



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 15

Conflitos dos dados

1. INTRODUÇÃO

Neste laboratório, tratamos dos **conflitos de dados**. Tal como explicado por Hennessy e Patterson na sua edição 6th de "Computer Architecture : A Quantitative Approach" [HePa], os conflitos de dados ocorrem quando o pipeline altera a ordem dos acessos de leitura/escrita aos operandos, de modo a que a ordem seja diferente da ordem observada pela execução sequencial de instruções num processador sem pipeline. Suponha que a instrução i é seguida pela instrução j no programa e que ambas as instruções utilizam o registo x . Podem ocorrer três tipos de Conflitos de dados entre i e j :

- **Conflito de dados lidos após a escrita (RAW – Read After Write):** Este é o tipo mais comum de conflito. Ocorre quando a instrução j lê o registo x antes da instrução i escrever o registo x . Assim, a instrução j utilizaria o valor errado de x .
- **Conflito de dados de escrita após leitura (WAR – Write After Read):** Os conflitos WAR ocorrem quando a instrução j escreve x e a instrução i lê x , e a instrução j é reordenada para ocorrer antes de i . Assim, a instrução i lê o valor incorreto de x . Este conflito só ocorre quando as instruções são reordenadas, o que só raramente acontece no SweRV EH1; especificamente, os conflitos WAR nunca acontecem no SweRV EH1.
- **Conflito de dados escrita após escrita (WAW - Write After Write):** Os conflitos WAW ocorrem quando as instruções são reordenadas e a instrução j escreve x antes de a instrução i escrever x . Este conflito só ocorre quando as instruções são reordenadas, o que raramente acontece no SweRV EH1; no entanto, no caso de leituras não-bloqueantes, pode ocorrer um conflito WAW, como analisaremos mais à frente neste laboratório.

Nas secções seguintes, analisamos a forma como os conflitos de dados RAW são resolvidos no processador SweRV EH1 e, em seguida, descrevemos tarefas e exercícios relacionados com os conflitos RAW. Também descrevemos um exercício que analisa uma situação em que ocorre um conflito WAW.

NOTA: Antes de analisar a lógica do conflito de dados do SweRV EH1, recomendamos a leitura da Secção 7.5 do DDCARV sobre a forma como os conflitos são resolvidos no processador em pipeline. Os conflitos de dados, especificamente, são analisados na Secção 7.5.3. Embora o processador em pipeline apresentado no livro seja mais simples do que o SweRV EH1, os conflitos de dados são resolvidos de forma semelhante em ambos os processadores.

2. RESOLUÇÃO DE CONFLITOS DE DADOS COM REENCAMINHAMENTO NO ANDAR DE DECODE

Tal como explicado na secção 7.5.3 do DDCARV, alguns conflitos dos dados RAW podem ser resolvidos através do reencaminhamento (também designado por *Forwarding* ou *Bypass*) de um resultado de uma instrução executada numa andar avançada do pipeline para uma instrução dependente executada num andar anterior do pipeline. Isto requer a adição de multiplexers à frente das unidades funcionais (ALUs, multiplicador, somador que calcula o endereço efetivo em DC1, etc.) para selecionar os seus operandos a partir do Register File ou de andares subsequentes.

A Figura 1 estende o andar de Decode mostrado na Figura 4 do Lab 11 com os valores de bypass. A lógica de Forwarding produz o Bypass (ou seja, encaminha) para cada um dos dois operandos de origem em cada uma das Vias:

- **Via-0:**
 - o Primeiro operando de entrada: `i0_rs1_bypass_data_d[31:0]`
 - o Segundo operando de entrada: `i0_rs2_bypass_data_d[31:0]`
- **Via-1:**
 - o Primeiro operando de entrada: `i1_rs1_bypass_data_d[31:0]`
 - o Segundo operando de entrada: `i1_rs2_bypass_data_d[31:0]`

Estas quatro entradas são distribuídas aos multiplexers 3:1 e 4:1 que determinam os operandos de entrada para cada um dos andares de execução dos caminhos do pipeline. Para efeitos de clareza na Figura 1 os sinais estão ligados por nomes. As entradas para a lógica de Forwarding são os resultados produzidos por instruções de programa anteriores que estão mais avançadas no pipeline, como veremos a seguir.

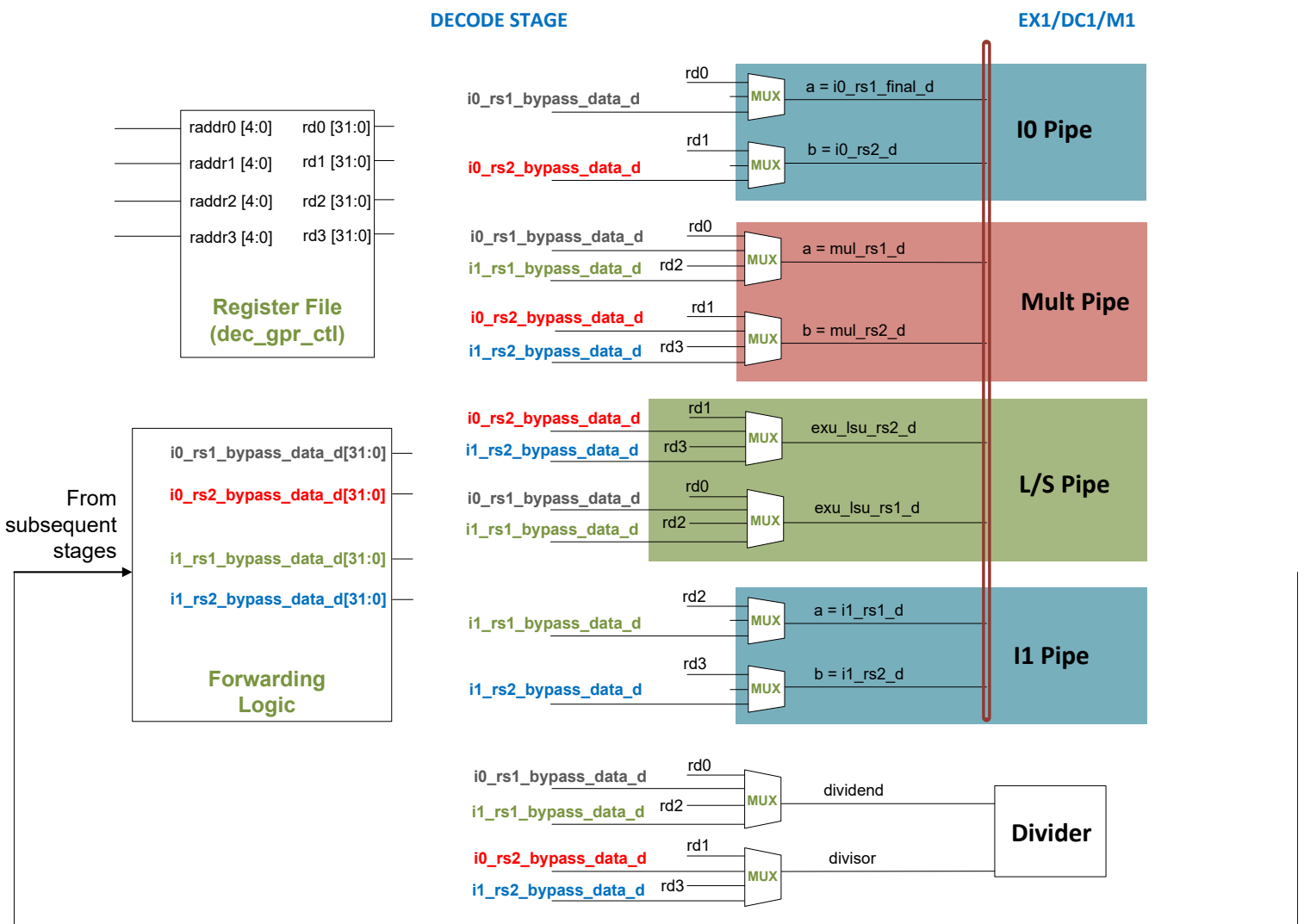


Figura 1. Entradas de bypass para as unidades funcionais.

Existem muitos caminhos de Forwarding no processador SweRV EH1 - nesta secção, concentramo-nos num caminho específico e analisamo-lo em pormenor. Em seguida, nas tarefas e exercícios, serão inspecionados outros casos. Analisamos a situação de duas instruções A-L dependentes executadas em simultâneo e a forma como os conflitos dos dados RAW são resolvidos. Tal como fizemos nos Labs 12 e 13, começamos com um estudo básico (Secção 2.A) e depois passamos a uma análise avançada (Secção 2.B). Pode optar por completar apenas a secção básica ou completar ambas as secções.

Vamos trabalhar com o exemplo apresentado na Figura 2 que executa duas instruções `add` contidas num ciclo que se repete durante `0xFFFF` iterações. A primeira instrução `add` escreve um valor em `t4` e a segunda instrução `add` usa `t4` como segundo operando de entrada. Uma instrução `add` independente (`add t6, t6, -1`), que é a instrução que atualiza o índice do ciclo) é inserida entre as duas instruções `add` para forçar as instruções `add` dependentes a utilizar o mesmo modo do processador.

```
.globl Test_Assembly

.text
Test_Assembly:

li t3, 0x3
li t4, 0x2
li t5, 0x1
li t6, 0xFFFF

REPEAT:
    INSERT_NOPS_8
    add t4, t4, t5      # t4 = t4 + t5 (t4 = 2 + 1)
    add t6, t6, -1
    add t3, t3, t4      # t3 = t3 + t4 (t3 = 3 + 3)
    INSERT_NOPS_9
    li t3, 0x3
    li t4, 0x2
    li t5, 0x1
    bne t6, zero, REPEAT    # Repete o ciclo

.end
```

Figura 2. Conflito de dados RAW entre duas instruções `add`

A pasta `[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL` fornece o projeto PlatformIO para que possa analisar, simular e modificar o programa como desejar. Abra o projeto no PlatformIO, construa-o e abra o ficheiro Disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_AL-AL/.pio/build/swervolf_nexys/firmware.dis`), verá que as duas instruções `add` que estamos a analisar estão colocadas nos endereços `0x000001A0` e `0x000001A8`:

<code>0x000001A0:</code>	<code>01ee8eb3</code>	<code>add t4, t4, t5</code>
<code>0x000001A4:</code>	<code>ffff8f93</code>	<code>addi t6, t6, -1</code>
<code>0x000001A8:</code>	<code>01de0e33</code>	<code>add t3, t3, t4</code>

A. Análise básica de um conflito de dados RAW entre instruções A-L

No exemplo que estamos a analisar, a segunda instrução `add` (`add t3, t3, t4`) necessita de utilizar o resultado da primeira instrução `add` (`add t4, t4, t5`) como segundo operando

de entrada. Este resultado está disponível no andar EX1, de onde pode ser encaminhado para o andar de Decode e usado pela segunda instrução `add`. No nosso exemplo (Figura 2), todas as iterações são iguais e t_4 é 2 inicialmente e 3 após a primeira adição. Este último valor (3) é o que a segunda adição deve usar como segundo operando de entrada, e não o valor lido do Register File (que é 2 até a primeira instrução `add` chegar ao andar do Writeback e o atualizar).

