



Imagination大学计划

RVfpga实验19

指令高速缓存

1. 简介

在本实验及下一实验中，我们将重点讨论RVfpga系统的存储器系统。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图1），RVfpga系统包含一个外部DDR主存储器、一个指令高速缓存（I\$）和两个分别用于存储数据（DCCM）和指令（ICCM）的便笺式存储器（也称为紧密耦合存储器）。

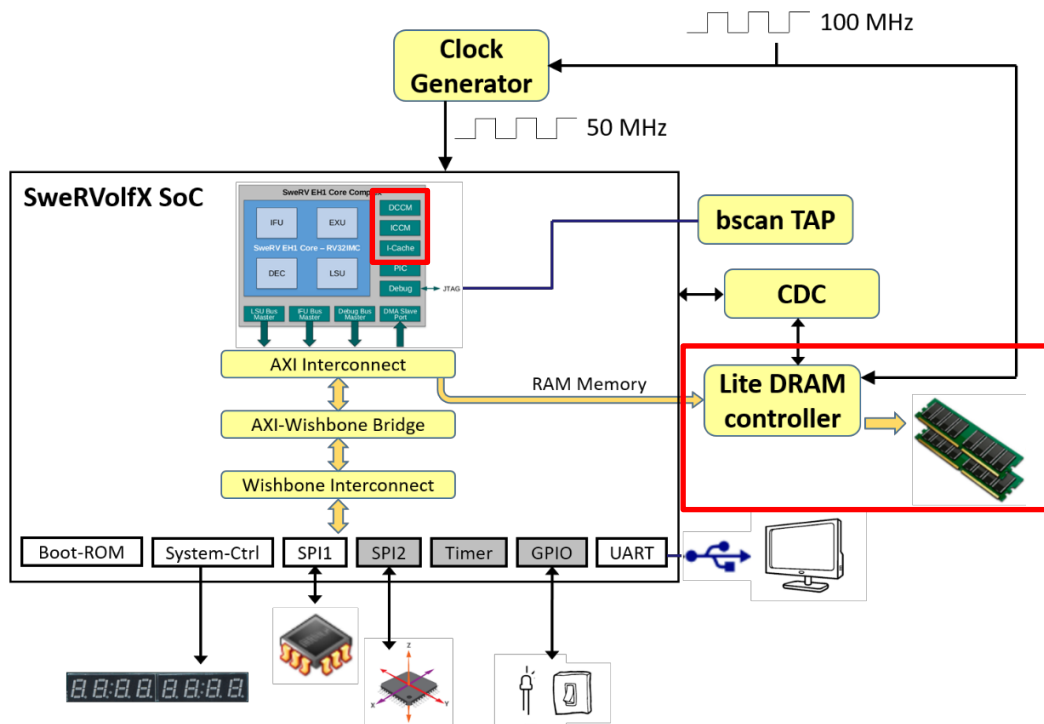


图1. RVfpgaNexys: RVfpga存储器系统以红色框突出显示

注：开始实验19前，建议您先阅读S.Harris和D.Harris所著的《数字设计和计算机体系结构》（RISC-V版本，Morgan Kaufmann出版，简称[DDCARV]）的第8.1至8.3节。

本实验将重点关注高速缓存的操作。遗憾的是，如图1所示，RVfpga系统并未包含数据高速缓存（D\$）。因此，我们无法使用分析程序数据存储器访问的典型方法来研究高速缓存。但我们可以使用RVfpga系统中包含的I\$来演示高速缓存的主要概念。[DDCARV]第8.3节中涉及的大多数概念同样适用于I\$，可以作为参考。

我们首先介绍如何从DDR外部存储器中读取数据及向其写入数据（第2部分），然后深入探讨RVfpga系统提供I\$的操作和管理功能（第3部分）。

2. DDR外部存储器数据访问

虽然我们无法在本实验中使用D\$介绍高速缓存，但我们会使用数据访问来说明RVfpga系统的整个存储器系统。在实验13和14中，我们展示了如何使用DDR外部存储器和DCCM进行装载和存储。正如这些实验中所述，每当内核需要访问数据时，都会在DC1中计算地址，然后在剩余阶段使用AXI总线从主存储器读取或向其写入数据。在访问DDR内存时，流水线必须暂停几个周期，但在访问DCCM时不会暂停。

以下示例是一个先后包含装载指令和存储指令的程序，主要用于对DDR外部存储器进行读/写操作。文件夹 *[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory* 提供 PlatformIO 项目，以便可以根据需要分析、仿真和修改程序。图2中所示的程序会遍历包含 10000 个元素的数组（未初始化，仅用于说明目的），读取数组中的每个元素（使用 `lw` 指令，该指令以红色突出显示），向元素添加常量，并将元素存储在数组同一位置中（使用 `sw` 指令，该指令以红色突出显示）。

```
.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
```

图2. 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/Lab19/LW-SW_Instruction_ExtMemory/.pio/build/swervolf_nexys/firmware.dis*）。请注意，`lw` 指令（0x000eae03）和 `sw` 指令（0x01cea023）分别位于地址 0x00000194 和 0x0000019c 处。

