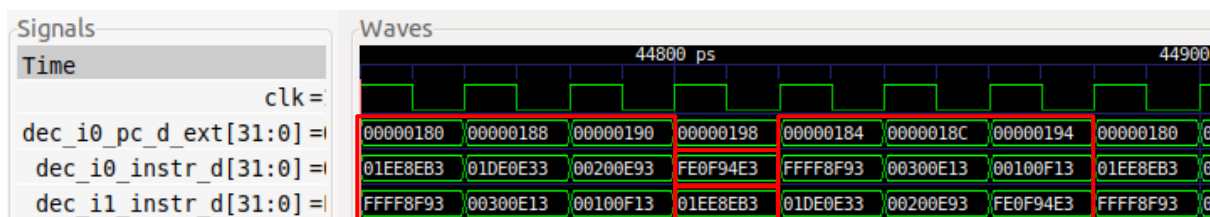


TAREFAS

TAREFA: Replicar a simulação da Figura 5 no seu computador. Pode usar o ficheiro `.tcl` fornecido em: `[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL/test_Basic.tcl`.

Solução fornecida no documento principal do Lab 15.

TAREFA: Remova todas as instruções `nop` do exemplo da Figura 2. Desenhe uma figura semelhante à Figura 3 para duas iterações consecutivas do ciclo, depois analise e confirme se a figura está correta comparando-a com uma simulação no Verilator e, por fim, calcule o IPC usando os Performance Counters durante a execução do programa na placa.

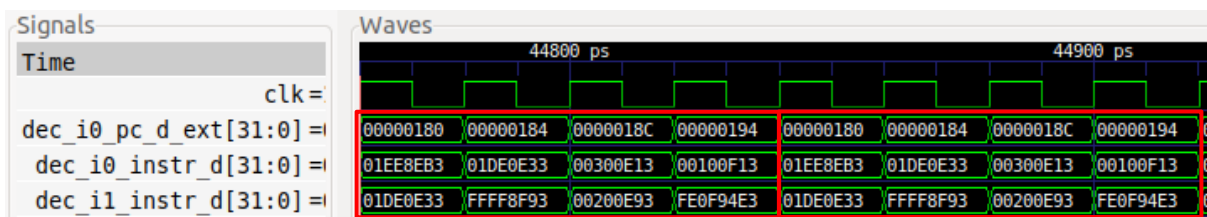


```
- add t4,t4,t5
  mul t3,t3,t4

- add t4,t4,t5
  div t3,t3,t4

- add t4,t4,t5
  lw t3, 0(t4)
```

- Duas instruções add:



Agora, cada iteração leva 4 ciclos para ser executada, pois a instrução de adição dependente deve ficar parada 1 ciclo, já que um de seus operandos de entrada não está disponível até que a primeira adição seja executada em EX1 e encaminhe o resultado.

```
PIO Home  C Test.c  asm Test_Assembly.S  asm startup.S

src > asm Test_Assembly.S

17 Test_Assembly:
18
19 li t3, 0x3
20 li t4, 0x2
21 li t5, 0x1
22 li t6, 0xFFFF
23
24 REPEAT:
25 add t4, t4, t5      # t4 = t4 + t5
26 add t3, t3, t4      # t3 = t3 + t4
27 add t6, t6, -1
28 li t3, 0x3
29 li t4, 0x2
30 li t5, 0x1
31 bne t6, zero, REPEAT # Repeat the loop
32
33 .end

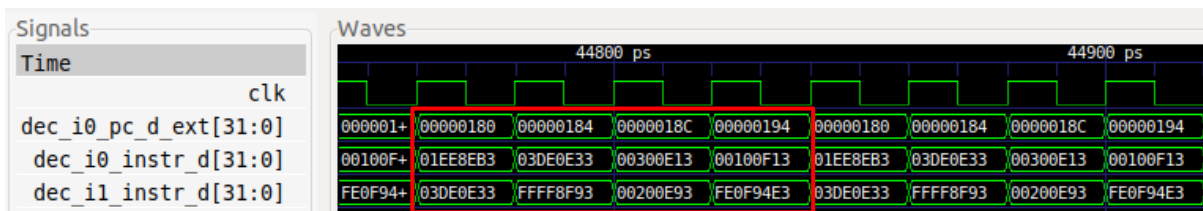
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

> 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 = 262392
Instructions = 458787
```

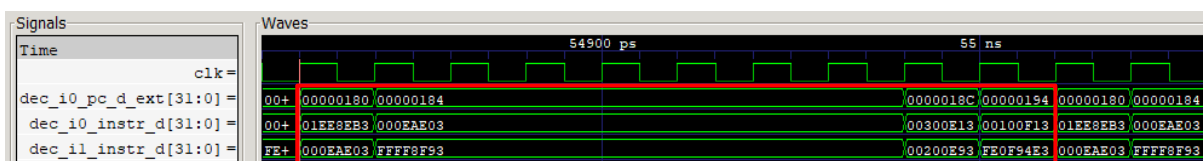
Agora o IPC não é o ideal: $IPC = 458 / 262 = 1,75$

- instrução add seguida de instrução mul:



Como antes, a instrução mul dependente deve ficar parada por 1 ciclo, já que um de seus operandos de entrada não está disponível até que a primeira adição seja executada em EX1 e encaminhe o resultado.

- instrução add seguida de instrução lw:



Como antes, a instrução lw dependente deve ficar parada por 1 ciclo, já que um de seus operandos de entrada não está disponível até que a primeira adição seja executada em EX1 e encaminhe o resultado.

TAREFA: Compare as equações anteriores com as explicadas para o processador com pipeline do DDCARV.

Equações para o processador com pipeline do DDCARV:

```
if      ((Rs1E == RdM) & RegWriteM) & (Rs1E != 0) then // Forward from Memory stage
    ForwardAE = 10
else if ((Rs1E == RdW) & RegWriteW) & (Rs1E != 0) then // Forward from Writeback stage
    ForwardAE = 01
else      ForwardAE = 00                                // No forwarding (use RF output)
```

TAREFA: Analise o código Verilog para explicar como o cálculo da equação anterior é realizado. Deve inspecionar as seguintes linhas do módulo **dec_decode_ctl**.

Solução não fornecida.

TAREFA: Escreva equações (semelhantes à anterior) para outros bits de controle de **i0_rs2bypass[9:0]**, **i0_rs1bypass[9:0]**, **i1_rs2bypass[9:0]** e **i1_rs1bypass[9:0]**.

Pode obter as equações das seguintes linhas no módulo **dec_decode_ctl**:

- 2372 - 2417

- 1721 - 1767
- 1497 - 1544
- 1130 - 1131 e 1255 - 1256

TAREFA: Replicar a simulação da Figura 8 no seu computador. Pode usar o ficheiro `.tcl` fornecido em: `[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL/test_Advanced.tcl`.

Solução fornecida no documento principal do Lab 15.

TAREFA: Para o programa da Figura 2, faça a mesma análise da Figura 8 para situações em que as duas instruções dependentes são colocadas a distâncias diferentes uma da outra. Pode controlar a distância alterando o número de nops entre as duas instruções `add` dependentes.

Além disso, crie outros exemplos em que o primeiro operando de entrada seja aquele que recebe os dados de encaminhamento.

