



Imagination大学计划

RVfpga实验20

ICCM、DCCM和基准测试

1. 简介

在本实验中，我们将分析SweRV EH1处理器中提供的便笺式存储器（ICCM和DCCM），然后会提供几个基准测试示例和练习，以演示实验11至20中的一些概念。

回想一下RVfpga入门指南的图25（为方便起见，我们已将该图复制为下面的图1），RVfpga系统包含两个便笺式存储器：一个用于存储数据，称为数据紧密耦合存储器（DCCM）；另一个用于存储指令，称为指令紧密耦合存储器（ICCM）。

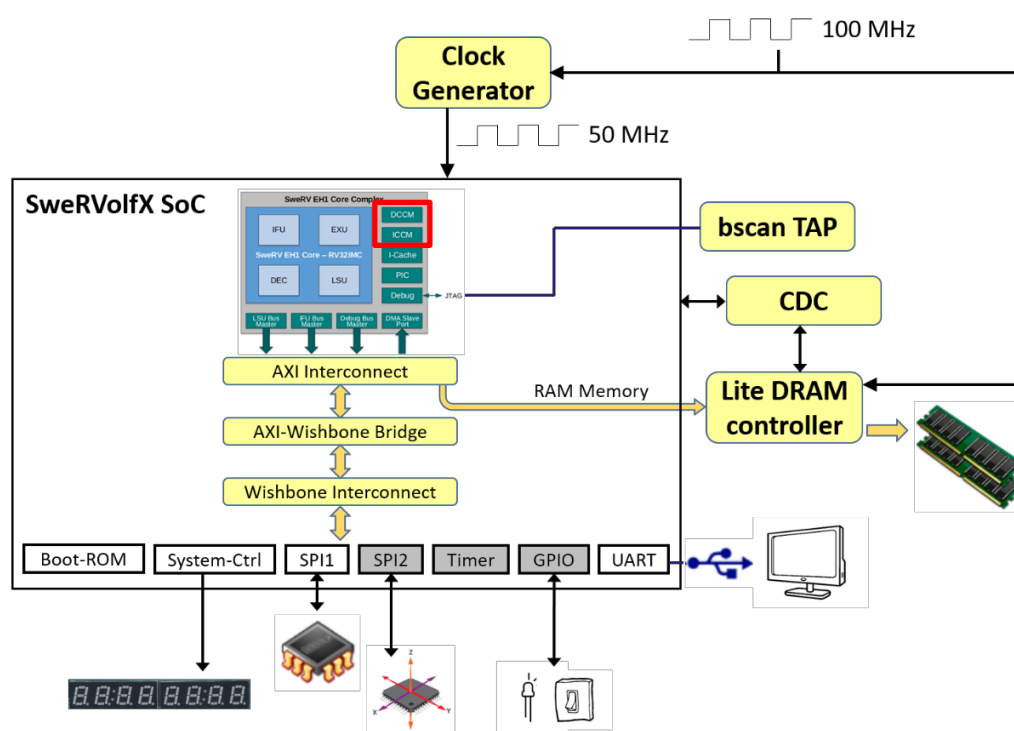
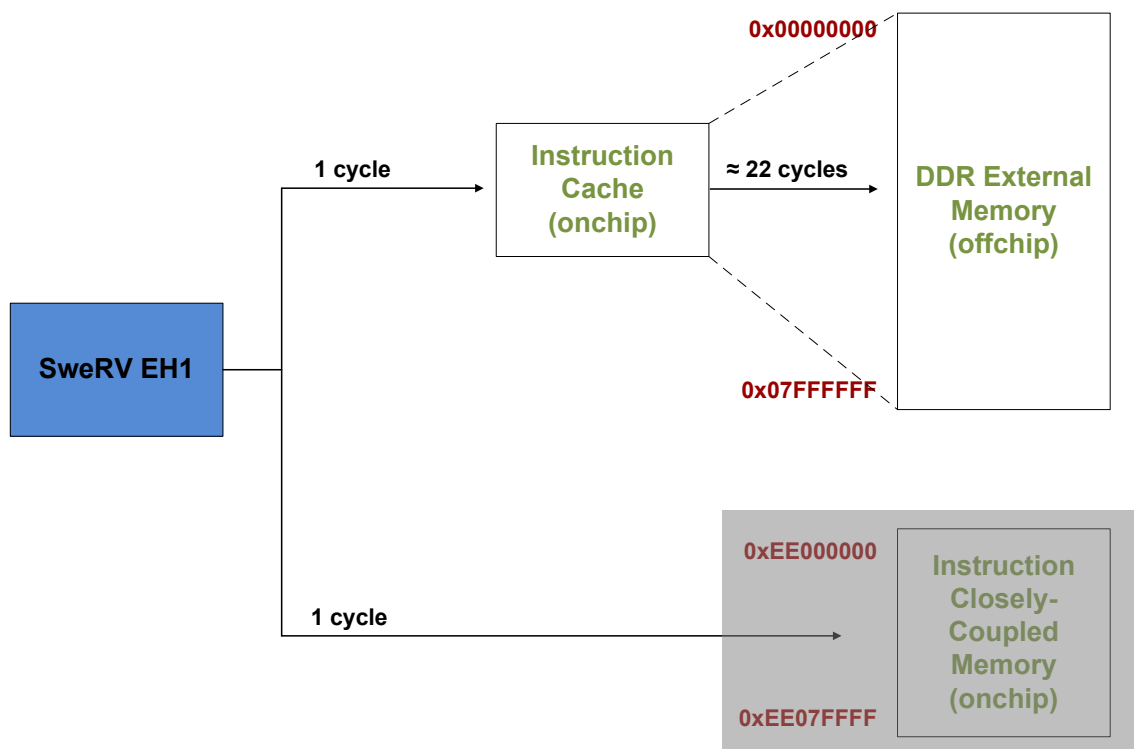


