

完成的工作

完成了 MuxCondPropagator

MuxCondPropagator 作用

该 Pass 的目标是识别 Chisel 电路中所有的 Mux 条件信号，并将它们沿着模块实例化层级向上传播。对于每个包含或使用了 Mux 条件的模块实例，会在其接口上添加一个新的 Bundle 类型的输出端口（默认为 `_mux_cond`）。这个 Bundle 包含了所有源自该模块内部或其子模块实例的 Mux 条件信号。

将 Mux 条件传播至顶层 Module 主要在于方便后续的仿真验证与分析。在仿真运行时，只需观测顶层的 `_mux_cond` 端口上的信号值（例如，记录它们是否都达到过 0 和 1 状态），就能有效地追踪设计中对应 Mux 的真/假分支是否都曾被执行。

这为统计分支覆盖率提供了清晰、直接的数据来源。

MuxCondPropagator 转换思路详解

一、transform 方法 (主入口)

`transform` 方法是 Mux 条件传播 Pass 的主入口点，负责协调整个电路的转换过程。

1. 准备阶段:

- **(1) 获取内部条件:** 调用 `ModuleInternalCondsMapProvider.getMap(circuit)` 获取或计算每个模块内部直接使用的（非字面量）Mux 条件表达式的映射 `internalCondMap` (类型 `Map[String, Seq[Expression]]`)。这是后续分析的基础。
- **(2) 构建模块映射:** 创建一个从模块名称到模块定义 (`DefModule`) 的映射 `moduleMap`，方便快速查找模块定义。
- **(3) 初始化缓存:** 创建一个可变映射 `processedModules` (类型 `mutable.Map[(String, Seq[String]), ModuleTransformResult]`)，用于缓存已经处理过的模块（以 **(模块名, 实例路径)** 为键）及其转换结果。这对于处理模块实例化和避免重复计算至关重要。
- **(4) 计算顶层名称:** 遍历电路中的所有模块 (`Module`) 和类 (`DefClass`) 定义，对每个定义调用 `collectTopLevelNames` 辅助函数，收集其顶层作用域（端口、Wire、Reg、Node、Instance 等）定义的名称集合。将结果存储在 `topLevelNamesMap` (类型 `mutable.Map[String, Set[String]]`) 中。这用于后续判断本地条件是否需要“提升” (hoist)。

2. 递归处理:

- 定义一个内部递归函数 `processModule(moduleName: String, instancePath: Seq[String]): ModuleTransformResult`。此函数负责处理单个模块（或类）的转换。
- **调用入口:** 从电路的主模块 (`circuit.main`) 开始，以指定的顶层实例名 `topInstName` 作为初始实例路径 (`Seq(topInstName)`)，调用 `processModule(circuit.main,`

`Seq(topInstName))` 启动递归处理。

3. `processModule` 函数内部逻辑:

- **缓存检查:** 使用 `(moduleName, instancePath)` 作为键检查 `processedModules` 缓存。如果命中, 直接返回缓存的 `ModuleTransformResult`。
- **查找模块定义:** 使用 `moduleName` 在 `moduleMap` 中查找对应的 `DefModule`。
- **处理不同模块类型:**
 - **Module:** 调用核心转换逻辑 `transformModule(module, moduleMap, internalCondMap, topLevelNamesMap(moduleName), instancePath, processModule)`。
 - **DefClass:** 将 `DefClass` 临时包装成一个 `Module` 对象, 然后调用 `transformModule` 进行处理。处理完成后, 将结果中的端口和主体放回一个新的 `DefClass` 定义中。
 - **ExtModule / IntModule:** 这些模块没有内部逻辑, 无需转换。直接创建 `ModuleTransformResult` 并返回原始模块定义和 `None` 端口信息。
 - **其他/未找到:** 抛出内部错误。
- **缓存结果:** 将得到的 `ModuleTransformResult` 存入 `processedModules` 缓存 (使用 `(moduleName, instancePath)` 作为键)。
- **返回结果:** 返回 `ModuleTransformResult`。

4. 结果整合与返回:

- 递归处理完成后, `processedModules` 缓存中包含了所有被访问到的模块 (在特定实例路径下) 的转换结果。
- 创建一个最终的模块映射 `finalModules` (类型 `mutable.Map[String, DefModule]`)。
- 遍历 `processedModules` 的值 (即 `ModuleTransformResult`), 将其中的 `transformedModule` 按模块名添加到 `finalModules` 中。
- 遍历原始电路 `circuit.modules`, 将任何未出现在 `finalModules` 中的模块 (例如未被实例化的模块或 `ExtModule/IntModule`) 添加到 `finalModules` 中, 以确保所有模块都被包含。
- 创建一个新的 `Circuit` 对象, 其 `modules` 列表包含 `finalModules` 中的所有模块定义, 并按名称排序以保证确定性。
- 返回这个新的 `Circuit` 对象。

二、`transformModule` 方法 (核心转换逻辑)

`transformModule` 方法负责对单个模块 (或模拟的类) 执行具体的 Mux 条件传播转换。

1. 收集本地条件 (`LocalCondSource`):

- 从传入的 `internalCondMap` 中获取当前模块 `module.name` 对应的 Mux 条件表达式列表。
- 遍历这些表达式:
 - 使用 `distinctBy(_.serialize)` 去重。
 - 规范化条件类型为 1 位宽 `GroundType` (通常是 `UInt<1>`)。
 - 使用 `getConditionBaseName(expr)` 获取表达式的基础名称。

- 结合 `currentInstancePath` 和 `LocalMarker` 生成全局唯一的字段名 (`fieldName`), 格式类似 `path__I__local__I__basename`。
- 创建 `LocalCondSource(fieldName, ConditionOriginInfo(expr, condType))` 并收集起来。

2. 收集子模块条件 (`ChildCondSource`):

- 定义内部递归函数 `findInstancesAndCollectChildConds` 来遍历模块主体语句 `module.body`。
- 当遇到 `DefInstance inst` 时:
 - 构建子模块的实例路径 `childInstancePath = currentInstancePath :+ inst.name`。
 - 递归调用 `processFunc(inst.module, childInstancePath)` (即外部传入的 `processModule`) 来处理子模块。
 - 获取子模块的转换结果 `childResult`。
 - 如果 `childResult.conditionPortInfo` 为 `Some((childPortName, childBundleType))`:
 - 遍历 `childBundleType` 的每个字段 `field`。
 - `field.name` 已经是子模块传递上来的全局唯一字段名。
 - 创建访问该子模块条件端口字段的表达式 `fieldAccess = SubField(Reference(inst.name, ...), childPortName, ...).field`。
 - 创建 `ChildCondSource`, 其中 `fieldName` 直接使用 `field.name`, `originInfo` 包含 `fieldAccess` 表达式。将 `ChildCondSource` 收集起来。
- 对 `Block`, `Conditionally`, `LayerBlock` 等结构递归调用 `findInstancesAndCollectChildConds`。

3. 整合条件并定义端口:

- 合并本地条件和子模块条件列表, 得到 `allCondSources`。
- 如果 `allCondSources` 为空, 说明此模块无需修改, 直接返回原始模块和 `None` 端口信息。
- 遍历 `allCondSources`, 为每个条件创建一个 `Field` 定义:
 - 字段名为 `source.fieldName` (已保证唯一)。
 - 字段类型规范化为 1 位宽 `GroundType`。
 - 方向为 `Default`。
- 将生成的 `Field` 列表按名称去重 (理论上已唯一) 并排序, 得到 `finalUniqueFields`。
- 使用 `finalUniqueFields` 创建最终的 `BundleType (combinedBundleType)`。
- 创建新的输出端口 `newOutputPort`: 名称为 `ConditionOutputPortName (_mux_cond)`, 方向为 `Output`, 类型为 `combinedBundleType`。
- 更新模块的端口列表 `updatedPorts`, 移除旧的同名端口 (若有) 并添加 `newOutputPort`。

4. 处理本地条件的提升 (Hoisting):

