



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 14

Conflitos Estruturais

1. INTRODUÇÃO

Nos próximos três laboratórios, Labs 14-16, discutiremos **os conflitos do pipeline**. Como explicado por D. Patterson e J. Hennessy no Capítulo 4, Secção 5 do seu recente livro RISC-V (Computer Organization and Design RISC-V Edition, Patterson & Hennessy, © Morgan Kaufmann 2017) [PaHe]: Existem situações no pipelining em que a próxima instrução não pode ser executada no ciclo de relógio seguinte. Estes eventos são designados por conflitos. Ocorrem três tipos diferentes de conflitos: **conflitos estruturais**, conflitos de dados e conflitos de controlo.

Tal como explicado por Patterson e Hennessy em [PaHe], **os conflitos estruturais** ocorrem quando uma instrução não pode ser executada porque o hardware não suporta a combinação de instruções que estão definidas para serem executadas. Neste laboratório, analisamos os conflitos estruturais no processador SweRV EH1.

No Lab 15 analisamos o segundo tipo de conflito, **os conflitos de dados**, no processador SweRV EH1. Como explicado por Hennessy e Patterson na edição de 6th do seu livro "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 que a ordem difere da ordem observada na execução sequencial de instruções num processador sem pipeline.

Por último, o terceiro tipo de conflito é designado por **conflito de controlo**. Tal como explicado por S. Harris e D. Harris na secção 7.5.3 do seu livro "*Digital Design and Computer Architecture: RISC-V Edition*" (a que chamamos DDCARV), um conflito de controlo ocorre quando a decisão sobre qual a instrução a ir buscar a seguir não foi tomada no momento em que é efetuada. No Lab 16 analisamos os conflitos de controlo no processador SweRV EH1.

2. CONFLITOS ESTRUTURAIS NO SweRV EH1

Nesta secção, ilustramos dois casos de conflitos estruturais que podem ocorrer no processador SweRV EH1. Cada um deles é resolvido de uma forma diferente, resultando assim num compromisso diferente entre desempenho e custo.

Para ilustrar a primeira situação, criamos um exemplo na Secção 2.A baseado na instrução de multiplicação de números inteiros (`mul`). Ao mesmo tempo, descrevemos a execução desta instrução no SweRV EH1, que ainda não analisámos em laboratórios anteriores. Recorde-se das secções 3 e 4 das GSG que esta instrução pertence à extensão RISC-V M (Standard Extension for Integer Multiplication and Division), que é suportada no SweRV EH1. Para executar esta instrução, o processador SweRV EH1 implementa uma unidade de multiplicação multi-ciclo em pipeline (ou seja, um multiplicador em pipeline que necessita de mais do que um ciclo para calcular o resultado) no pipe Multiply (ver Figura 4 do Lab 11), dividido em três andares: M1, M2 e M3.

Especificamente, neste exemplo, duas instruções `mul` chegam à fase de decodificação no mesmo ciclo. Como o SweRV EH1 tem apenas uma unidade de multiplicação, ocorre um conflito estrutural, uma vez que o processador "não suporta a combinação de instruções que estão definidas para execução" [PaHe]. Num processador que utilize um multiplicador multi-ciclo não-pipelined, a segunda instrução `mul` teria de esperar que a primeira terminasse a sua execução, o que teria um impacto importante no desempenho. No entanto, como já foi referido, o multiplicador utilizado no SweRV EH1 é pipelined, pelo que a

segunda instrução `mul` só é atrasada um ciclo e começa a ser executada assim que a primeira instrução de multiplicação termina a primeira fase da multiplicação (M1) e avança para a segunda fase (M2). Esta solução tem um impacto moderado no custo do hardware (uma estrutura em pipeline é mais cara do que uma não-pipelined), mas resolve o conflito estrutural com um impacto reduzido no desempenho (apenas um ciclo).

No segundo exemplo (Secção 2.B), três instruções chegam ao andar de Writeback no mesmo ciclo, sendo uma delas uma carga não bloqueante executada vários ciclos antes. Em princípio, como o SweRV EH1 é um núcleo superescalar de 2 vias, não seria possível completar três instruções no mesmo ciclo; no entanto, como mostrámos no Lab 11, o ficheiro de registo do SweRV EH1 tem uma terceira porta de escrita, o que evita o conflito estrutural nesta situação. Esta solução tem um impacto elevado no custo do hardware devido à porta extra do Register File, mas resolve este conflito estrutural sem perda de desempenho.

APÊNDICE A - Duas instruções `div` simultâneas no andar de Decode: Para além destes dois exemplos, no apêndice no final deste laboratório ilustramos outro exemplo baseado em instruções de divisão. Embora este exemplo não ilustre propriamente um conflito estrutural, não deixa de ser muito interessante e recomendamos que o analise também.

A. Duas instruções `mul` simultâneas na fase de decodificação

A extensão M do RISC-V inclui, entre outras, a instrução `mul`. Esta instrução executa a multiplicação de `rs1` por `rs2` e coloca os bits inferiores no registo de destino (`rd`). A instrução em linguagem de máquina para `mul` é a seguinte (ver Apêndice B de [DDCARV]):

```
0000001 | rs2 | rs1 | 000 | rd | 0110011
```

TAREFA: Pode realizar para a instrução `mul` um estudo semelhante ao realizado no Lab 12 para as instruções aritméticas-lógicas: ver o fluxo da instrução através das etapas do pipeline, analisar os bits de controlo (lembre-se da Secção 4 do SweRVref que existe um tipo de estrutura específico para a instrução `mul` chamado `mul_pkt_t`, e que existe um sinal definido no módulo `dec_decode_ctl` chamado `mul_p`), etc.

A unidade de multiplicação é implementada no módulo `exu_mul_ctl` (`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu_mul_ctl.sv`). Como já foi referido, esta unidade está em pipeline e necessita de 3 ciclos para calcular o resultado. A utilização de um multiplicador em pipeline - por oposição a um não-pipelined - reduz a perda de desempenho devido a conflitos estruturais.

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

Como exercício opcional, pode substituir a unidade de multiplicação pela sua própria unidade ou por outra da Internet.

