



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 19

Cache de Instruções

1. INTRODUÇÃO

Neste e no próximo laboratório, vamos nos concentrar no sistema de memória do sistema RVfpga. Lembre-se da Figura 25 no Guia de Introdução ao RVfpga (que replicamos na Figura 1 por uma questão de conveniência), que o sistema RVfpga tem uma memória principal DDR externa, uma cache para instruções (I\$) e duas memórias Scratchpad (também chamadas de memórias estreitamente acopladas), uma para dados (DCCM) e outra para instruções (ICCM).

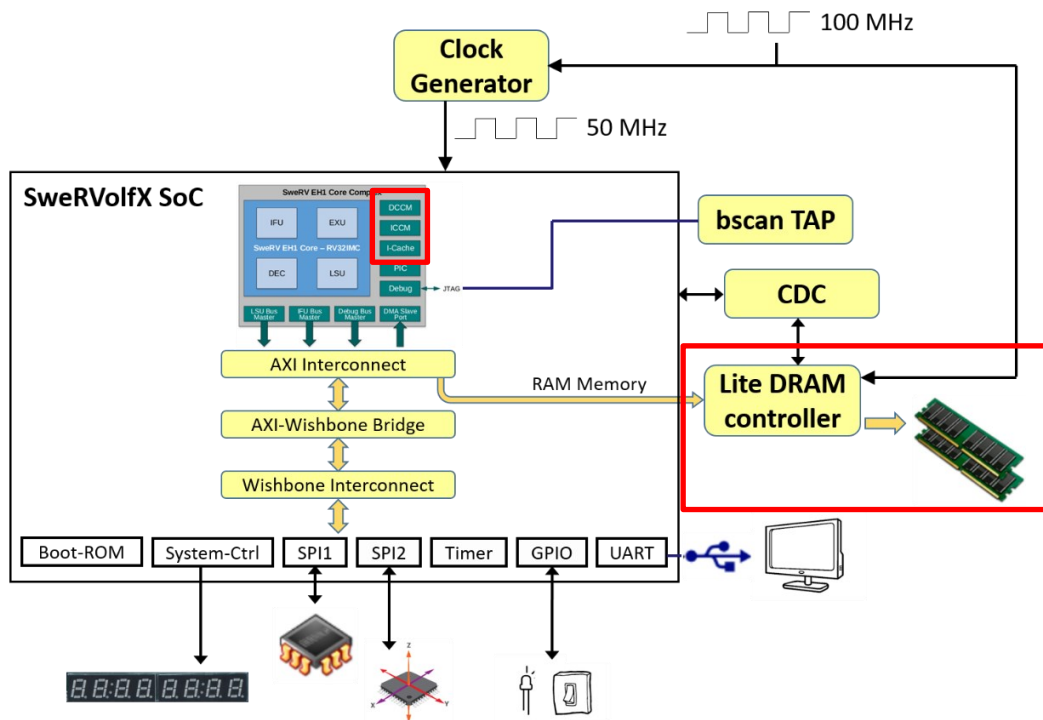


Figura 1 . RVfpgaNexys: o sistema de memória RVfpga é realçado por caixas vermelhas

NOTA: Antes de começar a trabalhar no Lab 19, recomendamos a leitura das Secções 8.1-8.3 do livro de S. Harris e D. Harris, "*Digital Design and Computer Architecture: RISC-V Edition*", Morgan Kaufmann [DDCARV].

Neste laboratório, vamos nos concentrar na operação da Cache. Infelizmente, como mostrado na Figura 1 o sistema RVfpga não inclui uma cache de dados (D\$). Assim, não podemos estudar uma cache usando as abordagens típicas em que os acessos à memória de dados do programa são analisados. No entanto, o sistema RVfpga inclui uma cache de instruções (I\$), que usamos neste laboratório para demonstrar os principais conceitos de uma memória cache. A maior parte dos conceitos explicados na Secção 8.3 de [DDCARV] são também aplicáveis a uma I\$ e, por isso, são úteis para os nossos propósitos.

Primeiro descrevemos como os dados são lidos e escritos na memória externa DDR (Secção 2), e depois aprofundamos o funcionamento e a gestão da I\$ disponível no sistema RVfpga (Secção 3).

2. ACESSOS DE DADOS DA MEMÓRIA EXTERNA DDR

Apesar de não podermos usar uma D\$ neste laboratório para explicar a cache, usamos os acessos a dados para descrever o sistema de memória geral do sistema RVfpga. Nos Labs 13 e 14, mostramos como leituras e escritas usam a memória externa DDR e o DCCM. Tal como explicado nesses laboratórios, sempre que o núcleo precisa de aceder a dados, o endereço é calculado em DC1 e, em seguida, esses dados são lidos ou escritos de/para a memória principal durante os restantes andares utilizando o barramento AXI. O pipeline tem de ficar parado durante alguns ciclos quando acede à memória DDR, mas não fica parado quando acede à DCCM.

O exemplo seguinte ilustra um programa que inclui uma instrução load seguida de uma instrução store, centrando-se na leitura/escrita da memória externa DDR. A pasta `[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory` fornece o projeto PlatformIO para que possa analisar, simular e modificar o programa como desejar. O programa, mostrado na Figura 2 percorre um vetor de 10.000 elementos (não inicializado e usada apenas para fins ilustrativos), lendo cada elemento (instrução `lw`, destacada em vermelho), adicionando uma constante a ele e armazenando o elemento (instrução `sw`, destacada em vermelho) no mesmo componente.

```
.data
D: .space 40000

.text
Test_Assembly:

li t2, 0x000
csrrs t1, 0x7F9, t2

la t4, D
li t5, 50
li t0, 40000
la t6, D
add t6, t6, t0

REPEAT:
    lw t3, (t4)
    add t3, t3, t5
    sw t3, (t4)
    add t4, t4, 4
    bne t4, t6, REPEAT    # Repete o ciclo
```

Figura 2. Programa de exemplo

Abra o projeto no PlatformIO, construa-o e abra o ficheiro Disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory/.pio/build/swervolf_nexys/firmware.dis`). Observe que a instrução `lw` (0x000eae03) e a instrução `sw` (0x01cea023) estão nos endereços 0x00000194 e 0x0000019c, respetivamente.

0x00000194:	000eae03	lw t3,0(t4)
...		
0x0000019c:	01cea023	sw t3,0(t4)

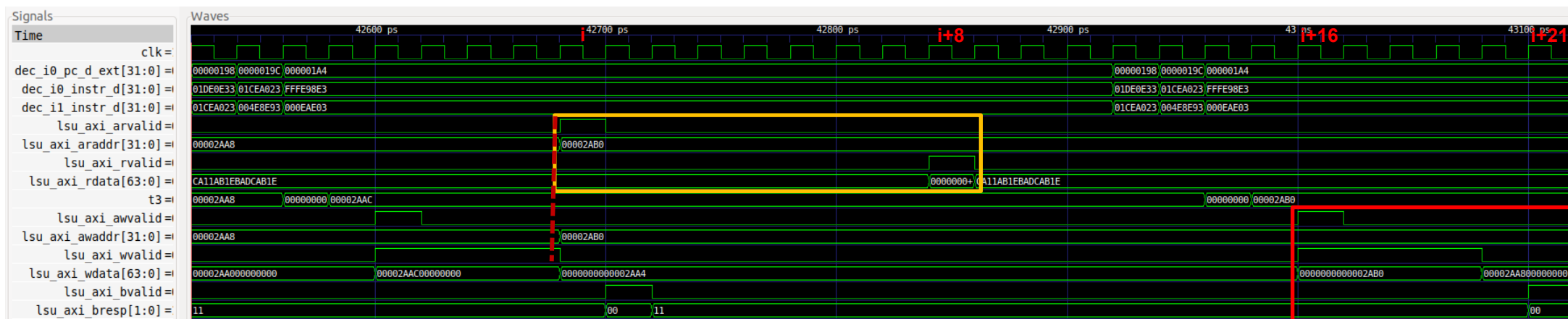



Figura 3. Simulação de uma iteração aleatória do programa da Figura 2

Figura 3 mostra a simulação de uma iteração aleatória do ciclo de Figura 2.

