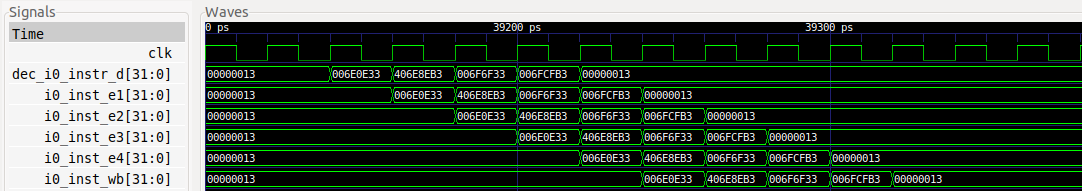
# TASKS

**TASK:** In the simulation from Figure 3, include the trace signals and highlight the instructions as they go through the pipeline from the Decode stage to the Writeback stage, similarly to Figure 4. You can use the *.tcl* file provided at: *[RVfpgaPath]/RVfpga/Labs/Lab17/Four\_AL\_Instructions/test\_task1.tcl*.

**add sub or xor**

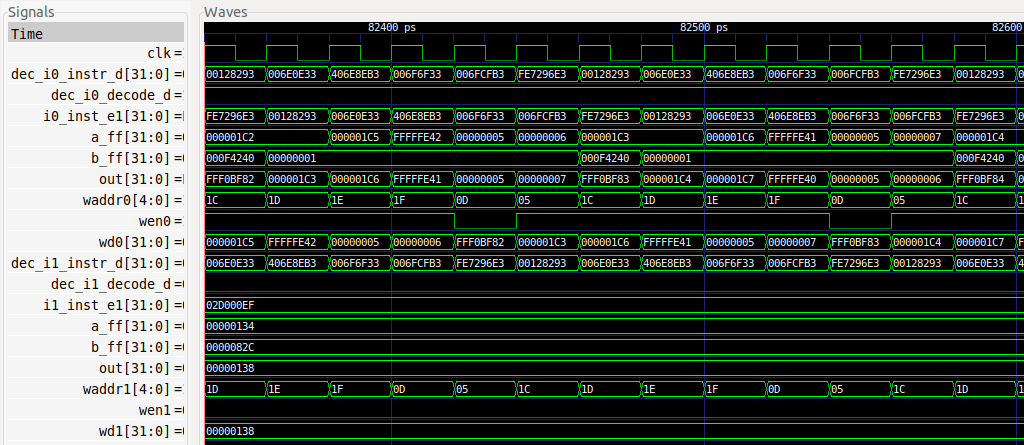


**1 2 3 4 5 6 7 8 9**

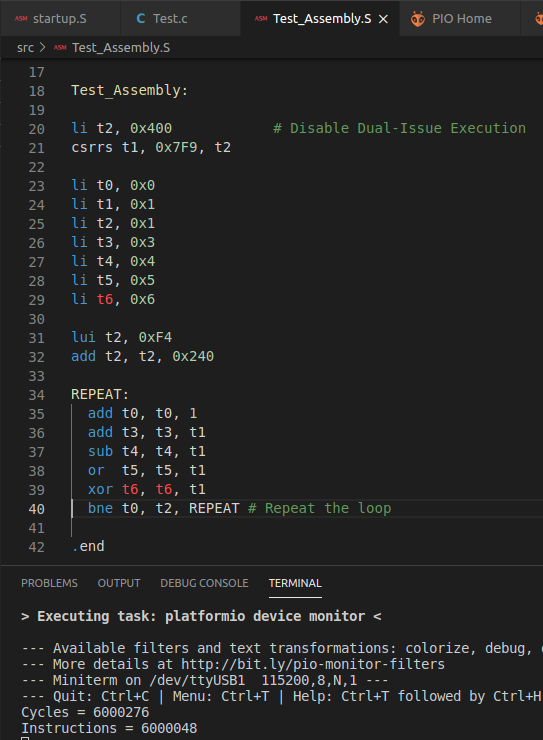
**TASK:** Removeall the nop instructions within the body of the loop from Figure 2.

Repeat the simulation from Figure 3. What is the expected IPC for this program?

Execute the program on the board and verify that the IPC obtained is the one that you expected.



The 6 instructions within the loop need 6 cycles to execute. Thus, the expected IPC is 1.

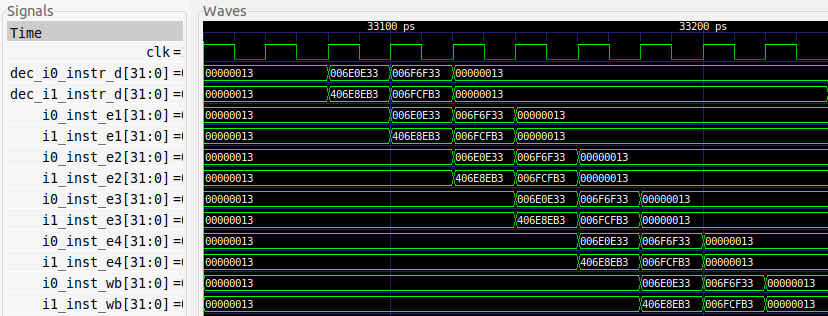


As expected:

IPC = 6000000 instructions / 6000000 cycles = 1

**TASK:** In the simulation from Figure 5, add trace signals and highlight the instructions as they go through the pipeline from the Decode to Writeback stages, similar to what’s shown in Figure 6. You can use the *.tcl* file provided at: *[RVfpgaPath]/RVfpga/Labs/Lab17/Four\_AL\_Instructions/test\_task2.tcl*.

**add sub or xor**

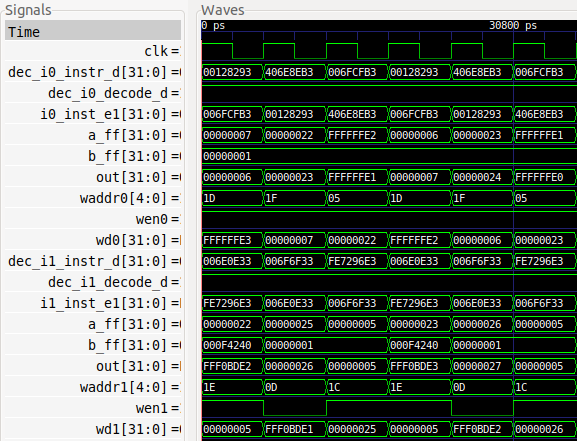


**1 2 3 4 5 6 7**

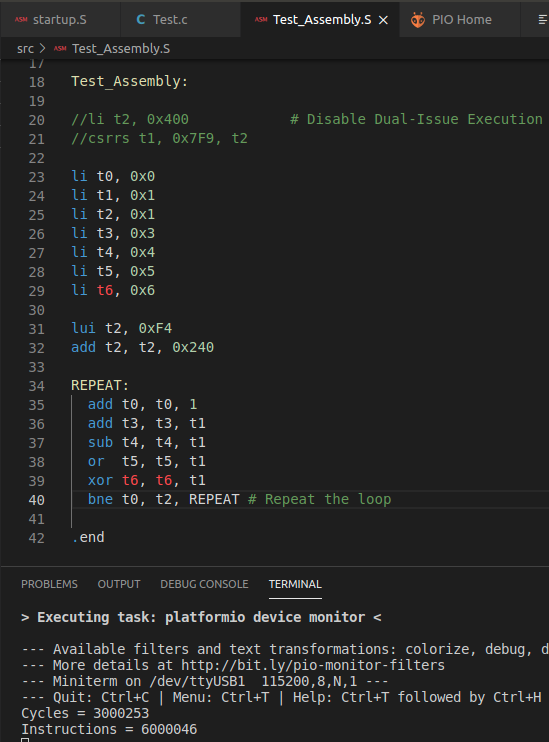
**TASK:** Removeall the nop instructions within the body of the loop from Figure 2.

