

TAREFAS

TAREFA: Verificar se esses 32 bits (0x0042a303) correspondem à instrução `lw t1, 4(t0)` na arquitetura RISC-V.

0x0042a303 → 000000000100 00101 010 00110 0000011

imm_{11:0} = 000000000100

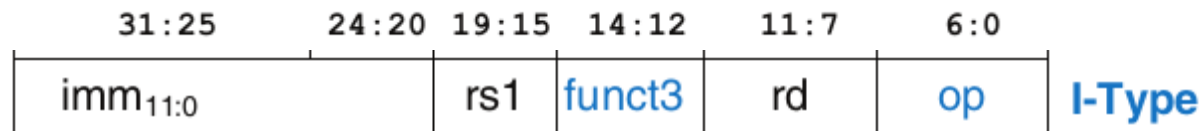
rs1 = 00101 = x5 (t0)

funct3 = 010

rd = 00110 = x6 (t1)

op = 0000011


Do Apêndice B do DDCARV:



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

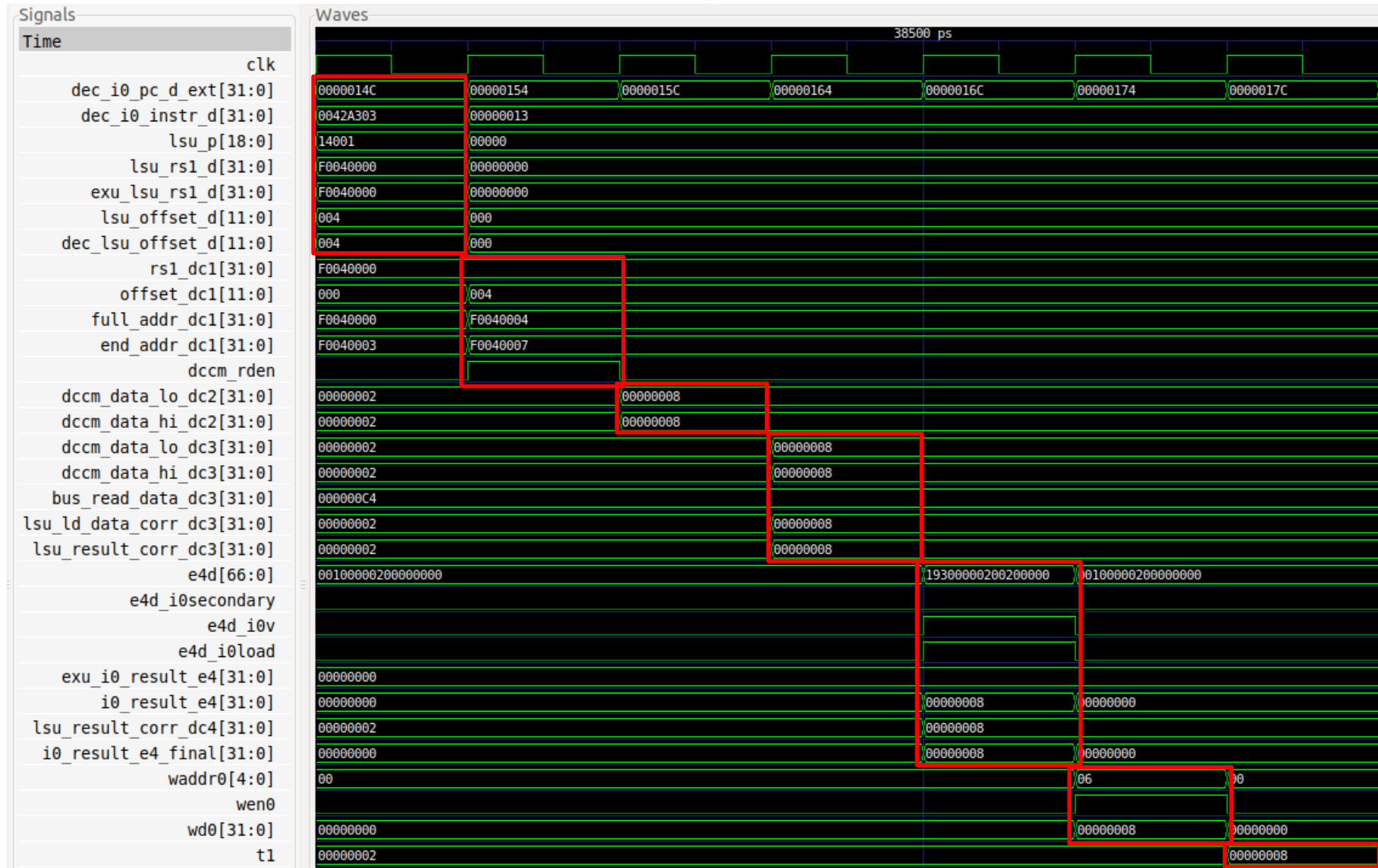
Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporary variables
s0/fp	x8	Saved variable / Frame pointer
s1	x9	Saved variable
a0-1	x10-11	Function arguments / Return values
a2-7	x12-17	Function arguments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