TAREFA: Replicar a simulação da Figura 3 no seu próprio computador. Para o fazer, siga os passos seguintes (descritos em pormenor na Secção 7 das GSG):

- Se necessário, gerar o binário de simulação (*Vrvfpgasim*).
- No PlatformIO, abra o projeto fornecido em: *[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory*.
- Estabelecer o caminho correto para o executável da simulação RVfpga (*Vrvfpgasim*) no ficheiro *platformio.ini*.
- Gerar o *trace* da simulação utilizando o Verilator (*Generate Trace*).
- Abrir o *trace* no GTKWave.
- Utilizar o ficheiro *test_Blocking_Extended.tcl* (disponibilizado em *[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory*) para abrir os mesmos sinais que os apresentados na Figura 6. Para isso, no GTKWave, clique em *File → Read Tcl Script File* e seleccione o ficheiro *test_Blocking_Extended.tcl*.
- Clicar várias vezes em *Zoom In* () e analisar a região que começa em 42500 ps.

A seguir, descrevemos como as leituras e escritas de memória ocorrem usando a memória externa DDR através do barramento AXI. Consulte a Secção 4.B.iii do Guia de Iniciação para obter mais detalhes sobre o funcionamento do barramento.

- O processador lê dados da Memória Externa DDR (quadrado amarelo) em t_3 . A leitura começa no ciclo i , quando a escrita da iteração anterior tiver sido concluída no barramento (ou seja, quando `lsu_axi_wvalid` passar de 1 para 0):
 - o **Ciclo i :** o endereço efetivo é enviado para a memória externa através do barramento AXI:
 - `lsu_axi_arvalid = 1`
 - `lsu_axi_araddr = 0x00002AB0`
 - o **Ciclo $i+8$:** (Note-se que a memória simulada não é igual à memória DDR real da placa Nexys A7, pelo que a latência observada na simulação não é igual à latência na placa, que analisaremos mais tarde), o valor de leitura é recebido através do barramento AXI da Memória Externa:
 - `lsu_axi_rvalid = 1`
 - `lsu_axi_rdata = 0x0`
- O processador calcula a adição (`add t3, t3, t5`) na ALU Secundária e escreve-a no Register File, como explicado no Laboratório 15. (Isto não é mostrado na figura, mas pode analisá-lo na sua própria simulação).
 - o **Ciclo $i+15$:** O processador escreve o resultado em t_3 : `t3 = 0x2AB0`.
- Finalmente, o processador escreve o valor de t_3 na memória externa DDR (quadrado vermelho):
 - o **Ciclo $i+16$:** o endereço efetivo e os dados são enviados para a memória externa através do bus AXI:
 - `lsu_axi_awvalid = 1`

- `lsu_axi_awaddr = 0x00002AB0`
 - `lsu_axi_wvalid = 1`
 - `lsu_axi_wdata = 0x0000000000002AB0`
- **Ciclo i+21:** (mais uma vez, esta latência é diferente na simulação e na placa), a Memória Externa notifica através do barramento AXI que a escrita foi corretamente efetuada:
 - `lsu_axi_bvalid = 1`
 - `lsu_axi_bresp = 00` (definido como: tudo funcionou corretamente)

TAREFA: Usando os contadores HW, medir o número de ciclos, instruções, leituras e escritas no programa da Figura 2. Quanto tempo no total (tanto para leitura como para escrita) é necessário para aceder à memória externa DDR? Pode comparar a execução quando utiliza a memória DDR como na Figura 3 e quando se utiliza o DCCM (outro projeto PlatformIO é fornecido em `[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_DCCM/`, que contém o mesmo programa preparado para ler/escrever no DCCM). Lembre-se de que a memória simulada não é a mesma que a memória DDR real na placa Nexys A7, portanto a latência de leitura/escrita observada na simulação e na execução na placa é diferente.

TAREFA: Utilize o exemplo de `[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_ExtMem` para estimar a latência de leitura da memória externa DDR utilizando os contadores HW. Tal como na tarefa anterior, pode utilizar o exemplo de `[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_DCCM` para comparar com um programa sem paragens (*stalls*) devido aos acessos à memória. Lembre-se de que a memória simulada não é a mesma que a memória DDR real na placa Nexys A7, portanto a latência de leitura observada na simulação e na execução na placa é diferente.

TAREFA: Um exercício bastante complexo, mas muito interessante é analisar o controlador de memória usado no sistema RVfpga. Lembre-se que pode encontrar os módulos que compõem este controlador na pasta `[RVfpgaPath]/RVfpga/src/LiteDRAM`, e que o módulo de topo está implementado no ficheiro `litedram_top.v` dentro dessa pasta. Pode começar com a simulação a partir da Figura 3 e adicionar e analisar alguns sinais do controlador da LiteDRAM.

3. FETCH DE INSTRUÇÕES NA CACHE DE INSTRUÇÕES

Nesta secção analisamos o funcionamento da Cache de Instruções (I\$) disponível no sistema RVfpga. Primeiro descrevemos como a I\$ pode ser configurada (Secção 3.A) e depois investigamos como são processados os *misses* e *hits* da cache (Secções 3.B e 3.C) e, finalmente, analisamos a política de substituição da I\$ utilizada no SweRV EH1 (Secção 3.D).

A. Configuração da cache de instruções

O I\$ do sistema RVfpga é altamente configurável com base num conjunto de parâmetros definidos no ficheiro

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.
vh. O sistema RVfpga padrão tem os seguintes parâmetros I\$:

```
`define RV_ICACHE_SIZE 16
`define RV_ICACHE_DATA_CELL ram_256x34
`define RV_ICACHE_IC_INDEX 8
`define RV_ICACHE_TAG_CELL ram_64x21
`define RV_ICACHE_ENABLE 1
`define RV_ICACHE_IC_ROWS 256
`define RV_ICACHE_TAG_DEPTH 64
`define RV_ICACHE_TAG_HIGH 12
`define RV_ICACHE_TAG_LOW 6
`define RV_ICACHE_IC_DEPTH 8
`define RV_ICACHE_TADDR_HIGH 5
```

No entanto, alguns dos parâmetros acima referidos são substituídos no ficheiro
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/global.h:

```
localparam ICACHE_TAG_HIGH = `RV_ICACHE_TAG_HIGH;
localparam ICACHE_TAG_LOW = `RV_ICACHE_TAG_LOW;
localparam ICACHE_IC_DEPTH = `RV_ICACHE_IC_DEPTH;
localparam ICACHE_TAG_DEPTH = `RV_ICACHE_TAG_DEPTH;
```

Assim, a I\$ tem a seguinte configuração:

Característica	Valor
Tamanho I\$	
Conjunto de dados (sem informação de paridade) Informação de paridade para dados:	16 Kibytes 1 Kibyte (4 Bytes por bloco)
Conjunto de etiquetas (sem informação de paridade) Informação de paridade para tags	640 Bytes 32 bytes (1 bit por etiqueta)
Estado LRU	24 Bytes (3 bits por conjunto)
Bit Valid	32 bytes (1 bit válido por etiqueta)
Associatividade (não configurável)	4 vias
Tamanho do bloco	64 Bytes
Número de blocos (Tamanho/Tamanho do bloco=16Ki/64)	256 blocos
Número de blocos por via (# blocos/Atribuição =256/4)	64 blocos

De acordo com esta configuração, a I\$ utilizada no sistema RVfpga é apresentada na Figura 4.