Repeat the simulation from Figure 5. What is the expected IPC for this program?

Execute the program on the board and verify that the IPC obtained is the one that you expected.



The 6 instructions within the loop need 3 cycles to execute. Thus, the expected IPC is 2.

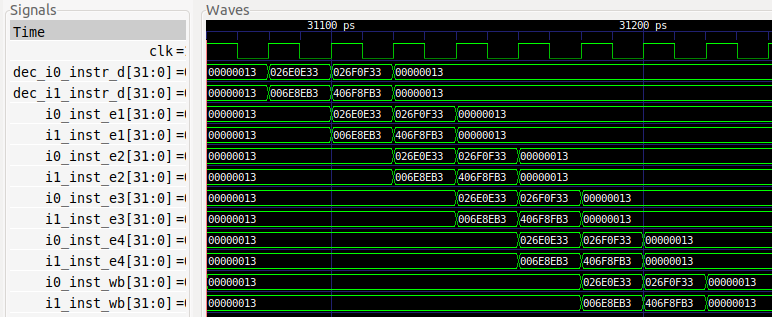


As expected:

IPC = 6000000 instructions / 3000000 cycles = 2

**TASK:** In the simulation from Figure 8, add trace signals and highlight the instructions as they go through the pipeline from the Decode to Writeback stages. You can use the *.tcl* file provided at: *[RVfpgaPath]/RVfpga/Labs/Lab17/TwoAL\_TwoMUL\_Instructions /test\_taskMuls.tcl*.

**mul add mul sub**



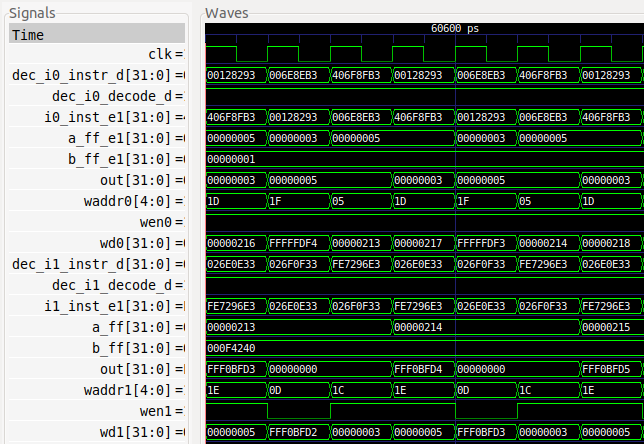
**TASK:** Removeall the nop instructions within the body of the loop from Figure 7.

Repeat the simulation from Figure 8. What is the expected IPC for this program?

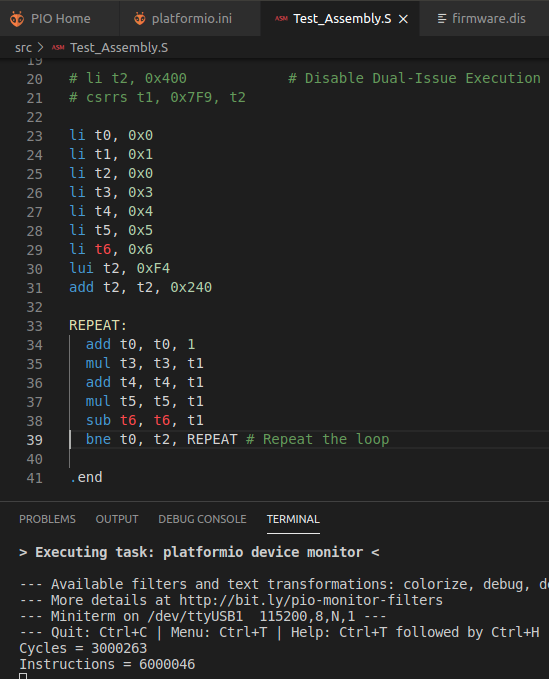
Execute the program on the board and verify that the IPC obtained is the one that you expected.

Repeat the same experiments for the single-issue configuration and compare the results.

**DUAL ISSUE:**



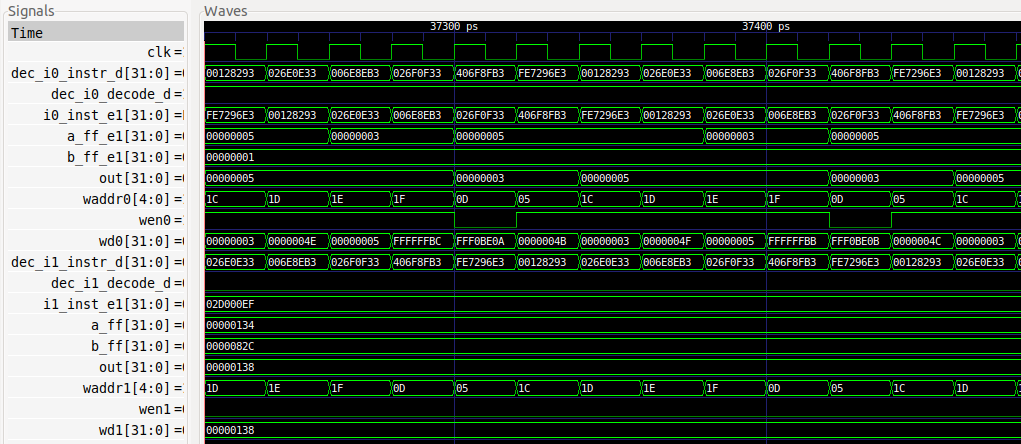
The 6 instructions within the loop need 3 cycles to execute. Thus, the expected IPC is 2.



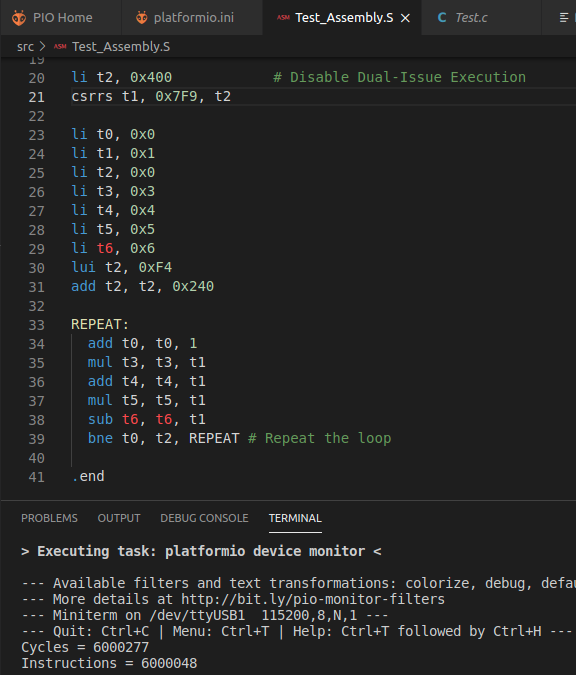
As expected:

IPC = 6000000 instructions / 3000000 cycles = 2

**SINGLE ISSUE:**



The 6 instructions within the loop need 6 cycles to execute. Thus, the expected IPC is 1.



