Quine-McCluskey Logic Minimization Program

Project Report

Team Members:

Moaz Allam - 900231984
Karim El Henawy - 900231975
Zeyad Maher - 900232155

Digital Design 1

Contents

1 Program Design

1.1 Overview

This project implements the Quine-McCluskey algorithm, a systematic tabular method for minimizing Boolean functions. The program accepts Boolean functions with up to 20 variables in minterm or maxterm notation, handles don't-care conditions, generates all prime implicants, identifies essential prime implicants, and produces minimal Boolean expressions along with Verilog HDL code.

1.2 Architecture

The program follows a modular design with clear separation of concerns:

Core Algorithm Pipeline:

Input Parsing → Prime Implicant Generation → PI Chart Construction →
Essential PI Detection → Minimal Cover Selection → Expression Generation →
Verilog Code Generation

1.3 Data Structures

1.3.1 Internal Representation

- **Binary Strings with Don't-Cares:** Terms represented as strings containing '0', '1', and '-' (don't care). Example: `"10-1"` represents AB'D
- **Prime Implicant Tuple:** `(binary_string, minterm_set)`. Example: `("0-1", {1, 3})`
- **PI Chart Dictionary:** Maps each minterm to list of covering PI indices

1.3.2 Algorithm-Specific Data Structures

- **Grouping Dictionary:** Groups terms by number of 1's for efficient combination
- **Sets for Tracking:** `used_terms`, `seen_binaries`, `essential_indices`

1.4 Key Algorithms

1.4.1 Prime Implicant Generation

Algorithm: Iterative pairwise combination

Listing 1: Prime Implicant Generation Pseudocode

```
Input: minterms, don't-cares, number of variables
Output: List of all prime implicants

1. Convert all terms to binary strings
2. Create initial term list with singleton minterm sets
3. WHILE terms can still be combined:
   a. Group terms by number of ones
   b. Compare terms in adjacent groups (differing by 1 bit)
   c. If two terms differ by exactly one bit position:
      - Combine them (replace different bit with '-')
      - Mark both terms as "used"
      - Add combined term to next iteration
   d. Add all "unused" terms to prime implicants list
4. RETURN prime implicants sorted by first minterm
```

Time Complexity: $O(3^n)$ worst case, typically much better with early pruning. **Space Complexity:** $O(m \cdot 2^n)$

1.4.2 Essential Prime Implicant Detection

Algorithm: Single-coverage detection. For each minterm in PI chart, if covered by exactly ONE PI, mark that PI as essential. **Time Complexity:** $O(m)$

1.4.3 Minimal Cover Finding

Algorithm: Backtracking with pruning

Listing 2: Minimal Cover Selection Pseudocode

```
BACKTRACK(current_solution, current_covered, depth):
    IF all minterms covered:
        IF solution size <= best solution:
            Add to solutions list
        RETURN
    IF current solution size >= best solution size:
        RETURN (prune)
    IF depth > max_depth:
        RETURN (prevent infinite loops)
    Pick an uncovered minterm
    FOR each PI that covers this minterm:
        Add PI to solution
        Update covered minterms
        BACKTRACK recursively
        Remove PI (backtrack)
```

Optimizations: Depth limiting (max_depth=15), best-first pruning, greedy minterm selection. **Time Complexity:** $O(p^k)$

1.5 Key Functions

2 Challenges and Solutions

2.1 Exponential Complexity

Problem: For large functions (15+ variables), prime implicants can grow exponentially. **Solution:** Depth-limited backtracking (max_depth=15), early pruning based on current best solution, greedy minterm selection.

Function	Purpose
`decimal_to_binary()`	Converts decimal to fixed-width binary
`can_combine()`	Checks if terms differ by one bit
`combine_terms()`	Combines terms with don't-care
`generate_prime_implicants()`	Generates all PIs
`build_pi_chart()`	Constructs PI coverage chart
`find_essential_pis()`	Identifies essential PIs
`find_minimal_covers()`	Finds minimal PI combinations
`term_to_expression()`	Converts to Boolean algebra
`term_to_verilog()`	Converts to Verilog syntax

2.2 Don't-Care Handling

Problem: Don't-cares should be used for minimization but not appear as required coverage. **Solution:** Include don't-cares in prime implicant generation, exclude from PI chart, allowing them to help create larger implicants without requiring coverage.

2.3 Multiple Minimal Solutions

Problem: Many Boolean functions have multiple equally minimal covers. **Solution:** Backtracking algorithm finds ALL solutions up to minimal size, clears solution list when better solution found, presents all minimal solutions.