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

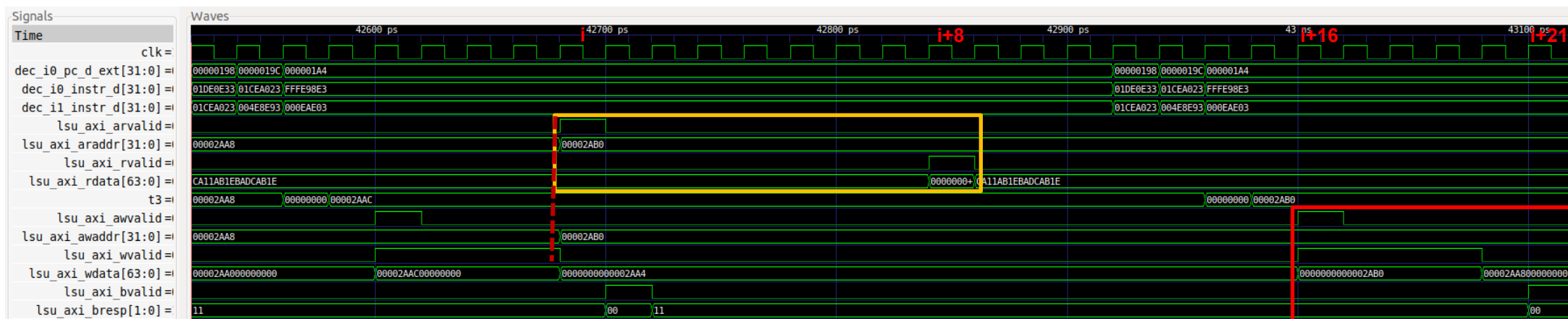



图3. 图2中程序任意一次迭代的仿真结果

图3所示为图2中循环的任意一次迭代的仿真结果。

任务：在自己的计算机上重复图3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/Lab19/LW-SW_Instruction_ExtMemory*。
- 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*test_Blocking_Extended.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_ExtMemory*中提供）打开与图6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*test_Blocking_Extended.tcl*文件。
- 单击几次“Zoom In”（放大）（），然后分析自42500 ps起的区域。

接下来，我们将描述如何通过AXI总线对DDR外部存储器进行读写操作。有关总线操作的更多详细信息，请参见入门指南的第4.B.iii部分。

- 处理器会从DDR外部存储器（黄框部分）读取数据并将数据放入t3。读操作始于周期i，当总线完成上一次迭代的写操作时（即lsu_axi_wvalid从1变为0），便会启动读操作：
 - **周期i：**通过AXI总线向外部存储器发送有效地址：
 - `lsu_axi_arvalid = 1`
 - `lsu_axi_araddr = 0x00002AB0`
 - **周期i+8：**（请注意，用于仿真的存储器不是Nexys A7开发板上实际使用的DDR存储器，因此仿真中观察到的延时与开发板上的延时并不相同，我们将在下文中对后者进行分析），通过AXI总线从外部存储器接收读取值：
 - `lsu_axi_rvalid = 1`
 - `lsu_axi_rdata = 0x0`
- 处理器在辅助ALU中进行加法运算（`add t3, t3, t5`），并将计算结果写入寄存器文件，如实验15所述。（图中并未显示该过程，但您可自行开展仿真分析。）
 - **周期i+15：**处理器将结果写入t3：`t3 = 0x2AB0`。
- 最后，处理器将t3的值写入DDR外部存储器（如红框所示）：
 - **周期i+16：**通过AXI总线向外部存储器发送有效地址和数据：
 - `lsu_axi_awvalid = 1`

- `lsu_axi_awaddr = 0x00002AB0`
- `lsu_axi_wvalid = 1`
- `lsu_axi_wdata = 0x00000000000002AB0`

- **周期i+21:** (同样, 仿真中的延时与开发板上的延时并不相同), 外部存储器通过AXI总线发出通知, 表明已正确执行写操作:

- `lsu_axi_bvalid = 1`
- `lsu_axi_bresp = 00` (表示一切正常)

任务: 使用硬件计数器测量图2中程序的周期数、指令数、装载次数和存储次数。访问DDR外部存储器所用的总时间是多少(包括读访问和写访问)? 可以比较使用图3中的DDR存储器与使用DCCM时的执行情况([RVfpgaPath]/RVfpga/Labs/Lab19/LW-SW_Instruction_DCCM/下提供另一个PlatformIO项目, 其中包含用于对DCCM进行读/写操作的相同程序)。请注意, 用于仿真的存储器不是Nexys A7开发板上实际使用的DDR存储器, 因此仿真中观察到的读/写延时与在开发板上执行程序时的延时并不相同。

任务: 使用[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_ExtMem中提供的示例, 借助硬件计数器估算DDR外部存储器的读延时。与上一任务一样, 可以使用[RVfpgaPath]/RVfpga/Labs/Lab19/LW_Instruction_DCCM中的示例, 将现有程序与因存储器访问而不存在暂停的程序进行比较。请注意, 用于仿真的存储器不是Nexys A7开发板上实际使用的DDR存储器, 因此仿真中观察到的读延时与在开发板上执行程序时的延时并不相同。

任务: 分析RVfpga系统中使用的存储器控制器, 本练习颇为复杂但十分有趣。请记住, 构成该控制器的模块位于[RVfpgaPath]/RVfpga/src/LiteDRAM中, 顶层模块在该文件夹内的litedram_top.v文件中实现。可以先进行图3所示的仿真, 然后添加并分析来自LiteDRAM控制器的一些信号。

3. 从指令高速缓存中取指

在本部分中, 我们将分析RVfpga系统中提供的指令高速缓存(I\$)的操作。我们首先介绍如何配置I\$(第3.A部分), 然后研究如何处理高速缓存未命中和命中(第3.B和3.C部分), 最后分析SweRV EH1中使用的I\$替换策略(第3.D部分)。

A. 指令高速缓存配置

RVfpga系统的I\$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。默认RVfpga系统具有以下I\$参数:

```

`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

