Chapter 5

Easily Testable Circuits for MAC Units

This chapter presents an easily testable realization of multiply and accumulate (MAC) units using Positive Polarity Reed-Muller expressions (PPRMs). An exhaustive branching algorithm to implement any logic function in RM form using Reed-Muller Universal Logic Modules (RM-ULMs) is presented. This approach reduces the delay and hardware requirement for synthesizing logic functions. A Genetic Algorithm (GA) based exhaustive branching approach for combinational logic synthesis using ULMs is also presented. The search algorithm finds an implementation that uses only a particular ULM with minimum number of modules and levels for any function.
5.1 Reed-Muller Expressions

Every Boolean function can be expressed in the form of Reed-Muller (RM) expression using AND and XOR operators. The AND-XOR algebra forms a complete Boolean algebra. This representation has various advantages such as ease of complementing and testing [Reddy, 1972]. The RM representations may be shorter with a reduced number of product terms leading to smaller circuits on-chip over the conventional descriptions [Harking, 1990]. Several papers have been published discussing the design and minimization techniques for RM logic, derivation of various polarities, as well as conversion between RM and Boolean forms [Sasao, 1993a], [Miller and Thomson, 1995], [Varma and Trachtenberg, 1991].

The following are the basic theorems of the XOR operator. The XOR logic has double duality property which can be verified by applying inversion theorem and transposition theorem. This makes the XOR logic a very flexible system of logic.

**Basic theorems:**

\[ x \oplus x = 0 \]
\[ x \oplus x' = 1 \]
\[ x \oplus 0 = x \]
\[ x \oplus 1 = x' \] (5.1)

\[ (x \oplus y)' = x' \oplus y = x \oplus y' \]
\[ x' \oplus y' = x \oplus y \] (5.2)

**Inversion theorems:**

\[ x \oplus y = y \oplus x \] (5.3)

**Commutative law:**

\[ (x \oplus y) \oplus z = x \oplus (y \oplus z) \] (5.4)

**Associative law:**

\[ x(y \oplus z) = xy \oplus xz \] (5.5)

**Distributive law:**

\[ (x \oplus y) \oplus z = x \oplus (y \oplus z) \]

**Disjunction theorem:** If \( f = g \oplus h \) and \( gh = 0 \), then \( f = g + h \) (5.6)
Transposition theorem: If $f = g \oplus h$, then $g = f \oplus h$ and $h = g \oplus f$ \hfill (5.7)

A function given in AND-OR form can be expressed in AND-XOR form using disjunction theorem. The function in terms of OR operators is expanded to minterm form initially, so that all the terms are disjoint. Then the OR operators can be directly replaced with XOR operators. As an example, the canonical form of a three variable function in AND-XOR logic is given in (5.8).

\[
f(a,b,c) = \alpha_0 a'b'c' \oplus \alpha_1 a'b c \oplus \alpha_2 a'bc' \oplus \alpha_3 a'bc \oplus \alpha_4 ab'c' \\
\quad \oplus \alpha_5 ab'c \oplus \alpha_6 abc' \oplus \alpha_7 abc
\] \hfill (5.8)

where $\alpha_i = 1$ or 0 depending on $i^{th}$ minterm is present or not. An alternate canonical form is obtained as in (5.9) by expanding all complemented variables using the basic theorem $x' = x \oplus 1$.

\[
f(a,b,c) = \beta_0 \oplus \beta_1 a \oplus \beta_2 b \oplus \beta_3 c \oplus \beta_4 ab \oplus \beta_5 ac \oplus \beta_6 bc \oplus \beta_7 abc
\] \hfill (5.9)

where $\beta_i = 1$ or 0 depending on $i^{th}$ term is present or not. This can be generalized for any arbitrary number of variables. There are various classes of AND-XOR expressions as defined below [Sasao, 1993b] [Sasao, 1997].

