THE IMAGINATION UNIVERSITY PROGRAMME

RVfpgaEL2 Lab 18

Adding New Instructions to the VeeR EL2 core

1. Adding Instructions `fadd.s`, `fmul.s` and `fdiv.s` to VeeR EL2

A. Introduction

In this lab, you will apply the knowledge acquired in previous labs to modify the VeeR EL2 processor to add three floating-point instructions that belong to the **RISC-V Single-Precision Floating-Point Zfinx** extension: `fadd.s`, `fmul.s` and `fdiv.s`. At https://wiki.riscv.org/display/HOME/Zfinx+TG and at https://github.com/riscv/riscv-zfinx, you can find more information about this extension, which in many aspects is similar to the **RISC-V Single-Precision Floating-Point F** extension, for which you can find details at https://five-embeddev.com/riscv-isa-manual/latest/f.html.

The Zfinx extension provides instructions similar to those in the standard floating-point F extension for single-precision floating-point instructions, but they operate on the **x** (integer) registers instead of the **f** (floating-point) registers. This makes the implementation simpler, given that the VeeR EL2 processor does not include a Floating Point Register File and adding it would be more complex.

Given the complexity of adding new instructions, we guide you through the process. Once you've learned how to add new instructions to the core, you can practice by adding other instructions from this same RISC-V extension or from any other RISC-V extension.

B. Floating point instructions `fadd`, `fmul` and `fdiv`

As stated by the Zfinx specification, the variants of these F-extension instructions have the same semantics, except that whenever such an instruction would have accessed an **f** register, it instead accesses the **x** register with the same number.

We next summarize some features of these three RISC-V instructions:

- **`fadd.s`**
  - Instruction `fadd.s rd, rs1, rs2` adds the two floating-point values in `rs1` and `rs2` and stores the result in `rd`.
  - Its format, as defined in the RISC-V F extension, is the following:
    `0000000 | rs2 | rs1 | Rounding-Mode | rd | 1010011`
- **`fmul.s`**
  - Instruction `fmul.s rd, rs1, rs2` adds the two floating-point values in `rs1` and `rs2` and stores the result in `rd`.
  - The instruction format, as defined in the RISC-V F extension, is the following:
    `0001000 | rs2 | rs1 | Rounding-Mode | rd | 1010011`
- **`fdiv.s`**
  - Instruction `fdiv.s rd, rs1, rs2` adds the two floating-point values in `rs1` and `rs2` and stores the result in `rd`.

- The instruction format, as defined in the RISC-V F extension, is the following:
  `0001100 | rs2 | rs1 | Rounding-Mode | rd | 1010011`

Floating-point instructions assume that the operands are represented in single-precision floating-point IEEE 754 format: https://people.eecs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF. To represent floating-point numbers, the register is logically divided into three fields: **Sign** (1 bit long), **Exponent** ($E_{7:0}$, 8 bits) and **Mantissa** ($M_{22:0}$, 23 bits).

Sign | E_7 ... E_0 | M_{22} ... M_0

C. Extension of the VeeR EL2 processor to support the new instructions

We next describe in detail how to perform the required changes for including the three instructions in VeeR EL2. Specifically, you must make changes in two parts of the core: the Execution Unit and the Control Unit.

Changes in the Control Unit:

Modify/create new control signals to support the new instructions. You must make changes in two files:

In file *[RVfpgaEL2NexysA7DDRPath]/src/VeeRwolf/VeeR_EL2CoreComplex/include/el2_def.sv*, perform the following changes:

1) Create a new structure type called `fp_pkt_t` which includes 3 bits: `fp_add`, `fp_mul`, and `fp_div`; these bits indicate, respectively, if the processor is executing a floating-point addition, a floating-point multiplication or a floating-point division.

```
// FP
typedef struct packed {
                        logic fp_add;
                        logic fp_mul;
                        logic fp_div;
                        } fp_pkt_t;
```

2) Create three new bits, called `fp_add`, `fp_mul`, and `fp_div`, that are part of the structure type `el2_dec_pkt_t`. Remember that this is the main structure type used in the Control Unit.

```
typedef struct packed {
                        logic fp_add;
                        logic fp_mul;
                        logic fp_div;
                        logic ctz;
                        logic ctz;
                        logic pcnt;
                        logic sext_b;
                        logic sext_h;
```