```

但部分上述参数将在文件

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

```

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，I\$采用以下配置：

特性	值
I\$大小	
数据数组（不含奇偶校验信息）	16 KiB
数据的奇偶校验信息：	1 KiB（每块4字节）
标记数组（不含奇偶校验信息）	640字节
标记的奇偶校验信息	32字节（每个标记1位）
LRU状态	24字节（每组3位）
有效位	32字节 （每个标记1个有效位）
相联性 （不可配置）	4条通路
块大小	64字节
块数 （大小/块大小 = 16Ki/64）	256个块
每条通路的块数 （块数/相联性 = 256/4）	64个块

根据该配置，RVfpga系统中使用的I\$如图4所示。

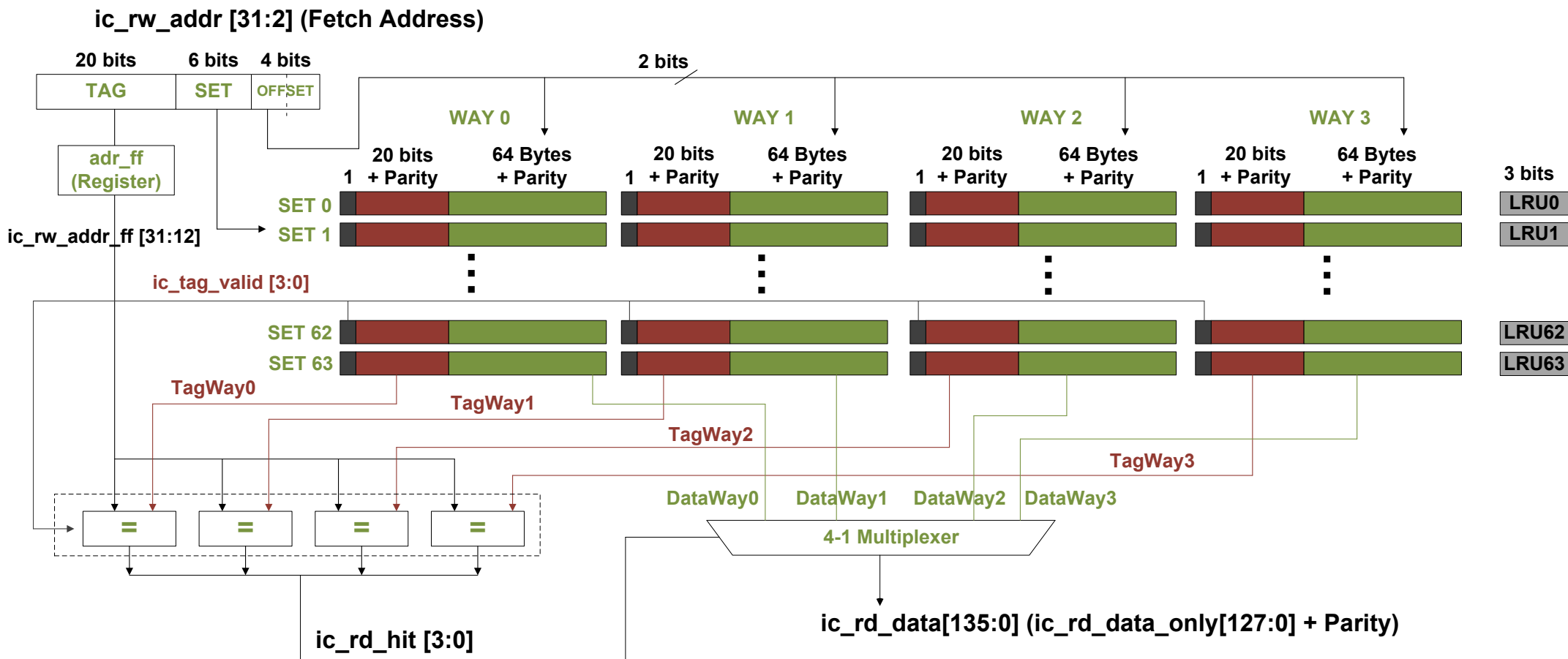


图4. I\$内部设计。高速缓存控制器（模块*ifu_mem_ctl*）向I\$提供输入信号（*ic_rw_addr*）/接收来自I\$的输出信号（*ic_rd_data*），如实验11的图3中所示（我们已将该图复制为下方的图8）。

RVfpga系统的I\$在文件

[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/ifu/ifu_ic_mem.sv所包含的模块ifu_ic_mem中实现。该模块会对另外两个模块进行实例化：

- **IC_TAG:** 此模块包含标记数组（如图4中的红框所示）以及计算命中信号（ic_rd_hit）的逻辑。模块会接收地址信号ic_rw_addr，并输出信号ic_rd_hit，数据数组将使用该信号选择用于执行命中的高速缓存通路。

访问I\$时所读取的标记由信号TagWay0–TagWay3提供，如图4及相关仿真（将在稍后展示）所示。请注意，处理器使用名为w_tout的信号读取标记。信号TagWay0–TagWay3是信号w_tout的一部分，需从文件ifu_ic_mem.sv的第583行至第590行中提取。

- **IC_DATA:** 此模块为数据数组，其中包含图4绿框中的部分，以及用于在发命中通路中选择数据的4:1多路开关。每条通路在物理层面分为4个库（图中未显示）。此模块从IC_TAG模块接收取指地址（ic_rw_addr）和命中信号（ic_rd_hit）。模块会基于取指地址的6位SET字段以及OFFSET字段中的2位，在信号ic_rd_data中选择必须发送至SweRV EH1处理器的128位指令束以及一些奇偶校验位，然后输出这些数据。请注意，信号ic_rd_data_only与去掉奇偶校验信息的信号ic_rd_data相同，因此我们将在下面的仿真中使用该信号。

从I\$中读取的数据位于信号DataWay0–DataWay3中。请注意，这些信号只是图中和仿真中使用的信号，处理器实际使用的信号为wb_dout_way_with_premux，DataWay0–DataWay3是该信号的一部分，需从文件ifu_ic_mem.sv的第313行至第320行中获取。

任务： 分析模块ifu_ic_mem，了解如何实现图4中的元素。

B. 指令高速缓存未命中管理

在本部分中，我们将展示处理器如何管理指令未命中。图5中的示例是一个包含16条未压缩的连续add指令（占用 $4 \times 16 = 64$ 字节，在图中显示为红色）的程序，这些指令处于一个迭代0x10000次的循环中。16条add指令之前有几条nop指令，其作用是强制将16条add指令映射到单个I\$块中。请记住，I\$块的大小为64字节。因此，必须以64字节边界对齐第一条add指令。文件夹[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example提供PlatformIO项目，以便可以根据需要分析、仿真和修改程序。