As expected:

IPC = 6000000 instructions / 6000000 cycles = 1

# EXERCISES

1. Create programs similar to those from Figure 2 and 7 using combinations of instructions that show new situations related to dual-issue execution.

Solution not provided.

1. Analyse the differences between the (dual-issue) SweRV EH1 processor and the example superscalar processor proposed in Section 7.7.4 of the textbook by S. Harris and D. Harris, “Digital Design and Computer Architecture: RISC-V Edition” [DDCARV] (shown in Figure 1 for convenience).

Solution not provided.

1. Analyse the program from Figure 7.70 in Section 7.7.4 of DDCARV, which is provided in a PlatformIO project in folder *[RVfpgaPath]/RVfpga/Labs/Lab17/DDCARV\_SuperscalarExample*. Run the program on SweRV EH1, both in simulation and on the board (for the latter remove the nop instructions). Explain the results. If necessary, reorder the program trying to obtain the optimal IPC.

Next, disable the dual-issue execution as explained in this lab – and in

SweRVref.docx (Section 2). Compare the simulation and the results obtained on the

board when compared to when the dual-issue feature is enabled.

000001c0 <REPEAT>:

1c0: 001a8a93 addi s5,s5,1

1c4: 0282ac03 lw s8,40(t0)

1c8: 006c0cb3 add s9,s8,t1

1cc: 41c38c33 sub s8,t2,t3

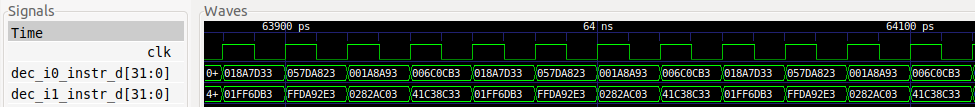
1d0: 018a7d33 and s10,s4,s8

1d4: 01ff6db3 or s11,t5,t6

1d8: 057da823 sw s7,80(s11)

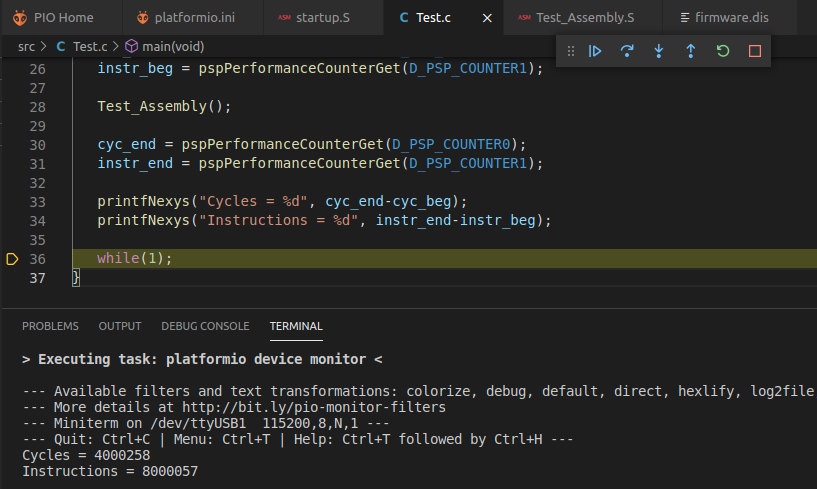
1dc: ffda92e3 bne s5,t4,1c0 <REPEAT>

**Simulation – Dual-Issue:**



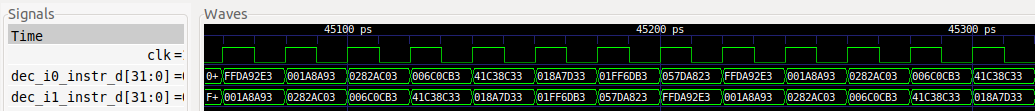
There are no stalls in the loop and 2 instructions per cycle are always executed. Some bypasses are performed and in some cases the Secondary ALU is used as explained in previous labs. You can further analyse these situations.

**Results on the board – Dual-Issue:**



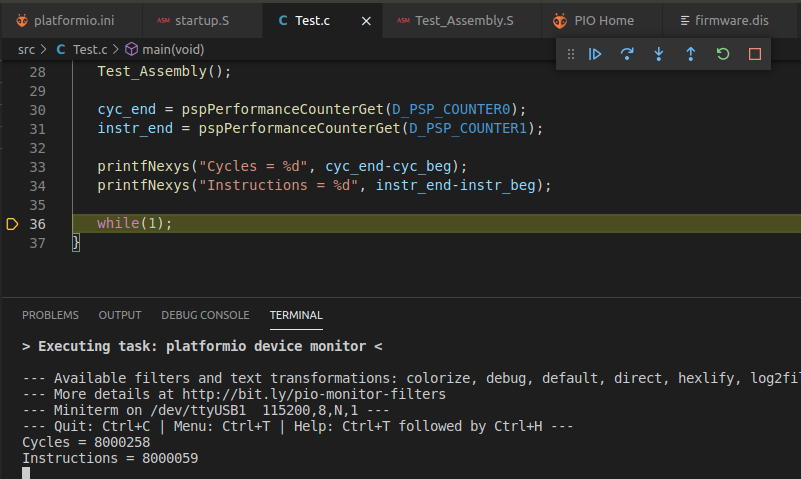
The IPC is equal to 2 with no need to reorder the code.

**Simulation – Single-Issue:**



There are no stalls in the loop and 1 instruction per cycle is always executed.

**Results on the board – Single-Issue:**



The IPC is equal to 1, which is the best we can achieve in a Single-Issue SweRV EH1.

1. Modify the program from Exercise 3 substituting instruction add s9, s8, t1 for instruction add t2, s8, t1. Explain the results. If necessary, reorder the program to try to obtain the optimal IPC.

Then disable the dual-issue execution as explained in this lab and in SweRVref.docx

(Section 2). Compare the simulation and the results obtained on the board when

compared to when the dual-issue feature is enabled.

