



Imagination大学计划

RVfpga实验13

访存指令：lw和sw指令

1. 简介

在前面的实验中，我们介绍了流水线的基本概念及其在SweRV EH1处理器中的使用，并分析了算术-逻辑指令在此处理器中的执行方式。在本实验中，我们将继续分析基本指令；具体来说分析存储器读写。

存储系统是现代计算机中最关键的性能瓶颈之一。存储器延时通常远高于内核时钟周期，因此处理器可能不得不在等待来自存储器的数据时暂停。

在本实验中，首先检查读取低延时存储单元（即不暂停处理器的存储单元）时的**装载/存储管道**（专门用于执行**装载/存储**操作的一组流水线阶段）。随后检查存储指令的执行。最后，重复我们的分析，忽略低延时存储器并直接与Nexys A7板上提供的DDR主存储器交互。

图1给出了SweRV EH1处理器微架构的高级视图。图中突出显示了与本实验相关的阶段：译码、DC1-3（数据访问阶段1-3）、提交和回写。

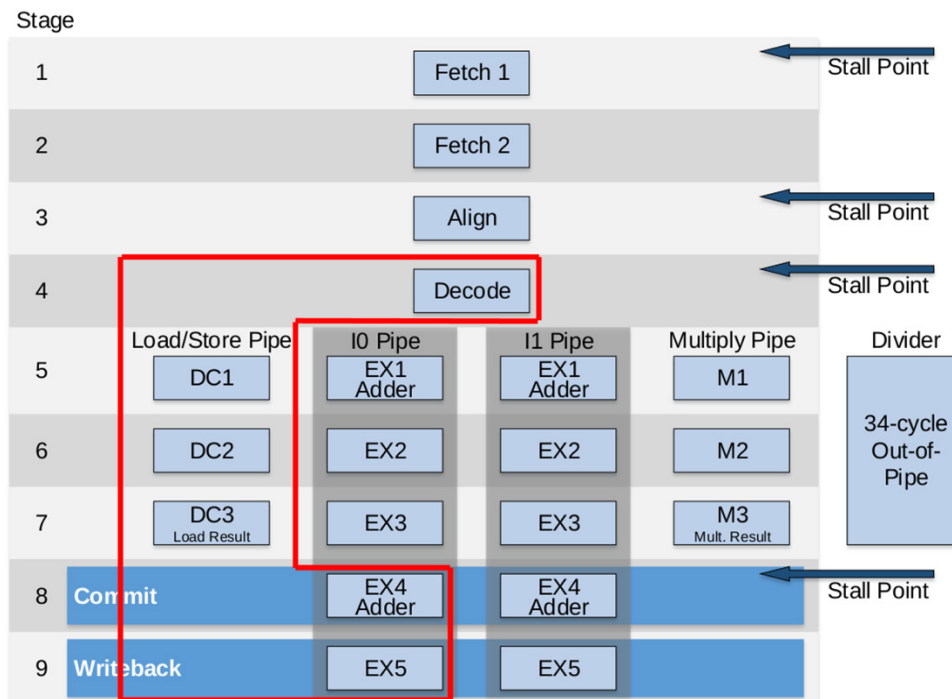


图1. SweRV EH1内核微架构

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

2. 访问低延时存储器的lw指令

在本部分中，我们将使用图2中的简单代码说明与装载指令的执行最相关的事件。示例程序由一个循环组成，此循环包含两个lw（装载字）操作（以红色突出显示），每个操作从连续的字对齐存储器地址读取一个32位字。所有迭代均访问相同的数据，但不执行任何操作。

与实验12一样，lw指令（图中以红色突出显示）前后存在几条nop（无操作）指令，以便将其与前方和后方的指令隔离。为简单起见，在本实验中，我们还禁止使用压缩指令（如SweRVref文档中所述）。

```
.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
```

图2. 具有两条lw指令的示例程序

文件夹[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM提供PlatformIO项目，以便可以分析、仿真和更改程序。在PlatformIO中打开、编译项目，然后打开反汇编文件（位于[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis）。在此文件中，找到第二条lw指令（位于地址0x0000014c处）。请注意指令的机器代码（0x0042a303）：

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

任务：验证这32位（0x0042a303）是否对应于RISC-V架构中的指令lw t1,4(t0)。

到目前为止，在入门指南（GSG）和之前的实验中，我们一直使用Nexys A7板上提供的DDR存储器来存储程序中的指令和数据。但是，访问外部存储器需要几个周期，这样便很难分析装载/存储指令的各个阶段；因此，在本部分中，我们将使用SweRV EH1中提供的低延时DCCM（数据紧耦合存储器）来存储程序数据。

DCCM是与内核紧密耦合的本地存储器。它提供低延时访问和SECDED ECC保护¹。它的大小在内核编译时以参数形式设置，范围为4 KiB至512 KiB（默认值为64 KiB）。在实验20中，我们将更详细地分析DCCM和ICCM；在本实验中，我们只是使用其来简化装载/存储指令的分析。请注意，采用这种方式时，一切操作均在包含SweRV EH1流水线和DCCM（以红色突出显示）的SweRV EH1内核组合（图3）内发生。

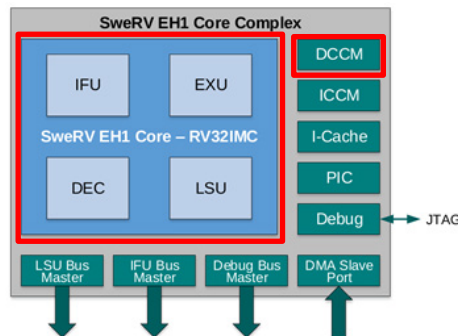


图3. SweRV EH1内核组合

图2中的代码定义了一个名为`midccm`的`ad-hoc`（特定的）部分来分配DCCM中的空间。默认情况下，在默认RVfpga系统中，DCCM的地址空间从0xF0040000开始。此项目提供的链接脚本（位于：[\[RVfpgaPath\]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/ld/link.lds](#)）将进行正确的地址分配。通过在文件

[\[RVfpgaPath\]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/platformio.ini](#)中包含以下命令来使用此链接脚本：

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

A. 1w指令的基本分析

图4给出了图2中循环的中间迭代期间第二条1w指令的执行情况。显示的信号是以下文件中指定的信号：[\[RVfpgaPath\]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/scriptLoad.tcl](#)。请注意，所有迭代均相同：第一次装载将DCCM的第一个数据字（2）读入`t1`（x6）；第二次装载将DCCM的第二个数据字（8）读入同一个寄存器（`t1`）。

