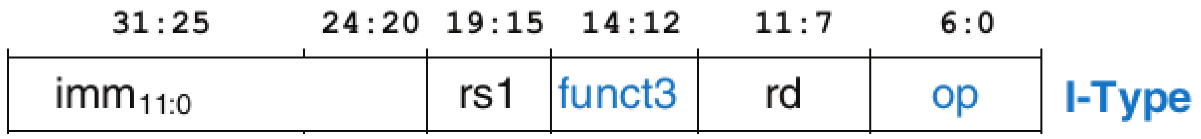
TASKS

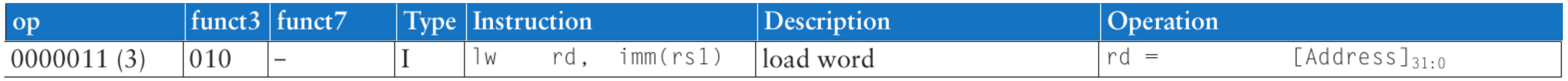
TASK: Verify that these 32 bits (0x0042a303) correspond to instruction `lw t1,4(t0)` in the RISC-V architecture.

0x0042a303 → 000000000100 00101 010 00110 0000011

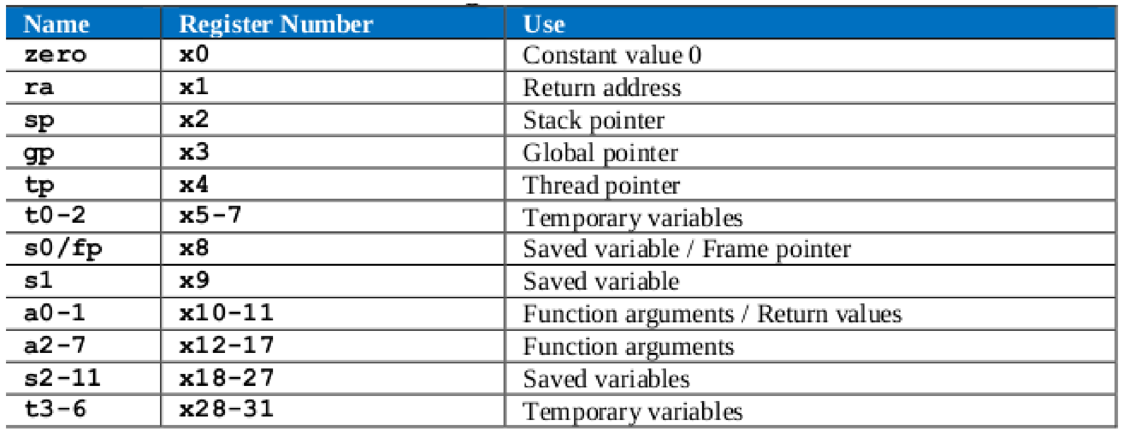
imm$_{11:0}$ = 000000000100
rs1 = 00101 = x5 (t0)
funct3 = 010
rd = 00110 = x6 (t1)
op = 0000011

From Appendix B of DDCARV:





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



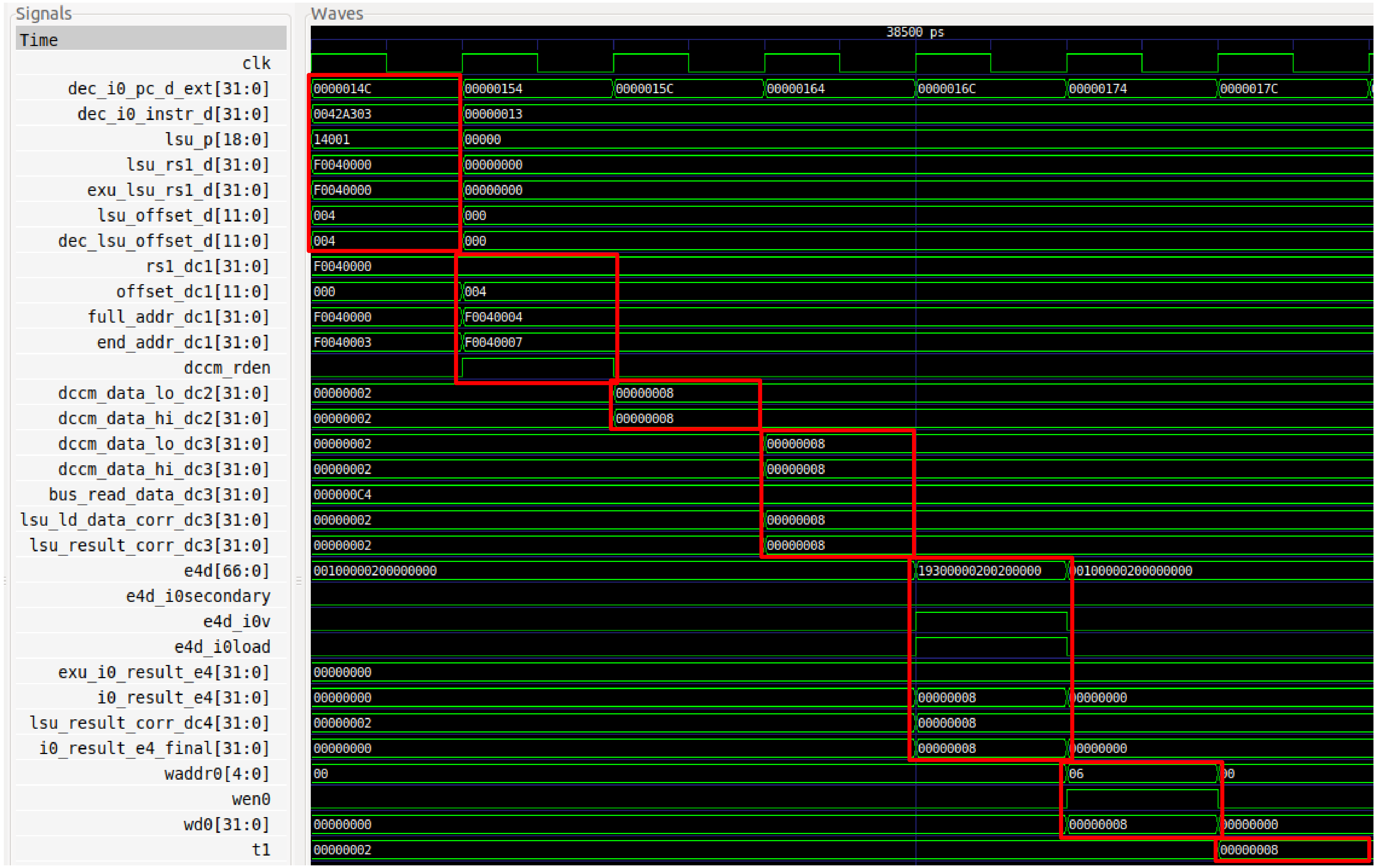
Name	Register Number	Use
`zero`	`x0`	Constant value 0
`ra`	`x1`	Return address
`sp`	`x2`	Stack pointer
`gp`	`x3`	Global pointer
`tp`	`x4`	Thread pointer
`t0-2`	`x5-7`	Temporary variables
`s0/fp`	`x8`	Saved variable / Frame pointer
`s1`	`x9`	Saved variable
`a0-1`	`x10-11`	Function arguments / Return values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