000001c0 <REPEAT>:

1c0: 001a8a93 addi s5,s5,1

1c4: 0282ac03 lw s8,40(t0)

1c8: 006c03b3 add t2,s8,t1

1cc: 41c38c33 sub s8,t2,t3

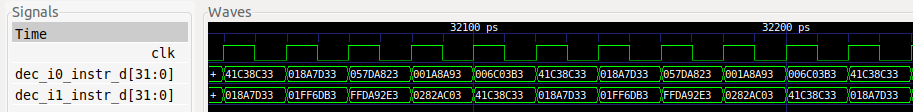
1d0: 018a7d33 and s10,s4,s8

1d4: 01ff6db3 or s11,t5,t6

1d8: 057da823 sw s7,80(s11)

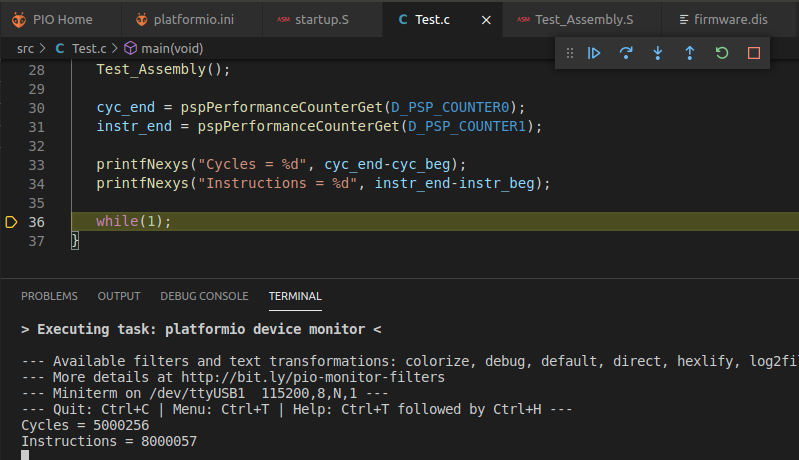
1dc: ffda92e3 bne s5,t4,1c0 <REPEAT>

**Simulation – Dual-Issue:**



There are 2 stalls in the loop due to the dependencies between the AL instructions.

**Results on the board – Dual-Issue:**



The IPC is smaller than 2 due to the stalls.

**Reorder the code:**

000001c0 <REPEAT>:

1c0: 0282ac03 lw s8,40(t0)

1c4: 006c03b3 add t2,s8,t1

1c8: 001a8a93 addi s5,s5,1

1cc: 41c38c33 sub s8,t2,t3

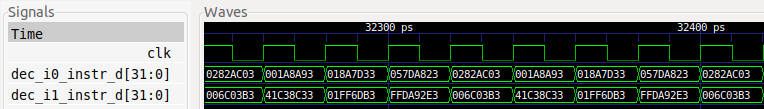
1d0: 018a7d33 and s10,s4,s8

1d4: 01ff6db3 or s11,t5,t6

1d8: 057da823 sw s7,80(s11)

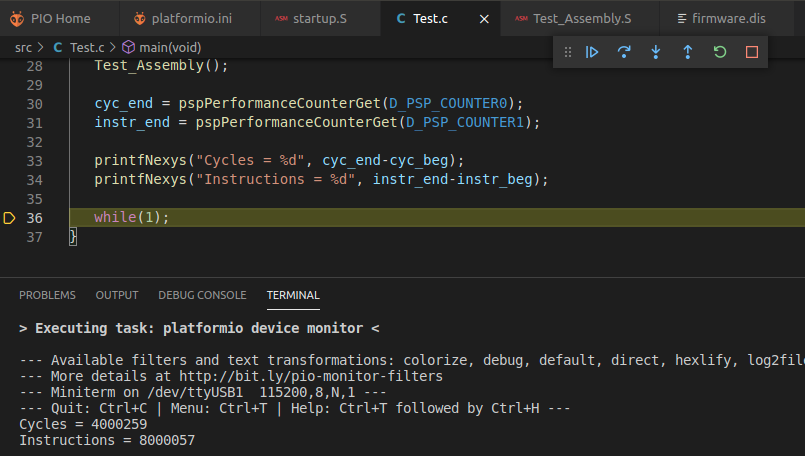
1dc: ffda92e3 bne s5,t4,1c0 <REPEAT>

**Simulation – Dual-Issue:**



There are no stalls in the loop.

**Results on the board – Dual-Issue:**



The IPC is 2.

**Back to the original program:**

000001c0 <REPEAT>:

1c0: 001a8a93 addi s5,s5,1

1c4: 0282ac03 lw s8,40(t0)

1c8: 006c03b3 add t2,s8,t1

1cc: 41c38c33 sub s8,t2,t3

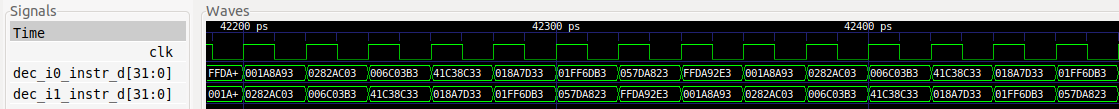
1d0: 018a7d33 and s10,s4,s8

1d4: 01ff6db3 or s11,t5,t6

1d8: 057da823 sw s7,80(s11)

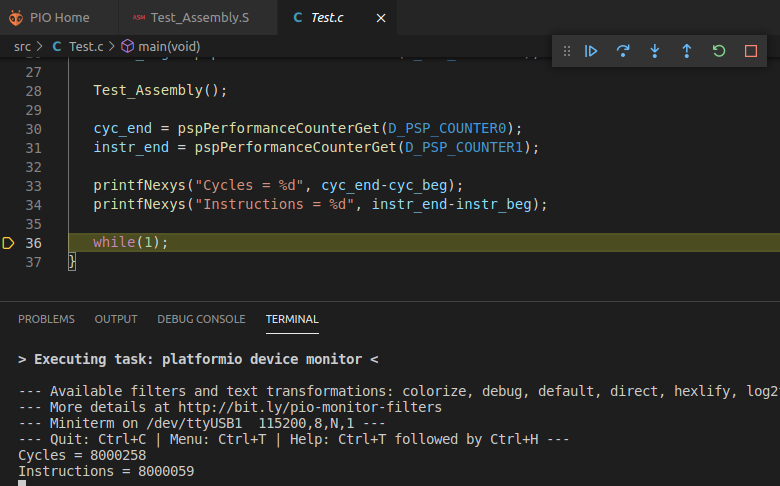
1dc: ffda92e3 bne s5,t4,1c0 <REPEAT>

**Simulation – Single-Issue:**



There are no stalls in the loop.

**Results on the board – Single-Issue:**



The IPC is 1.

1. (*The following exercise is based on exercise 4.31 from the book “Computer Organization and Design – RISC-V Edition”, by Patterson & Hennessy ([HePa]).*)

In this exercise we compare the performance of single- and dual-issue processors, taking into account program transformations that can be made to optimize for dual-issue execution. Problems in this exercise refer to the following loop (written in C):

for(i=0;i!=j;i+=2) b[i]=a[i]–a[i+1];

A compiler doing little or no optimization might produce the following RISC-V assembly code:

li x12, 0 li x13, 8000 li x14, 0 TOP: slli x5, x12, 2 add x6, x10, x5 lw x7, 0(x6) lw x29, 4(x6) sub x30, x7, x29 add x31, x11, x5 sw x30, 0(x31) addi x12, x12, 2 ENT: bne x12, x13, TOP

This code uses the following registers:

Table, calendar

Description automatically generated

This code is provided in *[RVfpgaPath]/RVfpga/Labs/Lab17/PaHe\_SuperscalarExample* with a few minor modifications compared with the code provided by the exercise from the book that do not affect to the behaviour of the program:

* Register x13 is initialized to 8, so that the loop will perform 4 iterations.
* The jal instruction is removed.
* The ld and sd instructions are substituted for lw and sw instructions. This implies changing the accesses from 4- to 8-bytes wide.

