

TAREFAS

TAREFA: Verificar se esses 32 bits (0x01de0e33) correspondem à instrução `add t3, t3, t4` na arquitetura RISC-V.

0x01de0e33 → 0000000 11101 11100 000 11100 0110011

funct7 = 0000000

rs2 = 11101 = x29 (t4)

rs1 = 11100 = x28 (t3)

funct3 = 000

rd = 11100 = x28 (t3)


op = 0110011

Do Apêndice B do DDCARV:

31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	R-Type
op	funct3	funct7	Type	Instruction	Description	Operation
0110011 (51)	000	0000000	R	add rd, rs1, rs2	add	rd = rs1 + rs2

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 3 no seu computador. Para fazer isso, 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/Lab12/ADD_Instruction*.
- Defina o caminho correto 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 *test_1.tcl* (fornecido em *[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/*) para abrir os mesmos sinais que os mostrados na Figura 3. Para isso, no GTKWave, clique em *File - Read Tcl Script File* e selecione o ficheiro *test_1.tcl*.
- Clique em *Zoom In* () várias vezes e vá para 15000ps.

Solução fornecida no documento principal do Lab 12.

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

- Unidade de controlo no módulo **dec_decode_ctl**
- Register File:
 - o Instanciação na linha 525 do módulo **dec**.
 - o Implementação no módulo **dec_gpr_ctl**.
- Muxes 3:1 no andarcodex: Linha 279 do módulo **exu**.
- Registos de pipeline para sinais de controlo: Distribuídos em vários módulos.
- Registos aff e bff: Linhas 90 e 92 do módulo **exu_alu_ctl**.
- IO ALU em EX1:
 - o Instanciação na linha 401 do módulo **exu**.
 - o Implementação no módulo **exu_alu_ctl**.
- Registos do pipeline com o resultado da operação (*i0e2resultff*, *i0e3resultff*, *i0e4resultff*, *i0wbresultff*): Linhas 2260-2283 do módulo **dec_decode_ctl**.
- Mux 3:1 no andar EX3: Linha 2268 do módulo **dec_decode_ctl**.
- Mux 3:1 no andar EX4: Linha 2277 do módulo **dec_decode_ctl**.
- Mux 2:1 no andar Writeback: Linha 2286 do módulo **dec_decode_ctl**.

TAREFA: Encontre no código Verilog (módulo **dec_decode_ctl**) como o sinal de controlo *i0r* é usado para ler o Register File.

- Os identificadores de registo são obtidos da instrução de 32 bits na Via-0: sinal `i0[31:0] = dec_i0_instr_d[31:0]`.
Numa instrução do tipo R, eles estão localizados nos seguintes campos:



No módulo **dec_decode_ctl**:

```
1121 assign i0r.rs1[4:0] = i0[19:15];
1122 assign i0r.rs2[4:0] = i0[24:20];
1123 assign i0r.rd[4:0] = i0[11:7];
```

- Os identificadores de registo e os sinais de habilitação de leitura são atribuídos a `dec_i0_rs1_d/dec_i0_rs2_d` e `dec_i0_rs1_en_d/dec_i0_rs2_en_d`.
Estes sinais são enviados do módulo **dec** para o módulo **dec_decode_ctl**. No módulo **dec_decode_ctl**:

```
1130 assign dec_i0_rs1_en_d = i0_dp.rs1 & (i0r.rs1[4:0] != 5'd0);
1131 assign dec_i0_rs2_en_d = i0_dp.rs2 & (i0r.rs2[4:0] != 5'd0);
1132 assign i0_rd_en_d = i0_dp.rd & (i0r.rd[4:0] != 5'd0);
1133
1134 assign dec_i0_rs1_d[4:0] = i0r.rs1[4:0];
1135 assign dec_i0_rs2_d[4:0] = i0r.rs2[4:0];
1136 assign i0_rd_d[4:0] = i0r.rd[4:0];
1137
```

- Os identificadores de registo e os sinais de habilitação de leitura são fornecidos ao Register File, que é instanciado no módulo **dec**. No módulo **dec**:

```
525 dec_gpr_ctl #(.GPR_BANKS(GPR_BANKS),
526               .GPR_BANKS_LOG2(GPR_BANKS_LOG2)) arf (.,
527               // inputs
528               .raddr0(dec_i0_rs1_d[4:0]), .rden0(dec_i0_rs1_en_d),
529               .raddr1(dec_i0_rs2_d[4:0]), .rden1(dec_i0_rs2_en_d),
530               .raddr2(dec_i1_rs1_d[4:0]), .rden2(dec_i1_rs1_en_d),
531               .raddr3(dec_i1_rs2_d[4:0]), .rden3(dec_i1_rs2_en_d),
532
533               .waddr0(dec_i0_waddr_wb[4:0]), .wen0(dec_i0_wen_wb), .wd0(dec_i0_wdata_wb[31:0]),
534               .waddr1(dec_i1_waddr_wb[4:0]), .wen1(dec_i1_wen_wb), .wd1(dec_i1_wdata_wb[31:0]),
535               .waddr2(dec_nonblock_load_waddr[4:0]), .wen2(dec_nonblock_load_wen), .wd2(lsu_nonblock_load_data[31:0]),
536
537               // outputs
538               .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0]),
539               .rd2(gpr_i1_rs1_d[31:0]), .rd3(gpr_i1_rs2_d[31:0])
540           );
```

TAREFA: Encontre no código Verilog (módulo **exu**) como os sinais de controlo **i0_ap** e **dd** são propagados do andar Decode para o andar Execução. Além disso, descubra como o sinal de controlo **dd** é usado pelo Register File no andar Write-Back, depois de passar por todos os andares Decode a Writeback.

O sinal **i0_ap** é obtido no módulo **dec_decode_ctl**. Ele é fornecido ao módulo **exu**, onde é propagado para EX1, EX2, EX3 e Commit (EX4). No módulo **exu**:

```
454 rvdffe #($bits(alu_pkt_t)) i0_ap_e1_ff (.*, .en(i0_e1_ctl_en), .din(i0_ap), .dout(i0_ap_e1) );
455 rvdffe #($bits(alu_pkt_t)) i0_ap_e2_ff (.*, .en(i0_e2_ctl_en), .din(i0_ap_e1), .dout(i0_ap_e2) );
456 rvdffe #($bits(alu_pkt_t)) i0_ap_e3_ff (.*, .en(i0_e3_ctl_en), .din(i0_ap_e2), .dout(i0_ap_e3) );
457 rvdffe #($bits(alu_pkt_t)) i0_ap_e4_ff (.*, .en(i0_e4_ctl_en), .din(i0_ap_e3), .dout(i0_ap_e4) );
```

O sinal **dd** é obtido no módulo **dec_decode_ctl** e propagado para EX1, EX2, EX3, Commit (EX4) e WB (EX5). No módulo **dec_decode_ctl**:

```
2139 rvdffe #($bits(dest_pkt_t)) e1ff (.*, .en(i0_e1_ctl_en), .din(dd), .dout(e1d));
2155 rvdffe #($bits(dest_pkt_t)) e2ff (.*, .en(i0_e2_ctl_en), .din(e1d_in), .dout(e2d));
2168 rvdffe #($bits(dest_pkt_t)) e3ff (.*, .en(i0_e3_ctl_en), .din(e2d_in), .dout(e3d));
2193 rvdffe #($bits(dest_pkt_t)) e4ff (.*, .en(i0_e4_ctl_en), .din(e3d_in), .dout(e4d));
2219 rvdffe #($bits(dest_pkt_t)) wbff (.*, .en(i0_wb_ctl_en | exu_div_finish | div_wen_wb), .din(e4d_in), .dout(wbd));
```

Note que a saída de cada registo é ligeiramente modificada (e, portanto, renomeada) antes de entrar no próximo registo. Pode consultar o código Verilog se quiser verificar os detalhes.

