

## 任务

**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)`。

0x0042a303 → 000000000100 00101 010 00110 0000011

imm<sub>11:0</sub> = 000000000100

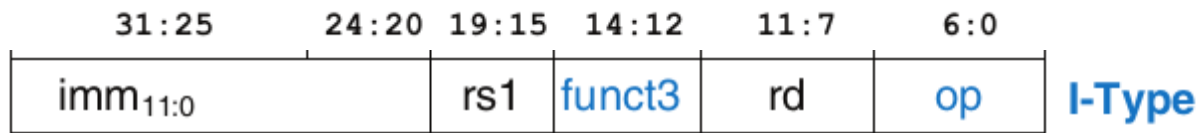
rs1 = 00101 = x5 (t0)

funct3 = 010

rd = 00110 = x6 (t1)

op = 0000011


来自DDCARV的附录B:



op	funct3	funct7	Type	Instruction	Description	Operation
0000011 (3)	010	–	I	lw rd, imm(rs1)	load word	rd = [Address] <sub>31:0</sub>

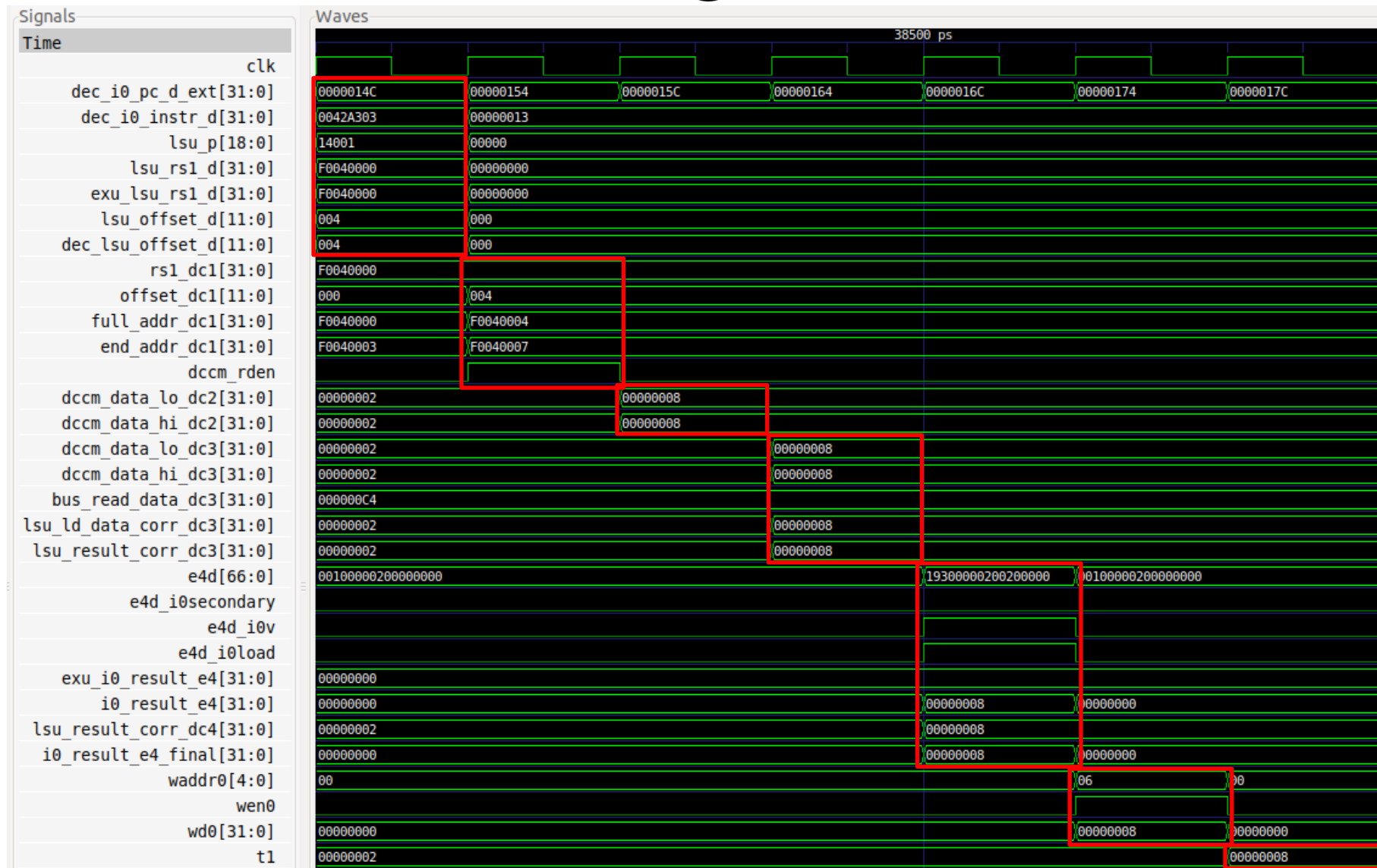
Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved variables
t3-6	x28-31	Temporary variables

