

**Imagination大学计划**

**RVfpga实验12**

## **算术/逻辑指令：add指令**

# 简介

在本实验中，我们将分析SweRV EH1流水线各个阶段的算术和逻辑指令流。图1给出了EH1微架构的高级视图，我们在本实验中分析的阶段以红色突出显示：I0管道的译码、EX1、EX2、EX3、提交（有时称为EX4）和回写阶段。（I1管道几乎与I0管道相同，但我们将其深入分析推迟到研究超标量处理的实验17。我们还会在实验11和实验16中分析取指和对齐阶段。）

****

**图1. SweRV EH1管道：突出显示了add指令的第4-9阶段**

在第2部分中，我们将分析add指令的译码到回写阶段（此时它将结果写入寄存器文件）。在说明过程中，我们会交错仿真add指令，您应当在自己的计算机上重复此仿真过程。在第3部分中，我们将提供相关练习以按照与add指令所述步骤类似的过程分析其他算术逻辑指令。

# SweRV EH1内核add指令的分析

在本部分中，我们将使用图2给出的示例，该示例执行无限循环中包含的add指令。文件夹*[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction*提供了PlatformIO项目，这样便可根据需要分析、仿真和更改程序。如SweRVref文档的第2部分所述，为简单起见，本项目中禁止使用压缩指令。此外，为方便起见，我们会在无限循环中插入add指令，这样一来，如果我们避免分析循环的第一次迭代，则可以在没有指令高速缓存（I$）未命中的情况下进行检查。这样做还能轻松地在仿真中找到关注的区域。最后，正如我们在相关实验包含的示例中执行的操作一样，add指令（在图2中以红色突出显示）前后存在几条nop（无操作）指令，以便将其与属于循环的其他迭代的前方/后方add指令隔离。

|  |
| --- |
| .globl main  main:  li t3, 0x4 # t3 = 4  li t4, 0x1 # t4 = 1    REPEAT:  INSERT\_NOPS\_10  **add t3, t3, t4** # t3 = t3 + t4  INSERT\_NOPS\_10  beq zero, zero, REPEAT # Repeat the loop  .end |

图2. add指令示例

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/Lab12/ADD\_Instruction/.pio/build/swervolf\_nexys/firmware.dis*中），则会发现add指令（0x01de0e33）位于该程序中的地址0x00000108处。

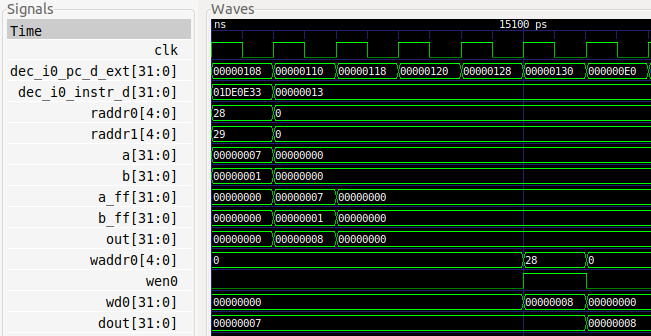
**0x00000108: 01de0e33 add t3,t3,t4**

**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。

1. **add指令的基本分析**

图3给出了图2中程序的Verilator仿真，其中显示了图2中的程序在循环的第四次迭代中执行add指令。这张图包括与译码、EX1和回写（Writeback，WB）阶段相关的一些信号。以红色突出显示的值对应于add指令，因为该指令通过I0管道遍历这三个阶段。请注意，图中所示的信号对应于I0管道。

**add t3,t3,t4**



**译码**

**i+5**

**i+1**

**周期i**

**EX1**

**WB**

图3. 图2中示例程序的Verilator仿真

图4给出了通过I0管道在循环的第四次迭代期间执行add指令的SweRV EH1流水线简化图（参见图2中的程序）。请注意图中合并了处理器在不同时钟周期的状态：

* **周期i： 译码：**对指令进行译码并读取寄存器文件。通过I0管道发送Add指令。
* **周期i+1： EX1：**通过ALU计算加法。
* **周期i+5： 回写：**使用写端口0将加法结果写入寄存器文件。



