

任务

任务：验证这32位（0x01de0e33）是否对应于RISC-V架构中的指令add t3,t3,t4。

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


来自DDCARV的附录B:

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

任务： 在自己的计算机上重复图3中的仿真过程。为此，请按照以下步骤操作（在GSG的第7部分中详述）：

- 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/Lab12/ADD_Instruction*。
- 在文件*platformio.ini*中建立到RVfpga仿真二进制文件（*Vrvfpgasim*）的正确路径。
- 使用Verilator生成仿真轨迹（生成轨迹）。
- 在GTKWave上打开轨迹。
- 使用文件*test_1.tcl*（在*[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/*中提供）打开与图3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*test_1.tcl*文件。
- 单击几次“*Zoom In*”（放大）（）移动至15000 ps。

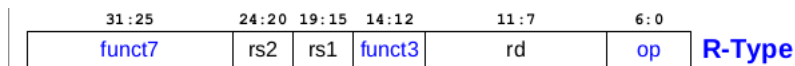
解答请参见实验12的主文档。

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

- 模块**dec_decode_ctl**中的控制单元
- 寄存器文件：
 - 模块**dec**第525行中的实例化。
 - 模块**dec_gpr_ctl**中的实现。
- 译码阶段的3:1多路开关：模块**exu**的第279行。
- 控制信号的流水线寄存器：分布在多个模块中。
- 寄存器**aff**和**bff**：模块**exu_alu_ctl**的第90行和第92行。
- EX1阶段的I0 ALU：
 - 模块**exu**第401行中的实例化。
 - 模块**exu_alu_ctl**中的实现。
- 包含运算结果的流水线寄存器（**i0e2resultff**、**i0e3resultff**、**i0e4resultff**和**i0wbresultff**）：模块**dec_decode_ctl**的第2260-2283行。
- EX3阶段的3:1多路开关：模块**dec_decode_ctl**的第2268行。
- EX4阶段的3:1多路开关：模块**dec_decode_ctl**的第2277行。
- 回写阶段的2:1多路开关：模块**dec_decode_ctl**的第2286行。

任务：在Verilog代码（模块**dec_decode_ctl**）中查找如何使用**i0r**控制信号读取寄存器文件。

- 寄存器标识符从通路0中的32位指令获得：信号**i0[31:0] = dec_i0_instr_d[31:0]**。
在R型指令中，它们位于以下字段中：



在模块**dec_decode_ctl**中:

```
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];
```

- 寄存器标识符和读使能信号分配给dec_i0_rs1_d/dec_i0_rs2_d和dec_i0_rs1_en_d/dec_i0_rs2_en_d。这些信号从模块**dec**发送到模块**dec_decode_ctl**。在模块**dec_decode_ctl**中:

```
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];
```

- 寄存器标识符和读使能信号提供给寄存器文件，该文件在模块**dec**中实例化。在模块**dec**中:

```
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               );
```

任务: 在Verilog代码（模块**exu**）中查找i0_ap和dd控制信号如何从译码阶段传播到执行阶段。此外，还需查找dd控制信号遍历译码到回写的所有阶段之后，如何在回写阶段被寄存器文件使用。

信号*i0_ap*在模块**dec_decode_ctl**中获得。它提供给模块**exu**，并从中传播到EX1、EX2、EX3和提交（EX4）阶段。在模块**exu**中：

```
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) );
```

信号*dd*在模块**dec_decode_ctl**中获得，并传播到EX1、EX2、EX3、提交（EX4）和WB（EX5）阶段。在模块**dec_decode_ctl**中：

```
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));
```

请注意，在进入下一个寄存器之前，每个寄存器的输出都会稍作修改（并因此重命名）。如果要查看详细信息，可以查看Verilog代码。

输出操作数的寄存器标识符在译码阶段分配：

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

信号*dd*从译码阶段传播到回写阶段（如上所示）：*dd* → *e1d* → *e2d* → *e3d* → *e4d* → *wbd*。随后，目标寄存器在回写阶段提供给寄存器文件：

```
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             );

```

任务：这两个信号（`i0_e1_ctl_en`和`dec_i0_alu_decode_d`）的产生过程相当复杂，这里不做详细说明，但您可自行在模块`dec_decode_ctl`和`exu`中进一步分析。

不提供解答。

任务：在Verilog代码（模块`exu`）中查找底部的3:1多路开关（第二个输入操作数）并尝试找到其输入的来源（图6中仅显示来自寄存器文件的输入）。不需要太仔细地查看输入，因为它们将在第3部分和后续实验提供的练习中进行分析。

```

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]);