O exemplo da Figura 1 executa duas instruções `mul` contidas num ciclo que se repete durante 0xFFFF iterações (ou seja, 65.535 em decimal). As instruções `mul` estão destacadas a vermelho na figura. Neste caso, as instruções `mul` estão rodeadas por várias instruções `nop` para isolar cada iteração uma da outra. Como de costume, o programa não faz nada de útil e tem apenas o objetivo de ilustrar os *conflitos estruturais* devidos às instruções `mul`.

```
.globl Test_Assembly
Test_Assembly:

li t2, 0xFFFF

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

REPEAT:
    beq t2, zero, OUT    # Permanece no ciclo?
    INSERT_NOPS_9
    mul t0, t3, t4        # t0 = t3 * t4
    mul t1, t5, t6        # t1 = t5 * t6
    INSERT_NOPS_9
    add t2, t2, -1
    add t0, zero, zero
    add t1, zero, zero
    j REPEAT
OUT:

.end
```

Figura 1. Exemplo com duas instruções `mul` consecutivas

A pasta `[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instruction` fornece o projeto PlatformIO para que possa analisar, simular e modificar o programa como desejar. A estrutura do projeto é baseada na estrutura fornecida no Lab 11 para a utilização dos contadores de desempenho: contém um ficheiro `.c` que inicializa, pára e imprime o valor dos contadores desejados e um ficheiro `.S` que contém o programa Assembly que queremos testar (neste caso, o ciclo com as duas instruções `mul`) e que é invocado a partir do ficheiro `.c`.

Abra o projeto no PlatformIO, compile-o e abra o ficheiro de Disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instruction/.pio/build/swervolf_nexys/firmware.dis`). Observe que as instruções `mul` são colocadas nos endereços 0x000001e8 e 0x000001ec.

```
0x000001e8:    03de02b3    mul t0,t3,t4
0x000001ec:    03ff0333    mul t1,t5,t6
```

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

Figura 2 mostra a simulação do programa da Figura 1 na segunda iteração do ciclo.

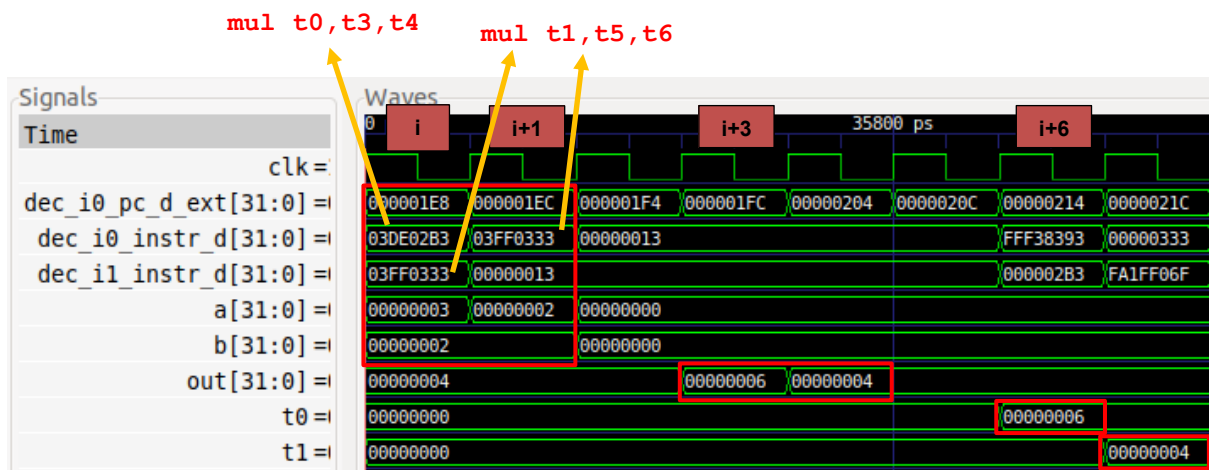


Figura 2. Simulação do Verilator do exemplo da Figura 1

TAREFA: Replicar a simulação da Figura 2 no seu próprio computador e analise-a mais detalhadamente. Pode utilizar o ficheiro .tcl
 [RVfpgaPath]/RVfpga/Labs/Lab14/MUL_Instruction/test.tcl

Analisar a forma de onda da Figura 2. Os valores destacados a vermelho correspondem a diferentes sinais relacionados com as instruções `mul` à medida que estas atravessam o pipeline.

- **Ciclo i:** As duas instruções `mul` chegam no mesmo ciclo ao andar de Decode. Um conflito estrutural impede que a segunda instrução `mul` (`dec_i1_instr_d = 0x03ff0333`) avance para o andar seguinte, dado que a primeira instrução `mul` (`dec_i0_instr_d = 0x03de02b3`) está programada para essa unidade.
- **Ciclo i+1:** A primeira instrução `mul` é executada na primeira fase do multiplicador em pipeline (M1), enquanto a segunda instrução `mul` aguarda no andar de Decode.
- **Ciclo i+2:** A primeira instrução `mul` é executada no segundo andar do multiplicador em pipeline (M2) e a segunda `mul` é executada no primeiro andar (M1).
- **Ciclo i+3:** A primeira instrução `mul` chega a EX3, quando o resultado da multiplicação é produzido (`out = 0x6` para a primeira instrução `mul`).
- **Ciclo i+4:** A segunda instrução `mul` chega a EX3, quando o resultado da multiplicação é produzido (`out = 0x4` para a segunda instrução `mul`).
- **Ciclo i+6:** O ficheiro de registo é atualizado com o resultado da primeira instrução `mul` (`t0 = 0x6`).
- **Ciclo i+7:** O ficheiro de registo é atualizado com o resultado da segunda instrução `mul` (`t1 = 0x4`).

A Figura 3 ilustra o fluxo das instruções do exemplo da Figura 1 através do pipeline SweRV EH1. **D** significa o andar de Decode, **A** significa o andar de alinhamento, **C** significa o andar de Commit e **WB** significa o andar de Writeback. Quando a primeira instrução `mul` é

descodificada (ciclo i), a maioria das instruções subsequentes pára na sua fase atual (assinalada na figura com o sufixo *st*) e são inseridas bolhas. No ciclo seguinte (i+1) as instruções são retomadas (note-se que a segunda instrução *mul* foi movida da Via-1 para a Via-0, e a Via-1 contém a instrução seguinte, que é um *nop*). No ciclo i+2 a primeira instrução *mul* está na segunda fase de execução (M2) e a segunda instrução *mul* está na primeira fase de execução (M1). Nos ciclos i+5 e i+6, as duas instruções *mul* escrevem o seu resultado de volta no ficheiro de registo, que pode ser visto atualizado na Figura 2 nos ciclos i+6 e i+7.