- 遍历 `localCondSources` (本地条件)。

- 对每个本地条件的 `sourceExpr` 调用 `getRootReferenceName` 获取其根引用名 `refName` (如果可能)。
- **判断是否需要提升:** 如果 `refName` 存在且 不在当前模块的顶层名称集合 `topLevelNames` 中, 则认为该条件源自局部定义 (如内部 `Node`) , 需要提升。
- **记录提升信息:** 如果需要提升:
 - 生成一个唯一的中间 Wire 名称 `wireName` (使用 `IntermediateWirePrefix` 和 字段名)。
 - 将 `(expr, tpe, wireName)` 存入 `localCondsToHoist` (以 `refName` 为键)。
 - 将 `refName` 到 `fieldName` 的映射存入 `localCondFieldNameMap`。
- **处理非引用条件:** 如果 `sourceExpr` 不是基于简单引用 (例如 `DoPrim`) , 尝试用 `getConditionBaseName` 获取基础名, 并同样检查是否在 `topLevelNames` 中来判断是否需要提升。

5. 生成中间结构:

- **(Wire 定义):** 根据 `localCondsToHoist` 中的信息, 为每个需要提升的条件生成 `DefWire(NoInfo, wireName, tpe)` 语句, 收集为 `intermediateWires`。
- **(默认连接):** 为避免 FIRRTL 的 "not fully initialized" 问题, 为 `intermediateWires` 中的每个 Wire 生成一条默认连接 `Connect(NoInfo, Reference(wireName, tpe), UIntLiteral(0, IntWidth(1)))`, 收集为 `defaultConnects`。

6. 修改主体以添加中间连接:

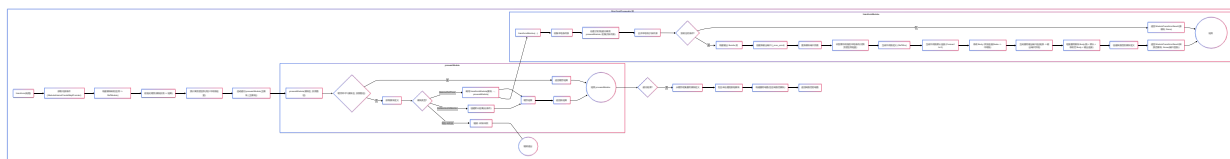
- 定义内部递归函数 `addIntermediateConnects(stmt: Statement): Seq[Statement]` 来遍历原模块主体语句 `module.body`。
- 当遇到 `DefNode(info, name, value)` 时:
 - 检查 `name` 是否在 `localCondsToHoist` 中且尚未添加连接 (通过 `connectAdded` 集合判断) 。
 - 如果是, 获取对应的 `wireName` 和 `tpe`。
 - 创建连接语句 `connectStmt = Connect(info, Reference(wireName, tpe), Reference(name, tpe))`。
 - 将 `name` 加入 `connectAdded`。
 - 返回 `Seq(node, connectStmt)`。
 - 否则, 只返回 `Seq(node)`。
- 对 `Block`, `Conditionally`, `LayerBlock` 等结构递归调用 `addIntermediateConnects`, 并处理返回的 `Seq[Statement]` (可能需要重新包装成 `Block`) 。
- 对原模块主体 `existingStmts` 应用 `addIntermediateConnects`, 得到包含中间连接的新主体语句列表 `bodyWithIntermediateConnects`。

7. 创建到输出端口的最终连接:

- 遍历最终端口字段列表 `finalUniqueFields`。
- 对每个 `field`:
 - 创建访问该端口字段的表达式 `portFieldAccess = SubField(Reference(newOutputPort.name, ...), field.name, ...)` (连接的 LHS)。
 - 在 `allCondSources` 中找到与 `field.name` 对应的原始 `source`。
 - **确定连接源 (RHS):**

- ## 8. 构建最终模块:

- ## MuxCondPropagator 转换流程图



- 进一步完成 Conditionally 的分支条件向上传播
- 尝试使用 Verilator 对 MuxCondPropagator Pass 后的电路进行 Mux 分支覆盖率分析