# 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)**

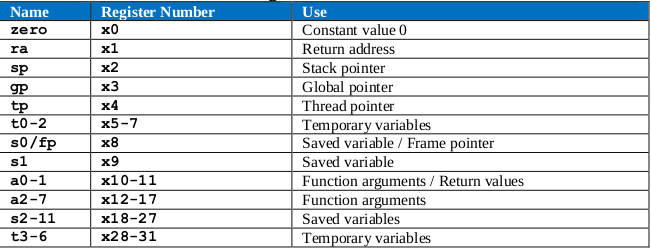
**op = 0110011**

来自DDCARV的附录B：









**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：

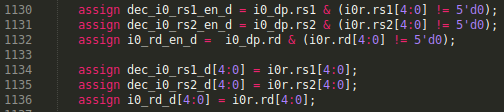


在模块**dec\_decode\_ctl**中：

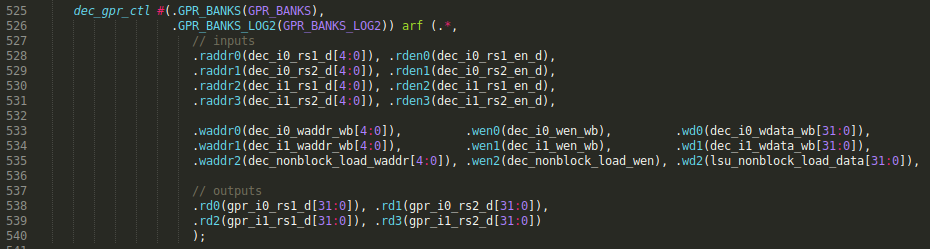


* 寄存器标识符和读使能信号分配给dec\_i0\_rs1\_d/dec\_i0\_rs2\_d和dec\_i0\_rs1\_en\_d/dec\_i0\_rs2\_en\_d。

这些信号从模块**dec**发送到模块**dec\_decode\_ctl**。在模块**dec\_decode\_ctl**中：

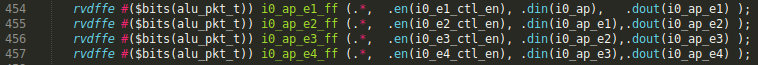


* 寄存器标识符和读使能信号提供给寄存器文件，该文件在模块**dec**中实例化。在模块**dec**中：



**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：



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：











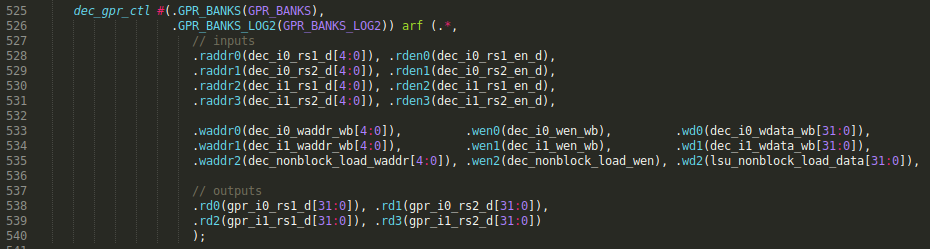
请注意，在进入下一个寄存器之前，每个寄存器的输出都会稍作修改（并因此重命名）。如果要查看详细信息，可以查看Verilog代码。

输出操作数的寄存器标识符在译码阶段分配：



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：





**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。



这些3:1多路开关接收3个输入：

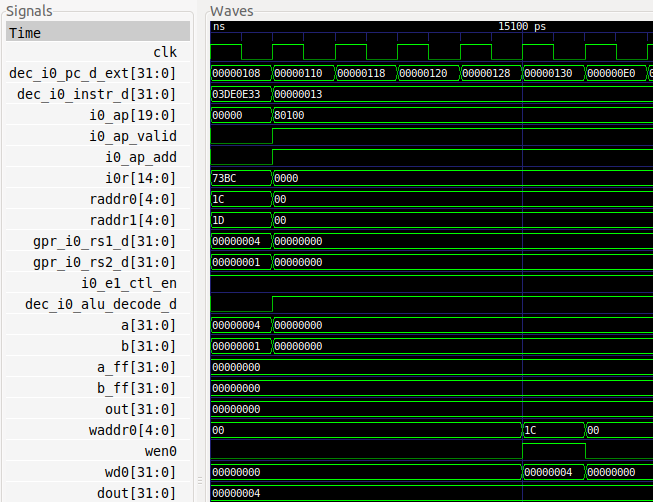
* 一个来自寄存器文件（gpr\_i0\_rs2\_d）
* 一个来自32位指令寄存器，它构成立即数（dec\_i0\_immed\_d）
* 一个来自旁路逻辑，我们将在实验15中分析（i0\_rs2\_bypass\_data\_d）

