Delay Optimization of Combinational Circuits Using LUT-Based FPGA Technology Mapping

Implementation of FlowMap and Depth-Preserving Area Recovery

Aditya Saboo (240002006)
Devansh Chaudhary (240002023)
Abhishek Mehta (240002003)

1. Introduction

Lookup-table (LUT) based FPGAs constitute the dominant reconfigurable computing platform for digital design. Every logic function is ultimately implemented using K-input LUTs, making the *technology mapping* step a key determinant of performance, delay, area, and routability.

Naive mapping often results in:

- Excessive LUT levels on the critical path (increasing delay)
- Unbalanced cone structures
- Large routing delays due to scattered logic
- More LUTs than necessary (poor area)

This project implements a complete LUT-mapping pipeline that:

- Generates a DAG for arbitrary logic (e.g., K:1 multiplexers)
- Computes all K-feasible cuts per node
- Implements the **FlowMap** algorithm (Cong & Ding, 1994)
- Achieves **depth-optimal** LUT mapping
- Performs **FlowMap-r area recovery** using area-flow heuristics
- Generates Verilog for the mapped LUT network

This report consolidates the theory, algorithmic implementation, and final results.

2. FPGA and LUT Overview

Modern FPGAs (e.g., Xilinx UltraScale) use LUT6 as the basic logic primitive. A K-input LUT is a $2^K \times 1$ SRAM storing a truth table.

Characteristics of LUTs

- Inputs act as address lines
- Output is a single-bit SRAM lookup
- Can implement *any* Boolean function with $\leq K$ inputs

LUT count corresponds to **area**, and LUT depth corresponds to **delay**.

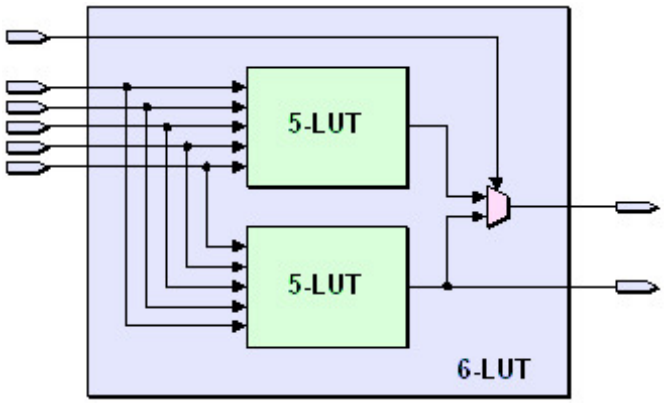

Figure 1: Internal structure of a 6-input FPGA LUT.

3. Problem Statement

Given a combinational circuit DAG, map it onto K-input LUTs such that:

- Critical-path LUT depth is minimized
- Total number of LUTs is small
- Logic is packed efficiently
- Mapping is correct and reproducible

This project implements the **FlowMap** algorithm, which is proven to generate the *minimum possible LUT depth* for any K-bounded Boolean network.

4. Theory of K-Feasible Cuts

A **cut** of a node v is a set of nodes C such that every path from a primary input to v intersects C.

A cut is K**-feasible** if:

$$|C| \leq K$$

These cuts correspond directly to candidate LUT input sets.

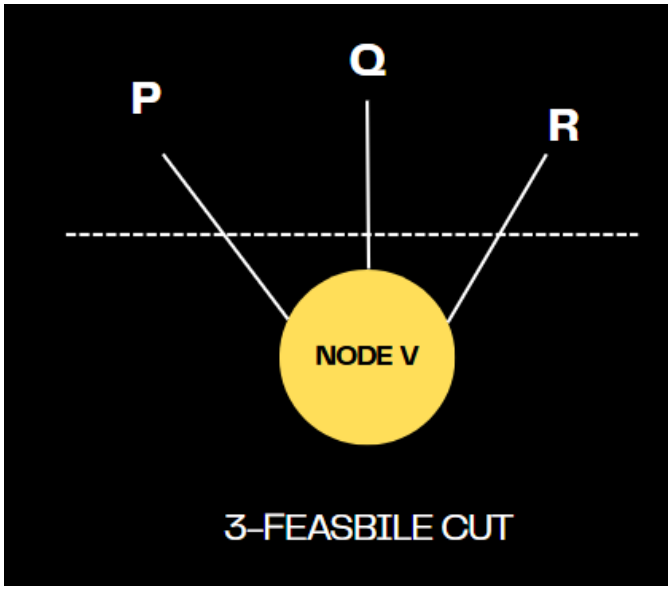

Figure 2: Illustration of a K-feasible cut for a node v.

