

**THE IMAGINATION UNIVERSITY PROGRAMME**

**RVfpga Lab 13**

## **Instruções de memória: Instruções lw e sw**

# INTRODUÇÃO

Nos laboratórios anteriores, apresentámos os conceitos básicos de *pipelining* e a sua utilização no processador SweRV EH1, e analisámos a forma como as instruções de Aritmética-Lógica são executadas neste processador. Neste laboratório, continuamos com a análise das instruções básicas; especificamente, analisamos as leituras e escritas de memória.

O sistema de memória é um dos pontos de estrangulamento de desempenho mais críticos nos computadores modernos. As latências da memória são normalmente muito superiores ao ciclo de relógio do núcleo, pelo que o processador pode ter de parar enquanto espera pelos dados da memória.

Neste laboratório, primeiro examinamos o *pipe de Load/Store* (o conjunto de andares do pipeline dedicados à execução de operações *de load/store*) ao ler um local de memória de baixa latência - ou seja, um que não pára o processador. Em seguida, examinamos a execução de instruções de escrita. Finalmente, repetimos a nossa análise ignorando a memória de baixa latência e fazendo interface direta com a memória principal DDR disponível na placa Nexys A7.

Figura 1 ilustra uma visão de alto-nível da microarquitetura do processador SweRV EH1. A figura destaca os andares que são relevantes neste laboratório: Decode, DC1-3 (fases de acesso a dados 1-3), Commit e Writeback.

Graphical user interface

Description automatically generated

Figura 1 Microarquitetura do núcleo SweRV EH1

(figura de https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V\_SweRV\_EH1\_PRM.pdf)

# A INSTRUÇÃO lw ACEDENDO A UMA MEMÓRIA DE BAIXA LATÊNCIA

Nesta secção, utilizamos o código simples da Figura 2 para ilustrar os eventos mais relevantes da execução de uma instrução de leitura. O programa de exemplo consiste num ciclo que contém duas operações lw (*load word*) (destacadas a vermelho), cada uma lendo uma palavra de 32 bits a partir de endereços de memória consecutivos e alinhados à palavra. Todas as iterações acessam os mesmos dados e não fazem nada com eles.

Tal como no Lab 12, as instruções lw (destacadas a vermelho na figura) estão rodeadas por várias instruções nop (no-operation) para as isolar das instruções anteriores e posteriores. Por uma questão de simplicidade, neste laboratório também descativamos a utilização de instruções comprimidas, tal como explicado no documento SweRVref.

|  |
| --- |
| .globl principal  .secção .midccm  A: .espaço 8  .texto  principal:  # Registo t3 = x28 (registo 28)  la t0, A # t0 = addr(A)  li t1, 0x2 # t1 = 2  sw t1, (t0) # A[0] = 2  adicionar t1, t1, 6 # t1 = 8  sw t1, 4(t0) # A[1] = 8  INSERT\_NOPS\_9  REPETIR:  INSERT\_NOPS\_1  **lw t1, (t0)**  INSERT\_NOPS\_9  INSERT\_NOPS\_4  **lw t1, 4(t0)**  INSERT\_NOPS\_10  INSERT\_NOPS\_4  beq zero, zero, REPEAT# Repetir o ciclo  .fim |

Figura 2 Programa de exemplo com duas instruções lw

A pasta *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_DCCM* fornece o projeto PlatformIO para que possa analisar, simular e modificar o programa. Abra o projeto em PlatformIO, construa-o e abra o ficheiro de *disassembly* (localizado em *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_DCCM/.pio/build/swervolf\_nexys/firmware.dis*). Nesse ficheiro, localize a segunda instrução lw, que se encontra no endereço 0x0000014c. Observe o código de máquina da instrução (0x0042a303):

**0x0000014c: 0042a303 lwt1 ,4(t0)**

**TAREFA:** Verificar se estes 32 bits (0x0042a303) correspondem à instrução lw t1,4(t0) na arquitetura RISC-V.

Até agora, no Guia de Iniciação (GSG) e nos laboratórios anteriores, temos usado a memória DDR disponível na placa Nexys A7 para armazenar as instruções e os dados do nosso programa. No entanto, o acesso a essa memória externa requer vários ciclos e dificulta a análise das etapas de uma instrução load/store; assim, nesta secção, utilizamos a memória DCCM (Data Closely-Coupled Memory) de baixa latência disponível no SweRV EH1 para armazenar os dados do programa.

A DCCM é uma memória local fortemente acoplada ao núcleo. Fornece acesso de baixa latência e proteção SECDED ECC[[1]](#footnote-2) . O seu tamanho é definido como um argumento no momento da construção do núcleo, variando entre 4 KiB e 512 KiB (64 KiB é a predefinição). No Lab 20, analisaremos o DCCM e o ICCM em mais pormenor; neste laboratório, usamo-lo simplesmente para simplificar a análise das instruções load/store. Note-se que, desta forma, tudo acontece dentro do SweRV EH1 Core Complex (Figura 3), onde estão colocados o pipeline SweRV EH1 e o DCCM (realçado a vermelho).



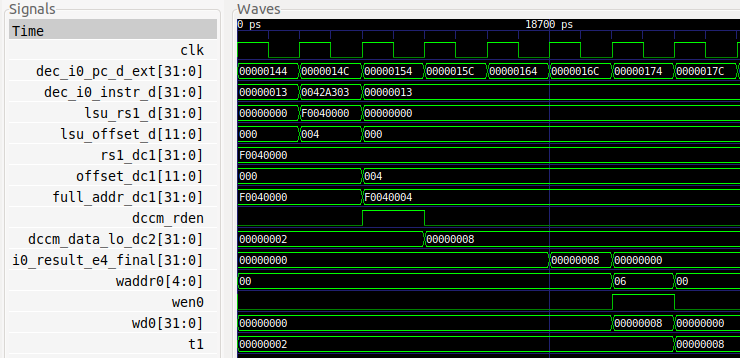
Figura 3 SweRV EH1 Core Complex

O código da Figura 2 define uma secção *ad-hoc* chamada *.midccm* para alocar espaço na DCCM. Por defeito, o espaço de endereçamento da DCCM começa em 0xF0040000 no nosso sistema RVfpga por omissão. O *linker script* fornecido neste projeto (disponível em: *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_DCCM/ld/link.lds*) tratará da atribuição correta dos endereços. Este linker script é usado incluindo o seguinte comando no ficheiro *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_DCCM/platformio.ini*:

board\_build.ldscript = ld/link.lds

1. **Análise de base da instrução lw**

Figura 4 mostra a execução da segunda instrução lw para uma iteração intermédia do ciclo da Figura 2. Os sinais mostrados são os especificados no ficheiro: *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_DCCM/scriptLoad.tcl*. Note-se que todas as iterações são iguais: o primeiro carregamento lê a primeira palavra de dados do DDCM (2) em *t1* (*x6*); o segundo carregamento lê a segunda palavra de dados do DDCM (8) no mesmo registo (*t1*).



