



THE IMAGINATION UNIVERSITY PROGRAMME

RVfpga Lab 20

ICCM, DCCM e Benchmarking

1. INTRODUÇÃO

Neste laboratório, analisamos as memórias *scratchpad* (ICCM e DCCM) disponíveis no processador SweRV EH1 e, em seguida, fornecemos vários exemplos de benchmarking e exercícios para demonstrar alguns dos conceitos dos laboratórios 11 a 20.

Recorde-se da Figura 25 do Guia de Iniciação do RVfpga (que repetimos abaixo na Figura 1 por uma questão de conveniência), que o sistema RVfpga inclui duas memórias de scratchpad (destacadas a vermelho na figura): uma para dados, chamada Data Closely-Coupled Memory (DCCM), e outra para instruções, chamada Instruction Closely-Coupled Memory (ICCM).

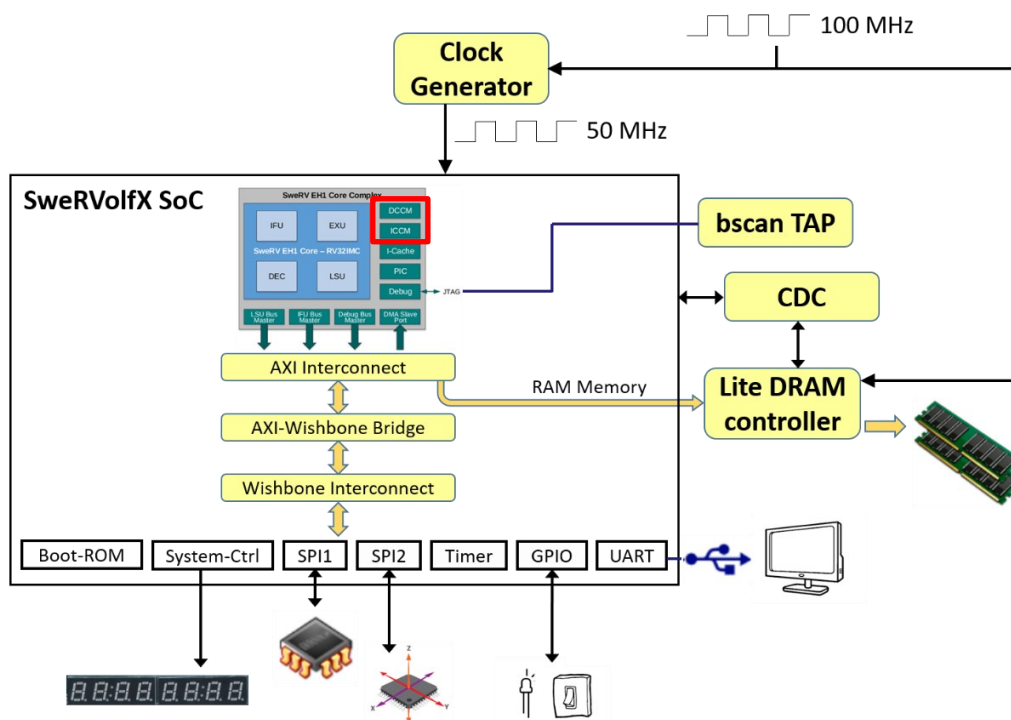
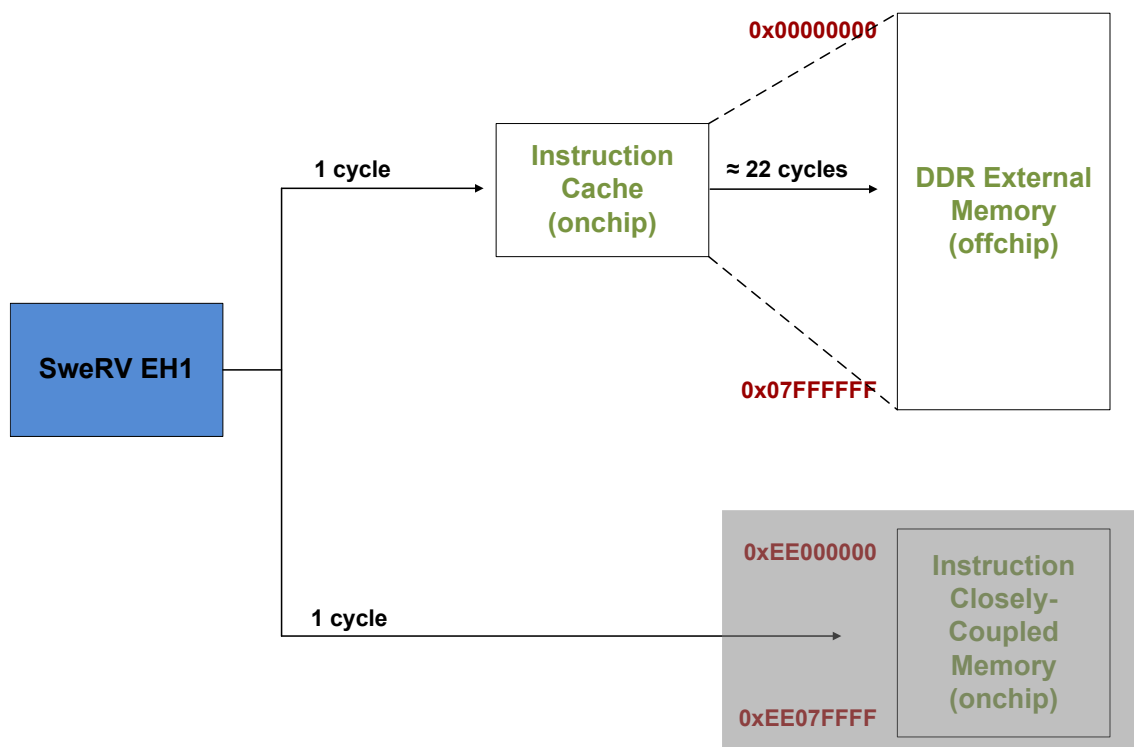