TAREFA: Replicar a simulação da Figura 4 no seu computador. Siga as próximas etapas (conforme descrito em detalhes na Seção 7 do GSG):

- Se necessário, gere o binário de simulação (*Vrvfpgasim*).
- No PlatformIO, abra o projeto fornecido em: *[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM*.
- Corrija o caminho para o binário de simulação do RVfpga (*Vrvfpgasim*) no ficheiro *platformio.ini*.
- Gere o trace da simulação com o Verilator (Generate Trace).
- Abra o trace usando o GTKWave.
- Use o ficheiro *scriptLoad.tcl* (fornecido em *[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_DCCM*) para abrir os mesmos sinais que os mostrados na Figura 4. Para isso, no GTKWave, clique em *File → Read Tcl Script File* e selecione o ficheiro *scriptLoad.tcl*.
- Clique em *Zoom In* () várias vezes e vá para 18600ps.

Solução fornecida no documento principal do Lab 13.

TAREFA: Amplie a simulação da Figura 4 para incluir os sinais mostrados na Figura 6, que são explicados abaixo.



TAREFA: Localize as estruturas e os sinais da Figura 6 nos ficheiros Verilog do processador SweRV EH1.

Solução não fornecida.

TAREFA: Inclua o sinal `lsu_p` na simulação da Figura 4 e analise seus bits de acordo com esta descrição.

Veja a simulação acima. Podemos ver que, quando a leitura é decodificada, `lsu_p = 0x14001`:

- `valid = 1`. A instrução é válida.
- `leitura = 1`. É uma leitura.
- `palavra = 1`. O tamanho do acesso é palavra.

TAREFA: Analisar no código Verilog o caminho seguido pelas duas entradas para a LSU (`exu_lsu_rsl_d` e `dec_lsu_offset_d`) a partir das fontes onde são obtidas. Vários módulos estão envolvidos nesse processo: **dec**, **exu**, **lsu**. Analise o comportamento desses sinais para obter outras instruções.

```
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]);
```

O endereço base pode vir do Register File ou do Bypass, tanto da Via-0 quanto da Via-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]});
```

O deslocamento vem dos 32 bits da instrução na Via-0 ou Via-1.

TAREFA: Analisar a implementação dos dois somadores do andar DC1, que são instanciados no módulo `lsu_lsc_ctl`. Fornecemos orientação na Figura 7 abaixo, mostrando a implementação desses somadores.

Ficheiro `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

