MIPSfpga by Imagination

Lab 19

User Defined Instructions using the CorExtend Interface

These materials produced in association with Imagination.
Join our University community for more resources.
community.imgtec.com/university

Lab 19
User Defined Instructions using the CorExtend Interface

1. Introduction

In this lab, you will learn how to add your own instructions using the MIPS CorExtend interface [1] available in MIPS processors. This feature allows designers to add to the processor User Defined Instructions (UDIs) that operate on data in the general-purpose registers in the same way as standard MIPS instructions. UDIs can enable significant performance improvement in critical algorithms beyond what is achievable with standard MIPS32 instructions.

Let´s start presenting a simple example that we will analyze in detail later. Release3 of MIPS ISA includes 4 logic instructions: `or`, `and`, `nor` and `xor`. Suppose that we execute an algorithm that performs many `nand` operations. Based on these four logic instructions, we can perform a `nand` operation in software simply combining an `and` instruction followed by an `xor` instruction. However, with this solution, every `nand` operation would take 2 cycles (assuming no additional stalls due to other reasons, such as I$ misses). A more efficient approach, requiring only 1 cycle, would be to implement a `nand` UDI using the CorExtend interface. Figure 1 illustrates a scheme of this approach: The CPU outputs the contents of the two source registers through the CorExtend interface; the UDI module performs the *nand* operation; and finally, the result is sent from the UDI module to the Register File through the CorExtend interface.

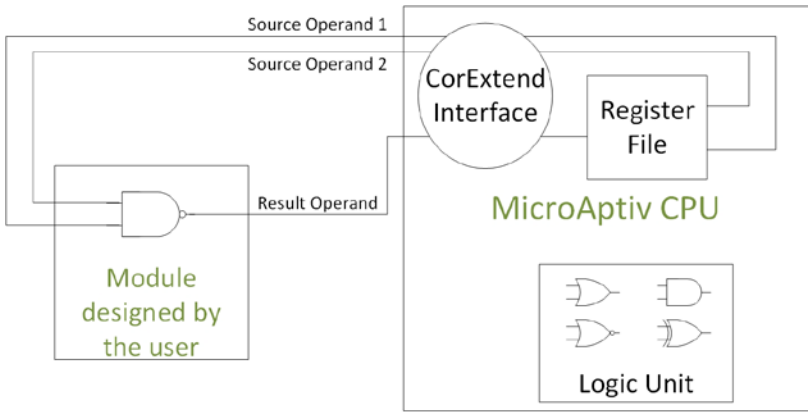

Figure 1. Example of a `nand` UDI.

2. CorExtend instruction requirements and formats

In this section we briefly analyze UDIs requirements and formats (you can find more details in [1]).

a. UDI requirements

UDI instructions defined with the CorExtend interface must accomplish several requirements, as extensively specified in Section 1.3 of [1]. In this section we summarize the main requirements:

- The CorExtend interface only allows the user to implement fixed integer instructions (i.e. jump, branch, load or store instructions are not allowed).
- The destination of the UDI may be a general-purpose register or a register inside the UDI block. Throughout this lab we will focus on the former. Those kind of instructions can complete in a single cycle or in multiple cycles, in which case they must stall the pipeline as we explain in Section 3.

b. UDI formats

UDI instructions defined with the CorExtend interface can use different formats, as specified in Section 1.4 of [1]. Along this lab we always use the same format, shown in Figure 2. Observe that UDIs are included within the *SPECIAL2* instruction category (Table 12.3 of [2]), with a function field equal to 01XXXX. Note also that the order of the three registers (Rs, Rt and Rd) is different than the one used for MIPS Arithmetic-Logic instructions.

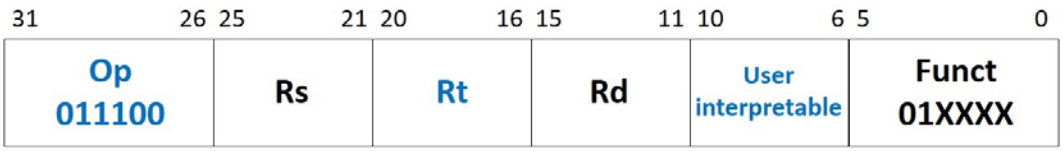

Figure 2. UDI format using three general-purpose register operands plus 5 immediate bits.

3. The CorExtend Interface: signals and operation

In this section we analyze the CorExtend interface in detail. Figure 3 illustrates the location of the CorExtend modules within the RTL hierarchy. The CorExtend interface is implemented in module **m14k_edp_buf_misc_pro**. It communicates the core (**m14k_core**) with the UDI module, located in module **m14k_udi_custom** (note that MIPSfpga provides an example implementation in **m14k_udi_custom** that you can analyze on your own, as proposed in Exercise 4). As shown in Figure 3, module **m14k_udi_custom** is instantiated in module **m14k_top**, a top-level module that also instantiates the processor itself (**m14k_cpu**).

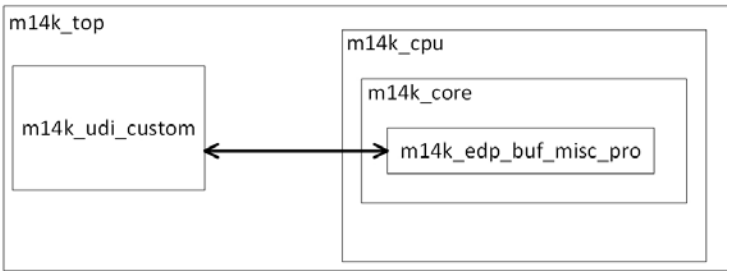

Figure 3. CorExtend UDI module location.

As analyzed in Labs 14 and 15, when an arithmetic or logic instruction is at the E-Stage, the Execution Datapath (module **m14k_edp**) decodes the instruction, reads the register file (RF) and performs the required operation (addition, subtraction, logic *and* operation, etc.). In the case of the UDIs, instruction decoding and the operation performed by the instruction are carried out within the UDI module (**m14k_udi_custom**). For that purpose, several signals must be communicated between module **m14k_edp_buf_misc_pro** and module **m14k_udi_custom**, as described in the next subsection. Subsection b explains the interaction of the UDI module with the pipeline.

a. Signals used by the CorExtend interface

In this subsection we analyze the main signals provided from/to the CorExtend interface (Table 1 of [1] specifies all the signals). We recommend you to analyze file **m14k_edp_buf_misc_pro.v** while reading the explanations.