O identificador de registo para o operando de saída é atribuído no andar Decode:

```
2070 assign dd.i0rd[4:0] = i0r.rd[4:0];
```

O sinal **dd** é propagado da Decodificação para o Writeback, conforme mostrado acima: **dd** → **e1d** → **e2d** → **e3d** → **e4d** → **wbd**. Em seguida, o registo de destino é fornecido ao Register File no andar Writeback:

```
2221 assign dec_i0_waddr_wb[4:0] = wbd.i0rd[4:0];
```

```

525     dec_gpr_ctl #(.GPR_BANKS(GPR_BANKS),
526                 .GPR_BANKS_LOG2(GPR_BANKS_LOG2)) arf (.*,
527                 // inputs
528                 .raddr0(dec_i0_rs1_d[4:0]), .rden0(dec_i0_rs1_en_d),
529                 .raddr1(dec_i0_rs2_d[4:0]), .rden1(dec_i0_rs2_en_d),
530                 .raddr2(dec_i1_rs1_d[4:0]), .rden2(dec_i1_rs1_en_d),
531                 .raddr3(dec_i1_rs2_d[4:0]), .rden3(dec_i1_rs2_en_d),
532
533                 .waddr0(dec_i0_waddr_wb[4:0]), .wen0(dec_i0_wen_wb), .wd0(dec_i0_wdata_wb[31:0]),
534                 .waddr1(dec_i1_waddr_wb[4:0]), .wen1(dec_i1_wen_wb), .wd1(dec_i1_wdata_wb[31:0]),
535                 .waddr2(dec_nonblock_load_waddr[4:0]), .wen2(dec_nonblock_load_wen), .wd2(lsu_nonblock_load_data[31:0]),
536
537                 // outputs
538                 .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0]),
539                 .rd2(gpr_i1_rs1_d[31:0]), .rd3(gpr_i1_rs2_d[31:0])
540             );

```

TAREFA: A geração desses dois sinais (`i0_e1_ctl_en` e `dec_i0_alu_decode_d`) é um processo bastante complexo que não explicamos aqui em detalhe, mas que pode analisar por conta própria nos módulos `dec_decode_ctl` e `exu`.

Solução não fornecida.

TAREFA: Localize no código Verilog (módulo `exu`) o multiplexer 3:1 na parte inferior (segundo operando de entrada) e tente encontrar a origem de suas entradas (na Figura 6, apenas a entrada proveniente do Register File é mostrada). Não é necessário examinar as entradas com muita atenção, pois elas serão analisadas nos exercícios propostos na Seção 3 e em Labs futuros.

```

286     assign i0_rs2_d[31:0] = ({32{~dec_i0_rs2_bypass_en_d}} & gpr_i0_rs2_d[31:0]) |
287                             ({32{~dec_i0_rs2_bypass_en_d}} & dec_i0_immed_d[31:0]) |
288                             ({32{ dec_i0_rs2_bypass_en_d}} & i0_rs2_bypass_data_d[31:0]);

```

Estes muxes 3:1 recebem 3 entradas:

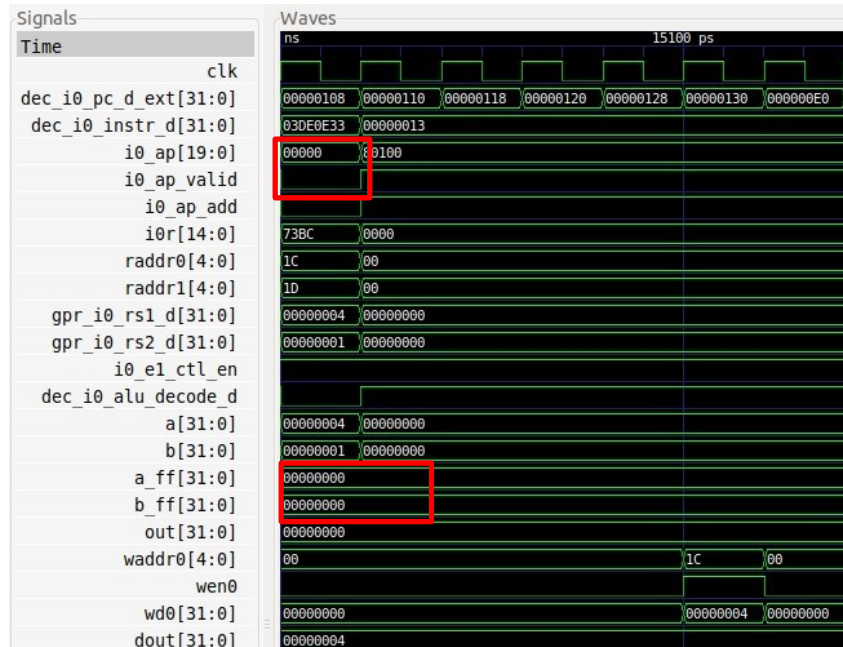
- Um do Register File (`gpr_i0_rs2_d`)
- Um do Instruction Register de 32 bits, que constitui o imediato (`dec_i0_immed_d`)
- Um da lógica de bypass, que analisamos no Lab 15 (`i0_rs2_bypass_data_d`)

TAREFA: Replicar a simulação da Figura 7 no seu computador. Pode usar o script `.tcl` fornecido em: `[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/test_2.tcl`. Observe que os aliases são usados nesse ficheiro `.tcl` para alguns dos bits de controlo.

Solução fornecida no documento principal do Lab 12.

TAREFA: No exemplo da Figura 2, substitua a instrução `add` por uma instrução não A-L (como uma instrução `mul`). Verifique se o sinal `i0_ap` tem todos os seus campos iguais a 0 e se isso faz com que a ALU I0 não funcione (verá que os sinais `a_ff` e `b_ff` para o Pipe I0 no andar EX1 permanecem estáveis para essa instrução). Pode usar o mesmo ficheiro `test_2.tcl` usado no exemplo da Figura 7.

Por exemplo, a simulação de `mul t3, t3, t4 (0x03de0e33)` fornece os seguintes resultados:

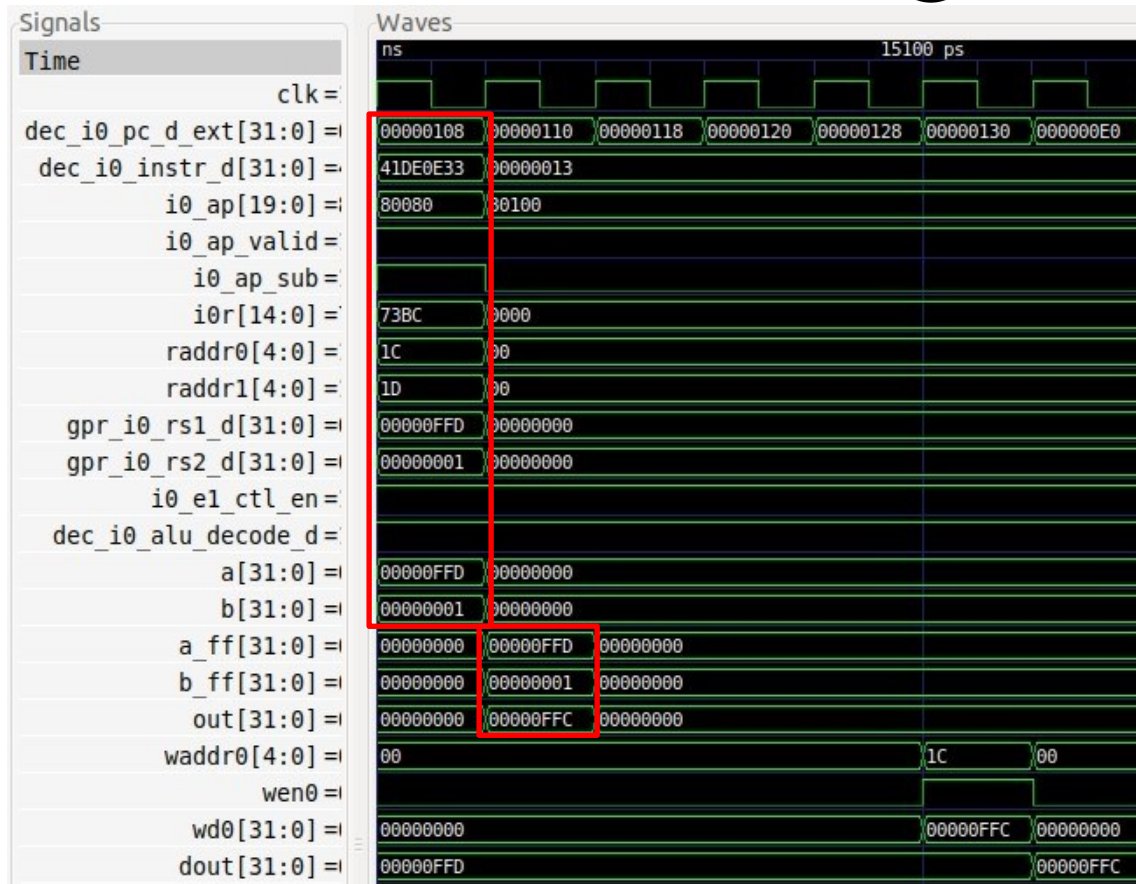


TAREFA: Inclua os novos sinais analisados nesta seção na simulação da Figura 7.

