



IMAGINATION大學計劃

RVfpga實驗13

記憶體存取指令：lw和sw指令

1. 簡介

在前面的實驗中，我們介紹了管線的基本概念及其在SweRV EH1處理器中的使用，並分析了算術-邏輯指令在此處理器中的執行方式。在本實驗中，我們將繼續分析基本指令；具體來說是分析記憶體讀寫。

儲存系統是現代電腦中最關鍵的效能瓶頸之一。記憶體延遲通常遠高於核心時週期，因此處理器可能不得不在等待來自記憶體的資料時暫停。

在本實驗中，首先檢查讀取低延遲儲存單元（即不暫停處理器的儲存單元）時的載入/儲存管道（專門用於執行載入/儲存操作的一組管線階段）。隨後檢查儲存指令的執行。最後，重複我們的分析，忽略低延遲記憶體並直接與Nexys A7板上提供的DDR主記憶體交互。

圖1提供了SweRV EH1處理器微架構的高階檢視。圖中強調顯示了與本實驗相關的階段：解碼、DC1-3（資料存取階段1-3）、提交和寫回。

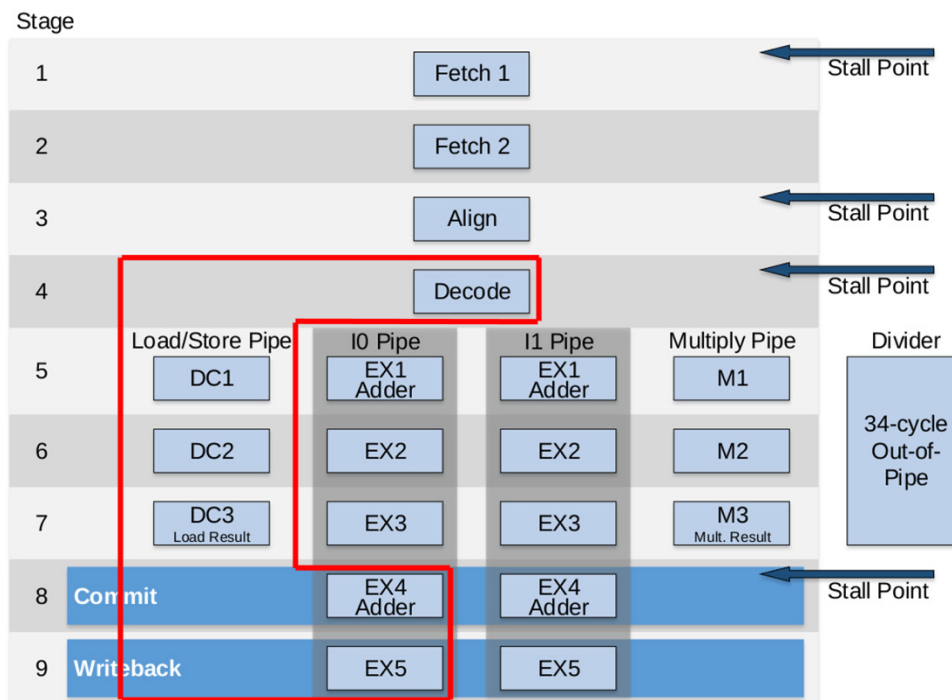


圖1 SweRV EH1核心微架構

（圖來自https://github.com/chipsalliance/Cores-SweRV/blob/master/docs/RISC-V_SweRV_EH1_PRM.pdf）

2. 存取低延遲記憶體的lw指令

在本部分中，我們將使用圖2中的簡單程式碼說明與載入指令的執行最相關的事件。範例程式由一個迴圈組成，此迴圈包含兩個lw（載入字）操作（以紅色強調顯示），每個操作從連續的字對齊記憶體位址讀取一個32位元字。所有迭代均存取相同的資料，但不執行任何操作。

與實驗12一樣，lw指令（圖中以紅色強調顯示）前後存在幾條nop（無操作）指令，以便將其與前方和後方的指令隔離。為簡單起見，在本實驗中，我們還停用了壓縮指令的使用（如SweRVref文件所述）。

```
.globl main

.section .midccm
A: .space 8

.text

main:

# Register t3 = x28 (register 28)
la t0, A                # t0 = addr(A)
li t1, 0x2              # t1 = 2
sw t1, (t0)              # A[0] = 2
add t1, t1, 6            # t1 = 8
sw t1, 4(t0)             # A[1] = 8
INSERT_NOPS_9

REPEAT:
    INSERT_NOPS_1
    lw t1, (t0)
    INSERT_NOPS_9
    INSERT_NOPS_4
    lw t1, 4(t0)
    INSERT_NOPS_10
    INSERT_NOPS_4
    beq zero, zero, REPEAT    # Repeat the loop

.end
```

圖2 具有兩條lw指令的範例程式

資料夾[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM提供PlatformIO專案，以便可以分析、模擬和變更程式。在PlatformIO中開啟、編譯專案，然後開啟反組譯檔案（位於[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis）。在此檔案中，找到第二條lw指令（位於位址0x0000014c處）。請注意指令的機器程式碼（0x0042a303）：

0x0000014c: 0042a303 lw t1, 4(t0)

任務：驗證這些32位元（0x0042a303）是否對應於RISC-V架構中的指令lw t1, 4(t0)。

到目前為止，在入門指南（GSG）和之前的實驗中，我們一直使用Nexys A7板上提供的DDR記憶體來儲存程式中的指令和資料。但是，存取外部記憶體需要幾個週期，這樣便很難分析載入/儲存指令的各個階段；因此，在本部分中，我們將使用SweRV EH1中提供的低延遲DCCM（資料緊密耦合記憶體）來儲存程式資料。

DCCM是與核心緊密耦合的本機記憶體。它提供低延遲存取和SECDED ECC保護¹。它的大小在核心編譯時以引數形式設定，範圍為4 KiB至512 KiB（預設值為64 KiB）。在實驗20中，我們將更詳細地分析DCCM和ICCM；在本實驗中，我們只是使用其來簡化載入/儲存指令的分析。請注意，採用這種方式時，一切操作均在包含SweRV EH1管線和DCCM（以紅色強調顯示）的SweRV EH1核心組合（圖3）內發生。

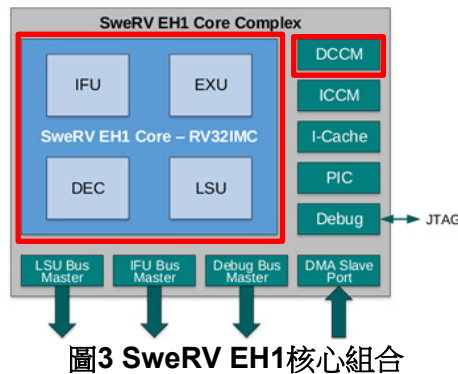


圖3 SweRV EH1核心組合

圖2中的程式碼定義了一個名為`.midccm`的`ad-hoc`（特定的）部分來配置DCCM中的空間。預設情況下，在預設RVfpga系統中，DCCM的位址空間從0xF0040000開始。此專案提供的連結指令碼（位於：`[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/ld/link.lds`）將進行正確的位址指定。透過在檔案`[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/platformio.ini`中包含以下命令來使用此連結器指令碼：

```
board_build.ldscript = ld/link.lds
```

A. 1w指令的基本分析

圖4顯示了圖2中迴圈的中間迭代期間第二條1w指令的執行情況。顯示的訊號是以下檔案中指定的訊號：`[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/scriptLoad.tcl`。請注意，所有迭代均相同：第一次載入將DCCM的第一個資料字（2）讀入`t1`（x6）；第二次載入將DCCM的第二個資料字（8）讀入同一個暫存器（`t1`）。