**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：

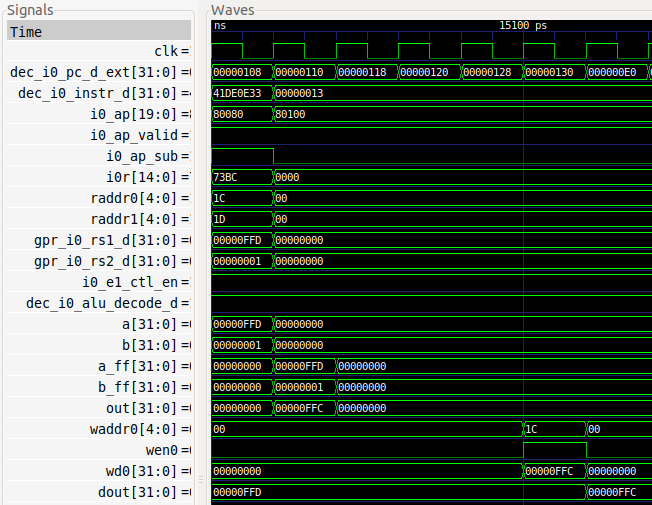


**任务：**将本部分中分析的新信号包含在图7的仿真中。

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。

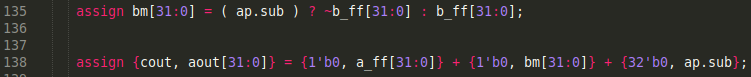
例如，sub t3, t3, t4的仿真（0x41de0e33）提供以下结果：



**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。



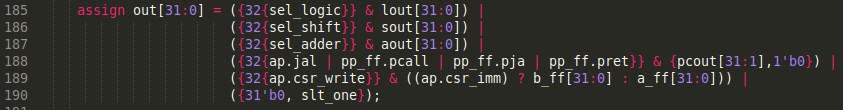
输入操作数从译码阶段（a和b）传播到执行阶段（a\_ff和b\_ff）。



这是加法器/减法器。

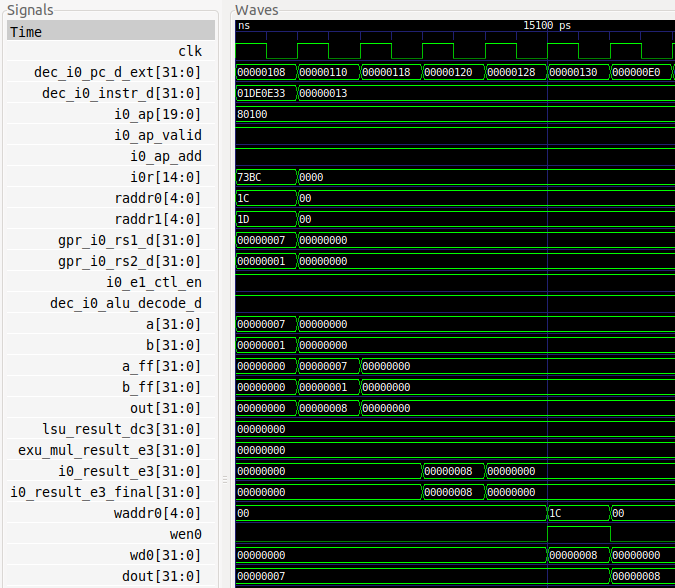
* 如果指令是加法，则aout = a\_ff + b\_ff
* 如果指令是减法，则先计算b\_ff的二进制补码，然后计算a\_out。



