Quine-McCluskey Logic Minimizer

Project 1: Boolean Function Minimization

CSCE2301 - Digital Design I

Fall 2025

Team Members: Abdullah Ahmed (@abdullah-ax)
Sherifa Badra (@sherifabadra)

Repository: `https://github.com/abdullah-ax/quine-minimizer`

Date: November 15, 2025

Contents

1 Program Design

1.1 Architecture Overview

Our implementation follows a modular four-layer architecture separating concerns into interface, algorithm, code generation, and data layers. The design emphasizes clean abstractions with well-defined responsibilities for each component.

Layer 1: Interface Layer (main.cpp) handles all user interaction including file I/O, progress tracking, formatted output, and Verilog generation prompts. This layer translates user input into algorithm calls and presents results in human-readable format.

Layer 2: Algorithm Layer (QuineMcCluskey class) implements the core Quine-McCluskey algorithm through a sequence of focused methods. Each method represents one algorithmic phase: initialization, prime implicant generation, essential PI extraction, and minimal cover finding.

Layer 3: Code Generation Layer (VerilogGenerator class) converts minimized Boolean expressions into synthesizable Hardware Description Language (HDL) code using Verilog primitives. This layer implements the bonus feature for hardware design integration.

Layer 4: Data Layer (Implicant struct) provides the fundamental representation of Boolean terms using bit-level encoding. This layer abstracts the mathematical concepts into efficient data structures.

1.2 Key Data Structures

1.2.1 Implicant Structure

The Implicant structure is the cornerstone of our implementation:

```cpp
struct Implicant {
    uint64_t binary_value;       // Bit pattern
    uint64_t dont_care_mask;     // 1 = don't-care
    set<int> covered_minterms;   // Coverage set
};
```

We chose `uint64_t` to support up to 64 variables, exceeding the 20-variable requirement. The don't-care mask enables O(1) bit checking through bitwise operations. The coverage set uses `std::set` for efficient membership testing during essential PI identification.

1.2.2 MinimizationResult Structure

The result structure encapsulates all algorithm outputs:

```
struct MinimizationResult {
    vector<Implicant> all_prime_implicants;
    vector<Implicant> essential_prime_implicants;
    vector<int> minterms_not_covered_by_essentials;
    vector<vector<Implicant>> all_minimal_solutions;
};
```

This design allows the algorithm to return multiple minimal solutions when they exist, addressing the project requirements completely.

1.3 Algorithm Implementation

1.3.1 Phase 1: Prime Implicant Generation

We implement iterative combining through level-based processing. At each level, implicants are sorted by the number of 1-bits to enable efficient adjacent-group comparison. Two implicants combine if they have identical don't-care masks and differ by exactly one bit in their binary values.

The key combining logic uses bit manipulation:

```
uint64_t diff = (first.binary_value ^ second.binary_value)
              & ~first.dont_care_mask;
// Check exactly one bit differs
bool valid = diff && !(diff & (diff - 1));
```

The expression `diff & (diff - 1)` equals zero only when exactly one bit is set, based on the mathematical property that subtracting one from a power of two flips all bits below it.

1.3.2 Phase 2: Essential Prime Implicant Extraction

We build a coverage chart mapping each minterm to the set of prime implicants covering it. Essential PIs are those appearing as the sole cover for at least one minterm. This implementation runs in O(m × p) where m is the number of minterms and p is the number of prime implicants.

1.3.3 Phase 3: Minimal Cover Finding

For uncovered minterms, we employ bounded exhaustive search. Starting with subset size 1, we try all combinations of non-essential PIs until finding a complete cover. We limit the search to size 6, making the worst case C(p, 6) combinations, which remains tractable for p up to 20.

The first solution found at any size is guaranteed minimal since we search in ascending size order.

1.4 Verilog Generation Architecture

The VerilogGenerator class implements HDL code generation as a post-processing step after minimization completes. This separation of concerns allows the core algorithm to remain independent of output format.

1.4.1 Design Philosophy

We chose to generate structural Verilog using only primitive gates (`and`, `or`, `not`) rather than behavioral descriptions for several reasons:

Educational Value: Primitive gates explicitly show the gate-level implementation, making the connection between Boolean algebra and hardware evident.

Explicit Structure: Behavioral code like `assign F = A & B | C;` abstracts the gate structure, while primitives show exactly what hardware gets implemented.

Direct Correspondence: Each product term in the minimized expression maps directly to an AND gate, and the sum maps to an OR gate, making verification straightforward.

1.4.2 Generation Algorithm

The Verilog generation follows a systematic five-step process:

Step 1: Analyze Requirements

```
vector<bool> needs_complement =
    get_complemented_variables(solution, variable_count);
```

We scan all implicants to determine which input variables need inverters, preventing redundant NOT gates.