¹更多信息，请参见《RISC-V SweRV™ EH1编程人员参考手册》。

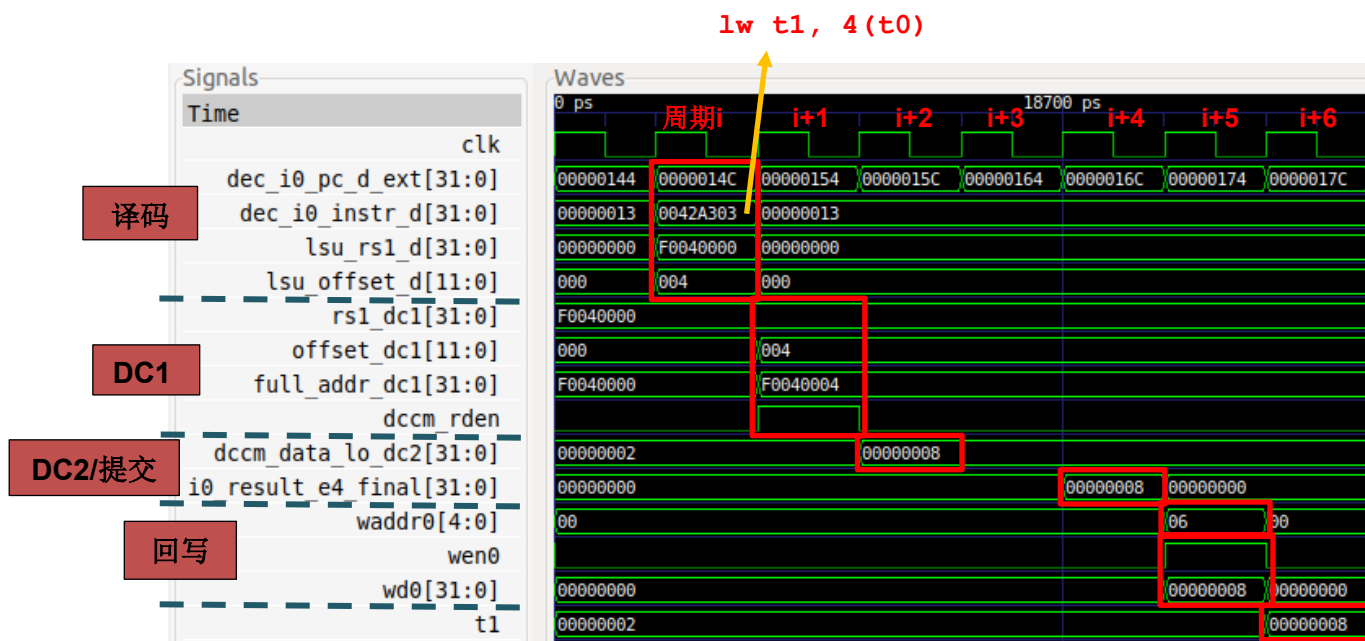


图4. 图2中示例程序的Verilator仿真

图5给出了第二条lw指令执行期间SweRV EH1流水线的高级视图。请注意图中合并了处理器在不同时钟周期的状态：

- 周期i: 对指令进行译码并读取寄存器文件。
- 周期i+1: 使用加法器计算有效地址。
- 周期i+2: 使用前一阶段计算的地址读取DDCM。
- 周期i+5: 将从存储器读取的值写入寄存器文件。

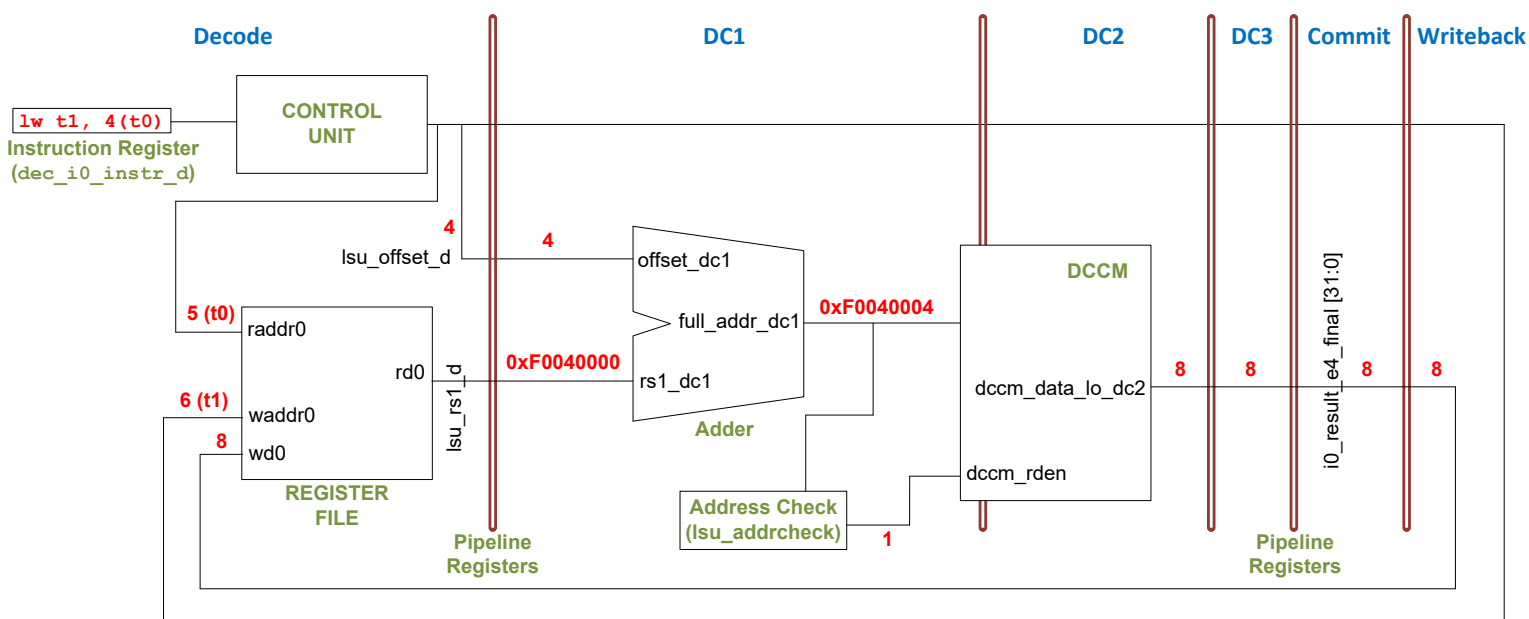



图5. 在SweRV EH1流水线中执行的lw指令的高级视图

任务： 在自己的计算机上重复图4中的仿真过程。请按照以下步骤操作（如GSG第7部分所详诉）：

- 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/Lab13/LW_Instruction_DCCM。
- 在文件*platformio.ini*中更正到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）。
- 使用GTKWave打开轨迹。
- 使用文件*scriptLoad.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM*中提供）打开与图4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*scriptLoad.tcl*文件。
- 单击几次“*Zoom In*”（放大）（）移动至18600 ps。

同时分析图4中的波形以及图5中的图。图中包括与译码、DC1-3、提交和回写阶段相关的一些信号。以红色突出显示的值对应于第二条lw指令，因为此指令遍历相应阶段。

- **周期i：** 译码：dec_i0_pc_d_ext保存lw指令的地址（0x0000014C），信号dec_i0_instr_d包含lw机器指令的32位（0x0042a303）。

在此阶段，将产生控制信号。此外，还将获得用于计算装载有效地址的操作数：信号lsu_rs1_d包含lw操作的基址（本例中保存在寄存器t0中，等于0xF0040000），信号lsu_offset_d包含从指令中提取的12位有符号立即数（本例中为0x004）。