The signals shown in Table 1 are computed at module **m14k_edp_buf_misc_pro** and are communicated to module **m14k_udi_custom**.

Table 1. Signals computed at m14k_edp_buf_misc_pro and passed to m14k_udi_custom

Module/Signal Name	Description
UDI_ir_e[31:0]	This signal is assigned the value of the Instruction Register (*mpc_ir_e*), and it is communicated to module **m14k_udi_custom** for performing instruction decoding.
UDI_irvalid_e	Validity of the IR contents
UDI_rs_e[31:0] and *UDI_rt_e[31:0]*	These two signals are assigned the value of signals *edp_abus_e* and *edp_bbus_e* respectively, which contain the value read from the register file (signals *rf_adt_e[31:0]* and *rf_bdt_e[31:0]*) or the value forwarded from a later stage in case of a RAW hazard (signals *preabus_e[31:0]* and *prebbus_e[31:0]*). They are communicated to module **m14k_udi_custom** for performing the operation.
UDI_start_e	This signal is assigned the value of *mpc_run_ie*
UDI_greset	This signal is assigned the value of the global reset (signal *greset*)
UDI_gscanenable	This signal is assigned the value of the global scan enable (signal *gscanenable*)

UDI_gclk	This signal is assigned the value of the global clock (signal *gclk*)

The signals shown in Table 2 are computed at module **m14k_udi_custom** and are communicated to the pipeline through module **m14k_edp_buf_misc_pro**.

Table 2. Signals computed at m14k_udi_custom and passed to m14k_edp_buf_misc_pro

Module/Signal Name	Description
UDI_wrreg_e[4:0]	This signal contains the destination register number of the UDI (decoding is performed within module **m14k_udi_custom**). It is inserted into the pipeline at the E-Stage.
UDI_ri_e	When set to high, this signal indicates that the *SPECIAL2* instruction currently being executed is illegal (i.e., reserved). This signal is used by the Master Pipeline Control block (module **m14k_mpc**) within the core to signal an illegal instruction. Note however that this signal is sampled by Master Pipeline Control only if the current instruction is within the *SPECIAL2* range of user-defined instructions (i.e. bits [5:4] of the instruction are 2'b01). It is inserted into the pipeline at the E-Stage.
UDI_rd_m[31:0]	This signal contains the result of the UDI instruction. It is inserted into the pipeline at the M-Stage.
UDI_stall_m	When the UDI requires N cycles to execute (being N>1), *UDI_stall_m* is set to 1 along N cycles, and, as a result, the pipeline is stalled for N cycles. When the result is ready, *UDI_stall_m* is set to 0, and the pipeline resumes. It is inserted into the pipeline at the M-Stage.

b. Operation of the CorExtend interface

In this subsection we describe how the UDI module (**m14k_udi_custom**) interacts with the pipeline through the CorExtend interface (**m14k_edp_buf_misc_pro**). Figure 4 illustrates the main signals communicated between the pipeline and the CorExtend interface.

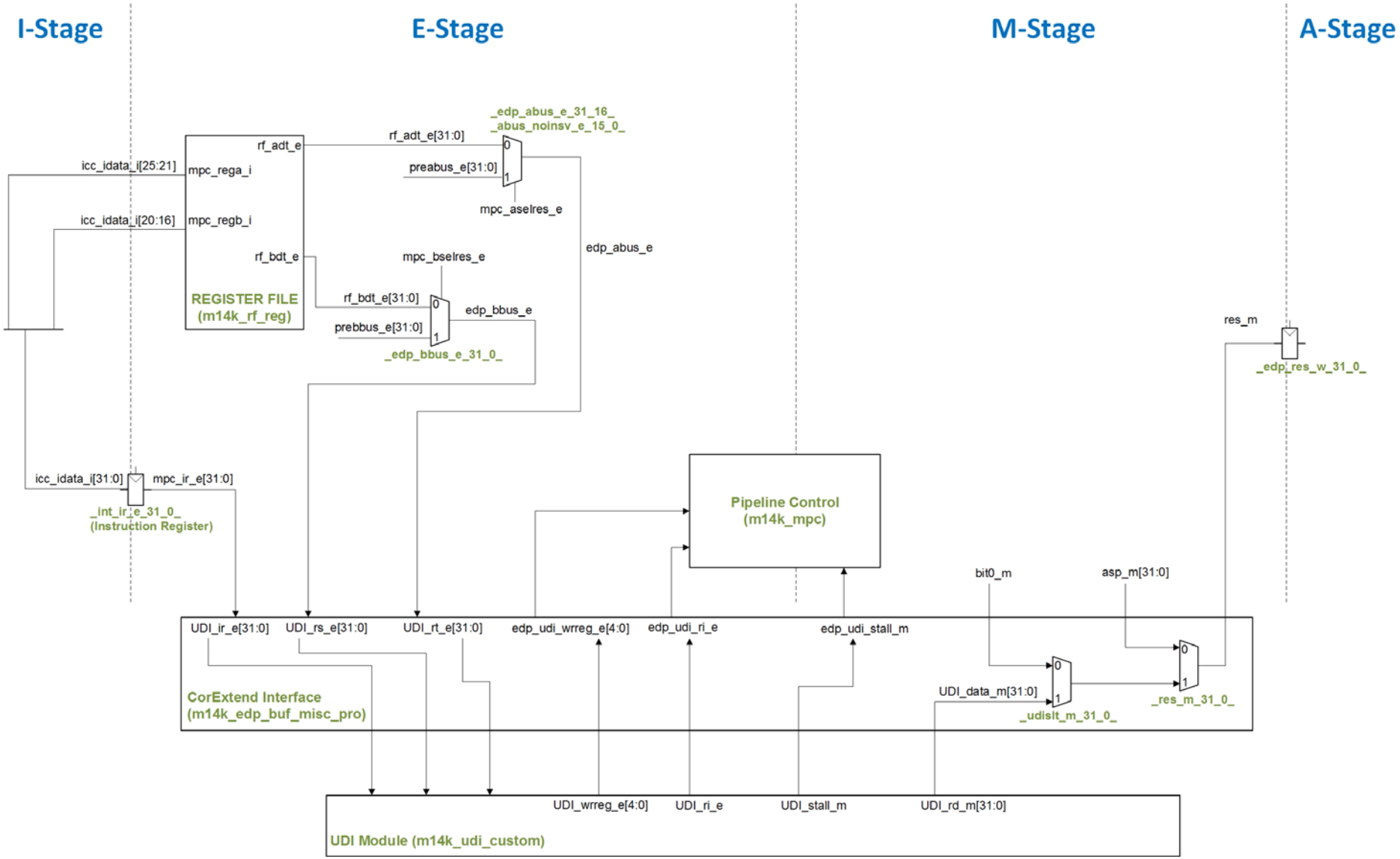

Figure 4. UDI block interaction with the pipeline.

