1. 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.

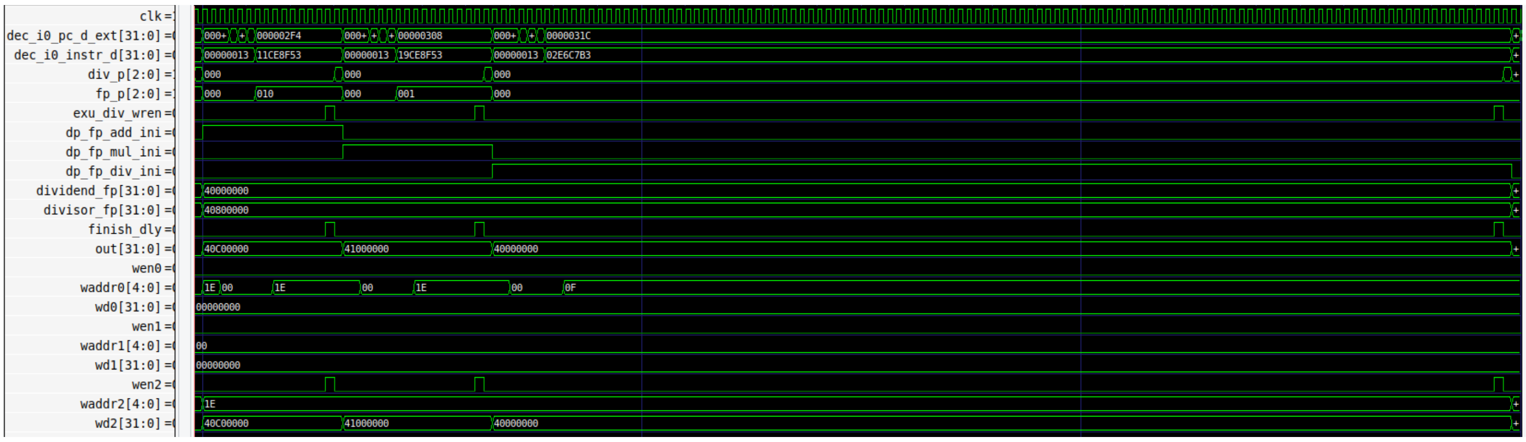
   Solution at: *[RVfpgaBooleanPath]/Labs/RVfpgaLabsSolutions/Lab18/Exercise1*

