基于 Mockturtle 框架的 AIG 电路 balance 和 rewrite 操作优化

项目作者：祝彦翔　课程：数字集成电路设计自动化

一、项目介绍

本项目是《数字集成电路设计自动化》课程的实践项目。在数字电路设计流程中，逻辑综合与优化是至关重要的一环，它直接影响到芯片的最终性能、面积和功耗。AIG 作为一种高效、标准的逻辑表示形式，是许多现代 EDA 工具进行逻辑优化的核心。

本次项目中，我的主要任务是基于开源的逻辑综合框架 Mockturtle，针对 AIG 数据结构，亲手实现并验证两种业界经典的优化算法：balance（平衡）和 rewrite（重写）。这两种算法代表了两种不同的优化方向：

其中，balance 致力于降低电路的逻辑深度（level），通过重构逻辑使其更加“平衡”，从而改善电路的时序性能，使其能工作在更高的时钟频率下。

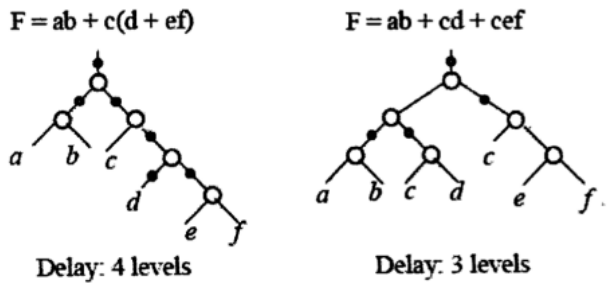
rewrite 的目标则是减少电路中的逻辑门数量（gates），通过寻找功能等价但结构更简单的子图进行替换，以达到优化电路面积、降低功耗的目的。

二、AIG 优化算法

2.1 balance 算法

在数字电路设计中，一个关键的目标是提升电路的速度，也就是时序性能。而逻辑深度的平衡，或者说让电路结构更“平衡”，是实现这一目标的重要手段。AIG balance 算法的核心思想就是通过重构网络，在不改变逻辑功能的前提下，降低关键路径上的逻辑层数。

该算法的流程通常是这样的：首先对整个 AIG 网络进行拓扑排序，确保每个节点在处理时，它的所有输入节点都已经被访问过。接着，算法会遍历每个节点，对它进行“cut 枚举”——也就是找出以它为根节点、输入数量在一定范围内的所有子图。对每一个找到的 cut，算法会计算出其对应的布尔函数，并将其转化为最优的 SOP（Sum-of-Products，积之和）形式。最后，基于这个 SOP 表达式，算法会以延迟最小为目标进行结构重构。如果在所有候选的 cut 中，有一个新的结构能够有效降低当前节点的逻辑深度，那么就用这个最优 cut 的新结构来替换掉原来的逻辑。



上图就是一个很好的例子。左侧的布尔函数 F = ab + c(d + ef)，其对应的 AIG 结构逻辑深

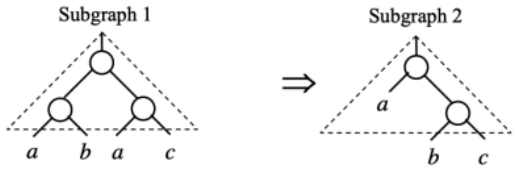
度为 4。这是一种不平衡的结构。通过 balance 操作，我们可以将其等价地转换为 F = ab + cd + cef，如右图所示。可以看到，新的结构逻辑深度降至 3，路径更加均匀，这有助于提升整个电路的最高工作频率。

2.2 rewrite 算法

与追求速度的 balance 不同，rewrite 算法的主要目标是减少电路网络中的逻辑门数量，从而优化芯片面积和资源利用率。它的核心策略同样是基于 cut 枚举，但这次不是为了构建 SOP，而是为了寻找“逻辑等价但结构更优”的子图进行替换。