TASK: Replicate the simulation from Figure 4 on your own computer. Follow the next steps (as described in detail in Section 7 of the GSG):

- If necessary, generate the simulation binary (*Vrvfpgasim*).
- In PlatformIO, open the project provided at: *[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM*.
- Correct the path to the RVfpga simulation binary (*Vrvfpgasim*) in file *platformio.ini*.
- Generate the simulation trace with Verilator (Generate Trace).
- Open the trace using GTKWave.
- Use file *scriptLoad.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM/*) to open the same signals as the ones shown in Figure 4. For that purpose, on GTKWave, click on *File → Read Tcl Script File* and select the *scriptLoad.tcl* file.
- Click on *Zoom In* (⊞) several times and move to 18600ps.

Solution provided in main document of Lab 13.

TASK: Extend the simulation from Figure 4 to include the signals shown in Figure 6, which are explained below.


Signals
Waves
Time
38500 ps
clk
dec_i0_pc_d_ext[31:0] 0000014C 00000154 0000015C 00000164 0000016C 00000174 0000017C
dec_i0_instr_d[31:0] 0042A303 00000013
lsu_p[18:0] 14001 00000
lsu_rs1_d[31:0] F0040000 00000000
exu_lsu_rs1_d[31:0] F0040000 00000000
lsu_offset_d[11:0] 004 000
dec_lsu_offset_d[11:0] 004 000
rs1_dc1[31:0] F0040000
offset_dc1[11:0] 000 004
full_addr_dc1[31:0] F0040000 F0040004
end_addr_dc1[31:0] F0040003 F0040007
dccm_rden
dccm_data_lo_dc2[31:0] 00000002 00000008
dccm_data_hi_dc2[31:0] 00000002 00000008
dccm_data_lo_dc3[31:0] 00000002 00000008
dccm_data_hi_dc3[31:0] 00000002 00000008
bus_read_data_dc3[31:0] 000000C4
lsu_ld_data_corr_dc3[31:0] 00000002 00000008
lsu_result_corr_dc3[31:0] 00000002 00000008
e4d[66:0] 00100000200000000 19300000200200000 00100000200000000
e4d_i0secondary
e4d_i0v
e4d_i0load
exu_i0_result_e4[31:0] 00000000
i0_result_e4[31:0] 00000000 00000008 00000000
lsu_result_corr_dc4[31:0] 00000002 00000008
i0_result_e4_final[31:0] 00000000 00000008 00000000
waddr0[4:0] 00 06 00
wen0
wd0[31:0] 00000000 00000008 00000000
t1 00000002 00000008

<u>TASK</u>: Locate the structures and signals from Figure 6 in the Verilog files of the SweRV EH1 processor.

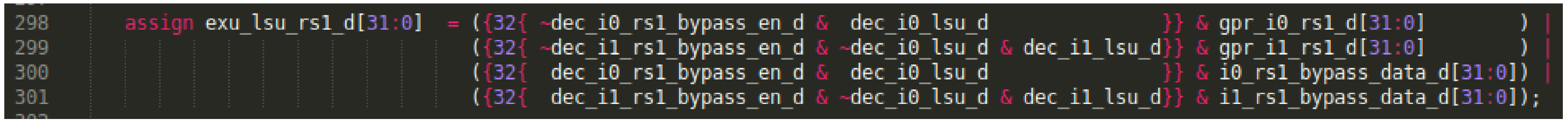
Solution not provided.

<u>TASK</u>: Include signal `lsu_p` in the simulation from Figure 4 and analyse its bits according to this description.

