



IMAGINATION大學計劃

RVfpga實驗20

ICCM、DCCM和基準測試

1. 簡介

在本實驗中，我們將分析SweRV EH1處理器中提供的暫存記憶體（ICCM和DCCM），然後會提供幾個基準測試範例和練習，以展示實驗11至20中的一些概念。

回想一下RVfpga入門指南的圖25（為方便起見，我們已將該圖複製為下面的圖1），RVfpga系統包含兩個暫存記憶體：一個用於儲存資料，稱為資料緊密耦合記憶體（DCCM）；另一個用於儲存指令，稱為指令緊密耦合記憶體（ICCM）。

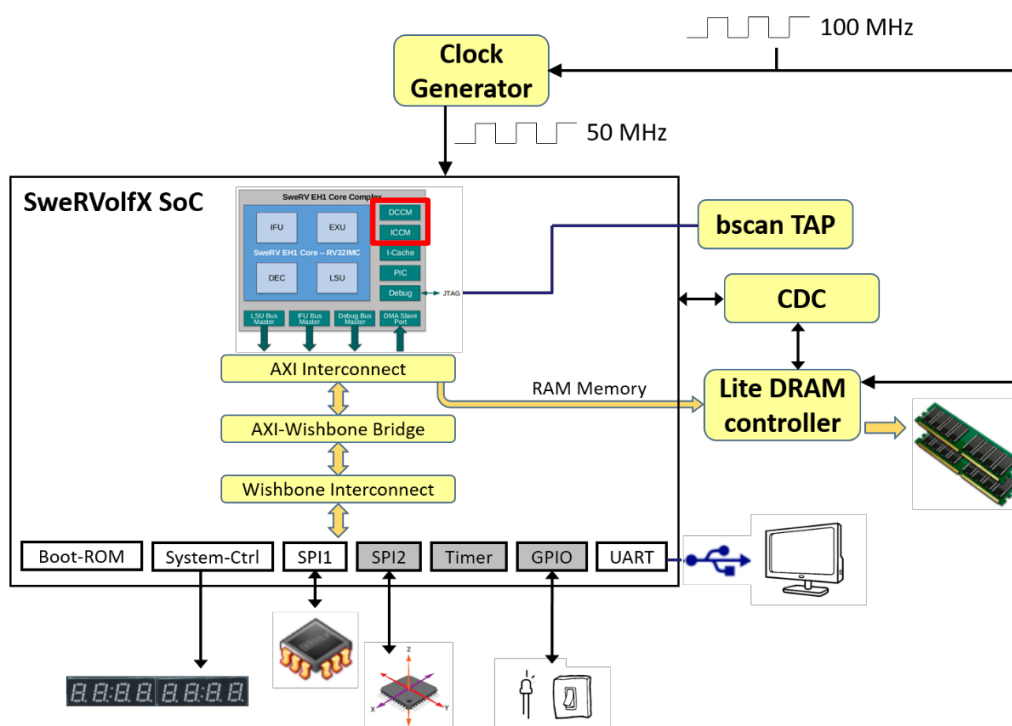
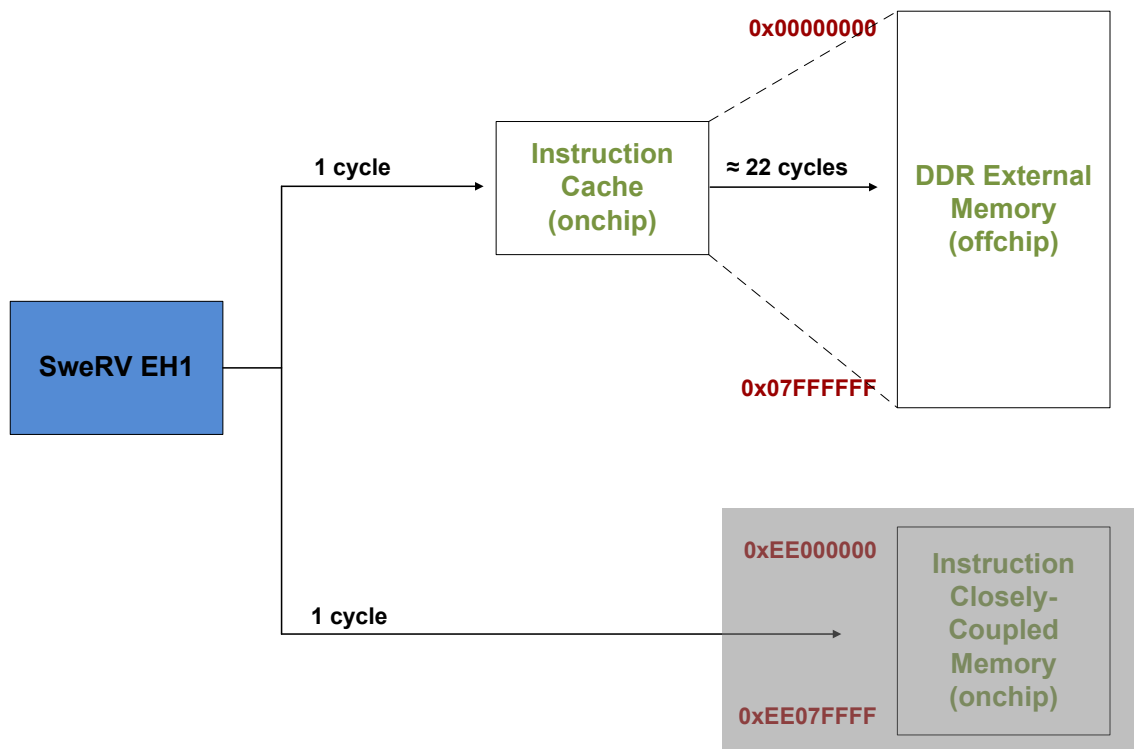


