

任務

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)。

0x0042a303 → 000000000100 00101 010 00110 0000011

imm_{11:0} = 000000000100

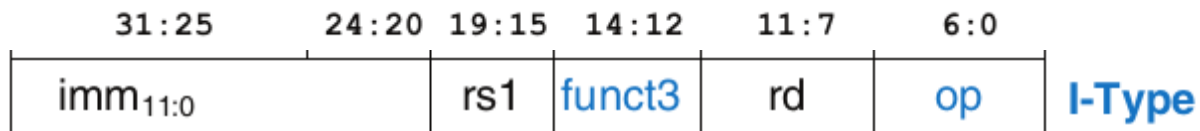
rs1 = 00101 = x5 (t0)

funct3 = 010

rd = 00110 = x6 (t1)

op = 0000011


來自DDCARV的附錄B：



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

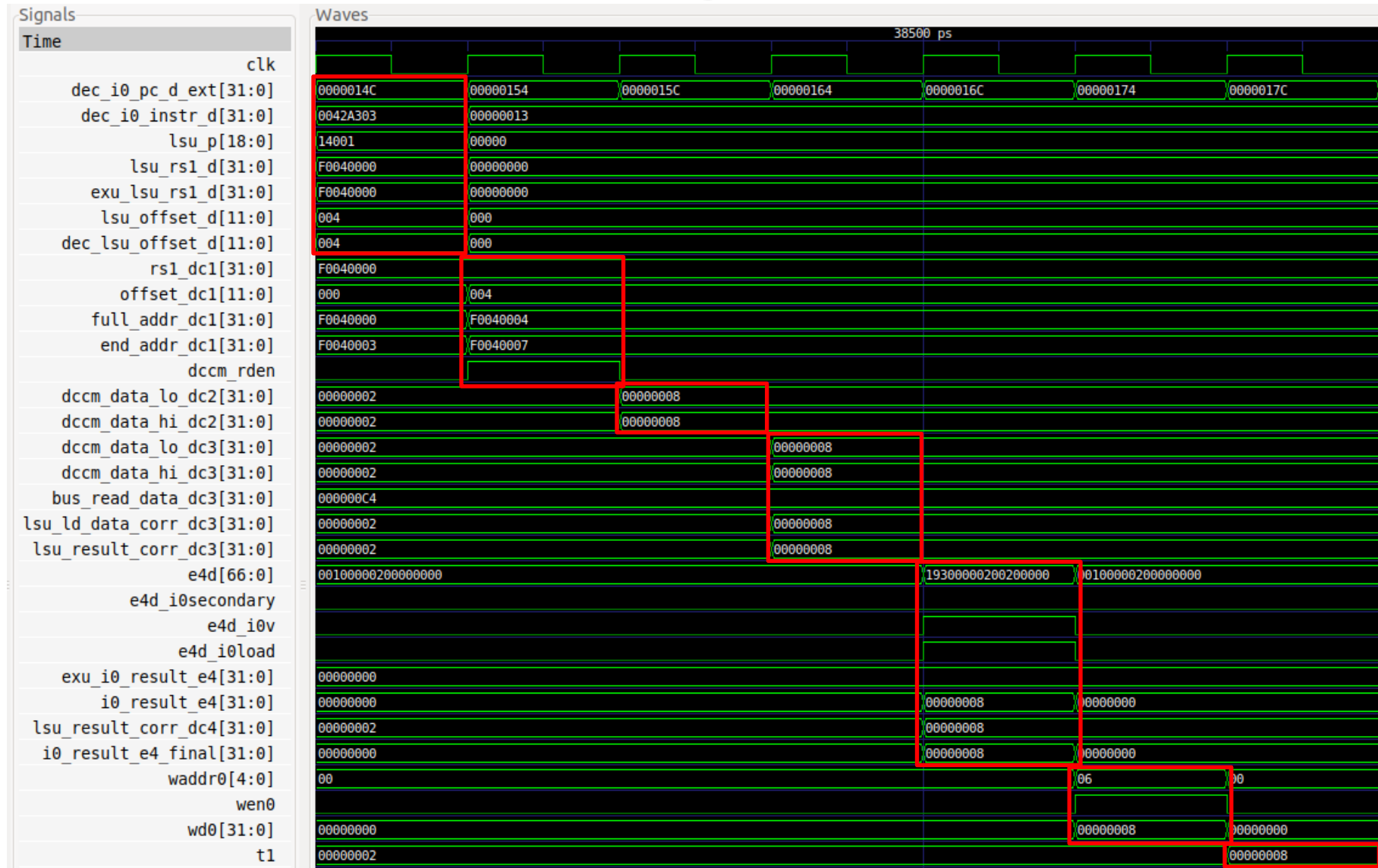
Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments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