**DC2 / Compromisso**

**DECODE**

**Writeback**

**DC1**

**i+4**

**i+3**

**i+6**

**i+5**

**i+2**

**i+1**

**Ciclo i**

**lw t1, 4(t0)**

Figura 4 . Simulação do Verilator para o programa de exemplo da Figura 2

Figura 5 mostra uma vista de alto-nível do pipeline do SweRV EH1 durante a execução da segunda instrução lw. Note-se que a figura junta o estado do processador em diferentes ciclos:

* **Ciclo i**: A instrução é descodificada e o Register File é lido.
* **Ciclo i+1**: O endereço efetivo é calculado utilizando o somador.
* **Ciclo i+2**: O DDCM é lido utilizando o endereço calculado no andar anterior.
* **Ciclo i+5**: O valor lido da memória é escrito no Register File.



Figura 5 . Vista de alto-nível da instrução lw em execução no pipeline EH1 da SweRV

**TAREFA:** Replicar a simulação da Figura 4 no seu próprio computador. Siga os passos seguintes (descritos em pormenor na Secção 7 das GSG):

* Se necessário, gerar o binário de simulação (*Vrvfpgasim*).
* Na PlatformIO, abra o projeto fornecido em: *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_DCCM*.
* Corrija o caminho para o binário de simulação RVfpga (*Vrvfpgasim*) no ficheiro *platformio.ini*.
* Gere o *trace* da simulação com o Verilator (*Generate trace*).
* Abra o *trace* usando o GTKWave.
* Utilizar o ficheiro *scriptLoad.tcl* (fornecido em *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_DCCM/*) para abrir os mesmos sinais que os mostrados na Figura 4. Para isso, no GTKWave, clicar em *File → Read Tcl Script File* e selecionar o ficheiro *scriptLoad.tcl*.
* Clicar várias vezes em *Zoom In* ( ) e passar para 18600ps.

Analisar a forma de onda da Figura 4 e o diagrama da Figura 5 ao mesmo tempo. As figuras incluem alguns sinais associados aos andares Decode, DC1-3, Commit e Writeback. Os valores destacados a vermelho correspondem à segunda instrução lw à medida que atravessa estes andares.

* **Ciclo i: Descodificação:** dec\_i0\_pc\_d\_ext contém o endereço da instrução lw (0x0000014C) e o sinal dec\_i0\_instr\_d contém os 32 bits da instrução de máquina lw (0x0042a303).

Neste andar, **são gerados** os **sinais de controlo**. Além disso, **são obtidos** os **operandos para calcular o endereço efetivo de leitura**: o sinal lsu\_rs1\_d contém o endereço de base da operação lw (que neste exemplo é mantido no registo *t0* e é igual a 0xF0040000), e o sinal lsu\_offset\_d contém o imediato assinado de 12 bits extraído da instrução (0x004 neste exemplo).

* **Ciclo i+1: DC1**: O endereço é calculado utilizando um somador localizado dentro do módulo **lsu\_lsc\_ctl**. O endereço é o endereço de base (rs1\_dc1 = 0xF0040000) mais o *offset* com extensão de sinal (offset\_dc1 = 0x00000004); o endereço final é full\_addr\_dc1 = 0xF0040004. Este endereço é verificado (Verificação de endereço) para determinar a região de memória do acesso (DCCM, PIC ou memória externa). Neste exemplo, dado que o endereço final pertence ao intervalo DCCM (0xF0040004), dccm\_rden é ativado para permitir a leitura do banco DCCM correspondente. O endereço final (full\_addr\_dc1) e o sinal de ativação (dccm\_rden) são fornecidos ao DCCM, que é lido no ciclo seguinte.
* **Ciclo i+2: DC2**: o DCCM é lido e os dados são colocados em dccm\_data\_lo\_dc2 = 0x8, que é propagado para o andar seguinte.
* **Ciclo i+3:** **DC3**: os dados lidos do DCCM são propagados para o andar seguinte.
* **Ciclo i+4:** **Commit**: os dados lidos do DCCM (sinal i0\_result\_e4\_final = 0x8) são propagados para o andar seguinte.
* **Ciclo i+5:** **Writeback**: Finalmente, o valor lido da memória é **escrito de volta** para o Register File através do sinal wd0 = 0x8. Dado que wen0 = 1, o valor é escrito no final desse ciclo no registo x6 (waddr0 = 0x6). Pode observar que, no ciclo seguinte (último ciclo mostrado na Figura 4), o registo x6 (também chamado *t1*) contém o novo valor (t1 = 0x8). Note que o sinal t1 mostrado na forma de onda é um aliás definido no script *.tcl* para o sinal dout.

1. **Análise avançada da instrução lw**

Nesta secção, analisamos mais detalhadamente as etapas percorridas pela instrução lw. Figura 6 mostra um diagrama dos principais elementos que a instrução load do nosso exemplo atravessa durante a sua execução ao longo do pipe Load/Store (andares DC1, DC2 e DC3). Pode ser necessário fazer zoom na figura para ver os detalhes. Os blocos pretos rotulados como *LOGIC* na figura contêm vários blocos, como multiplexers e portas lógicas. Por uma questão de simplicidade, apenas alguns dos sinais de interface do bloco estão incluídos na figura.

Os andares de Decode e Writeback são idênticos aos apresentados para as instruções A-L (ver Figura 6 no Lab 12). No entanto, destacamos alguns detalhes do andar de Decodificação. Recorde-se que no andar de Decode são gerados sinais de controlo e as instruções e operandos são programados para os Pipes adequados:

* O *offset* imediato da leitura está no sinal lsu\_offset\_d.
* O endereço base da leitura está no sinal exu\_lsu\_rs1\_d. (Este sinal é produzido a partir de um multiplexer 4:1 (mostrado na Figura 4 do Laboratório 11), e propagado para o estágio DC1 depois de passar por alguma lógica).
* Os sinais para instruções load/store estão em lsu\_p, um novo pacote de sinais de controlo mostrado na Figura 6.