图4. 执行add指令的SweRV EH1流水线

**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。

我们将同时分析图3中的波形和图4中的图，以此了解流水线中的add指令，具体如下所述。

* **周期i**： **译码：**信号dec\_i0\_instr\_d包含32位机器指令0x01DE0E33。在RISC-V中，add指令的操作码如下（参见[DDCARV]的附录B）：

1. | rs1 | 000 | rd | 0110011

可以轻松验证0x01DE0E33是否对应于：add t3, t3, t4（请记住，t3=x28且t4=x29）。

在这一阶段，将生成**控制信号**并**读取寄存器文件**。在下一阶段（EX1），操作数将通过I0管道发送到ALU。信号raddr0和raddr1（图中以十进制表示）包含add指令的两个源寄存器编号，信号a和b包含将在下一个（EX1）阶段发送到ALU的值。本例中的a和b是从寄存器文件中读取的值。对于其他指令，a和b可能是不同的值；例如，b可能是立即数。我们将在后面的实验中分析其他指令。

* 周期i+1： **EX1：执行**add指令。信号a\_ff和b\_ff包含ALU的输入（本例中分别为7和1），而信号out包含加法的结果（8）。
* 周期i+5： **回写**：最后，在4个周期后，加法的结果通过信号wd0 = 0x8**回写**到寄存器文件中，其中包含要写入的数据。鉴于本周期内wen0 = 1（写使能），加法结果会在周期结束时写入寄存器x28（以十进制表示，waddr0 = 28）。可以发现，在接下来的周期（图中最后一个周期）中，寄存器x28已更新为新值（dout = 8）。

请记住，可通过GTKwave轻松更改信号的数据格式。为此，将光标置于信号上，单击鼠标右键，然后选择所需的“数据格式”。例如，waddr0采用十进制格式（28）而不是十六进制格式（0x1C）将更方便查看，如图5所示。



图5. 以十进制格式显示的信号waddr0

1. **add指令的高级分析**

本部分将比A部分更详细地分析add指令遍历的各个阶段（从译码到回写），逐渐向图3中的仿真添加更多信号。

图6给出了add指令通过I0管道执行期间遍历的主要元素的细节图。实验11的图4已给出相关图示（建议将两个图进行比较），但现在只关注I0管道并提供与add指令相关的详细信息。可能需要将图放大才能看到细节。控制信号的名称显示为红色，而数据信号的名称显示为黑色。这些名称是SweRV EH1 Verilog模块中使用的实际名称。等号（=）表示Verilog代码中的信号分配。

**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。



图6. 经过I0管道的算术逻辑指令使用的主要单元

1. **译码阶段**

如实验11中所述，译码阶段负责两个主要任务：

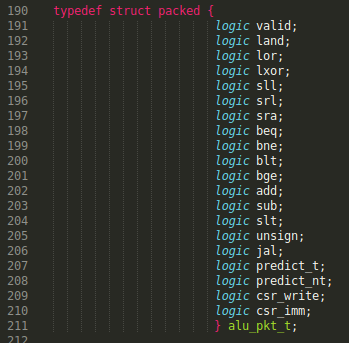
* **对指令进行译码并产生控制信号**。
* **读取或组合源操作数并将指令发送到适当的管道**。

接下来将针对add指令分析上述每项任务并将一些相关信号添加到仿真中。

**对指令进行译码并产生控制信号**：

如实验11的第2.C.i部分所述，*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/include/swerv\_types.sv*中定义了多个结构来对控制位进行分组。其中三个结构与算术逻辑（Arithmetic-Logic，A-L）指令直接相关：

* **alu\_pkt\_t**：这是A-L指令的主要结构：

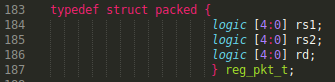


该类型的两个信号i0\_ap（通路0）和i1\_ap（通路1）于译码阶段在模块**dec\_decode\_ctl**内部定义和分配，并在后续执行（EX1-4）阶段传播（通路0为信号i0\_ap\_e1、i0\_ap\_e2、i0\_ap\_e3和i0\_ap\_e4，通路1为信号i1\_ap\_e1、i1\_ap\_e2、i1\_ap\_e3和i1\_ap\_e4）。它们包含用于向ALU告知其必须执行的操作的控制信号。当执行add指令时，i0\_ap/i1\_ap除了下面两位控制信号外的所有位均设置为0：

