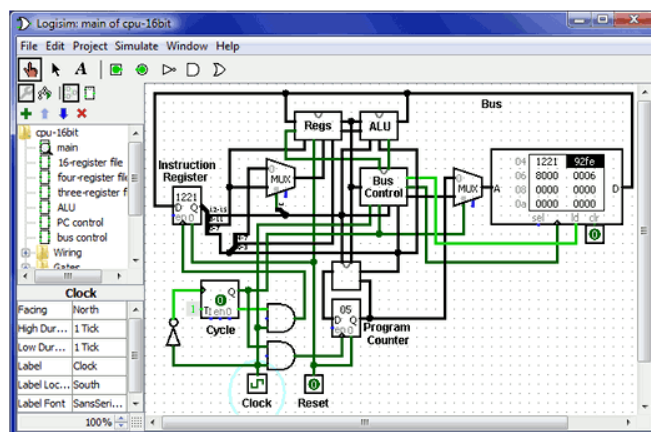


Logisim 用户指南



Logisim 是一种用于设计和模拟数字逻辑电路的教育工具。具有简单的工具栏界面和电路模拟功能。它简单易用，有助于初学者学习数字逻辑电路的基本概念。由于 Logisim 能够从较小的子电路构建较大的电路，并通过鼠标拖动绘制电路之间的连线，因此可以用于设计和模拟整个 CPU，在全球众多高校中有广泛的应用。

可以使用 Logisim 的课程，包括但不限于：

- 数字逻辑电路课程
- 计算机系统概论课程
- 计算机组成原理课程
- 计算机体系结构课程

您现在正在阅读的《Logisim 用户指南》是 Logisim 的官方参考。第一部分是介绍 Logisim 主要部分的一系列章节。编写这些部分的目的是为了“从头到尾”阅读，了解 Logisim 的所有重要功能，主要包括：

- 初级教程
- 库和属性
- 子电路
- 线束
- 组合分析

此外还包含了 Logisim 相关的各类参考资料和扩展功能介绍

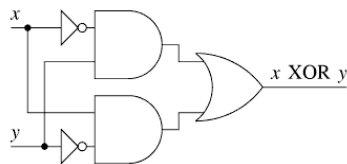
- 菜单参考
- 内存组件
- 日志记录
- 命令行验证
- 应用程序首选项
- 项目选项
- 值传播
- JAR 库
- 关于程序

第一节：初级教程

Logisim 允许用户设计和模拟数字电路。作为一款教育工具软件，它能够帮助您了解电路的结构和工作原理。为了练习使用 Logisim，让我们构建一个异或门电路（XOR）——也就是说，一个接受两个输入（我们称之为 x 和 y ）的电路，如果输入相同，则输出 0；如果输入不同，则输出 1。下面的真值表说明了这一点。

x	y	$x \text{ XOR } y$
0	0	0
0	1	1
1	0	1
1	1	0

我们可以在纸上设计出这样的电路。



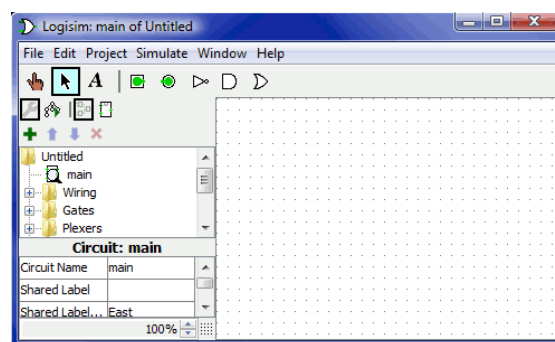
在纸上绘制电路很容易，但无法确保它是正确的。为了验证该电路，我们将通过如下步骤在 Logisim 中绘制并测试它。我们将得到一个看起来比在纸上画的更好的电路图。

- 步骤 0：打开 Logisim
- 步骤 1：添加门
- 步骤 2：添加导线
- 步骤 3：添加文本
- 步骤 4：测试电路

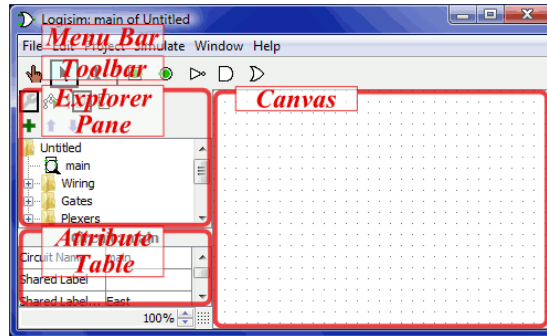
下面开始电路的搭建过程，希望你能够乐在其中。

步骤 0：打开 Logisim

启动 Logisim 时，您将看到一个类似以下的窗口。根据所使用的操作系统，细节可能略有不同。



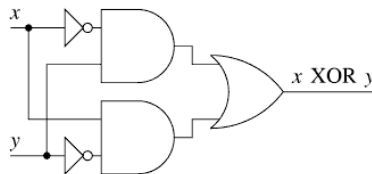
Logisim 程序界面主要分为三个部分，称为资源管理器窗格、属性表和画布。在这些部分的上方是菜单栏和工具栏。



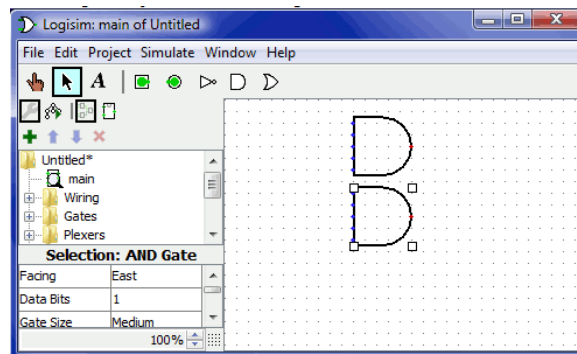
资源管理器窗格和属性表将在后面专门介绍，这里可以先暂时忽略；画布是绘制电路的地方；工具栏中包含了用于完成此操作的工具。接大多数软件都包含了菜单栏，这里不再介绍。

步骤 1：添加门

我们正试图在 Logisim 中构建以下电路。

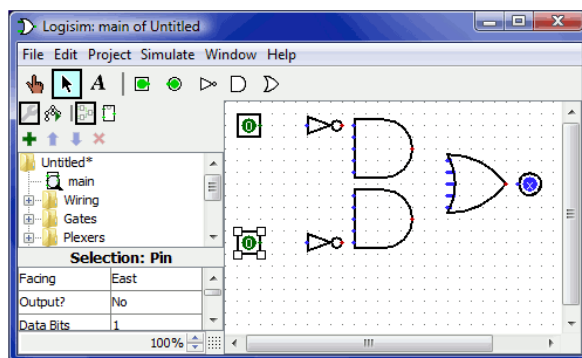


建议先放置门电路并调整其位置和布局，然后再使用导线连接，从而构建电路。我们首先要做的是添加两个与门（AND）。单击工具栏中的 AND 工具（工具栏右起第二个图标）。然后在编辑区域中单击要放置第一个与门的位置。确保在左边为电路其余部分留有足够的空间。然后再次单击 AND 工具，并在其下方放置第二个与门。



请注意 AND 门左侧的五个点。这些是可以连接导线的点。这里，我们将只使用其中两个用于异或电路；但对于其他电路，您会发现两条以上的导线连接到 AND 门是有用的。

现在添加其他门。首先单击 OR 工具（工具栏右起第一个图标）；然后单击所需位置。并使用 NOT 工具（工具栏右起第三个图标）将两个非门放入画布中。



如果需要对已经放置好的元件位置进行调整，可以使用编辑工具（工具栏左起第二个图标）选择它，然后将其拖动到所需的位置。

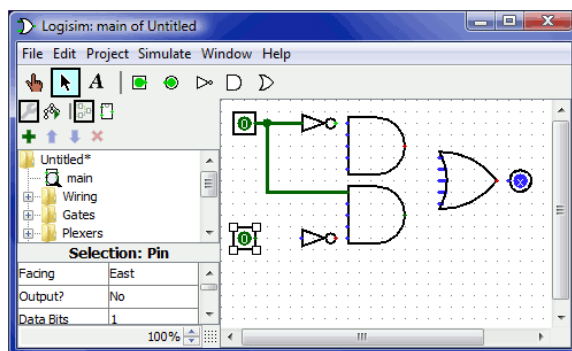
如果需要删除元件，可以先使用鼠标将其框选，然后从“编辑”菜单中选择“删除”或按 delete 键进行删除。

在放置电路的每个元件时，您会注意到，只要放置元件，Logisim 就会返回到“编辑”工具，以便您可以移动最近放置的元件，或者通过创建导线将元件连接到其他元件（详见步骤 2）。

如果要重复添加最近放置的元件，可以使用快捷键 Control-D 进行操作。

步骤 2：添加导线

在画布上放置好所有组件后，就可以开始添加连线了。选择编辑工具（工具栏左起第二个图标）。当光标位于接收导线的点上时，将围绕该点绘制一个小的绿色圆。在那里按下鼠标按钮，并将其拖动到所需的位置。添加导线时，Logisim 相当智能：只要导线在另一条导线上结束，Logisim 就会自动连接它们。也可以通过使用编辑工具拖动导线的一个端点来“延伸”或“缩短”导线。Logisim 中的导线必须水平或垂直。为了将上面的输入连接到 NOT 门和 AND 门，我们添加了三条不同的线路。

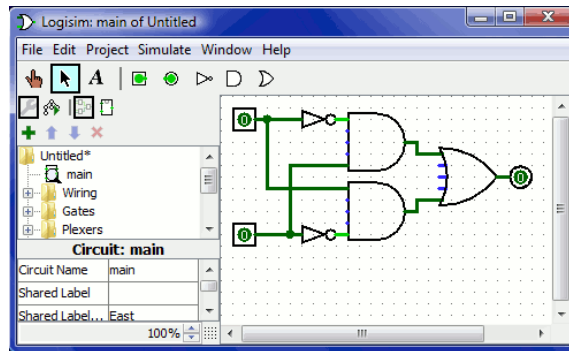


Logisim 自动将电线连接到门和彼此。这包括如上所示在 T 交点处自动绘制圆，表示导线已连接。

绘制导线时，可能会看到一些蓝色或灰色导线。Logisim 中的蓝色表示该点的值为“未知”，灰色表示导线未连接到任何东西。这没什么大不了的，因为你正在建造一条赛道。但当完成之后，电路的电线都不应该是蓝色或灰色的。（手术室门的未连接支腿仍为蓝色：很好。）

如果在你认为所有东西都应该连接起来之后，你确实有一根蓝色或灰色的电线，那么就是出了问题。将电线连接到正确的位置很重要。Logisim 在组件上绘制小点，以指示电线应连接的位置。当你继续，你会看到这些点从蓝色变成浅绿色或深绿色。

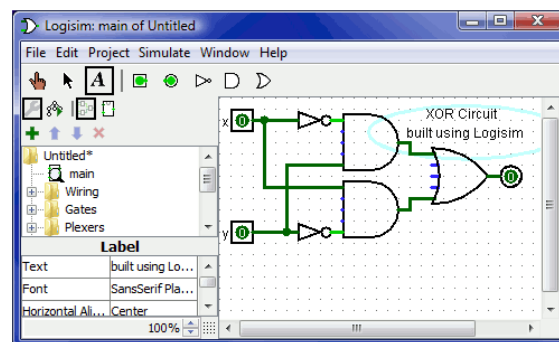
连接完所有导线后，插入的所有导线本身都将为浅绿色或深绿色。



步骤 3：添加文本

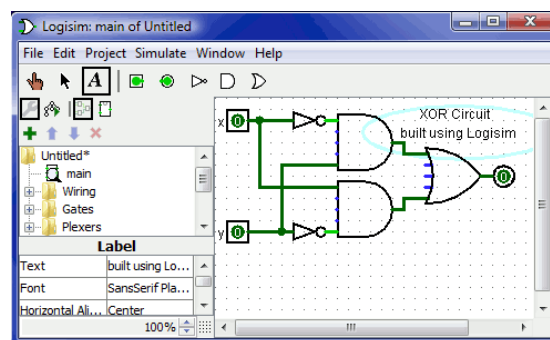
在电路中添加文本是不必要的；但如果你想向某人（如老师）展示你的电路，那么一些标签有助于传达电路不同部分的用途。

选择文本工具 (A)。您可以单击输入管脚并开始键入以给它添加标签。（最好直接单击输入端号，而不是单击文本所在的位置，因为这样标签就会随着端号移动。）您可以对输出端号执行相同的操作。或者，您可以单击任何旧位置，然后开始键入，将标签放在其他位置。



步骤 4：测试电路

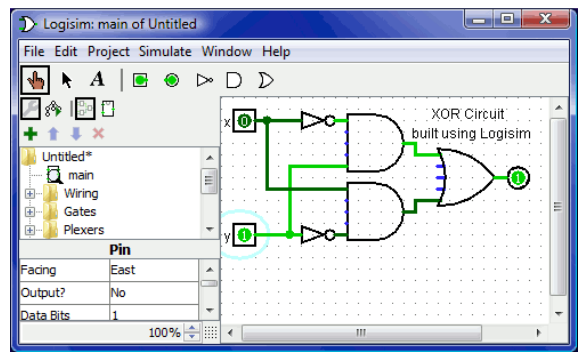
我们的最后一步是测试我们的电路，以确保它真正达到我们的预期。Logisim 已经在模拟电路。让我们再看我们在哪里。



注意，输入引脚都包含 0s；输出引脚也是如此。这已经告诉我们，当两个输入都为 0 时，电路已经计算了 0。

现在尝试另一种输入组合。选择 poke 工具 (P)，然后单击输入开始 poke。每次插入输入时，其值都会切换。

例如，我们可能首先插入底部输入。



更改输入值时，Logisim 将显示哪些值沿导线向下移动，方法是将它们绘制为浅绿色以指示 1 值，或绘制为深绿色（几乎为黑色）以指示 0 值。您还可以看到输出值已更改为 1。

到目前为止，我们已经测试了真值表的前两行，并且输出（0 和 1）与所需的输出相匹配。

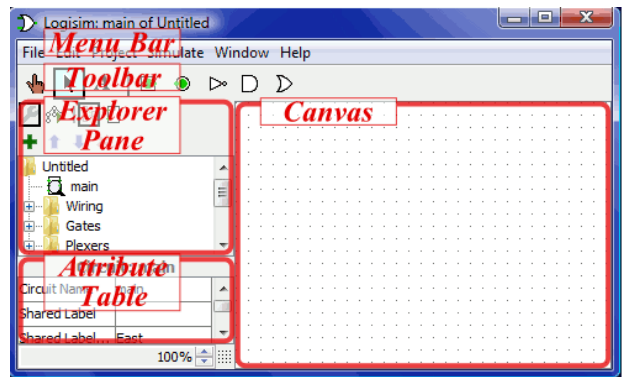
x	y	x XOR y
0	0	0
0	1	1
1	0	1
1	1	0

通过将开关插入不同的组合，我们可以验证其他两行。如果它们都匹配，那么我们就完了：电路正常！要存档完成的工作，您可能需要保存或打印电路。File（文件）菜单允许这样做，当然也允许您退出 Logisim。但为什么现在就放弃呢？

现在您已经完成了教程，您可以通过构建自己的电路来实验 Logisim。如果要构建具有更复杂功能的电路，则应浏览帮助系统的其余部分，看看还能做什么。Logisim 是一个功能强大的程序，允许您构建和测试大型电路；这个循序渐进的过程只是表面现象。

第二节：库和属性表

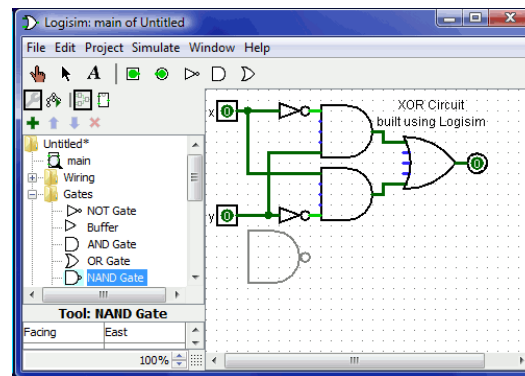
在本节中，我们将研究如何使用 Logisim 中的其他个主要区域，即资源管理器窗口和属性表。最后，对 Logisim 的工具和组件属性进行介绍。



资源管理器窗格

Logisim 将工具组织到库中。它们在资源管理器窗格中显示为文件夹；要访问库的组件，只需双击相应的文件夹。下面，我打开了盖茨库并从中选择了 NAND 工具。您可以看到，Logisim 现在已准备好将 NAND 门

添加到电路中。

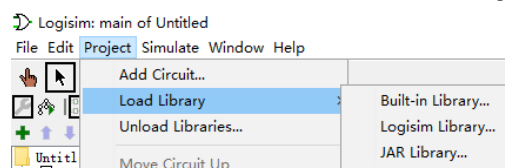


如果你仔细查看逻辑门库中的选项,你会发现我们没有必要自己搭建 XOR 电路,它已经内置于 Logisim 中。

创建项目时, Logisim 会自动包含多个库, 统称为内置库:

- 导线库: 与导线直接交互的元件。
- 逻辑门库: 执行简单逻辑功能的部件。
- 复用器库: 更复杂的组合组件, 如多路选择和解码器。
- 运算器库: 执行算术的组件。
- 存储器库: 记忆数据的组件, 如触发器、寄存器和 RAM。
- 输入输出库: 为与用户交互而存在的组件。
- 基础库: 使用 Logisim 不可或缺的工具, 但您可能不需要经常深入研究这个库。

Logisim 允许用户使用项目菜单的“加载库”子菜单添加更多库。可以看到 Logisim 有三类库。



- 内置库: 内置库是与 Logisim 一起分发的库。这些都记录在 Library Reference 中。
- Logisim 库: Logisim 库是在 Logisim 中构建并作为项目文件保存到磁盘的项目。可以在单个项目中开发一组电路 (如本指南的“子电路”部分所述), 然后将该组电路作为其他项目的库。
- JAR 库: JAR 库是用 Java 开发的库, 但不是用 Logisim 分发的。您可以下载其他人编写的 JAR 库, 也可以按照本指南的 JAR libraries 部分所述编写自己的库。开发 JAR 库要比开发 Logisim 库困难得多, 但组件可能会更复杂, 包括属性和与用户的交互。内置库 (Base 除外) 是使用与 JAR 库相同 API 编写的, 因此它们恰当地展示了 JAR 库能够支持的功能范围。

一些 JAR 库没有提供需要导入的 Java 类的信息。加载这样的 JAR 时, Logisim 将提示您键入类名。这个类名应该由分发 JAR 文件的人提供。

删除库时, 可以从“项目”菜单中选择“卸载库...”。Logisim 会阻止用户卸载包含电路中使用的、出现在工具栏中的或映射到鼠标按钮的元件的库。

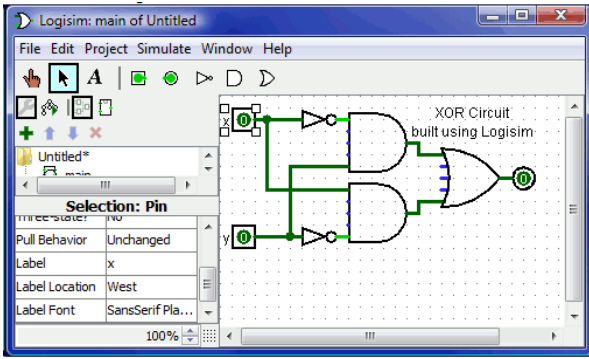
顺便提一下, 从技术上讲, 库包含的是工具, 而不是组件。因此, 在基础库中, 您可以找到 Poke Tool (戳工具)、Edit Tool (编辑工具) 以及其他与各个组件不直接对应的工具。然而, 大多数库只包含添加单个组件的工具; 除基础库之外的所有内置库都是这样的。