À semelhança do Decode, o andar de Commit também foi analisado no Lab 12, mas agora incluímos a entrada para o multiplexer 3:1 final relacionado com as instruções de leitura (lsu\_result\_corr\_dc4), que foi omitido no Lab 12 por uma questão de simplicidade. Lembre-se que a saída desse multiplexer 3:1 é i0\_result\_e4\_final[31:0], como mostrado na Figura 6. Além disso, neste laboratório só nos concentramos na Via-0, mas uma leitura/escrita pode ser executada através de qualquer uma das vias do processador superescalar de duas vias. Note, no entanto, que existe apenas um pipe L/S (Load/Store). Assim, a Via-1 também tem um multiplexer 3:1 (cuja saída é i1\_result\_e4\_final[31:0] e uma de suas entradas é lsu\_result\_corr\_dc4), como mostrado na Figura 4 do Lab 11.

**TAREFA:** Ampliar a simulação de Figura 4 para incluir os sinais mostrados em

Figura 6 , que são explicadas a seguir. Um ficheiro *.tcl* que pode utilizar é fornecido em: *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_DCCM/scriptLoadExtended.tcl*

**TAREFA:** Localizar os módulos e sinais da Figura 6 nos ficheiros Verilog do processador SweRV EH1.



Figura 6 Principais elementos percorridos pelas instruções de leitura no circuito de load/store

1. **Fase de descodificação**

Os pormenores gerais do andar de Decode já foram analisados nos Lab 11 e 12. Lembre-se que o andar de Decode é responsável por duas tarefas principais:

* **Descodificar** as instruções e gerar **sinais de controlo**.
* **Distribuir** as instruções pelos tubos apropriados e fornecer os **operandos de entrada**.

**Descodificar as instruções e gerar sinais de controlo**:

Para além de outras estruturas de sinais de controlo já analisadas nos Labs 11 e 12, uma estrutura adicional, *lsu\_pkt\_t*, contém sinais de instruções load/store. Como é habitual, esta estrutura está definida no ficheiro *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv\_types.sv*. O sinal *lsu\_p* é um exemplo de um sinal deste tipo que é propagado a partir do andar de Decode através dos andares de load/store do Pipe.

Este sinal contém algumas informações relevantes para a leitura/escrita na memória:

* **O bit 0** (valid) é definido como 1 se a operação for válida.
* **O bit 12** (unsign) é definido para 1 quando os dados a ler/escrever não têm sinal.
* **O bit 13** (store) é definido como 1 se a operação for uma escrita (sb, sh, sw...).
* **O bit 14** (load) é posto a 1 se a operação for uma leitura (lb, lh, lw...).
* **Os bits 15-18** codificam o tamanho do acesso (byte, meia palavra, palavra, palavra dupla).

**TAREFA:** Incluir o sinal *lsu\_p* na simulação de Figura 4 e analisar os seus bits de acordo com esta descrição.

**Distribuir as instruções pelos pipes apropriados e fornecer os operandos de entrada**:

Como explicado no Lab 11, o processador SweRV EH1 inclui vários pipes para a execução de instruções. No andar de Decode as instruções, depois de descodificadas, devem ser agendadas através dos pipes apropriados. No programa que estamos a analisar neste laboratório (Figura 2), a instrução lw é enviada para execução no Pipe LSU (andares DC1-3). Especificamente, exu\_lsu\_rs1\_d é o valor mantido no registo base. O sinal dec\_lsu\_offset\_d é o deslocamento imediato assinado de 12 bits, que é extraído da instrução e enviado para o estágio DC1.

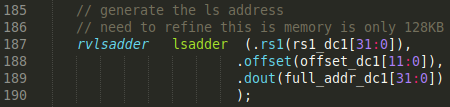
**TAREFA:** Analisar no código Verilog o caminho percorrido pelas duas entradas para o LSU (exu\_lsu\_rs1\_d e dec\_lsu\_offset\_d) a partir das fontes onde são obtidas. Vários módulos estão envolvidos neste processo: **dec**, **exu**, **lsu**.

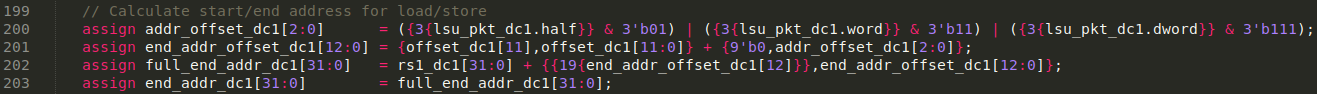
1. **Andar DC1**

No andar DC1, rs1\_dc1 (endereço de base, propagado a partir da descodificação) e offset\_dc1 (offset, propagado a partir da descodificação) são adicionados no módulo **lsadder** para calcular o *endereço principal efetivo* (sinal full\_addr\_dc1[31:0], que é atribuído a lsu\_addr\_dc1[31:0]). Este é o endereço de memória a ser lido.

Além do endereço a ser lido, o *endereço final* (end\_addr\_dc1[31:0]) também é computado noutro somador (vale lembrar que esse segundo somador não foi mostrado na Figura 5 nem na Figura 4 do Lab 11, por uma questão de simplicidade). Este é o endereço do último byte que deve ser lido da memória. Este endereço é usado para tratar acessos não alinhados e acessos de sub-palavras (byte, meia-palavra).

**TAREFA:** Analisar a implementação dos dois somadores do estágio DC1, que estão instanciados no módulo **lsu\_lsc\_ctl**. Fornecemos orientações na Figura 7 abaixo, mostrando a implementação desses somadores.





**Figura 7. Verilog para somadores na fase DC1 do ficheiro *lsu\_lsc\_ctl.sv*.**

Por exemplo, uma palavra de leitura (lw) para o endereço que começa em 0xF0040003 teria: full\_addr\_dc1=0xF0040003 e end\_addr\_dc1=0xF0040006 (ver Figura 8). Desta forma, o pipe LSU pode extrair a palavra do pacote lido, que consiste em duas palavras que começam em endereços que são múltiplos de quatro (neste caso 0xF0040000 e 0xF0040004).



Figura 8 . Exemplo de uma instrução lw para o endereço 0xF0040003

Os dois endereços (lsu\_addr\_dc1[31:0] e end\_addr\_dc1[31:0]) são enviados para a memória de dados (no nosso exemplo, a DCCM), que será acedida no ciclo seguinte.

**TAREFA:** No programa da Figura 2experimente diferentes tamanhos de acesso (byte, half-word) e acessos não alinhados. Para isso, altere o *offset* ou o tipo de acesso de lw para lb (*load byte*) ou lh (*load half-word*). Por exemplo, se alterar o deslocamento de 4 para 3, a instrução *load word* executa um acesso não alinhado aos 32 bits que começam no endereço 0xF0040003, como mostrado na Figura 8. Analise o valor dos sinais lsu\_addr\_dc1[31:0] (ou full\_addr\_dc1[31:0]) e end\_addr\_dc1[31:0] nestas diferentes situações.