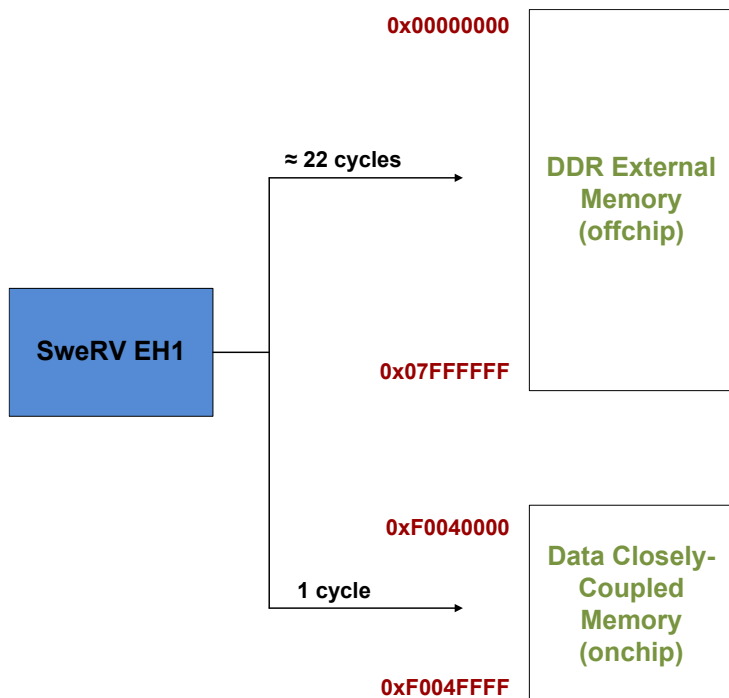
Figura 1. Sistema RVfpgaNexys

NOTA: Antes de começar a trabalhar neste laboratório, recomendamos a leitura das secções 1 e 3 do artigo de Preeti Ranjan Panda, Nikil D. Dutt e Alexandru Nicolau. "On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems". ACM Trans. Design Autom. Electr. Syst. 5(3): 682-704 (2000) (disponível em: <https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.2430&rep=rep1&type=pdf>). Este documento apresenta uma boa introdução à utilização de memórias Scratchpad em processadores embebidos.

O mapa de memória do sistema RVfpga foi descrito na Secção 4.B do Guia de Iniciação. A figura seguinte complementa essa descrição com uma ilustração do espaço de endereçamento ocupado pela memória de instruções (Figura 2a) e pela Memória de Dados (Figura 2b) disponíveis no sistema RVfpga.



(a) Espaço de endereçamento da memória de instruções, constituído por uma cache de instruções (I\$) e pela memória externa DDR. O ICCM está desativado no sistema predefinido.



(b) Espaço de endereçamento da memória de dados, constituído por uma DCCM e uma memória externa DDR.

Figura 2. Espaço de endereçamento do sistema RVfpga para memórias de instrução e de dados

Neste laboratório, concentramo-nos na configuração e no funcionamento das memórias de Dados/Instruções estreitamente acopladas (secções 2.A e 2.B, respetivamente) e, em seguida, apresentamos vários exemplos e exercícios de avaliação comparativa (secção 3), nos quais utilizamos programas elementares *ad-hoc* que ilustram situações específicas e aplicações reais.

2. MEMÓRIAS DE DADOS/INSTRUÇÕES ESTREITAMENTE ACOPLADAS (DCCM E ICCM)

Nesta secção, analisamos a memória de dados (DCCM) e a memória de instruções (ICCM) estreitamente acoplada disponíveis no sistema RVfpga. Começamos por descrever a forma como estas duas estruturas podem ser configuradas (Secção 3.A) e, em seguida, ilustramos como é efetuado um acesso à DCCM (Secção 3.B).

A. Configuração da DCCM e da ICCM no Sistema RVfpga

A DCCM e a ICCM do sistema RVfpga são altamente configuráveis com base num conjunto de parâmetros definidos no ficheiro

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh. O sistema RVfpga por omissão tem os seguintes parâmetros para estas duas estruturas:

DCCM:

```
`define RV_DCCM_EADR 32'hf004ffff
`define RV_DCCM_FDATA_WIDTH 39
`define RV_LSU_SB_BITS 16
`define RV_DCCM_SIZE 64
`define RV_DCCM_ECC_WIDTH 7
`define RV_DCCM_SADR 32'hf0040000
`define RV_DCCM_BYTE_WIDTH 4
`define RV_DCCM_NUM_BANKS 8
`define RV_DCCM_SIZE_64
`define RV_DCCM_NUM_BANKS_8
`define RV_DCCM_OFFSET 28'h40000
`define RV_DCCM_WIDTH_BITS 2
`define RV_DCCM_ENABLE 1
`define RV_DCCM_DATA_CELL ram_2048x39
`define RV_DCCM_RESERVED 'h1000
`define RV_DCCM_ROWS 2048
`define RV_DCCM_BANK_BITS 3
`define RV_DCCM_DATA_WIDTH 32
`define RV_DCCM_INDEX_BITS 11
`define RV_DCCM_BITS 16
`define RV_DCCM_REGION 4'hf
```

ICCM:

```
`define RV_ICCM_DATA_CELL ram_16384x39
`define RV_ICCM_BITS 19
`define RV_ICCM_ROWS 16384
`define RV_ICCM_INDEX_BITS 14
`define RV_ICCM_NUM_BANKS 8
`define RV_ICCM_NUM_BANKS_8
```

```

`define RV_ICCM_BANK_BITS 3
`define RV_ICCM_SIZE_512
`define RV_ICCM_RESERVED 'h1000
`define RV_ICCM_SIZE 512
`define RV_ICCM_REGION 4'he
`define RV_ICCM_OFFSET 10'he000000
`define RV_ICCM_SADR 32'hee000000
`define RV_ICCM_EADR 32'hee07ffff

```

No entanto, tal como na I\$, alguns dos parâmetros acima referidos são substituídos no ficheiro `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/global.h`:

DCCM:

```

localparam DCCM_BITS = `RV_DCCM_BITS;
localparam DCCM_BANK_BITS = `RV_DCCM_BANK_BITS;
localparam DCCM_NUM_BANKS = `RV_DCCM_NUM_BANKS;
localparam DCCM_DATA_WIDTH = `RV_DCCM_DATA_WIDTH;
localparam DCCM_FDATA_WIDTH = `RV_DCCM_FDATA_WIDTH;
localparam DCCM_BYTE_WIDTH = `RV_DCCM_BYTE_WIDTH;
localparam DCCM_ECC_WIDTH = `RV_DCCM_ECC_WIDTH;

```

ICCM:

```

localparam ICCM_SIZE = `RV_ICCM_SIZE;
localparam ICCM_BITS = `RV_ICCM_BITS;
localparam ICCM_NUM_BANKS = `RV_ICCM_NUM_BANKS;
localparam ICCM_BANK_BITS = `RV_ICCM_BANK_BITS;
localparam ICCM_INDEX_BITS = `RV_ICCM_INDEX_BITS;
localparam ICCM_BANK_HI = 4 + (`RV_ICCM_BANK_BITS/4);

```

Note-se que, como mostra a Figura 2 a DCCM está ativada no nosso sistema de base (`RV_DCCM_ENABLE = 1`), mas a ICCM está desativada (`RV_ICCM_ENABLE` não definido), pelo que não está incluído qualquer ICCM no SoC utilizado nos laboratórios anteriores.

Tabela 1 resume as configurações das ICCM e DCCM no sistema RVfpga.

Tabela 1. Configurações das DCCM e ICCM

Característica	Valor
DCCM	
Ativação	1
Espaço de endereçamento	0xF0040000 - 0xF004FFFF
Tamanho	64 KiB
Número de bancos	8
Dimensão do banco	2048x39 bits (7 bits para paridade)
ICCM	
Ativação	0

Figura 3 apresenta um diagrama de blocos da configuração do DCCM do RVfpga. Os sinais de entrada para a DCCM (`lsu_addr_dc1`, `end_addr_dc1`, `stbuf_addr_any`, `stbuf_ecc_any` e `stbuf_data_any`) e os sinais de saída da DCCM (`dccm_data_lo_dc2` e `dccm_data_hi_dc2`) são fornecidos de/para a Load Store Unit (lsu), como explicado no Lab 13 (ver Figuras 6 e 13 no Lab 13).

