

## TAREFAS

**TAREFA:** Pode realizar um estudo semelhante para a instrução `mul` como o realizado no Lab 12 para as instruções aritméticas-lógicas: veja o fluxo da instrução passando pelos andares do pipeline, analise os bits de controle (lembre-se do Apêndice D do Lab 11 que há um tipo de estrutura específica para a instrução `mul` chamada `mul_pkt_t` e há um sinal definido no módulo `dec_decode_ctl` chamado `mul_p`) etc.

Solução não fornecida.

**TAREFA:** Inspeccione o código Verilog do `exu_mul_ctl` e veja como a multiplicação é computada. Lembre-se de que o RISC-V inclui 4 instruções de multiplicação (`mul`, `mulh`, `mulhsu` e `mulhu`), e todas elas devem ser compatíveis com o hardware.

Como um exercício opcional, substitua a Multiply Unit por sua própria unidade ou por uma da 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.load),           .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 logic signed [65:0] prod_e2;
106
107 assign prod_e2[65:0] = a_ff_e2 * b_ff_e2;
108
109 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));
110
111 rvdffe    #(64) prod_e3_ff     (*, .din(prod_e2[63:0]),     .dout(prod_e3[63:0]),     .en(mul_c1_e3_clken));
112
113
114
115 // ----- E3 Logic Stage -----
116
117
118
119 assign out[31:0] = low_e3 ? prod_e3[31:0] : prod_e3[63:32];
120

```

- As entradas e os bits de controle produzidos no andar Decode são registrados nas linhas 72-81.

M1:

- No caso de uma dependência de dados entre a multiplicação e uma leitura anterior, ocorre um forwarding nas linhas 87 a 88.
- Além disso, o tratamento do sinal dos operandos de entrada é determinado nas linhas 90-91. Lembre-se de que o RISC-V inclui três versões da operação "multiply high": mulh, mulhsu e mulhu.

- Esses valores são propagados para M2.

M2:

- A multiplicação real é realizada na linha 108.

M3:

- A parte baixa/alta é retornada em `out[31:0]` na linha 119. A parte baixa é selecionada no caso de uma instrução `mulh`, enquanto a parte alta é selecionada no caso de qualquer uma das três instruções `mulh`.

**TAREFA:** Verificar se esse par de 32 bits (0x03de02b3 e 0x03ff0333) corresponde às instruções `mul t0,t3,t4` e `mul t1,t5,t6` na arquitetura RISC-V.

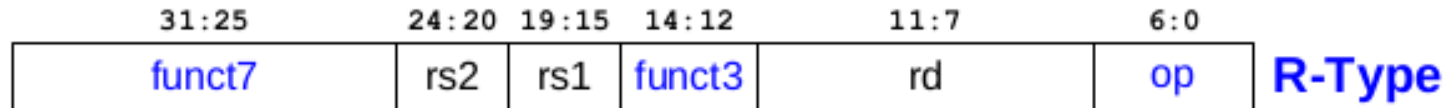
**0x03de02b3 → 0000001 11101 11100 000 00101 0110011**

**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**

Do Apêndice B do DDCARV:



op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000001	R	mul rd, rs1, rs2	multiply	rd = (rs1 x rs2) <sub>31:0</sub>

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

**TAREFA:** Replique a simulação da Figura 2 no seu computador e analise-a mais detalhadamente.

Solução fornecida no documento principal do Lab 14.

**TAREFA:** Compare a ilustração da Figura 3 com a simulação da Figura 2, concentrando-se nas duas instruções `mul`. Especificamente, analise como as duas instruções são atribuídas às duas vias nos andares Align e Decode.

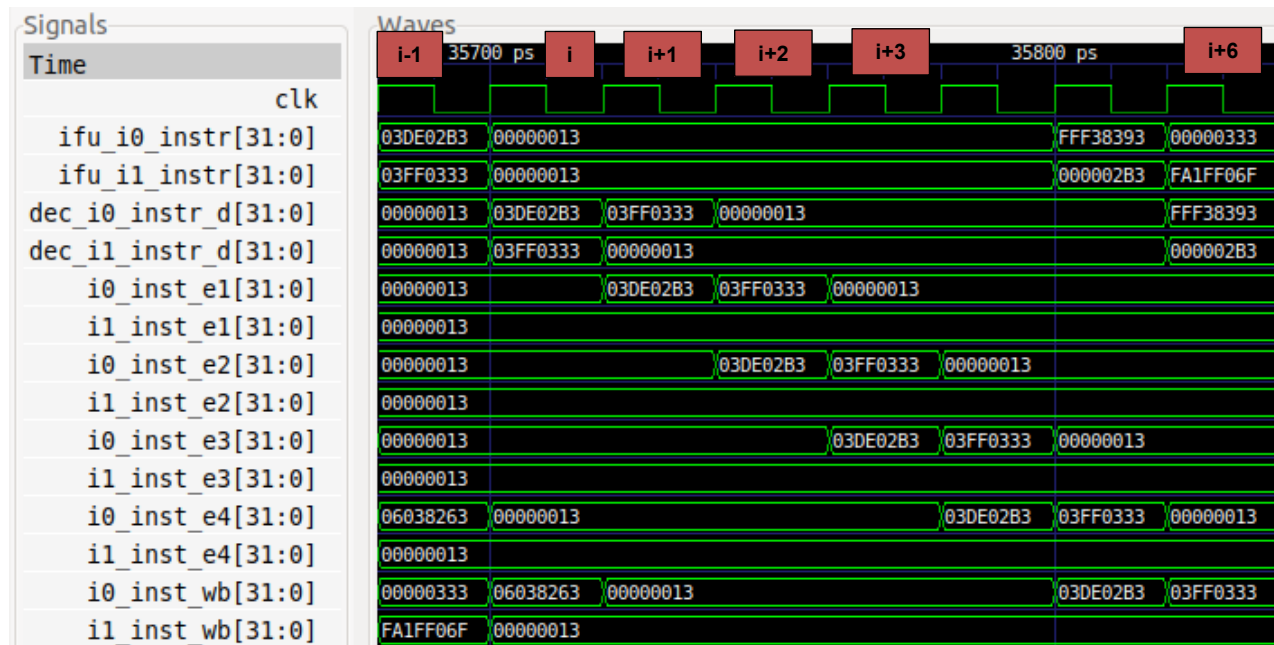
- No módulo `ifu_aln_ctl` (andar Align), as duas instruções são atribuídas a:
  - Via 0: `ifu_i0_instr`

- Via 1: ifu\_i1\_instr
- No módulo **dec\_ib\_ctl**, as duas instruções são armazenadas no buffer de Align para Decode:
  - Via 0: ifu\_i0\_instr → dec\_i0\_instr\_d
  - Via 1: ifu\_i1\_instr → dec\_i1\_instr\_d
- No módulo **dec\_decode\_ctl** (andar Decode), as duas instruções são programadas para os pipes correspondentes, se possível.

Depois de enviadas, elas continuam pelos três andares de execução, Commit e Writeback:

- Via 0: i0\_inst\_e1 - i0\_inst\_e2 - i0\_inst\_e3 - i0\_inst\_e4 - i0\_inst\_wb
- Via 1: i1\_inst\_e1 - i1\_inst\_e2 - i1\_inst\_e3 - i1\_inst\_e4 - i1\_inst\_wb

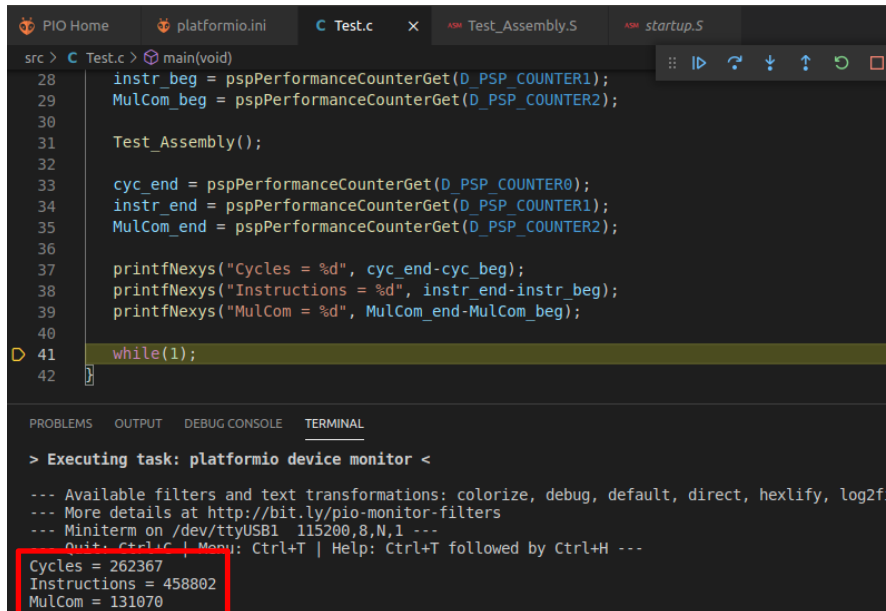
Fornecemos um ficheiro `.tcl` chamado `[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instruction/test_AssignmentWays.tcl` que inclui todos esses sinais.



- No ciclo i-1 (não mostrado na Figura 2 nem na Figura 3), as duas instruções `mul` estão no andar Align: a primeira é atribuída à Via 0 (`ifu_i0_instr = 0x03de02b3`) e a segunda é atribuída à Via 1 (`ifu_i1_instr = 0x03ff0333`) no módulo **ifu\_aln\_ctl**.

- No ciclo  $i$ , as duas instruções foram propagadas para o andar Decode no módulo **dec\_ib\_ctl**: a primeira continua na Via 0 (`dec_i0_instr_d = 0x03de02b3`) e a segunda continua na Via 1 (`dec_i1_instr_d = 0x03ff0333`).
- No ciclo  $i+1$ , a primeira instrução `mul` foi propagada para o andar M1 no módulo **dec\_decode\_ctl** (`i0_inst_e1 = 0x03de02b3`). Entretanto, a segunda instrução `mul` não pôde ser propagada devido ao conflito estrutural analisado no Lab e, portanto, uma bolha foi inserida no primeiro andar de execução da Via 1: `i1_inst_e1 = 0x00000013`. Além disso, como a via 0 foi libertada no andar Decode, o segundo `mul` foi reatribuído a essa via: `dec_i0_instr_d = 0x03ff0333`.
- No ciclo  $i+2$ , a segunda instrução `mul` é propagada para o andar M1, que agora está livre (`i0_inst_e1 = 0x03ff0333`), e a primeira instrução `mul` é propagada para o andar M2.
- Nos ciclos  $i+3$  a  $i+6$ , as duas instruções `mul` progridem pelo pipeline sem paradas até o andar Writeback.

**TAREFA:** Remova as instruções `nop` incluídas no ciclo e meça os diferentes eventos (ciclos, instruções/multiplicações completadas, etc.) usando os contadores de desempenho disponíveis no SweRV EH1, conforme explicado no Lab 11. O número de ciclos está de acordo com o esperado depois de analisar a simulação da Figura 2? Justifique sua resposta. Agora, reordene o código dentro do ciclo tentando alcançar a taxa de transferência ideal. Justifique os resultados obtidos no código original e no código reordenado.