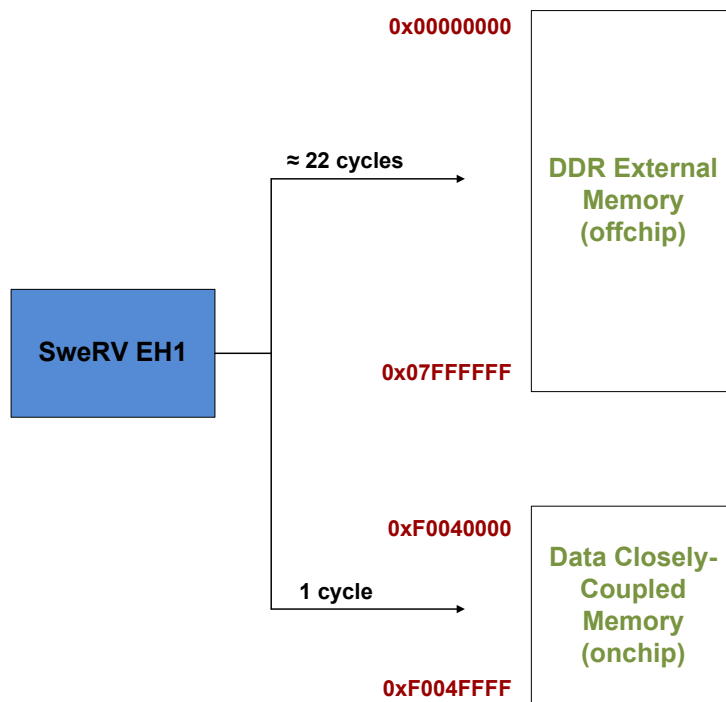
圖1. RVfpgaNexys系統

附註：開始本實驗前，建議您先閱讀Preeti Ranjan Panda、Nikil D. Dutt和Alexandru Nicolau的論文「晶片上與晶片外記憶體：基於嵌入式處理器的系統中的資料分區問題」（ACM Trans. Design Autom. Electr. Syst. 5(3): 682-704 (2000)）的第1部分和第3部分。（文件連結：<https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.2430&rep=rep1&type=pdf>）。該論文詳細介紹了暫存記憶體在嵌入式處理器中的應用。

入門指南的第4.B部分說明了RVfpga系統的記憶體對映。下圖為相關說明的補充，說明了RVfpga系統中提供的指令記憶體（圖2a）和資料記憶體（圖2b）所佔用的位址空間。



(a) 指令記憶體的位址空間，由指令快取（I\$）和DDR外部記憶體組成。預設系統中會停用ICCM。



(b) 資料記憶體的位址空間，由DCCM和DDR外部記憶體組成。

圖2. 指令記憶體和資料記憶體的RVfpga系統位址空間

在本實驗室中，我們將重點介紹資料/指令緊密耦合記憶體組態和操作（分別為第2.A部分和第2.B部分），然後提供幾個基準測試範例和練習（第3節），其中會用到專用於描繪特定情景的範例程式以及實際應用程式。

2. 資料/指令緊密耦合記憶體（DCCM/ICCM）

在本部分中，我們將分析RVfpga系統中提供的資料緊密耦合記憶體（DCCM）和指令緊密耦合記憶體（ICCM）。我們首先介紹如何配置這兩個結構（第3.A部分），然後說明如何執行對DCCM的存取（第3.B部分）。

