TASKS

TASK: You can perform a similar study for the `mul` instruction as the one performed in Lab 12 for arithmetic-logic instructions: view the flow of the instruction through the pipeline stages, analyse the control bits (remember from Appendix D of Lab 11 that there is a specific structure type for the `mul` instruction called `mul_pkt_t`, and there is a signal defined in module **dec_decode_ctl** called `mul_p`), etc.

Solution not provided.

TASK: Inspect the Verilog code from **exu_mul_ctl** and see how the multiplication is computed. Remember that RISC-V includes 4 multiply instructions (`mul`, `mulh`, `mulhsu` and `mulhu`), and all of them must be supported by the hardware.

As an optional exercise, replace the Multiply Unit with your own unit or one from the Internet.

```
70   // -------------------------- Input flops    -------------------------------
71
72   rvdffs      #(1)  valid_e1_ff      (.*, .din(mp.valid),                  .dout(valid_e1),              .clk(active_clk),        .en(~freeze));
73
74   rvdff_fpga  #(1)  rs1_sign_e1_ff   (.*, .din(mp.rs1_sign),               .dout(rs1_sign_e1),           .clk(exu_mul_c1_e1_clk), .clken(mul_c1_e1_clken), .rawclk(clk));
75   rvdff_fpga  #(1)  rs2_sign_e1_ff   (.*, .din(mp.rs2_sign),               .dout(rs2_sign_e1),           .clk(exu_mul_c1_e1_clk), .clken(mul_c1_e1_clken), .rawclk(clk));
76   rvdff_fpga  #(1)  low_e1_ff        (.*, .din(mp.low),                    .dout(low_e1),                .clk(exu_mul_c1_e1_clk), .clken(mul_c1_e1_clken), .rawclk(clk));
77   rvdff_fpga  #(1)  ld_rs1_byp_e1_ff (.*, .din(mp.load_mul_rs1_bypass_e1), .dout(load_mul_rs1_bypass_e1), .clk(exu_mul_c1_e1_clk), .clken(mul_c1_e1_clken), .rawclk(clk));
78   rvdff_fpga  #(1)  ld_rs2_byp_e1_ff (.*, .din(mp.load_mul_rs2_bypass_e1), .dout(load_mul_rs2_bypass_e1), .clk(exu_mul_c1_e1_clk), .clken(mul_c1_e1_clken), .rawclk(clk));
79
80   rvdffe  #(32) a_e1_ff          (.*, .din(a[31:0]),                   .dout(a_ff_e1[31:0]),          .en(mul_c1_e1_clken));
81   rvdffe  #(32) b_e1_ff          (.*, .din(b[31:0]),                   .dout(b_ff_e1[31:0]),          .en(mul_c1_e1_clken));
82
83
84
85   // -------------------------- E1 Logic Stage --------------------------------
86
87   assign a_e1[31:0]             = (load_mul_rs1_bypass_e1)  ?  lsu_result_dc3[31:0]  :  a_ff_e1[31:0];
88   assign b_e1[31:0]             = (load_mul_rs2_bypass_e1)  ?  lsu_result_dc3[31:0]  :  b_ff_e1[31:0];
89
90   assign rs1_neg_e1             =  rs1_sign_e1 & a_e1[31];
91   assign rs2_neg_e1             =  rs2_sign_e1 & b_e1[31];
92
93
94   rvdffs       #(1)  valid_e2_ff      (.*, .din(valid_e1),                   .dout(valid_e2),          .clk(active_clk),        .en(~freeze));
95
96   rvdff_fpga   #(1)    low_e2_ff      (.*, .din(low_e1),                     .dout(low_e2),            .clk(exu_mul_c1_e2_clk), .clken(mul_c1_e2_clken), .rawclk(clk));
97
98   rvdffe  #(33) a_e2_ff          (.*, .din({rs1_neg_e1, a_e1[31:0]}),  .dout(a_ff_e2[32:0]),          .en(mul_c1_e2_clken));
99   rvdffe  #(33) b_e2_ff          (.*, .din({rs2_neg_e1, b_e1[31:0]}),  .dout(b_ff_e2[32:0]),          .en(mul_c1_e2_clken));
100
101
102
103  // ---------------------- E2 Logic Stage -------------------------
104
105
106  logic signed [65:0]  prod_e2;
107
108  assign prod_e2[65:0]          =  a_ff_e2  *  b_ff_e2;
109
110  rvdff_fpga  #(1)    low_e3_ff      (.*, .din(low_e2),                    .dout(low_e3),                .clk(exu_mul_c1_e3_clk), .clken(mul_c1_e3_clken), .rawclk(clk));
111
112  rvdffe      #(64) prod_e3_ff       (.*, .din(prod_e2[63:0]),             .dout(prod_e3[63:0]),         .en(mul_c1_e3_clken));
113
114
115
116  // ----------------------- E3 Logic Stage ------------------------
117
118
119  assign out[31:0]            = low_e3  ?  prod_e3[31:0]  :  prod_e3[63:32];
```

- The inputs and control bits produced at the decode stage are registered in lines 72-81.

M1:

- In case of a data dependency between the multiplication and a previous load, a forwarding takes place in lines 87-88.
- Moreover, the treatment of the sign of the input operands is determined in lines 90-91. Remember that RISC-V includes three versions of the “multiply high” operation: `mulh`, `mulhsu` and `mulhu`.

- These values are propagated to M2.

M2:
- The actual multiplication is performed in line 108.

M3:
- The low/high part is returned in `out[31:0]` in line 119. The low part is selected in case of a `mul` instruction, whereas the high part is selected in case of any of the three `mulh` instructions.

TASK: Verify that this pair of 32 bits (0x03de02b3 and 0x03ff0333) correspond to instructions `mul t0,t3,t4` and `mul t1,t5,t6` in the RISC-V architecture.

0x03de02b3 → 0000001 11101 11100 000 00101 0110011

funct7 = 0000001
rs2 = 11101 = x29 (t4)
rs1 = 11100 = x28 (t3)
funct3 = 000
rd = 00101 = x5 (t0)
op = 0110011

0x03ff0333 → 0000001 11111 11110 000 00110 0110011

funct7 = 0000001
rs2 = 11111 = x31 (t6)
rs1 = 11110 = x30 (t5)
funct3 = 000
rd = 00110 = x6 (t1)
op = 0110011

From Appendix B of DDCARV:

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	**R-Type**

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000001	R	`mul rd, rs1, rs2`	multiply	`rd = (rs1 X rs2)`$_{31:0}$

Name	Register Number	Use
`zero`	`x0`	Constant value 0
`ra`	`x1`	Return address
`sp`	`x2`	Stack pointer
`gp`	`x3`	Global pointer
`tp`	`x4`	Thread pointer
`t0-2`	`x5-7`	Temporary variables
`s0/fp`	`x8`	Saved variable / Frame pointer
`s1`	`x9`	Saved variable
`a0-1`	`x10-11`	Function arguments / Return values
`a2-7`	`x12-17`	Function arguments
`s2-11`	`x18-27`	Saved variables
`t3-6`	`x28-31`	Temporary variables

TASK: Replicate the simulation from Figure 2 on your own computer and analyse it more closely.

Solution provided in the main document of Lab 14.

