.e
```

```
./espresso.linux -Dso -oeqntott coredecode.e |
./addassign -pre out.  > equations
```

These two commands will generate files *coredecode.e* and *equations*.

3. In the same folder (*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/*), generate the *legal equation*.

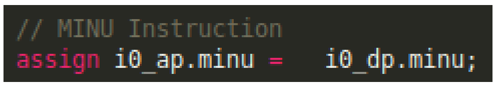
```
./coredecode -in decode -legal > legal.e
```

```
./espresso.linux -Dso -oeqntott legal.e |
./addassign -pre out. > legal_equation
```

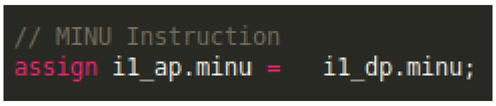
These two commands will generate files *legal.e* and *legal_equations*.

4. Modify the **dec_dec_ctl** module by substituting the existing equations (lines 2497-2672 of file *dec_decode_ctl.sv*), for the new ones, as defined in files *equations* and *legal_equations*.

- In module **dec_decode_ctl**, assign a value to the new *minu* bit in signals *i0_ap* and *i1_ap*, using signals *i0_dp* and *i1_dp* (Figure 8).



```
// MINU Instruction
assign i0_ap.minu =   i0_dp.minu;
```



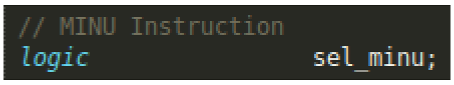
```
// MINU Instruction
assign i1_ap.minu =   i1_dp.minu;
```

Figure 8. Assign value to *minu* bits

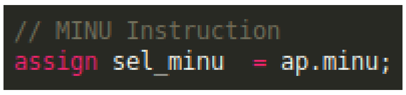
These steps describe the general procedure that must be followed for modifying the control unit when including a new instruction in the SweRV EH1 processor.

<u>Execution Unit Changes</u>:

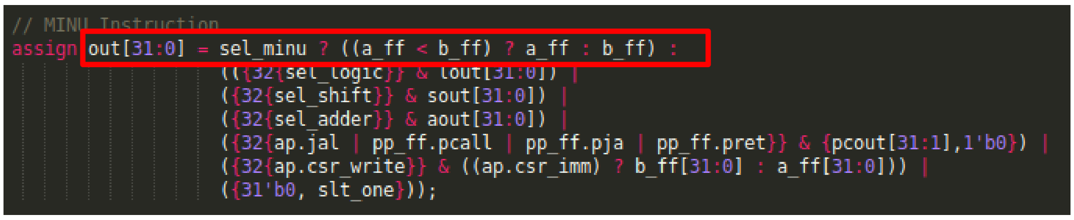
Next, modify the Execution Unit, which is implemented in modules **exu**, **exu_alu_ctl**, **exu_mul_ctl**, **exu_div_ctl** (the files that contain these modules are named after the modules). In future exercises, we will analyse complex situations where a new whole pipe is necessary. However, in this exercise, only a few small changes are required in module **exu_alu_ctl** (Figure 9).



```
// MINU Instruction
logic                   sel_minu;
```



```
// MINU Instruction
assign sel_minu   = ap.minu;
```



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

Figure 9. Modify the ALUs

Once you have completed these changes, you are ready to test the new instruction. Perform a simulation in Verilator that illustrates the use of the new instruction. You can use the program provided in Figure 10, or you can create your own program.

The program in Figure 10 creates an endless loop that computes the minimum value of two registers in each iteration. Note that the new instruction cannot be used normally (with a mnemonic), but it has to be used directly in machine format, as the RISC-V compiler does not support it yet.

```
.globl main
main:
```

```
li t3, 0x2
li t4, 0x30
li t6, -0x5

REPEAT:
        nop
        nop
        add t3, t3, t3
        add t4, t4, t6
        nop
        .word 0x0bde5f33 # minu t5, t4, t3    0000 101 | 1 1101  | 1110 0 | 101 | 1111 0 | 011 0011
        nop
        nop
        beq  zero, zero, REPEAT    # Repeat the loop
     nop