¹更多詳細資訊，請參見《RISC-V SweRV™ EH1程式設計師參考手冊》。

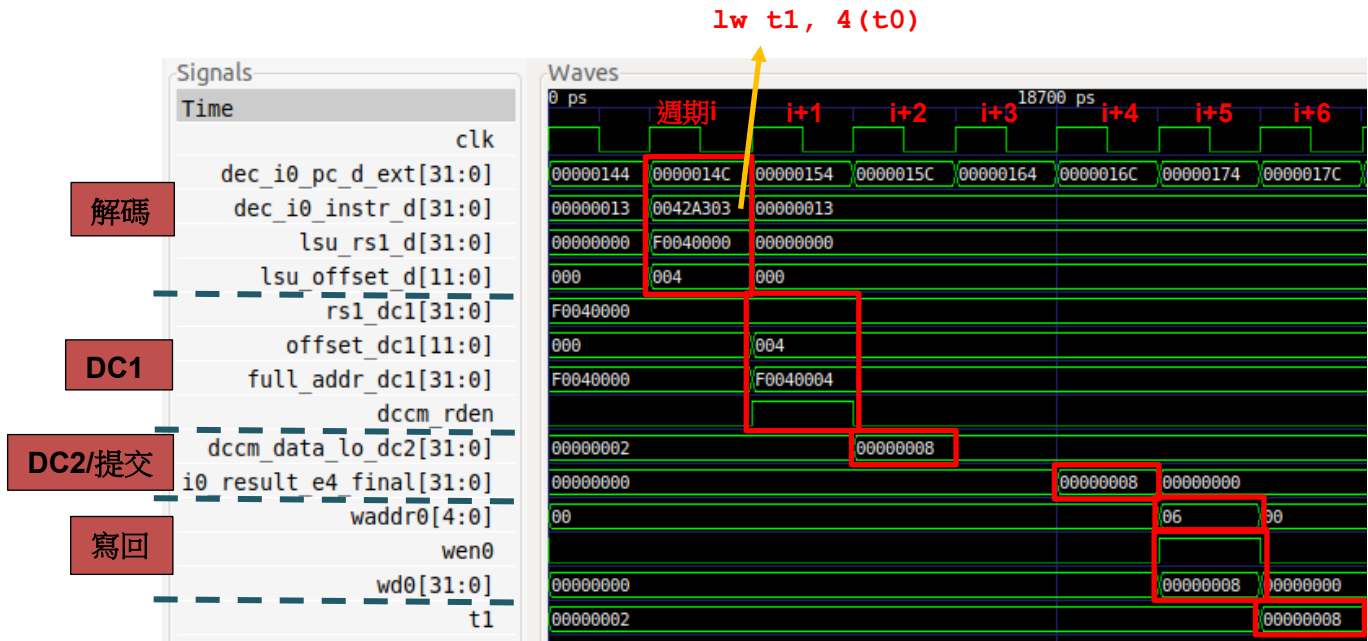


圖4. 圖2中範例程式的Verilator模擬

圖5顯示了第二條`lw`指令執行期間SweRV EH1管線的高階檢視。請注意圖中合併了處理器在不同週期的狀態：

- 週期 i ：對指令進行解碼並讀取暫存器檔案。
- 週期 $i+1$ ：使用加法器計算有效位址。
- 週期 $i+2$ ：使用前一階段計算的位址讀取DDCM。
- 週期 $i+5$ ：將從記憶體讀取的值寫入暫存器檔案。

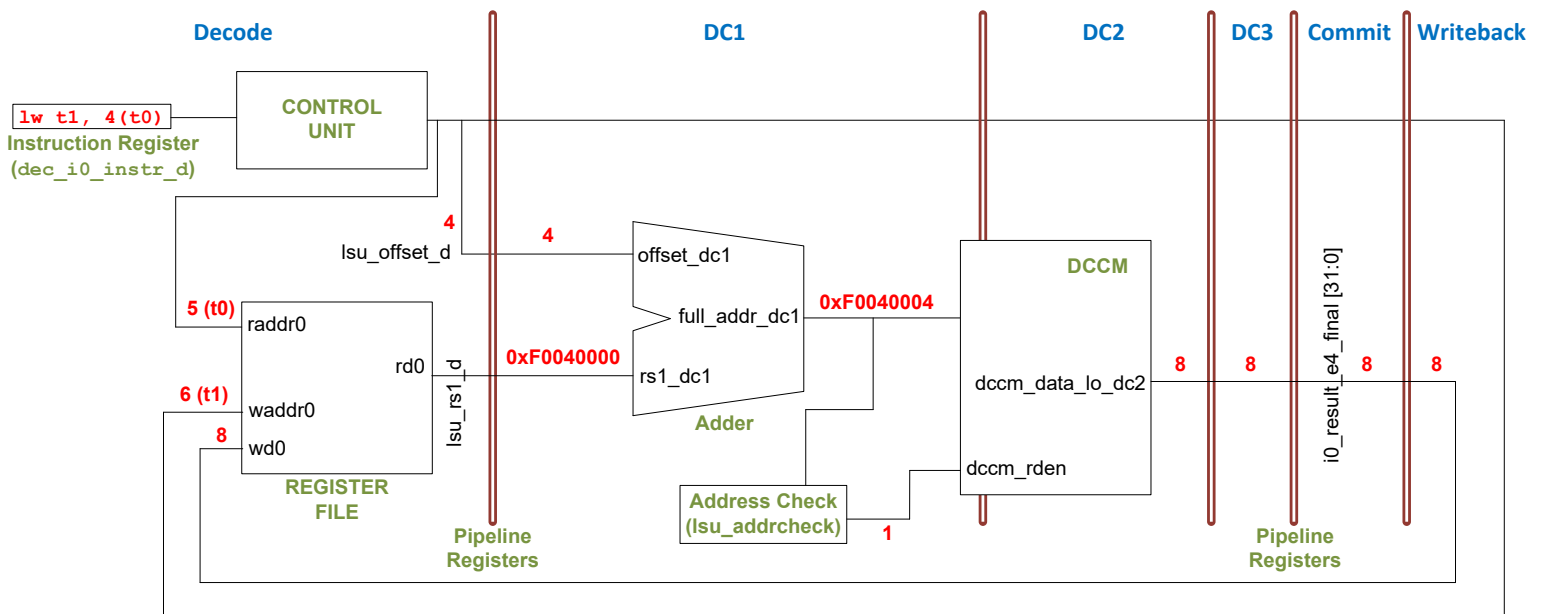


圖5. 在SweRV EH1管線中執行的lw指令的高階檢視

任務：在自己的電腦上重複圖4中的模擬過程。請按照以下步驟操作（如GSG第7部分所詳述）：

- 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/Lab13/LW_Instruction_DCCM。
- 在檔案*platformio.ini*中更正到RVfpga模擬二進位檔案（*Vrvfpgasim*）的路徑。
- 使用Verilator產生模擬軌跡（產生軌跡）。
- 使用GTKWave開啟軌跡。
- 使用檔案*scriptLoad.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM*中提供）開啟與圖4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*scriptLoad.tcl*檔案。
- 按幾次「**Zoom In**」（放大）（）移動至18600 ps。

同時分析圖4中的波形以及圖5中的圖。圖中包括與解碼、DC1-3、提交和寫回階段相關的一些訊號。以紅色強調顯示的值對應於第二條lw指令，因為此指令遍歷相應階段。

- **週期i：解碼：**dec_i0_pc_d_ext保存lw指令的位址（0x0000014C），訊號dec_i0_instr_d包含lw機器指令的32位元（0x0042a303）。

在此階段，將產生控制訊號。此外，還將取得用於計算載入有效位址的運算元：訊號lsu_rsl_d包含lw操作的基本位址（本例中保存在暫存器t0中，等於0xF0040000），訊號lsu_offset_d包含從指令中擷取的12位元有符號立即數（本例中為0x004）。

- **週期i+1：DC1：**使用模組lsu_lsc_ctl內部的加法器計算位址。位址等於基本位址（rsl_dc1 = 0xF0040000）加上符號延伸偏移量（offset_dc1 = 0x00000004）；最終位址full_addr_dc1 = 0xF0040004。檢查此位址（位址檢查）以確定存取的記憶體區域（DCCM、PIC或外部記憶體）。在本例中，假設最終位址屬於DCCM範圍（0xF0040004），dccm_rden置為有效可允許相應DCCM儲存區的讀取。最終位址（full_addr_dc1）和啟用訊號（dccm_rden）提供給DCCM，然後在下一個週期讀取。
- **週期i+2：DC2：**讀取DCCM並將資料置於dccm_data_lo_dc2 = 0x8中，隨後傳播到下一階段。
- **週期i+3：DC3：**將從DCCM讀取的資料傳播到下一階段。
- **週期i+4：提交：**將從DCCM讀取的資料（訊號i0_result_e4_final = 0x8）傳播到下一階段。
- **週期i+5：寫回：**最後，透過訊號wd0 = 0x8將從記憶體讀取的值寫回到暫存器檔案中。鑒於wen0 = 1，此值將在此週期結束時寫入暫存器x6（waddr0 = 0x6）。可以發現，在接下來的週期（圖4中的最後一個週期）中，暫存器x6（也稱為t1）包含新值（t1 = 0x8）。請注意，波形中顯示的訊號t1是.tcl指令碼中為訊號dout定義的別名。