具体来说，rewrite 算法会遍历 AIG 中的每个节点，并枚举其所有满足特定输入规模（比如 k=4）的 cut。对每一个 cut，算法会计算出其布尔函数的真值表。然后，它会用这个真值表在一个预先计算好的、包含各种最优逻辑实现的数据库（例如 NPN 等价类库）中进行查找。

如果查找到了一个或多个逻辑等价的候选结构，算法会评估每一个替换方案的“收益（Gain）”。这个收益可以通过 Gain = NodesSaved - NodesAdded 来计算，即替换后能节省的旧节点数减去实现新结构所需的新增节点数。如果收益大于零，或者在特定策略下等于零，算法就会选择其中收益最大的方案，用这个更简洁的新结构来替换掉网络中原有的子图。



上图展示了 rewrite 的一个简单场景。假设子图 subgraph1 中计算 a&b 的结果在外部没有被再次使用，那么它就可以被逻辑等价的 subgraph2 所替代。经过这样的转换，总的门数量就减少了，从而实现了向面积更优化的方向演进。

三、代码实现

3.1 mockturtle 框架介绍

我的项目是基于 Mockturtle 这一强大的开源框架完成的。Mockturtle 由 EPFL 实验室开发，是目前学术界进行逻辑综合与优化研究的主流工具之一。它完全使用 C++17 标准编写，设计上高度模块化和可扩展，能够灵活地处理 AIG、XAG、MIG 等多种网络结构。

Mockturtle 最核心的设计理念是“组合与视图（views）”，它可以通过不同的视图层（如 fanout_view, depth_view）来包装和增强一个已有的网络，赋予其额外的功能，而无需改变底层的数据结构。此外，框架内已经为我们准备好了许多高性能的算法模块，比如 cut 枚举、balancing、rewriting 等，这使得我可以把精力更集中在算法的调用和流程设计上，而不是从零开始。

3.2 balance 代码实现

在我的代码中，balance 的核心思路与伪代码完全一致 ，即通过 SOP 重构来优化逻辑深度。

我直接调用了 Mockturtle 提供的 balancing 算法接口来实现这一功能。

首先，我设置了 cut 枚举的相关参数，其中最重要的 cut_size 被设为 4，这意味着算法在优化时会考虑所有输入数不大于 4 的子图。然后，我实例化了一个专门用于 AIG 网络的 SOP 重平衡器 sop_rebalancing<Ntk>。

```
// 定义 SOP 重平衡器
mockturtle::sop_rebalancing<Ntk> rebalancer;
```

接着，我定义了一个回调函数 rebalancing_fn，这个函数会在 balancing 主流程遍历到每个节点时被调用。它接收当前 cut 的真值表和子节点的到达时间等信息，然后调用我们之前定义的 rebalancer 来执行实际的、基于延迟最优的重构。

```
// 定义重平衡回调函数
mockturtle::rebalancing_function_t<Ntk> rebalancing_fn = [&](...) {
    rebalancer(dest, function, children, best_level, best_cost, callback);
};
```

最后，通过一行核心代码启动整个优化流程：

```
// 调用 balancing 函数，并传入网络、回调函数和参数
ntk = mockturtle::balancing(ntk, rebalancing_fn, ps);
```

在执行完 balancing 后，网络中可能会留下一些因为逻辑替换而不再被任何节点引用的“悬空”节点。为此，我调用了 cleanup_dangling(ntk)函数来清理这些无用节点，确保网络结构的干净整洁。整个过程中，我无需手动进行 cut 枚举或 SOP 计算，Mockturtle 的接口为我方便地封装了这些复杂的底层操作

3.3 rewrite 代码实现

我的 rewrite 算法实现也严格遵循了伪代码的逻辑，目标是通过查表和替换来减少门的数量，优化电路面积。在 Mockturtle 中，这个过程可以借助 cut_rewriting_with_compatibility_graph 函数高效完成。

算法实现上，我首先选择并实例化了一个用于 AIG 网络的 NPN 等价类综合器 xag_npn_resynthesis。这个对象内部封装了一个预先计算好的、针对 4 输入布尔函数的 AIG 最优结构数据库。