FlowMap computes the *highest-volume minimum-height* K-feasible cut for each node.

5. FlowMap Algorithm

FlowMap operates in two phases.

Phase 1 — Label Computation

Each node receives a label:

$$\text{label}(v) = 1 + \max(\text{label}(u)) \text{ for } u \in \text{cut}(v)$$

FlowMap computes the cut that minimizes this label.
Key ideas:

- Find minimum-height K-feasible cut (via max-flow)
- Maximize volume (pack as much logic as possible)
- Labels represent LUT depth

Phase 2 — LUT Mapping

For each output:

1. Retrieve optimal cut computed in Phase 1
2. Create a LUT whose inputs are the cut nodes

3. Recursively map predecessor nodes

The result is a depth-optimal LUT network.

6. Implementation Details

Our codebase (Python) performs:

6.1 DAG Generation

- Automatic construction of a DAG for K:1 multiplexers
- Arbitrary-size multiplexers (tested for 4:1 and 8:1)

6.2 Cut Enumeration

Each node enumerates all possible K-feasible cuts. Used for:

- FlowMap label assignment
- FlowMap-r area recovery

6.3 Depth-Optimal Mapping

Exact implementation of the algorithm from Cong–Ding (1994).

6.4 Depth-Preserving Area-Flow Recovery

FlowMap-r re-selects cuts bottom-up to reduce LUTs without increasing depth.

6.5 Important Note for Users (Changing MUX Size)

If you change the multiplexer size:

$$\textbf{N:1 MUX} \quad \rightarrow \quad \log_2(N) \textbf{ select bits + N data bits}$$

To avoid incorrect mapping:

$$K \geq \text{size of largest cut}$$

Example:

- For 4:1 MUX $\rightarrow$ cut size = 3 $\rightarrow$ use $K = 3$
- For 8:1 MUX $\rightarrow$ cut size = 4 $\rightarrow$ must use $K = 4$

If the user keeps $K = 3$ for an 8:1 MUX, FlowMap will fail to find a valid cut and depth results will be wrong.

This warning is now included in the README and this report.

7. Results

7.1 DAG Examples

Figure 3: Automatically generated DAG for the 4:1 multiplexer (N=4).