B. lw指令的進階分析

在本部分中，我們將更詳細地分析lw指令遍歷的階段。圖6顯示了本例中的載入指令沿載入/儲存管道（DC1、DC2和DC3階段）執行期間遍歷的主要模組。可能需要將圖放大才能看到詳細資訊。圖中標有**LOGIC**的黑色模組包含多路開關和邏輯門等各種模組。為簡單起見，圖中僅包含模組的部分區塊的介面訊號。

解碼和寫回階段與A-L指令所示的階段相同（參見實驗12中的圖6）。但是，我們指出了解碼階段的一些詳細資訊。回想一下，解碼階段將產生控制訊號並將指令和運算元調度到適當的管道：

- 載入的立即偏移量位於訊號lsu_offset_d中。
- 載入的基本位址位於訊號exu_lsu_rs1_d中。（此訊號由4:1多路開關（如實驗11的圖4所示）產生，然後在遍歷某些邏輯後傳播到DC1階段。）
- 載入/儲存指令的訊號位於lsu_p（圖6所示的新控制訊號封包）中。

與解碼類似，實驗12中也分析了提交階段，但本實驗將包含與載入指令相關的最終3:1多路開關的輸入（lsu_result_corr_dc4），而實驗12為簡單起見對此進行了省略。請記住，此3:1多路開關的輸出為i0_result_e4_final[31:0]，如圖6所示。此外，我們在本實驗中只關注通路0，但載入/儲存可透過雙路超標量處理器中的任何一路執行。不過請注意，只有一個L/S（載入/儲存）管道。因此，通路1也有一個3:1多路開關（其輸出為i1_result_e4_final[31:0]，輸入之一為lsu_result_corr_dc4），如實驗11的圖4所示。

任務：擴展圖 4 中的模擬以包含圖 6 所示的訊號（在下文說明）。可以使用的.tcl 檔案位於：`[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/scriptLoadExtended.tcl`

任務：在SweRV EH1處理器的Verilog檔案中找到圖6中的模組和訊號。

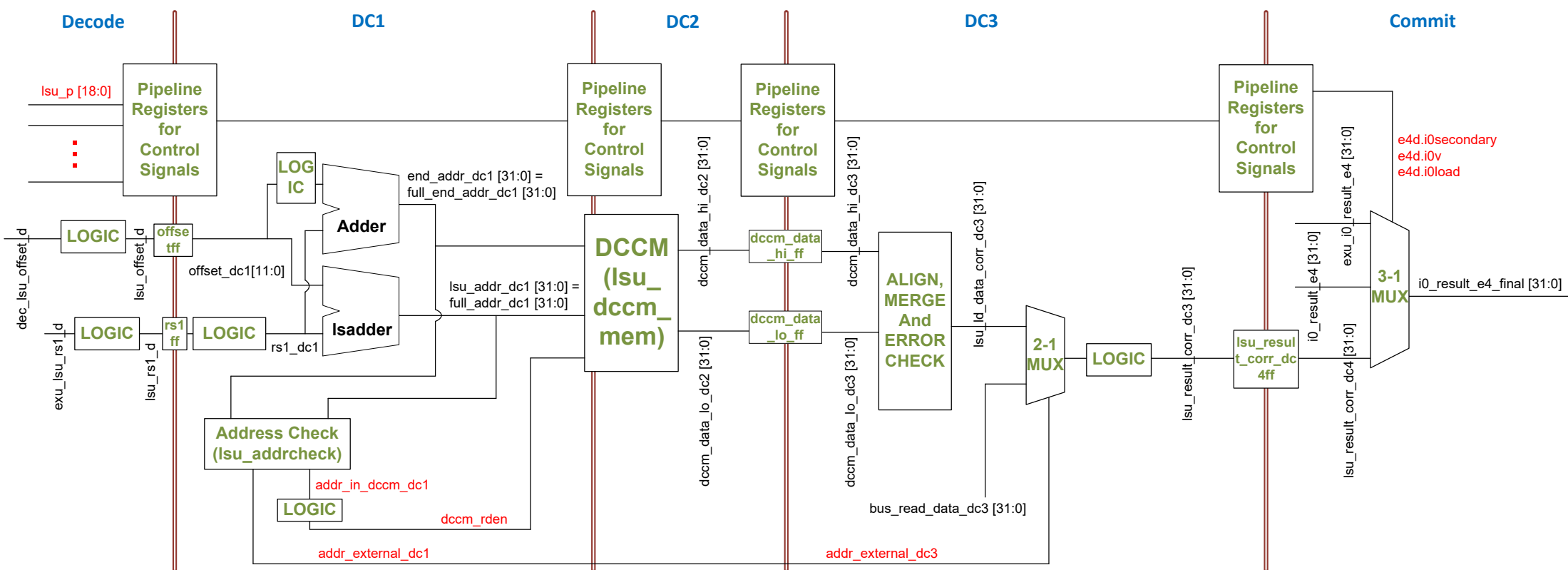


圖6 載入指令在載入/儲存管道中遍歷的主要模組

i. 解碼階段

解碼階段的一般詳細資訊已在實驗11和12中進行了分析。請記住，解碼階段負責兩個主要任務：

- 對指令進行解碼並產生控制訊號。
- 將指令分發到適當的管道並提供輸入運算元。

對指令進行解碼並產生控制訊號：

除了實驗11和12中已經分析過的其他控制訊號結構之外，還有一個附加結構`lsu_pkt_t`，其中包含載入/儲存指令訊號。通常，此結構在檔案
`[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv`中定義。訊號`lsu_p`是此類訊號的一個範例，從解碼階段傳播到載入/儲存管道階段。

此訊號封裝了記憶體讀/寫的一些相關資訊：

- o 位元0 (`valid`) 在操作有效時置1。
- o 位元12 (`unsign`) 在要讀/寫的資料為無符號時置1。
- o 位元13 (`store`) 在儲存操作 (`sb`、`sh`、`sw`...) 時置1。
- o 位元14 (`load`) 在載入操作 (`lb`、`lh`、`lw`...) 時置1。
- o 位元15-18：對存取的大小（位元組、半字、字和雙字）進行編碼。

任務：在圖4的模擬中包含訊號`lsu_p`並根據上述說明分析其各個位元。

將指令分發到適當的管道並提供輸入運算元：

如實驗11中所述，SweRV EH1處理器包括多個用於執行指令的管道。在解碼階段，指令一旦被解碼，就必須透過適當的管道進行調度。在本實驗分析的程式（圖2）中，待執行的`lw`指令傳送到LSU管道（階段DC1-3）。具體來說，`exu_lsu_rs1_d`是基本位址暫存器中保存的值。訊號`dec_lsu_offset_d`是12位元有符號立即偏移量，從指令中擷取並傳送到DC1階段。

任務：在Verilog程式碼中從LSU的兩個輸入（`exu_lsu_rs1_d`和`dec_lsu_offset_d`）的取得來源分析這兩個輸入所遵循的路徑。此過程涉及幾個模組：**dec**、**exu**和**lsu**。

ii. DC1階段

在DC1階段，`rs1_dc1`（基本位址，從解碼階段傳播）和`offset_dc1`（偏移量，從解碼階段傳播）新增到模組**lsadder**中以計算**主要有效位址**（訊號`full_addr_dc1[31:0]`，此位址會指定給`lsu_addr_dc1[31:0]`）。這是要讀取的記憶體位址。

除了要讀取的位址外，**結束位址**（`end_addr_dc1[31:0]`）也在另一個加法器中計算（應強調一點，為簡單起見，第二個加法器不在圖5中顯示，也不在實驗11的圖4中顯示）。這是需要從記憶體中讀取的最後一個位元組的位址。此位址用於處理未對齊存取和子字（位元組和半字）存取。

任務：分析DC1階段中兩個加法器的實作，這兩個加法器在模組`lsu_lsc_ctl`中實例化。我們透過展示這些加法器的實作在下面的圖7中提供指導。

```

185 // generate the ls address
186 // need to refine this is memory is only 128KB
187 rvlsadder lsadder (.rs1(rs1_dc1[31:0]),
188                  .offset(offset_dc1[11:0]),
189                  .dout(full_addr_dc1[31:0])
190                  );

```

```

199 // Calculate start/end address for load/store
200 assign addr_offset_dc1[2:0] = ({3{lsu_pkt_dc1.half}} & 3'b01) | ({3{lsu_pkt_dc1.word}} & 3'b11) | ({3{lsu_pkt_dc1.dword}} & 3'b111);
201 assign end_addr_offset_dc1[12:0] = {offset_dc1[11], offset_dc1[11:0]} + {9'b0, addr_offset_dc1[2:0]};
202 assign full_end_addr_dc1[31:0] = rs1_dc1[31:0] + ({19{end_addr_offset_dc1[12]}}, end_addr_offset_dc1[12:0]);
203 assign end_addr_dc1[31:0] = full_end_addr_dc1[31:0];