- **周期i+1：DC1：** 使用模块lsu_lsc_ctl内部的加法器计算地址。地址等于基址（rs1_dc1 = 0xF0040000）加上符号扩展偏移量（offset_dc1 = 0x00000004）；最终地址full_addr_dc1 = 0xF0040004。检查此地址（地址检查）以确定访问的存储器区域（DCCM、PIC或外部存储器）。在本例中，假设最终地址属于DCCM范围（0xF0040004），dccm_rden置为有效可允许相应DCCM存储区的读取。最终地址（full_addr_dc1）和使能信号（dccm_rden）提供给DCCM，然后在下一个周期读取。
- **周期i+2：DC2：** 读取DCCM并将数据置于dccm_data_lo_dc2 = 0x8中，随后传播到下一阶段。
- **周期i+3：DC3：** 将从DCCM读取的数据传播到下一阶段。
- **周期i+4：提交：** 将从DCCM读取的数据（信号i0_result_e4_final = 0x8）传播到下一阶段。
- **周期i+5：回写：** 最后，通过信号wd0 = 0x8将从存储器读取的值写回到寄存器文件中。鉴于wen0 = 1，此值将在此周期结束时写入寄存器x6（waddr0 = 0x6）。可以发现，在接下来的周期（图4中的最后一个周期）中，寄存器x6（也称为t1）包含新值（t1 = 0x8）。请注意，波形中显示的信号t1是.tcl脚本中为信号dout定义的别名。

B. lw指令的高级分析

在本部分中，我们将更详细地分析lw指令遍历的阶段。图6给出了本例中的装载指令沿装载/存储管道（DC1、DC2和DC3阶段）执行期间遍历的主要模块。可能需要将图放大才能看到细节。图中标有**LOGIC**的黑色模块包含多路开关和逻辑门等各种模块。为简单起见，图中仅包含模块的部分接口信号。

译码和回写阶段与A-L指令所示的阶段相同（参见实验12中的图6）。但是，我们指出了译码阶段的一些细节。回想一下，译码阶段将产生控制信号并将指令和操作数调度到适当的管道：

- 装载的立即偏移量位于信号lsu_offset_d中。
- 装载的基址位于信号exu_lsu_rsl_d中。（此信号由4:1多路开关（如实验11的图4所示）产生，然后在遍历某些逻辑后传播到DC1阶段。）
- 装载/存储指令的信号位于lsu_p（图6所示的新控制信号包）中。

与译码类似，实验12中也分析了提交阶段，但本实验将包含与装载指令相关的最终3:1多路开关的输入（lsu_result_corr_dc4），而实验12为简单起见对此进行了省略。请记住，此3:1多路开关的输出为i0_result_e4_final[31:0]，如图6所示。此外，我们在本实验中只关注通路0，但装载/存储可通过双路超标量处理器中的任何一路执行。不过请注意，只有一个L/S（装载/存储）管道。因此，通路1也有一个3:1多路开关（其输出为i1_result_e4_final[31:0]，输入之一为lsu_result_corr_dc4），如实验11的图4所示。

任务：扩展图4中的仿真以包含图6所示的信号（在下文说明）。可以使用的.tcl文件位于：
`[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/scriptLoadExtended.tcl`

任务：在SweRV EH1处理器的Verilog文件中找到图6中的模块和信号。

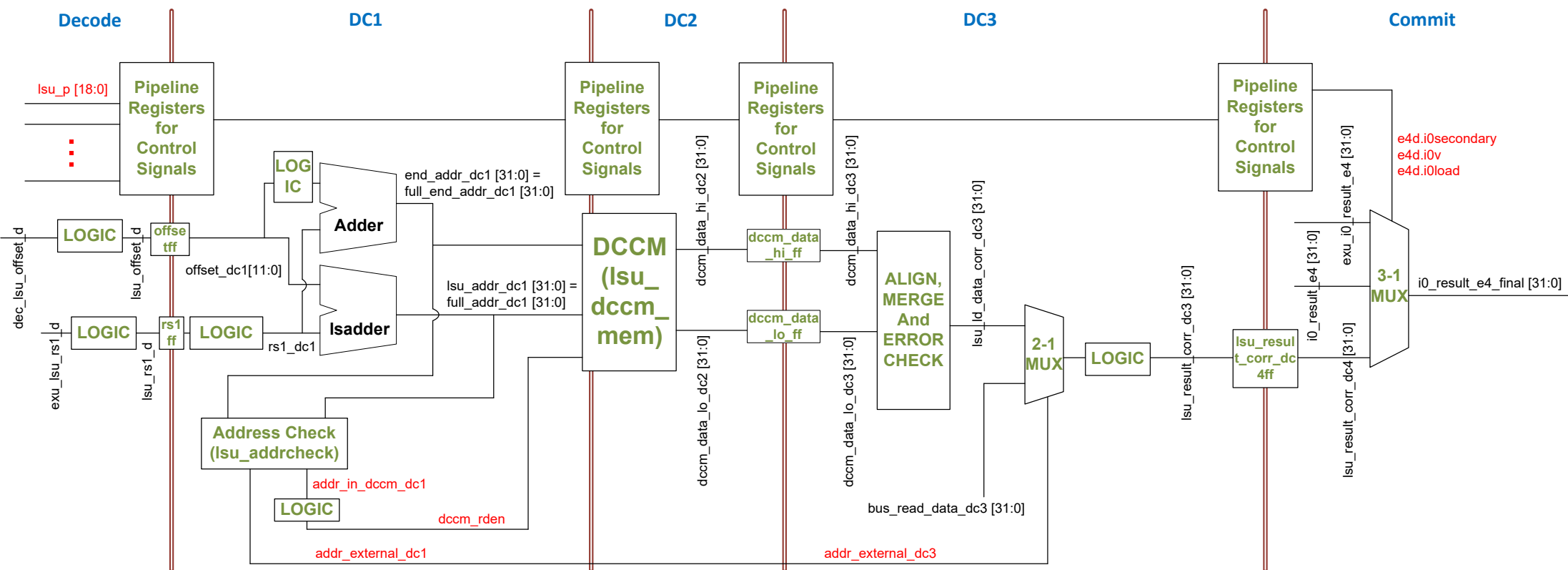


图6. 装载指令在装载/存储管道中遍历的主要模块

i. 译码阶段

译码阶段的一般细节已在实验11和12中进行了分析。请记住，译码阶段负责两个主要任务：

- 对指令进行**译码**并产生**控制信号**。
- 将指令**分发**到适当的管道并提供**输入操作数**。

对指令进行译码并产生控制信号：

除了实验11和12中已经分析过的其他控制信号结构之外，还有一个附加结构`lsu_pkt_t`，其中包含装载/存储指令信号。通常，此结构在文件 `[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv_types.sv` 中定义。信号`lsu_p`是此类信号的一个示例，从译码阶段传播到装载/存储管道阶段。

此信号封装了存储器读/写的一些相关信息：

- o **位0** (`valid`) 在操作有效时置1。
- o **位12** (`unsign`) 在要读/写的数据为无符号时置1。
- o **位13** (`store`) 在存储操作 (`sb`、`sh`、`sw`...) 时置1。
- o **位14** (`load`) 在装载操作 (`lb`、`lh`、`lw`...) 时置1。
- o **位15-18**：对访问的大小（字节、半字、字和双字）进行编码。

任务：在图4的仿真中包含信号`lsu_p`并根据上述说明分析其各个位。

将指令分发到适当的管道并提供输入操作数：

如实验11中所述，SweRV EH1处理器包括多个用于执行指令的管道。在译码阶段，指令一旦被译码，就必须通过适当的管道进行调度。在本实验分析的程序（图2）中，待执行的`lw`指令发送到LSU管道（阶段DC1-3）。具体来说，`exu_lsu_rs1_d`是基址寄存器中保存的值。信号`dec_lsu_offset_d`是12位有符号立即偏移量，从指令中提取并发送到DC1阶段。