See the simulation above. We can see that when the load is decoded `lsu_p` = 0x14001:

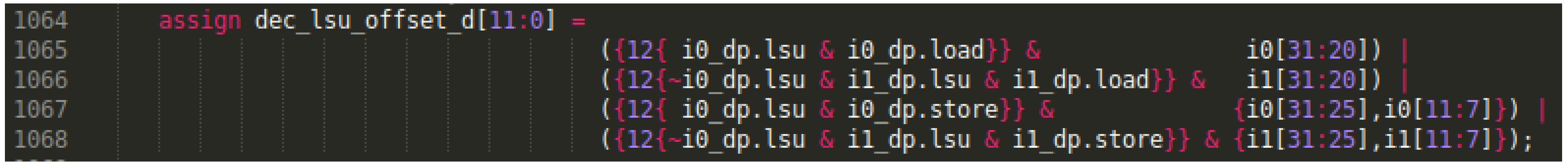
- `valid` = 1. The instruction is valid.
- `load` = 1. It is a load.
- `word` = 1. The size of the access is word.

<u>TASK</u>: Analyse in the Verilog code the path followed by the two inputs to the LSU (`exu_lsu_rs1_d` and `dec_lsu_offset_d`) from the sources where they are obtained. Several modules are involved in this process: **dec**, **exu**, **lsu**. Analyse the behaviour of these signals for other instructions.



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

The base address can come from the Register File or from the Bypass, either from Way-0 or Way-1.

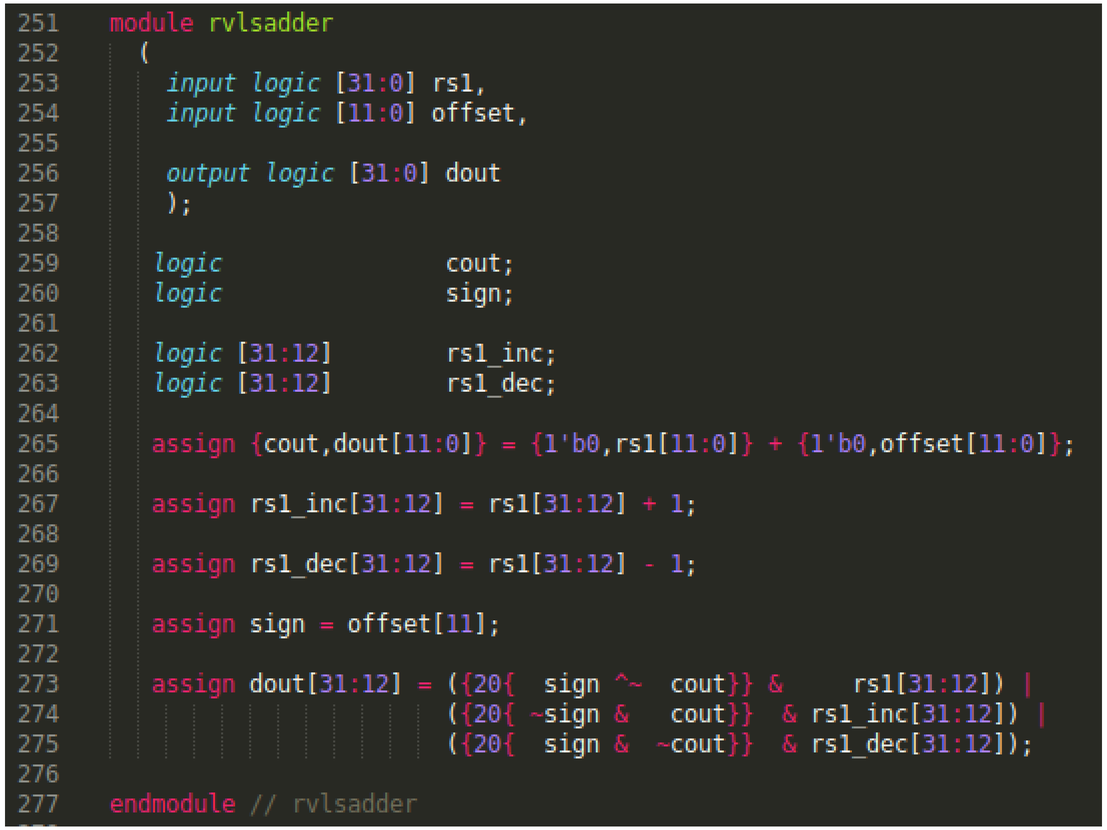


```
   assign dec_lsu_offset_d[11:0] =