**Positive Polarity Reed-Muller Expression (PPRM)**

When an arbitrary $n$-variable function is represented as in (5.10), it is called a positive polarity Reed-Muller (PPRM) expression.

\[
f(x_1, x_2, ..., x_n) = \beta_0 \oplus \beta_1 x_1 \oplus \beta_2 x_2 \oplus .... \oplus \beta_n x_n \\
\quad \oplus \beta_{12} x_1 x_2 \oplus \beta_{13} x_1 x_3 \oplus .... \oplus \beta_{n-1n} x_{n-1} x_n \oplus \\
\quad \cdots \cdots \\
\quad \oplus \beta_{12...n} x_1 x_2 x_3 ... x_n
\] \hfill (5.10)

For a given function the coefficients $\beta$'s are unique. Hence PPRM is a canonical representation. All the literals are positive for PPRM.
Fixed Polarity Reed-Muller Expression (FPRM)

The fixed polarity Reed-Muller (FPRM) expression allows only one polarity for each input variable. Each variable $x_i$ in (5.9) can choose either of positive ($x_i$) or negative ($\overline{x_i}$) polarity. Thus, there are $2^n$ different sets of polarities for an $n$-variable function. There exists a unique set of coefficients $(\beta_0, \beta_1, ..., \beta_{12...n})$, for a given function and for a given set of polarity. Thus FPRM is a canonical representation.

Generalized Reed-Muller Expression (GRM)

There is no restriction on the allowed polarities of input variables in generalized Reed-Muller (GRM) expressions. The variables can take both positive and negative polarities, but it does not allow the same set of variables in more than one product term. Each of the $n2^{n-1}$ literal in (5.9) can take two polarities. So there are $2^{n2^{n-1}}$ different sets of polarities for an $n$-variable function. As there exists a unique set of coefficients $(\beta_0, \beta_1, ..., \beta_{12...n})$ for a given set of polarities, GRM forms a canonical expression for a logic function.

Exclusive OR Sum-of-Products Expressions (ESOP)

Arbitrary product terms combined by XOR operators is called exclusive OR sum-of-products (ESOP) expressions. It has no restrictions on the allowed polarities of variables or on the allowed product terms. It is the most general form of AND-XOR expression [Kalay et al., 2000]. The different Reed-Muller forms follow the relationship: $\text{PPRM} \subseteq \text{FPRM} \subseteq \text{GRM} \subseteq \text{ESOP}$. 
5.2 Easily Testable Circuits

A Boolean function can be implemented in different circuit design methods. Each realization may require different number of test vectors. The logic function implementation in AND-XOR logic plays an important role in design-for-testability. Several researchers have explored various design techniques and testability issues of AND-XOR logic. Reddy showed that only $n + 4$ test vectors are needed to detect all single stuck-at faults in a PPRM network, where $n$ is the number of input variables [Reddy, 1972]. Later, Saluja and Reddy extended the results for detection of multiple stuck-at faults [Saluja and Reddy, 1975]. But the realization uses a cascaded XOR structure where the propagation delay is large. Pradhan introduced a test method for multiple fault detection in ESOP circuits [Pradhan, 1978]. The testing of a multilevel two-input XOR tree is done by a test set consisting of four vectors irrespective of the depth of the tree [Rahaman et al., 2004]. The XOR tree propagates any single fault to the output [Dubrova and Muzio, 1996]. This property minimizes the number of test vectors needed for fault detection and simplifies the test pattern generation for RM circuits. Sasao presented a GRM implementation to detect multiple stuck-at faults where a XOR tree structure is used to reduce circuit propagation delay [Sasao, 1997]. Kalay et al. presented an ESOP implementation with a minimal universal test set of size $n + 6$ to detect all possible single stuck-at faults [Kalay et al., 2000]. A bit parallel multiplier over Galois field with a constant test set of length 8 to detect all the single stuck-at Faults is presented [Rahaman et al., 2007].

The adders and multipliers in MAC units of a FIR filter can be implemented in RM form by taking the easiness of testing into consideration.
The basic element of a MAC unit is a full adder whose sum and carry outputs are expressed in RM form as in (5.11).

\[ s = a \oplus b \oplus c_{in}, \quad \text{and} \]
\[ c_{out} = ab \oplus bc \oplus ac_{in} \quad (5.11) \]

The present decimation filters designed for dual-mode operation has MAC units using adders and ROMs as the basic elements. The AND-XOR realizations of adders are obtained with full adders implemented in RM form. The simulation results using the test tool ATALANTA that show the number of test vectors and the test patterns needed to test the full adder and the adders of various sizes implemented in both AND-OR and AND-XOR forms are given in Section 6.9.