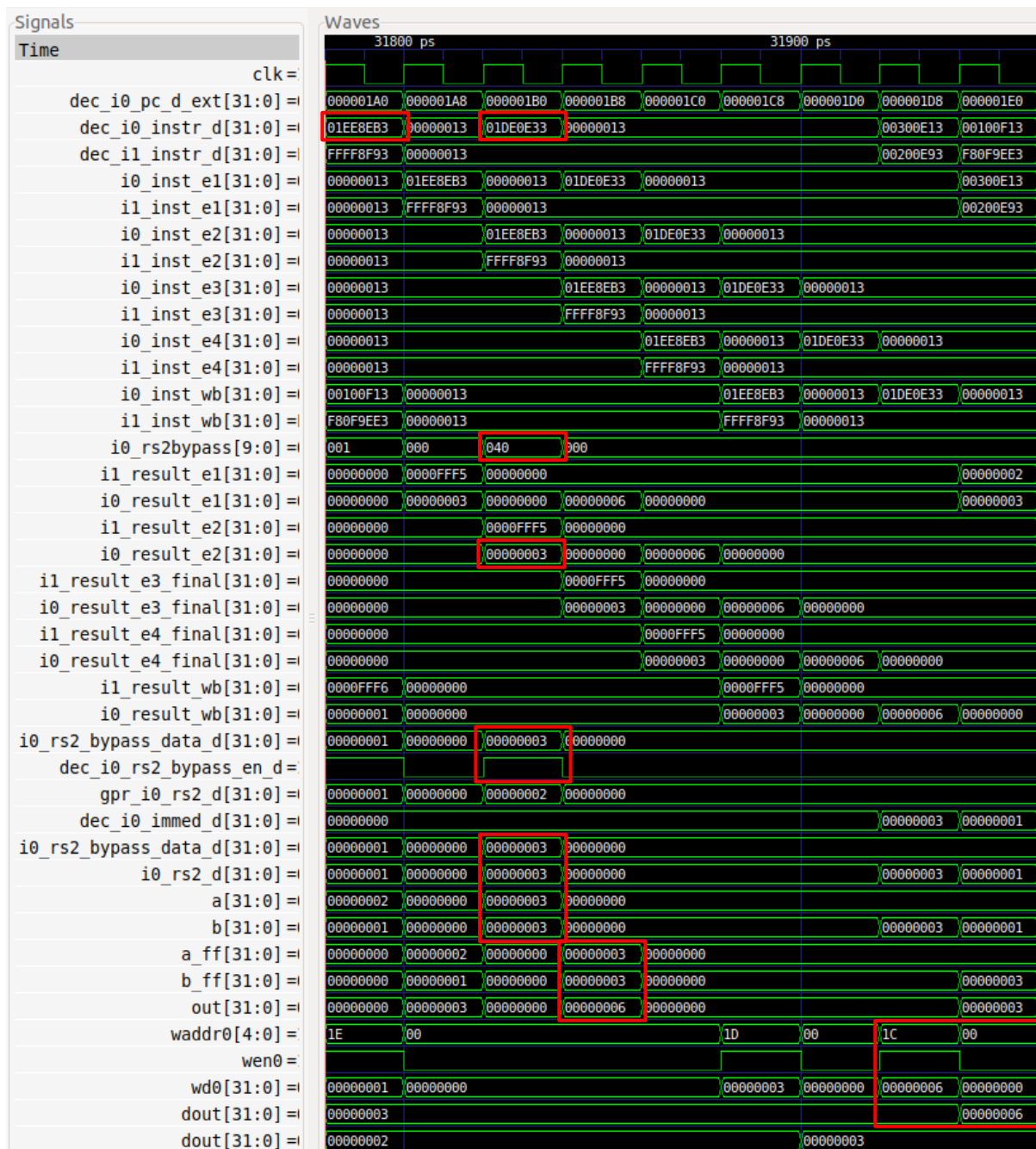
Também pode criar outros exemplos em que as duas instruções de adição estejam sendo executadas pelo Pipe I1 e confirmar que o comportamento é o mesmo.

Por fim, substitua a instrução `add` dependente (`add t3, t3, t4`) por outras instruções dependentes executadas por meio de outros pipes e analise os resultados da simulação. Por exemplo, em vez da segunda instrução `add`, poderia incluir uma das seguintes instruções:

- `lw t3, (t4)` (força o valor de leitura da DCCM, conforme explicado no Lab 13)
- `mul t3, t3, t4`
- `div t3, t3, t4`

Exemplo de novo programa simulado: Bypass do andar EX2 para o andar Decode:

| | |
|---------------|---|
| 1a0: 01ee8eb3 | add t4, t4, t5 |
| 1a4: ffff8f93 | <code>addi</code> <code>t6, t6, -1</code> |
| 1a8: 00000013 | <code>nop</code> |
| 1ac: 00000013 | <code>nop</code> |
| 1b0: 01de0e33 | add t3, t3, t4 |
| 1b4: 00000013 | <code>nop</code> |

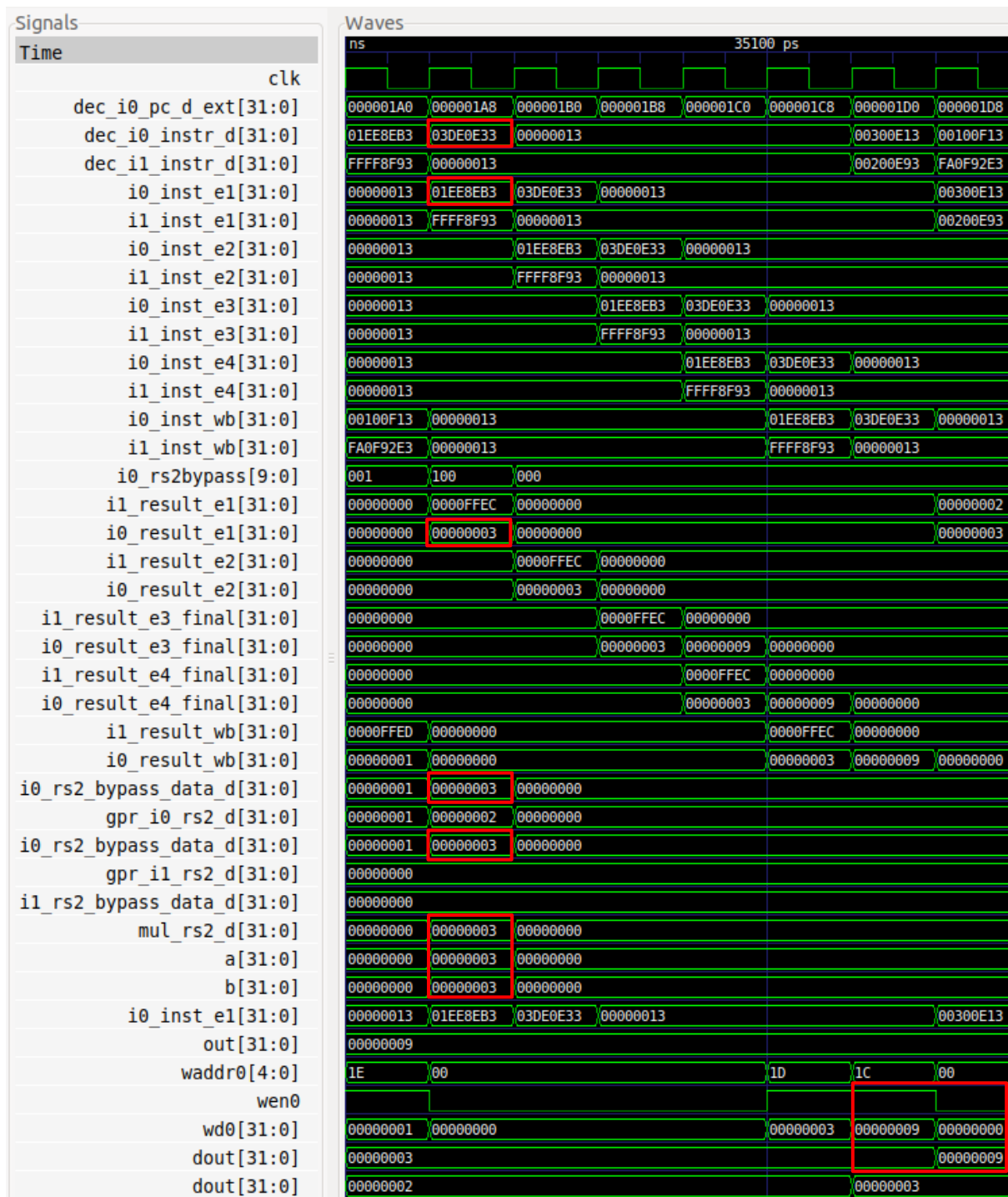


Exemplo de novo programa simulado: Execução de mul em vez do segundo add:

```

1a0: 01ee8eb3      add    t4,t4,t5
1a4: ffff8f93      addi   t6,t6,-1
1a8: 03de0e33      mul    t3,t3,t4

```



TAREFA: Adicione lógica à Figura 10 para produzir o primeiro operando de entrada (a) da ALU secundária no Pipe I0.

Solução não fornecida.