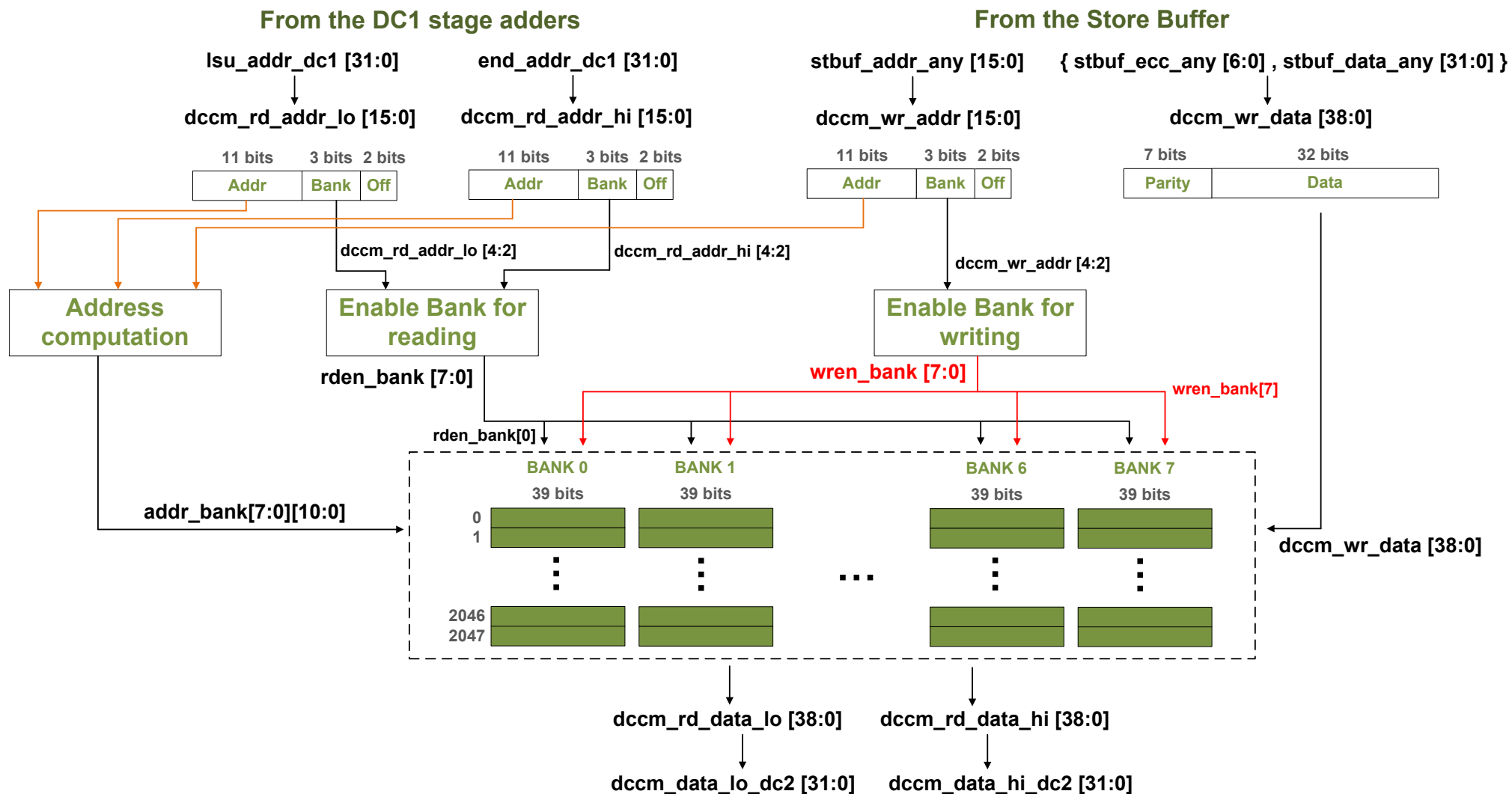


Figura 3. Arquitetura interna da DCCM.

A DCCM do sistema RVfpga está implementada no módulo `Isu_dccm_mem`, incluído no ficheiro

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/Isu/Isu_dccm_mem.sv`.

Como se pode ver na Figura 3 a DCCM está dividida em 8 bancos. São fornecidos dois endereços de leitura para suportar acessos não alinhados: `dccm_rd_addr_lo[15:0] = lsu_addr_dc1[15:0]` e `dccm_rd_addr_hi[15:0] = end_addr_dc1[15:0]`. Estes endereços estão logicamente divididos em 3 campos:

- **Bank:** Banco selecionado.
- **Addr:** Endereço da palavra de 32 bits lida dentro do banco.
- **Off:** Byte lido dentro da palavra de 32 bits.
- Note que são adicionados 7 bits de paridade a cada palavra de 32 bits.

Como também explicado no Lab 13 e como pode ser visto na Figura 3 um endereço de escrita é fornecido no sinal `dccm_wr_addr[15:0]` pelo Store Buffer (veja o apêndice do Lab 13 para maiores descrições do funcionamento do Store Buffer). O endereço de escrita é dividido como os endereços de leitura (veja o item anterior). Com base no campo de 3 bits desses endereços **Bank** (além de outros sinais não especificados na figura que analisará a seguir), são obtidos 8 bits de habilitação de leitura/escrita em `rden_bank[7:0]` e `wren_bank[7:0]`, respetivamente. Cada bit determina se o banco correspondente deve ser ativado ou desativado para leitura e escrita.

Com base no campo de 11 bits destes endereços **Addr** (e noutros sinais não especificados na figura que analisará numa tarefa abaixo), são obtidos oito endereços de 11 bits em `addr_bank[7:0][10:0]`, um endereço de 11 bits por banco.

Cada um dos 8 bancos pode ser acedido de forma independente, como será analisado numa tarefa seguinte. Assim, por exemplo, na situação mais extrema, seria possível efetuar duas leituras e uma escrita no mesmo ciclo, desde que os três acessos fossem a três bancos diferentes:

- Numa leitura não alinhada, os bancos *j* e *k* podem ser lidos no mesmo ciclo fornecendo os endereços de 11 bits nos sinais `addr_bank[j]` (que é obtido a partir do campo **Addr** de 11 bits do sinal `dccm_rd_addr_lo`) e `addr_bank[k]` (que é obtido a partir do campo **Addr** de 11 bits do sinal `dccm_rd_addr_hi`), e definindo os sinais de ativação correspondentes: `rden_bank[j] = 1` e `rden_bank[k] = 1`.
- Ao mesmo tempo, também é possível escrever no banco *i*, fornecendo o endereço de 11 bits no sinal `addr_bank[i]` (obtido a partir do campo **Addr** de 11 bits do sinal `dccm_wr_addr`), e definindo o sinal de ativação correspondente: `wren_bank[i] = 1`.