随后，我配置了 cut_rewriting 的相关参数，同样将 cut_size 设为 4，与伪代码保持一致。

```
// 定义基于 NPN 数据库的重综合函数
xag_npn_resynthesis<aig_network, aig_network, xag_npn_db_kind::aig_complete> resyn;
// 设置 cut rewriting 参数
cut_rewriting_params ps;
ps.cut_enumeration_ps.cut_size = 4;
ps.min_cand_cut_size = 2;
ps.min_cand_cut_size_override = 3;
// 调用 cut_rewriting 核心函数
cut_rewriting_with_compatibility_graph(ntk, resyn, ps);
```

```
// 清理悬空节点
ntk = cleanup_dangling(ntk);
```

当 cut_rewriting_with_compatibility_graph 函数被调用时，Mockturtle 框架会在内部自动完成伪代码中描述的一系列操作：对每个节点进行拓扑遍历和 cut 枚举，计算 cut 的布尔函数，去 resyn 提供的数据库中查找所有等价的最优结构，评估每个替换方案的增益（Gain），并选择最佳方案进行替换。像 DereferenceNode 和 ReferenceNode 这类复杂的引用计数操作，也由框架在底层安全、自动地完成了。

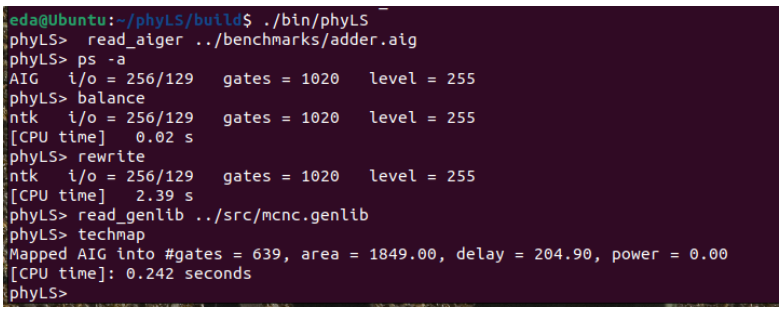
最后，同样调用 cleanup_dangling(ntk)来移除优化后产生的无用节点，保证最终输出的 AIG 网络是简洁和正确的。

四、代码结果分析

为了验证我实现的 balance 和 rewrite 操作的有效性，我使用了几个 benchmark 电路进行了测试。

4.1 adder.aig 分析

在编写 balance 和 rewrite 代码前，先使用 balance 和 rewrite 操作，如下图，可以看到使用前后并没有任何的变化（符合预期，因为 balance 和 rewrite 内部为空）。此时可见 adder.aig 这个 AIG 电路内，参数为 1020 个门电路，电路层级数为 255。



使用优化过的 balance 和 rewrite 代码后，结果如下图，可见此时经过 balance 优化之后，门的数目增加到了 1397 而电路的层数则被大大减少，到了 171 层。这也是符合我们对于 balance 操作的预期的。然后经过 rewrite 操作，巧合的是电路的门数目和电路层数都回到了初始状态（但内部结构还是不同的），这是因为 rewrite 操作将之后用 balance 加工过一遍的 AIG 电路又朝着门数目方向优化，而原先的电路可能恰好是门电路数方向优化的一个极端故遇到了这个巧合。

```
phyLS> read_aiger ../benchmarks/adder.aig
phyLS> ps -a
AIG   i/o = 256/129   gates = 1020   level = 255
phyLS> balance
ntk   i/o = 256/129   gates = 1397   level = 171
phyLS> rewrite
ntk   i/o = 256/129   gates = 1020   level = 255
phyLS> read_genlib ../src/mcnc.genlib
phyLS> techmap
Mapped AIG into #gates = 680, area = 1849.00, delay = 204.90, power =
 0.00
```