A. RVfpga系統中的DCCM和ICCM組態

RVfpga系統的DCCM和ICCM可使用檔案

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh`
中定義的一系列參數進行多種配置。對於這兩個結構，預設RVfpga系統使用以下參數：

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

```

但是，類似於IS\$中的情況，部分上述參數將在檔案

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/global.h中被改寫：

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);

```

請注意，如圖2所示，我們的基線系統中啟用了DCCM（RV_DCCM_ENABLE = 1），但停用了ICCM（未定義RV_ICCM_ENABLE），因此先前實驗中使用的SoC中不包括ICCM。

表1摘要了RVfpga系統中的ICCM和DCCM組態。

表1. DCCM和ICCM組態

特性	值
DCCM	
啟用位元	1
位址空間	0xF0040000 – 0xF004FFFF
大小	64 KiB
儲存區數量	8
儲存區大小	2048x39位元（有7位元為同位檢查位元）
ICCM	
啟用位元	0

圖3所示為RVfpga的DCCM組態區塊圖。載入儲存單元（lsu）向DCCM提供輸入訊號（lsu_addr_dc1、end_addr_dc1、stbuf_addr_any、stbuf_ecc_any和stbuf_data_any）/接收來自DCCM的輸出訊號（dccm_data_lo_dc2和dccm_data_hi_dc2），如實驗13所述（參見實驗13中的圖6和圖13）。