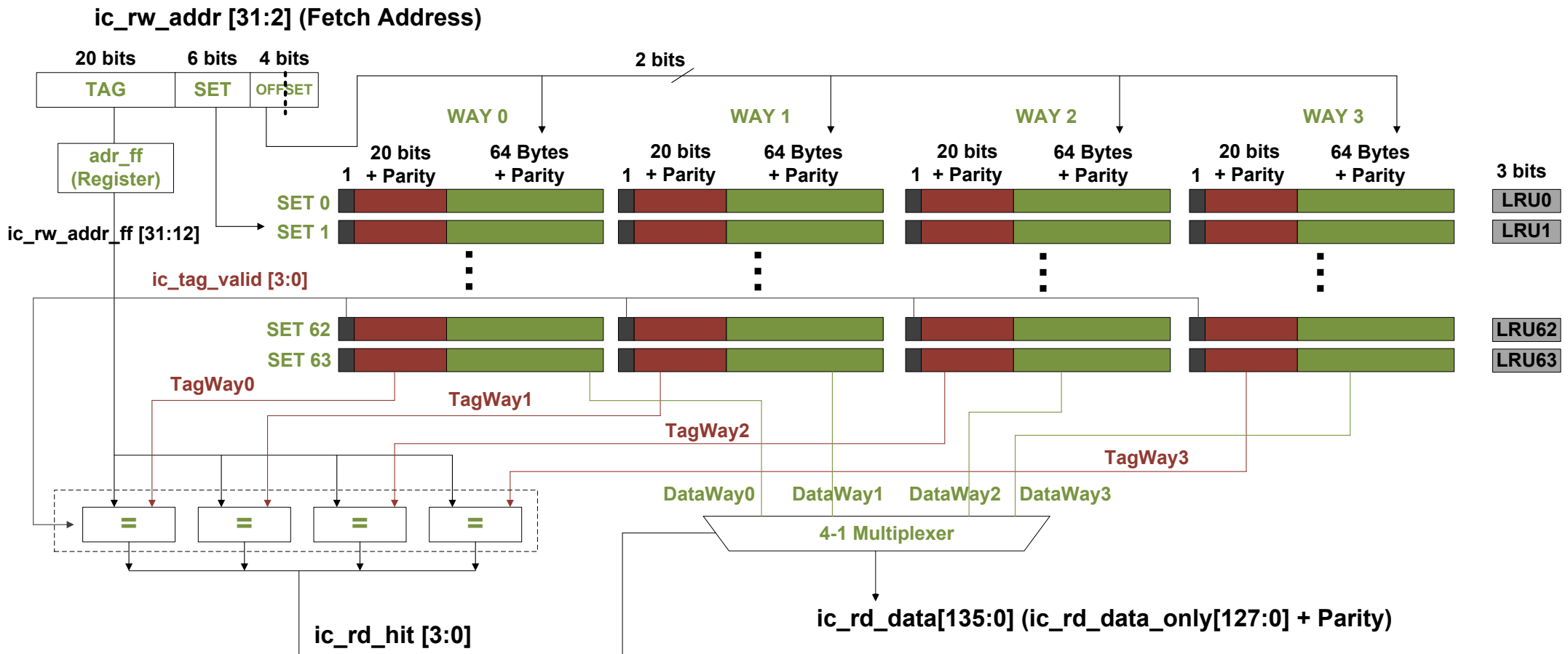


Figura 4. Conceção interna da I\$. O sinal de entrada para a I\$ (**ic_rw_addr**) e o sinal de saída da I\$ (**ic_rd_data**) são fornecidos de/para o controlador de cache (módulo *ifu_mem_ctl*), como explicámos na Figura 3 do Lab 11 (repetido abaixo como Figura 8).

A I\$ do sistema RVfpga é implementada no módulo **ifu_ic_mem**, incluído no ficheiro *[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/ifu/ifu_ic_mem.sv*. Este módulo instancia dois outros módulos:

- **IC_TAG**: Este módulo inclui a matriz de etiquetas (as caixas vermelhas indicadas na Figura 4) e a lógica para calcular o sinal de *hit*, *ic_rd_hit*. O módulo recebe, entre outros sinais, o endereço, *ic_rw_addr*. Tem como saída, entre outros sinais, o sinal *ic_rd_hit*, que é utilizado pelo Data Array para seleccionar a via da cache onde ocorre o *hit*.

As etiquetas lidas quando é efetuado um acesso à I\$ são fornecidas nos sinais *TagWay0-TagWay3*, como se mostra na Figura 4 e na simulação (apresentada mais à frente). Note-se que o processador utiliza um sinal chamado *w_tout* para ler as etiquetas. Os sinais *TagWay0-TagWay3* são extraídos de *w_tout* nas linhas 583-590 do ficheiro *ifu_ic_mem.sv*.

- **IC_DATA**: Este módulo é a matriz de dados, que inclui as caixas verdes apresentadas na Figura 4 bem como o multiplexer 4:1 que selecciona os dados da via onde ocorre o *hit*. Cada via está fisicamente dividida em 4 bancos (não mostrados na figura). O módulo recebe, entre outros sinais, o endereço de Fetch (*ic_rw_addr*) e o sinal de *hit* do módulo IC_TAG (*ic_rd_hit*). Com base no campo SET de 6 bits e em 2 bits do campo OFFSET do Fetch Address, este módulo selecciona e emite, no sinal *ic_rd_data*, o pacote de instruções de 128 bits mais alguns bits de paridade que devem ser enviados para o processador SweRV EH1. Note-se que o sinal *ic_rd_data_only* é o mesmo que o sinal *ic_rd_data* sem a informação de paridade, pelo que o utilizamos nas simulações que se seguem.

Os dados lidos da I\$ estão nos sinais *DataWay0-DataWay3*. Note-se que estes sinais são utilizados tanto na figura como na simulação, mas não são utilizados pelo processador. O sinal real do processador chama-se *wb_dout_way_with_premux*, a partir do qual os sinais *DataWay0-DataWay3* são obtidos nas linhas 313-320 do ficheiro *ifu_ic_mem.sv*.

TAREFA: Analisar o módulo **ifu_ic_mem** para entender como os elementos da Figura 4 são implementados.