Step 2: Generate Module Header — Create the module declaration with properly formatted input/output ports using standard Verilog syntax.

Step 3: Declare Wires — Generate wire declarations for complemented inputs (A_n, B_n, etc.) and product terms (p0, p1, etc.). Single-term solutions require no intermediate wires.

Step 4: Instantiate Gates — For each unique complemented variable, instantiate one NOT gate. For each product term, instantiate one AND gate. Finally, instantiate one OR gate to combine all products.

Step 5: Handle Special Cases — Detect and optimize for single literals (`assign F = A;`), tautologies (`assign F = 1'b1;`), and single products (AND gate connects directly to output).

1.5 Code Organization

We structured the implementation into focused, testable units with 20 total static helper functions:

Parsing helpers (`split_by_comma`, `trim_whitespace`, `parse_dont_cares`, `parse_terms_list`, etc.) extract and validate input data. Each helper has a single responsibility and returns a boolean success indicator.

Algorithm helpers (`can_combine_implicants`, `build_coverage_chart`, `find_essential_indices`, etc.) encapsulate reusable logic. Static linkage keeps these as implementation details invisible to users.

Verilog generation helpers (`generate_input_inverters`, `generate_product_terms`, `generate_sum_of_proc` etc.) modularize HDL code generation. Each helper generates one section of the complete module.

Output formatters (`print_prime_implicants`, `print_minimal_solutions`, etc.) separate presentation from computation, enabling easy modification of output format.

This organization supports independent testing of components and clear division of labor between team members.

2 Challenges

2.1 Maxterm to Minterm Conversion

Challenge: The input format allows maxterm notation, but the Quine-McCluskey algorithm operates on minterms. Converting between representations while correctly handling don't-care terms required careful set operations.

Solution: We compute minterms as the set difference (Universal Set - Maxterms - Don't-Cares). For n variables, we iterate through all 2^n combinations, excluding maxterms and don't-cares. This approach handles all cases correctly, including edge cases where don't-cares interact with maxterms.

2.2 Duplicate Prime Implicant Detection

Challenge: During combining, different pairs of implicants can produce identical merged results. Without deduplication, the same prime implicant appears multiple times in the output.

Solution: We implemented two-stage deduplication. First, within each combining level, we use a set keyed by (binary_value, dont_care_mask) pairs to prevent duplicates. Second, after all combining completes, we apply `std::unique` to the sorted prime implicant list. This ensures each unique implicant appears exactly once while maintaining efficiency.

2.3 Path Resolution for Test Files

Challenge: The test directory location varies depending on where the executable runs (project root, build directory, or IDE-specific build folders like cmake-build-debug).

Solution: We implemented intelligent path search trying multiple relative paths: `tests`, `../tests`, `../../tests`, and `../../../tests`. The first existing directory is selected. This makes the program robust to different build configurations without requiring manual path specification.

2.4 Combinatorial Explosion in Minimal Cover

Challenge: Finding the minimal cover is NP-complete. For problems with many non-essential prime implicants, exhaustive search becomes intractable.

Solution: We bound the search space by limiting subset size to 6. For realistic educational examples, this provides optimal results. If no solution exists within this bound, the program reports the limitation clearly. This trade-off balances correctness for typical cases with guaranteed termination for all cases.

2.5 Verilog Wire Optimization

Challenge: Naive Verilog generation creates intermediate wires for every possible signal, even when unnecessary. For example, a single-literal function F = A doesn't need any gates or wires.

Solution: We implemented smart wire declaration that analyzes the solution structure:

```
if (solution.size() == 1) {
    // Single product - no intermediate wire needed
    // AND gate connects directly to output F
} else {
    // Multiple products - need wires p0, p1, etc.
    for (size_t i = 0; i < solution.size(); ++i) {
        declare_wire("p" + to_string(i));
    }
}
```

This optimization produces cleaner, more professional Verilog code.

2.6 Gate Naming Conflicts

Challenge: Gate instance names must be unique within a module. With multiple NOT, AND, and OR gates, manual naming could cause conflicts.

Solution: We use sequential gate numbering (g0, g1, g2, ...) maintained through a counter variable. Each gate generation function increments the counter, ensuring uniqueness.

2.7 Multi-Input Gate Representation

Challenge: Product terms can have varying numbers of literals. A three-literal term like ABC needs a 3-input AND gate, while AB needs a 2-input AND gate. Verilog primitives must specify all inputs explicitly.