属性表

许多组件都有属性, 这些属性用于配置组件的行为或显示方式。属性表用于查看和显示组件的属性值。

要选择要查看的组件属性, 请使用编辑工具 () 单击该组件。(您也可以右键单击 (或控制单击) 组件, 然后从弹出菜单中选择显示属性。此外, 通过 Poke 工具 () 或 Text 工具 () 操作组件将显示该组件的属性。)

下面的屏幕截图演示了选择 XOR 电路的上部输入并向下滑动以查看 Label Font 属性后的情况。



要修改属性值，请单击该值。修改属性的界面取决于要更改的属性；对于“标签字体”属性，将出现一个对话框，用于选择新字体；但有些属性（如标签）允许您将值编辑为文本字段，而其他属性（如标记位置）将显示下拉菜单，从中选择值。

每个组件类型都有一组不同的属性；要了解其含义，请参阅 Library Reference 中的相关文档。

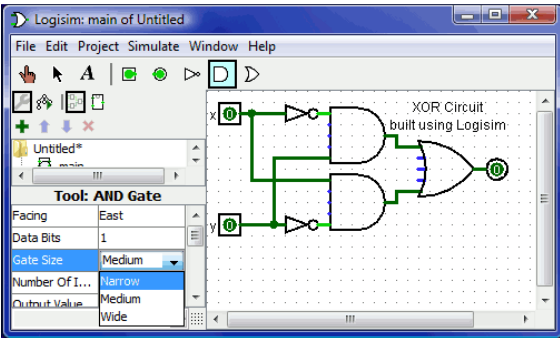
如果使用“编辑”工具选择了多个组件，则属性表将显示所有选定组件（不包括任何导线）之间共享的属性。

如果选定组件的属性值不同，则显示的值将为空。可以使用属性表一次更改所有选定组件属性的值。

工具和组件属性

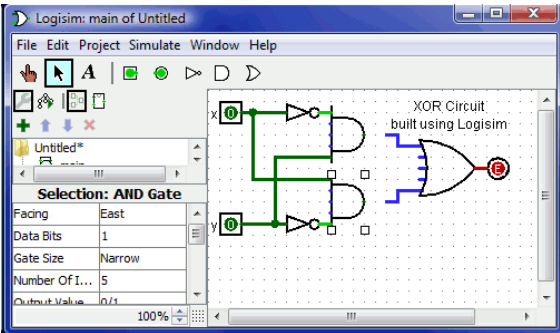
每个用于将元件添加到电路的工具都有一组属性，这些属性将被赋予该工具创建的元件，尽管元件的属性稍后可能会更改，而不会影响工具的属性。选择工具时，Logisim 将更改属性表以显示该工具的属性。

例如，假设我们要创建更小的 AND 门。现在，每次我们选择 AND 工具时，它都会创建一个大型 AND 门。但是，如果我们在选择工具后编辑“门大小”属性（在将其 AND 门放置到电路中之前），我们将更改该工具的属性，以便将来使用该工具添加的 AND 门会变小。



现在，我们可以删除现有的两个 AND 门，并在其位置添加两个新的 AND 门。这一次，它们将变得狭窄。

（如果您选择将输入数量减少到 3，那么 AND 门在左侧将不会有垂直扩展。但您还必须重新布线，以便电线碰到 AND 门的左侧。）



对于某些工具，该工具的图标反映了某些属性的值。其中一个示例是“Pin”工具，其图标的朝向与其“Facing”属性所示的方式相同。

工具栏中的每个工具都有一个独立于资源管理器窗格中相应工具的属性集。因此，即使我们更改了工具栏的 AND 工具以创建窄 AND 门，但除非您也更改其属性，否则 gates 库中的 AND 工具仍将创建宽 AND 门。事实上，默认工具栏中的输入端号和输出端号工具都是 Wiring 库的端号工具的实例，但属性集不同。根据“输出”的值，“锁定”工具的图标绘制为圆形或方形属性

Logisim 提供了一个方便的快捷方式来更改控制许多零部件朝向的方向的“朝向”属性：在选择该工具时键入箭头键会自动更改零部件的方向。

第三节：子电路

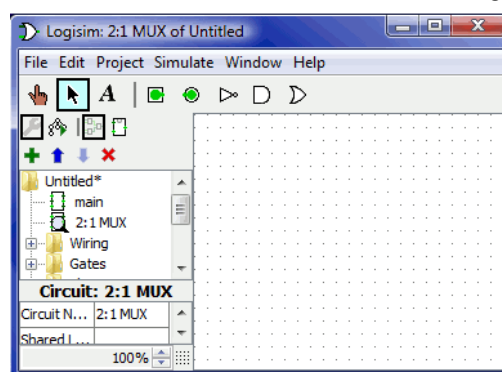
当您构建越来越复杂的电路时，您会希望构建更小的电路，可以将其作为嵌套在较大电路中的模块多次使用。在 Logisim 中，用于较大电路的较小电路称为子电路。

如果你熟悉计算机编程，你就会熟悉子程序的概念，无论它是用你喜欢的语言称为子程序、函数、方法还是过程。子电路的概念与此类似，它的作用是相同的：将一个大的工作分解为小的部分，以节省多次定义同一概念的工作量，并方便调试。

创建电路

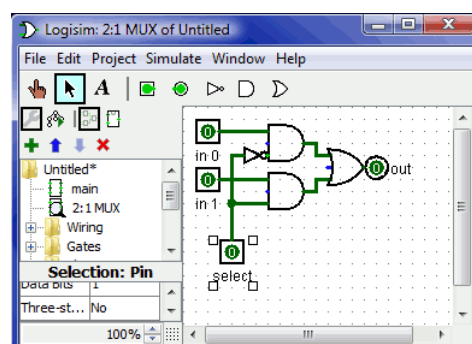
每个 Logisim 项目实际上都是一个电路库。在最简单的形式中，每个项目只有一个电路（默认情况下称为“主电路”），但很容易添加更多电路：从“项目”菜单中选择“添加电路...”，然后为要创建的新电路键入任意名称。

假设我们想构建一个名为“2:1 MUX”的 2 对 1 多路复用器。添加电路后，Logisim 将如下所示。



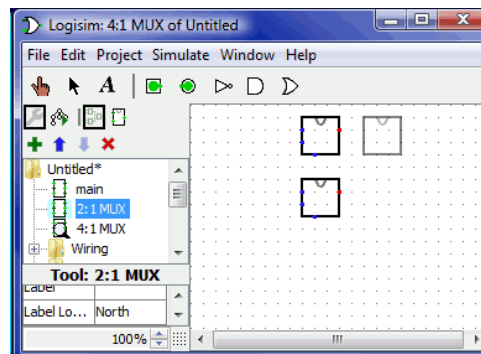
在资源管理器窗格中，您现在可以看到项目现在包含两个电路，“main”和“2:1 MUX”。Logisim 在当前查看的电路图标上绘制放大镜；当前电路名称也显示在窗口的标题栏中。

在编辑电路使其看起来像 2:1 多路复用器之后，我们可能会得到以下电路。

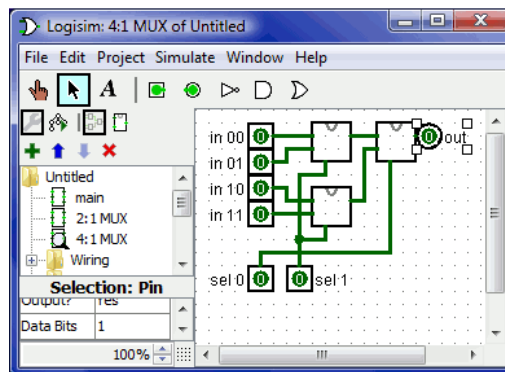


使用子电路

现在假设我们想使用我们的 2 对 1 多路复用器的实例来构建一个 4 对 1 的多路复用器。当然，我们首先会创建一个新的电路，我们将其称为“4:1 MUX”。要向电路中添加 2 对 1 多路复用器，我们在资源管理器窗格中单击一次 2:1 MUX 电路，将其选为工具，然后我们可以通过在画布中单击来添加其副本，以方框表示。

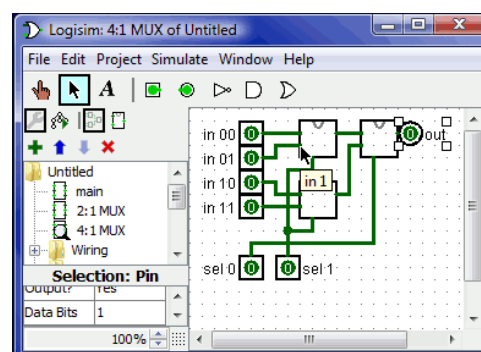


如果要双击资源管理器窗格中的 2:1 MUX 电路，则窗口将切换到编辑 2:1 MUX 电路。在建立电路之后，我们得出以下结论。



我们的四对一多路复用器电路使用了三个二对一多路复用器的副本，每个副本都画成一个盒子，旁边有引脚。此框上的引脚对应于 2:1 MUX 电路中的输入和输出引脚。盒子西侧的两个引脚对应于 2:1 MUX 电路中朝东的两个引脚；盒子东侧的引脚对应于 2:1 MUX 的西向引脚（碰巧是一个输出引脚）；盒子南侧的引脚对应于 2:1 MUX 的朝北引脚。盒子西侧的两个引脚的顺序对应于子电路设计中相同的自上而下的顺序。（如果盒子的北侧或南侧有多个引脚，它们将对应于子电路中的相同左右顺序。）

如果子电路布局中的引脚具有与其关联的标签，则当用户将鼠标悬停在子电路组件的相应位置上时，Logisim 将在提示（即临时文本框）中显示该标签。（如果您觉得这些提示很烦人，可以通过“首选项”窗口的“布局”选项卡禁用它们。）



其他几个组件也会显示这些提示：例如，对于内置触发器的一些引脚，将鼠标悬停在上面可以解释该引脚的功能。

顺便提一下，电路的每个引脚都必须是输入或输出。许多制造的芯片具有引脚，在某些情况下用作输入，在其他情况下用作输出；您不能在 Logisim 中构造这样的芯片（至少在当前版本中是这样的）。Logisim 将为电路中出现的所有子电路维护不同的状态信息。例如，如果一个电路包含一个触发器，并且该电路多次用作子电路，那么在模拟较大的电路时，每个子电路的触发器都有自己的值。既然我们已经定义了 4 对 1 多路复用器，我们现在可以在其他电路中使用它。Logisim 对电路的嵌套深度没有限制——尽管它会反对在电路内部嵌套电路！

注意：编辑用作子电路的电路没有什么错；事实上，这很常见。但请注意，对电路管脚的任何更改（添加、删除或移动管脚）也会在包含的电路中重新排列管脚。因此，如果更改电路中的任何接点，还需要编辑其用作子电路的任何电路。

编辑子电路外观

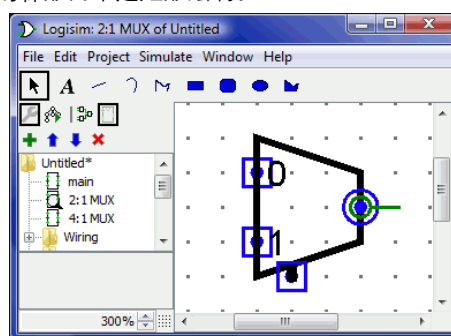
默认外观

默认情况下，当子电路放置在较大的电路中时，它被绘制为一个矩形，其中有一个凹口指示子电路布局的北端。图钉将基于其朝向放置在矩形的边界上：布局中朝东的图钉（通常出现在布局的西侧）将根据其在布局中的自上而下顺序放置在矩形西侧。布局中朝南的接点（通常朝向布局的北侧）将根据布局中从左到右的顺序放置在矩形的北侧。

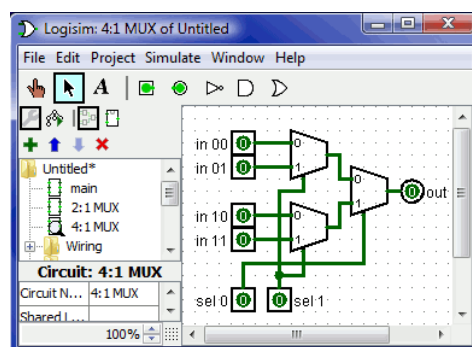
默认矩形可以选择包含一些字母，这些字母将出现在矩形的中间。要指定此选项，请选择选择工具 (A)，然后单击电路布局的背景。这将显示属性表中的电路属性，包括“共享标签”、“共享标签朝向”和“共享标签字体”属性。共享标签属性的值将绘制在矩形的中心；“共享标签朝向”属性自定义文本的绘制方向，当然，“共享标签字体”属性自定义使用的字体。

定制外观

默认外观非常有用，实际上 Logisim 已经存在很多年了，没有其他选择。但是，如果希望以不同的方式绘制子电路，可以从“项目”菜单中选择“编辑电路外观”，Logisim 的界面将从常规布局编辑界面切换到绘制电路外观的界面。（您也可以单击资源管理器窗格上部工具栏中最右侧的图标 (A)。）下面，我们正在编辑 2:1 多路复用器的外观，以使用通常的梯形而不是矩形绘制。












如果 2:1 多路复用器的外观如上所示，那么 4:1 多路复用的布局将如下所示。




外观编辑器类似于传统的绘图程序，但有一些特殊符号用于指示图形在放置到电路布局中时的工作方式。这些特殊符号无法删除。

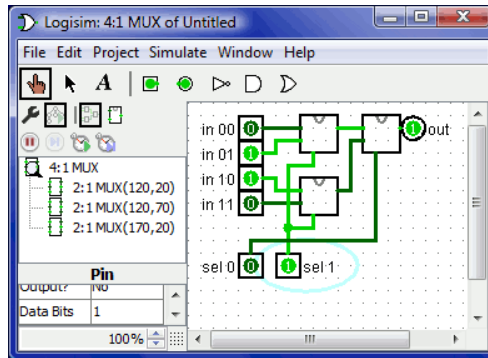
- 带一条线的绿色圆圈，我们称之为锚。每个子电路外观中只有一个锚。电路中的每个元件都有一个点来标识其位置；用户在创建新组件时会看到这一点：鼠标单击仅标识一个位置，组件相对于该位置放置（通常主输出位于鼠标位置）。创建子电路时，锚标识鼠标相对于整个图形的位置。锚点还标识外观的朝向，如锚点线从其圆指向的方向所示。当将子电路放置到布局中时，用户可以更改子电路的朝向；锚的面向指示外观的方向。在我们的示例中，锚点朝东，4:1 多路复用器中的每个子电路实例也朝东，因此它们都以与 2:1 多路复用器外观相同的方向绘制。
- 蓝色圆圈和带有圆点的正方形是子电路的端口。电路中的端口数量与输入和输出引脚数量完全相同。与输入对应的端口绘制为正方形，而与输出对应的端口则绘制为圆形。每个端口指示连接到电路中的导线如何与布局中的输入或输出引脚相对应。当您选择一个端口时，Logisim 将通过在窗口右下角弹出布局的微型图来指示相应的管脚，对应的管脚用蓝色绘制。选择所有端口时不会发生这种情况。

工具栏包含用于添加其他形状的工具，如下所示，并描述了 shift 和 alt 键如何修改工具行为。此外，在按住控制键的情况下单击或拖动鼠标会定期将鼠标位置捕捉到最近的栅格点。

	选择、移动、复制和粘贴形状
	添加或编辑文本
	创建线段。按住 Shift 键并拖动将使线的角度保持为 45° 的倍数
	创建二次贝塞尔曲线。对于指定曲线端点的第一次拖动，按住 shift 键并拖动会使端点保持 45° 的倍数。然后单击以指示控制点的位置；按住 shift 键并单击可确保曲线对称，而按住 alt 键并单击则通过控制点绘制曲线
	创建一系列连接的线，其顶点由连续的单击指示。按住 Shift 键单击可确保上一个顶点与当前顶点之间的角度是 45° 的倍数。双击或按 Enter 键完成形状
	通过从一个角拖动到另一个角来创建矩形。按住 Shift 键并拖动以创建正方形，按住 alt 键并拖动可从中心开始创建矩形
	通过从一个角拖动到另一个角，创建圆角矩形。按住 Shift 键并拖动以创建正方形，按住 alt 键并拖动可从中心开始创建矩形
	通过从边界框的一个角拖动到另一个角来创建椭圆形。按住 Shift 键并拖动以创建圆，按住 alt 键并拖动可从中心开始创建椭圆
	创建任意多边形，其顶点由连续的单击指示。按住 Shift 键并单击可确保顶点与上一个顶点成 45° 角。双击，按 Enter 键，或单击起始顶点以完成形状

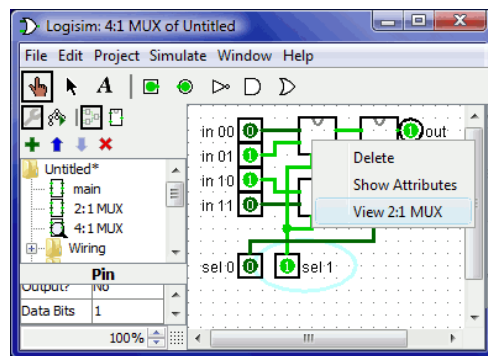
调试子电路

当您测试较大的电路时，您可能会发现错误。为了弄清哪里出了问题，在运行整个电路的同时，探索子电路中的情况会有所帮助。要进入子电路的状态，可以使用三种不同的技术中的任何一种。最简单的方法可能是通过单击资源管理器窗格上部工具栏中的第二个图标 ，或从“项目”菜单中选择“查看仿真树”来查看仿真层次结构。这会切换资源管理器窗格，以便显示正在模拟的子电路的层次结构。



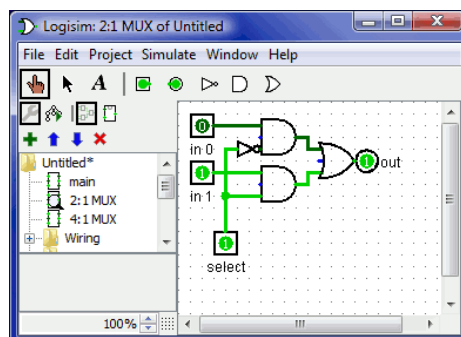
双击此层次结构中的元素将显示该子电路中发生的情况。

进入子电路的第二种方法是，右键单击或控制单击子电路，然后选择 View（视图）选项，弹出弹出菜单。



第三种方法是首先确保选择了 Poke Tool，然后单击要进入的子电路；一个放大镜将出现在子电路的中心，双击放大镜将进入子电路的状态。

在任何情况下，一旦您进入子电路，您将看到子电路中的管脚值与通过它们从包含电路发送的值相匹配。



在子电路中，您可以更改电路。如果更改影响任何子电路的输出，则它们将传播到包含的电路中。一个例外：子电路输入是根据从超级电路进入电路的值来确定的，因此切换这些值没有意义。如果你试图插入一个子电路的输入，会弹出一个对话框，询问：引脚与超级电路状态相关联。创建新电路状态？单击“否”将取消切换请求，而单击“是”将创建已查看状态的副本，该副本与外部电路分离，输入引脚已切换。