**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。

解答请参见实验13的主文档。

**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）。



**任务：**在SweRV EH1处理器的Verilog文件中找到图6中的结构和信号。

不提供解答。

**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。

请参见上面的仿真。可以看到，当装载进行译码时，`lsu_p = 0x14001`：

- `valid = 1`。指令有效。
- `load = 1`。指令为装载指令。
- `word = 1`。访问的大小是字。

**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**。为其他指令分析这些信号的行为。

```
298     assign exu_lsu_rs1_d[31:0] = ({32{ ~dec_i0_rs1_bypass_en_d & dec_i0_lsu_d }} & gpr_i0_rs1_d[31:0] ) |
299     ({32{ ~dec_i1_rs1_bypass_en_d & ~dec_i0_lsu_d & dec_i1_lsu_d }} & gpr_i1_rs1_d[31:0] ) |
300     ({32{ dec_i0_rs1_bypass_en_d & dec_i0_lsu_d }} & i0_rs1_bypass_data_d[31:0]) |
301     ({32{ dec_i1_rs1_bypass_en_d & ~dec_i0_lsu_d & dec_i1_lsu_d }} & i1_rs1_bypass_data_d[31:0]);
```

基址可以来自寄存器文件或旁路（来自通路0或通路1）。

```
1064     assign dec_lsu_offset_d[11:0] =
1065     ({12{ i0_dp.lsu & i0_dp.load }} & i0[31:20]) |
1066     ({12{ ~i0_dp.lsu & i1_dp.lsu & i1_dp.load }} & i1[31:20]) |
1067     ({12{ i0_dp.lsu & i0_dp.store }} & {i0[31:25], i0[11:7]}) |
1068     ({12{ ~i0_dp.lsu & i1_dp.lsu & i1_dp.store }} & {i1[31:25], i1[11:7]});
```

偏移量来自通路0或通路1指令的32位。

**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。

文件**beh\_lib.sv**:

```

251  module rvlsadder
252  (
253      input logic [31:0] rs1,
254      input logic [11:0] offset,
255
256      output logic [31:0] dout
257  );
258
259      logic          cout;
260      logic          sign;
261
262      logic [31:12]  rs1_inc;
263      logic [31:12]  rs1_dec;
264
265      assign {cout,dout[11:0]} = {1'b0,rs1[11:0]} + {1'b0,offset[11:0]};
266
267      assign rs1_inc[31:12] = rs1[31:12] + 1;
268
269      assign rs1_dec[31:12] = rs1[31:12] - 1;
270
271      assign sign = offset[11];
272
273      assign dout[31:12] = ({20{ sign ^~ cout}} & rs1[31:12]) |
274      | ({20{ ~sign & cout}} & rs1_inc[31:12]) |
275      | ({20{ sign & ~cout}} & rs1_dec[31:12]);
276
277  endmodule // rvlsadder