任务：在Verilog代码中从LSU的两个输入（`exu_lsu_rs1_d`和`dec_lsu_offset_d`）的获取来源分析这两个输入所遵循的路径。此过程涉及几个模块：**dec**、**exu**和**lsu**。

ii. DC1阶段

在DC1阶段，`rs1_dc1`（基址，从译码阶段传播）和`offset_dc1`（偏移量，从译码阶段传播）添加到模块**lsadder**中以计算**主要有效地址**（信号`full_addr_dc1[31:0]`，此地址会分配给`lsu_addr_dc1[31:0]`）。这是要读取的存储器地址。

除了要读取的地址外，*结束地址*（`end_addr_dc1[31:0]`）也在另一个加法器中计算（应强调一点，为简单起见，第二个加法器不在图5中显示，也不在实验11的图4中显示）。这是需要从存储器中读取的最后一个字节的地址。此地址用于处理未对齐访问和子字（字节和半字）访问。

任务：分析DC1阶段中两个加法器的实现，这两个加法器在模块`lsu_lsc_ctl`中实例化。我们通过展示这些加法器的实现在下面的图7中提供指导。

```

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];

```

图7. DC1阶段加法器的Verilog代码（来自文件`lsu_lsc_ctl.sv`）

例如，针对从`0xF0040003`开始的地址的装载字（`lw`）满足：`full_addr_dc1=0xF0040003`且`end_addr_dc1=0xF0040006`（参见图8）。这样，LSU管道便可从读数据束中提取字，读数据束由两个字组成，这两个字从等于四的倍数的地址开始（本例中为`0xF0040000`和`0xF0040004`）。

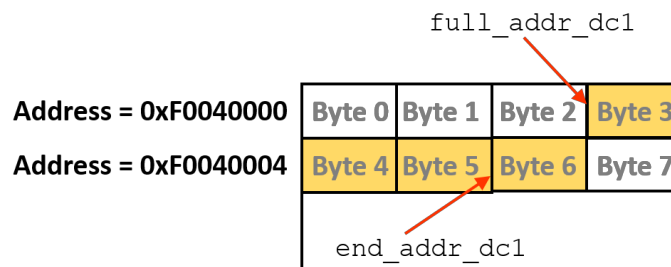


图8. 针对地址`0xF0040003`的`lw`指令示例

两个地址（`lsu_addr_dc1[31:0]`和`end_addr_dc1[31:0]`）发送到数据存储器（本例中为DCCM），将在下一个周期访问。

任务：在图2的程序中，尝试不同的访问大小（字节和半字）和未对齐访问。为此，请更改偏移量或将访问类型从`lw`更改为`lb`（装载字节）或`lh`（装载半字）。例如，如果将偏移量从4更改为3，则装载字指令将对从地址`0xF0040003`开始的32位执行未对齐访问，如图8所示。分析上述不同情况下信号`lsu_addr_dc1[31:0]`（或`full_addr_dc1[31:0]`）和`end_addr_dc1[31:0]`的值。
在实验20中，我们将从DCCM的内部分析这种情况。

除了地址计算之外，DC1阶段还在模块`lsu_addrcheck`中执行地址范围检查（参见图9）来确定访问的目标存储器（本例中为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]),
    .*
);
```

图9. 检查存储器地址的范围和位置

通过地址检查的结果可确定必须访问的存储器：DCCM、PIC或外部DDR存储器（参见图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
```

图10. 每个存储器单元的地址

在本例中，DCCM读使能信号变为高电平（`addr_in_dccm_dc1 = 1`）。此信号在遍历一些逻辑后提供给DCCM（信号`dccm_rden`）以允许/禁止访问（本例中允许访问）。信号`addr_external_dc1`在必须使能外部DDR存储器时为1，其他情况时为0，由DC3阶段传播和使用，如图6所示。

iii. DC2阶段

如果使能了DCCM读取（`dccm_rden = 1`），则会在此阶段读取数据。请注意，将读取两个32位值（`dccm_data_lo_dc2[31:0]`和`dccm_data_hi_dc2[31:0]`），因为数据访问可能未对齐，因此分布在两个字上（例如图8中的示例）。

任务：在图2的程序中，当对地址0xF0040004和地址0xF0040003执行lw时，比较信号`dccm_data_lo_dc2[31:0]`和`dccm_data_hi_dc2[31:0]`的值。

iv. DC3阶段

来自DCCM的两个32位数据值从DC2（信号`dccm_data_lo_dc2[31:0]`和`dccm_data_hi_dc2[31:0]`）传播到DC3（信号`dccm_data_lo_dc3[31:0]`和`dccm_data_hi_dc3[31:0]`）。对于对齐访问（例如本例中的访问），两个信号相等，仅使用`dccm_data_lo_dc3[31:0]`。

在DC3阶段，前一个周期读取的两个字（`dccm_data_lo_dc3[31:0]`和`dccm_data_hi_dc3[31:0]`）通过执行多项任务的逻辑：

- **错误检查：**使用ECC检查数据是否存在错误。
- **处理装载/存储冒险：**如果针对同一地址的存储指令仍在执行，则将数据从存储指令转发到装载指令，而不是从存储器中读取。我们将在实验15中分析这种情况。
- **对齐：**对齐请求的数据。

上述所有过程的结果是，lsu_ld_data_corr_dc3[31:0]信号中提供最终数据。

任务：分析lsu_dccm_ctl和lsu_ecc模块中的Verilog代码中使用的对齐、合并和错误检查逻辑。

任务：在图2的程序中，当对地址0xF0040004和地址0xF0040003执行lw时，比较信号lsu_result_corr_dc3[31:0]的值。

在执行错误检查、装载/存储冒险处理和对齐的逻辑之后，2:1多路开关将在来自DCCM的数据（lsu_ld_data_corr_dc3[31:0]）或来自DDR存储器的数据（bus_read_data_dc3[31:0]）之间进行选择。此多路开关由信号addr_external_dc3控制，该信号于DC1阶段在模块lsu_addrcheck中生成（信号addr_external_dc1）。

任务：在Verilog代码中分析信号addr_external_dc1如何于DC1阶段在模块lsu_addrcheck中计算。

此2:1多路开关的输出（lsu_result_corr_dc3[31:0]）传播到提交阶段。

v. 提交阶段

在提交阶段，3:1多路开关选择要发送到回写阶段的读取数据（i0_result_e4_final[31:0]）（参见图6）。此3:1多路开关还可选择ALU的输出（实验11和12中已进行说明）以及辅助ALU的输出（将在实验15中分析）。

vi. 回写阶段

此阶段已在实验11和12中说明，因此在图6中未显示，其中加法的结果已写入目标寄存器。在这种情况下，此阶段将DCCM数据写入目标寄存器。

3. 访问低延时存储器的sw指令

在本部分中，我们将使用图11给出的代码说明与执行存储指令最相关的事件。此代码包含一个具有1000次迭代的循环，此循环将写入存储器的连续地址。向量A包含1000个字，位于DCCM（0xF0040000 – 0xF004FFFF）。每条sw指令后跟一条lw指令，前者用于检查是否已

存储正确的值。通常会插入nop来隔离指令，在这种情况下，还需确保数据实际写入寄存器以及从寄存器读取，而不仅仅是从sw指令转发到lw。通常将禁止使用压缩指令（如SweRVref文档中所述）。此外，与前一部分的示例一样，我们将使用DCCM来存储和装载数据。

```
.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
```