Figura 3 ilustra o fluxo das instruções do exemplo da Figura 2 através do pipeline SweRV EH1 para uma iteração aleatória do ciclo. No ciclo i , o valor computado no andar EX1 do Pipe I0 deve ser encaminhado para a instrução que está no andar Decode da Via-0, devido ao conflito de dados RAW entre as duas instruções `add` em análise.

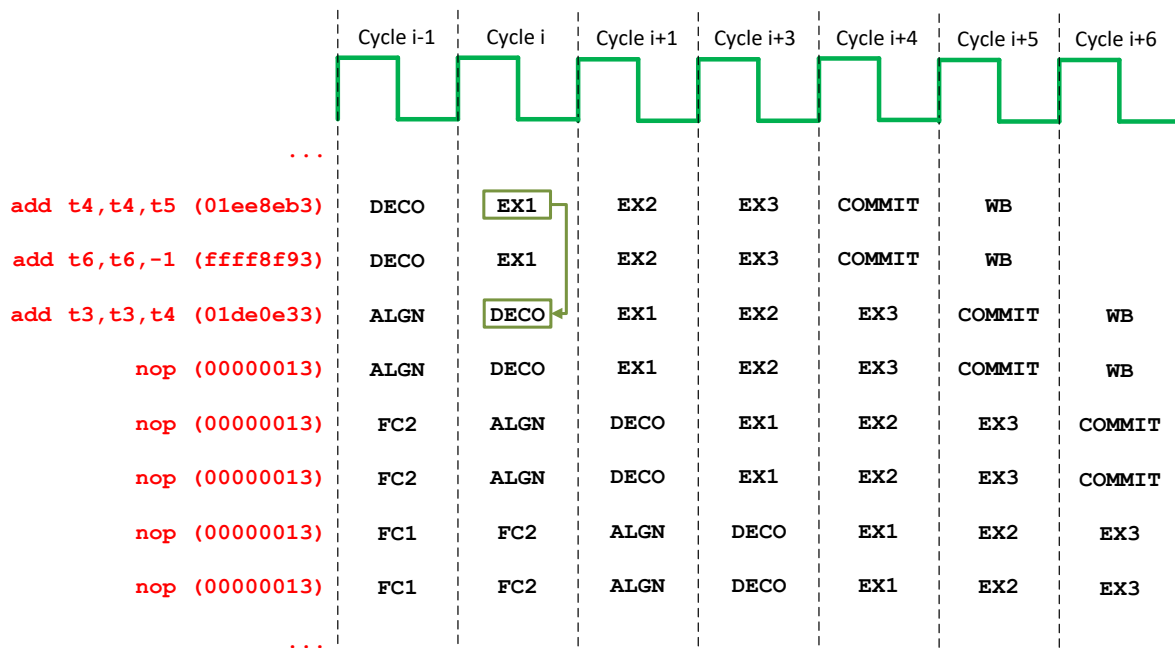


Figura 3. Execução da Figura 2 código de exemplo. O reencaminhamento é efectuado no ciclo i .

Figura 4 ilustra as andares SweRV EH1 Via-0 Decode e EX1 durante o ciclo i da Figura 3. Neste ciclo, a primeira instrução `add` (`add t4,t4,t5`) está no andar EX1 e a segunda instrução `add` (`add t3,t3,t4`) está no andar de Decodificação. Como mostra a figura, o resultado da primeira instrução `add` é encaminhado para a andar de Decode, é selecionado pela lógica de Forwarding (como analisaremos em detalhe na secção seguinte) e é utilizado como segundo operando de entrada para a segunda instrução `add`.

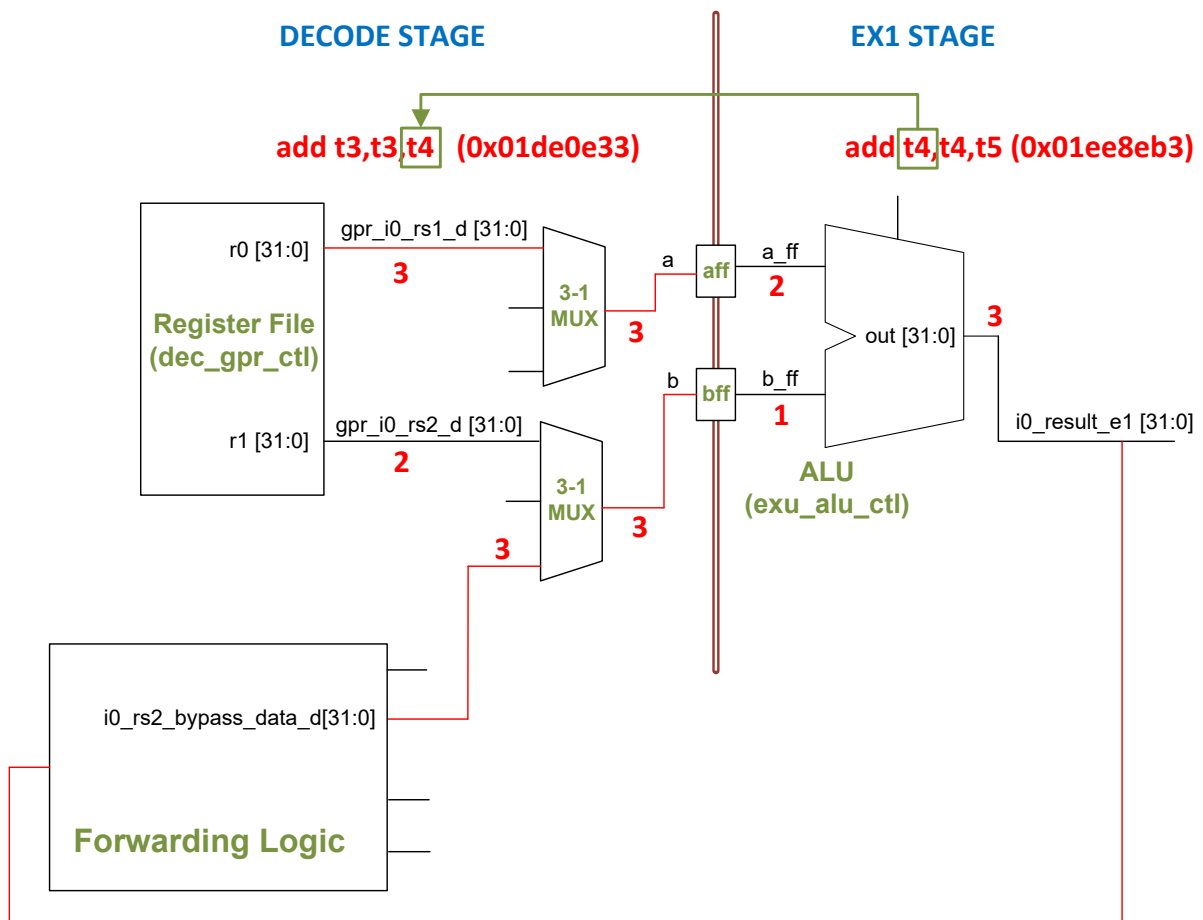


Figura 4. Resultado enviado de EX1 para o Decode (segundo operando) da Via-0

Finalmente, Figura 5 mostra a simulação do programa da Figura 2 durante os ciclos i e $i+1$ da Figura 3.

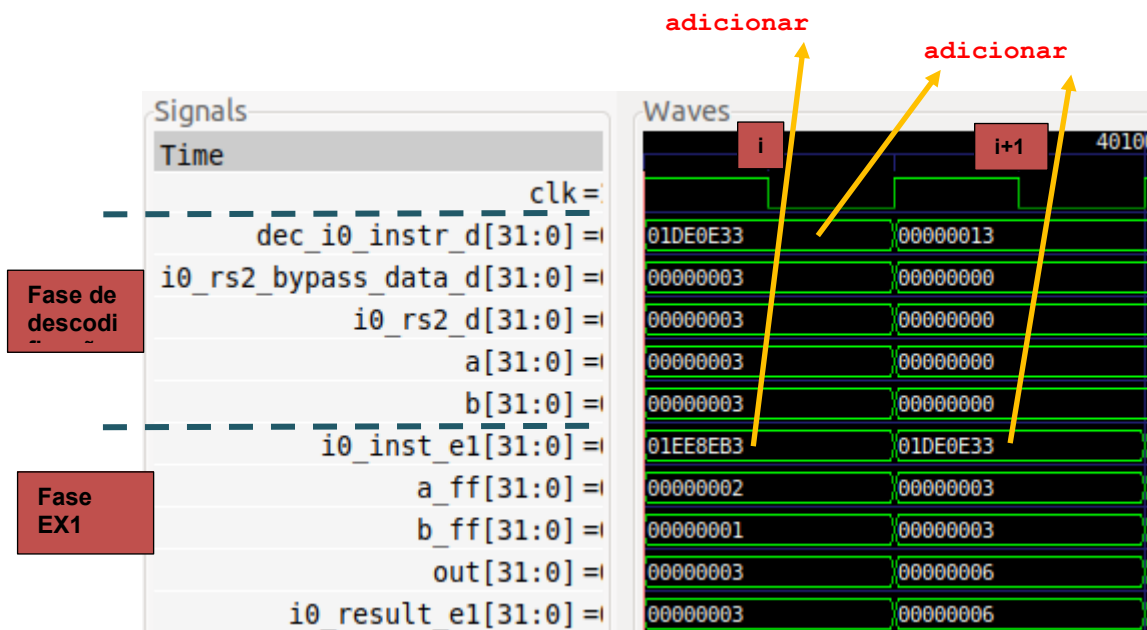


Figura 5. Simulação do código de exemplo da Figura 2

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

Analisar a simulação a partir de Figura 5 e o diagrama da Figura 4 em simultâneo.