```

这些3:1多路开关接收3个输入：

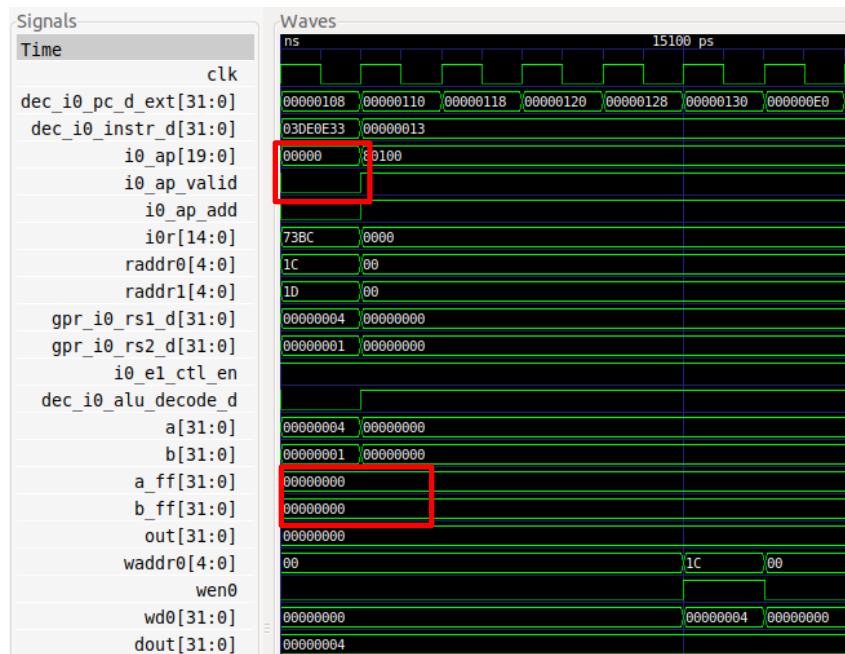
- 一个来自寄存器文件（`gpr_i0_rs2_d`）
- 一个来自32位指令寄存器，它构成立即数（`dec_i0_immed_d`）
- 一个来自旁路逻辑，我们将在实验15中分析（`i0_rs2_bypass_data_d`）

任务：在自己的计算机上重复图7中的仿真过程。可以使用以下位置提供的`.tcl`脚本：
`[RVfpgaPath]/RVfpga/Labs/Lab12/ADD_Instruction/test_2.tcl`。请注意，该`.tcl`文件中为一些控制位使用了别名。

解答请参见实验12的主文档。

任务：在图2的示例中，将add指令替换为非A-L指令（例如mul指令）。验证i0_ap信号的所有字段是否均等于0，等于0时I0 ALU不起作用（对于该指令，EX1阶段I0管道的信号a_ff和b_ff将保持不变）。可以使用与图7中示例所用test_2.tcl文件相同的文件。

例如，mul t3, t3, t4的仿真（0x03de0e33）提供以下结果：

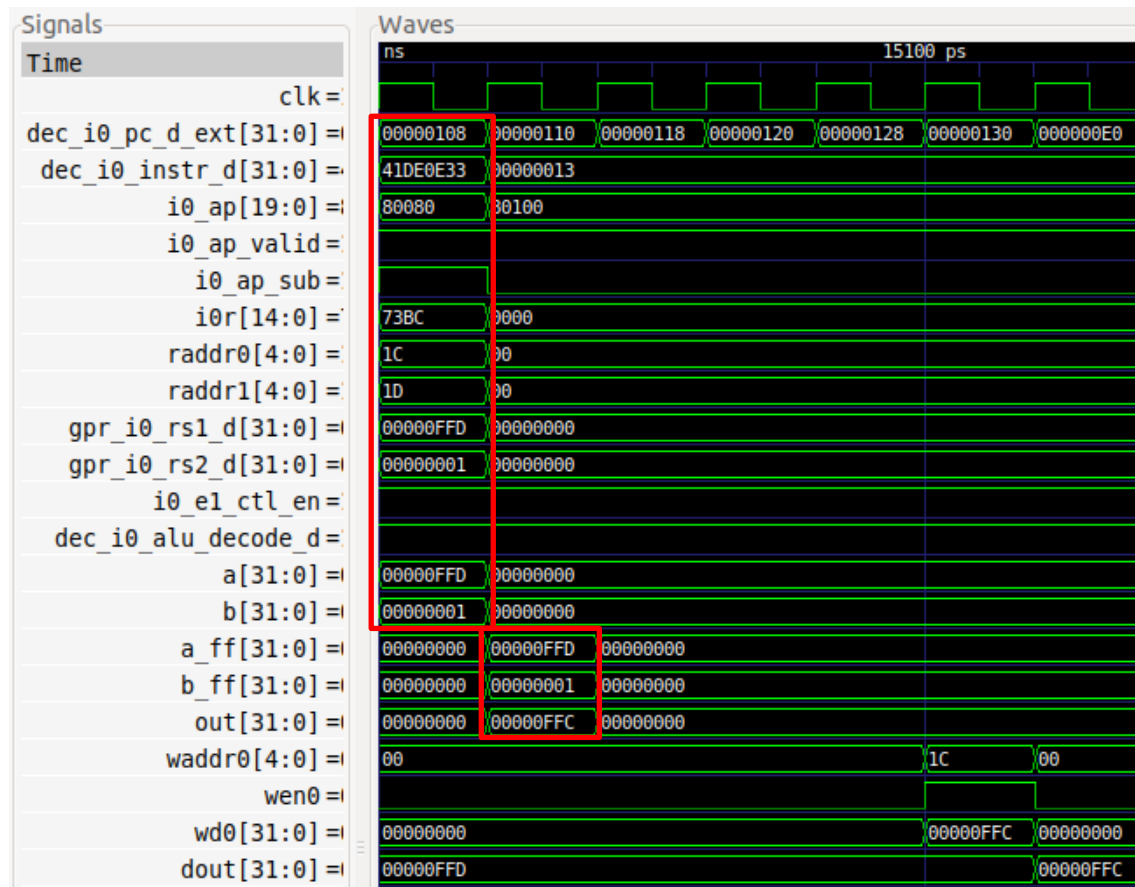


任务：将本部分中分析的新信号包含在图7的仿真中。

不提供解答。

任务：对sub指令执行与图7中的仿真类似的仿真。请记住，可以通过.tcl文件将新信号添加到仿真中。

例如，sub t3, t3, t4的仿真（0x41de0e33）提供以下结果：



任务： 分析模块`exu_alu_ctl`中实现的加法器/减法器的Verilog实现。图8通过显示与加法和减法运算直接相关的逻辑来提供一些帮助。

```
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]));
```