```

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

```


图5. 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/Lab19/InstructionMemory_Example/.pio/build/swervolf_nexys/firmware.dis*）。请注意，第一条add指令（0x005282b3）位于地址0x000001c0处（以64字节边界对齐），第16条指令（0x011888b3）位于地址0x000001fc处（块中的最后一个字）。

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

图6所示为16条add指令周围区域的仿真结果（28900 ps至30220 ps）。中间的图片（主图）是我们的目标分析区域的执行结果。顶部的一张图和底部的两张图是主图特定区域的放大视图。

任务： 在自己的计算机上重复图6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/Lab19/InstructionMemory_Example。
- 在文件*platformio.ini*中更新到RVfpga仿真二进制文件（*Vrvfpgasim*）的路径。
- 使用Verilator生成仿真轨迹（生成轨迹）。
- 在GTKWave上打开轨迹。
- 使用文件*test1_Miss.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example*中提供）打开与图6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*test1_Miss.tcl*文件。
- 单击几次“*Zoom In*”（放大）（），然后分析28900 ps至30220 ps范围内的区域。

还可以进行一些更深入的分析，例如对IS的写操作或初始指令的旁路。

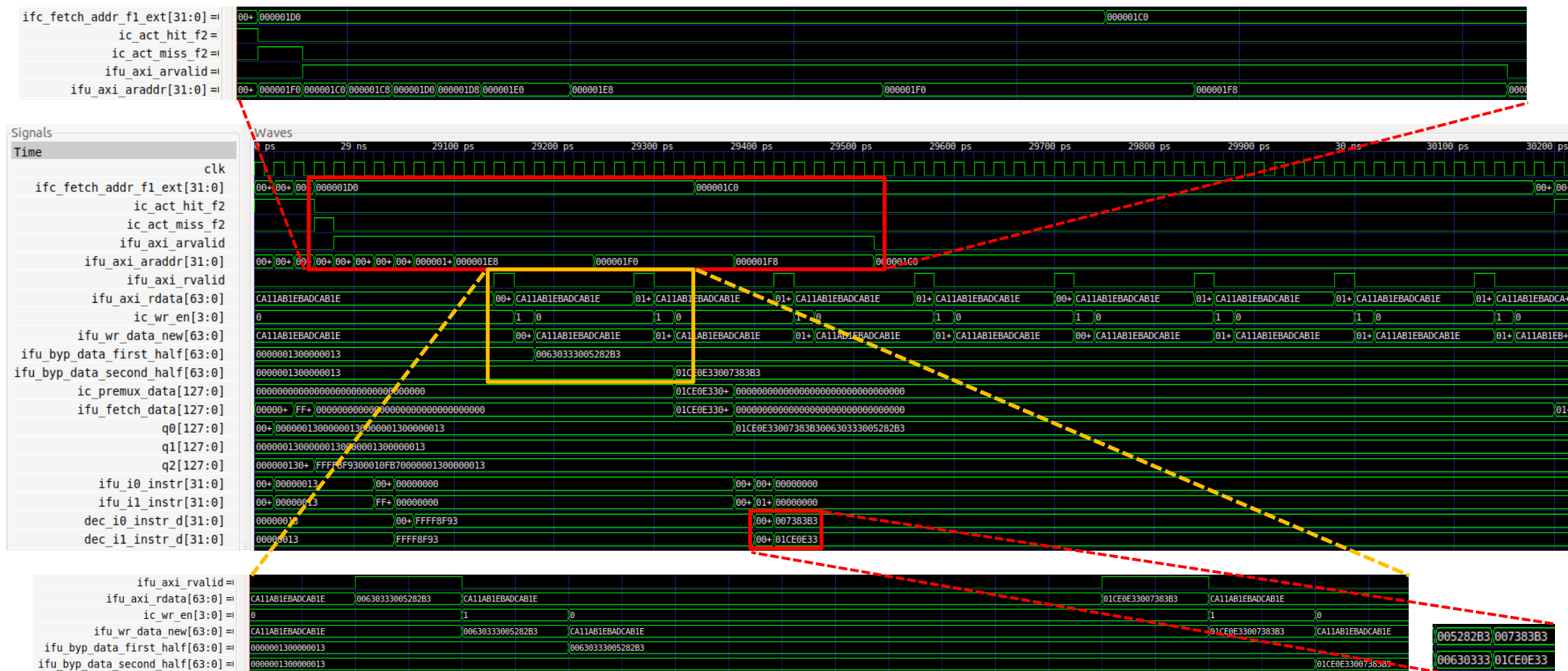


图6. 图5中程序的仿真波形（显示I\$未命中）

本示例用于说明SweRV EH1如何处理I\$未命中。其中展示了首次执行16条add指令时的取指操作。如果这些指令不在I\$中，则必须将其从DDR外部存储器复制到I\$中。

- 从上面的图可以看出，每隔约29ns便会出现一次I\$未命中（ic_act_miss_f2 = 1），该事件将触发通过AXI总线发起的块请求（ifu_axi_arvalid = 1）。
- 随后，系统将通过AXI总线依次请求构成目标块的八个64位块。
 - o 信号ifu_axi_arvalid连续27个周期保持高电平。该信号表示通道正在发送有效的读地址和控制信息。
 - o 在ifu_axi_arvalid = 1的这27个周期内，信号ifu_axi_araddr将通过AXI总线提供8个64位块的初始地址，这8个地址必须从DDR存储器读取：
 - 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
- 中间的图则展示了八个64位数据块通过信号ifu_axi_rdata中的AXI总线依次到达处理器。
 - o 信号ifu_axi_rvalid用于指示通道正在发送所需的读数据，该信号每经7个周期便会保持一个周期的高电平。
 - o 八个64位块（每个块包含两条指令）均由信号ifu_axi_rdata提供（图6中未予显示，您可在自己的计算机上重复该仿真进行验证）：