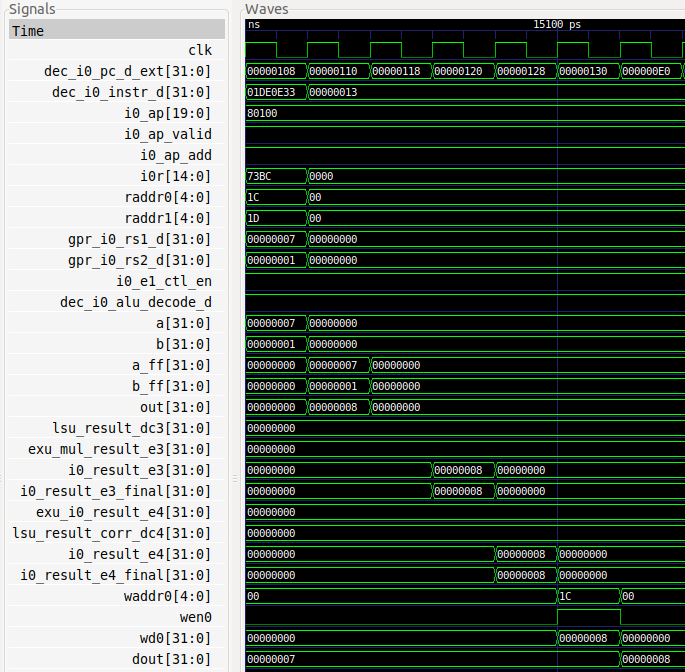


如果指令是加法或减法，则out = aout。

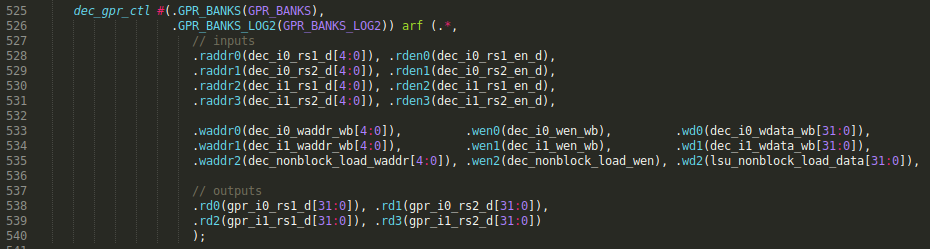
**任务：**对于图2中的示例，在仿真中验证该多路开关是否从add指令的预期管道中选择结果。



**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。









# 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。

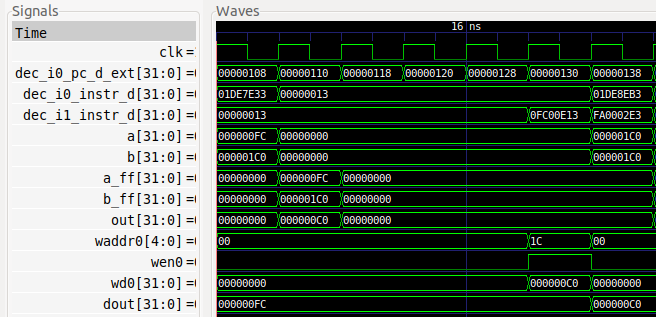
以下示例（在*[RVfpgaPath]/RVfpga/Labs/RVfpgaLabsSolutions/Programs\_Solutions/Lab12/AND\_Instruction*中提供）说明了无限循环中包含的and指令的执行情况。与add指令的示例中一样，and指令（以红色突出显示）前后有几条nop指令。循环末尾包含两条指令，用于修改存储在t3和t4中的值。

|  |
| --- |
| #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）：

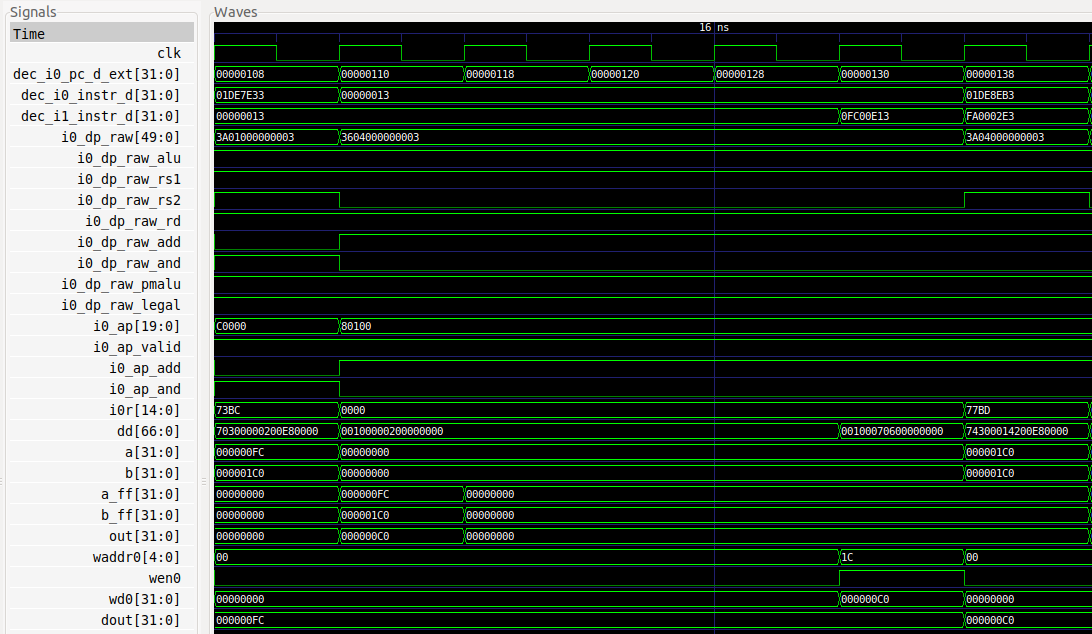
0000000 | rs2 | rs1 | 111 | rd | 0110011