                                    ({12{ i0_dp.lsu & i0_dp.load}} &               i0[31:20]) |
                                    ({12{~i0_dp.lsu & i1_dp.lsu & i1_dp.load}} &   i1[31:20]) |
                                    ({12{ i0_dp.lsu & i0_dp.store}} &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

The offset comes from the 32 bits of the instruction at Way-0 or Way-1.

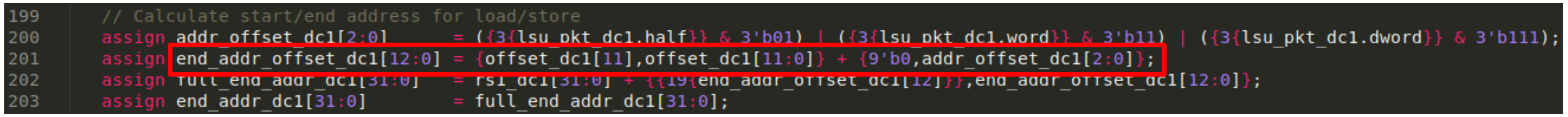
TASK: Analyse the implementation of the two adders from the DC1 stage, which are instantiated in module **lsu_lsc_ctl**. We provide guidance in Figure 7 below by showing the implementation of these adders.

File *beh_lib.sv*:



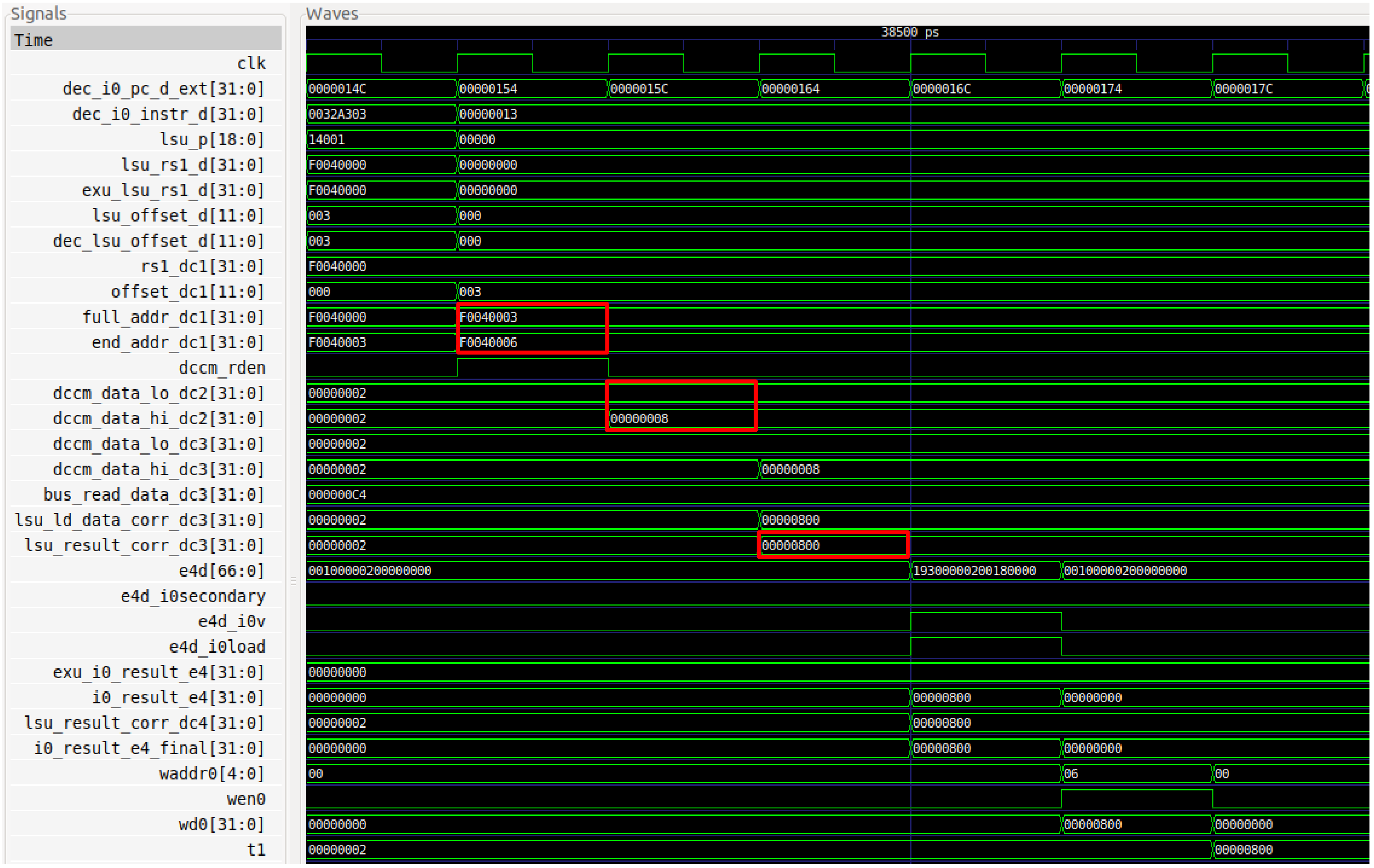
```
module rvlsadder
  (
    input logic [31:0] rs1,
    input logic [11:0] offset,