Solução não fornecida.

TAREFA: Realize uma simulação de uma sub-instrução semelhante à da Figura 7. Lembre-se de que pode incluir novos sinais na simulação por meio do ficheiro `.tcl`.

Por exemplo, a simulação de `sub t3, t3, t4 (0x41de0e33)` fornece os seguintes resultados:



TAREFA: Analisar a implementação Verilog do somador/subtrator implementado no módulo **exu_alu_ctl**. A Figura 8 lhe dá alguma ajuda, mostrando a lógica diretamente relacionada às operações de adição e subtração.

```

90   rvdfte #(32) aff (.*, .en(enable & valid), .din(a[31:0]), .dout(a_ff[31:0]));
91
92   rvdfte #(32) bff (.*, .en(enable & valid), .din(b[31:0]), .dout(b_ff[31:0]));

```

Os operandos de entrada são propagados do andar Decode (a e b) para o andar Execução (a_ff e b_ff).

```

135     assign bm[31:0] = ( ap.sub ) ? ~b_ff[31:0] : b_ff[31:0];
136
137
138     assign {cout, aout[31:0]} = {1'b0, a_ff[31:0]} + {1'b0, bm[31:0]} + {32'b0, ap.sub};
139

```

Esse é o somador/subtrator.

- Se a instrução for uma adição, aout = a_ff + b_ff
- Se a instrução for uma subtração, b_ff será primeiro complementado por dois e, em seguida, a_out será computado.

```

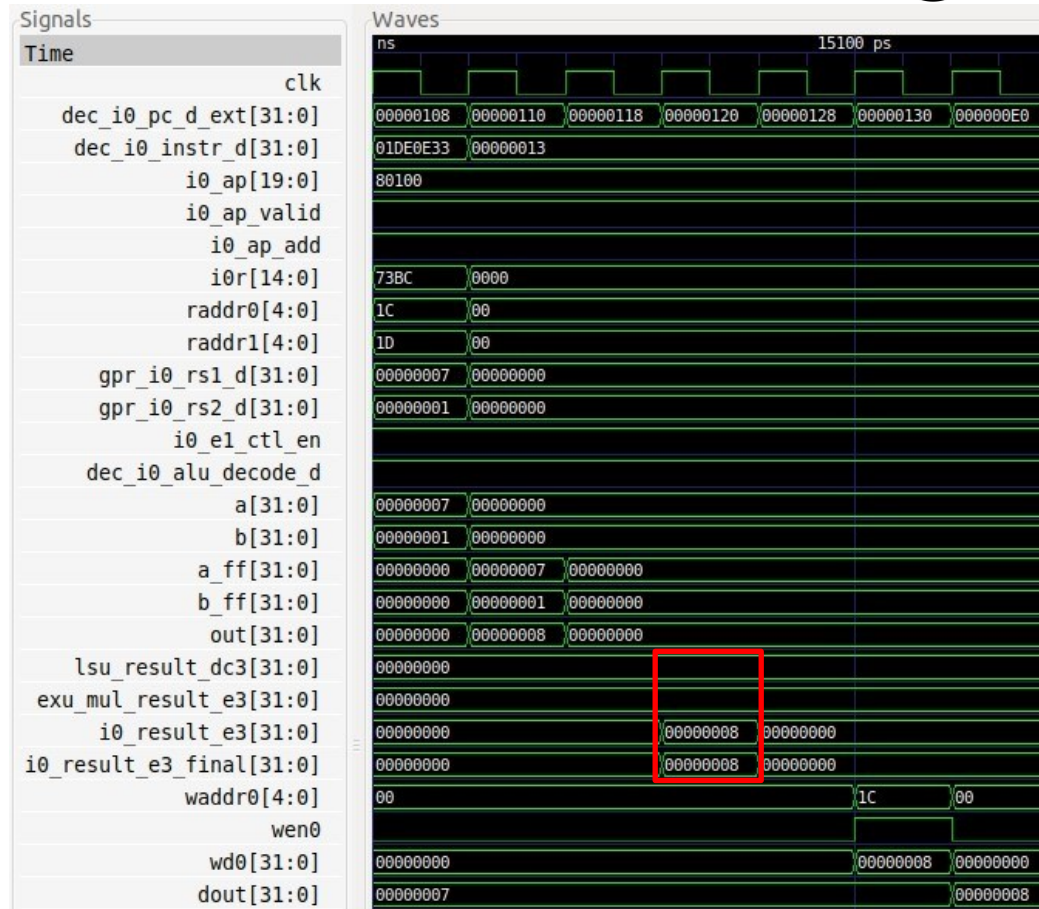
172     assign sel_adder = (ap.add | ap.sub) & ~ap.slt;

185     assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                       ({32{sel_shift}} & sout[31:0]) |
187                       ({32{sel_adder}} & aout[31:0]) |
188                       ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1], 1'b0}) |
189                       ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                       ({31'b0, slt_one});
191

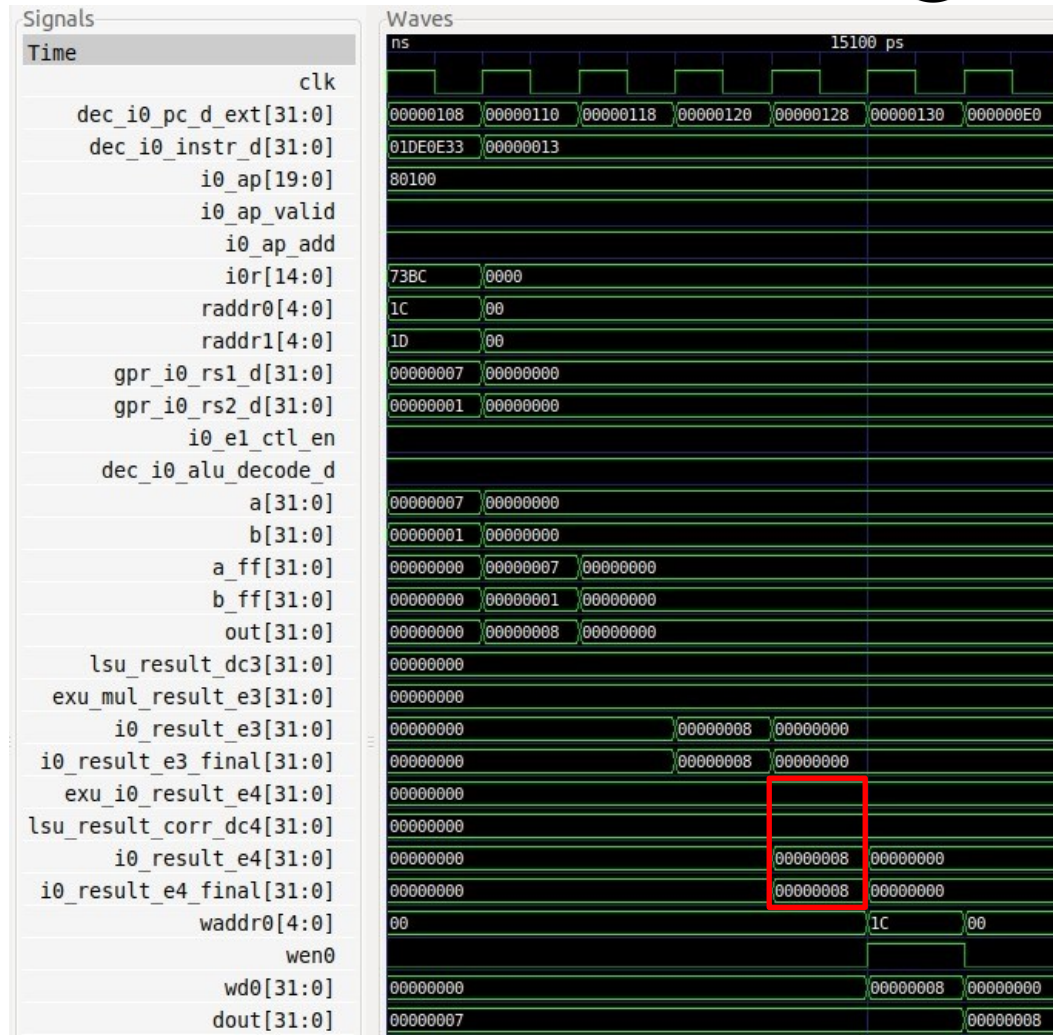
```