 - 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
- 下面的两幅图显示，八个64位块在到达高速缓存控制器后均会立即写入I\$。例如，前两个64位块的写入方式如下：
 - o 信号ic_wr_en变为高电平，与此同时，ic_wr_data = 0x00630333005282b3。因此，第一条和第二条add指令会写入I\$。
 - o 几个周期后，信号ic_wr_en变为高电平，与此同时，ic_wr_data = 0x01ce0e33007383b3。因此，第三条和第四条add指令会写入I\$。

- 最终，可以看到上述四条指令从I\$控制器旁路到流水线（信号 `ifu_byp_data_first_half` 和 `ifu_byp_data_second_half`），以便在I\$未命中后尽快重新启动执行。几个周期后，四条指令会到达译码阶段（参见右下角的信号 `dec_i0_instr_d` 和 `dec_i1_instr_d` 放大图）。

C. 指令高速缓存命中管理

在本部分中，我们仍将使用第3.B部分（图5）的示例，但会重点分析I\$命中。图7所示为执行图5中的程序时循环的第二次迭代（与图6中的分析类似，第一次迭代会发生I\$未命中，除此之外，可使用任何一次迭代）。

任务： 在自己的计算机上重复图7中的仿真过程。使用文件 `test1_Hit.tcl`（在 `[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_Example` 中提供）。单击几次“Zoom In”（放大）（）移动至34680 ps。

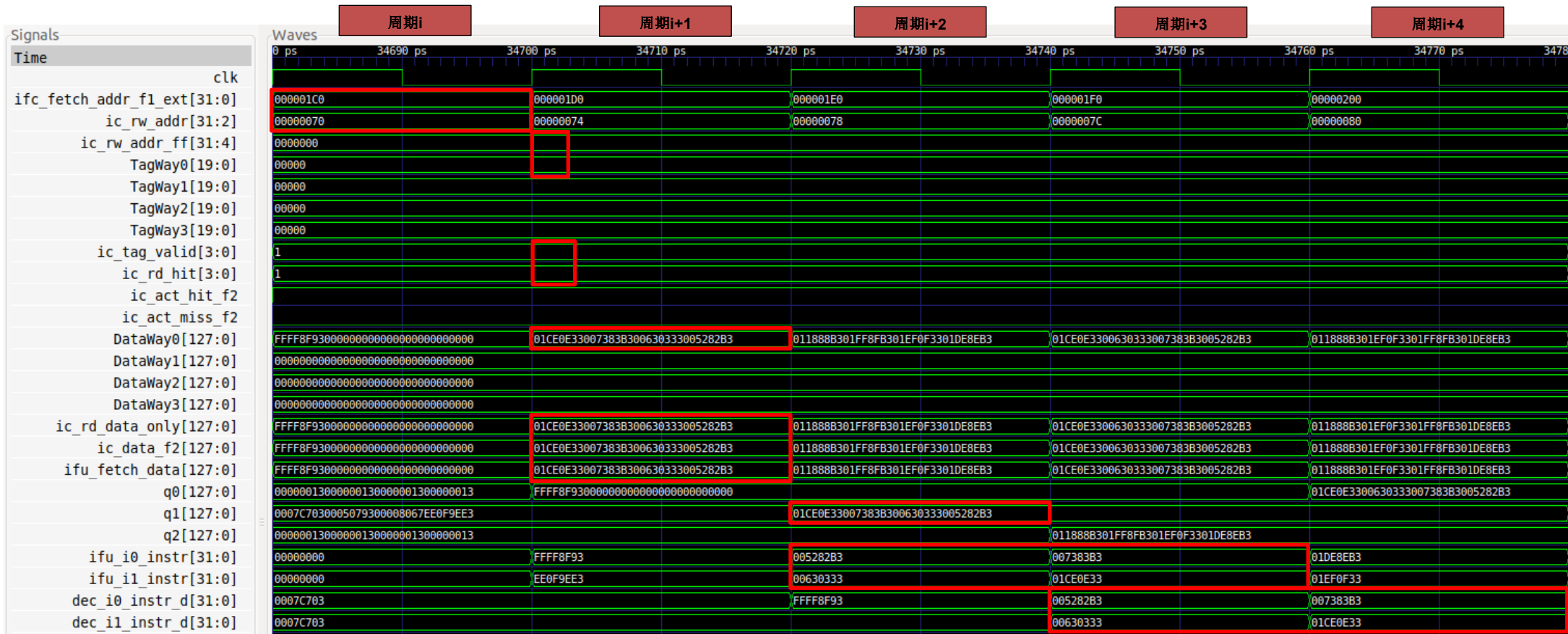


图7. 图5中程序的仿真波形（显示I\$命中）

分析图7中的仿真波形，其中包括实验11图3中所涉及的FC1和FC2阶段的一些信号。为方便起见，我们将该图复制为下面的图8。该仿真还包含本实验的图4中所示的部分信号。请注意，图4为I\$的设计图，在下面的图8中以黑框显示。

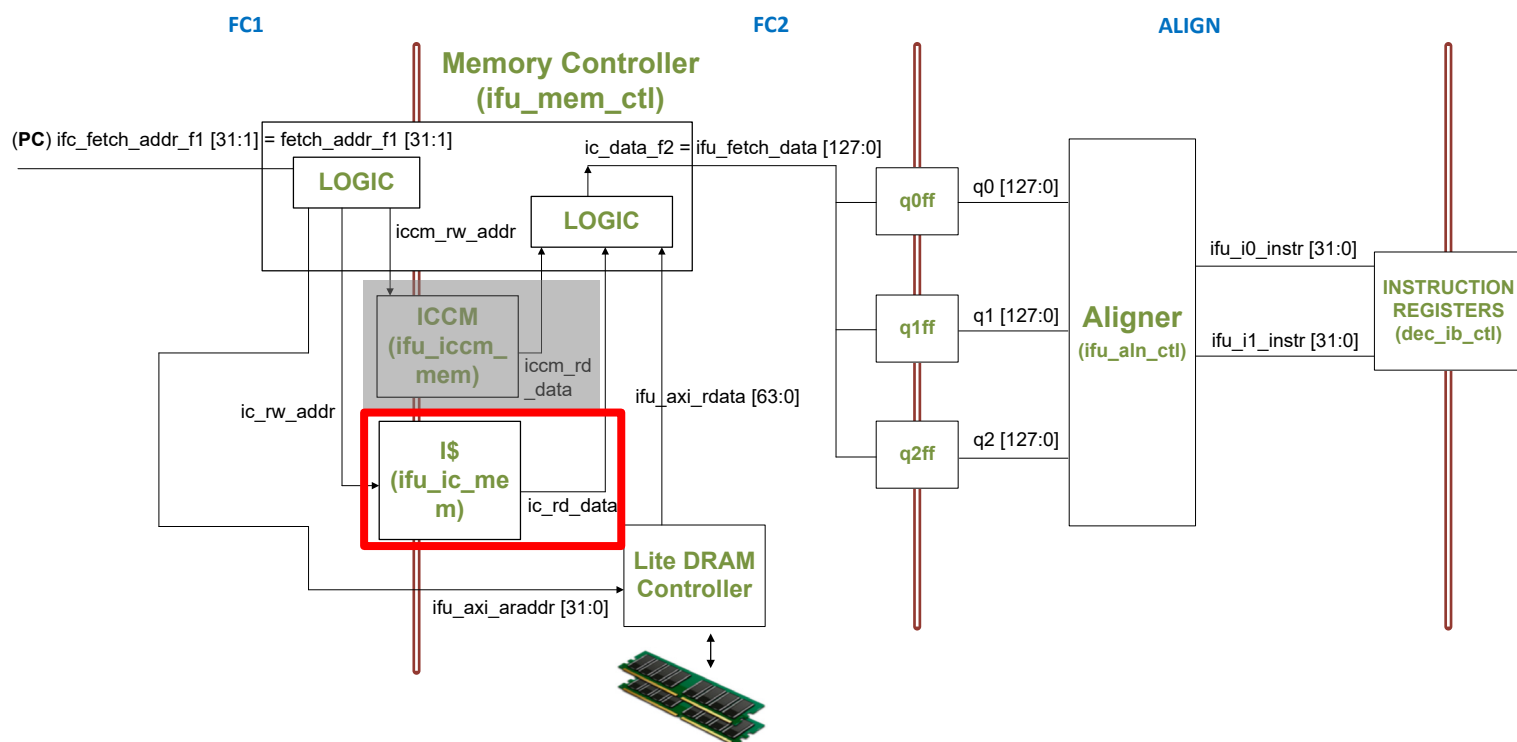


图8. FC1、FC2和对齐阶段

如图7所示，I\$命中情况如下：

- **周期i:** 图5所示程序中第一条add指令（add t0,t0,t0）的地址由信号 ifc_fetch_addr_f1_ext（ifc_fetch_addr_f1_ext = 0x000001c0）提供。该信号将传递到I\$，但需要去掉两个最低有效位，因为指令以4字节（32位）边界对齐。因此，ic_rw_addr = 0x0000070。

使用取指地址的子集访问标记数组和数据数组，如图4所示。访问结果将在下一周期提供。

- **周期i+1:** 四个标记（每个高速缓存通路一个）位于信号TagWay0-TagWay3中。这些标记将与adr_ff中寄存的取指地址（输出信号ic_rw_addr_ff）的TAG字段进行比较。在本例中，所有标记均与TAG字段相同，但只有一条通路（通路0）有效（ic_tag_valid = 0001），因此将在通路0中发出命中消息：ic_rd_hit = 0001。

四个128位指令束同样位于信号DataWay0-DataWay3中。图4所示的4:1多路开关用于选择通路0（即DataWay0）提供的数据。因此：

ic_rd_data_only = 0x01ce0e33007383b300630333005282b3