Assume a dual-issue, statically scheduled processor that has the following properties:

1. One instruction must be a memory operation; the other must be an arithmetic/logic instruction or a branch.

2. The processor has all possible forwarding paths between stages.

3. The processor has perfect branch prediction.

4. Two instruction may not issue together if one depends on the other.

5. If a stall is necessary, both instructions in a stage must stall.

1. Compare the properties of this example processor and the properties of the SweRV EH1 processor.
2. Draw a pipeline diagram and a simulation showing how a random iteration of the loop (except the first one) of the RISC-V code given above executes on the dual-issue SweRV EH1 processor. Assume that the loop exits after four thousand iterations (this is the case in the above code).
3. What is the speedup of going from a single-issue to a dual-issue SweRV EH1 processor? Explain the results. Test the program on the board and enable/disable dual-issue execution.
4. Rearrange/rewrite the RISC-V code given above to achieve better performance on the two-issue SweRV EH1 processor. (However, do not unroll the loop.)
5. Now, unroll the RISC-V code so that each iteration of the unrolled loop handles two iterations of the original loop. Then, rearrange/rewrite your unrolled code to achieve better performance on the two-issue SweRV EH1 processor.

b)

0000018c <TOP>:

18c: 00261293 slli t0,a2,0x2

190: 00550333 add t1,a0,t0

194: 00032383 lw t2,0(t1)

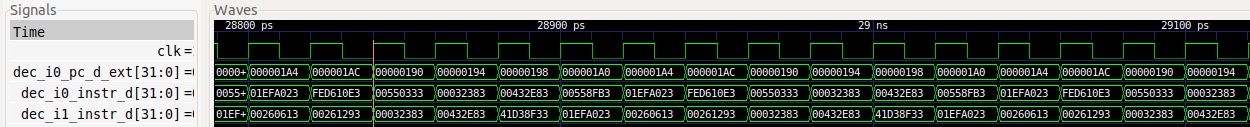
198: 00432e83 lw t4,4(t1)

19c: 41d38f33 sub t5,t2,t4

1a0: 00558fb3 add t6,a1,t0

1a4: 01efa023 sw t5,0(t6)

1a8: 00260613 addi a2,a2,2



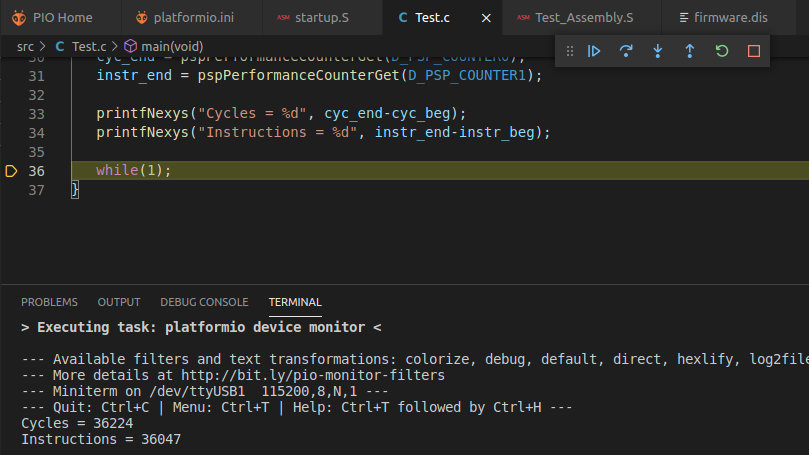
sw t5,0(t6) delayed because of RAW Data Hazard with add t6,a1,t0.

Delayed because of Structural Hazard between the two lw instructions.

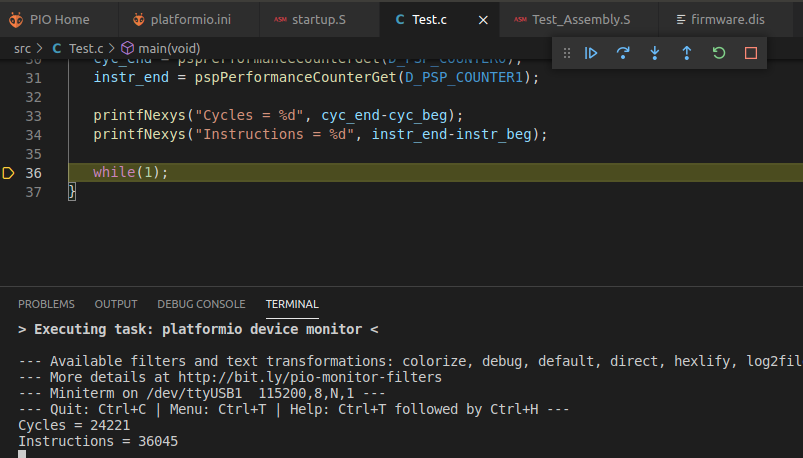
lw t2,0(t1) delayed because of RAW Data Hazard with add t1,a0,t0. This situation was examined in the last task of Lab 15 – Section 2.A.

c)

**Single Issue:**



**Dual Issue:**



SPEEDUP = Time\_Single / Time\_Dual = 36224 / 24221 ≈ 1.5 (the ideal speedup would be 2, but due to the stalls analysed at (b), it is only of 1.5)

d)

TOP: slli x5, x12, 2 addi x12, x12, 2 add x6, x10, x5 lw x7, 0(x6) add x31, x11, x5 lw x29, 4(x6) sub x30, x7, x29 sw x30, 0(x31) ENT: bne x12, x13, TOP

0000018c <TOP>:

18c: 00261293 slli t0,a2,0x2

190: 00260613 addi a2,a2,2

194: 00550333 add t1,a0,t0

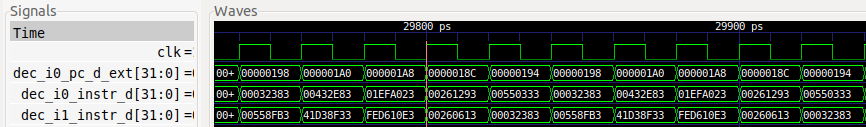
198: 00032383 lw t2,0(t1)