```

圖7. DC1階段加法器的Verilog程式碼（來自檔案`lsu_lsc_ctl.sv`）

例如，針對從`0xF0040003`開始的位址的載入字（`lw`）滿足：`full_addr_dc1=0xF0040003`且`end_addr_dc1=0xF0040006`（參見圖8）。這樣，LSU管道便可從讀取資料束中擷取字，讀取資料束由兩個字組成，這兩個字從等於四的倍數的位址開始（本例中為`0xF0040000`和`0xF0040004`）。

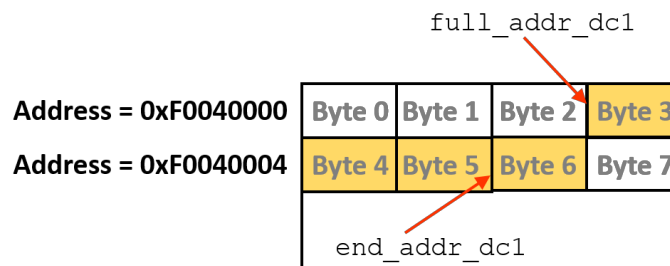


圖8. 針對位址`0xF0040003`的`lw`指令範例

兩個位址（`lsu_addr_dc1[31:0]`和`end_addr_dc1[31:0]`）傳送到資料記憶體（本例中為DCCM），將在下一個週期存取。

任務：在圖2的程式中，嘗試不同的存取大小（位元組和半字）和未對齊存取。為此，請變更偏移量或將存取類型從`lw`變更為`lb`（載入位元組）或`lh`（載入半字）。例如，如果將偏移量從4變更為3，則載入字指令將對從位址`0xF0040003`開始的32位元執行未對齊存取，如圖8所示。分析上述不同情況下訊號`lsu_addr_dc1[31:0]`（或`full_addr_dc1[31:0]`）和`end_addr_dc1[31:0]`的值。
在實驗20中，我們將從DCCM的內部分析這種情況。

除了位址計算之外，DC1階段還在模組**lsu_addrcheck**中執行位址範圍檢查（參見圖9）來確定存取的目標記憶體（本例中為DCCM）。

```
// Module to generate the memory map of the address
lsu_addrcheck addrcheck (
    .start_addr_dc1(full_addr_dc1[31:0]),
    .end_addr_dc1(full_end_addr_dc1[31:0]),
    .*
);
```

圖9. 檢查記憶體位址的範圍和位置

透過位址檢查的結果可確定必須存取的記憶體：DCCM、PIC或外部DDR記憶體（參見圖10）。

```
43 output logic addr_in_dccm_dc1, // address in dccm
44 output logic addr_in_pic_dc1, // address in pic
45 output logic addr_external_dc1, // address in external
```

圖10. 每個記憶體單元的位址

在本例中，DCCM讀啟用訊號變為高電平（`addr_in_dccm_dc1 = 1`）。此訊號在遍歷一些邏輯後提供給DCCM（訊號`dccm_rden`）以啟用/停用存取（本例中啟用存取）。訊號`addr_external_dc1`在必須啟用外部DDR記憶體時為1，其他情況時為0，由DC3階段傳播和使用，如圖6所示。

iii. DC2階段

如果啟用了DCCM讀取（`dccm_rden = 1`），則會在此階段讀取資料。請注意，將讀取兩個32位元值（`dccm_data_lo_dc2[31:0]`和`dccm_data_hi_dc2[31:0]`），因為資料存取可能未對齊，因此分布在兩個字上（例如圖8中的範例）。

任務：在圖2的程式中，當對位址0xF0040004和位址0xF0040003執行lw時，比較訊號`dccm_data_lo_dc2[31:0]`和`dccm_data_hi_dc2[31:0]`的值。

iv. DC3階段

來自DCCM的兩個32位元資料值從DC2（訊號`dccm_data_lo_dc2[31:0]`和`dccm_data_hi_dc2[31:0]`）傳播到DC3（訊號`dccm_data_lo_dc3[31:0]`和`dccm_data_hi_dc3[31:0]`）。對於對齊存取（例如本例中的存取），兩個訊號相等，僅使用`dccm_data_lo_dc3[31:0]`。

在DC3階段，前一個週期讀取的兩個字（`dccm_data_lo_dc3[31:0]`和`dccm_data_hi_dc3[31:0]`）通過執行多項任務的邏輯：

- **錯誤檢查**：使用ECC檢查資料是否存在錯誤。
- **處理載入/儲存冒險**：如果針對同一位址的儲存指令仍在執行，則將資料從儲存指令轉送到載入指令，而不是從記憶體中讀取。我們將在實驗15中分析這種情況。
- **對齊**：對齊要求的資料。

上述所有過程的結果是，lsu_ld_data_corr_dc3[31:0]訊號中提供最終資料。

任務：分析lsu_dccm_ctl和lsu_ecc模組中的Verilog程式碼中使用的對齊、合併和錯誤檢查邏輯。

任務：在圖2的程式中，當對位址0xF0040004和位址0xF0040003執行lw時，比較訊號lsu_result_corr_dc3[31:0]的值。

在執行錯誤檢查、載入/儲存冒險處理和對齊的邏輯之後，2:1多路開關將在來自DCCM的資料（lsu_ld_data_corr_dc3[31:0]）或來自DDR記憶體的資料（bus_read_data_dc3[31:0]）之間進行選擇。此多路開關由訊號addr_external_dc3控制，該訊號於DC1階段在模組lsu_addrcheck中產生（訊號addr_external_dc1）。

任務：在Verilog程式碼中分析訊號addr_external_dc1如何於DC1階段在模組lsu_addrcheck中計算。

此2:1多路開關的輸出（lsu_result_corr_dc3[31:0]）傳播到提交階段。

v. 提交階段

在提交階段，3:1多路開關選擇要傳送到寫回階段的讀取資料（io_result_e4_final[31:0]）（參見圖6）。此3:1多路開關還可選擇ALU的輸出（實驗11和12中已進行說明）以及輔助ALU的輸出（將在實驗15中分析）。

vi. 寫回階段

此階段已在實驗11和12中說明，因此在圖6中未顯示，其中加法的結果已寫入目標暫存器。在這種情況下，此階段將DCCM資料寫入目標暫存器。

3. 存取低延遲記憶體的sw指令

在本部分中，我們將使用圖11顯示的程式碼說明與執行儲存指令最相關的事件。此程式碼包含一個具有1000次迭代的迴圈，此迴圈將寫入記憶體的連續位址。向量A包含1000個字，位於DCCM（0xF0040000 – 0xF004FFFF）。每條sw指令後跟一條lw指令，前者用於檢查是否已儲存正確的值。通常會插入nop來隔離指令，在這種情況下，還需確保資料實際寫入暫存器以及從暫存器讀取，而不僅僅是從sw指令轉送到lw。通常將停用壓縮指令的使用（如

SweRVref文件中所描述)。此外，與前一部分的範例一樣，我們將使用DCCM來儲存和載入資料。

```
.globl main

.section .midccm
A: .space 4000

.text

main:
la t0, A           # t0 = addr(A)
li t1, 0x2         # t1 = 2
li t2, 1000        # t2 = 1000
INSERT_NOPS_2

REPEAT:
    sw t1, (t0)
    INSERT_NOPS_10
    INSERT_NOPS_4
    lw t1, (t0)
    INSERT_NOPS_10
    add t1, t1, t1
    add t0, t0, 0x04
    add t2, t2, -1
    INSERT_NOPS_10
    bne t2, zero, REPEAT # Repeat the loop
    nop
    nop

.end
```

圖11 使用sw指令的範例程式碼

資料夾[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM提供PlatformIO專案，以便可以分析、模擬和變更程式。在PlatformIO中開啟、編譯專案，然後開啟反組譯檔案（位於[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis）。可以看到sw指令位於位址0x00000110處，還可以看到指令的機器程式碼（0x0062a023）：