* + valid： 表示这是一条有效的ALU指令
  + add： 表示这是一条add指令

当通路0/1中的指令不是A-L指令时，信号i0\_ap/i1\_ap的所有位均为0（具体来说，valid = 0），这样一来I0/I1 ALU根本不起作用。

* **reg\_pkt\_t**：该类型的信号i0r（通路0）和i1r（通路1）在模块**dec\_decode\_ctl**内部定义、分配和使用。它们包含两个源寄存器（字段rs1和rs2）的编号和目标寄存器（字段rd）的编号：



* **dest\_pkt\_t**：该类型的信号dd于译码阶段在模块**dec\_decode\_ctl**内部定义和分配，并在所有剩余阶段（信号e1d、e2d、e3d、e4d和wbd）传播。它包含多个字段，例如通路0和通路1中指令的目标寄存器：分别为i0rd[4:0]和i1rd[4:0]。

其中一些信号用于译码阶段，不会通过控制流水线寄存器传播到后面的阶段。i0r.rs1/i1r.rs1和i0r.rs2/i1r.rs2就属于这种情况，它们在译码阶段直接提供给寄存器文件以读取两个输入操作数（信号raddr0、raddr1、raddr2和raddr3）。

**任务：**在Verilog代码（模块**dec\_decode\_ctl**）中查找如何使用i0r控制信号在译码阶段读取寄存器文件。

不过，其他控制信号必须传播到后面的阶段。i0\_ap/i1\_ap就属于这种情况，ALU使用它来了解必须执行的操作（本例中为加法），dd也属于这种情况，寄存器文件使用它来写入两个结果。

**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（EX1）阶段。此外，还需要查找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。

**读取或组合源操作数并将指令发送到适当的管道**：

如实验11中所述，SweRV EH1处理器包含多个用于执行指令的管道。在译码阶段，指令一旦被译码，就必须通过适当的管道进行调度。具体来说，如果A-L指令在通路0上，则必须尽可能发送到I0管道；类似地，如果A-L指令在通路1上，则必须尽可能发送到I1管道。在本实验分析的程序（图2）中，一旦处理器在通路0上对add指令进行译码（即，其“已知”这是一条A-L指令，因此必须将其发送到I0管道），则必须检查是否满足通过I0管道执行的所有条件：有效译码？2个输入操作数可用？流水线未阻塞？...在本例中，此项检查的结果通过两个状态信号发送到I0管道，这两个信号在**dec\_decode\_ctl**模块中计算，供**exu**模块中的ALU使用（在下一小节中，我们将更详细地说明ALU）。这两个状态信号为：

* i0\_e1\_ctl\_en（在ALU内部重命名为enable）：该信号取决于dec\_i0\_ctl\_en[4:1]，它在译码时确定通路0的每个执行阶段（EX1-3）以及提交阶段必须使能（1）还是禁止（0）。请注意，根据不同情况（分支预测错误、除法计算错误等）的影响，该指令可能是非法的，或者流水线可能被阻塞和清除等，这些都会禁用流水线。
* dec\_i0\_alu\_decode\_d（在ALU内部重命名为valid）：如果通路0处的算术/逻辑指令已合法译码并且不使用辅助ALU（我们将在实验15中说明该结构），则该信号为1。

两个信号均必须为1，ALU才能在下一阶段（EX1阶段）执行add操作。

**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。

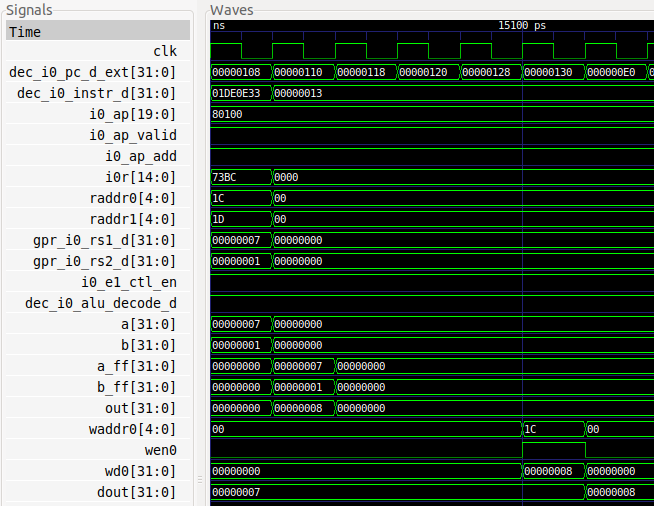
同样如实验11中所述，输入操作数通过译码阶段实现的两个3:1多路开关提供给I0管道（i0\_rs1\_final\_d和i0\_rs2\_d）（参见图6）。在本示例的add指令中，两个输入操作数均从寄存器文件中直接获得：

