



IMAGINATION大學計劃

# RVfpga實驗19

## 指令快取

## 1. 簡介

在本實驗及下一實驗中，我們將重點討論RVfpga系統的記憶體系統。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圖1），RVfpga系統包含一個外部DDR主記憶體、一個指令快取（I\$）和兩個分別用於儲存資料（DCCM）和指令（ICCM）的暫存記憶體（也稱為緊密耦合記憶體）。

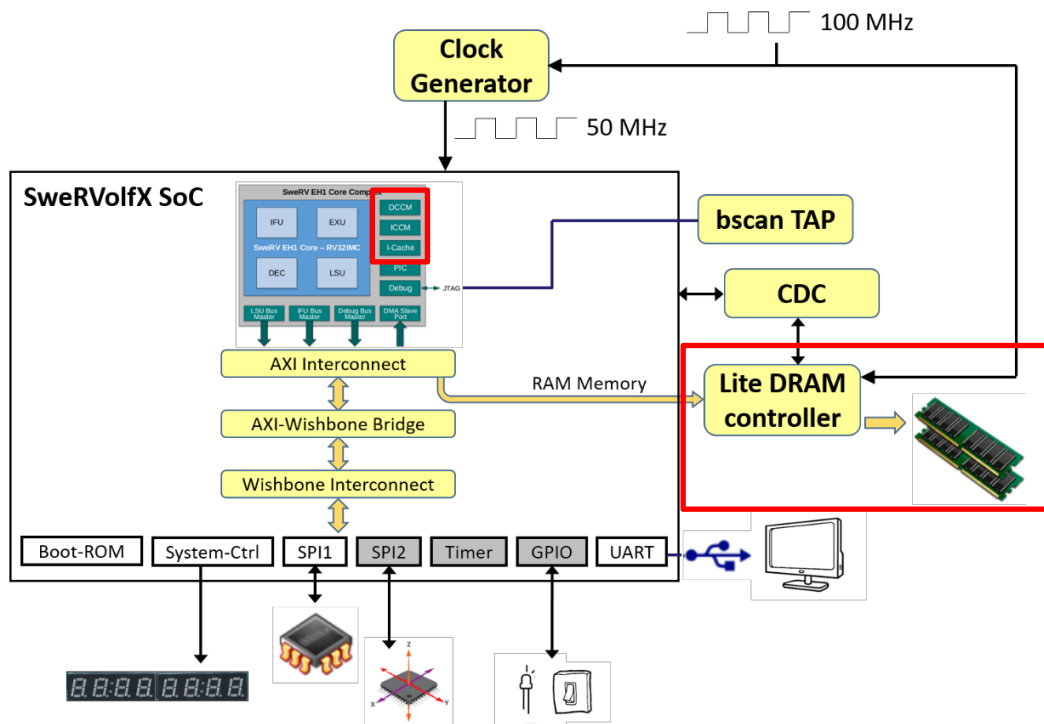


圖1. RVfpgaNexys：RVfpga記憶體系統以紅色方塊強調顯示

**附註：**開始實驗19前，建議您先閱讀S.Harris和D.Harris所著的《數位設計和電腦體系結構》（RISC-V版本，Morgan Kaufmann出版，簡稱[DDCARV]）的第7.7.4節。

本實驗將重點關注快取的操作。遺憾的是，如圖1所示，RVfpga系統並未包含資料快取（D\$）。因此，我們無法使用分析程式資料記憶體存取的典型方法來研究快取。但我們可以使用RVfpga系統中包含的I\$來展示快取的主要概念。[DDCARV]第8.3節中涉及的大多數概念同樣適用於I\$，可以作為參考。

我們首先介紹如何從DDR外部記憶體中讀取資料及向其寫入資料（第2部分），然後深入探討RVfpga系統提供I\$的操作和管理功能（第3部分）。

## 2. DDR外部記憶體資料存取

雖然我們無法在本實驗中使用D\$介紹快取，但我們會使用資料存取來說明RVfpga系統的整個記憶體系統。在實驗13和14中，我們顯示了如何使用DDR外部記憶體和DCCM進行載入和儲存。正如這些實驗中所述，每當核心需要存取資料時，都會在DC1中計算位址，然後在剩餘階段使用AXI匯流排從主記憶體讀取或向其寫入資料。在存取DDR記憶體時，管線必須暫停幾個週期，但在存取DCCM時不會暫停。

以下範例是一個先後包含載入指令和儲存指令的程式，主要用於對DDR外部記憶體進行讀/寫操作。資料夾[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW\_Instruction\_ExtMemory提供PlatformIO專案，以便可以根據需要分析、模擬和修改程式。圖2中所示的程式會遍歷包含10000個元素的陣列（未初始化，僅用於說明目的），讀取陣列中的每個元素（使用lw指令，該指令以紅色強調顯示），向元素新增常數，並將元素儲存在數組同一位置中（使用sw指令，該指令以紅色強調顯示）。

```
.data
D: .space 40000

.text
Test_Assembly:

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

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

REPEAT:
    lw t3, (t4)
    add t3, t3, t5
    sw t3, (t4)
    add t4, t4, 4
    bne t4, t6, REPEAT    # Repeat the loop
```

圖2. 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/Lab19/LW-SW\_Instruction\_ExtMemory/.pio/build/swervolf\_nexys/firmware.dis）。請注意，lw指令（0x0000eae03）和sw指令（0x01cea023）分別位於位址0x00000194和0x0000019c處。