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

任務：驗證這些32位元（0x0062a023）是否對應於RISC-V架構中的指令sw t1, 0(t0)。

圖12顯示了圖11中迴圈的第四次迭代期間sw指令的執行情況。可以分析除第一個迭代之外的任何迭代。通常，不應使用指令的第一次執行以避免指令快取（I\$）未命中。

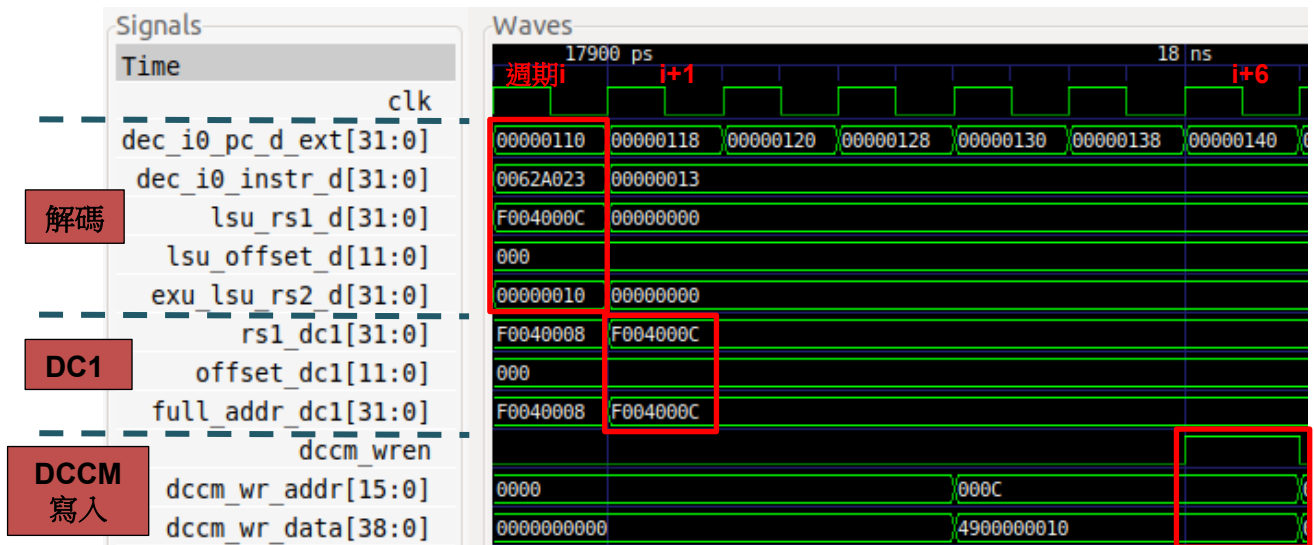


圖12 圖11範例的Verilator模擬

圖13顯示了圖11中迴圈的第四次迭代期間執行sw指令時SweRV EH1管線的高階檢視。暫存器t1（保存要寫入記憶體的值）為0x10，t0（保存基本位址）為0xF004000C。因此，sw將值0x10寫入DCCM位址0xF004000C。圖中顯示了SweRV EH1處理器的Verilog模組中使用的實際名稱。請注意圖中合併了處理器在不同週期的狀態：

- **週期i：** store指令在解碼階段進行解碼，並指定給LSU管道，在這種情況下，運算元由在此週期中讀取的指令立即數欄位和暫存器檔案提供。
- **週期i+1：** 有效位址在加法器單元計算（如load部分所述）。請注意，為簡單起見，圖中僅包含圖6所示的lsadder。
- **週期i+6：** 第二個運算元（從暫存器t1讀取）在遍歷儲存緩衝區後儲存在DCCM中（相關說明請參見附錄）。

請注意，就程式執行時間而言，儲存不是關鍵操作，因此它可以延遲幾個週期而不影響效能。相比之下，載入指令十分關鍵，因為它們通常會讀取後續指令所需的值，因此（如前一部分所述）實作的是儲存-載入轉送路徑（圖13中未顯示），這樣有助於減少記憶體存取，並避免在針對同一記憶體位址的儲存和後續載入操作之間發生資料冒險時管線暫停。我們將在實驗15中分析這種情況。

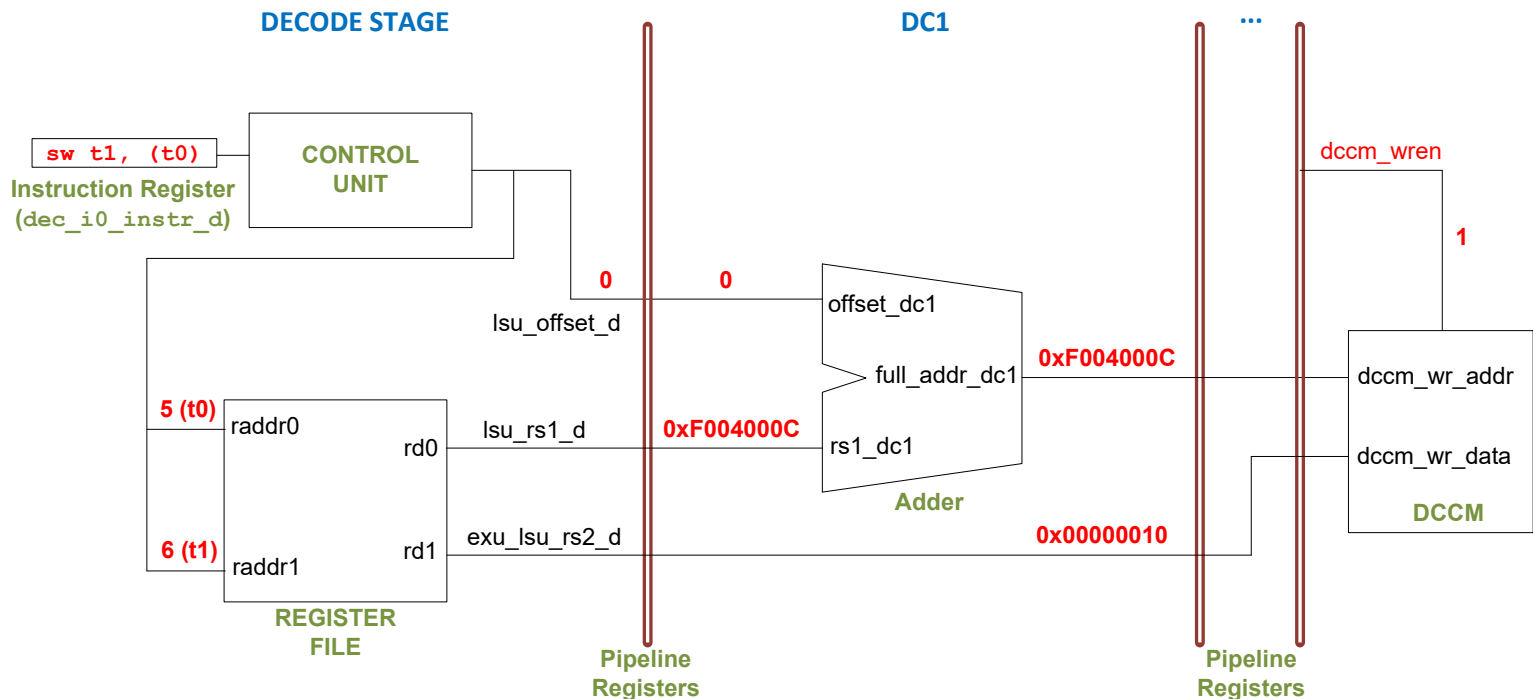


圖13 SweRV EH1中`sw`指令執行的高階檢視

任務：在自己的電腦上重複圖12中的模擬過程。請按照以下步驟操作（如GSG第7部分所詳述）：

- 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/Lab13/SW_Instruction_DCCM。
- 在檔案platformio.ini中更新到RVfpga模擬二進位檔案（*Vrvfpgasim*）的路徑。
- 使用Verilator產生模擬軌跡（產生軌跡）。
- 在GTKWave上開啟軌跡。
- 使用檔案scriptStore.tcl（在[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM/中提供）顯示與圖4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scriptStore.tcl檔案。
- 按幾次「Zoom In」（放大）（）移動至17900 ps。

同時分析圖12中的波形以及圖13中的圖。圖中包括一些與解碼和DC1階段相關的訊號，以及一些與DCCM寫入相關的訊號，寫入發生在幾個週期之後。以紅色強調顯示的值對應於`sw`指令，因為該指令遍歷相應階段。

- **週期i：** **解碼：**如載入指令部分所述，訊號dec_i0_pc_d_ext包含`sw`指令的地址（0x00000110），訊號dec_i0_instr_d包含32位元`sw`指令（0x0062a023）。訊號lsu_rs1_d包含`sw`操作的基本位址（本例中為0xF004000C，由暫存器t0提供），訊號lsu_offset_d包含從指令中擷取並隨後新增到基本位址的12位元立即數（本例中為0x000）。對於儲存指令，從第二個暫存器（本例中為t1）讀取的值最終寫入記憶體（exu_lsu_rs2_d = 0x10）。因此，它必須傳播到後續階段。