TAREFA: Usando as instruções fornecidas no Lab 1, implemente um novo sistema RVfpga que inclua uma ICCM de 64 KiB.

Lembre-se que a ICCM está desativada no nosso sistema por omissão. Assim, tal como explicado na Secção 2.A do documento SweRVref, para ativar a ICCM é necessário incluir a seguinte linha no ficheiro

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_define.s.vh`:

```
`define RV_ICCM_ENABLE 1
```

Além disso, os parâmetros fornecidos no sistema RVfpga padrão são para uma ICCM de 512 KiB. Assim, para implementar uma ICCM de 64 KiB, é necessário modificar as seguintes linhas do mesmo ficheiro (ficheiro *common_defines.vh*):

```
RV_ICCM_DATA_CELL ram_16384x39 → RV_ICCM_DATA_CELL ram_2048x39
RV_ICCM_BITS 19 → RV_ICCM_BITS 16
RV_ICCM_ROWS 16384 → RV_ICCM_ROWS 2048
RV_ICCM_INDEX_BITS 14 → RV_ICCM_INDEX_BITS 11
RV_ICCM_SIZE_512 → RV_ICCM_SIZE_64
RV_ICCM_SIZE 512 → RV_ICCM_SIZE 64
RV_ICCM_EADR 32'hee07ffff → RV_ICCM_EADR 32'hee00ffff
```

Como explicado na Secção 2.A do documento SweRVref, em vez de modificar manualmente o ficheiro *common_defines.vh*, também é possível modificar a configuração do processador SweRV EH1 utilizando o script *swerv.config*.

TAREFA: Desenhar uma figura semelhante à Figura 3 para a ICCM implementada na tarefa anterior.

B. Acesso à DCCM

À semelhança da I\$ que analisámos no Lab 19, a ICCM e a DCCM têm uma latência de acesso baixa - ou seja, que permite que os dados sejam lidos ou escritos num único ciclo (ver Figura 2). No entanto, ao contrário da I\$, a ICCM e a DCCM são controladas por software.

Nesta secção, ilustramos e descrevemos um acesso à DCCM. Utilizamos a arquitetura interna do DCCM apresentada na Figura 3 como referência e executamos um programa semelhante a um já utilizado no Lab 19. Este programa, apresentado na Figura 4 é fornecido na pasta *[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/*. Percorre um *array* de 250 elementos, lendo cada elemento (instrução *lw*, destacada a vermelho), adicionando-lhe um elemento e armazenando o elemento (instrução *sw*, destacada a vermelho) de volta no mesmo elemento do *array*. O ciclo contém 20 instruções *nop* para isolar as iterações umas das outras. O *array* é inicializado antes de ser acedido (o ciclo de inicialização não é mostrado na Figura 4 mas pode ver a inicialização do *array* no projeto PlatformIO).

```
// Acesso ao array
la t4, D
li t5, 50
li t0, 1000
la t6, D
add t6, t6, t0
li t5, 1

REPEAT_Access:
    lw t3, (t4)
    add t3, t3, t5
    sw t3, (t4)
    add t4, t4, 4
    INSERT_NOPS_10
    INSERT_NOPS_10
    bne t4, t6, REPEAT_Access # Repete o ciclo
```

Figura 4. Programa de exemplo

Abra o projeto no PlatformIO, compile-o e abra o ficheiro Disassembly (disponível em *[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis*). Repare que a instrução `lw` (0x000eae03) e a instrução `sw` (0x01cea023) são colocadas nos endereços 0x000001c0 e 0x000001c8, respetivamente.

0x000001c0:	000eae03	lw t3,0(t4)
...		
0x000001c8:	01cea023	sw t3,0(t4)

Figura 5 mostra a simulação de uma iteração aleatória do ciclo de Figura 4. A figura inclui alguns dos sinais apresentados na Figura 3 bem como alguns dos sinais do IP core do LSU que descrevemos no Lab 13.

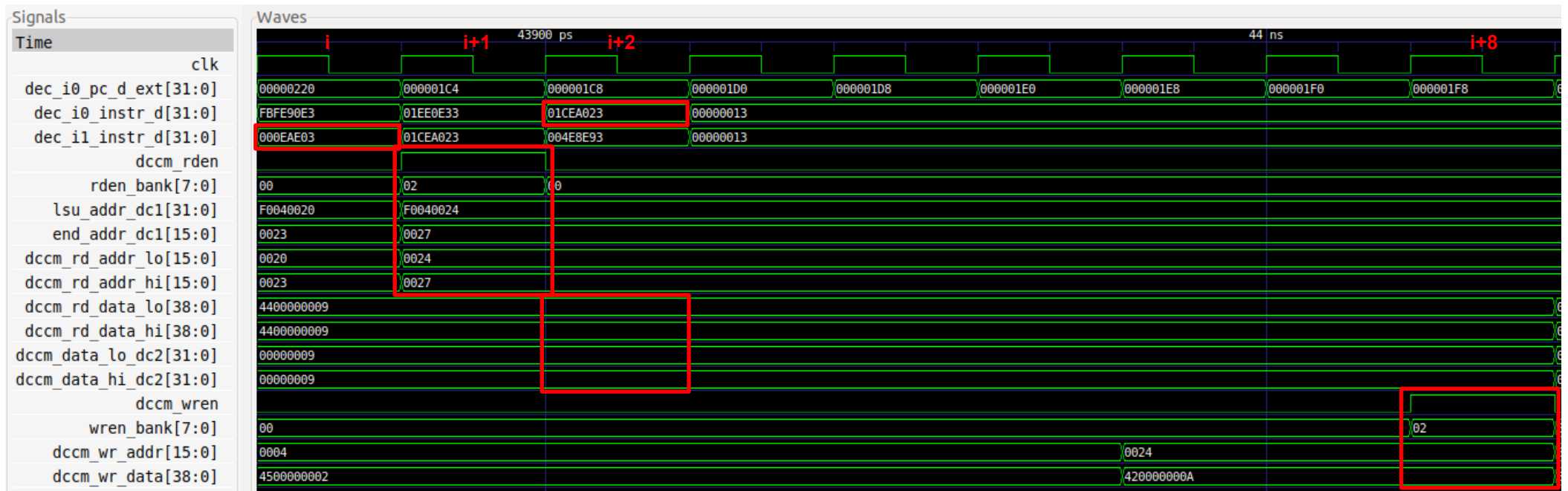



Figura 5. Simulação de uma iteração aleatória do programa de Figura 4

TAREFA: Replicar a simulação da Figura 5 no seu próprio computador. Para o fazer, siga os passos seguintes (descritos em pormenor na Secção 7 das GSG):

- Se necessário, gerar o executável da simulação (*Vrvfpgasim*).
- No PlatformIO, abra o projeto fornecido em: `[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM`.
- 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 *scriptLoadStore.tcl* (fornecido em `[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM`) para abrir os mesmos sinais que os mostrados na Figura 5. Para isso, no GTKWave, clique em *File* → *Read Tcl Script File* e selecione o ficheiro *scriptLoadStore.tcl*.
- Clicar várias vezes em *Zoom In* () e analisar a região que começa em 43900 ps.