Se a instrução for uma adição ou uma subtração, então out = aout.

TAREFA: Verificar na simulação se esse multiplexer seleciona o resultado do Pipe esperado para a instrução `add`, para o exemplo da Figura 2.



TAREFA: Verificar na simulação se esse multiplexer seleciona o resultado da fonte de entrada adequada (`i0_result_e4`) para a instrução `add` do nosso exemplo da Figura 2.



TAREFA: No código Verilog, analise como os sinais `wen0` e `waddr0` são gerados no andar Decode e propagados para o andar Writeback.

```

525     dec_gpr_ctl #(.GPR_BANKS(GPR_BANKS),
526                 .GPR_BANKS_LOG2(GPR_BANKS_LOG2)) arf (.*,
527                 // inputs
528                 .raddr0(dec_i0_rs1_d[4:0]), .rden0(dec_i0_rs1_en_d),
529                 .raddr1(dec_i0_rs2_d[4:0]), .rden1(dec_i0_rs2_en_d),
530                 .raddr2(dec_i1_rs1_d[4:0]), .rden2(dec_i1_rs1_en_d),
531                 .raddr3(dec_i1_rs2_d[4:0]), .rden3(dec_i1_rs2_en_d),
532
533                 .waddr0(dec_i0_waddr_wb[4:0]), .wen0(dec_i0_wen_wb), .wd0(dec_i0_wdata_wb[31:0]),
534                 .waddr1(dec_i1_waddr_wb[4:0]), .wen1(dec_i1_wen_wb), .wd1(dec_i1_wdata_wb[31:0]),
535                 .waddr2(dec_nonblock_load_waddr[4:0]), .wen2(dec_nonblock_load_wen), .wd2(lsu_nonblock_load_data[31:0]),
536
537                 // outputs
538                 .rd0(gpr_i0_rs1_d[31:0]), .rd1(gpr_i0_rs2_d[31:0]),
539                 .rd2(gpr_i1_rs1_d[31:0]), .rd3(gpr_i1_rs2_d[31:0])
540             );
541

```

```

2221     assign dec_i0_waddr_wb[4:0] = wbd.i0rd[4:0];

```

```

2224     assign i0_wen_wb = wbd.i0v & ~(~dec_tlu_i1_kill_writeb_wb & ~i1_load_kill_wen & wbd.i0v & wbd.i1v & (wbd.i0rd[4:0] == wbd.i1rd[4:0])) & ~dec_tlu_i0_kill_writeb_wb;
2225     assign dec_i0_wen_wb = i0_wen_wb & ~i0_load_kill_wen; // don't write a nonblock load 1st time down the pipe
2226

```

```

2070     assign dd.i0rd[4:0] = i0r.rd[4:0];
2071     assign dd.i0v = i0_rd_en_d & i0_legal_decode_d;

```

EXERCÍCIOS

1) Faça uma análise semelhante à apresentada neste Lab para instruções lógicas (`and`, `or`, `xor`).

O exemplo a seguir, fornecido em `[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab12/AND_Instruction`, ilustra a execução de uma instrução `and` contida num ciclo que se repete para sempre. Como no exemplo da instrução `add`, a instrução `and` (destacada em vermelho) é rodeada por várias instruções `nop`. Duas instruções são incluídas no final do ciclo para modificar os valores armazenados em `t3` e `t4`.

```
#define INSERT_NOPS_1      nop;
#define INSERT_NOPS_2      nop; INSERT_NOPS_1
#define INSERT_NOPS_3      nop; INSERT_NOPS_2
#define INSERT_NOPS_4      nop; INSERT_NOPS_3
#define INSERT_NOPS_5      nop; INSERT_NOPS_4
#define INSERT_NOPS_6      nop; INSERT_NOPS_5
#define INSERT_NOPS_7      nop; INSERT_NOPS_6
#define INSERT_NOPS_8      nop; INSERT_NOPS_7
#define INSERT_NOPS_9      nop; INSERT_NOPS_8
#define INSERT_NOPS_10     nop; INSERT_NOPS_9

.globl main
main:

    li t3, 0xFC           # t3 = 0xFC
    li t4, 0x7            # t4 = 0x7

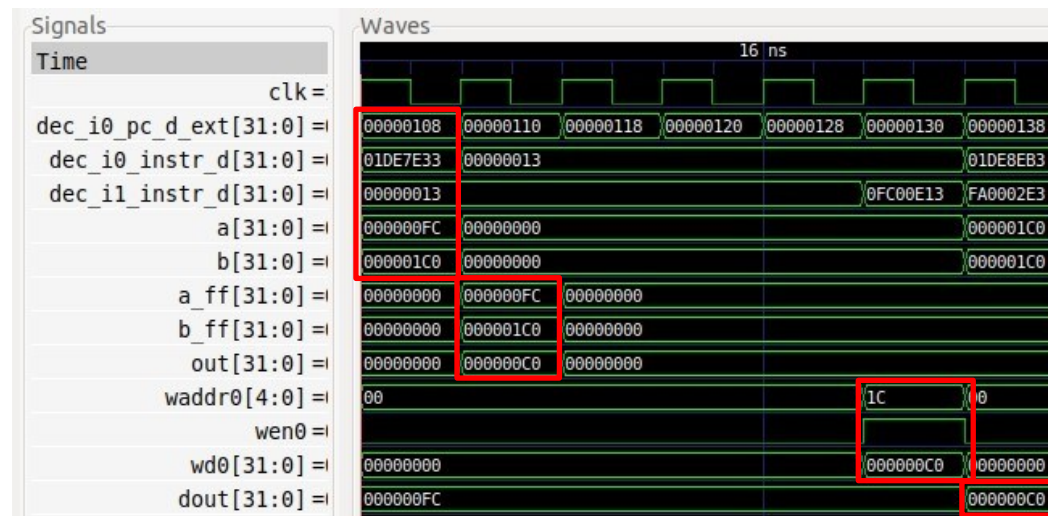
REPEAT:
    INSERT_NOPS_10
    and t3, t3, t4         # t3 = t3 & t4
    INSERT_NOPS_10
    li t3, 0xFC           # t3 = 0xFC
    add t4, t4, t4
    beq zero, zero, REPEAT # Repete o ciclo

.end
```

Se abrir o projeto no PlatformIO, compilá-lo e abrir o ficheiro Disassembly (disponível em `[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab12/AND_Instruction/.pio/build/swervolf_nexys/firmware.dis`) irá ver que a instrução `and` é colocada no endereço `0x00000108`, e também pode ver o código de máquina da instrução (`0x01de7e33`):

```
0x00000108:    01de7e33          and    t3, t3, t4
```

Em seguida, simulamos o programa no Verilator e abrimos o ficheiro trace gerado pelo simulador no GTKWave. Vá para qualquer iteração do ciclo, exceto a primeira.