- **週期i+1：** **DC1：**正如載入部分所述，此階段計算位址（ $\text{full_addr_dc1} = \text{rs1_dc1} + \text{offset_dc1} = 0xF004000C$ ）。
- **週期i+6：** **DCCM寫入：**五個週期後，DCCM從儲存緩衝區收到寫入資料和寫入位址（ $\text{dccm_wr_addr}=0x000C$ 和 $\text{dccm_wr_data}=0x4900000010$ ）。請注意，DCCM僅接收位址（ $0x000C$ ）的最後16位元，因為在我們的組態中其實大小為64 KiB（參見檔案`common_defines.vh`），16位元足以定址 2^{16} 個位元組。資料已在前面加上了一些ECC位元（ $0x49$ ）。當訊號`dccm_wren`置為有效時（在圖12的週期i+6中），對DCCM的寫入操作完成。

附錄A – 儲存緩衝區的操作：附錄A將說明儲存緩衝區，這是一個重要結構，用於暫時保存儲存指令必須寫入記憶體的值和位址。

任務：在模擬中分析儲存指令之後的載入指令，以驗證值是否已正確寫入DCCM。需要新增圖4和圖6中的一些訊號來分析載入。

任務：按照與第2.B部分中對lw指令執行的進階分析類似的方式擴展本部分中對sw指令執行的基礎分析。

任務：分析針對DCCM的未對齊儲存以及子字儲存：儲存位元組（sb）或儲存半字（sh）。

4. 存取外部記憶體

在第2部分和第3部分中，我們使用DCCM來儲存和載入資料。在本部分中，我們將分析存取Nexys A7上所提供外部記憶體的載入指令。請注意，這種情況（參見圖14）與第2部分中分析的情況（參見圖3）相反，SweRV EH1核心必須透過AXI匯流排與外部記憶體通訊，才能取得載入指令要求的資料。

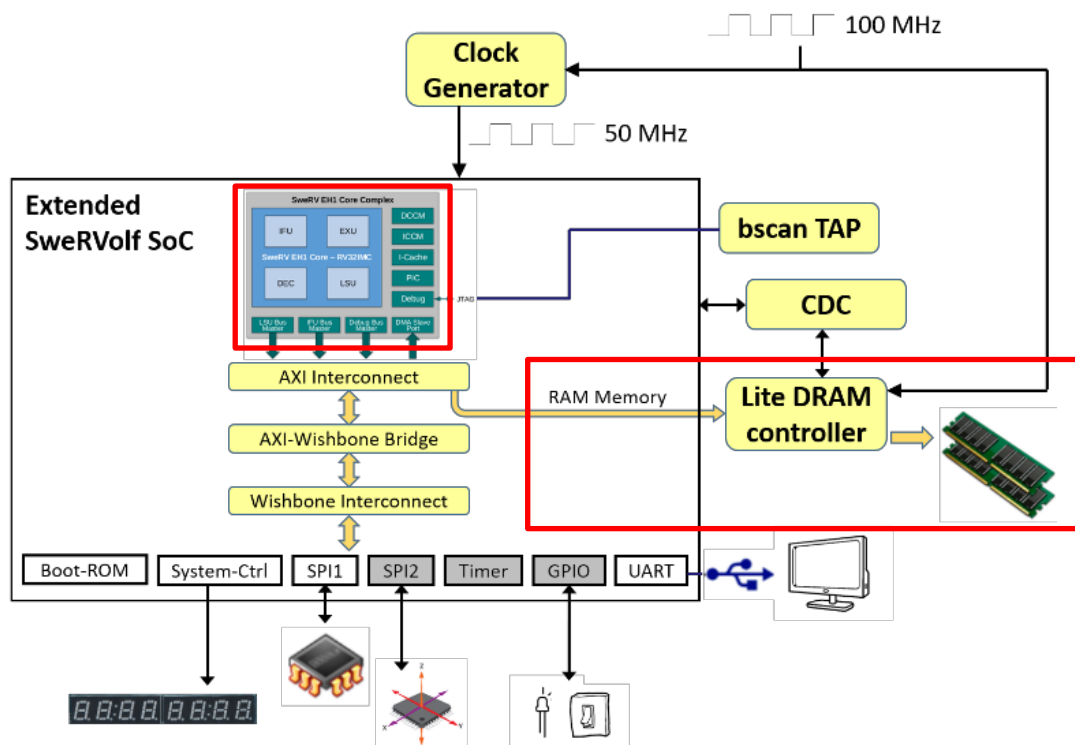


圖14. RVfpgaNexys

我們將分析阻塞和非阻塞存取。阻塞載入會完全停止處理器，直到其接收到從記憶體讀取的資料為止。這意味著在載入接收到其資料之前沒有其他指令執行。相比之下，只要指令不依賴於載入讀取的資料，非阻塞載入便允許程式繼續執行；僅當執行依賴於載入的指令時，才會停止執行。在這兩種情況下，從記憶體中讀取的資料遵循管線中的不同路徑；在本實驗中，我們將分析第一種情況（阻塞載入），而在下一個實驗（實驗14）中，我們將在結構冒險的背景分析第二種情況（非阻塞載入）。

圖15中的程式碼描繪了說明如何執行讀取外部DDR記憶體的lw指令的簡單範例。此程式碼包含一個迴圈，此迴圈讀取一個12元素的陣列（lw t3, (t4)）並在暫存器t6中累加其元素的總和（add t6, t3, t6）。通常將插入幾個nop操作來隔離指令，以使指令更易於分析，此外還會停用壓縮指令。

向量D包含12個字，位於主記憶體中。為此，需在.data部分中宣告陣列並為專案使用一般連結器指令碼（位於~/platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/link.ld）。這樣，.data部分中定義的資料將置於外部記憶體中，而不是像圖2的程式中一樣置於DCCM中。

預設情況下，SweRV EH1中的載入指令是非阻塞的。如果希望載入指令為阻塞指令，則必須在要分析的組合語言程式碼的開頭包含接下來的兩條指令，如圖15所示（有關啟用/停用核心功能的更多說明，請參見SweRVref文件的第2.C部分）：

```
li t2, 0x020
csrrs t1, 0x7F9, t2
```

```
.globl main

.data
D: .word 3,5,6,8,7,10,12,2,1,4,11,9

.text
main:

    li t2, 0x020
    csrrs t1, 0x7F9, t2

    la t4, D
    li t5, 12
    li t6, 0x0
    INSERT_NOPS_1

REPEAT:
    lw t3, (t4)
    add t5, t5, -1
    INSERT_NOPS_10
    add t6, t3, t6
    add t4, t4, 4
    INSERT_NOPS_9
    bne t5, zero, REPEAT    # Repeat the loop

    INSERT_NOPS_4

.end
```

圖15 阻塞lw指令的範例

資料夾[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory提供PlatformIO專案，以便可以分析、模擬和變更程式。如果在PlatformIO中開啟、編譯專案，然後開啟反組譯檔案（位於[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory/.pio/build/swervolf_nexys/firmware.dis中），可以看到lw指令位於位址0x000000f4處，還可以看到指令的機器程式碼（0x000eae03）：

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

存取外部DDR2記憶體體的阻塞載入的路徑幾乎與第2部分中所述存取DDCM的載入的路徑相同，如圖16所示。但是，有一個重要的區別：在某些週期內，處理器將暫停來等待外部記憶體體提供的資料；隨後，當收到要求的資料時，指令可以繼續執行。

在SweRV EH1中，透過AXI匯流排控制外部記憶體存取的模組稱為lsu_bus_intf。它負責向Lite DRAM控制器提供位址，並在一些週期後接收和對齊要求的資料，然後在DC3階段將其插入核心。請注意，AXI匯流排用於與DDR2外部記憶體通訊。在本例（圖15）中，DC3階段的2:1多路開關（也包含在圖6中）選擇來自外部記憶體體的輸入（即lsu_result_corr_dc3 = bus_read_data_dc3）而不是圖2範例中所選擇的來自DDCM的輸入（lsu_ld_data_corr_dc3）。提交階段的3:1多路開關則選擇與圖2範例中的輸入相同的輸入（即i0_result_e4_final = lsu_result_corr_dc4）。