```

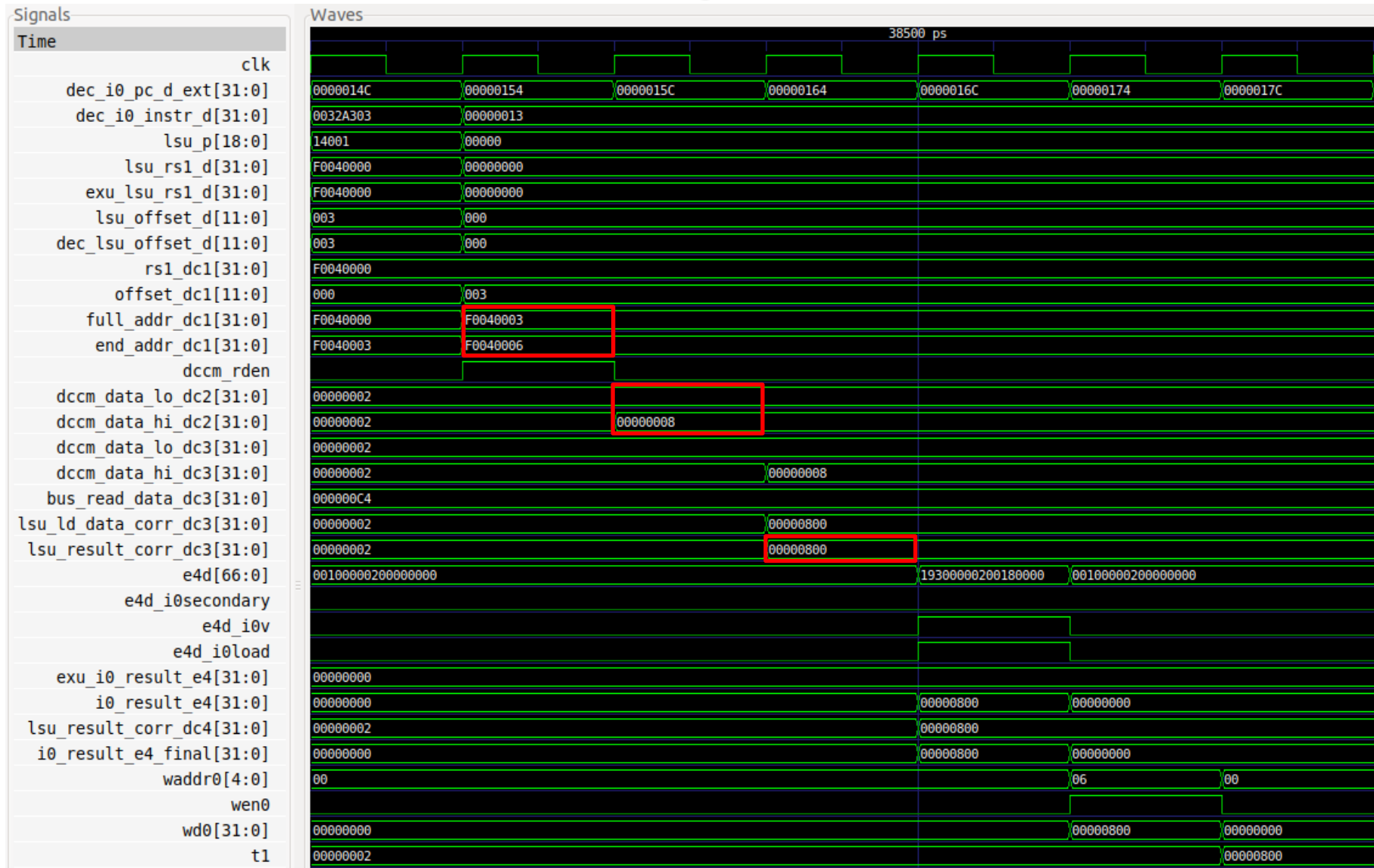
文件**lsu\_lsc\_ctl.sv**:

```

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] = rsl_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];

```

**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。



信号`lsu_addr_dc1[31:0]`和`end_addr_dc1[31:0]`的值将访问的起始地址和结束地址传达给存储器：`0xF0040003`和`0xF0040007`。读取两个字（`0x00000002`和`0x00000008`），并在对齐器中提取最后一个字（`0x00000800`）。

**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。

上文有两个仿真。

- 指向地址`0xF0040004`的`lw`

```
dccm_data_lo_dc2[31:0]: 0x00000008  
dccm_data_hi_dc2[31:0]: 0x00000008
```

两个信号都包含从请求地址读取的值。

- 指向地址`0xF0040003`的`lw`

```
dccm_data_lo_dc2[31:0]: 0x00000002 (来自地址0xF0040000的值)  
dccm_data_hi_dc2[31:0]: 0x00000008 (来自地址0xF0040004的值)
```