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

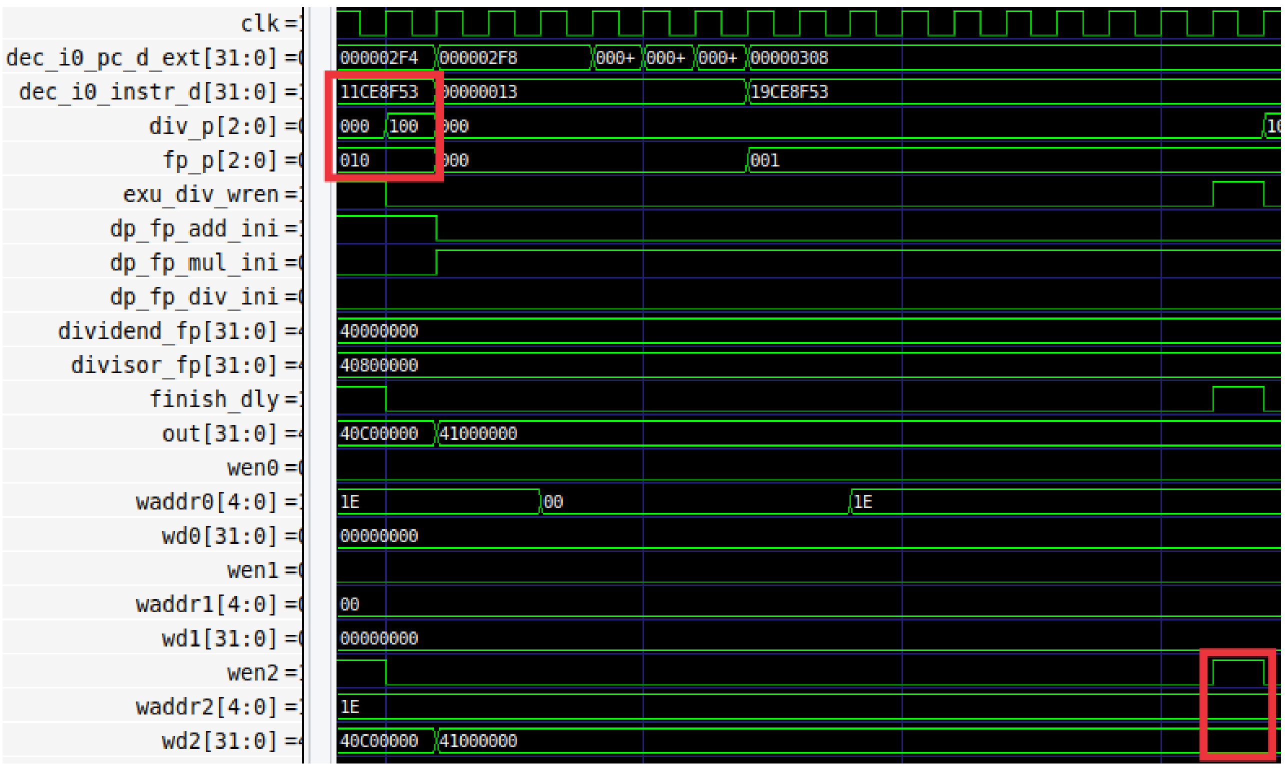
   Catapult project at: *[RVfpgaBooleanPath]/Labs/RVfpgaLabsSolutions/Lab18/Exercise2*

`RVfpga-Trace:`

`One whole iteration – fadd fmul fdiv:`

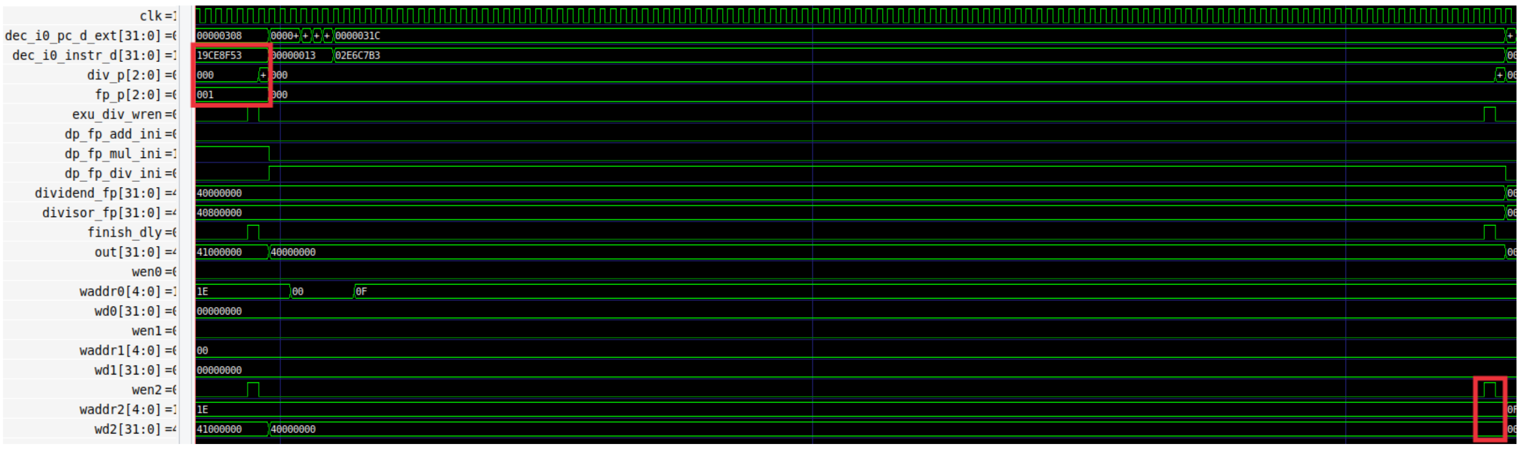


`fmul` instruction:



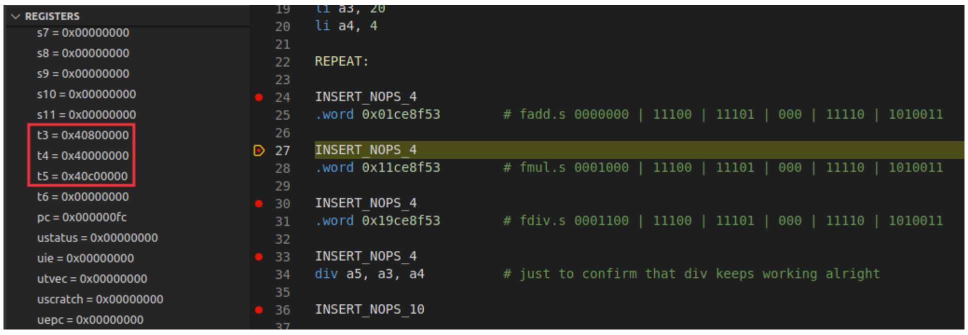
The fmul is executed correctly and obtains the expected result: 0x41000000 which is 8 (4*2).

`fdiv` instruction:



The fdiv is executed correctly and obtains the expected result: 0x40000000 which is 2 (4/2).

Boolean Board:

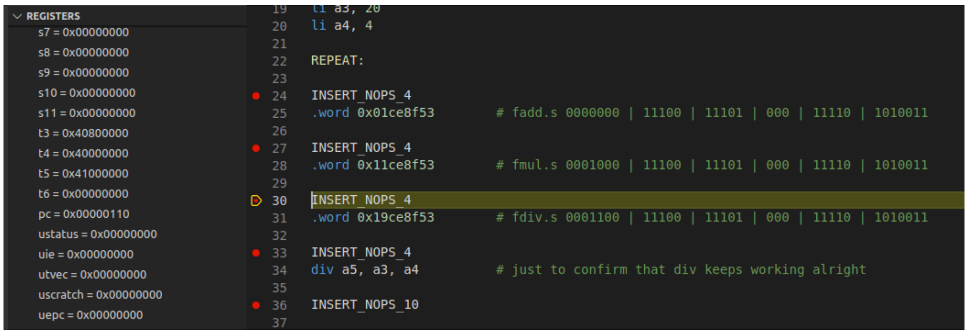


```
REGISTERS
s7 = 0x00000000
s8 = 0x00000000
s9 = 0x00000000
s10 = 0x00000000
s11 = 0x00000000
t3 = 0x40800000
t4 = 0x40000000
t5 = 0x40c00000
t6 = 0x00000000
pc = 0x000000fc
ustatus = 0x00000000
uie = 0x00000000
utvec = 0x00000000
uscratch = 0x00000000
uepc = 0x00000000
```

```
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
```

The result of the fadd is correct:

0x40800000 (t3) + 0x40000000 (t4) = 0x40c00000 (t5)

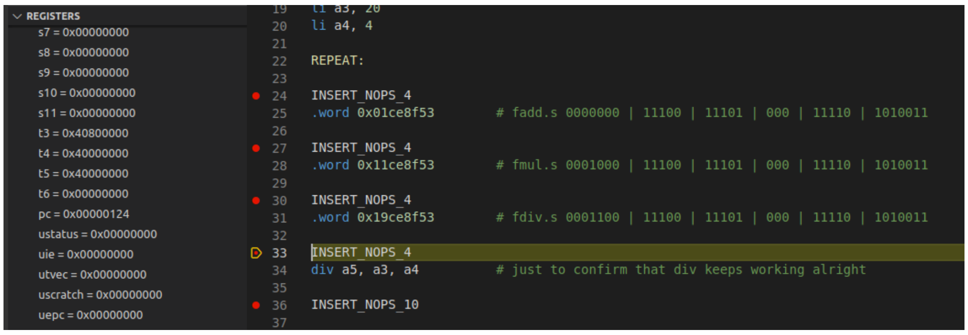


```
REGISTERS
s7 = 0x00000000
s8 = 0x00000000
s9 = 0x00000000
s10 = 0x00000000
s11 = 0x00000000
t3 = 0x40800000
t4 = 0x40000000
t5 = 0x41000000
t6 = 0x00000000
pc = 0x00000110
ustatus = 0x00000000
uie = 0x00000000
utvec = 0x00000000
uscratch = 0x00000000
uepc = 0x00000000
```

```
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
```