19c: 00558fb3 add t6,a1,t0

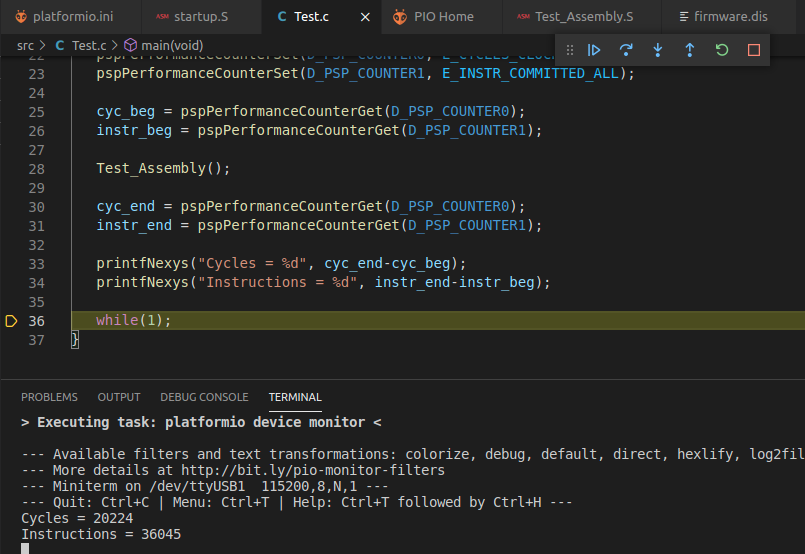
1a0: 00432e83 lw t4,4(t1)

1a4: 41d38f33 sub t5,t2,t4

1a8: 01efa023 sw t5,0(t6)



Delayed because of RAW Data Hazard add-lw



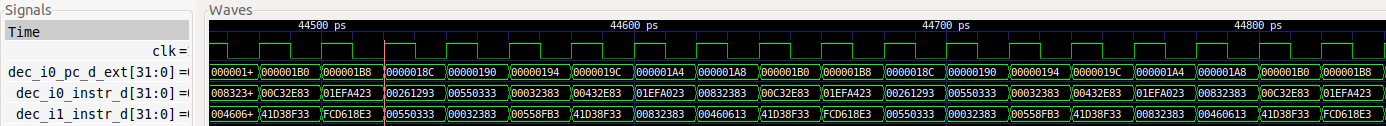
SPEEDUP = Time\_Single / Time\_Dual = 36224 / 20224 ≈ 1.8

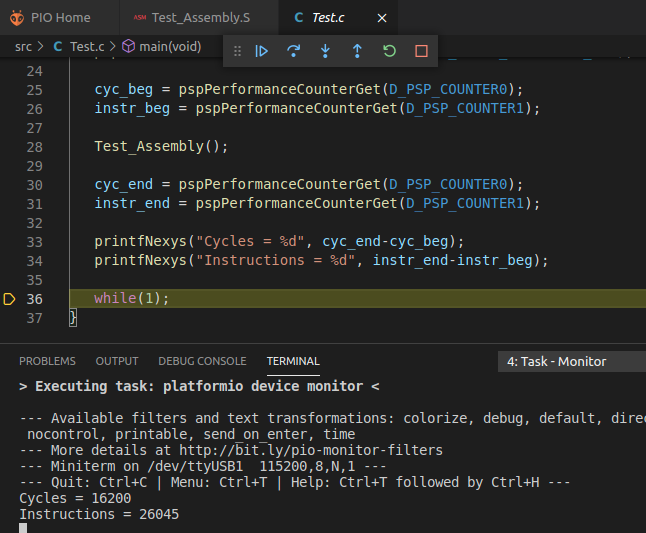
e)

TOP: slli x5, x12, 2 add x6, x10, x5

lw x7, 0(x6) add x31, x11, x5 lw x29, 4(x6) sub x30, x7, x29 sw x30, 0(x31) lw x7, 8(x6) addi x12, x12, 4 lw x29, 12(x6) sub x30, x7, x29 sw x30, 8(x31)

ENT: bne x12, x13, TOP





Thanks to unrolling and rewriting, performance increases even more.

1. (*The following exercise is based on exercises 7.30, 7.32 and 7.34 from Chapter 7 of DDCARV.*)

Suppose the SweRV EH1 processor is running the following code snippet. Recall that SweRV EH1 has a Hazard Unit. You may assume a memory system that returns the result within one cycle (for that purpose we use the DCCM, and we insert the code snippet into a loop and avoid the first iteration so that there are no I$ misses).

addi s1, t0, 11 # t0 contains the base address of the DCCM lw s2, 25(s1) lw s5, 16(s2) add s3, s2, s5 or s4, s3, t4 and s2, s3, s4

1. Simulate the program with Verilator and GTKWave. Analyse the results and for each cycle, specify:

\* Which instructions are decoded, issued to execution and commited?

\* Which registers are being written and which are being read?

\* What forwarding and stalls occur?

1. What is the CPI of the processor on this program? First answer theoretically and then confirm your answer by executing the program on the board.
2. Perform the same analysis on the single-issue processor and compare the results with the results from the dual-issue processor.

A PlatformIO project is provided at: *[RVfpgaPath]/RVfpga/Labs/Lab17/DDCARV\_Exercises-30-32-34*. The program under analysis is inserted in a loop so that it is easier to understand in the simulation (any iteration but the first one can be used for analysis) and it can be measured using performance counters.

1dc: 00b28493 addi s1,t0,11

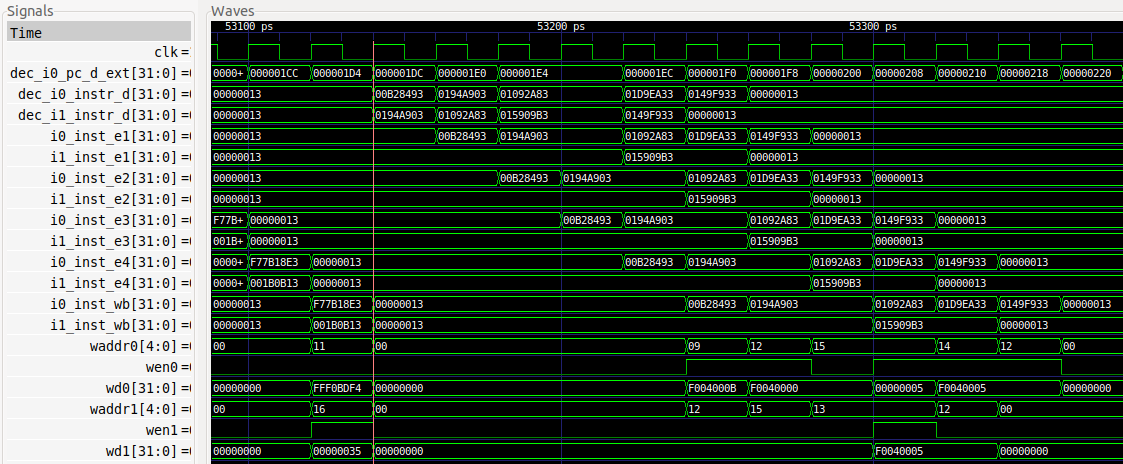
1e0: 0194a903 lw s2,25(s1)

1e4: 01092a83 lw s5,16(s2)