TAREFA: Replicar a simulação da Figura 12 no seu computador. Pode usar o ficheiro .tcl fornecido em: [RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL/scriptLoad.tcl

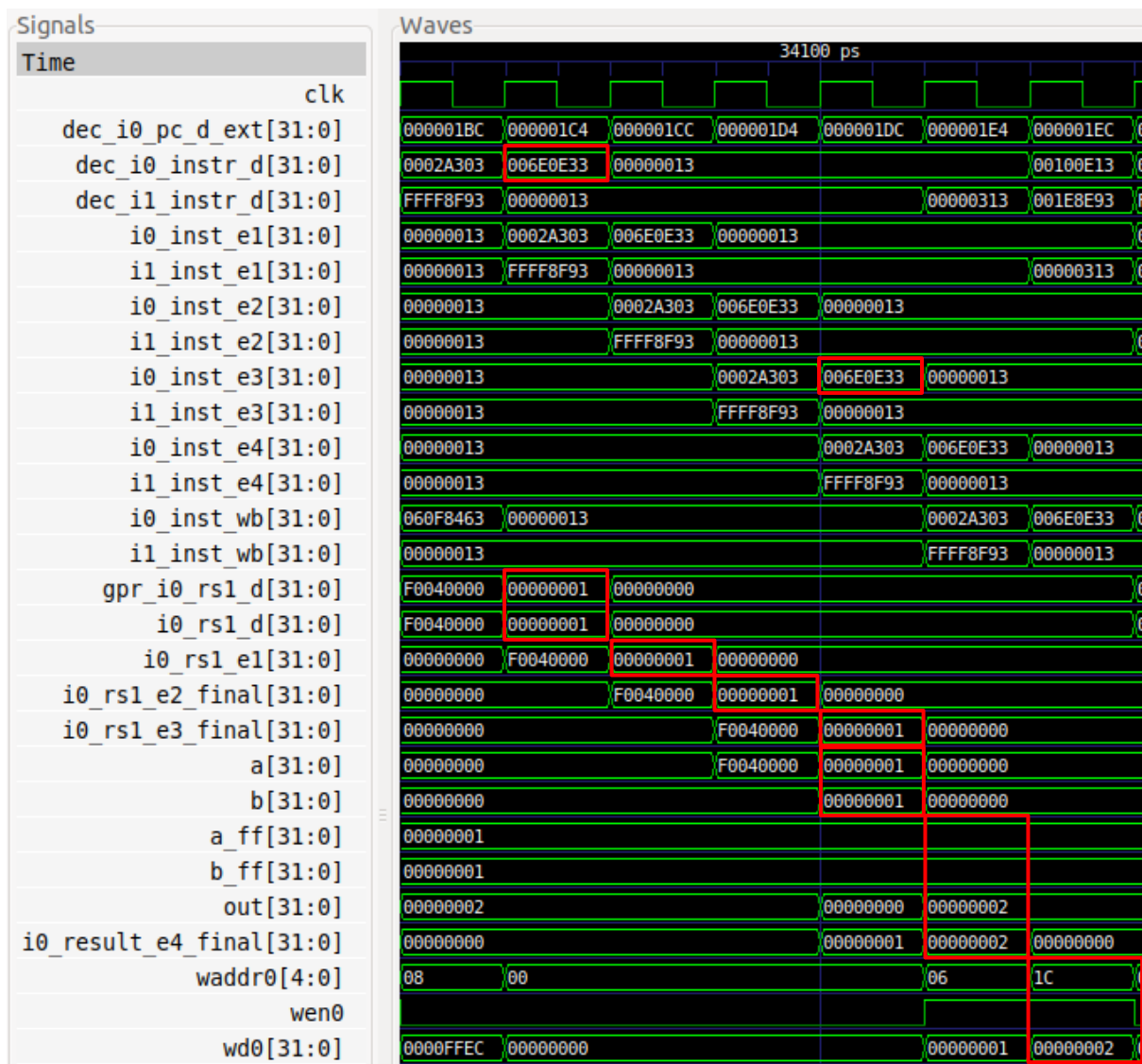
Solução fornecida no documento principal do Lab 15.

TAREFA: Desenhe uma figura semelhante à Figura 3 para o exemplo da Figura 11.

Solução não fornecida.

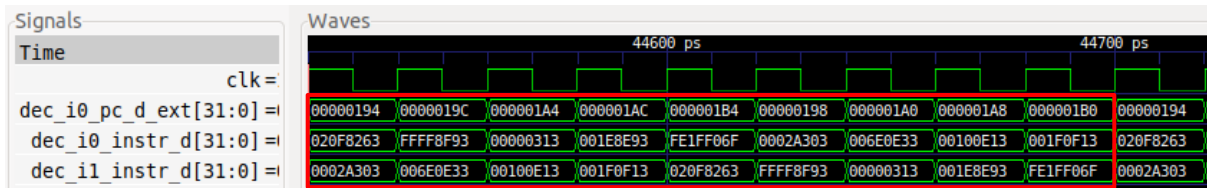
TAREFA: No exemplo anterior, analise como o primeiro operando da instrução `add t3, t3, t1` (`t3`) é obtido. Pode usar o ficheiro `.tcl` fornecido em:
`[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL/scriptLoad_FirstOperand.tcl`

O primeiro operando não depende de instruções anteriores, portanto, é obtido diretamente do Register File.



TAREFA: Remover as instruções `nop` no exemplo da Figura 11 e obter o IPC usando os

contadores HW.



2 iterações em 9 ciclos. Cada iteração contém 9 instruções. Portanto: $IPC = 18 / 9 = 2$

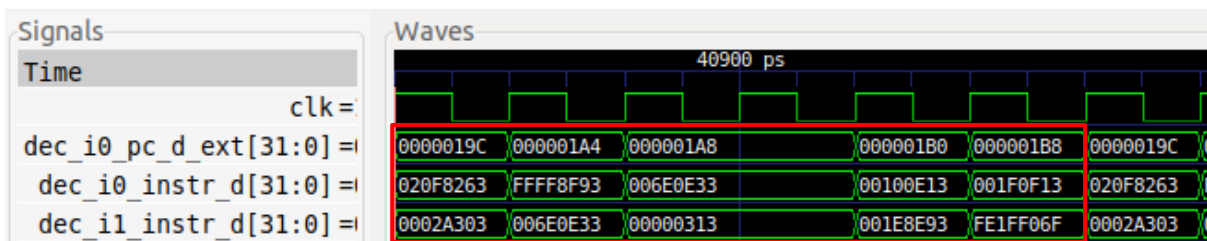
```

src > C Test.c x Test_Assembly.S platformio.ini startup.S
23 pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
24
25 cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
26 instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
27
28 Test_Assembly();
29
30 cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
31 instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
32
33 printfNexys("Cycles = %d", cyc_end-cyc_beg);
34 printfNexys("Instructions = %d", instr_end-instr_beg);
35
36 while(1);
37
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 = 295144
Instructions = 589863

```

Graças à lógica de forwarding e à ALU secundária, não há atrasos e o IPC é o ideal: $IPC = 5898 / 2951 = 1,998$

TAREFA: Desative a ALU secundária conforme explicado no Lab 11 e analise o exemplo da Figura 11 com uma simulação do Verilator e com uma execução na placa.



Após o `lw` (0x0002a303), a instrução `add` dependente (0x006e0e33) fica parada por alguns ciclos. 1 iteração leva 6 ciclos.

```

PIO Home  C Test.c  x  Test_Assembly.S  platformio.ini  Firmware.dis  startup.S
src > C Test.c > main(void)
29
30     cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
31     instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
32
33     printfNexys("Cycles = %d", cyc_end-cyc_beg);
34     printfNexys("Instructions = %d", instr_end-instr_beg);
35
36     while(1);
37
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 = 393469
Instructions = 589865

```

O IPC agora está longe do ideal, devido ao conflito criado pelo conflito de dados `lw-add`:
 $IPC = 5898 / 3934 = 1,499$

TAREFA: No exemplo da Figura 11, mova a instrução `add t6,t6,-1` após a instrução `add t3,t3,t1` e reexamine o programa na simulação e na placa.

Pode usar o programa fornecido em:

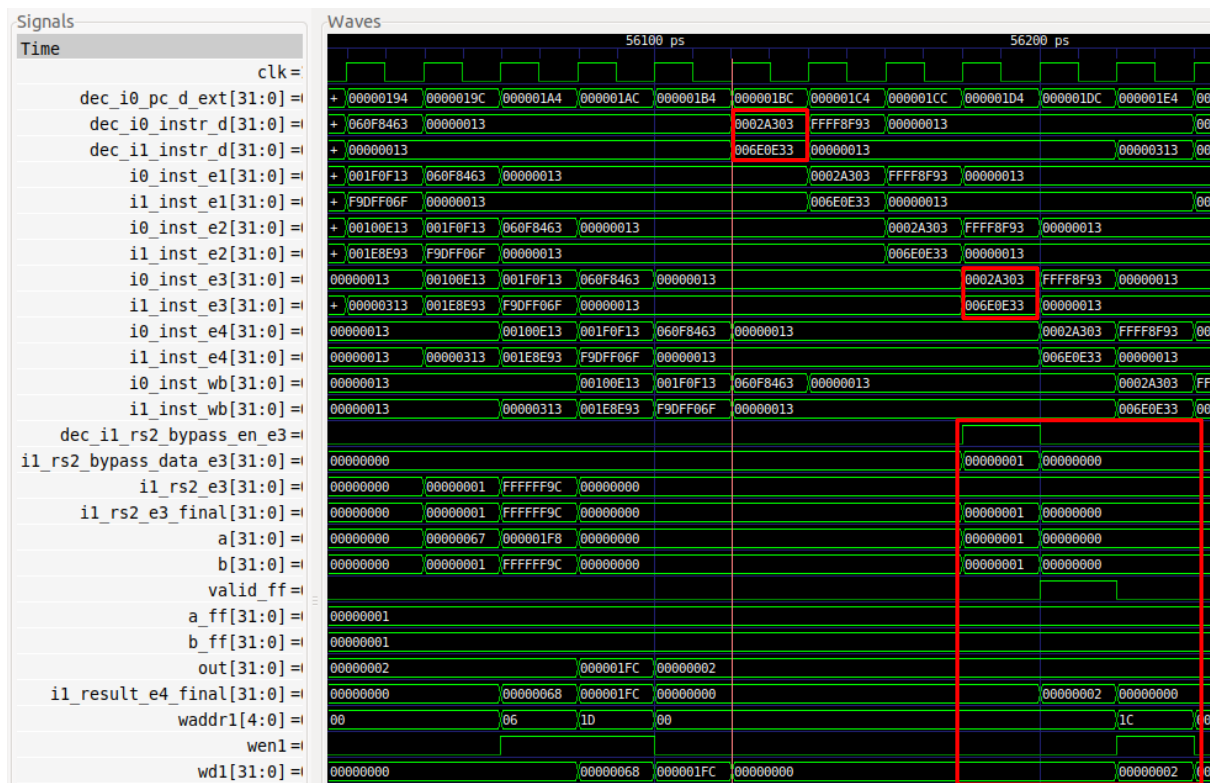
`[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab15/DataHazards_SameCycle-LW-AL`

A instrução `add t3,t3,t1` depende do load e ambas são executadas em paralelo pelas duas vias. Nesse caso, a adição é reexecutada na ALU secundária. Observe que a Via-1 pode receber o operando de entrada dos outros pipes, de modo que nenhum ciclo é perdido nessa situação.

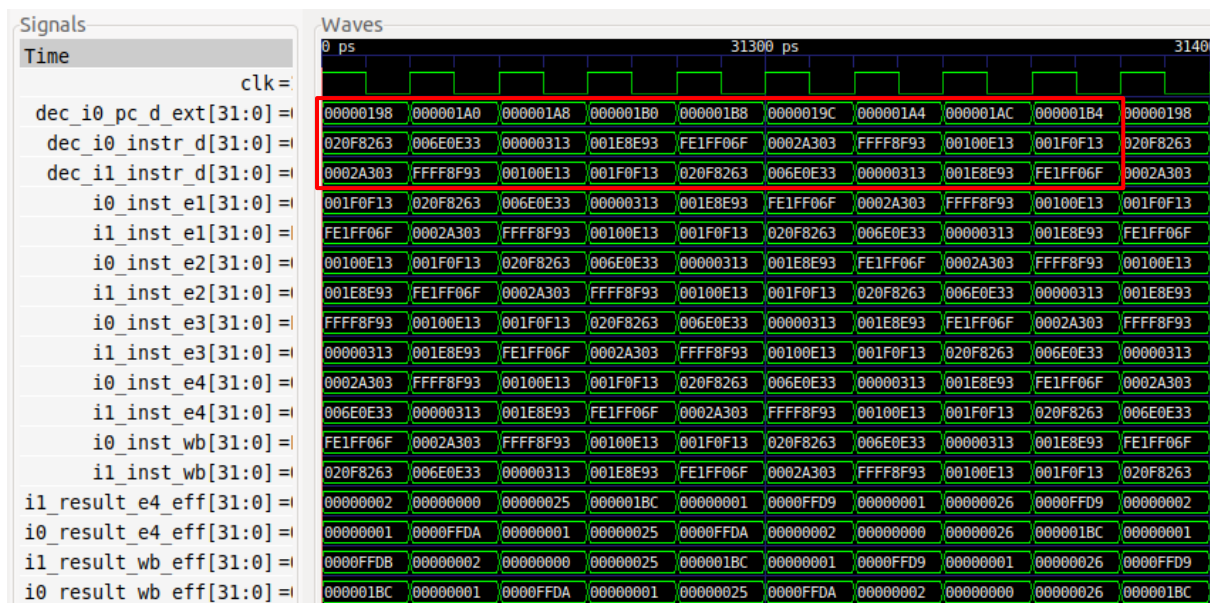
```

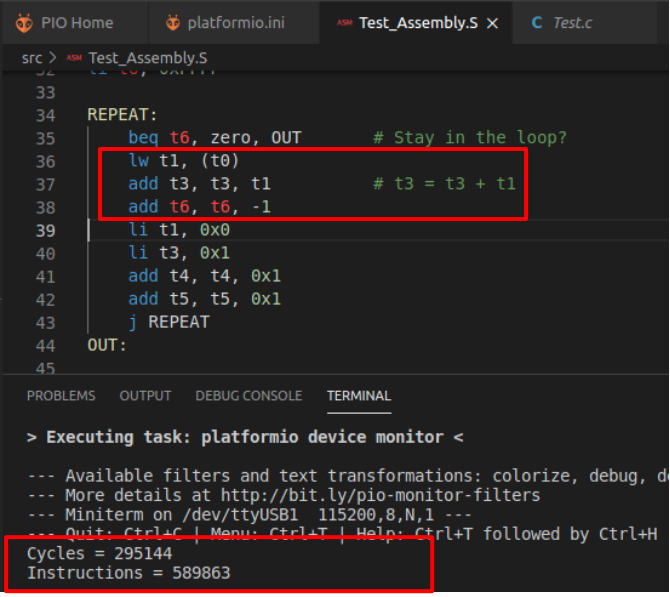
1688     assign i0_rs1_bypass_data_e3[31:0] = ({32{e3d.i0rs1bype3[3]}} & i1_result_e4_eff[31:0]) |
1689     ({32{e3d.i0rs1bype3[2]}} & i0_result_e4_eff[31:0]) |
1690     ({32{e3d.i0rs1bype3[1]}} & i1_result_wb_eff[31:0]) |
1691     ({32{e3d.i0rs1bype3[0]}} & i0_result_wb_eff[31:0]);
1692
1693     assign i0_rs2_bypass_data_e3[31:0] = ({32{e3d.i0rs2bype3[3]}} & i1_result_e4_eff[31:0]) |
1694     ({32{e3d.i0rs2bype3[2]}} & i0_result_e4_eff[31:0]) |
1695     ({32{e3d.i0rs2bype3[1]}} & i1_result_wb_eff[31:0]) |
1696     ({32{e3d.i0rs2bype3[0]}} & i0_result_wb_eff[31:0]);
1697
1698     assign i1_rs1_bypass_data_e3[31:0] = ({32{e3d.i1rs1bype3[6]}} & i0_result_e3[31:0]) |
1699     ({32{e3d.i1rs1bype3[5]}} & exu_mul_result_e3[31:0]) |
1700     ({32{e3d.i1rs1bype3[4]}} & lsu_result_dc3[31:0]) |
1701     ({32{e3d.i1rs1bype3[3]}} & i1_result_e4_eff[31:0]) |
1702     ({32{e3d.i1rs1bype3[2]}} & i0_result_e4_eff[31:0]) |
1703     ({32{e3d.i1rs1bype3[1]}} & i1_result_wb_eff[31:0]) |
1704     ({32{e3d.i1rs1bype3[0]}} & i0_result_wb_eff[31:0]);
1705
1706
1707     assign i1_rs2_bypass_data_e3[31:0] = ({32{e3d.i1rs2bype3[6]}} & i0_result_e3[31:0]) |
1708     ({32{e3d.i1rs2bype3[5]}} & exu_mul_result_e3[31:0]) |
1709     ({32{e3d.i1rs2bype3[4]}} & lsu_result_dc3[31:0]) |
1710     ({32{e3d.i1rs2bype3[3]}} & i1_result_e4_eff[31:0]) |
1711     ({32{e3d.i1rs2bype3[2]}} & i0_result_e4_eff[31:0]) |
1712     ({32{e3d.i1rs2bype3[1]}} & i1_result_wb_eff[31:0]) |
1713     ({32{e3d.i1rs2bype3[0]}} & i0_result_wb_eff[31:0]);
1714

```



Se removermos as instruções `nop`, realizarmos a simulação e executarmos na placa, obteremos:





```

src > Test_Assembly.S
32  li t0, 0x1
33
34  REPEAT:
35      beq t6, zero, OUT      # Stay in the loop?
36      lw t1, (t0)
37      add t3, t3, t1          # t3 = t3 + t1
38      add t6, t6, -1
39      li t1, 0x0
40      li t3, 0x1
41      add t4, t4, 0x1
42      add t5, t5, 0x1
43      j REPEAT
44  OUT:
45
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
> Executing task: platformio device monitor <
--- Available filters and text transformations: colorize, debug, d
--- 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 = 295144
Instructions = 589863