In file *[RVfpgaEL2NexysA7DDRPath]/src/VeeRwolf/VeeR_EL2CoreComplex/dec/el2_dec_decode_ctl.sv*, perform the following changes:

1) Assign the value to the new bits in the D Stage, using signal `i0_dp_raw`. To do so, you must modify the equations from module `el2_dec_dec_ctl` (lines 1541-1867 of file *el2_dec_decode_ctl.sv*), as explained next (note that these explanations are summarized in lines 1526-1539 of module `el2_dec_decode_ctl`, from where we have obtained and extended them):
    a. Generate the new equations:
        - File *[RVfpgaEL2NexysA7DDRPath]/src/VeeRwolf/VeeR_EL2CoreComplex/dec/decode* is a human readable file that has all of the instruction decodes defined in the VeeR EL2 processor, and that you must modify as explained next.
            - In section `.definition`, create a new line for each of the new instructions according to their format, shown above.

```
.definition

fadd =    [0000000.................1010011]
fmul =    [0001000.................1010011]
fdiv =    [0001100.................1010011]

clz       =  [011000000000.....001.....0010011]
ctz       =  [011000000001.....001.....0010011]
cpop      =  [011000000010.....001.....0010011]
sext_b    =  [011000000100.....001.....0010011]
sext_h    =  [011000000101.....001.....0010011]
```

            - In section `.output`, create a new bit for each instruction.

```
.output

rv32i = {
      fp_add
      fp_mul
      fp_div
      alu
      rs1
      rs2
      imm12
```

            - In section `.decode`, create a new line for each instruction that indicates the control bits enabled by each of them. As we explain below, for the sake of simplicity we treat the new instructions similarly to the `div` instructions. Thus, the same signals enabled for a division must be included plus the specific new signal created before for each instruction (`fp_add`, `fp_mul`, and `fp_div`).

```
.decode

rv32i[fadd]   = { div rs1 rs2 rd        presync postsync  fp_add }
rv32i[fmul]   = { div rs1 rs2 rd        presync postsync  fp_mul }
rv32i[fdiv]   = { div rs1 rs2 rd        presync postsync  fp_div }

rv32i[clz]       =  { alu zbb     rs1     rd                     clz   }
rv32i[ctz]       =  { alu zbb     rs1     rd                     ctz   }
```

        - In the same folder (*[RVfpgaEL2NexysA7DDRPath]/src/VeeRwolf/VeeR_EL2CoreComplex/dec*),

generate the *general equations*, which, after the modification of the *decode* file, will include the instructions supported by VeeR EL2 plus the three floating point 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`.

- In the same folder (*[RVfpgaEL2NexysA7DDRPath]/src/VeeRwolf/VeeR_EL2CoreComplex/dec*), generate the *legal equations*.

```
./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`.

b. Substitute the old equations with the new ones generated in the previous step. Specifically, you must substitute the existing equations in lines 1554-1862 of file *el2_dec_decode_ctl.sv*, for the new ones defined in files *equations* and *legal_equations*.

2) In module **`el2_dec_decode_ctl`**, assign a value to the new floating point control bits in signal `fp_p`, using signal `i0_dp`.

```
assign div_p.valid    =  div_decode_d;

assign div_p.unsign   =  i0_dp.unsign;
assign div_p.rem      =  i0_dp.rem;

// FP
assign fp_p.fp_add  = i0_dp.fp_add;
assign fp_p.fp_mul  = i0_dp.fp_mul;
assign fp_p.fp_div  = i0_dp.fp_div;