1e8: 015909b3 add s3,s2,s5

1ec: 01d9ea33 or s4,s3,t4

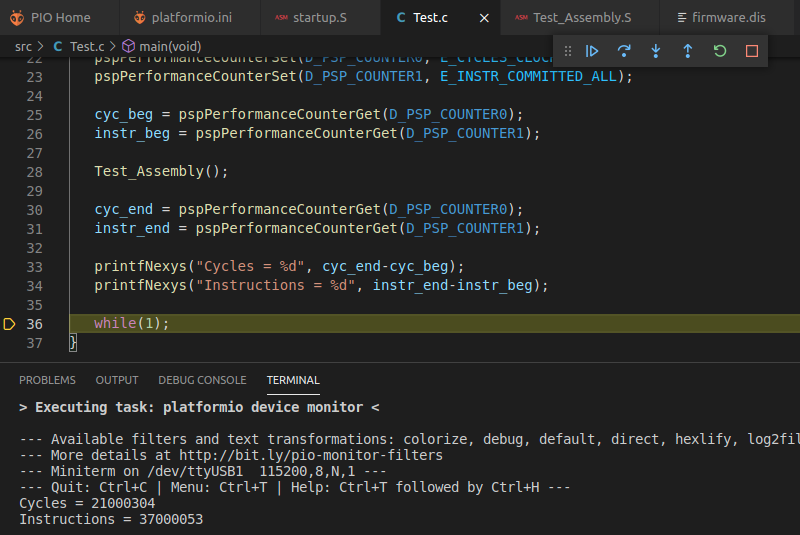
1f0: 0149f933 and s2,s3,s4



**1 2 3 4 5 6 7 8 9 10 11**

* 1. 1st cycle:
     1. Way-0:
        1. Instruction addi (0x00b28493) is at the Decode Stage. At the end of the cycle, this instruction progresses to EX1.
     2. Way-1:
        1. Instruction lw (first load, 0x0194a903) is at the Decode Stage. At the end of the cycle, this instruction stays at Decode Stage, because it depends on the result of the previous addi instruction.
  2. 2nd cycle:
     1. Way-0:
        1. Instruction lw (first load) is again at the Decode Stage, but now it has moved from Way-1 to Way-0. The input operand for the effective address is obtained from the addi instruction. It receives the input operand for the effective address computation through forwarding from the addi instruction. At the end of the cycle, this instruction progresses to DC1.
        2. Instruction addi is at the EX1 stage. During this cycle it obtains the result of the addition and forwards it to the first load.
     2. Way-1:
        1. Instruction lw (second load, 0x01092a83) is at the Decode Stage. At the end of this cycle it cannot progress due to a structural hazard with the first load.
  3. 3rd cycle:
     1. Way-0:
        1. Instruction lw (second load) is again at the Decode Stage, but now it has moved from Way-1 to Way-0. It will not be able to advance given that the second load needs the value read by the first load for computing the effective address. This value is obtained at the end of DC2 (see Lab 13).
        2. Instruction lw (first load) is at the DC1 Stage.
        3. Instruction addi is at the EX2 stage.
     2. Way-1:
        1. Instruction add (0x015909b3) is at the Decode Stage. It will not be able to progress due to a data hazard with the two load instructions.
  4. 4th cycle:
     1. Way-0:
        1. Instruction lw (second load) is again at the Decode Stage. Now it will be able to progress, as the LSU Pipe is free and its operand is obtained through forwarding.
        2. Instruction lw (first load) is at the DC2 Stage.
        3. Instruction addi is at the EX3 stage.
     2. Way-1:
        1. Instruction add is again at the Decode Stage. It obtains its two input operands from the two loads, so it will execute in the Secondary ALU (see Lab 15).
  5. 5th cycle:
     1. Way-0:
        1. The or (0x01d9ea33) instruction is at the Decode Stage. It will be able to progress, obtaining the first operand through forwarding.
        2. Instruction lw (second load) is at the DC1 Stage. It receives the input operand for computing the effective address from the first load, and it will be able to progress.
        3. Instruction lw (first load) is at the DC3 Stage.
        4. Instruction addi is at the Commit Stage (EX4).
     2. Way-1:
        1. The and instruction is at the Decode Stage. It cannot advance because it has a data hazard with the or instruction which is at Way-0, Decode stage (see Lab 15).
        2. Instruction add is at the EX1 Stage.
  6. 6th cycle:
     1. Way-0:
        1. The and instruction is again at the Decode Stage, but it has moved from Way-1 to Way-0. It obtains its two input operands through forwarding and it will be able to progress.
        2. The or instruction is at the EX1 Stage. It forwards the result to the and instruction.
        3. Instruction lw (second load) is at the DC2 Stage.
        4. Instruction lw (first load) is at the Commit Stage.
        5. Instruction addi is at the Write-Back Stage (EX5). Register x9 is written to 0xF004000B.
     2. Way-1:
        1. Instruction add is at the EX2 Stage.
  7. 7th cycle:
     1. Way-0:
        1. The and (0x0149f933) instruction is at the EX1 Stage.
        2. The or instruction is at the EX2 Stage.
        3. Instruction lw (second load) is at the DC3 Stage.
        4. Instruction lw (first load) is at the WB Stage. Register x12 is written to 0xF0040000.
     2. Way-1:
        1. Instruction add is at the EX3 Stage.

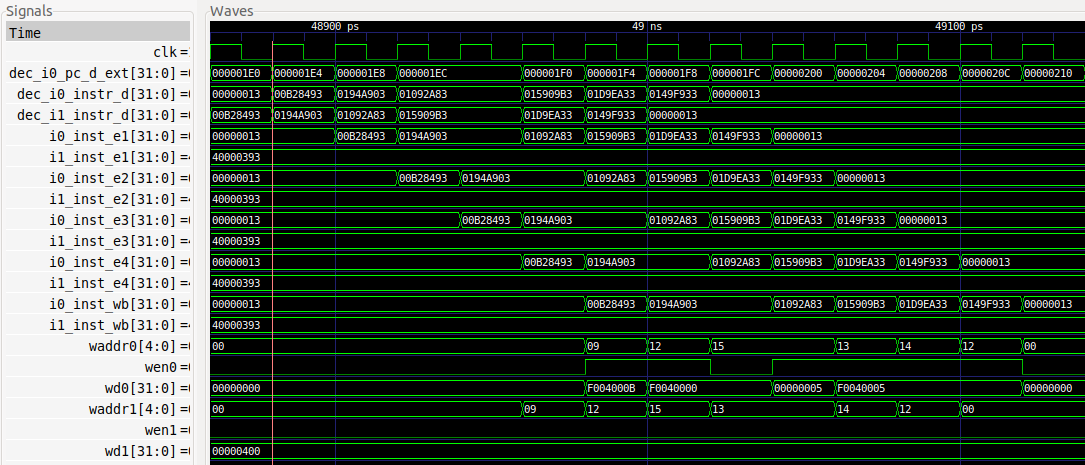
1. 8th cycle:
   * 1. Way-0:
        1. The and instruction is at the EX2 Stage.
        2. The or instruction is at the EX3 Stage.
        3. Instruction lw (second load) is at the Commit Stage.
     2. Way-1:
        1. Instruction add is at the Commit Stage.
2. 9th cycle:
   * 1. Way-0:
        1. The and instruction is at the EX3 Stage.
        2. The or instruction is at the Commit Stage.
        3. Instruction lw (second load) is at the WB Stage. Register x15 is written to 0x00000005.
     2. Way-1:
        1. Instruction add is at the WB Stage. Register x13 is written to 0xF0040005.
3. 10th cycle:
   * 1. Way-0:
        1. The and instruction is at the Commit Stage.
        2. The or instruction is at the WB Stage. Register x14 is written to 0xF0040005.
4. 11th cycle:
   * 1. Way-0:
        1. The and instruction is at the WB Stage. Register x12 is written to 0xF0040005.