```

Graças à lógica de forwarding e à ALU secundária, não há atrasos e o IPC é o ideal: $IPC = 5898 / 2951 = 1,998$

1. EXERCÍCIOS

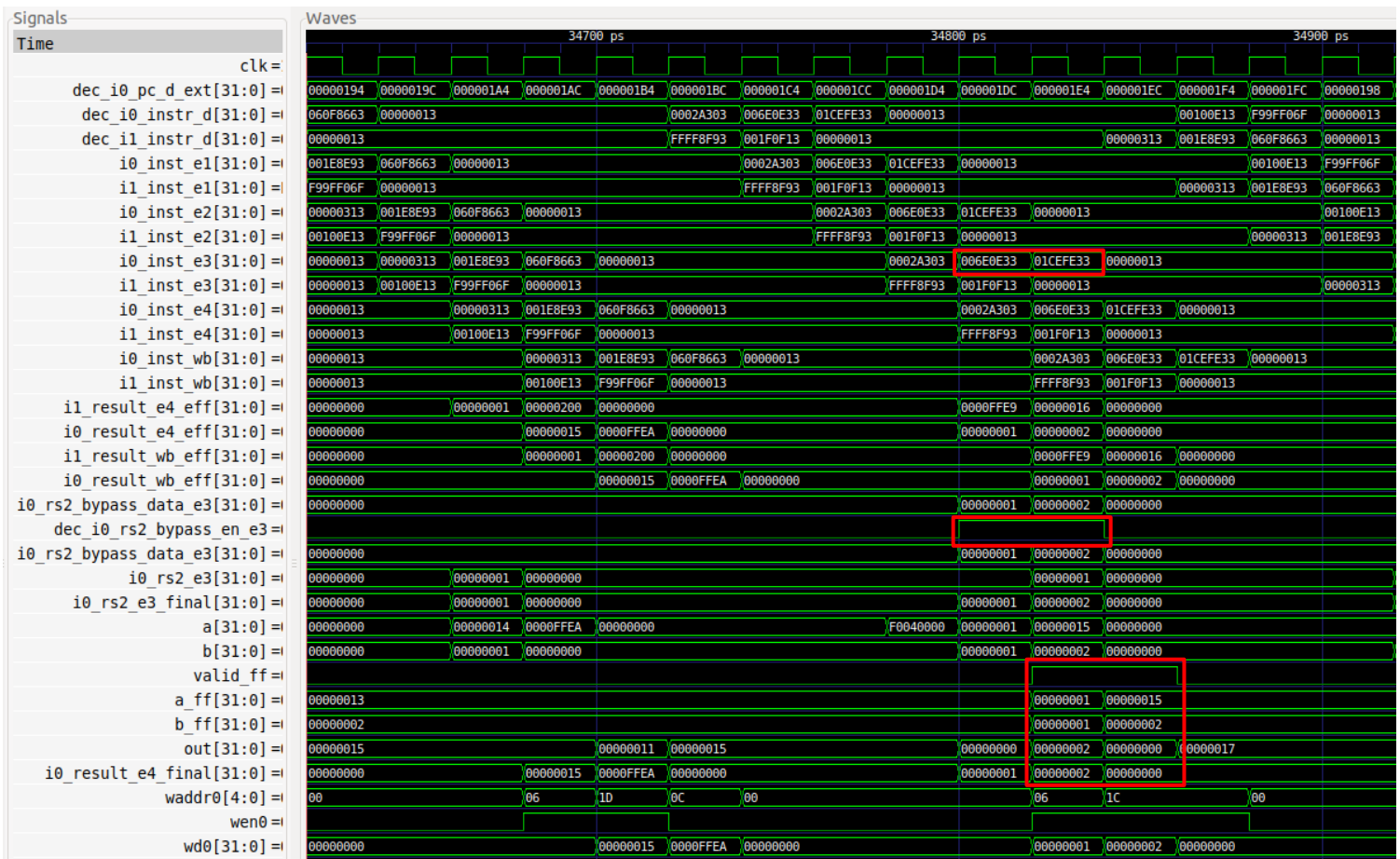
- 1) Modifique o programa usado na Seção 3 adicionando uma instrução aritmética-lógica extra que dependa do resultado da instrução `add`. Por exemplo, pode substituir o ciclo da Figura 11 pelo código a seguir, em que uma nova instrução `AND` foi incluída (**e** **t3, t4, t3**) e em que reordenamos ligeiramente o código movendo para frente a instrução `add t5, t5, 0x1`:

```

REPEAT:
    beq t6, zero, OUT
    INSERT_NOPS_9
    lw t1, (t0)
    add t6, t6, -1
    add t3, t3, t1
    add t5, t5, 0x1
    and t3, t4, t3
    INSERT_NOPS_8
    li t1, 0x0
    li t3, 0x1
    add t4, t4, 0x1
    j REPEAT
OUT:

```

Analise a simulação do Verilator e explique como os conflitos de dados são tratados na nova instrução A-L. Em seguida, remova todas as instruções `nop` e analise os resultados fornecidos pelos contadores de HW.



As instruções `add` e `and` dependentes usam a ALU secundária para recalcular o resultado. Observe que o segundo operando de entrada para a instrução `and` é bypassed em EX3.

```

src > C Test.c > main(void)
22  pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
23  pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
24
25  cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
26  instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
27
28  Test_Assembly();
29
30  cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
31  instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
32
33  printfNexys("Cycles = %d", cyc_end-cyc_beg);
34  printfNexys("Instructions = %d", instr_end-instr_beg);
35
36  while(1);
37

```

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 ---
Cycles = 327910
Instructions = 655398

```

$$IPC = 6553 / 3279 = 1,998$$

Instruções executadas por iteração: $655398 / 65535 = 10$
 Número de ciclos por iteração: $327910 / 65535 = 5$

Graças aos forwardings e à ALU secundária, o IPC ideal é alcançado nesse programa.

- 2) Analise a mesma situação descrita na Seção 2.C para uma instrução `mul` seguida de uma instrução `add` que usa o resultado da multiplicação. No programa da Figura 11, pode simplesmente substituir o `lw` por um `mul` que escreve no registo `t1`.

Solução não fornecida.

- 3) Analise uma situação com uma instrução `lw` seguida por uma instrução `mul` que depende do valor lido pelo load. No programa da Figura 11, pode simplesmente substituir a instrução `add` dependente por uma instrução `mul`.

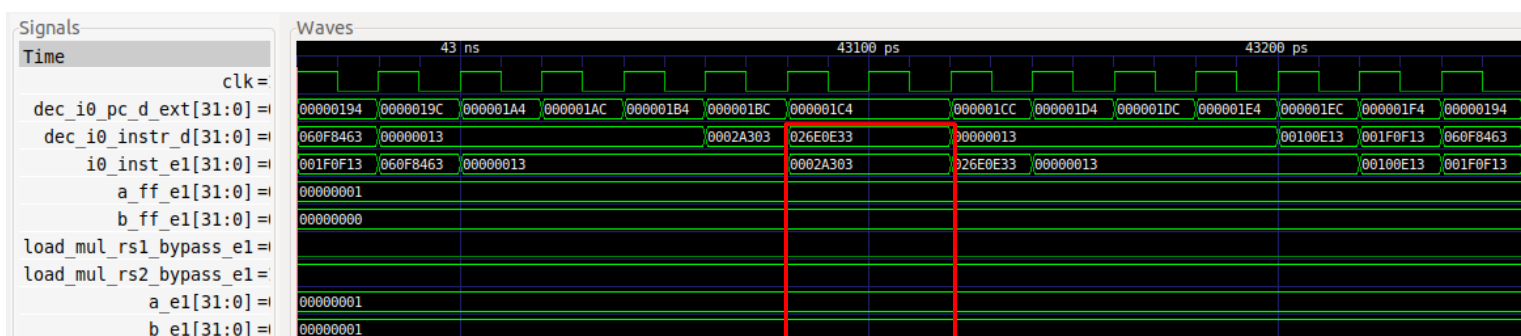
Pode usar o programa fornecido em:

`[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab15/DataHazards_Close-LW-MUL`

A instrução `mul` não pode ser executada pela ALU secundária. Um novo caminho de bypass é implementado dentro do multiplicador (módulo `exu_mul_ctl`) que encaminha o valor lido por um load para o andar M1.