解答請參見實驗13的主文件。

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）。



任務：在SweRV EH1處理器的Verilog檔案中找到圖6中的結構和訊號。

不提供解答。

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。

請參見上面的模擬。可以看到，當載入進行解碼時，`lsu_p = 0x14001`：

- `valid = 1`。指令有效。
- `load = 1`。指令為載入指令。
- `word = 1`。存取的大小是字。

任務：在Verilog程式碼中從LSU的兩個輸入（`exu_lsu_rsl_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**。為其他指令分析這些訊號的行為。

```

298     assign exu_lsu_rsl_d[31:0] = ({32{ ~dec_i0_rsl_bypass_en_d & dec_i0_lsu_d          }} & gpr_i0_rsl_d[31:0]      ) |
299                                ({32{ ~dec_i1_rsl_bypass_en_d & ~dec_i0_lsu_d & dec_i1_lsu_d }} & gpr_i1_rsl_d[31:0]      ) |
300                                ({32{ dec_i0_rsl_bypass_en_d & dec_i0_lsu_d          }} & i0_rsl_bypass_data_d[31:0]) |
301                                ({32{ dec_i1_rsl_bypass_en_d & ~dec_i0_lsu_d & dec_i1_lsu_d }} & i1_rsl_bypass_data_d[31:0]);
302

```

基本位址可以來自暫存器檔案或旁路（來自通路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]});
1069

```

偏移量來自通路0或通路1指令的32位元。

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。

檔案**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

```

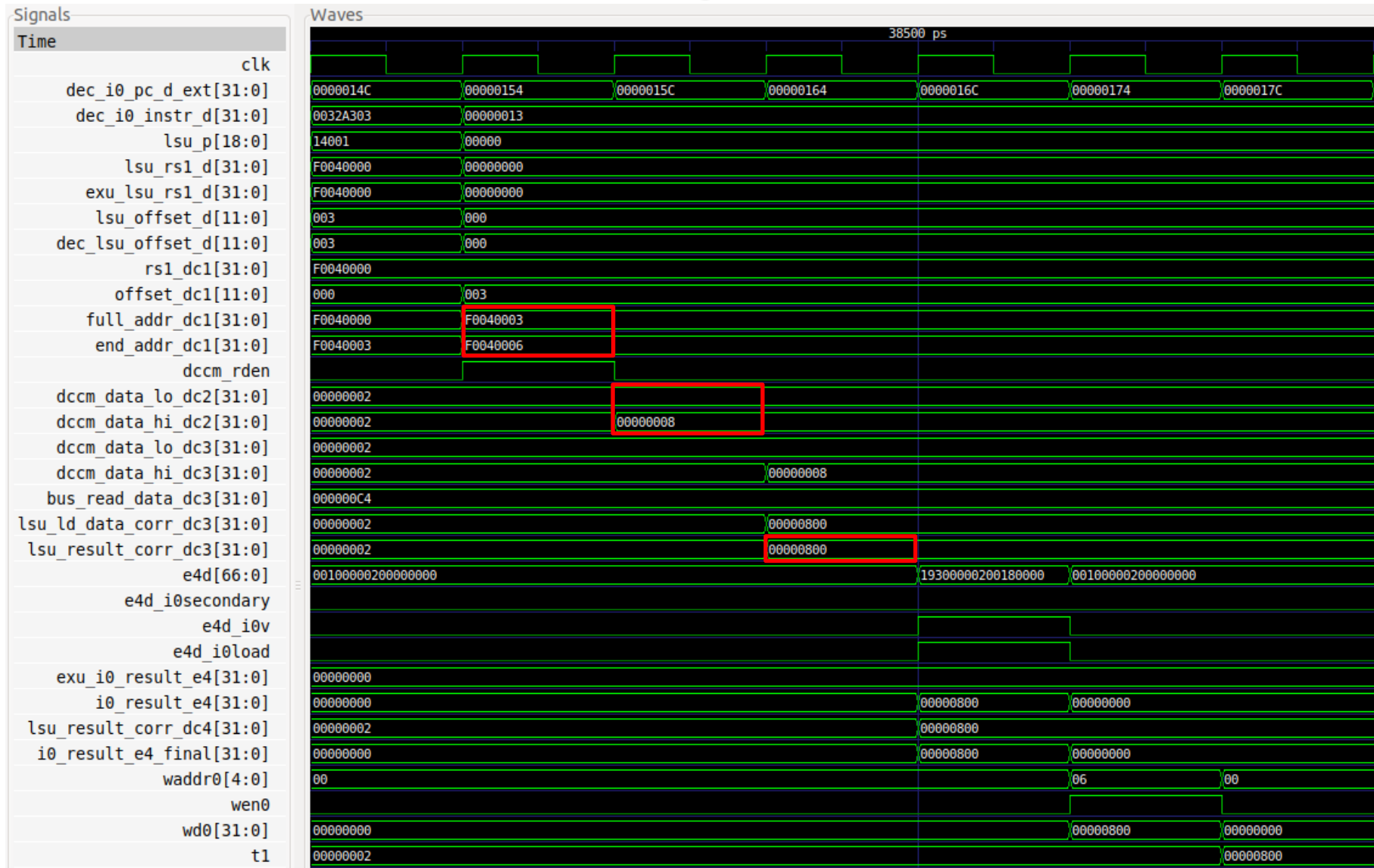
檔案**lsu_lsc_ctl.sv**：

```

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];