请注意，图8中显示的信号为ic_rd_data，该信号与加入奇偶校验信息的ic_rd_data_only信号相同。

上述128位将传播到对齐阶段，如图8所示。

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

请注意，这128位所对应的是前四条add指令。

- **周期i+2:** 在对齐阶段，从缓冲区q1中提取第一条和第二条add指令：
 - o ifu_i0_instr = 0x005282b3
 - o ifu_i1_instr = 0x00630333
- **周期i+3:** 在对齐阶段提取第三条和第四条add指令，同时对第一条和第二条add指令进行译码：
 - o ifu_i0_instr = 0x007383b3
 - o ifu_i1_instr = 0x01ce0e33
 - o dec_i0_instr_d = 0x005282b3
 - o dec_i1_instr_d = 0x00630333
- **周期i+4:** 最后，对第三条和第四条add指令进行译码：
 - o dec_i0_instr_d = 0x007383b3
 - o dec_i1_instr_d = 0x01ce0e33

D. 指令高速缓存替换策略

本部分介绍RVfpga系统的高速缓存替换策略。正如Harris & Harris在[DDCARV]第8.3.3节中所述，在组相联高速缓存中，如果高速缓存组已满，则必须选择一个块将其逐出。根据时间局部性原则，最好的选择是逐出近期使用最少的块，因为短期内再次使用该块的可能性最低。因此，大多数组相联高速缓存采用最近最少使用（Least Recently Used, LRU）的替换策略。然而，跟踪最近最少使用的通路较为复杂，因此一般使用简化的LRU策略（通常称为伪LRU），该策略已足够满足实际需求。具体来说，SweRV EH1使用名为二进制树伪LRU的简化策略。

注：如果您尚未阅读[DDCARV]第8.3.3节，请先阅读该节。另外，建议您阅读Gille Damien的硕士论文《有关嵌入式系统中不同高速缓存行替换算法的研究》（2007年3月8日）的第4部分，链接如下：<https://people.kth.se/~ingo/MasterThesis/ThesisDamienGille2007.pdf>。为方便起见，该文档在下文中简称为[GiDa]。

i. 二叉树伪LRU策略在SweRV EH1中的实现

如[GiDa]中所述，二叉树LRU策略是LRU策略的简化版本，要实现该策略，N路相连高速缓存中的每组需要有N-1个位（称为LRU状态）。因此，对于使用4路指令高速缓存的SweRV EH1，每组需要有3个位，以跟踪不同通路的访问历史。

如第3.B部分所述，发生I\$未命中时，必须通过DDR外部存储器请求该块。使用DDR外部内存提供高速缓存块时，必须将该块写入I\$。取指地址的SET字段决定了必须向其中写入新块的I\$组（参见图4）。可能发生两种情况：

- 高速缓存组未满，意味着有一个或多个块无效。在这种情况下，新块将写入包含无效块的最低通路。
- 高速缓存组已满，意味着四个块均有效。在我们的处理器中，二叉树LRU替换策略会决定必须逐出哪个块。该策略根据组的3位LRU状态确定替换的通路，如下表所示（x表示无需考虑该位）：

LRU状态	替换的通路
x00	通路0
x10	通路1
0x1	通路2
1x1	通路3

如前文所述，从模块ifu_mem_ctl中提取的以下Verilog代码片段（图9）可实现选择存储新I\$块所必需的通路的逻辑。