Analise a forma de onda (os valores destacados em vermelho correspondem à instrução `and`). Neste Lab, saltamos os andares Fetch e Align, que serão explicados num próximo Lab.

- Andar **Decode**: O sinal `dec_i0_pc_d_ext` contém o endereço da instrução (nos livros, isso geralmente é chamado de Program Counter), que para o `and` é `0x00000108`, e o sinal `dec_i0_instr_d` contém a instrução de máquina de 32 bits `0x01DE7E33` (nos livros, isso geralmente é chamado de Instruction Register).

No RISC-V, o opcode para a instrução `and` é (consulte o Apêndice B de [Harris&Harris]):

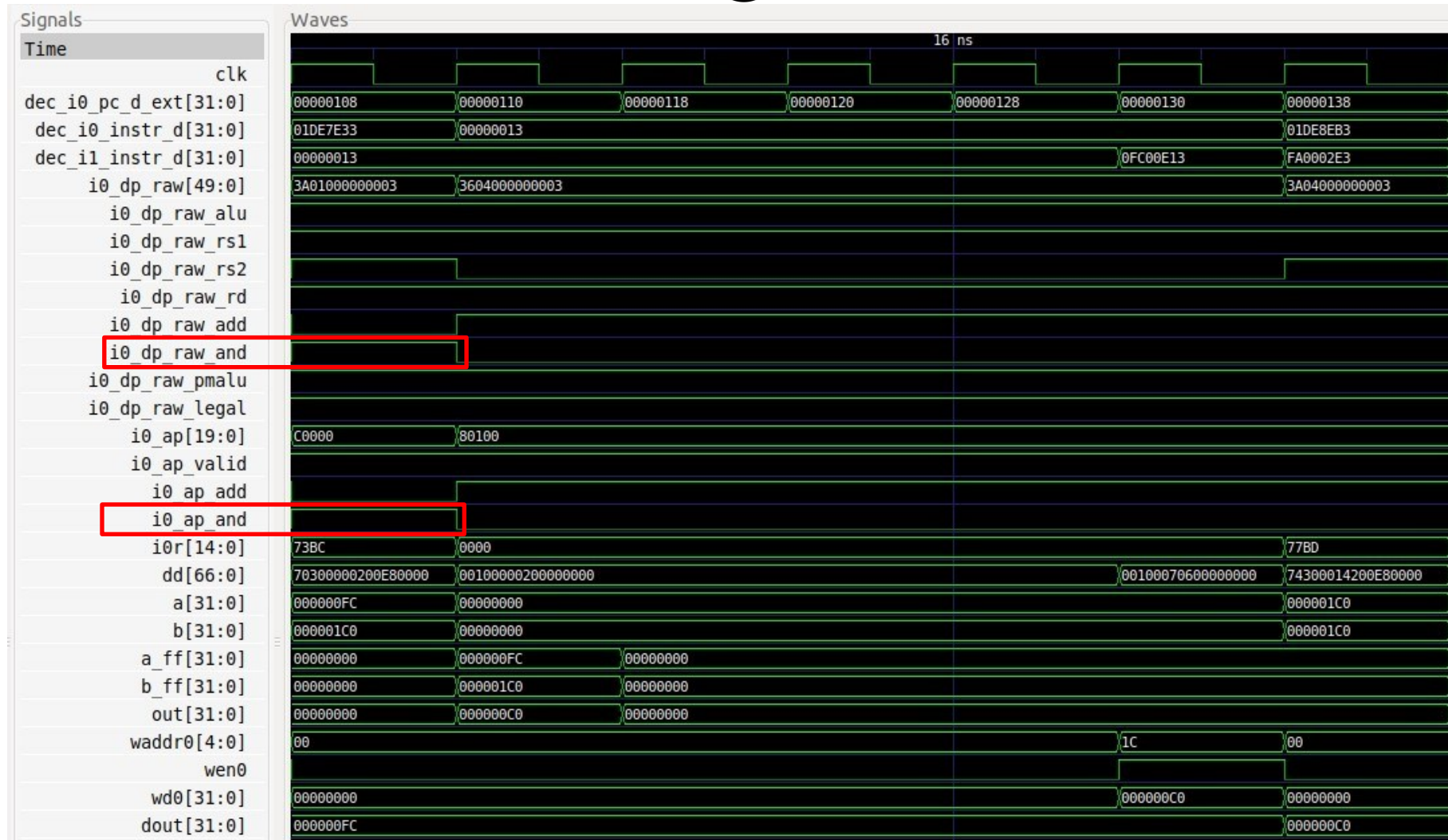
```
00000000 | rs2 | rs1 | 111 | rd | 0110011
```

Assim, pode verificar facilmente que `0x01DEFE33` corresponde a: `and t3, t3, t4` (lembre-se de que `t3=x28` e `t4=x29`).

Neste andar, os **sinais de controlo do pipeline são gerados** (mostraremos alguns detalhes na próxima seção). Além disso, os **O Register File é lido** nesse andar. Os sinais `a` e `b` contêm as entradas para a ALU, que, nesse caso, coincidem com os valores lidos do Register File (noutros casos que analisaremos nos próximos Labs, esse não será o caso).

- Andar **EX1**: no próximo ciclo, a instrução `and` é **executada**. Os sinais `a_ff` e `b_ff` contêm as entradas para a ALU (0xFC e 0x1C0, respectivamente), enquanto `out` contém o resultado da adição (0xC0).
- Andar **EX5**, também chamado de **Writeback**: Finalmente, 4 ciclos depois, o resultado da adição é **escrito de volta** no Register File por meio do sinal `wd0=0xC0`, que contém os dados a serem gravados. Dado que `wen0=1` (habilitação de gravação), o resultado da operação `and` é gravado no final desse ciclo no registo x28 (o índice do registo, `waddr0=0x1C`). É possível observar que, no ciclo seguinte (último ciclo mostrado na figura), o registo x28 contém o novo valor (`dout=0xC0`).

Em seguida, adicionamos os sinais de controlo à simulação anterior:



Pode ver que o bit de controlo para a instrução `and` é 1 no primeiro ciclo.

Os segmentos de código Verilog a seguir mostram a unidade lógica do SweRV EH1.

```

90     rvdffe #(32) aff (.*, .en(enable & valid), .din(a[31:0]), .dout(a_ff[31:0]));
91
92     rvdffe #(32) bff (.*, .en(enable & valid), .din(b[31:0]), .dout(b_ff[31:0]));

149     assign logic_sel[3] = ap.land | ap.lor;
150     assign logic_sel[2] = ap.lor | ap.lxor;
151     assign logic_sel[1] = ap.lor | ap.lxor;
152
153
154
155     assign lout[31:0] = ( a_ff[31:0] & b_ff[31:0] & {32{logic_sel[3]}} ) |
156                       ( a_ff[31:0] & ~b_ff[31:0] & {32{logic_sel[2]}} ) |
157                       ( ~a_ff[31:0] & b_ff[31:0] & {32{logic_sel[1]}} );

158
159
168     assign sel_logic = {ap.land,ap.lor,ap.lxor};

185     assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                       ({32{sel_shift}} & sout[31:0]) |
187                       ({32{sel_adder}} & aout[31:0]) |
188                       ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1],1'b0}) |
189                       ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                       ({31'b0, slt_one});
191

```

Quando o bit de controlo da and é 1, o resultado da operação and é selecionado:

$\text{logic_sel}[3]=1 \text{ and } \text{logic_sel}[2]=\text{logic_sel}[1]=0 \rightarrow \text{lout} = \text{a_ff} \& \text{b_ff}$

2) (O exercício a seguir é baseado no exercício 4.1 do livro "Computer Organization and Design - RISC-V Edition", de Patterson & Hennessy ([HePa]).