As leituras e escritas de memória utilizando a DCCM ocorrem da seguinte forma:

- o **Ciclo i:** A instrução `lw` é decodificada na Via 1: `dec_il_instr_d = 0x000eae03`.
- o **Ciclo i+1:** O endereço é gerado nos andares DC1, como descrito no Lab 13 (ver Figura 6 desse laboratório), e fornecido à DCCM:
 - `lsu_addr_dc1[31:0] = 0xF0040024` → `dccm_rd_addr_lo[15:0] = 0x0024`
 - `end_addr_dc1[15:0] = 0x0027` → `dccm_rd_addr_hi[15:0] = 0x0027`

Como resultado da verificação do endereço, a leitura da DCCM é ativada: `dccm_rden = 1`. Este sinal é fornecido à DCCM e, juntamente com o campo *Bank* de 3 bits do endereço, determina o banco que deve ser lido. Neste caso, apenas o segundo banco do acesso precisa de ser lido, uma vez que o acesso está alinhado por palavras: `rden_bank = 0x02` (em binário 00000010).

- o **Ciclo i+2:** Os dados de leitura são obtidos da DCCM e fornecidos ao núcleo. Dado que se trata de um acesso alinhado, os dois sinais de leitura são iguais e apenas o `dccm_data_lo_dc2` é efetivamente utilizado pelo núcleo (mais uma vez, isto foi explicado no Lab 13):
 - `dccm_rd_data_lo = 0x4400000009` → `dccm_data_lo_dc2 = 0x00000009`
 - `dccm_rd_data_hi = 0x4400000009` → `dccm_data_hi_dc2 = 0x00000009`
- o **Ciclo i+8:** Depois de adicionar 1 (o imediato) ao valor lido (`0x00000009 + 1 = 0x0000000A`) e de atravessar o Store Buffer, como explicado no apêndice do Lab 13, os dados e o endereço são fornecidos à DCCM, e a escrita do banco correto é ativada utilizando os seguintes sinais:
 - `dccm_wren = 1`
 - `wren_bank = 0x02` (em binário 00000010; ou seja, o segundo banco)
 - `dccm_wr_addr = 0x0024`
 - `dccm_wr_data = 0x420000000A`

TAREFA: Explique como os sinais `rden_bank`, `wren_bank` e `addr_bank` são obtidos nas linhas 103, 104 e 105 do módulo `lsu_dccm_mem`.

TAREFA: Simular uma leitura não alinhada para a DCCM e analisar como é tratada no interior da DCCM. Pode usar o programa usado acima ([RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/) e simplesmente substituir a instrução load da seguinte forma:

`lw t3, (t4) → lw t3, 1(t4)`

TAREFA: Simular um conflito de banco DCCM modificando o programa da Figura 4 ([RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/).

1st modificação: Remover as 20 instruções `nop`, regenerar a simulação e analisar o `lw` e o `sw` numa iteração aleatória do ciclo.

2nd modification: Modificar o imediato da instrução `sw` para fazer com que o `lw` e o `sw` tenham acesso ao mesmo banco no mesmo ciclo:

`sw t3, (t4) → sw t3, 8(t4)`

3. BENCHMARKING

Para avaliar um processador, um programa (ou conjunto de programas) é executado e o desempenho do processador é medido. Comparamos os processadores executando os mesmos benchmarks (ou seja, conjuntos de programas) nesses processadores. Apresentamos dois benchmarks comuns: **CoreMark** e **Dhrystone**. Esses benchmarks estão na pasta [RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks. Descrevemos estes benchmarks, juntamente com o programa de **Processamento de Imagem** do Lab 5, a seguir.

A pasta [RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters contém um projeto PlatformIO do CoreMark destinado ao sistema RVfpga. Adaptámos o CoreMark ao sistema RVfpga utilizando as fontes fornecidas pela Chips Alliance em <https://github.com/chipsalliance/Cores-SweRV>. Para qualquer parâmetro de referência, usamos os contadores em hardware (HW Counters) para medir vários eventos do processador, como o número de instruções executadas e o número de ciclos do processador, conforme explicado no Lab 11. Além de modificar o benchmark para usar os contadores HW do RISC-V, adicionámos algum suporte para usar o DCCM/ICCM e para usar otimizações do compilador.

Na próxima secção, mostramos como executar o CoreMark na placa Nexys A7 em vários cenários.

A. Variante 1: Sem otimizações do compilador ou DCCM/ICCM

Primeiro, mostramos como executar o benchmark CoreMark nas condições do processador usadas nos laboratórios anteriores: modo de depuração e sem uso de DCCM/ICCM. Para isso, siga os próximos passos:

- Abra o projeto *CoreMark_HwCounters* no PlatformIO.

- Abrir o ficheiro `src/Test.c` (ver Figura 6), que inclui a função *principal* do nosso programa:
 - o A função *principal* começa por configurar os contadores HW para medir quatro eventos: número de ciclos, transações do barramento I (instruções) e transações do barramento D (instruções `ld/st`). Para este efeito, é utilizada a função `pspPerformanceCounterSet()`.
 - o Em seguida, configura as diferentes características do processador SweRV EH1, utilizando duas instruções de montagem (`li` e `csrrs`), tal como explicado na Secção 2.C do documento SweRVref. Neste caso, todas as características são deixadas nos seus valores por omissão.
 - o De seguida, o programa executa um ciclo que só é encerrado quando qualquer um dos interruptores da placa é invertido. O objetivo deste ciclo é permitir que o utilizador abra o monitor série (serial monitor) antes de o parâmetro de referência ser executado e apresentar os seus resultados.
 - o O programa invoca então a função `main_cmark()`, que implementa o próprio benchmark CoreMark, que é implementado no ficheiro `src/cmark.c`.
 - o Finalmente, imprime os quatro eventos utilizando a função `printfNexys()`.

```

25 int main(void)
26 {
27     /* Initialize Uart */
28     uartInit();
29
30     pspEnableAllPerformanceMonitor(1);
31
32     pspPerformanceCounterSet(D_PSP_COUNTER0, E_CYCLES_CLOCKS_ACTIVE);
33     pspPerformanceCounterSet(D_PSP_COUNTER1, E_INSTR_COMMITTED_ALL);
34     pspPerformanceCounterSet(D_PSP_COUNTER2, E_D_BUD_TRANSACTIONS_LD_ST);
35     pspPerformanceCounterSet(D_PSP_COUNTER3, E_I_BUS_TRANSACTIONS_INSTR);
36
37     /* Modify core features as desired */
38     __asm("li t2, 0x000");
39     __asm("csrrs t1, 0x7F9, t2");
40
41     /* Invert Switch to execute CoreMark*/
42     int switches_value, switches_init;
43     WRITE_GPIO(GPIO_INOUT, 0xFFFF);
44     switches_init = (READ_GPIO(GPIO_SWs) >> 16);
45     switches_value = switches_init;
46     while (switches_value==switches_init) {
47         switches_value = (READ_GPIO(GPIO_SWs) >> 16);
48         printfNexys("Invert any Switch to execute CoreMark");
49     }
50
51     main_cmark();
52
53     printfNexys("Cycles = %d", cyc_end-cyc_beg);
54     printfNexys("Instructions = %d", instr_end-instr_beg);
55     printfNexys("Data Bus Transactions = %d", LdSt_end-LdSt_beg);
56     printfNexys("Inst Bus Transactions = %d", Inst_end-Inst_beg);
57
58     while(1);
59 }