- Instrução `add t4, t4, t5` (0x01ee8eb3):
 - o No ciclo i , esta instrução está no andar EX1 do Pipe I0 (`i0_inst_e1 = 0x01ee8eb3`). Ela calcula a seguinte adição na ALU:

$$a_ff(2) + b_ff(1) = out(3)$$
 O resultado da adição é fornecido como entrada para a lógica de Forwarding no andar de Decode, como se mostra na Figura 4.
- Instrução `add t3, t3, t4` (0x01de0e33):
 - o No ciclo i , esta instrução está no andar de Decode da Via-0 (`dec_i0_instr_d = 0x01de0e33`). A lógica de Forwarding conecta `i0_result_e1` com `i0_rs2_bypass_data_d`. Os dois multiplexers 3:1 selecionam os operandos de entrada para a adição que será calculada no ciclo seguinte (ciclo $i+1$) no andar EX1 do Pipe I0; especificamente:
 - $a = 3$ (do ficheiro de registo)
 - $b = 3$ (da saída da ALU no andar EX1 do Pipe I0, através da lógica de Forwarding, sinal `i0_rs2_bypass_data_d`)
 - o No ciclo $i+1$, esta instrução está no andar EX1 do Pipe I0 (`i0_inst_e1 = 0x01de0e33`). Ela calcula a seguinte adição na ALU:

$$a_ff(3) + b_ff(3) = out(6)$$

TAREFA: Remover todas as instruções `nop` no exemplo da Figura 2. Desenhe uma figura semelhante à Figura 3 para duas iterações consecutivas do ciclo, depois analise e confirme que a figura está correta comparando-a com uma simulação do Verilator e, finalmente, calcule o IPC utilizando os contadores de desempenho enquanto executa o programa na placa.

TAREFA: No exemplo da Figura 2 remova todas as instruções `nop` e mova a instrução `add t6, t6, -1` após a instrução `add t3, t3, t4` e, em seguida, reexamine o programa tanto na simulação quanto na placa. Neste programa reordenado, as duas instruções `add` dependentes (`add t4, t4, t5` e `add t3, t3, t4`) chegam à andar de Decode no mesmo ciclo, o que tem um impacto no desempenho. Explique o impacto destas alterações, utilizando tanto a simulação como a execução na placa.

Teste situações semelhantes em que substitui a instrução `add` dependente por outras instruções dependentes, como por exemplo:

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

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

B. Análise avançada de um conflito de dados RAW entre instruções A-L

i. Explicação teórica

Figura 6 estende os diagramas da Figura 1 e Figura 4 adicionando um multiplexer 10:1 (rodeado por um quadrado azul na Figura 6) que produz o sinal `i0_rs2_bypass_data_d`, que no nosso exemplo da Figura 2 fornece o segundo operando de entrada para a segunda instrução `add` (`add t3,t3,t4`). Este multiplexer 10:1 é implementado dentro da caixa de lógica de Forwarding mostrada na Figura 1 e na Figura 4.

A figura também mostra as ligações de entrada deste multiplexer 10:1. O valor encaminhado pode provir de uma instrução executada através da Via 0 ou da Via 1. Assim, são necessários cinco caminhos de Forwarding por via. Especificamente, as entradas para o multiplexer 10:1 podem vir de qualquer um dos andares subsequentes (EX1, EX2, EX3, Commit e Writeback) da Via 0 ou Via 1. Por uma questão de simplicidade, ligamos as cinco entradas provenientes da Via 0 utilizando fios, enquanto as 5 entradas provenientes da Via 1 são ligadas por nomes.

Três multiplexers 10:1 adicionais dentro da lógica de Forwarding computam os três outros operandos de origem: sinais `i0_rs1_bypass_data_d`, `i1_rs1_bypass_data_d` e `i1_rs2_bypass_data_d`. No entanto, não os mostramos na figura porque não são utilizados no exemplo que analisamos nesta secção (Figura 2). Todos os quatro multiplexers podem ser encontrados nas linhas 2429-2473 do módulo `dec_decode_ctl`.

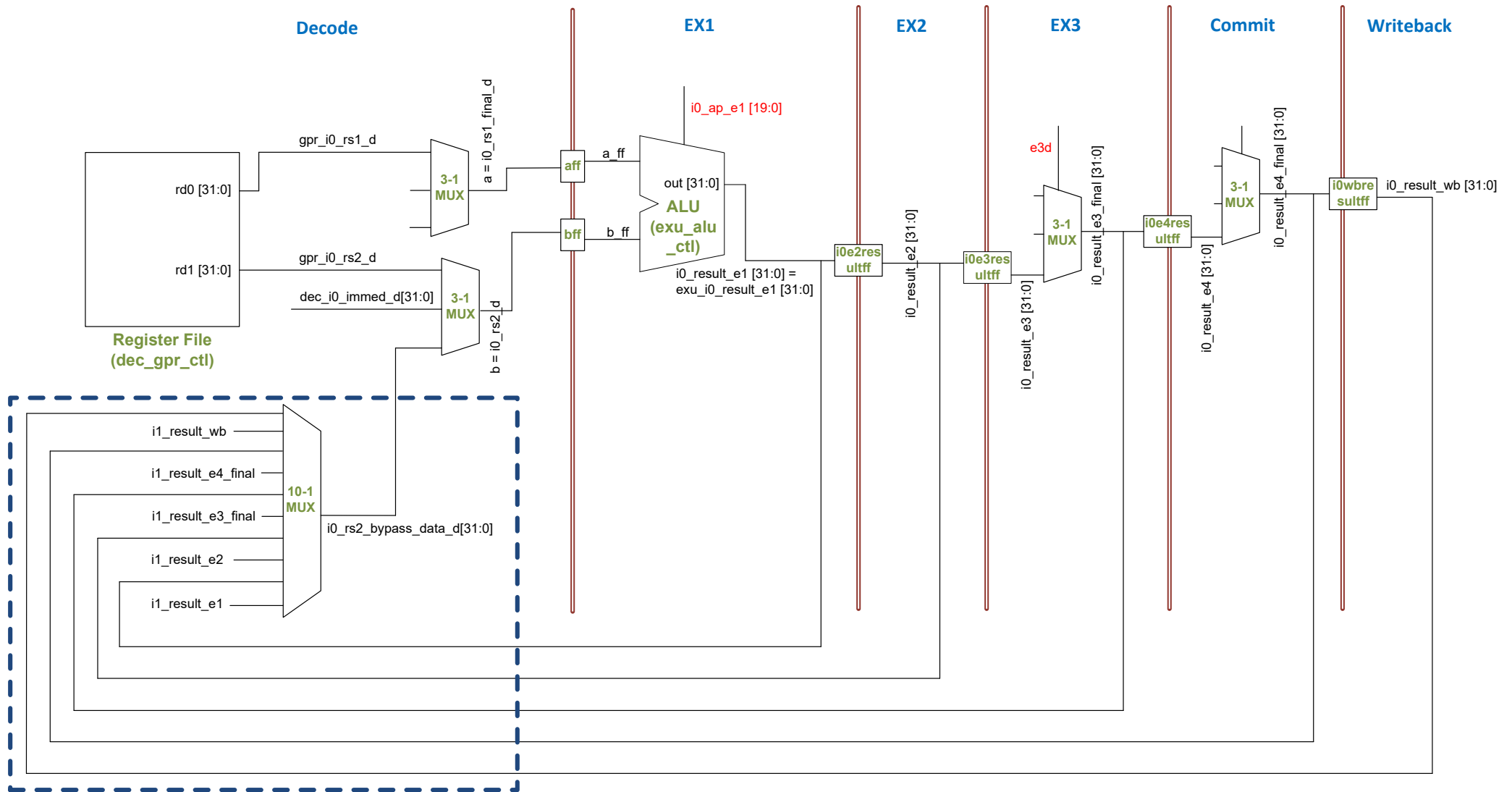


Figura 6. Pipe I0 incluindo a lógica de Forwarding utilizada para a segunda fonte de entrada da ALU