3 Testing

3.0.1 Test Case Organization

We have two main directories:

- `./simple_test_cases/`: Contains 10 foundational test cases covering basic functionality
- `./complex_test_cases/`: Contains 10 advanced test cases covering edge cases and complex scenarios

Each test case is stored in a separate file (`testN.txt`) following the standardized input format. Expected outputs for both directories are documented in text files to enable automated verification of program correctness.

3.1 Simple Test Cases (Tests 1-10)

The simple test cases validate core algorithm functionality and were verified using the online Quine-McCluskey minimization tool at `https://geeekyboy.github.io/Quine-McCluskey-Solver/`.

3.1.1 Coverage of Core Functionality

Key Features Tested:

- Basic prime implicant generation
- Essential prime implicant identification

Test	Variables	Purpose
1	3	Basic minterm minimization
2	4	Minterm minimization with don't-cares
3	2	Minimal variable count (2 variables)
4	4	All odd minterms (pattern recognition)
5	3	Maxterm notation conversion
6	4	Complex don't-care integration
7	5	Larger variable space (5 variables)
8	3	Nearly complete function (7/8 minterms)
9	4	Symmetric function with don't-cares
10	4	Mixed coverage with strategic don't-cares

Table 1: Simple Test Case Coverage

- Don't-care condition handling
- Maxterm to minterm conversion
- Multiple variable counts (2-5 variables)
- Various minterm densities

3.2 Complex Test Cases (Tests 11-20)

Complex test cases were designed to stress-test the algorithm with challenging scenarios and edge cases. These test cases and their expected outputs were generated and verified using Claude AI (Claude Sonnet 4.5) to ensure correctness for scenarios beyond simple manual verification.

3.2.1 Advanced Scenario Coverage

Test	Variables	Scenario
11	6	Sparse minterm distribution with many don't-cares
12	6	Irregular minterm patterns
13	6	Extremely sparse function (5 minterms, 64 possible)
14	6	Single minterm (edge case: $F = m_{63}$)
15	7	Structured patterns in 7-variable space
16	7	Arithmetic progression minterms
17	7	Contiguous minterm blocks
18	7	Irregular minterm patterns
19	7	All don't-cares (degenerate case: $F = d$)
20	6	Complex essential PI interaction

Table 2: Complex Test Case Coverage

3.3 Verification

Simple Test Cases:

- Hand verification of basic cases (Tests 1-3)

- Cross-validation with online QM tool for all 10 tests

Complex Test Cases:

- AI-assisted generation of expected outputs using Claude Sonnet 4.5
- Verification of prime implicant completeness
- Validation of essential PI detection

3.4 Test Results

All 20 test cases (10 simple + 10 complex) pass successfully with 100% correctness rate. The test suite validates:

4 AI Usage Documentation

4.1 AI Tools Used

Primary Tool: Claude AI (Claude Sonnet 4.5) by Anthropic
Usage Period: Throughout project development and testing phases
Purpose: Code assistance, debugging, and test case generation

4.2 Areas of AI Assistance

4.2.1 Code Development and Debugging (30% of codebase)

AI assistance was utilized for the following utility and I/O functions:

1. Input/Output Functions:

- `parse_input_file()`: File reading and parsing logic
- `parse_testcase_input()`: Flexible test case number parsing (ranges, lists)
- `print_results()`: Formatted output generation
- `process_testcase()`: Test case execution orchestration

Prompts Used:

Listing 3: Example Prompt for File Parsing

```
"Write a Python function to parse a Quine-McCluskey input file with the
following format:
- Line 1: number of variables
- Line 2: minterms (m0,m1,m3) or maxterms (M0,M1,M3)
- Line 3: don't cares (d0,d1,d4)

The function should handle errors, validate ranges, and convert maxterms
to minterms if needed."
```

Verification Process:

- Reviewed generated code line-by-line to ensure understanding
- Modified variable names and comments for consistency with project style
- Tested with invalid inputs (out-of-range terms, malformed files)
- Verified against 20 test cases

2. Debugging Assistance:
AI was used to identify and fix specific bugs:

- **Issue:** Duplicate prime implicants appearing in results
- **Prompt:** "My QM algorithm generates duplicate prime implicants. Here's the code for `generate_prime_implicants()`. How can I prevent duplicates?"
- **Solution:** Implemented `seen_binaries` set to track generated patterns
- **Verification:** Tested on Test 11 (which previously produced 8 duplicates)

4.2.2 Test Case Generation and Verification

Complex Test Case Creation:

Claude AI was used to generate and verify the 10 complex test cases (Tests 11-20) that exceed manual verification capabilities.