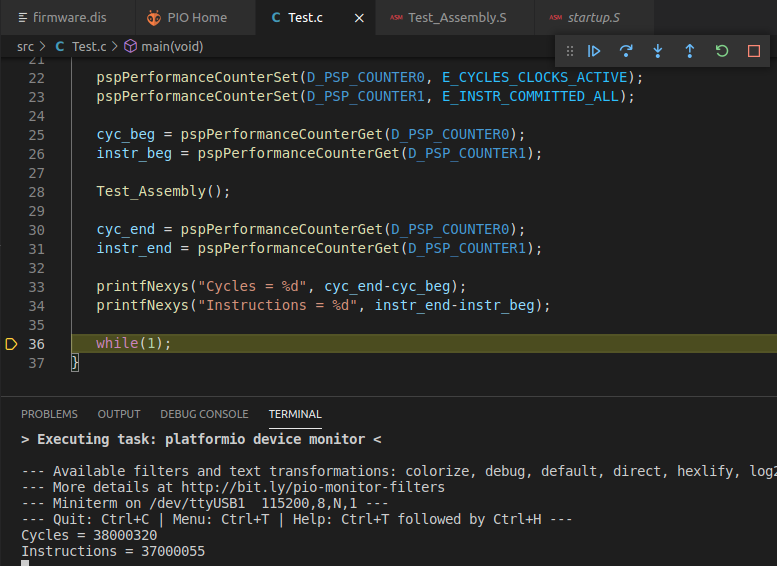
Note that we must take into account that in this calculation we have to remove all the extra instructions that we have included: first addi, 29 nops and a final bne. They all can be executed in ½ cycle (there is a free slot in the execution of the last instruction of our program).

Thus: **IPC** = (37-31) (21-15.5-0.5) = 6 / 6 = **1**

Which is exactly what we observe in our simulation.



1 Instruction is executed per cycle except in the data hazard between the two loads (0x0194a903 and 0x01092a83), which inserts one bubble in the pipeline and 1 cycle is lost.



As before, there are 31 extra instructions that we have to remove for computing the IPC. They all can be executed in 1 cycle.

Thus: **IPC** = (37-31) / (38-31) = **6 / 7**

Which is exactly what we observe in our simulation.

1. (*The following exercise is based on exercises 7.31, 7.33 and 7.35 from Chapter 7 of DDCARV.*)

Repeat exercise 7 for the following code snippet.

addi s1, t0, 52 addi s0, s1, -4 lw s3, 16(s0) sw s3, 20(s0) xor s2, s0, s3 or s2, s2, s3

A PlatformIO project is provided at: *[RVfpgaPath]/RVfpga/Labs/Lab17/DDCARV\_Exercises-31-33-35*.

1d4: 03428493 addi s1,t0,52

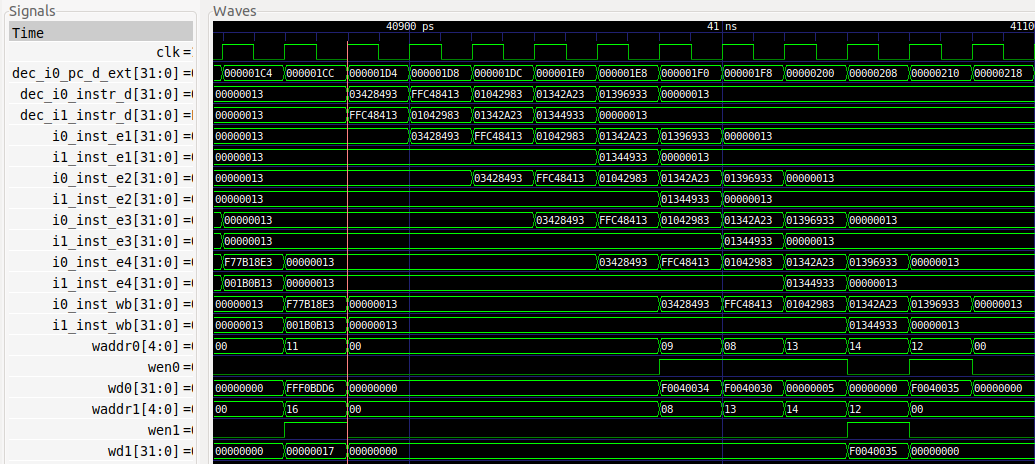
1d8: ffc48413 addi s0,s1,-4

1dc: 01042983 lw s3,16(s0)

1e0: 01342a23 sw s3,20(s0)

1e4: 01344933 xor s2,s0,s3

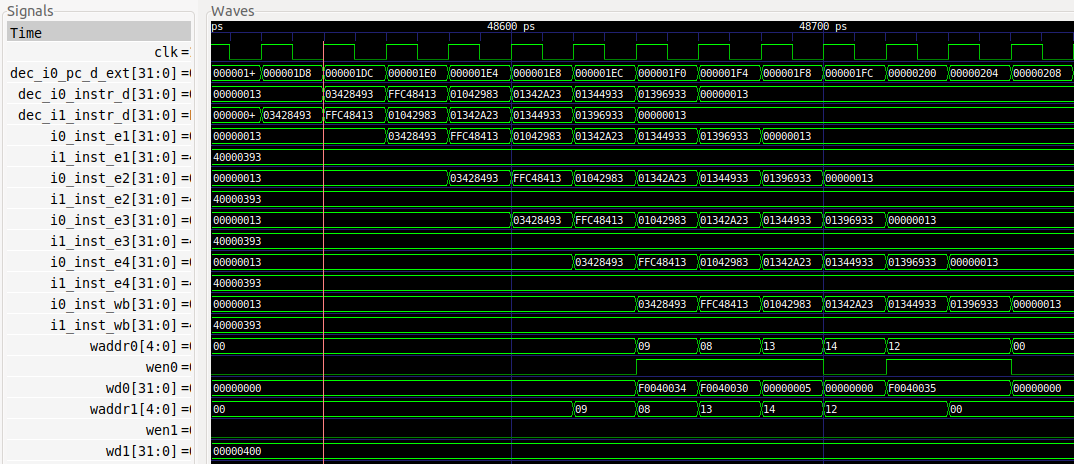
1e8: 01396933 or s2,s2,s3



**1 2 3 4 5 6 7 8 9 10**

There is a hazard between all consecutive instructions except in the case of the sw (0x01342a23) and the xor (0x01344933). All of them are data hazards except in the case of the lw (0x01042983) and the sw (0x01342a23), which present a structural hazard. According to the simulation, only 1 instruction is executed per cycle except for the pair sw-xor, for which the xor instruction is executed through Way 1.

In this case: **IPC = 6 / 5**



In this case: **IPC** = 6 / 6 = **1**