assign mul_p.valid    =  mul_decode_d;
```

Changes in the Execution Unit:

The Execution Unit is implemented in modules **`el2_exu`**, **`el2_exu_alu_ctl`**, **`el2_exu_mul_ctl`**, **`el2_exu_div_ctl`** (the files that contain these modules are named after the modules). You will add hardware for computing floating-point addition, multiplication, and division (you may find some sources on the Internet as we detail below). The processor will then use this hardware when a `fadd.s`, `fmul.s`, or `fdiv.s` instruction is executed. To do so, complete the following steps:

- **Obtain the units for the floating-point computations**: Download the multi-cycle floating-point Adder, Multiplier, and Divider provided at: https://github.com/dawsonjon/fpu. These are non-pipelined multi-cycle units like the Integer Divider available in VeeR EL2. You do not need to understand their internal design, but you need to understand their interface (their input and output signals), so that you are able to integrate the new units in the processor. For example:
    - Signals `input_a` and `input_b` are used to provide the two input operands to the FP unit.
    - Signal `output_z_stb` is used to indicate if the operation has finished.

- **Integrate the new instructions in the processor**: For the sake of simplicity, we recommend the new instructions to be treated similarly to the `div` instructions (for example, the result of the new instructions will also be written through port 2 of the Register File) and the floating-point units to be instantiated from inside the **`el2_exu_div_ctl`** module. That's why, in the previous section ("Changes in the Control Unit"), for the new floating-point instructions we set the same control bits as `div` instructions.
  The **`el2_exu_div_ctl`** module provides some signals that are useful for integrating the new instructions:
    - Signals `dividend` and `divisor`: These signals are assigned with the input operands to the divisor. In the new implementation we will redefine them to provide the input operands both to the divisor and to the FPU. Note that you do not need to make changes in the Verilog codes, and you can simply use these two signals as they are.
    - Signal `out`: This is assigned with the result of the division. In the new implementation we will redefine it to provide both the result of the `div` instruction and the result of the floating-point instructions. This requires a new multiplexer inside the **`el2_exu_div_ctl`** module to select the correct output for the different instructions supported in the module (`div`, `fadd.s`, `fmul.s` and `fdiv.s`).
    - Signal `finish_dly`: This signal goes high when the division ends and it is used as the enable signal of write port 2 of the Register File. In the new implementation we will redefine it for signalling both the completion of a `div` instruction and the completion of a floating-point instruction.

D. Experiments

After modifying the hardware, we will perform a simulation in RVfpgaEL2-Trace that illustrates the use of the new instructions. You can use the program provided in Figure 1, or you can create your own one. The program in Figure 1 creates an endless loop that computes three instructions: floating-point add, multiply, and divide. It also includes a division in order to confirm that this instruction keeps working correctly.

```
main:

li t3, 0x40800000
li t4, 0x40000000

li a3, 20
```

```
li a4, 4

REPEAT:
INSERT_NOPS_4
.word 0x01ce8f53     # fadd.s 0000000 | 11100 | 11101 | 000 | 11110 | 1010011

INSERT_NOPS_4
.word 0x11ce8f53     # fmul.s 0001000 | 11100 | 11101 | 000 | 11110 | 1010011

INSERT_NOPS_4
.word 0x19ce8f53     # fdiv.s 0001100 | 11100 | 11101 | 000 | 11110 | 1010011

INSERT_NOPS_4
div a5, a3, a4       # just to confirm that div keeps working alright