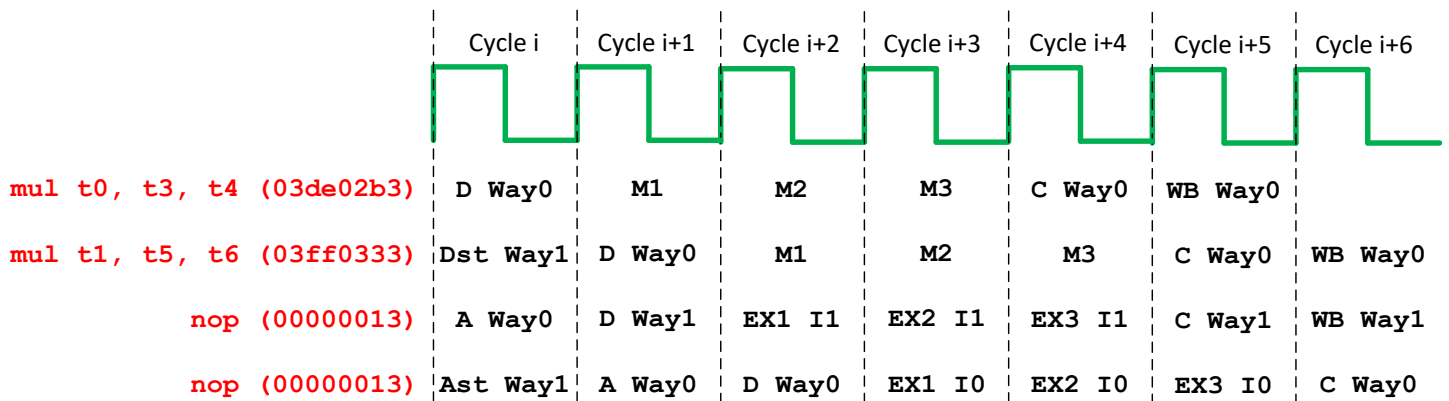


Figura 3. Execução da Figura 1 código de exemplo

TAREFA: Comparar a ilustração da Figura 3 com a simulação da Figura 2 concentrando-se nas duas instruções *mul*. Especificamente, analise a forma como as duas instruções são atribuídas às duas vias nas fases de Alinhamento e Descodificação e como progridem através do pipeline.

- No módulo **ifu_aln_ctl** (andar Align), as duas instruções são atribuídas aos seguintes sinais, sempre que possível:

- Via 0: *ifu_i0_instr*
- Via 1: *ifu_i1_instr*

- No módulo **dec_ib_ctl**, as duas instruções são colocadas num buffer desde o Align até ao Decode. Note-se que, nalguns casos, as instruções podem ser bloqueadas nestes buffers e reatribuídas a uma via diferente:

- Via 0: *ifu_i0_instr* → *dec_i0_instr_d*
- Via 1: *ifu_i1_instr* → *dec_i1_instr_d*

- No módulo **dec_decode_ctl** (andar de Decode), as duas instruções são programadas para os pipes correspondentes, sempre que possível. Uma vez enviadas, continuam a passar pelos três andares de execução, o andar de Commit e o de 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 estes sinais.

TAREFA: Remover as instruções *nop* incluídas no ciclo da Figura 1 e meça diferentes eventos (ciclos, instruções/multiplicações cometidas, etc.) usando os Contadores de Desempenho disponíveis no SweRV EH1, como explicado no Lab 11. O número de ciclos está de acordo com o esperado depois de analisar a simulação de Figura 2? Justifique a sua resposta.

Agora reordene o código dentro do loop tentando atingir o débito ideal. Justifique os resultados obtidos no código original e no código reordenado.

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

- Analisar o programa C.
- Faça uma simulação e inspecione uma iteração aleatória do ciclo. Note que o programa C é compilado sem optimizações.
- Meça diferentes eventos (ciclos, instruções/multiplicações efectuadas, etc.) utilizando os contadores de desempenho disponíveis no Swe RV EH1, como explicado no Lab 11.
O número de ciclos é o esperado após a análise da simulação da Figura 2? Justifique a sua resposta.
- Crie um programa análogo em Assembly RISC-V e compare-o com a versão C. Reordene as instruções tentando obter o melhor IPC possível.

- Desativar a extensão **M** RISC-V no programa C e comparar os resultados com o programa original. Para isso, modificar a seguinte linha no arquivo `platformio.ini` de:

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

Para:

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

Isto evita a utilização das instruções da extensão M RISC-V e emula-as utilizando outras instruções.

TAREFA: Modificar o programa da Figura 1 substituindo as duas instruções `mul` por duas instruções `lw` para o DCCM. Deverá observar um conflito estrutural análogo ao analisado nesta secção e resolvido de forma semelhante.

B. Três instruções em execução simultânea no Writeback

O SweRV EH1 é um processador superescalar de 2 vias (discutimos brevemente esta característica no GSG e em laboratórios anteriores, e iremos analisá-la em mais pormenor no Lab 17). Isto significa que duas instruções podem ser executadas neste processador por ciclo. Numa situação em que três instruções chegassem à mesma fase no mesmo ciclo, poderia ocorrer um conflito estrutural. Pode parecer que tal situação não é possível dada a estrutura do SweRV EH1, no entanto, há um caso específico em que tal pode acontecer:

- A Memória DDR2 Externa tem uma latência moderada que força as instruções de leitura a pararem. Quando a leitura eventualmente recebe os seus dados da memória, prossegue para o andar de Writeback, onde escreve o valor lido no ficheiro de registo (vamos assumir que este Writeback acontece no ciclo *i*).
- Se as leituras forem não-bloqueantes (ou seja, enquanto a leitura está à espera que os dados cheguem da memória, o processador continua a executar instruções que não dependem desses dados), pode acontecer que duas outras instruções cheguem à fase de Writeback no ciclo *i* e também precisem de escrever no ficheiro de registo

(por exemplo, duas instruções `add`).

- Nesta situação, três instruções estariam a tentar escrever no ficheiro de registo no mesmo ciclo (ciclo *i*).