接下来我又继续尝试了连续多次的 balance 和 rewrite 操作，如下两图可以看到连接 balance 操作后 gates 数目连续的增加，而 level 数目则相应连续减小。原因在于 balance 仅简单进行一遍遍历，故并不能达到优化目标，多次 balance 可以尽量向优化目标步进（贪婪）。而连续 rewrite 操作后同理，level 数目增加的同时可以有 gates 数目的连续减少。

```
phyLS> balance
ntk   i/o = 256/129   gates = 1397   level = 171
phyLS> balance
ntk   i/o = 256/129   gates = 1604   level = 88
phyLS> balance
ntk   i/o = 256/129   gates = 1965   level = 61
phyLS> balance
ntk   i/o = 256/129   gates = 2216   level = 34
phyLS> balance
ntk   i/o = 256/129   gates = 2583   level = 25
phyLS> balance
ntk   i/o = 256/129   gates = 2790   level = 17
phyLS> balance
ntk   i/o = 256/129   gates = 3059   level = 15
phyLS> balance
ntk   i/o = 256/129   gates = 3153   level = 13
phyLS> balance
ntk   i/o = 256/129   gates = 3153   level = 13
```

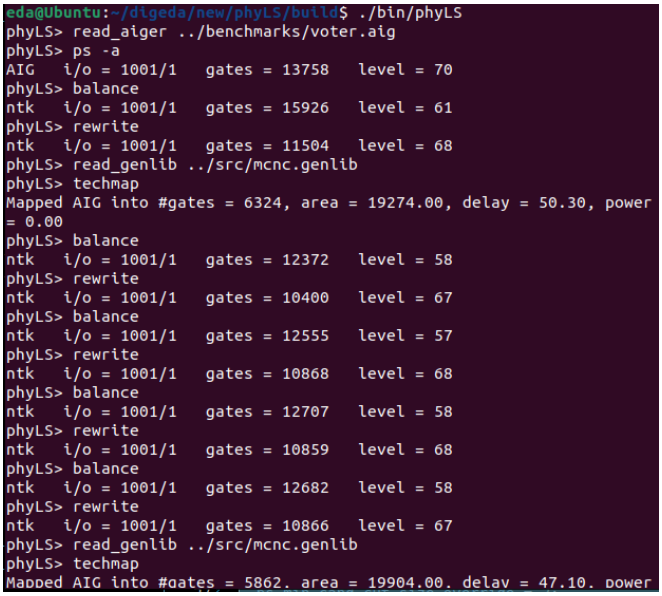
连续 balance 操作

```
phyLS> balance
ntk   i/o = 256/129   gates = 3153   level = 13
phyLS> rewrite
ntk   i/o = 256/129   gates = 2358   level = 19
phyLS> rewrite
ntk   i/o = 256/129   gates = 1916   level = 24
phyLS> rewrite
ntk   i/o = 256/129   gates = 1643   level = 30
phyLS> rewrite
ntk   i/o = 256/129   gates = 1457   level = 35
phyLS> rewrite
ntk   i/o = 256/129   gates = 1383   level = 41
phyLS> rewrite
ntk   i/o = 256/129   gates = 1341   level = 50
phyLS> rewrite
ntk   i/o = 256/129   gates = 1320   level = 60
phyLS> rewrite
ntk   i/o = 256/129   gates = 1298   level = 72
phyLS> rewrite
ntk   i/o = 256/129   gates = 1276   level = 82
phyLS> rewrite
ntk   i/o = 256/129   gates = 1260   level = 92
phyLS> rewrite
ntk   i/o = 256/129   gates = 1245   level = 104
phyLS> rewrite
ntk   i/o = 256/129   gates = 1224   level = 117
phyLS> rewrite
ntk   i/o = 256/129   gates = 1206   level = 128
phyLS> rewrite
ntk   i/o = 256/129   gates = 1186   level = 134
phyLS> rewrite
ntk   i/o = 256/129   gates = 1164   level = 146
```