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

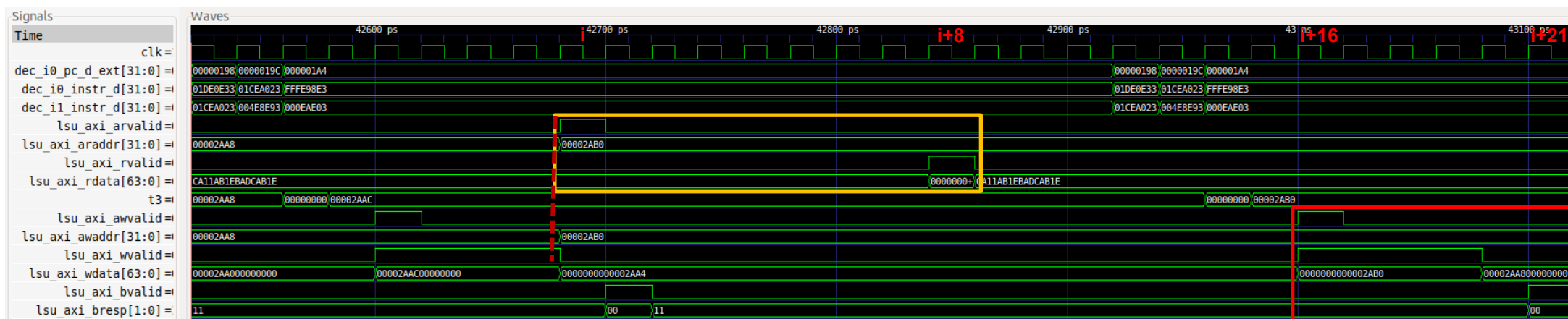



圖3. 圖2中程式任意一次迭代的模擬結果

圖3所示為圖2中迴圈的任意一次迭代的模擬結果。

**任務：**在自己的電腦上重複圖3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/Lab19/LW-SW\_Instruction\_ExtMemory*。
- 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*test\_Blocking\_Extended.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW\_Instruction\_ExtMemory*中提供）開啟與圖6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*test\_Blocking\_Extended.tcl*檔案。
- 按幾次「Zoom In」（放大）（），然後分析自42500 ps起的區域。

接下來，我們將描述如何透過AXI匯流排對DDR外部記憶體進行讀寫操作。有關匯流排操作的更多詳細資訊，請參見入門指南的第4.B.iii部分。

- 處理器會從DDR外部記憶體（黃框部分）讀取資料並將資料放入t3。讀取操作始於週期i，當匯流排完成上一次迭代的寫入操作時（即lsu\_axi\_wvalid從1變為0），便會啟動讀取操作：
  - **週期i：**透過AXI匯流排向外部記憶體傳送有效位址：
    - `lsu_axi_arvalid = 1`
    - `lsu_axi_araddr = 0x00002AB0`
  - **週期i+8：**（請注意，用於模擬的記憶體不是Nexys A7開發板上實際使用的DDR記憶體，因此模擬中觀察到的延遲與開發板上的延遲並不相同，我們將在下文對後者進行分析），透過AXI匯流排從外部記憶體接收讀取值：
    - `lsu_axi_rvalid = 1`
    - `lsu_axi_rdata = 0x0`
- 處理器在輔助ALU中進行加法運算（`add t3, t3, t5`），並將計算結果寫入暫存器檔案，如實驗15所述。（圖中並未顯示該過程，但您可自行展開模擬分析。）
  - **週期i+15：**處理器將結果寫入t3：`t3 = 0x2AB0`。
- 最後，處理器將t3的值寫入DDR外部記憶體（如紅框所示）：
  - **週期i+16：**透過AXI匯流排向外部記憶體傳送有效位址和資料：
    - `lsu_axi_awvalid = 1`

- `lsu_axi_awaddr = 0x00002AB0`
  - `lsu_axi_wvalid = 1`
  - `lsu_axi_wdata = 0x00000000000002AB0`
- **週期i+21：**（同樣的，模擬中的延遲與開發板上的延遲並不相同），外部記憶體透過AXI匯流排發出通知，表明已正確執行寫入操作：
- `lsu_axi_bvalid = 1`
  - `lsu_axi_bresp = 00`（表示一切正常）

**任務：**使用硬體計數器測量圖2中程式的週期數、指令數、載入次數和儲存次數。存取DDR外部記憶體所用的總時間是多少（包括讀取存取和寫入存取）？可以比較使用圖3中的DDR記憶體與使用DCCM時的執行情況（`[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_DCCM/`下提供另一個PlatformIO專案，其中包含用於對DCCM進行讀/寫操作的相同程式）。請注意，用於模擬的記憶體不是Nexys A7開發板上實際使用的DDR記憶體，因此模擬中觀察到的讀/寫延遲與在開發板上執行程式時的延遲並不相同。

**任務：**使用`[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_ExtMem`中提供的範例，藉助硬體計數器估算DDR外部記憶體的讀取延遲。與上一任務一樣，可以使用`[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_DCCM`中的範例，將現有程式與因記憶體存取而不存在暫停的程式進行比較。請注意，用於模擬的記憶體不是Nexys A7開發板上實際使用的DDR記憶體，因此模擬中觀察到的讀取延遲與在開發板上執行程式時的延遲並不相同。

**任務：**分析RVfpga系統中使用的記憶體控制器，本練習頗為複雜但十分有趣。請記住，構成該控制器的模組位於`[RVfpgaPath]/RVfpga/src/LiteDRAM`中，頂層模組在該資料夾內的`litedram_top.v`檔案中實作。可以先進行圖3所示的模擬，然後新增並分析來自LiteDRAM控制器的一些訊號。

### 3. 從指令快取中擷取

在本部分中，我們將分析RVfpga系統中提供的指令快取（I\$）的操作。我們首先介紹如何配置I\$（第3.A部分），然後研究如何處理快取未命中和命中（第3.B和3.C部分），最後分析SweRV EH1中使用的I\$替換策略（第3.D部分）。

#### A. 指令快取組態

RVfpga系統的I\$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。預設RVfpga系統具有以下I\$參數：

```

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

```

但部分上述參數將在檔案

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

```

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

```

因此，IS採用以下組態：

特性	值
<b>IS大小</b>	
資料陣列（不合同位檢查資訊）	16 KiB
資料的同位檢查資訊：	1 KiB（每區塊4個位元組）
標籤陣列（不合同位檢查資訊）	640個位元組
標籤的同位檢查資訊	32個位元組（每個標籤1位元）
<b>LRU狀態</b>	24個位元組（每組3位元）
有效位元	32個位元組 （每個標籤1個有效位元）
<b>相聯性</b> （不可配置）	4條通路
<b>區塊大小</b>	64個位元組
<b>區塊數</b> （大小/區塊大小 = 16Ki/64）	256個區塊
<b>每條通路的區塊數</b> （區塊數/相連性 = 256/4）	64個區塊