```

Figura 6. Ficheiro `src/Test.c` do projeto CoreMark no PlatformIO

- Analise brevemente as funções do benchmark CoreMark implementadas no ficheiro `src/cmark.c`. Note que os contadores HW são iniciados e parados dentro da função `main_cmark()` (linhas 1109-1112 e 1130-1133), e que o próprio benchmark é executado entretanto (linhas 1114-1128).

```

1104      /* perform actual benchmark */
1105      start_time();
1106
1107      __asm("__perf_start:");
1108
1109      cyc_beg = pspPerformanceCounterGet(D_PSP_COUNTER0);
1110      instr_beg = pspPerformanceCounterGet(D_PSP_COUNTER1);
1111      LdSt_beg = pspPerformanceCounterGet(D_PSP_COUNTER2);
1112      Inst_beg = pspPerformanceCounterGet(D_PSP_COUNTER3);
1113
1114      #if (MULTITHREAD>1)
1115          if (default_num_contexts>MULTITHREAD) {
1116              default_num_contexts=MULTITHREAD;
1117          }
1118          for (i=0 ; i<default_num_contexts; i++) {
1119              results[i].iterations=results[0].iterations;
1120              results[i].execs=results[0].execs;
1121              core_start_parallel(&results[i]);
1122          }
1123          for (i=0 ; i<default_num_contexts; i++) {
1124              core_stop_parallel(&results[i]);
1125          }
1126      #else
1127          iterate(&results[0]);
1128      #endif
1129
1130      cyc_end = pspPerformanceCounterGet(D_PSP_COUNTER0);
1131      instr_end = pspPerformanceCounterGet(D_PSP_COUNTER1);
1132      LdSt_end = pspPerformanceCounterGet(D_PSP_COUNTER2);
1133      Inst_end = pspPerformanceCounterGet(D_PSP_COUNTER3);
1134
1135      __asm("__perf_end:");
1136
1137      stop_time();
1138      total_time=get_time();

```

Figura 7. Ficheiro *src/cmark.c* do projeto CoreMark no PlatformIO

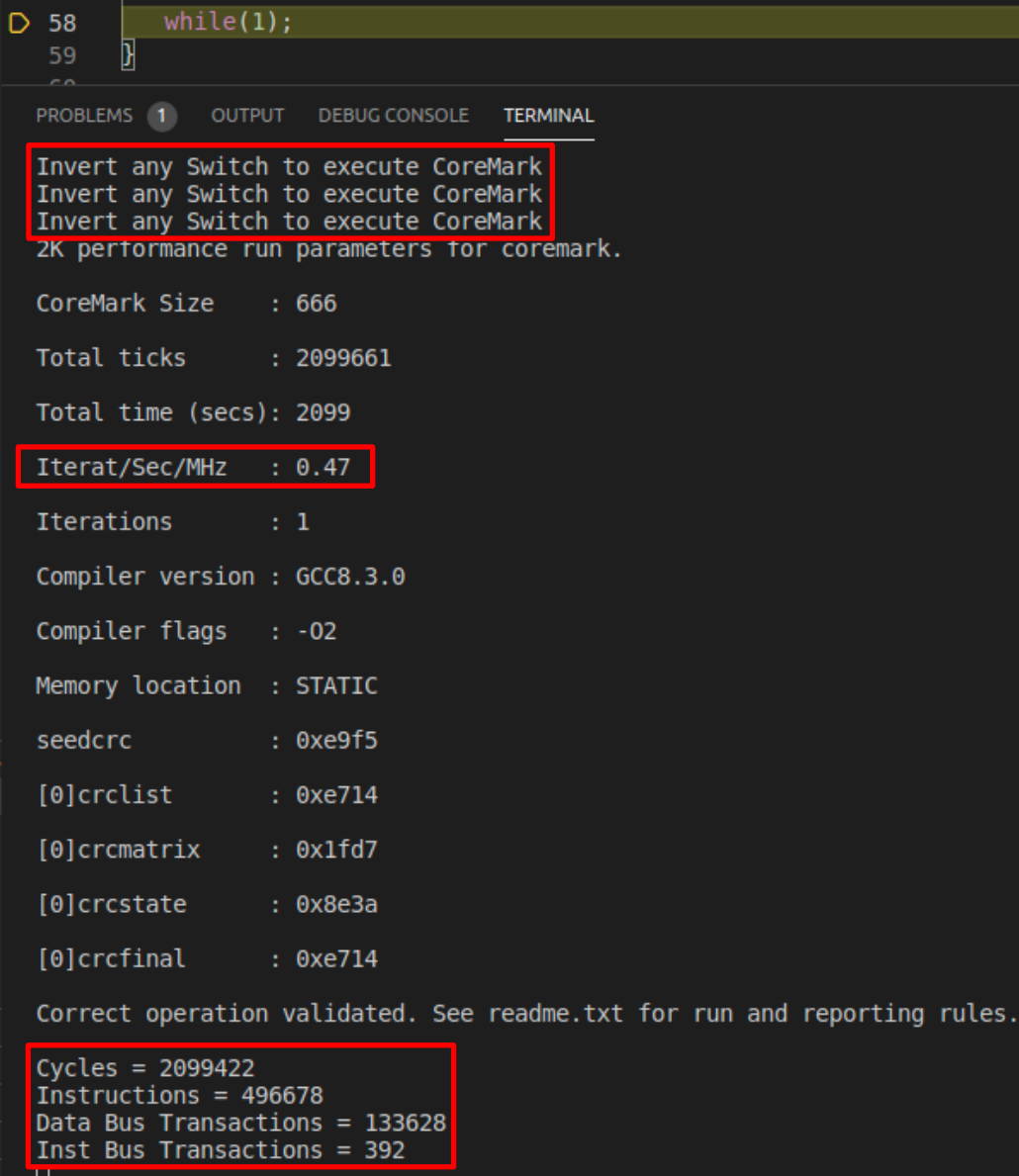
- Executar o programa na placa. Em seguida, abra o monitor série como explicado na Secção 6.F do GSG.

Depois de abrir o monitor série, verá primeiro uma mensagem repetida que lhe pede para inverter um interruptor na placa para executar o teste de referência CoreMark (ver a caixa vermelha superior na Figura 8). Depois de inverter um interruptor, o teste de referência é executado e apresenta os resultados, como mostra a Figura 8.

O CoreMark executa várias iterações de um ciclo (pode facilmente modificar o número de iterações através de um parâmetro chamado `ITERATIONS` e definido no ficheiro *src/cmark.c*). O número de iterações que completa por segundo é chamado de *pontuação do CoreMark* (CM). O número de iterações por MHz é *CM/MHz*. O parâmetro de comparação fornece o CM/MHz - também chamado *Iterat/Sec/MHz* (iteraões/segundo/MHz) - que é 0,47. Também pode ver os valores fornecidos pelos contadores em hardware, que foram utilizados para calcular o CM/MHz.