No Lab 20, analisamos esta situação a partir dos elementos internos da DCCM.

Para além do cálculo do endereço, a fase DC1 efetua uma verificação do intervalo de endereços no módulo **lsu\_addrcheck** (ver Figura 9) para determinar a memória de destino do acesso, que é a DCCM no nosso exemplo.

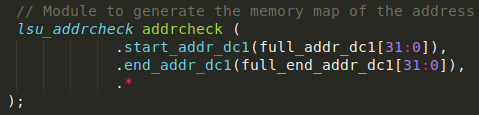


Figura 9 . Verificar a gama e a localização do endereço de memória

Como resultado da verificação do endereço, é determinado qual a memória a que se deve aceder: a DCCM, o PIC ou a memória DDR externa (ver Figura 10).



Figura 10 . Endereços de cada unidade de memória

No nosso exemplo, o sinal de ativação de leitura do DCCM fica ativo (addr\_in\_dccm\_dc1 = 1). Este sinal passa por alguma lógica e depois é fornecido ao DCCM (sinal dccm\_rden) para ativar/desativar o acesso (no nosso exemplo, para o ativar). O sinal addr\_external\_dc1, que é 1 quando a Memória DDR Externa deve ser habilitada e 0 caso contrário, é propagado e utilizado pelo andar DC3, como mostra a Figura 6.

1. **Andar DC2**

Se a leitura DCCM estiver ativada (dccm\_rden = 1), os dados são lidos neste andar. Note-se que são lidos dois valores de 32 bits (dccm\_data\_lo\_dc2[31:0] e dccm\_data\_hi\_dc2[31:0]), uma vez que o acesso aos dados pode ser desalinhado e, por conseguinte, repartido por duas palavras (como no exemplo da Figura 8).

**TAREFA:** No programa da Figura 2 compare o valor dos sinais dccm\_data\_lo\_dc2[31:0] e dccm\_data\_hi\_dc2[31:0] ao fazer um lw para o endereço 0xF0040004 e para o endereço 0xF0040003.

1. **Andar DC3**

Os dois valores de dados de 32 bits do DCCM são propagados de DC2 (sinais dccm\_data\_lo\_dc2[31:0] e dccm\_data\_hi\_dc2[31:0]) para DC3 (sinais dccm\_data\_lo\_dc3[31:0] e dccm\_data\_hi\_dc3[31:0]). Para acessos alinhados, como o do nosso exemplo, ambos os sinais são iguais e apenas o dccm\_data\_lo\_dc3[31:0] é utilizado.

No andar DC3, as duas palavras lidas no ciclo anterior (dccm\_data\_lo\_dc3[31:0] e dccm\_data\_hi\_dc3[31:0]) passam por uma lógica que executa várias tarefas:

* **Controlo de erros:** Os dados são verificados quanto a erros utilizando ECC.
* **Gestão de conflitos de leitura/escrita:** Se uma instrução *store* para o mesmo endereço ainda estiver a ser executada, os dados são encaminhados da instrução store para a instrução *load* em vez de serem lidos da memória. Analisaremos esta situação no Lab 15.
* **Alinhamento:** Os dados solicitados estão alinhados.

Como resultado de todos estes processos, os dados finais são fornecidos no sinal lsu\_ld\_data\_corr\_dc3[31:0].

**TAREFA:** Analisar os circuitos lógicos de Align, Merge e Error Check utilizada no código Verilog nos módulos **lsu\_dccm\_ctl** e **lsu\_ecc**.

**TAREFA:** No programa da Figura 2 compare o valor do sinal lsu\_result\_corr\_dc3[31:0] ao efetuar um lw para o endereço 0xF0040004 e para o endereço 0xF0040003.

Após esta lógica que executa a verificação de erros, o tratamento de conflitos de leitura/escrita e o alinhamento, está um multiplexer 2:1 que seleciona entre dados do DCCM (lsu\_ld\_data\_corr\_dc3[31:0]) ou da memória DDR (bus\_read\_data\_dc3[31:0]). Este multiplexer é controlado pelo sinal addr\_external\_dc3, que foi gerado no módulo **lsu\_addrcheck** no andar DC1 (sinal addr\_external\_dc1).

**TAREFA:** Analisar no código Verilog como o sinal addr\_external\_dc1 foi computado no andar DC1 no módulo **lsu\_addrcheck**.

A saída deste multiplexer 2:1 (lsu\_result\_corr\_dc3[31:0]) é propagada para o andar de Commit.

1. **Andar de Commit**

No andar de Commit, um multiplexer 3:1 seleciona os dados de leitura (i0\_result\_e4\_final[31:0]) para serem enviados para o andar de Writeback (ver Figura 6). Este multiplexer 3:1 também pode selecionar a saída da ALU, como já foi explicado nos Labs 11 e 12, e a saída da ALU Secundária, como analisaremos no Lab 15.

1. **Andar de Writeback**

Este andar já foi explicado nos labs 11 e 12, pelo que não é apresentado na Figura 6 onde o resultado da adição foi escrito no registo de destino. Neste caso, este andar escreve os dados do DCCM no registo de destino.

# A INSTRUÇÃO sw DE ACEDENDO A UMA MEMÓRIA DE BAIXA LATÊNCIA

Nesta secção, utilizamos o código apresentado na Figura 11 para ilustrar os eventos mais relevantes da execução de uma instrução de escrita. O código contém um ciclo com 1000 iterações que escreve em endereços de memória consecutivos. O vetor A contém 1000 palavras e é colocado no DCCM (0xF0040000 - 0xF004FFFF). Cada sw é seguido de um lw que verifica se o valor correto foi armazenado. Como é habitual, são inseridos nops para isolar as instruções e, neste caso, também para garantir que os dados são efetivamente escritos e lidos da memória e não apenas encaminhados da instrução sw para a lw. Como é habitual, desativamos a utilização de instruções comprimidas, tal como explicado no documento SweRVref. Além disso, tal como no exemplo da secção anterior, usamos o DCCM para armazenar e carregar dados.

|  |
| --- |
| .globl principal  .secção .midccm  A: .space 4000  .texto  principal:  la t0, A # t0 = addr(A)  li t1, 0x2 # t1 = 2  li t2, 1000 # t2 = 1000  INSERT\_NOPS\_2  REPETIR:  **sw t1, (t0)**  INSERT\_NOPS\_10  INSERT\_NOPS\_4  lw t1, (t0)  INSERT\_NOPS\_10  adicionar t1,t1,t1  adicionar t0,t0,0x04  adicionar t2,t2,-1  INSERT\_NOPS\_10  bne t2, zero, REPEAT # Repete o ciclo  não  não  .fim |