完成查看和/或编辑后，可以通过双击资源管理器窗格中的父电路或通过“模拟”菜单的“输出到状态”子菜单返回到父电路。

Logisim 库

每个 Logisim 项目都自动成为一个库，可以加载到其他 Logisim 工程中：只需将其保存到一个文件中，然后在另一个项目中加载该库即可。然后，第一个项目中定义的所有电路将作为第二个项目的子电路。此功能允许您跨项目重用通用组件，并与您的朋友（或学生）共享喜爱的组件。

每个项目都有一个指定的“主电路”，可以通过“项目”菜单中的“设置为主电路”选项将其更改为引用当前电路。这样做的唯一意义是，主电路是首次打开项目时显示的电路。新创建的文件（“main”）中电路的默认名称根本没有意义，您可以随意删除或重命名该电路。

使用加载的 Logisim 库，您可以查看电路并操作其状态，但 Logisim 会阻止您更改电路的设计和文件中存储的其他数据。

如果要更改加载的 Logisim 库中的电路，则需要先在 Logisim 中单独打开它。保存后，其他项目应立即自动加载修改后的版本；但如果没有，您可以右键单击资源管理器窗格中的库文件夹，然后选择重新加载库。

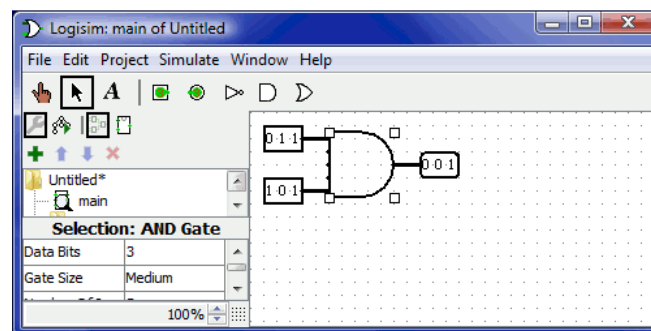
第四节：线束

在简单的 Logisim 电路中，大多数导线只携带一个位；但 Logisim 还允许您创建将多个位捆绑在一起的连线。沿导线移动的比特数就是导线的比特宽度。

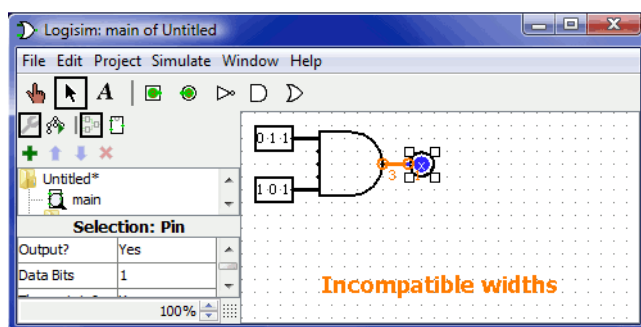
创建线束

电路中每个元件上的每个输入和输出都有一个与其相关联的位宽度。通常位宽度为 1，无法更改，但 Logisim 的许多内置组件都包含属性，允许您自定义其输入和输出的位宽度。

下面的屏幕截图说明了一个简单的电路，用于查找两个三位输入的位 AND。请注意，三位输出如何是两个输入的逐位 AND。所有组件都已通过其“数据位”属性进行定制，以处理三位数据；屏幕截图显示 AND 门属性，包括数据位属性 3。

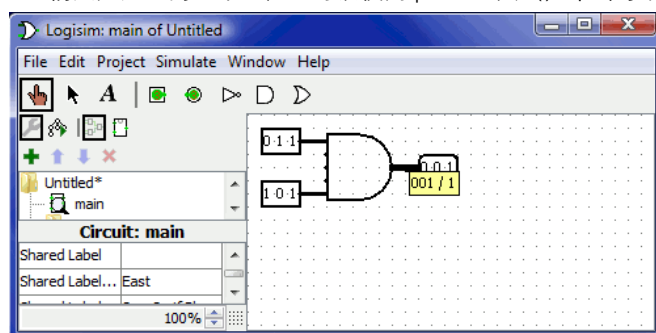


Logisim 中的所有组件都为每个输入和输出定义一个位宽度。相比之下，导线的位宽度是未定义的：相反，导线的宽度会适应它所附着的组件。如果一条线连接两个要求不同位宽度的组件，Logisim 将投诉“不兼容宽度”，并用橙色指示违规位置。在下文中，输出引脚的“数据位”属性已更改为 1，因此 Logisim 抱怨导线无法将三位值连接到一个位值。



连接不兼容位置（以橙色绘制）的导线不包含值。

对于单位导线，您可以一眼看到导线所承载的值，因为 Logisim 会根据值将导线涂成浅绿色或深绿色。它不显示多位导线的值：它们只是黑色的。不过，您可以使用 poke 工具 () 单击导线来探测它。

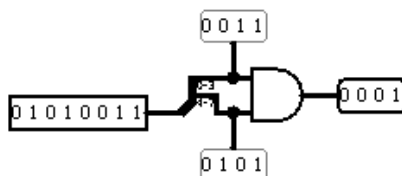


此探测功能有助于使用线束调试电路。

分线器

当您使用多位值时，您通常希望在不同的方向路由不同的位。Wiring 库的拆分器工具 () 允许您完成此操作。

例如，假设我们需要一个电路来计算其八位输入的两个半字节（上四位和下四位）的位 AND。我们将有一个来自输入引脚的 8 位值，我们想把它分成两个 4 位值。在下面的电路中，我们使用了一个分路器来实现这一点：8 位输入端进入分路器，分路器将 8 位分为两个 4 位值，然后将这两个值送入 AND 门并从那里输入到输出端。



在此示例中，拆分器将传入值拆分为多个传出处。但是拆分器也可以以另一种方式工作：它可以将多个值合并为单个值。事实上，它们是非定向的：它们可以一次以一种方式发送值，稍后以另一种方式进行发送，甚至可以同时发送这两种方式，如下面的示例所示，一个值通过两个拆分器向东移动，然后再通过它们向西返回，然后再向东返回，最终到达输出。



理解拆分器的关键是它们的属性。在下文中，术语拆分端是指一侧的多条导线中的一条，而术语组合端是指另一侧的单条导线。

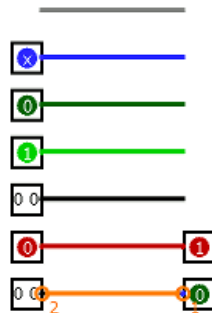
- “面”属性指示分割端点相对于组合端点的位置。

- “扇出”（Fan Out）属性指定有多少个分割端点。
- “输入位宽度”属性指定组合端点的位宽度。
- “位 x”属性表示哪个分割端对应于组合端的位 x。如果多个位对应于同一个拆分端，那么它们的相对顺序将与组合端相同。Logisim 拆分器的组合端不能有对应于多个拆分端的位。

请注意，对“扇出”（Fan Out）或“位宽度输入”（Bit Width In）属性的任何更改都将重置所有“位 x”属性，以便它们将组合值的位尽可能均匀地分布在拆分端。

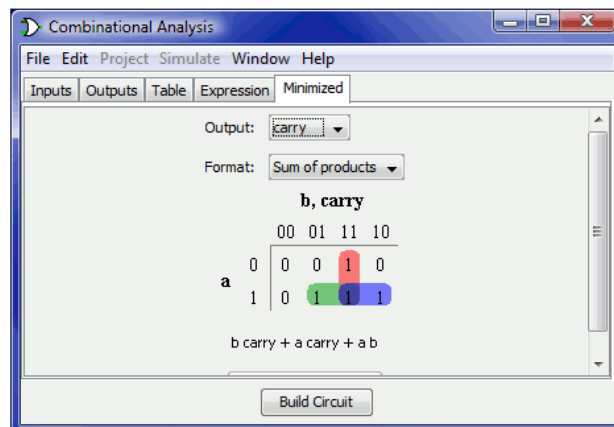
导线颜色

我们现在可以总结出 Logisim 电线可以呈现的彩虹色。下面的小电路一次演示了所有这些颜色。



- 灰色：导线的位宽度未知。这是因为导线未连接到任何组件的输入和输出。（所有输入和输出都有定义的位宽度。）
- 蓝色：导线携带一个位值，但没有任何东西将特定值驱动到导线上。我们称之为浮动位；有些人称之为高阻抗值。在本示例中，将值放置到导线上的元件是一个三状态接点，因此它可以发出此浮动值。
- 深绿色：导线携带一位 0 值。
- 亮绿色：导线带有一位 1 值。
- 黑色：导线带有多位值。部分或全部位可能未指定。
- 红色：导线带有错误值。这通常是因为门无法确定正确的输出，可能是因为它没有输入。这也可能是因为两个组件试图向导线发送不同的值；这就是上例中发生的情况，其中一个输入引脚将 0 放置在导线上，而另一个将 1 放置在同一导线上，从而导致冲突。当携带的任何位为错误值时，多位线将变为红色。
- 橙色：连接到导线的组件在位宽度上不一致。橙色导线实际上是“断开”的：它在组件之间不携带值。这里，我们将一个两位组件附加到一位组件，因此它们是不兼容的。

第五节：组合逻辑分析



所有电路都属于两个众所周知的类别之一：在组合电路中，所有电路输出都是电流电路输入的严格组合，而在顺序电路中，一些输出可能取决于过去的输入（随时间变化的输入序列）。

组合电路是这两种电路中比较简单的一种。从业者使用三种主要技术来总结此类电路的行为。

- 逻辑电路
- 布尔表达式，允许电路工作方式的代数表示
- 真值表，列出所有可能的输入组合和相应的输出

Logisim 的组合分析模块允许您在这三种表示法之间进行全方位的转换。这是一种创建和理解具有少量一位输入和输出的电路的特别方便的方法。

打开组合逻辑分析窗口

组合分析模块的大部分内容是通过一个同名窗口访问的，允许您查看真值表和布尔表达式。此窗口可以通过两种方式打开。

通过窗口菜单

选择组合分析，将显示当前的组合分析窗口。如果您以前没有查看过窗口，则打开的窗口将表示根本没有电路。

无论打开多少个项目，Logisim 中只存在一个组合分析窗口。无法同时打开两个不同的分析窗口。

通过“项目”菜单

在用于编辑电路的窗口中，还可以通过从“项目”菜单中选择“分析电路”选项来请求 Logisim 分析当前电路。在 Logisim 打开窗口之前，它将计算布尔表达式和对应于电路的真值表，并将它们放在那里供您查看。

为了成功进行分析，每个输入必须连接到输入引脚，每个输出必须连接到输出引脚。Logisim 将只分析每种类型最多八个的电路，并且都应该是单位引脚。否则，您将看到一条错误消息，窗口将无法打开。

在构造与电路相对应的布尔表达式时，Logisim 将首先尝试构造与电路中的门精确对应的布尔表达式。但是，如果电路使用一些非门元件（例如多路复用器），或者如果电路深度超过 100 层（不太可能），则会弹出一个对话框，告诉您无法推导布尔表达式，Logisim 将根据真值表推导表达式，它将通过悄悄尝试每个输入组合并读取结果输出而得到。

分析电路后，电路与“组合分析”窗口之间没有连续关系。也就是说，对电路的更改不会反映在窗口中，对窗口中布尔表达式和/或真值表的更改也不会反映在电路中。当然，您可以随时重新分析电路；并且，正如我们稍后将看到的，您可以与“组合分析”窗口中显示的内容相对应的电路替换该电路。

局限性

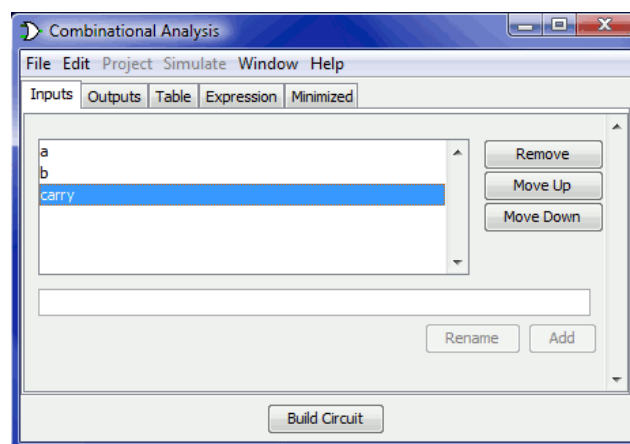
Logisim 不会尝试检测时序电路：如果您告诉它分析时序电路，它仍然会创建一个真值表和相应的布尔表达式，尽管这些不会准确地总结电路行为。（事实上，检测顺序电路被证明是不可能的，因为这将相当于解决暂停问题。当然，您可能希望 Logisim 至少做一些尝试，也许在导线中寻找触发器或周期，但事实并非如此。）因此，组合分析系统不应该被任意使用：只有当你确实确定你正在分析的电路确实是组合电路时才使用它！

Logisim 将对原始电路进行可能出乎意料的更改：组合分析系统要求每个输入和输出都有一个符合 Java 标识符规则的唯一名称。（大概每个字符都必须是字母或数字，第一个字符必须是字母。不允许有空格！）它尝试使用管脚的现有标签，如果不存在标签，则使用默认列表。如果现有标签不遵循 Java 标识符规则，Logisim 将尽可能尝试从标签中提取有效名称。

顺便说一句，真值表中输入的顺序将与原始电路中输入的自上而下顺序相匹配，连接按左右顺序断开。（这同样适用于输出的排序。）

编辑真值表

打开 Combinational Analysis 窗口时，您将看到它由五个选项卡组成。



本页介绍前三个选项卡：输入、输出和表格。指南的下一页介绍最后两个选项卡，表达式和最小化。

输入和输出选项卡

输入选项卡允许您查看和编辑输入列表。要添加新输入，请在窗格底部的字段中键入，然后单击添加。如果要重命名现有输入，请在窗格左上角区域的列表中选择它；然后键入名称并单击“重命名”。

要删除输入，请从列表中选择它，然后单击“删除”。您还可以使用输入上的“上移”或“下移”按钮对输入重新排序（这会影响真值表和生成的电路中的顺序）。

所有操作都会立即影响真相表。

Outputs 选项卡的工作方式与 Inputs 选项卡完全相同，当然不同的是它使用的是输出列表。

表格选项卡

Table 选项卡下的唯一项是当前真值表，按常规顺序绘制，输入构成左侧的列，输出构成右侧的列。

您可以通过单击感兴趣的值来编辑输出列中显示的当前值。这些值将在 0、1 和 x 之间循环（表示“不在乎”）。正如我们将在下一页看到的那样，任何不精确的值都允许最小化表达式的计算具有一定的灵活性。

您还可以使用键盘导航和编辑真相表。您可以使用剪贴板复制和粘贴值。剪贴板可以传输到任何支持制表符分隔文本的应用程序（如电子表格）。

如果真值表基于现有电路，您可能在输出列中看到一些带“!!”的粉红色方块在他们身上。这些错误对应于计算该行的值时发生的错误-电路似乎在振荡，或者输出值是错误值（在 Logisim 电路中显示为红线）。将鼠标悬停在条目上应该会显示一个工具提示，说明错误的类型。单击错误条目后，您将进入 0-1-x 周期；没有办法回去。

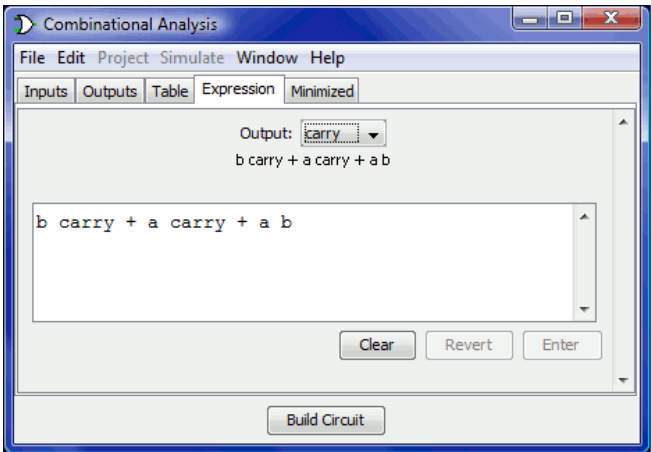
创建表达式

对于每个输出变量，组合分析窗口维护两个结构-真值表的相关列和布尔表达式-指定每个输出与其输入的关系。您可以编辑真值表或表达式；另一个将根据需要自动更改以保持一致。

正如我们将在下一页看到的那样，布尔表达式特别有用，因为组合分析窗口在被告知构建与当前状态对应的电路时将使用这些表达式。

可以使用窗口的最后两个选项卡（“表达式”选项卡和“最小化”选项卡）查看和编辑表达式。

表达式选项卡



表达式选项卡允许您查看和编辑与每个输出变量关联的当前表达式。您可以使用窗格顶部标记为“output:”的选择器选择要查看和编辑的输出表达式。

在选择器的正下方，将出现以一种特别常见的符号格式化的表达式，其中 OR 表示为加法，AND 表示为乘法，NOT 表示为受 NOT 影响的部分上方的条。

下面的文本窗格以 ASCII 格式显示相同的信息。这里，NOT 用波浪号 (“~”) 表示。

您可以在文本窗格中编辑表达式，然后单击 Enter 按钮使其生效；这样做还将更新真值表以使其对应。“清除”按钮清除文本窗格，“还原”按钮将窗格改回表示当前表达式。

请注意，如果编辑真值表，则编辑的表达式将丢失。

除了表示 and 和 OR 的乘法和加法之外，您键入的表达式可以包含任何 C/Java 逻辑运算符，也可以只包含单词本身。

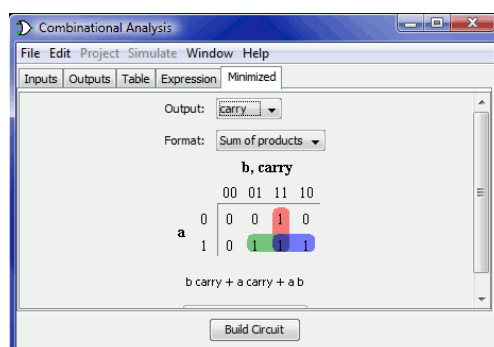
highest precedence	~ ! ' NOT
	(none) & && AND
	^ XOR
lowest precedence	+ OR

以下示例都是同一表达式的有效表示。您还可以混合使用运算符。

- a' (b + c)
- !a && (b || c)
- NOT a AND (b OR c)

通常，AND 序列（或 or 或 XOR）中的括号无关紧要。（特别是，当 Logisim 创建相应的电路时，它将忽略这些括号。）

最小化选项卡



最后一个选项卡显示对应于真值表列的最小化表达式。您可以使用顶部的选择器选择要查看的输出的最小化表达式，并可以使用下面的选择器指示是要派生乘积和表达式还是求和乘积表达式。

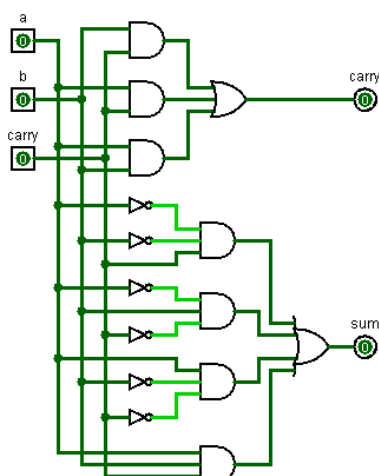
如果有四个或更少的输入，则与变量对应的卡诺图将显示在选择器下方。您可以单击卡诺图来更改相应的真值表值。卡诺图还将以实心半透明圆形矩形显示当前选定的最小化表达式术语。