```

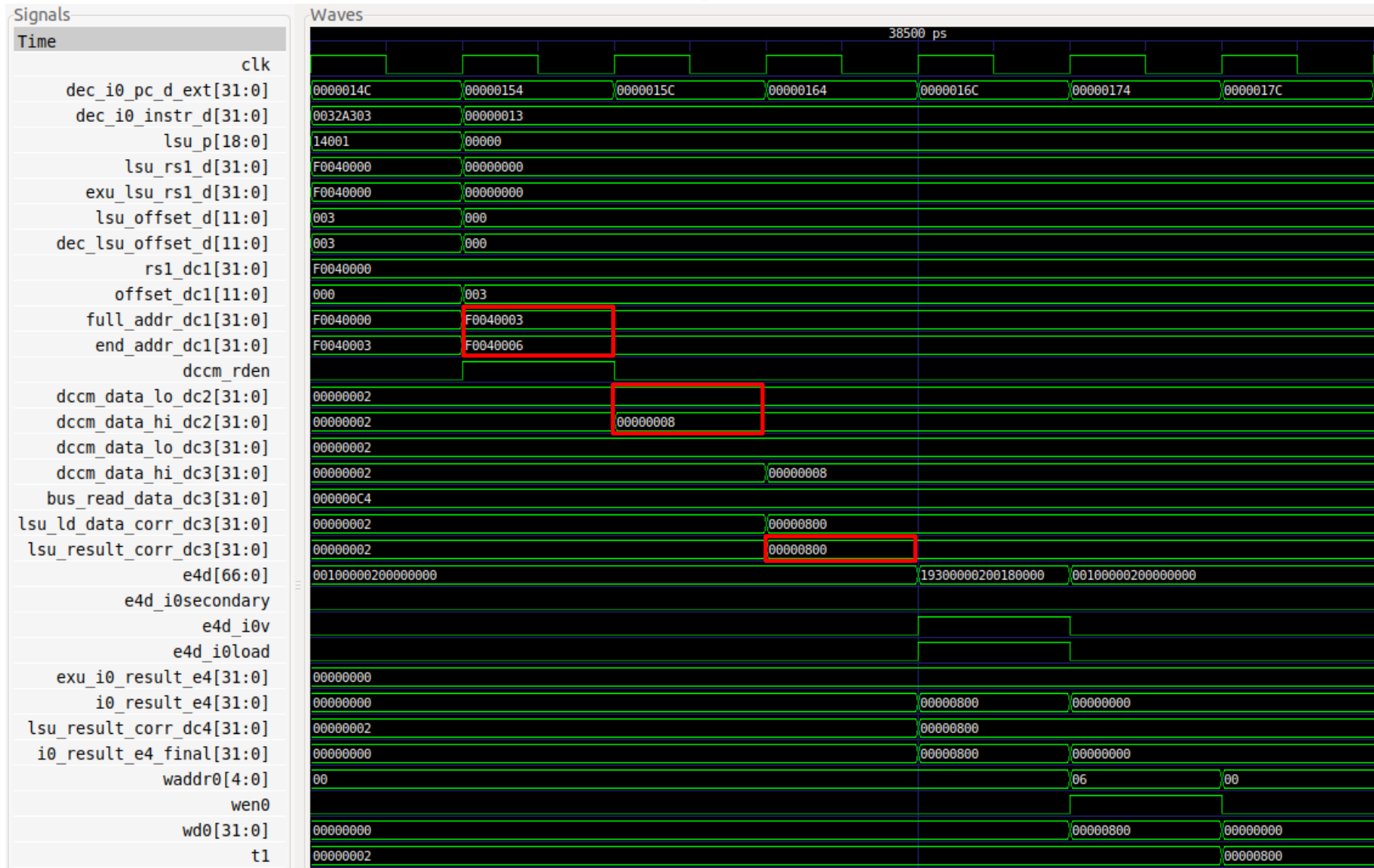
Ficheiro `lsu_lsc_ctl.sv`:

```

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];