    output logic [31:0] dout
    );

   logic                 cout;
   logic                 sign;

   logic [31:12]         rs1_inc;
   logic [31:12]         rs1_dec;

   assign {cout,dout[11:0]} = {1'b0,rs1[11:0]} + {1'b0,offset[11:0]};

   assign rs1_inc[31:12] = rs1[31:12] + 1;

   assign rs1_dec[31:12] = rs1[31:12] - 1;

   assign sign = offset[11];

   assign dout[31:12] = ({20{  sign ^~  cout}} &     rs1[31:12]) |
                        ({20{ ~sign &   cout}}  & rs1_inc[31:12]) |
                        ({20{  sign &  ~cout}}  & rs1_dec[31:12]);

endmodule // rvlsadder
```

File *lsu_lsc_ctl.sv*:



```
    // Calculate start/end address for load/store
    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);
    assign end_addr_offset_dc1[12:0] = {offset_dc1[11],offset_dc1[11:0]} + {9'b0,addr_offset_dc1[2:0]};
    assign full_end_addr_dc1[31:0]   = rs1_dc1[31:0] + {{19{end_addr_offset_dc1[12]}},end_addr_offset_dc1[12:0]};
    assign end_addr_dc1[31:0]         = full_end_addr_dc1[31:0];
```

TASK: In the program from Figure 2, try different access sizes (`byte`, `half-word`) and unaligned accesses. To do so, change the offset or the access type from `lw` to `lb` (*load byte*) or `lh` (*load half-word*). For example, if you change the offset from 4 to 3, the load word instruction performs an unaligned access to the 32-bits starting at address 0xF0040003, as shown in Figure 8. Analyse the value of signals `lsu_addr_dc1[31:0]` (or `full_addr_dc1[31:0]`) and `end_addr_dc1[31:0]` under these different situations.
In Lab 20 we analyse this situation from the internals of the DCCM.


Signals
Waves
Time
38500 ps
clk
dec_i0_pc_d_ext[31:0] 0000014C 00000154 0000015C 00000164 0000016C 00000174 0000017C
dec_i0_instr_d[31:0] 0032A303 00000013
lsu_p[18:0] 14001 00000
lsu_rs1_d[31:0] F0040000 00000000
exu_lsu_rs1_d[31:0] F0040000 00000000
lsu_offset_d[11:0] 003 000
dec_lsu_offset_d[11:0] 003 000
rs1_dc1[31:0] F0040000
offset_dc1[11:0] 000 003
full_addr_dc1[31:0] F0040000 F0040003
end_addr_dc1[31:0] F0040003 F0040006
dccm_rden
dccm_data_lo_dc2[31:0] 00000002
dccm_data_hi_dc2[31:0] 00000002 00000008
dccm_data_lo_dc3[31:0] 00000002
dccm_data_hi_dc3[31:0] 00000002 00000008
bus_read_data_dc3[31:0] 000000C4
lsu_ld_data_corr_dc3[31:0] 00000002 00000800
lsu_result_corr_dc3[31:0] 00000002 00000800
e4d[66:0] 00100000200000000 19300000200180000 00100000200000000
e4d_i0secondary
e4d_i0v
e4d_i0load
exu_i0_result_e4[31:0] 00000000
i0_result_e4[31:0] 00000000 00000800 00000000
lsu_result_corr_dc4[31:0] 00000002 00000800
i0_result_e4_final[31:0] 00000000 00000800 00000000
waddr0[4:0] 00 06 00
wen0
wd0[31:0] 00000000 00000800 00000000
t1 00000002 00000800

The values of signals `lsu_addr_dc1[31:0]` and `end_addr_dc1[31:0]` communicate to Memory the starting and final address of the access: 0xF0040003 and 0xF0040007. Two words are read (0x00000002 and 0x00000008) and the final word is extracted in the Aligner (0x00000800).

TASK: In the program from Figure 2, compare the value of signals `dccm_data_lo_dc2[31:0]` and `dccm_data_hi_dc2[31:0]` when doing a `lw` to address 0xF0040004 and to address 0xF0040003.

Above you can see the two simulations.

- `lw` to address 0xF0040004

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

  Both signals contain the value read from the requested address.

- `lw` to address 0xF0040003

  `dccm_data_lo_dc2[31:0]`: 0x00000002 (value from address 0xF0040000)
  `dccm_data_hi_dc2[31:0]`: 0x00000008 (value from address 0xF0040004)

TASK: Analyse the Align, Merge, and Error Check logic used in the Verilog code in modules **lsu_dccm_ctl** and **lsu_ecc**.

Solution not provided.

TASK: In the program from Figure 2, compare the value of signal `lsu_result_corr_dc3[31:0]` when doing a `lw` to address 0xF0040004 and to address 0xF0040003.

Above you can see the two simulations.

- `lw` to address 0xF0040004

  `lsu_result_corr_dc3[31:0]`: 0x00000008

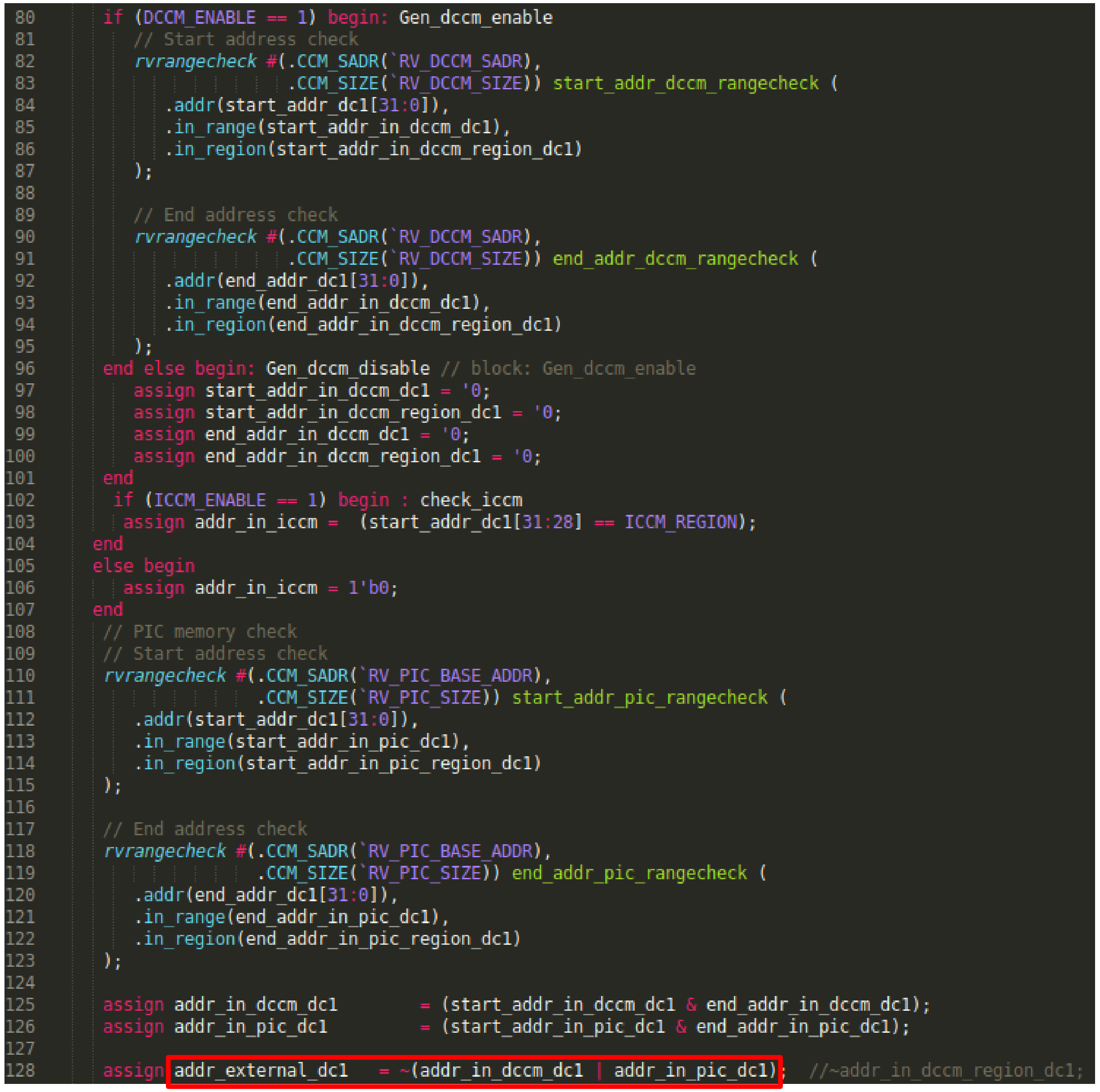
  It contains the value read from the requested address.

- `lw` to address 0xF0040003

  `lsu_result_corr_dc3[31:0]`: 0x00000800

  It contains the value read from the requested address. Take into account that RISC-V is little-endian.

TASK: Analyse in the Verilog code how signal `addr_external_dc1` was computed in the DC1 stage in module **lsu_addrcheck**.