```

任務：在圖2的程序中，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。



訊號`lsu_addr_dc1[31:0]`和`end_addr_dc1[31:0]`的值將存取的起始位址和結束位址傳達給記憶體：`0xF0040003`和`0xF0040007`。讀取兩個字（`0x00000002`和`0x00000008`），並在對齊器中擷取最後一個字（`0x00000800`）。

任務：在圖2的程序中，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。

上文有兩個模擬。

- 指向位址`0xF0040004`的`lw`

`dccm_data_lo_dc2[31:0] : 0x00000008`

`dccm_data_hi_dc2[31:0] : 0x00000008`

兩個訊號都包含從要求位址讀取的值。

- 指向位址`0xF0040003`的`lw`

`dccm_data_lo_dc2[31:0] : 0x00000002`（來自位址`0xF0040000`的值）

`dccm_data_hi_dc2[31:0] : 0x00000008`（來自位址`0xF0040004`的值）

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。

不提供解答。

任務：在圖2的程序中，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。

上文有兩個模擬。

- 指向位址0xF0040004的lw

lsu_result_corr_dc3[31:0] : 0x00000008

它包含從要求位址讀取的值。

- 指向位址0xF0040003的lw

lsu_result_corr_dc3[31:0] : 0x00000800

它包含從要求位址讀取的值。需考慮RISC-V採用位元組由小到大模式。

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。

```

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;

```

模組**rvrangecheck**用於檢查要求位址：

- 如果它處於DCCM/ICCM位址範圍中（第80-107行），在這種情況下，訊號`addr_in_dccm_dc1 = 1`
- 如果它處於PIC位址範圍中（第108-123行），在這種情況下，訊號`addr_in_pic_dc1 = 1`
- 如果它不處於上述任一位址範圍中，則處於DDR外部記憶體中，在這種情況下：`addr_external_dc1 = 1`

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)`。

0x0062a023 → 0000000 00110 00101 010 00000 0100011

imm_{11:0} = 000000000000

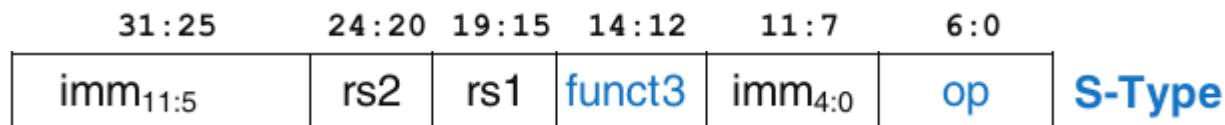
rs2 = 00110 = x6 (t1)

rs1 = 00101 = x5 (t0)