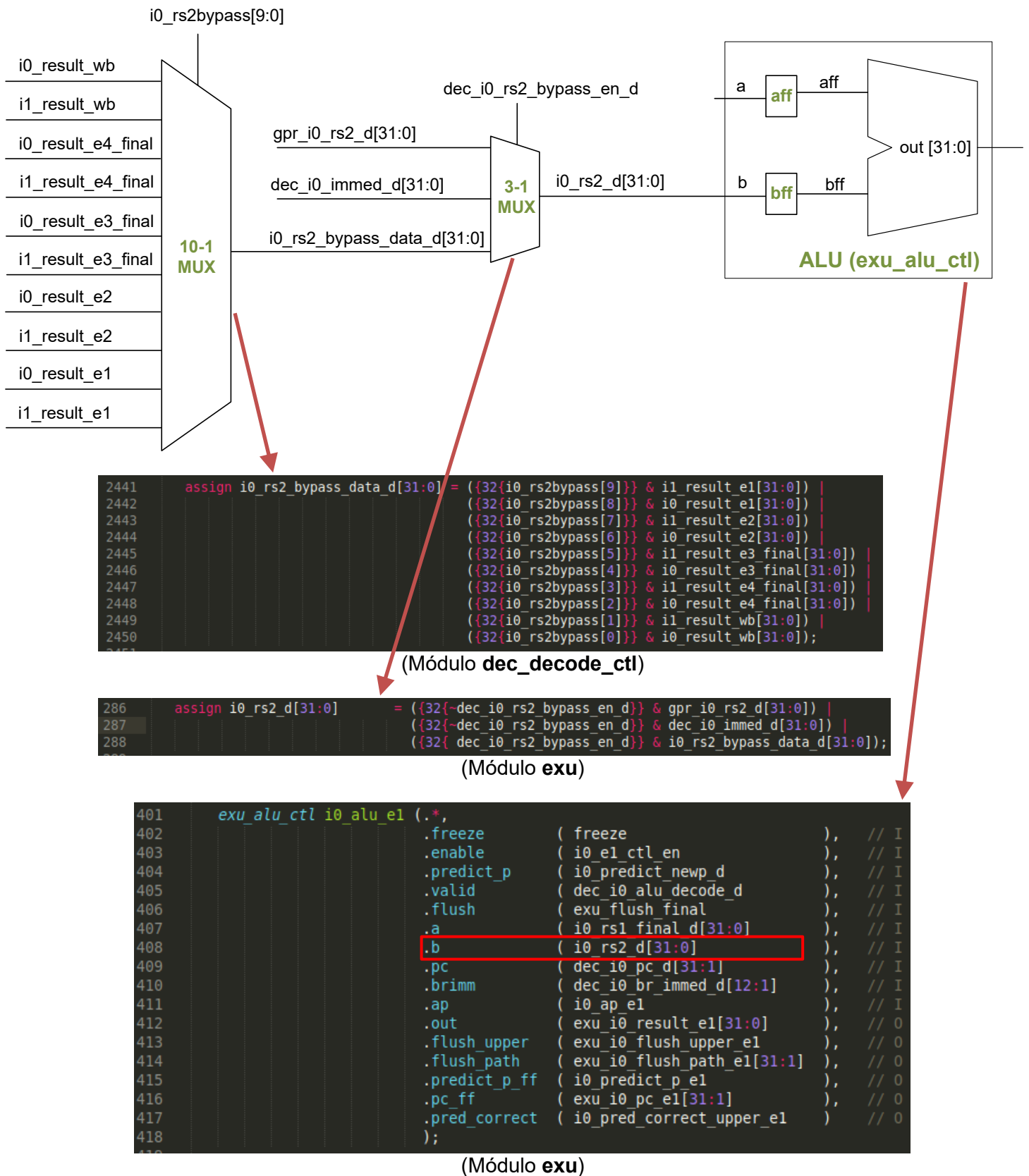


Figura 7. Multiplexers 10:1 e 3:1 destacados na Figura 6

Figura 7 faz uma ampliação dos dois multiplexers (multiplexers 10:1 e 3:1) da Figura 6 que calculam o segundo operando de entrada para a ALU do Pipe I0 (b). A figura mostra um diagrama de blocos e o código Verilog onde estes multiplexers são implementados nos módulos **dec_decode_ctl** e **exu**.

NOTA: Os dois multiplexers da Figura 7 também existem no processador do DDCARV. Os dados são encaminhados para o andar Execute nesse processador e existem menos caminhos de Forwarding porque não é superescalar e tem um pipeline mais curto. Pode analisar os caminhos de Forwarding na Figura 7.55 do DDCARV.

Em seguida, analisamos as entradas, as saídas e o sinal de controlo dos dois multiplexers apresentados na Figura 7.

Multiplexer 10:1:

Saída: A saída do multiplexer 10:1 é `i0_rs2_bypass_data_d[31:0]`. Este sinal contém o valor que deve ser encaminhado (bypass) para a instrução no andar de Decode.

Entradas: As entradas para o multiplexer 10:1 são os resultados de instruções anteriores no programa que estão a ser executadas em andares posteriores (EX1, EX2, EX3, Commit ou Writeback). Cinco desses sinais vêm do Pipe I0 (como mostrado na Figura 6) e os outros cinco sinais vêm do Pipe I1 (não mostrado na Figura 6), uma vez que a instrução no andar de Decode pode potencialmente depender de uma instrução executada de qualquer uma das duas formas.

Sinal de controlo: O sinal de controlo (`i0_rs2bypass[9:0]`) seleciona qual a entrada que está ligada à saída do multiplexer. Tem 10 bits, mas apenas um bit pode ser ativo (podem ser zero se não houver conflito de dados). O multiplexer atua do seguinte modo:

- Se `i0_rs2bypass[9] == 1` → `i0_rs2_bypass_data_d = i1_result_e1`
- Se `i0_rs2bypass[8] == 1` → `i0_rs2_bypass_data_d = i0_result_e1`
- Se `i0_rs2bypass[7] == 1` → `i0_rs2_bypass_data_d = i1_result_e2`
- Se `i0_rs2bypass[6] == 1` → `i0_rs2_bypass_data_d = i0_result_e2`
- Se `i0_rs2bypass[5] == 1` → `i0_rs2_bypass_data_d = i1_result_e3_final`
- Se `i0_rs2bypass[4] == 1` → `i0_rs2_bypass_data_d = i0_result_e3_final`
- Se `i0_rs2bypass[3] == 1` → `i0_rs2_bypass_data_d = i1_result_e4_final`
- Se `i0_rs2bypass[2] == 1` → `i0_rs2_bypass_data_d = i0_result_e4_final`
- Se `i0_rs2bypass[1] == 1` → `i0_rs2_bypass_data_d = i1_result_wb`
- Se `i0_rs2bypass[0] == 1` → `i0_rs2_bypass_data_d = i0_result_wb`

Para compreender como é calculado este sinal de controlo de 10 bits, explicamos o cálculo do sinal `i0_rs2bypass[8]`, que é o que fica ativo no nosso exemplo da Figura 2 para o bypass add-add.

- Se `i0_rs2bypass[8]` for 1, o valor de bypass selecionado é `i0_result_e1`, que é o resultado da instrução executada no andar EX1 do Pipe I0 (ver Figura 6).
- Para que EX1 encaminhe dados para o andar de Decode (ambos no pipe I0), devem

ocorrer as seguintes condições (ver secção 4 do documento SweRVref para rever os sinais de controlo):

- O segundo operando de entrada da instrução no andar de Decode é lido a partir do Register File, e não é lido a partir do registo zero. Na Unidade de Controlo SweRV EH1, isto ocorre quando `dec_i0_rs2_en_d` é 1. O código Verilog correspondente é:

```
dec_i0_rs2_en_d = i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
```

- A instrução no andar EX1 do Pipe I0 é válida:
`e1d.i0v == 1`
- O registo de destino da instrução no andar EX1 (do pipe I0) e o segundo registo de origem da instrução no andar de Decode (da Via 0) são os mesmos:

```
e1d.i0rd[4:0] == i0r.rs2[4:0]
```

- A instrução no andar EX1 (do Pipe I0) é uma operação ALU:

```
i0_rs2_class_d.alu == 1
```

- Tendo tudo isto em conta, podemos concluir que:

```
i0_rs2bypass[8] =
    (i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0)) &
    e1d.i0v &
    (e1d.i0rd[4:0] == i0r.rs2[4:0]) &
    i0_rs2_class_d.alu ;
```

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

TAREFA: Analisar o código Verilog para explicar como é efetuado o cálculo da equação anterior. Deve inspecionar as seguintes linhas do módulo `dec_decode_ctl`.

```
2384 assign i0_rs2bypass[9:0] = { i0_rs2_depth_d[3:0] == 4'd1 & i0_rs2_class_d.alu,
2385                               i0_rs2_depth_d[3:0] == 4'd2 & i0_rs2_class_d.alu,
2386                               i0_rs2_depth_d[3:0] == 4'd3 & i0_rs2_class_d.alu,
```

```
1733 assign {i0_rs2_class_d, i0_rs2_depth_d[3:0]} =
1734 (i0_rs2_depend_i1_e1) ? { i1_elc, 4'd1 } :
1735 (i0_rs2_depend_i0_e1) ? { i0_elc, 4'd2 } :
1736 (i0_rs2_depend_i1_e2) ? { i1_e2c, 4'd3 } :
```

```
1509 assign i0_rs2_depend_i0_e1 = dec_i0_rs2_en_d & e1d.i0v & (e1d.i0rd[4:0] == i0r.rs2[4:0]);
```

```
1131 assign dec_i0_rs2_en_d = i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
```

TAREFA: Escrever equações (semelhantes à anterior) para outros bits de controlo de `i0_rs2bypass[9:0]`, `i0_rs1bypass[9:0]`, `i1_rs2bypass[9:0]` e `i1_rs1bypass[9:0]`.

Multiplexer 3:1:

Saída: A saída do multiplexer 3:1 é `i0_rs2_d[31:0]`. Este sinal é enviado para a segunda entrada (b) da ALU no Via 0.

Entradas: As entradas para o multiplexer 3:1 são:

- O valor lido do Register File (`gpr_i0_rs2_d`).
- O valor Imediato (`dec_i0_immed_d`), obtido a partir da instrução.
- O valor encaminhado a partir de andares posteriores (`i0_rs2_bypass_data_d`) obtido a partir do multiplexer 10:1 descrito anteriormente.

Sinal de controlo: O sinal de controlo para o multiplexer 3:1 (`dec_i0_rs2_bypass_en_d`) seleciona uma das duas opções:

- O valor ignorado dos andares posteriores (`i0_rs2_bypass_data_d`), se `dec_i0_rs2_bypass_en_d == 1`
- Ou o valor proveniente do Register File ou do Imediato (`gpr_i0_rs2_d` e `dec_i0_immed_d`, respetivamente), se `dec_i0_rs2_bypass_en_d == 0`. Pode parecer estranho que o mesmo sinal selecione duas entradas; no entanto, o sinal que não deve ser selecionado (`gpr_i0_rs2_d` ou `dec_i0_immed_d`) é forçado a zero no código Verilog.

O sinal de seleção do multiplexer 3:1 (`dec_i0_rs2_bypass_en_d`) é simplesmente calculado como o OR lógico do sinal de controlo de 10 bits do multiplexer 10:1:

```
assign dec_i0_rs2_bypass_en_d = |i0_rs2bypass[9:0];
```

Assim, sempre que o segundo operando de entrada de uma instrução depende do resultado de uma instrução anterior que ainda está a ser executada (ou seja, qualquer um dos 10 bits do sinal `i0_rs2bypass[9:0]` é 1), `dec_i0_rs2_bypass_en_d == 1` e o operando é obtido através de Forwarding. Inversamente, se não depender de nenhuma instrução anterior, `dec_i0_rs2_bypass_en_d == 0` e o operando vem do Register File ou do Imediato.

ii. Experiência

A Figura 8 mostra a simulação do programa da Figura 2 numa iteração aleatória do ciclo. O ciclo *i* da Figura 3 está indicado na parte superior da figura.