```
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];
```

Dessa forma, apenas um ciclo é perdido devido à dependência RAW.



O segundo operando é bypassed do valor lido pelo load. Dessa forma, apenas um ciclo é perdido devido à dependência.

- 4) (O exercício a seguir é baseado nos exercícios 4.18, 4.19, 4.20 e 4.26 de [HePa]). Suponha que executava o código abaixo numa versão do processador SweRV EH1 que não lida com conflitos de dados (ou seja, o programador é responsável por lidar com os conflitos de dados inserindo instruções `nop` quando necessário). Adicione instruções `nop` ao código para que ele seja executado corretamente.
- ```
addi x11, x12, 5
```

```
add x13, x11, x12
addi x14, x11, 15
add x15, x13, x12
```

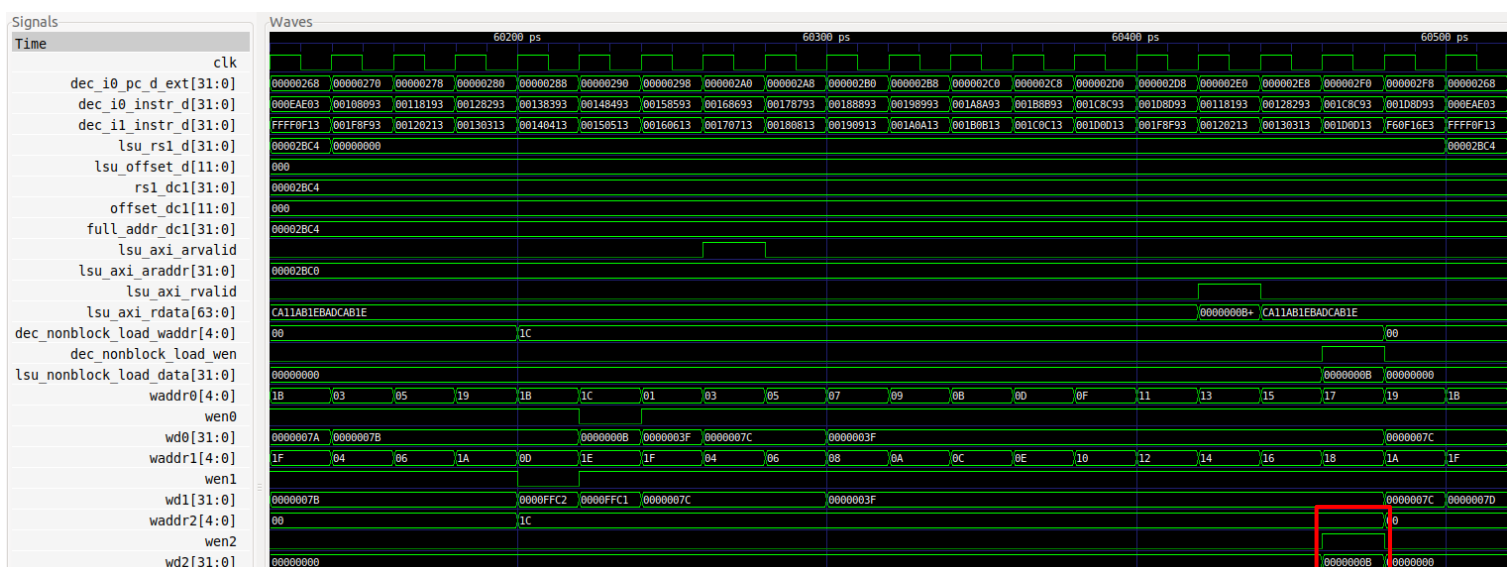
Em seguida, crie sequências de pelo menos três trechos de código Assembly que exibam diferentes tipos de conflitos de dados RAW. O tipo de dependência de dados RAW é identificado pelo andar que produz o resultado e a próxima instrução que consome o resultado.

Para cada sequência, quantos `nop` precisariam ser inseridos e onde, para permitir que seu código seja executado corretamente em um processador SweRV EH1 sem forwarding ou detecção de conflito? Qual é o CPI se usarmos o forwarding disponível no SweRV EH1 e não inserirmos `nops`?

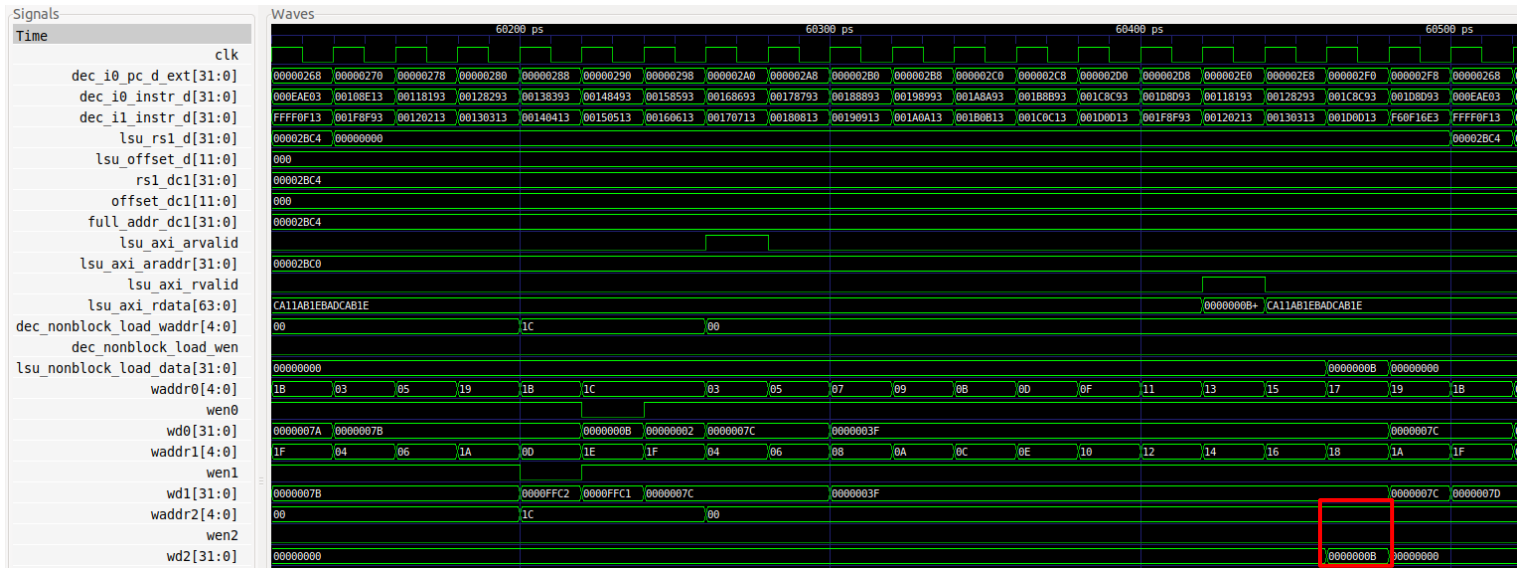
Solução não fornecida.

5) No programa da Seção 2.C do Lab 14 (disponível em [\[RVfpgaPath\]/RVfpga/Labs/Lab14/LW\\_Instruction\\_ExtMemory](#)), substitua a instrução `add x1, x1, 1` por `add x28, x1, 1`. Isso introduz um conflito WAW entre a instrução `add` modificada e a instrução de leitura não-bloqueante no início do ciclo (`lw x28, (x29)`). Analise na simulação como esse conflito é tratado no SweRV EH1, para o qual pode observar o valor do sinal `wen2` no Register File. Tente entender como esse sinal é calculado na Unidade de Controle (módulo `dec`).

A simulação para o programa original é a seguinte. Como analisamos no Lab 14, há um ciclo (destacado na figura) em que 3 escritas simultâneas são realizadas no Register File (2 instruções `add` e uma instrução `load` não-bloqueante).



No novo programa, mostrado abaixo, em que substituímos a instrução `add x1, x1, 1` pela instrução `add x28, x1, 1`, é detectado que uma instrução posterior na ordem do programa modifica o mesmo registo, portanto, a escrita do load é desativada (o sinal `wen2` nunca fica alto), resolvendo o conflito de dados WAW.



6) No programa da Seção 2.C do Lab 14 (disponível em [\[RVfpgaPath\]/RVfpga/Labs/Lab14/LW\\_Instruction\\_ExtMemory](#)), substitua a instrução `add x1, x1, 1` por `add x1, x28, 1`. Isso introduz um conflito RAW entre a instrução `add` modificada e a instrução de leitura não-bloqueante no início do ciclo (`lw x28, (x29)`). Analise numa simulação como esse conflito é tratado no SweRV EH1.



O conflito RAW é detectado, o pipeline é paralisado e o encaminhamento ocorre conforme explicado no Lab.