**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。

不提供解答。

**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。

上文有两个仿真。



- 指向地址0xF0040004的lw

`lsu_result_corr_dc3[31:0]: 0x00000008`

它包含从请求地址读取的值。

- 指向地址0xF0040003的lw

`lsu_result_corr_dc3[31:0]: 0x00000800`

它包含从请求地址读取的值。需考虑RISC-V采用小端模式。

**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。

```

80  if (DCCM_ENABLE == 1) begin: Gen_dccm_enable
81      // Start address check
82      rvrangecheck #(.CCM_SADR(`RV_DCCM_SADR),
83          .CCM_SIZE(`RV_DCCM_SIZE)) start_addr_dccm_rangecheck (
84          .addr(start_addr_dc1[31:0]),
85          .in_range(start_addr_in_dccm_dc1),
86          .in_region(start_addr_in_dccm_region_dc1)
87      );
88
89      // End address check
90      rvrangecheck #(.CCM_SADR(`RV_DCCM_SADR),
91          .CCM_SIZE(`RV_DCCM_SIZE)) end_addr_dccm_rangecheck (
92          .addr(end_addr_dc1[31:0]),
93          .in_range(end_addr_in_dccm_dc1),
94          .in_region(end_addr_in_dccm_region_dc1)
95      );
96  end else begin: Gen_dccm_disable // block: Gen_dccm_enable
97      assign start_addr_in_dccm_dc1 = '0;
98      assign start_addr_in_dccm_region_dc1 = '0;
99      assign end_addr_in_dccm_dc1 = '0;
100     assign end_addr_in_dccm_region_dc1 = '0;
101  end
102  if (ICCM_ENABLE == 1) begin : check_iccm
103      assign addr_in_iccm = (start_addr_dc1[31:28] == ICCM_REGION);
104  end
105  else begin
106      assign addr_in_iccm = 1'b0;
107  end
108  // PIC memory check
109  // Start address check
110  rvrangecheck #(.CCM_SADR(`RV_PIC_BASE_ADDR),
111      .CCM_SIZE(`RV_PIC_SIZE)) start_addr_pic_rangecheck (
112      .addr(start_addr_dc1[31:0]),
113      .in_range(start_addr_in_pic_dc1),
114      .in_region(start_addr_in_pic_region_dc1)
115  );
116
117  // End address check
118  rvrangecheck #(.CCM_SADR(`RV_PIC_BASE_ADDR),
119      .CCM_SIZE(`RV_PIC_SIZE)) end_addr_pic_rangecheck (
120      .addr(end_addr_dc1[31:0]),
121      .in_range(end_addr_in_pic_dc1),
122      .in_region(end_addr_in_pic_region_dc1)
123  );
124
125  assign addr_in_dccm_dc1      = (start_addr_in_dccm_dc1 & end_addr_in_dccm_dc1);
126  assign addr_in_pic_dc1      = (start_addr_in_pic_dc1 & end_addr_in_pic_dc1);
127
128  assign addr_external_dc1 = ~(addr_in_dccm_dc1 | addr_in_pic_dc1); //~addr_in_dccm_region_dc1;

```

模块**rvrangecheck**用于检查请求地址：

- 如果它处于DCCM/ICCM地址范围中（第80-107行），在这种情况下，信号addr\_in\_dccm\_dc1 = 1
- 如果它处于PIC地址范围中（第108-123行），在这种情况下，信号addr\_in\_pic\_dc1 = 1
- 如果它不处于上述任一地址范围中，则处于DDR外部存储器中，在这种情况下：addr\_external\_dc1 = 1

**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)。

**0x0062a023 → 0000000 00110 00101 010 00000 0100011**

**imm<sub>11:0</sub> = 000000000000**

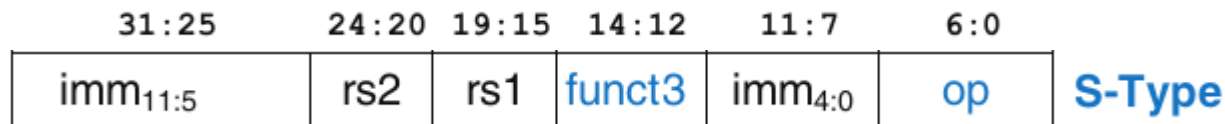
**rs2 = 00110 = x6 (t1)**

**rs1 = 00101 = x5 (t0)**

**funct3 = 010**

**op = 0100011**


来自DDCARV的附录B：



op	funct3	funct7	Type	Instruction	Description	Operation
0100011 (35)	010	–	S	sw rs2, imm(rs1)	store word	[Address] <sub>31:0</sub> = rs2