E-Stage

As shown in the figure, at the E-Stage, the UDI module receives, through signal *UDI_ir_e[31:0]*, the 32 bits that constitute the instruction fetched in the I-Stage (signal *mpc_ir_e[31:0]*). The UDI module must decode the instruction internally, according to any of the supported formats (as we said before, in this lab we stick to the one shown in Figure 2). In the case that a *SPECIAL2* instruction is detected (i.e. bits [31:26] of the instruction are 6'b011100) and that the *SPECIAL2* instruction belongs to the range of user-defined instructions (i.e. bits [5:4] of the instruction are 2'b01), the UDI module must take the following actions:

- If bits [3:0] of the instruction correspond to a supported UDI, *UDI_ri_e* must be set to 0. In this case, the UDI module starts executing the instruction (using the operands available in *UDI_rs_e[31:0]* and *UDI_rt_e[31:0]*) and, when the result is ready (depending whether the operation is single or multi-cycle), registers it to *UDI_rd_m[31:0]*.
- If bits [3:0] of the instruction do not correspond to a supported UDI, *UDI_ri_e* must be set to 1, which triggers a Reserved Instruction exception at the Main Pipeline Control block.

In this stage, the UDI module is also required to send the destination register number (signal *UDI_wrreg_e[4:0]*) to the Pipeline Control after the instruction is decoded.

M-Stage

At the M-Stage, for a single-cycle UDI, the result, available in signal *UDI_rd_m[31:0]*, is inserted into the pipeline through signal *res_m[31:0]* (see Figure 2). Note that this signal (*res_m*) can also be assigned with the result of *arithmetic-logic* instructions (signal *asp_m[31:0]*) or with the result of *slt-type* instructions (signal *bit0_m*), as explained in Labs 14 and 15. Moreover, signal *UDI_stall_m* must be set to 0.

In case of a multi-cycle UDI, signal *UDI_stall_m* is asserted at the M-Stage by the UDI module. This indicates to the pipeline that the UDI result is not valid yet and will cause the M-Stage and all earlier stages to stall until the result is valid. When the result is valid, signal *UDI_stall_m* is de-asserted so that the processor can resume execution.

4. Example implementation, simulation and exectution of a `seleqz` UDI in Verilog

In this section we guide you on the process of including a `seleqz` UDI using the CorExtend interface, simulating it on XSIM, and executing it on the board. This instruction, which you also

implement in Exercise 3 of Lab 15 by changing the core, uses the format from Figure 2. The description of this instruction is the following:

if GPR[rt] = 0 then GPR[rd] ← GPR[rs]
else then GPR[rd] ← 0

a. Implementation

You can follow the next steps for implementing a `seleqz` UDI. Note that the modified Verilog files are provided in folder *Lab19_CorExtend\Example_SELEQZ-CorExtend_Verilog\rtl_up_CorExtend*:

1. Copy the soft-core folder (**rtl-up**) into a new folder (**rtl_up_CorExtend**). Use the new folder in the next steps.
2. In the new folder, expand the capability of the MIPSfpga system so that it can write to the 7-segment displays on the Nexys4 DDR board, as explained in Lab 5.
3. Make a copy of file **m14k_udi_stub.v** to file **m14k_udi_seleqz.v**.
4. The UDI module is instantiated at module **m14k_top**, line 629:

   ```
   `M14K_UDI_MODULE udi (
   …
   ```

   Change the definition of *M14K_UDI_MODULE*, in file **m14k_config.vh**, as follows:

   ```
   `define M14K_UDI_MODULE m14k_udi_custom → `define M14K_UDI_MODULE m14k_udi_seleqz
   ```

5. In the new file **m14k_udi_seleqz.v**:
   1. In line 78, change the name of the module from *m14k_udi_stub* to *m14k_udi_seleqz*.
   2. Signal *UDI_ri_e*:
      i. In line 147, this signal is assigned to 1:

      ```
      assign UDI_ri_e = 1'b1;
      ```

      This will trigger a Reserved Instruction exception if the current instruction is a *SPECIAL2* instruction (i.e. bits [31:26] of the instruction are 6'b011100) in the range reserved for UDIs (i.e. bits [5:4] of the instruction are 2'b01).

      ii. For implementing a `seleqz` instruction we use `UDI0` mnemonic. Thus, we should avoid triggering an exception for a *SPECIAL2* instruction whose bits [5:0] are 6'b010000. As such, we should replace line 147 for the following code:

      ```
      assign UDI_ri_e = (UDI_irvalid_e && UDI_ir_e[31:26]==6'o34 &&
      UDI_ir_e[5:0]==6'o20) ? 1'b0 : 1'b1;
      ```

   3. Signal *UDI_present*:

i. In line 151, this signal is set to 0:

```
assign UDI_present = 1'b0;
```

ii. We are now including a UDI, thus we should replace this line for:

```
assign UDI_present = 1'b1;
```

4. Signal *UDI_wrreg_e*:
   i. In line 149 this signal is set to 1:

```
assign UDI_wrreg_e = 5'b0;
```

   ii. We are using the UDI format shown in Figure 2. Thus, we should replace line 149 for:

```
assign UDI_wrreg_e = UDI_ir_e[15:11];
```

5. Instruction operation: We should include new logic for implementing the functionality for a `seleqz` instruction. Thus, we should define a new signal (*udi_res_e[31:0]*) and the following logic:

```
assign udi_res_e[31:0] = (UDI_rt_e == 0) ? UDI_rs_e : 32'b0;
```