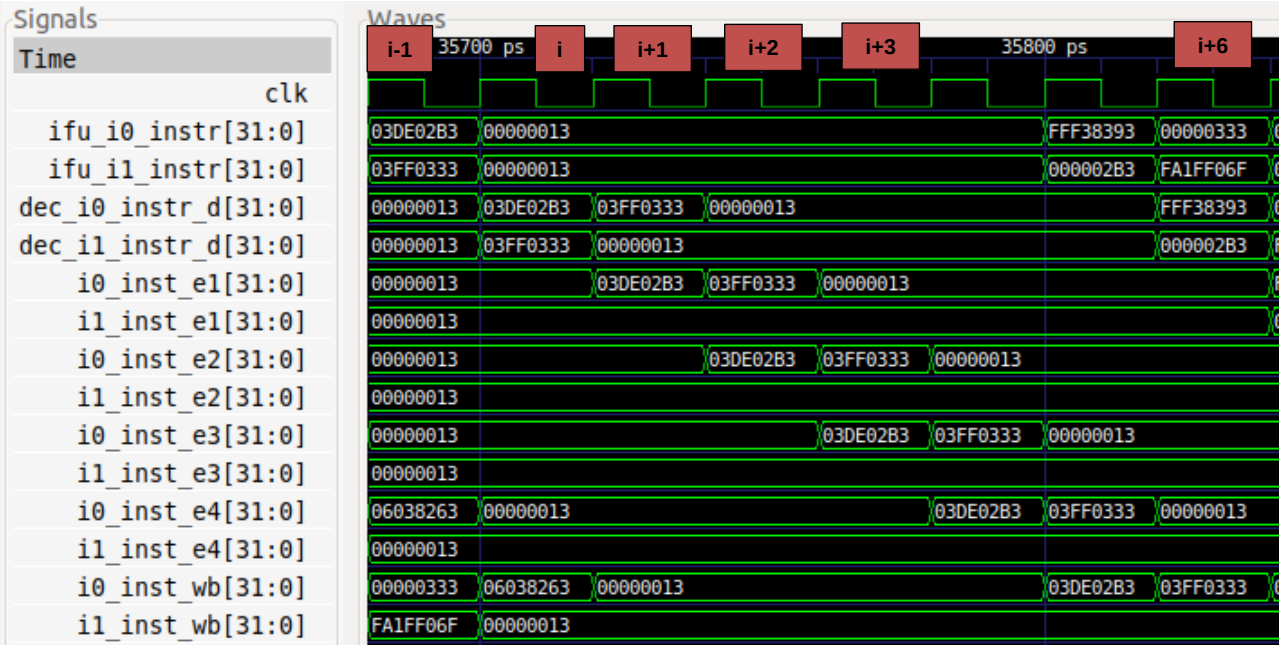
TASK: Compare the illustration from Figure 3 with the simulation from Figure 2 focusing on the two `mul` instructions. Specifically, analyse how the two instructions are assigned to the two ways in the Align and Decode stages.

- In module **ifu_aln_ctl** (Align stage) the two instructions are assigned to:
  - Way 0: `ifu_i0_instr`

- Way 1: `ifu_i1_instr`

- In module **dec_ib_ctl** the two instructions are buffered from Align to Decode:
  - Way 0: `ifu_i0_instr` → `dec_i0_instr_d`
  - Way 1: `ifu_i1_instr` → `dec_i1_instr_d`
- In module **dec_decode_ctl** (Decode stage) the two instructions are scheduled to the corresponding pipes if possible. Once they are sent, they continue through the three execution stages, the Commit stage and the Writeback stage:
  - Way 0: `i0_inst_e1` – `i0_inst_e2` – `i0_inst_e3` – `i0_inst_e4` – `i0_inst_wb`
  - Way 1: `i1_inst_e1` – `i1_inst_e2` – `i1_inst_e3` – `i1_inst_e4` – `i1_inst_wb`

We provide a *.tcl* file called *[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instruction/test_AssignmentWays.tcl* that includes all these signals.



- In cycle i-1 (not shown in Figure 2 nor in Figure 3) the two `mul` instructions are at the Align stage: the first is assigned to Way 0 (`ifu_i0_instr` = 0x03de02b3) and the second is assigned to Way 1 (`ifu_i1_instr` = 0x03ff0333) at module **ifu_aln_ctl**.

- In cycle i the two instructions have been propagated to the Decode stage at module **dec_ib_ctl**: the first continues in Way 0 (`dec_i0_instr_d` = 0x03de02b3) and the second continues in Way 1 (`dec_i1_instr_d` = 0x03ff0333).

- In cycle i+1 the first `mul` instruction has been propagated to the M1 stage at the **dec_decode_ctl** module (`i0_inst_e1` = 0x03de02b3). However, the second `mul` instruction couldn't be propagated due to the structural hazard analysed in the lab, and thus a bubble has been inserted in the first execution stage of Way 1: `i1_inst_e1` = 0x00000013.
  Moreover, given that Way 0 has been released at the Decode stage, the second `mul` has been reassigned to that Way: `dec_i0_instr_d` = 0x03ff0333.

- In cycle i+2 the second `mul` instruction is propagated to the M1 stage, which is now free (`i0_inst_e1` = 0x03ff0333), and the first `mul` instruction is propagated to the M2 stage.

- In cycles i+3 to i+6 the two `mul` instructions progress through the pipeline with no stalls until the Writeback stage.

TASK: Remove the `nop` instructions included within the loop and measure different events (cycles, instructions/multiplies committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11. Is the number of cycles as expected after analysing the simulation from Figure 2? Justify your answer.
Now reorder the code within the loop trying to reach the ideal throughput. Justify the results obtained in the original code and in the reordered one.

```c
    instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
    MulCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);

    Test_Assembly();

    cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
    instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
    MulCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);

    printfNexys("Cycles = %d", cyc_end-cyc_beg);
    printfNexys("Instructions = %d", instr_end-instr_beg);
    printfNexys("MulCom = %d", MulCom_end-MulCom_beg);

    while(1);
}
```

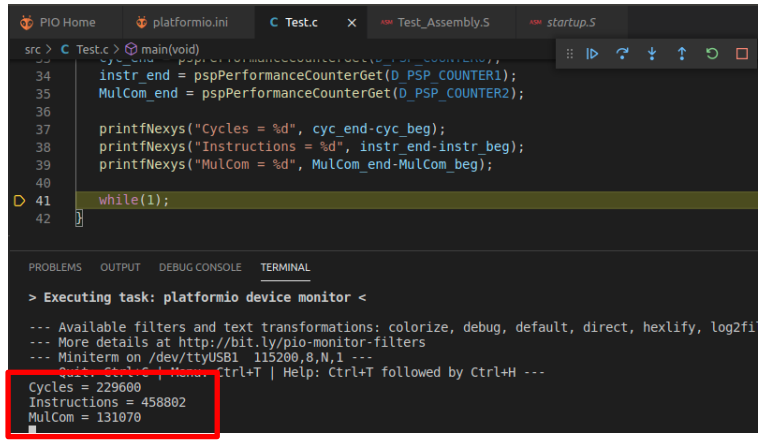
```
> Executing task: platformio device monitor <

--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2fi
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1  115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Cycles = 262367
Instructions = 458802
MulCom = 131070
```

IPC = 458000 / 262000 = 1.748. The IPC is a bit smaller than the ideal one because the second mul instruction must wait one cycle due to the structural hazard, as explained in the lab.

If we reorder the code, inserting in between the two mul instructions the update of the loop index, we obtain the ideal IPC, as we fill the bubble introduced by the structural hazard with a useful instruction.

```
REPEAT:
    beq t2, zero, OUT      # Stay in the loop?
    mul t0, t3, t4         # t0 = t3 * t4
    add t2, t2, -1
    mul t1, t5, t6         # t1 = t5 * t6
    add t0, zero, zero
    add t1, zero, zero
    j REPEAT
OUT:
```



IPC = 458000 / 229000 = 2

TASK: Folder *[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instr_Accumul_C-Lang* provides the PlatformIO project of a C program that accumulates the subtraction of two multiplications within a loop.

- Analyse the C program.

- Perform a simulation and inspect a random iteration of the loop. Note that the C program is compiled without optimizations.

- Measure different events (cycles, instructions/multiplications committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11.
Is the number of cycles as expected after analysing the simulation from Figure 2? Justify your answer.

- Create an analogous program in RISC-V assembly and compare it with the C version. Reorder the instructions trying to obtain the best possible IPC.

- Disable the **M** RISC-V extension in the C program and compare the results with the original program. To do so, modify the following line in file *platformio.ini* from:
  `build_flags = -Wa,-march=rv32ima -march=rv32ima`
To:

```
build_flags = -Wa,-march=rv32ia -march=rv32ia
```

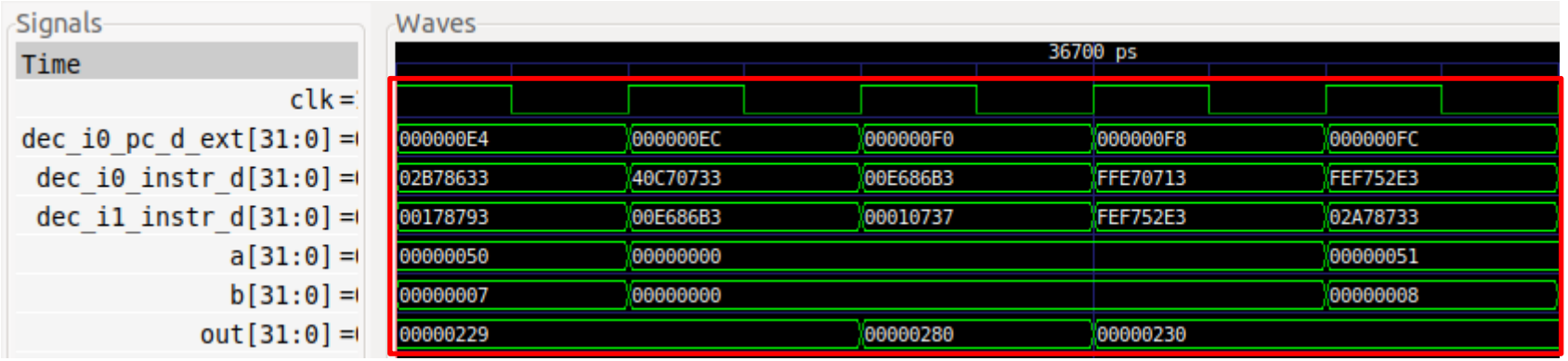
This avoids the use of the instructions from the M RISC-V extension and emulates them using other instructions instead.