下面是最小化表达式本身，格式与“表达式”选项卡的显示相同。如果有四个以上的输入，卡诺图将不会出现；但仍将计算最小化表达式。(Logisim 使用 Quine-McCluskey 算法计算最小化表达式。这相当于卡诺图，但它适用于任何数量的输入变量。)

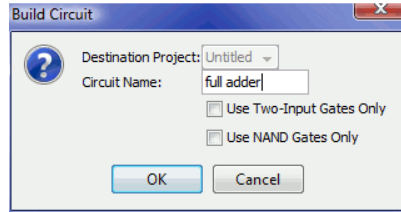
“设置为表达式”按钮允许您选择最小化表达式作为变量对应的表达式。这通常是不必要的，因为对真值表的编辑会导致对更改的列使用最小化表达式；但是，如果通过“表达式”选项卡输入表达式，则可以方便地切换到相应的最小化表达式。

生成电路

“构建电路”按钮将构建一个电路，其门对应于每个输出的当前选定表达式。电路的输入和输出将以自上而下的顺序显示，对应于它们在输入和输出选项卡下的显示方式。一般来说，所构建的电路将具有吸引力；事实上，Logisim 的组合分析模块的一个应用是美化绘制不良的电路。尽管如此，与任何自动格式化一样，它不会表达人工绘制电路所能表达的结构细节。



单击“构建电路”按钮时，将出现一个对话框，提示您选择要在其中创建电路的项目以及要为其指定的名称。



如果键入现有电路的名称，则该电路将被替换（在 Logisim 提示您确认确实要执行此操作后）。

“构建电路”对话框包含两个选项。“仅使用两个输入门”选项指定要构造的所有门都有两个输入。（当然，非门构成此规则的例外。）“仅使用与非门”选项指定您希望它将电路转换为仅使用与门的电路。如果只想使用两个输入与非门，则可以选择这两个选项。

Logisim 无法为包含任何 XOR 运算符的表达式构造仅 NAND 电路。因此，如果任何输出表达式包含 XOR，则将禁用此选项。

第六节：菜单功能

本节介绍每个主要 Logisim 窗口附带的六个菜单。

- 文件菜单
- 编辑菜单
- 项目菜单
- 模拟菜单
- 窗口和帮助菜单

许多菜单项专门与当前打开的项目相关。但一些 Logisim 窗口（尤其是 Combinational Analysis 窗口和 Application Preferences 窗口）与项目没有关联。对于这些窗口，将禁用项目特定的菜单项。

文件菜单

New	在新窗口中打开新项目。项目最初将是当前选定模板的副本
Open...	在新窗口中将现有文件作为项目打开
Open Recent	在新窗口中打开最近打开的项目，而不提示用户浏览文件选择对话框
Close	关闭与当前查看的项目关联的所有窗口
Save	保存当前查看的项目，覆盖文件中以前的内容
Save As...	保存当前查看的项目，提示用户保存到与以前不同的文件中
Export Image...	创建与电路对应的图像文件。配置对话框如下所述
Print...	将电路发送到打印机打印。配置对话框如下所述
Preferences...	显示应用程序首选项窗口
Exit	关闭所有当前打开的项目并退出 Logisim 程序

导出配置

选择“导出图像...”时，Logisim 将显示一个包含四个选项的对话框。

- 电路：一个列表，从中可以选择一个或多个应导出到图像文件中的电路。（空电路不会显示为选项。）
- 图像格式：您可以创建 PNG、GIF 和 JPEG 文件。我建议使用时使用 PNG 文件：GIF 格式已经过时了，JPEG 格式会在图像中引入伪影，因为 JPEG 真正适用于摄影图像。

- 比例因子：使用此滑块，可以在图像转储到图像文件中时缩放图像。
 - 打印机视图：在导出电路时是否使用“打印机视图”。
- 单击确定后，Logisim 将显示文件选择对话框。如果选择了一个电路，请选择要将图像放置到其中的文件。如果选择了多个电路，请选择应放置文件的目录；Logisim 将根据电路名称（例如 main.png）为图像命名。

打印配置

当您选择“Print...”时，Logisim 会显示一个对话框，用于配置打印的内容。

- 电路：一个列表，从中可以选择要打印的一个或多个电路。（空电路不显示为选项。）Logisim 将每页打印一个电路。如果电路对于页面来说太大，图像将缩小以适合页面。
- 页眉：应在每页顶部居中显示的文本。以下替换将纳入文本。

%n	Name of circuit on page
%p	Page number
%P	Total page count
%%	A single percent sign ("%")

- 旋转到合适的位置：如果选中此选项，则当电路太大而无法适合页面时，Logisim 会将每个电路旋转 90 度，并且旋转 90 度时不需要将其缩放到最小。
- 打印机视图：在打印电路时是否使用“打印机视图”。

单击“确定”后，Logisim 将在打印电路之前显示标准页面设置对话框。

编辑菜单

Undo XX	撤消最近完成的影响线路在文件中保存方式的操作。请注意，这不包括对电路状态的更改（与 Poke Tool 执行的操作一样）。
Cut	<p>将当前选定的元件从电路中删除到 Logisim 的剪贴板上。</p> <p>注：Logisim 的剪贴板与整个系统的剪贴板分开维护；因此，剪切/复制/粘贴无法在不同的应用程序之间工作，甚至包括 Logisim 的其他运行副本。但是，如果在同一个 Logisim 流程下打开了多个项目，那么您应该能够在它们之间剪切/复制/粘贴。</p>
Copy	将电路中当前选定的元件复制到 Logisim 的剪贴板上。（请参见“剪切”菜单项下的注释）。
Paste	<p>将 Logisim 剪贴板上的组件粘贴到当前选择中。（请参见“剪切”菜单项下的注释。）</p> <p>粘贴组件时，它们不会立即被丢弃；相反，它们将以浅灰色绘制。在您移动选择或更改选择以使组件不再位于其中之前，它们实际上不会“丢弃”到电路中。</p> <p>这种奇怪行为的原因是：为了与其他行为保持一致，Logisim 必须在任何导线掉入电路后立即合并它们；此合并过程会更改电路中的现有导线。然而，当您从剪贴板粘贴导线时，您可能希望它们出现在不同的位置，合并过程中固有的更改将违背您的意愿。</p>
Delete	从电路中删除当前选择中的所有元件，而不修改剪贴板。
Duplicate	创建当前选择中所有零部件的副本。这类似于选择“复制”，然后选择“粘贴”，但“复制”不会修改或使用剪贴板。
Select All	选择当前电路中的所有元件。
Raise Selection	<p>此菜单项仅在编辑电路外观时可用。它将提升当前选定的对象，以便在当前与选择重叠的对象上绘制（或绘制）该对象。如果选择被多个对象重叠，则只会将其提升到最低的对象之上；反复选择菜单项，直到它达到应有的顺序。</p> <p>（确定两个任意对象是否重叠是很困难的。Logisim 使用一种算法，在两个对象中的每一个中选择几个随机点，并查看另一个对象中是否也有任何点。如果重叠很小（例如，小于任一</p>

	对象的 5%)，有时它将无法检测到重叠。)
Lower Selection	此菜单项仅在编辑电路外观时可用。它会降低当前选定的对象，以便在当前选择重叠的对象下方绘制（或绘制）该对象。如果选择的对象与多个对象重叠，则只会降低到最高对象的下方；反复选择菜单项，直到它达到应有的顺序。
Raise To Top	仅当编辑电路的外观时，此菜单项才可用，它会将当前选定的对象提升到所有其他对象之上。（锚点和端口是例外-它们总是在顶部。）
Lower To Bottom	仅当编辑电路的外观时，此菜单项才可用，它会降低当前选定的对象，以便在其上绘制所有其他对象。
Add Vertex	仅当编辑电路的外观并且在直线、多段线或多边形上选择了点时，此菜单项才可用，它会在图形上插入一个新顶点。在插入之前，选定点绘制为菱形。
Remove Vertex	仅当编辑电路的外观并且在多段线或多边形上选择了现有顶点时，此菜单项才可用，它会删除选定的顶点。在删除之前，选定顶点在表示顶点的正方形内绘制为菱形。Logisim 不允许删除只有三个顶点的多边形或只有两个顶点的折线上的顶点。

项目菜单

Add Circuit...	将新电路添加到当前项目中。Logisim 将坚持让您为新电路命名。名称不得与项目中的任何现有电路匹配。
Load Library	将库加载到项目中。您可以加载三种类型的库，如《用户指南》中其他部分所述。
Unload Libraries...	从项目中卸载当前库。Logisim 将不允许您卸载当前正在使用的任何库，包括包含出现在任何项目电路中的元件的库，以及那些带有出现在工具栏中的工具或映射到鼠标的工具的库。
Move Circuit Up	将当前显示的电路在项目中的电路列表上移一步，如资源管理器窗格中所示。
Move Circuit Down	将当前显示的电路在项目中的电路列表中下移一步，如资源管理器窗格中所示。
Set As Main Circuit	将当前显示的电路设置为主电路。（如果当前电路已经是项目的主电路，则此菜单项将灰显。）主电路的唯一意义在于，它是打开项目文件时第一个出现的电路。
Revert To Default Appearance	如果编辑了电路的外观，此菜单项会将外观恢复为具有凹口外观的默认矩形。仅当编辑电路的外观时，才会启用菜单项。
View Toolbox	将资源管理器窗格更改为显示已加载的项目电路和库的列表。
View Simulation Tree	将资源管理器窗格更改为显示当前仿真中的子电路层次。
Edit Circuit Layout	用于编辑元件布局的开关，该布局决定电路的工作方式。此菜单项通常被禁用，因为您通常会编辑布局。
Edit Circuit Appearance	用于编辑将电路用作另一个电路中的子电路时该电路的外观。默认情况下，电路表示为一个上端带有灰色凹口的矩形，但此菜单选项允许您为子电路绘制不同的外观。
Remove Circuit	从项目中删除当前显示的电路。Logisim 将阻止您删除用作子电路的电路，并阻止您删除项目中的最终电路。
Analyze Circuit	计算与当前电路对应的真值表和布尔表达式，并在“组合分析”窗口中显示它们。分析过程仅对组合电路有效。组合分析部分对分析过程进行了全面描述。

Get Circuit Statistics	<p>显示一个对话框，其中包含有关当前查看的电路使用的元件的统计信息。该对话框包括一个包含五列的表格：</p> <ul style="list-style-type: none"> ● 组件：组件的名称。 ● 库：组件所在库的名称。 ● 简单：元件直接出现在查看的电路中的次数。 ● 唯一：元件在电路层次中出现的次数，其中层次中的每个子电路仅计数一次。 ● 递归：元件在电路层次结构中出现的次数，其中我们将每个子电路计数为它在层次结构中显示的次数。 <p>通过考虑使用子电路部分中使用三个 2:1 复用器构建的 4:1 复用器，Unique 和 Recursive 之间的区别最容易解释。2:1 多路复用器包含两个 AND 门（4:1 电路不包含），因此 AND 门的唯一计数为 2；但如果您要使用此图构建 4:1 多路复用器，那么三个 2:1 多路复用器中的每一个实际上都需要 2 个 AND 门，因此递归计数为 6。如果使用的是加载的 Logisim 库中的电路，则这些元件将被视为黑盒：库中电路的内容不包含在唯一计数和递归计数中。</p>
Options...	打开“项目选项”窗口。

模拟菜单

Simulation Enabled	<p>如果选中，则查看的电路将是“活动”的：即，通过电路传播的值将随着电路的每次插入或更改而更新。</p> <p>如果检测到电路振荡，菜单选项将自动取消选中。</p>
Reset Simulation	清除有关当前电路状态的所有内容，使其看起来就像刚刚再次打开文件一样。如果正在查看子电路的状态，则会清除整个层次。
Step Simulation	将模拟向前推进一步。例如，信号可能在一个步骤中最终进入一个门，但门在下一个模拟步骤之前不会显示不同的信号。为了帮助识别整个电路中的哪些点发生了变化，任何值发生变化的点都用蓝色圆圈表示；如果子电路中包含任何已更改的点（或其子电路，递归地），那么它将被绘制为蓝色轮廓。
Go Out To State	当您通过子电路的弹出菜单深入研究其状态时，“输出到状态”子菜单将列出当前查看的电路状态上方的电路。选择一个将显示相应的电路。
Go In To State	如果您深入研究了一个子电路的状态，然后又移出，则此子菜单将列出当前电路下方的子电路。选择其中一个电路将显示相应的电路。
Tick Once	在模拟中向前移动一个周期。当您想要手动步进时钟时，尤其是当时钟不在当前查看的同一电路中时，这将非常有用。
Ticks Enabled	开始自动驱动时钟。只有当电路包含任何时钟设备（在布线库中）时，这才有效。默认情况下，该选项处于禁用状态。
Tick Frequency	<p>允许您选择驱动频率。例如，8 Hz 意味着该时钟每秒产生 8 个 tick（滴答）。tick 是计量时钟速率的基本单位。</p> <p>请注意，时钟周期速度将慢于滴答速度：最快的时钟将有一个滴答上升周期和一个滴答下降周期；如果滴答声发生在 8 赫兹，这样的时钟将具有 4 赫兹的上升/下降周期速率，即时钟速率为 4 赫兹。</p>
Logging...	进入日志模块，该模块有助于在模拟过程中自动记录和保存电路中的值。

窗口和帮助菜单

窗口菜单

Minimize	最小化（图标化）当前窗口。
Maximize	将当前窗口调整为其首选大小。
Close	关闭当前窗口。
Combinational Analysis	显示当前的组合分析窗口，而不更改其任何内容。
Preferences	显示“应用程序首选项”窗口。
Individual Window Titles	将相应的窗口移到前面。

帮助菜单

Tutorial	打开帮助系统至《成为 Logisim 用户指南》的“初学者教程”部分。
User's Guide	打开 Logisim 用户指南的帮助系统。
Library Reference	打开“库参考”的帮助系统。
About...	显示一个包含版本号的窗口，混合在启动屏幕图形中。

第七节：存储组件

RAM 和 ROM 组件是 Logisim 内置库中两个非常有用的组件。然而，由于它们可以存储大量信息，因此它们也是最复杂的两个组件。

关于它们如何在电路中工作的文档可以在库参考的 RAM 和 ROM 页面上找到。本节介绍用户查看和编辑内存内容的几种方式。

- Poking memory
- 弹出菜单和文件
- Logisim 的集成十六进制编辑器

Poking memory

您可以使用 Poke Tool 操作内存的内容，但其界面受到空间限制：除了最简单的编辑之外，您可能会发现集成的十六进制编辑器更加方便。

然而，要查看和编辑电路内的值，Poke Tool 有两种操作模式：您可以编辑显示的地址，也可以编辑单个值。要编辑显示的地址，请在显示矩形外单击。Logisim 将在顶部地址周围绘制一个红色矩形。

- 键入十六进制数字将相应地更改顶部地址。
- 键入 Enter 键将向下滚动一行。
- 键入 Backspace 键将向上滚动一行。
- 键入空格键将向下滚动一页（四行）。

要编辑特定值，请单击显示矩形中的值。Logisim 将围绕该地址绘制一个红色矩形。

- 键入十六进制数字将更改当前正在编辑的地址的值。
- 键入 Enter 键将移动到编辑显示器中其正下方的值（向下一行）。
- 键入 Backspace 键将转到编辑以前地址的值。
- 键入空格键将转到编辑以下地址的值。

弹出菜单和文件

除了所有组件通用的选项外，内存弹出菜单还包括四个选项：

- 编辑内容：打开十六进制编辑器来编辑内存内容。
- 清除内容：将内存中的所有值重置为 0。
- 加载图像...：根据在文件中找到的值，使用下面描述的格式重置内存中的所有值。
- 保存图像...：使用下面描述的格式将内存中的所有值存储到文件中。

用于映像文件的文件格式非常简单；用户可以编写一个程序，例如汇编程序，生成内存映像，然后将其加载到内存中。作为此文件格式的示例，如果我们有一个 256 字节的内存，其前五个字节是 2、3、0、20 和 -1，并且所有后续值都是 0，那么映像将是以下文本文件。

```
v2.0 raw
```

```
02
```

```
03
```

```
00
```

```
14
```

```
ff
```

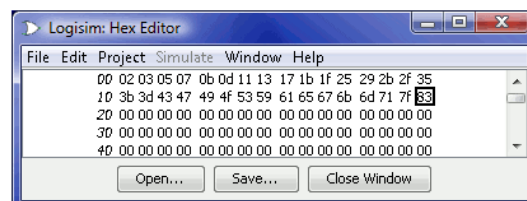
第一行标识所使用的文件格式（目前只支持这一种文件格式）。后续值以十六进制形式列出内存数值，从地址 0 开始；您可以在同一行上放置多个这样的值。如果内存位置比文件中标识的多，Logisim 会将 0 加载到其他内存位置。

映像文件可以使用行程编码；例如，文件可以写成 16*00，而不是将值 00 在一行中写 16 次。请注意，重复次数以 10 为基数。Logisim 生成的文件时，如果相同数据连续重复次数达到 4 次以上，将对其使用运行长度编码。

您可以使用“#”符号将注释放入文件中：Logisim 将忽略行中以“#”开头的所有字符。

Hex 编辑器

Logisim 包括一个集成的十六进制编辑器，用于查看和编辑内存内容。要访问它，请打开内存组件的弹出菜单并选择编辑内容...。对于将内存内容作为属性值一部分的 ROM 组件，您也可以通过单击相应的属性值来访问十六进制编辑器。



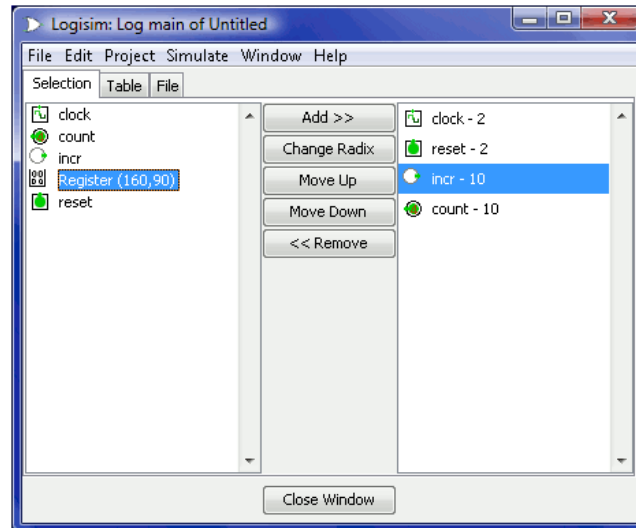
左侧斜体数字显示十六进制的内存地址。其他数字显示从该存储器地址开始的值；十六进制编辑器可以根据窗口中的大小，每行显示四个、八个或十六个值。为了便于计数，每组四个值之间有一个较大的间距。您可以使用滚动条或键盘（箭头键、主页、结束、向上翻页和向下翻页）在内存中导航。键入十六进制字符将更改当前选定的值。

您可以通过拖动鼠标、按住 shift 键并单击鼠标，或在按住 shift 键的同时使用键盘在内存中导航来选择一系列值。可以使用“编辑”菜单复制和粘贴值；剪贴板也可以传输到其他应用程序中。

第八节：日志

在测试大型电路和记录电路行为时，记录过去的电路行为可能很有用。这是 Logisim 日志模块的用途，它允许您选择其值应被记录的组件；或者，您可以指定一个文件，将日志放在其中。