B. Gestão de Miss na Cache de Instruções

Nesta secção, mostramos como os erros (*miss*) de instrução são geridos no processador. O exemplo da Figura 5 ilustra um programa que inclui 16 instruções *add* sequenciais e não comprimidas (que ocupam $4 \times 16 = 64$ bytes), mostradas a vermelho na figura, dentro de um ciclo com 0x10000 iterações. Várias instruções *nop* são colocadas antes das 16 instruções *add* para forçar as 16 instruções *add* a serem mapeadas num único bloco I\$. Recorde-se que o tamanho do bloco I\$ é de 64 bytes. Assim, a primeira instrução *add* deve ser alinhada num limite de 64 bytes. A pasta *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example* fornece o projeto PlatformIO para que possa analisar, simular e modificar o programa como desejar.

```

Test_Assembly:

    INSERT_NOPS_3
    INSERT_NOPS_8
    INSERT_NOPS_8

    li t6, 0x10000

REPEAT:
    add t6, t6, -1

    add t0, t0, t0
    add t1, t1, t1
    add t2, t2, t2
    add t3, t3, t3
    add t4, t4, t4
    add t5, t5, t5
    add t6, t6, t6
    add a7, a7, a7
    add t0, t0, t0
    add t2, t2, t2
    add t1, t1, t1
    add t3, t3, t3
    add t4, t4, t4
    add t6, t6, t6
    add t5, t5, t5
    add a7, a7, a7

    INSERT_NOPS_8
    INSERT_NOPS_8

    INSERT_NOPS_8
    INSERT_NOPS_8
    INSERT_NOPS_8
    INSERT_NOPS_8

    bne t6, zero, REPEAT    # Repete o ciclo

ret

```

Figura 5. Programa de exemplo

Abra o projeto no PlatformIO, compile-o e abra o ficheiro Disassembly (disponível em *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example/.pio/build/swervolf_nexys/firmware.dis*). Repare que a primeira instrução `add` (0x005282b3) é colocada no endereço 0x000001c0 (que está alinhado num limite de 64 bytes) e a décima sexta (0x011888b3) é colocada no endereço 0x000001fc (a última palavra do bloco).


0x000001c0:	005282b3	add	t0, t0, t0
...
0x000001fc:	011888b3	add	a7, a7, a7

Figura 6 mostra a simulação da região em torno das 16 instruções de `add` (de 28900 ps a 30220 ps). A figura do meio (principal) mostra a execução da região de interesse para a nossa análise. As figuras em cima e as duas em baixo fazem zoom em regiões específicas da figura principal.

TAREFA: Replicar a simulação da Figura 6 no seu próprio computador. Para o fazer, siga os seguintes passos (descritos em pormenor na Secção 7 do GSG):

- Se necessário, gere o binário de simulação (*Vrvfpgasim*).
- No PlatformIO, abra o projeto fornecido em:

[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example.

- Atualizar o caminho para o executável da simulação RVfpga (*Vrvfpgasim*) no ficheiro *platformio.ini*.
- Gerar o *trace* da simulação com o Verilator (*Generate Trace*).
- Abrir o *trace* no GTKWave.
- Use o ficheiro *test1_Miss.tcl* (fornecido em *[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example*) para abrir os mesmos sinais que os mostrados na Figura 6. Para isso, no GTKWave, clique em *File → Read Tcl Script File* e selecione o ficheiro *test1_Miss.tcl*.
- Clicar várias vezes em *Zoom In* () e analisar a região de 28900 ps a 30220 ps.

Também é possível analisar algumas coisas com mais pormenor, como a escrita na I\$ ou o bypass das instruções iniciais.

Este exemplo ilustra a forma como um *miss* de I\$ é tratado no SweRV EH1. Mostra a busca das 16 instruções de *add* na primeira vez que são executadas. Quando estas instruções ainda não estão na I\$ e têm de ser copiadas da memória externa DDR para a I\$.


- Na figura em cima, pode ver que um *miss* de I\$ é assinalado por volta dos 29ns (*ic_act_miss_f2* = 1), o que desencadeia o pedido do bloco através do barramento AXI (*ifu_axi_arvalid* = 1).
- Em seguida, os oito blocos de 64 bits que compõem o bloco de destino são solicitados sequencialmente através do barramento AXI.
 - o O sinal *ifu_axi_arvalid* fica tivo durante 27 ciclos. Este sinal indica que o canal está a sinalizar um endereço de leitura e informação de controlo válidos.
 - o Durante estes 27 ciclos em que *ifu_axi_arvalid* = 1, os endereços iniciais dos oito blocos de 64 bits são fornecidos sequencialmente através do barramento AXI utilizando o sinal *ifu_axi_araddr*, que fornece os 8 endereços que devem ser lidos da memória DDR:
 - *ifu_axi_araddr* = 0x000001c0
 - *ifu_axi_araddr* = 0x000001c8
 - *ifu_axi_araddr* = 0x000001d0
 - *ifu_axi_araddr* = 0x000001d8
 - *ifu_axi_araddr* = 0x000001e0
 - *ifu_axi_araddr* = 0x000001e8
 - *ifu_axi_araddr* = 0x000001f0
 - *ifu_axi_araddr* = 0x000001f8
- A figura do meio mostra os oito blocos de 64 bits que chegam sequencialmente ao processador através do barramento AXI no sinal *ifu_axi_rdata*.
 - o O sinal *ifu_axi_rvalid*, que indica que o canal está a sinalizar os dados de leitura necessários, fica alto durante um ciclo a cada 7 ciclos.
 - o Cada um dos oito blocos de 64 bits (cada um contendo duas instruções) é fornecido no sinal *ifu_axi_rdata* (isto não pode ser visto na Figura 6 - pode reproduzir a simulação no seu computador para o verificar):
 - *ifu_axi_rdata* = 0x00630333005282b3
 - *ifu_axi_rdata* = 0x01ce0e33007383b3
 - *ifu_axi_rdata* = 0x01ef0f3301de8eb3
 - *ifu_axi_rdata* = 0x011888b301ff8fb3
 - *ifu_axi_rdata* = 0x007383b3005282b3
 - *ifu_axi_rdata* = 0x01ce0e3300630333
 - *ifu_axi_rdata* = 0x01ff8fb301de8eb3
 - *ifu_axi_rdata* = 0x011888b301ef0f33
- As duas figuras inferiores mostram que cada um dos oito blocos de 64 bits é escrito na I\$ logo após a sua chegada ao controlador de cache. Por exemplo, os dois primeiros blocos de 64 bits são escritos da seguinte forma:
 - o O sinal *ic_wr_en* fica alto e, ao mesmo tempo, *ic_wr_data* = 0x00630333005282b3. Assim, a primeira e a segunda instruções *add* são escritas na I\$.
 - o Vários ciclos mais tarde, o sinal *ic_wr_en* volta a ficar alto e, ao mesmo tempo, *ic_wr_data* = 0x01ce0e33007383b3. Assim, a terceira e a quarta instruções *add* são escritas na I\$.

- Finalmente, pode ver que as quatro instruções são desviadas do controlador I\$ para o pipeline (sinais `ifu_byp_data_first_half` e `ifu_byp_data_second_half`) para que este possa reiniciar a execução o mais rapidamente possível após um *miss* de I\$. Vários ciclos mais tarde, as quatro instruções chegam ao andar de Decode (ver a figura no canto inferior direito que faz zoom nos sinais `dec_i0_instr_d` e `dec_i1_instr_d`).