Considere a seguinte instrução: `and rd, rs1, rs2`

- Quais são os valores dos sinais de controlo gerados pelo SweRV EH1 para essa instrução?
- Quais recursos (blocos) que desempenham uma função útil para essa instrução?
- Quais recursos (blocos) que não produzem saída para essa instrução? Quais recursos produzem saída que não é usada?

Solução não fornecida.

3) Analise, tanto numa simulação do Verilator quanto diretamente no código Verilog, as instruções de *deslocamento à esquerda/direita* disponíveis no conjunto de instruções do RV32I Base Integer: `srl`, `sra` e `sll`.

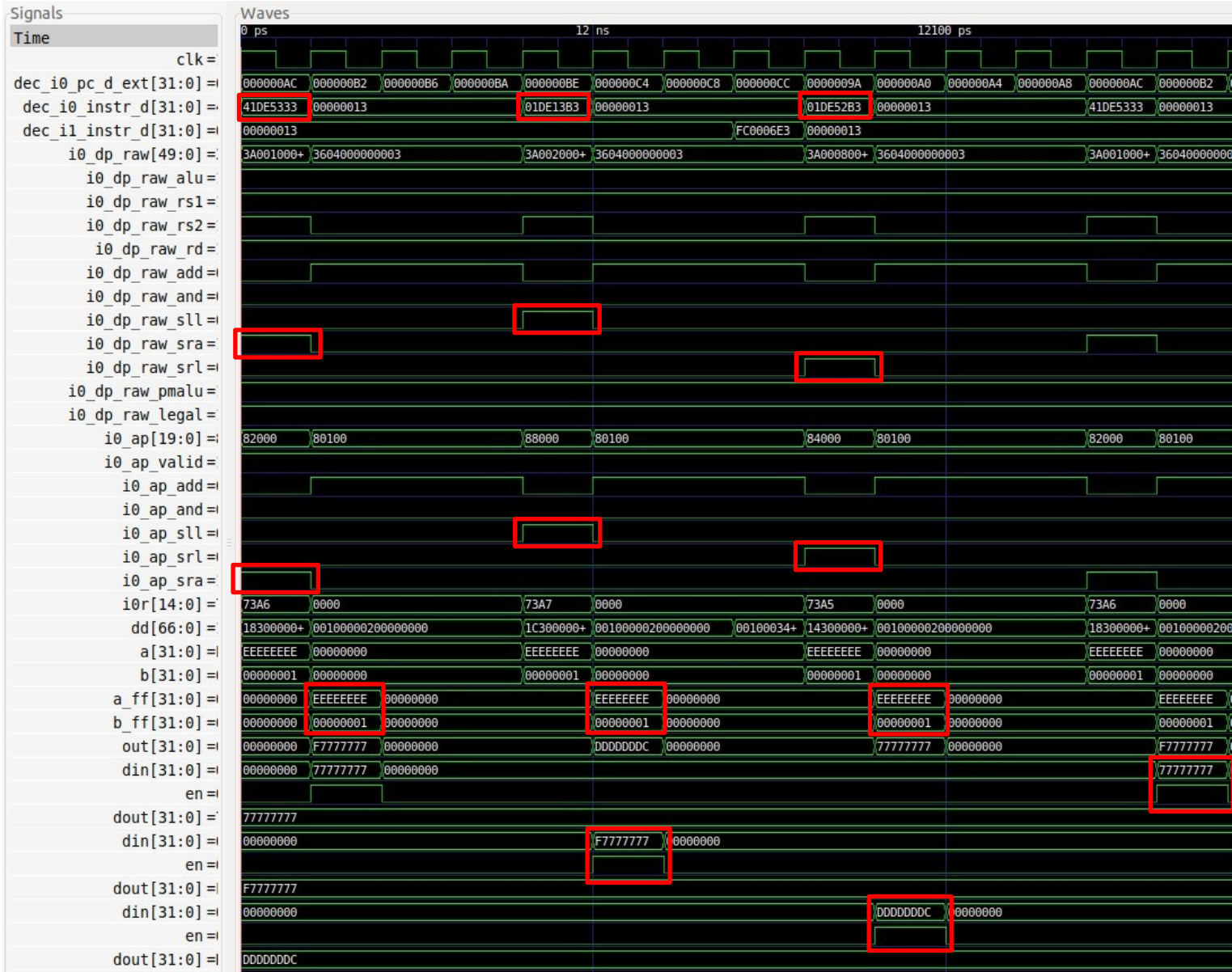
```
#define INSERT_NOPS_0
#define INSERT_NOPS_1    nop; INSERT_NOPS_0
#define INSERT_NOPS_2    nop; INSERT_NOPS_1
#define INSERT_NOPS_3    nop; INSERT_NOPS_2
#define INSERT_NOPS_4    nop; INSERT_NOPS_3
#define INSERT_NOPS_5    nop; INSERT_NOPS_4
#define INSERT_NOPS_6    nop; INSERT_NOPS_5
#define INSERT_NOPS_7    nop; INSERT_NOPS_6
#define INSERT_NOPS_8    nop; INSERT_NOPS_7
#define INSERT_NOPS_9    nop; INSERT_NOPS_8
#define INSERT_NOPS_10   nop; INSERT_NOPS_9

.globl main
main:

li t3, 0xEEEEEEEEE
li t4, 0x1

REPEAT:
    srl t0, t3, t4
    INSERT_NOPS_7
    sra t1, t3, t4
    INSERT_NOPS_7
    sll t2, t3, t4
    INSERT_NOPS_6
    beq zero, zero, REPEAT # Repete o ciclo

.end
```