您可以通过 Simulate 菜单中的 logging... 选项进入日志模块。它会打开一个包含三个选项卡的窗口。



我们将分别讨论这些选项卡。

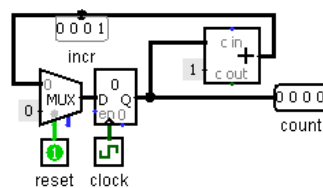
- 选择选项卡
- 表格选项卡
- 文件选项卡

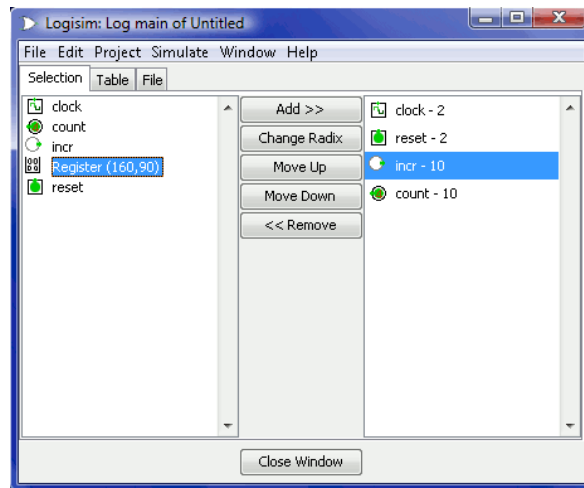
每个项目只有一个日志记录窗口；当您切换到查看项目中的另一个电路时，日志窗口会自动切换到记录其他电路。也就是说，除非您在同一模拟中向上或向下移动，否则日志模块不会改变。

请注意，当日志模块切换到记录另一个模拟时，它将停止记录到文件中。如果您再次切换回模拟，它将记住该模拟的配置，但您需要手动重新启用文件日志记录。

“选择”选项卡

选择选项卡允许您选择日志中应包含的值。下面的窗口对应于以下电路。





选项卡分为三个垂直区域。第一个（最左边）是电路中可以记录其值的所有元件的列表。在内置库中，以下类型的组件支持日志记录。

- 线路库：引脚、探针和时钟组件
- I/O 库：按钮和 LED 组件
- 内存库：除 ROM 外的所有组件

对于有相关标签的组件，其名称与标签相对应；其他元件的名称指定了它们的类型及其在电路中的位置。任何子电路也将出现在列表中；不能选择它们进行日志记录，但可以选择其中符合条件的组件。注意，RAM 组件要求您选择应该记录的内存地址；它只允许记录前 256 个地址。

最后一个（最右侧）垂直区域列出了已选定的组件。此外，它还指示记录组件多位值的基数（基数）；基数对一位值没有显著影响。

中间一栏按钮允许操作所选条目的内容。

- Add：将左侧当前选定项目添加到选定内容中。
- Change Radix：在 2（二进制）、10（十进制）和 16（十六进制）之间循环选择当前选定组件的基数。
- Move Up：将左侧当前选定项目的零部件向前移动一个点。
- Move Down：将左侧当前选定项目的零部件向后移动一个点。
- Remove：将删除左侧当前选定项目的零部件。

表格选项卡

表格选项卡以图形方式显示当前日志。

clock	reset	incr	count
0	0	1	0
1	0	2	1
0	0	2	1
1	0	3	2
0	0	3	2
1	0	4	3
0	0	4	3
1	0	5	4
0	0	5	4
0	1	5	4
1	1	1	0
1	0	1	0
0	0	1	0

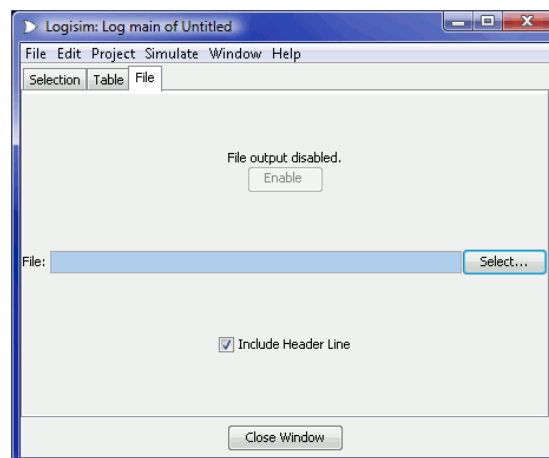
该表包含所选内容中每个组件的一列。表中的每一行显示值传播完成后模拟的快照。任何重复的行都不会添加到日志中。请注意，仅显示最近的 400 行。如果计算行时选择的内容中没有相应的组件，则某些行可

能包含空条目。

显示的表格仅供查看；它不是交互式的。

“文件”选项卡

“文件”选项卡允许您指定日志应放置到的文件。



顶部是文件日志记录是否正在进行的指示器，以及启用或禁用该日志记录的按钮。（请注意，在选择下面的文件之前，您无法启用它。）该按钮允许您暂停并重新启动文件输入。当您在项目窗口中切换到查看另一个模拟时，文件记录将自动停止；如果返回到原始日志并希望继续记录，则需要使用顶部的按钮手动重新启动文件日志记录。

中间是一个指示正在记录到哪个文件的指示器。要更改它，请使用“选择...”按钮。选择文件后，将自动启动文件日志记录。如果您选择一个预先存在的文件，Logisim 将询问您是要覆盖该文件还是将新条目附加到末尾。

在底部，您可以控制是否应在文件中放置标题行，以指示选择中的项目。如果添加了标题行，则每当选择更改时，都会在文件中放置一个新的标题行。

文件格式

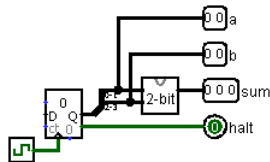
条目以制表符分隔的格式放置到文件中，与“表格”选项卡下显示的内容密切相关。（一个不同之处是，任何标题行都会给出子电路中组件的完整路径。）该格式有意简单，以便您可以将其输入到另一个程序中进行处理，例如 Python/Perl 脚本或电子表格程序。

为了使脚本可以在 Logisim 运行的同时处理该文件，Logisim 将每隔 500 毫秒将新记录刷新到磁盘上。请注意，在模拟过程中 Logisim 也可能间歇性地关闭文件，然后重新打开文件，特别是在几秒钟后没有添加任何新记录的情况下。

第九节：命令行验证

Logisim 包括从命令行执行电路的基本支持。这既有助于电路设计的脚本验证，也有助于教师对学生的解决方案进行自动化测试。

我们将从如何以命令行执行电路开始。对于我们的示例，我们假设已经在名为 `adder-test.circ` 的文件中构建了以下电路。它使用一个两位加法器作为子电路，并使用计数器遍历所有 16 个可能的输入。



构建完这个电路后，我们从命令行执行 Logisim，提供项目的文件名和带有表参数的 -tty 选项。

```
java -jar logisim-filename.jar adder-test.circ -tty table
```

在不打开任何窗口的情况下，Logisim 加载电路并开始执行它，以尽可能快的速度计时，同时完成每个计时之间的传播。每次传播完成后，Logisim 加载输出引脚的当前值；如果与上一次传播相比有任何更改，则所有值都将以制表符分隔格式显示。如果有一个标有特殊单词 halt 的输出管脚，则不会显示其输出，但一旦传播完成后管脚的值达到 1，Logisim 将结束模拟。

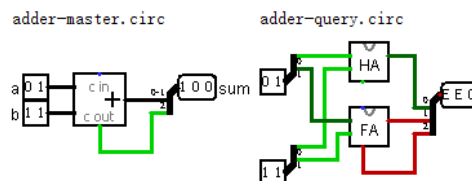
对于我们的示例，Logisim 显示了下表。因为我们有二个输出引脚，对应于两位加法器的二个输入 a 和 b，所以这些输出包括在输出的前两列。还有另一个输出引脚对应于两位加法器的输出，所以它是第三列。这些列是根据电路中的自上而下顺序从左到右排序的。

00	00	000
01	00	001
10	00	010
11	00	011
00	01	001
01	01	010
10	01	011
11	01	100
00	10	010
01	10	011
10	10	100
11	10	101
00	11	011
01	11	100
10	11	101
11	11	110

替换库

现在假设我们有两个逻辑电路，它们应该做同样的事情。作为一名讲师，你可能已经让学生完成了一项作业：你有一个文件包含你的解决方案，但你有几个学生文件包含他们的作业。也许任务是构建一个两位加法器。

设想我们有两个文件，名为 adder-master.circ 和 adder-query.circ。每个文件都包含一个名为 2 位加法器的电路（重要的是要测试的电路名称完全相同），其外观如下。



如您所见，adder-master.circ 使用 Logisim 的内置加法器，而 adder-query.circ 使用两个子电路表示半加法器和全加法器（它们本身由简单的门构成）。在我们的例子中，adder-query.circ 有一个愚蠢的错误：半加

法器的进位没有连接到全加法器。

我们将测试电路构建到不同的文件中。在那里，我们加载 `adder-master.circ` 作为 Logisim 库 (Project>Load Library>Logisim Library...)，我们将其 2 位加法器作为子电路插入。我们可以直接执行此电路，以获得理想的输出，从而获得完美的解决方案。

```
java -jar logisim-filename.jar adder-test.circ -tty table
```

但我们想使用 `adder-query.circ` 而不是 `adder-master.circ` 作为加载的库。最简单的方法是打开 Logisim 并加载该库；或者您可以简单地删除 `adder-master.circ` 文件和重命名 `adder-query.circ` 为 `adder-master`。但 Logisim 包含一个方便的子选项，可以在会话期间临时将一个文件替换为另一个文件，而不会在磁盘上进行任何更改。

```
java -jar logisim-filename.jar adder-test.circ -tty table -sub adder-master.circ adder-query.circ
```

您将从中看到的输出如下所示：当然，它与我们在前一节中看到的有所不同，因为现在它是使用错误的 `adder-query.circ` 执行的。

00	00	0E0
01	00	0E1
10	00	EE0
11	00	EE1
00	01	0E1
01	01	0E0
10	01	EE1
11	01	EE0
00	10	EE0
01	10	EE1
10	10	1E0
11	10	1E1
00	11	EE1
01	11	EE0
10	11	1E1
11	11	1E0

其他验证选项

还有一些与命令行执行相关的附加选项。

-load 命令行参数

一个更复杂的电路可能包括一个 RAM 组件，需要加载一个程序，以便电路可以做任何事情。您可以在命令行指定内存图像文件，该文件将在模拟开始之前加载到电路中的任何 RAM 组件中。(这在加载 GUI 时不起作用-它仅用于命令行执行。)

```
java -jar logisim-filename.jar cpu.circ -tty table -load mem-image.txt
```

参数的顺序并不重要（表参数必须紧跟 `-tty` 之后，内存映像的文件名必须紧跟 `-load` 之后）。内存映像文件应为 Logisim 的内存映像格式。

Logisim 递归搜索 RAM，因此如果 RAM 嵌套在子电路中，这仍然有效。但是，无法区分不同的 RAM 组件：Logisim 会尝试将同一个文件加载到它能找到的每个 RAM 中。

-tty 参数的选项

到目前为止，在我们的示例中，我们一直使用 `-tty` 表来指示应该显示输出值表。您可以通过列出一个或多个以逗号分隔的选项，以其他方式自定义行为。例如，您可以编写 `-tty` 表、暂停、速度，程序将执行下面列

出的所有三个行为。（它们的列出顺序无关紧要。）

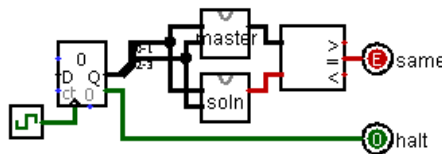
halt	模拟结束后，将显示一条单行消息，解释模拟结束的原因。在任何情况下都会显示错误条件，例如检测到的振荡。
speed	请注意，在模拟过程中显示信息会使模拟速度慢得多。作为一个比较，相同的电路和图像仅在速度选项下以 714 Hz 以上的频率运行，但在表格选项下也以 490 Hz 的频率运行。
stats	<p>显示一个以制表符分隔的表，其中包含有关项目中顶级主电路使用的元件的统计信息。该表包括四列：</p> <ul style="list-style-type: none"> ● 唯一：元件在电路层次中出现的次数，其中层次中的每个子电路仅计数一次。 ● 递归：元件在电路层次结构中出现的次数，其中我们将每个子电路计数为它在层次结构中显示的次数。 ● 组件：组件的名称。 ● 库：组件所在库的名称。 <p>“项目”菜单部分进一步解释了“唯一”和“递归”之间的区别。如果文件使用加载的 Logisim 库中的电路，则这些元件将被视为黑盒：库电路的内容不包含在唯一和递归计数中。（此功能对于指定学生使用 Logisim 库子集构建项目的教师非常有用。）</p>
table	（如前所述）
tty	任何 TTY 组件都会将其输出发送到显示器（标准输出），在键盘上键入的任何信息都会发送到电路中的所有键盘组件。即使这些组件深深嵌套在子电路层次结构中，也会包含这些组件。

测试多个文件

在课堂示例中，您将有許多要测试其等效性的文件，并且您不希望读取每个学生解决方案的输出。

在电路中建立比较器

一种方法是构建一个直接进行比较的测试电路。在这里，我们在包含我们的解决方案电路的测试文件中创建了一个额外的电路。在我们的整体测试电路中，我们包括了加法器主电路的两个子电路。电路和子电路从解决方案电路直接定位到嵌套电路中。我们把它连接起来，这样只有一个输出，只要两个子电路一致，这个输出就是 1。



现在我们可以简单地运行 Logisim 来替换每个查询文件。对于任何正确的解决方案，唯一的输出都是 1。

使用重定向和 shell 脚本

如果您对命令行很熟悉，可以构建自己的 shell 脚本来完成此任务。这里，我们将使用重定向（>操作符）将每个电路的输出保存到一个文件中。例如，我们可能发出以下两个命令来收集 master circuit 和 query circuit 的输出。

```
java -jar logisim-filename.jar adder-test.circ -tty table > output-master.txt
```

```
java -jar logisim-filename.jar adder-test.circ -tty table -sub adder-master.circ adder-query.circ > output-query.txt
```

现在我们已经创建了两个不同的文件。然后，我们可以使用为此目的构建的程序来比较这两个输出文件。

要处理多个查询文件，您可能需要构建一个简单的程序，例如一个 shell 脚本，来遍历每个文件并比较输出。

下面是在 Linux bash 下的一种操作方法：

```
RUN_TEST="java -jar logisim-filename.jar adder-test.circ -tty table"
```



```

${RUN_TEST} > output-master.txt
for QUERY_FILE in adder-query*.circ
do
    if ${RUN_TEST} -sub adder-master.circ ${QUERY_FILE} | cmp -s output-master.txt
    then
        echo "${QUERY_FILE} OK"
    else
        echo "${QUERY_FILE} different"
    fi
done

```

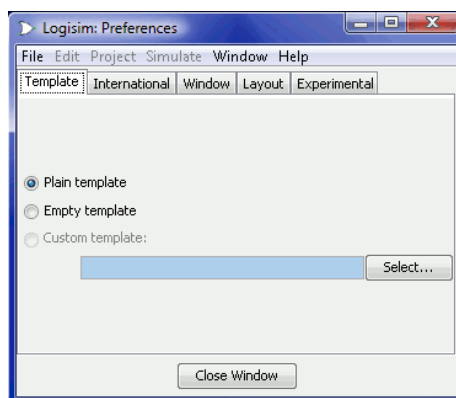
第十节：应用偏好

Logisim 支持两类配置选项：应用程序首选项和项目选项。应用程序首选项处理跨越所有打开项目的首选项，而项目选项特定于该项目。本节讨论应用程序首选项；项目选项将在另一节中介绍。

您可以通过 File（文件）菜单（或 Mac OS 下的 Logisim 菜单）中的 preferences…（首选项…）选项查看和编辑应用程序首选项，窗口将显示多个选项卡。我们将分别讨论这些选项卡，然后我们将看到如何从命令行配置首选项。

- “模板”选项卡
- “国际”选项卡
- “窗口”选项卡
- “布局”选项卡
- “实验”选项卡
- 命令行

模板选项卡



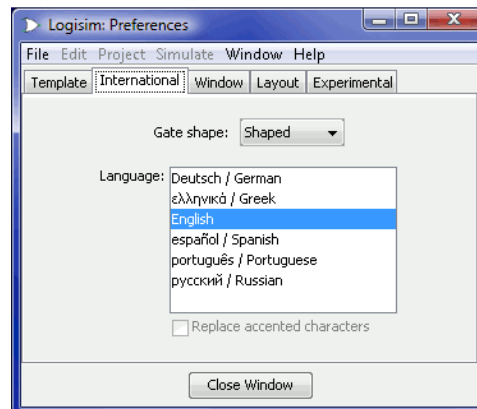
模板实际上是一个 Logisim 文件，每当 Logisim 创建新项目时，它都会用作起点。此外，如果您有一个具有奇怪配置环境的现有 Logisim 文件，则可以使用窗口中用于编辑“项目选项”的“全部还原为模板”按钮“重置”环境。

尽管模板在其他情况下也很有用，但它们特别适合课堂使用，在课堂上，教师可能希望分发模板供学生开始使用。特别是如果课堂上大量使用 Logisim，包括许多更高级的特性，那么这种情况尤其可能发生，在这种情况下，简单的默认配置可能被证明过于简单。当讲师打开一个学生提交的文件时，模板在课堂设置中

也很有用，该学生已经对环境进行了大量配置。

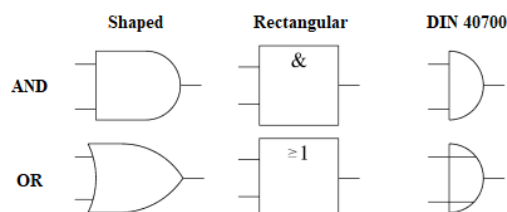
默认情况下，将使用 Logisim 附带的默认模板选择“普通模板”选项。如果您想要一个基本配置，您可以选择“Empty template”（清空模板）。但是如果您想要指定另一个文件作为模板，请通过 select…（选择…）按钮选择一个模板，然后选择“Custom templates”（自定义模板）选项。

“国际”选项卡



此选项卡允许根据区域首选项配置 Logisim。

- 逻辑门形状：Logisim 支持绘制逻辑门的三种标准：shaped gates、rectangular gates 和 DIN 40700 gates。下表说明了这种区别。



因为 shaped gates 风格在美国更流行，而 rectangular gates 风格在欧洲更流行，一些人根据这些地区来参考这些风格；但首选形状和矩形的区域中性术语。DIN 40700 标准是德国标准组织 DIN 采用的数字和模拟电子元件起草标准。1976 年，德国工业标准协会（DIN）对数字元件采用了矩形标准，但一些工程师继续使用旧的样式；它们似乎越来越罕见。

Logisim 没有严格遵循任何标准；它引导了一个中间地带，允许在它们之间切换。特别是，成形闸门比相关 IEEE 标准规定的尺寸更为方形。而且，尽管 XOR 和 XNOR 门实际上应该与矩形样式的 OR 和 NOR 门的宽度相同，但这并不是因为难以压缩成形的 XOR 门。