- C program (original and disassembly):

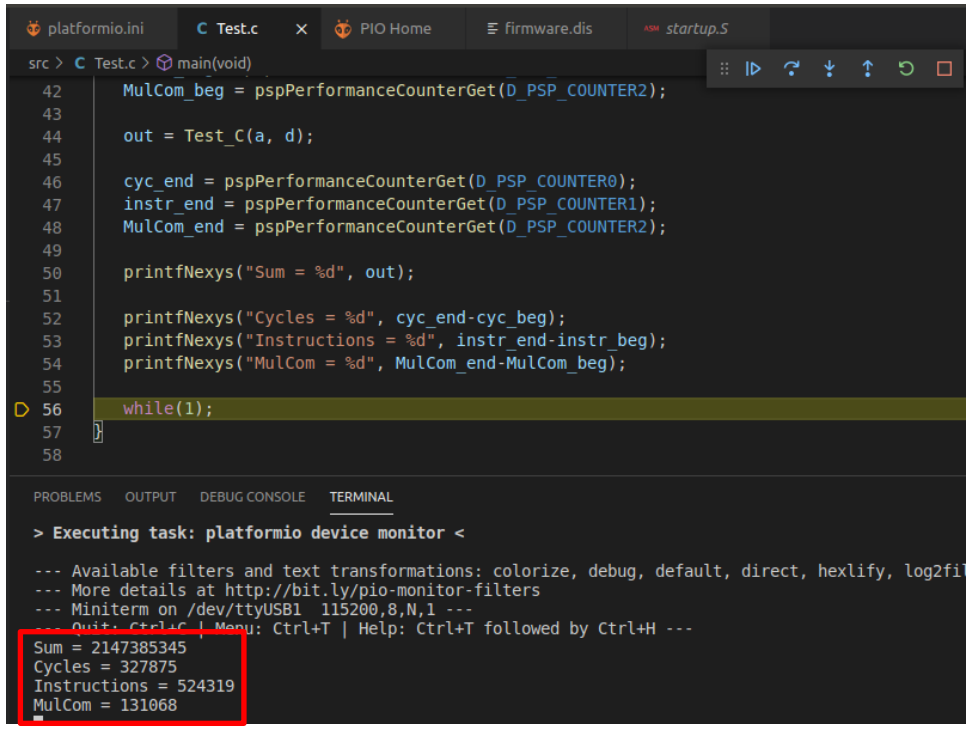
```c
int Test_C(int a, int d)
{
    int b, c, e=0, i=1;
    do {
        b = a*i;
        c = d*i;
        i = i+1;
        e = e + (b-c);
    } while(i<65535);
    return(e);
}
```

```
000000d8 <Test_C>:
  d8: 00100793          li  a5,1
  dc: 00000693          li  a3,0
  e0: 02a78733          mul a4,a5,a0
  e4: 02b78633          mul a2,a5,a1
  e8: 00178793          addi  a5,a5,1
  ec: 40c70733          sub a4,a4,a2
  f0: 00e686b3          add a3,a3,a4
  f4: 00010737          lui a4,0x10
  f8: ffe70713          addi  a4,a4,-2 # fffe <_sp+0xc386>
  fc: fef752e3          bge a4,a5,e0 <Test_C+0x8>
 100: 00068513          mv  a0,a3
 104: 00008067          ret
```

- Simulation of the C program:



- HW Counters:



IPC = 524000 / 327000 = 1.6. Some cycles are lost due to RAW data hazards, that we will analyse in Lab 15.

- The Assembly program can be found at:
  [RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab14/MUL_Instr_Accumul_Assembly

```
000001c4 <Test_Assembly>:
 1c4: 00100393            li  t2,1
 1c8: 00000e93            li  t4,0
 1cc: 00000f93            li  t6,0
 1d0: 00010637            lui a2,0x10
 1d4: fff60613            addi  a2,a2,-1 # ffff <_sp+0xc567>
 1d8: 00050e33            add t3,a0,zero
 1dc: 00058f33            add t5,a1,zero

000001e0 <REPEAT>:
 1e0: 027e02b3            mul t0,t3,t2
 1e4: 027f0333            mul t1,t5,t2
 1e8: 00138393            addi  t2,t2,1
 1ec: 40628eb3            sub t4,t0,t1
 1f0: 01df8fb3            add t6,t6,t4
 1f4: fec396e3            bne t2,a2,1e0 <REPEAT>
 1f8: 000f8533            add a0,t6,zero
 1fc: 00008067            ret
```

PIO Home | platformio.ini | Test.c | Test_Assembly.S | firmware.dis | startup.S

src > Test.c > main(void)

```c
    out = Test_Assembly(a, d);

    cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
    instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
    MulCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);

    printfNexys("Sum = %d", out);

    printfNexys("Cycles = %d", cyc_end-cyc_beg);
    printfNexys("Instructions = %d", instr_end-instr_beg);
    printfNexys("MulCom = %d", MulCom_end-MulCom_beg);

    while(1);
}
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
> Executing task: platformio device monitor <

--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2file,
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1  115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Sum = 2147385345
Cycles = 262367
Instructions = 393256
MulCom = 131068
```

The result of the Sum is the same, as the program is the same.
The number of cycles is a bit smaller, as the assembly version programmed by hand is more efficient than the one obtained by the compiler without optimizations.
The number of instructions is also a bit smaller.

We reorder the loop as follows:

```
li  t2, 0x1
li  t4, 0x0
li  t6, 0x0
li  a2, 0xFFFF
add t3, a0, zero
add t5, a1, zero

REPEAT:
   mul t0, t3, t2        # t0 = t3 * t2
   mul t1, t5, t2        # t1 = t5 * t2
   sub t4, t0, t1
   add t2, t2, 1
   add t6, t6, t4
   bne t2, a2, REPEAT   # Repeat the loop

add a0, t6, zero
```

PIO Home | platformio.ini | Test.c | Test_Assembly.S | startup.S

src > Test.c > main(void)

```c
    out = Test_Assembly(a, d);

    cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
    instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
    MulCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);

    printfNexys("Sum = %d", out);

    printfNexys("Cycles = %d", cyc_end-cyc_beg);
    printfNexys("Instructions = %d", instr_end-instr_beg);
    printfNexys("MulCom = %d", MulCom_end-MulCom_beg);

    while(1);
}
```

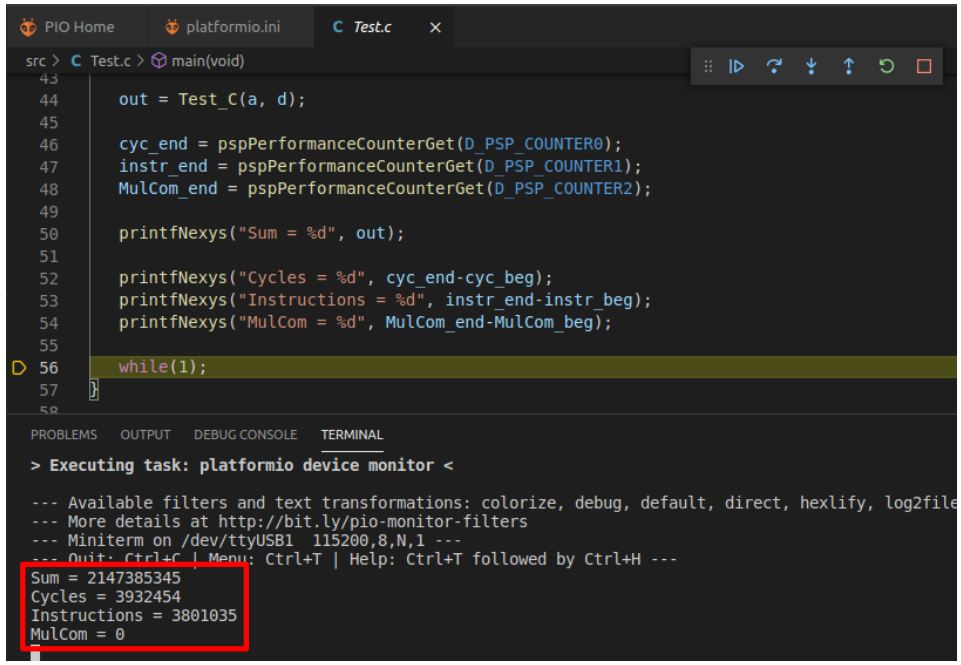
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
> Executing task: platformio device monitor <