输入操作数从译码阶段（a和b）传播到执行阶段（a_ff和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};
```

这是加法器/减法器。

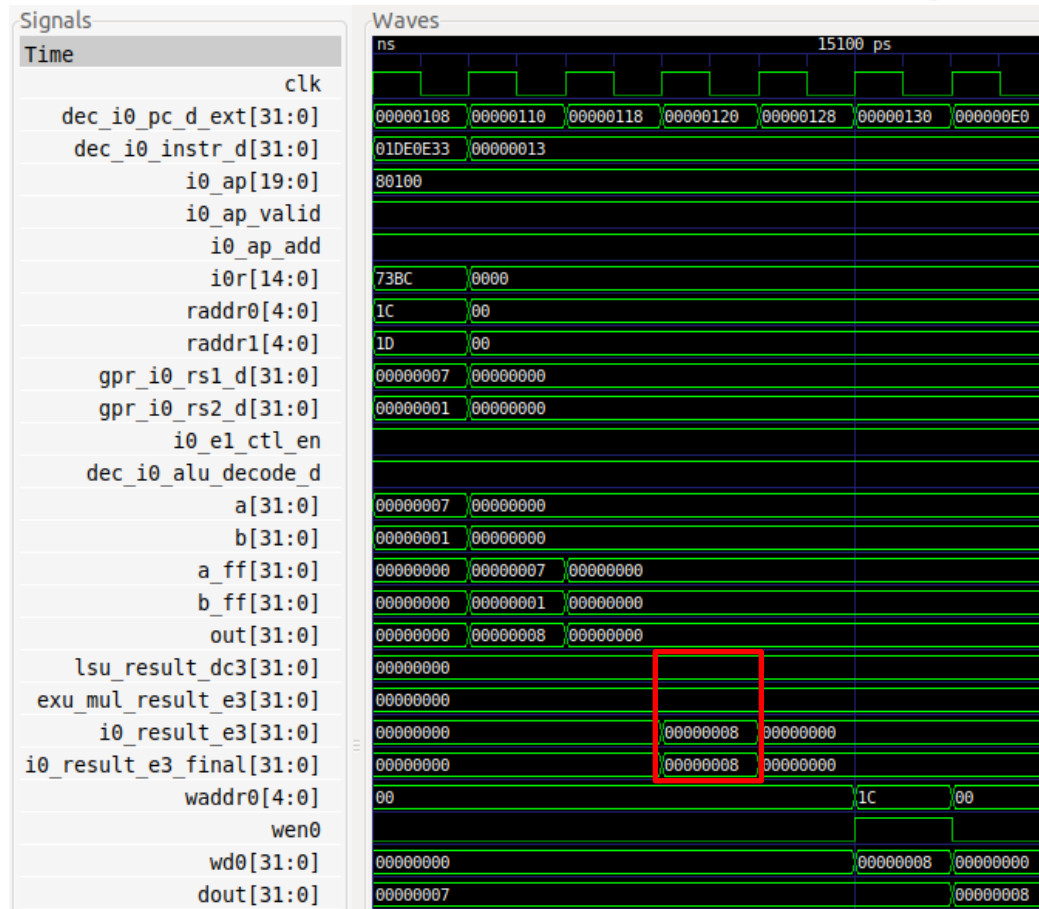
- 如果指令是加法，则`aout = a_ff + b_ff`
- 如果指令是减法，则先计算`b_ff`的二进制补码，然后计算`a_out`。

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

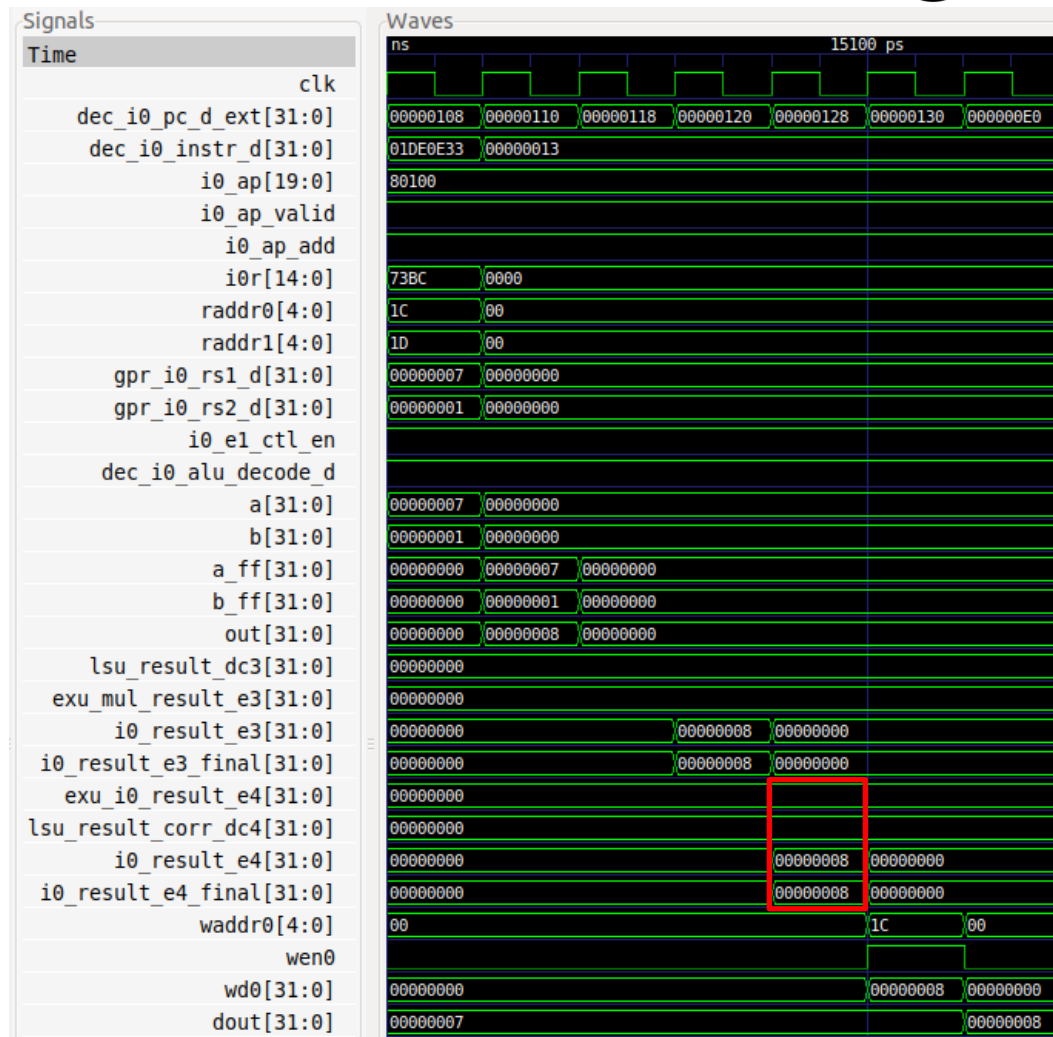
185      assign out[31:0] = ({32{sel_logic}} & lout[31:0]) |
186                        ({32{sel_shift}} & sout[31:0]) |
187                        ({32{sel_adder}} & aout[31:0]) |
188                        ({32{ap.jal | pp_ff.pcall | pp_ff.pja | pp_ff.pret}} & {pcout[31:1], 1'b0}) |
189           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

如果指令是加法或减法，则`out = aout`。

任务： 对于图2中的示例，在仿真中验证该多路开关是否从add指令的预期管道中选择结果。



任务：对于图2中示例的add指令，在仿真中验证该多路开关是否从正确的输入源选择结果（i0_result_e4）。



任务： 在Verilog代码中，分析信号wen0和waddr0如何在译码阶段生成并传播到回写阶段。