根據該組態，RVfpga系統中使用的IS如圖4所示。

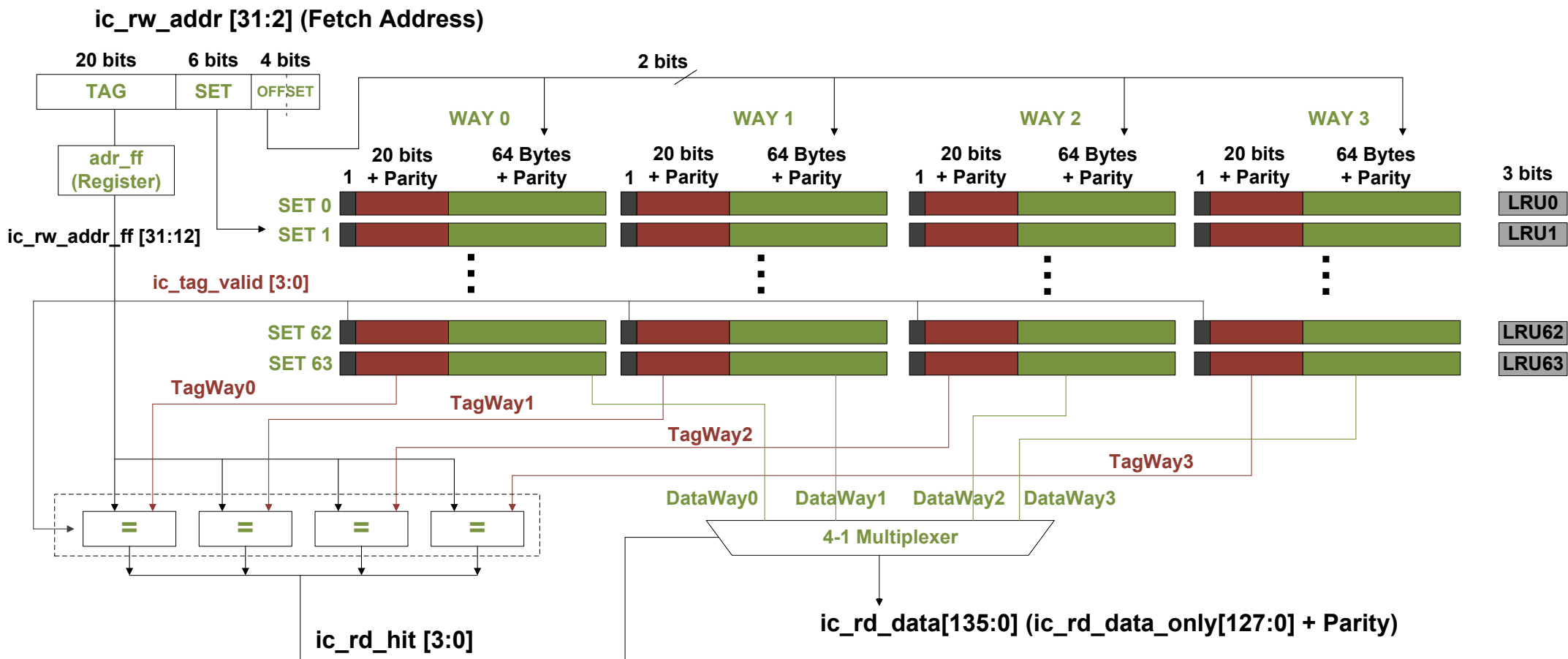


圖4. I\$內部設計。快取控制器（模組 *ifu\_mem\_ctl*）向I\$提供輸入訊號（*ic\_rw\_addr*）/接收來自I\$的輸出訊號（*ic\_rd\_data*），如實驗11的圖3中所示（我們已將該圖複製為下方的圖8）。



RVfpga系統的I\$在檔案

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/ifu/ifu\_ic\_mem.sv所包含的模組ifu\_ic\_mem中實作。該模組會對另外兩個模組進行實例化：

- **IC\_TAG**：此模組包含標籤陣列（如圖4中的紅色方塊所示）以及計算命中訊號（ic\_rd\_hit）的邏輯。模組會接收位址訊號ic\_rw\_addr，並輸出訊號ic\_rd\_hit，資料陣列將使用該訊號選擇用於執行命中的快取通路。

存取I\$時所讀取的標籤由訊號TagWay0–TagWay3提供，如圖4及相關模擬（將在稍後顯示）所示。請注意，處理器使用名為w\_tout的訊號讀取標籤。訊號TagWay0–TagWay3是訊號w\_tout的一部分，需從檔案ifu\_ic\_mem.sv的第583行至第590行中擷取。

- **IC\_DATA**：此模組為資料陣列，其中包含圖4綠色方塊中的部分，以及用於在發生命中的通路中選擇資料的4:1多路開關。每條通路在實體層面分為4個儲存區（圖中未顯示）。此模組從IC\_TAG模組接收擷取位址（ic\_rw\_addr）和命中訊號（ic\_rd\_hit）。模組會基於擷取位址的6位元SET欄位以及OFFSET欄位中的2位元，在訊號ic\_rd\_data中選擇必須傳送至SweRV EH1處理器的128位元指令束以及一些同位檢查位元，然後輸出這些資料。請注意，訊號ic\_rd\_data\_only與去掉同位檢查資訊的訊號ic\_rd\_data相同，因此我們將在下面的模擬中使用該訊號。

從I\$中讀取的資料位於訊號DataWay0–DataWay3中。請注意，這些訊號只是圖中和模擬中使用的訊號，處理器實際使用的訊號為wb\_dout\_way\_with\_premux，DataWay0–DataWay3是該訊號的一部分，需從檔案ifu\_ic\_mem.sv的第313行至第320行中取得。

**任務：**分析模組ifu\_ic\_mem，瞭解如何實作圖4中的元素。

## B. 指令快取未命中管理

在本部分中，我們將顯示處理器如何管理指令未命中。圖5中的範例是一個包含16條未壓縮的連續add指令（佔用 $4 \times 16 = 64$ 位元組，在圖中顯示為紅色）的程式，這些指令處於一個迭代0x10000次的迴圈中。16條add指令之前有幾條nop指令，其作用是強制將16條add指令對映到單個I\$區塊中。請記住，I\$區塊的大小為64個位元組。因此，必須以64位元組邊界對齊第一條add指令。資料夾 [RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory\_Example 提供 PlatformIO專案，以便可以根據需要分析、模擬和修改程式。

```

Test_Assembly:

    INSERT_NOPS_3
    INSERT_NOPS_8
    INSERT_NOPS_8

    li t6, 0x10000

REPEAT:
    add t6, t6, -1

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

    INSERT_NOPS_8
    INSERT_NOPS_8

    INSERT_NOPS_8
    INSERT_NOPS_8
    INSERT_NOPS_8
    INSERT_NOPS_8

    bne t6, zero, REPEAT    # Repeat the loop

ret