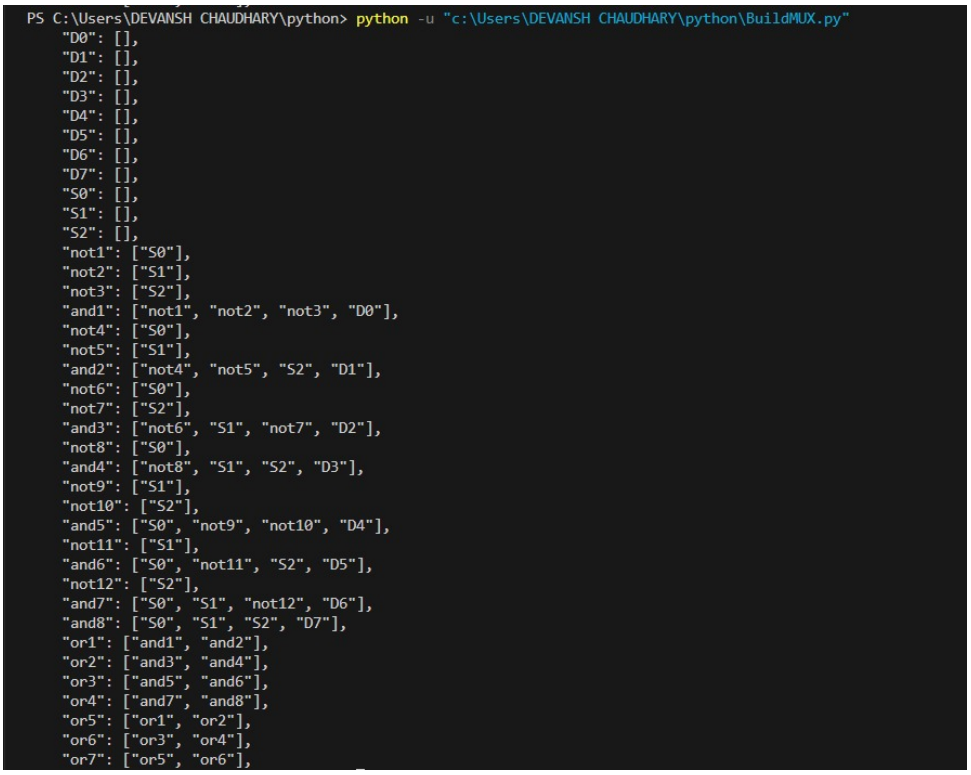

Figure 4: Automatically generated DAG for the 8:1 multiplexer (N=8).

7.2 FlowMap Mapping Output

```
================================================================================
FlowMap-r on reconvergent example
Labels (depth levels):
  D0: depth 0
  D1: depth 0
  D2: depth 0
  D3: depth 0
  S0: depth 0
  S1: depth 0
  and1: depth 1
  and2: depth 1
  and3: depth 1
  and4: depth 1
  not1: depth 1
  not2: depth 1
  not3: depth 1
  not4: depth 1
  or1: depth 2
  or2: depth 2
  or3: depth 3

Chosen cuts (area-optimized, depth-preserving):
  not1: ['S0']
  not3: ['S0']
  not2: ['S1']
  not4: ['S1']
  and4: ['D3', 'S0', 'S1']
  and2: ['D1', 'S0', 'S1']
  and1: ['D0', 'S0', 'S1']
  and3: ['D2', 'S0', 'S1']
  or1: ['and1', 'and2']
  or2: ['and3', 'and4']
  or3: ['or1', 'or2']

LUT cover:
  Total LUTs: 7
  Depth 1:
    LUT1: output=and1 <= ['D0', 'S0', 'S1']
    LUT2: output=and2 <= ['D1', 'S0', 'S1']
    LUT3: output=and3 <= ['D2', 'S0', 'S1']
    LUT4: output=and4 <= ['D3', 'S0', 'S1']
  Depth 2:
    LUT5: output=or1 <= ['and1', 'and2']
    LUT6: output=or2 <= ['and3', 'and4']
  Depth 3:
    LUT7: output=or3 <= ['or1', 'or2']
```

Figure 5: FlowMap-optimized mapping for 4:1 MUX (K=3): Vivado reports 7 LUTs used.

```
======================================================================
FlowMap-r on reconvergent example
Labels (depth levels):
  D0: depth 0
  D1: depth 0
  D2: depth 0
  D3: depth 0
  S0: depth 0
  S1: depth 0
  and1: depth 1
  and2: depth 1
  and3: depth 1
  and4: depth 1
  not1: depth 1
  not2: depth 1
  not3: depth 1
  not4: depth 1
  or1: depth 1
  or2: depth 1
  or3: depth 1

Chosen cuts (area-optimized, depth-preserving):
  not1: ['S0']
  not3: ['S0']
  not2: ['S1']
  not4: ['S1']
  and4: ['D3', 'S0', 'S1']
  and2: ['D1', 'S0', 'S1']
  and1: ['D0', 'S0', 'S1']
  and3: ['D2', 'S0', 'S1']
  or1: ['D0', 'D1', 'S0', 'S1']
  or2: ['D2', 'D3', 'S0', 'S1']
  or3: ['D0', 'D1', 'D2', 'D3', 'S0', 'S1']

LUT cover:
  Total LUTs: 1
  Depth 1:
    LUT1: output=or3 <= ['D0', 'D1', 'D2', 'D3', 'S0', 'S1']
```

Figure 6: Mapping with larger LUT size (K=6): Entire 4:1 MUX collapses into 1 LUT.