6. Signal *UDI_rd_m*:
   i. In line 148 this signal is set to 0:

```
assign UDI_rd_m = 32'b0;
```

   ii. We have to assign the result of the instruction at the M-Stage. As such, we should replace line 148 for:

```
mvp_cregister_wide #(32) _seleqz_result_(UDI_rd_m[31:0],
UDI_gscanenable, ~UDI_ri_e, UDI_gclk, udi_res_e[31:0]);
```

b. Simulation

In this section we illustrate the simulation of a `seleqz` UDI as it flows through the different stages of the microAptiv pipeline. Folder *Lab19_CorExtend\Example_SELEQZ-CorExtend\Simulation* includes both the waveform configuration file (*testbench_boot_behav.wcfg*) and the simulation source files (folder *SimulationSources*) that you must use in this section. You can view the source program in file **main.c**. This simple program initializes registers t3 and t4, executes two `seleqz` instructions with those registers as source operands, and places the results in t5 and t6. You may also be interested in viewing file **program.dis** which shows the disassembled executable interspersed with the assembly or C source code. If you look for "<main>:" in that file, you can see the assembly instructions, as well as the memory address and machine code of each instruction (Figure 5).

```
80000204:   240b0000    li    t3,0
80000208:   240c0006    li    t4,6
```

```
8000020c:   240d0000    li    t5,0
80000210:   240e0000    li    t6,0
80000214:   718b6810    s32ldd xr0,t4,-1176  // UDI0 $t4, $t3, $t5, 0
80000218:   718c7010    s32ldd xr0,t4,-912   // UDI0 $t4, $t4, $t6, 0
8000021c:   1000ffff    b     8000021c <main+0x18>
```

Figure 5. Example assembly program, with the two UDIs highlighted in blue.

Using the program shown in Figure 5, debug your implementation with Vivado's XSIM simulator. For that purpose, create the new MIPSfpga system as explained in Section 4.a, and follow the steps explained in Section 4 of Lab 14 for creating a Vivado project, configuring and executing the simulation. The fetch of the first `UDI0` instruction is done around time 91440ns, thus configure the simulation runtime as explained in Lab 14. As for the waveform configuration file and the text files for initializing the memories, use the files provided in folder *Lab19_CorExtend\Example_SELEQZ-CorExtend\Simulation*. Recall that you can add new signals to the simulation as explained in Step 3 of Exercise 1 in Lab 14.

Figure 11 illustrates a timing diagram of the different stages of the execution of the `seleqz` instructions.

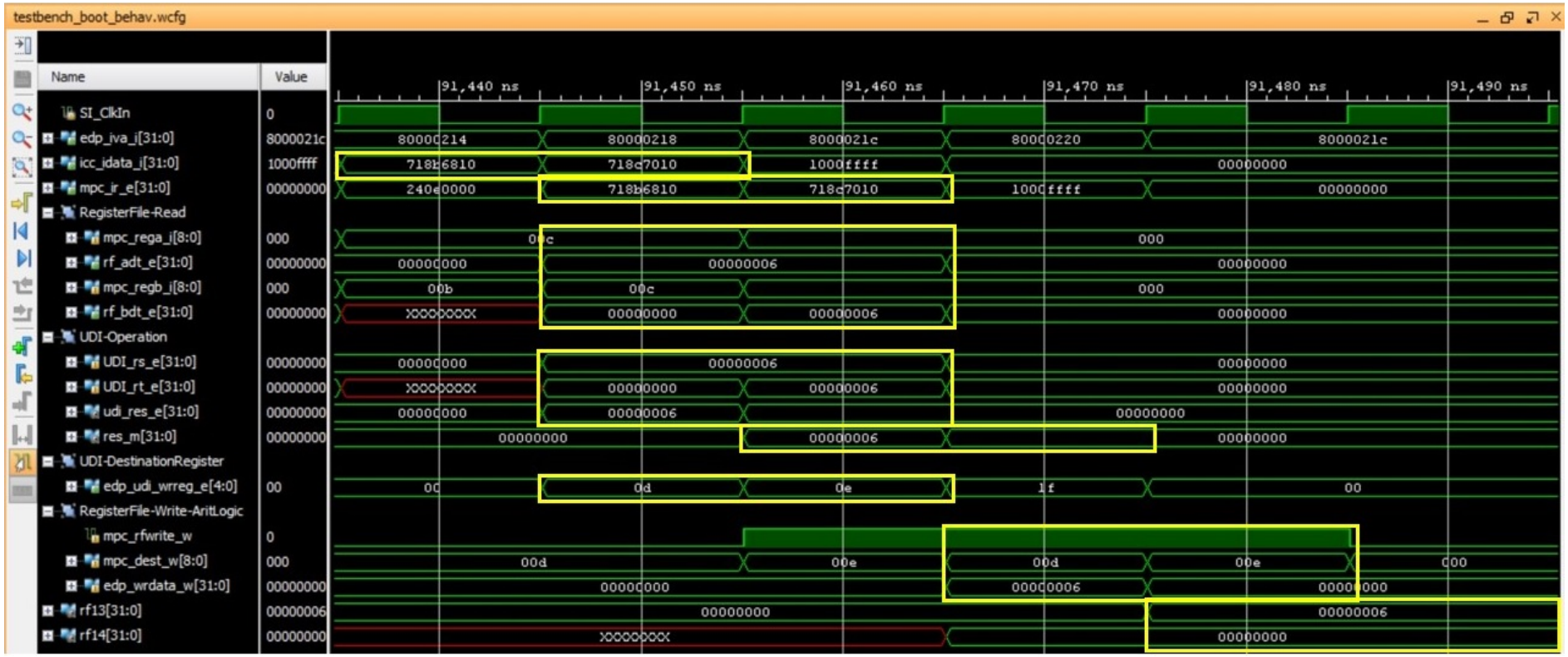

Figure 6. Results of the execution of the code shown in Figure 5.

Next we detail the results obtained in the execution of the program from Figure 5, focusing on the two `seleqz` UDIs.