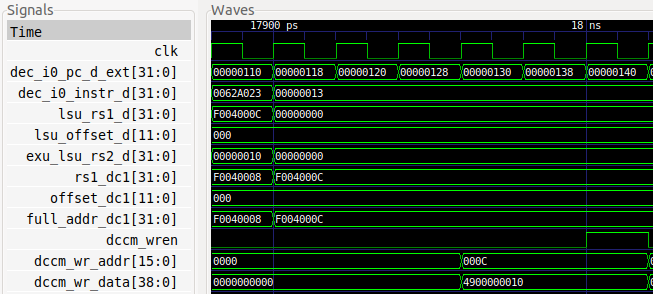
Figura 11 Exemplo de código com a instrução sw

A pasta *[RVfpgaPath]/RVfpga/Labs/Lab13/SW\_Instruction\_DCCM* fornece o projeto PlatformIO para que possa analisar, simular e modificar o programa. Abra o projeto em PlatformIO, compile-o e abra o ficheiro de *disassembly* (disponível em *[RVfpgaPath]/RVfpga/Labs/Lab13/SW\_Instruction\_DCCM/.pio/build/swervolf\_nexys/firmware.dis*). Verá que a instrução sw é colocada no endereço 0x00000110, e pode também ver o código de máquina para a instrução (0x0062a023):

**0x00000110: 0062a023 sw t1, 0(t0)**

**TAREFA:** Verificar se estes 32 bits (0x0062a023) correspondem à instrução sw t1, 0(t0) na arquitetura RISC-V.

Figura 12 mostra a execução da instrução sw durante a quarta iteração do ciclo da Figura 11. Qualquer iteração, exceto a primeira, pode ser analisada. Como é habitual, a primeira execução de uma instrução não deve ser utilizada para evitar *misses* na cache de instruções (I$).



**DECODE**

**i+6**

**Ciclo i**

**DCCM**

**Escrever**

**DC1**

**i+1**

Figura 12 Simulação Verilator para o exemplo de Figura 11

Figura 13 mostra uma vista de alto-nível do pipeline SweRV EH1 durante a execução da instrução sw durante a quarta iteração do ciclo da Figura 11. O registo t1 (que contém o valor a escrever na memória) é 0x10 e t0 (que contém o endereço base) é 0xF004000C. Assim, sw escreve o valor 0x10 no endereço 0xF004000C do DCCM. A figura mostra os nomes reais utilizados nos módulos Verilog do processador SweRV EH1. Note-se que a figura funde o estado do processador em diferentes ciclos:

* **Ciclo i**: A instrução store é descodificada no andar Decode, é atribuída ao LSU Pipe e os operandos são fornecidos, neste caso a partir do campo imediato da instrução e do Register File, que é lido neste ciclo.
* **Ciclo i+1**: O endereço efetivo é calculado na unidade de somador como explicado para a load. Note-se que apenas o lsadder mostrado na Figura 6 está incluído na figura por uma questão de simplicidade.
* **Ciclo i+6:** O segundo operando (lido do registo t1) é armazenado no DCCM, depois de passar pelo *Store Buffer*, que explicamos no Apêndice.

Note-se que a escrita não é uma operação crítica em termos de tempo de execução do programa, pelo que pode ser atrasada vários ciclos sem afetar o desempenho. Em contrapartida, as instruções de leitura podem ser críticas, uma vez que leem frequentemente um valor necessário para uma instrução subsequente, pelo que, como mencionado na secção anterior, é implementado um caminho de encaminhamento store-load (não mostrado na Figura 13), que poupa os acessos à memória e evita paragens no pipeline em caso de conflito de dados entre uma escrita e uma leitura subsequente para o mesmo endereço de memória. Analisamos esta situação no Lab 15.



Figura 13 Vista de alto-nível da execução da instrução sw no SweRV EH1

**TAREFA:** Replicar a simulação da Figura 12 no seu próprio computador. Siga os passos seguintes (descritos em pormenor na Secção 7 das GSG):

* Se necessário, gerar o binário de simulação (*Vrvfpgasim*).
* Abrir no PlatformIO o projeto fornecido em: *[RVfpgaPath]/RVfpga/Labs/Lab13/SW\_Instruction\_DCCM*.
* Atualizar o caminho para o binário de simulação RVfpga (*Vrvfpgasim*) no ficheiro *platformio.ini*.
* Gerar o *trace* da simulação com o Verilator (Gerar traço).
* Abrir o *trace* no GTKWave.
* Utilize o ficheiro *scriptStore.tcl* (fornecido em *[RVfpgaPath]/RVfpga/Labs/Lab13/SW\_Instruction\_DCCM/*) para apresentar os mesmos sinais que os mostrados na Figura 4. Para isso, no GTKWave, clique em *File → Read Tcl Script File* e seleccione o ficheiro *scriptStore.tcl*.
* Clicar várias vezes em *Zoom In* ( ) e passar para 17900ps.

Analisar a forma de onda da Figura 12 e o diagrama da Figura 13 em simultâneo. A figura inclui alguns sinais associados aos andares de Decode e DC1, bem como alguns sinais relacionados com a escrita na DCCM, que ocorre vários ciclos depois. Os valores destacados a vermelho correspondem à instrução sw à medida que esta atravessa estas etapas.

* **Ciclo i: Descodificação:** Como explicado para a instrução load, o sinal dec\_i0\_pc\_d\_ext contém o endereço da instrução sw (0x00000110) e o sinal dec\_i0\_instr\_d contém a instrução sw de 32 bits (0x0062a023). O sinal lsu\_rs1\_d contém o endereço base da operação sw (que neste exemplo é 0xF004000C, como fornecido pelo registo t0), e o sinal lsu\_offset\_d contém o imediato de 12 bits (0x000 neste exemplo) que foi extraído da instrução e subsequentemente adicionado ao endereço base. Para instruções store, o valor lido do segundo registo (neste caso t1) será eventualmente escrito na memória (exu\_lsu\_rs2\_d = 0x10). Assim, ele deve ser propagado para os andares subsequentes.
* **Ciclo i+1: DC1: Tal** como explicado para as leituras, durante esta fase o endereço é calculado (full\_addr\_dc1 = rs1\_dc1 + offset\_dc1 = 0xF004000C).
* **Ciclo i+6: Escrita DCCM:** Após cinco ciclos, o DCCM recebe os dados e o endereço de escrita (dccm\_wr\_addr=0x000C e dccm\_wr\_data=0x4900000010) do *Store Buffer*. Note-se que o DCCM só recebe os últimos 16 bits do endereço (0x000C), porque o seu tamanho real é de 64 KiB na nossa configuração (ver ficheiro *common\_defines.vh*) e 16 bits são suficientes para endereçar 216 bytes. Os dados foram precedidos de alguns bits ECC (0x49). Quando o sinal dccm\_wren é ativado (no ciclo i+6 na Figura 12), a escrita no DCCM fica concluída.