C. Gestão de *Hits* da Cache de Instruções

Nesta secção, trabalhamos com o mesmo exemplo da Secção 3.B (Figura 5), mas centramo-nos agora na análise dos *hits* de I\$. Figura 7 mostra a segunda iteração do ciclo quando se executa o programa da Figura 5 (qualquer iteração seria válida, exceto a primeira, que, como analisámos na Figura 6 que, como analisámos na Figura 6, sofre *misses* na I\$).

TAREFA: Replicar a simulação da Figura 7 no seu próprio computador. Utilize o ficheiro `test1_Hit.tcl` (fornecido em

`[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example`). Aumentar o zoom () várias vezes e passar para 34680ps.

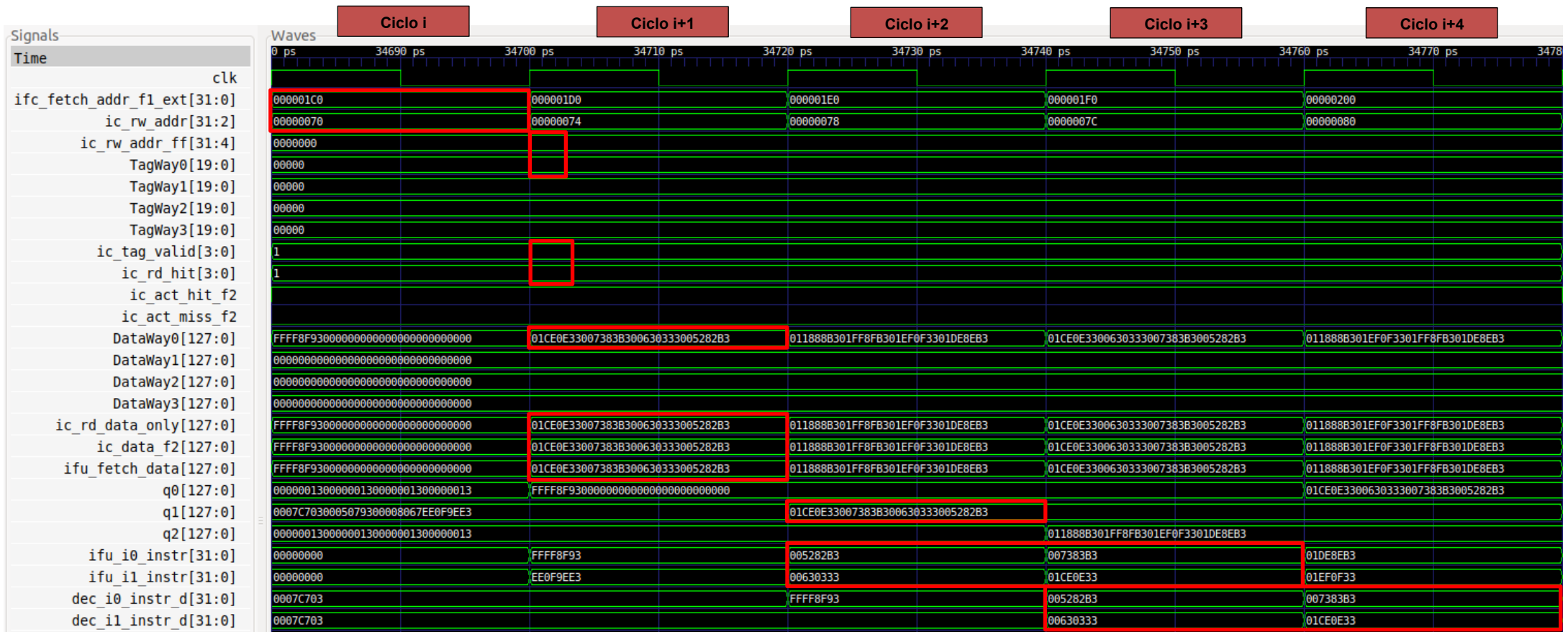


Figura 7. Forma de onda de simulação para o programa da Figura 5 ilustrando um acerto I\$

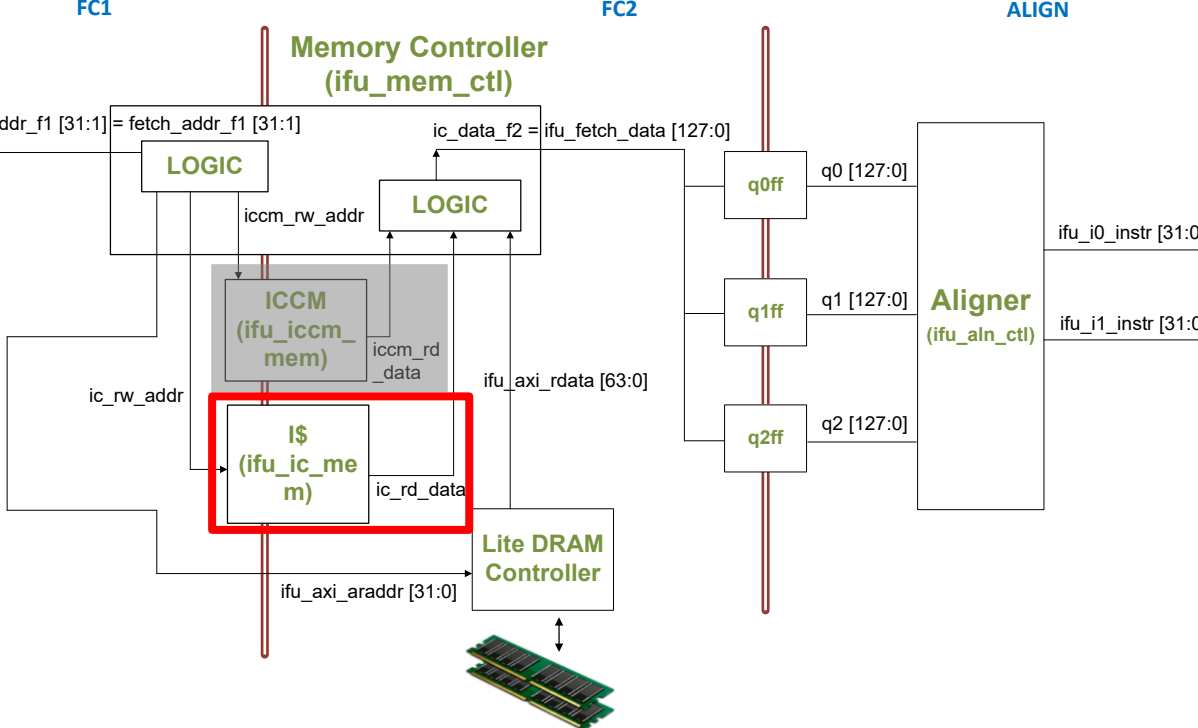


Figura 8. Andares FC1, FC2 e Align

O *hit* de i\$ ocorre da seguinte forma, como mostra a Figura 7:

- **Ciclo i:** O endereço da primeira instrução de adição (add t0,t0,t0) no programa de Figura 5 é dado no sinal `ifc_fetch_addr_fl_ext` (`ifc_fetch_addr_fl_ext = 0x000001c0`). Este sinal é passado para a I\$ exceto nos seus dois bits menos significativos, que são necessários porque as instruções são alinhadas em 4 bytes (32 bits). Assim, `ic_rw_addr = 0x0000070`.