```
src > C 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
```

```
> Executing task: platformio device monitor <

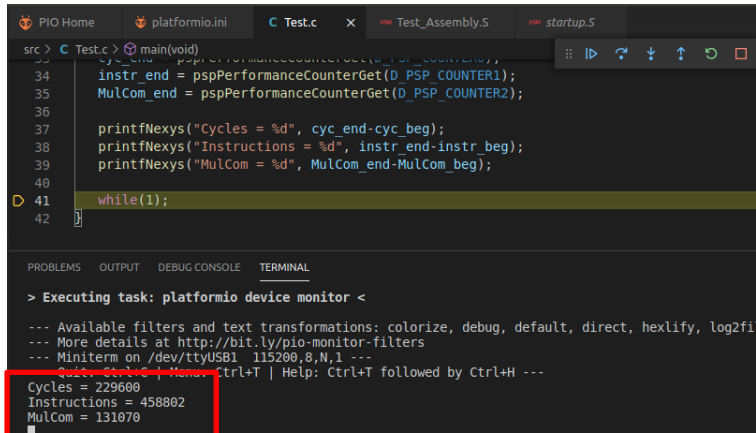
--- Available filters and text transformations: colorize, debug, default, direct, hexlify, log2fi
--- More details at http://bit.ly/pio-monitor-filters
--- Miniterm on /dev/ttyUSB1 115200,8,N,1 ---
--- Quit: Ctrl+C | Menu: Ctrl+T | Help: Ctrl+T followed by Ctrl+H ---

Cycles = 262367
Instructions = 458802
MulCom = 131070
```

$IPC = 458000 / 262000 = 1,748$ . O IPC é um pouco menor do que o ideal porque a segunda instrução mul deve esperar um ciclo devido ao conflito estrutural, conforme explicado no Lab.

Se reordenarmos o código, inserindo entre as duas instruções mul a atualização do índice do ciclo, obteremos o IPC ideal, pois preenchemos a bolha introduzida pelo conflito estrutural com uma instrução útil.

```
22 REPEAT:
23     beq t2, zero, OUT      # Stay in the loop?
24     mul t0, t3, t4         # t0 = t3 * t4
25     add t2, t2, -1
26     mul t1, t5, t6         # t1 = t5 * t6
27     add t0, zero, zero
28     add t1, zero, zero
29     j REPEAT
30 OUT:
```



```
src > C Test.c > main(void)
34   cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
35   instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
36   MulCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
37
38   printfNexys("Cycles = %d", cyc_end-cyc_beg);
39   printfNexys("Instructions = %d", instr_end-instr_beg);
40   printfNexys("MulCom = %d", MulCom_end-MulCom_beg);
41
42   while(1);
43 }
```

```
> 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 = 229600
Instructions = 458802
MulCom = 131070
```

$$\text{IPC} = 458000 / 229000 = 2$$

**TAREFA:** A pasta `[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instr_Accumul_C-Lang` fornece o projeto PlatformIO de um programa em C que acumula a subtração de duas multiplicações num ciclo.

- Analisar o programa C.
  - Execute uma simulação e inspecione uma iteração aleatória do ciclo. Observe que o programa em C é compilado sem otimizações.
  - Meça diferentes eventos (ciclos, instruções/multiplicações completadas, etc.) usando os contadores de desempenho disponíveis no Swe RV EH1, conforme explicado no Lab 11.
- O número de ciclos está de acordo com o esperado após a análise da simulação da Figura 2? Justifique sua resposta.
- Crie um programa análogo em Assembly RISC-V e compare-o com a versão em C. Reordene as instruções tentando obter o melhor IPC possível.
  - Desative a extensão **M** RISC-V no programa C e compare os resultados com o programa original. Para fazer isso, modifique a seguinte linha no ficheiro `platformio.ini` de:

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

Para:

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

Isso evita o uso das instruções da extensão M RISC-V e emula-as usando outras instruções.

- Programa C (original e Disassembly):

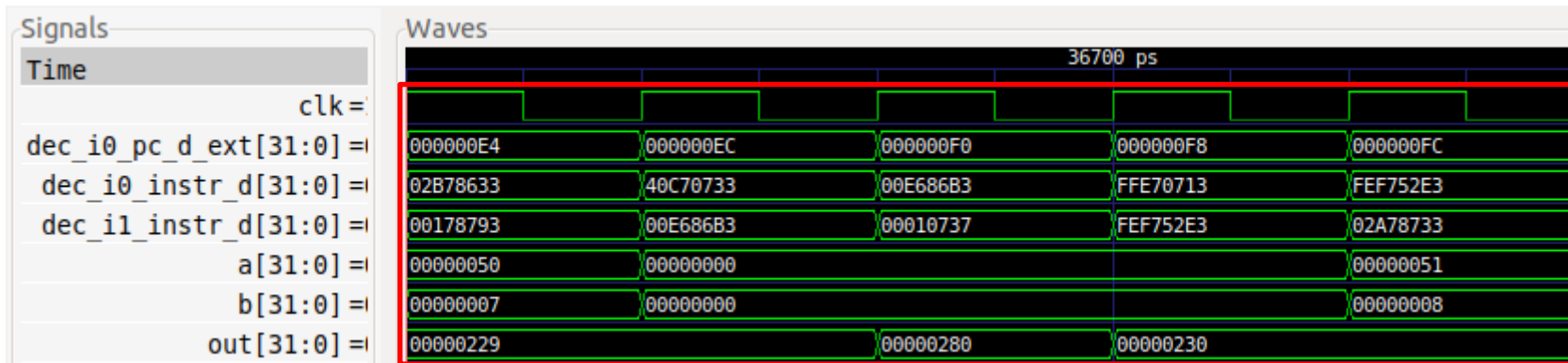
```
11 int Test_C(int a, int d)
12 {
13     int b, c, e=0, i=1;
14     do {
15         b = a*i;
16         c = d*i;
17         i = i+1;
18         e = e + (b-c);
19     } while(i<65535);
20     return(e);
21 }
```

```

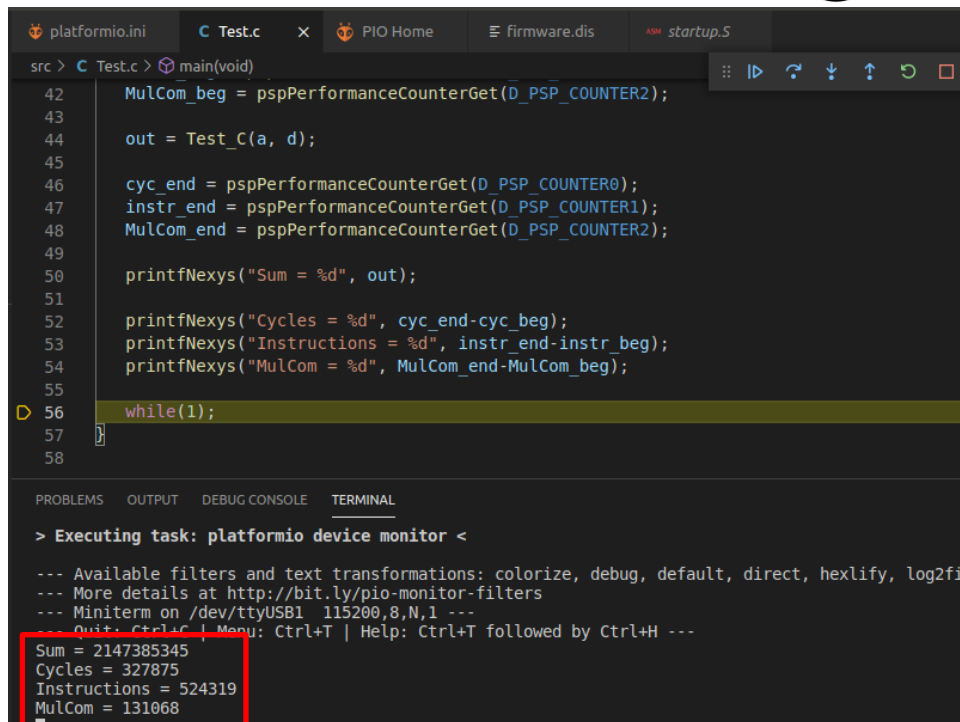
66 000000d8 <Test_C>:
67      d8: 00100793      li  a5,1
68      dc: 00000693      li  a3,0
69      e0: 02a78733      mul a4,a5,a0
70      e4: 02b78633      mul a2,a5,a1
71      e8: 00178793      addi a5,a5,1
72      ec: 40c70733      sub a4,a4,a2
73      f0: 00e686b3      add a3,a3,a4
74      f4: 00010737      lui a4,0x10
75      f8: ffe70713      addi a4,a4,-2 # fffe <_sp+0xc386>
76      fc: fef752e3      bge a4,a5,e0 <Test_C+0x8>
77      100: 00068513      mv  a0,a3
78      104: 00008067      ret

```

- Simulação do programa em C:



- Contadores HW:



```

src > C Test.c > main(void)
42 MulCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
43
44 out = Test_C(a, d);
45
46 cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
47 instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
48 MulCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
49
50 printfNexys("Sum = %d", out);
51
52 printfNexys("Cycles = %d", cyc_end-cyc_beg);
53 printfNexys("Instructions = %d", instr_end-instr_beg);
54 printfNexys("MulCom = %d", MulCom_end-MulCom_beg);
55
56 while(1);
57
58

```

```

> 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 ---

Sum = 2147385345
Cycles = 327875
Instructions = 524319
MulCom = 131068

```

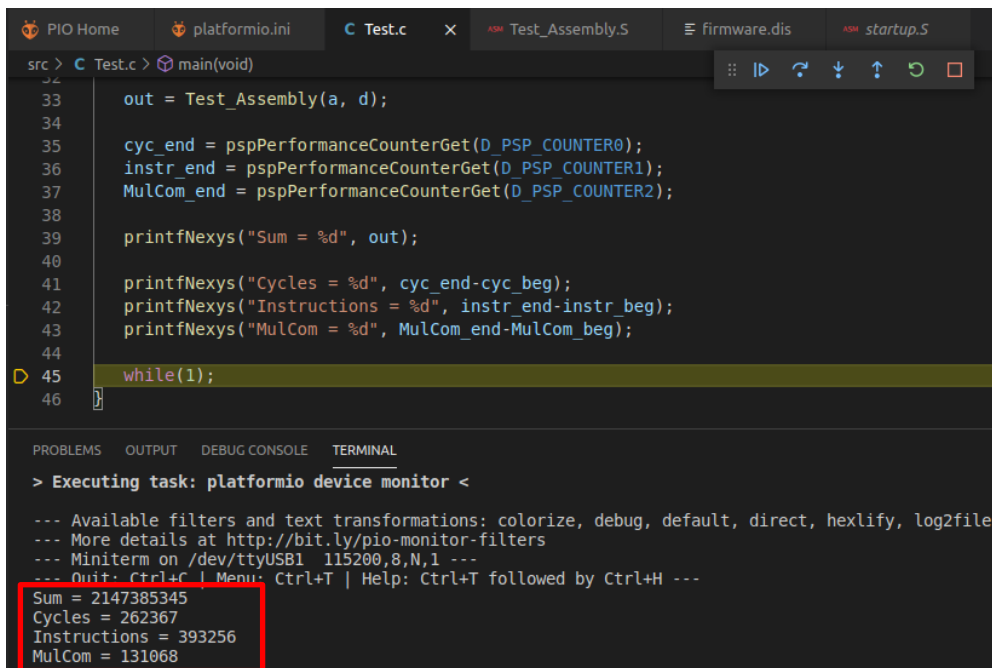
$IPC = 524000 / 327000 = 1,6$ . Alguns ciclos são perdidos devido aos conflitos de dados RAW, que analisaremos no Lab 15.

- O programa Assembly pode ser encontrado em:  
`[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab14/MUL_Instr_Accumul_Assembly`

```

127 000001c4 <Test_Assembly>:
128 1c4: 00100393      li t2,1
129 1c8: 00000e93      li t4,0
130 1cc: 00000f93      li t6,0
131 1d0: 00010637      lui a2,0x10
132 1d4: fff60613      addi a2,a2,-1 # ffff <_sp+0xc567>
133 1d8: 00050e33      add t3,a0,zero
134 1dc: 00058f33      add t5,a1,zero
135
136 000001e0 <REPEAT>:
137 1e0: 027e02b3      mul t0,t3,t2
138 1e4: 027f0333      mul t1,t5,t2
139 1e8: 00138393      addi t2,t2,1
140 1ec: 40628eb3      sub t4,t0,t1
141 1f0: 01df8fb3      add t6,t6,t4
142 1f4: fec396e3      bne t2,a2,1e0 <REPEAT>
143 1f8: 00018533      add a0,t6,zero
144 1fc: 00008067      ret

```



The screenshot shows the PlatformIO IDE with the following components:

- Editor:** Displays the C code in `Test.c`. The code includes headers, defines, and a `main` function that calls `Test_Assembly(a, d)` and prints performance metrics. A `while(1);` loop is at the bottom.
- Terminal:** Shows the output of the program execution. It includes a header for the platformio device monitor and the following results:
 

```

Sum = 2147385345
Cycles = 262367
Instructions = 393256
MulCom = 131068

```

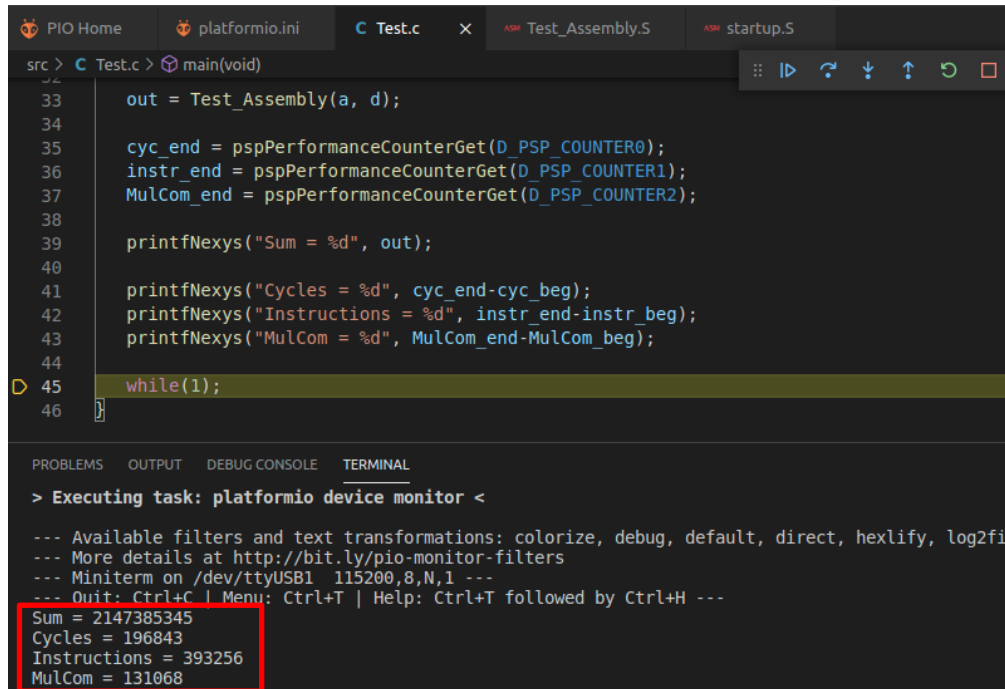
O resultado da soma é o mesmo, pois o programa é o mesmo.

O número de ciclos é um pouco menor, pois a versão Assembly programada manualmente é mais eficiente do que a obtida pelo compilador sem otimizações.

O número de instruções também é um pouco menor.

Reordenamos o ciclo da seguinte forma:

```
15  li  t2, 0x1
16  li  t4, 0x0
17  li  t6, 0x0
18  li  a2, 0xFFFF
19  add t3, a0, zero
20  add t5, a1, zero
21
22  REPEAT:
23      mul t0, t3, t2      # t0 = t3 * t2
24      mul t1, t5, t2      # t1 = t5 * t2
25      sub t4, t0, t1
26      add t2, t2, 1
27      add t6, t6, t4
28      bne t2, a2, REPEAT  # Repeat the loop
29
30  add a0, t6, zero
```



The screenshot shows a code editor with a C file named `Test.c`. The code defines a `main` function that calls `Test_Assembly(a, d)`, then uses Pico Performance Counter (PSP) functions to get cycle, instruction, and multiply-compare counts. It prints these values using `printfNexys` and enters a `while(1);` loop. Below the code, the terminal window shows the output of the program, which is highlighted with a red box:

```

> 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

```

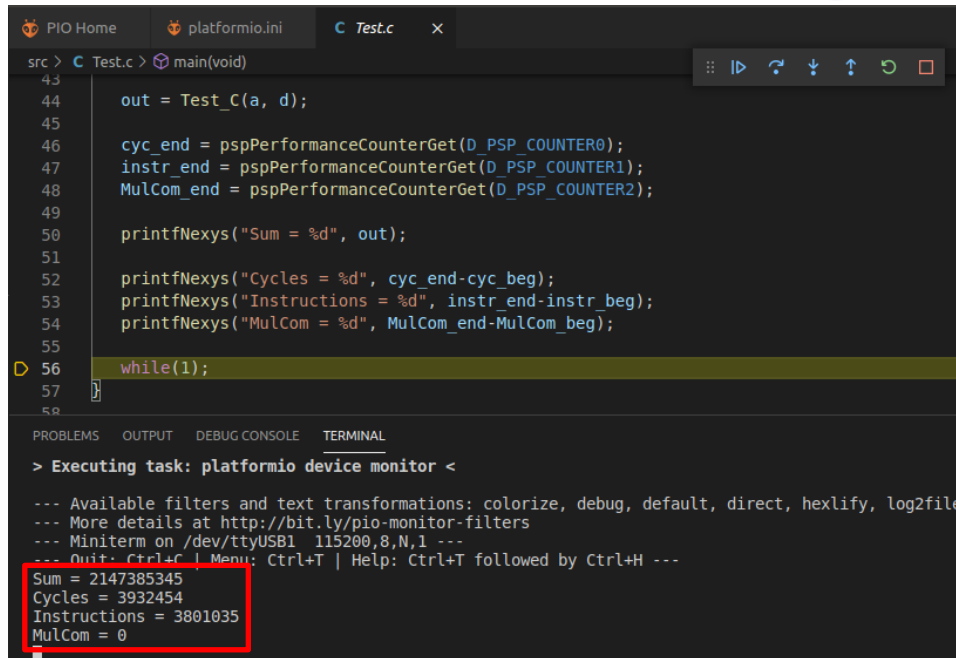
O resultado da soma é o mesmo, pois o programa é o mesmo.

Número de ciclos por iteração =  $196800 / 65500 = 3$

O número de instruções é o mesmo. Número de instruções por iterações =  $393000 / 65500 = 6$

IPC =  $393 / 197 = 1,994$ . Obtemos o IPC ideal.

- Desativar a extensão M:



The screenshot shows a code editor with a C program in `Test.c`. The program calculates a sum, counts cycles, instructions, and multiplies. The output in the terminal shows the results of the execution.