A execução demorou ~2 milhões de ciclos e foram processadas cerca de meio milhão de instruções, resultando num IPC (instruções por ciclo) $\approx 0,25$; especificamente, $\frac{1}{2}$ milhão de instruções / 2 milhões de ciclos $\approx 0,25$. Este desempenho é realmente mau:

recorde-se que o IPC ideal no processador SweRV EH1 é 2 porque é superescalar de duas vias. No entanto, o desempenho é fraco devido ao grande número de leituras/escritas de dados e à lentidão da memória externa DDR. O número de transações de dados através do barramento é de cerca de 133.000. O número de transações de instruções através do barramento é de apenas 392, porque a maioria dos acessos a instruções ocorre na I\$. Recorde-se que o sistema RVfpga não tem uma D\$ (cache de dados).



```

58 while(1);
59
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
2K performance run parameters for coremark.

CoreMark Size      : 666
Total ticks        : 2099661
Total time (secs): 2099
Iterat/Sec/MHz     : 0.47
Iterations         : 1
Compiler version   : GCC8.3.0
Compiler flags     : -O2
Memory location    : STATIC
seedcrc            : 0xe9f5
[0]crclist         : 0xe714
[0]crcmatrix       : 0x1fd7
[0]crcstate        : 0x8e3a
[0]crcfinal        : 0xe714

Correct operation validated. See readme.txt for run and reporting rules.

Cycles = 2099422
Instructions = 496678
Data Bus Transactions = 133628
Inst Bus Transactions = 392
  
```

Figura 8. Resultados da execução do benchmark CoreMark

B. Variante 2: Utilização da DCCM

Agora habilitamos a DCCM no sistema RVfpga para que a maioria dos acessos a dados use a DCCM (em vez da memória DDR externa). Como veremos, essa alteração aumenta o desempenho, tal como esperado. Siga as próximas etapas para executar o CoreMark numa versão do sistema RVfpga que usa a DCCM:

- O *linker script* padrão que usamos até agora na maioria dos laboratórios está disponível em `.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/link.lds`. No entanto, para utilizar a DCCM para armazenar alguns dados do programa, utilizamos um *linker script* específico que é fornecido como parte do projeto PlatformIO que está a utilizar e que está disponível em: `[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM.ld`. Abra este ficheiro e inspecione-o. Figura 9 mostra algumas partes deste ficheiro, que descrevemos brevemente.

A captura de ecrã no topo da Figura 9 define uma secção de memória para a DCCM (denominada `dccm`), que corresponde ao espaço de endereço definido na Figura 2(b) para esta memória: `dccm (wxa!ri) : ORIGEM = 0xf0040000, COMPRIMENTO = 64K`

As restantes capturas de ecrã mapeiam várias secções de código para a memória DCCM: `.rodata`, `.data`, `.sdata`, `.bss` e `.stack`.

```

26 MEMORY
27 {
28     ram (wxa!ri) : ORIGIN = 0x00000000, LENGTH = 64M
29     ram2 (wxa!ri) : ORIGIN = 0x04000000, LENGTH = 64M
30     dccm (wxa!ri) : ORIGIN = 0xf0040000, LENGTH = 64K
31     ovl          : ORIGIN = 0xE0000000, LENGTH = 8k
32 }

```

```

72 .rodata :
73 {
74     *(.rodata)
75     *(.rodata .rodata.*)
76     *(.gnu.linkonce.r.*)
77     KEEP(*(COMRV_RODATA_SEC))
78     . = ALIGN(4);
79 } > dccm : dccm_load

```

```

104 .data :
105 {
106     *(.data .data.*)
107     *(.gnu.linkonce.d.*)
108     . += 10; /* fix for linker false error message */
109     . = ALIGN(8);
110 } > dccm : dccm_load

```

```

122 .sdata :
123 {
124     . = ALIGN(8);
125     __global_pointer$ = . + 0x800;
126     *(.sdata .sdata.*)
127     *(.gnu.linkonce.s.*)
128     . = ALIGN(8);
129     *(.srodata .srodata.*)
130     . = ALIGN(8);
131 } > dccm : dccm_load

```

```

142 .bss :
143 {
144     *(.sbss .sbss.* .gnu.linkonce.sb.*)
145     *(.scommon)
146     *(.bss)
147     . = ALIGN(8);
148 } >dccm : dccm load

```

```

152 .stack :
153 {
154     _heap_end = .;
155     . = . + __stack_size;
156     sp = .;
157 } > dccm : dccm load

```

Figura 9. Ficheiro *ld/link_DCCM.ld* do projeto CoreMark no PlatformIO

- Abra o ficheiro *platformio.ini* e descomente a linha 18 (ver Figura 10) para que o programa use o *linker script* da Figura 9 em vez do *linker script* padrão. Por esta via, como explicado acima, a maioria dos dados será acedida na memória rápida DCCM em vez da memória lenta DDR.



```

11 [env:swervolf_nexys]
12 platform = chipsalliance
13 board = swervolf_nexys
14 framework = wd-riscv-sdk
15
16
17 # DCCM/ICCM link scripts
18 #board_build.ldscript = ld/link_DCCM.ld
19 #board_build.ldscript = ld/link_DCCM-ICCM.ld

```

```

11 [env:swervolf_nexys]
12 platform = chipsalliance
13 board = swervolf_nexys
14 framework = wd-riscv-sdk
15
16
17 # DCCM/ICCM link scripts
18 board_build.ldscript = ld/link_DCCM.ld
19 #board_build.ldscript = ld/link_DCCM-ICCM.ld

```

Figura 10. Ficheiro *platformio.ini* do projeto CoreMark no PlatformIO

- Executar o programa na placa e abrir o monitor de série. Em seguida, inverte um interruptor na placa. Obterá os resultados mostrados na Figura 11.

Neste caso, o CM/MHz (ou seja, o valor de Iterat/Sec/MHz) é 1,88. O número de ciclos diminuiu para cerca de meio milhão de ciclos. Tal como na versão anterior do processador, são processadas cerca de meio milhão de instruções; obtemos assim um IPC de 1. Ao mapear secções do programa para o DCCM, o desempenho aumentou por um fator de quatro.