Os segmentos de código Verilog a seguir mostram a unidade de deslocamento (Shift Unit) do SweRV EH1.

```

161      assign ashift[31:0] = a_ff >>> b_ff[4:0];
162
163      assign sout[31:0] = ( {32{ap.sll}} & (a_ff[31:0] << b_ff[4:0]) ) |
164                          ( {32{ap.srl}} & (a_ff[31:0] >> b_ff[4:0]) ) |
165                          ( {32{ap.sra}} & ashift[31:0] );
166
167
168      assign sel_shift = |{ap.sll,ap.srl,ap.sra};
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185      assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                          ({32{sel_shift}} & sout[31:0]) |
187                          ({32{sel_adder}} & aout[31:0]) |
188                          ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1],1'b0}) |
189                          ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                          ({31'b0, slt_one});
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

4) Analise, tanto numa simulação do Verilator quanto diretamente no código Verilog, as instruções *set-less-than* disponíveis no Instruction Set do RV32I Base Integer: `slt` e `sltu`.

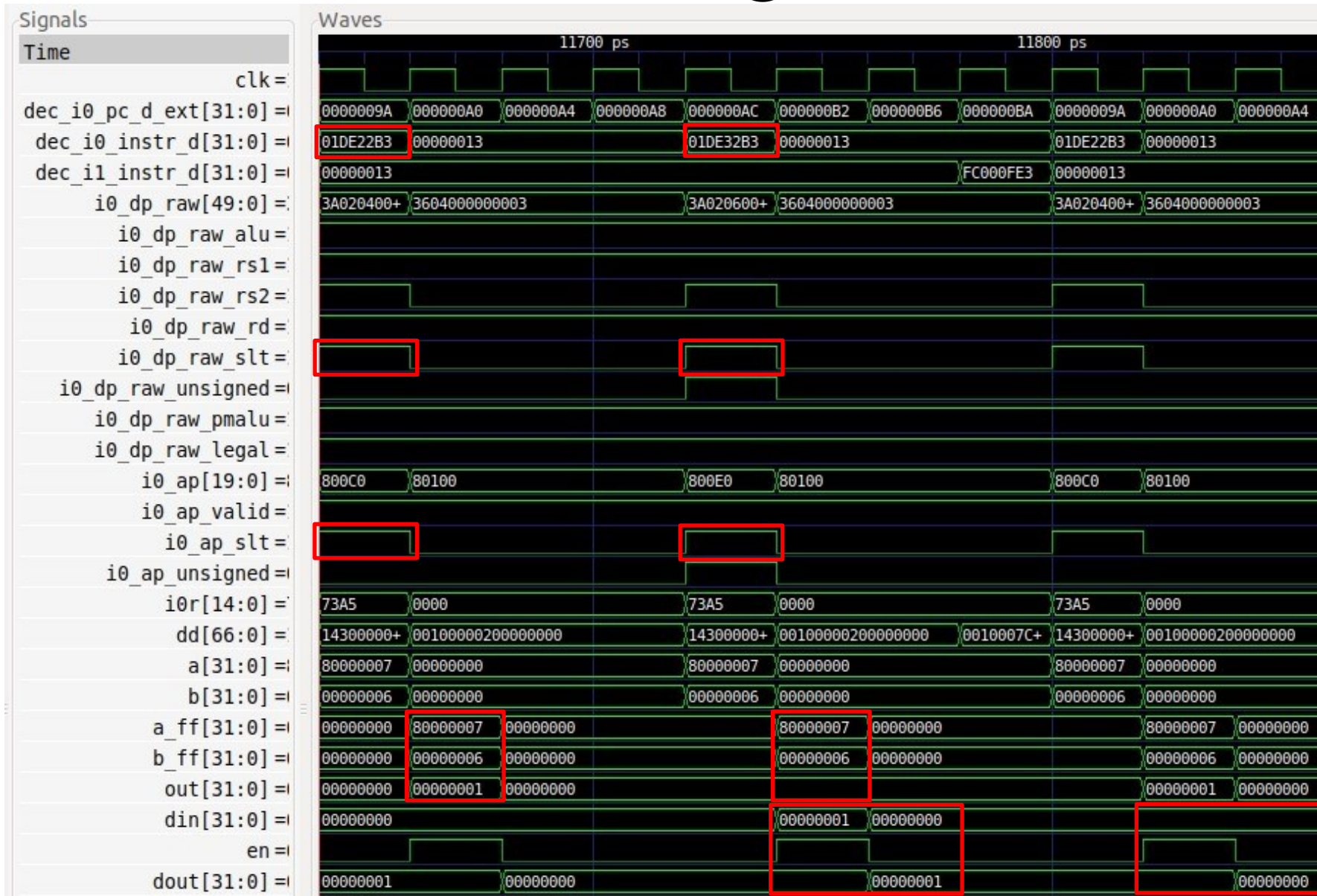
```
#define INSERT_NOPS_0
#define INSERT_NOPS_1    nop; INSERT_NOPS_0
#define INSERT_NOPS_2    nop; INSERT_NOPS_1
#define INSERT_NOPS_3    nop; INSERT_NOPS_2
#define INSERT_NOPS_4    nop; INSERT_NOPS_3
#define INSERT_NOPS_5    nop; INSERT_NOPS_4
#define INSERT_NOPS_6    nop; INSERT_NOPS_5
#define INSERT_NOPS_7    nop; INSERT_NOPS_6
#define INSERT_NOPS_8    nop; INSERT_NOPS_7
#define INSERT_NOPS_9    nop; INSERT_NOPS_8
#define INSERT_NOPS_10   nop; INSERT_NOPS_9

.globl main
main:

li t3, 0x80000007
li t4, 0x6

REPEAT:
    slt  t0, t3, t4
    INSERT_NOPS_7
    sltu t0, t3, t4
    INSERT_NOPS_6
    beq  zero, zero, REPEAT # Repete o ciclo

.end
```



Os segmentos de código Verilog a seguir mostram a lógica que executa essas operações no SweRV EH1.

```

135   assign bm[31:0] = ( ap.sub ) ? ~b_ff[31:0] : b_ff[31:0];
136
137
138   assign {cout, aout[31:0]} = {1'b0, a_ff[31:0]} + {1'b0, bm[31:0]} + {32'b0, ap.sub};
139
140   assign ov = (~a_ff[31] & ~bm[31] & aout[31]) |
141             ( a_ff[31] & bm[31] & ~aout[31] );
142
143   assign neg = aout[31];
144

```

```

177   assign lt = (~ap.unsign & (neg ^ ov)) |
178             ( ap.unsign & ~cout);
179
180   assign ge = ~lt;
181
182
183   assign slt_one = (ap.slt & lt);
184
185   assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                     ({32{sel_shift}} & sout[31:0]) |
187                     ({32{sel_adder}} & aout[31:0]) |
188                     ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1], 1'b0}) |
189                     ({32{ap.csr_write}} & ((ap.csr_imm) ? b_ff[31:0] : a_ff[31:0])) |
190                     ({31'b0, slt_one});
191

```

5) Analise, tanto numa simulação do Verilator quanto diretamente no código Verilog, algumas das instruções *imediatas* disponíveis no Instruction Set do RV32I Base Integer: `addi`, `andi`, `ori`, `xori`, `srli`, `srai`, `slli`, `slti` e `sltui`.

```

#define INSERT_NOPS_0
#define INSERT_NOPS_1    nop; INSERT_NOPS_0
#define INSERT_NOPS_2    nop; INSERT_NOPS_1
#define INSERT_NOPS_3    nop; INSERT_NOPS_2

```

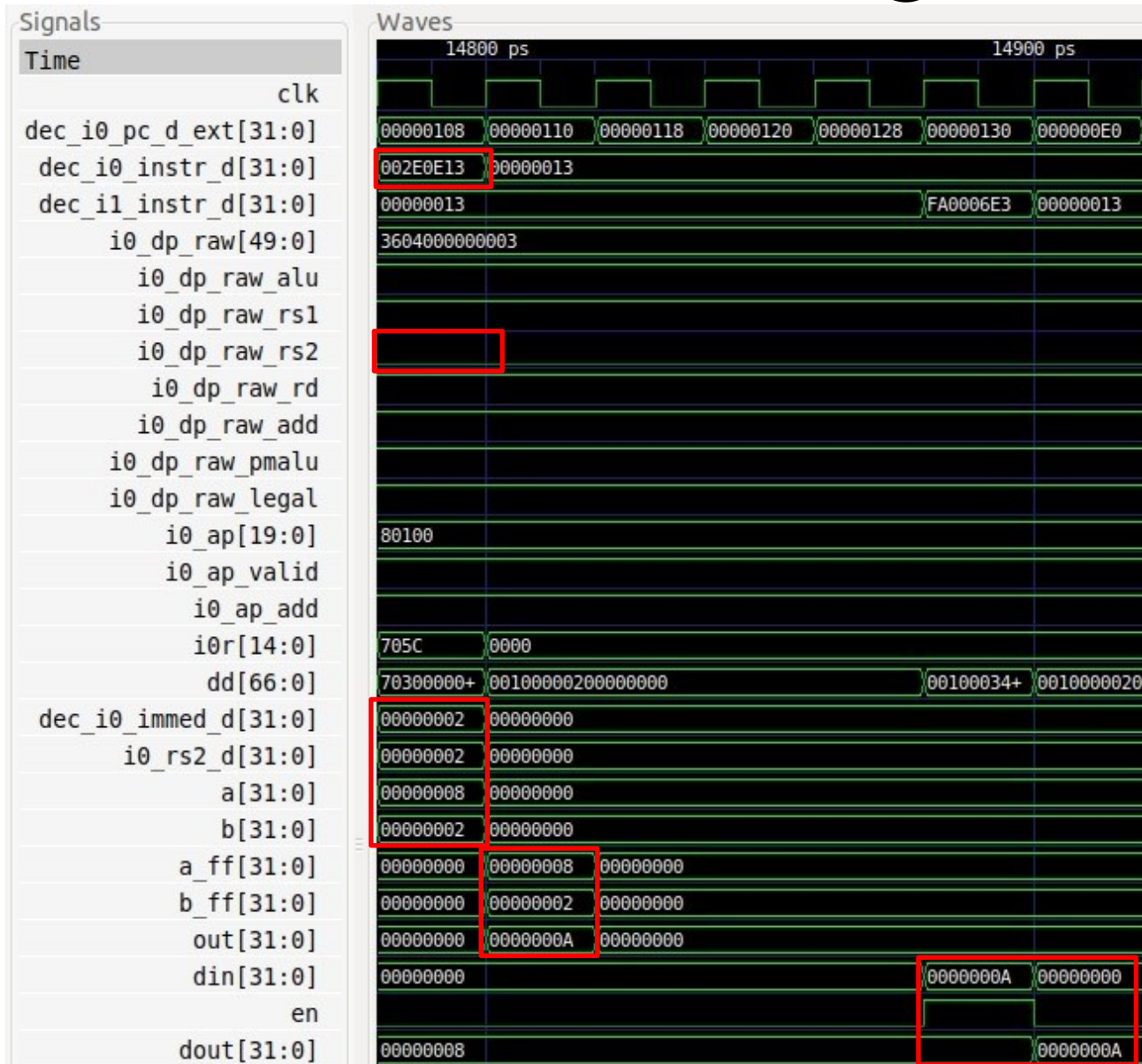
```
#define INSERT_NOPS_4    nop; INSERT_NOPS_3
#define INSERT_NOPS_5    nop; INSERT_NOPS_4
#define INSERT_NOPS_6    nop; INSERT_NOPS_5
#define INSERT_NOPS_7    nop; INSERT_NOPS_6
#define INSERT_NOPS_8    nop; INSERT_NOPS_7
#define INSERT_NOPS_9    nop; INSERT_NOPS_8
#define INSERT_NOPS_10   nop; INSERT_NOPS_9