```
src > C Test.c > main(void)
43
44     out = Test_C(a, d);
45
46     cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
47     instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
48     MulCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
49
50     printfNexys("Sum = %d", out);
51
52     printfNexys("Cycles = %d", cyc_end-cyc_beg);
53     printfNexys("Instructions = %d", instr_end-instr_beg);
54     printfNexys("MulCom = %d", MulCom_end-MulCom_beg);
55
56     while(1);
57
58
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

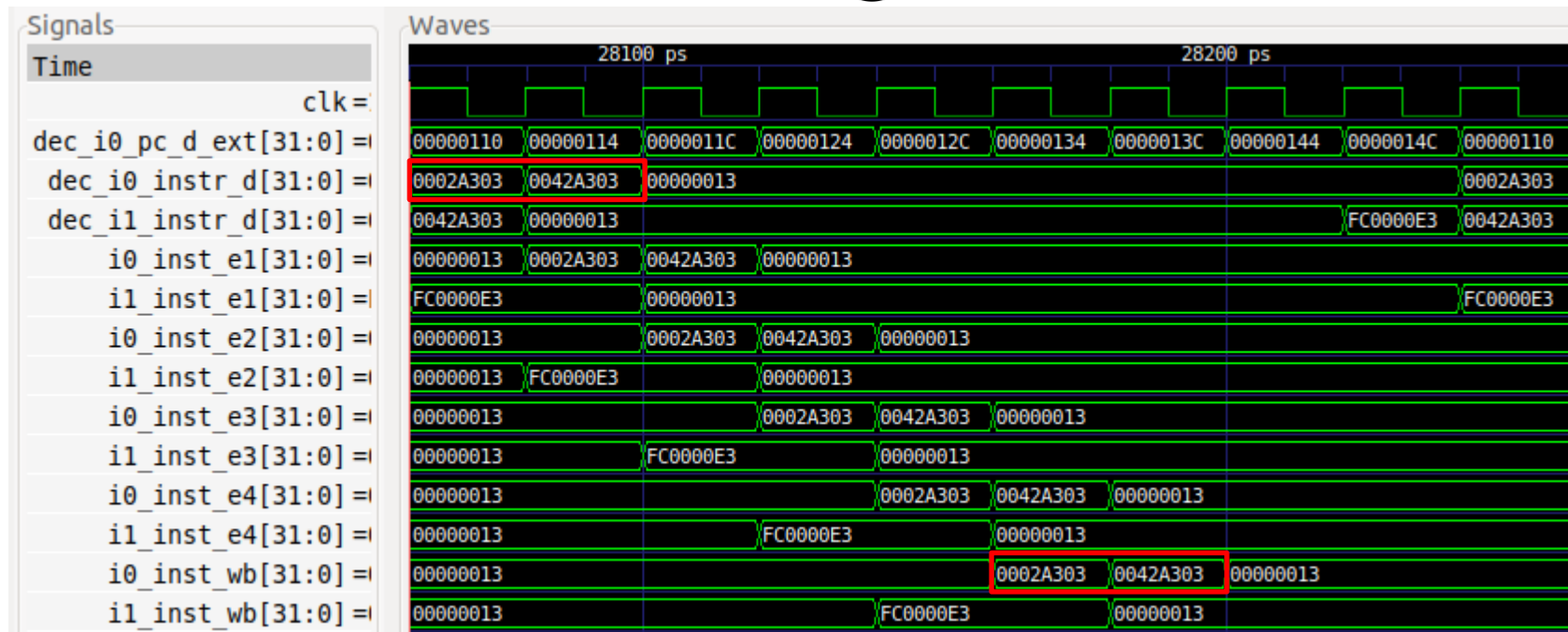
> Executing task: platformio device monitor <

--- 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 = 3932454
Instructions = 3801035
MulCom = 0
```

O resultado da soma é o mesmo, pois o programa é o mesmo.  
 O número de ciclos é muito maior: Cerca de 4 milhões contra cerca de 0,3 milhão.  
 O número de instruções também é muito maior: Cerca de 3 milhões contra cerca de 0,5 milhão.  
 O IPC está melhor agora.  
 Não há multiplicações completadas.

**TAREFA:** Modifique o programa da Figura 1, substituindo as duas instruções `mul` por duas instruções `lw` para a DCCM. Deve observar um conflito estrutural análogo ao analisado nesta seção e resolvido por uma via semelhante.



Como podemos ver na simulação, o comportamento para duas leituras consecutivas é exatamente o mesmo que no caso de duas instruções mul consecutivas.

**TAREFA:** Replique a simulação da Figura 6 no seu computador. Use o ficheiro *test\_NonBlocking.tcl* (fornecido em `[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory`). Aumente o zoom (  ) várias vezes e vá para 60120ps.

Solução fornecida no documento principal do Lab 14.

**TAREFA:** Compare a simulação mostrada na Figura 6 (leitura não-bloqueante) com a simulação mostrada na Figura 14 do Lab 13 (leitura bloqueante). Adicione todos os sinais necessários para a comparação.

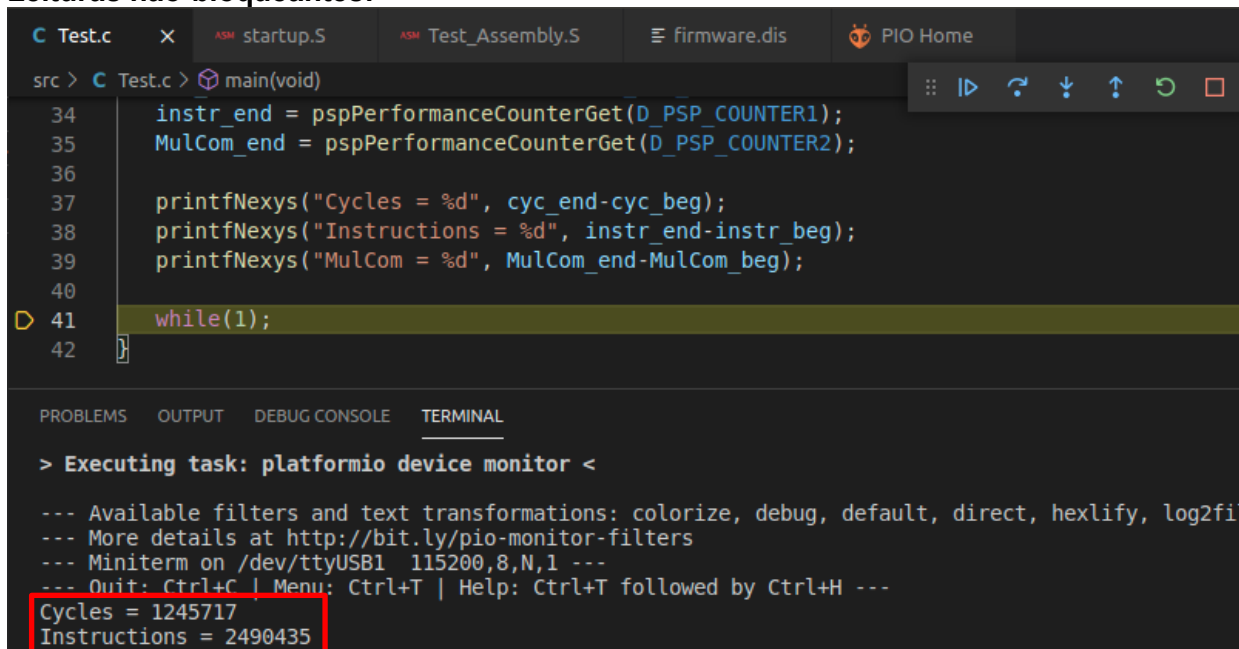
Solução não fornecida.

**TAREFA:** Compare a ilustração da Figura 7 com a simulação da Figura 6 que reproduziu no seu computador. Adicione sinais para ampliar a simulação e aprofundar a compreensão, conforme desejar.

Solução não fornecida.

**TAREFA:** Meça diferentes eventos (ciclos, instruções/leituras completadas, etc.) usando os contadores de desempenho disponíveis no SweRV EH1, conforme explicado no Lab 11. O número de ciclos está de acordo com o esperado após a análise da simulação da Figura 6? Justifique sua resposta.  
Compare esses resultados com os obtidos quando as leituras são configuradas como leituras bloqueantes.

#### Leituras não-bloqueantes:



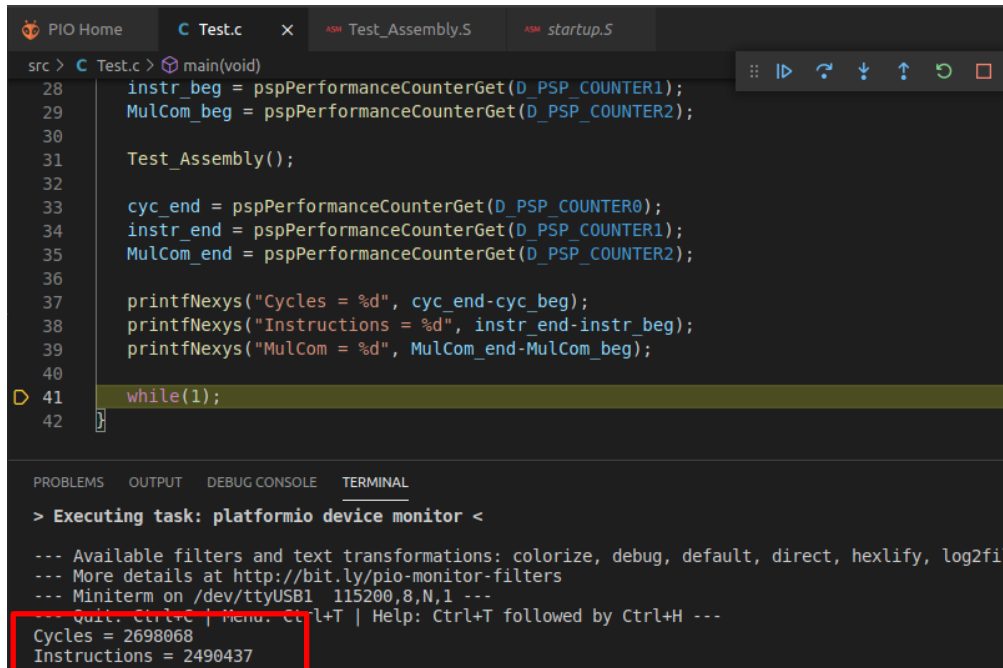
```
C Test.c x ASM startup.S ASM Test_Assembly.S firmware.dis PIO Home
src > C Test.c > main(void)
34 instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
35 MulCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
36
37 printfNexys("Cycles = %d", cyc_end-cyc_beg);
38 printfNexys("Instructions = %d", 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 = 1245717
Instructions = 2490435
```

O IPC obtido ( $IPC = 2490 / 1245 = 2$ ) é o ideal graças à carga sem bloqueio.

Leituras bloqueantes:

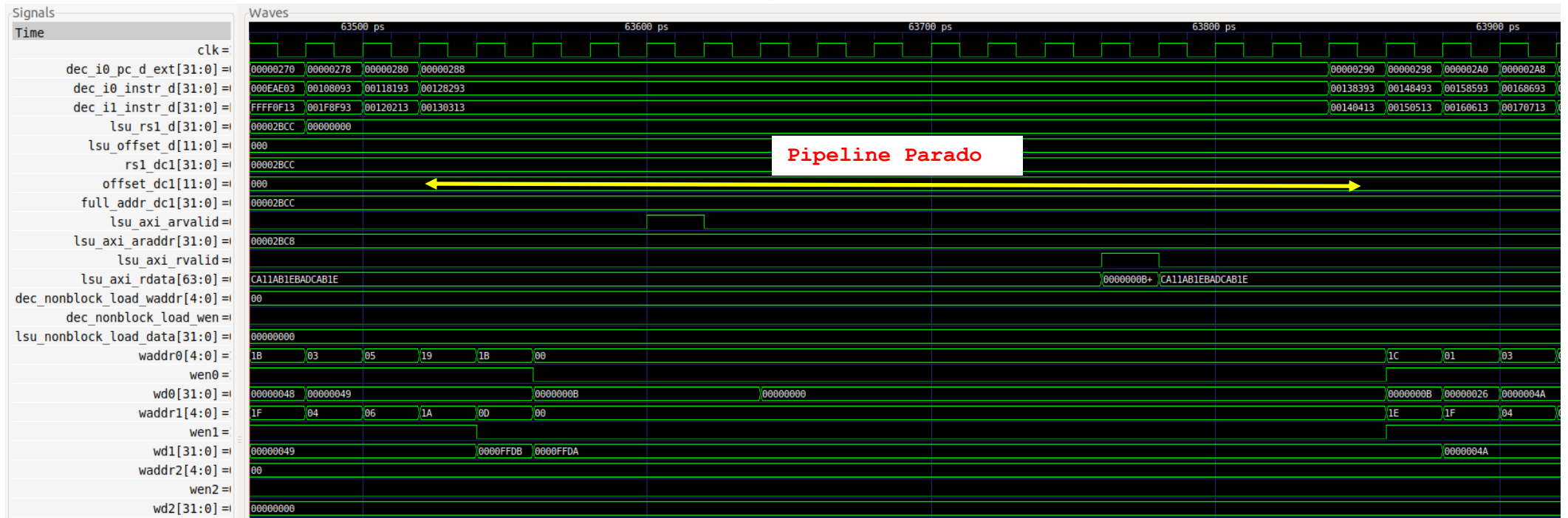


The screenshot shows a code editor with a C file named `Test.c`. The code defines performance counters and prints the number of cycles, instructions, and multiplies completed. A `while(1);` loop is present at the bottom. The terminal output shows the execution results, with the last two lines highlighted in red:

```
src > C 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
```

O número de instruções é o mesmo, mas agora são necessários muito mais ciclos para executar o ciclo, já que as instruções de leitura fazem com que as instruções subsequentes fiquem paradas para que os dados venham da memória. A simulação demonstra isso com mais clareza.



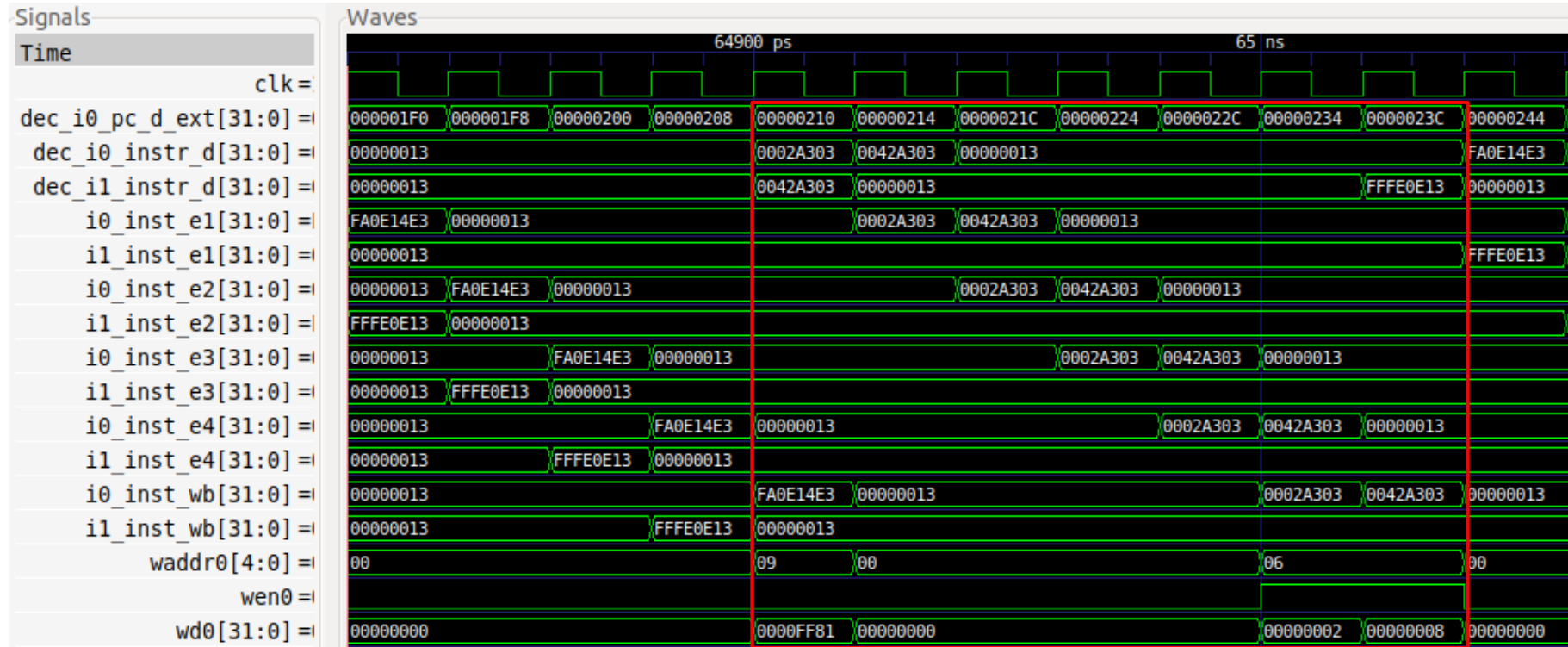
## EXERCÍCIOS

1. Analise, tanto na simulação como na placa, o conflito estrutural que ocorre entre duas instruções de memória consecutivas (pode analisar qualquer combinação de duas instruções de memória consecutivas, como leituras e escritas) que chegam ao L/S Pipe no mesmo ciclo. Teste tanto para leituras não-bloqueantes quanto para leituras bloqueantes. Pode usar o projeto PlatformIO fornecido em: [\[RVfpgaPath\]/RVfpga/Labs/Lab14/TwoConsecutiveLW\\_Instructions](#).

Duas leituras consecutivas:

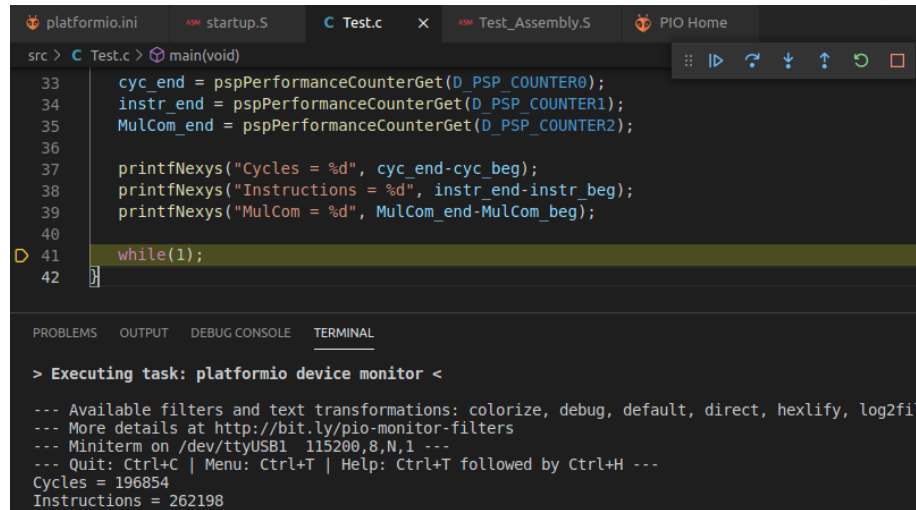
```
210: 0002a303    lw    t1,0(t0)
214: 0042a303    lw    t1,4(t0)
```

- Simulação:



Devido ao conflito estrutural no Pipe L/S, o segundo lw deve ficar parado por 1 ciclo, de forma semelhante ao Pipe Mult que lida com duas instruções mul consecutivas.

- Execução na placa:



```

src > C Test.c > main(void)
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, 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 ---
Cycles = 196854
Instructions = 262198

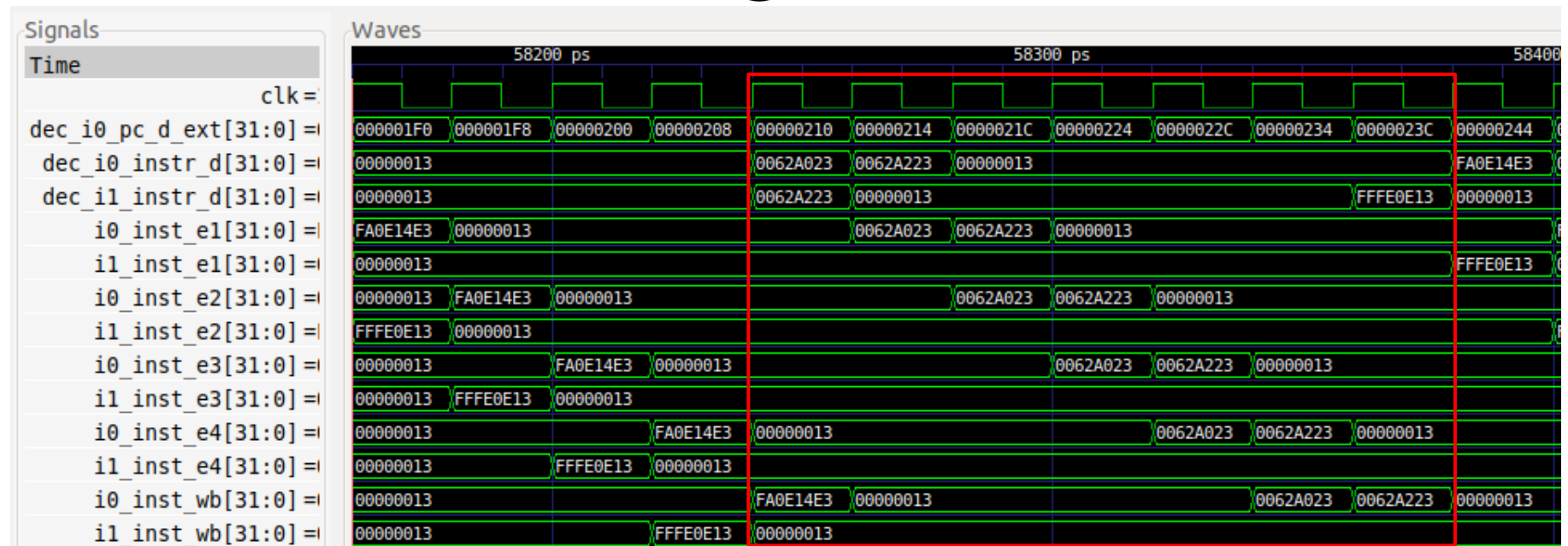
```

$$\text{IPC} = 262 / 196 = 1,33$$

Duas escritas consecutivas:

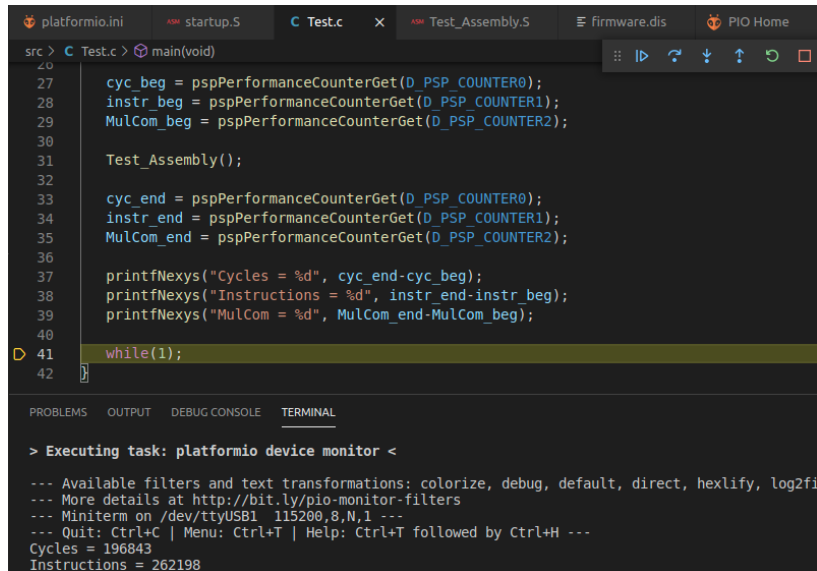
210:	0062a023	sw	t1,0 (t0)
214:	0062a223	sw	t1,4 (t0)

- Simulação:



Devido ao conflito estrutural no L/S Pipe, o segundo sw deve ser parar por 1 ciclo, da mesma forma que o Pipe Mult lida com duas instruções mul consecutivas.

- Execução na placa:



```

src > C Test.c > main(void)
27   cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
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 = 196843
Instructions = 262198

```

$$IPC = 262 / 196 = 1,33$$

2. (O exercício a seguir é baseado no exercício 4.22 do livro "Computer Organization and Design - RISC-V Edition", de Patterson & Hennessy ([HePa]).

Considere o fragmento do conjunto RISC-V abaixo:

```

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

```

Suponha que modifica o processador SweRV EH1 para que ele tenha apenas uma memória (que manipula instruções e dados). Nesse caso, haverá um conflito estrutural toda vez que um programa precisar carregar (Fetch) uma instrução durante o mesmo ciclo em que outra instrução acede a dados.

- Desenhe um diagrama de pipeline para mostrar onde o código acima irá parar nessa versão imaginária do processador

SweRV EH1.

- b. Em geral, é possível reduzir o número de paragens/nops resultantes desse conflito estrutural reordenando o código?
- c. Esse conflito estrutural deve ser tratado no hardware? Vimos que os conflitos de dados podem ser eliminados com a adição de nops ao código. Pode fazer o mesmo com esse conflito estrutural? Em caso afirmativo, explique como. Caso contrário, explique por que não.

Solução não fornecida.

## APÊNDICE A - DUAS INSTRUÇÕES `DIV` SIMULTÂNEAS NO ANDAR DECODE

**TAREFA:** Pode realizar um estudo semelhante para a instrução `div` como o realizado no Lab 12 para as instruções aritméticas-lógicas: veja o fluxo da instrução passando pelos andares do pipeline, analise os bits de controle (lembre-se do Apêndice D do Lab 11 que há um tipo de estrutura específica para a instrução `div` chamada `div_pkt_t`, e há um sinal definido no módulo `dec_decode_ctl` chamado `div_p`), etc.

Solução não fornecida.

**TAREFA:** Inspeção o código Verilog do `exu_div_ctl` para entender como a divisão é computada. Analise também o efeito dos sinais `div_stall`, `finish_early` e `finish`. Como exercício opcional, substitua a Divide Unit por sua própria unidade ou por uma da Internet.

Solução não fornecida.

**TAREFA:** Verificar se esse par de 32 bits (0x03de42b3 e 0x03ff4333) corresponde às instruções `div t0, t3, t4` e `div t1, t5, t6` na arquitetura RISC-V.

**0x03de42b3** → 0000001 11101 11100 100 00101 0110011

**funct7 = 0000001**

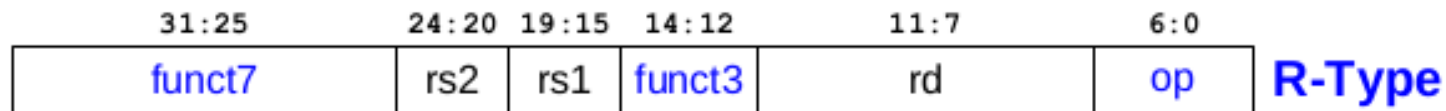
**rs2 = 11101 = x29 (t4)**

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

Do Apêndice B do DDCARV:



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

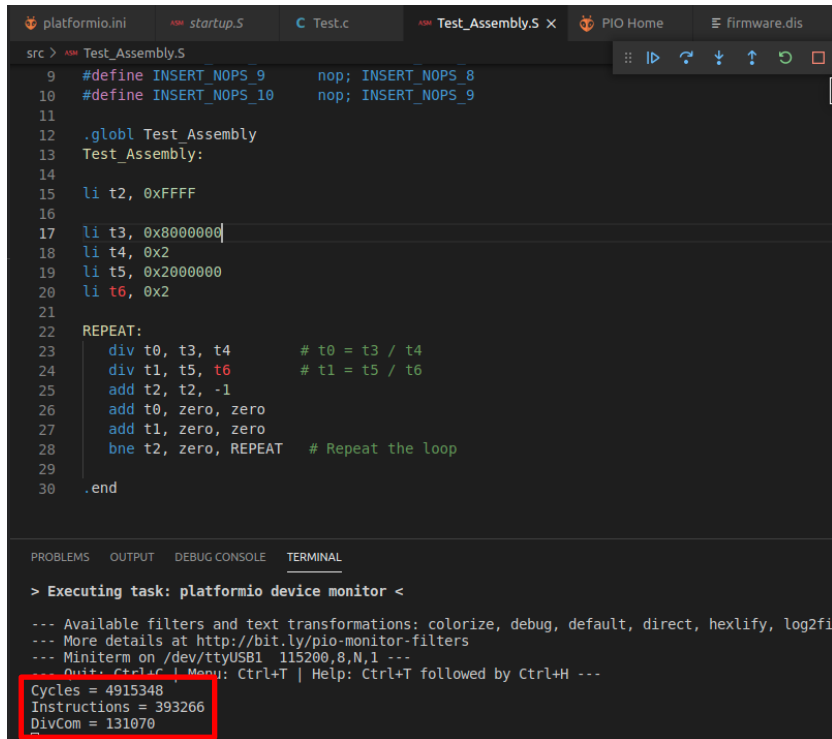
**TAREFA:** Replique a simulação da Figura 9 no seu computador e analise-a detalhadamente.

Solução fornecida no documento principal do Lab 14.

**TAREFA:** Compare a ilustração da Figura 10 e a simulação da Figura 9 que reproduziu no seu computador. Adicione sinais para ampliar a simulação e aprofundar a compreensão, conforme desejado.

Solução não fornecida.

**TAREFA:** Meça diferentes eventos (ciclos, instruções/divisões completadas etc.) usando os contadores de desempenho disponíveis no SweRV EH1, conforme explicado no Lab 11. O número de ciclos está de acordo com o esperado após a análise da simulação da Figura 9? Justifique sua resposta.