Se o ficheiro de registo tivesse apenas dois portos de escrita, ocorreria um conflito estrutural e uma das três instruções que tentasse escrever teria de esperar que o ficheiro de registo ficasse livre. No entanto, no SweRV EH1, como mostrámos no Lab 11, é implementada uma terceira porta de escrita, o que permite que este conflito estrutural seja resolvido sem paragens e, portanto, sem perda de desempenho.

O exemplo da Figura 4 ilustra esta situação. Executa uma instrução `lw` não bloqueante seguida de 36 instruções `add` contidas num ciclo que se repete durante 0xFFFF iterações (ou seja, 65.535). A instrução `lw` está destacada a vermelho na figura. As duas instruções `add`, que chegam ao andar de Writeback no mesmo ciclo que a instrução `lw` (ciclo *i*), também estão destacadas. Neste caso, as instruções `nop` não estão incluídas. Como é habitual, o programa não faz nada de útil e destina-se apenas a ilustrar o exemplo desta secção.

```
REPEAT:
    lw x28, (x29)
    add x30, x30, -1
    add x1, x1, 1
    add x31, x31, 1
    add x3, x3, 1
    add x4, x4, 1
    add x5, x5, 1
    add x6, x6, 1
    add x7, x7, 1
    add x8, x8, 1
    add x9, x9, 1
    add x10, x10, 1
    add x11, x11, 1
    add x12, x12, 1
    add x13, x13, 1
    add x14, x14, 1
    add x15, x15, 1
    add x16, x16, 1
    add x17, x17, 1
    add x18, x18, 1
    add x19, x19, 1
    add x20, x20, 1
    add x21, x21, 1
    add x22, x22, 1
    add x23, x23, 1
    add x24, x24, 1
    add x25, x25, 1
    add x26, x26, 1
    add x27, x27, 1
    add x31, x31, 1
    add x3, x3, 1
    add x4, x4, 1
    add x5, x5, 1
    add x6, x6, 1
    add x25, x25, 1
    add x26, x26, 1
    add x27, x27, 1
    bne x30, zero, REPEAT    # Repeat the loop
```

Figura 4. Exemplo de uma instrução `lw` não-bloqueante seguida de 36 instruções A-L

A pasta *[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory* fornece o projeto PlatformIO para que possa analisar, simular e modificar o programa como desejar. A estrutura do projeto é baseada na estrutura fornecida no Lab 11 para a utilização dos contadores de desempenho: contém um ficheiro .c que inicializa, pára e exhibe o valor dos contadores desejados e um ficheiro .S que contém o programa Assembly que queremos testar (neste caso, o ciclo com a instrução `lw` não bloqueante) e que é invocado a partir do ficheiro .c.

Como mostra a Figura 5 os dados de 32 bits obtidos no módulo **lsu_bus_intf** (Bus Interface) são fornecidos ao ficheiro de registos através do sinal `lsu_nonblock_load_data[31:0]`. Além disso, os sinais de controlo que dizem ao Register File onde escrever esses dados e quando escrevê-los, que foram gerados no andar de Decode e propagados através dos Registos do Pipeline, são fornecidos ao ficheiro de registo através dos sinais `dec_nonblock_load_waddr[4:0]` e `dec_nonblock_load_wen`, respetivamente. Estes três sinais vão para o ficheiro de registo através do terceiro porto de escrita disponível nesta estrutura (`waddr2`, `wen2` e `wd2`), como ilustrado na figura. Lembre-se que na Figura 6 do Lab 11 ilustrámos o Register File em detalhe.