```
 if (DCCM_ENABLE == 1) begin: Gen_dccm_enable
    // Start address check
    rvrangecheck #(.CCM_SADR(`RV_DCCM_SADR),
                   .CCM_SIZE(`RV_DCCM_SIZE)) start_addr_dccm_rangecheck (
       .addr(start_addr_dc1[31:0]),
       .in_range(start_addr_in_dccm_dc1),
       .in_region(start_addr_in_dccm_region_dc1)
    );

    // End address check
    rvrangecheck #(.CCM_SADR(`RV_DCCM_SADR),
                   .CCM_SIZE(`RV_DCCM_SIZE)) end_addr_dccm_rangecheck (
       .addr(end_addr_dc1[31:0]),
       .in_range(end_addr_in_dccm_dc1),
       .in_region(end_addr_in_dccm_region_dc1)
    );
 end else begin: Gen_dccm_disable // block: Gen_dccm_enable
    assign start_addr_in_dccm_dc1 = '0;
    assign start_addr_in_dccm_region_dc1 = '0;
    assign end_addr_in_dccm_dc1 = '0;
    assign end_addr_in_dccm_region_dc1 = '0;
 end
  if (ICCM_ENABLE == 1) begin : check_iccm
   assign addr_in_iccm =  (start_addr_dc1[31:28] == ICCM_REGION);
end
else begin
   assign addr_in_iccm = 1'b0;
end
 // PIC memory check
 // Start address check
 rvrangecheck #(.CCM_SADR(`RV_PIC_BASE_ADDR),
                .CCM_SIZE(`RV_PIC_SIZE)) start_addr_pic_rangecheck (
    .addr(start_addr_dc1[31:0]),
    .in_range(start_addr_in_pic_dc1),
    .in_region(start_addr_in_pic_region_dc1)
 );

 // End address check
 rvrangecheck #(.CCM_SADR(`RV_PIC_BASE_ADDR),
                .CCM_SIZE(`RV_PIC_SIZE)) end_addr_pic_rangecheck (
    .addr(end_addr_dc1[31:0]),
    .in_range(end_addr_in_pic_dc1),
    .in_region(end_addr_in_pic_region_dc1)
 );

 assign addr_in_dccm_dc1        = (start_addr_in_dccm_dc1 & end_addr_in_dccm_dc1);
 assign addr_in_pic_dc1         = (start_addr_in_pic_dc1 & end_addr_in_pic_dc1);