.end
```

Figure 10. Simple program for testing the new instruction, highlighted in red

Figure 11 shows the Verilator simulation (as usual, we use a *.tcl* script for including the signals). The waveform shows two iterations of the loop, which shows two executions of the new instruction (`ifu_i0_instr` or `ifu_i1_instr` = 0x0BDE5F33). Its main control bits (`i0_dp_raw` or `i1_dp_raw` = 0x7A00000000003) and its ALU control bits (`i0_ap` or `i1_ap` = 0x180000) are the same as in an `add` instruction except for the `minu` bit and the `add` bit. The result written in *t5* (shown in the bottom of the figure) is the minimum value of the two numbers read from *t3* and *t4*. Note that the second `minu` execution compares 0xFFFFFFFE with 0x00000800; given that it is an *unsigned min* instruction, 0xFFFFFFFE represents a large positive number, and thus the minimum value among the two is 0x00000800.

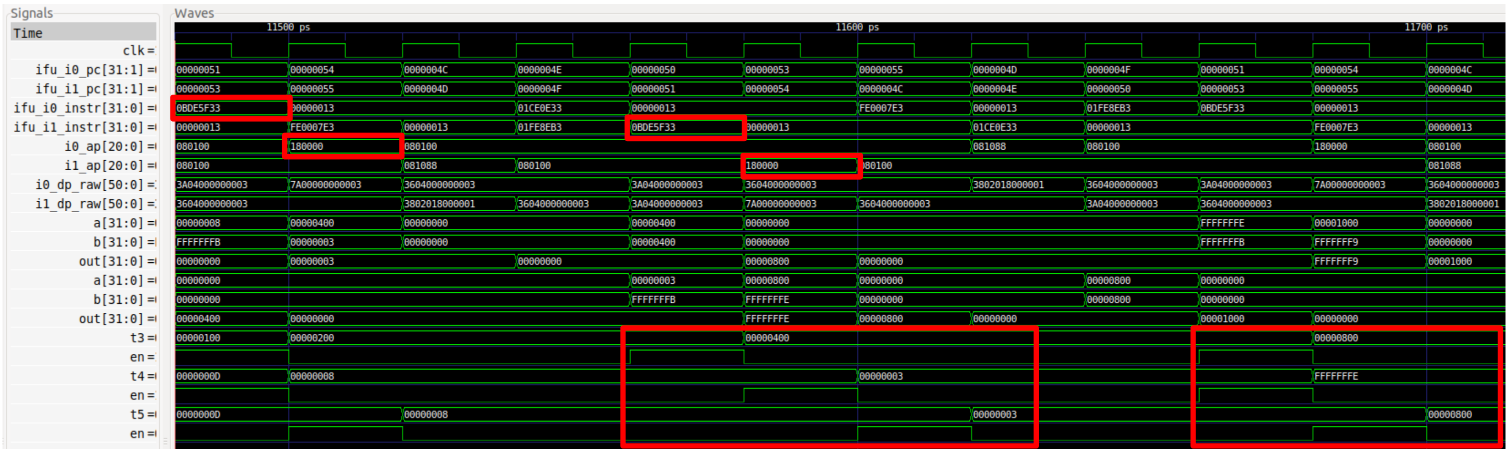


Figure 11. Verilator simulation of the program from Figure 10

Modify the program to perform different comparisons, and then simulate the program using Verilator.

Once you have checked that your implementation works correctly, generate the new bitstream in Vivado and test the new instruction on the board using any of the tests built for simulation.

Build a program that reads the 16 switches and compares the binary value of the 8 least significant switches with the binary value of the 8 most significant ones using the new instruction. Then, display the minimum value on the 7-segment displays.

Finally, build different tests to confirm that the instruction works as expected, and demonstrate the results on the board.

2) Implement other instructions that belong to the RISC-V *bitmanip* extension. Begin by completing the remaining `min/max` instructions: `min`, `max`, and `maxu`.

3) In this exercise you will extend the SweRV EH1 processor to include three new instructions that belong to the RISC-V Single-Precision Floating-Point extension (F extension): `fadd.s`, `fmul.s` and `fdiv.s`.

- The instructions assume that the operands are represented in single-precision floating-point IEEE 754 format (https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF). In a floating-point number, the register is logically divided into three fields: **Sign** (1 bit long), **Exponent** (8 bits) and **Mantissa** (23 bits).

Sign | E_7 … E_0 | M_{22} … M_0

- Instruction `fadd.s rd, rs1, rs2` adds the two floating-point values in `rs1` and `rs2` and stores the result in `rd`. Instruction `fmul.s rd, rs1, rs2` multiplies the two floating-point values in `rs1` and `rs2` and stores the result in `rd`. Finally, instruction `fdiv.s rd, rs1, rs2` divides the two floating-point values in `rs1` and `rs2` and stores the result in `rd`.

- The formats used for these instructions, as defined in the RISC-V F extension, are:

```
fadd.s: 0000000 | rs2 | rs1 | Rounding-Mode | rd | 1010011
fmul.s: 0001000 | rs2 | rs1 | Rounding-Mode | rd | 1010011
fdiv.s: 0001100 | rs2 | rs1 | Rounding-Mode | rd | 1010011
```

- Although this extension assumes a processor that has 32 floating-point registers, in this exercise, for the sake of simplicity, you will use the existing Register File used by any other instruction (i.e. the *x* registers). Also, we assume other simplifications: only one floating-point instruction can execute at once and floating-point instructions are blocking.

In order to include support for these instructions in the SweRV EH1 processor, you must make the following modifications:

<u>Execution Unit Changes</u>:

You will add hardware for floating-point addition, multiplication, and division (you may find some sources on the Internet as we detail below). You will then use this hardware when a `fadd`, a `fmul`, or a `fdiv` instruction is executed. To do so, complete the following:

  - Download the multi-cycle floating-point Adder, Multiplier, and Divider provided at: https://github.com/dawsonjon/fpu. These are non-pipelined multi-cycle units similar to the integer Divider available in SweRV EH1.

- Even though the new units constitute new pipes and thus could be treated independently, you can instantiate the three floating-point units inside the **exu_div_ctl** module, given that this execution pipe provides some signals that are useful for supporting the new instructions, such as signals *finish* and *div_stall*. If you do it this way, you must enable the same bits as a `div` instruction, plus the new floating-point bits, when generating the equations for the Control Unit as explained below.

Control Unit Changes:

Modify/create new control signals to support the new instructions.

- Create new bits and structure types in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv*.
    - Create a new structure type called `fp_pkt_t` which includes 3 bits: *fp_add*, *fp_mul*, and *fp_div*, which indicate, respectively, if the processor is executing a floating-point addition, a floating-point multiplication or a floating-point division.
    - Create three new bits, called *fp_add*, *fp_mul* and *fp_div*, that are part of the structure type `dec_pkt_t`. Remember that this is the main structure type used in the Control Unit.
- Assign a value to the new control signals in module **dec_decode_ctl** (implemented in file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv*).
    - Assign values to the new bits in signals `i0_dp_raw` and `i1_dp_raw`. For that purpose, you must regenerate the equations from module **dec_dec_ctl** as explained in Exercise 1. As mentioned above, if you manage the new instructions as a `div` instruction, you must enable the same bits as a `div` instruction, plus the new floating-point bits, when generating the equations from module **dec_dec_ctl**.
    - Create a new signal of type `fp_pkt_t` called `fp_p`. Then assign values to the three bits of this structure, using signals `i0_dp` and `i1_dp`. Note that, similarly to the `mul` or `div` instructions, only one signal of this type is required, because only one floating point instruction can execute in a given cycle.

After modifying the hardware, perform a simulation in Verilator that illustrates the use of the new instructions. You can use the program provided in Figure 12, or you can create your own one. The program in Figure 12 creates an endless loop that computes three instructions: floating-point add, multiply, and divide.

```
.globl main
main:
```

```
li t0, 0x4
li t1, 0x2
li t3, 0x40800000
li t4, 0x40000000

REPEAT:
       div t5, t0, t1
       nop
       nop
       .word 0x01ce8f53     # fadd.s 0000000 | 11100 | 11101 | 000 | 11110 | 1010011
       nop
       nop
       .word 0x11ce8f53     # fmul.s 0001000 | 11100 | 11101 | 000 | 11110 | 1010011
       nop
       nop
       .word 0x19ce8f53     # fdiv.s 0001100 | 11100 | 11101 | 000 | 11110 | 1010011
       nop
       nop
       beq  zero, zero, REPEAT    # Repeat the loop
    nop