```

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

```

图9. 用于选择必须替换的通路的Verilog代码

图9所示的Verilog代码片段使用以下信号：

- **replace_way_mb_any**（4位）：保存独热编码，其中与必须替换的通路相对应的位为1。
- **way_status_mb_ff**（3位）：保存新块所在组的LRU状态。
- **tagv_mb_ff**（4位）：保存新块所在组的有效位；有效通路所对应的有效位为1，无效通路所对应的有效位为0。

任务：分析图9中的Verilog代码，并基于上述说明解释代码如何运行。

当I\$中发生命中或未命中时，必须根据下表更新组的LRU状态（其中“-”表示位保持不变）：

写入的通路	下一LRU状态
通路0	-11
通路1	-01
通路2	1-0
通路3	0-0

通过分析该表可以看出，如[GiDa]所述，在发命中或未命中时，指向命中/插入行的路径上的位将会反转，将树的相反部分指定为伪LRU。其原理是通过反转指向最后访问数据的节点，确保最后访问的数据不会被逐出。

从模块ifu_mem_ctl提取的以下Verilog代码片段（图10）可实现上述更新LRU状态的逻辑。

```

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];

```

图10. 用于更新LRU状态的Verilog代码片段

图10所示的Verilog代码片段使用以下信号：

- **ic_rd_hit**（4位）：保存发命中中的通路。
- **way_status**和**way_status_mb_ff**（各3位）：保存发命中或替换的组的上一LRU状态。
- **ifu_wr_en_new_q**（1位）：如果发生替换，则信号值为1。
- **way_status_new**（3位）：保存刚刚发命中或未命中的组的新LRU状态。
- **replace_way_mb_any**（4位）：保存独热编码，其中与必须替换的通路相对应的位为1。图9下方也提供了关于该信号的说明。

任务：分析图10中的Verilog代码，并基于上述说明解释代码如何运行。

ii. 二叉树LRU策略的工作演示示例

为了分析SweRV EH1的替换策略，我们在文件夹
[RVfpgaPath]/RVfpga/Labs/Lab19/InstructionMemory_LRU_Example中提供了新的示例。该

- 第五条j指令（j Set8_Block1）位于地址0x00004200处。根据图4，访问IS时的地址划分如下：

IS地址（二进制）= 00000000000000000000000010000100000000

TAG = 0x4

SET = 0x8

OFFSET = 0x0

当该程序（图11）执行第一次迭代时，组8的初始状态为空。图12所示为执行第一次迭代后，IS中组8理论上的变化情况。随后，我们将展示几项Verilator仿真，以证实这些理论说明。

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

图12. 图11中的循环执行第一次迭代时的IS组8

下面的Verilator仿真展示了循环第一次迭代期间的高速缓存信号，仿真结果证实了图12中所示的分析。图13所示的Verilator仿真为程序执行第一条j指令（j Set8_Block2）后的状态。该指令地址（0x200）同样映射到I\$的组8。该组的初始状态为空：tagv_mb_ff = 0000。因此，根据二叉树LRU策略，必须将新块写入通路0。replace_way_mb_any = ic_wr_en = 0001。组8的LRU状态更新如下：way_status_new = 011。

回想一下第3.B部分，该块从DDR存储器中读取，并以64位块的形式写入I\$。图13展示了将新块的标记和前两条指令写入组8的过程：

```
ic_rw_addr_q[11:4] = 00100000 (组8)
ic_tag_wr_data[19:0] = 0x0 (最高有效位用作纠错位，未包含在此处)
ic_wr_data1[31:0] = 0x0000106F (j Set8_Block2)
ic_wr_data2[31:0] = 0x00000013 (nop)
```

（ic_wr_data1和ic_wr_data2是为了清楚起见而创建的信号，但I\$中实际使用的信号为ic_wr_data[67:0]，其中包括两条指令和一些奇偶校验信息）。

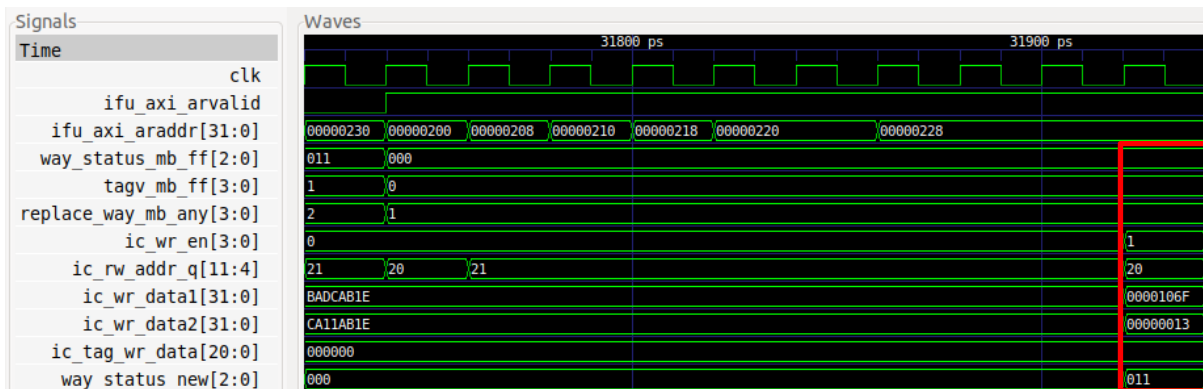


图13. 执行第一条j指令后组8的LRU状态

图14所示的Verilator仿真为程序执行第二条j指令（j Set8_Block3）后的状态。该指令地址（0x1200）同样映射到I\$的组8。在该组中，仅通路0有效：tagv_mb_ff = 0001。因此，根据二叉树LRU策略，必须将新块写入通路1：replace_way_mb_any = ic_wr_en = 0010。组8的LRU状态更新如下：way_status_new = 001。

与上文类似，图14展示了将新块的前两条指令写入组8的过程：

