CS 516000 FPGA Architecture & CAD

Problem Description

We can represent a Boolean network as a directed acyclic graph in which each node represents a logic gate, and in which a directed edge (i,j) exists if the output of gate i is an input of gate j. A primary input (PI) node has no incoming edge, and a primary output (PO) node has no outgoing edge. We use $inputs(u)$ to denote the set of nodes that supply inputs to gate u. Given a subgraph H of a Boolean network, $inputs(H)$ denotes the set of distinct nodes outside H which supply inputs to the gates in H. For a node v in the network, a K-feasible cone at v, denoted Cv , is a subgraph consisting of v and its predecessors such that $|inputs(Cv)| \leq K$ and any path connecting a node in Cv and v lies entirely in Cv . A Boolean network is K-bounded if $inputs(v) \leq K$ for each node v. We assume that each programmable logic block in an FPGA is a K-input one-output lookup-table (K-LUT) that can implement any Kinput Boolean function. Thus, each K-LUT can implement any K-feasible cone of a Boolean network. The technology mapping problem is to cover a given Boolean network with K-feasible cones. In this assignment, you have to write a C/C++ program to generate a technology mapping of the given Boolean network. The objective of the project is to minimize the number of K-LUTs used.

Input Format

There will be one input file for each testcase which represents a Boolean network. Format: The first line of the file contains the name of the Boolean network and three integers N,I,O. N denotes the number of gates in the Boolean network. I denotes the number of primary inputs and O denotes the number of primary outputs. The following I lines will be the IDs of primary input nodes. And the following O lines will be the IDs of primary output nodes. Every line in the rest of the file lists the node ID of a gate and its input node IDs.

Constraints

- $1 \leq N \leq 105$
- $3 \leq K \leq 8$
- Runtime limit: 10 minutes. (If your program fails to generate a feasible mapping result within 10 minutes, it fails the testcase.)

Output Format

Each line in the output fille represents a LUT in the following format:

<Output ID> <1st Input ID> <2nd Output ID> <Kth Input ID>

My approach

In order to solve this problem, I looked for some inspiration in the papers that are shown in the bibliography of the lectures, however I found them to be too complex so I tried to find some useful information on the slides directly, and there is page 18 I found the diagram shown in figure 1.

From there I started to think how did it go from the original network to only use 3 3-LUTS.

I realized that the circuit could be model as a graph in the following order based on their levels

Level 1 : PI {1, 2, 3, 4, 5}

Level 2 : Gate 6

Level 3 : Gates {8 and 7}

Level 4 : Gates {10 and 9}

Level 5: PO {11}
So I said I can create a LUT for Node 11 and put its inputs inside
So LUT 11 = {10, 9}
I can further expand this as long as inputs less than K(3), starting by the input with higher depth
LUT 11 = {1, 8, 9} , here I expand given than 9, might share inputs with 10
LUT 11 = {1,8,8, 7} after removing repeating entries, inputs are still 3.
so LUT 11 = {1,8,7}

I check for more expansions, I check 8

LUT 11 = {2,6,1,7}, it has more than K entries, so I reverted.

LUT 11= {1,8,7} I expand 7 now, but same happen so cannot be a valid expansion.

And as for one since it is a primary input, I cannot expand it either.
I realized If I do that for the whole network, I can optimize it, however nodes 10 and 9 were merged so don't want to process them again so I marked them as merged so they are skipped.

But for further cases, for example where LUT 8 has input {2 3 4}, it marked 6 as a merged node, meaning it will be skipped in further processing which can be leading to unconnected components, so I create a rule that if a merged node has been merged, but it is still a part of an input a LUT must be created for it.
This case lead me to a more robust implementation that I detail below.
Also figure 1 and 2 show the network I used and the graph representation.