- 1st and 2nd cycles:
  - The two `seleqz` UDIs are fetched: *icc_idata_i[31:0]*=0x718b6810 for the first UDI and *icc_idata_i[31:0]*=0x718c7010 for the second UDI.
- 2nd and 3rd cycles:
  - The first `seleqz` UDI is executed in the 2nd cycle. *Rt* is 0 (*UDI_rt_e*=0), thus *Rs* is copied to *Rd* (*udi_res_e*=0x6).
  - The second `seleqz` UDI is executed in the 3rd cycle. *Rt* is different than 0 (*UDI_rt_e*=6), thus a 0 is copied to *Rd* (*udi_res_e*=0x0).
- 3rd and 4th cycles:
  - The UDI results are registered to the M-Stage and selected in the multiplexers: *res_m[31:0]*=0x6 for the first UDI and *res_m[31:0]*=0x0 for the second UDI.
- 4th and 5th cycles:
  - The inputs for writing the RF are generated:
    - *mpc_rfwrite_w*=1. Write enabled.
    - *mpc_dest_w*=0x0d for the first UDI and *mpc_dest_w*=0x0e for the second UDI.
    - *edp_wrdata_w*=0x6 for the first UDI and *edp_wrdata_w*=0x0 for the second UDI.
- 5th and 6th cycles:
  - The RF is written:
    - rf13 ($t5) = 0x6
    - rf14 ($t6) = 0x0

c. Execution on the board

Finally, we are going to execute the program from Figure 5 on the board. Follow the steps explained in detail in Exercise 3 of Lab 14 and summarized below, or simply use the *bitfile* provided in *Lab19_CorExtend\Example_SELEQZ-CorExtend_Verilog*.

- Step 1 – Prepare the source files for execution on the board: Modify the program shown above for this example, provided in folder *Lab19_CorExtend\Example_SELEQZ-CorExtend\Simulation\SimulationSources*, by commenting line "`b .;`". Then, re-generate the executable files, as explained in Section 7.2 of the Getting Started Guide, by using the *make* command (the **Makefile** is also provided in *Lab19_CorExtend\Example_SELEQZ-CorExtend\Simulation\SimulationSources*). Finally, analyze on your own the code after the commented branch. This code will output, on

the 7-segment displays, the value of registers $t5 (switches equal to 0) and $t6 (switches equal to 1).

- Step 2 – Synthesize the new processor, as explained in Step 3 – Lab 1. If necessary, set module **mfp_nexys4_ddr** as the top module, by right-clicking on it in the **Project Manager** and selecting **Set as Top**.
- Step 3 – Program the FPGA board, as explained in Step 4 – Lab 1.
- Step 4 – Download the program to the board using the script **loadMIPSfpga.bat**, as explained in Step 3 – Section 2 of Lab 2: The program will start running on the board, and you will be able to see the value of register $t5 (0x6 in the example) when the switches are equal to 0, and the value of register $t6 (0x0 in the example) when the switches are equal to 1 on the 7-segment displays.
- Step 5 – Debug the program as explained in Step 4 – Section 2 of Lab 2: Figure 12 illustrates the register file contents after the execution of the assembly code from Figure 5. Observe that, as expected, t5=0x6 and t6=0x0 after the execution of the two `seleqz` UDIs included in the executed program.

```
(gdb) monitor reset halt
JTAG tap: mAUP.cpu tap/device found: 0x00000001 (mfg: 0x000 (<invalid>), part: 0x0000, ver: 0x0)
target halted in MIPS32 mode due to debug-request, pc: 0xbfc00000
(gdb) b *0x80000214
Breakpoint 1 at 0x80000214: file main.c, line 8.
(gdb) c
Continuing.

[Remote target] #1 stopped.
0x80000214 in main () at main.c:8
8           asm volatile
(gdb) i r
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 00000000 00000000 80000280 00000000 00000002 80001000 00000000
           t0       t1       t2       t3       t4       t5       t6       t7
 R8   80000204 00000002 00000000 00000000 00000006 00000000 00000000 00000000
           s0       s1       s2       s3       s4       s5       s6       s7
 R16  9fc0013c 00000000 00000000 00000000 00000000 00000000 00000000 00000000
           t8       t9       k0       k1       gp       sp       s8       ra
 R24  00000000 00000000 00000000 00000000 80008280 8003fff0 00000000 9fc001a4
         status       lo       hi badvaddr    cause       pc
      00000000 00000100 00000000 00000000 00000000 80000214
(gdb) stepi
0x80000218      8           asm volatile
(gdb) stepi
19          asm volatile
(gdb) i r
          zero       at       v0       v1       a0       a1       a2       a3
 R0   00000000 00000000 00000000 80000280 00000000 00000002 80001000 00000000
           t0       t1       t2       t3       t4       t5       t6       t7
 R8   80000204 00000002 00000000 00000000 00000006 00000006 00000000 00000000
           s0       s1       s2       s3       s4       s5       s6       s7
 R16  9fc0013c 00000000 00000000 00000000 00000000 00000000 00000000 00000000
           t8       t9       k0       k1       gp       sp       s8       ra
 R24  00000000 00000000 00000000 00000000 80008280 8003fff0 00000000 9fc001a4
         status       lo       hi badvaddr    cause       pc
      00000000 00000100 00000000 00000000 00000000 8000021c
(gdb)
```

Figure 7. Register file contents after the execution of the program from Figure 5.

5. Exercises

Exercise 1. Implement a nand UDI

The introduction illustrated a diagram of a nand UDI. In this exercise, you will implement this new instruction using the CorExtend Interface.

i. Implementation of a nand UDI

Follow the next steps for implementing a seleqz UDI (we recommend you to work with the Vivado project created in Section 4):

1. Start by making a copy of file **rtl_up_CorExtend\m14k_udi_seleqz.v** to file **rtl_up_CorExtend\m14k_udi_seleqz_nand.v**.
2. The UDI module is instantiated at module **m14k_top**, line 629:

   ```
   `M14K_UDI_MODULE udi (
   …
   ```

   Change the definition of *M14K_UDI_MODULE*, at file **m14k_config.vh**, as follows:

   ```
   `define M14K_UDI_MODULE m14k_udi_seleqz → `define M14K_UDI_MODULE m14k_udi_seleqz_nand
   ```

3. In file **m14k_udi_seleqz_nand.v**:
   1. Change the name of the module from *m14k_udi_seleqz* to *m14k_udi_seleqz_nand*.
   2. Implement the new nand UDI following a similar process as the one explained in the previous section for a seleqz UDI. Maintain the UDI0 mnemonic for the seleqz UDI and use the UDI1 mnemonic (*SPECIAL2* instruction with function field of 010001, according to document Table 12.3 in [2]) for the nand UDI.
4. In the Vivado project created in Section 4, we must add a new design source file. In Vivado, click on **Add Sources** in the **Flow Navigator** window on the left, select the **Add or create design sources** option, and then click **Next**. Click on **Add Files…**, select **All Files** in the *Files of type* filter, browse to the new **m14k_udi_seleqz_nand.v** file, select it, and click **OK** (Figure 13). Leave the *Copy sources into project box* unselected, but leave the *Include all design sources for simulation* box checked. Then click **Finish**. The new design source file should be added to the project hierarchy.