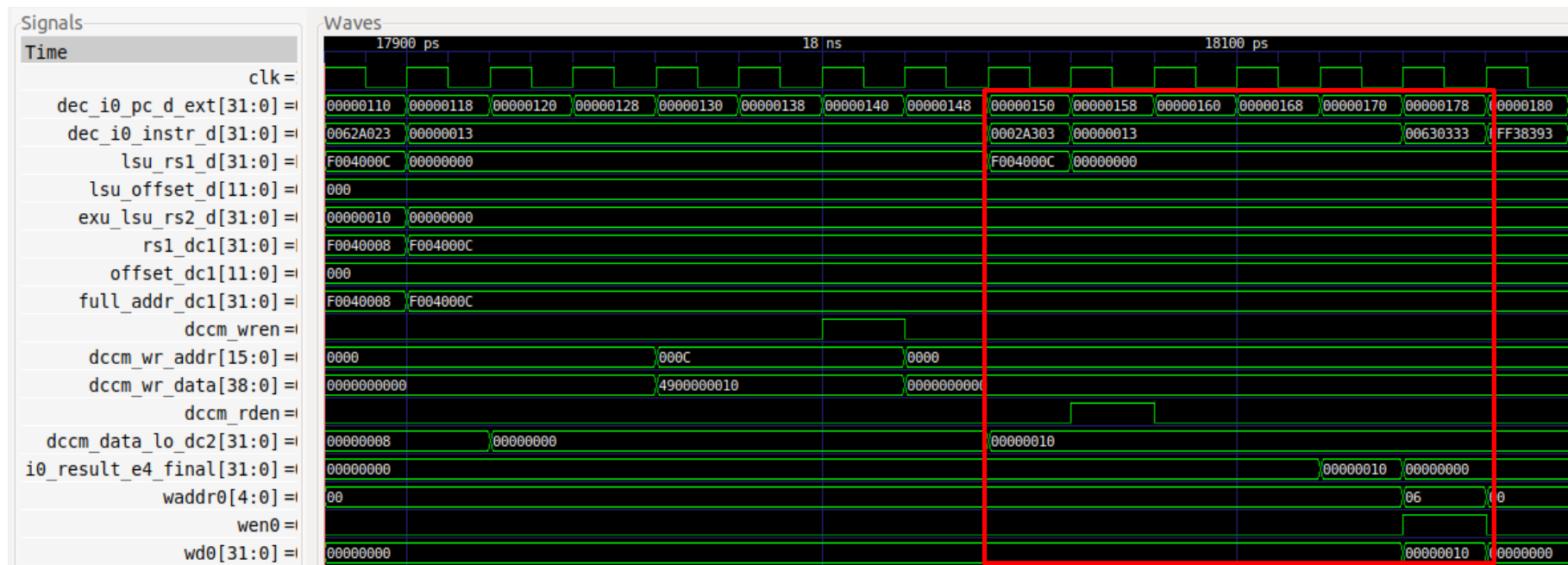
**任务：**在自己的计算机上重复图12中的仿真过程。请按照以下步骤操作（如GSG第7部分所详诉）：

- 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。

解答请参见实验13的主文档。

**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。

不提供解答。

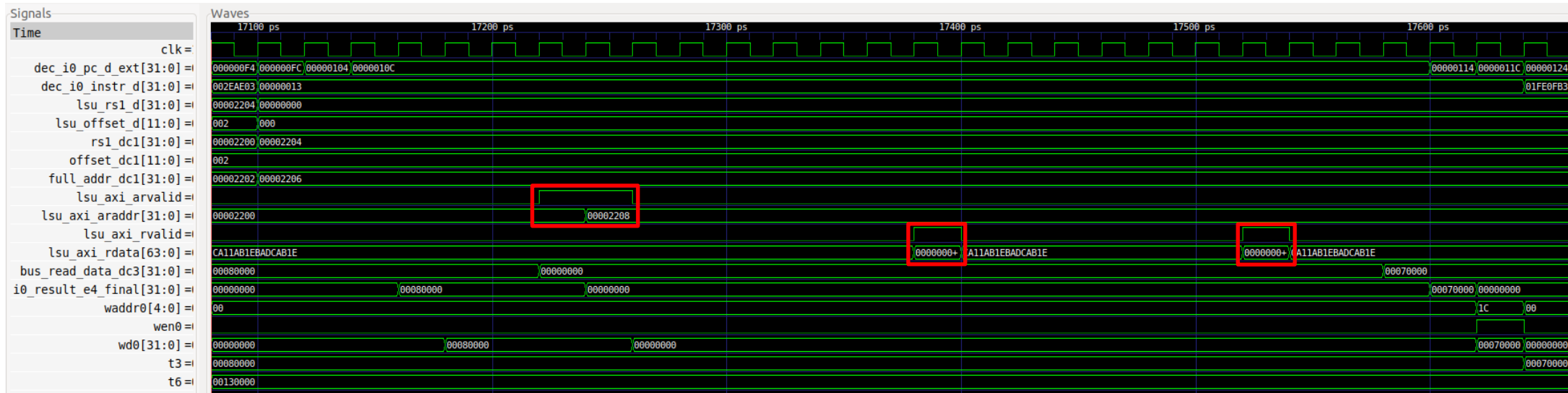
**任务：**分析针对DCCM的未对齐存储以及子字存储：存储字节（sb）或存储半字（sh）。