```

TAREFA: No programa da Figura 2, experimente diferentes tamanhos de acesso (byte, half-word) e acessos não alinhados. Para isso, altere o deslocamento ou o tipo de acesso de `lw` para `lb` (*load byte*) ou `lh` (*load half-word*). Por exemplo, se alterar o deslocamento de 4 para 3, a instrução de leitura de palavra de leitura executará um acesso não alinhado aos 32 bits que começam no endereço 0xF0040003, conforme mostrado na Figura 8. Analise o valor dos sinais `lsu_addr_dc1[31:0]` (ou `full_addr_dc1[31:0]`) e `end_addr_dc1[31:0]` nessas diferentes situações. No Lab 20, analisamos essa situação a partir dos aspectos internos da DCCM.



Os valores dos sinais `lsu_addr_dc1[31:0]` e `end_addr_dc1[31:0]` comunicam à memória o endereço inicial e final do acesso: `0xF0040003` e `0xF0040007`. Duas palavras são lidas (`0x00000002` e `0x00000008`) e a palavra final é extraída no alinhador (`0x00000800`).

TAREFA: No programa da Figura 2, compare o valor dos sinais `dccm_data_lo_dc2[31:0]` e `dccm_data_hi_dc2[31:0]` ao fazer um `lw` para o endereço `0xF0040004` e para o endereço `0xF0040003`.

Acima, pode ver as duas simulações.

- `lw` para o endereço `0xF0040004`

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

Ambos os sinais contêm o valor lido do endereço solicitado.

- `lw` para o endereço `0xF0040003`

```
dccm_data_lo_dc2[31:0]: 0x00000002 (valor do endereço 0xF0040000)  
dccm_data_hi_dc2[31:0]: 0x00000008 (valor do endereço 0xF0040004)
```

TAREFA: Analisar a lógica de Align, Merge, e Error Check usada no código Verilog nos módulos `lsu_dccm_ctl` e `lsu_ecc`.

Solução não fornecida.

TAREFA: No programa da Figura 2, compare o valor do sinal `lsu_result_corr_dc3[31:0]` ao fazer um `lw` para o endereço `0xF0040004` e para o endereço `0xF0040003`.

Acima, pode ver as duas simulações.

- `lw` para o endereço `0xF0040004`

```
lsu_result_corr_dc3[31:0]: 0x00000008
```

Ele contém o valor lido do endereço solicitado.

- `lw` para o endereço `0xF0040003`

```
lsu_result_corr_dc3[31:0]: 0x00000800
```

Ele contém o valor lido do endereço pedido. Tenha em consideração que o RISC-V é little-endian.

TAREFA: Analise no código Verilog como o sinal `addr_external_dc1` foi computado no andar DC1 no módulo `lsu_addrcheck`.

```

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;

```

O módulo **rvrangecheck** é usado para verificar o endereço pedido:

- Se estiver dentro do intervalo de endereços DCCM/ICCM (linhas 80-107), nesse caso o sinal `addr_in_dccm_dc1 = 1`
- Se estiver dentro do intervalo de endereços do PIC (linhas 108-123), nesse caso o sinal `addr_in_pic_dc1 = 1`
- Se não estiver em nenhum desses intervalos de endereços, ele estará na memória externa DDR e, nesse caso:
`addr_external_dc1 = 1`

TAREFA: Verificar se esses 32 bits (0x0062a023) correspondem à instrução `sw t1, 0(t0)` na arquitetura RISC-V.