 assign addr_external_dc1    = ~(addr_in_dccm_dc1 | addr_in_pic_dc1);  //~addr_in_dccm_region_dc1;
```

Module **rvrangecheck** is used to check the requested address:

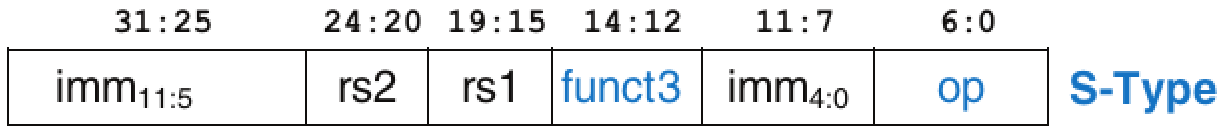
- If it is within the DCCM/ICCM address range (lines 80-107), in which case signal `addr_in_dccm_dc1` = 1
- If it is within the PIC address range (lines 108-123), in which case signal `addr_in_pic_dc1` = 1
- If it is not in any of these address ranges, then it is at the DDR External Memory, in which case: `addr_external_dc1` = 1

<u>TASK</u>: Verify that these 32 bits (0x0062a023) correspond to instruction `sw t1,0(t0)` in the RISC-V architecture.

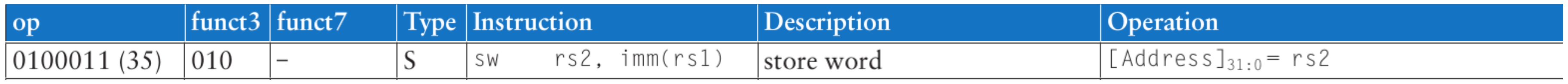
0x0062a023 → 0000000 00110 00101 010 00000 0100011

$imm_{11:0}$ = 000000000000
rs2 = 00110 = x6 (t1)
rs1 = 00101 = x5 (t0)
funct3 = 010
op = 0100011

From Appendix B of DDCARV:



31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type



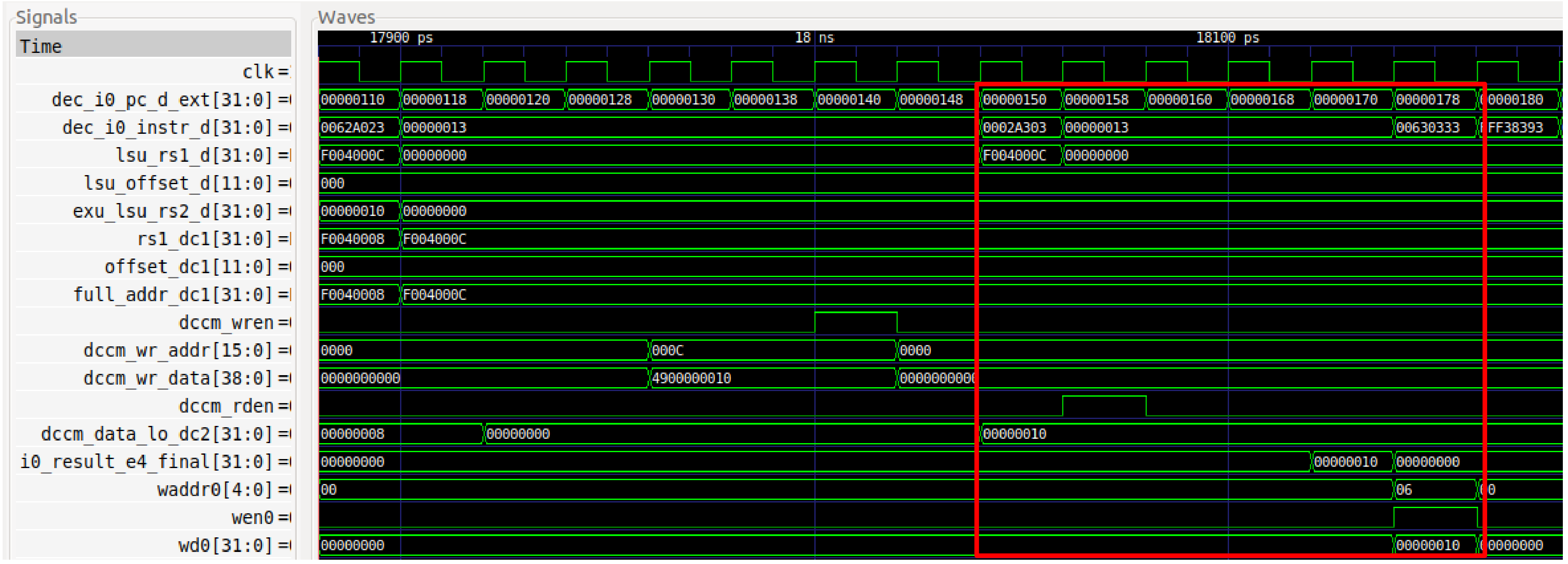
op	funct3	funct7	Type	Instruction	Description	Operation
0100011 (35)	010	–	S	`sw rs2, imm(rs1)`	store word	$[Address]_{31:0}$ = rs2

<u>TASK</u>: Replicate the simulation from Figure 12 on your own computer. Follow the next steps (as described in detail in Section 7 of the GSG):
- If necessary, generate the simulation binary (*Vrvfpgasim*).

- Open in PlatformIO the project provided at: *[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM*.
- Update the path to the RVfpga simulation binary (*Vrvfpgasim*) in file *platformio.ini*.
- Generate the simulation trace with Verilator (Generate Trace).
- Open the trace on GTKWave.
- Use file *scriptStore.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM/*) to display the same signals as the ones shown in Figure 4. For that purpose, in GTKWave, click on *File → Read Tcl Script File* and select the *scriptStore.tcl* file.
- Click on *Zoom In* ( ) several times and move to 17900ps.

Solution provided in the main document of Lab 13.