funct3 = 010

op = 0100011


來自DDCARV的附錄B：



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

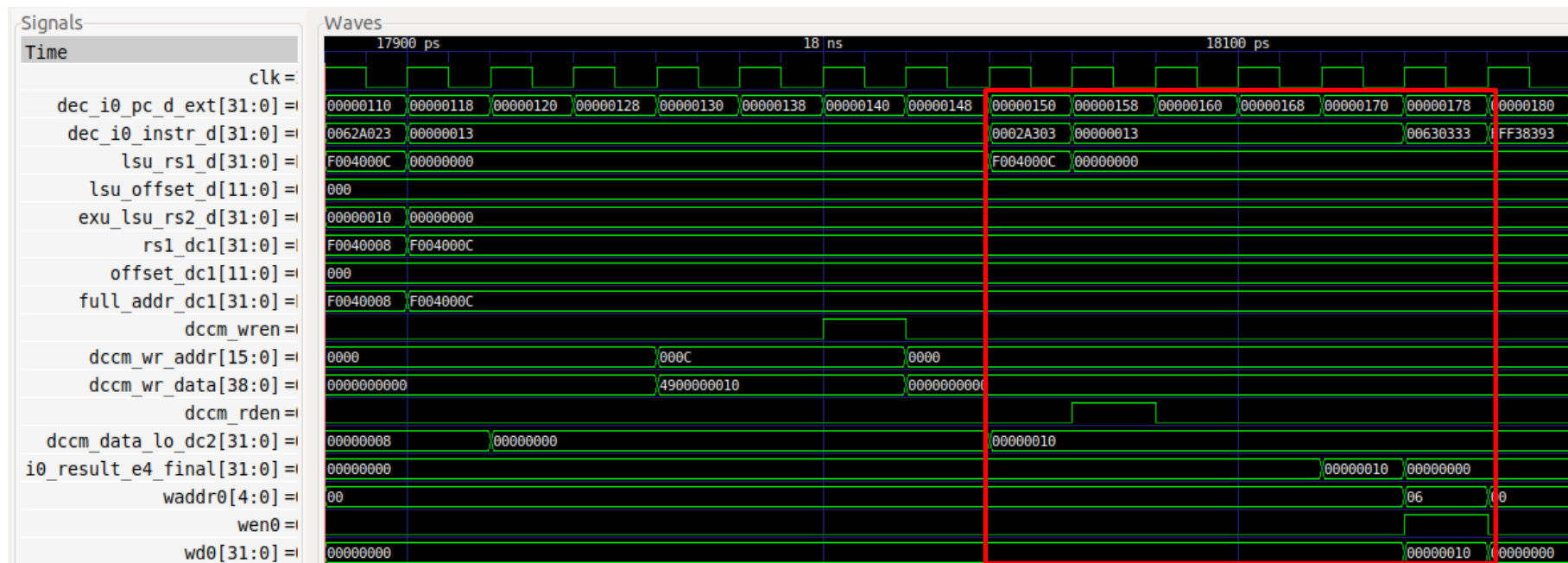
任務：在自己的電腦上重複圖12中的模擬過程。請按照以下步驟操作（如GSG第7部分所詳述）：

- 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。

解答請參見實驗13的主文件。

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。

不提供解答。

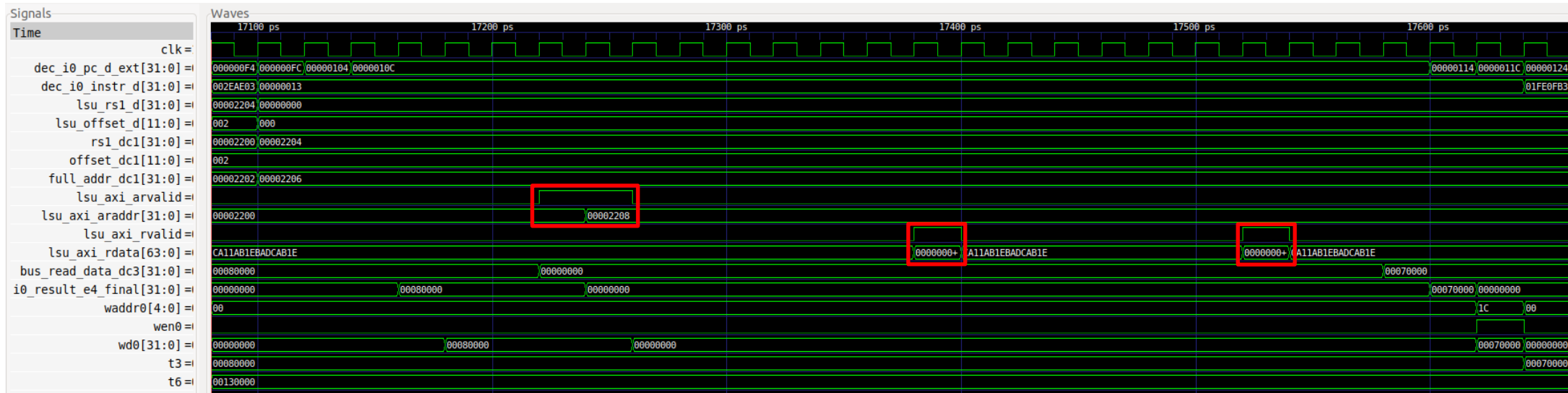
任務：分析針對DCCM的未對齊儲存以及子字儲存：儲存位元組（sb）或儲存半字（sh）。