0x0062a023 → 00000000 00110 00101 010 00000 0100011

`imm11:0 = 000000000000`

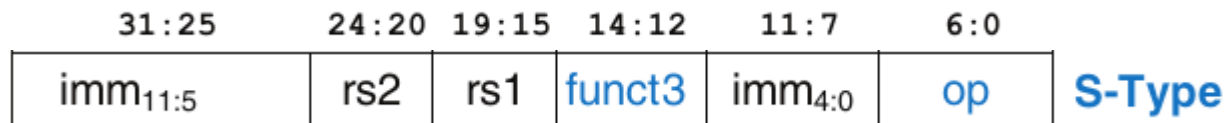
`rs2 = 00110 = x6 (t1)`

`rs1 = 00101 = x5 (t0)`

`funct3 = 010`

`op = 0100011`


Do Apêndice B do DDCARV:



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

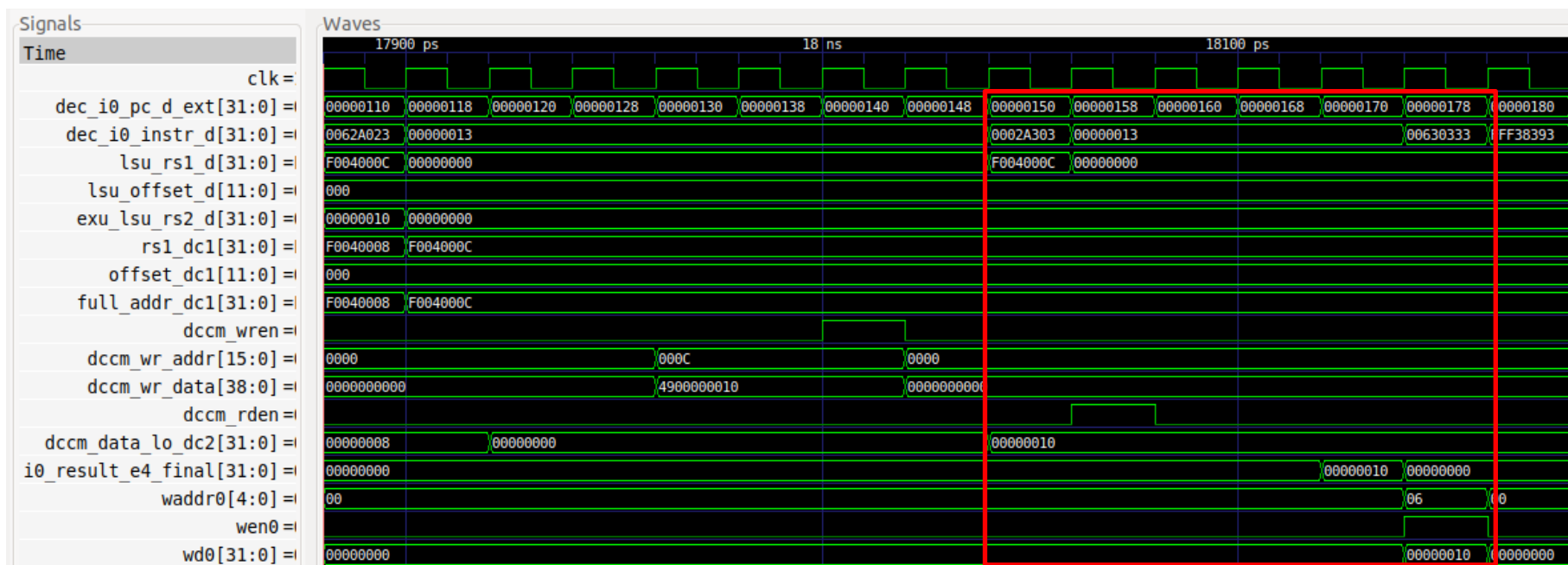
TAREFA: Replicar a simulação da Figura 12 no seu computador. Siga as próximas etapas (conforme descrito em detalhes na Seção 7 do GSG):

- Se necessário, gere o binário de simulação (*Vrvfpgasim*).