```

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;

```

练习

- 1) 对逻辑指令（and、or和xor）执行与本实验中提供的分析类似的分析。

以下示例（在[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab12/AND_Instruction中提供）说明了无限循环中包含的and指令的执行情况。与add指令的示例中一样，and指令（以红色突出显示）前后有几条nop指令。循环末尾包含两条指令，用于修改存储在t3和t4中的值。

```
#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

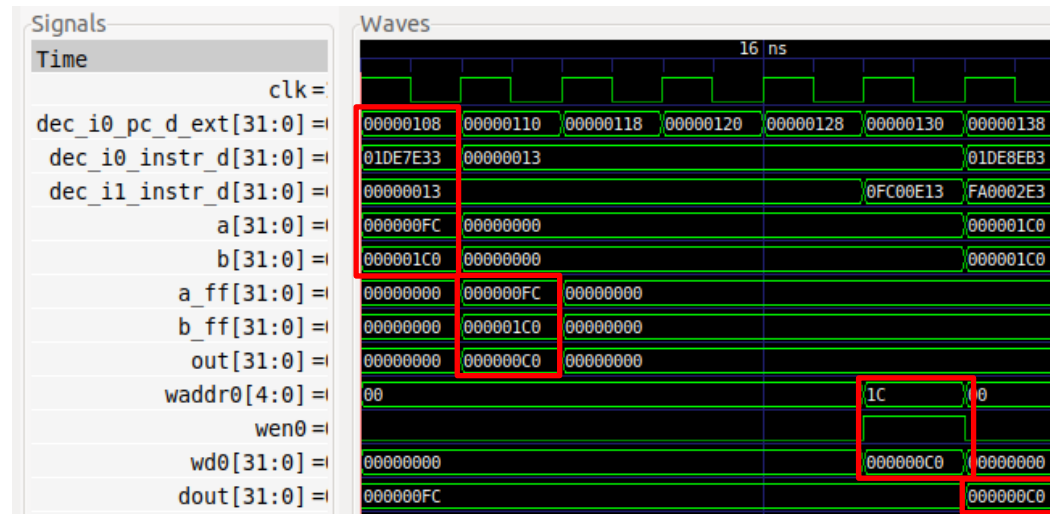
REPEAT:
    INSERT_NOPS_10
    and t3, t3, t4          # t3 = t3 & t4
    INSERT_NOPS_10
    li t3, 0xFC             # t3 = 0xFC
    add t4, t4, t4
    beq zero, zero, REPEAT # Repeat the loop