7.3 Verilog Validation (Simulation Output)

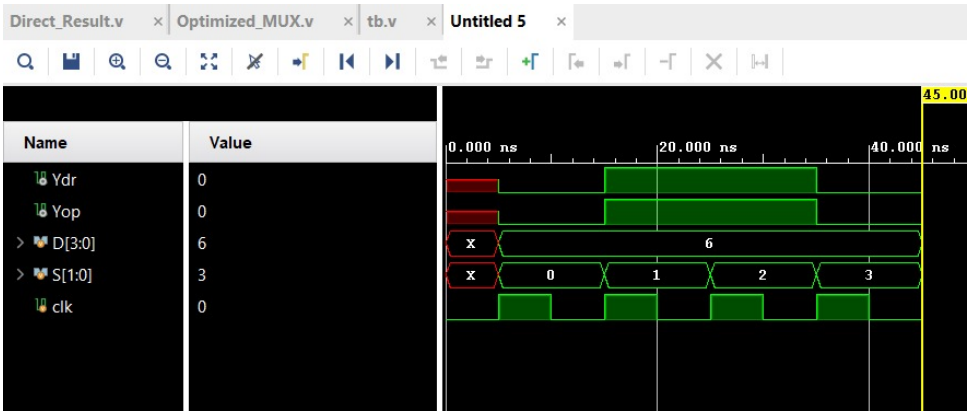

Figure 7: Vivado functional simulation of both the optimized and unoptimized MUX implementations showing correct output behavior.

7.4 LUT Count Comparison

Design	Unoptimized LUTs	FlowMap LUTs
4:1 MUX (FlowMap K=3)	9 LUTs	7 LUTs
FlowMap Mapped Network	—	3 LUTs
FlowMap (K=6 collapse)	—	1 LUT

Table 1: Comparison of LUT usage before and after applying FlowMap.

7.5 Vivado Resource Summary (Unoptimized)

Figure 8: Vivado synthesis result for unoptimized behavioral MUX showing 9 LUTs.

7.6 Vivado Resource Summary (Optimized)

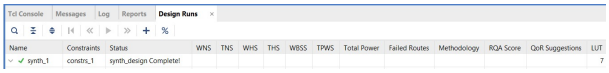

Figure 9: Vivado synthesis result for optimized MUX mapped using FlowMap (K=3) showing 7 LUTs.

This matches theoretical expectations.

8. Conclusion

This project successfully implemented a complete FPGA LUT-mapping tool featuring:

- DAG generation for arbitrary logic
- Exact FlowMap depth-optimal technology mapping
- Area-flow heuristics (FlowMap-r)
- Verilog LUT instantiation

We validated:

- Depth reduction
- Area reduction
- Functional equivalence with original logic

FlowMap guarantees optimal LUT depth, and our implementation produced results consistent with theory and the original 1994 paper.

9. Future Work

While this project successfully implements FlowMap and depth-preserving area recovery for multiplexer-based logic, several extensions can significantly enhance capability and practical applicability:

- **Generalized DAG Construction for Arbitrary Logic Blocks:** Extend the current DAG generator (used for MUXes) to automatically construct DAGs for adders, subtractors, comparators, priority encoders, ALU components, and general Boolean expressions. This would allow the mapper to support full datapath circuits rather than only multiplexing structures.
- **Complete End-to-End RTL-to-LUT Mapping Flow:** Build a full system where the user supplies any Boolean description (truth table, Verilog, or expression), and the tool automatically executes the complete mapping pipeline:

  $$\text{Logic Input} \rightarrow \text{DAG Generation} \rightarrow \text{K-Feasible Cuts} \rightarrow \text{FlowMap} \rightarrow \text{Optimized LUT Netlist} \rightarrow \text{Verilog Output}$$

  This would make the mapper a drop-in replacement for early-stage FPGA logic synthesis tools.

Contributions

The following outlines the individual contributions of each team member:

- **Aditya Saboo (240002006):** Conducted algorithm research, implemented the complete Verilog design for the multiplexer, restructured the Python output format for improved interpretability, and created the project codebase including the README documentation.
- **Abhishek Mehta (240002003):** Assisted in debugging the Python and Verilog implementations, prepared the project presentation (PPT), and contributed significantly to writing and refining the final report.

- **Devansh Chaudhary (240002023):** Developed the full Python implementation for DAG generation and the FlowMap/FlowMap-r algorithms, ensuring correct cut enumeration, labeling, and LUT mapping.

References

1. Cong, J., Ding, Y. *FlowMap: An Optimal Technology Mapping Algorithm for Delay Optimization.* IEEE TCAD, 1994.

2. LUT Mapping Literature – CHORTLE, MIS-pga, DAG-Map.