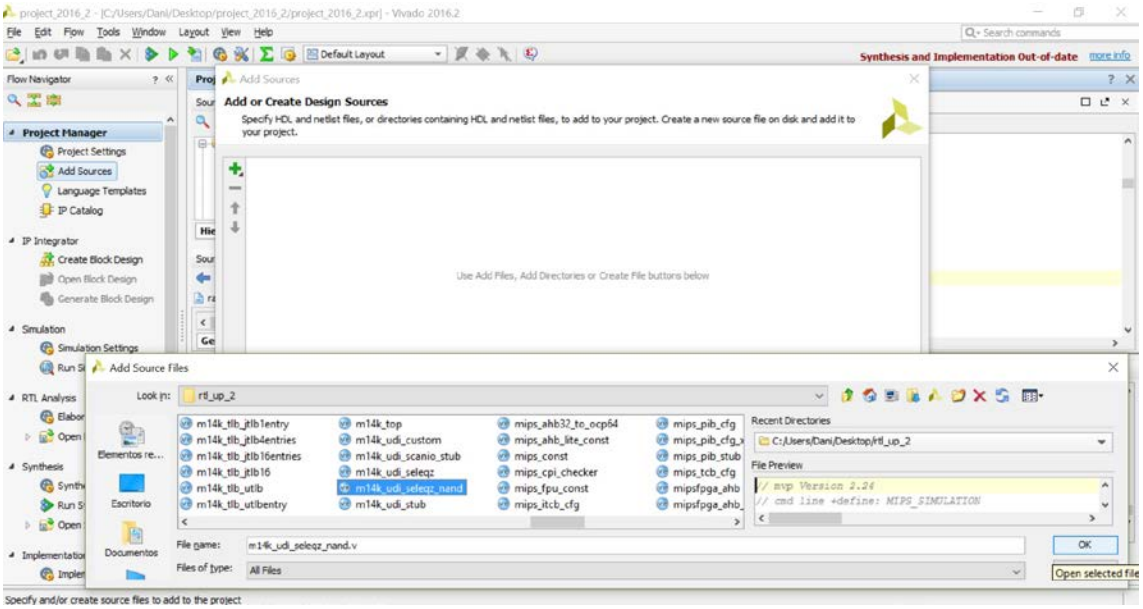

Figure 8. Add *m14k_udi_seleqz_nand.v* file as a new design source.

ii. Simulation of a nand UDI

For simulating the nand instruction, you must substitute the old text files that initialize the memories with the new ones, provided in folder *Lab19_CorExtend\Exercise_NAND-CorExtend\Simulation\SimulationSources*. For that purpose, follow the instructions provided in Exercise 1 of Lab 14.

You can find the program in the **main.c** file of folder *Lab19_CorExtend\Exercise_NAND-CorExtend\Simulation\SimulationSources*, where everything is ready (the *.elf* file, the text files for initializing memory, etc.).

```
"   li $t3, 2;"          // Initialize first source operand
"   li $t4, 3;"          // Initialize second source operand
"   li $t5, 0;"          // Initialize destination operand
"   nop;"                // Avoid RAW dependencies
"   UDI1 $t4, $t3, $t5, 0;"
"   b .;"                // Stay here"
```

iii. Execution on the board of a nand UDI

Follow the steps explained in Section 4.c to execute on the board the program containing a nand instruction.

Exercise 2. Implement a seq UDI

MicroAptiv includes the `slt` instruction, which has the following description:

Reg[Rd] = (Reg[Rs] < Reg[Rs])

A similar instruction to `slt`, called `seq`, not included in microAptiv, is presented in Exercise 4.2.a of [3], with the following description:

Reg[Rd] = (Reg[Rs] == Reg[Rs])

In this exercise you must implement a `seq` UDI in MIPSfpga using the CorExtend Interface. As in Exercise 1, implement the new UDI in a new file called **m14k_udi_seleqz_nand_seq.v** using `UDI2` mnemonic (*SPECIAL2* instruction with function field of 010010, according to document Table 12.3 in [2]), incorporate the new file into your design sources, and modify **m14k_config.vh** accordingly. Then, simulate the new instruction as explained in Exercise 1 (in folder *Lab19_CorExtend**Exercise_SEQ-CorExtend**Simulation* everything is provided). Finally, execute the new instruction on the board.

Exercise 3. Implement the `seleqz`, `nand` and `seq` UDIs in VHDL

Given that UDIs are implemented in an independent module, we can implement them in any hardware description language that we desire, such as VHDL. Thus, implement the three instructions implemented above (`seleqz`, `nand` and `seq`) in VHDL, and then simulate them on XSIM and execute them on the FPGA board.

Exercise 4. Analyze and test the example implementation

Section 6.2 of [1] explains a "Pipelined Bit Swap with Local UDI State" sample implementation, which is included in file **m14k_udi_custom.v**. You can first analyze this implementation, and then simulate it on Vivado XSIM and synthesize and test it on the board.

Exercise 5. Analyze and test the implementation proposed in [4]

In [4] a DSP accelerator is implemented using the CorExtend Interface. You can first analyze this implementation, and then simulate it on Vivado XSIM and synthesize and test it on the board.