.end
```

如果在PlatformIO中打开、编译项目，然后打开反汇编文件（位于[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs_Solutions/Lab12/AND_Instruction/.pio/build/swervolf_nexys/firmware.dis中），可以看到and指令位于地址0x00000108处，还可以看到指令的机器代码（0x01de7e33）：

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

接下来，我们在Verilator中仿真程序，然后在GTKWave上打开仿真器生成的跟踪文件。移至循环的任何一次迭代（第一次除外）。



分析波形（以红色突出显示的值对应于and指令）。在本实验中，我们将跳过取指和对齐阶段，这两个阶段将在后面的实验中说明。

- **译码阶段：**信号dec_i0_pc_d_ext包含指令的地址（在教材中，该信号通常称为程序计数器），and的地址为0x00000108，信号dec_i0_instr_d包含32位机器指令0x01DE7E33（在教材中，该信号通常称为指令寄存器）。

在RISC-V中，and指令的操作码如下（参见[Harris&Harris]的附录B）：

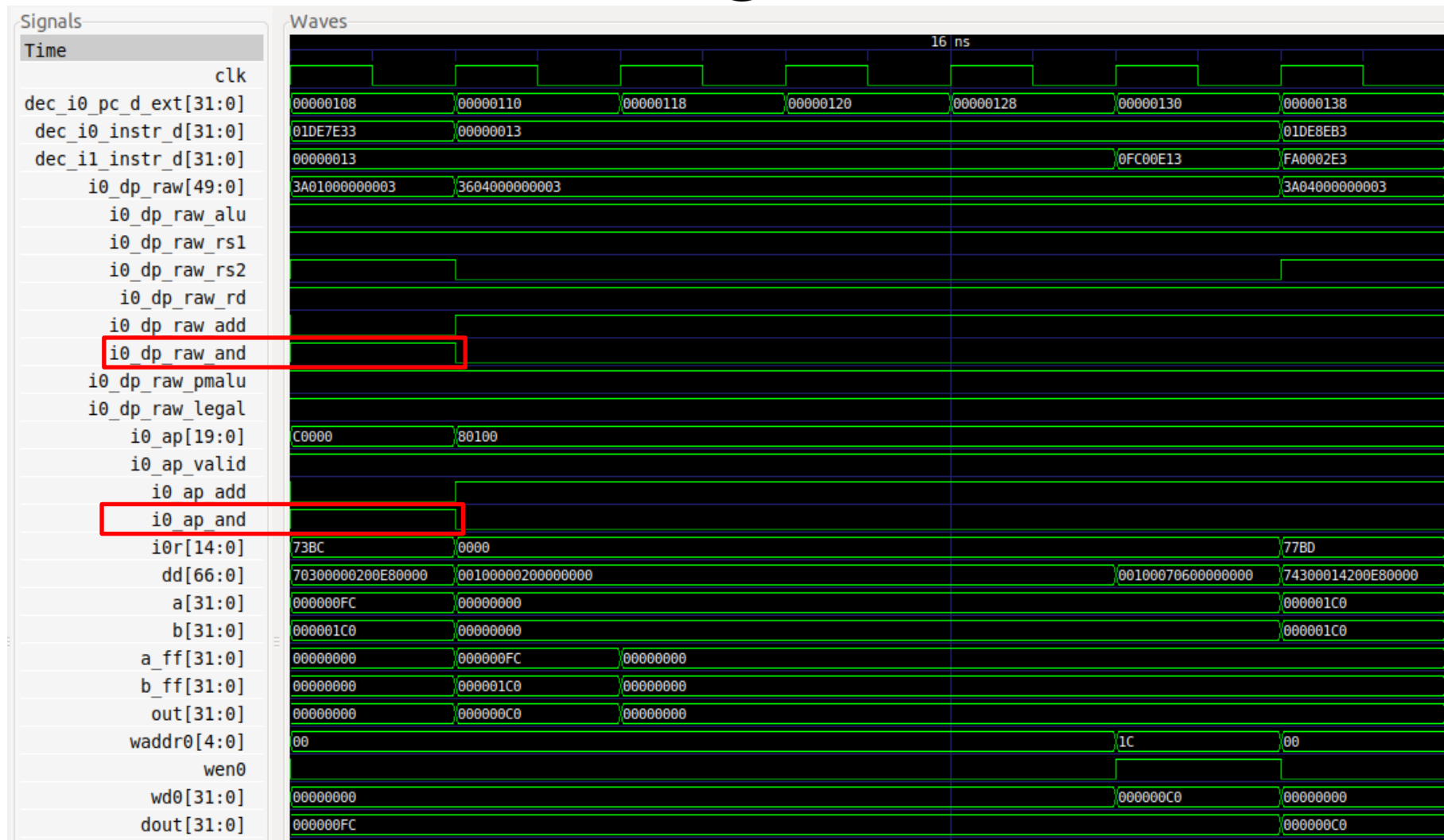
```
00000000 | rs2 | rs1 | 111 | rd | 0110011
```

因此可以轻松验证0x01DEFE33是否对应于：and t3, t3, t4（请记住，t3=x28且t4=x29）。

在此阶段将产生流水线控制信号（我们将在下一部分中详细介绍）。此外，在此阶段还将读取寄存器文件。信号a和b包含ALU的输入，本例中与从寄存器文件读取的值一致（对于后续实验中将分析的其他示例，情况并非如此）。

- **EX1阶段**：在下一周期中，将执行and指令。信号a_ff和b_ff包含ALU的输入（分别为0xFC和0x1C0），而out包含加法的结果（0xC0）。
- **EX5阶段（也称为回写）**：最后，在4个周期后，加法结果通过信号wd0=0xC0回写到寄存器文件中，其中包含要写入的数据。鉴于wen0=1（写使能），逻辑与运算结果在相应周期结束时写入寄存器x28（寄存器索引，waddr0=0x1C）。可以发现，在接下来的周期（图中最后一个周期）中，寄存器x28包含新值（dout=0xC0）。

接下来，我们将控制信号添加到之前的仿真中：



可以看到，在第一个周期中，and指令的控制位为1。

以下Verilog片段显示了SweRV EH1的逻辑单元。