```

platformio.ini  startup.S  Test.c  Test_Assembly.S x  PIO Home  firmware.dis
src > Test_Assembly.S
9  #define INSERT_NOPS_9    nop; INSERT_NOPS_8
10 #define INSERT_NOPS_10  nop; INSERT_NOPS_9
11
12 .globl Test_Assembly
13 Test_Assembly:
14
15 li t2, 0xFFFF
16
17 li t3, 0x8000000
18 li t4, 0x2
19 li t5, 0x2000000
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

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 = 4915348
Instructions = 393266
DivCom = 131070

```

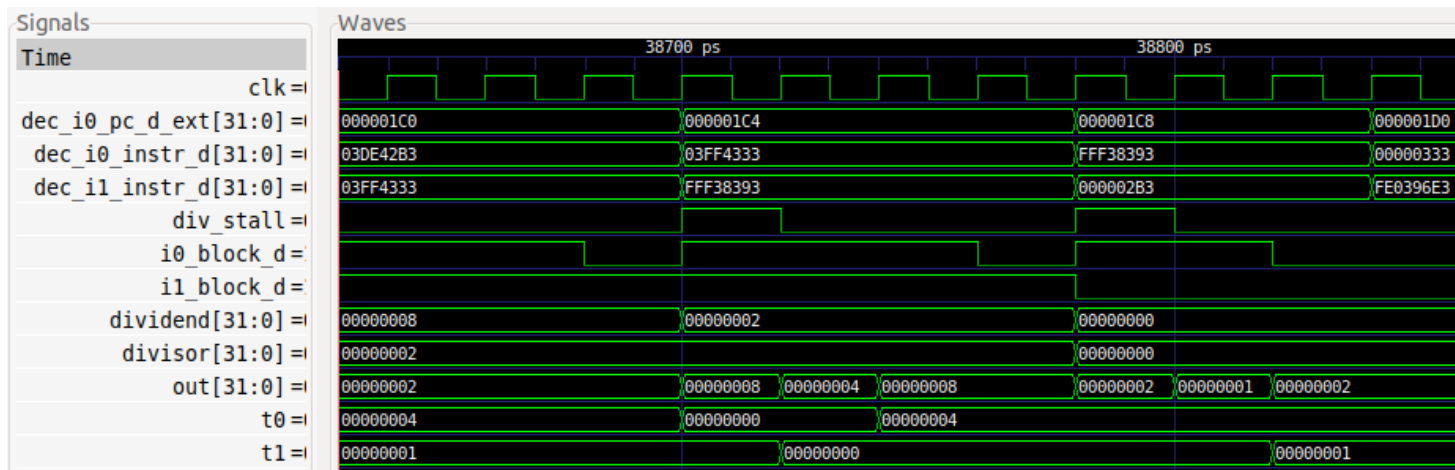
$CPI = 4910000 / 393000 = 12$ . Levando em conta que cada divisão leva cerca de 34 ciclos para ser executada e que as outras instruções levam  $\frac{1}{2}$  ciclo cada, isso é aproximadamente o que poderíamos esperar: um cálculo teórico aproximado poderia ser: 6 instruções executadas em  $34 + 34 + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2} + \frac{1}{2}$  ciclos  $\rightarrow CPI = 70 / 6 = 11$

**TAREFA:** Experimente dividendos e divisores diferentes e veja como o número de ciclos para computar o resultado depende do valor deles. Veja o experimento na simulação e com os HW Counters.

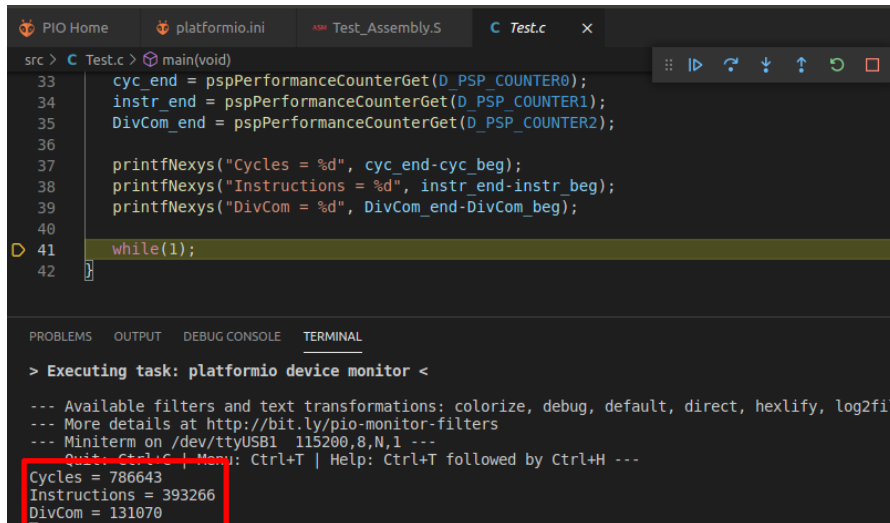
```

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

```



Agora, as divisões são computadas em apenas cerca de 5 ciclos.



```
src > C Test.c > main(void)
33   cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
34   instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
35   DivCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
36
37   printfNexys("Cycles = %d", cyc_end-cyc_beg);
38   printfNexys("Instructions = %d", instr_end-instr_beg);
39   printfNexys("DivCom = %d", DivCom_end-DivCom_beg);
40
41   while(1);
42
```

```
> 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 ---
Cycles = 786643
Instructions = 393266
DivCom = 131070
```

O CPI diminui muito (cerca de 2 por ciclo), já que o tempo para computar cada divisão também diminui muito.

**TAREFA:** A pasta `[RVfpgaPath]/RVfpga/Labs/Lab14/DIV_Instr_Accumul_C-Lang` fornece o projeto PlatformIO de um programa em C que acumula a subtração de duas divisões num ciclo.

- Analisar o programa C.
  - Execute uma simulação e inspecione uma iteração aleatória do ciclo. Observe que o programa em C é compilado sem otimizações.
  - Meça diferentes eventos (ciclos, instruções/divisões completadas, etc.) usando os contadores de desempenho disponíveis no SweRV EH1, conforme explicado no Lab 11.
- O número de ciclos está de acordo com o esperado após a análise da simulação da Figura 9? Justifique sua resposta.
- Crie um programa análogo em Assembly RISC-V e compare-o com a versão em C.
  - Desative a extensão **M** RISC-V no programa C e compare os resultados com o programa original. Para fazer isso, modifique a seguinte linha no ficheiro `platformio.ini` de:

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

Para:

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

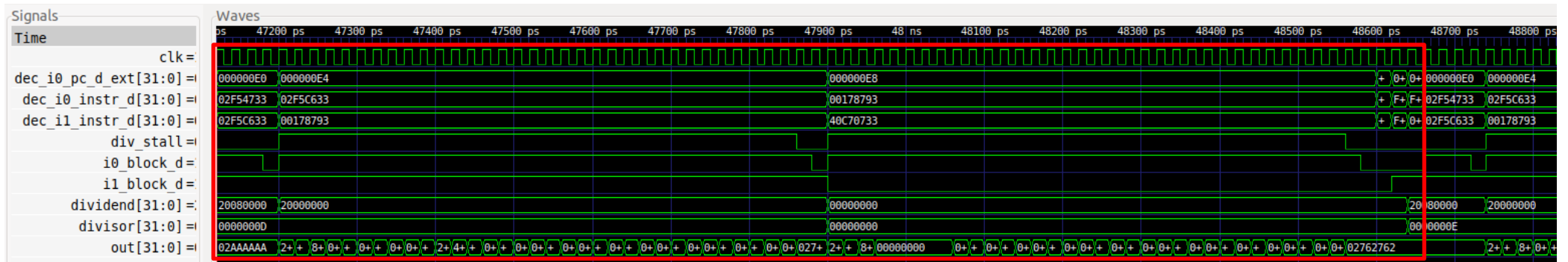
Isso evita o uso das instruções da extensão RISC-V M e as emula usando outras instruções.

- Programa C (original e desmontagem):

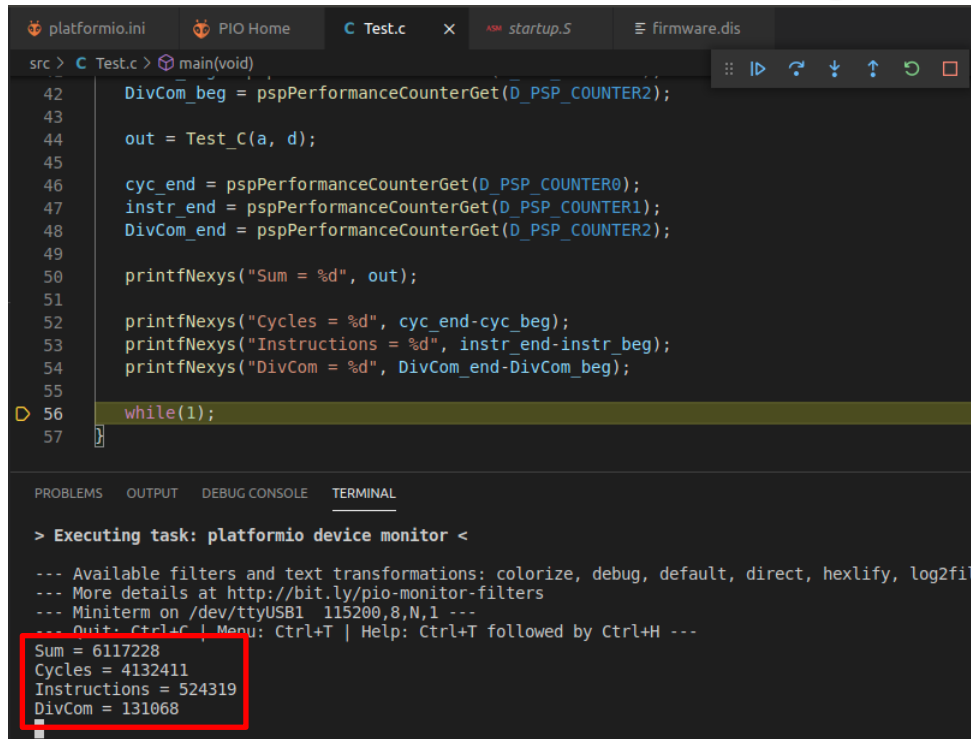
```
11 int Test_C(int a, int d)
12 {
13     int b, c, e=0, i=1;
14     do {
15         b = a/i;
16         c = d/i;
17         i = i+1;
18         e = e + (b-c);
19     } while(i<65535);
20     return(e);
21 }
```