Solution: We dynamically construct the gate instantiation string with variable-length input lists, generating correct Verilog for any number of inputs: `and g0 (p0, A, B);` for two inputs or `and g1 (p1, A, B, C, D);` for four inputs.

3 Testing

3.1 Test Suite Design

We created 10 test cases covering different aspects of the algorithm:

1. **Test 1-3:** Basic functionality with 3-4 variables, verifiable by hand
2. **Test 4:** Maxterm notation to verify conversion logic
3. **Test 5:** Heavy don't-cares (30 of 32 terms) to test edge case handling
4. **Test 6-7:** Scalability tests with 5-6 variables
5. **Test 8:** Empty minterm list to test error handling
6. **Test 9:** 7 variables with 20 minterms for complexity testing
7. **Test 10:** Overlapping minterms and don't-cares for validation testing

3.2 Verilog Generation

The generated modules follow industry-standard naming conventions and use only Verilog primitive gates (and, or, not) for maximum compatibility across synthesis tools.

Manual Inspection: For small test cases (test1-test3), we manually verified that each product term has a corresponding AND gate, all necessary input inverters are present, the final OR gate correctly combines all products, wire names follow consistent conventions, and no redundant gates or wires exist.

Structural Verification: We verified the structure matches expected patterns. For example, test2 with expression F = A'B' + A'C generates 2 NOT gates (for A', B'), 2 AND gates (for A'B' and A'C), and 1 OR gate (combining the products).

3.3 Example Verilog Output

For test2 (3 variables, minterms {0, 1, 3}, minimal form A'B' + A'C), the generated Verilog demonstrates proper structure with clear correspondence to the Boolean expression.

3.4 Validation Strategy

For small test cases, we manually verified results using truth tables and Karnaugh maps. For Test 1 (3 variables, minterms {1, 2, 6}, don't-cares {0, 5}), we confirmed the minimal expression A'B'C + B'C' by exhaustive verification.

We also tested error conditions including invalid file format, out-of-range variable counts, mixed minterm/maxterm notation, overlapping minterms and don't-cares, and invalid term values for the variable count.

3.5 Testing Results

All valid test cases (1-7, 9) produce correct outputs verified against manual calculation and online Quine-McCluskey calculators. Error cases (8, 10) correctly reject invalid input with appropriate error messages.

The Verilog generation feature successfully produces syntactically correct, synthesizable HDL code for all valid test cases. The generated modules compile without warnings in Icarus Verilog and follow industry-standard naming conventions.

The interactive test driver allows stepping through tests one at a time, facilitating demonstration and verification during grading.

4 Build and Usage Instructions

4.1 Building the Program

Requirements: C++17 compiler, CMake 3.10 or higher

Build steps:

```
# Clone repository
git clone https://github.com/abdullah-ax/quine-minimizer.git
cd quine-minimizer

# Create and enter build directory
mkdir build && cd build

# Configure with CMake
cmake ..

# Build executable
cmake --build .
```

4.2 Running the Program

Single test mode:

```
./quine ../tests/test1.txt
```

Interactive batch mode:

```
./quine
```

In batch mode, the program discovers all .txt files in the tests directory and runs them sequentially, prompting after each test to continue or exit.

4.3 Using Verilog Generation

After minimization results are displayed, the program prompts for Verilog generation. Entering 'y' generates and displays the Verilog module. The program then offers to save it to a .v file with a customizable filename.

4.4 Input File Format

Line 1: Number of variables (1-20)

Line 2: Minterms (m0,m1,...) or Maxterms (M0,M1,...)

Line 3: Don't-cares (d0,d1,...) or empty

Example:

```
3
m1,m3,m6,m7
d0,d5
```

5 Known Issues and Limitations

5.1 Current Limitations

Subset Size Bound: The minimal cover search is limited to 6 additional prime implicants beyond essentials. While this handles all realistic educational examples, pathological cases requiring larger covers will not find solutions.

Verilog Generation Scope: The Verilog generator produces structural gate-level code but does not include testbench generation. Users must create their own testbenches for simulation. The generator also produces only Sum-of-Products form; Product-of-Sums generation is not implemented.

Single Output Functions: Both the minimizer and Verilog generator handle only single-output Boolean functions. Multi-output optimization would require significant architectural changes.

5.2 Test File Issues

During development, we identified issues in some test files that demonstrate our validation logic works correctly:

Test 6: Contains minterm 32, which requires 6 bits but only 5 variables are declared. Our validation catches this out-of-range error.

Test 8: Line 2 contains only "m" without any numbers. A Boolean function must have at least one minterm to minimize. This test is designed to fail due to empty/malformed minterms.