7) Finalmente, na Seção 2.C do programa do Lab 14 (disponível em [\[RVfpgaPath\]/RVfpga/Labs/Lab14/LW\\_Instruction\\_ExtMemory](#)), substitua a instrução

add x1, **x1**, 1 por add x1, **x28**, 1 e a instrução add **x7**, x7, 1 por add **x28**, x7, 1. Isso causa um conflito RAW e WAW. Analise numa simulação como esses dois conflitos são tratados no SweRV EH1.

Solução não fornecida.

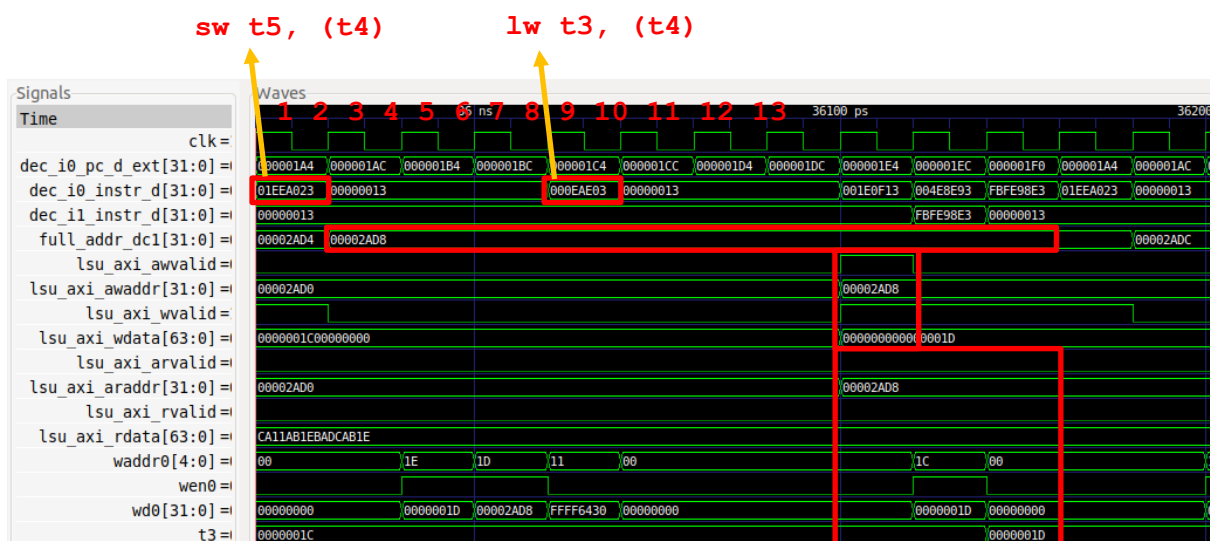
## 8) Forwarding de Escrita para Leitura

Essa é uma situação muito interessante que não analisámos neste Lab e que analisará neste exercício. Quando uma escrita seguida de uma leitura acede ao mesmo endereço, os dados podem ser encaminhados da escrita para a leitura dentro do núcleo e a leitura da memória externa DDR pode ser evitada, economizando tempo e energia.

A lógica que implementa esse encaminhamento está incluída na LSU e, especificamente, nos módulos `lsu_bus_intf` e `lsu_bus_buffer`, que deve inspecionar neste exercício.

O projeto PlatformIO de `[RVfpgaPath]/RVfpga/Labs/Lab15/Sw-Lw-Forwarding` ilustra um encaminhamento de carregamento de armazenamento. Um script `.tcl` é fornecido nessa pasta, que pode usar para analisar uma iteração aleatória do ciclo e entender como o forwarding é realizado.

## Simulação no Verilator:



Analisar a simulação:

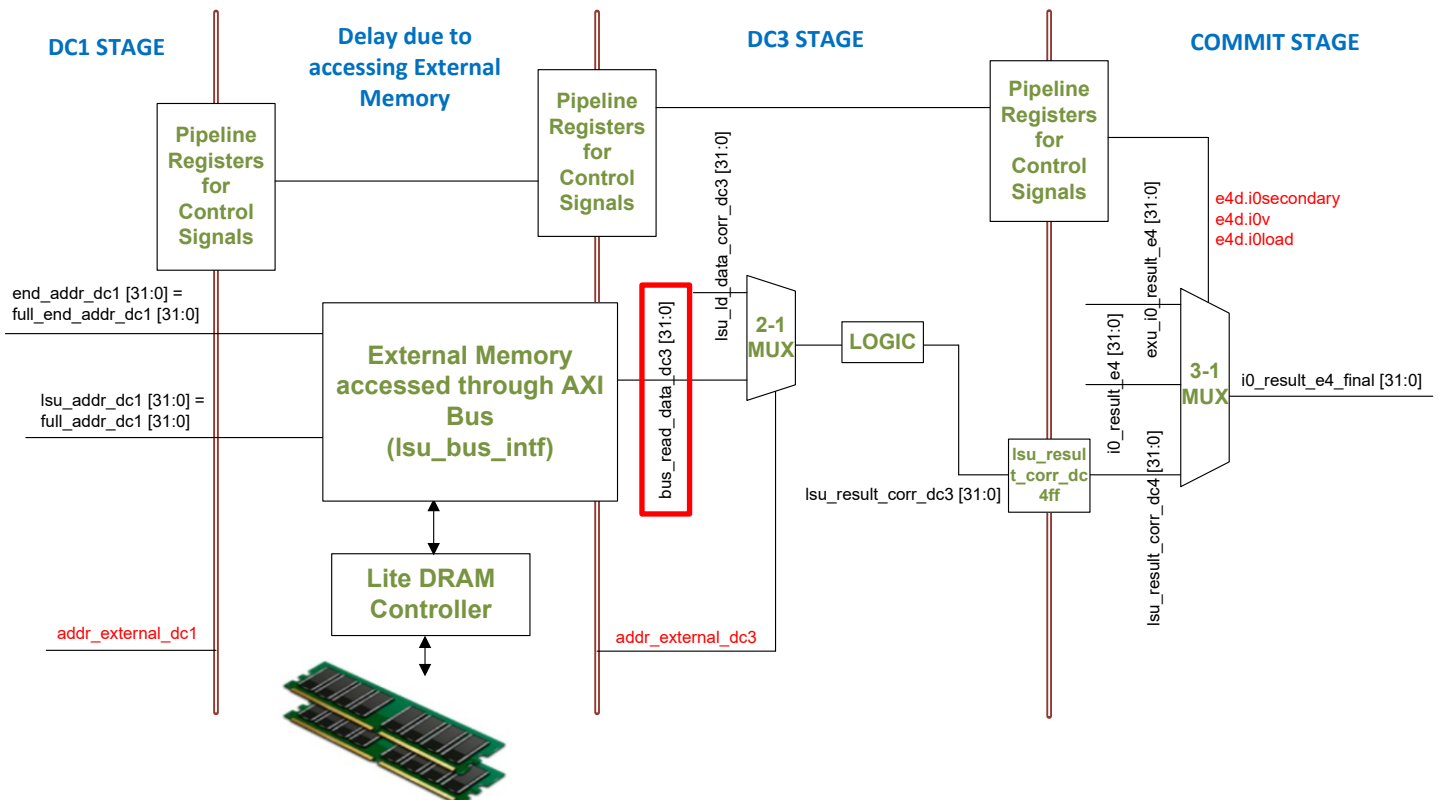
- **Ciclo 1:** A instrução `sw` é decodificada.
- **Ciclo 5:** A instrução `lw` é decodificada.
- **Ciclos 2 a 11:** O sinal `full_addr_dc1` = 0x00002AD8 durante toda a iteração. Isso acontece porque o endereço de escrita e o endereço de leitura são os mesmos.
- **Ciclo 9:** O store escreve na memória externa DDR por meio dos sinais de escrita do barramento AXI.

- o `lsu_axi_awvalid = 1`
  - o `lsu_axi_awaddr = 0x00002AD8`
  - o `lsu_axi_wvalid = 1`
  - o `lsu_axi_wdata = 0x000000000000001D`
- **Ciclos 9, 10 e 11:** A leitura recebe seus dados imediatamente por meio da lógica de bypass e escreve-os no Register File. A leitura nunca é enviada à memória DDR (veja os sinais de habilitação de leitura do barramento AXI: `lsu_axi_arvalid = lsu_axi_rvalid = 1`):
- o `waddr0 = 0x1C` (que é o registro `x28 = t3`)
  - o `wen0 = 1`
  - o `wd0 = 0x0000001D`
  - o `t3 = 0x0000001D`

## Como o forwarding é realizado dentro do núcleo?

Para a análise do encaminhamento de carga de armazenamento, é necessário inspecionar dois módulos: **lsu\_bus\_intf** e **lsu\_bus\_buffer**.

- 1) Na Seção 4 do Lab 13, analisamos um acesso de leitura à memória externa DDR. Ilustramos as estruturas SweRV EH1 envolvidas nesse acesso na Figura 16 do Lab 13:



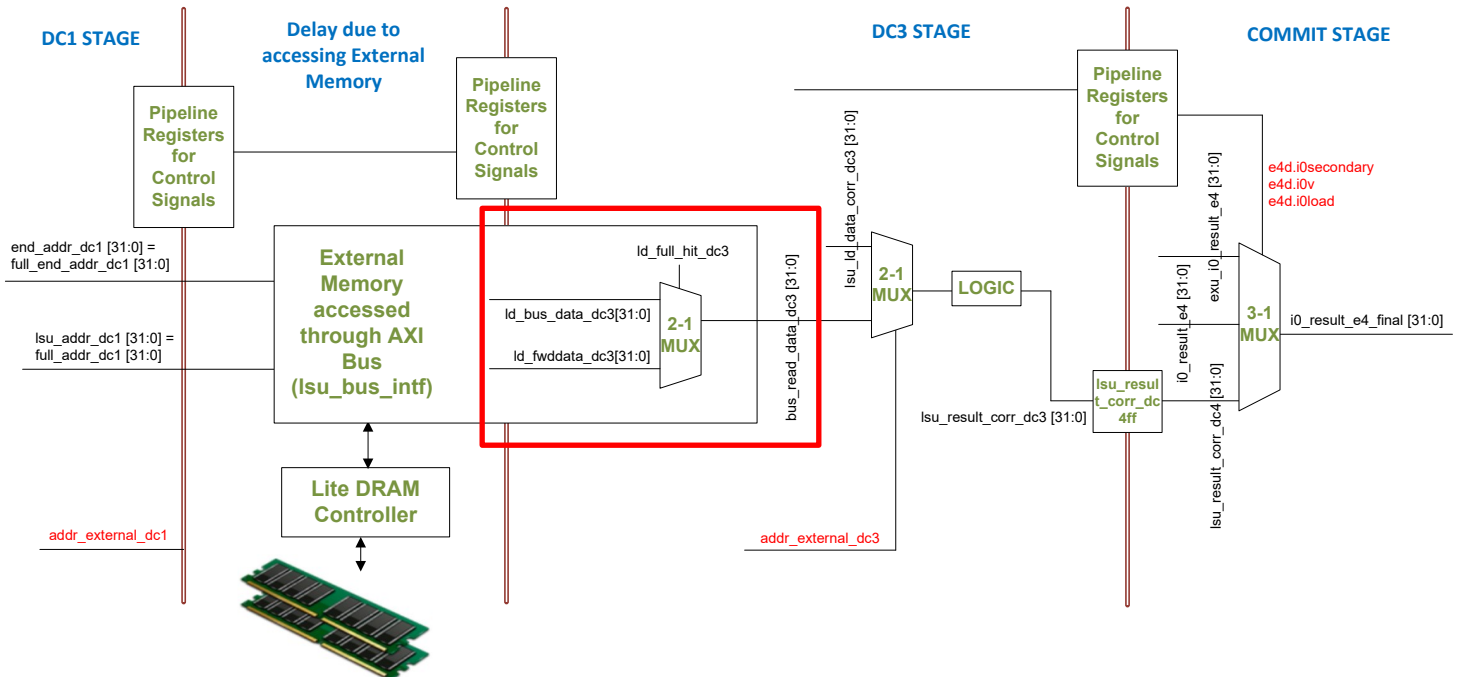
Os dados são fornecidos no sinal: `bus_read_data_dc3`.

- 2) O valor atribuído ao `bus_read_data_dc3` pode vir da memória DDR ou da lógica de

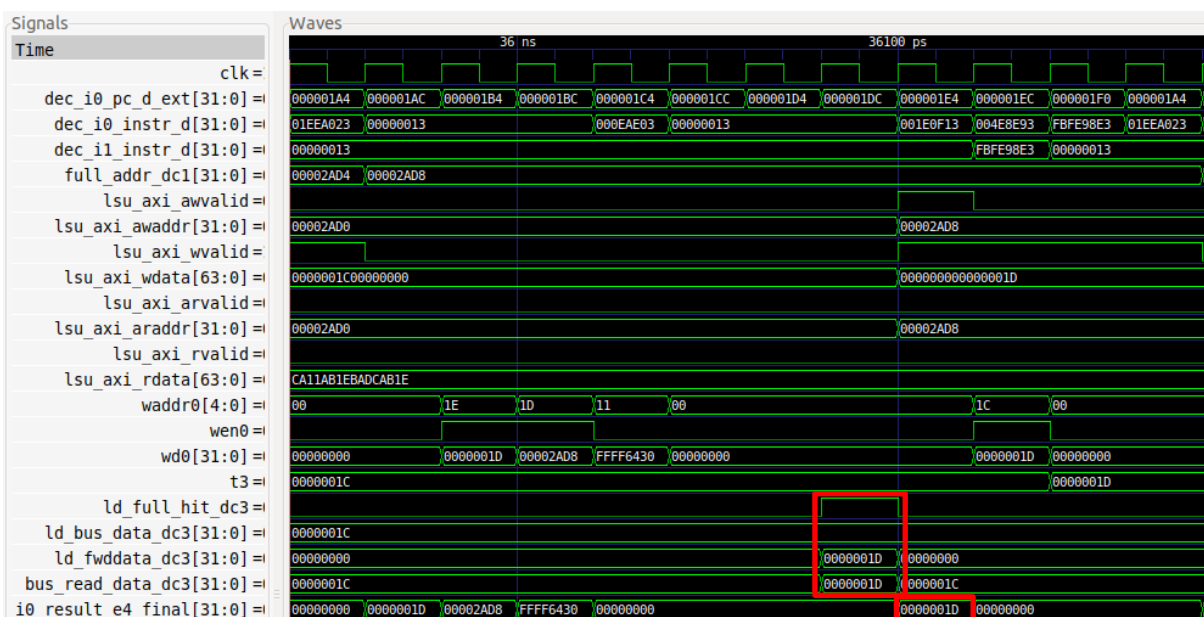
forwarding. Para esse fim, o módulo **lsu\_bus\_intf** inclui um multiplexer 2:1 que seleciona entre o valor lido da memória DDR (**ld\_bus\_data\_dc3**) e o valor do bypass (**ld\_fwddata\_dc3**).