The result of the fmul is correct:

0x40800000 (t3) + 0x40000000 (t4) = 0x41000000 (t5)

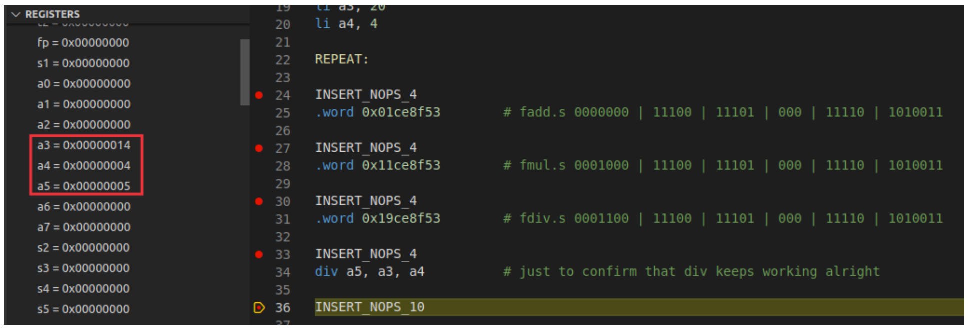


```
REGISTERS
s7 = 0x00000000
s8 = 0x00000000
s9 = 0x00000000
s10 = 0x00000000
s11 = 0x00000000
t3 = 0x40800000
t4 = 0x40000000
t5 = 0x40000000
t6 = 0x00000000
pc = 0x00000124
ustatus = 0x00000000
uie = 0x00000000
utvec = 0x00000000
uscratch = 0x00000000
uepc = 0x00000000
```

```
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
```

The result of the fdiv is correct:

0x40800000 (t3) + 0x40000000 (t4) = 0x40000000 (t5)



```
REGISTERS
fp = 0x00000000
s1 = 0x00000000
a0 = 0x00000000
a1 = 0x00000000
a2 = 0x00000000
a3 = 0x00000014
a4 = 0x00000004
a5 = 0x00000005
a6 = 0x00000000
a7 = 0x00000000
s2 = 0x00000000
s3 = 0x00000000
s4 = 0x00000000
s5 = 0x00000000
```

```
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
```

The result of the div is correct:

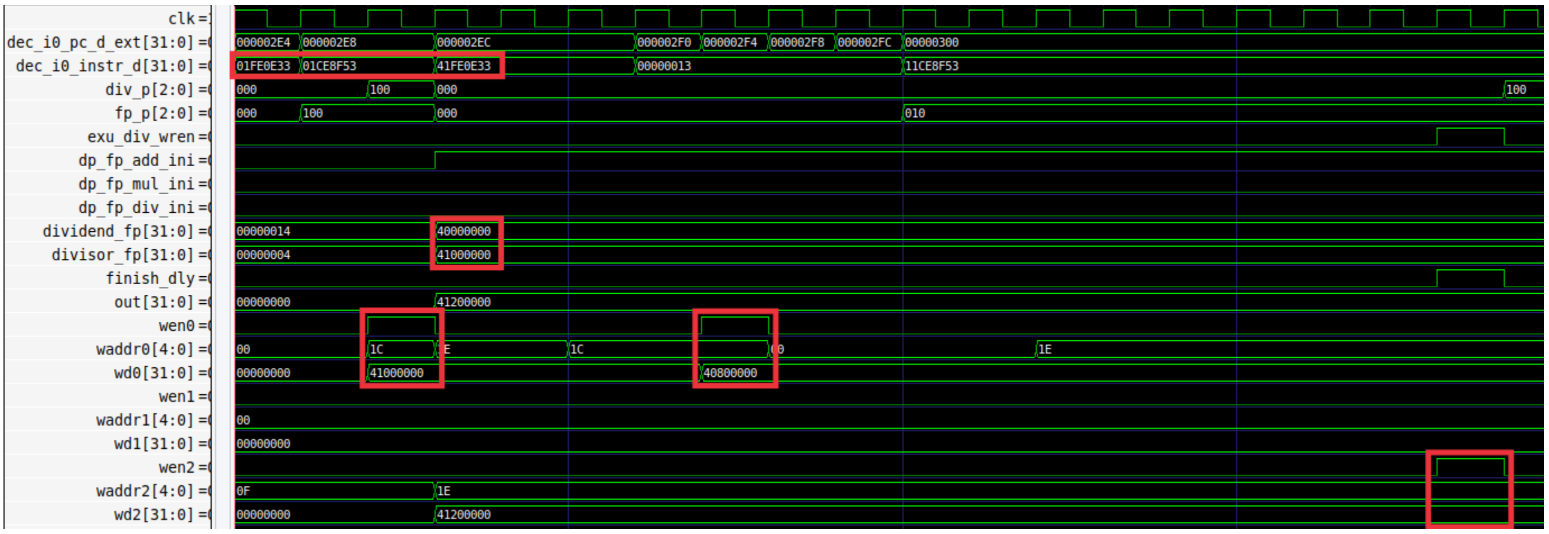
0x14 (a3) + 0x4 (a4) = 0x5 (a5)

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.

Based in the Test Program used in the lab, we insert two dependent instructions, one before the fadd and another after the fadd.

```
main:

li t3, 0x40800000
li t4, 0x40000000
li t6, 0x800000

li a3, 20
li a4, 4

REPEAT:

INSERT_NOPS_4
add t3, t3, t6
.word 0x01ce8f53 # fadd.s t5, t3, t4
sub t3, t3, t6
```

add: 0x40800000 + 0x00800000 = 0x41000000 (8)
fadd: 0x41000000 (8) + 0x40000000 (2) = 0x41200000 (10)
sub: 0x41200000 - 0x00800000 = 0x41000000 (8)



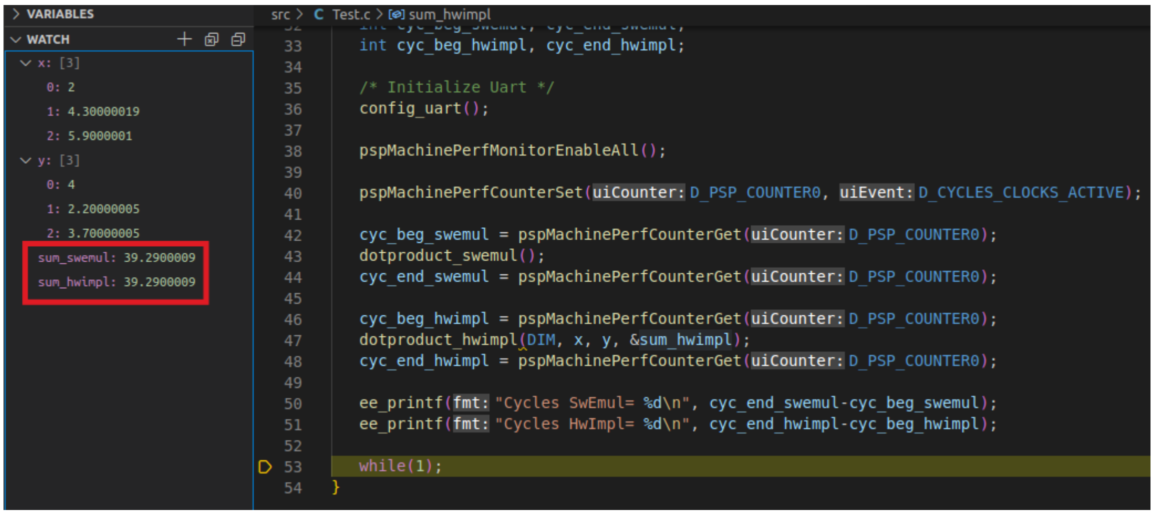
- The integer add and sub are computed correctly.
- The values are also correctly forwarded.
- The result of the fadd is correct (0x41200000).