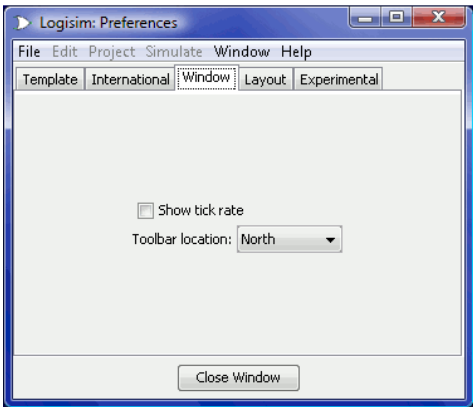
- 语言：语言之间的变化。当前版本提供了英语、西班牙语、俄语和德语翻译。
 - 德文译本是在 Logisim 2.6.1 中引入的，并保持最新。这是由瑞典乌普萨拉大学的一名教员 Uwe Zimmermann 写的。
 - 希腊语翻译采用 Logisim 2.7.0，并保持最新。它是由希腊爱奥尼亚群岛技术教育学院教师 Thanos Kakarountas 撰写的。
 - 葡萄牙语翻译是在 Logisim 2.6.2 中引入的，并保持最新。这是由 Theldo Cruz Franqueira 写的，他是蓬蒂夫大学的一名教员
 - 俄文译本采用 Logisim 2.4.0，并保持最新。这是来自俄罗斯的 Thanos Kakarountas 写的。
 - 自 Logisim 2.1.0 起，西班牙语翻译已完成，但随后的 Logisim 版本添加了未翻译的新选项。它是由西班牙的 Pablo Leal Ramos 提供的。

欢迎将 Logisim 翻译成其他语言！如果你有兴趣，请联系我，卡尔·伯奇。这不是一个承诺：我很高兴听到

你感兴趣，我会告诉你我是否知道有人已经在做这项工作，准备一个版本供你合作，并向你发送指示。翻译过程不需要了解 Java。

- 替换重音字符：一些平台对不出现在 7 位 ASCII 字符集中的字符（如“no örö”）的支持较差。选中此选项后，Logisim 将用适当的等效 7 位 ASCII 字符替换所有字符实例。如果当前语言没有可用的等效语言（如英语），则禁用该复选框。

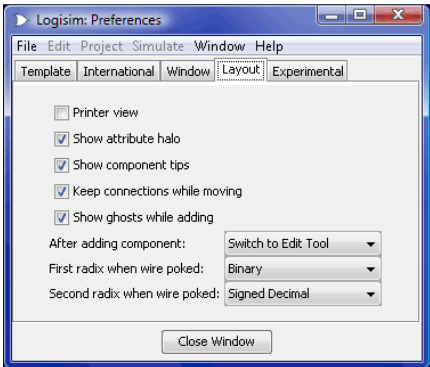
“窗口”选项卡



此选项卡包括影响 Logisim 所用主窗口外观的首选项。

- 显示 tick 速率：如果选中，则在启用 tick 时，Logisim 会显示其能够完成 tick 的速率测量值。tick 率是通过前 1000 次 tick 的平均值来测量的。（禁用 tick 或更改最大 tick 率将清除其历史记录。）由于 Logisim 无法以非常快的速度模拟较大的电路，因此实际的时钟频率可能远低于所选的时钟频率。例如，对于一个相当大的电路，Logisim 的最大速度可能是 16 赫兹；您可以选择更快的刻度率，但实际速度不会超过 16 赫兹。
- 工具栏位置：此下拉菜单配置工具栏在整个窗口中的位置。工具栏可以放置在窗口的四个边界中的任何一个，即北、南、东和西。它也可能被隐藏，或者可以“放在中间”，也就是说，它位于画布的左侧，但位于资源管理器窗格和属性表的右侧。

布局选项卡

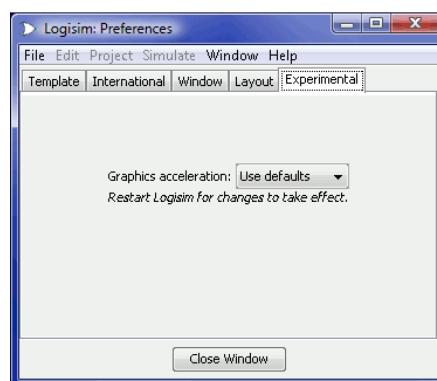


此选项卡包括影响电路布局编辑器行为的首选项。

- 打印机视图：指定是否以通过打印机显示电路的相同方式在屏幕上显示电路。通常这是关闭的，Logisim 在屏幕上显示带有当前电路状态指示的电路，并显示有关组件接口的一些提示（最明显的是，它在 OR 门上绘制支腿以指示它们将连接的位置）。不过，打印机视图省略了状态指示，也省略了此类接口提示。

- 显示属性光晕：指定是否围绕其属性当前显示在属性表中的组件或工具绘制淡蓝色椭圆形。
- 显示组件提示：指定是否显示当鼠标悬停在支持它们的组件上时将临时显示的“工具提示”。例如，如果将鼠标悬停在子电路元件的管脚上，它将显示子电路中相应管脚的标签。将鼠标悬停在拆分器的一端会告诉您该端对应的位。此外，Plexers、Arithmetic 和 Memory 库中的所有组件都将通过提示提供有关其输入和输出的信息。
- 移动时保持连接：指示在移动元件时 Logisim 是否应添加新导线以保持其连接。默认情况下，这是打开的，但可以在移动组件时按 shift 键暂时关闭。如果未选中此框，则默认情况下不会在移动过程中添加导线，但您可以在移动期间按 shift 键临时启用它。
- 添加时显示重影：选中此选项后，当选择用于添加新组件的工具时，当鼠标在画布上移动时，将绘制要添加的组件的浅灰色轮廓。例如，如果选择 AND 门工具并将鼠标移动到窗口中（不按鼠标按钮），则单击鼠标时，AND 门将显示灰色的轮廓。
- 添加组件后：默认情况下，添加每个单独的组件后，Logisim 会切换回“编辑工具”，以允许您移动组件和添加导线。下拉框允许您更改此行为，以便 Logisim 停留在同一工具上添加更多相同组件，直到您自己选择编辑工具。（这是 Logisim 2.3.0 之前的默认行为。虽然更直观，但此行为需要更多鼠标移动才能在工具之间切换。）
- 插入导线时的第一个基数：配置使用插入工具单击导线时的值显示方式。单击导线将临时显示该值，直到用户单击电路中的其他位置。
- 导线插入时的第二基数：配置导线值显示方式的第二部分。

实验选项卡



这些首选项支持被视为实验性的功能，插入这些功能以获得用户反馈。

- 图形加速：一位 Logisim 用户观察到添加了 -Dsun.java2d.d3d=True to the command line 似乎通过告诉 Logisim 使用硬件图形加速来提高其图形性能。此下拉框尝试配置 Logisim 以进行设置；欢迎报告此下拉框是否对性能有任何影响。在 Logisim 重新启动之前，它不会产生任何影响。

命令行选项

您可以通过命令行选项配置 Logisim 的许多应用程序首选项。这在一个只提供学生公用电脑的实验室中特别有用，您希望 Logisim 每次都为学生启动相同的程序，而不管以前的学生如何配置程序。

总体命令行语法如下。

```
java -jar jarFileName [options] [filenames]
```

命令行上命名的可选附加文件将在 Logisim 中作为单独的窗口打开。

以下示例在基本配置中启动 Logisim。

`java-jar jarFileName-plain-gates shaped-locale en`

支持的选项包括以下选项。

`-plain`

`-empty`

`-template templateFile`

配置 Logisim 使用的模板

`-gates [shaped|rectangular]`

配置所适用的逻辑门符号标准

`-locale localeIdentifier`

配置要使用的语言。Logisim 支持的语言环境包括：德语、英语西班牙语、俄语、希腊语

de German

en English

es Spanish

ru Russian

el Greek

`-accents [yes|no]`

这仅适用于使用 7 位 ASCII 字符集以外字符的语言；这将包括使用重音字符的语言，而不包括英语。如果不是，7 位 ASCII 字符集之外的字符将替换为适合该语言的等效字符；这对于 Java/OS 组合非常有用，因为这些字符不受很好的支持。

`-clearprops`

在启动时清除所有应用程序首选项，这样 Logisim 就会像第一次在主机系统上执行一样。

`-nosplash`

隐藏初始 Logisim 初始屏幕。

`-help`

显示命令行选项的摘要。

`-version`

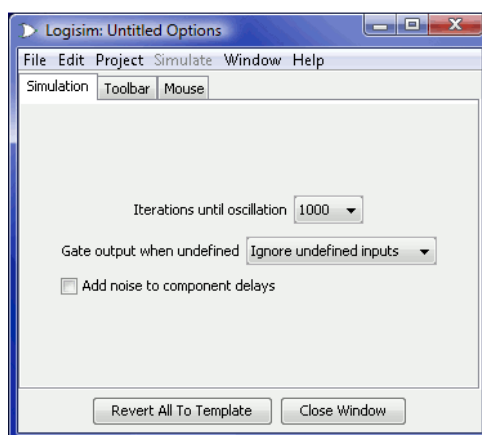
显示 Logisim 版本号。

第十一节：工程选项

项目选项

Logisim 支持两类配置选项：应用程序首选项和项目选项。应用程序首选项处理跨越所有打开项目的首选项，而项目选项特定于该项目。本节讨论项目选项；应用程序首选项将在另一节中介绍。

您可以通过“项目”菜单中的“选项…”选项查看和编辑项目选项。它会打开带有几个选项卡的“选项”窗口。



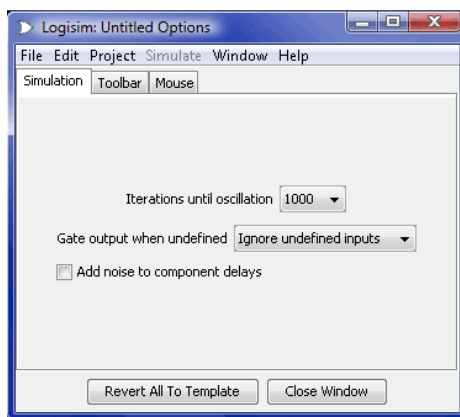
我们将分别讨论这些选项卡。

- “仿真”选项卡
- “工具栏”选项卡
- “鼠标”选项卡

窗口底部是“全部还原为模板”按钮。单击时，所有选项和工具属性都将更改为当前模板中的设置（在应用程序首选项下选择）。

“仿真”选项卡

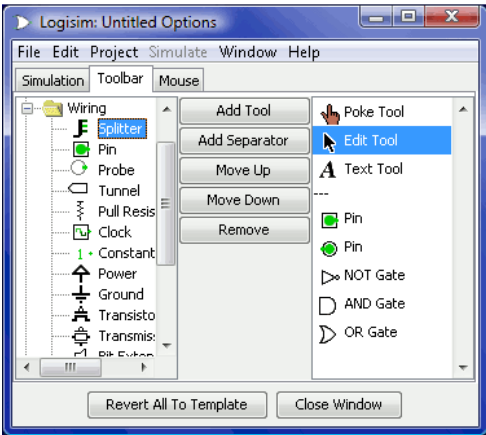
“仿真”选项卡允许配置用于仿真电路的算法。这些参数适用于在同一窗口中模拟的所有电路，即使是存在于项目中加载的其他库中的电路。



- “直到振荡为止的迭代次数”下拉菜单指定在确定电路振荡之前模拟电路的时间。数字表示内部隐藏时钟的点击次数（一个简单的门只需点击一次）。默认值 1000 对于几乎所有用途都足够了，甚至对于大型电路也是如此。但是，如果使用 Logisim 报告错误振荡的电路，则可能需要增加迭代次数。这在实践中不太可能是一个问题，但这样的情况之一是电路包含许多启用了随机噪声的低于锁存器的电路。如果您使用的电路容易振荡，并且您使用的处理器速度非常慢，那么您可能需要减少迭代次数。
- “未定义时的门输出”下拉菜单配置了内置逻辑门在某些输入未连接或浮动时的行为。默认情况下，Logisim 会忽略这些输入，从而允许门处理的输入比设计的更少。然而，在现实生活中，门在这种情况下行为是不可预测的，因此此下拉菜单允许更改门，以便将此类断开的输入视为错误。
- “向组件延迟添加噪波”复选框允许您启用或禁用添加到组件延迟的随机噪波。内部模拟使用一个隐藏的时钟进行模拟，为了提供更逼真的模拟，每个组件（不包括导线和分路器）在接收输入和发出输出之间都有延迟。如果启用此选项，Logisim 会偶尔（大约每 16 个组件反应一次）使组件的单击时间比正常时间长。我建议不要使用此选项，因为此技术确实会在正常电路中引入罕见错误。

工具栏选项卡

工具栏选项卡允许您配置工具栏中显示的工具。



左侧是一个资源管理器，列出了所有可用的工具，右侧的列表显示了工具栏的当前内容。（三个破折号“---”表示分隔符，以灰色线绘制。）在浏览器和列表之间有五个按钮和一个组合框：

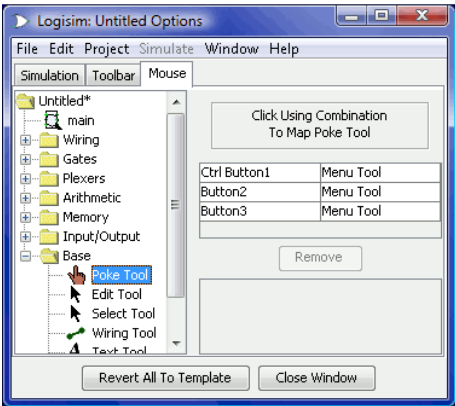
- 添加工具将资源管理器中当前选定的工具添加到工具栏左侧的末尾。
- 添加分隔符将分隔符添加到工具栏的末尾。
- “上移”将工具栏中当前选定的项目上移/左移一个位置。
- “下移”将工具栏中当前选定的项目下移/右移一个位置。
- 移除将从工具栏中移除当前选定的项目。

与工具关联的属性不在此窗口中显示；相反，您可以在主图形窗口中查看和编辑它们。

鼠标选项卡

默认情况下，当您在 Logisim 的绘图区域中单击鼠标时，将使用当前选定的工具。如果右键单击或按住 Ctrl 键单击，它将在鼠标下方显示当前组件的弹出菜单。

Logisim 允许您修改此行为，从而无需一直转到工具栏和/或资源管理器。（如果你是左撇子，这可能也很方便。）鼠标按钮和修改键（shift、control 和 alt 的任何子集）的每个组合都可以映射到不同的工具。鼠标选项卡允许您配置这些映射。



- 左侧是一个资源管理器，您可以在其中选择要映射的工具。
- 在右上方是一个矩形，您可以使用要单击的鼠标组合在其中单击。例如，如果要通过按住 shift 键并拖动来创建新导线，则应首先选择资源管理器中的 Wiring Tool（位于基础库下）；然后将鼠标移动到

显示“click Using Combination To Map Wiring Tool”的位置。如果该组合已被使用，则映射将被新工具替换。

- 在此区域下方是当前映射的列表。请注意，未列出的任何组合只使用当前选定的工具。
- 下面是“删除”按钮，从中可以删除按钮上方表格中当前选定的映射。那么，在未来，该鼠标组合将映射到工具栏或资源管理器窗格中当前选定的任何工具。
- 下面是映射列表中当前选定工具的属性列表。每个鼠标映射工具都有自己的一组属性，不同于资源管理器窗格和工具栏中使用的属性。您可以在此处编辑这些属性值。

第十二节：值传递

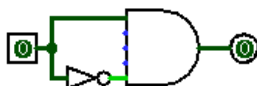
Logisim 用于模拟值在电路中传播的算法通常不需要担心。可以说，该算法足够复杂，足以解释门延迟，但不够现实，无法解释更困难的现象，如电压变化或竞争条件。

你还想知道更多吗？

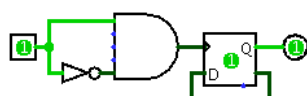
- 门延迟
- 振荡误差
- 缺点

闸门延迟

作为 Logisim 算法复杂程度的示例，请考虑以下电路。



这“显然”总是输出 0。但在现实中，NOT 门不会对它们的输入做出即时反应，在 Logisim 中也不会。因此，当该电路的输入从 0 变为 1 时，AND 门将短暂地看到两个 1 输入，并短暂地发出 1。你不会在屏幕上看到它。但当我们使用 AND 门的输出作为 D 触发器时钟的输入时，这种效果是可以观察到的。



将 0 输入插入为 1 会导致瞬时 1 进入 D 触发器，因此每次电路输入从 0 变为 1 时，触发器的值都会切换。每个组件都有一个相关的延迟。Logisim 中内置的更复杂的组件往往具有更大的延迟，但这些延迟有些武断，可能无法反映实际情况。

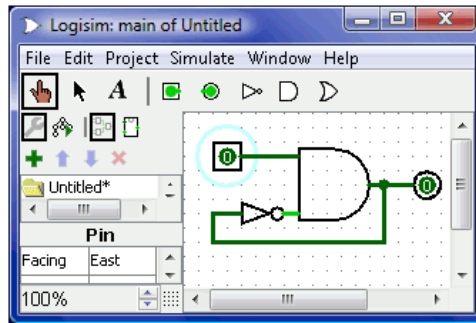
从技术角度来看，在单个电路中处理这种复杂程度相对容易。然而，处理好跨子电路的门延迟有点复杂；Logisim 确实试图通过将所有基本组件的传播值放入单个计划中来正确解决这个问题，而不考虑组件所在的子电路。

（通过“项目选项”窗口的“模拟”选项卡，您可以配置 Logisim 为组件传播添加随机、偶尔的延迟。这旨在模拟实际电路的不均匀性。特别是，使用两个 NOR 门构建的 R-s 门锁将在没有这种随机性的情况下振荡，因为两个门都将同步处理其输入。默认情况下，这种随机性被禁用。）

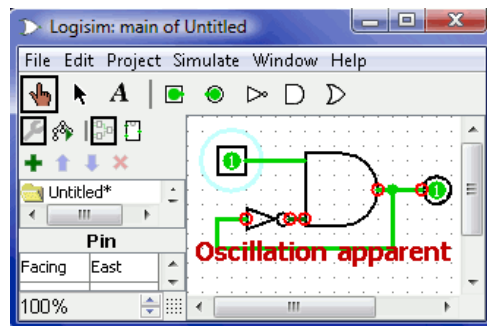
请注意，我没有说 Logisim 总是很好地解决门延迟问题。但至少它尝试了。

振荡误差

传播算法通常无声工作，没有任何问题，当您创建振荡的电路时，它将变得非常明显。



该电路目前处于稳定状态。但如果将输入更改为 1，电路将有效地进入无限循环。过了一会儿，Logisim 就会放弃，并显示“明显振荡”消息，告诉您它认为电路正在振荡。



它将显示放弃时的值。这些值看起来是错误的-在这个屏幕截图中，虽然 AND 门的一个输入为 0，但它发出的是 1，但可能是 NOT 门有一个 1 输入和一个 1 输出。

Logisim 在每个似乎与振荡有关的位置用红色圈出。如果涉及的点位于子电路内，Logisim 将用红色绘制该子电路的轮廓。

当 Logisim 检测到振荡时，它会关闭所有进一步的模拟。可以使用“模拟”菜单的“模拟已启用”选项重新启用模拟。

Logisim 使用一种相当简单的技术检测振荡：如果电路模拟似乎需要多次迭代，那么它将简单地放弃并报告振荡。（它确定涉及的点是在最后 25% 的迭代中接触到的点。）因此，它可能会错误地报告振荡，特别是如果您使用的是非常大的电路；但它将比我用 Logisim 构建的任何一个都要大。在任何情况下，如果您确信报告有误，可以通过“项目选项”窗口的“模拟”选项卡配置振荡发生之前完成的迭代次数。

缺点

Logisim 的传播算法对于几乎所有的教育目的来说都足够复杂；但对于工业电路设计来说，它还不够成熟。从最坏到最坏，Logisim 传播技术的缺点包括：

- 除了门延迟问题外，Logisim 并不特别关心时间问题。这是非常理想的，因此 S-R 触发器中的一对 NOR 门将无限锁定地切换，而不是电路最终进入稳定状态。
- Logisim 无法模拟其管脚有时作为输入，有时作为输出的子电路。不过，使用 Java 构建的组件可以有这样的管脚：在内置库中，内存库的 RAM 电路包含一个 D 管脚，它既可以作为输入，也可以作为输出。
- 假定存在振荡误差，Logisim 在固定迭代次数后停止模拟。可以想象，不振荡的大型电路可能会导致故障。
- Logisim 对于区分电压等级没有任何作用：一个位只能是开、关、未指定或错误。