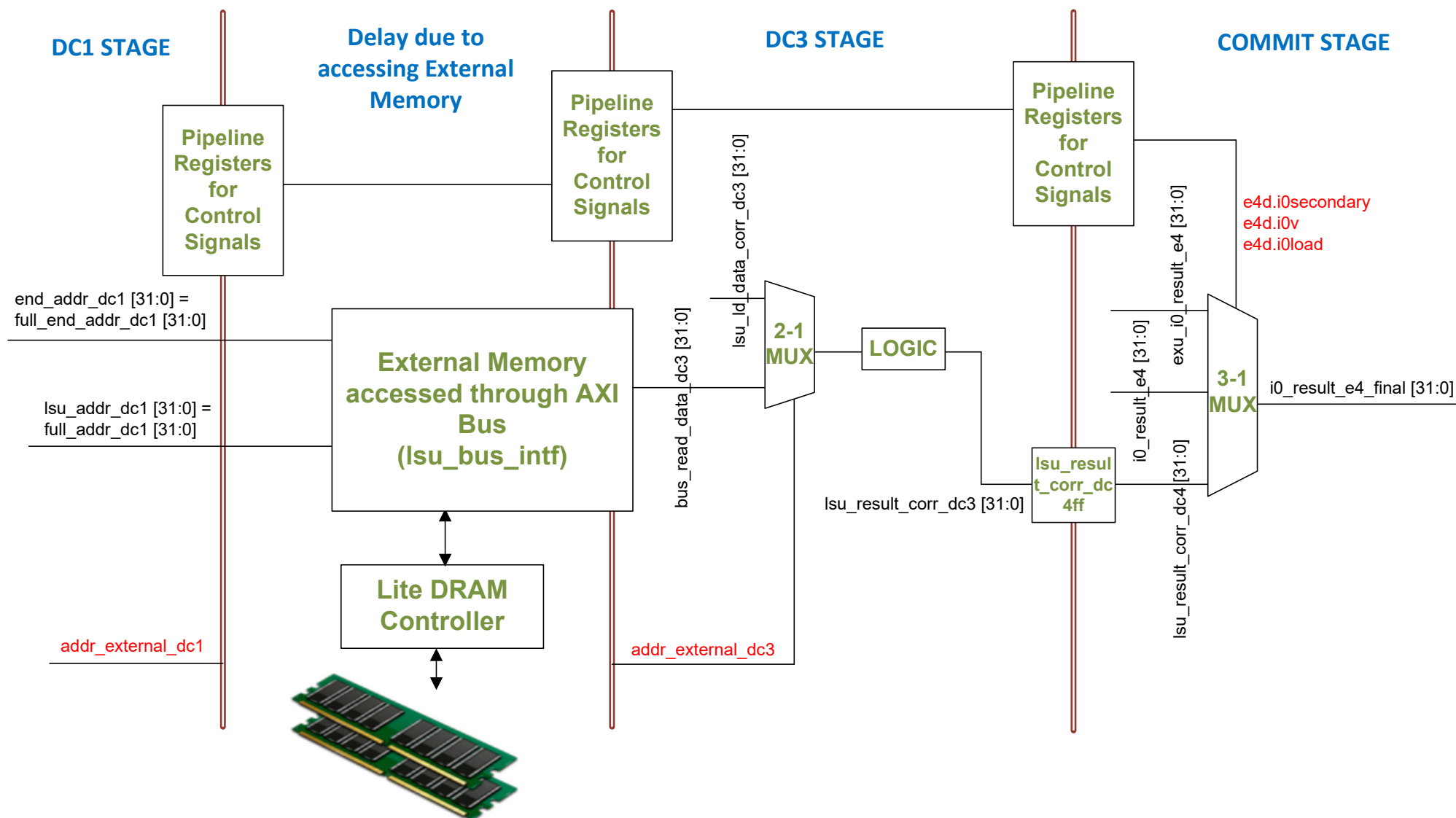


圖16. 存取外部記憶體的阻塞lw指令

圖17顯示了圖15中迴圈的第四次迭代期間lw的執行情況，執行期間會將儲存在位址0x00002204中的值讀入暫存器t3。請注意，對於此程式，D陣列從位址0x000021F8開始。

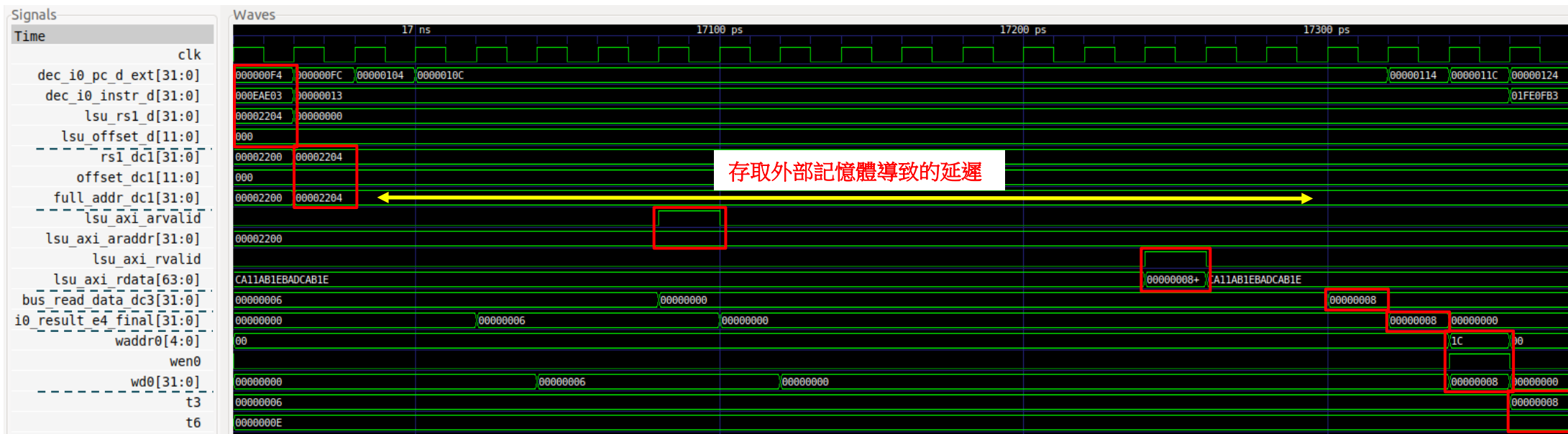



圖17. 圖15中範例的Verilator模擬

任務：在自己的電腦上重複圖17中的模擬過程。使用檔案`test_Blocking.tcl`（在`[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory`中提供）。按幾次「Zoom In」（放大）（）移動至16940 ps。

分析波形。圖中包括與各管線階段相關聯的一些訊號。請注意，頂部訊號組（clk至full_addr_dc1）和底部訊號組（i0_result_e4_final至wd0）與圖4所示的訊號組相同。以紅色強調顯示的值對應於lw指令，因為該指令遍歷相應階段。

- 位址在解碼階段計算，如第2部分所述。訊號full_addr_dc1[31:0]包含位址，本例第四次迭代中的位址（圖17所示的位址）為0x00002204。訊號end_addr_dc1[31:0]（圖中未顯示）也按照第2部分所述進行計算，其中包含要存取的最後一個位元組的位址。
- 幾個週期後，位址透過AXI匯流排使用以下訊號傳送到外部記憶體：
lsu_axi_arvalid = 1和lsu_axi_araddr = 0x00002200。請注意，傳送的位址是雙字對齊的，因為每次存取都會從記憶體中讀取64位元。資料讀入訊號lsu_axi_rdata（在實驗19和20中，我們將詳細分析記憶體層級）。如果存取需要多個位址（由於未對齊存取），則透過匯流排傳送多個位址並按順序返回資料。

任務：修改圖15中的程式以分析需要透過AXI匯流排向外部記憶體傳送兩個位址的未對齊載入存取。

- 幾個週期後，外部記憶體透過AXI匯流排傳回讀取的64位元資料（lsu_axi_rdata = 0x0000000800000006和lsu_axi_rvalid = 1）。此資料在LSU（模組lsu_bus_buffer）內部緩衝。
- 要求的32位元資料透過記憶體讀取的64位資料擷取，將插入到主管線路徑中：
bus_read_data_dc3 = 0x00000008。
- 隨後，此資料遵循與第2部分中的範例相同的路徑寫入暫存器檔案：
i0_result_e4_final → wd0。

任務：將控制多路開關的訊號新增到模擬中（在圖16的DC3和提交階段），其中多路開關選擇由DDR外部記憶體提供的資料。可以在Verilog程式碼的以下幾行中找到這些多路開關：

- 2:1多路開關：模組lsu_lsc_ctl的第264行。
- 3:1多路開關：模組dec_decode_ctl的第2277行。

可以使用的.tcl檔案位於：

`[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory/test_Blocking_Extended.tcl`