Process:

1. **Test Case Design:** Specified desired characteristics (e.g., "sparse 6-variable function with many don't-cares")
2. **Expected Output Generation:** AI computed prime implicants, essential PIs, and minimal expressions
3. **Cross-Verification:** Ran test cases through our implementation and compared outputs
4. **Discrepancy Resolution:** For any differences, manually verified correctness or identified bugs

Example Prompt:

Listing 4: Test Case Generation Prompt

```
"For a 6-variable Boolean function with minterms m0,m1,m3,m5,m7,m8,m10,
m14,m15 and don't-cares d4,d6,d12, compute:
1. All prime implicants
2. Essential prime implicants
3. Minimal Boolean expression
Show step-by-step Quine-McCluskey reduction."
```

Verification Strategy:

- AI-generated outputs saved as expected results in text files

- Program output compared against expected results
- For Test 14 (single minterm), manually verified: $F = ABCDEF$
- For Test 19 (all don't-cares), verified empty result: $F = 0$
- Selected complex tests (15, 16, 17) manually spot-checked for correctness

5 Build and Usage Instructions

5.1 System Requirements

Python 3.7+, any OS (Windows/macOS/Linux), no external dependencies (uses only Python standard library), optional pytest for test suites.

5.2 Installation and Running

```
# Run single test
python QM.py
> 1

# Run multiple tests
python QM.py
> 1 3 5 7

# Run range of tests
python QM.py
> 1-10

# Run mixed specification
python QM.py
> 1 3-5 8-10
```

5.3 Input File Format

Files located in `simple_test_cases/` and `complex_test_cases/` directories.

Format: `<number_of_variables>`, `<minterms>`, `<dont_cares>`

Example:

```
6
m0,m1,m3,m5,m7,m8,m10,m14,m15
d4,d6,d12,d20
```

Maxterm Example: `M3,M4,M6` (program auto-converts to minterms)

5.4 Output

Console: Prime implicants, essential PIs, minimized expressions. **Verilog File:** `boolean_function_<N>.v` synthesizable HDL code.

6 Known Issues and Limitations

6.1 Performance Limitations

Very large functions (15+ variables with dense minterms) may experience slower performance. Depth limiting (max_depth=15) prevents infinite loops. Most practical cases (up to 10 variables) complete in ¡1 second.

6.2 Minimal Cover Selection

The current implementation uses backtracking for minimal cover selection. Future improvement could implement Petrick's method or branch-and-bound optimization for more efficient minimal cover selection in complex cases.

7 Team Contributions

Moaz Allam: Core algorithm implementation (prime implicant generation, term combination), data structure design, performance optimization, algorithm complexity analysis, core debugging of QM logic.

Karim El Henawy: Minimal cover selection (backtracking algorithm), essential PI detection, Verilog code generation, integration testing framework, algorithm optimization.

Zeyad Maher: Input/output handling and file parsing (with AI assistance), expression formatting (Boolean algebra and Verilog), comprehensive testing infrastructure (simple and complex test cases), test case verification using online tools and AI, project documentation.

Collaboration: All members participated in algorithm design, code reviews, debugging, testing, AI-generated code verification, and documentation. The core algorithmic components were developed independently without AI assistance, with AI used primarily for I/O utilities and test validation.

8 Conclusion

This project successfully implements a complete Quine-McCluskey logic minimizer with full QM algorithm, essential PI detection, don't-care support, maxterm conversion, and Verilog HDL generation supporting up to 20 variables. Quality assurance includes comprehensive two-tier testing with 20 test cases (10 simple + 10 complex) achieving 100% pass rate and truth table verification proving correctness.

AI tools were used responsibly for approximately 30% of the codebase (primarily I/O and test validation), with the core algorithmic implementation developed independently by the team. All AI-generated code was thoroughly reviewed, understood, tested, and integrated following best practices.

References

1. Quine, W. V. (1952). "The Problem of Simplifying Truth Functions". *American Mathematical Monthly.*

2. McCluskey, E. J. (1956). "Minimization of Boolean Functions". *Bell System Technical Journal.*

3. Morris Mano, M. & Ciletti, M. D. (2013). *Digital Design* (5th Edition). Pearson.

4. Roth, C. H. & Kinney, L. L. (2014). *Fundamentals of Logic Design* (7th Edition). Cengage Learning.

5. Quine-McCluskey Online Tool: `https://geeekyboy.github.io/Quine-McCluskey-Solver/`

6. Anthropic. (2024). Claude AI (Claude Sonnet 4.5). Used for code assistance and test validation. `https://www.anthropic.com/claude`