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.

Catapult project at:
[RVfpgaBooleanPath]/Labs/RVfpgaLabsSolutions/Lab18/Exercise4

Boolean Board:

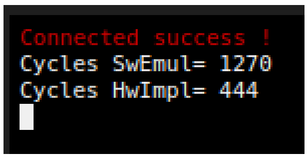


```
x: [3]
  0: 2
  1: 4.30000019
  2: 5.9000001
y: [3]
  0: 4
  1: 2.20000005
  2: 3.70000005
sum_swemul: 39.2900009
sum_hwimpl: 39.2900009
```

```
    int cyc_beg_hwimpl, cyc_end_hwimpl;

    /* Initialize Uart */
    config_uart();

    pspMachinePerfMonitorEnableAll();

    pspMachinePerfCounterSet(uiCounter: D_PSP_COUNTER0, uiEvent: D_CYCLES_CLOCKS_ACTIVE);

    cyc_beg_swemul = pspMachinePerfCounterGet(uiCounter: D_PSP_COUNTER0);
    dotproduct_swemul();
    cyc_end_swemul = pspMachinePerfCounterGet(uiCounter: D_PSP_COUNTER0);

    cyc_beg_hwimpl = pspMachinePerfCounterGet(uiCounter: D_PSP_COUNTER0);
    dotproduct_hwimpl(DIM, x, y, &sum_hwimpl);
    cyc_end_hwimpl = pspMachinePerfCounterGet(uiCounter: D_PSP_COUNTER0);

    ee_printf(fmt: "Cycles SwEmul= %d\n", cyc_end_swemul-cyc_beg_swemul);
    ee_printf(fmt: "Cycles HwImpl= %d\n", cyc_end_hwimpl-cyc_beg_hwimpl);

    while(1);
}
```

The result of the dot product, both in the program that uses software emulation and in the program that uses hardware execution, is the same and correct:

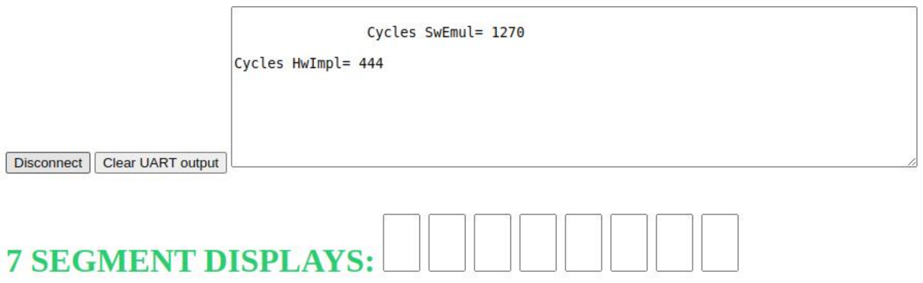
{2.0, 4.3, 5.9} * {4.0, 2.2, 3.7} = 2*4 + 4.3*2.2 + 5.9*3.7 = 39.29