O Tag Array e o Data Array são acedidos utilizando um subconjunto do endereço de Fetch, como mostra na Figura 4. O resultado do acesso estará disponível no ciclo seguinte.

- **Ciclo i+1:** As quatro etiquetas, uma por via da cache, estão nos sinais TagWay0-TagWay3. Estes são comparados com o campo TAG do endereço de pesquisa que foi registado em `adr_ff` (sinal de saída `ic_rw_addr_ff`). Neste caso, todas as etiquetas são iguais ao campo TAG, mas apenas uma via (Via 0) é válida (`ic_tag_valid = 0001`), pelo que é assinalado um acerto na Via 0: `ic_rd_hit = 0001`.

Além disso, quatro pacotes de 128 bits estão nos sinais `DataWay0`–`DataWay3`. O multiplexer 4:1 da Figura 4 seleciona os dados fornecidos pela via 0 (ou seja, `DataWay0`). Assim:

```
ic_rd_data_only = 0x01ce0e33007383b300630333005282b3
```

Note-se que o sinal que é mostrado na Figura 8 é `ic_rd_data`, que é o mesmo que `ic_rd_data_only` junto com a informação de paridade.

Estes 128 bits são propagados para os andares de Align como mostra a Figura 8.

```
ifu_fetch_data = ic_data_f2 = ic_rd_data_only =  
0x01ce0e33007383b300630333005282b3
```

Note-se que estes 128 bits correspondem às primeiras quatro instruções de adição.

- **Ciclo i+2:** A primeira e a segunda instruções `add` são extraídas no andar de alinhamento do buffer `q1`:
 - o `ifu_i0_instr` = 0x005282b3
 - o `ifu_i1_instr` = 0x00630333
- **Ciclo i+3:** A terceira e a quarta instruções `add` são extraídas no andar de Align e, ao mesmo tempo, a primeira e a segunda instruções `add` são decodificadas:
 - o `ifu_i0_instr` = 0x007383b3
 - o `ifu_i1_instr` = 0x01ce0e33
 - o `dec_i0_instr_d` = 0x005282b3
 - o `dec_i1_instr_d` = 0x00630333
- **Ciclo i+4:** Finalmente, a terceira e a quarta instruções `add` são decodificadas:
 - o `dec_i0_instr_d` = 0x007383b3
 - o `dec_i1_instr_d` = 0x01ce0e33

D. Política de Substituição na Cache de Instruções

Esta secção descreve a política de substituição na cache do sistema RVfpga. Tal como explicado por Harris & Harris na Secção 8.3.3 de [DDCARV], nas caches associativas (*set associative*), a cache tem de escolher o bloco a despejar quando um conjunto (*set*) de cache está cheio. O princípio da localidade temporal sugere que a melhor escolha é despejar o bloco utilizado menos recentemente, porque é menos provável que volte a ser utilizado em breve. Por isso, a maioria das caches associativas tem uma política de substituição do bloco usado menos recentemente (Least Recently Used - LRU). No entanto, o rastreio da via menos recentemente utilizada torna-se complicado, pelo que as políticas LRU aproximadas (normalmente designadas por Pseudo LRU) são frequentemente utilizadas e suficientemente boas na prática. Especificamente, o SweRV EH1 utiliza uma política aproximada denominada **Pseudo LRU de árvore binária**.

NOTA: Se ainda não o fez, leia a Secção 8.3.3 de [DDCARV]. Além disso, recomendamos a leitura da Secção 4 da tese de mestrado de Gille Damien, "Study of Different Cache Line Replacement Algorithms in Embedded Systems" (8 de março de 2007), que pode encontrar online em: <https://people.kth.se/~ingo/MasterThesis/ThesisDamienGille2007.pdf>. Referimo-nos a esse documento como [GiDa].

i. Implementação da Política Pseudo LRU em Árvore Binária no SweRV EH1

Como explicado em [GiDa], uma política LRU em árvore binária, que é uma aproximação de uma política LRU, que requer N-1 bits por conjunto (a que chamamos Estado LRU) numa cache associativa de N-vias. Assim, no caso do SweRV EH1, onde é usada uma cache de instruções de 4 vias, são necessários 3 bits por conjunto para seguir o histórico de acesso às diferentes vias.

Como explicado na Secção 3.B, quando ocorre um *miss* na I\$, o bloco deve ser solicitado à Memória Externa DDR. Quando a Memória Externa DDR fornece o bloco de cache, este deve ser escrito na I\$. O campo SET do Fetch Address determina o conjunto de I\$ onde o novo bloco deve ser escrito (ver Figura 4). Duas coisas podem acontecer:

- O conjunto não está completo, o que significa que um ou mais blocos não são válidos. Neste caso, o novo bloco é escrito na via mais baixa que contém um bloco não-válido.
- O conjunto está completo, o que significa que os quatro blocos são válidos. No nosso processador, a Política de Substituição LRU em Árvore Binária determina qual bloco deve ser expulso. Essa política determina a via de substituição com base no estado LRU de 3 bits do conjunto, de acordo com a tabela a seguir (onde x significa *don't care*):

Estado LRU	Via a substituir
x00	Via 0
x10	Via 1
0x1	Via 2
1x1	Via 3

O seguinte trecho de Verilog (Figura 9), extraído do módulo **ifu_mem_ctl**, implementa a lógica para a seleção da via que deve ser usada para armazenar o novo bloco I\$, de acordo com a explicação anterior.

```

545 assign replace_way_mb_any[3] = ( way_status_mb_ff[2] & way_status_mb_ff[0] & (&tagv_mb_ff[3:0])) |
546 (~tagv_mb_ff[3] & tagv_mb_ff[2] & tagv_mb_ff[1] & tagv_mb_ff[0]) ;
547 assign replace_way_mb_any[2] = (~way_status_mb_ff[2] & way_status_mb_ff[0] & (&tagv_mb_ff[3:0])) |
548 (~tagv_mb_ff[2] & tagv_mb_ff[1] & tagv_mb_ff[0]) ;
549 assign replace_way_mb_any[1] = ( way_status_mb_ff[1] & ~way_status_mb_ff[0] & (&tagv_mb_ff[3:0])) |
550 (~tagv_mb_ff[1] & tagv_mb_ff[0] ) ;
551 assign replace_way_mb_any[0] = (~way_status_mb_ff[1] & ~way_status_mb_ff[0] & (&tagv_mb_ff[3:0])) |
552 (~tagv_mb_ff[0] ) ;
553

```

Figura 9. Código Verilog para selecionar a via que deve ser substituída

Os sinais utilizados no excerto Verilog da Figura 9 são os seguintes:

- **replace_way_mb_any** (4 bits): contém um valor pontual que é 1 para a via que deve ser substituída.
- **way_status_mb_ff** (3 bits): contém o estado LRU do conjunto do novo bloco.
- **tagv_mb_ff** (4 bits): contém os bits válidos do conjunto do novo bloco; as vias válidas têm bits válidos de 1, enquanto as vias inválidas têm bits válidos de 0.

TAREFA: Analisar o código Verilog da Figura 9 e explique como ele funciona com base

nas explicações acima.