**APÊNDICE A - FUNCIONAMENTO DO STORE BUFFER:** O Apêndice A explica o *Store Buffer*, que é uma estrutura importante que guarda temporariamente o valor e o endereço que devem ser escritos na memória pela instrução store.

**TAREFA:** Analisar na simulação a instrução load que se segue à store para verificar se o valor foi corretamente escrito no DCCM. Terá de acrescentar alguns dos sinais da Figura 4 e Figura 6 para analisar a leitura.

**TAREFA:** Estender a análise básica realizada nesta secção para a instrução sw de forma semelhante à análise avançada realizada para a instrução lw na Secção 2.B.

**TAREFA:** Analisar as escritas não alinhados no DCCM, bem como as escritas de sub-palavras: escrever byte (sb) ou *escrever meia-palavra* (sh).

# ACESSO À MEMÓRIA EXTERNA

Nas secções 2 e 3, utilizámos o DCCM para armazenar e carregar os dados. Nesta secção, analisamos as instruções de leitura que acedem à memória externa disponível no Nexys A7. Note-se que, neste caso (ver Figura 14), ao contrário do cenário analisado na Secção 2 (ver Figura 3), o núcleo SweRV EH1 tem de comunicar com a memória externa através do barramento AXI para obter os dados solicitados pela instrução de leitura.



Figura 14 . RVfpgaNexys

Analisamos tanto os acessos bloqueantes como os não bloqueantes. As leituras bloqueantes param completamente o processador até receberem os dados lidos da memória. Isto significa que nenhuma outra instrução avança até que a leitura receba os seus dados. Em contrapartida, as leituras não bloqueantes permitem que a execução do programa continue desde que as instruções não dependam dos dados lidos; a execução só pára quando é executada uma instrução que depende de uma leitura. Nestes dois cenários, os dados lidos da memória seguem caminhos diferentes através do pipeline; neste laboratório analisamos o primeiro (leituras bloqueantes) e no laboratório seguinte (Lab 14) analisamos o segundo caso (leituras não bloqueantes) no contexto dos conflitos estruturais.

O código da Figura 15 apresenta um exemplo simples para ilustrar a execução de uma instrução lw que lê a memória DDR externa. O código contém um ciclo que lê uma matriz de 12 elementos (lw t3, (t4)) e acumula a soma dos seus elementos no registo t6 (add t6, t3, t6). Como é habitual, são inseridas várias operações nop para isolar as instruções e facilitar a sua análise, e as instruções comprimidas são desativadas.

O vetor D contém 12 palavras e é colocado na memória principal. Para isso, o vetor é declarado na secção *.data* e é utilizado o script de ligação habitual para o projeto (disponível em *~/.platformio/packages/framework-wd-riscv-sdk/board/nexys\_a7\_eh1/link.lds*). Desta forma, os dados definidos na secção *.data* são colocados na memória externa e não na DCCM, como foi feito no programa da Figura 2.

Por omissão, as instruções de leitura são não bloqueantes no SweRV EH1. Se quisermos que as instruções de leitura sejam bloqueantes, temos de incluir as duas instruções seguintes no início do código de Assembly que vamos analisar, como mostra a Figura 15 (consulte a Secção 2.C do documento SweRVref para obter mais explicações sobre a ativação/desativação das características principais):

li t2, 0x020

csrrs t1, 0x7F9, t2

|  |
| --- |
| .globl main  **.dados**  D: .word 3,5,6,8,7,10,12,2,1,4,11,9  .text  main:  **li t2, 0x020**  **csrrs t1, 0x7F9, t2**  la t4, D  li t5, 12  li t6, 0x0  INSERT\_NOPS\_1  REPEAT:  **lw t3, (t4)**  add t5, t5, -1  INSERT\_NOPS\_10  add t6, t3, t6  add t4, t4, 4  INSERT\_NOPS\_9  bne t5, zero, REPEAT # Repete o ciclo  INSERT\_NOPS\_4  .end |

Figura 15 Exemplo de instrução lw bloqueante

A pasta *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_ExtMemory* fornece o projeto PlatformIO para que possa analisar, simular e modificar o programa. Se abrir o projeto em PlatformIO, compilá-lo e abrir o ficheiro *Disassembly* (disponível em *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_ExtMemory/.pio/build/swervolf\_nexys/firmware.dis*), verá que a instrução lw é colocada no endereço 0x000000f4, e pode também ver o código de máquina para a instrução (0x000eae03):

**0x000000f4: 000eae03 lw t3, 0(t4)**

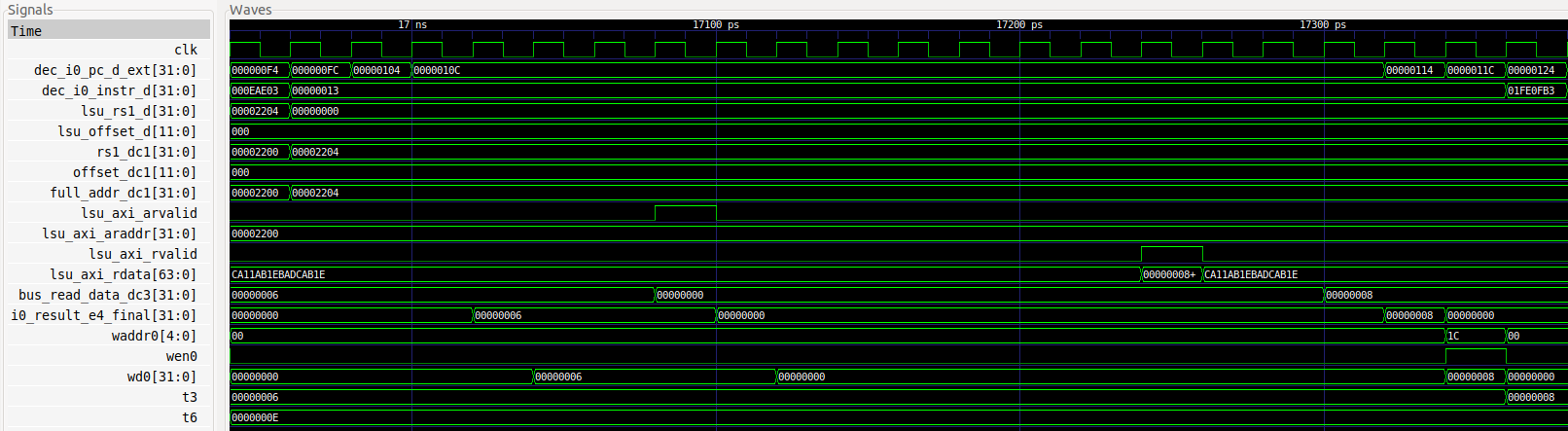
As leituras bloqueantes que acedem à Memória DDR2 Externa seguem praticamente o mesmo caminho explicado na Secção 2 para as leituras que acedem ao DDCM, como mostraremos na Figura 16. No entanto, existe uma diferença importante: durante alguns ciclos, o processador fica parado à espera dos dados fornecidos pela Memória Externa; depois, quando os dados solicitados são recebidos, as instruções podem continuar a ser executadas.