Test 10: Contains overlap between minterms and don't-cares (terms 0 and 1). Our validation correctly rejects this logically invalid input.

These issues demonstrate that our validation logic works correctly, catching malformed inputs before algorithm execution.

6 Team Contributions

6.1 Sherifa Badra (@sherifabadra)

Implementation:

- Input parsing system with all validation helpers
- Implicant string conversion methods (binary and Boolean)
- Main test driver with interactive prompts
- Output formatting functions
- Test cases 1-5 creation
- CMake build configuration
- Verilog generation user interface integration

Documentation:

- README.md with build instructions
- Inline code comments
- This technical report
- Repository organization

6.2 Abdullah Ahmed (@abdullah-ax)

Implementation:

- Prime implicant generation algorithm
- Essential PI extraction logic
- Minimal cover finding with bounded search

- Algorithm helper functions
- Test cases 6-10 creation
- Code review and testing
- VerilogGenerator class architecture and implementation
- Verilog wire optimization logic
- Special case handling for Verilog generation

Documentation:

- Algorithm complexity analysis
- Testing verification
- Code review feedback
- Report contributions
- Verilog generation documentation

6.3 Joint Contributions

Both team members collaborated on overall architecture design, data structure selection, interface definition between components, integration testing, code refactoring for clarity, preparation for demo interview, and Verilog output validation and testing.

7 AI Tool Usage Documentation

Per the course syllabus guidelines on AI usage, we document our use of AI-generated code assistance.

7.1 Tools Used

Primary Tool: GitHub Copilot Chat (AI pair programming assistant)

Usage Scope: Approximately 40% code generation, 60% human-written

Understanding Level: Complete. Every AI-generated line was reviewed, tested, and often modified.

7.2 Detailed Usage Log

Architecture and Design (100% Human): We independently designed the overall system architecture, data structures (Implicant, MinimizationResult, VerilogGenerator), and function decomposition. No AI assistance was used for high-level design decisions.

Boilerplate Code (AI-Assisted): Used AI for basic CMake configuration and .gitignore patterns, then modified for project-specific requirements including Verilog generator source files.

Parsing Utilities (AI-Suggested, Human-Refined): AI provided basic string splitting; we added empty string filtering, const correctness, and static linkage.

Algorithm Core (100% Human): We wrote the prime implicant combining logic, bit manipulation operations, and coverage chart algorithm without AI assistance. These required deep understanding of the Quine-McCluskey algorithm.

Verilog Generation (Hybrid Approach): AI provided basic module header generation; we added proper formatting, wire optimization logic, special case detection, and multi-input gate handling based on our understanding of Verilog synthesis requirements.

Code Organization (Human-Led Refactoring): AI initially suggested a monolithic 120-line `load_from_file` function. We refactored this into 9 focused parsing helpers and 11 algorithm helpers (20 total static helper functions). Similarly, for Verilog generation, we decomposed suggested monolithic code into 5 modular helpers. These transformations were entirely our design decisions to improve code clarity and testability.

Test Documentation (AI-Assisted): We used AI to generate a `tests.md` file documenting all test cases, then reviewed and refined it to ensure accuracy.

7.3 Learning and Value Addition

What AI provided: Standard library usage patterns, common C++ idioms, boilerplate structure, and basic Verilog syntax patterns.

What we provided: Algorithm correctness and completeness, meaningful variable and function names, code organization and modularity, comprehensive error handling, test case design and validation, edge case identification, Verilog optimization strategies, hardware design insights, and integration of minimization with code generation.

We can explain every line of code, including AI-generated portions, and have tested all functionality thoroughly. The AI was a productivity tool for routine coding tasks, not a substitute for algorithmic understanding or hardware design knowledge.

8 Conclusion

This project successfully implements the Quine-McCluskey algorithm with a clean, modular architecture supporting up to 20 variables and includes Verilog HDL code generation as a bonus feature. Our design emphasizes code clarity through focused functions, comprehensive validation, and separation of concerns.

Key achievements include robust input handling for both minterm and maxterm notation, efficient prime implicant generation using bit manipulation, correct identification of essential PIs, bounded exhaustive search for minimal covers with support for multiple solutions, and automatic generation of synthesizable Verilog using primitive gates.

The Verilog generation feature demonstrates the practical application of Boolean minimization in hardware design workflows. By producing structural HDL code, our tool bridges the gap between theoretical minimization and actual hardware implementation, providing value for both educational and practical design scenarios.

The implementation demonstrates strong software engineering practices: modular design, reusable components, comprehensive testing, and clear documentation. Through this project, we gained deep understanding of Boolean minimization algorithms, practical experience in C++ system design, and insights into hardware description language generation and digital design workflows.