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）。

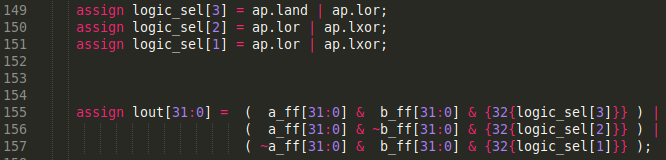
接下来，我们将控制信号添加到之前的仿真中：



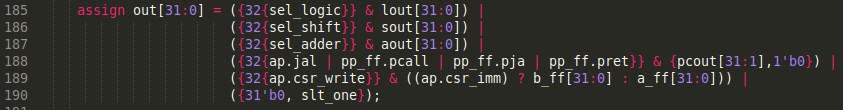
可以看到，在第一个周期中，and指令的控制位为1。

以下Verilog片段显示了SweRV EH1的逻辑单元。









当and控制位为1时，选择逻辑与运算的结果：

logic\_sel[3]=1且logic\_sel[2]=logic\_sel[1]=0 🡪 lout = a\_ff & b\_ff

1. （*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

* 1. SweRV EH1为该指令生成的控制信号的值是多少？
  2. 哪些资源（块）对该指令执行有用的功能？
  3. 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。

1. 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, 0xEEEEEEEE

li t4, 0x1

REPEAT:

srl t0, t3, t4

INSERT\_NOPS\_7

sra t1, t3, t4

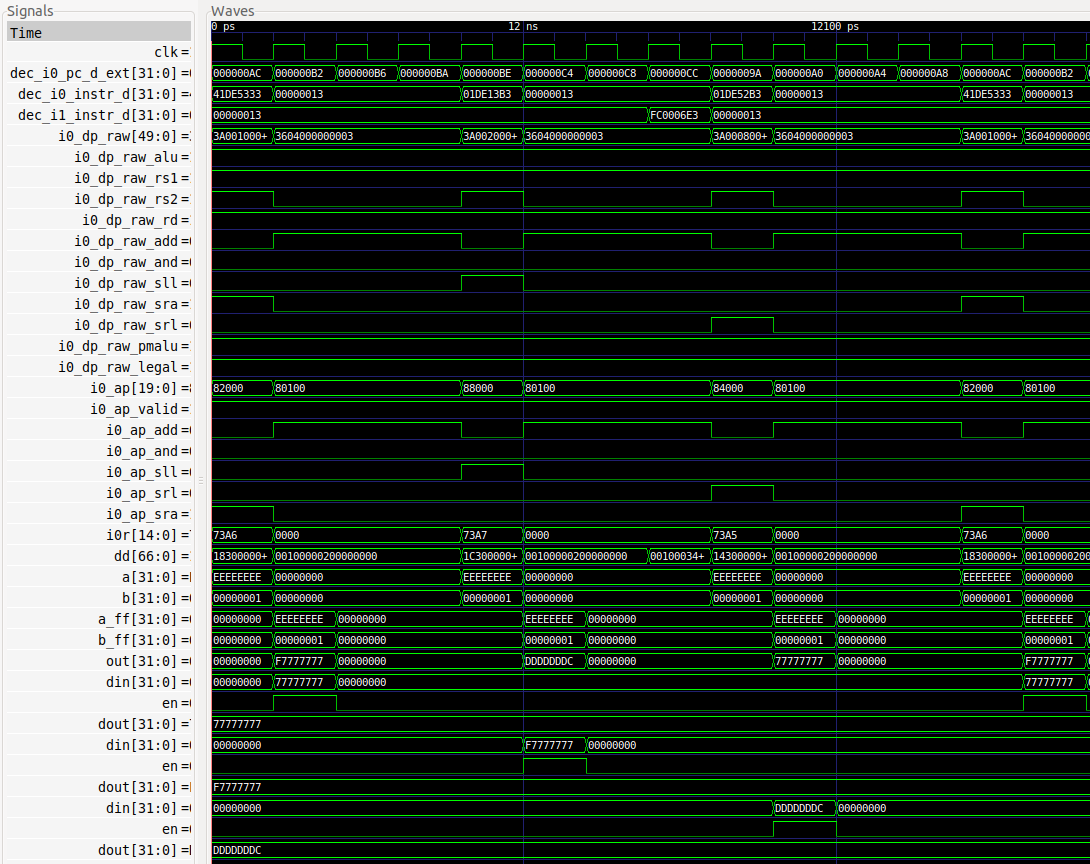
INSERT\_NOPS\_7

sll t2, t3, t4

INSERT\_NOPS\_6

beq zero, zero, REPEAT # Repeat the loop

.end

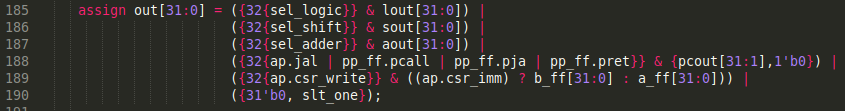


以下Verilog片段显示了SweRV EH1的移位单元。









1. 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

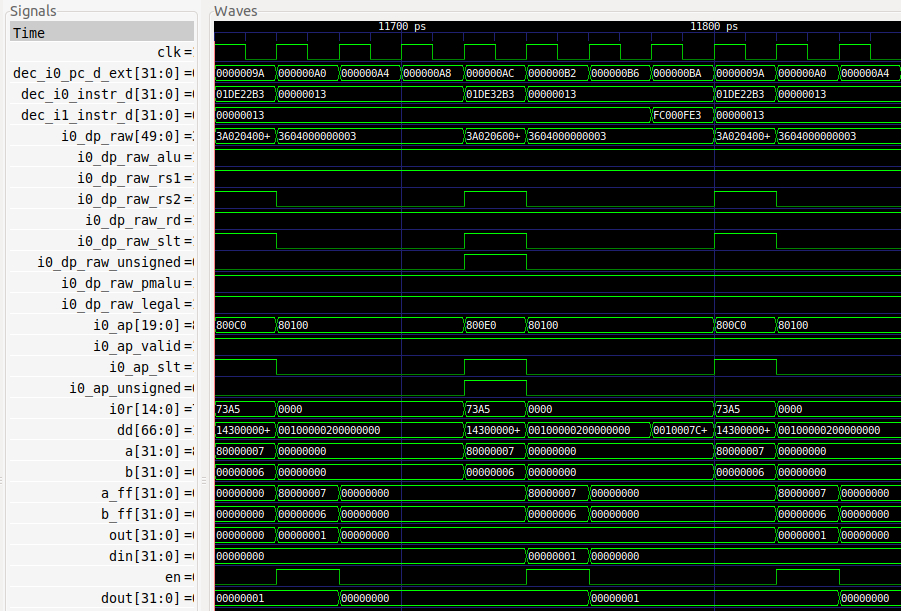
INSERT\_NOPS\_7

sltu t0, t3, t4

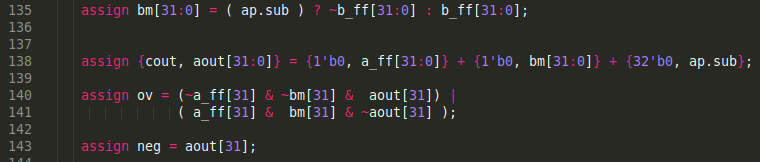
INSERT\_NOPS\_6

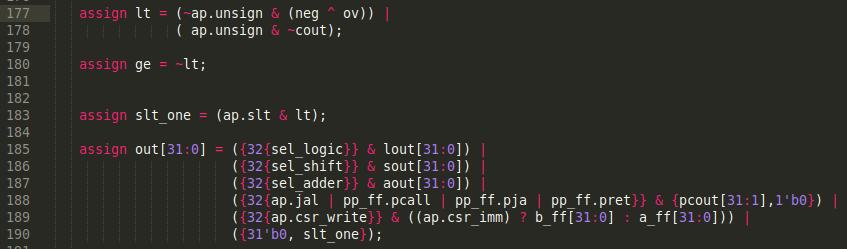
beq zero, zero, REPEAT # Repeat the loop