O módulo que controla o acesso à Memória Externa através do barramento AXI é chamado **lsu\_bus\_intf** no SweRV EH1. Ele é responsável por fornecer o endereço ao controlador da Lite DRAM e, alguns ciclos depois, receber e alinhar os dados solicitados e inseri-los no núcleo no andar DC3. Note-se que é utilizado um barramento AXI para comunicar com a memória externa DDR2. Neste exemplo (Figura 15), um multiplexer 2:1 no andar DC3, que também foi incluído na Figura 6 seleciona a entrada proveniente da memória externa (ou seja, lsu\_result\_corr\_dc3 = bus\_read\_data\_dc3), em vez da entrada proveniente do DCCM (lsu\_ld\_data\_corr\_dc3) que foi selecionada no exemplo da Figura 2. Quanto ao multiplexer 3:1 no andar Commit, seleciona a mesma entrada que no exemplo da Figura 2 (ou seja, i0\_resultado\_e4\_final = lsu\_resultado\_corr\_dc4).



Figura 16. Bloqueio da instrução lw que acede à memória externa

Figura 17 mostra a execução do lw na quarta iteração do loop de Figura 15 onde se lê o valor armazenado no endereço 0x00002204 no registo *t3*. Note-se que para este programa a matriz D começa no endereço 0x000021F8.



**Atraso devido ao acesso à memória externa**

Figura 17 . Simulação Verilator do exemplo da Figura 15

**TAREFA:** Replicar a simulação da Figura 17 no seu próprio computador. Utilize o ficheiro *test\_Blocking.tcl* (fornecido em *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_ExtMemory*). *Aumentar o zoom* ( ) várias vezes e passar para 16940ps.

Analise a forma de onda. A figura inclui alguns sinais associados aos andares do pipeline. Observe que o conjunto de sinais na parte superior (clk até full\_addr\_dc1) e o conjunto de sinais na parte inferior (i0\_result\_e4\_final até wd0) são os mesmos que os mostrados na Figura 4. Os valores destacados a vermelho correspondem à instrução lw à medida que atravessa estas etapas.

* O endereço é calculado no andar de Decode, como explicado na secção 2. O sinal full\_addr\_dc1[31:0] contém o *endereço*, que na quarta iteração do nosso exemplo (a mostrada na Figura 17) é 0x00002204. O sinal end\_addr\_dc1[31:0] (não mostrado na figura) é também computado como explicado na Secção 2 e contém o endereço do último byte a ser acedido.
* Alguns ciclos depois, o endereço é enviado para a memória externa através do barramento AXI usando os seguintes sinais: lsu\_axi\_arvalid = 1 e lsu\_axi\_araddr = 0x00002200. Note-se que o endereço enviado está alinhado com duas palavras, uma vez que são lidos 64 bits da memória por acesso. Os dados são lidos no sinal lsu\_axi\_rdata (nos Labs 19 e 20 analisaremos em pormenor a hierarquia da memória). Se for necessário mais do que um endereço para o acesso (devido a um acesso não alinhado), são enviados múltiplos endereços e os dados são devolvidos sequencialmente através do barramento.

**TAREFA:** Modificar o programa de Figura 15 para analisar um acesso de leitura não alinhado que precisa de enviar dois endereços para a memória externa através do barramento AXI.

* Alguns ciclos depois, a memória externa retorna um dado de 64 bits lido através do barramento AXI (lsu\_axi\_rdata = 0x0000000800000006 e lsu\_axi\_rvalid = 1). Esses dados são armazenados em buffer dentro da LSU (módulo **lsu\_bus\_buffer**).
* Os dados de 32 bits solicitados são extraídos dos dados de 64 bits lidos da memória e inseridos no pipeline principal: bus\_read\_data\_dc3 = 0x00000008.
* Em seguida, estes dados são escritos no Register File seguindo o mesmo caminho que no exemplo da secção 2: i0\_resultado\_e4\_final → wd0.

**TAREFA**: Adicione à simulação os sinais que controlam os multiplexers (nos andares DC3 e Commit na Figura 16) que selecionam os dados fornecidos pela Memória Externa DDR. Pode encontrar estes multiplexers nas seguintes linhas do código Verilog:

- Multiplexer 2:1: Linha 264 do módulo **lsu\_lsc\_ctl**.

- Multiplexer 3:1: Linha 2277 do módulo **dec\_decode\_ctl**.

Um ficheiro *.tcl* que pode utilizar é fornecido em: *[RVfpgaPath]/RVfpga/Labs/Lab13/LW\_Instruction\_ExtMemory/test\_Blocking\_Extended.tcl*

**TAREFA:** Também pode ser interessante analisar a implementação do barramento AXI para aceder ao controlador DRAM, para o qual pode inspecionar o módulo **lsu\_bus\_intf**.

# APÊNDICE A - FUNCIONAMENTO DO BUFFER DE ESCRITA

O *buffer de escrita* é uma fila circular de 8 entradas, localizada dentro da LSU, onde cada operação de escrita (sw) é rastreada, anotando seu endereço de destino e dados. Geralmente, um *buffer de escrita* pode ser usado para:

* Satisfazer as operações *de leitura* subsequentes com um *armazenamento* de dados anterior se os seus endereços de destino coincidirem. Este reencaminhamento de dados resolve os problemas de leitura após escrita e poupa os acessos à memória.
* Permitir que as operações *de leitura* independentes avancem rapidamente sobre as operações de *escrita* pendentes anteriores, dado que as operações *de leitura* estão provavelmente no caminho crítico.
* Fundir operações *de escrita* compatíveis numa única operação, resolvendo assim os problemas de escrita após escrita e poupando acessos à memória.