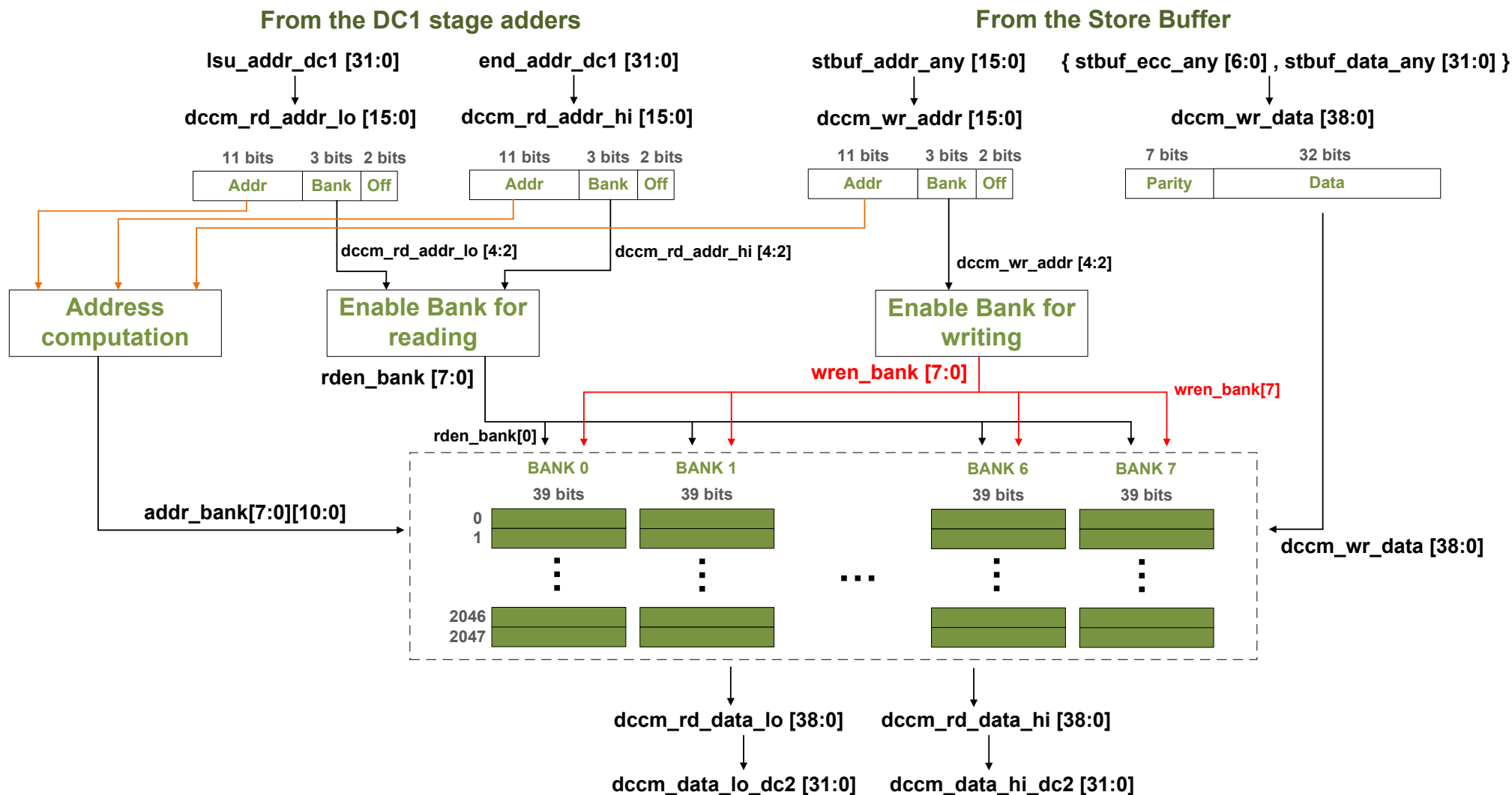


圖3. DCCM內部設計。

RVfpga系統的DCCM在檔案

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/lsu/lsu_dccm_mem.sv`所包含的模組`lsu_dccm_mem`中實作。如圖3所示，DCCM分為8個儲存區。系統提供兩個讀支援取位址來未對齊存取：`dccm_rd_addr_lo[15:0] = lsu_addr_dc1[15:0]`和`dccm_rd_addr_hi[15:0] = end_addr_dc1[15:0]`。這些位址在邏輯上分為3個欄位：

- **Bank**：選擇的儲存區。
- **Addr**：在儲存區內讀取的32位元字的位址。
- **Off**：在32位元字內讀取的位元組。
- 請注意，每個32位元字會新增7個同位檢查位元。

如實驗13所述以及圖3所示，儲存緩衝區在訊號`dccm_wr_addr[15:0]`中提供一個寫入位址（有關儲存緩衝區操作的更多說明，請參見實驗13中的附錄）。寫入位址的劃分方式與讀取位址相同（參見上文）。基於這些位址的3位元**Bank**欄位（加上圖中未指定的、將在下文的任務中分析的其他訊號），在`rden_bank[7:0]`和`wren_bank[7:0]`中將分別取得8個讀/寫啟用位元。每一位元用於確定是否必須啟用或停用相應儲存區的讀寫。

基於這些位址的11位元**Addr**欄位（加上圖中未指定的、將在下文的任務中分析的其他訊號），在`addr_bank[7:0][10:0]`中將取得8個11位元位址，每個儲存區對應一個11位元位址。

8個儲存區中的每一個都可以獨立存取，下文的任務中將進行具體分析。因此，在最極端的情況下，可以在同一週期內執行兩次讀取存取和一次寫入存取，前提是三次存取必須針對三個不同的儲存區：

- 在未對齊讀存取中，透過在訊號`addr_bank[j]`（從訊號`dccm_rd_addr_lo`的11位元**Addr**欄位取得）和`addr_bank[k]`（從訊號`dccm_rd_addr_hi`的11位元**Addr**欄位取得）中提供11位元位址並將相應的啟用訊號置1（`rden_bank[j] = rden_bank[k] = 1`），可以在同一週期內讀取儲存區`j`和`k`。
- 同時，透過在訊號`addr_bank[i]`（從訊號`dccm_wr_addr`的**Addr**欄位取得）中提供11位元位址並將相應的啟用訊號置1（`wren_bank[i] = 1`），還可以對儲存區`i`進行寫入存取。

任務：使用實驗1中提供的指令，實作一個包含64 KiB ICCM的新RVfpga系統。

請注意，預設系統中會停用ICCM。因此，如SweRVref文件的第2.A部分所述，要啟用ICCM，必須在檔案

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh`中包含以下行：

```
`define RV_ICCM_ENABLE 1
```

此外，預設RVfpga系統中提供的參數適用於512 KiB ICCM。因此，要實作64 KiB ICCM，必須修改上述檔案（檔案`common_defines.vh`）的以下行：

```
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
```

如SweRVref文件的第2.A部分所述，除手動修改檔案`common_defines.vh`外，還可以使用`swerv.config`指令碼修改SweRV EH1處理器的組態。

任務：為上一任務中實作的ICCM繪製一張與圖3類似的圖。

B. 存取DCCM

與我們在實驗19中分析的I\$類似，ICCM和DCCM具有較低的存取延遲，因此支援在單個週期內讀取或寫入資料（參見圖2）。但是，與I\$的情況相反，ICCM和DCCM受軟體控制。

在本部分中，我們將說明並描述針對DCCM的存取。我們使用圖3中所示的DCCM內部設計作為參考，並執行與實驗19中所使用的程式類似的程式。該程式位於資料夾 `[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/` 中，具體內容如圖4所示。該程式會遍歷包含250個元素的陣列，讀取每個元素（使用`lw`指令，以紅色強調顯示），將元素加1並儲存回同一陣列元素（使用`sw`指令，以紅色強調顯示）。該迴圈包含20條`nop`指令，以將迭代互相分隔開。在存取陣列之前會先對其進行初始化（圖4中未顯示初始化迴圈，但可在PlatformIO專案中查看陣列的初始化過程）。

```
// Access 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    # Repeat the loop
```

