Digital Circuit Optimization and Manipulation

Markus Gasser
TU Wien
markus.gasser@tuwien.ac.at

Omid Mirmotahari
Universitetet i Oslo
omidmi@ifi.uio.no

August 21, 2016

Abstract

This paper explains the optimization process of Boolean expressions through K-maps.

I. Two-Level Circuit Optimization

Although the truth table is a unique representation of a logic function an algebraic expression for it is not unique. As we have already discussed each can be represented by standardized forms such as sum of minterms or product of maxterm but also simplified into sum of products or product of sums.

The *K-map* or *Karnaugh map* offers an easy way to simplify algebraic expression to an optimal sum of products or product of sums form.

WX \ YZ	00	01	11	10
00	0	1	3	2
01	4	5	7	6
11	12	13	15	14
10	8	9	11	10

Figure 1: *K-map structure for 4 variables*

i. Cost Criteria

To define the optimal solution for our expressions we first have to define a cost criteria to then be able to measure the optimality. Two possibilities are the *literal cost* and the *gate-input cost*.

The literal cost is the number of literal appearances in the Boolean expression. This number is easy to determine but is not a good indication of the complexity of the resulting logic circuit. Gate-input cost on the other hand is defined as the number of inputs to the gates in the implementation through gate logic corresponding to the equation. This is a very good indication for complexity of the circuit since it's directly proportional to the number of transistors and wires used in the realization.

It is however important to note, that independent of the used cost criteria the solution is not necessarily unique, meaning that there can be more than one Boolean expression with identical lowest cost.

ii. Map Structures

Each cell in a K-map corresponds to one minterm in the function's Boolean expression. A K-map for 4 variables, resulting in $2^4 = 16$ cells is depicted in figure 1. The cells are arranged according to a Gray code, resulting in the fact that minterms that differ in exactly one value of a variable are share the same edge in the map. Two terms that fulfill this fact are called *adjacent*. Additionally in a 3 variable K-map The terms of cells in the left and

right border of the map are adjacent (building a cylinder). In a 3 variable K-map the terms of cells in the left and right border and in the bottom and top border are adjacent (resulting in a torus).

When placing a the minterms of a functions onto the map the decimal indexes of the minterms written inside the cells or the binary variable representations along the top and the left of figure 1 can be used as guidance.

iii. Optimization Process

The whole process of optimizing a Boolean expression through a K-map can be executed in 4 steps.

1. Filling the values (1 or 0) into the cells of the minterms into the cells of the K-map.
2. Identifying *rectangles* of groups of 1s in the map. These rectangles have to contain number of cells that are powers of 2, each 1 on the map needs to be inside of at least one rectangle and each rectangle should be constructed as large as possible. As mentioned previously adjacent cells are also placed on opposite borders of the map which can result in rectangles running over the borders.
3. Eliminating all unneeded rectangles.
4. Reading off the sum-of-products expression, determining the corresponding product terms for the required rectangles in the map.

II. Map Manipulation

To define the optimization process explained in the last section more systematic a new terms need to be defined.

i. Essential Prime Implicants

The new terms introduced are:

- An *implicant* is a product of minterms in a sum of products expression of a function. Equivalently the function has the value 1 for all the minterms of the product term. In the map representation all rectangles of 1s correspond to implicants.
- A *prime implicant* is a implicant that is not an implicant anymore if any literal is removed from it. In the map representation the set of prime implicants of a n-variable function is constructed by rectangles made up of 2^m squares ($m = 0, 1, \ldots, n$) with rectangles containing as many squares as possible.
- A prime implicant is *essential* if a minterm is not included in any other implicant. Otherwise the implicant is said to be *nonessential*.

We can now formulate a more systematic optimization procedure:

1. Prime implicants can be obtained from the map representation of a function by building all possible maximum rectangles of size $2^m, \quad m = 0, 1, \ldots, n$ squares.
2. Essential prime implicants are the found by searching for squares that are only contained in one implicant.
3. The optimized Boolean expression of the function is finally obtained by the logical sum of all essential prime implicants in addition to other prime implicants that contain remaining minterms not yet included in the expression.

ii. Nonessential Prime Implicants

To simplify the expression format further a selection rule for the needed nonessential prime implicants can be used. It states that the overlap among primes used in the expression should be minimized as much as possible.

iii. Product-of-Sums Optimization

The previous sections explained the procedure to obtained the sum-of-products expression for a function. To obtain the product-of-sums one, we mark the empty squares in the map by 0,

combine them into rectangles and obtain in an analog way as previously the complement of the function $\overline{F}$. Taking the complement results in the function F expressed as product of sums.

iv. Don't-Care Conditions

In case of a *incompletely specified function* where there are unspecified output for certain input combinations (e.g. BCD code) the Boolean expression can be further optimized. The unspecified minterms are called *don't-care conditions*, are marked with "X" in the K-map and can be used in the optimization process when building rectangles each of these squares may but does not have to be included in one or more rectangles.

III. Summary

In this paper we explained the concepts of K-maps and how they can be used to optimize Boolean expressions to their product of sums and sum of product forms. We detailed a more systematic approach to optimization and explain the concept of don't care conditions.

References

[Kime and Mano, 2004] Kime, Charles R and Mano, M Morris (2004). Logic and computer design fundamentals. *Pearson Education*.