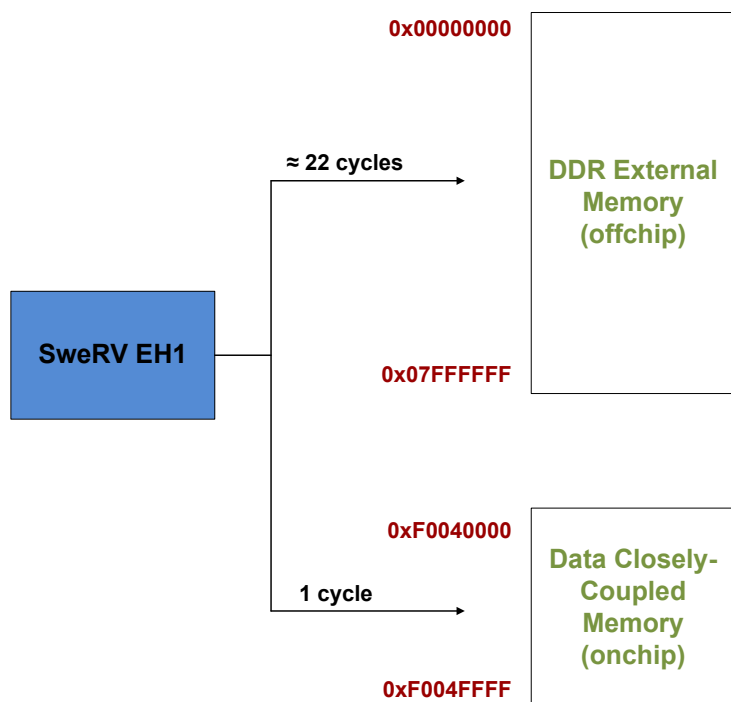
图1. RVfpqaNexys系统

注：开始本实验前，建议您先阅读Preeti Ranjan Panda、Nikil D. Dutt和Alexandru Nicolau的论文“片上与片外存储器：基于嵌入式处理器的系统中的数据分区问题”（ACM Trans.Design Autom.Electr.Syst.5(3): 682-704 (2000)）的第1部分和第3部分。（文档链接：<https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.472.2430&rep=rep1&type=pdf> 该论文详细介绍了便笺式存储器在嵌入式处理器中的应用。）

入门指南的第4.B部分说明了RVfpga系统的存储器映射。下图为相关说明的补充，展示了RVfpga系统中提供的指令存储器（图2a）和数据存储器（图2b）所占用的地址空间。



(a) 指令存储器的地址空间，由指令缓存（I\$）和DDR外部存储器组成。默认系统中会禁止ICCM。



(b) 数据存储器的地址空间，由DCCM和DDR外部存储器组成。

图2. 指令存储器和数据存储器的RVfpga系统地址空间

在本实验室中，我们将重点介绍数据/指令紧密耦合存储器的配置和操作（分别为第2.A部分和第2.B部分），然后提供几个基准测试示例和练习（第3节），其中会用到专用于展示特定情景的示例程序以及实际应用程序。

2. 数据/指令紧密耦合存储器（DCCM/ICCM）

在本部分中，我们将分析RVfpga系统中提供的数据紧密耦合存储器（DCCM）和指令紧密耦合存储器（ICCM）。我们首先介绍如何配置这两个结构（第3.A部分），然后说明如何执行对DCCM的访问（第3.B部分）。