```

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]));
93
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
168     assign sel_logic = {ap.land,ap.lor,ap.lxor};
169
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

```

当and控制位为1时，选择逻辑与运算的结果：

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

2) (以下练习基于《计算机组织结构和设计》(RISC-V版本, 作者Patterson & Hennessy ([HePa])) 中的练习4.1。)

请看下面的指令: `and rd, rs1, rs2`

- a. SweRV EH1为该指令生成的控制信号的值是多少?
- b. 哪些资源(块)对该指令执行有用的功能?
- c. 哪些资源(块)不为该指令产生输出? 哪些资源产生不使用的输出?

不提供解答。

3) 在Verilator仿真中以及直接在Verilog代码中分析RV32I基本整数指令集中提供的*shift left/right*指令: srl、sra和sll。

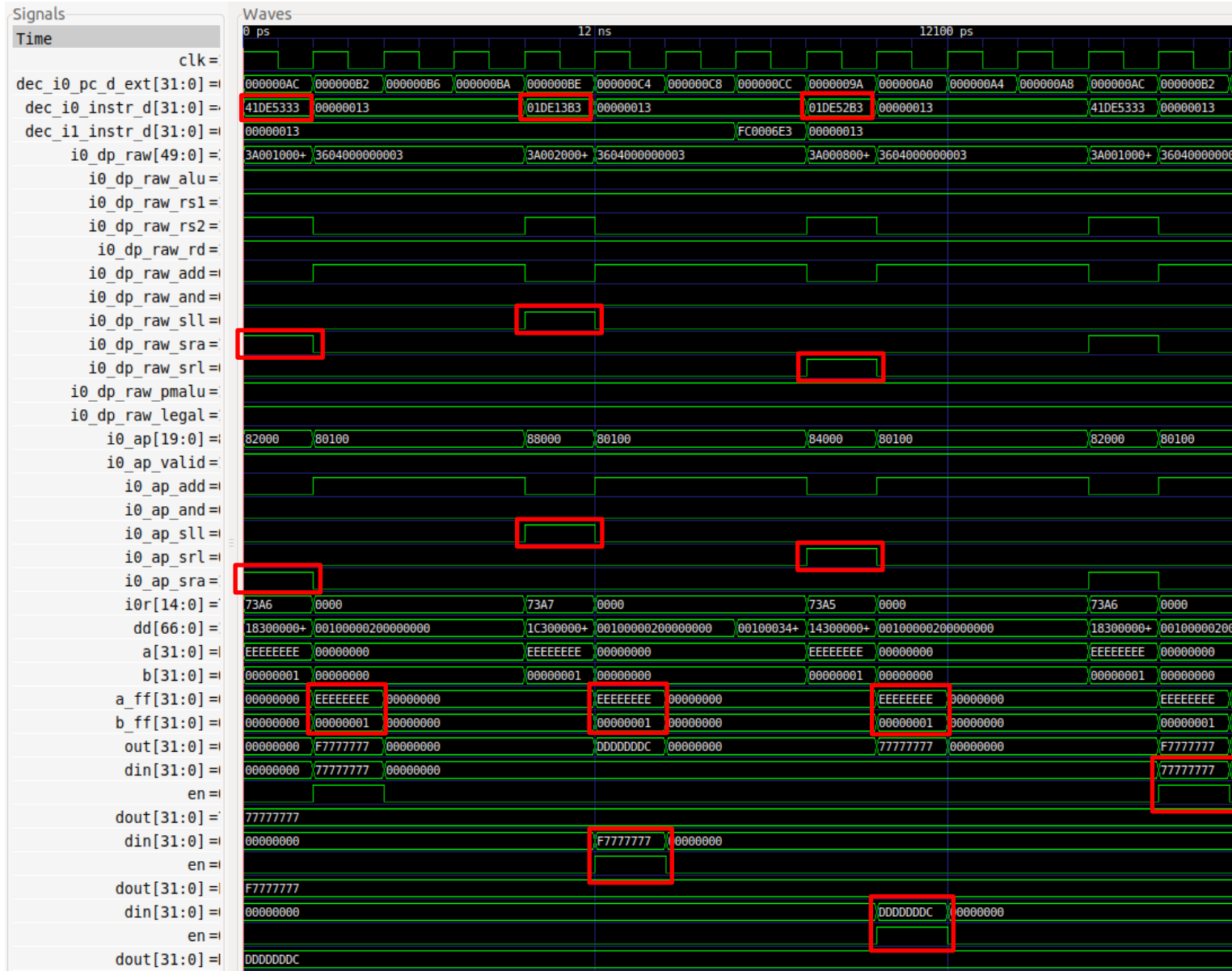
```
#define INSERT_NOPS_0
#define INSERT_NOPS_1      nop; INSERT_NOPS_0
#define INSERT_NOPS_2      nop; INSERT_NOPS_1
#define INSERT_NOPS_3      nop; INSERT_NOPS_2
#define INSERT_NOPS_4      nop; INSERT_NOPS_3
#define INSERT_NOPS_5      nop; INSERT_NOPS_4
#define INSERT_NOPS_6      nop; INSERT_NOPS_5
#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 # Repeat the loop