图11. 使用sw指令的示例代码

文件夹[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM提供PlatformIO项目，以便可以分析、仿真和更改程序。在PlatformIO中打开、编译项目，然后打开反汇编文件（位于[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM/.pio/build/swervolf_nexys/firmware.dis）。可以看到sw指令位于地址0x00000110处，还可以看到指令的机器代码（0x0062a023）：

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

任务：验证这32位（0x0062a023）是否对应于RISC-V架构中的指令sw t1, 0(t0)。

图12 给出了图11中循环的第四次迭代期间sw指令的执行情况。可以分析除第一个迭代之外的任何迭代。通常，不应使用指令的第一次执行以避免指令高速缓存（I\$）未命中。

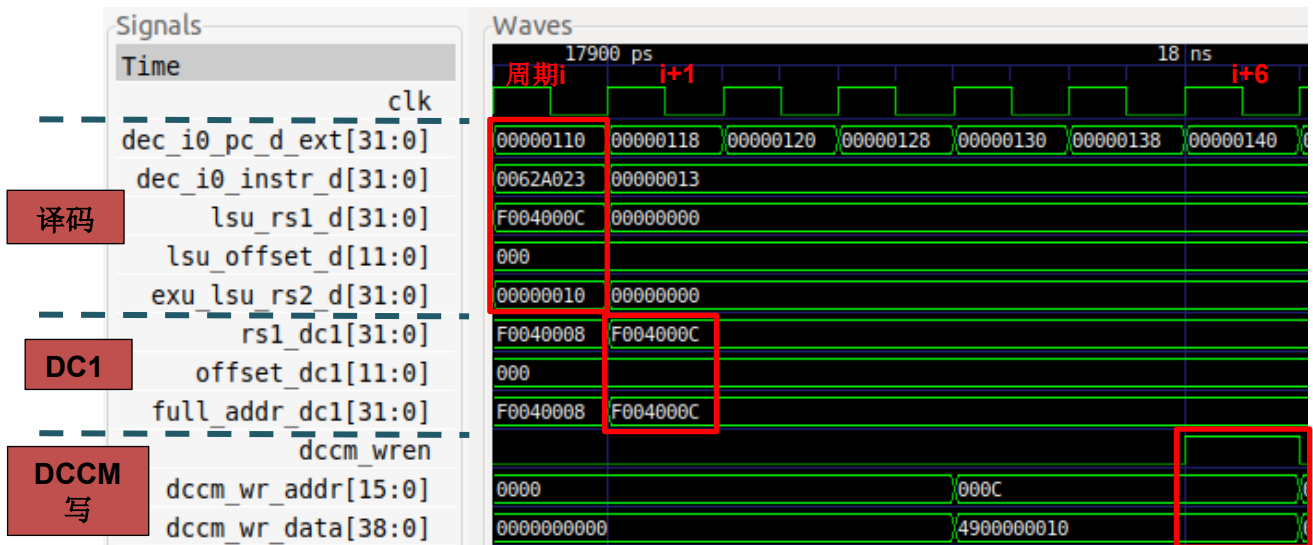


图12. 图11示例的Verilator仿真

图13给出了图11中循环的第四次迭代期间执行sw指令时SweRV EH1流水线的高级视图。寄存器t1（保存要写入存储器的值）为0x10，t0（保存基地址）为0xF004000C。因此，sw将值0x10写入DCCM地址0xF004000C。图中给出了SweRV EH1处理器的Verilog模块中使用的实际名称。请注意图中合并了处理器在不同时钟周期的状态：

- **周期i:** 存储指令在译码阶段进行译码，并分配给LSU管道，在这种情况下，操作数由在此周期中读取的指令立即数字段和寄存器文件提供。
- **周期i+1:** 有效地址在加法器单元计算（如load部分所述）。请注意，为简单起见，图中仅包含图6所示的lsadder。
- **周期i+6:** 第二个操作数（从寄存器t1读取）在遍历存储缓冲区后存储在DCCM中（相关说明请参见附录）。

请注意，就程序执行时间而言，存储不是关键操作，因此它可以延迟几个周期而不影响性能。相比之下，装载指令十分关键，因为它们通常会读取后续指令所需的值，因此（如前一部分所述）实现的是存储-装载转发路径（图13中未显示），这样有助于减少存储器访问，并避免在针对同一存储器地址的存储和后续装载操作之间发生数据冒险时流水线暂停。我们将在实验15中分析这种情况。

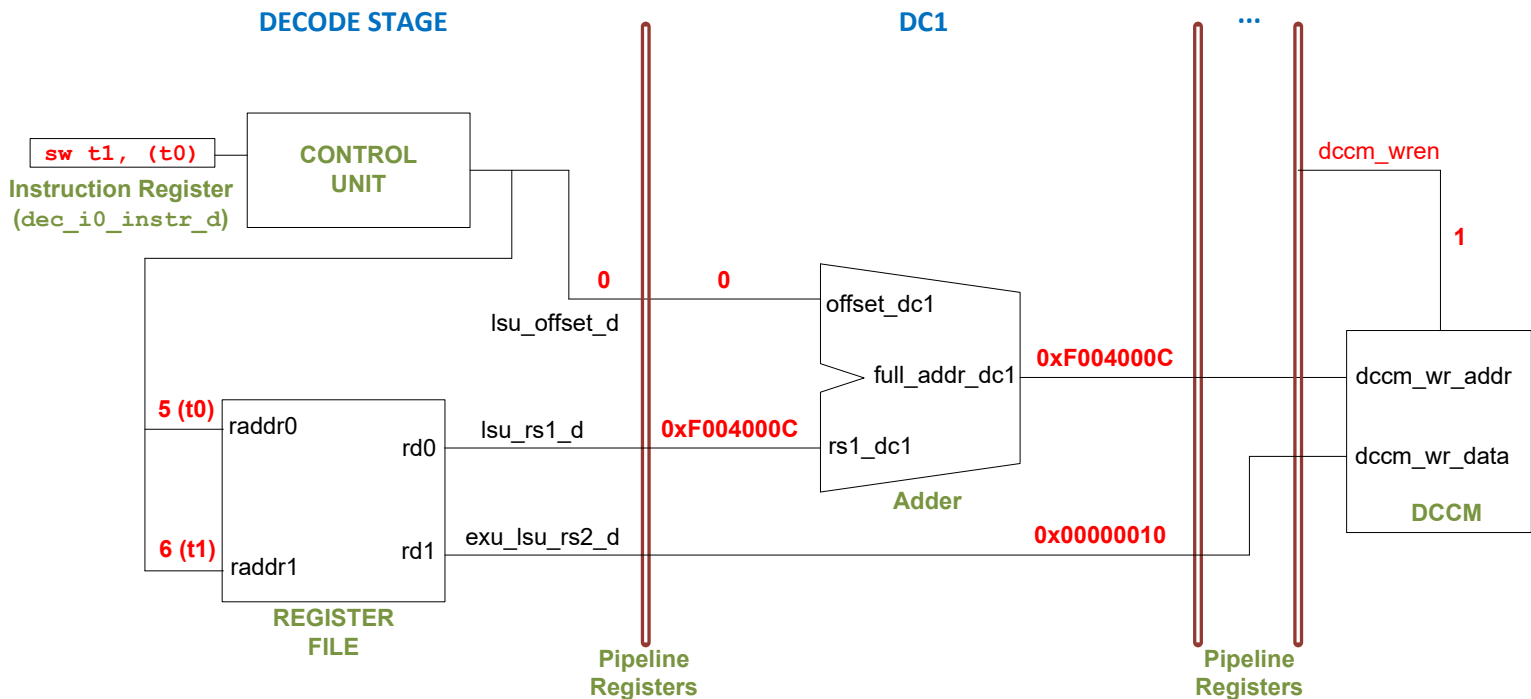



图13. SweRV EH1中sw指令执行的高级视图

任务： 在自己的计算机上重复图12中的仿真过程。请按照以下步骤操作（如GSG第7部分所详诉）：

- 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/Lab13/SW_Instruction_DCCM。
- 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*scriptStore.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM*中提供）显示与图4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*scriptStore.tcl*文件。
- 单击几次“Zoom In”（放大）（）移动至17900 ps。