A. RVfpga系统中的DCCM和ICCM配置

RVfpga系统的DCCM和ICCM可使用文件

`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/common_defines.vh`中定义的一系列参数进行多种配置。对于这两个结构，默认RVfpga系统使用以下参数：

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'h0
`define RV_ICCM_OFFSET 10'h000000
`define RV_ICCM_SADR 32'h000000
`define RV_ICCM_EADR 32'h007ffff

```

但是，类似于IS\$中的情况，部分上述参数将在文件
[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/global.h中被改写：

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);

```

请注意，如图2所示，我们的基线系统中使能了DCCM（RV_DCCM_ENABLE = 1），但禁止了ICCM（未定义RV_ICCM_ENABLE），因此先前实验中使用的SoC中不包括ICCM。

表1总结了RVfpga系统中的ICCM和DCCM配置。

表1. DCCM和ICCM配置

特性	值
DCCM	
使能位	1
地址空间	0xF0040000 – 0xF004FFFF
大小	64 KiB
存储区数量	8
存储区大小	2048x39位（有7位为奇偶校验位）
ICCM	
使能位	0

图3所示为RVfpga的DCCM配置框图。装载存储单元（lsu）向DCCM提供输入信号（lsu_addr_dc1、end_addr_dc1、stbuf_addr_any、stbuf_ecc_any和stbuf_data_any）/接收来自DDCM的输出信号（dccm_data_lo_dc2和dccm_data_hi_dc2），如实验13所述（参见实验13中的图6和图13）。