TASK: Analyse in the simulation the load instruction that follows the store to verify that the value has been correctly written to the DCCM. You will need to add some of the signals from Figure 4 and Figure 6 to analyse the load.



TASK: Extend the basic analysis performed in this section for the `sw` instruction in a similar way as the advanced analysis performed for the `lw` instruction in Section 2.B.

Solution not provided.

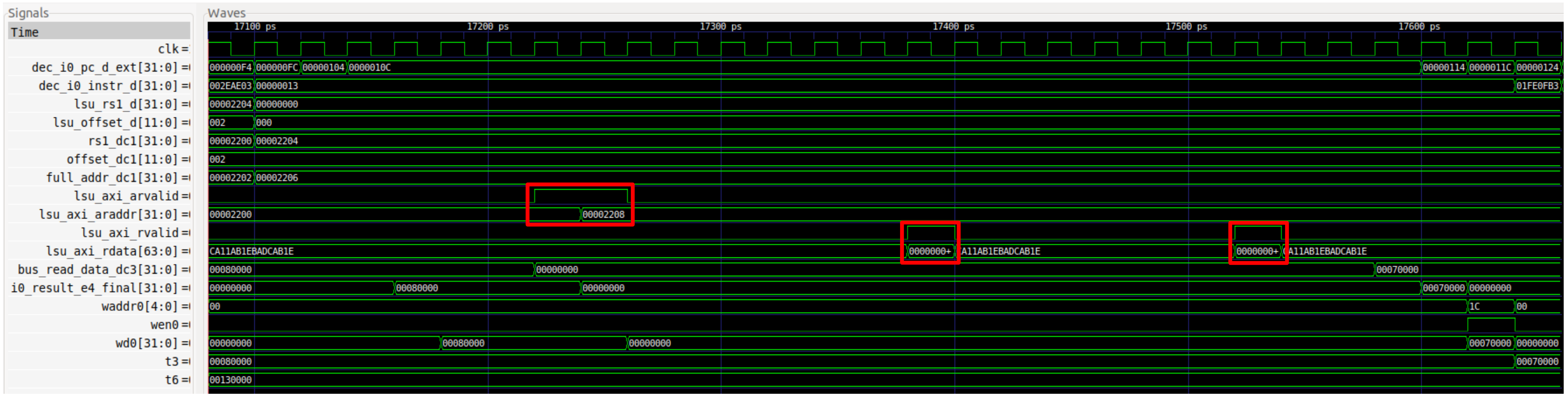
TASK: Analyse unaligned stores to the DCCM, as well as sub-word stores: store byte (`sb`) or *store half-word* (`sh`).

Solution not provided.

TASK: Replicate the simulation from Figure 17 on your own computer. Use file *test_Blocking.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory*). *Zoom In* (⊞) several times and move to 16940ps.

Solution provided in the main document of Lab 13.

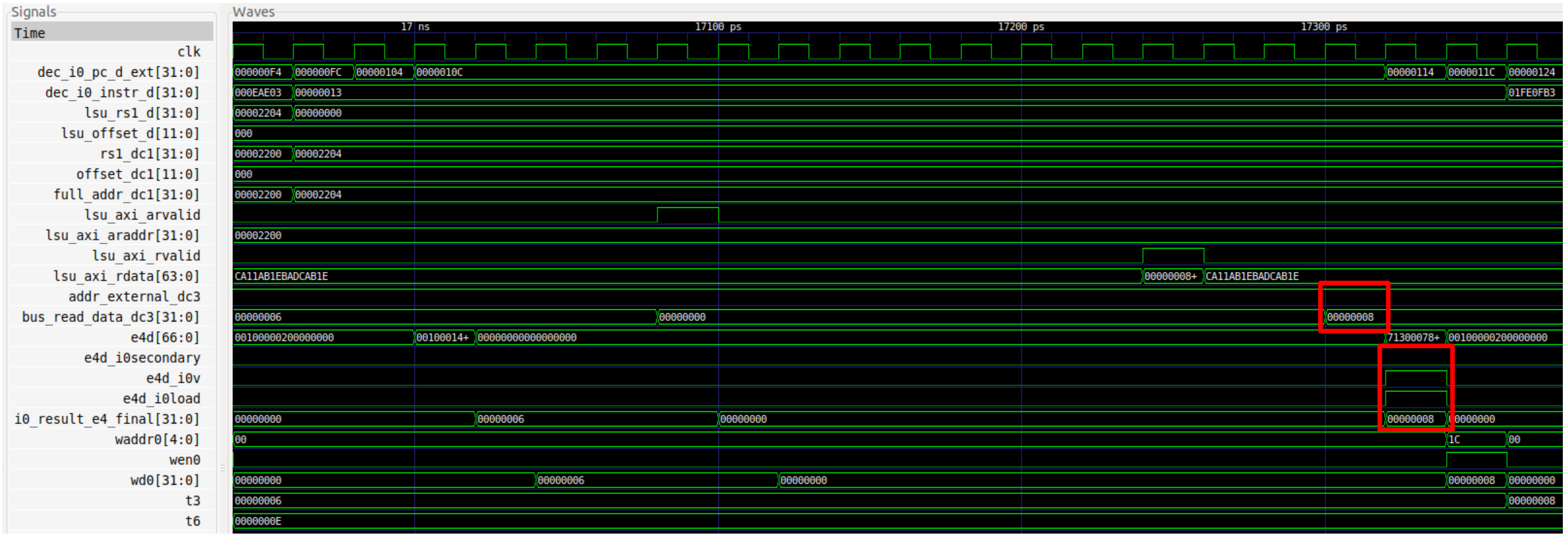
TASK: Modify the program from Figure 15 in order to analyse an unaligned load access that needs to send two addresses to the External Memory through the AXI Bus.



TASK: Add to the simulation the signals that control the multiplexers (in the DC3 and Commit stages in Figure 16) that select the data provided by the DDR External Memory. You can find these multiplexers at the following lines of the Verilog code:

- 2:1 Multiplexer: Line 264 of module **lsu_lsc_ctl**.
- 3:1 Multiplexer: Line 2277 of module **dec_decode_ctl**.

A *.tcl* file that you can use is provided at: *[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory/test_Blocking_Extended.tcl*



TASK: It can also be interesting to analyse the AXI Bus implementation for accessing the DRAM Controller, for which you can inspect the **lsu_bus_intf** module.

Solution not provided.

TASK: Replicate the simulation from Figure 18 on your own computer. Use file *scriptStoreBuffer.tcl* (provided at *[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM*). *Zoom In* ( ) several times and move to 17900ps.

Solution provided in the main document of Lab 13.

TASK: Modify the program from Figure 11 in order to have two outstanding stores and perform a similar analysis to the one from Figure

18.

Solution not provided.