```


圖5. 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/Lab19/InstructionMemory_Example/.pio/build/swervolf_nexys/firmware.dis`）。請注意，第一條add指令（0x005282b3）位於位址0x000001c0處（以64位元組邊界對齊），第16條指令（0x011888b3）位於位址0x000001fc處（區塊中的最後一個字）。

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

圖6所示為16條add指令週圍區域的模擬結果（28900 ps至30220 ps）。中間的圖片（主圖）是我們的目標分析區域的執行結果。頂部的一張圖和底部的兩張圖是主圖特定區域的放大檢視。

**任務：**在自己的電腦上重複圖6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/Lab19/InstructionMemory\_Example*。
- 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*test1\_Miss.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory\_Example*中提供）開啟與圖6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*test1\_Miss.tcl*檔案。
- 按幾次「*Zoom In*」（放大）（），然後分析28900 ps至30220 ps範圍內的區域。

還可以進行一些更深入的分析，例如對I\$的寫入操作或初始指令的旁路。




本範例用於說明SweRV EH1如何處理I\$未命中。其中顯示了初次執行16條add指令時的擷取操作。如果這些指令不在I\$中，則必須將其從DDR外部記憶體複製到I\$中。

- 從上面的圖可以看出，每隔約29ns便會出現一次I\$未命中（ic\_act\_miss\_f2 = 1），該事件將觸發透過AXI匯流排觸發的區塊要求（ifu\_axi\_arvalid = 1）。
- 隨後，系統將透過AXI匯流排依次要求構成目標區塊的八個64位元區塊。
  - o 訊號ifu\_axi\_arvalid連續27個週期保持高電平。該訊號表示通道正在傳送有效的讀取位址和控制資訊。
  - o 在ifu\_axi\_arvalid = 1的這27個週期內，訊號ifu\_axi\_araddr將透過AXI匯流排提供8個64位元區塊的初始位址，這8個位址必須從DDR記憶體讀取：
    - ifu\_axi\_araddr = 0x000001c0
    - ifu\_axi\_araddr = 0x000001c8
    - ifu\_axi\_araddr = 0x000001d0
    - ifu\_axi\_araddr = 0x000001d8
    - ifu\_axi\_araddr = 0x000001e0
    - ifu\_axi\_araddr = 0x000001e8
    - ifu\_axi\_araddr = 0x000001f0
    - ifu\_axi\_araddr = 0x000001f8
- 中間的圖則展示了八個64位元資料區塊透過訊號ifu\_axi\_rdata中的AXI匯流排依次到達處理器。
  - o 訊號ifu\_axi\_rvalid用於指示通道正在傳送所需的讀取資料，該訊號每經7個週期便會保持一個週期的高電平。
  - o 八個64位元區塊（每個區塊包含兩條指令）均由訊號ifu\_axi\_rdata提供（圖6中未予顯示，您可在自己的電腦上重複該模擬進行驗證）：
    - ifu\_axi\_rdata = 0x00630333005282b3
    - ifu\_axi\_rdata = 0x01ce0e33007383b3
    - ifu\_axi\_rdata = 0x01ef0f3301de8eb3
    - ifu\_axi\_rdata = 0x011888b301ff8fb3
    - ifu\_axi\_rdata = 0x007383b3005282b3
    - ifu\_axi\_rdata = 0x01ce0e3300630333
    - ifu\_axi\_rdata = 0x01ff8fb301de8eb3
    - ifu\_axi\_rdata = 0x011888b301ef0f33
- 下面的兩張圖顯示，八個64位元區塊在到達快取控制器後均會立即寫入I\$。例如，前兩個64位元區塊的寫入方式如下：
  - o 訊號ic\_wr\_en變為高電平，與此同時，ic\_wr\_data = 0x00630333005282b3。因此，第一條和第二條add指令會寫入I\$。
  - o 幾個週期後，訊號ic\_wr\_en變為高電平，與此同時，ic\_wr\_data = 0x01ce0e33007383b3。因此，第三條和第四條add指令會寫入I\$。

- 最終，可以看到上述四條指令從IS控制器旁路到管線（訊號 `ifu_byp_data_first_half` 和 `ifu_byp_data_second_half`），以便在IS未命中後盡快重新啟動執行。幾個週期後，四條指令會到達解碼階段（參見右下角的訊號 `dec_i0_instr_d` 和 `dec_i1_instr_d` 放大圖）。

## C. 指令快取命中管理

在本部分中，我們仍將使用第3.B部分（圖5）的範例，但會重點分析IS命中。圖7所示為執行圖5中的程式時迴圈的第二次迭代（與圖6中的分析類似，第一次迭代會發生IS未命中，除此之外，可使用任何一次迭代）。

**任務：**在自己的電腦上重複圖7中的模擬過程。使用檔案 `test1_Hit.tcl`（在 `[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example` 中提供）。按幾次「Zoom In」（放大）（）移動至34680 ps。