- 还有一些其他的缺点，我已经忽略了，因为它们太模糊了，如果你意识到了，很明显 Logisim 离这个水平还差得很远。作为一个极端的例子，我有一个朋友在一家大型芯片制造商工作，他的工作是担心芯片纳米宽电线中的“气泡”会增长并导致随机断开。
- 除此之外，我还不是电路设计专家；因此，传播技术中很可能存在我不知道的错误。我欢迎专家的更正。

第十三节：JAR 库

使用 JAR 库

Logisim 有两种类型的电路元件：在 Logisim 中设计为元件组合的元件，以及用 Java 编写的基本元件。Logisim 电路更容易设计，但它们无法支持复杂的用户交互，而且效率相对较低。

Logisim 包含相当全面的 Java 组件内置库集合，但它也可以加载您或其他人编写的其他库。下载库后，您可以通过右键单击资源管理器窗格（顶行）中的项目并选择 Load library>JAR library...将其导入到项目中。然后，Logisim 将提示您选择 JAR 文件。（在某些情况下，您可能需要在出现提示时键入起始类名，该类名将由库开发人员提供。但是，开发人员通常会配置 JAR 库以避免这种情况（通过在 JAR 中包含一个清单文件，其中包含一个库类属性，指定主类名）。）

创建 JAR 库

本节的其余部分将介绍一系列经过详细注释的示例，说明如何自己开发 Logisim 库。只有当您是经验丰富的 Java 程序员时，才应该尝试这样做。您会发现这些示例之外的文档相当贫乏。

您可以下载一个 JAR 文件，该文件允许通过 Logisim 网站的链接部分将这些示例导入 Logisim。该 JAR 文件还包含这些示例中包含的源代码。

格雷码增量	使用一个简单的组件示例来说明任何组件类型的基本组件，该组件接受多位输入并计算其后的下一个格雷码值。
库类	说明如何定义库。这是任何 JAR 文件的入口点——用户在加载 JAR 库时输入其名称的类。
简单格雷码计数器	说明如何制作具有内部状态的组件，特别是迭代格雷码的 8 位计数器。
格雷码计数器	演示一个完整、相当复杂的组件，用户可以与之交互。它实现了一个格雷码计数器，其中记忆的位数是可定制的，用户可以通过使用 Poke Tool 点击当前值并键入值来编辑当前值。
指导方针	开发第三方库的人员的一般信息。

许可证

这个示例 JAR 库中的代码是在 MIT 许可证下发布的，这是一个比 GPL 更宽松的许可证，在 GPL 下发布其余的 Logisim。

版权所有 (c) 2009, Carl Burch。

特此免费授予获得本软件和相关文档文件（“软件”）副本的任何人无限制地处理本软件的权限，包括但不限于使用、复制、修改、合并、发布、分发、再许可和/或销售本软件副本的权利，并允许向其提供本软件的人这样做，受以下条件约束：

上述版权声明和本许可声明应包含在软件的所有副本或实质部分中。

本软件“按原样”提供，不提供任何类型的明示或暗示担保，包括但不限于适销性、特定用途适用性和非侵权担保。在任何情况下，无论是合同诉讼、侵权诉讼还是其他形式的索赔、损害赔偿或其他责任，无论是由软件或软件中的使用或其他交易引起、引起或与之相关的，作者或版权持有人概不负责。

格雷码增量

库中包含的每个组件都是通过创建 `InstanceFactory` 的子类来定义的，`InstanceFactory` 位于在 `com.cburch.logisim` 实例包。这个子类包含所有涉及的代码

(这里我们描述的是当前版本 Logisim 的 API。您可能会发现一些为较旧版本的 Logisim 开发的库，其中的组件是通过定义两个类来开发的，一个扩展 `Component`，另一个扩展 `ComponentFactory`。2.3.0 版引入了更简单的 `InstanceFactory` API；不推荐使用较旧的技术。)

三个 Logisim 包定义了与定义组件库相关的大多数类。

<code>com.cburch.logisim.instance</code>	包含与定义组件特别相关的类，包括 <code>InstanceFactory</code> 、 <code>InstanceState</code> 、 <code>InstancePainter</code> 和 <code>Instance</code> 类。
<code>com.cburch.logisim.data</code>	包含与组件关联的数据元素相关的类，例如用于表示边界矩形的 <code>Bounds</code> 类或用于表示导线上可能存在的值的 <code>Value</code> 类。
<code>com.cburch.logisim.tools</code>	包含与库定义相关的类。

关于格雷码

在继续之前，让我简要描述一下这些示例所基于的格雷码。理解这些示例的工作原理并不重要，因此如果您愿意，可以安全地跳到下面的代码，尤其是如果您已经知道 Gray 代码的话。

格雷码是一种技术（以 Frank Gray 命名），用于迭代 n 位序列，每个步骤只更改一位。例如，考虑下面列出的 4 位格雷码。

```
0000 0001 0011 0010
0110 0111 0101 0100
1100 1101 1111 1110
1010 1011 1001 1000
```

每个值都有带下划线的位，该位将更改为序列中的下一个值。例如，0000 之后是 0001，其中最后一位已切换，因此最后一位带有下划线。

Logisim 的内置组件不包含任何格雷码功能。但电子设计师发现格雷码有时很有用。格雷码的一个特别值得注意的例子是沿着卡诺地图的轴。

格雷增量

这是一个简单的例子，说明了定义组件的基本要素。这个特殊的组件是一个递增器，它接受一个多位输入，并按顺序生成下一个 Gray 代码。

```
package com.cburch.gray;

import com.cburch.logisim.data.Attribute;
import com.cburch.logisim.data.BitWidth;
import com.cburch.logisim.data.Bounds;
import com.cburch.logisim.data.Value;
import com.cburch.logisim.instance.InstanceFactory;
import com.cburch.logisim.instance.InstancePainter;
import com.cburch.logisim.instance.InstanceState;
import com.cburch.logisim.instance.Port;
import com.cburch.logisim.instance.StdAttr;

/** This component takes a multibit input and outputs the value that follows it
 * in Gray Code. For instance, given input 0100 the output is 1100. */
class GrayIncrementer extends InstanceFactory {
```

```

/* Note that there are no instance variables. There is only one instance of
 * this class created, which manages all instances of the component. Any
 * information associated with individual instances should be handled
 * through attributes. For GrayIncrementer, each instance has a "bit width"
 * that it works with, and so we'll have an attribute. */

/** The constructor configures the factory. */
GrayIncrementer() {
    super("Gray Code Incrementer");

    /* This is how we can set up the attributes for GrayIncrementers. In
     * this case, there is just one attribute - the width - whose default
     * is 4. The StdAttr class defines several commonly occurring
     * attributes, including one for "bit width." It's best to use those
     * StdAttr attributes when appropriate: A user can then select several
     * components (even from differing factories) with the same attribute
     * and modify them all at once. */
    setAttributes(new Attribute[] { StdAttr.WIDTH },
        new Object[] { BitWidth.create(4) });

    /* The "offset bounds" is the location of the bounding rectangle
     * relative to the mouse location. Here, we're choosing the component to
     * be 30x30, and we're anchoring it relative to its primary output
     * (as is typical for Logisim), which happens to be in the center of the
     * east edge. Thus, the top left corner of the bounding box is 30 pixels
     * west and 15 pixels north of the mouse location. */
    setOffsetBounds(Bounds.create(-30, -15, 30, 30));

    /* The ports are locations where wires can be connected to this
     * component. Each port object says where to find the port relative to
     * the component's anchor location, then whether the port is an
     * input/output/both, and finally the expected bit width for the port.
     * The bit width can be a constant (like 1) or an attribute (as here).
     */
    setPorts(new Port[] {
        new Port(-30, 0, Port.INPUT, StdAttr.WIDTH),
        new Port(0, 0, Port.OUTPUT, StdAttr.WIDTH),
    });
}

/** Computes the current output for this component. This method is invoked
 * any time any of the inputs change their values; it may also be invoked in
 * other circumstances, even if there is no reason to expect it to change
 * anything. */

```

```

public void propagate(InstanceState state) {
    // First we retrieve the value being fed into the input. Note that in
    // the setPorts invocation above, the component's input was included at
    // index 0 in the parameter array, so we use 0 as the parameter below.
    Value in = state.getPort(0);

    // Now compute the output. We've farmed this out to a helper method,
    // since the same logic is needed for the library's other components.
    Value out = nextGray(in);

    // Finally we propagate the output into the circuit. The first parameter
    // is 1 because in our list of ports (configured by invocation of
    // setPorts above) the output is at index 1. The second parameter is the
    // value we want to send on that port. And the last parameter is its
    // "delay" - the number of steps it will take for the output to update
    // after its input.
    state.setPort(1, out, out.getWidth() + 1);
}

/** Says how an individual instance should appear on the canvas. */
public void paintInstance(InstancePainter painter) {
    // As it happens, InstancePainter contains several convenience methods
    // for drawing, and we'll use those here. Frequently, you'd want to
    // retrieve its Graphics object (painter.getGraphics) so you can draw
    // directly onto the canvas.
    painter.drawRectangle(painter.getBounds(), "G+I");
    painter.drawPorts();
}

/** Computes the next gray value in the sequence after prev. This static
 * method just does some bit twiddling; it doesn't have much to do with
 * Logisim except that it manipulates Value and BitWidth objects. */
static Value nextGray(Value prev) {
    BitWidth bits = prev.getBitWidth();
    if(!prev.isFullyDefined()) return Value.createError(bits);
    int x = prev.toIntValue();
    int ct = (x >> 16) ^ x; // compute parity of x
    ct = (ct >> 8) ^ ct;
    ct = (ct >> 4) ^ ct;
    ct = (ct >> 2) ^ ct;
    ct = (ct >> 1) ^ ct;
    if((ct & 1) == 0) { // if parity is even, flip 1's bit
        x = x ^ 1;
    } else { // else flip bit just above last 1

```

```

        int y = x ^ (x & (x - 1)); // first compute the last 1
        y = (y - 1) & bits.getMask();
        x = (y == 0 ? 0 : x ^ y);
    }
    return Value.createKnown(bits, x);
}
}
}

```

这个例子本身不足以创建一个工作 JAR 文件；还必须提供 Library 类，如下一页所示。

库类

JAR 库的访问点是一个扩展 library 类的类。图书馆的主要工作是列出图书馆中可用的工具；通常，这些工具都是用来添加定义的各种组件的工具，即使用不同组件工厂的 AddTool 类的实例。

Components

```

package com.cburch.gray;

import java.util.Arrays;
import java.util.List;

import com.cburch.logisim.tools.AddTool;
import com.cburch.logisim.tools.Library;

/** The library of components that the user can access. */
public class Components extends Library {
    /** The list of all tools contained in this library. Technically,
     * libraries contain tools, which is a slightly more general concept
     * than components; practically speaking, though, you'll most often want
     * to create AddTools for new components that can be added into the circuit.
     */
    private List<AddTool> tools;

    /** Constructs an instance of this library. This constructor is how
     * Logisim accesses first when it opens the JAR file: It looks for
     * a no-arguments constructor method of the user-designated class.
     */
    public Components() {
        tools = Arrays.asList(new AddTool[] {
            new AddTool(new GrayIncrementer()),
            new AddTool(new SimpleGrayCounter()),
            new AddTool(new GrayCounter()),
        });
    }

    /** Returns the name of the library that the user will see. */

```

```

    public String getDisplayName() {
        return "Gray Tools";
    }

    /** Returns a list of all the tools available in this library. */
    public List<AddTool> getTools() {
        return tools;
    }
}

```

简单格雷码计数器

通常，我们希望组件在本质上不是完全组合的，也就是说，我们希望该组件具有一些内存。定义这样的组件有一个重要的微妙之处：您不能让组件本身存储状态，因为单个组件可以在同一电路中多次出现。它不能在电路中直接出现多次，但如果它出现在多次使用的子电路中，则可以出现多次。

解决方案是创建一个新类来表示对象的当前状态，并通过父电路的状态将其实例与组件相关联。在这个实现边缘触发 4 位 Gray 代码计数器的示例中，我们定义了一个 CounterData 类来表示计数器的状态，此外还定义了前面所示的 InstanceFactory 子类。CounterData 对象会记住计数器的当前值以及上次看到的时钟输入（以检测上升沿）。

计数器数据

```

package com.cburch.gray;

import com.cburch.logisim.data.BitWidth;
import com.cburch.logisim.data.Value;
import com.cburch.logisim.instance.InstanceData;
import com.cburch.logisim.instance.InstanceState;

/** Represents the state of a counter. */
class CounterData implements InstanceData, Cloneable {

    /** Retrieves the state associated with this counter in the circuit state,
     * generating the state if necessary.
     */

    public static CounterData get(InstanceState state, BitWidth width) {
        CounterData ret = (CounterData) state.getData();
        if(ret == null) {
            // If it doesn't yet exist, then we'll set it up with our default
            // values and put it into the circuit state so it can be retrieved
            // in future propagations.
            ret = new CounterData(null, Value.createKnown(width, 0));
            state.setData(ret);
        } else if(!ret.value.getBitWidth().equals(width)) {
            ret.value = ret.value.extendWidth(width.getWidth(), Value.FALSE);
        }
        return ret;
    }
}

```

```

/** The last clock input value observed. */
private Value lastClock;

/** The current value emitted by the counter. */
private Value value;

/** Constructs a state with the given values. */
public CounterData(Value lastClock, Value value) {
    this.lastClock = lastClock;
    this.value = value;
}

/** Returns a copy of this object. */
public Object clone() {
    // We can just use what super.clone() returns: The only instance variables are
    // Value objects, which are immutable, so we don't care that both the copy
    // and the copied refer to the same Value objects. If we had mutable instance
    // variables, then of course we would need to clone them.
    try { return super.clone(); }
    catch(CloneNotSupportedException e) { return null; }
}

/** Updates the last clock observed, returning true if triggered. */
public boolean updateClock(Value value) {
    Value old = lastClock;
    lastClock = value;
    return old == Value.FALSE && value == Value.TRUE;
}

/** Returns the current value emitted by the counter. */
public Value getValue() {
    return value;
}

/** Updates the current value emitted by the counter. */
public void setValue(Value value) {
    this.value = value;
}
}

SimpleCounter
package com.cburch.gray;

import com.cburch.logisim.data.BitWidth;

```



```

import com.cburch.logisim.data.Bounds;
import com.cburch.logisim.data.Direction;
import com.cburch.logisim.instance.InstanceFactory;
import com.cburch.logisim.instance.InstancePainter;
import com.cburch.logisim.instance.InstanceState;
import com.cburch.logisim.instance.Port;
import com.cburch.logisim.util.GraphicsUtil;
import com.cburch.logisim.util.StringUtil;

/** Manufactures a simple counter that iterates over the 4-bit Gray Code. This
 * example illustrates how a component can maintain its own internal state. All
 * of the code relevant to state, though, appears in CounterData class. */
class SimpleGrayCounter extends InstanceFactory {
    private static final BitWidth BIT_WIDTH = BitWidth.create(4);

    // Again, notice how we don't have any instance variables related to an
    // individual instance's state. We can't put that here, because only one
    // SimpleGrayCounter object is ever created, and its job is to manage all
    // instances that appear in any circuits.

    public SimpleGrayCounter() {
        super("Gray Counter (Simple)");
        setOffsetBounds(Bounds.create(-30, -15, 30, 30));
        setPorts(new Port[] {
            new Port(-30, 0, Port.INPUT, 1),
            new Port( 0, 0, Port.OUTPUT, BIT_WIDTH.getWidth()),
        });
    }

    public void propagate(InstanceState state) {
        // Here I retrieve the state associated with this component via a helper
        // method. In this case, the state is in a CounterData object, which is
        // also where the helper method is defined. This helper method will end
        // up creating a CounterData object if one doesn't already exist.
        CounterData cur = CounterData.get(state, BIT_WIDTH);

        boolean trigger = cur.updateClock(state.getPort(0));
        if(trigger) cur.setValue(GrayIncrementer.nextGray(cur.getValue()));
        state.setPort(1, cur.getValue(), 9);

        // (You might be tempted to determine the counter's current value
        // via state.getPort(1). This is erroneous, though, because another
        // component may be pushing a value onto the same point, which would
        // "corrupt" the value found there. We really do need to store the

```

```

        // current value in the instance.)
    }

    public void paintInstance(InstancePainter painter) {
        painter.drawBounds();
        painter.drawClock(0, Direction.EAST); // draw a triangle on port 0
        painter.drawPort(1); // draw port 1 as just a dot

        // Display the current counter value centered within the rectangle.
        // However, if the context says not to show state (as when generating
        // printer output), then skip this.
        if(painter.getShowState()) {
            CounterData state = CounterData.get(painter, BIT_WIDTH);
            Bounds bds = painter.getBounds();
            GraphicsUtil.drawCenteredText(painter.getGraphics(),
                StringUtil.toHexString(BIT_WIDTH.getWidth(), state.getValue().toIntValue()),
                bds.getX() + bds.getWidth() / 2,
                bds.getY() + bds.getHeight() / 2);
        }
    }
}

```

格雷码计数器

Logisim 库的这种定位以一个相当复杂的 Gray 代码计数器结束，该计数器允许用户使用 Poke Tool 更改其当前值，并使用 Text Tool 在组件上放置标签。它还自定义与工具关联的资源管理器中显示的图标。

格雷计数器

```

package com.cburch.gray;

import java.net.URL;

import javax.swing.ImageIcon;
import com.cburch.logisim.data.Attribute;
import com.cburch.logisim.data.BitWidth;
import com.cburch.logisim.data.Bounds;
import com.cburch.logisim.data.Direction;
import com.cburch.logisim.instance.Instance;
import com.cburch.logisim.instance.InstanceFactory;
import com.cburch.logisim.instance.InstancePainter;
import com.cburch.logisim.instance.InstanceState;
import com.cburch.logisim.instance.Port;
import com.cburch.logisim.instance.StdAttr;
import com.cburch.logisim.util.GraphicsUtil;
import com.cburch.logisim.util.StringUtil;

```

```

/** Manufactures a counter that iterates over Gray codes. This demonstrates
 * several additional features beyond the SimpleGrayCounter class. */
class GrayCounter extends InstanceFactory {
    public GrayCounter() {
        super("Gray Counter");
        setOffsetBounds(Bounds.create(-30, -15, 30, 30));
        setPorts(new Port[] {
            new Port(-30, 0, Port.INPUT, 1),
            new Port( 0, 0, Port.OUTPUT, StdAttr.WIDTH),
        });

        // We'll have width, label, and label font attributes. The latter two
        // attributes allow us to associate a label with the component (though
        // we'll also need configureNewInstance to configure the label's
        // location).
        setAttributes(
            new Attribute[] { StdAttr.WIDTH, StdAttr.LABEL, StdAttr.LABEL_FONT },
            new Object[] { BitWidth.create(4), "", StdAttr.DEFAULT_LABEL_FONT });

        // The following method invocation sets things up so that the instance's
        // state can be manipulated using the Poke Tool.
        setInstancePoker(CounterPoker.class);

        // These next two lines set it up so that the explorer window shows a
        // customized icon representing the component type. This should be a
        // 16x16 image.
        URL url = getClass().getClassLoader().getResource("com/cburch/gray/counter.gif");
        if(url != null) setIcon(new ImageIcon(url));
    }

    /** The configureNewInstance method is invoked every time a new instance
     * is created. In the superclass, the method doesn't do anything, since
     * the new instance is pretty thoroughly configured already by default. But
     * sometimes you need to do something particular to each instance, so you
     * would override the method. In this case, we need to set up the location
     * for its label. */
    protected void configureNewInstance(Instance instance) {
        Bounds bds = instance.getBounds();
        instance.setTextField(StdAttr.LABEL, StdAttr.LABEL_FONT,
            bds.getX() + bds.getWidth() / 2, bds.getY() - 3,
            GraphicsUtil.H_CENTER, GraphicsUtil.V_BASELINE);
    }

    public void propagate(InstanceState state) {