同时分析图12中的波形以及图13中的图。图中包括一些与译码和DC1阶段相关的信号，以及一些与DCCM写入相关的信号，写入发生在几个周期之后。以红色突出显示的值对应于sw指令，因为该指令遍历相应阶段。

- **周期i：** **译码：** 如装载指令部分所述，信号*dec_i0_pc_d_ext*包含sw指令的地址（0x00000110），信号*dec_i0_instr_d*包含32位sw指令（0x0062a023）。信号*lsu_rs1_d*包含sw操作的基址（本例中为0xF004000C，由寄存器t0提供），信号*lsu_offset_d*包含从指令中提取并随后添加到基址的12位立即数（本例中为0x000）。对于存储指令，从第二个寄存器（本例中为t1）读取的值最终写入存储器（*exu_lsu_rs2_d* = 0x10）。因此，它必须传播到后续阶段。

- **周期i+1: DC1:** 正如装载部分所述，此阶段计算地址 ($\text{full_addr_dc1} = \text{rs1_dc1} + \text{offset_dc1} = 0xF004000C$)。
- **周期i+6: DCCM写入:** 五个周期后，DCCM从存储缓冲区收到写数据和写地址 ($\text{dccm_wr_addr}=0x000C$ 和 $\text{dccm_wr_data}=0x4900000010$)。请注意，DCCM仅接收地址 (0x000C) 的最后16位，因为在我们的配置中其实际大小为64 KiB (参见文件 *common_defines.vh*)，16位足以寻址 2^{16} 个字节。数据已预先添加了一些ECC位 (0x49)。当信号 dccm_wren 置为有效时 (在图12的周期i+6中)，对DCCM的写操作完成。

附录A – 存储缓冲区的操作: 附录A将说明存储缓冲区，这是一个重要结构，用于临时保存存储指令必须写入存储器的值和地址。

任务: 在仿真中分析存储指令之后的装载指令，以验证值是否已正确写入DCCM。需要添加图4和图6中的一些信号来分析装载。

任务: 按照与第2.B部分中对lw指令执行的高级分析类似的方式扩展本部分中对sw指令执行的基础分析。

任务: 分析针对DCCM的未对齐存储以及子字存储：存储字节 (sb) 或存储半字 (sh)。

4. 访问外部存储器

在第2部分和第3部分中，我们使用DCCM来存储和装载数据。在本部分中，我们将分析访问Nexys A7上所提供外部存储器的装载指令。请注意，这种情况 (参见图14) 与第2部分中分析的情况 (参见图3) 相反，SweRV EH1内核必须通过AXI总线与外部存储器通信，才能获得装载指令请求的数据。

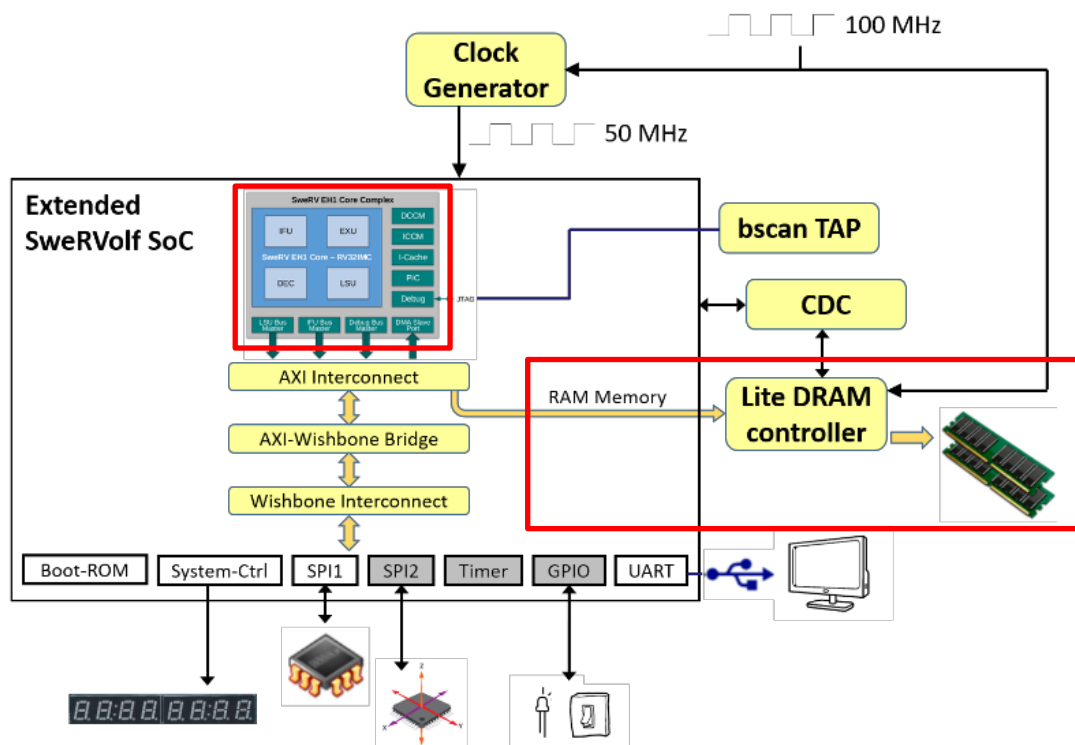


图14. RVfpgaNexys

我们将分析阻塞和非阻塞访问。阻塞装载会完全停止处理器，直到其接收到从存储器读取的数据。这意味着在装载接收到其数据之前没有其他指令执行。相比之下，只要指令不依赖于装载读取的数据，非阻塞装载便允许程序继续执行；仅当执行依赖于装载的指令时，才会停止执行。在这两种情况下，从存储器中读取的数据遵循流水线中的不同路径；在本实验中，我们将分析第一种情况（阻塞装载），而在下一个实验（实验14）中，我们将在结构冒险的背景下分析第二种情况（非阻塞装载）。

图15中的代码给出了说明如何执行读取外部DDR存储器的lw指令的简单示例。此代码包含一个循环，此循环读取一个12元素的数组（lw t3, (t4)）并在寄存器t6中累加其元素的总和（add t6, t3, t6）。通常将插入几个nop操作来隔离指令，以使指令更易于分析，此外还会禁止压缩指令。

向量D包含12个字，位于主存储器中。为此，需在.data部分中声明数组并为项目使用常规链接器脚本（位于~/platformio/packages/framework-wd-riscv-sdk/board/nexys_a7_eh1/link.lds）。这样，.data部分中定义的数据将置于外部存储器中，而不是像图2的程序中一样置于DCCM中。