Os sinais na parte superior (Sinais de *Trace*) são incluídos para ajudar a seguir as instruções à medida que elas progridem pelo pipeline. Observe que esses sinais já foram usados em laboratórios anteriores. O significado de cada sinal na Via 0 é o seguinte (o mesmo se aplica à Via 1, bastando substituir `i0` por `i1` nos nomes dos sinais):

- `dec_i0_instr_d` → instrução no andar Decode
- `i0_inst_e1` → instrução no andar EX1
- `i0_inst_e2` → instrução no andar EX2
- `i0_inst_e3` → instrução no andar EX3

- `i0_inst_e4` → instrução no andar Commit
- `i0_inst_wb` → instrução no andar Writeback

Abaixo dos sinais de *trace*, são mostrados os sinais principais de cada multiplexer analisado acima. Cada multiplexer está rodeado por duas linhas azuis, enquanto o sinal de controlo, as entradas e a saída de cada multiplexer estão separados por linhas vermelhas.

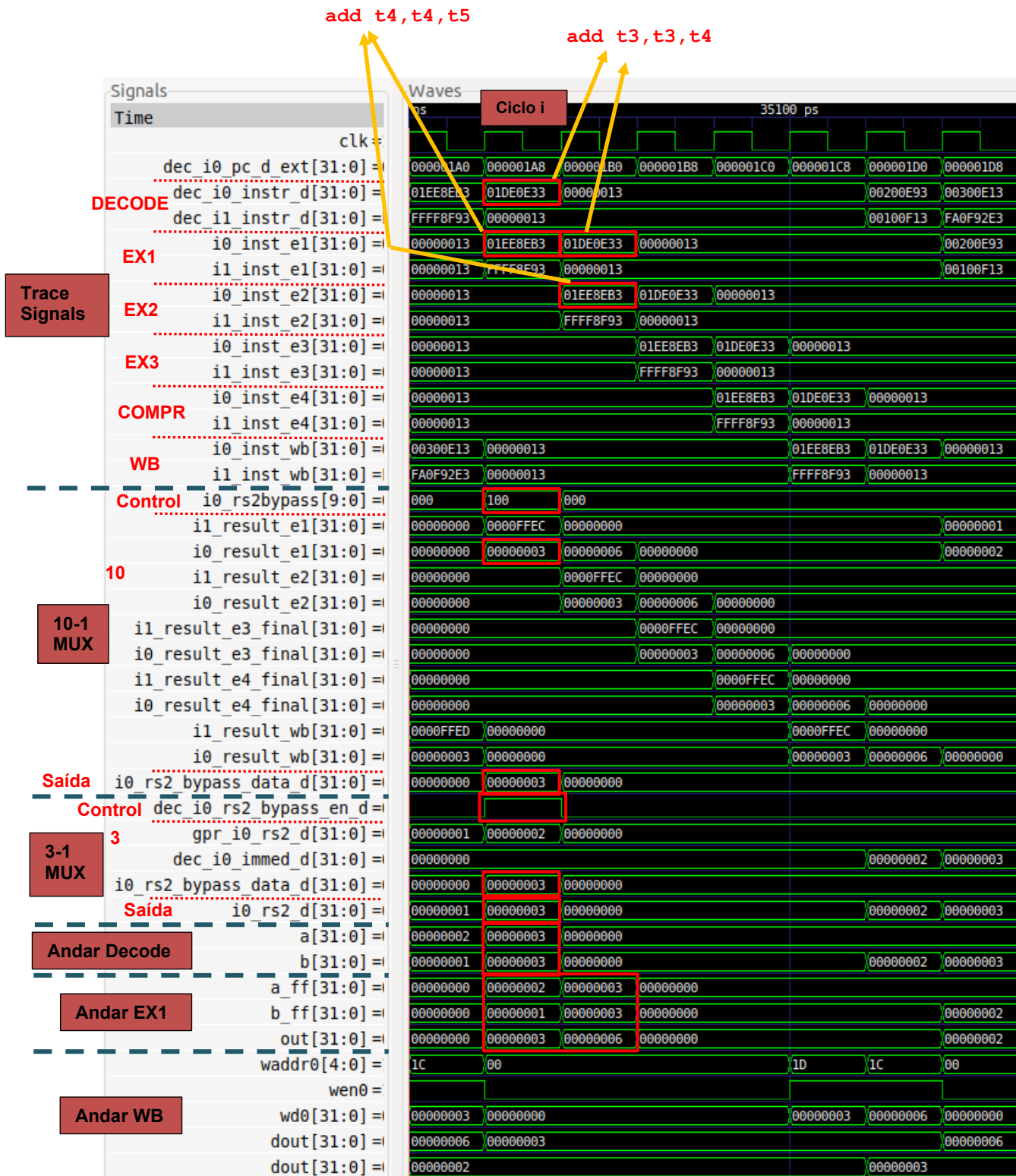


Figura 8. Simulação do programa da Figura 2 numa iteração aleatória do ciclo

Figura 9 mostra os andares Decode e EX1 durante a execução do programa da Figura 2 no ciclo *i* (tal como definido na Figura 8).

Cycle i

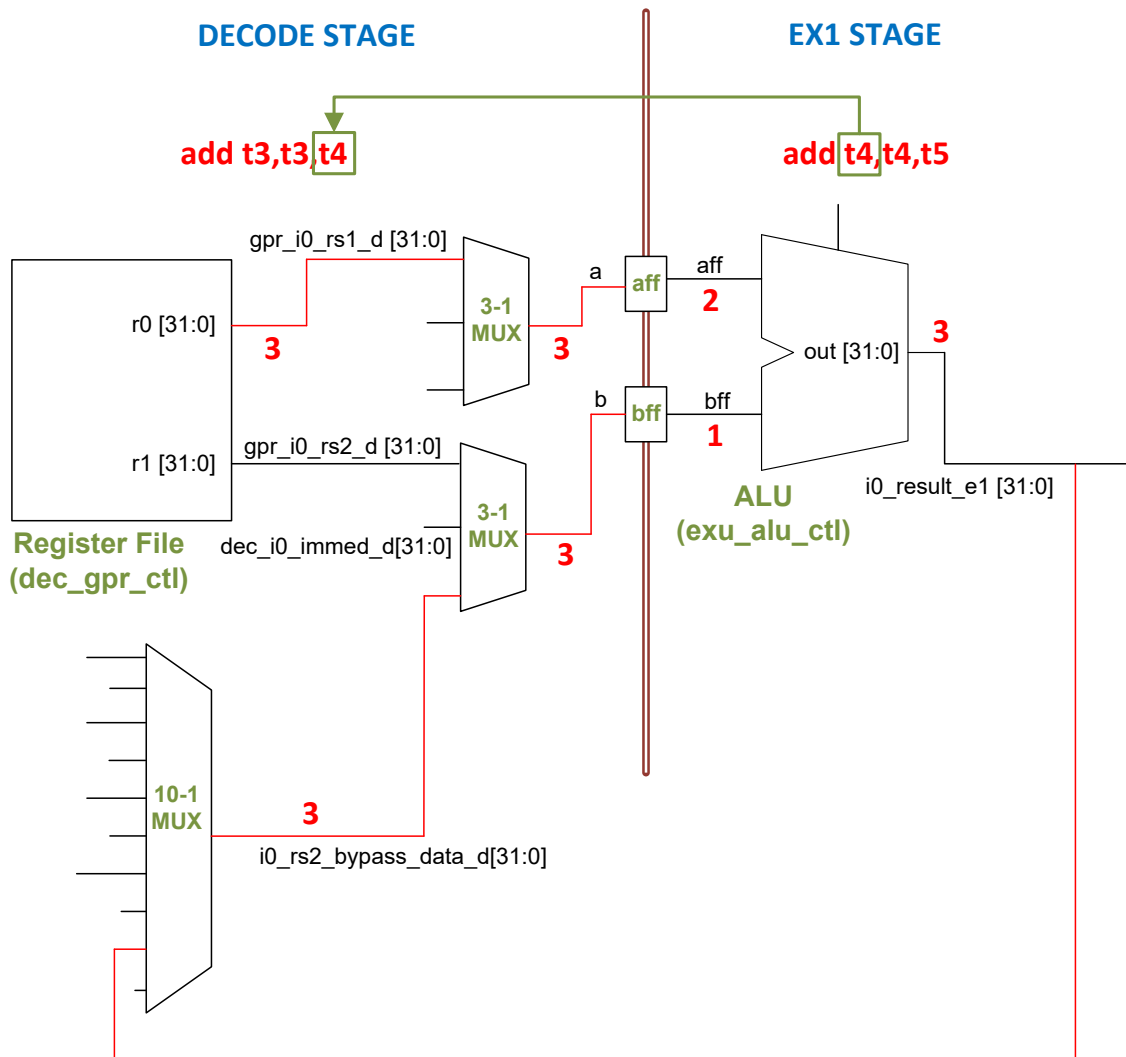


Figura 9. Andar Decode e EX1 durante a execução da Figura 2 no ciclo *i* (tal como definido na Figura 8)

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

Analisar a simulação a partir de Figura 8 e o diagrama da Figura 9 em simultâneo.

- Sinais de *trace* apresentados na Figura 8:

- No ciclo *i*, a segunda instrução `add` está a ser executada no andar de Decode da via 0 (`dec_i0_instr_d = 0x01DE0E33`) e a primeira instrução `add` está a ser executada no andar EX1 do pipeline I0 (`i0_inst_e1 = 0x01EE8EB3`).
- No ciclo *i+1*, o segundo `add` progride para o andar EX1 do I0 Pipe (`i0_inst_e1 = 0x01DE0E33`) e o primeiro `add` progride para o andar EX2

do I0 Pipe ($i0_inst_e2 = 0x01EE8EB3$).

- **Multiplexer 10:1:** No ciclo i , o sinal $i0_rs2bypass[9:0] = 0x100$ (ou seja, $i0_rs2bypass[8] = 1$), pelo que a saída é ligada ao valor proveniente do andar EX1 do pipe I0 (ver Figura 9):
 $i0_rs2_bypass_data_d = i0_result_e1 = 0x00000003$
- **Multiplexer 3:1:** No ciclo i , o sinal $dec_i0_rs2_bypass_en_d = 1$, pelo que a saída é ligada ao valor proveniente da lógica de bypass (ver Figura 9):
 $i0_rs2_d = i0_rs2_bypass_data_d = 0x00000003$
- **Andar EX1 representado na Figura 8:**
 - o No ciclo i , a primeira instrução `add` calcula a adição na ALU do pipe I0:
 $a_ff(2) + b_ff(1) = out(3)$.
 - o No ciclo $i+1$, a segunda instrução `add` calcula a adição na ALU do pipe I0:
 $a_ff(3) + b_ff(3) = out(6)$.