```

```

        // This is the same as with SimpleGrayCounter, except that we use the
        // StdAttr.WIDTH attribute to determine the bit width to work with.
        BitWidth width = state.getAttributeValue(StdAttr.WIDTH);
        CounterData cur = CounterData.get(state, width);
        boolean trigger = cur.updateClock(state.getPort(0));
        if(trigger) cur.setValue(GrayIncrementer.nextGray(cur.getValue()));
        state.setPort(1, cur.getValue(), 9);
    }

    public void paintInstance(InstancePainter painter) {
        // This is essentially the same as with SimpleGrayCounter, except for
        // the invocation of painter.drawLabel to make the label be drawn.
        painter.drawBounds();
        painter.drawClock(0, Direction.EAST);
        painter.drawPort(1);
        painter.drawLabel();

        if(painter.getShowState()) {
            BitWidth width = painter.getAttributeValue(StdAttr.WIDTH);
            CounterData state = CounterData.get(painter, width);
            Bounds bds = painter.getBounds();
            GraphicsUtil.drawCenteredText(painter.getGraphics(),
                StringUtil.toHexString(width.getWidth(), state.getValue().toIntValue()),
                bds.getX() + bds.getWidth() / 2,
                bds.getY() + bds.getHeight() / 2);
        }
    }
}

CounterPoker
package com.cburch.gray;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.MouseEvent;

import com.cburch.logisim.data.BitWidth;
import com.cburch.logisim.data.Bounds;
import com.cburch.logisim.data.Value;
import com.cburch.logisim.instance.InstancePainter;
import com.cburch.logisim.instance.InstancePoker;
import com.cburch.logisim.instance.InstanceState;
import com.cburch.logisim.instance.StdAttr;

```

```

/** When the user clicks a counter using the Poke Tool, a CounterPoker object
 * is created, and that object will handle all user events. Note that
 * CounterPoker is a class specific to GrayCounter, and that it must be a
 * subclass of InstancePoker in the com.cburch.logisim.instance package. */
public class CounterPoker extends InstancePoker {
    public CounterPoker() { }

    /** Determines whether the location the mouse was pressed should result
     * in initiating a poke.
     */
    public boolean init(InstanceState state, MouseEvent e) {
        return state.getInstance().getBounds().contains(e.getX(), e.getY());
        // Anywhere in the main rectangle initiates the poke. The user might
        // have clicked within a label, but that will be outside the bounds.
    }

    /** Draws an indicator that the caret is being selected. Here, we'll draw
     * a red rectangle around the value. */
    public void paint(InstancePainter painter) {
        Bounds bds = painter.getBounds();
        BitWidth width = painter.getAttributeValue(StdAttr.WIDTH);
        int len = (width.getWidth() + 3) / 4;

        Graphics g = painter.getGraphics();
        g.setColor(Color.RED);
        int wid = 7 * len + 2; // width of caret rectangle
        int ht = 16; // height of caret rectangle
        g.drawRect(bds.getX() + (bds.getWidth() - wid) / 2,
            bds.getY() + (bds.getHeight() - ht) / 2, wid, ht);
        g.setColor(Color.BLACK);
    }

    /** Processes a key by just adding it onto the end of the current value. */
    public void keyTyped(InstanceState state, KeyEvent e) {
        // convert it to a hex digit; if it isn't a hex digit, abort.
        int val = Character.digit(e.getKeyChar(), 16);
        BitWidth width = state.getAttributeValue(StdAttr.WIDTH);
        if((val == 0 || (val & width.getMask()) != val) return;

        // compute the next value
        CounterData cur = CounterData.get(state, width);
        int newVal = (cur.getValue().toIntValue() * 16 + val) & width.getMask();
        Value newValue = Value.createKnown(width, newVal);
        cur.setValue(newValue);
    }
}

```

```

state.fireInvalidated();

// You might be tempted to propagate the value immediately here, using
// state.setPort. However, the circuit may currently be propagating in
// another thread, and invoking setPort directly could interfere with
// that. Using fireInvalidated notifies the propagation thread to
// invoke propagate on the counter at its next opportunity.
}
}

```

指导方针

了解更多信息

除了这里提供的示例序列之外，Logisim 源代码还提供了大量其他示例，尽管它们并不总是说明对可读性和良好设计的同样关注。

为了实现对未来版本的最大可移植性，您应该尽可能地坚持使用…实例、…数据和…工具包中的类。当然，您可以使用其他软件包的 API，但它们更容易受到未来版本 Logisim 更改的影响。

我通常愿意回答偶尔的帮助请求。当然，bug 报告和改进建议总是受欢迎的。

分配

您可以自由分发您开发的任何 JAR，不受任何限制。然而，如果您的部分工作是从 Logisim 源代码（根据 GPL 发布）的部分派生而来的，则 GPL 限制确实适用。从用户指南本节中的示例代码派生不会产生此类限制；这些示例是根据 MIT 许可证发布的。

如果您想与其他 Logisim 用户共享您的库，我很乐意通过 Logisim 网站提供指向托管网页或 JAR 文件本身的链接。如果您认为您的库应该构建到基本的 Logisim 版本中，那么我欢迎您的建议，并且我很高兴感谢您在 Logisim 发布中的贡献，包括工作。

第十四节：关于 Logisim

Logisim 是开源软件。源代码包含在分布式 JAR 文件的 src 子目录中。

如果您觉得 Logisim 有用，请告诉我。如果你是一家教育机构，尤其要这样做；这些信息将帮助我获得对这项工作的支持。

我欢迎关于 Logisim 的电子邮件，包括错误报告、建议和修复。当您给我发电子邮件时，请记住我一直在努力制作 Logisim，但没有收到您的任何付款。如果你想有权投诉软件，那么我建议你拿出钱，为 Logisim 的一个竞争项目买单。（据我所知，没有任何开源竞争对手可以接近 Logisim 的功能集。）尽管如此，我仍对继续改进 Logisim 感兴趣，欢迎您的建议。

版权声明

版权所有 (c) 2005, Carl Burch。

Logisim 是自由软件；您可以根据自由软件基金会发布的 GNU 通用公共许可条款重新发布和/或修改它；许可证的版本 2 或（根据您的选择）任何更高版本。

Logisim 的发行是为了希望它有用，但没有任何保证；甚至没有对适销性或特定用途适用性的默示保证。有关更多详细信息，请参阅 GNU 通用公共许可证。

致谢

Logisim 的源代码主要是我自己的工作；我必须感谢资助我教授工作的雇主，包括这个项目：我于 2000-

2004 年在美国明尼苏达州圣约翰大学 (Collegeville, Minnesota, USA) 开始了这个项目, 从 2004 年到现在, 我一直在美国阿肯色州康威的亨德里克斯学院 (Hendrix College) 继续学习。我非常感谢这些大学给我时间和资源来完成这个项目。要是所有的学院和大学都能像这些学院一样共同行动, 关心优秀的教学就好了!

其他一些特别乐于助人的人:

- Thelido Cruz Franqueira、Thanos Kakaruntas、Ilia Lilov、Pablo Leal Ramos 和 Uwe Zimmermann, 他们为 Logisim 打包的翻译做出了贡献。有关翻译的更多信息, 请访问“国际首选项”页面。
- 加州大学伯克利分校 2005 年春季 CS61C 班, 该班经历了 Logisim 2.0 的测试版。这些学生忍受了许多错误, 我非常感谢他们的耐心和建议!
- 2001 年春季, 圣本笃学院和圣约翰大学的 CSCI 150 课程, 在开发过程中使用了最基本的 Logisim 版本。

几个 Logisim 来自 Logisim 使用的其他软件包; 其中的几个部分是作为 Logisim 的一部分分发的。

Sun 的 Java API (显然)

Sun 的 JavaHelp 项目

从“帮助”菜单提供集成的帮助系统。

MJAdapter, 来自 Steve Roy

与 Macintosh OS X 平台集成。

launch4j, 来自 Grzegorz Kowalt

允许将 Logisim 作为 Windows 可执行文件分发。

GIFEncoder, 来自 Adam Doppelt

将图像保存为 GIF 文件。这本身基于 Sverre H.Huseby 编写的 C 代码。

Jeremy Wood 的 ColorPicker

提供配置颜色时弹出的颜色对话框 (与 LED 组件一样)。

JFontChooser, 来自 Christos Bohoris

提供选择字体属性时弹出的字体选择对话框 (例如, 许多组件的“标签字体”属性)。

TableSorter, 归于 Philip Milne、Brendon McLean、Dan van Enkevort、Parwinder Sekhon 和 ouroborus@ouroborus.org

通过单击列标题, 可以在“获取电路统计信息”对话框中对表格进行排序。

农场新鲜网页图标, <http://www.fatcow.com/free-icons>

提供用于控制仿真树下显示的仿真的图标。这些图标是根据 Creative Commons Attribution 3.0 许可证发布的, 并且不能根据 GPL 条款重新发布。

最后, 我想感谢所有与我联系过的用户, 无论是提供错误报告、建议, 还是让我知道他们在课堂上使用 Logisim。我不得不让这些建议者匿名, 因为我没有他们的许可在这里提及它们, 但是: 谢谢!

通用公共许可证

第 2 版, 1991 年 6 月

版权所有 (C) 1989, 1991 自由软件基金会, 股份有限公司。

美国马萨诸塞州波士顿市富兰克林街 51 号 5 楼, 邮编: 02110-1301

每个人都可以复制和分发原稿

此许可证文档, 但不允许更改。

序言

大多数软件的许可证都是为了剥夺您的分享和改变的自由。相比之下, GNU 公众许可旨在保证您可以自由共享和更改软

件——确保软件对所有用户都是免费的。这通用公共许可证适用于大多数自由软件基金会的软件以及作者承诺的任何其他程序使用它。(其他一些自由软件基金会软件包含在改为 GNU 库通用公共许可证。)您可以将其应用于你的程序也一样。当我们谈到自由软件时,我们指的是自由,而不是价格我们的通用公共许可证旨在确保您有分发自由软件副本的自由(并收取此服务(如果您愿意),您可以接收或获取源代码如果需要,可以更改软件或使用其中的部分。在新的免费程序中:你知道你可以做这些事情。为了保护你的权利,我们需要做出禁止的限制任何人拒绝你这些权利或要求你放弃这些权利。如果您分发软件的副本,或者如果您修改了它。例如,如果您分发此类程序的副本您必须向收件人授予以下所有权利:你有。你必须确保他们也收到或能够得到源代码。%1 你必须向他们展示这些术语,让他们知道权利。我们通过两个步骤保护您的权利:(1)对软件进行版权保护,以及(2)向您提供此许可证,允许您合法复制,分发和/或修改软件。此外,为了保护每一位作者和我们,我们要确保每个人都知道这是免费的软件如果软件被其他人修改并传递,我们希望收件人知道他们所拥有的不是原件,所以别人提出的任何问题都不会影响到原来的作者的声誉。最后,任何自由程序都会不断受到软件的威胁专利。我们希望避免风险该计划将单独获得专利许可,实际上使程序专有。为了防止这种情况发生,我们明确表示专利必须授权给每个人自由使用或根本不授权。复制、分发和随后进行修改。

GNU 通用公共许可证

复制、分发和修改的条款和条件

0. 本许可证适用于包含以下内容的任何程序或其他作品版权所有人发布的通知,表示可以分发根据本通用公共许可条款。下面的“程序”,指任何此类计划或作品,以及“基于计划的作品”是指本程序或版权法规定的任何衍生作品:也就是说,包含程序或其一部分的作品,逐字记录或修改和/或翻译成另一个语言(以下,翻译包括但不限于术语“修改”)。每个持牌人都称呼为“您”。除复制、分发和修改之外的活动本许可证涵盖的范围;它们超出了其范围。行为运行程序不受限制,程序的输出仅当其内容构成基于程序(独立于通过运行程序生成的程序)。这是否属实取决于该计划所做的工作。

1. 您可以复制和分发计划的逐字副本源代码,只要您在每个副本上醒目且适当地发布适当的版权声明和免责声明;保持所有提及本许可证和无任何保证的通知;并向该计划的任何其他接收者提供本许可证的副本与计划一起。您可以就转让副本的实际行为收取费用,并且您可以选择提供保修保护以换取费用。

2. 您可以修改您的一份或多份计划或任何部分从而形成一部基于该程序的作品,并复制和根据第 1 节的条款分发此类修改或工作前提是也满足所有这些条件:a) 您必须使修改后的文件带有明显的通知说明您更改了文件和任何更改的日期。b) 您必须在全部或部分包含或衍生自程序或任何部分,作为整体获得许可,第三方不收取任何费用本许可条款下的各方。c) 如果修改后的程序通常以交互方式读取命令当你跑步的时候,你必须引起它,当你开始跑步的时候以最普通的方式进行交互使用,以打印或显示公告,包括适当的版权声明和请注意,没有担保(或者说,您提供保证),用户可以根据这些条件,并告诉用户如何查看此的副本许可证(例外:如果程序本身是交互式的,但通常不会打印这样的公告,您的工作基于程序不需要打印公告。)这些要求适用于整个修改工程。如果该作品的可识别部分并非源自该计划,可以合理地认为是独立和独立的工程则本许可证及其条款不适用于当您将它们作为单独的作品分发时。但当你以作品为基础,将相同的部分作为整体的一部分进行分发给计划中,整体分配必须符合本许可证,其对他被许可方的权限扩展至整个,因此,无论是谁写的,都是每个部分。因此,本节的目的不是要求权利或抗辩您完全由自己撰写的工作权利;相反,其目的是行使控制衍生品分销的权利,或基于计划的集体工作。此外,仅仅是基于该计划的其他作品的聚合在一卷存储或分发介质不会影响其他工作本许可证的范围。

3. 您可以复制和分发程序(或基于程序的作品,根据第 2 节)的条款,以目标代码或可执行形式上述第 1 节和第 2 节规定,您还应执行以下操作之一:a) 随附完整的相应机器可读文件源代码,必须根据章节条款分发通常用于软件交换的介质上的上述 1 和 2;或 b) 随附书面报价,有效期至少为三年,给任何第三方,费用不超过实际执行源分发的成本,完整相应源代码的机器可读副本根据上述第 1 节和第 2 节的条款在介质上分发通常用于软件交换;或 c) 随附您收到的有关报价的信息分发相应的源代码。(该替代方案是仅允许非商业分销,且仅当您以目标代码或可执行形式接收程序符合上述 b 小节的要约。)作品的源代码是指作品的首选形式对其进行修改。对于可执行的工作,请填写源代码代码是指它包含的所有模块的所有源代码,以及任何关联的接口定义文件,以及用于控制可执行文件的编译和安装。然而,作为一个特殊例外,分发的源代码不需要包括正常分布的任何内容(源或二进制表单)的主要组件(编译器、内核等)运行可执行文件的操作系统,除非该组件它本身伴随着可执行文件。如果可执行代码或目标代码的分发是通过提供从指定地

点获取副本，然后提供同等内容从同一位置复制源代码的权限视为分发源代码，即使第三方不是被迫复制源代码和目标代码。

4. 您不得复制、修改、再授权或分发本程序除非本许可证明确规定。任何尝试否则，复制、修改、再许可或分发程序是无效，并将自动终止您在本许可证下的权利。但是，根据本许可证不会终止其许可证各方仍然完全遵守。

5. 您不需要接受本许可证，因为您没有签名了。但是，没有任何其他内容授予您修改或分发程序或其衍生作品。这些操作是如果您不接受本许可证，将被法律禁止。因此，通过修改或分发程序（或基于程序），您表示接受本许可证，并且复制、分发或修改的所有条款和条件程序或基于程序工作。

6. 每次您重新发布程序（或任何基于程序），则收件人会自动从原始许可方复制、分发或修改程序这些条款和条件。您不能再强加对收件人行使本协议授予的权利的限制。您不负责强制第三方遵守本许可证。

7. 如果，由于法院判决或专利指控侵权或任何其他原因（不限于专利问题），对您施加的条件（无论是法院命令、协议还是否则，他们不会请原谅您不遵守本许可证的条件。如果你不能分配以同时满足您在本协议项下的义务许可和任何其他相关义务可能根本不会分发该程序。例如，如果一项专利许可证不允许通过以下方式免费重新分配程序所有直接或间接通过您接收副本的人，那么您能够同时满足它和本许可证的唯一方法是完全避免分发程序。如果本节的任何部分根据在任何特殊情况下，本节的余额旨在适用，而整个章节意在适用于其他情况。本节的目的是不是诱导您违反专利或其他产权主张或对任何此类索赔；本节的唯一目的是保护自由软件分发系统的完整性由公共许可实践实施。许多人已经为广泛的分布式软件做出慷慨贡献依靠该系统的一致应用系统由作者/捐赠者决定他/她是否愿意通过任何其他系统分发软件，而被许可方不能强加这种选择。本节旨在彻底阐明是本许可证其余部分的后果。

8. 如果程序的分发和/或使用受到限制某些国家通过专利或受版权保护的接口将程序置于本许可之下的原始版权所有者的可以添加明确的地理分布限制，不包括这些国家，因此只允许在这些国家或地区之间进行分销因此没有被排除在外的国家。在这种情况下，本许可证包括该限制如同写在本许可证正文中一样。

9. 自由软件基金会可发布修订版和/或新版本通用公共许可证。这样的新版本将在精神上与当前版本相似，但在细节上可能有所不同解决新的问题或担忧。每个版本都有一个不同的版本号。如果程序指定适用于本许可证的版本号以及“任何更新版本”，您可以选择遵循条款和条件该版本或 Free 发布的任何后续版本软件基础。如果程序未指定版本号本许可证，您可以选择自由软件发布的任何版本地基

10. 如果您希望将本计划的部分内容纳入其他免费内容分发条件不同的程序，请写信给作者请求许可。对于受免费软件版权保护的软件基金会，写信给自由软件基金会；我们有时对此要破例处理。我们的决定将以这两个目标为指导保持我们自由软件的所有衍生产品的自由状态普遍促进软件的共享和重用。无担保

11. 由于该程序是免费许可的，因此没有任何担保对于该计划，在适用法律允许的范围内。除非何时版权持有人和/或其他方另有书面说明“按原样”提供程序，无任何明示的担保或暗示，包括但不限于特定用途的适销性和适用性。整个风险课程的质量和性能与您息息相关。应该程序证明有缺陷，您承担所有必要维修的成本，修理或纠正。

12. 在任何情况下，除非适用法律要求或书面同意任何版权持有人或任何其他世界卫生组织可以修改和/或按照上述许可重新分发程序，对您造成的损失负责，包括任何一般、特殊、偶然或后果性损害不使用或无法使用程序（包括但不限于数据丢失或数据不准确或持续丢失您或第三方或程序未能与任何其他方一起运行计划），即使该持有人或其他方已被告知此类损害的可能性。

条款和条件结束