Quando ocorre um *hit* ou um *miss* na I\$, o estado LRU do conjunto deve ser atualizado de acordo com a tabela seguinte (em que "-" significa que o bit permanece inalterado):

Via Escrita	Próximo Estado LRU
Via 0	-11
Via 1	-01
Via 2	1-0
Via 3	0-0

Se analisar esta tabela, verá que, como explicado por [GiDa], após um *hit* ou *miss*, os bits no caminho em direção à linha de *hit*/inserção são invertidos para indicar a parte oposta da árvore como pseudo LRU. A ideia subjacente é proteger os últimos dados acedidos do despoje, invertendo os nós no seu caminho.

O seguinte trecho de Verilog (Figura 10), extraído do módulo **ifu_mem_ctl**, implementa a lógica para esta atualização do estado LRU.

```

554 assign way_status_hit_new[2:0] = ({3{ic_rd_hit[0]} & {way_status[2], 1'b1, 1'b1}} |
555 {3{ic_rd_hit[1]} & {way_status[2], 1'b0, 1'b1}} |
556 {3{ic_rd_hit[2]} & {1'b1, way_status[1], 1'b0}} |
557 {3{ic_rd_hit[3]} & {1'b0, way_status[1], 1'b0}});
558
559 assign way_status_rep_new[2:0] = ({3{replace_way_mb_any[0]} & {way_status_mb_ff[2], 1'b1, 1'b1}} |
560 {3{replace_way_mb_any[1]} & {way_status_mb_ff[2], 1'b0, 1'b1}} |
561 {3{replace_way_mb_any[2]} & {1'b1, way_status_mb_ff[1], 1'b0}} |
562 {3{replace_way_mb_any[3]} & {1'b0, way_status_mb_ff[1], 1'b0}});
563
564 // Make sure to select the way_status_hit_new even when in hit under miss.
565 assign way_status_new[2:0] = (ifu_wr_en_new_q) ? way_status_rep_new[2:0] :
566 way_status_hit_new[2:0];
567

```

Figura 10. Trecho de Verilog para atualização do estado LRU

Os sinais utilizados no trecho Verilog da Figura 10 são os seguintes:

- **ic_rd_hit** (4 bits): indica a via em que ocorreu um *hit*.
- **way_status** e **way_status_mb_ff** (3 bits cada): indicam o estado LRU anterior do conjunto em que se registou a resposta positiva ou a substituição.
- **ifu_wr_en_new_q** (1 bit): é 1 se tiver ocorrido uma substituição.
- **way_status_new** (3 bits): contém o novo estado LRU para o conjunto que acaba de ser referenciado num *hit* ou num *miss*.
- **replace_way_mb_any** (4 bits): contém um valor em *one-hot* que é 1 para a via que deve ser substituída. Este sinal também foi explicado a seguir Figura 9.

TAREFA: Analisar o código Verilog da Figura 10 e explique como ele funciona com base nas explicações acima.

com a divisão de endereços mostrada na Figura 4 para aceder à I\$:

!\$ Endereço em binário = 00000000000000000001100100000000

TAG = 0x3

SET = 0x8

OFFSET = 0x0

- A quinta instrução `j` (`j Set8_Block1`) está no endereço `0x00004200`. De acordo com a divisão de endereços mostrada na Figura 4 para aceder à `I$`:

!\$ Endereço em binário = 000000000000000010001000000000

TAG = 0x4

SET = 0x8

OFFSET = 0x0

Neste programa (Figura 11), quando a primeira iteração é executada, o Set 8 está inicialmente vazio. Figura 12 mostra as alterações teóricas do Set 8 na I\$ durante a execução da primeira iteração. Mais adiante, mostramos várias simulações do Verilator que confirmam estas explicações teóricas.

SET 8 after execution of the first j instruction at 0x200

Valid	Tag	Data		
1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0	LRU STATE = 011
0			WAY 1	
0			WAY 2	
0			WAY 3	

SET 8 after execution of the second j instruction at 0x1200

1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0	LRU STATE = 001
1	00000000000000000001	j Set8_Block3 nop ... nop	WAY 1	
0			WAY 2	
0			WAY 3	

SET 8 after execution of the third j instruction at 0x2200

1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0	LRU STATE = 100
1	00000000000000000001	j Set8_Block3 nop ... nop	WAY 1	
1	00000000000000000010	j Set8_Block4 nop ... nop	WAY 2	
0			WAY 3	

SET 8 after execution of the fourth j instruction at 0x3200

1	00000000000000000000	j Set8_Block2 nop ... nop	WAY 0	LRU STATE = 000
1	00000000000000000001	j Set8_Block3 nop ... nop	WAY 1	
1	00000000000000000010	j Set8_Block4 nop ... nop	WAY 2	
1	00000000000000000011	j Set8_Block5 nop ... nop	WAY 3	

SET 8 after execution of the fifth j instruction at 0x4200

1	000000000000000000100	j Set8_Block1 nop ... nop	WAY 0	LRU STATE = 011
1	000000000000000000001	j Set8_Block3 nop ... nop	WAY 1	
1	000000000000000000010	j Set8_Block4 nop ... nop	WAY 2	
1	0000000000000000000011	j Set8_Block5 nop ... nop	WAY 3	

Figura 12. Set 8 da I\$ durante a execução da primeira iteração do ciclo na Figura 11

As simulações do Verilator a seguir mostram os sinais da cache durante a primeira iteração do ciclo e confirmam a análise mostrada na Figura 12. Figura 13 mostra a simulação do Verilator do programa após a execução da primeira instrução *j* (*j* Set8_Block2). Mais uma vez, o endereço desta instrução (0x200) mapeia para o Set 8 do I\$. Esse conjunto está inicialmente vazio: *tagv_mb_ff* = 0000. Assim, de acordo com a política LRU da Árvore Binária, o novo bloco deve ser escrito na Via 0: *replace_way_mb_any* = *ic_wr_en* = 0001. O estado LRU do Set 8 é atualizado do seguinte modo: *way_status_new* = 011.

Recorde-se da Secção 3.B que o bloco é lido da memória DDR e escrito na I\$ em blocos de 64 bits. Figura 13 ilustra a escrita da etiqueta e das duas primeiras instruções do novo bloco no SET 8:

```
ic_rw_addr_q[11:4] = 00100000 (SET 8)
ic_tag_wr_data[19:0] = 0x0 (o bit mais significativo é utilizado para correção
de erros e não está incluído aqui)
ic_wr_data1[31:0] = 0x0000106F (j Set8_Block2)
ic_wr_data2[31:0] = 0x00000013 (nop)
```

(*ic_wr_data1* e *ic_wr_data2* são sinais criados por uma questão de clareza, mas o sinal utilizado na I\$ chama-se *ic_wr_data*[67:0], que inclui as duas instruções mais alguma informação de paridade).