不提供解答。

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。

解答請參見實驗13的主文件。

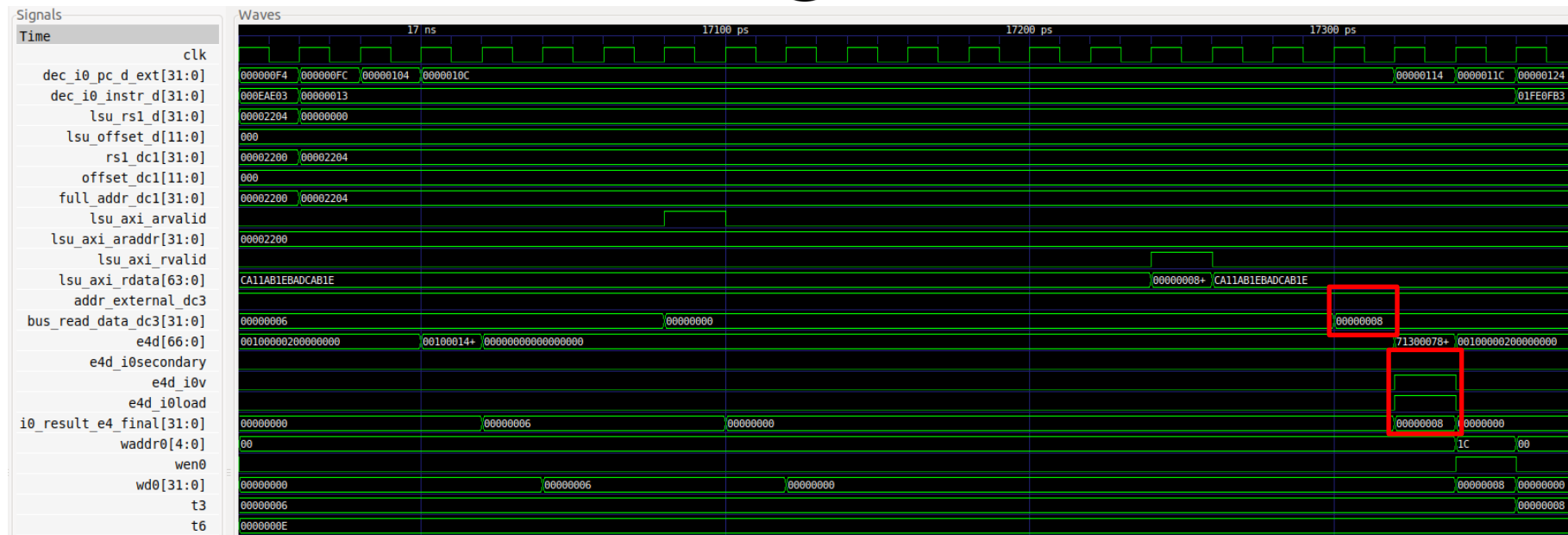
任務：修改圖15中的程序以分析需要透過AXI匯流排向外部記憶體傳送兩個位址的未對齊載入存取。



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。

不提供解答。

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。

解答請參見實驗13的主文件。

任務：修改圖11中的程序以實作兩個出色的儲存操作，並執行與圖18中的分析類似的分析。

不提供解答。