```
66 000000d8 <Test_C>:
67 d8: 00100793      li a5,1
68 dc: 00000693      li a3,0
69 e0: 02f54733      div a4,a0,a5
70 e4: 02f5c633      div a2,a1,a5
71 e8: 00178793      addi a5,a5,1
72 ec: 40c70733      sub a4,a4,a2
73 f0: 00e686b3      add a3,a3,a4
74 f4: 00010737      lui a4,0x10
75 f8: ffe70713      addi a4,a4,-2 # fffe <_sp+0xc386>
76 fc: fef752e3      bge a4,a5,e0 <Test_C+0x8>
77 100: 00068513      mv a0,a3
78 104: 00008067      ret
```

- Simulação do programa C:



- Contadores HW:



The screenshot shows a code editor with a C program in `Test.c`. The program calculates the sum of a series of numbers and measures performance metrics using PSP performance counters. The code is as follows:

```

src > C Test.c > main(void)
42   DivCom_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
43
44   out = Test_C(a, d);
45
46   cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
47   instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
48   DivCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
49
50   printfNexys("Sum = %d", out);
51
52   printfNexys("Cycles = %d", cyc_end-cyc_beg);
53   printfNexys("Instructions = %d", instr_end-instr_beg);
54   printfNexys("DivCom = %d", DivCom_end-DivCom_beg);
55
56   while(1);
57

```

The terminal output shows the results of the program execution:

```

> 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 = 6117228
Cycles = 4132411
Instructions = 524319
DivCom = 131068

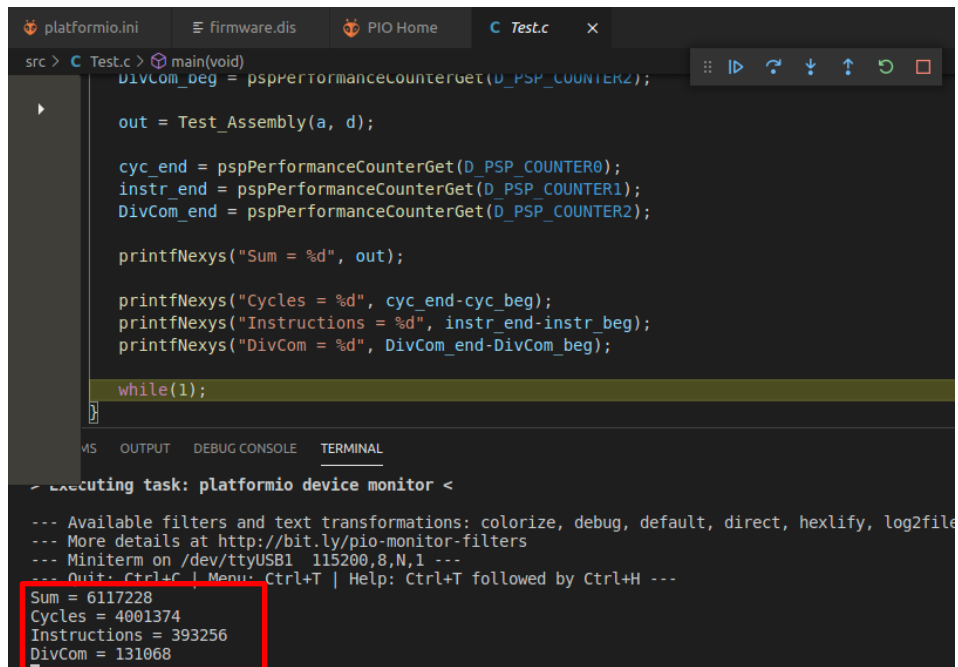
```

- O programa Assembly pode ser encontrado em:  
`[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab14/DIV_Instr_Accumul_Assembly`

```

128 000001c8 <Test_Assembly>:
129 1c8: 00100393      li t2,1
130 1cc: 00000e93      li t4,0
131 1d0: 00000f93      li t6,0
132 1d4: 00010637      lui a2,0x10
133 1d8: fff60613      addi a2,a2,-1 # ffff <_sp+0xc377>
134 1dc: 00050e33      add t3,a0,zero
135 1e0: 00058f33      add t5,a1,zero
136
137 000001e4 <REPEAT>:
138 1e4: 027e42b3      div t0,t3,t2
139 1e8: 027f4333      div t1,t5,t2
140 1ec: 00138393      addi t2,t2,1
141 1f0: 40628eb3      sub t4,t0,t1
142 1f4: 01df8fb3      add t6,t6,t4
143 1f8: fec396e3      bne t2,a2,1e4 <REPEAT>
144 1fc: 000f8533      add a0,t6,zero
145 200: 00008067      ret

```



```

src > C 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);
}

MS  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

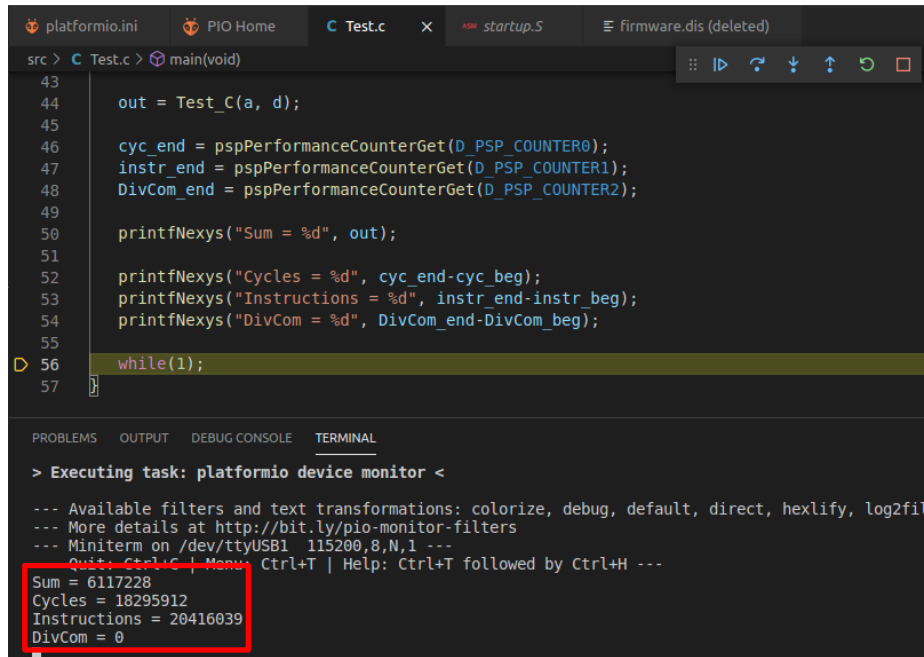
```

O resultado da soma é o mesmo, pois o programa é o mesmo.

O número de ciclos é um pouco menor, pois a versão Assembly programada manualmente é mais eficiente do que a obtida pelo compilador sem otimizações.

O número de instruções também é um pouco menor.

- Desativar a extensão M:



The screenshot shows a code editor with a C program in `Test.c`. The program calculates the sum of elements in an array and prints performance metrics. The terminal output shows the results of the execution.

```
src > C Test.c > main(void)
43
44     out = Test_C(a, d);
45
46     cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
47     instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
48     DivCom_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
49
50     printfNexys("Sum = %d", out);
51
52     printfNexys("Cycles = %d", cyc_end-cyc_beg);
53     printfNexys("Instructions = %d", instr_end-instr_beg);
54     printfNexys("DivCom = %d", DivCom_end-DivCom_beg);
55
56     while(1);
57 }
```

```
> 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 ---
Sum = 6117228
Cycles = 18295912
Instructions = 20416039
DivCom = 0
```

O resultado da soma é o mesmo, pois o programa é o mesmo.

O número de ciclos é muito maior: Cerca de 18 milhões contra cerca de 4 milhões.

O número de instruções também é muito maior: Cerca de 20 milhões contra cerca de 0,5 milhão.

O IPC está melhor agora.

Não há divisões completadas.

**TAREFA:** No SweRV EH1, as instruções `div` são bloqueantes. Modifique o processador para permitir instruções `div` não-bloqueantes.

Em seguida, adicione um segundo divisor ao processador SweRV EH1, de modo que duas instruções `div` do exemplo da Figura 8 possam ser executadas em paralelo.

Solução não fornecida.