--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2fil
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1  115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Sum = 2147385345
Cycles = 196843
Instructions = 393256
MulCom = 131068
```

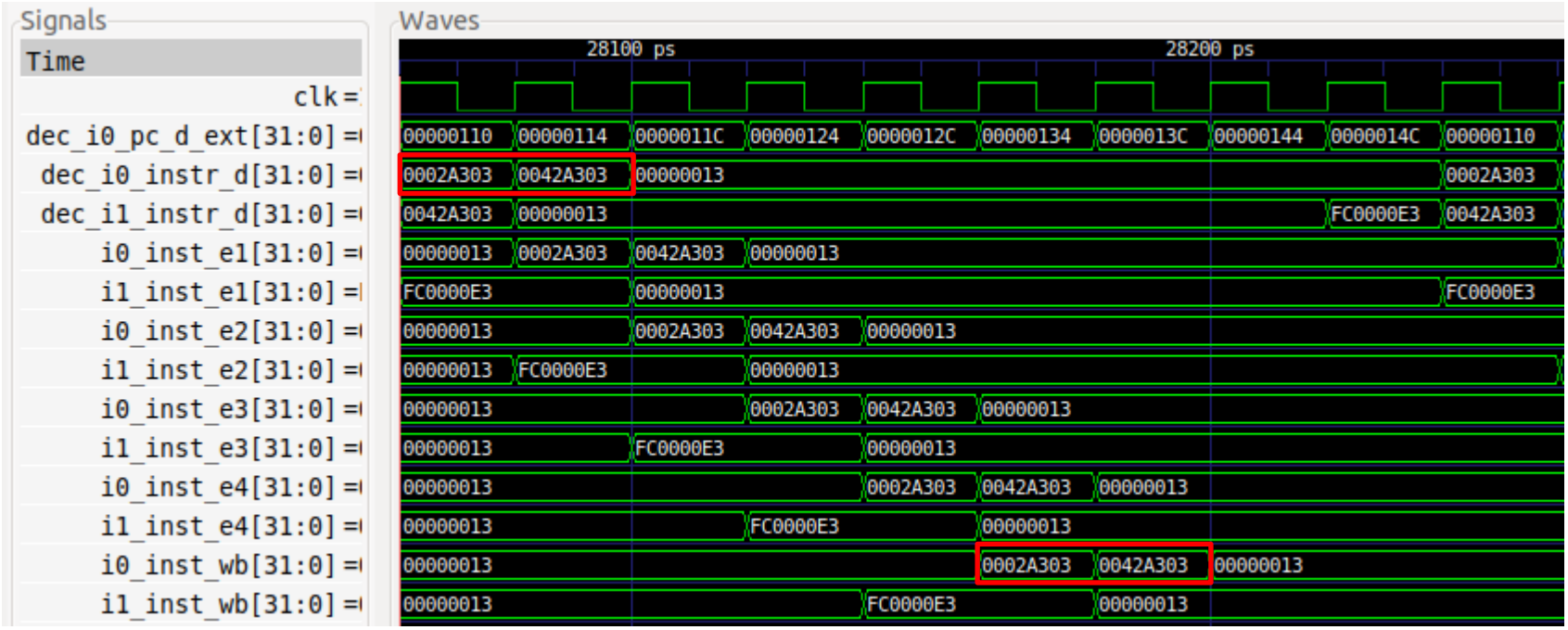
The result of the Sum is the same, as the program is the same.
Number of cycles per iteration = 196800 / 65500 = 3
The number of instructions is the same. Number of instructions per iterations = 393000 / 65500 = 6
IPC = 393 / 197 = 1.994. We obtain the optimal IPC.

- Disable M Extension:



The result of the Sum is the same, as the program is the same.
The number of cycles is much higher: Around 4M vs. around 0.3M.
The number of instructions is also much higher: Around 3M vs. around 0.5M.
The CPI is better now.
There are no multiplications committed.

TASK: Modify the program from Figure 1, replacing the two `mul` instructions for two `lw` instructions to the DCCM. You should observe a structural hazard analogous to the one analysed in this section and resolved in a similar way.



As we can see in the simulation, the behaviour for two consecutive loads is exactly the same as in the case of two consecutive mul instructions.

TASK: Replicate the simulation from Figure 6 on your own computer. Use file *test_NonBlocking.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory*). *Zoom In* ( ) several times and move to 60120ps.

Solution provided in the main document of Lab 14.

TASK: Compare the simulation shown in Figure 6 (non-blocking load) with the simulation shown in Figure 14 of Lab 13 (blocking load). Add all of the signals needed for the comparison.

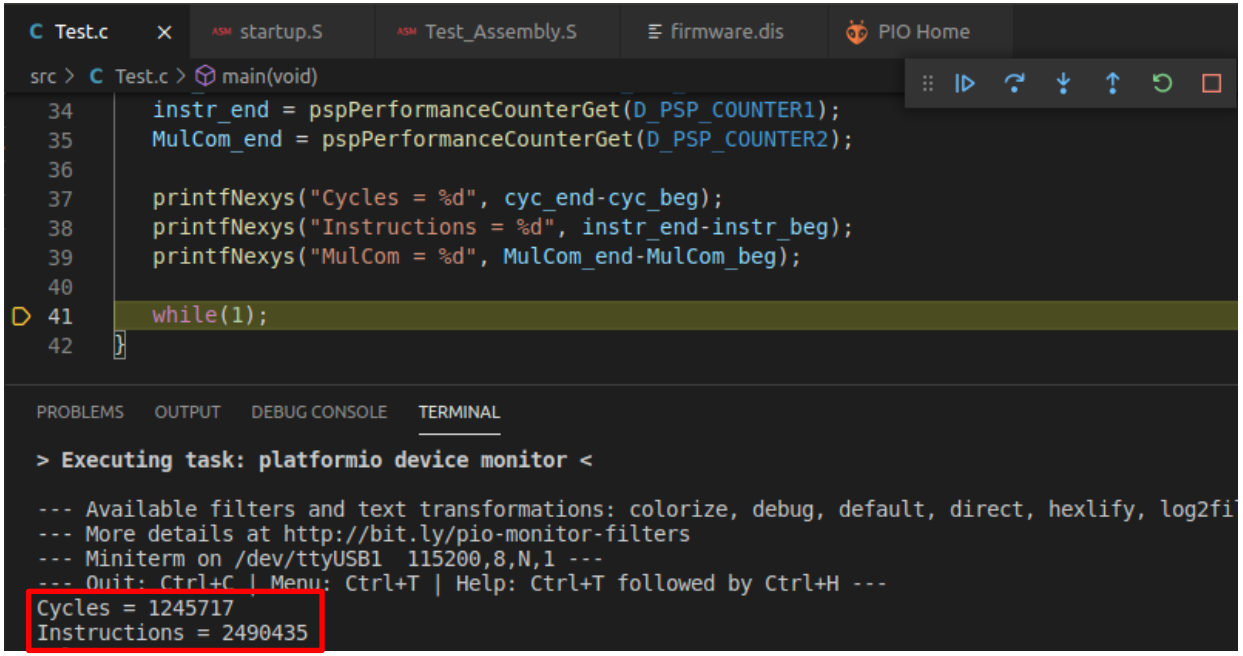
Solution not provided.

TASK: Compare the illustration from Figure 7 with the simulation from Figure 6 that you have replicated on your own computer. Add signals to extend the simulation and deepen understanding, as desired.

Solution not provided.

TASK: Measure different events (cycles, instructions/loads committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11. Is the number of cycles as expected after analysing the simulation from Figure 6? Justify your answer. Compare these results with those obtained when loads are configured as blocking loads.

Non-blocking loads:



The IPC obtained (IPC = 2490 / 1245 = 2) is the ideal thanks to the non-blocking load.

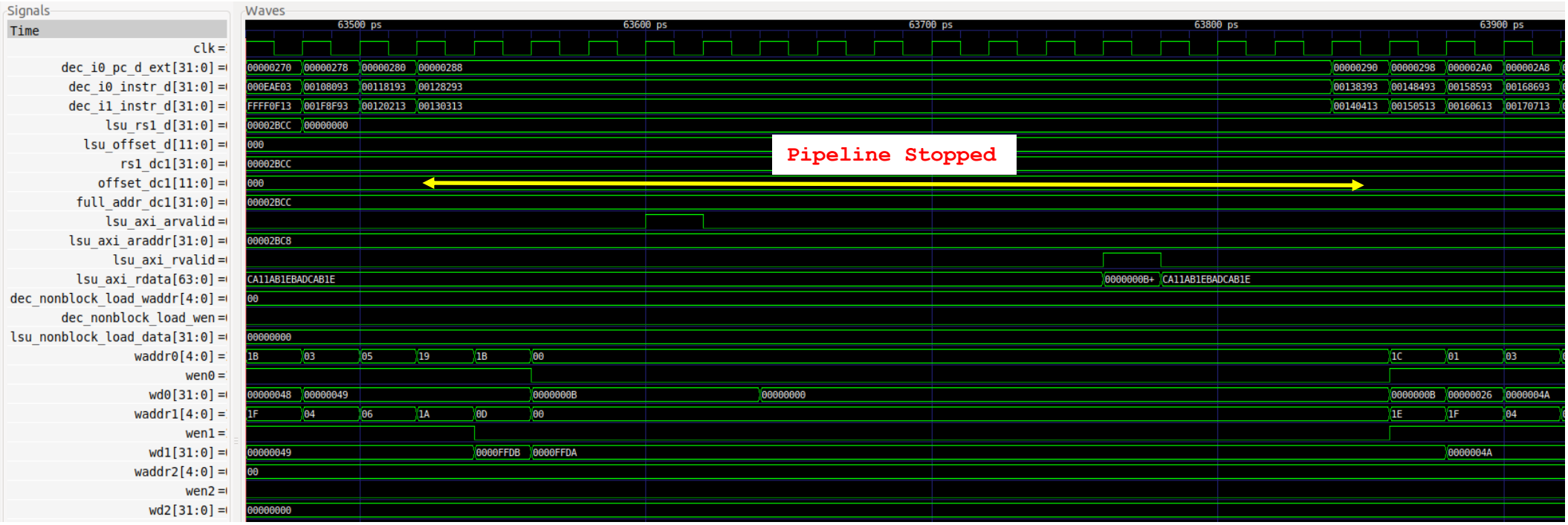
Blocking loads:

```
PIO Home   Test.c   Test_Assembly.S   startup.S
src > Test.c > main(void)
28    instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
29    MulCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
30
31    Test_Assembly();
32
33    cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
34    instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
35    MulCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
36
37    printfNexys("Cycles = %d", cyc_end-cyc_beg);
38    printfNexys("Instructions = %d", instr_end-instr_beg);
39    printfNexys("MulCom = %d", MulCom_end-MulCom_beg);
40
41    while(1);
42  }

PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL
> Executing task: platformio device monitor <

--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2fi
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1  115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Cycles = 2698068
Instructions = 2490437
```

The number of instructions is the same, but now it takes much more cycles to execute the loop as the loads make the subsequent instructions to stall for the data to come from memory. The simulation demonstrates it more clearly.



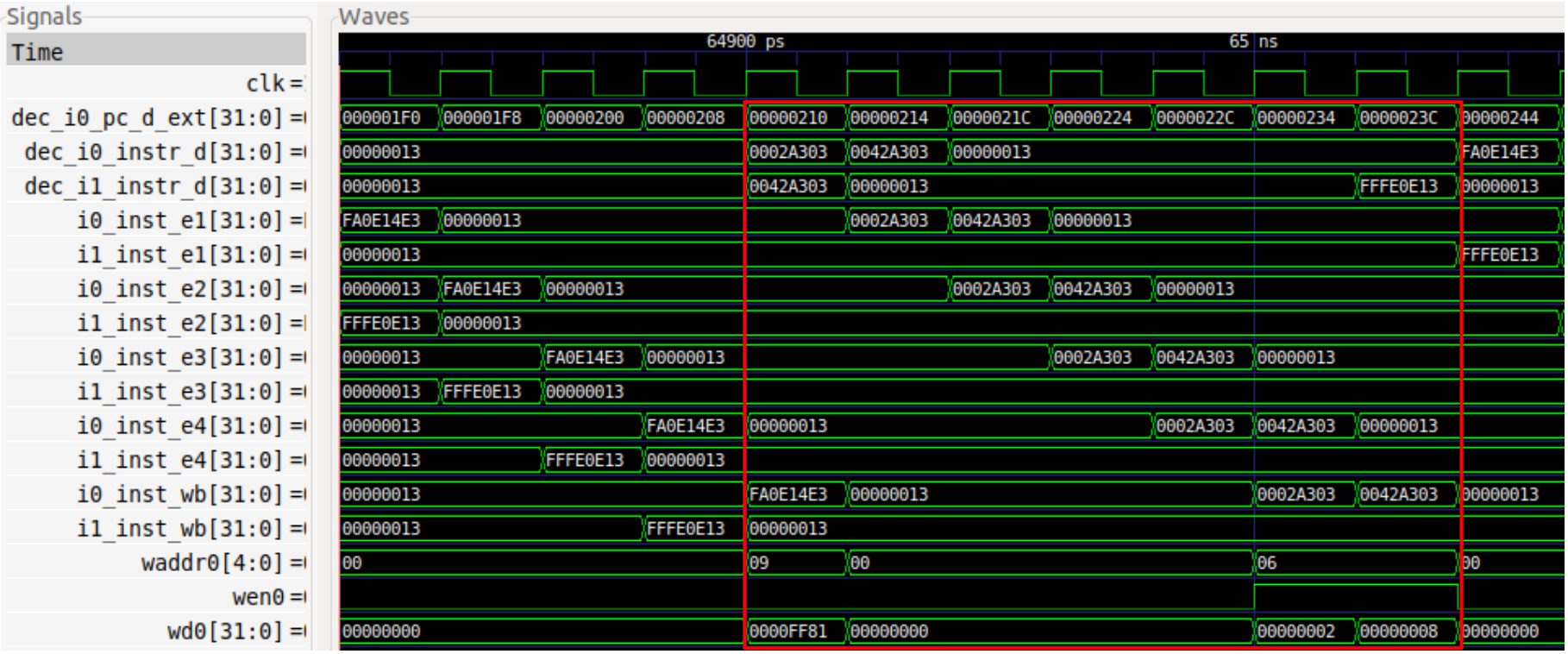
EXERCISES