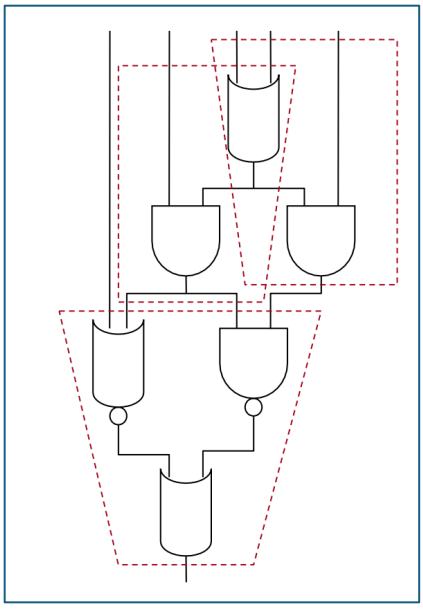

Figure 1 Circuit

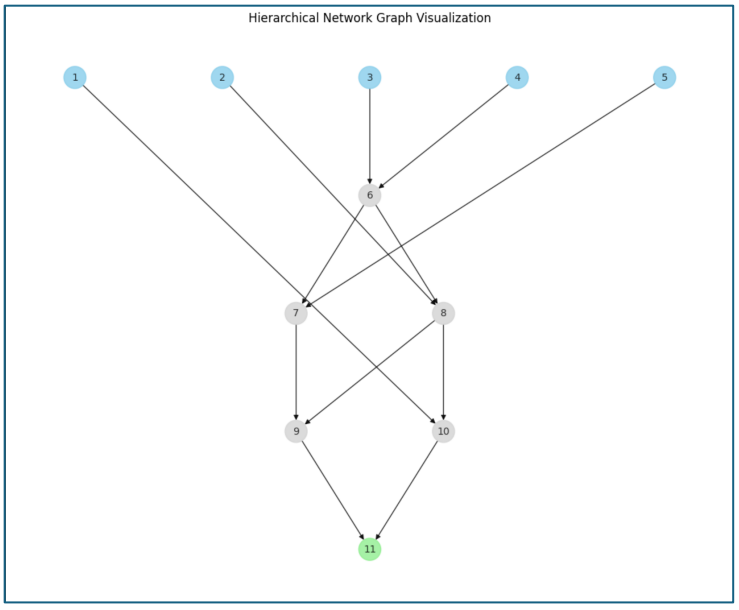

Figure 2 Graph Representation

Step 1. Graph Initialization

Initially, I conceptualize the boolean network as a graph where nodes represent logic gates and primary inputs, and edges denote the connections between them.

Step 2: Reading and Constructing the Network

The ***readFromFile*** function parses the input contained in the file, reading the inputs in the way outlined by the input section.

As the file is being read the data to build the graph is stored here:
I created 3 data structures to save the primary inputs, primary outpurs and the gates.

Step 3: Assigning Levels to Nodes

Using the assignLevelsOptimized function, each node (gate) in the graph is assigned a level based on the longest path from the node to the nearest primary input. This hierarchical structuring is crucial for understanding which gates can be combined into a single LUT without violating the logical integrity of the circuit.

The function uses a breadth-first search (BFS) strategy starting from each primary input and traverses through the network, marking the depth (or level) of each node as it proceeds.

Step 4: Generating the lUTS

The generateLUTs function is the core of my algorithm designed to optimize the circuit by creating Look-Up Tables (LUTs) from a network of gates. This function takes as input the primary outputs, the connections between gates, levels of nodes, primary inputs, and a predefined limit KKK which specifies the maximum number of inputs allowed per LUT.

Step 4.1: Setting Up Data Structures

To get started, I first set up some essential data structures. I use a vector<pair<int, set<int>>> luts to store the final LUTs, where each element is a pair consisting of an output node and its set of inputs. I also have a set<int> visited that tracks nodes already processed and included in an LUT to avoid redundant processing.

Additionally, I use set<int> mergedNodes to track nodes that have been merged into any LUT, which helps in managing dependencies as the LUTs are expanded. Lastly, the set<int> inputTracker monitors nodes that serve as inputs to other LUTs, ensuring that if these nodes are reused later, they are properly handled.