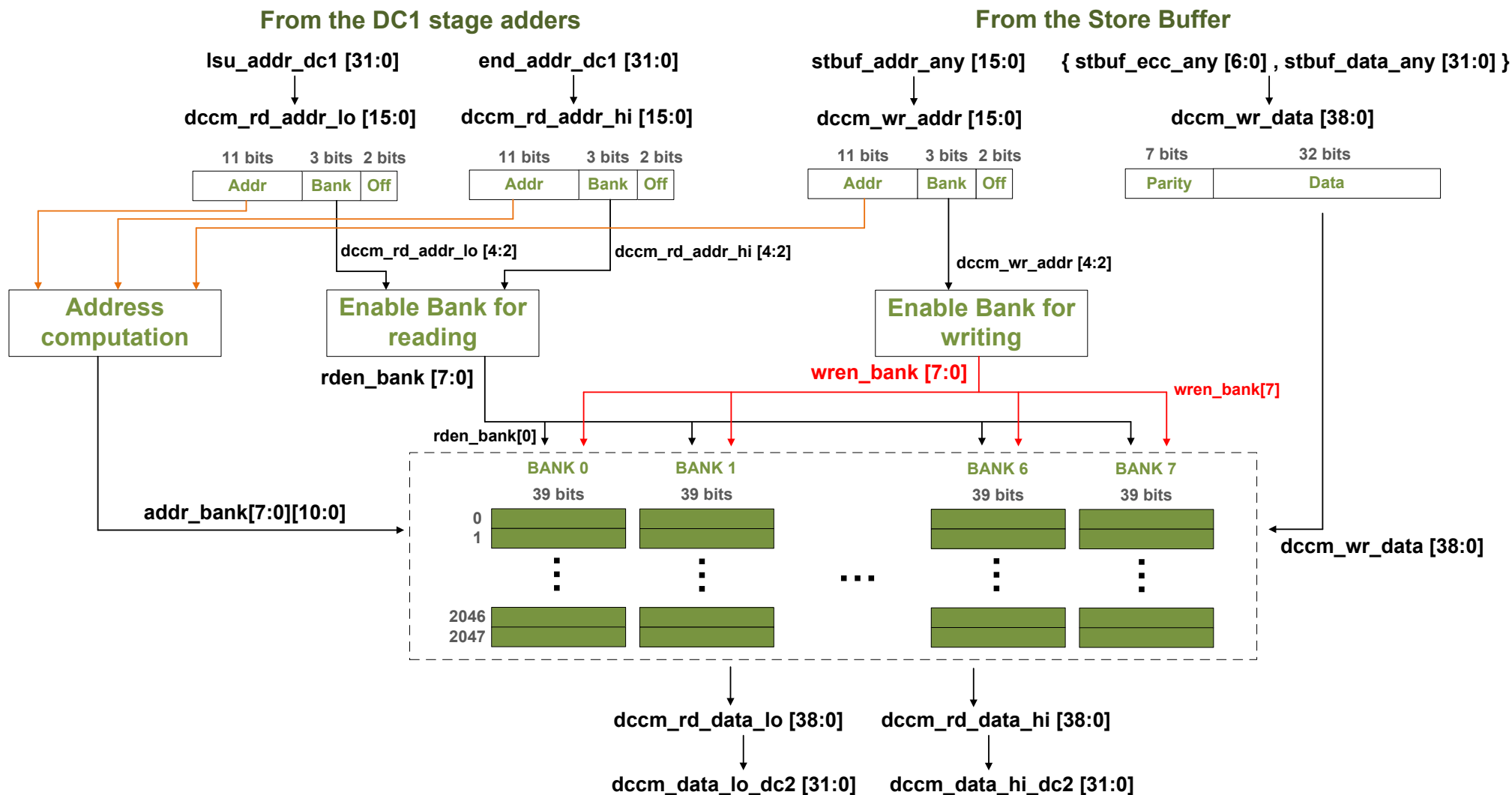


图3. DCCM内部设计。

RVfpga系统的DCCM在文件

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/lsu/lsu_dccm_mem.sv所包含的模块lsu_dccm_mem中实现。如图3所示，DCCM分为8个存储区。系统提供两个读地址来支持未对齐访问： $\text{dccm_rd_addr_lo}[15:0] = \text{lsu_addr_dc1}[15:0]$ and $\text{dccm_rd_addr_hi}[15:0] = \text{end_addr_dc1}[15:0]$ 。这些地址在逻辑上分为3个字段：

- **Bank**: 选择的存储区。
- **Addr**: 在存储区内读取的32位字的地址。
- **Off**: 在32位字内读取的字节。
- 请注意，每个32位字会添加7个奇偶校验位。

如实验13所述以及图3所示，存储缓冲区在信号 $\text{dccm_wr_addr}[15:0]$ 中提供一个写地址（有关存储缓冲区操作的更多说明，请参见实验13中的附录）。写地址的划分方式与读地址相同（参见上文）。基于这些地址的3位Bank字段（加上图中未指定的、将在下文的任务中分析的其他信号），在 $\text{rden_bank}[7:0]$ 和 $\text{wren_bank}[7:0]$ 中将分别获得8个读/写使能位。每一位用于确定是否必须允许或禁止相应存储区的读写。

基于这些地址的11位Addr字段（加上图中未指定的、将在下文的任务中分析的其他信号），在 $\text{addr_bank}[7:0][10:0]$ 中将获得8个11位地址，每个存储区对应一个11位地址。

8个存储区中的每一个都可以独立访问，下文的任务中将进行具体分析。因此，在最极端的情况下，可以在同一周期内执行两次读访问和一次写访问，前提是三次访问必须针对三个不同的存储区：

- 在未对齐读访问中，通过在信号 $\text{addr_bank}[j]$ （从信号 dccm_rd_addr_lo 的11位Addr字段获取）和 $\text{addr_bank}[k]$ （从信号 dccm_rd_addr_hi 的11位Addr字段获取）中提供11位地址并将相应的使能信号置1（ $\text{rden_bank}[j] = \text{rden_bank}[k] = 1$ ），可以在同一周期内读取存储区j和k。
- 同时，通过在信号 $\text{addr_bank}[i]$ （从信号 dccm_wr_addr 的Addr字段获取）中提供11位地址并将相应的使能信号置1（ $\text{wren_bank}[i] = 1$ ），还可以对存储区i进行写访问。

任务：使用实验1中提供的指令，实现一个包含64 KiB ICCM的新RVfpga系统。

请注意，默认系统中会禁止ICCM。因此，如SweRVref文档的第2.A部分所述，要使能ICCM，必须在文件

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

```
`define RV_ICCM_ENABLE 1
```

此外，默认RVfpga系统中提供的参数适用于512 KiB ICCM。因此，要实现64 KiB ICCM，必须修改上述文件（文件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文档的第2.A部分所述，除手动修改文件`common_defines.vh`外，还可以使用`swerv.config`脚本修改SweRV EH1处理器的配置。

任务：为上一任务中实现的ICCM绘制一张与图3类似的图。

B. 访问DCCM

与我们在实验19中分析的I\$类似，ICCM和DCCM具有较低的访问延迟，因此支持在单个周期内读取或写入数据（参见图2）。但是，与I\$的情况相反，ICCM和DCCM受软件控制。