Finalmente, o número de transações de dados através do barramento é agora 0, dado que os dados são armazenados na DCCM.

```

58 while(1);
59
60
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL

Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
2K performance run parameters for coremark.

CoreMark Size      : 666
Total ticks         : 530028
Total time (secs): 530
Iterat/Sec/MHz      : 1.88
Iterations          : 1
Compiler version    : GCC8.3.0
Compiler flags      : -O2
Memory location     : STATIC
seedcrc             : 0xe9f5
[0]crclist          : 0xe714
[0]crcmatrix        : 0x1fd7
[0]crcstate         : 0x8e3a
[0]crcfinal         : 0xe714

Correct operation validated. See readme.txt for run and reporting rules.

Cycles = 529897
Instructions = 496678
Data Bus Transactions = 0
Inst Bus Transactions = 392

```

Figura 11. Resultados da execução do benchmark CoreMark

TAREFA: No ficheiro *platformio.ini* (ver Figura 10), comente a linha 18 e descomente a linha 19 para que o programa use o *linker script* fornecido em: *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM-ICCM.ld*. Analise esse novo *linker script*, que usa a DCCM para armazenar a maioria dos dados e a ICCM para armazenar as instruções. Execute o benchmark CoreMark e compare os resultados com os obtidos nesta secção. Neste caso, como o nosso sistema RVfpga padrão não inclui uma ICCM, use o *bitstream* criado na primeira tarefa deste laboratório ou o *bitstream* que fornecemos em: *[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Bitstream/rvfpganexys_DCCM-ICCM.bit*.

C. Variante: Utilizar a DCCM e as optimizações do compilador

Agora adicionamos outra via para melhorar o desempenho: as otimizações do compilador. Tal como na secção anterior, utilizamos a DCCM para armazenar a maioria das secções de dados da aplicação - mas agora também ativamos as otimizações do compilador. Até este ponto, executámos programas em modo de depuração sem otimizações do compilador. Para ativar as otimizações do compilador, siga as próximas etapas:

- Use o *linker script*
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM.ld novamente. Para isso, abra o ficheiro *platformio.ini* e descomente a linha 18 (ver Figura 10) e comente a linha 19.
- **Utilizando um procedimento diferente do utilizado anteriormente**, execute o programa na placa: Carregue o *bitstream* habitual, mas depois utilize a opção "Upload and Monitor" disponível nas Project Tasks do PlatformIO (ver Figura 12).

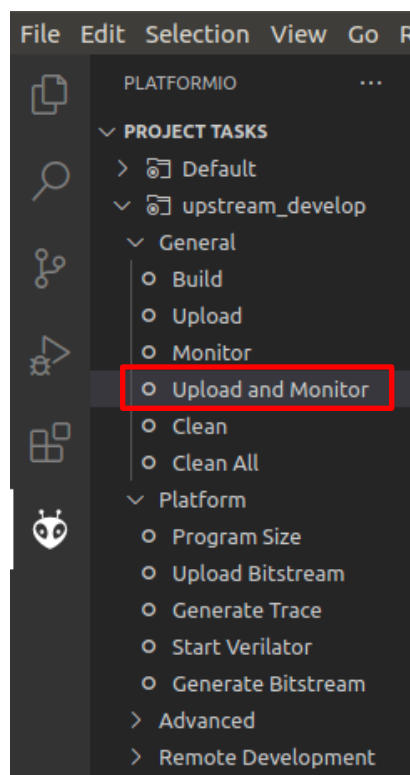


Figura 12. Carregar e monitorizar

Esta opção irá compilar o programa, executá-lo na placa e abrir o monitor série. Esta opção compila usando as opções de otimização determinados pela opção *build_flags* no *platformio.ini*, neste caso -O2 (veja Figura 13).

```
25 build_unflags = -Wa,-march=rv32imac -march=rv32imac -Os
26 build_flags = -Wa,-march=rv32ima -march=rv32ima -O2
27 extra_scripts = extra_script.py
```

Figura 13. Ficheiro *platformio.ini*, opção *build_flags*

Quando o programa começar a ser executado, como habitualmente, inverte um interruptor na placa. Obterá os resultados mostrados na Figura 14.

O CM/MHz (Iterat/Sec/MHz) é agora de 3,47. O número de ciclos diminuiu para cerca de 288.000, e o número de instruções é agora de cerca de 309.000. Apesar de o IPC ≈ 1 , o desempenho (CM/MHz e, por conseguinte, o tempo de execução) é agora muito melhor do que no cenário analisado na secção B, uma vez que tanto o número de ciclos como o de instruções diminuíram significativamente. Esta melhoria deve-se à ativação das otimizações do compilador. O número de transações do barramento de dados continua a ser 0, dado que os dados são armazenados na DCCM.

```
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
Invert any Switch to execute CoreMark
2K performance run parameters for coremark.

CoreMark Size      : 666
Total ticks        : 288490
Total time (secs): 288
Iterat/Sec/MHz     : 3.47
Iterations         : 1
Compiler version   : GCC8.3.0
Compiler flags     : -O2
Memory location    : STATIC
seedcrc           : 0xe9f5
[0]crclist         : 0xe714
[0]crcmatrix       : 0x1fd7
[0]crcstate        : 0x8e3a
[0]crcfinal        : 0xe714

Correct operation validated. See readme.txt for run and reporting rules.

Cycles = 288337
Instructions = 309637
Data Bus Transactions = 0
Inst Bus Transactions = 504
```

Figura 14. Resultados de execução do CoreMark quando se utilizam otimizações do compilador

TAREFA: Modificar a otimização de compilação para -O3 e explicar os resultados.

4. EXERCÍCIOS

- 1) Faça a mesma análise que foi feita para o CoreMark, mas desta vez usando o benchmark Dhrystone. Um projeto PlatformIO que contém o benchmark Dhrystone está em:
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Dhrystone_HwCounters. Como

requerido por todos os benchmarks, este benchmark Dhrystone foi adaptado para o sistema específico, neste caso o sistema RVfpga, usando os ficheiros fonte fornecidos em <https://github.com/chipsalliance/Cores-SweRV>. O ficheiro *Test.c* é semelhante ao do CoreMark (Figura 6) mas invoca a função `main_dhry()`, que inclui o próprio benchmark Dhrystone.

- 2) Faça a mesma análise que foi feita para o CoreMark, mas desta vez para a aplicação ImageProcessing. Um projeto PlatformIO que contém a aplicação ImageProcessing está em:
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/ImageProcessing_HwCounters. Estas são as aplicações que usámos no Lab 5 para transformar uma imagem RGB em escala de cinzentos. O ficheiro *Test.c* é semelhante ao do CoreMark (Figura 6) mas invoca a função `ImageTransformation()`, que inclui o benchmark Image Transformation que analisámos no Lab 5. A DCCM do sistema RVfpga predefinida não é suficientemente grande para armazenar a imagem, pelo que se deve usar o sistema RVfpga (bitstream) que tem uma DCCM de 128 KiB, que se encontra em:
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Bitstreams/rvfpganexys_DCCM-128.bit.
- 3) Ativar/desativar várias funcionalidades do núcleo, conforme descrito na Secção 2.C deste laboratório. Compare os resultados de desempenho - ou seja, os valores dos contadores HW ao executar os programas nesses núcleos modificados. Execute os três programas (CoreMark, Dhrystone e ImageProcessing) nesses sistemas RVfpga modificados na placa Nexys A7. As variações incluem:
 - Usando diferentes configurações e implementações do preditor de saltos (salto não tomado, Gshare, e o preditor bimodal implementado no Exercício 1 de Lab 16).
 - Ativar/desativar a função de *dual-issue*.
 - Utilizando várias configurações de I\$/DCCM/ICCM (como diferentes tamanhos ou diferentes Políticas de substituição de I\$).