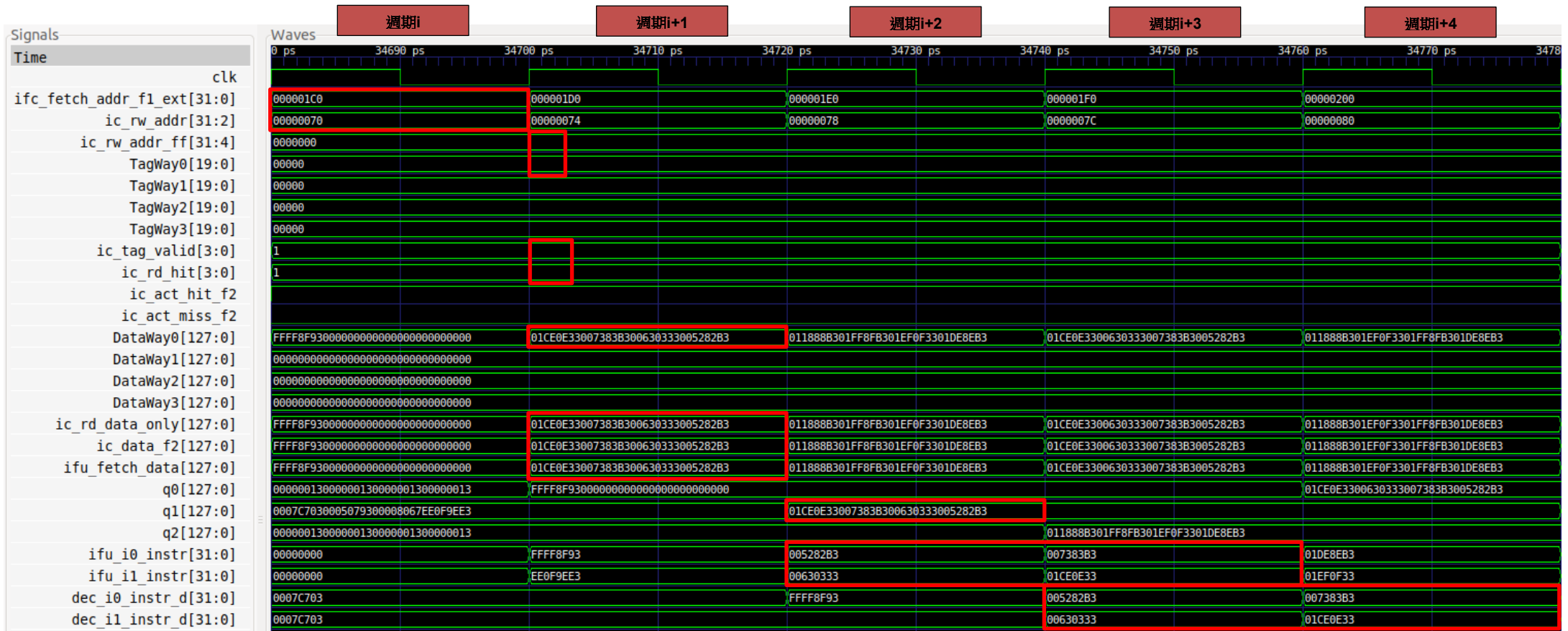


圖7. 圖5中程式的模擬波形（顯示I\$命中）



分析圖7中的模擬波形，其中包括實驗11圖3中所涉及的FC1和FC2階段的一些訊號。為方便起見，我們將該圖複製為下面的圖8。該模擬還包含本實驗的圖4中所示的部分訊號。請注意，圖4為I\$的設計圖，在下面的圖8中以黑色方塊顯示。

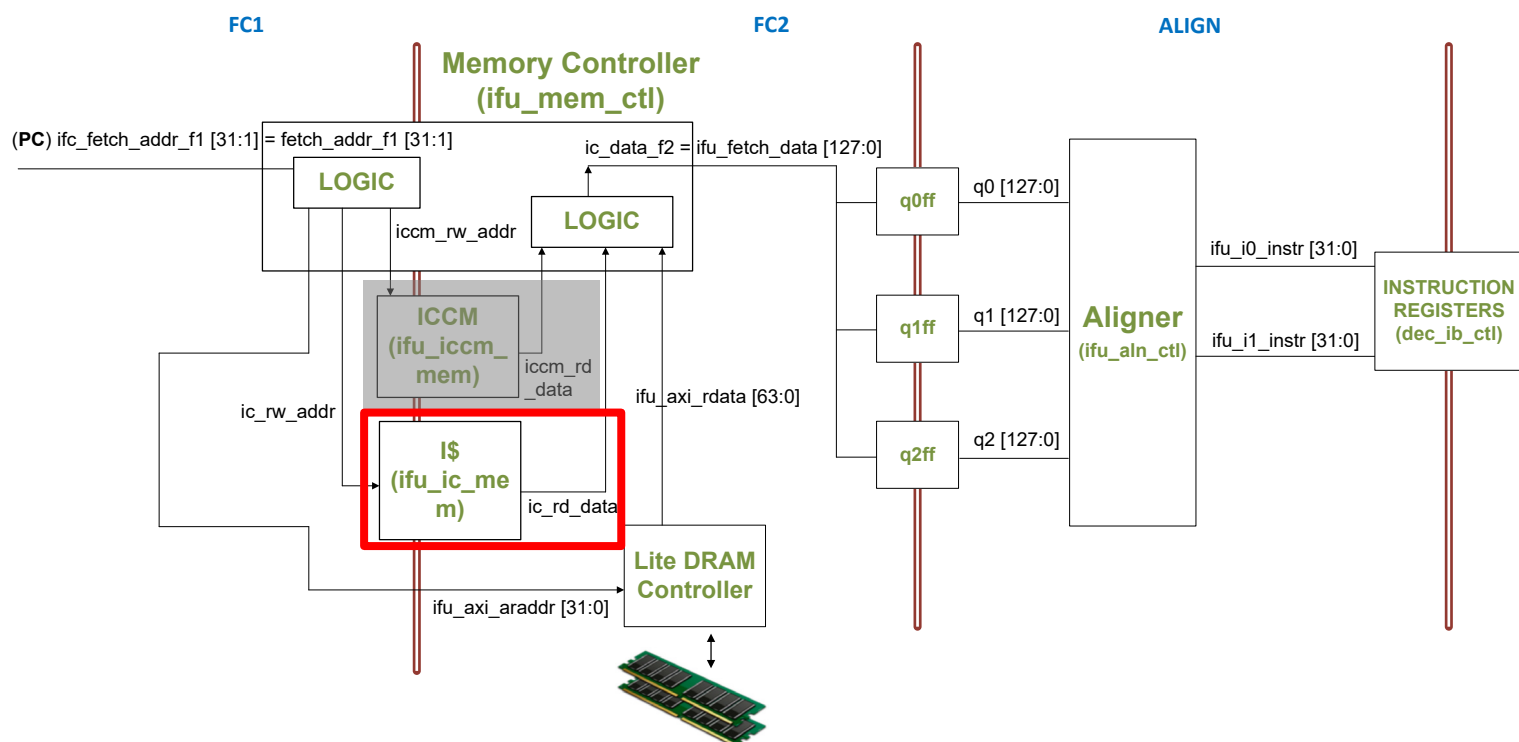


圖8. FC1、FC2和對齊階段

如圖7所示，I\$命中情況如下：

- **週期i**：圖5所示程式中第一條add指令（add t0,t0,t0）的位址由訊號 ifc\_fetch\_addr\_f1\_ext (ifc\_fetch\_addr\_f1\_ext = 0x000001c0) 提供。該訊號將傳遞到I\$，但需要去掉兩個最低有效位元，因為指令以4個位元組（32位元）邊界對齊。因此，ic\_rw\_addr = 0x0000070。