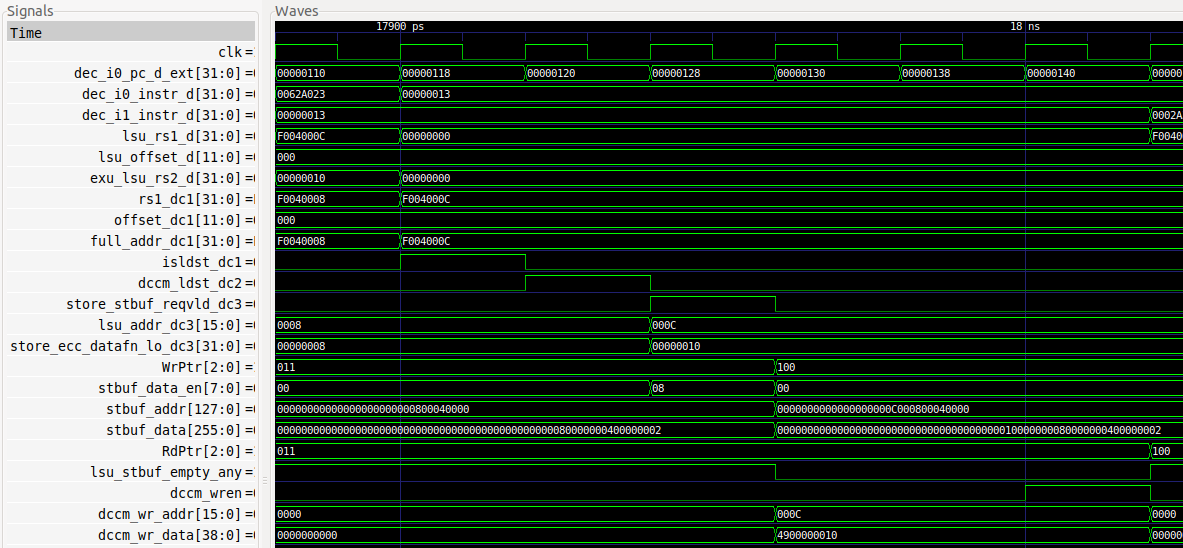
Figura 18 mostra alguns dos sinais relevantes do *Store Buffer* quando se executa o código da Figura 11. Estes novos sinais são adicionados aos que são mostrados na Figura 12. Como nessa figura, Figura 18 mostra a execução da instrução sw na quarta iteração do ciclo.

**i+4**

**i+3**

**i+1**

**Ciclo i**



**DC3**

**Store Buffer**

**DCCM**

Figura 18 . Simulação do Verilator para o exemplo da Figura 11 ilustrando a operação *Store Buffer*

**TAREFA:** Replicar a simulação da Figura 18 no seu próprio computador. Utilize o ficheiro *scriptStoreBuffer.tcl* (fornecido em *[RVfpgaPath]/RVfpga/Labs/Lab13/SW\_Instruction\_DCCM*). *Aumentar o zoom* ( ) várias vezes e passar para 17900ps.

Os sinais de topo (dos andares de Decode, DC1 e DC2) são apresentados na Figura 12 e aí explicados, pelo que não voltamos a explicá-los aqui. Em DC3 (ciclo i na figura), prepara-se a escrita no *Store Buffer*. O endereço final e os dados a serem escritos na memória pela instrução store são enviados para o *Store Buffer* através dos sinais lsu\_addr\_dc3 = 0x000C e store\_ecc\_datafn\_lo\_dc3 = 0x00000010. O sinal store\_stbuf\_reqvld\_dc3 é ativado quando uma operação de escrita é identificada na fase DC3, o que desencadeia a operação do buffer de escrita.

O próximo conjunto de sinais corresponde aos sinais internos do *Store Buffer* (pode encontrá-los no módulo **lsu\_stbuf**). WrPtr codifica a entrada do *Store Buffer* onde a próxima operação de sw irá colocar o seu endereço e dados. No exemplo, WrPtr é 0b011 (ou seja, a entrada número três, que é a quarta entrada, uma vez que começa em 0).

Durante a fase DC3 (ciclo i), a quarta entrada do buffer de escrita é ativada através da ativação do quarto bit do sinal stbuf\_data\_en (note-se que 0x08 se traduz em 00001000 em codificação de um só bit, e o único valor '1' está na posição do quarto bit). O sinal lsu\_stbuf\_empty\_any fica baixo no final deste ciclo para indicar que o *buffer de escrita* não está vazio - ou seja, o buffer de escrita contém dados que estão pendentes para serem escritos na memória.

Na fase Commit (ciclo i+1), ocorre a atualização da quarta entrada do buffer de escrita. Os sinais stbuf\_addr e stbuf\_data contêm na sua quarta entrada: 0x000C (que corresponde ao endereço do DCCM a ser escrito) e 0x00000010 (que corresponde aos dados a serem armazenados no DCCM), respetivamente. WrPtr foi incrementado para apontar para a entrada seguinte da memória intermédia (b100), e RdPtr regista o valor mais antigo da memória intermédia que ainda não foi confirmado (b011).

Um ciclo após a fase de Writeback (ciclo i+3), o sinal de ativação de escrita do DCCM (dccm\_wren) é ativado, escrevendo assim na memória e libertando a quarta entrada do buffer. Finalmente, no ciclo i+4, RdPtr é atualizado (para b100) e a memória intermédia está novamente vazia, pelo que lsu\_stbuf\_empty\_any volta a ser elevado.

Figura 19 mostra como o *buffer de escrita de* 8 entradas evolui no exemplo mostrado na Figura 18. No ciclo i, o *buffer de escrita* está vazio, indicado por WrPtr == RdPtr. No ciclo i+1, o *Store Buffer* contém um par *endereço/dados* (0x000C/0x00000010), que corresponde ao armazenamento analisado na Figura 18. Finalmente, no ciclo i+4, os dados são escritos no DCCM e o *Store Buffer* fica novamente vazio (WrPtr == RdPtr).



Figura 19 . *O Store Buffer* muda durante o exemplo da Figura 18

**TAREFA:** Modificar o programa de Figura 11 de modo a ter duas escritas pendentes e efetuar uma análise semelhante à da Figura 18.

1. Ver o *Manual de Referência do Programador RISC-V SweRVTM EH1* para mais pormenores. [↑](#footnote-ref-2)