.end
```

Figure 12. Simple program for testing the new instructions, highlighted in red

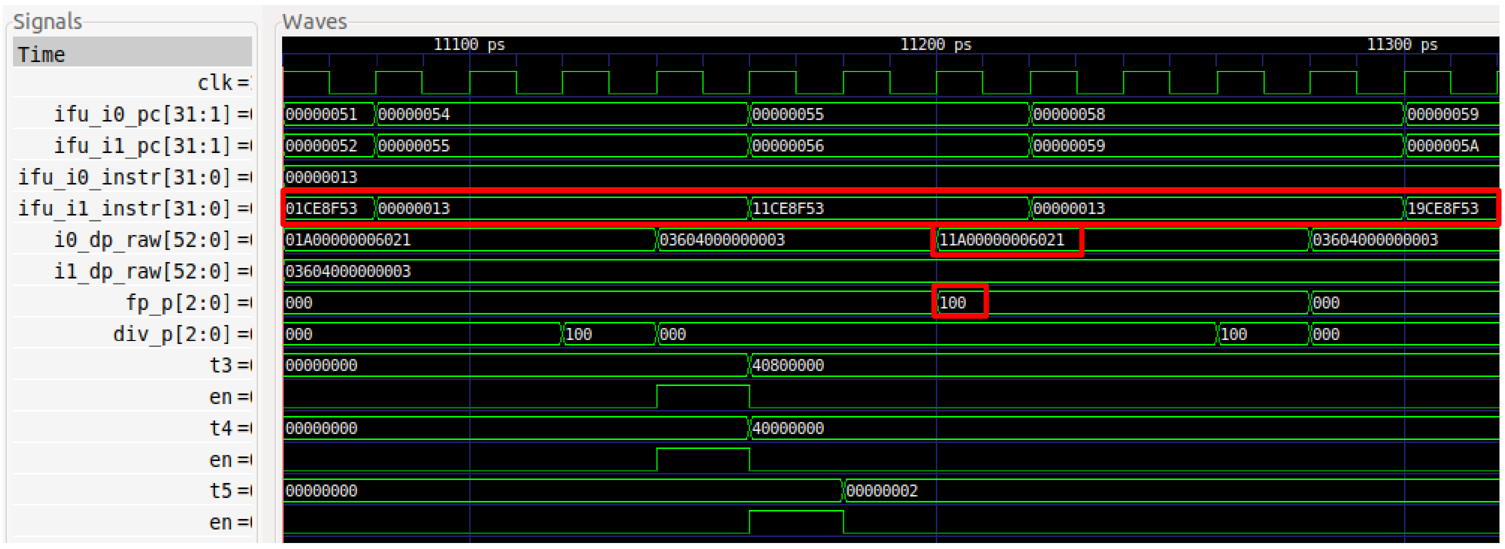
Figure 13 shows the results of the Verilator simulation. To check the results, you can use a floating-point converter, such as the one available at: https://www.h-schmidt.net/FloatConverter/IEEE754.html.

In Figure 13-a, the three floating-point instructions are fetched into `ifu_i0_instr` or `ifu_i1_instr`. Their main control bits (`dec_pkt_t`) are the same as those for a `div` instruction (`i0_dp_raw` = 0x11A00000006021) with the three extra bits added as described above. The FP (floating-point) control bits (`fp_pkt_t`) are 100 for `fadd` (as shown in the figure), 010 for `fmul`, and 001 for `fdiv`.

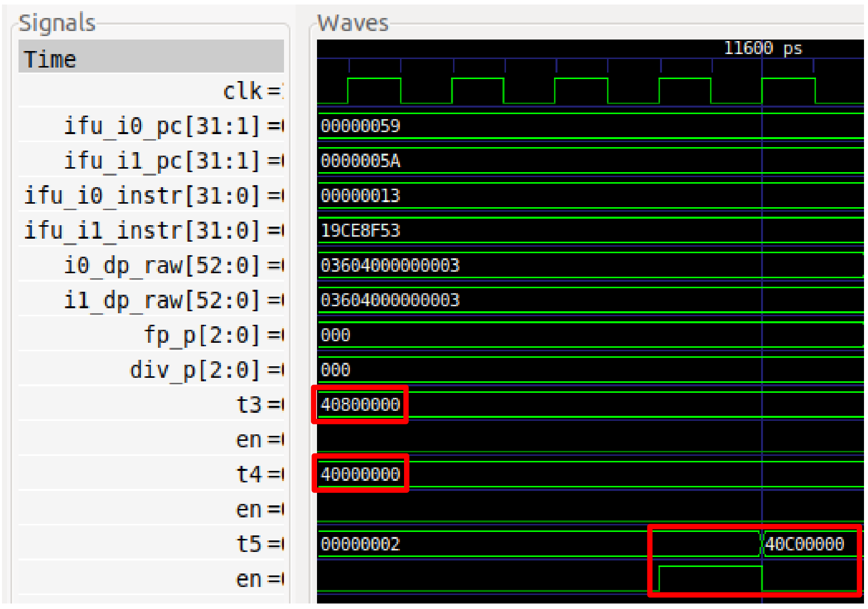
Figure 13-b shows the FP add writing its result into *t5* several cycles later. Note that the input values are 0x40800000 and 0x40000000, thus the result of the addition is 0x40c00000.

Figure 13-c shows the FP multiply writing its result into *t5* several cycles later. Note that the input values are 0x40800000 and 0x40000000, thus the result of the multiplication is 0x41000000.

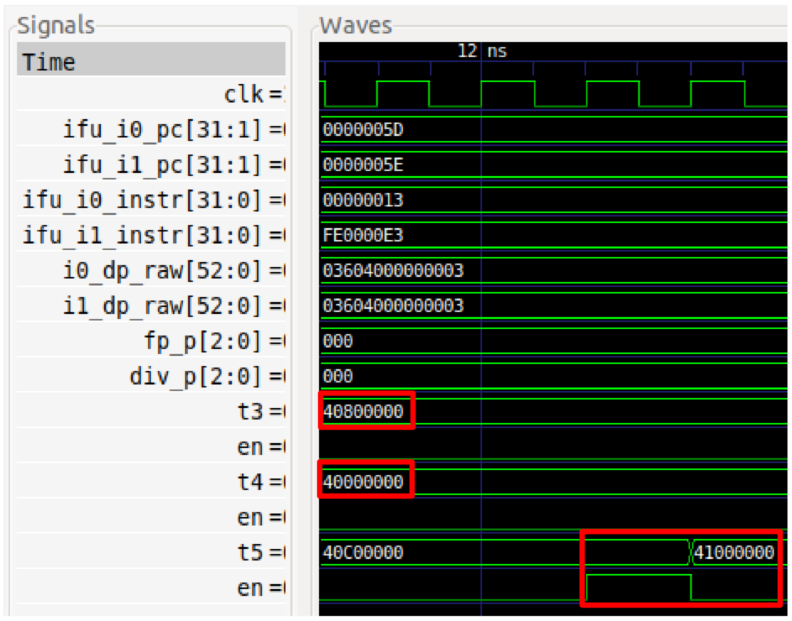
Finally, Figure 13-d shows the FP divide writing its result into *t5* several cycles later. Note that the input values are 0x40800000 and 0x40000000, thus the result of the division is 0x40000000.


Signals
Waves
Time
11100 ps
11200 ps
11300 ps
clk
ifu_i0_pc[31:1]
00000051 00000054 00000055 00000058 00000059
ifu_i1_pc[31:1]
00000052 00000055 00000056 00000059 0000005A
ifu_i0_instr[31:0]
00000013
ifu_i1_instr[31:0]
01CE8F53 00000013 11CE8F53 00000013 19CE8F53
i0_dp_raw[52:0]
01A000000006021 03604000000003 11A00000006021 03604000000003
i1_dp_raw[52:0]
03604000000003
fp_p[2:0]
000 100 000
div_p[2:0]
000 100 000 100 000
t3
00000000 40800000
en
t4
00000000 40000000
en
t5
00000000 00000002
en

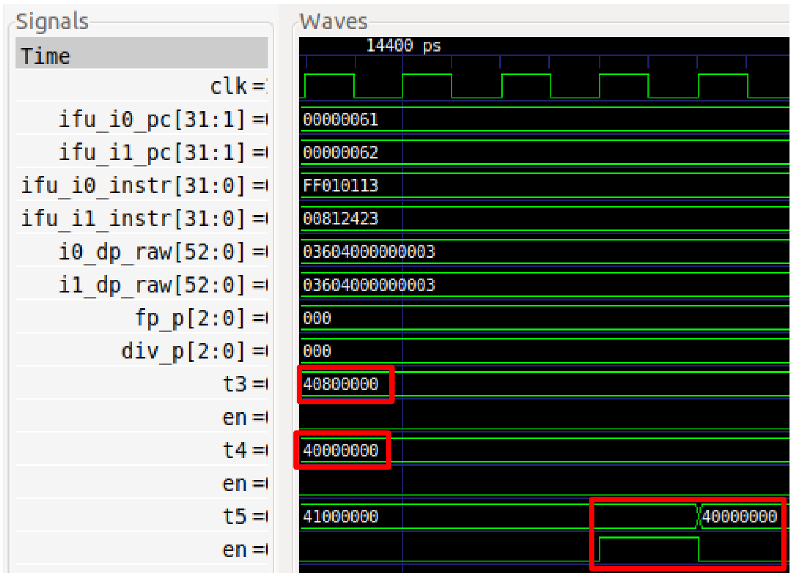
(a)


Signals
Waves
Time
11600 ps
clk
ifu_i0_pc[31:1]
00000059
ifu_i1_pc[31:1]
0000005A
ifu_i0_instr[31:0]
00000013
ifu_i1_instr[31:0]
19CE8F53
i0_dp_raw[52:0]