TAREFA: Para o programa da Figura 2 efetuar a mesma análise que na 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 é o que recebe os dados de Forwarding.

Também pode criar outros exemplos em que as duas instruções `add` estejam a ser executadas através do pipe I1 e confirmar que o comportamento é o mesmo.

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

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

3. RESOLVER OS CONFLITOS DOS DADOS COM O FORWARDING NO ANDAR DE COMMIT

Uma situação mais delicada ocorre quando uma instrução depende de uma instrução anterior que necessita de vários ciclos para obter o resultado (i.e. uma operação multi-ciclo), como por exemplo uma instrução `lw`, uma instrução `mul`, uma instrução `div`, etc. Nesta secção analisamos uma situação específica que pode ocorrer na execução de uma instrução `lw` e de uma instrução `add` dependente, e deixamos como exercício a análise de outras instruções e situações.

Como explicado no Lab 13, uma instrução `lw` necessita de três ciclos (andares DC1, DC2 e DC3) para obter o seu resultado quando é utilizada a memória DCCM de baixa latência. É este o cenário utilizado nesta secção. (Como também analisámos nos Labs 13 e 14, ocorre um atraso maior quando é utilizada a Memória DDR2 Externa - os efeitos desta maior latência da memória nos conflitos dos dados são deixados como exercício).

Se a instrução `lw` for executada três ou mais ciclos antes da instrução `add` dependente, o conflito é resolvido como explicado na Secção 2. Neste caso, os mesmos multiplexers 10:1 e 3:1 descritos nessa secção são usados para encaminhar os dados lidos pela instrução `lw` para a instrução subsequente que dela depende.

APÊNDICE A: O apêndice no final do documento inclui um exemplo de um conflito de dados RAW `lw-add` que é tratado como explicado na Secção 2.

No entanto, se a instrução `lw` for executada mais perto da instrução `add` dependente, o conflito é resolvido de uma forma diferente da descrita na Secção 2. O problema agora é que quando a instrução `add` atinge o andar EX1, o valor lido pela instrução `lw` ainda não está disponível.

No processador em pipeline explicado em DDCARV, são introduzidas bolhas neste caso, que fazem a instrução dependente esperar e só utilizam o valor lido quando este está disponível. Isto requer pouco hardware adicional, mas tem impacto no desempenho. Assim, o SweRV EH1 permite que a instrução dependente continue através do pipeline e depois recalcule a operação no andar Commit, se necessário devido a uma dependência de dados.

Especificamente, o SweRV EH1 adiciona uma ALU extra (a ALU secundária) no andar Commit de cada via. Esta ALU recalcula a operação aritmética-lógica com as entradas corretas quando necessário. Assim, não se perdem ciclos devido a bloqueios - mas à custa da adição de duas ALUs extra (uma por via), bem como de sinais de controlo e lógica adicionais. Figura 10 ilustra a implementação desta ALU secundária no andar Commit da via 0 (a ALU está rodeada por um quadrado azul), bem como a lógica de Forwarding adicionada no andar EX3 para o segundo operando de entrada (esta lógica está rodeada por um quadrado vermelho). (Na Figura 4 do Lab 11, estas duas ALU extra e os caminhos de encaminhamento não foram incluídos por uma questão de simplicidade).

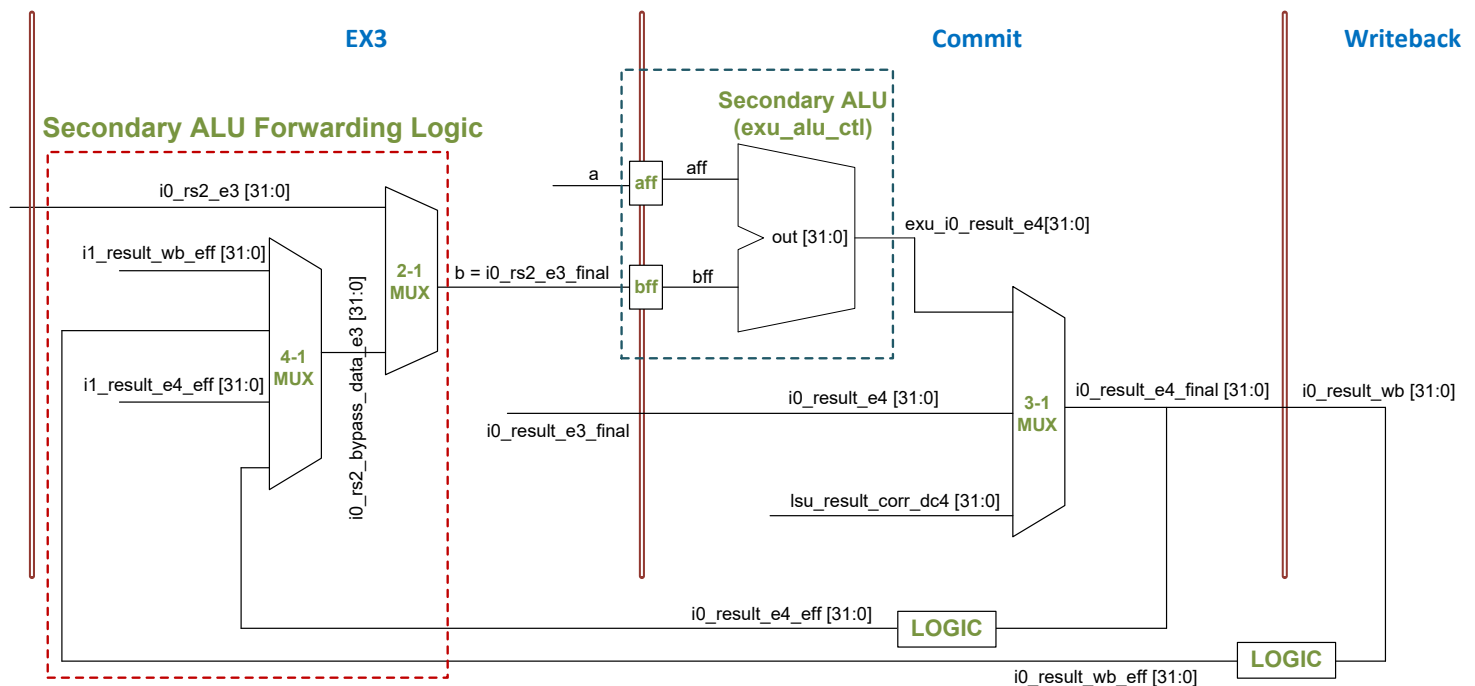


Figura 10. ALU secundária no andar de Commit da Via 0

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

Figura 11 mostra o código de exemplo utilizado nesta secção. Executa uma instrução `lw` seguida imediatamente por uma instrução de adição independente (`add t6, t6, -1`: que calcula o índice do ciclo) e depois uma instrução `add` que depende da leitura. A instrução `add` independente é incluída para forçar tanto a instrução `lw` como a instrução `add` dependente a serem executadas através da Via 0. Assim, a única diferença em relação ao programa do Apêndice é que as instruções `lw` e `add` estão agora mais próximas; no entanto, esta pequena diferença no programa traduz-se numa enorme diferença na forma como é executado, como acabámos de explicar e vamos demonstrar de seguida.

```
.globl Test_Assembly

.section .midccm
#.data
A: .space 4

.text
Test_Assembly:
la t0, A                # t0 = addr(A)
li t1, 0x1              # t1 = 1
sw t1, (t0)             # A[0] = 1
li t1, 0x0
li t3, 0x1
li t6, 0xFFFF

REPEAT:
    beq t6, zero, OUT    # Stay in the loop?
    INSERT_NOPS_9
    lw t1, (t0)
    add t6, t6, -1
    add t3, t3, t1        # t3 = t3 + t1
    INSERT_NOPS_8
```

```
li t1, 0x0
li t3, 0x1
add t4, t4, 0x1
add t5, t5, 0x1
j REPEAT
OUT:
.end
```

Figura 11. Programa que executa um `lw`, um `add` independente e um `add` dependente

Como habitualmente, a pasta `[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL` tem o projeto PlatformIO para que se possa analisar, simular e modificar o programa como desejado. Abra o projeto no PlatformIO, compile-o e abra o ficheiro de Disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_Close-LW-AL/.pio/build/swervolf_nexys/firmware.dis`). Observe que as instruções `lw` e `add` são colocadas nos endereços `0x000001bc` e `0x000001c4`.

<code>0x000001bc:</code>	<code>0002a303</code>	<code>lw t1,0(t0)</code>
<code>0x000001c0:</code>	<code>ffff8f93</code>	<code>addi t6,t6,-1</code>
<code>0x000001c4:</code>	<code>006e0e33</code>	<code>add t3,t3,t1</code>

Figura 12 mostra a simulação do programa da Figura 11 numa iteração aleatória do ciclo. Mais uma vez, qualquer iteração seria válida, exceto a primeira, que deve ser evitada devido aos erros na cache de instruções. Tal como no exemplo da secção anterior, os sinais no topo (Sinais de *trace*) são incluídos para ajudar a seguir as instruções à medida que progridem através do pipeline. Abaixo dos sinais de *trace*, são mostrados os principais sinais dos multiplexers 4:1 e 2:1 e a nova ALU de Figura 11.