在本部分中，我们将演示并描述针对DCCM的访问。我们使用图3中所示的DCCM内部设计作为参考，并执行与实验19中所使用的程序类似的程序。该程序位于文件夹 `[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/` 中，具体内容如图4所示。该程序会遍历包含250个元素的数组，读取每个元素（使用`lw`指令，以红色突出显示），将元素加1并存储回同一数组元素（使用`sw`指令，以红色突出显示）。该循环包含20条`nop`指令，以将迭代互相分隔开。在访问数组之前会先对其进行初始化（图4中未显示初始化循环，但可在PlatformIO项目中查看数组的初始化过程）。

```
// 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
```

图4. 示例程序

在PlatformIO中打开、编译项目，然后打开反汇编文件（位于 `[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis`）。请注意，`lw`指令（`0x000eae03`）和`sw`指令（`0x01cea023`）分别位于地址`0x000001c0`和`0x000001c8`处。

```

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

```

图5所示为图4中循环的任意一次迭代的仿真结果。图中包括图3所示的部分信号以及我们在实验13中描述的一些LSU内核信号。

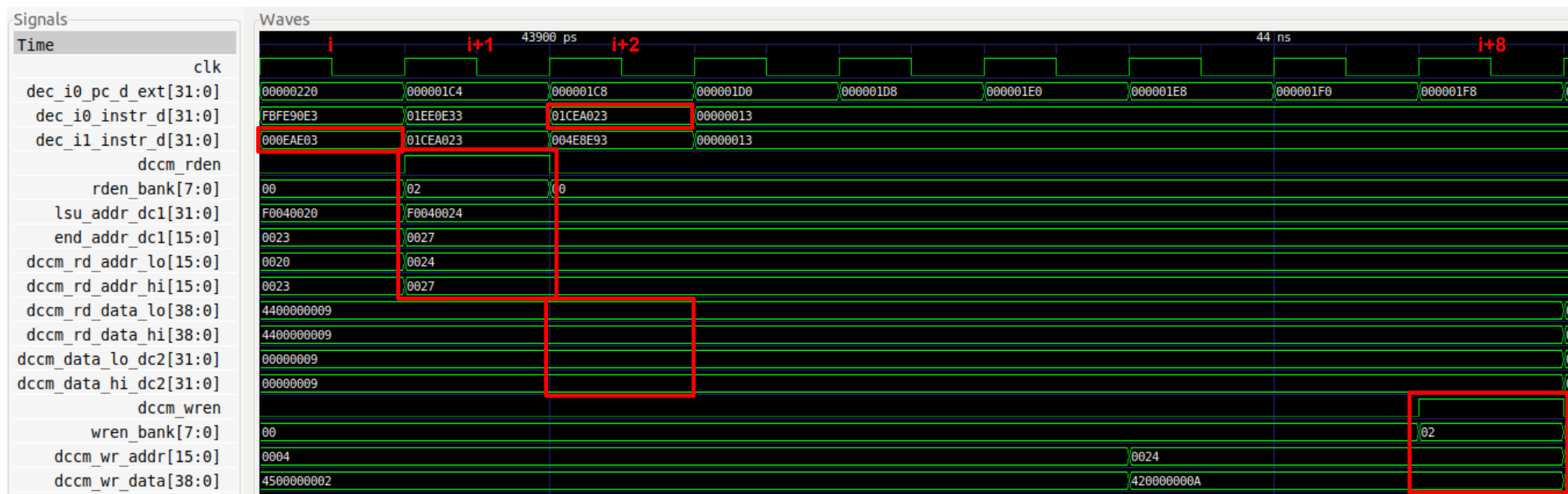



图5. 图4中程序任意一次迭代的仿真结果

任务： 在自己的计算机上重复图5中的仿真过程。为此，请按照以下步骤操作（在GSG的第7部分中详述）：

- 必要时生成仿真二进制文件（*Vrvfpgasim*）。
- 在PlatformIO中，打开在以下位置提供的项目：*[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM*。
- 在文件*platformio.ini*中建立到RVfpga仿真二进制文件（*Vrvfpgasim*）的正确路径。
- 使用Verilator生成仿真轨迹（生成轨迹）。
- 在GTKWave上打开轨迹。
- 使用文件*scriptLoadStore.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM*中提供）打开与图5所示信号相同的信号。为此，在GTKWave上，单击“**File → Read Tcl Script File**”（文件 → 读取Tcl脚本文件）并选择*scriptLoadStore.tcl*文件。
- 单击几次“**Zoom In**”（放大）（），然后分析自43900 ps起的区域。

使用DCCM读写存储器的过程如下：

- **周期i：** 在通路1中对lw指令进行译码：*dec_i1_instr_d = 0x000eae03*。
- **周期i+1：** 在DC1阶段生成地址，如实验13所述（参见该实验中的图6），然后将地址提供给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*