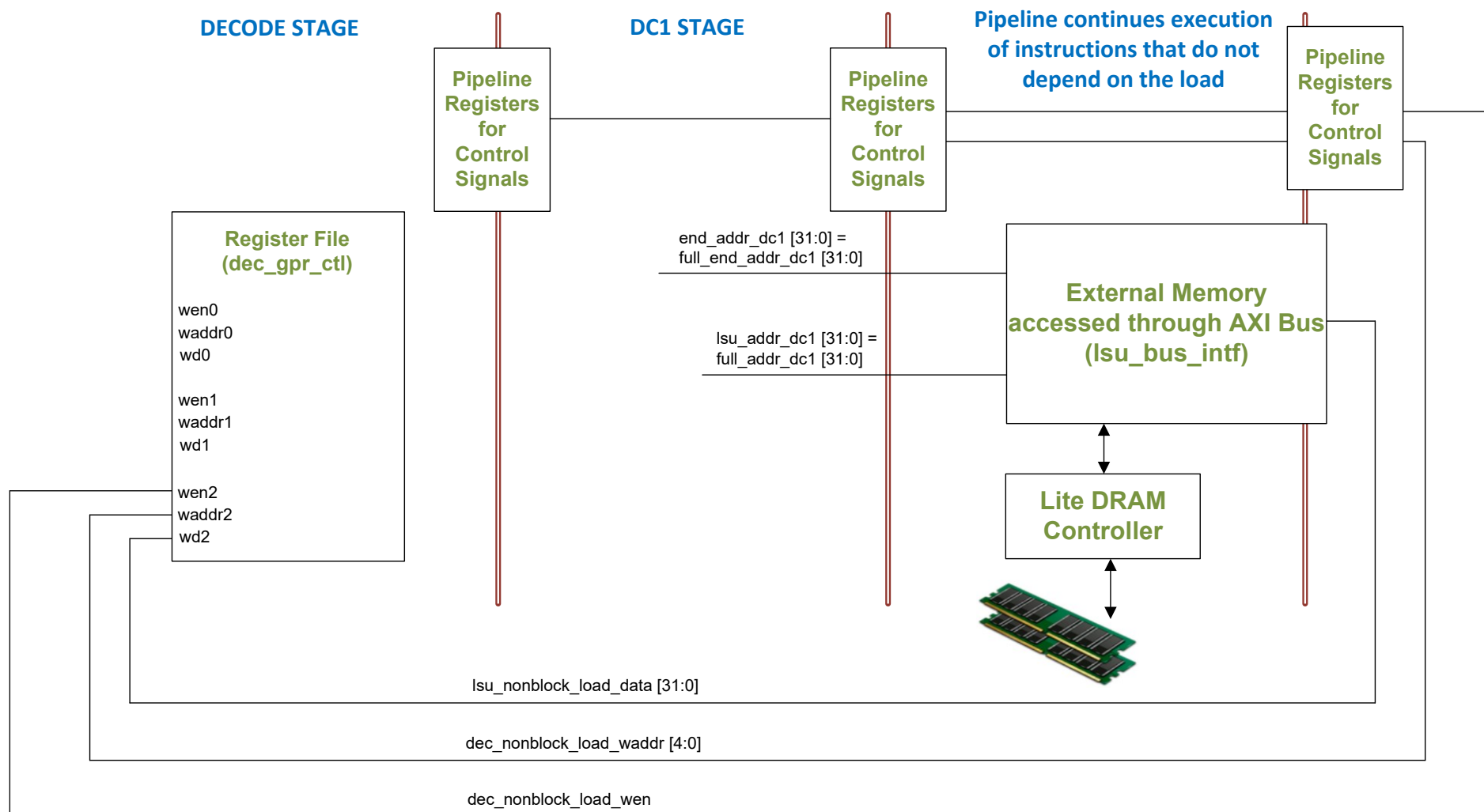


Figura 5. Instrução de leitura não-bloqueante que acede à memória externa

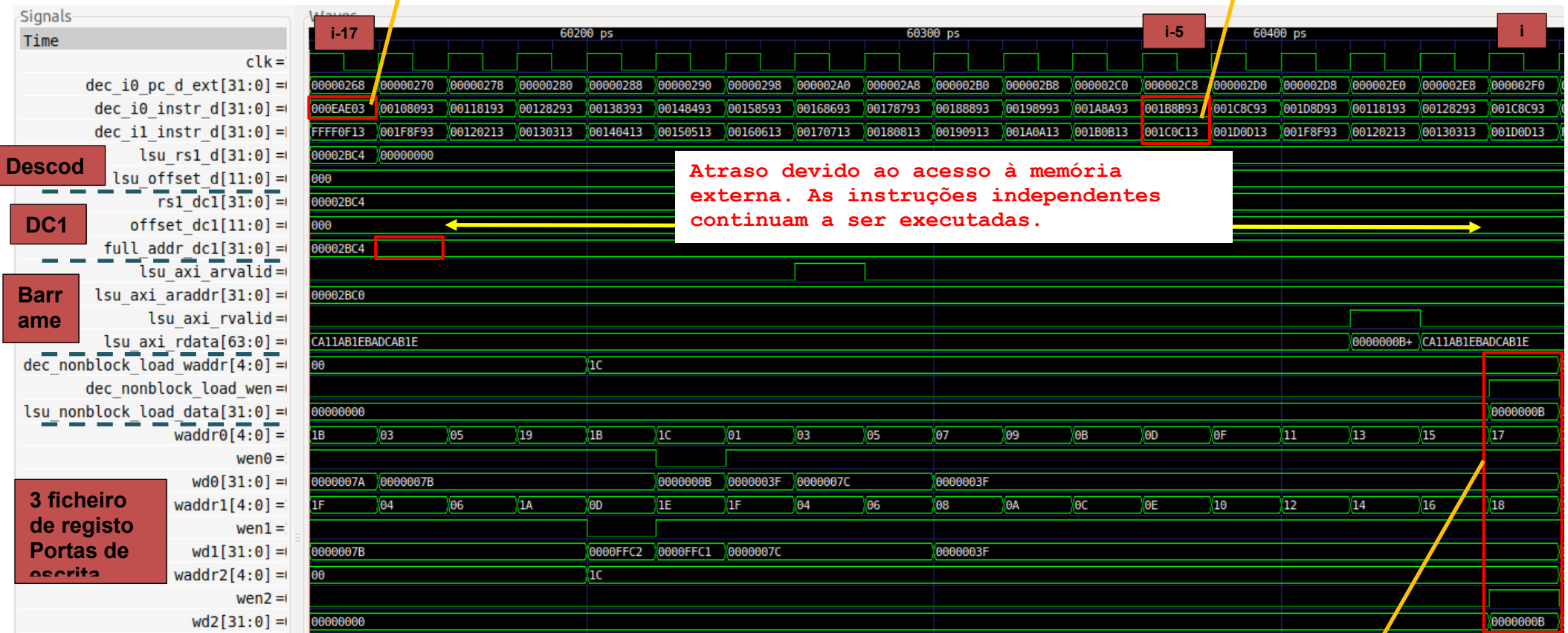


Figura 6. Simulação do Verilator para o exemplo da Figura 4

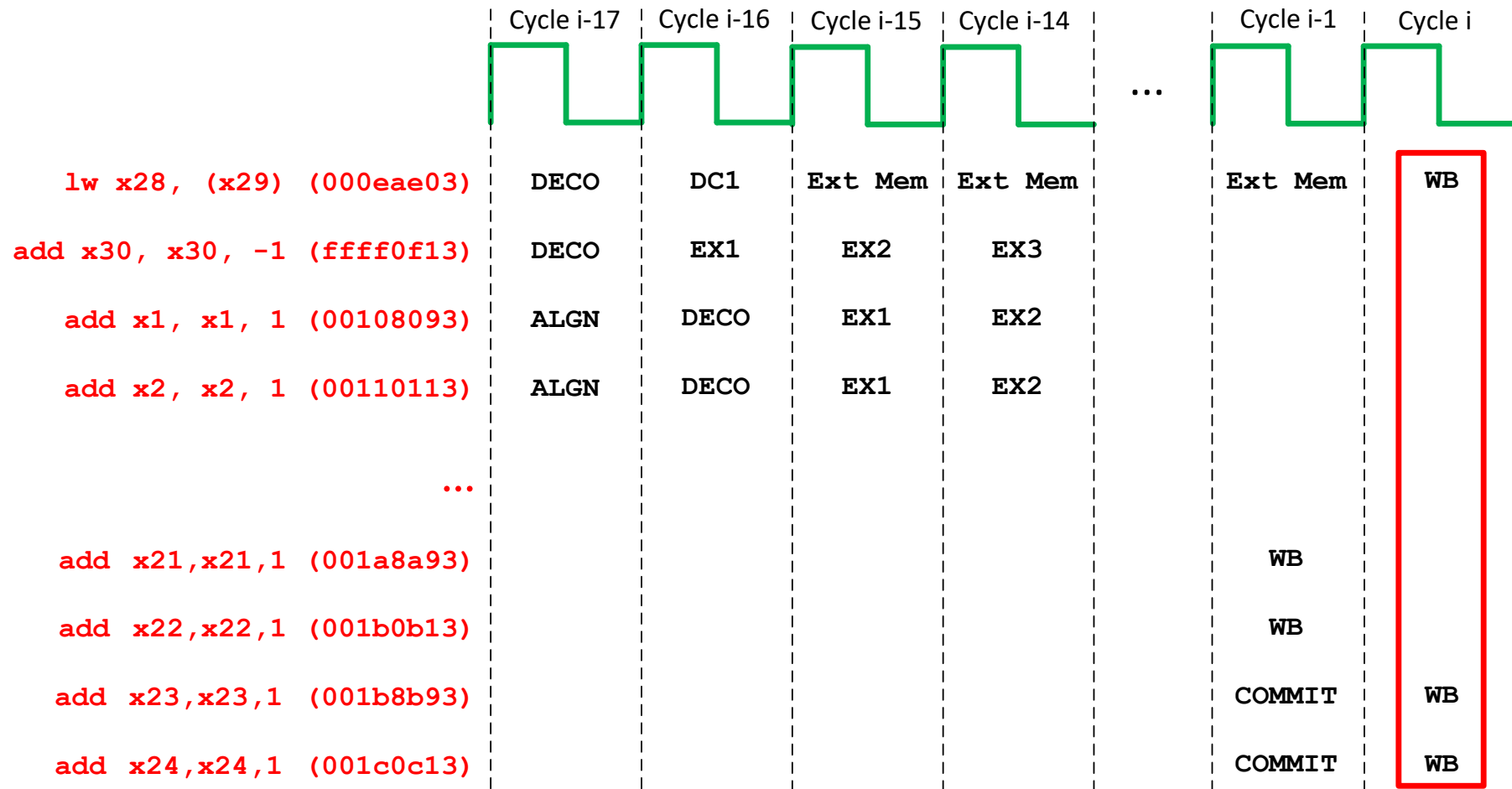



Figura 7. Execução do código de exemplo da Figura 4

Figura 6 e Figura 7 mostram a simulação do Verilator para o programa da Figura 4 e um diagrama que ilustra a execução deste programa para uma iteração aleatória do ciclo.

TAREFA: Replicar a simulação da Figura 6 no seu próprio computador. Utilize o ficheiro `test_NonBlocking.tcl` (fornecido em

`[RVfpgaPath]/RVfpga/Labs/Lab14/LW_Instruction_ExtMemory`). Aumentar o zoom () várias vezes e passar para 60120ps.

Analisar a forma de onda da Figura 6 e o diagrama da Figura 7.

- **Ciclo i-17:** A instrução `lw` está no andar de Decode.
- **Ciclo i-16:** O endereço de memória efetivo é calculado e enviado para a memória externa através do barramento AXI. A latência da memória externa obriga a instrução de leitura a esperar vários ciclos para que os dados cheguem ao núcleo.
- **Ciclo i-5:** As duas instruções `add` em conflito são decodificadas.
- **Ciclo i:** A instrução `lw` e as duas instruções `add` em conflito seguem para o andar de Writeback, onde todas elas devem escrever o ficheiro de registos. Isto é possível graças às três portas de escrita disponíveis no ficheiro de registos do SweRV EH1. Note que os números dos registos são mostrados em hexadecimal na simulação. `x23`, `x24` e `x28` (registos `0x17`, `0x18` e `0x1c`) estão a ser escritos.

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

TAREFA: Comparar a ilustração da Figura 7 com a simulação da Figura 6 que reproduziu no seu próprio computador. Acrescente sinais para alargar a simulação e aprofundar a compreensão, conforme pretendido.

TAREFA: Medir diferentes eventos (ciclos, instruções/leituras efetuadas, etc.) usando os contadores de desempenho disponíveis no SweRV EH1, como explicado no Lab 11. O número de ciclos está de acordo com o esperado depois de analisar a simulação da Figura 6? Justifique a sua resposta. Compare estes resultados com os obtidos quando as cargas são configuradas como leituras bloqueantes.