圖4. 範例程式

在PlatformIO中開啟、編譯專案，然後開啟反組譯檔案（位於 `[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis`）。請注意，`lw`指令（`0x000eae03`）和`sw`指令（`0x01cea023`）分別位於位址`0x000001c0`和`0x000001c8`處。

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

圖5所示為圖4中迴圈的任意一次迭代的模擬結果。圖中包括圖3所示的部分訊號以及我們在實驗13中描述的一些LSU核心訊號。

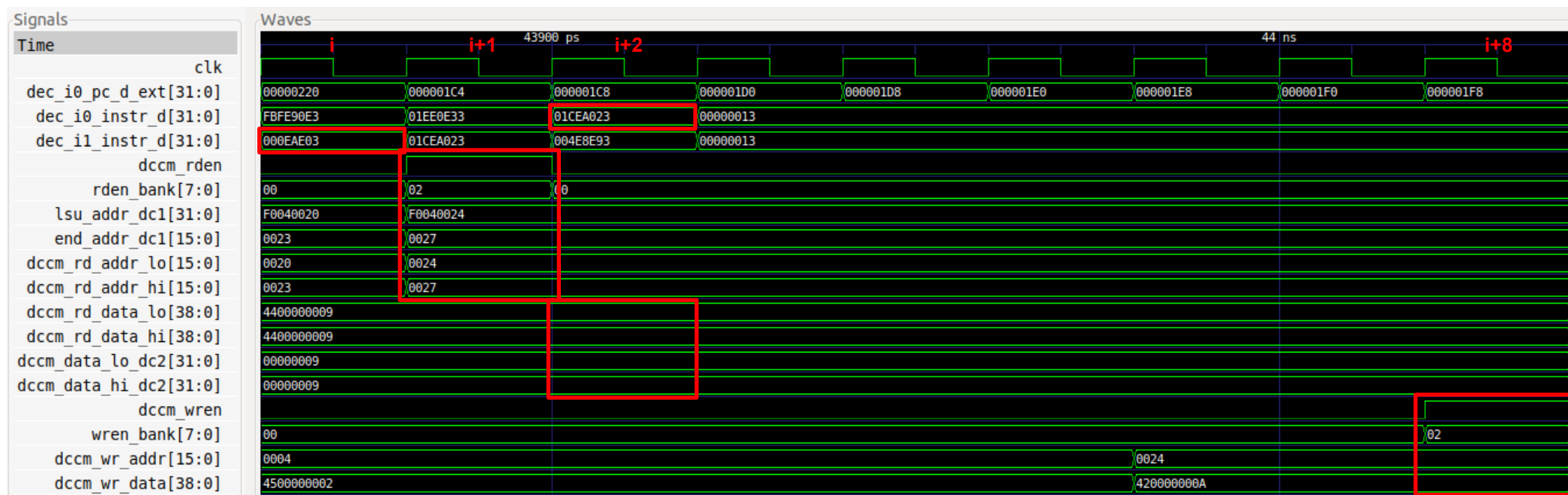


圖5. 圖4中程式任意一次迭代的模擬結果

任務：在自己的電腦上重複圖5中的模擬過程。為此，請按照以下步驟操作（在GSG的第7部分中詳述）：

- 必要時產生模擬二進位檔案（*Vrvfpgasim*）。
- 在PlatformIO中，開啟在以下位置提供的專案：*[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM*。
- 在檔案*platformio.ini*中建立到RVfpga模擬二進位檔案（*Vrvfpgasim*）的正確路徑。
- 使用Verilator產生模擬軌跡（產生軌跡）。
- 在GTKWave上開啟軌跡。
- 使用檔案*scriptLoadStore.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM*中提供）開啟與圖5所示訊號相同的訊號。為此，在GTKWave上，按一下「File → Read Tcl Script File」（檔案 → 讀取Tcl指令碼檔案）並選擇*scriptLoadStore.tcl*檔案。
- 按幾次「Zoom In」（放大）（），然後分析自43900 ps起的區域。