完成地址检查后，将使能DCCM的读操作：*dccm_rden = 1*。该信号将与地址的3位*Bank*字段（用于确定必须读取的存储区）一同提供给DCCM。在这种情况下，只需读取第二个存储区，因为访问为字对齐访问：*rden_bank = 0x02 (in binary 00000010)*。

- **周期i+2：** 从DCCM获取读数据，并将其提供给内核。由于该访问为对齐访问，两个读信号相等，内核只能有效地使用*dccm_data_lo_dc2*（实验13中同样给出了具体的说明）：
 - *dccm_rd_data_lo = 0x4400000009 → dccm_data_lo_dc2 = 0x00000009*
 - *dccm_rd_data_hi = 0x4400000009 → dccm_data_hi_dc2 = 0x00000009*
- **周期i+8：** 如实验13的附录所述，将读取值加1（立即数）（*0x00000009 + 1 = 0x0000000A*）并遍历存储缓冲区后，系统会将数据和地址提供给DCCM，并使用以下信号为相应的缓冲区使能写访问：
 - *dccm_wren = 1*
 - *wren_bank = 0x02*（二进制值为00000010；即第二个存储区）
 - *dccm_wr_addr = 0x0024*
 - *dccm_wr_data = 0x420000000A*

任务： 解释如何在模块*lsu_dccm_mem*的第103、104和105行中获取信号*rden_bank*、*wren_bank*和*addr_bank*。

任务： 仿真DCCM的未对齐读访问，分析DCCM内部的处理方式。仍可使用上述程序（`[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/`），只需将装载指令进行如下替换：

```
lw t3, (t4) → lw t3, 1(t4)
```

任务： 通过修改图4中的程序（`[RVfpgaPath]/RVfpga/Labs/Lab20/LW-SW_Instruction_DCCM/`），仿真DCCM存储区的冲突。

第1项修改： 删除20条nop指令，重新生成仿真，并随机选取循环的某次迭代以分析lw和sw。

第2项修改： 修改sw指令的立即数，使lw和sw尝试在同一周期内访问同一存储区：

```
sw t3, (t4) → sw t3, 8(t4)
```

3. 基准测试

如需对处理器进行基准测试，应运行程序（或程序集）并测定处理器性能。通过在多个处理器上运行相同的基准（即程序集），可对这些处理器进行比较。我们引入了两种常用的基准：

CoreMark和**Dhrystone**。这些基准位于文件夹

`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks`中。我们接下来详细介绍这些基准以及实验5中的图像处理程序。

文件夹`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters`中提供了专用于RVfpga系统的CoreMark基准PlatformIO项目。我们使用Chips Alliance提供的源代码（<https://github.com/chipsalliance/Cores-SweRV>）对CoreMark进行了修改，使其能够适用于RVfpga系统。在任何基准测试中，我们均使用硬件计数器（HW计数器）来测量各种处理器事件，如执行的指令数和处理器周期数（参见实验11）。除了修改基准以便使用RISC-V硬件计数器外，我们还添加了一些对使用DCCM/ICCM和编译器优化的支持。

在下一部分，我们将展示如何在各种情景下使用Nexys A7开发板运行CoreMark。

A. 情景1：无编译器优化和DCCM/ICCM

首先，我们将展示如何在先前实验所使用的处理器条件下执行CoreMark基准测试，即采用调试模式，不使用DCCM/ICCM。为此，请按以下步骤操作：

- 在PlatformIO中打开**CoreMark_HwCounters**项目。
- 打开文件**src/Test.c**（参见图6），其中包含程序的**main**函数：
 - o **main**函数首先配置用于测量以下四个事件的硬件计数器：周期数、指令总线事务（指令）和数据总线事务（ld/st指令）。为此，需使用函数 `pspPerformanceCounterSet()`。

- 然后使用两条汇编指令（`li`和`csrrs`）配置SweRV EH1处理器的不同功能，如SweRVref文档的第2.C部分所述。在这种情况下，所有功能均使用默认值。
- 随后程序将执行一个循环，只有开发板上的任何开关状态发生反转时，才会退出循环。该循环的作用是允许用户在执行基准测试并输出结果之前打开串行监视器。
- 然后，程序将调用函数`main_cmark()`，该函数会在文件`src/cmark.c`中自行实现CoreMark基准测试。
- 最后，程序使用函数`printfNexys()`打印四个事件。

```

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  }