Exercise 6. Implement an FPU using the CorExtend Interface and test it with a simple algorithm

The microAptiv processor configuration used in MIPSfpga does not include a floating-point unit (FPU). In this exercise you are going to include an IEEE754 single-precision FPU, made up of an adder, a multiplier and a divider, using the CorExtend Interface. Then, you will implement two algorithms, one using a floating point unit and the other one using software emulation for floating point operations. Finally, you will compare both implementations using performance counters. Follow the next steps:

Step 1. Create an FPU

You have several options:

1. You can implement your own IEEE754 single-precision FPU in Verilog or VHDL.

2. You can download an IEEE754 single-precision FPU. There are many open-source alternatives in the Internet, both in Verilog and in VHDL.
3. You can use the IEEE754 single-precision FPU provided with the package (*Lab19_CorExtend\FPU\FPU_verilog-files*), which was obtained from https://github.com/dawsonjon/fpu. It includes an IEEE754 single-precision multi-cycle adder (**adder.v**), an IEEE754 single-precision multi-cycle multiplier (**multiplier.v**) and an IEEE754 single-precision multi-cycle divider (**divider.v**). The following copyright applies to this software:

   Copyright (c) 2012 Jonathan P Dawson.
   Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
   The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
   THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Step 2. Implement three new UDIs

Implement three new UDIs, using mnemonics `UDI3`, `UDI4` and `UDI5`, where `UDI3` performs multi-cycle floating-point addition, `UDI4` performs multi-cycle floating-point multiplication, and `UDI5` performs multi-cycle floating-point division. The new UDIs use the format specified in Figure 2. Note that both the source operands and the destination operand are stored in the general purpose register file. Note that, given that the floating point operations require multiple cycles, you need to use signal *UDI_stall_m* (Table 2) for stalling the processor until the result of the floating point operation is ready.

Step 3. Simulate and execute on the board a simple program

Create a simple program for testing your new UDIs both in simulation and on the board. We are providing an example program in *Lab19_CorExtend\FPU\Basic_Step3\SimulationSources* (Figure 18), where $t3 and $t4 are first initialized and then the three new UDIs are executed, using

these two registers as operands and interpreting them as single-precision floating-point values. The results, placed in $t5 (addition), $t6 (multiplication) and $t7 (division), are shown through the 7 segment displays (comment the "b ." instruction for that purpose).

```
...
"   lui $t4, 0x4080;"   // = 4
"   lui $t3, 0x4000;"   // = 2
"   lui $t5, 0x0000;"
"   lui $t6, 0x0000;"
"   lui $t7, 0x0000;"
"   nop;"
"   UDI3 $t4, $t3, $t5, 0;" // 4 + 2 = 6 (0x40c00000)
"   nop;"
"   UDI4 $t4, $t3, $t6, 0;" // 4 * 2 = 8 (0x41000000)
"   nop;"
"   UDI5 $t4, $t3, $t7, 0;" // 4 / 2 = 2 (0x40000000)
"   nop;"
...
```

Figure 9. Example assembly program, with the add instruction highlighted in blue.

Note that IEEE754 single-precision has 32 bits, distributed in a sign bit, 8 exponent bits, and 23 bits for the mantissa, organized as follows:

According to this format, normalized numbers are represented as follows:

$$\pm \; 1.M_{23}...M_0 \; \times \; 2^{(E_7...E_0 - 127)}$$

Moreover, de-normalized numbers, which have all exponent bits equal to 0, are represented as follows:

$$\pm \; 0.M_{23}...M_0 \; \times \; 2^{-126}$$

Finally, there are some reserved encodings, such as an encoding with all bits in the exponent and in the mantissa equal to 0, which represents a 0, or all bits in the exponent equal to 1, which represents infinite, if all bits in the mantissa are 0, or several Not-a-Numbers (NaNs), if the mantissa is different than 0.

Thus, in the example from Figure 18, we should obtain the following results:

- t4 = **0100 0000 1000 0000 0000 0000 0000** (0x40800000) = + 1.0 x $2^{129-127}$ = 4.0
- t3 = **0100 0000 0000 0000 0000 0000 0000** (0x40000000) = + 1.0 x $2^{128-127}$ = 2.0
- t5 = t4 + t3 = 6.0 = + 1.5 x 2^2 = **0100 0000 1100 0000 0000 0000 0000** (0x40c00000)
- t6 = t4 x t3 = 8.0 = + 1.0 x 2^3 = **0100 0001 0000 0000 0000 0000 0000** (0x41000000)
- t7 = t4 / t3 = 2.0 = + 1.0 x 2^1 = **0100 0000 0000 0000 0000 0000 0000** (0x40000000)

You can use the following web-page for transforming real numbers to the IEEE754 format and vice versa: https://www.h-schmidt.net/FloatConverter/IEEE754.html.

Step 4. Implement the bisection root-finding method using the new UDIs

Using the UDIs implemented in the previous step, you will now implement the bisection root-finding method. The bisection method is one of the simplest iterative methods for finding a root of a function in a given interval. It is based on the Bolzano theorem, which enunciates that if a function *f(x)* is continuous in an interval [a,b] and its sign is different at the beginning and end of the interval (i.e. *f(a)* x *f(b)* < 0), it must contain a root in that interval (see the example shown in Figure 19, where the red point corresponds to *a*, the blue point corresponds to *b*, and the green point is the root of the function in the interval [a,b]).

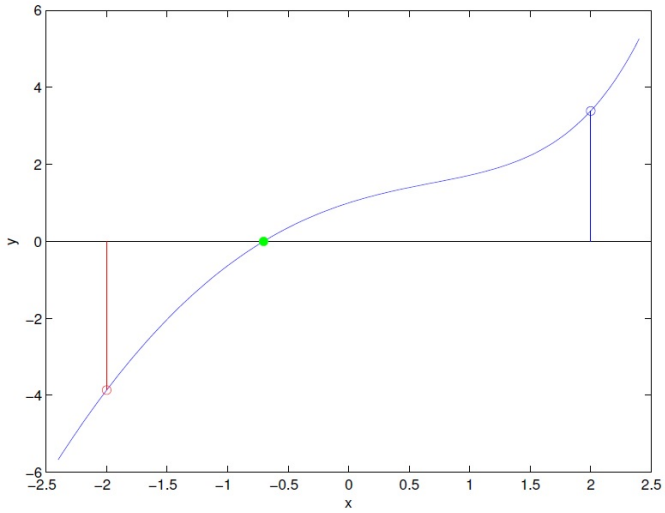

Figure 10. Continuous function with one root in the interval [-2,2].