使用DCCM讀寫記憶體的过程如下：

- **週期i：**在通路1中對lw指令進行解碼：*dec_i1_instr_d = 0x000eae03*。
- **週期i+1：**在DC1階段產生位址，如實驗13所述（參見該實驗中的圖6），然後將位址提供給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*
- 完成位址檢查後，將啟用DCCM的讀取操作：*dccm_rden = 1*。該訊號將與位址的3位元Bank欄位（用於確定必須讀取的儲存區）一起提供給DCCM。在這種情況下，只需讀取第二個儲存區，因為存取為字對齊存取：*rden_bank = 0x02 (in binary 00000010)*。
- **週期i+2：**從DCCM取得讀取資料，並將其提供給核心。由於該存取為對齊存取，兩個讀取訊號相等，核心只能有效地使用*dccm_data_lo_dc2*（實驗13中同樣提供了具體的說明）：
 - *dccm_rd_data_lo = 0x4400000009* → *dccm_data_lo_dc2 = 0x00000009*
 - *dccm_rd_data_hi = 0x4400000009* → *dccm_data_hi_dc2 = 0x00000009*
- **週期i+8：**如實驗13的附錄所述，將讀取值加1（立即數）（*0x00000009 + 1 = 0x0000000A*）並遍歷儲存緩衝區後，系統會將資料和位址提供給DCCM，並使用以下訊號為相應的緩衝區啟用寫入存取：
 - *dccm_wren = 1*
 - *wren_bank = 0x02*（二進位值為00000010；即第二個儲存區）
 - *dccm_wr_addr = 0x0024*
 - *dccm_wr_data = 0x420000000A*

任務：解釋如何在模組*lsu_dccm_mem*的第103、104和105行中取得訊號*rden_bank*、*wren_bank*和*addr_bank*。

任務：模擬DCCM的未對齊讀取存取，分析DCCM內部的處理方式。仍可使用上述程式（`[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/`），只需將載入指令進行如下替換：

`lw t3, (t4) → lw t3, 1(t4)`

任務：透過修改圖4中的程式（`[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/`），模擬DCCM儲存區的衝突。

第1項修改：刪除20條nop指令，重新產生模擬，並隨機選取迴圈的某次迭代以分析lw和sw。

第2項修改：修改sw指令的立即數，使lw和sw嘗試在同一週期內存取同一儲存區：

`sw t3, (t4) → sw t3, 8(t4)`

3. 基準測試

如需對處理器進行基準測試，應執行程式（或程式集）並測定處理器效能。透過在多個處理器上執行相同的基準（即程式集），可對這些處理器進行比較。我們引入了兩種常用的基準：

CoreMark和**Dhrystone**。這些基準位於資料夾

`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks`中。我們接下來詳細介紹這些基準以及實驗5中的影像處理程式。

資料夾`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters`中提供了專用於RVfpga系統的CoreMark基準PlatformIO專案。我們使用Chips Alliance提供的原始程式碼（<https://github.com/chipsalliance/Cores-SweRV>）對CoreMark進行了修改，使其能夠適用於RVfpga系統。在任何基準測試中，我們均使用硬體計數器（HW計數器）來測量各種處理器事件，如執行的指令數和處理器週期數（參見實驗11）。除了修改基準以便使用RISC-V硬體計數器外，我們還新增了一些對使用DCCM/ICCM和編譯器最佳化的支援。

在下一部分，我們將顯示如何在各種情景下使用Nexys A7開發板執行CoreMark。

A. 情景1：無編譯器最佳化和DCCM/ICCM

首先，我們將顯示如何在先前實驗所使用的處理器條件下執行CoreMark基準測試，即採用除錯模式，不使用DCCM/ICCM。為此，請按以下步驟操作：

- 在PlatformIO中開啟**CoreMark_HwCounters**專案。
- 開啟檔案**src/Test.c**（參見圖6），其中包含程式的**main**函數：
 - o **main**函數首先配置用於測量以下四個事件的硬體計數器：週期數、指令匯流排交易（指令）和資料匯流排交易（ld/st指令）。為此，需使用函數 `pspPerformanceCounterSet()`。

- 然後使用兩條組合語言指令（li和csrrs）配置SweRV EH1處理器的不同功能，如SweRVref文件的第2.C部分所述。在這種情況下，所有功能均使用預設值。
- 隨後程式將執行一個迴圈，只有開發板上的任何開關狀態發生反轉時，才會退出迴圈。該迴圈的作用是允許使用者在執行基準測試並輸出結果之前開啟序列監視器。
- 然後，程式將叫用函數main_cmark()，該函數會在檔案src/cmark.c中自行實作CoreMark基準測試。
- 最後，程式使用函數printfNexys()輸出四個事件。

```

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 }

```

圖6. CoreMark PlatformIO專案中的src/Test.c檔案

- 簡要分析在檔案src/cmark.c中實作的CoreMark基準測試所包含的函數。請注意，HW計數器在main_cmark()函數中啟動和停止（第1109-1112行和第1130-1133行），基準測試本身則在上述程式碼行（第1114-1128行）之間執行。

```

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();