### 5.3 Combinational Logic Synthesis using Reed-Muller Universal Logic Modules

RM functions can be implemented using discrete components or more conveniently by Reed-Muller universal logic modules (RM-ULMs). An RM-ULM is a device with \( c \)-control inputs, \( 2^c \) data inputs and a single output \( f(c) \) and is designated as RM-ULM(\( c \)). The behaviour of this module is described as in (5.12):

\[ f(c) = b_0 \oplus b_1 x_1 \oplus b_2 x_2 \oplus b_3 x_2 x_1 \oplus \ldots \oplus b_{2^c-1} x_c x_{c-1} \ldots x_1 \quad (5.12) \]

\[ = \bigoplus_{i=0}^{2^c-1} b_i P_i \]

where \( b_i = 0 \) or 1, and the product term (or piterm) \( P_i \) is,

\[ P_i = x_{i_1} x_{i_{i-1}} \ldots x_{i_1} \quad \text{where} \quad i = \sum_{j=0}^{c-1} 2^j x_j \quad (5.13) \]

\( x_{i_k} \) will be present in \( P_i \) if the \( k^{th} \) bit of binary representation for \( i \) is 1. The logic symbol for RM-ULM(\( c \)) is shown in Figure 5.1.
VLSI implementations using only one type of modular building blocks can decrease system design and manufacturing cost. Circuit delay and cost can be reduced by using RM-ULMs connected in tree structure for functions in RM form. A tree network is very suitable for VLSI realization because of the uniform interconnection structure and the repeated use of identical modules.

The use of RM-ULM for realization of logic functions has already been explored by researchers. Programmed algorithms have been developed for optimization of number of modules at sub-system level in a tree network [Xu et al., 1993], [Almaini et al. 1992]. The algorithm looks for possible cascade networks, and if it is not found a tree structure is implemented. Tan and Chia presented an alternate algorithm which performs similar optimization of fixed polarity Reed-Muller expansions (FPRM) with a reduced computation time [Tan and Chia, 1996]. The above algorithms do not explore all the possible branching options of the tree structure and hence the delay of the circuit synthesized may not be minimal.

In this research, further delay reduction is achieved by using a novel tree-structured exhaustive branching network using RM-ULM(1) for implementing a logic function given in positive polarity Reed-Muller (PPRM)
form. A logic function with \( n \)-variables can be implemented using \( 2^n - 1 \) RM-ULM(1)s in \( n \)-levels by standard implementation. Any implementation using less than \( 2^n - 1 \) number of modules and / or lesser number of levels can be considered as an improvement in cost and / or speed.

5.3.1 N-ary Exhaustive Branching Technique

For a given number of input variables \( n \), there is a well-defined number of functions, which is equal to \( 2^{2^n} \) [Correia and Reis, 2001]. Standard implementation of a tree network requires \( n \)-levels to implement these functions. Xu presented a programmed algorithm to reduce the complexity of the network in terms of number of modules and levels. In his approach 1's, 0's, \( \hat{x}_i \) (where \( \hat{x}_i \) is a variable \( x_i \) or its complement \( x_i' \), \( 1 \leq i \leq n \)) or functions using any number of variables can be given to any data inputs of the RM-ULM. But the control inputs accept variables only. In this research, the performance is further improved by an exhaustive branching technique with \( \hat{x}_i \) or functions of two variables at control input. Since \( x_i' \) or functions are also given to control input, the utilization of all branching options are made possible. This decreases the number of levels, and hence the delay is reduced for any logic function implementation using RM-ULMs.

The first level (output stage) will have a single RM-ULM, the second level will have a maximum of \( (2^c + c) \) RM-ULMs, where \( c \) is the number of control inputs (\( c = 1 \) in this case) and so on. In general, the maximum number of RM-ULMs in a level can be expressed as \( (2^c + c)^{L-1} \) where \( L \) indicates the number of levels. The maximum number of RM-ULMs, \( N \) in the complete network having \( L \) levels is given in (5.14).
\[ N = \sum_{x=1}^{L} (2^x + c)^{x-1} \]  

(5.14)

A network with 1-level can realize functions up to 3 variables, since there are 3 inputs. By connecting \( x_i \) to the control input, the remaining \( x_j \) variables \((j \neq i)\) or constants \((0\ or\ 1)\) can be connected to each of the 2 data input lines. So there are 6 possible values for each data input line, resulting in \(6^2\) functions. Selecting 1 variable as control input from the total of 3 variables and its complements, can take \(6C_1\) combinations. Out of 62 distinct functions implemented at level 1, 24 are 3 variable functions which require 3 levels in standard implementation.