The bisection method starts with an initial interval [a_0,b_0] for which: *f(a_0)* x *f(b_0)* < 0. In every iteration, the interval is reduced in one half, so that in the new interval [a_i,b_i], again: *f(a_i)* x *f(b_i)* < 0. After several iterations, the middle point of the interval [a_i,b_i] is close enough (based on a tolerance value) to the root of the function. The detailed method is illustrated in Figure 20.

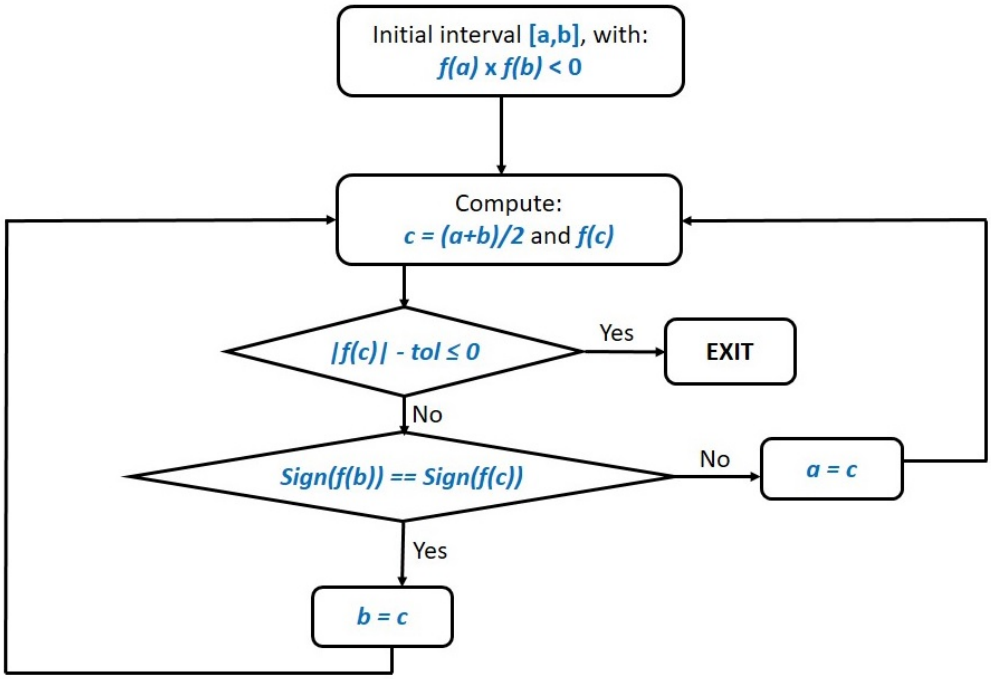

Figure 11. Flow diagram for the bisection method.

Using the floating-point addition (UDI3), multiplication (UDI4) and division (UDI5) included in previous steps, implement the bisection method. Note that you can do all required floating-point arithmetic operations with these new UDIs. As for the first comparison (***|f(c)|-tol ≤ 0***), you can compute the left part of the inequality, mask the result so that only the sign remains, and check if the sign is positive or negative. Note that you can compute the absolute value of *f(c)* by simply applying a mask, given that the sign in this format is explicit. Finally, as for the second comparison (***Sign(f(b)) == Sign(f(c))***), you can simply compare the signs of the two floating-point values, once again by applying a mask.

Folder *Lab19_CorExtend\FPU\Bisection_HW_Step4* provides a skeleton that you have to complete, simulate and debug, and then execute on the board. Note that the skeleton initializes registers $t3 and $t4 to 1.25 and 2.5 respectively. You should evaluate the bisection method for function $f(x)=x^2-2$ in the interval [1.25, 2.5], where the function presents one root, and you can use a tolerance value of 0.01. The initial values for the interval, using the IEEE754 representation, are the following:

- t3 = **0011 1111 1010 0000 0000 0000 0000** (0x3fa00000) = + 1.01 x $2^{127-127}$ = 1.25
- t4 = **0100 0000 0010 0000 0000 0000 0000** (0x40200000) = + 1.01 x $2^{128-127}$ = 2.5

You should obtain the following results:

- 1st iteration: 1.875 (0x3ff00000)
- 2nd iteration: 1.5625 (0x3fc80000)
- 3rd iteration: 1.40625 (0x3fb40000)
- 4th iteration: 1.484375 (0x3fbe0000)
- 5th iteration: 1.4453125 (0x3fb90000)
- 6th iteration: 1.4257812 (0x3fb68000)
- 7th iteration (solution within the tolerance range): **1.4160156 (0x3fb54000)**

Step 5. Implement the bisection root-finding method using floating-point emulation

In this exercise, you must implement the bisection method using the software floating point library (https://gcc.gnu.org/onlinedocs/gccint/Soft-float-library-routines.html) provided by *gcc*. This library can be used on machines which do not have hardware support for floating point. It is used whenever *-msoft-float* is used to disable generation of floating point instructions (note that the provided *makefile* already uses this option).

Folder *Lab19_CorExtend\FPU\Bisection_SW_Step5* provides a similar skeleton to the one used in the previous step, that you have to complete in order to resolve this exercise. As in the previous step, the skeleton is ready to execute the bisection method for function $f(x)=x^2-2$, in the interval [1.25, 2.5] and for a tolerance value of 0.01.

Step 6. Compare the implementations from steps 4 and 5 with the performance counters

Using the performance counters as explained in Lab 13, compare the HW and SW versions of the bisection method developed in steps 4 and 5 respectively. Folder *Lab19_CorExtend\FPU\Bisection_HW-SW-Comparison_Step6* provides the skeleton that you can use for this exercise, which is configured to measure *number of cycles* and *number of instructions completed*.

6. References

[1] "CorExtend Instruction Integrator's Guide for M4K®/4KE®/4KS™ and M14K Family Cores -- MD00324". Document provided with the GSG Package.

[2] "MIPS32® microAptiv™ UP Processor Core Family Software User's Manual -- MD00942".

[3] "Computer Organization and Design". David A. Patterson and John L. Hennesy. Morgan Kaufmann.

[4] "MIPS microAptiv UP Processor CorExtend UDI interface protocol guide". Kirill Zats. https://drive.google.com/file/d/0BzSz4P83rJyRUDAwZ25SeURZSnc/view.