```

图6. CoreMark PlatformIO项目中的`src/Test.c`文件

- 简要分析在文件`src/cmark.c`中实现的CoreMark基准测试所包含的函数。请注意，HW计数器在`main_cmark()`函数中启动和停止（第1109-1112行和第1130-1133行），基准测试本身则在上述代码行（第1114-1128行）之间执行。

```

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();

```

图7. CoreMark PlatformIO项目中的src/cmark.c文件

- 在开发板上运行程序。然后根据GSG第6.F部分的说明打开串行监视器。

打开串行监视器后，首先会看到一条重复出现的信息，要求您反转开发板中的开关以执行CoreMark基准测试（参见图8上方的红框）。开关反转后，将执行基准测试并输出结果，如图8所示。

CoreMark会运行循环的多次迭代（可通过文件src/cmark.c中定义的参数ITERATIONS轻松修改迭代次数）。每秒钟完成的迭代次数称为**CoreMark score**（CM）。每MHz的迭代次数为**CM/MHz**。基准测试会给出CM/MHz，该参数也称为Iterat/Sec/MHz（迭代数/秒/兆赫），在本例中的值为0.47。也可以查看硬件计数器提供的值，并使用该值计算CM/MHz。

执行用时约200万个周期，处理了约50万条指令，相应的IPC（每个周期执行的指令数）≈0.25（具体计算过程为：50万条指令/200万个周期 ≈ 0.25）。这样的性能远远无法满足要求：回想一下，由于SweRV EH1处理器为双路超标量处理器，其理想的IPC值应为2。由于处理器要进行大量的数据读/写访问，并且DDR外部存储器的速度较慢，因此处理器的性能明显下降。通过总线处理的数据事务数约为133000。通过总线处理的指令事务数仅为

392, 因为大多数指令访问会在I\$中发生命中。请记住, RVfpga系统中不存在D\$ (数据高速缓存)。

```

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

```

图8. CoreMark基准测试的执行结果

B. 情景2: 使用DCCM

我们接下来在RVfpga系统中使能DCCM, 以便使用DCCM (而不是外部DDR存储器) 完成大部分数据访问。从下文的结果可以看出, 这一更改能够实现预期的性能提升。请按照以下步骤在使用DCCM的RVfpga系统版本上运行CoreMark:

- 到目前为止, 我们在大多数实验中均使用默认链接器脚本 (路径为 `.platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/link.lids`)。但是, 在本例中, 为了利用DCCM来存储程序的某些数据, 我们需要使用随PlatformIO项目提供

的一个特定的链接器脚本，其路径为：

[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM.ld。打开该文件并对其进行检查。图9所示为该文件的一部分，我们将进行简要说明。

图9中最上方的屏幕截图定义了DCCM的存储器部分（称为dccm），对应于图2（b）中为该存储器定义的地址空间：dccm (wxa!ri) : ORIGIN = 0xf0040000, LENGTH = 64K

其余屏幕截图将几个代码部分映射到DCCM存储器：.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

```

图9. CoreMark PlatformIO项目中的ld/link_DCCM.ld文件

- 打开文件`platformio.ini`并取消注释第18行（参见图10），以便程序使用图9中的链接器脚本替代默认链接器脚本。如上文所述，通过这种方式，可在高速DCCM（而非速度较慢的DDR存储器）中访问大部分数据。



```

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

```

图10. CoreMark PlatformIO项目中的platformio.ini文件

- 在开发板上运行程序，并打开串行监视器。然后，反转开发板上的某个开关。得到的结果如图11所示。

在本例中，CM/MHz（即Iterat/Sec/MHz）的值为1.88。周期数减少为50万左右。与上一版本一样，处理器会处理大约50万条指令；因此可得出IPC为1。通过将程序的各个部分映射到DCCM，处理器的性能提高了四倍。

最后，由于数据存储存储在DCCM中，通过总线处理的数据事务数变为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

```

图11. CoreMark基准测试的执行结果

任务：在文件`platformio.ini`（参见图10）中，注释掉第18行并取消注释第19行，以便程序使用以下路径中的链接器脚本：

`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM-ICCM.ld`。分析新的链接器脚本，该脚本使用DCCM存储大部分数据，使用ICCM存储指令。执行CoreMark基准测试，将结果与本部分提供的结果进行比较。在本例中，由于我们的默认RVfpga系统不包括ICCM，请使用您在本实验的第一个任务中创建的比特流或以下路径中的比特流：`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Bitstreams/rvfpganexys_DCCM-ICCM.bit`。