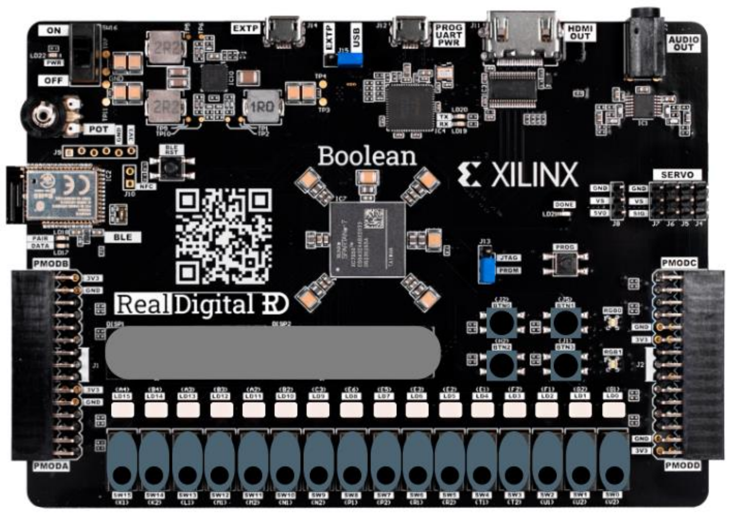


```
Connected success !
Cycles SwEmul= 1270
Cycles HwImpl= 444
```

The program needs in the hw implementation around 1/3 of the cycles needed in the sw implementation.

RVfpgaEL2-ViDBo:

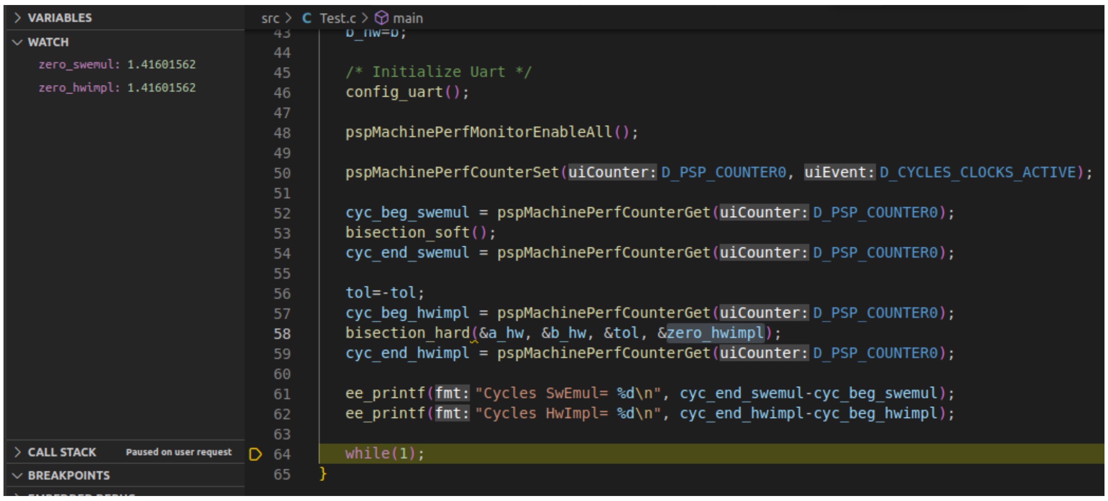

```
                Cycles SwEmul= 1270
Cycles HwImpl= 444
```



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.

   Catapult project at:
   [RVfpgaBooleanPath]/Labs/RVfpgaLabsSolutions/Lab18/Exercise5

Boolean Board:

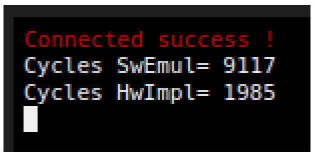


```
zero_swemul: 1.41601562
zero_hwimpl: 1.41601562
```

```
    b_hw=b;

    /* Initialize Uart */
    config_uart();

    pspMachinePerfMonitorEnableAll();

    pspMachinePerfCounterSet(uiCounter: D_PSP_COUNTER0, uiEvent: D_CYCLES_CLOCKS_ACTIVE);

    cyc_beg_swemul = pspMachinePerfCounterGet(uiCounter: D_PSP_COUNTER0);
    bisection_soft();
    cyc_end_swemul = pspMachinePerfCounterGet(uiCounter: D_PSP_COUNTER0);

    tol=-tol;
    cyc_beg_hwimpl = pspMachinePerfCounterGet(uiCounter: D_PSP_COUNTER0);
    bisection_hard(&a_hw, &b_hw, &tol, &zero_hwimpl);
    cyc_end_hwimpl = pspMachinePerfCounterGet(uiCounter: D_PSP_COUNTER0);

    ee_printf(fmt: "Cycles SwEmul= %d\n", cyc_end_swemul-cyc_beg_swemul);
    ee_printf(fmt: "Cycles HwImpl= %d\n", cyc_end_hwimpl-cyc_beg_hwimpl);

    while(1);
}
```