INSERT_NOPS_10

beq zero, zero, REPEAT # Repeat the loop
```

Figure 1. Program for testing the new floating point instructions

Figure 2 shows the results of the RVfpgaEL2-Trace simulation for the `add` instruction. 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.

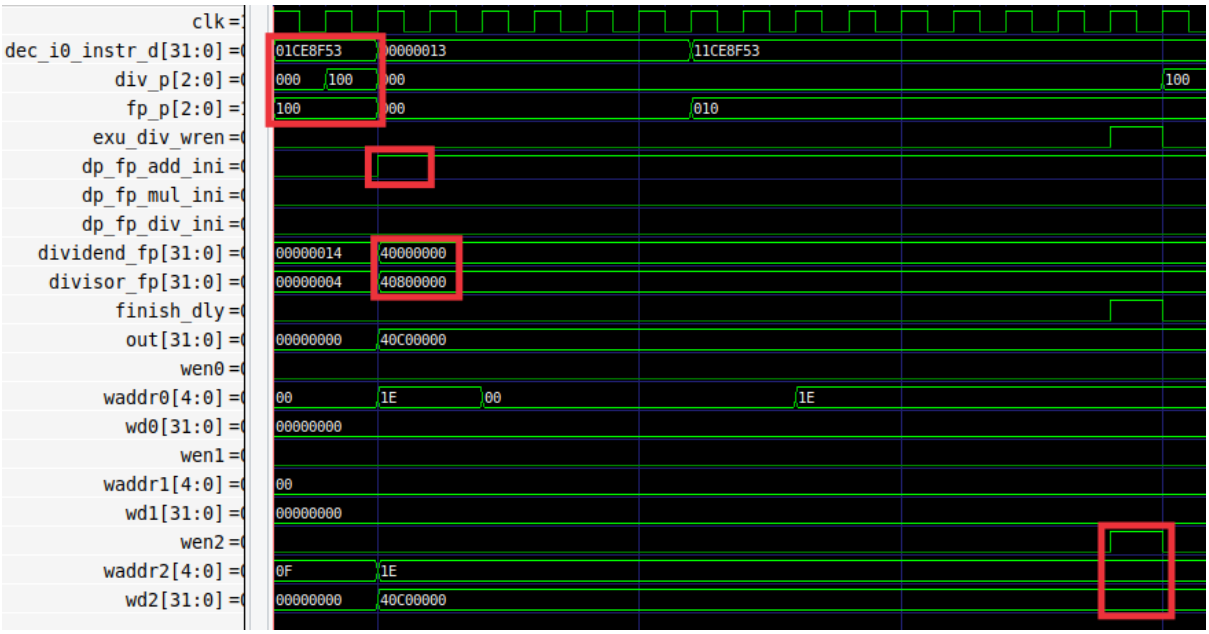

Figure 2. Simulation of `fadd`

In the first cycle, signal `dec_i0_instr_d` has the `fadd` instruction (0x01ce8f53), which is at the D Stage. The two operands are read from the Register File and provided to the Floating Point Unit (`dividend_fp` = 0x40000000 and `divisor_fp` = 0x40800000). After a few cycles the `finish_dly` signal goes high, indicating that the floating point operation finishes, and the result is written to the Register File through Write Port 2: `wen2` = 1, `waddr2` = 0x1E and `wd2` = 0x40C00000.

Figure 3 shows the results of the RVfpgaEL2-Trace simulation for the `div` instruction.

Figure 3. Simulation of `div`

In the first cycle, signal `dec_i0_instr_d` has the `div` instruction (0x02e6c7b3), which is in the D Stage. The two operands are read from the Register File and provided to the Floating-Point Unit (`dividend_fp` = 0x14 and `divisor_fp` = 0x4). After a few cycles the result is written to the Register File through Write Port 2: `wen2` = 1, `waddr2` = 0x0F and `wd2` = 0x5.

2. Exercises

1) Modify the SoC to include the `fadd`, `fmul` and `fdiv` instructions, as explained in Section C. Generate the bitstream in Vivado and RVfpgaEL2-Trace, RVfpgaEL2-ViDBo and RVfpgaEL2-Pipeline binaries with Verilator.

2) Test the program from Figure 1, both in RVfpgaEL2-Trace and on the board. Analyse the `fmul` and `fdiv` instructions.

3) Modify the provided program to test other cases and test if the instructions work correctly. For example, test negative numbers, data dependencies with previous/subsequent instructions, etc.

4) Implement 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.

5) 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.

6) Replace the FPU (floating-point unit) with the following one: https://github.com/openhwgroup/cvfpu. The Final Degree Project “Extensiones de punto flotante para el core SweRV EH1” should be helpful, as it performs the same extension on VeeR EH1. You will find the project on the Internet and the sources at: https://github.com/aperea01/TFG-SweRV-EH1-FP

7) 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 (note that floating point instructions that use a Floating-Point Register File are described in the RISC-V F extension), your own FP unit implementation, etc.

8) Add instructions from other RISC-V Extensions that are not available in the VeeR EL2 processor.

9) Verify the processor, including the new instructions. The Final Degree Project “Extensiones de punto flotante para el core SweRV EH1[Floating-point extensions for the SweRV EH1 core]” should be helpful, as it performs the same extension on VeeR EH1. You will find the project on the Internet and the sources at: https://github.com/aperea01/TFG-SweRV-EH1-FP