使用擷取位址的子集存取標籤陣列和資料陣列，如圖4所示。存取結果將在下一週期提供。

- **週期i+1**：四個標籤（每個快取通路一個）位於訊號TagWay0-TagWay3中。這些標籤將與adr\_ff中暫存的擷取位址（輸出訊號ic\_rw\_addr\_ff）的TAG欄位進行比較。在本例中，所有標籤均與TAG欄位相同，但只有一條通路（通路0）有效（ic\_tag\_valid = 0001），因此將在通路0中發出命中訊息：ic\_rd\_hit = 0001。

四個128位元指令令同樣位於訊號DataWay0-DataWay3中。圖4所示的4:1多路開關用於選擇通路0（即DataWay0）提供的資料。因此：

ic\_rd\_data\_only = 0x01ce0e33007383b300630333005282b3



請注意，圖8中顯示的訊號為ic\_rd\_data，該訊號與加入同位檢查資訊的ic\_rd\_data\_only訊號相同。

上述128位元將傳播到對齊階段，如圖8所示。

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

請注意，這128位元所對應的是前四條add指令。

- **週期i+2**：在對齊階段，從緩衝區q1中擷取第一條和第二條add指令：
  - o ifu\_i0\_instr = 0x005282b3
  - o ifu\_i1\_instr = 0x00630333
- **週期i+3**：在對齊階段擷取第三條和第四條add指令，同時對第一條和第二條add指令進行解碼：
  - o ifu\_i0\_instr = 0x007383b3
  - o ifu\_i1\_instr = 0x01ce0e33
  - o dec\_i0\_instr\_d = 0x005282b3
  - o dec\_i1\_instr\_d = 0x00630333
- **週期i+4**：最後，對第三條和第四條add指令進行解碼：
  - o dec\_i0\_instr\_d = 0x007383b3
  - o dec\_i1\_instr\_d = 0x01ce0e33

## D. 指令快取替換策略

本部分介紹RVfpga系統的快取替換策略。正如Harris & Harris在[DDCARV]第8.3.3節中所述，在組相聯快取中，如果快取組已滿，則必須選擇一個區塊將其逐出。根據時間局部性原則，最好的選擇是逐出近期使用最少的區塊，因為短期內再次使用該區塊的可能性最低。因此，大多陣列相連快取採用最近最少使用（Least Recently Used, LRU）的替換策略。然而，追蹤最近最少使用的通路較為複雜，因此一般使用簡化的LRU策略（通常稱為偽LRU），該策略已足以滿足實際需求。具體來說，SweRV EH1使用名為二進位樹偽LRU的簡化策略。

**附註：**如果您尚未閱讀[DDCARV]第8.3.3節，請先閱讀該節。另外，建議您閱讀Gille Damien的碩士論文《有關嵌入式系統中不同快取行替換演算法的研究》（2007年3月8日）的第4部分，連結如下：<https://people.kth.se/~ingo/MasterThesis/ThesisDamienGille2007.pdf>。為方便起見，該文件在下文中簡稱為[GiDa]。

### i. 二叉樹偽LRU策略在SweRV EH1中的實作

如[GiDa]中所述，二叉樹LRU策略是LRU策略的簡化版本，要實現該策略，N路相連快取中的每組需要N-1個位元（稱為LRU狀態）。因此，對於使用4路指令快取的SweRV EH1，每組需要3個位元，以追蹤不同通路的存取歷史。

如第3.B部分所述，發生I\$未命中時，必須透過DDR外部記憶體要求該區塊。使用DDR外部記憶體提供快取區塊時，必須將該區塊寫入I\$。擷取位址的SET欄位決定了必須向其中寫入新區塊的I\$組（參見圖4）。可能發生兩種情況：

- 快取組未滿，意味著有一個或多個區塊無效。在這種情況下，新區塊將寫入包含無效區塊的最低通路。
- 快取組已滿，意味著四個區塊均有效。在我們的處理器中，二叉樹LRU替換策略會決定必須逐出哪個區塊。該策略根據組的3位元LRU狀態確定替換的通路，如下表所示（x表示無需考慮該位元）：

LRU狀態	替換的通路
x00	通路0
x10	通路1
0x1	通路2
1x1	通路3

如前文所述，從模組ifu\_mem\_ctl中擷取的以下Verilog程式碼片段（圖9）可實作選擇儲存新I\$區塊所必需的通路的邏輯。

```

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

```

圖9. 用於選擇必須替換的通路的Verilog程式碼

圖9所示的Verilog程式碼片段使用以下訊號：

- **replace\_way\_mb\_any**（4位元）：保存獨熱編碼，其中與必須替換的通路相對應的位元為1。
- **way\_status\_mb\_ff**（3位元）：保存新區塊所在組的LRU狀態。
- **tagv\_mb\_ff**（4位元）：保存新區塊所在組的有效位元；有效通路所對應的有效位元為1，無效通路所對應的有效位元為0。

**任務：**分析圖9中的Verilog程式碼，並基於上述說明解釋程式碼如何運作。

當I\$中發生命中或未命中時，必須根據下表更新組的LRU狀態（其中「-」表示位元保持不變）：

寫入的通路	下一LRU狀態
通路0	-11
通路1	-01
通路2	1-0
通路3	0-0

透過分析該表可以看出，如[GiDa]所述，在發生命中或未命中時，指向命中/插入行的路徑上的位元將會反轉，將樹的相反部分指定為偽LRU。其原理是透過反轉指向最後存取資料的節點，確保最後存取的資料不會被逐出。

從模組ifu\_mem\_ctl擷取的以下Verilog程式碼片段（圖10）可實作上述更新LRU狀態的邏輯。

```

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

```

圖10. 用於更新LRU狀態的Verilog程式碼片段

圖10所示的Verilog程式碼片段使用以下訊號：

- **ic\_rd\_hit**（4位元）：保存發生命中的通路。
- **way\_status**和**way\_status\_mb\_ff**（各3位元）：保存發生命中或替換的組的上一LRU狀態。
- **ifu\_wr\_en\_new\_q**（1位元）：如果發生替換，則訊號值為1。
- **way\_status\_new**（3位元）：保存剛剛發生命中或未命中的組的新LRU狀態。
- **replace\_way\_mb\_any**（4位元）：保存獨熱編碼，其中與必須替換的通路相對應的位元為1。圖9下方也提供了關於該訊號的說明。

**任務：**分析圖10中的Verilog程式碼，並基於上述說明解釋程式碼如何運作。

## ii. 二叉樹LRU策略的工作展示範例

為了分析SweRV EH1的替換策略，我們在資料夾

[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory\_LRU\_Example中提供了新的範例。該範例（圖11）在一個無限迴圈中存取五個不同的IS區塊，並且五個區塊均對映到同一IS組：