The result of the bisection algorithm, both in the program that uses software emulation and in the program that uses hardware execution, is the same and correct (within the tolerance of 0.01):

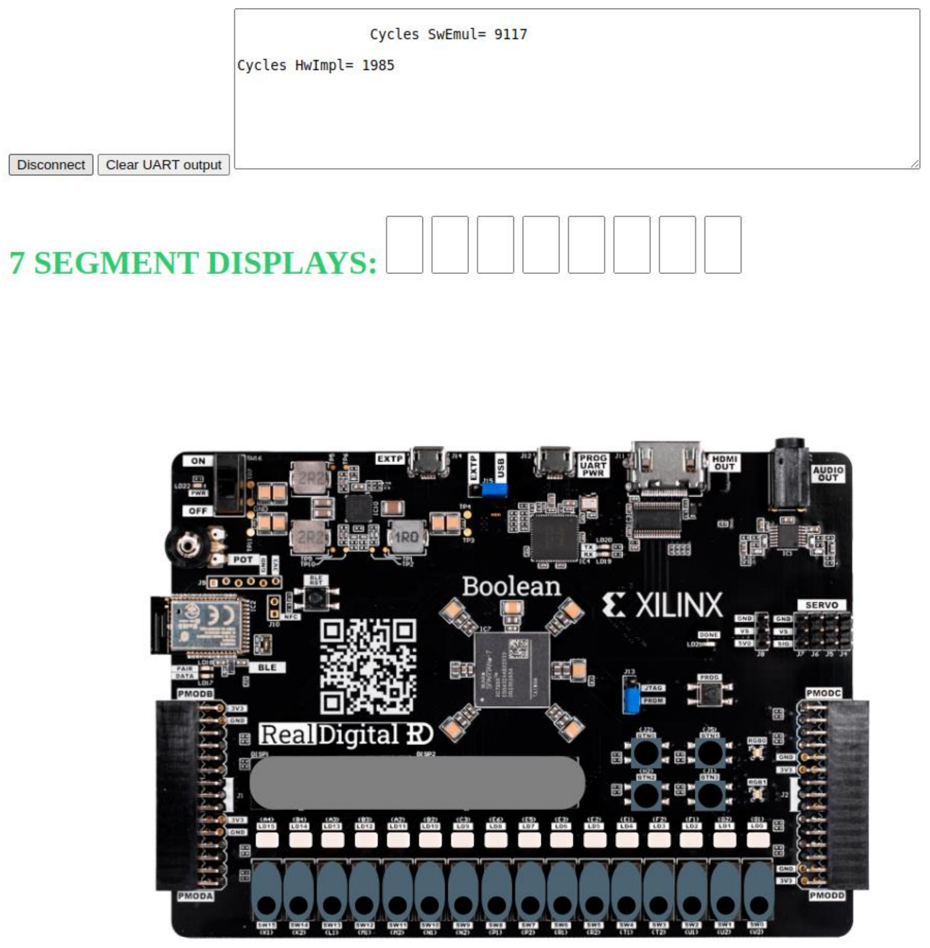
> The function is f(x) = x^2-2, and the initial points are 1.25 and 2.5. There is a zero at x=1.4142



```
Connected success !
Cycles SwEmul= 9117
Cycles HwImpl= 1985
```

The program needs in the hw implementation around 1/5 of the cycles needed in the sw implementation.

RVfpgaEL2-ViDBo:



```
                Cycles SwEmul= 9117
Cycles HwImpl= 1985
```

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

   Solution not provided for this exercise.

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.

   Solution not provided for this exercise.

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

   Solution not provided for this exercise.

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

Solution not provided for this exercise.