03604000000003
i1_dp_raw[52:0]
03604000000003
fp_p[2:0]
000
div_p[2:0]
000
t3
40800000
en
t4
40000000
en
t5
00000002 40C00000
en

(b)



(c)



(d)

Figure 13. Verilator simulation of the program from Figure 12

Modify the program to test other cases and demonstrate that the instructions work correctly. For example, test negative numbers, data dependencies with previous/subsequent instructions, etc. Then simulate them using Verilator.

Next, test the new instructions in hardware on the board. To do so, program the example *DotProduct_C-Lang* provided in the GSG, using the new `fmul` and `fadd` instructions for performing the floating-point computations. Compare the execution of this algorithm when floating-point instructions are emulated vs. when these instructions are implemented in hardware.

You can also add more functionality, such as providing support for: other floating-point formats (such as *double precision*), other floating-point rounding modes, a new register file for the floating-point values, your own FP unit implementation, etc.

4) Implement the Bisection Method. You can find a lot of information about this root-finding algorithm on the internet, for example, at: https://en.wikipedia.org/wiki/Bisection_method.

   Compare the execution of this algorithm when floating-point instructions are emulated vs. when these instructions are implemented in hardware.

5) Implement any of the instructions proposed in the exercises from Chapter 4 from the book "Computer Organization and Design – RISC-V Edition", by Patterson & Hennessy ([HePa]), such as:

   a. (from [HePa] Exercise 4.11):
      i. Instruction "Load With Increment": `lwi.d rd, rs1, rs2`
      ii. Interpretation: `rd = Mem[rs1 + rs2]`

   b. (from [HePa] Exercise 4.12):
      i. Instruction "Swap": `swap rs1, rs2`
      ii. Interpretation: `rs2 = rs1; rs1 = rs2`

   c. (from [HePa] Exercise 4.13):
      i. Instruction "Store Sum": `ss rs1, rs2, imm`
      ii. Interpretation: `Mem[rs1] = rs2 + imm`