SET = 8。為此，我們建立了一個包含五條j（跳轉）指令的無限迴圈，其中每對j指令由

1023個nop分隔。請注意，j指令與nop共佔用4KiB空間（1024 \* 4個位元組/指令），等同於IS中每條通路的大小（參見第3.A部分和圖4）。

```
Set8_Block1:    j Set8_Block2          # This j instruction is at address 0x00000200
                 INSERT_NOPs_1023

Set8_Block2:    j Set8_Block3          # This j instruction is at address 0x00001200
                 INSERT_NOPs_1023

Set8_Block3:    j Set8_Block4          # This j instruction is at address 0x00002200
                 INSERT_NOPs_1023

Set8_Block4:    j Set8_Block5          # This j instruction is at address 0x00003200
                 INSERT_NOPs_1023

Set8_Block5:    j Set8_Block1          # This j instruction is at address 0x00004200
```

**圖11. 顯示對映到相同組的j指令的範例程式**

在PlatformIO中開啟、編譯專案，然後開啟反組譯檔案（位於 `[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_LRU_Example/.pio/build/swervolf_nextys/firmware.dis`）。注意以下事項：

- 第一條j指令（j Set8\_Block2）位於位址0x00000200處。根據圖4，存取IS時的位址劃分如下：
  - IS位址（二進位）= 000000000000000000000000001000000000
  - TAG = 0x0
  - SET = 0x8
  - OFFSET = 0x0
- 第二條j指令（j Set8\_Block3）位於位址0x00001200處。根據圖4，存取IS時的位址劃分如下：
  - IS位址（二進位）= 000000000000000000000000100100000000
  - TAG = 0x1
  - SET = 0x8
  - OFFSET = 0x0
- 第三條j指令（j Set8\_Block4）位於位址0x00002200處。根據圖4，存取IS時的位址劃分如下：
  - IS位址（二進位）= 000000000000000000000000100010000000
  - TAG = 0x2
  - SET = 0x8
  - OFFSET = 0x0
- 第四條j指令（j Set8\_Block5）位於位址0x00003200處。根據圖4，存取IS時的位址劃分如下：
  - IS位址（二進位）= 000000000000000000000000110010000000
  - TAG = 0x3
  - SET = 0x8
  - OFFSET = 0x0

- 第五條j指令 (j Set8\_Block1) 位於位址0x00004200處。根據圖4，存取IS時的位址劃分如下：

IS位址 (二進位) = 00000000000000000000000010000100000000

TAG = 0x4

SET = 0x8

OFFSET = 0x0

當該程式 (圖11) 執行第一次迭代時，組8的初始狀態為空。圖12所示為執行第一次迭代後，IS中組8理論上的變化情況。隨後，我們將顯示幾項Verilator模擬，以證實這些理論說明。

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

Valid	Tag	Data	
1	00000000000000000000	j Set8_Block2   nop   ...   nop	WAY 0
0			WAY 1
0			WAY 2
0			WAY 3

LRU STATE = 011

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

1	00000000000000000000	j Set8_Block2   nop   ...   nop	WAY 0
1	00000000000000000001	j Set8_Block3   nop   ...   nop	WAY 1
0			WAY 2
0			WAY 3

LRU STATE = 001

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

1	00000000000000000000	j Set8_Block2   nop   ...   nop	WAY 0
1	00000000000000000001	j Set8_Block3   nop   ...   nop	WAY 1
1	00000000000000000010	j Set8_Block4   nop   ...   nop	WAY 2
0			WAY 3

LRU STATE = 100

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

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

LRU STATE = 000

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

1	000000000000000000100	j Set8_Block1   nop   ...   nop	WAY 0
1	000000000000000000001	j Set8_Block3   nop   ...   nop	WAY 1
1	000000000000000000010	j Set8_Block4   nop   ...   nop	WAY 2
1	000000000000000000011	j Set8_Block5   nop   ...   nop	WAY 3

LRU STATE = 011

圖12. 圖11中的迴圈執行第一次迭代時的IS組8

下面的Verilator模擬顯示了迴圈第一次迭代期間的快取訊號，模擬結果證實了圖12中所示的分析。圖13所示的Verilator模擬為程式執行第一條j指令（j Set8\_Block2）後的狀態。該指令位址（0x200）同樣對映到I\$的組8。該組的初始狀態為空：tagv\_mb\_ff = 0000。因此，根據二叉樹LRU策略，必須將新區塊寫入通路0。replace\_way\_mb\_any = ic\_wr\_en = 0001。組8的LRU狀態更新如下：way\_status\_new = 011。

回想一下第3.B部分，該區塊從DDR記憶體中讀取，並以64位元區塊的形式寫入I\$。圖13描繪了將新區塊的標籤和前兩條指令寫入組8的過程：

```
ic_rw_addr_q[11:4] = 00100000 (組8)
ic_tag_wr_data[19:0] = 0x0 (最高有效位元用作糾錯位元，未包含在此處)
ic_wr_data1[31:0] = 0x0000106F (j Set8_Block2)
ic_wr_data2[31:0] = 0x00000013 (nop)
```

（ic\_wr\_data1和ic\_wr\_data2是為了清楚起見而建立的訊號，但I\$中實際使用的訊號為ic\_wr\_data[67:0]，其中包括兩條指令和一些同位檢查資訊）。