.end
```



以下Verilog片段显示了SweRV EH1的移位单元。

```
161      assign ashift[31:0] = a_ff >>> b_ff[4:0];
```

```
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] );
```

```
170      assign sel_shift = |{ap.sll,ap.srl,ap.sra};
```

```
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});
```

4) 在Verilator仿真中以及直接在Verilog代码中分析RV32I基本整数指令集中提供的小于则置位指令：slt和sltu。

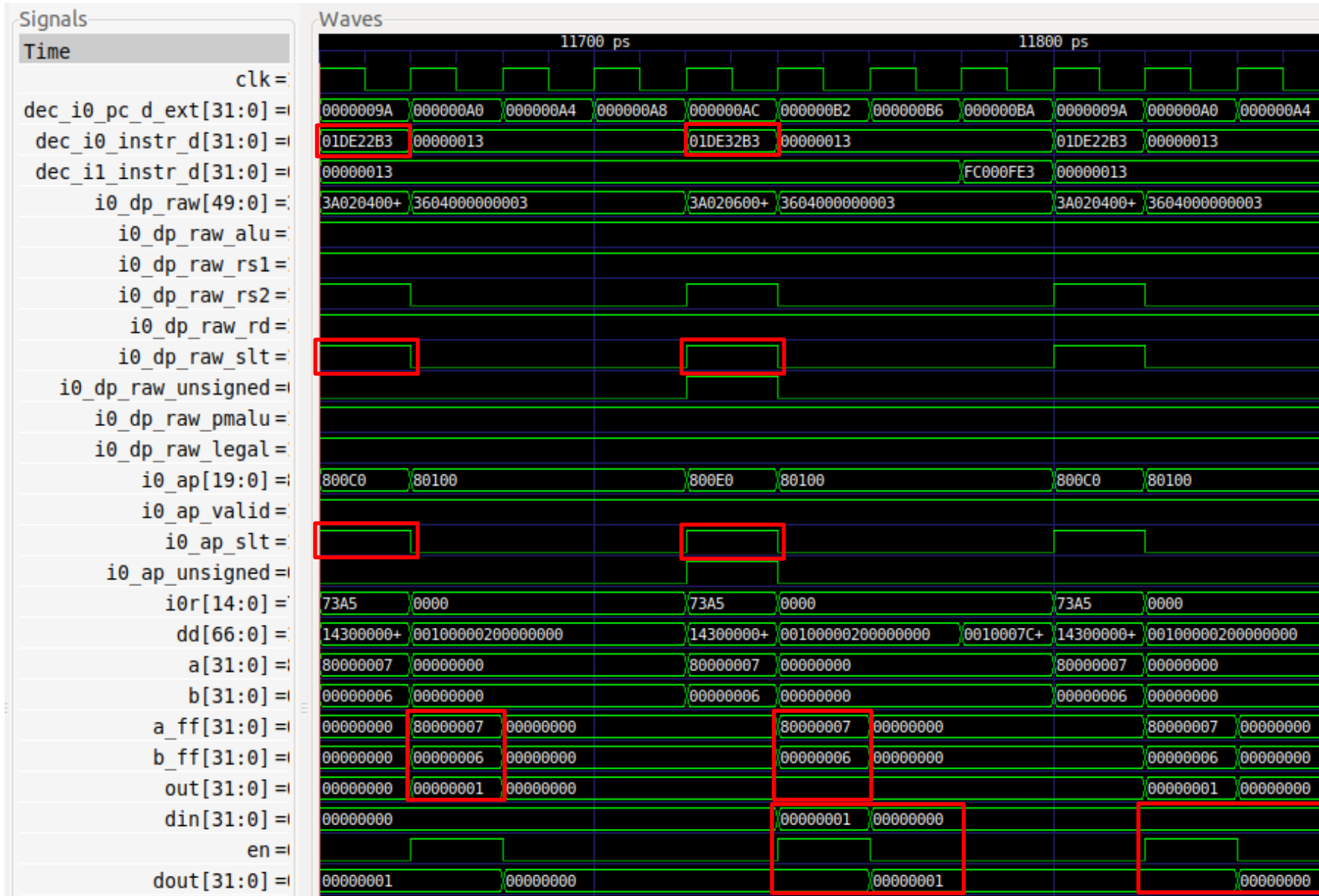
```
#define INSERT_NOPS_0
#define INSERT_NOPS_1      nop; INSERT_NOPS_0
#define INSERT_NOPS_2      nop; INSERT_NOPS_1
#define INSERT_NOPS_3      nop; INSERT_NOPS_2
#define INSERT_NOPS_4      nop; INSERT_NOPS_3
#define INSERT_NOPS_5      nop; INSERT_NOPS_4
#define INSERT_NOPS_6      nop; INSERT_NOPS_5
#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 # Repeat the loop

.end
```



以下Verilog片段显示了在SweRV EH1中执行这些运算的逻辑。

```

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) 在Verilator仿真中以及直接在Verilog代码中分析RV32I基本整数指令集中提供的*immediate*指令：addi、andi、ori、xori、srli、srai、slli、slti和sltui。