Step 4.2: Defining the expandLUT Function

To handle LUT expansion, I define a lambda function called expandLUT. This function is responsible for recursively expanding the inputs of a node's inputs and is crucial for optimizing the size of the LUTs. I chose to use a lambda function here because it keeps the code cleaner and more contained within the context of the main function. The expandLUT function takes three arguments: lutSet, which holds the current list of inputs for the LUT; node, which is the current node being processed; and expandedSet, which tracks nodes that have already been expanded in the current context to prevent infinite loops. The function starts by checking if the node is a primary input—since primary inputs cannot be expanded further, they are simply returned as-is. If the node exists in the gates map and has not been expanded before, I add its inputs to lutSet.

Step 4.3: Defining the createLUT Function

The core of the LUT generation process is handled within another lambda function called createLUT. This function is responsible for creating an LUT for a given node, starting with its immediate inputs and then expanding them recursively. The first step in createLUT is to check if the node has already been processed (i.e., it's in the visited set) or if it's a primary input; if either condition is true, an empty vector is returned, indicating that there is no need to generate an LUT for this node. For nodes that need to be processed, I initialize a set called lutSet with the direct inputs of the node and use expandedInputs to keep track of nodes that have already been expanded. Additionally, I use tempMergedNodes to record nodes that are merged during the expansion. To expand the LUT efficiently, I use a deque<int> called queue to manage nodes that need to be expanded. I initially populate the queue with the direct inputs of the node.

Step 4.4: Expanding the LUT

The main logic for expanding the LUT involves iterating over the queue and processing each node. I sort the nodes in the queue by their level, giving priority to

nodes with the highest depth. This approach ensures that deeper nodes, which are likely to have more dependencies, are expanded first. For each node in the queue, I call the expandLUT function to attempt to add its inputs to the current LUT. If adding these inputs causes the total number of inputs in the LUT to exceed KKK, I revert to a previous state (previousLutSet) and stop expanding this LUT. This approach ensures that the input limit is respected and that the LUT remains within the hardware constraints. If the expansion does not exceed the limit, I update the LUT with the new inputs and continue expanding.

Step 4.5: Finalizing the LUT

Once an LUT is fully expanded and remains within the input limit, I finalize it. At this point, I mark all nodes that were merged into this LUT by adding them to the mergedNodes set. I also add the current node to the visited set to avoid processing it again in future iterations. Additionally, I insert the inputs used in this LUT into the inputTracker to keep track of nodes that may serve as inputs in other LUTs. The createLUT function then returns the finalized list of inputs as a vector.

Step 4.6: Processing Nodes by Level

After defining the helper functions, I proceed to generate LUTs for the entire circuit by processing nodes according to their levels. To do this, I first group nodes by their levels using a map called nodesByLevel, where each key represents a level and the corresponding value is a list of nodes at that level. I then iterate through the levels in reverse order, starting with the deepest level. For each node at a given level, I check if it has already been visited or if it has been merged but is no longer required. If a node has not been processed, I call createLUT to generate an LUT for it. If an LUT is successfully created, I add it to the luts vector.

Step 4.7: Handling Prematurely Merged Nodes

After generating LUTs for the main nodes, I handle any nodes that were prematurely merged but still need their own LUTs. I do this by iterating through the inputTracker set to find nodes that are used as inputs in other LUTs but do not yet have their own LUTs. For each such node, if it hasn’t been visited and isn’t a primary input, I generate an additional LUT to cover it.

Step 4.8: Returning the Final LUTs

Finally, the function returns the luts vector, which represents the optimized structure of the circuit. Each entry in this vector includes an output node and its corresponding set of inputs, effectively compressing the circuit into a format suitable for FPGA implementation. This approach ensures efficient compression of the circuit into LUTs while adhering to the input limit KKK, leveraging sets, queues, and a backtracking mechanism to ensure both efficiency and correctness.

Step 5: Saving the LUts to a file

I think the description of the function it is very straightforward and somehow trivial, but in short it just save the resulting LUTS into an output file.