不提供解答。

**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。

解答请参见实验13的主文档。

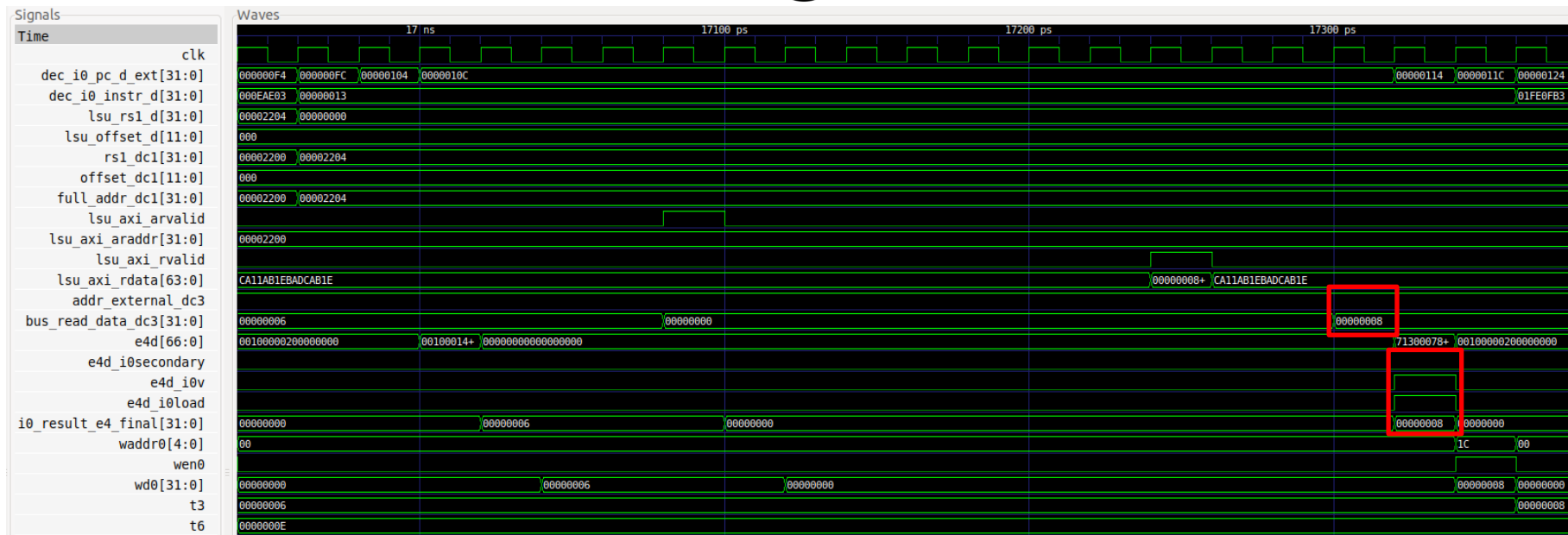
**任务：**修改图15中的程序以分析需要通过AXI总线向外部存储器发送两个地址的未对齐装载访问。



**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。

不提供解答。

**任务：** 在自己的计算机上重复图18中的仿真过程。使用文件scriptStoreBuffer.tcl（在 [RVfpgaPath]/RVfpga/Labs/Lab13/SW\_Instruction\_DCCM中提供）。单击几次“Zoom In”（放大）（）移动至17900 ps。

解答请参见实验13的主文档。

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

不提供解答。