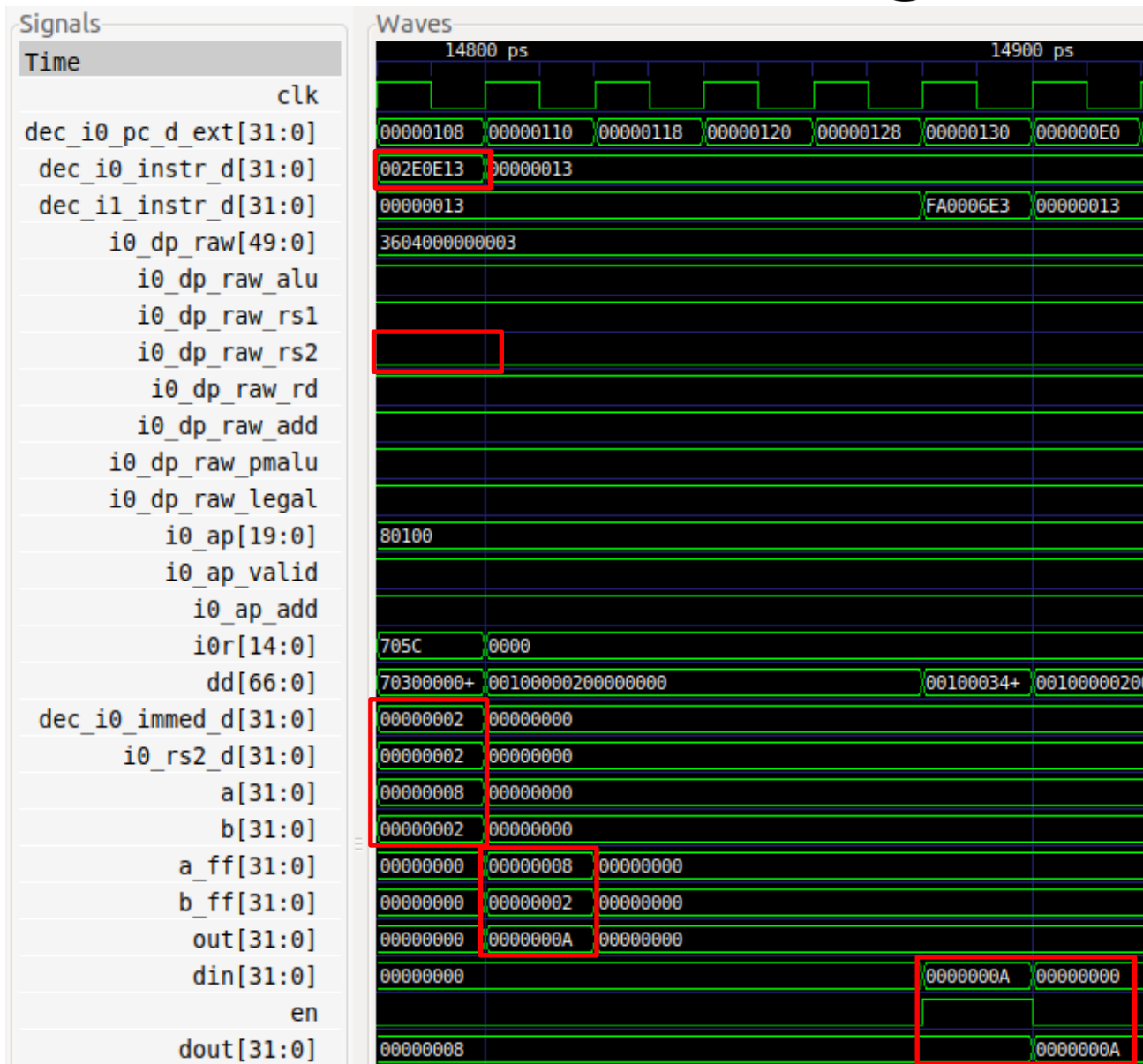
```
#define INSERT_NOPS_0
#define INSERT_NOPS_1      nop; INSERT_NOPS_0
#define INSERT_NOPS_2      nop; INSERT_NOPS_1
#define INSERT_NOPS_3      nop; INSERT_NOPS_2
#define INSERT_NOPS_4      nop; INSERT_NOPS_3
#define INSERT_NOPS_5      nop; INSERT_NOPS_4
#define INSERT_NOPS_6      nop; INSERT_NOPS_5
#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 # Repeat the loop

.end
```



在模块**dec_decode_ctl**中，计算32位立即数。

```

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

```

在模块**exu**中，选择正确的rs2源。本例中我们使用**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]);

```

在模块**dec_gpr_ctl**中，使能信号**rden1**确定是否访问寄存器文件来获取第二个操作数。如果指令使用立即数操作数：**i0_dp.rs2=0 → rden1=0 → rd1[31:0]=0x00000000 → 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) （以下练习基于[HePa]的练习4.4以及S. Harris和D. Harris所编教材《数字设计和计算机体系结构：RISC-V版本》[DDCARV]第7章的练习1。）

制造硅芯片时，材料（如硅）中的缺陷和制造错误会导致有缺陷的电路。一个非常常见的缺陷是一根信号线“损坏”，逻辑始终为0。这通常称为“**stuck-at-0**”（固定为0）故障。确定信号**i0_ap**（**alu_pkt_t**类型）中包含的每个控制位发送“固定为0”故障的影响。

结构类型在文件`swerv_types.sv`中定义:

```
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;
```

- 信号`valid`固定为**0**: 无法执行任何**A-L**指令, 因为任何**A-L**指令都将被视为无效。
- 信号`land`、`lor`、`lxor`、`sll`、`srl`、`sra`、`beq`、`bne`、`blt`、`bge`、`add`、`sub`、`slt`和`jal`固定为**0**: 对于上述每一位, 都无法执行相应的**A-L**指令; 例如, 如果`land`固定为**0**, 将无法执行`and`指令。
- 信号`unsign`固定为**0**: 无法向处理器传达运算必须为无符号运算的信息。
- 信号`predict_t`和`predict_nt`: 无法向处理器传达预测采用或不采用分支的信息。
- 信号`csr_write`和`csr_imm`: 无法在**CSR**寄存器中写入或使用立即数进行运算。

7) (以下练习基于[HePa]的练习4.6。)

图5不讨论**I**型指令, 如`addi`或`andi`。

- 需要哪些额外的逻辑块（如果有）来支持SweRV EH1中I型指令的执行？将所有必要的逻辑块添加到图5并说明其用途。
- 列出addi的控制单元产生的信号的值。

译码阶段两个3-1多路开关的输入之一来自信号dec_i0_immed_d[31:0]中的立即数。立即数是一个32位信号，根据执行的I型指令进行不同的计算。它是组成指令的一个子集（32位），相应位的选择和符号扩展过程如下：

```

1231 // read the csr value through rs2 immid 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_uimm20}} & {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

```

addi的控制信号的值位于练习5的仿真中。