```

圖7. CoreMark PlatformIO專案中的src/cmark.c檔案

- 在開發板上執行程式。然後根據GSG第6.F部分的說明開啟序列監視器。

開啟序列監視器後，首先會看到一條重複出現的訊息，要求您反轉開發板中的開關以執行CoreMark基準測試（參見圖8上方的紅色方塊）。開關反轉後，將執行基準測試並輸出結果，如圖8所示。

CoreMark會執行迴圈的多次迭代（可透過檔案src/cmark.c中定義的參數ITERATIONS輕鬆修改迭代次數）。每秒鐘完成的迭代次數稱為**CoreMark score (CM)**。每MHz的迭代次數為**CM/MHz**。基準測試會提供CM/MHz，該參數也稱為Iterat/Sec/MHz（迭代數/秒/兆赫），在本例中的值為0.47。也可以查看硬體計數器提供的值，並使用該值計算CM/MHz。

執行用時約200萬個週期，處理了約50萬條指令，相應的IPC（每個週期執行的指令數）≈ 0.25（具體計算過程為：50萬條指令/200萬個週期 ≈ 0.25）。這樣的效能遠遠無法滿足要求：回想一下，由於SweRV EH1處理器為雙路超標量處理器，其理想的IPC值應為2。由於處理器要進行大量的資料讀/寫存取，並且DDR外部記憶體的速度較慢，因此處理器的效能明顯下降。透過匯流排處理的資料交易數約為133000。透過匯流排處理的指令交易數

僅為392，因為大多數指令存取會在I\$中發生命中。請記住，RVfpga系統中不存在D\$（資料快取）。

```

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
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

```

圖8. CoreMark基準測試的執行結果

B. 情景2：使用DCCM

我們接下來在RVfpga系統中啟用DCCM，以便使用DCCM（而不是外部DDR記憶體）完成大部分資料存取。從下文的結果可以看出，這一更改能夠實現預期的效能提升。請按照以下步驟在使用DCCM的RVfpga系統版本上執行CoreMark：

- 到目前為止，我們在大多數實驗中均使用預設連結器指令碼（路徑為`.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/link.lids`）。但是，在本例中，為了利用DCCM來儲存程式的某些資料，我們需要使用隨PlatformIO專案提供的一個特定的連結器指令碼，其路徑為：

[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM.ld。開啟該檔案並對其進行檢查。圖9所示為該檔案的一部分，我們將進行簡要說明。

圖9中最上方的螢幕擷取畫面定義了DCCM的記憶體部分（稱為dccm），對應於圖2（b）中為該記憶體定義的位址空間：dccm (wxa!ri) : ORIGIN = 0xf0040000, LENGTH = 64K

其餘螢幕擷取畫面將幾個程式碼部分對映到DCCM記憶體：
.rodata、.data、.sdata、.bss和.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

```

圖9. CoreMark PlatformIO專案中的ld/link_DCCM.ld檔案

- 開啟檔案*platformio.ini*並取消註解第18行（參見圖10），以便程式使用圖9中的連結器指令碼替代預設連結器指令碼。如上文所述，透過這種方式，可在高速DCCM（而非速度較慢的DDR記憶體）中存取大部分資料。



```

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

```

圖10. CoreMark PlatformIO專案中的platformio.ini檔案

- 在開發板上執行程式，並開啟序列監視器。然後，反轉開發板上的某個開關。得到的結果如圖11所示。

在本例中，CM/MHz（即Iterat/Sec/MHz）的值為1.88。週期數減少為50萬左右。與上一版本一樣，處理器會處理大約50萬條指令；因此可得出IPC為1。透過將程式的各個部分對映到DCCM，處理器的效能提高了四倍。

最後，由於資料儲存在DCCM中，透過匯流排處理的資料交易數變為0。