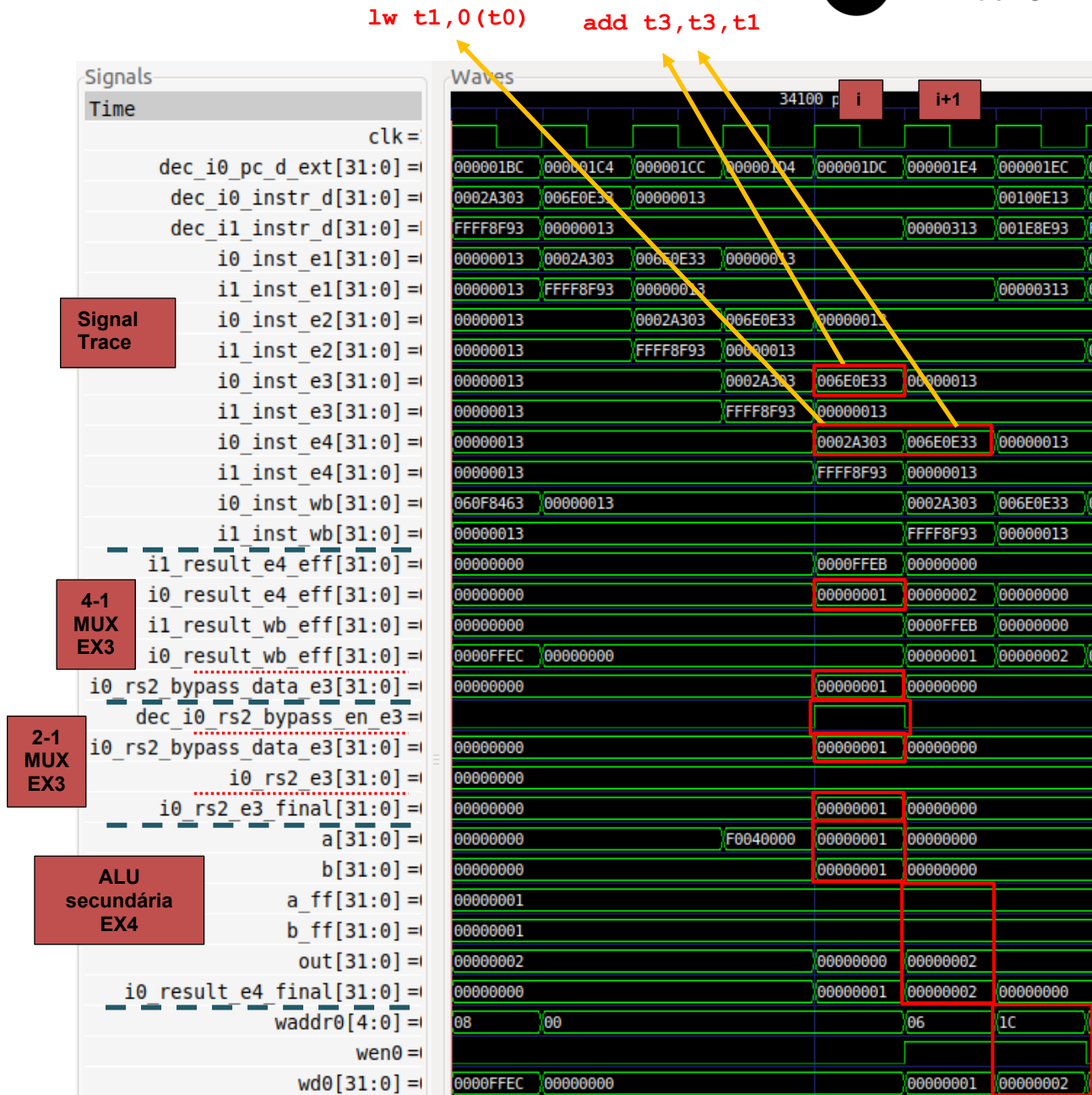


Figura 12. Simulação do programa de Figura 11 na terceira iteração do ciclo

Figura 13 mostra um diagrama da execução do programa da Figura 11 na sétima iteração do ciclo e para o ciclo i mostrado na Figura 12, quando a instrução `add` está no andar EX3 e a instrução `lw` está no andar Commit, e para o ciclo $i+1$, quando a instrução `add` está no andar Commit (i.e., EX4) e recalcula a operação nas entradas corretas.

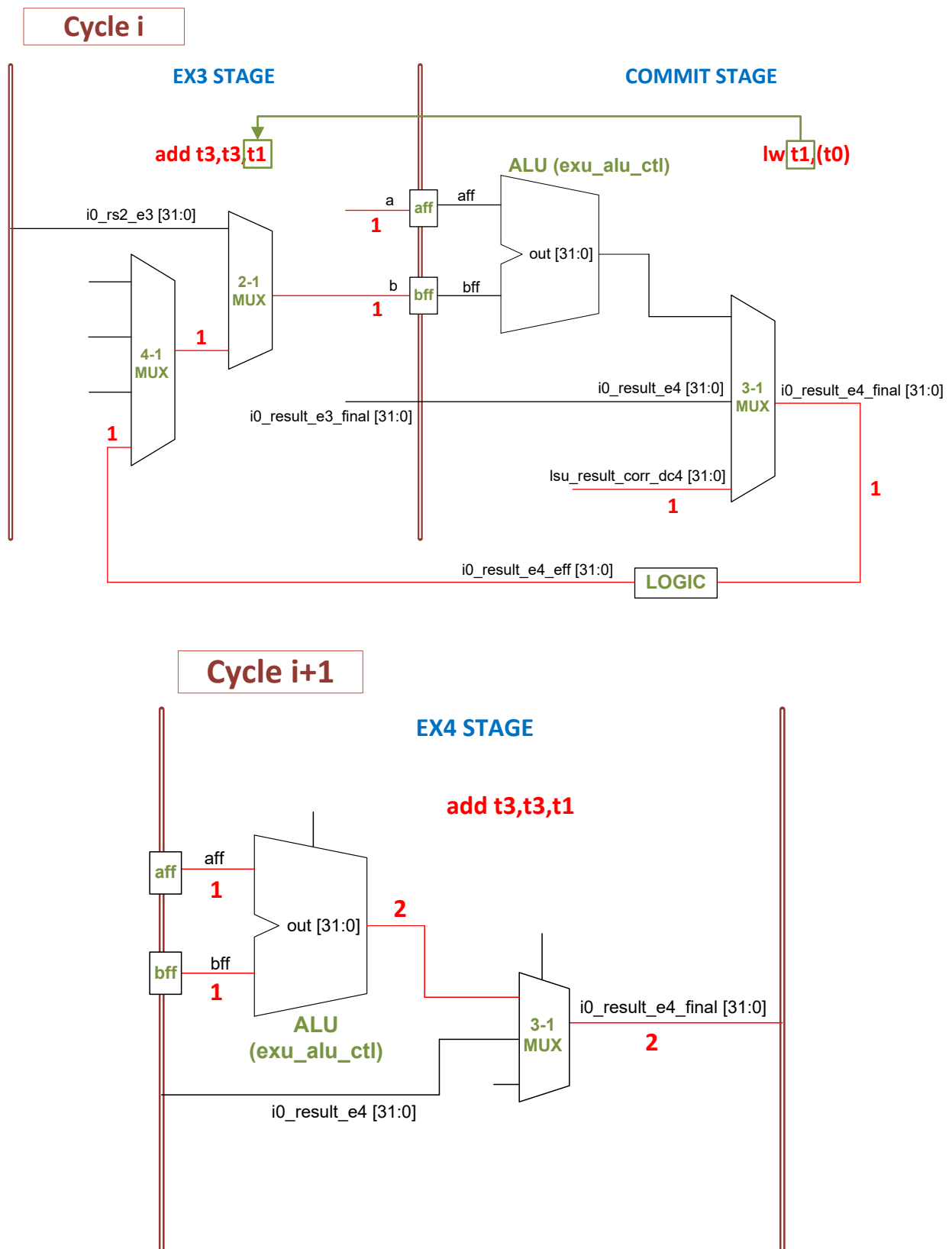


Figura 13. Diagrama da execução do programa da Figura 11 na sétima iteração do ciclo e para os ciclos i e $i+1$ de Figura 12

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

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

Analisar a forma de onda da Figura 12 e o diagrama da Figura 13 em simultâneo.

- **Sinais de trace apresentados na Figura 12:**
 - o No ciclo i , a instrução `add` está no andar EX3 da Via 0 (`i0_inst_e3 = 0x006E0E33`), e a instrução `lw` está no andar Commit do Pipe I0 (`i0_inst_e4 = 0x0002A303`).
 - o No ciclo $i+1$, a instrução `add` está no andar Commit da Via 0 (`i0_inst_e4 = 0x006E0E33`).
- **4-1 Multiplexer:** No ciclo i , é selecionado o valor lido pela instrução de leitura, que neste ciclo se encontra no andar Commit:


```
i0_rs2_bypass_data_e3 = i0_result_e4_eff = 0x00000001
```
- **Multiplexer 2-1:** No ciclo i , devido à dependência entre a leitura e a adição, o valor de `bypass` é selecionado (`dec_i0_rs2_bypass_en_e3 = 1`). Assim:


```
i0_rs2_e3_final = i0_rs2_bypass_data_e3 = 0x00000001
```
- **ALU do andar de compromisso:** No ciclo $i+1$, a adição é recalculada utilizando os valores corretos:


```
out = a_ff + b_ff = 0x00000001 + 0x00000001 = 0x00000002
```

 Depois, no multiplexer 3:1, a saída da ALU é selecionada (`exu_i0_result_e4`). Note-se que, se não existir qualquer dependência, o valor no sinal `i0_result_e4` seria selecionado.

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

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

TAREFA: Desativar a ALU secundária como explicado no Lab 11 e analisar o exemplo da Figura 11 tanto com uma simulação no Verilator como com uma execução na placa.

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

4. EXERCÍCIOS

- 1) Modifique o programa utilizado na Secção 3, adicionando uma instrução aritmética-lógica extra que depende do resultado da instrução `add`. Por exemplo, pode substituir o ciclo da Figura 11 pelo seguinte código, onde foi incluída uma nova instrução AND (`and t3, t4, t3`), e onde reordenámos ligeiramente o código movendo para a 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 são tratados os conflitos de dados para a nova instrução A-L. Em seguida, remova todas as instruções nop e analise os resultados fornecidos pelos contadores HW.

- 2) Analise a mesma situação que a descrita na Secção 3 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`.

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

- 4) (O exercício seguinte baseia-se nos exercícios 4.18, 4.19, 4.20 e 4.26 de [PaHe]). Suponha que executou o código abaixo numa versão do processador SweRV EH1 que não trata os conflitos de dados (ou seja, o programador é responsável por tratar os conflitos de dados inserindo nops quando necessário). Adicione nops 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 apresentem diferentes tipos de conflitos de dados RAW. O tipo de dependência de dados

RAW é identificado pelo andar que produz o resultado e pela instrução seguinte que consome o resultado.

Para cada sequência, quantos nops teriam de ser inseridos e onde, para permitir que o seu código fosse executado corretamente num processador SweRV EH1 sem encaminhamento ou deteção de conflito? Qual é o IPC se utilizarmos o encaminhamento disponível no SweRV EH1 e não inserirmos nops?