```
387 assign bus_read_data_dc3[31:0] = ld_full_hit_dc3 ? ld_fwddata_dc3[31:0] : ld_bus_data_dc3[31:0];
```

Em seguida, incluímos esse multiplexer 2:1 na figura anterior:



- 3) Se adicionar as entradas/saída/sinal de controle desse multiplexer na simulação anterior, obterá:



Pode ver que os dados fornecidos para o carregamento são o bypass da escrita.

- 4) É possível analisar mais detalhadamente como o sinal de controle (`ld_full_hit_dc3`) e o valor bypassado (`ld_fwddata_dc3`) são computados nos módulos `lsu_bus_intf` e `lsu_bus_buffer`.

## APÊNDICE A

**TAREFA:** Replicar a simulação da Figura 15 no seu computador.

Solução fornecida no documento principal do Lab 15.

**TAREFA:** Compare como o cenário acima é tratado no SweRV EH1 e no processador com pipeline do DDCARV.

Solução não fornecida.

**TAREFA:** Se comparar cuidadosamente a Figura 16 e a Figura 6 do Lab 13, verá que o valor que a instrução `lw` lê para o Register File na Figura 6 do Lab 13 (sinal `lsu_ld_data_corr_dc3[31:0]`) é diferente do valor encaminhado pelo `lw` na Figura 16 (sinal `lsu_ld_data_dc3[31:0]`). A diferença entre os dois valores é que o primeiro foi verificado pela lógica ECC no módulo `lsu_ecc`, enquanto o segundo não foi. Explique por que não é problemático o fato de o valor encaminhado pelo `lw` não ser verificado quanto a erros.

Se for detectado um erro nos dados lidos pelo load, o pipeline será interrompido e descarregado. Assim, tanto a instrução de leitura quanto todas as instruções subsequentes, algumas das quais são as instruções que recebem o valor encaminhado incorreto, são descartadas e nunca entram em “commit”.

**TAREFA:** No exemplo da Figura 14, remova todas as instruções `nop` antes do `lw` e depois do `add`. Não remova os 5 `nop` entre as duas instruções dependentes. Analise a simulação e, em seguida, calcule o IPC com os Performance Counters executando o programa na placa (pode parecer estranho manter as instruções `nop` ao medir o IPC, pois são instruções inúteis; no entanto, o programa em si é inútil e nosso único objetivo aqui é analisar os conflitos de dados e entendê-los).

Solução não fornecida.