Implementation of Quine-McCluskey Logic Minimization (V.1)

Islam Faisal Ibrahim Hussam Ashraf El-Araby

March 12, 2015

Abstract

In this report, we demonstrate the design and implementation of the Quine-McCluskey logic minimization algorithm in C++. The representation of implicants, minterms are presented. The method of grouping implicants together is also demonstrated. Finally, a short note about future optimizations to the implementation is presented.

1 Introduction

The source code and binary files supported by this report are designed to list all implicants, find the prime implicants, list them and finally list the essential prime implicants. The minterms and the representation of the implicants are encoded using dynamic bit-sets.

2 Memory Representation

2.1 Representation of three states

One of the problems with representing an implicant using minterm is the presence of the arbitrary dashes (-) since a bit can only represent two states. This was solved using two bits to represent three states. To describe a minterm, you have two bit strings x_1 and x_2(Of course, in C++, we use 0-indexation). Let x^i be the corresponding state of the x^i location. x^i is determined according to the following rule(any following i indexation is 1-indexation where i is the right most bit):

1. $x_1^i = 0 \wedge x_2^i = 0 \leftrightarrow x^i = 0$
2. $(x_1^i = 1 \wedge x_2^i = 0) \leftrightarrow x^i = 1$
3. $(x_1^i = 1 \wedge x_2^i = 1) \leftrightarrow x^i = -$
4. $x_1^i = 1 \rightarrow (x^i = 1 \vee x^i = -)$

Although this implementation involves redundancy, we adopted it because of its simplicity. The option brings back also some advantages such as:

1. $Rank(x) = \sum_i x_1^i$ is what we define to be the *rank* which is used to group the implicants and sort them in the implicant table of this tabulation method.

2. If we let the cost of an implicant be the sum of all input literals (symbols and negations) and we have a circuit to design of n inputs, then the following, easily computed, expression is the cost of this implicant since they represent symbols omitted from the final expression
$Cost(x) = n - \sum_i x_2^i$

3. Leading zeros starting at x_1^i will result immediately in leading zeros starting at x_2^i. We omit any leading zeros from our bitset in the implicant representation.

Additionally, we used the Boost dynamic bitset which allows efficient representation of bitsets and omitted leading zeros from the implicant representation.

2.2 Representation of implicant lists

We chose to represent an implicant using a C++ map of a dynamic bitset and an implicant class. The implicant part is to show the description(representation) of its state-string and the bitset part to show what minterms and do not care's it is covering. However, during the tabulation algorithm, we embed this as a value type to a map of integeral key type which is the rank. Therefore, the picture is that you have a map that does not allow duplication of grouped implicants. Each group contains pairs of an implicant and the minterms it covers without duplication. Grouping is based upon the rank.

3 Algorithm Implementation

3.1 Joining implicants

Because of the choices indicated in the previous section, we were able to find a compact method of how to decide whether two implicants can be joined and how to join them.

3.1.1 Join checking

If we have two implicants x and y and we need to check if they can be joined. In order for the two implicants to be combined, they should satisfy these conditions:

1. x and y should have the same number of inputs.

2. $max(x_1 \oplus y_1, x_2 \oplus y_2) = 1$

3.1.2 Joining implicants

Once we checked they can be joined, we perform the joining process as follows:

1. Extend the bitsets by adding leading zeros to be of same size.

2. $z_1 = x_1 \vee (x_1 \oplus y_1) \vee (x_2 \oplus y_2)$
3. $z_2 = x_2 \vee (x_1 \oplus y_1) \vee (x_2 \oplus y_2)$

3.2 Finding essential prime implicants

After generating the prime implicants table, we generate the essential ones. Firstly, we generate and print an adjacency matrix representing the minterms(without the do not cares) and what prime implicants it is covered by. During this generation we mark the ones with only one prime implicant. Their implicants are then printed as the essential prime implicants.

4 Testing and Results

The code was tested against several datasets to make sure it works properly. The program succeeded in finding the list of prime implicants and deduced the correct essential prime implicants. However, a limitation was found and we provide the solution to be implemented in future work. When providing a large input file (for example 16 input and 65536 minterms) to the program, it runs out of memory because it exhausts its heap.

5 Limitations and Future Work

5.1 Serialization of objects

Due to the limitation of running out of memory discovered when dealing with large files, we want to serialize our objects to the hard disk and de-serialize them when needed instead of keeping all objects occupying space in the stack and heap.

5.2 Threading

During the joining process of implicants among groups, the outcome of joining two groups is independent of joining any two other groups. This fact induces the idea of multi-threading which may be helpful in reducing the time required to finish the algorithm.

5.3 Generating SoP

After generating the essential prime implicants, we want the program to continue finding the SoP after observing row and column dominance which shall be easy using the adjacency matrix generated as indicated above. Additionally, we want to embed the branching method when no dominance can be found to guarantee that we found the minimum.

6 Software Information

Information about how to compile the software, how to invoke it on a file and some other notes are included in the readMe.txt file.