C. 情景3：使用DCCM和编译器优化

我们接下来介绍另一种提高性能的方法：编译器优化。与上一部分相同，我们使用DCCM存储应用程序的大部分数据，但我们另外使能了编译器优化。在此之前，我们均在调试模式下执行程序，未进行编译器优化。要使能编译器优化，请按以下步骤操作：

- 再次使用
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/CoreMark_HwCounters/ld/link_DCCM.ld
路径下的链接器脚本。为此，应打开文件platformio.ini，取消注释第18行（参见图10）并注释掉第19行。
- **使用与先前不同的步骤**在开发板上运行程序：上传常用的比特流，但改为使用PlatformIO的“Project Tasks”（项目任务）中提供的“Upload and Monitor”（上传并监视）选项（参见图12）。

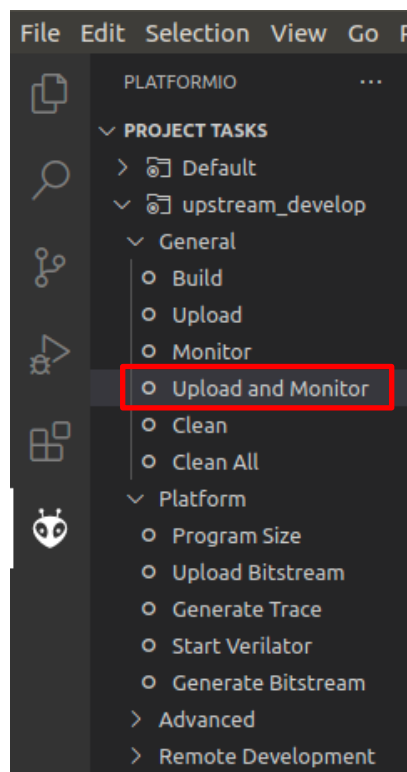


图12. 上传并监视

该选项将编译程序、在开发板上执行程序并打开串行监视器。该选项使用platformio.ini中的build_flags选项所确定的优化标志进行编译，在本例中为-O2（参见图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
```

图13. 文件platformio.ini中的build_flags选项

等待程序开始执行后，像往常一样，反转开发板上的开关。得到的结果如图14所示。

此时，CM/MHz（Iterat/Sec/MHz）的值为3.47。周期数和指令数分别减少为288000和309000左右。虽然IPC仍然约等于1，但此时的性能（CM/MHz，以及对应的执行时间）相比B部分中分析的场景大幅提升，因为周期数和指令数都显著减少。性能得到改善的原因是使能了编译器优化。由于数据存储在DCCM中，数据总线事务数仍为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
```

图14. 使用编译器优化时的CoreMark执行结果

任务：将编译器优化方式改为-O3，执行程序并解释结果。

4. 练习

- 1) 使用Dhrystone基准替代CoreMark基准，执行相同的分析。可访问以下路径获取包含Dhrystone基准的PlatformIO项目：
`[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Dhrystone_HwCounters`。根据所有基准的要求，我们已使用<https://github.com/chipsalliance/Cores-SweRV>中提供的源代码对该Dhrystone基准进行了修改，使其能够适用于特定系统（在本例中为

RVfpga系统)。文件**Test.c**与CoreMark中的文件(图6)类似,但会调用函数**main_dhry()**,该函数包含Dhrystone基准本身。

- 2) 使用图像处理应用程序替代CoreMark,执行相同的分析。可访问以下路径获取包含图像处理应用程序的PlatformIO项目:
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/ImageProcessing_HwCounters。
我们在实验5中曾使用这些应用程序将RGB图像转换为灰度图像。文件**Test.c**与CoreMark中的文件(图6)类似,但会调用函数**ImageTransformation()**,该函数包含我们在实验5中分析的图像转换基准。默认RVfpga系统的DCCM没有足够的空间来存储图像,因此需使用具有128 KiB DCCM的RVfpga系统(比特流),该文件路径为:
[RVfpgaPath]/RVfpga/Labs/Lab20/RealBenchmarks/Bitstreams/rvfpganexys_DCCM-128.bit。
- 3) 根据本实验第2.C部分所述,允许/禁止各种内核功能。比较相应的性能结果(即在这些修改后的内核上执行程序时HW计数器的值)。在Nexys A7开发板上,使用这些修改后的RVfpga系统运行全部三个程序(CoreMark、Dhrystone和图像处理)。可能的变化包括:
 - 使用不同的分支预测器配置和实现方案(如始终不采取分支、使用Gshare分支预测器或使用实验16的练习1中实现的双模分支预测器)。
 - 允许/禁止双发射功能。
 - 使用各种不同的I\$/DCCM/ICCM配置(例如不同的大小或不同的I\$替换策略)。