```

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

```

圖11. CoreMark基準測試的執行結果

任務：在檔案*platformio.ini*（參見圖10）中，註解掉第18行並取消註解第19行，以便程式使用以下路徑中的連結器指令碼：

[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM-ICCM.ld。分析新的連結器指令碼，該指令碼使用DCCM儲存大部分資料，使用ICCM儲存指令。執行CoreMark基準測試，將結果與本部分提供的結果進行比較。在本例中，由於我們的預設RVfpga系統不包括ICCM，請使用您在本實驗的第一個任務中建立的位元流或以下路徑中的位元流：

[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Bitstreams/rvfpganexys_DCCM-ICCM.bit。

C. 情景3：使用DCCM和編譯器最佳化

我們接下來介紹另一種提高效率的方法：編譯器最佳化。與上一部分相同，我們使用DCCM儲存應用程式的大部分資料，但我們另外啟用了編譯器最佳化。在此之前，我們均在除錯模式下執行程式，未進行編譯器最佳化。要啟用編譯器最佳化，請按以下步驟操作：

- 再次使用
`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM.ld`路徑下的連結器指令碼。為此，應開啟檔案`platformio.ini`，取消註解第18行（參見圖10）並註解掉第19行。
- **使用與先前不同的步驟**在開發板上執行程式：上傳常用的位元流，但改為使用PlatformIO的「Project Tasks」（專案任務）中提供的「Upload and Monitor」（上傳並監視）選項（參見圖12）。

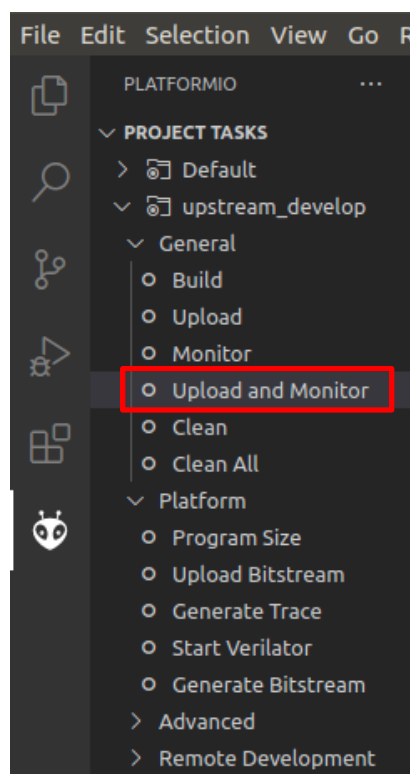


圖12. 上傳並監視

該選項將編譯程式、在開發板上執行程式並開啟序列監視器。該選項使用`platformio.ini`中的`build_flags`選項所確定的最佳化旗標進行編譯，在本例中為`-O2`（參見圖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
```

圖13. 檔案`platformio.ini`中的`build_flags`選項

等待程式開始執行後，像往常一樣，反轉開發板上的開關。得到的結果如圖14所示。

此時，CM/MHz（Iterat/Sec/MHz）的值為3.47。週期數和指令數分別減少為288000和309000左右。雖然IPC仍然約等於1，但此時的效能（CM/MHz，以及對應的執行時間）相比B部分中分析的場景大幅提升，因為週期數和指令數都顯著減少。效能得到改善的原因是啟用了編譯器最佳化。由於資料儲存在DCCM中，資料匯流排交易數仍為0。

```
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
```

圖14. 使用編譯器最佳化時的CoreMark執行結果

任務：將編譯器最佳化方式改為-O3，執行程式並解釋結果。

4. 練習

- 1) 使用Dhrystone基準替代CoreMark基準，執行相同的分析。可存取以下路徑取得包含Dhrystone基準的PlatformIO專案：
`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Dhrystone_HwCounters`。根據所有基準的要求，我們已使用<https://github.com/chipsalliance/Cores-SweRV>中提供的原始程式碼對該Dhrystone基準進行了修改，使其能夠適用於特定系統（在本例中為

RVfpga系統)。檔案**Test.c**與CoreMark中的檔案(圖6)類似,但會叫用函數**main_dhry()**,該函數包含Dhrystone基準本身。

- 2) 使用影像處理應用程式替代CoreMark,執行相同的分析。可存取以下路徑取得包含影像處理應用程式的PlatformIO專案:
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/ImageProcessing_HwCounters。我們在實驗5中曾使用這些應用程式將RGB影像轉換為灰階影像。檔案**Test.c**與CoreMark中的檔案(圖6)類似,但會叫用函數**ImageTransformation()**,該函數包含我們在實驗5中分析的影像轉換基準。預設RVfpga系統的DCCM沒有足夠的空間來儲存影像,因此需使用具有128 KiB DCCM的RVfpga系統(位元串流),該檔案路徑為:
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Bitstreams/rvfganexys_DCCM-128.bit。
- 3) 根據本實驗第2.C部分所述,啟用/停用各種核心功能。比較相應的效能結果(即在這些修改後的核心上執行程式時HW計數器的值)。在Nexys A7開發板上,使用這些修改後的RVfpga系統執行全部三個程式(CoreMark、Dhrystone和影像處理)。可能的變化包括:
 - 使用不同的分支預測器組態和實作方案(如始終不採取分支、使用Gshare分支預測器或使用實驗16的練習1中實作的雙模分支預測器)。
 - 啟用/停用雙指令功能。
 - 使用各種不同的IS/DCCM/ICCM組態(例如不同的大小或不同的IS替換策略)。