默认情况下，SweRV EH1中的装载指令是非阻塞的。如果希望装载指令为阻塞指令，则必须在要分析的汇编代码的开头包含接下来的两条指令，如图15所示（有关允许/禁止内核功能的更多说明，请参见SweRVref文档的第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
```

图15. 阻塞lw指令的示例

文件夹[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory提供PlatformIO项目，以便可以分析、仿真和更改程序。如果在PlatformIO中打开、编译项目，然后打开反汇编文件（位于[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory/.pio/build/swervolf_nexys/firmware.dis中），可以看到lw指令位于地址0x000000f4处，还可以看到指令的机器代码（0x000eae03）：

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

访问外部DDR2存储器的阻塞装载的路径几乎与第2部分中所述访问DDCM的装载的路径相同，如图16所示。但是，有一个重要的区别：在某些周期内，处理器将暂停来等待外部存储器提供的数据；随后，当收到请求的数据时，指令可以继续执行。

在SweRV EH1中，通过AXI总线控制外部存储器访问的模块称为lsu_bus_intf。它负责向Lite DRAM控制器提供地址，并在一些周期后接收和对齐请求的数据，然后在DC3阶段将其插入内核。请注意，AXI总线用于与DDR2外部存储器通信。在本例（图15）中，DC3阶段的2:1多路开关（也包含在图6中）选择来自外部存储器的输入（即lsu_result_corr_dc3 = bus_read_data_dc3）而不是图2示例中所选择的来自DDCM的输入（lsu_ld_data_corr_dc3）。提交阶段的3:1多路开关则选择与图2示例中的输入相同的输入（即i0_result_e4_final = lsu_result_corr_dc4）。

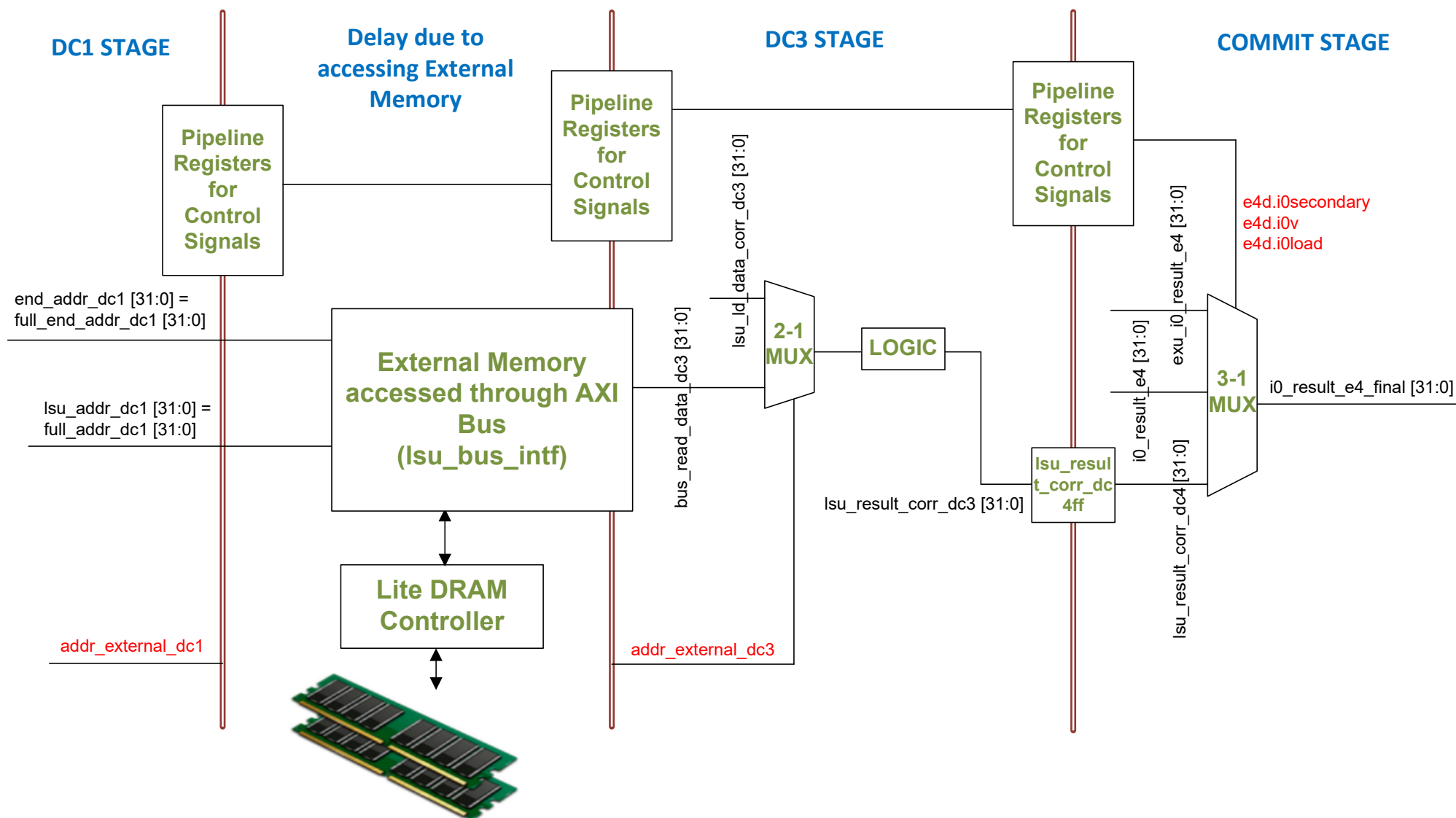


图16. 访问外部存储器的阻塞`lw`指令

图 17 给出了图 15 中循环的第四次迭代期间 `lw` 的执行情况，执行期间会将存储在地址 `0x00002204` 中的值读入寄存器 `t3`。请注意，对于此程序，`D` 数组从地址 `0x000021F8` 开始。

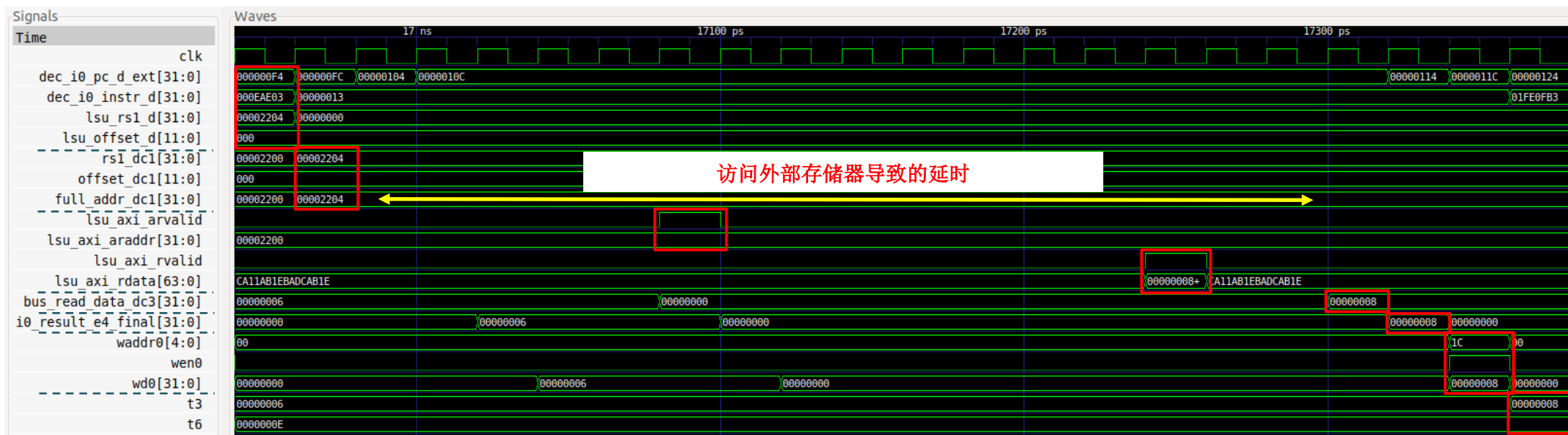



图17. 图15中示例的Verilator仿真

任务： 在自己的计算机上重复图17中的仿真过程。使用文件 `test_Blocking.tcl`（在 `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory` 中提供）。单击几次“Zoom In”（放大）（）移动至16940 ps。

分析波形。图中包括与各流水线阶段相关联的一些信号。请注意，顶部信号组（`clk`至`full_addr_dc1`）和底部信号组（`i0_result_e4_final`至`wd0`）与图4所示的信号组相同。以红色突出显示的值对应于lw指令，因为该指令遍历相应阶段。

- 地址在译码阶段计算，如第2部分所述。信号`full_addr_dc1[31:0]`包含地址，本例第四次迭代中的地址（图17所示的地址）为`0x00002204`。信号`end_addr_dc1[31:0]`（图中未显示）也按照第2部分所述进行计算，其中包含要访问的最后一个字节的地址。
- 几个周期后，地址通过AXI总线使用以下信号发送到外部存储器：`lsu_axi_arvalid = 1`和`lsu_axi_araddr = 0x00002200`。请注意，发送的地址是双字对齐的，因为每次访问都会从存储器中读取64位。数据读入信号`lsu_axi_rdata`（在实验19和20中，我们将详细分析存储器层级）。如果访问需要多个地址（由于未对齐访问），则通过总线发送多个地址并按顺序返回数据。