3. EXERCÍCIOS

1. Analisar, tanto em 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 loads e stores) que chegam ao pipe L/S no mesmo ciclo. Pode utilizar o projeto PlatformIO fornecido em: [\[RVfpgaPath\]/RVfpga/Labs/Lab14/TwoConsecutiveLW_Instructions](#).

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

Considere o fragmento de montagem RISC-V abaixo:

```
sw x29, 12(x16)
lw x29, 8(x16)
submarino x17, x15, x14
beqz x17, etiqueta
adicionar x15, x11, x14
sub x15, x30, x14
```

Suponhamos que modificamos o processador SweRV EH1 de modo a que tenha apenas uma memória (que gere tanto instruções como dados). Neste caso, haverá um conflito estrutural sempre que um programa precisar de ir buscar uma instrução durante o mesmo ciclo em que outra instrução acede a dados.

- a. Desenhe um diagrama de pipeline para mostrar onde o código acima irá parar nesta versão imaginária do processador SweRV EH1.
- b. Em geral, é possível reduzir o número de paragens/nops reordenando o código?
- c. Este conflito estrutural tem de ser tratado no hardware? Vimos que os conflitos de dados podem ser eliminados adicionando nops ao código. É possível fazer o mesmo com este conflito estrutural? Em caso afirmativo, explique como. Se não, explique porquê.

APÊNDICE A - DUAS INSTRUÇÕES `div` EM SIMULTÂNEO NO ANDAR DE DECODE

Este exemplo adicional é baseado na instrução de divisão de números inteiros (`div`). Tal como a instrução `mul`, a instrução `div` pertence à extensão RISC-V M (Standard Extension for Integer Multiplication and Division), que é suportada no SweRV EH1.

A instrução `div` efetua a divisão inteira assinada de `rs1` por `rs2` e armazena o resultado em `rd`. A instrução em linguagem de máquina para `div` é (ver Apêndice B de [DDCARV]):

```
0000001 | rs2 | rs1 | 100 | rd | 0110011
```

TAREFA: Pode realizar para a instrução `div` um estudo semelhante ao realizado no Lab 12 para as instruções aritmético-lógicas: ver o fluxo da instrução através das etapas do pipeline, analisar os bits de controlo (lembre-se da Secção 4 do SweRVref que existe um tipo de estrutura específico para a instrução `div` chamado `div_pkt_t`, e que existe um sinal definido no módulo `dec_decode_ctl` chamado `div_p`), etc.