6) Similar to the previous exercise, implement the instructions proposed in Exercises 3-6 from Chapter 7 of the textbook by S. Harris and D. Harris, "Digital Design and Computer Architecture: RISC-V Edition" [DDCARV]. We repeat below all of the instructions included in these four exercises. Some of them are already supported by our SweRV EH1 processor, in which case, instead of implementing them, you can simply explain how they are implemented.

   a. Exercise 3: `xor, sll, srl, bne`. (Already implemented in SweRV EH1)

   b. Exercise 4: `lui, sra, lbu, blt, bltu, bge, bgeu, jalr, auipc, sb, slli, srai`. (Already implemented in SweRV EH1)

   c. Exercise 5: `lwpostinc rd,imm(rs)` (the instruction is equivalent to the following two instructions: `lw rd, 0(rs)` followed by `addi rs, rs, imm`).

   d. Exercise 6: `lwpreinc rd, imm(rs)` (the instruction is equivalent to the following two instructions: `lw rd, imm(rs)` followed by `addi rs, rs, imm`).

7) Include a new event for counting the number of I-Type instructions executed in a program. We provide some guidance to help you complete this exercise:

   ○ You will need to modify some structures from file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv* *_types.sv*. Specifically, you should add another field in the following structure type:

- Structure `inst_t`: new field for an I-Type instruction.

- As you know, the control bits are assigned in module **dec_decode_ctl** (file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_decode_ctl.sv*). Modify the assignment of signals `i0_itype` and `i1_itype` to add the new instruction type included in the previous item.

- The hardware counters are implemented in module **dec_tlu_ctl** (file *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/dec/dec_tlu_ctl.sv*). Open that file and analyse the code included in lines 1882 to 2143. You will have to modify this part of the code to include the new counter.

After the new counter has been included in the Verilog code, debug the implementation using Verilator. Once your implementation has been verified through simulation, generate the new bitstream for the SoC and test the operation of the new counter in hardware on the board.