任務：分析用於存取DRAM控制器的AXI匯流排實作也很有趣，為此可以檢查lsu_bus_intf模組。

附錄A – 儲存緩衝區的操作

儲存緩衝區是位於LSU內部的一個8項目迴圈佇列，在此佇列中，會追蹤每個儲存（sw）操作，同時註記其目標位址和資料。通常，儲存緩衝區可用於：

- 滿足先前儲存資料的後續載入操作（如果其目標位址相符）。這種資料轉送消除了「寫後讀」冒險並減少了記憶體存取。
- 令獨立的載入操作能夠快速執行而不用等待先前待處理的儲存操作（鑒於載入操作很可能處於關鍵路徑中）。
- 將相容的儲存操作合併為單個操作，進而解決「寫後寫」冒險並減少記憶體存取。

圖18顯示了執行圖11中的程式碼時儲存緩衝區的一些相關訊號。這些新訊號將新增到圖12所示的訊號中。與該圖一樣，圖18顯示了迴圈的第四次迭代期間sw指令的執行情況。

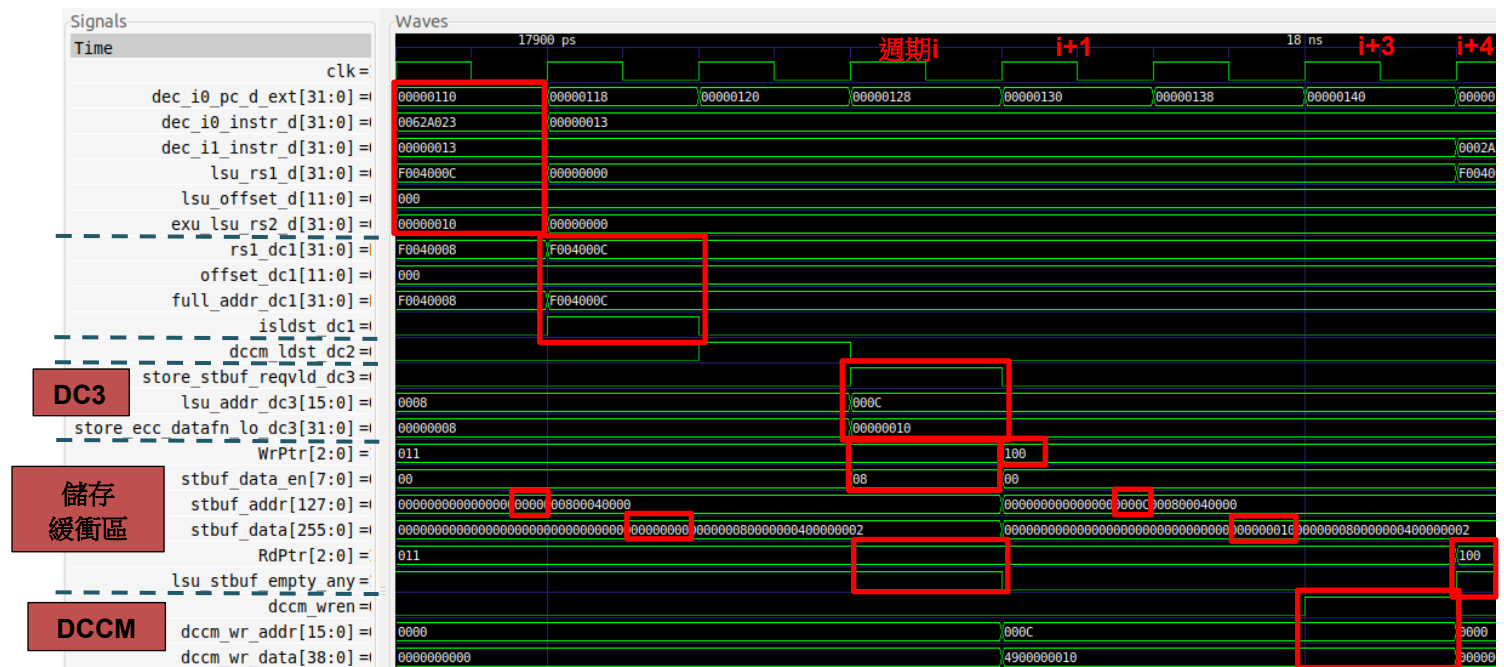



圖18. 圖11中範例的Verilator模擬，用於說明儲存緩衝區操作

任務：在自己的電腦上重複圖18中的模擬過程。使用檔案 `scriptStoreBuffer.tcl`（在 `[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM` 中提供）。按幾次「Zoom In」（放大）（）移動至17900 ps。

頂部訊號（來自解碼、DC1和DC2階段）在圖12中顯示和說明，因此此處不再闡述。DC3（圖中的週期i）準備寫入儲存緩衝區。儲存指令要寫入記憶體的最終位址和資料透過訊號 `lsu_addr_dc3 = 0x000C` 和 `store_ecc_datafn_lo_dc3 = 0x00000010` 傳送到儲存緩衝區。當DC3階段識別出儲存操作時，訊號 `store_stbuf_reqvld_dc3` 置為有效，這將觸發儲存緩衝區操作。

下一組訊號對應於內部儲存緩衝區訊號（位於模組lsu_stbuf中）。WrPtr對儲存緩衝區的項目進行編碼，下一個sw操作將在儲存緩衝區中放置其位址和資料。在範例中，WrPtr為0b011（即項目編號3，這是第4個項目，因為編號從0開始）。

DC3階段（週期i）透過將訊號stbuf_data_en的第4位元置為有效來啟用儲存緩衝區的第4個項目（請注意，0x08以獨熱編碼方式轉換為00001000，並且唯一的「1」值位於第4位元的位址）。訊號lsu_stbuf_empty_any在此週期結束時變為低電平，表示儲存緩衝區不為空 – 也就是說，儲存緩衝區保存等待寫入記憶體的资料。

在提交階段（週期i+1），更新儲存緩衝區的第4個項目。訊號stbuf_addr和stbuf_data的第4個項目中包含：0x000C（對應於要寫入的DCCM位址）和0x00000010（對應於要儲存在DCCM中的資料）。WrPtr已遞增為指向下一個緩衝區項目（b100），並且RdPtr會追蹤緩衝區中尚未提交的最早值（b011）。

在寫回階段後的一個週期（週期i+3），DCCM寫入啟用訊號（dccm_wren）置為有效，以便寫入記憶體並釋放緩衝區的第4個項目。最後，在週期i+4，更新RdPtr（b100）並且緩衝區再次為空，這樣lsu_stbuf_empty_any就會再次變為高電平。

圖19說明了在圖18顯示的範例中的8項目儲存緩衝區如何變化。在週期i，儲存緩衝區為空，此狀態由WrPtr == RdPtr表示。在週期i+1，儲存緩衝區包含一個位址/資料對（0x000C/0x00000010），對應於圖18中分析的儲存。最後，在週期i+4，資料寫入DCCM，儲存緩衝區再次變為空（WrPtr == RdPtr）。

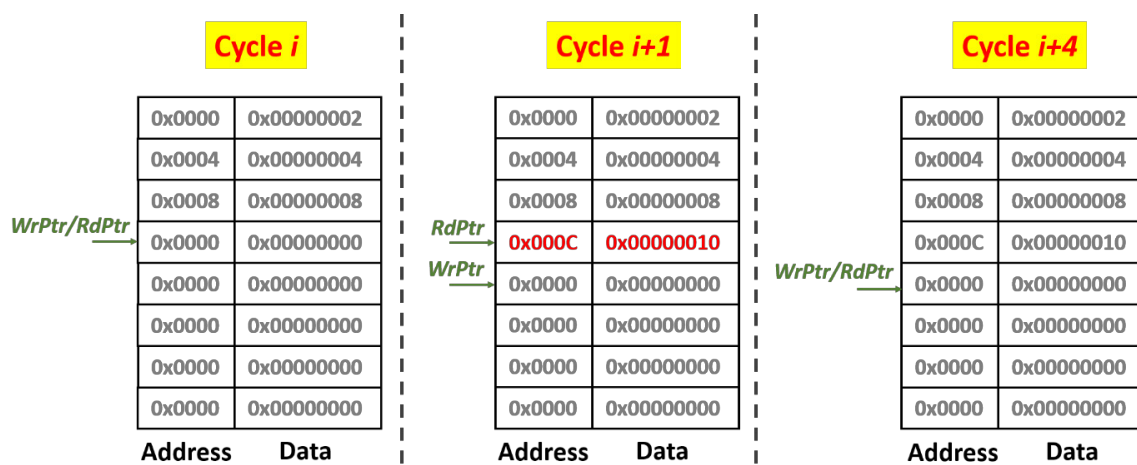


圖19. 圖18的範例期間儲存緩衝區的變化

任務：修改圖11中的程式以實現兩個未完成的儲存操作，並執行與圖18中的分析類似的分析。