Para executar esta instrução, o processador SweRV EH1 implementa uma unidade de divisão multi-ciclo de bloqueio não-pipelined no módulo `exu_div_ctl` (`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu_div_ctl.sv`). Esta unidade necessita de até 34 ciclos para computar o resultado, no entanto, dependendo das entradas, pode ser muito menor. A unidade de divisão emite vários sinais para o processador (`div_stall`, `finish_early`, `finish`) para indicar o estado de uma instrução de divisão.

TAREFA: Inspeccionar o código Verilog do `exu_div_ctl` para perceber como é calculada a divisão. Analise também o efeito dos sinais `div_stall`, `finish_early` e `finish`. Como exercício opcional, substitua a unidade de divisão pela sua própria unidade ou por outra da Internet.

O exemplo da Figura 8 executa duas instruções `div` contidas num ciclo que se repete durante 0xFFFF iterações (ou seja, 65.535 em decimal). As instruções `div` estão destacadas a vermelho na figura. Neste caso, ao contrário de muitos outros exemplos, as instruções `nop` não são necessárias, pois as instruções `div` já estão isoladas de qualquer outra instrução devido à alta latência da unidade de divisão. Como nos programas de exemplo anteriores que usámos, o programa não faz nada de útil.

```
.globl Test_Assembly
Test_Assembly:

li t2, 0xFFFF

li t3, 0x8000000
li t4, 0x2
li t5, 0x2000000
li t6, 0x2

REPEAT:
    div t0, t3, t4      # t0 = t3 / t4
    div t1, t5, t6      # t1 = t5 / t6
    add t2, t2, -1
    add t0, zero, zero
    add t1, zero, zero
```

```
bne t2, zero, REPEAT # repeat the loop
.end
```

Figura 8. Exemplo de duas instruções `div` consecutivas

A pasta `[RVfpgaPath]/RVfpga/Labs/Lab14/DIV_Instruction` fornece o projeto PlatformIO para que possa analisar, simular e alterar o programa como desejar. A estrutura do projeto é semelhante à utilizada para a instrução `mul` e baseia-se na estrutura incluída no Lab 11 para a utilização dos contadores de desempenho.

Se abrir o projeto no PlatformIO, construí-lo e abrir o ficheiro de *Disassembly* (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab14/DIV_Instruction/.pio/build/swervolf_nexys/firmware.dis`), verá que as instruções `div` são colocadas nos endereços `0x000001c0` e `0x000001c4`.

```
0x000001c0:    03de42b3    div t0,t3,t4
0x000001c4:    03ff4333    div t1,t5,t6
```

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

A Figura 9 mostra a simulação do programa da Figura 8 numa iteração aleatória do ciclo.

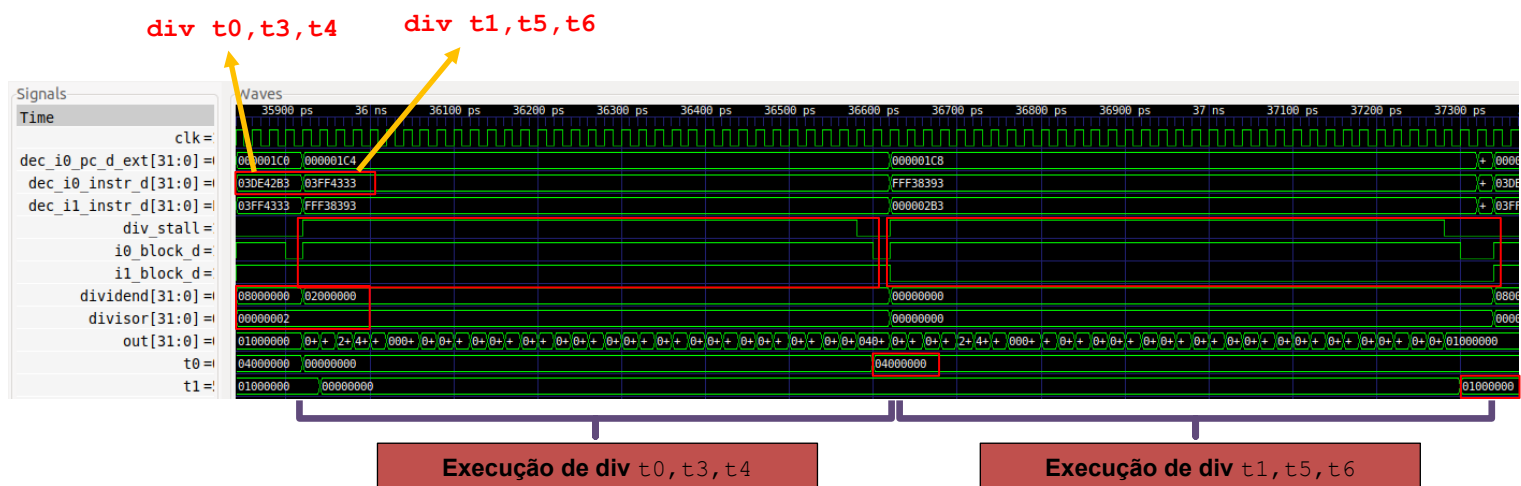


Figura 9. Simulação do Verilator para o exemplo da Figura 8

TAREFA: Replicar a simulação da Figura 9 no seu próprio computador e analisá-la em pormenor.

Analisar a forma de onda da Figura 9. Os valores destacados a vermelho são sinais relacionados com as duas instruções `div` à medida que atravessam o pipeline.

- As duas instruções `div` chegam ao andar de decodificação no mesmo ciclo (`dec_i0_pc_d_ext` = `0x000001c0`, que é o endereço de instrução da primeira `div`). A primeira instrução `div` (`0x03de42b3`) está programada para ser executada na unidade de divisão, pelo que envia o dividendo e o divisor (`dividend` = `0x08000000` e `divisor` = `0x00000002`) para esta unidade. Note-se que

selecionámos valores elevados para os dividendos para que o tempo de cálculo da divisão seja próximo do máximo (34 ciclos).

Dado que a divisão é bloqueante no SweRV EH1, qualquer outra instrução após o `div` é bloqueada. Note-se, no entanto, que, mesmo que a divisão não fosse bloqueante, um conflito estrutural devido ao facto de haver apenas um divisor faria com que a segunda instrução `div` ficasse bloqueada. Tal como explicado na Secção 2, haveria outras abordagens para melhorar o desempenho, tais como a colocação do divisor em pipelines ou a inclusão de outro. No entanto, dado que a divisão não é uma operação frequente, a redução dos custos de hardware prevalece neste caso.