* 第一个输入操作数： i0\_rs1\_final\_d[31:0] = gpr\_i0\_rs1\_d[31:0]
* 第二个输入操作数： i0\_rs2\_d[31:0] = gpr\_i0\_rs2\_d[31:0]

**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。

图7通过添加上文所述信号扩展图3中的Verilator仿真：

* i0\_ap[19:0]
* i0\_ap.valid（通过下述*.tcl*脚本中的别名在图中命名为i0\_ap\_valid，脚本位于*[RVfpgaPath]/RVfpga/Labs/Lab12/ADD\_Instruction/test\_2.tcl*）
* i0\_ap.add（通过下述*.tcl*脚本中的别名在图中命名为i0\_ap\_add）
* i0r[14:0]
* raddr0
* raddr1
* i0\_e1\_ctl\_en（在ALU内部重命名为enable）
* dec\_i0\_alu\_decode\_d（在ALU内部重命名为valid）
* gpr\_i0\_rs1\_d[31:0]
* gpr\_i0\_rs2\_d[31:0]



**寄存器文件  
读取**

**有效/使能**

**i0\_ap**

图7. 图2示例程序的Verilator仿真，包括控制信号和寄存器文件读端口

**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。

分析图7中的波形。如上文所述，除valid和add位以外，i0\_ap的所有位均为0。此外，信号i0r包含add指令的两个源寄存器和一个目标寄存器的标识符。使用i0r.rs1和i0r.rs2访问寄存器文件（参见信号raddr0和raddr1）并将读取的值提供给I0管道：gpr\_i0\_rs1\_d = a = 0x7且gpr\_i0\_rs2\_d = b = 0x1。最后，可以看到valid和enable信号均为1，因此I0管道ALU将在下一个周期中使用。

**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示例所用文件相同的*test\_2.tcl*文件。

1. **执行阶段**

如实验11中所述，SweRV EH1包含四个执行管道（参见实验11中的图4）：I0/I1、乘法和L/S管道。此外，它还包含一个非流水化除法器。每个管道分为3个阶段：**EX1/EX2/EX3**（I0/I1管道）、**M1/M2/M3**（乘法管道）和**DC1/DC2/DC3**（L/S管道）。在本实验中，我们将重点关注I0管道，其中会执行add指令。Add指令的I0管道的主要任务是在ALU中计算加法并将其传播到提交阶段。

1. **EX1阶段**

该阶段执行ALU操作 – 本例中为加法。SweRV EH1的算术逻辑单元（Arithmetic-Logical Unit，ALU）在模块**exu\_alu\_ctl**（位于*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu/exu\_alu\_ctl.sv*）中实现，并在模块**exu**（位于*[RVfpgaPath]/RVfpga/src/SweRVolfSoC/SweRVEh1CoreComplex/exu.sv*）中实例化。

图8给出了I0管道EX1阶段中包含的ALU的实例化过程，以及ALU及其一些输入/输出端口的简化图。请注意，大多数输入/输出信号在ALU中均已重命名。



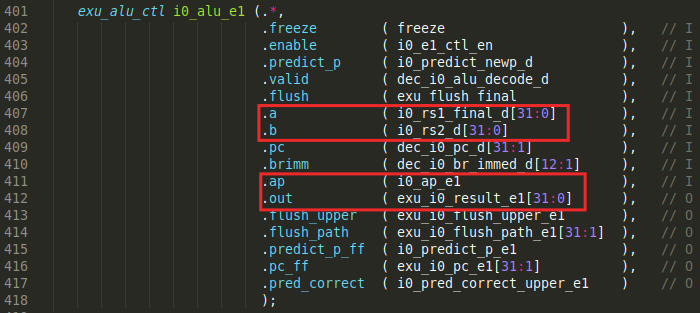


图8. I0的ALU（exu\_alu\_ctl模块）：高级视图和Verilog代码

**ALU输入：**ALU输入（a和b）在译码阶段由图6中所示的两个3:1多路开关选择（如上一部分所述）。在**exu\_alu\_ctl**模块内部，当valid和enable信号均为1时，两个寄存器（aff和bff）将操作数从译码阶段传播到EX1阶段。