.end



以下Verilog片段显示了在SweRV EH1中执行这些运算的逻辑。





1. 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:

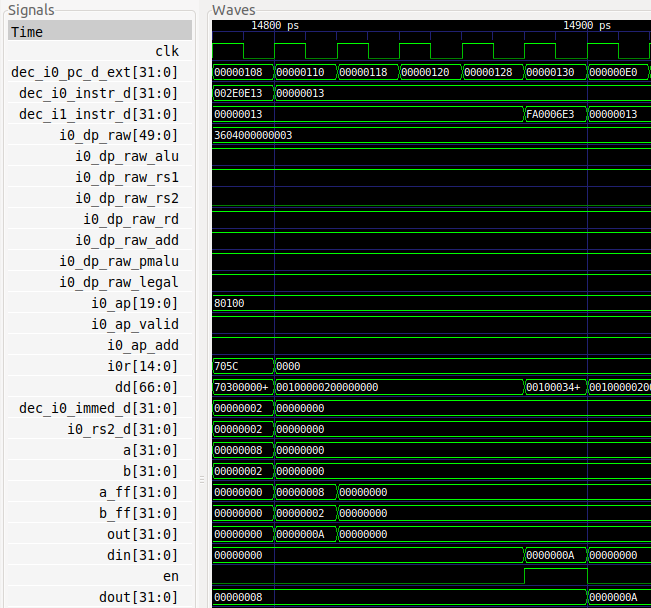
INSERT\_NOPS\_10

addi t3, t3, 2 # t3 = t3 + t4

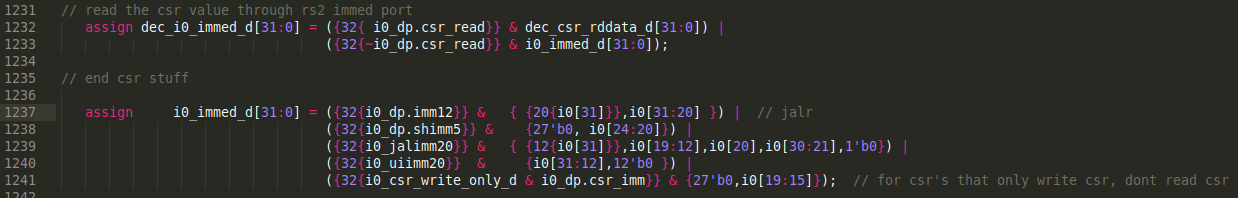
INSERT\_NOPS\_10

beq zero, zero, REPEAT # Repeat the loop

.end



在模块**dec\_decode\_ctl**中，计算32位立即数。



在模块**exu**中，选择正确的*rs2*源。本例中我们使用*dec\_i0\_immed\_d*。



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.



1. （*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;

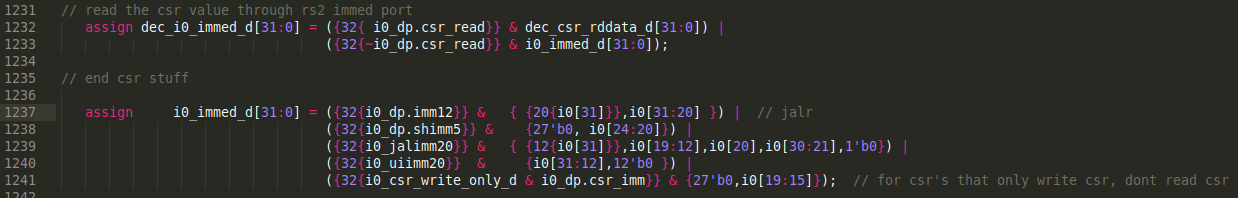
* 信号valid固定为0：无法执行任何A-L指令，因为任何A-L指令都将被视为无效。
* 信号land、lor、lxor、sll、srl、sra、beq、bne、blt、bge、add、sub、slt和jal固定为0：对于上述每一位，都无法执行相应的A-L指令；例如，如果land固定为0，将无法执行and指令。
* 信号unsign固定为0：无法向处理器传达运算必须为无符号运算的信息。
* 信号predict\_t和predict\_nt：无法向处理器传达预测采用或不采用分支的信息。
* 信号csr\_write和csr\_imm：无法在CSR寄存器中写入或使用立即数进行运算。

1. （*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。

* 1. 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。
  2. 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：



addi的控制信号的值位于练习5的仿真中。