- Abra no PlatformIO o projeto fornecido em: *[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM*.
- Atualize o caminho para o binário de simulação do RVfpga (*Vrvfpgasim*) no ficheiro *platformio.ini*.
- Gere o trace da simulação com o Verilator (Generate Trace).
- Abra o trace no GTKWave.
- Use o ficheiro *scriptStore.tcl* (fornecido em *[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM/*) para exibir os mesmos sinais que os mostrados na Figura 4. Para isso, no GTKWave, clique em *File → Read Tcl Script File* e selecione o ficheiro *scriptStore.tcl*.
- Clique em *Zoom In* () várias vezes e vá para 17900ps.

Solução fornecida no documento principal do Lab 13.

TAREFA: Analise na simulação a instrução de leitura que segue a escrita para verificar se o valor foi escrito corretamente na DCCM. precisará adicionar alguns dos sinais da Figura 4 e da Figura 6 para analisar a leitura.




TAREFA: Estenda a análise básica realizada nesta seção para a instrução `sw` por uma via semelhante à análise avançada realizada para a instrução `lw` na Seção 2.B.

Solução não fornecida.

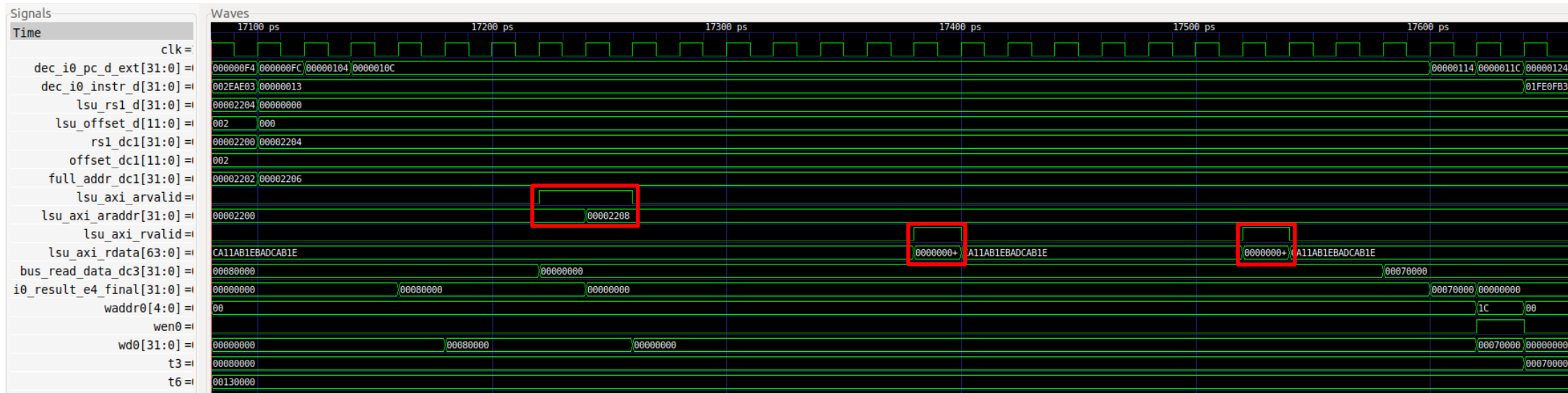
TAREFA: Analisar as escritas não alinhadas na DCCM, bem como escritas de sub-palavras: *store byte* (`sb`) ou *store half-word* (`sh`).

Solução não fornecida.

TAREFA: Replique a simulação da Figura 17 no seu computador. Use o ficheiro *test_Blocking.tcl* (fornecido em `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory`). Aumente o zoom () várias vezes e passe para 16940ps.

Solução fornecida no documento principal do Lab 13.