1. Analyse, both in simulation and on the board, the structural hazard that happens between two consecutive memory instructions (you can analyse any combination of two consecutive memory instructions such as loads and stores) that arrive at the L/S Pipe in the same cycle. Test both for non-blocking and for blocking loads. You can use the PlatformIO project provided at: *[RVfpgaPath]/RVfpga/Labs/Lab14/TwoConsecutiveLW_Instructions*.

Two consecutive loads:

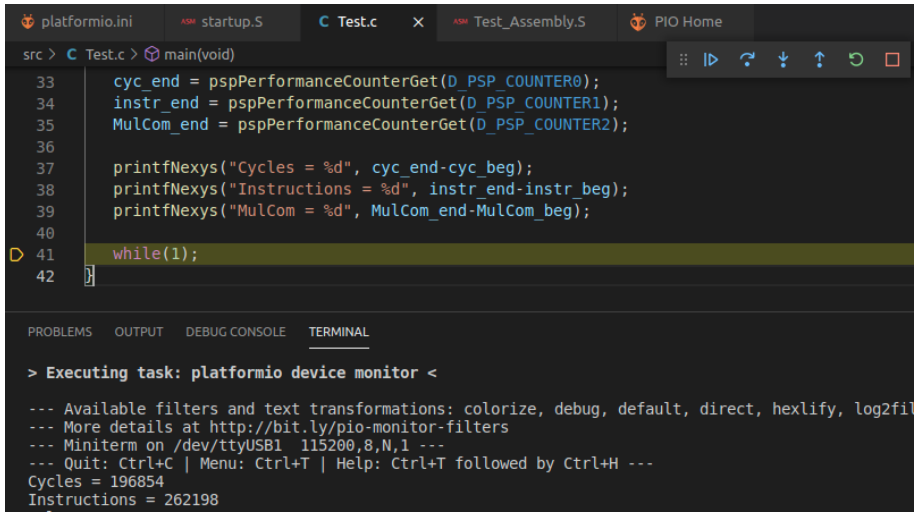
```
210:  0002a303          lw    t1,0(t0)
214:  0042a303          lw    t1,4(t0)
```

- Simulation:



Due to the structural hazard in the L/S Pipe, the second lw must stall for 1 cycle, similarly to the Mult Pipe handling two consecutive mul instructions.

- Execution on the board:

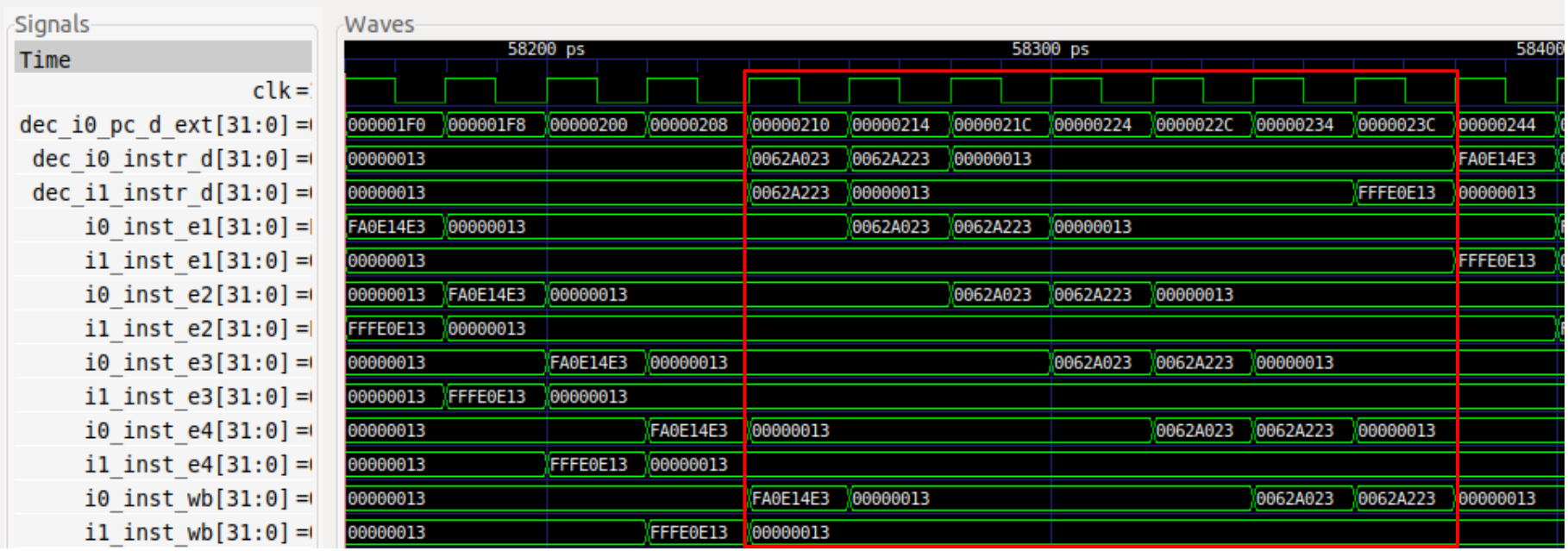


IPC = 262 / 196 = 1.33

Two consecutive stores:

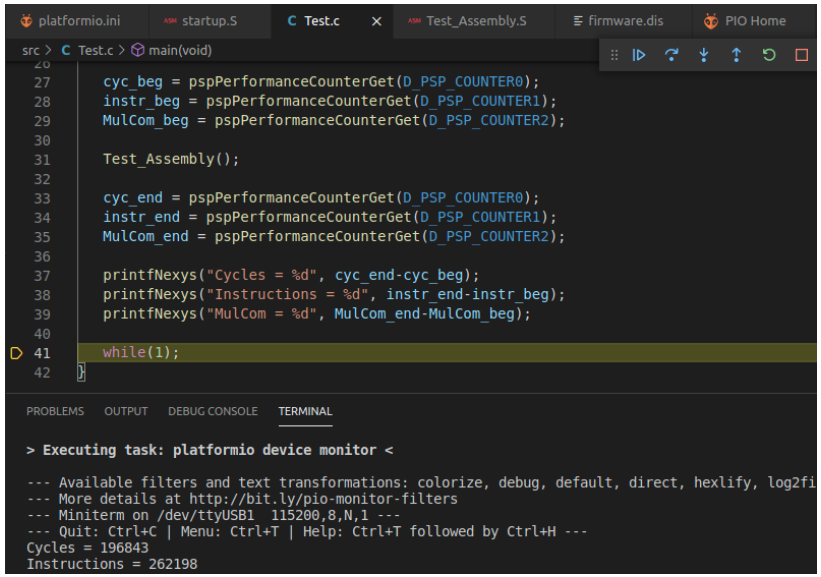
```
210:  0062a023          sw    t1,0(t0)
214:  0062a223          sw    t1,4(t0)
```

- Simulation:



Due to the structural hazard in the L/S Pipe, the second sw must stall for 1 cycle, similarly to the Mult Pipe handling two consecutive mul instructions.

- Execution on the board:



IPC = 262 / 196 = 1.33

2. (*The following exercise is based on exercise 4.22 from the book "Computer Organization and Design – RISC-V Edition", by Patterson & Hennessy ([HePa]).*)

   Consider the fragment of RISC-V assembly below:

```
sd x29, 12(x16)
ld x29, 8(x16)
sub x17, x15, x14
beqz x17, label
add x15, x11, x14
sub x15, x30, x14
```

   Suppose we modify the SweRV EH1 processor so that it has only one memory (that handles both instructions and data). In this case, there will be a structural hazard every time a program needs to fetch an instruction during the same cycle in which another instruction accesses data.

   a. Draw a pipeline diagram to show where the code above will stall in this imaginary version of the SweRV EH1 processor.

b. In general, is it possible to reduce the number of stalls/nops resulting from this structural hazard by reordering code?
c. Must this structural hazard be handled in hardware? We have seen that data hazards can be eliminated by adding nops to the code. Can you do the same with this structural hazard? If so, explain how. If not, explain why not.

Solution not provided.

APPENDIX A – TWO SIMULTANEOUS `DIV` INSTRUCTIONS IN THE DECODE STAGE

TASK: You can perform a similar study for the `div` instruction as the one performed in Lab 12 for arithmetic-logic instructions: view the flow of the instruction through the pipeline stages, analyse the control bits (remember from Appendix D of Lab 11 that there is a specific structure type for the `div` instruction called `div_pkt_t`, and there is a signal defined in module **dec_decode_ctl** called `div_p`), etc.

Solution not provided.

TASK: Inspect the Verilog code from **exu_div_ctl** to understand how the division is computed. Also analyse the effect of signals `div_stall`, `finish_early`, and `finish`. As an optional exercise, replace the Divide Unit with your own unit or one from the Internet.

Solution not provided.

TASK: Verify that this pair of 32 bits (0x03de42b3 and 0x03ff4333) correspond to instructions `div t0,t3,t4` and `div t1,t5,t6` in the RISC-V architecture.

0x03de42b3 → 0000001 11101 11100 100 00101 0110011

funct7 = 0000001
rs2 = 11101 = x29 (t4)
rs1 = 11100 = x28 (t3)
funct3 = 100
rd = 00101 = x5 (t0)
op = 0110011

0x03ff4333 → 0000001 11111 11110 100 00110 0110011

funct7 = 0000001
rs2 = 11111 = x31 (t6)
rs1 = 11110 = x30 (t5)
funct3 = 100
rd = 00110 = x6 (t1)
op = 0110011

From Appendix B of DDCARV:

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	**R-Type**

op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	100	0000001	R	`div rd, rs1, rs2`	divide (signed)	`rd = rs1 / rs2`

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved variables
t3-6	x28-31	Temporary variables

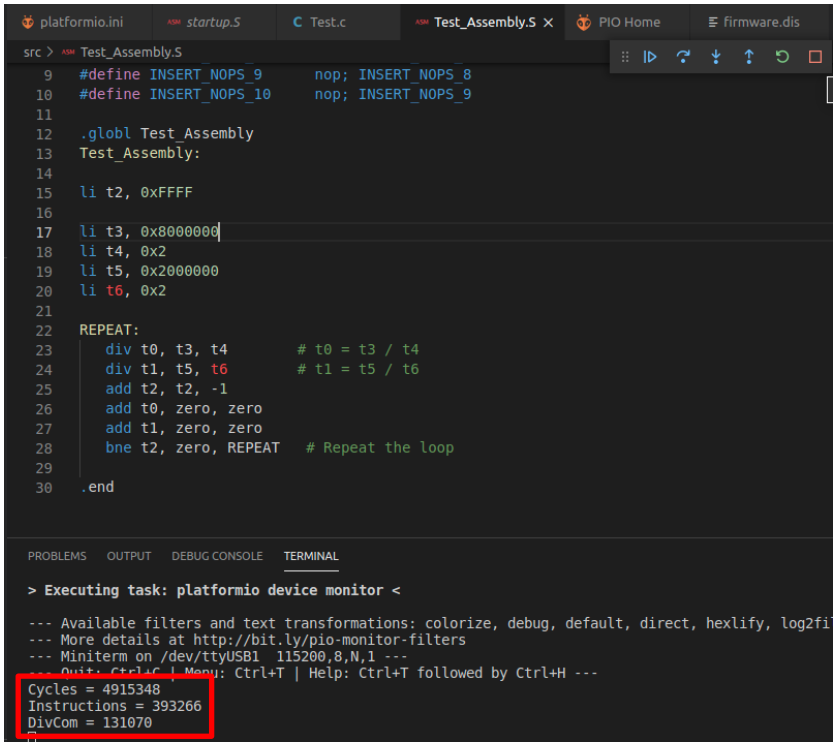
TASK: Replicate the simulation from Figure 9 on your own computer and analyse it in detail.

Solution provided in the main document of Lab 14.

TASK: Compare the illustration from Figure 10 and the simulation from Figure 9 that you have replicated on your own computer. Add signals to extend the simulation and deepen understanding, as desired.

Solution not provided.

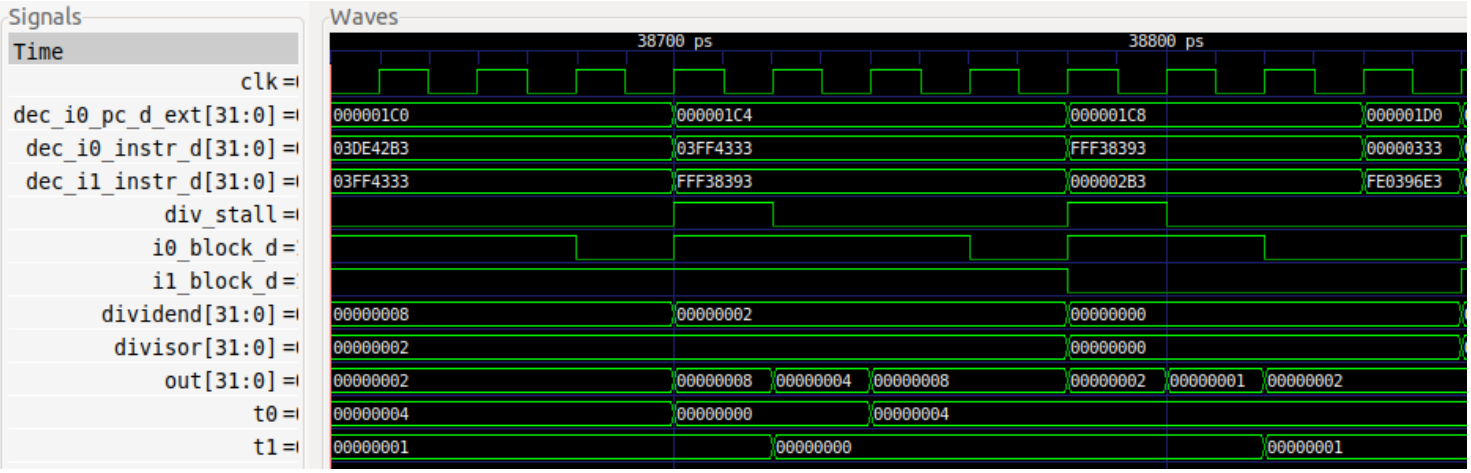
TASK: Measure different events (cycles, instructions/divisions committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11. Is the number of cycles as expected after analysing the simulation from Figure 9? Justify your answer.



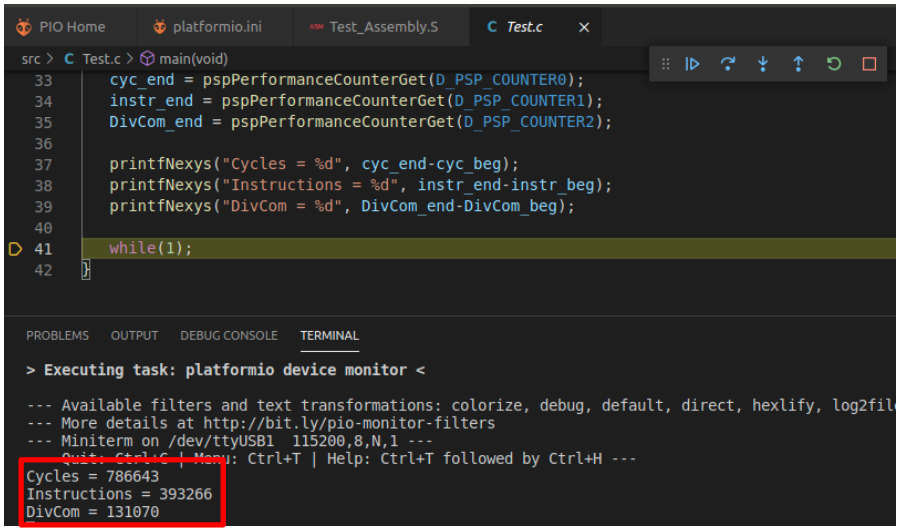
CPI = 4910000 / 393000 = 12. Taking into account that each division takes around 34 cycles to execute and that the other instructions take ½ cycle each, this is approximately what we could expect: an approximate theoretical computation could be: 6 instructions executed in 34 + 34 + ½ + ½ + ½ + ½ cycles → CPI = 70 / 6 = 11

TASK: Try different dividends and divisors and see how the number of cycles for computing the result depends on their value. View the experiment both in simulation and with the HW Counters.

```
12  .globl Test_Assembly
13  Test_Assembly:
14
15  li t2, 0xFFFF
16
17  li t3, 0x8
18  li t4, 0x2
19  li t5, 0x2
20  li t6, 0x2
21
22  REPEAT:
23     div t0, t3, t4        # t0 = t3 / t4
24     div t1, t5, t6        # t1 = t5 / t6
25     add t2, t2, -1
26     add t0, zero, zero
27     add t1, zero, zero
28     bne t2, zero, REPEAT   # Repeat the loop
29
30  .end
```



Now the divisions are computed in only around 5 cycles.



The CPI decreases a lot (around 2 per cycle) given that the time for computing each division decreases a lot too.

TASK: Folder *[RVfpgaPath]/RVfpga/Labs/Lab14/DIV_Instr_Accumul_C-Lang* provides the PlatformIO project of a C program that accumulates the subtraction of two divisions within a loop.

- Analyse the C program.

- Perform a simulation and inspect a random iteration of the loop. Note that the C program is compiled without optimizations.

- Measure different events (cycles, instructions/divisions committed, etc.) using the Performance Counters available in SweRV EH1, as explained in Lab 11.
Is the number of cycles as expected after analysing the simulation from Figure 9? Justify your answer.

- Create an analogous program in RISC-V assembly and compare it with the C version.

- Disable the **M** RISC-V extension in the C program and compare the results with the original program. To do so, modify the following line in file *platformio.ini* from:

```
build_flags = -Wa,-march=rv32ima -march=rv32ima
```

To:

```
build_flags = -Wa,-march=rv32ia -march=rv32ia
```

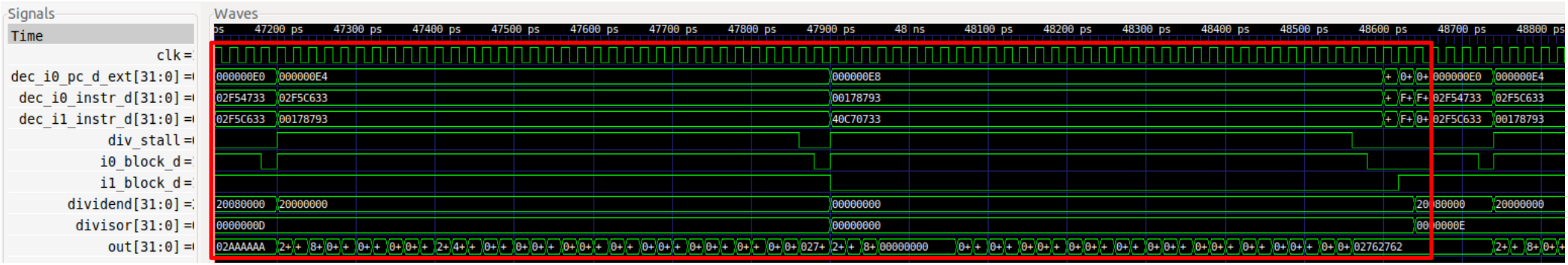
This avoids the use of the instructions from the RISC-V M extension and emulates them using other instructions instead.

- C program (original and disassembly):

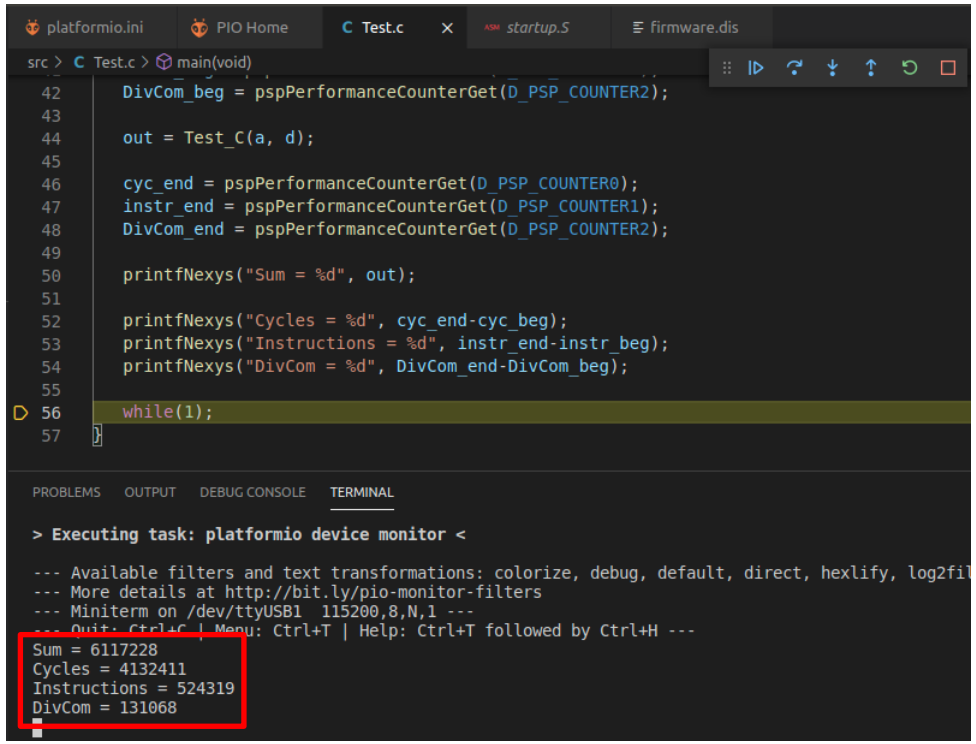
```c
int Test_C(int a, int d)
{
    int b, c, e=0, i=1;
    do {
        b = a/i;
        c = d/i;
        i = i+1;
        e = e + (b-c);
    } while(i<65535);
    return(e);
}
```

```
000000d8 <Test_C>:
  d8: 00100793          li  a5,1
  dc: 00000693          li  a3,0
  e0: 02f54733          div a4,a0,a5
  e4: 02f5c633          div a2,a1,a5
  e8: 00178793          addi  a5,a5,1
  ec: 40c70733          sub a4,a4,a2
  f0: 00e686b3          add a3,a3,a4
  f4: 00010737          lui a4,0x10
  f8: ffe70713          addi  a4,a4,-2 # fffe <_sp+0xc386>
  fc: fef752e3          bge a4,a5,e0 <Test_C+0x8>
 100: 00068513          mv  a0,a3
 104: 00008067          ret
```

- Simulation of the C program:



- HW Counters:



- The Assembly program can be found at:
  [RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab14/DIV_Instr_Accumul_Assembly

```
000001c8 <Test_Assembly>:
 1c8: 00100393            li  t2,1
 1cc: 00000e93            li  t4,0
 1d0: 00000f93            li  t6,0
 1d4: 00010637            lui a2,0x10
 1d8: fff60613            addi  a2,a2,-1 # ffff <_sp+0xc377>
 1dc: 00050e33            add t3,a0,zero
 1e0: 00058f33            add t5,a1,zero

000001e4 <REPEAT>:
 1e4: 027e42b3            div t0,t3,t2
 1e8: 027f4333            div t1,t5,t2
 1ec: 00138393            addi  t2,t2,1
 1f0: 40628eb3            sub t4,t0,t1
 1f4: 01df8fb3            add t6,t6,t4
 1f8: fec396e3            bne t2,a2,1e4 <REPEAT>
 1fc: 000f8533            add a0,t6,zero
 200: 00008067            ret
```

platformio.ini | firmware.dis | PIO Home | Test.c

src > Test.c > main(void)

```
    DivCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);

    out = Test_Assembly(a, d);

    cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
    instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
    DivCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);

    printfNexys("Sum = %d", out);

    printfNexys("Cycles = %d", cyc_end-cyc_beg);
    printfNexys("Instructions = %d", instr_end-instr_beg);
    printfNexys("DivCom = %d", DivCom_end-DivCom_beg);

    while(1);
}
```

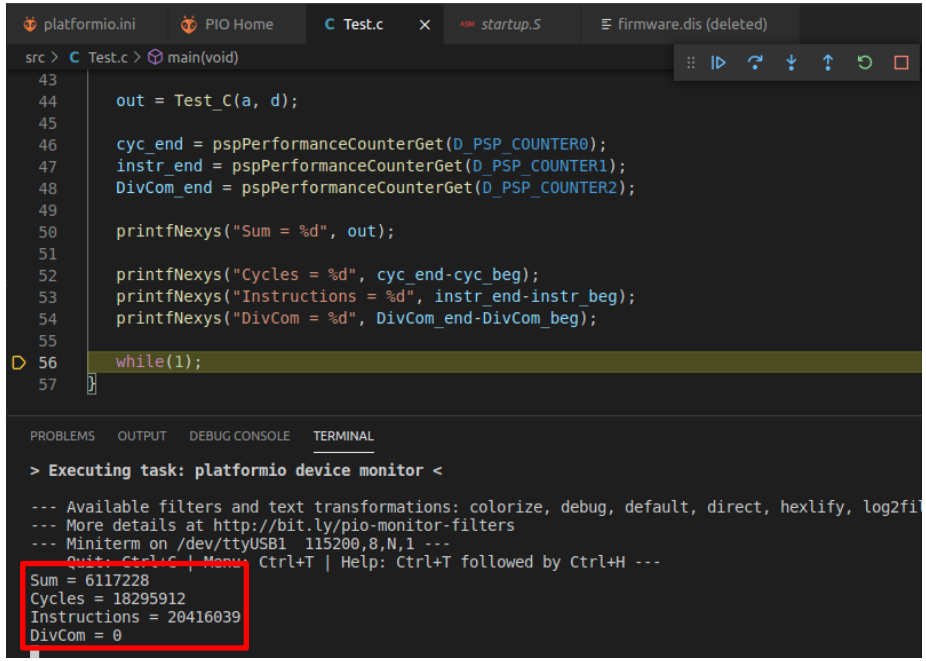
OUTPUT | DEBUG CONSOLE | TERMINAL

```
> Executing task: platformio device monitor <

--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2file
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1  115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---
Sum = 6117228
Cycles = 4001374
Instructions = 393256
DivCom = 131068
```

The result of the Sum is the same, as the program is the same.
The number of cycles is a bit smaller, as the assembly version programmed by hand is more efficient than the one obtained by the compiler without optimizations.
The number of instructions is also a bit smaller.

- Disable M Extension:



The result of the Sum is the same, as the program is the same.
The number of cycles is much higher: Around 18M vs. around 4M.
The number of instructions is also much higher: Around 20M vs. around 0.5M.
The CPI is better now.
There are no divisions commited.

TASK: In SweRV EH1, `div` instructions are blocking. Modify the processor to allow non-blocking `div` instructions.

Then add a second divider to the SweRV EH1 processor, so that two `div` instructions of the example from Figure 8 are allowed to execute in parallel.

Solution not provided.