- O pipeline é bloqueado durante a execução da primeira instrução `div` (ver sinal `div_stall = 1` durante o cálculo da primeira divisão). Também se pode ver que tanto a `Via-0` como a `Via-1` estão bloqueados com os sinais `i0_block_d` e `i1_block_d` a serem 1. Além disso, agora `dec_i0_pc_d_ext = 0x000001c4`, que é o endereço da segunda instrução de divisão, que está bloqueada no andar de decodificação.
- O sinal `out` da unidade de divisão fornece o resultado após 34 ciclos, que é escrito no registo de destino (`t0 = 0x04000000`). Pode ver como o valor de saída muda a cada ciclo à medida que a operação de divisão se transforma sucessivamente no resultado final.
- Quando o resultado é obtido, o divisor é libertado, o pipeline é autorizado a continuar (`div_stall = 0`) e a segunda instrução `div` é programada para a unidade de divisão. Depois, 34 ciclos mais tarde, o resultado da segunda instrução `div` é escrito no ficheiro de registos (`t1 = 0x01000000`).

Tal como na primeira instrução `div`, todas as instruções após a segunda `div` têm de parar devido ao bloqueio do divisor. Neste caso, no entanto, as instruções que não dependem de `t1` poderiam continuar, caso se tratasse de uma divisão não bloqueante.

A Figura 10 ilustra o fluxo das instruções no exemplo da Figura 8 através do pipeline SweRV EH1. Quando a primeira instrução `div` é decodificada (ciclo `i`), a segunda `div` e as instruções subsequentes ficam bloqueadas na sua fase atual, devido à divisão bloqueante da SweRV EH1 e ao conflito estrutural na unidade de divisão. (Uma instrução parada é marcada na figura com o sufixo `-st`.) Depois, 34 ciclos mais tarde (ciclo `i+34`), a primeira instrução `div` termina a execução e escreve o resultado no ficheiro de registos através do multiplexer 2:1 que foi mostrado na Figura 4 do Lab 11. No ciclo seguinte (`i+35`), as instruções subsequentes são retomadas. Depois, no ciclo 36, a segunda instrução `div` inicia a execução e as instruções subsequentes ficam novamente paradas devido ao bloqueio da divisão da SweRV EH1.

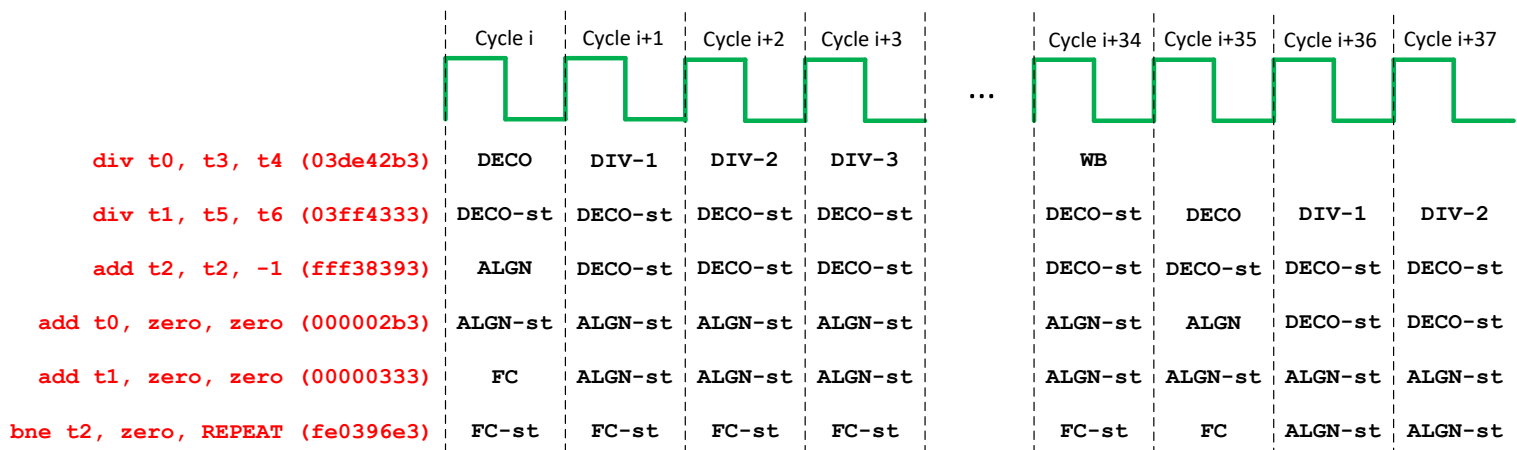


Figura 10. Execução da Figura 8 código de exemplo (o sufixo *-st* indica uma instrução bloqueada)

TAREFA: Comparar a ilustração da Figura 10 e a simulação da Figura 9 que replicou no seu próprio computador. Acrescente sinais para alargar a simulação e aprofundar a compreensão, conforme pretendido.

TAREFA: Medir diferentes eventos (ciclos, instruções/divisões executadas, etc.) usando os Contadores de Desempenho disponíveis no SweRV EH1, como explicado no Lab 11. O número de ciclos é o esperado após a análise da simulação da Figura 9? Justifique a sua resposta.

TAREFA: Experimentar diferentes dividendos e divisores e ver como o número de ciclos para calcular o resultado depende do seu valor. Ver a experiência tanto em simulação como com os contadores HW.

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

- Analisar o programa C.
- Faça uma simulação e inspecione uma iteração aleatória do ciclo. Note que o programa C é compilado sem otimizações.
- Meça diferentes eventos (ciclos, instruções/divisões executadas, etc.) utilizando os contadores de desempenho disponíveis no SweRV EH1, como explicado no Lab 11. O número de ciclos é o esperado após a análise da simulação da Figura 9? Justifique a sua resposta.
- Criar um programa análogo em Assembly RISC-V e compará-lo com a versão C.
- Desativar a extensão **M** RISC-V no programa C e comparar os resultados com o programa original. Para isso, modificar a seguinte linha no arquivo `platformio.ini` de:

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

Para:

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

Isto evita a utilização das instruções da extensão RISC-V M e emula-as utilizando outras instruções.

TAREFA: No SweRV EH1, as instruções `div` estão a bloquear. Modificar o processador para permitir instruções `div` sem bloqueio.

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