5) No programa da Secção 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`. Isto introduz um conflito WAW entre a instrução `add` modificada e a leitura não bloqueante no início do ciclo (`lw x28, (x29)`). Analise em simulação como este conflito é tratado no SweRV EH1, para o qual pode olhar para o valor do sinal `wen2` no Register File. Tente compreender como este sinal é calculado na Unidade de Controlo (módulo **dec**).

6) No programa da Secção 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`. Isto introduz um conflito RAW entre a instrução `add` modificada e a leitura não bloqueante no início do ciclo (`lw x28, (x29)`). Analise em simulação como é que este conflito é tratado no SweRV EH1.

7) No programa da Secção 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`, e a instrução `add x7, x7, 1` por `add x28, x7, 1`. Isto faz com que ocorra um conflito RAW e um conflito WAW. Analise em simulação como é que estes dois conflitos são tratados no SweRV EH1.

8) Store-Load Forwarding (Escrita-Leitura)

Esta é uma situação muito interessante que não analisámos neste laboratório e que irá analisar neste exercício. Quando um store seguido de um load acedem ao mesmo endereço, os dados podem ser encaminhados do store para o load dentro do núcleo e a leitura da Memória Externa DDR pode ser evitada, poupando tempo e energia.

A lógica que implementa este reencaminhamento está incluída no 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 Store-Load Forwarding. É fornecido um script *.tcl* nessa pasta, que pode utilizar para analisar uma iteração aleatória do ciclo e compreender como é efetuado o Store-Load Forwarding no SweRV EH1.

APÊNDICE A

Neste apêndice, incluímos um exemplo de um conflito de dados RAW `lw-add` que é tratado como explicado na Secção 2. Figura 14 mostra o código de exemplo utilizado neste apêndice. Executa uma instrução `lw` seguida de 5 instruções `nop` e uma instrução `add` que depende da leitura. As instruções `nop` intermédias são incluídas para separar as duas instruções dependentes.

```
.globl Test_Assembly

.section .midccm
#.data
A: .space 4

.text
Test_Assembly:

# Register t3 is also called register 28 (x28)
la t0, A                # t0 = addr(A)
li t1, 0x1              # t1 = 1
sw t1, (t0)             # A[0] = 1
li t1, 0x0
li t3, 0x1
li t6, 0xFFFF

REPEAT:
    beq t6, zero, OUT    # Stay in the loop?
    INSERT_NOPS_8
    lw t1, (t0)
    INSERT_NOPS_5
    add t3, t3, t1        # t3 = t3 + t1
    INSERT_NOPS_8
    li t1, 0x0
    li t3, 0x1
    add t6, t6, -1
    j REPEAT
OUT:

.end
```

Figura 14. Programa que executa `lw`, 5 `nops` e uma instrução `add` dependente

Como é habitual, a pasta `[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_FarAway-LW-AL` fornece o projeto PlatformIO para que possa analisar, simular e modificar o programa como desejar. Abra o projeto, construa-o e abra o ficheiro Disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab15/DataHazards_FarAway-LW-AL/.pio/build/swervolf_nexys/firmware.dis`). Observe que as instruções `lw` e `add` são colocadas nos endereços `0x000001b0` e `0x000001c8`.

<code>0x000001b0:</code>	<code>0002a303</code>	<code>lw t1, 0(t0)</code>
<code>0x000001b4:</code>	<code>00000013</code>	<code>nop</code>
<code>0x000001b8:</code>	<code>00000013</code>	<code>nop</code>
<code>0x000001bc:</code>	<code>00000013</code>	<code>nop</code>
<code>0x000001c0:</code>	<code>00000013</code>	<code>nop</code>
<code>0x000001c4:</code>	<code>00000013</code>	<code>nop</code>
<code>0x000001c8:</code>	<code>006e0e33</code>	<code>add t3, t3, t1</code>

Figura 15 mostra a simulação do programa da Figura 14 na terceira iteração do ciclo. Mais uma vez, qualquer iteração seria válida, exceto a primeira, que deve ser evitada devido aos

misses na cache de instruções. Como nos exemplos do laboratório principal, os sinais na parte superior (Sinais de *trace*) ajudam a rastrear as instruções à medida que elas progridem no pipeline. Abaixo dos sinais de *trace*, são mostrados os sinais principais de cada multiplexer. Os sinais de cada multiplexer estão rodeados por linhas azuis tracejadas. O sinal de controlo, as entradas e a saída de cada multiplexer são ilustrados, tal como foi feito no laboratório principal.



Figura 15. Simulação do programa da Figura 14 na terceira iteração do ciclo

Figura 16 mostra um diagrama da execução do programa da Figura 14 no ciclo i (tal como definido na Figura 15), quando a instrução `add` está no andar Decode e a instrução `lw` está em DC3.

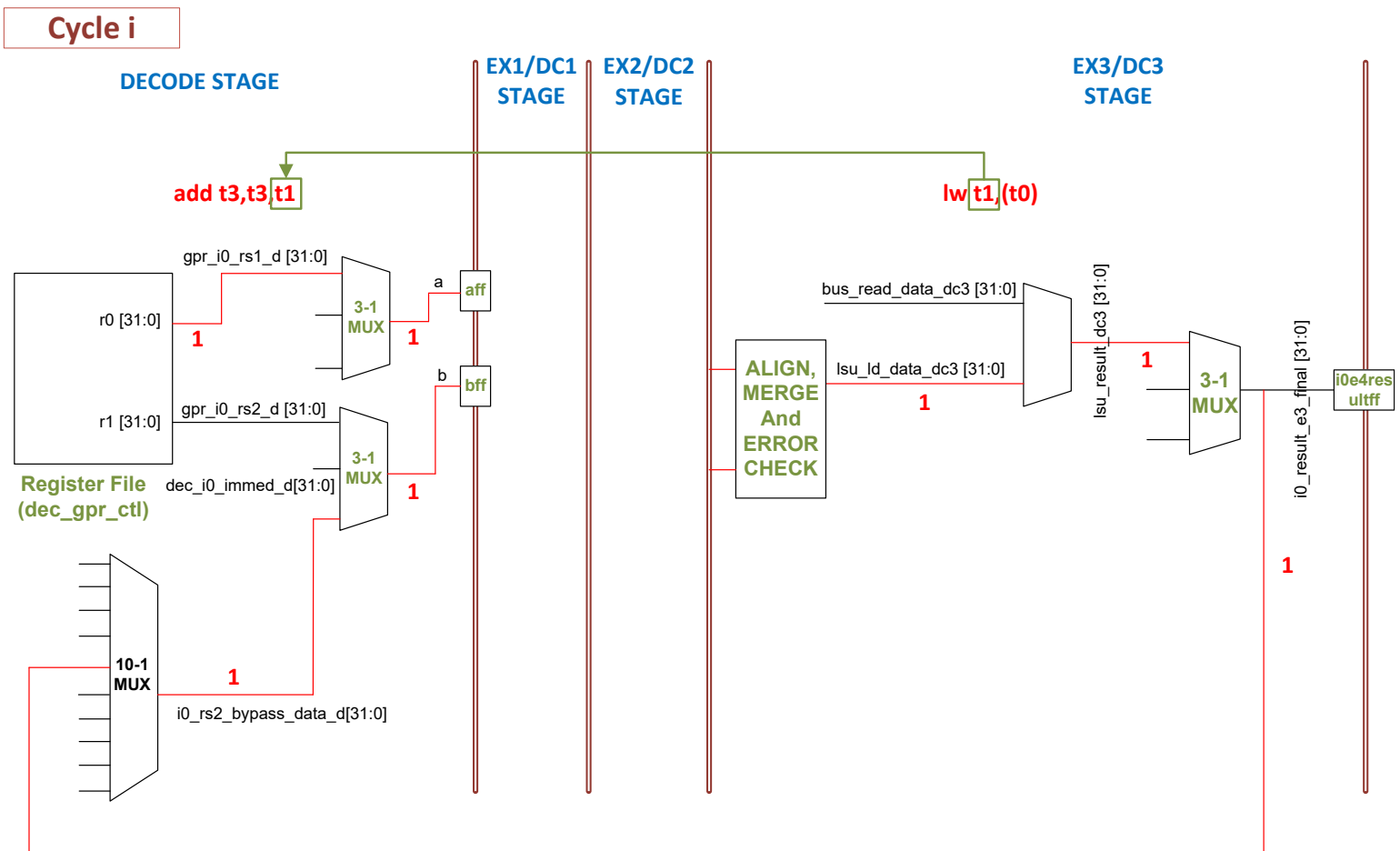


Figura 16. Hardware durante a execução do programa da Figura 14 na terceira iteração do ciclo e no quarto ciclo mostrado na Figura 15

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

Analisar a forma de onda da Figura 15 e o diagrama da Figura 16 em simultâneo.

- **Sinais de *trace* apresentados na Figura 15:**
 - o No ciclo *i*, a instrução `add` está no andar de Decode da Via 0 (`dec_i0_instr_d = 0x006E0E33`), e `lw` está no andar DC3 do pipe I0 (`i0_inst_e3 = 0x0002A303`).
- **Multiplexer 10:1:** No ciclo *i*, o sinal `i0_rs2bypass[9:0] = 0x010` (i.e. `i0_rs2bypass[4] = 1`), pelo que a saída é ligada ao valor proveniente do andar EX3/DC3 do pipe I0 (ver Figura 16):
`i0_rs2_bypass_data_d = i0_result_e3_final = 0x00000001`
- **Multiplexer 3:1:** No ciclo *i*, o sinal `dec_i0_rs2_bypass_en_d = 1`, pelo que a saída é ligada ao valor proveniente da lógica de bypass (ver Figura 9):
`i0_rs2_d = i0_rs2_bypass_data_d = 0x00000001`

TAREFA: Comparar como o cenário acima é tratado no SweRV EH1 e no processador em pipeline do DDCARV.

TAREFA: Se compararmos cuidadosamente Figura 16 e a Figura 6 do Lab 13, verá que o valor que a instrução `lw` lê no 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 ambos os valores reside no facto de o primeiro ter sido verificado pela lógica ECC no módulo `lsu_ecc`, enquanto o segundo não o foi. Explique por que razão não é problemático que o valor enviado pelo `lw` não seja verificado quanto a erros.

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