```
ic_rw_addr_q[11:4] = 00100000 (组8)
ic_tag_wr_data[19:0] = 0x1 (最高有效位用作纠错位，未包含在此处)
ic_wr_data1[31:0] = 0x0000106F (j Set8_Block3)
ic_wr_data2[31:0] = 0x00000013 (nop)
```

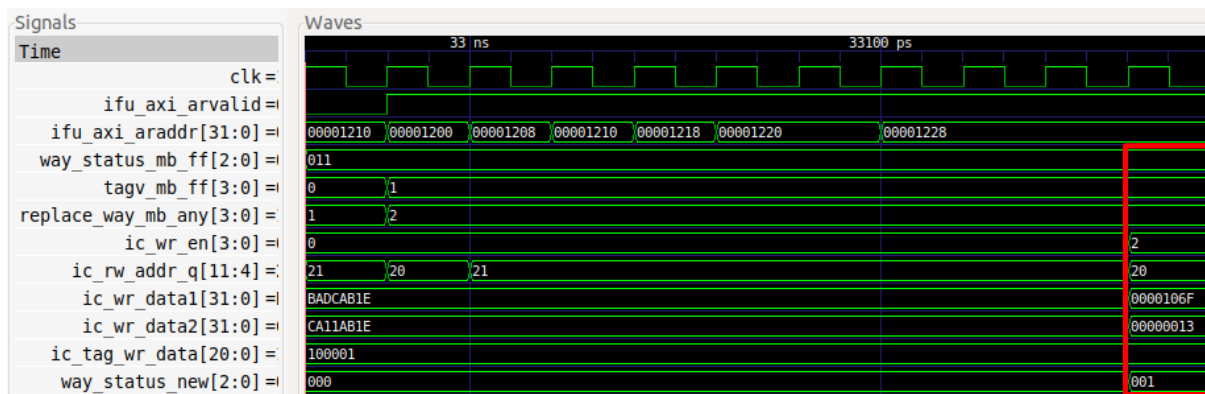


图14. 执行第二条j指令后组8的LRU状态

图15所示的Verilator仿真为程序执行第五条j指令（j Set8_Block1）后的状态。该指令地址（0x4200）同样映射到I\$的组8。但是，与先前的情况相反，此时组已满：tagv_mb_ff = 1111。因此，根据二叉树LRU策略，必须将新块写入通路1：replace_way_mb_any = ic_wr_en = 0001。组8的LRU状态更新如下：way_status_new = 011。

与上文类似，图15展示了将新块的前两条指令写入组8的过程：

```
ic_rw_addr_q[11:4] = 00100000 (组8)
ic_tag_wr_data[19:0] = 0x4 (最高有效位用作纠错位，未包含在此处)
ic_wr_data1[31:0] = 0x800fc06f (j Set8_Block1)
ic_wr_data2[31:0] = 0x00008067 (ret)
```

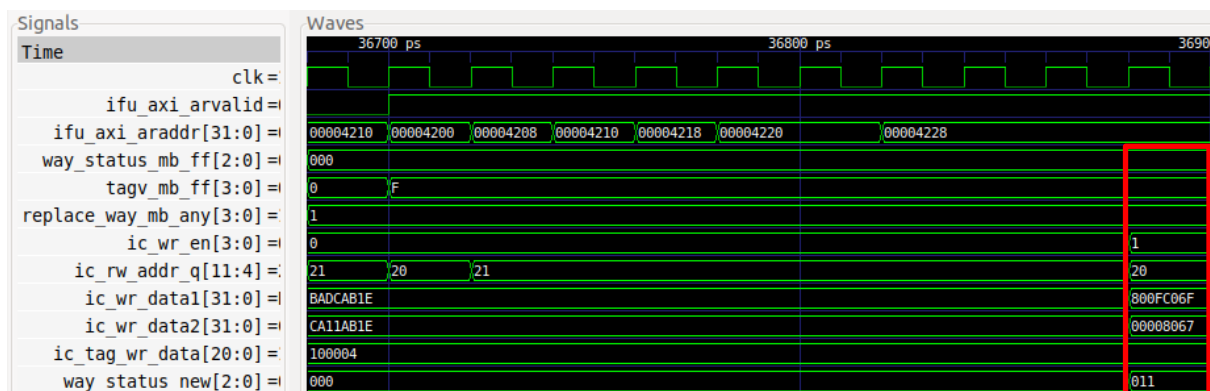


图15. 执行第五条j指令后组8的LRU状态

任务： -在自己的计算机上重复图13-图15中的仿真过程。

4. 练习

- 1) 将图11所示的循环转换为0x10000次迭代的循环，但为j指令保持原有的地址。测量周期数以及I\$命中和未命中的次数。然后删除其中一条j指令，再次测量上述指标。比较测量结果，并给出解释。
- 2) 使用图5中的程序，从I\$替换策略的角度分析I\$命中。
- 3) 扩展图6，详细分析每个64位块如何写入I\$。
- 4) 通过仿真器和开发板分析其他I\$配置，例如具有不同块大小的I\$。请注意，无法修改通路的数量。
- 5) 分析用于检查数据数组和标记数组奇偶校验信息正确性的逻辑。