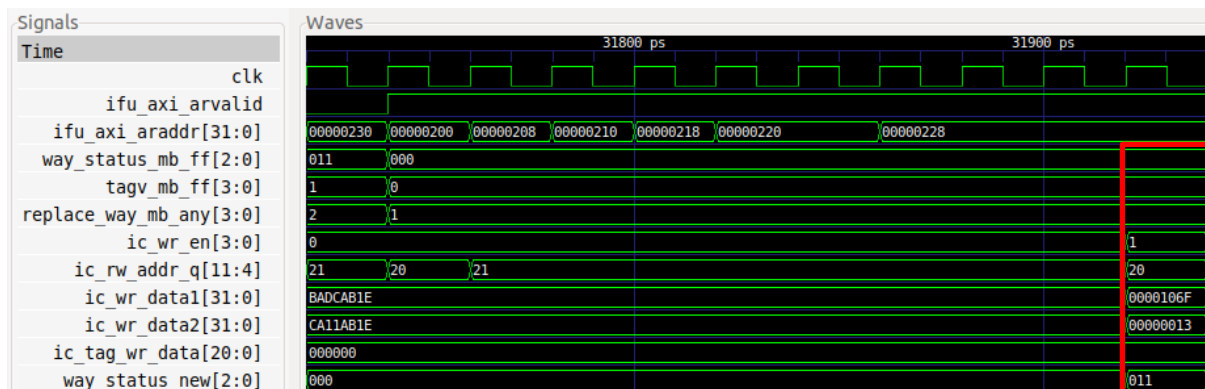


圖13. 執行第一條j指令後組8的LRU狀態

圖14所示的Verilator模擬為程式執行第二條j指令（j Set8\_Block3）後的狀態。該指令位址（0x1200）同樣對映到I\$的組8。在該組中，僅通路0有效：tagv\_mb\_ff = 0001。因此，根據二叉樹LRU策略，必須將新區塊寫入通路1：replace\_way\_mb\_any = ic\_wr\_en = 0010。組8的LRU狀態更新如下：way\_status\_new = 001。

與上文類似，圖14描繪了將新區塊的前兩條指令寫入組8的過程：

```
ic_rw_addr_q[11:4] = 00100000 (組8)
ic_tag_wr_data[19:0] = 0x1 (最高有效位元用作糾錯位元，未包含在此處)
ic_wr_data1[31:0] = 0x0000106F (j Set8_Block3)
ic_wr_data2[31:0] = 0x00000013 (nop)
```



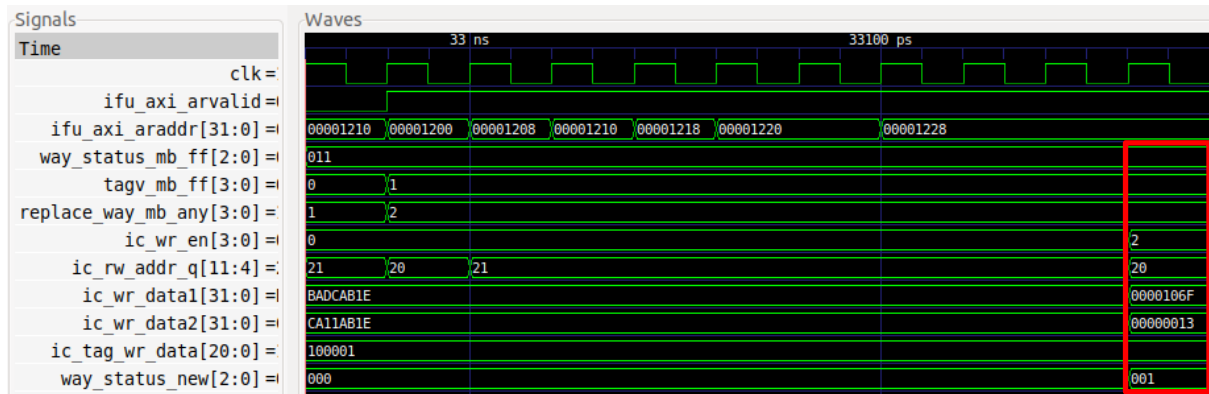


圖14. 執行第二條j指令後組8的LRU狀態

圖15所示的Verilator模擬為程式執行第五條j指令（j Set8\_Block1）後的狀態。該指令位址（0x4200）同樣對映到I\$的組8。但是，與先前的情況相反，此時組已滿：tagv\_mb\_ff = 1111。因此，根據二叉樹LRU策略，必須將新區塊寫入通路1：replace\_way\_mb\_any = 1，ic\_wr\_en = 0001。組8的LRU狀態更新如下：way\_status\_new = 011。

與上文類似，圖15描繪了將新區塊的前兩條指令寫入組8的過程：

```
ic_rw_addr_q[11:4] = 00100000 (組8)
ic_tag_wr_data[19:0] = 0x4 (最高有效位元用作糾錯位元，未包含在此處)
ic_wr_data1[31:0] = 0x800fc06f (j Set8_Block1)
ic_wr_data2[31:0] = 0x00008067 (ret)
```

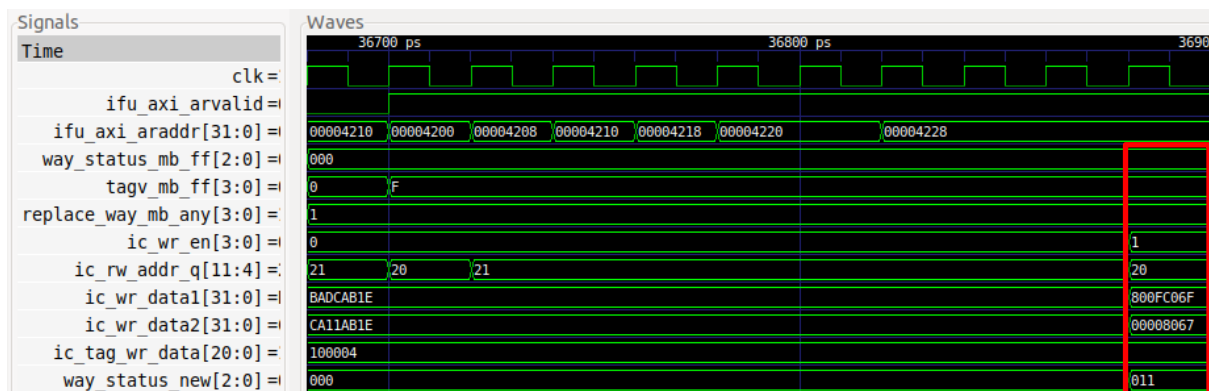


圖15. 執行第五條j指令後組8的LRU狀態

**任務：**在自己的電腦上重複圖13-圖15中的模擬過程。



## 4. 練習

- 1) 將圖11所示的迴圈轉換為0x10000次迭代的迴圈，但為j指令保持原有的位址。測量週期數以及I\$命中和未命中數。然後刪除其中一條j指令，再次測量上述指標。比較測量結果，並做出解釋。
- 2) 使用圖5中的程式，從I\$替換策略的角度分析I\$命中。
- 3) 展開圖6，詳細分析每個64位元區塊如何寫入I\$。
- 4) 透過模擬器和開發板分析其他I\$組態，例如具有不同區塊大小的I\$。請注意，無法修改通路的數量。
- 5) 分析用於檢查資料陣列和標籤陣列同位檢查資訊正確性的邏輯。