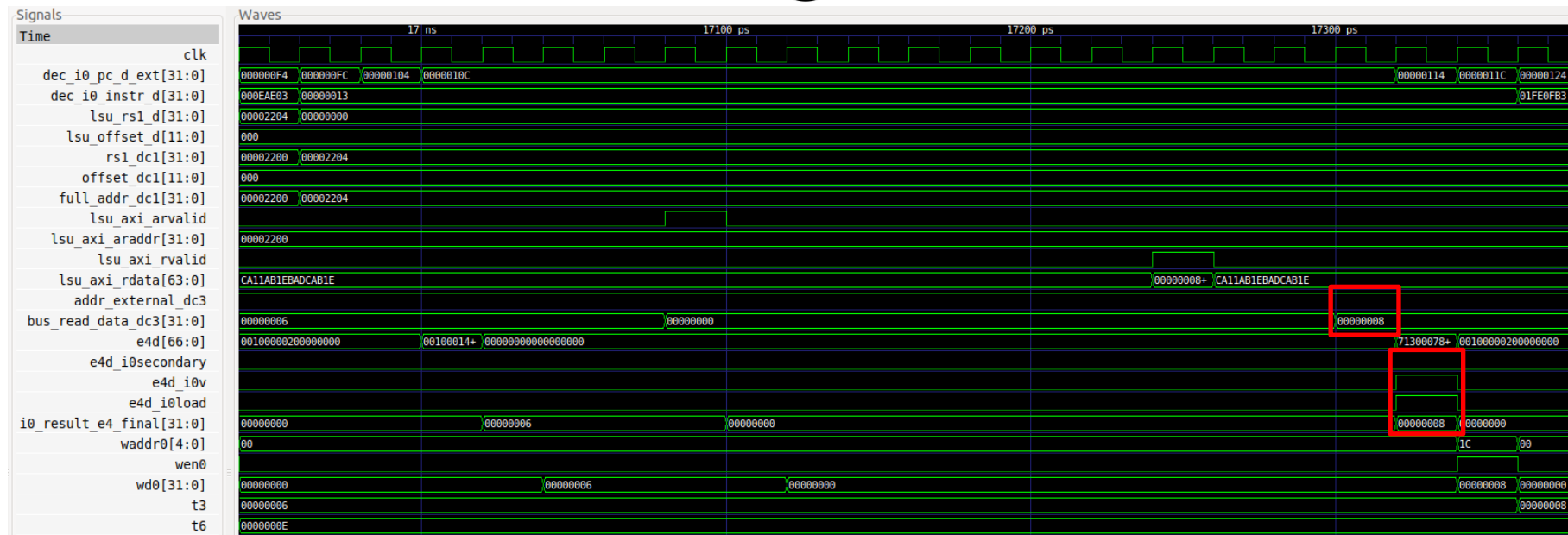
TAREFA: Modifique o programa da Figura 15 para analisar um acesso de leitura não alinhado que precisa enviar dois endereços para a memória externa por meio do barramento AXI.



TAREFA: Adicione à simulação os sinais que controlam os multiplexers (nos andares DC3 e Commit na Figura 16) que selecionam os dados fornecidos pela memória externa DDR. pode encontrar esses multiplexers nas seguintes linhas do código Verilog:


- Multiplexador 2:1: Linha 264 do módulo **lsu_lsc_ctl**.
- Multiplexador 3:1: Linha 2277 do módulo **dec_decode_ctl**.

Um ficheiro `.tcl` que pode usar é fornecido em: `[RVfpgaPath]/RVfpga/Labs/Lab13/LW_Instruction_ExtMemory/test_Blocking_Extended.tcl`



TAREFA: Também pode ser interessante analisar a implementação do barramento AXI para aceder ao controlador DRAM, para o qual pode inspecionar o módulo **lsu_bus_intf**.

Solução não fornecida.

TAREFA: Replique a simulação da Figura 18 no seu computador. Use o ficheiro *scriptStoreBuffer.tcl* (fornecido em `[RVfpgaPath]/RVfpga/Labs/Lab13/SW_Instruction_DCCM`). Aumente o zoom () várias vezes e passe para 17900ps.

Solução fornecida no documento principal do Lab 13.

TAREFA: Modifique o programa da Figura 11 para ter duas escritas pendentes e realizar uma análise semelhante à da Figura 18.

Solução não fornecida.