**ALU控制信号：**ALU由信号i0\_ap在译码阶段生成的控制位支配（请记住，这是一种alu\_pkt\_t类型结构）。该信号通过流水线寄存器传播（如上一部分所述）。在EX1中，该信号称为i0\_ap\_e1，它在ALU内部重命名为ap（参见图8）。

**ALU输出：**ALU输出在信号exu\_i0\_result\_e1中获得（参见图8）。该信号使用新的流水线寄存器传播到EX2（参见图6），它位于模块**dec\_decode\_ctl**中（信号首先分配给i0\_result\_e1）：



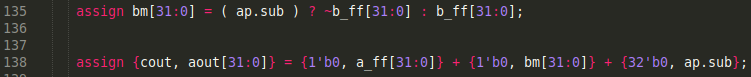


**任务：**将本部分中分析的新信号添加到图7的仿真中。

**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。

**任务：**分析模块**exu\_alu\_ctl**中实现的加法器/减法器的Verilog实现。图9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。可以使用Verilator仿真作为帮助。







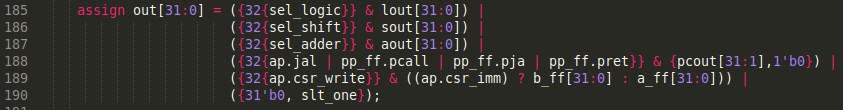


图9. exu\_alu\_ctl内部的加法器

1. **EX2和EX3阶段**

这些阶段在算术-逻辑指令中执行的任务很少；但是，要将这些指令与其他需要三个周期来计算其操作的指令类型（例如装载、存储、乘法等）同步，这些阶段必不可少。请记住，在多周期设计中，每条指令可以有不同数量的阶段，但在SweRV EH1等顺序流水线处理器中，所有指令必须遍历相同数量的阶段。

在本例中，加法的结果通过新的流水线寄存器传播，它位于模块**dec\_decode\_ctl**中。在EX3的3:1多路开关中，将选择i0\_result\_e3（参见图6）。该3:1多路开关也在实验11的图4和图8中显示。正如该实验中所述，它从正确的管道中选择结果，图2示例中的结果由I0管道提供：（i0\_result\_e3\_final = i0\_result\_e3）。





**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。

1. **提交阶段**

与EX2和EX3类似，该阶段对独立add指令执行少量操作（在实验15中，我们将分析依赖于前一条指令的add指令必须在辅助ALU中重新计算加法的情况，图6中未显示）。在本例中，此阶段可用的3:1多路开关选择输入i0\_result\_e4。该3:1多路开关也在实验11的图4和图9中显示。在图2的示例中，选择的值仍是I0管道提供的结果（i0\_result\_e4\_final = i0\_result\_e4）。

**任务：**对于图2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（i0\_result\_e4）选择结果。

1. **回写阶段**

在最后一个阶段，add指令的结果写入寄存器文件，如图6所示。32位结果（i0\_result\_wb\_raw[31:0]）在EX1阶段计算并传播到此阶段。它在传递到寄存器文件之前遍历2:1多路开关（此多路开关的另一个输入来自除法器，我们将在实验14中进行分析）。寄存器地址（在图7中，信号waddr0以十六进制显示，但也可以如前文所述以十进制显示）和写使能信号通过控制流水线寄存器提供。

**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. 对以下逻辑指令执行与本实验中提供的分析类似的分析：and、or和xor。
2. （*以下练习基于《计算机组织结构和设计》（RISC-V版本，Patterson & Hennessy（[PaHe]））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？

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*左移/右移*指令：srl、sra和sll。
2. 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。
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*立即数*指令：addi、andi、ori、xori、srli、srai、slli、slti和sltui。
4. （*以下练习基于[PaHe]的练习4.6。*）

图6不讨论I型指令，例如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6并说明其用途。
  2. 列出addi的控制单元产生的信号的值。

1. （*以下练习基于[PaHe]的练习4.4以及S. Harris和D. Harris所著教科书《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。