连续 rewrite 操作

4.2 其他 benchmark

我还使用了 voter.aig 作为 testbench 对优化操作进行验证，结果如下：



这里可见看到在经过 balance 和 rewrite 的两个操作后，gates 和 level 的数目都小于初始状态（13758/70），可以肯定生成了一个结构上更好的 AIG 电路。

另外，我尝试不断重复 balance 和 rewrite 两个操作，可以发现在多次重复后，gates 和 level 的水平都向着更小的方向发展（如最终生成的电路 gates=10866, level=67）。

五、总结

通过本次课程项目，我成功基于 Mockturtle 框架，实现并验证了 AIG 电路的 balance 和 rewrite 两种核心优化算法。在这个过程中，我不仅深入学习了 balance 为时序、rewrite 为面积的优化原理，还理解了它们背后基于 cut 枚举、SOP 重构和查表替换等技术细节，并通过调用 Mockturtle 的 API 将这些理论成功应用到了实践中。实验结果直观地展示了数字电路设计中“面积”与“速度”的权衡关系，让我对 EDA 算法的目标导向性有了更深刻的认识。特别是在对 voter.aig 的测试中，我发现将两种不同目标的优化算法组合使用，能够避免单一策略的局部最优问题，最终在门数和深度上都取得了比初始电路更好的效果。

六、附录

6.1 伪代码

Balance 算法的伪代码：

```
subject_graph performSopBalancing(subject_graph S, int K, int C)
{
   for each node n in S, in topological order {
      compute C structural K-input cuts of n;
      for each cut {
         compute truth table;
         compute irredundant SOP;
         perform delay-optimal balancing of the SOP;
         If (the cut has smaller AIG level than the best cut)
            save the cut as the best cut;
      }
      if (root node AIG level is reduced using the best cut)
          update AIG structure;
   }
   return S;
}
```

Rewrite 算法的伪代码：

```
Rewriting( network AIG, hash table PrecomputedStructures, bool UseZeroCost )
{
   for each node N in the AIG in the topological order {
      for each 4-input cut C of node N computed using cut enumeration {
          F = Boolean function of N in terms of the leaves of C;
          PossibleStructures = HashTableLookup( PrecomputedStructures, F );
          BestS = NULL;
          BestGain = -1;
          for each structure S in PossibleStructures {
             NodesSaved = DereferenceNode( AIG, N );
             NodesAdded = ReferenceNode( AIG, S );
             Gain = NodesSaved - NodesAdded;
             Dereference( AIG, S );
             Reference( AIG, N );
             if ( Gain > 0 || (Gain == 0 && UseZeroCost) ) {
                if ( BestS == NULL || BestGain < Gain ) {
                   BestS = S;
                   BestGain = Gain;
          }
          if ( BestS == NULL ) continue;
```

```
            NodesSaved = DereferenceNode( AIG, N );
            NodesAdded = ReferenceNode( AIG, S );
            assert( BestGain == NodesSaved - NodesAdded );
        }
    }
}
```

6.2 项目环境及运行

Gcc version 11.4.0

Clang version 14.0.0

由于 ubuntu 虚拟机的版本问题，并没能成功安装指定的 gcc 7.0 等低版本工具链（此虚拟机上安装 9.0 以下版本就会报错）。不过经过实验本项目环境同样可以正常运行 mockturtle 框架。

```
Supported LTO compression algorithms: zlib zstd
gcc version 11.4.0 (Ubuntu 11.4.0-1ubuntu1~22.04)
eda@Ubuntu:~$ clang -v
Ubuntu clang version 14.0.0-1ubuntu1.1
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
Found candidate GCC installation: /usr/bin/../lib/gcc/x86_64-linux-gnu/11
Found candidate GCC installation: /usr/bin/../lib/gcc/x86_64-linux-gnu/9
Selected GCC installation: /usr/bin/../lib/gcc/x86_64-linux-gnu/11
```

项目的运行方式与 github 开源项目上方式相同，包括 cmake 及 make 编译等。

6.3 附件

附件包括代码：balance.hpp, rewrite.hpp；及测试截图 adder_test.png, voter_test.png。