.globl main
main:

li t3, 0x4                # t3 = 4
INSERT_NOPS_1

REPEAT:
    INSERT_NOPS_10
    addi t3, t3, 2        # t3 = t3 + 2
    INSERT_NOPS_10
    beq zero, zero, REPEAT # Repete o ciclo

.end
```



No módulo **dec_decode_ctl**, o imediato de 32 bits é computado.

```

1231 // read the csr value through rs2 immed port
1232 assign dec_i0_immed_d[31:0] = ({32{ i0_dp.csr_read}} & dec_csr_rddata_d[31:0]) |
1233                               ({32{~i0_dp.csr_read}} & i0_immed_d[31:0]);
1234
1235 // end csr stuff
1236
1237 assign i0_immed_d[31:0] = ({32{i0_dp.imm12}} & { {20{i0[31]}}, i0[31:20] }) | // jalr
1238                           ({32{i0_dp.shimm5}} & {27'b0, i0[24:20]}) |
1239                           ({32{i0_jalimm20}} & {12{i0[31]}, i0[19:12], i0[20], i0[30:21], 1'b0}) |
1240                           ({32{i0_uiimm20}} & {i0[31:12], 12'b0 }) |
1241                           ({32{i0_csr_write_only_d & i0_dp.csr_imm}} & {27'b0, i0[19:15]}); // for csr's that only write csr, dont read csr
1242

```

No módulo **exu**, a entrada *rs2* adequada é selecionada. Nesse caso, usamos *dec_i0_immed_d*.

```

286 assign i0_rs2_d[31:0] = ({32{~dec_i0_rs2_bypass_en_d}} & gpr_i0_rs2_d[31:0]) |
287                           ({32{~dec_i0_rs2_bypass_en_d}} & dec_i0_immed_d[31:0]) |
288                           ({32{ dec_i0_rs2_bypass_en_d}} & i0_rs2_bypass_data_d[31:0]);

```

No módulo **dec_gpr_ctl**, o sinal de habilitação *rden1* determina se o ficheiro de registo é acessado para o segundo operando ou não. Se uma instrução usar um operando imediato: $i0_dp.rs2=0 \rightarrow rden1=0 \rightarrow rd1[31:0]=0x00000000 \rightarrow gpr_i0_rs2_d[31:0]=0x00000000$.

```

90 rd1[31:0] |= ({32{rden1 & (raddr1[4:0]== 5'(j)) & (gpr_bank_id[GPR_BANKS_LOG2-1:0] == 1'(i))}} & gpr_out[i][j][31:0]);

```

6) (O exercício a seguir é baseado no exercício 4.6 de [HePa]).

A Figura 5 não aborda instruções do tipo I, como *addi* ou *andi*.

- Quais blocos lógicos adicionais, se houver, são necessários para dar suporte à execução de instruções do tipo I no SweRV EH1? Adicione os blocos lógicos necessários à Figura 5 e explique sua finalidade.
- Liste os valores dos sinais gerados pela unidade de controlo para *addi*.

Uma das entradas para os dois multiplexers 3:1 no andar Decode vem do imediato no sinal *dec_i0_immed_d[31:0]*. O imediato é um sinal de 32 bits que é computado de forma diferente, dependendo da instrução I-Type que é executada. É um subconjunto de 32 bits que compõem a instrução, que são selecionados e estendidos por sinal da seguinte forma:

```

1231 // read the csr value through rs2 immed port
1232 assign dec_i0_immed_d[31:0] = ({32{i0_dp.csr_read}} & dec_csr_rddata_d[31:0]) |
1233                               ({32{~i0_dp.csr_read}} & i0_immed_d[31:0]);
1234
1235 // end csr stuff
1236
1237 assign i0_immed_d[31:0] = ({32{i0_dp.imm12}} & {20{i0[31]},i0[31:20]}) | // jalr
1238                           ({32{i0_dp.shimm5}} & {27'b0, i0[24:20]}) |
1239                           ({32{i0_jalimm20}} & {12{i0[31]},i0[19:12],i0[20],i0[30:21],1'b0}) |
1240                           ({32{i0_uiimm20}} & {i0[31:12],12'b0}) |
1241                           ({32{i0_csr_write_only_d} & i0_dp.csr_imm}} & {27'b0,i0[19:15]}); // for csr's that only write csr, dont read csr
1242

```

Os valores dos sinais de controlo para o `addi` podem ser vistos na simulação do Exercício 5.

7) (O exercício a seguir é baseado no exercício 4.4 de [HePa] e no exercício 1 do Capítulo 7 do livro de S. Harris e D. Harris, "Digital Design and Computer Architecture": RISC-V Edition" [DDCARV]).

Quando os chips de silício são fabricados, os defeitos nos materiais (por exemplo, silício) e os erros de fabricação podem resultar em circuitos defeituosos. Um defeito muito comum é uma ligação de um sinal ficar "quebrada" e registrar sempre um 0 lógico, o que é chamado de falha permanente ou "stuck-at-0".

Determine o efeito de cada um dos bits de controlo incluídos no sinal `i0_ap` (um sinal do tipo `alu_pkt_t`) ficar preso em 0.

O tipo de estrutura é definido no ficheiro `swerv_types.sv`:

```

typedef struct packed {
    logic valid;
    logic land;
    logic lor;
    logic lxor;
    logic sll;
    logic srl;
    logic sra;
    logic beq;
    logic bne;
    logic blt;
    logic bge;
    logic add;
    logic sub;
    logic slt;
    logic unsign;

```

```
logic jal;  
logic predict_t;  
logic predict_nt;  
logic csr_write;  
logic csr_imm;  
} alu_pkt_t;
```

- Sinal `valid` em "stuck-at-0": não seria possível executar nenhuma instrução A-L, pois qualquer instrução A-L seria considerada inválida.
- Sinais `land`, `lor`, `lxor`, `sll`, `srl`, `sra`, `beq`, `bne`, `blt`, `bge`, `add`, `sub`, `slt` e `jal` em "stuck-at-0": para cada um desses bits, não seria possível executar a instrução A-L correspondente; por exemplo, se `land` estiver em "stuck-at-0", não será possível executar uma instrução `and`.
- Sinal `unsign` em "stuck-at-0": não seria possível comunicar ao processador que a operação deve ser sem sinal.
- Sinais `predict_t` e `predict_nt`: Não seria possível comunicar ao processador que um salto foi previsto como tomado ou não tomado.
- Sinais `csr_write` e `csr_imm`: não seria possível escrever ou operar com um imediato no Registo CSR.