任务： 修改图15中的程序以分析需要通过AXI总线向外部存储器发送两个地址的未对齐装载访问。

- 几个周期后，外部存储器通过AXI总线返回读取的64位数据（`lsu_axi_rdata = 0x0000000800000006`和`lsu_axi_rvalid = 1`）。此数据在LSU（模块`lsu_bus_buffer`）内部缓存。
- 请求的32位数据通过存储器读取的64位数据提取，将插入到主流水线路径中：`bus_read_data_dc3 = 0x00000008`。
- 随后，此数据遵循与第2部分中的示例相同的路径写入寄存器文件：`i0_result_e4_final` → `wd0`。

任务： 将控制多路开关的信号添加到仿真中（在图16的DC3和提交阶段），其中多路开关选择由DDR外部存储器提供的数据。可以在Verilog代码的以下几行中找到这些多路开关：

- 2:1多路开关：模块`lsu_lsc_ctl`的第264行。
- 3:1多路开关：模块`dec_decode_ctl`的第2277行。

可以使用的`.tcl`文件位于：

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

任务： 分析用于访问DRAM控制器的AXI总线实现也很有趣，为此可以检查`lsu_bus_intf`模块。

下一组信号对应于内部 **存储缓冲区** 信号（位于模块 **lsu_stbuf** 中）。**WrPtr** 对 **存储缓冲区** 的条目进行编码，下一个 **sw** 操作将在 **存储缓冲区** 中放置其地址和数据。在示例中，**WrPtr** 为 **0b011**（即条目编号 **3**，这是第 **4** 个条目，因为编号从 **0** 开始）。

DC3 阶段（周期 **i**）通过将信号 **stbuf_data_en** 的第 **4** 位置为有效来使能 **存储缓冲区** 的第 **4** 个条目（请注意，**0x08** 以独热编码方式转换为 **00001000**，并且惟一的“**1**”值位于第 **4** 位的位置）。信号 **lsu_stbuf_empty_any** 在此周期结束时变为低电平，表示 **存储缓冲区** 不为空 – 也就是说，**存储缓冲区** 保存等待写入存储器的数据。

在提交阶段（周期 **i+1**），更新 **存储缓冲区** 的第 **4** 个条目。信号 **stbuf_addr** 和 **stbuf_data** 的第 **4** 个条目中包含：**0x000C**（对应于要写入的 **DCCM** 地址）和 **0x00000010**（对应于要存储在 **DCCM** 中的数据）。**WrPtr** 已递增为指向下一个缓冲区条目（**b100**），并且 **RdPtr** 会跟踪缓冲区中尚未提交的最早值（**b011**）。

在回写阶段后的一个周期（周期 **i+3**），**DCCM** 写使能信号（**dccm_wren**）置为有效，以便写入存储器并释放缓冲区的第 **4** 个条目。最后，在周期 **i+4**，更新 **RdPtr**（**b100**）并且缓冲区再次为空，这样 **lsu_stbuf_empty_any** 就会再次变为高电平。

图 **19** 说明了在图 **18** 给出的示例中的 **8** 条目 **存储缓冲区** 如何变化。在周期 **i**，**存储缓冲区** 为空，此状态由 **WrPtr == RdPtr** 表示。在周期 **i+1**，**存储缓冲区** 包含一个地址/数据对（**0x000C/0x00000010**），对应于图 **18** 中分析的存储。最后，在周期 **i+4**，数据写入 **DCCM**，**存储缓冲区** 再次变为空（**WrPtr == RdPtr**）。

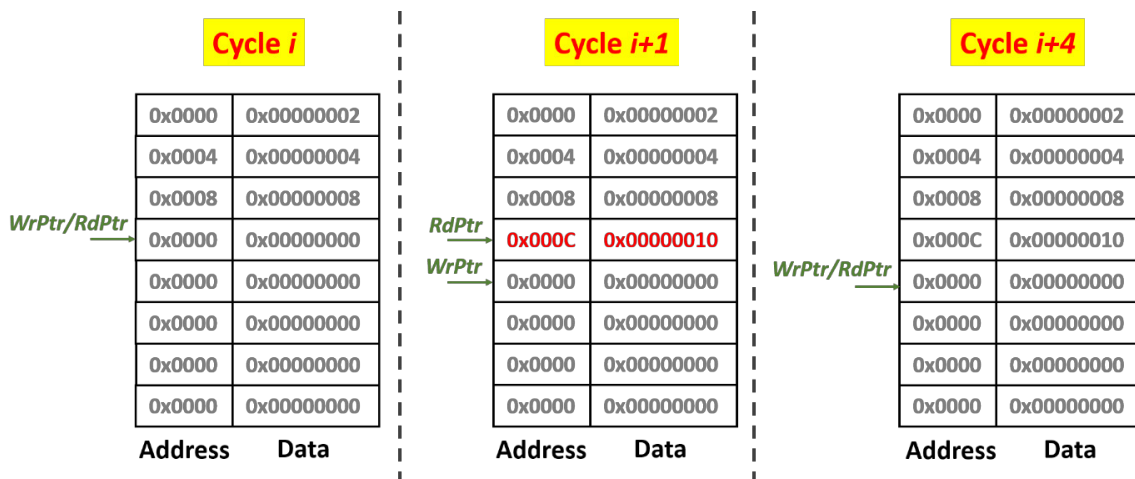


图 **19**. 图 **18** 的示例期间 **存储缓冲区** 的变化

任务： 修改图 **11** 中的程序以实现两个未完成的存储操作，并执行与图 **18** 中的分析类似的分析。