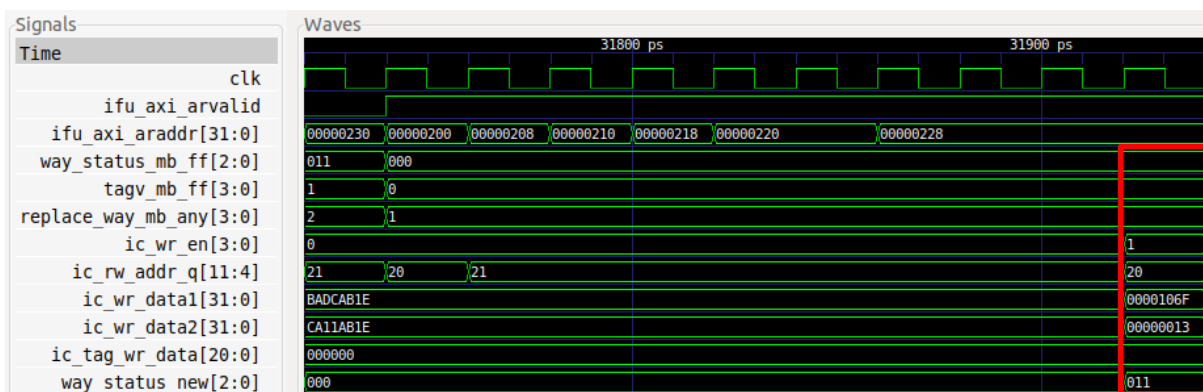


Figura 13. Estado de LRU do Set 8 após a execução da primeira instrução *j*

Figura 14 ilustra a simulação do Verilator após a execução da segunda instrução *j* (*j* Set8_Block3). Este endereço de instrução (0x1200) também mapeia para o Set 8 da I\$. Apenas a via 0 é válida nesse conjunto: *tagv_mb_ff* = 0001. Assim, de acordo com a política LRU de Árvore Binária, o novo bloco deve ser escrito na Via 1: *replace_way_mb_any* = *ic_wr_en* = 0010. O estado de LRU do conjunto 8 é atualizado do seguinte modo: *way_status_new* = 001.

Como anteriormente, Figura 14 ilustra a escrita das duas primeiras instruções do novo bloco no SET 8:

```
ic_rw_addr_q[11:4] = 00100000 (SET 8)
ic_tag_wr_data[19:0] = 0x1 (o bit mais significativo é utilizado para correção
de erros e não está incluído aqui)
ic_wr_data1[31:0] = 0x0000106F (j Set8_Block3)
ic_wr_data2[31:0] = 0x00000013 (nop)
```

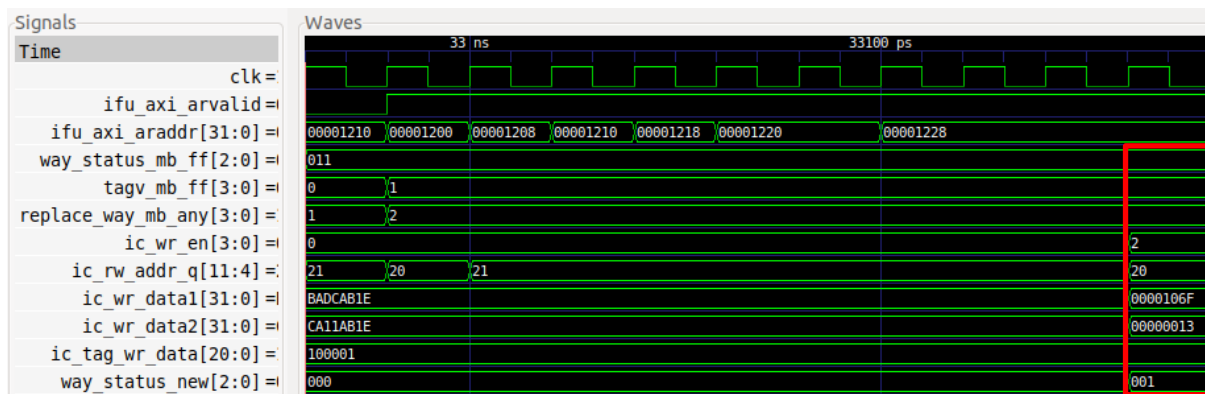


Figura 14. Estado de LRU do Set 8 após a execução da segunda instrução j

Figura 15 ilustra a simulação do Verilator após a execução da quinta instrução j (j Set8_Block1). O endereço desta instrução (0x4200) também mapeia para o Set 8 do I\$. No entanto, ao contrário da situação anterior, neste caso o conjunto está cheio: tagv_mb_ff = 1111. Assim, de acordo com a política de LRU da Árvore Binária, o novo bloco deve ser escrito na Via 1: replace_way_mb_any = ic_wr_en = 0001. O estado LRU do Set 8 é atualizado do seguinte modo: way_status_new = 011.

Como anteriormente, Figura 15 ilustra a escrita das duas primeiras instruções do novo bloco no SET 8:

```
ic_rw_addr_q[11:4] = 00100000 (SET 8)
ic_tag_wr_data[19:0] = 0x4 (o bit mais significativo é utilizado para correção de erros e não está incluído aqui)
ic_wr_data1[31:0] = 0x800fc06f (j Set8_Block1)
ic_wr_data2[31:0] = 0x00008067 (ret)
```

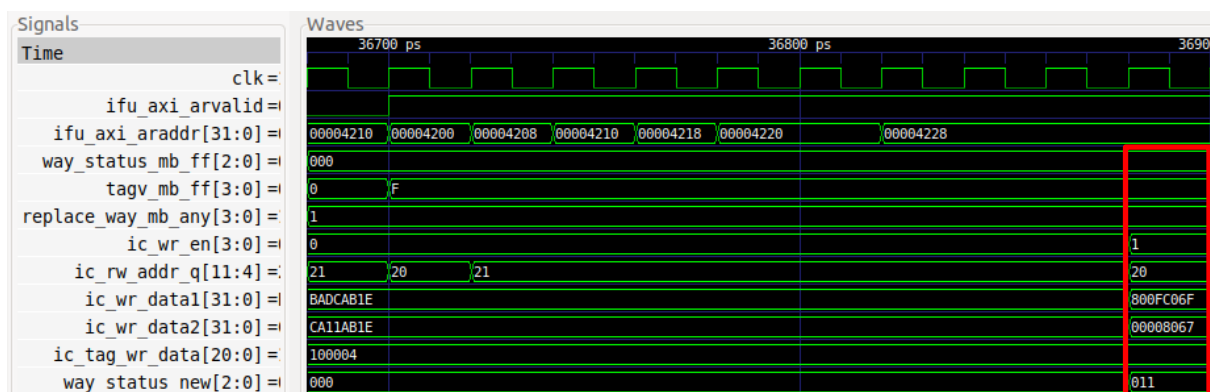


Figura 15. Estado de LRU do conjunto 8 após a execução da quinta instrução j

TAREFA: Replicar a simulação da Figura 13-Figura 15 no seu próprio computador.

4. EXERCÍCIOS

- 1) Transformar o ciclo infinito de Figura 11 num ciclo com 0x10000 iterações, mas mantendo as instruções `j` nos mesmos endereços. Meça o número de ciclos e *hits* e *misses* na I\$. Em seguida, remova uma das instruções `j` e meça as mesmas métricas. Compare e explique os resultados.
- 2) Utilizar o programa da Figura 5 para analisar um hit de I\$ do ponto de vista da política de substituição de I\$.
- 3) Estender a Figura 6 para analisar em pormenor como cada pedaço de 64 bits é escrito na I\$.
- 4) Analisar em simulação e na placa outras configurações de I\$, como uma I\$ com um tamanho de bloco diferente. Recorde-se que o número de vias não pode ser modificado.
- 5) Analisar a lógica que verifica a correção da informação de paridade do Data Array e do Tag Array.