Level 2 allows the implementation of functions having maximum of 9 variables, using 3 control lines and 6 data lines. Selecting 3 variables from the total of 9 variables and its complements, results in \(18C_3\times3!\) combinations. The remaining 6 variables and its complements or constants \((0\ or\ 1)\) at 6 data lines give rise to \(14^6\) distinct functions with one combination at control input. In the tree structure given by Xu, at level 2 maximum number of variables possible is only 7, which result in \(10^4\) distinct functions with one combination at control input. The exhaustive branching approach increases the number of variables and functions that can be implemented in level 2. As the number of levels increases this difference becomes more and more significant, and more delay reduction can be achieved for functions with large number of variables. In general with \(L\) levels, the number of functions that can be implemented using RM-ULMs in the exhaustive branching method is \(2(y+1)^y\) for one combination at control inputs. The number of combinations possible at control inputs is \(\{2z^L Cz^{L-1}\}\times\{z^{L-1}\}\) where \(y = (z^L - z^{L-1})\) and \(z = (2^c + c)\). Maximum number of variables at level \(L\) is
$n_{max} = 2^L$. For a given function if there are $n$ dependent variables, the levels $L$ required for implementation in this approach is given as $\left\lfloor \log_{(2^c + c)} n \right\rfloor \leq L \leq (n - 1)$, whereas in the tree structure $L$ can be in the range $\left\lfloor \log_{(2^c)} n \right\rfloor \leq L \leq (n - 1)$. This clearly demonstrates a reduction in delay attained by the exhaustive branching technique over the implementation using tree structure.

5.3.2 Exhaustive Branching Algorithm

Behavior of an RM-ULM(1) can be expressed as $F_j \oplus F_s F_k$, where $F_s$, $F_j$ and $F_k$ are functions of $t$ variables ($1 \leq t \leq n$). The number of variables of $F_s$, $F_j$ and $F_k$ varies according to the complexity of the function to be realized. The maximum number of variables in $F_s$, $F_j$ or $F_k$ determines the delay of the network. The network terminates when $F_s$, $F_j$ and $F_k$ are 1's, 0's or $\hat{x}_i$ ($1 \leq i \leq n$). If all inputs except one terminate with a variable $\hat{x}_i$ or a logical constant and only one input continues into the next level, a cascade is generated where a single module is used in each level. The new algorithm aims to identify $\hat{x}_j$ or functions of 2 variables at each control input, that eliminate as many branches as possible and reduce the number of levels and modules required for implementation. The algorithm for any function given piterms ($P_i$), is as follows:

**Exhaustive Branching Algorithm:**

*Step 1*: Get the piterms in decimal, and the number of variables, $n$. Set level, $L = 1$, number of modules, $M = 1$.

*Step 2*: List the piterms in $n$-bit binary as a piterm table.

*Step 3*: Check whether any column in the table is all zeros. Eliminate the variable corresponding to that column and get the reduced piterm table.
Step 4: Get the reduced piterm tables for each variable $x_i$ (one table for $x_i = 1$ and another table for $x_i = 0$) and find the $x_i$ for which the reduced piterm tables correspond to constants (0 or 1) or $\hat{x}_j$ ($j \neq i$) by checking the number of ones in each piterm table, $c_i$. If $c_i \leq 1$, terminate.

Step 5: For each $x_i$ check the following conditions:

(i) Number of zeros ≥ number of ones
(ii) For each (1, 0) pair, the remaining bits are constants
(iii) Number of such pairs is equal to 2
(iv) One pair has remaining bits as all zeros and the other has ones in one column only.

Terminate if all the above conditions are satisfied as implementation is obtained with $(1 \oplus x_i)$ at the control input.

Step 6: $L = L + 1$, $M = M + 1$. Get the reduced piterm tables for each variable and find the $x_i$ for which the following conditions are satisfied.

(i) One reduced piterm table corresponds to a constant (0 or 1) or $\hat{x}_j$ ($j \neq i$).
(ii) The other reduced piterm table is a single module implementation by repeating the steps 4 & 5.

Step 7: Get reduced piterm tables for each possible $(1 \oplus x_i)$ (by checking conditions (i) & (ii) of step 5), and find the $(1 \oplus x_i)$ for which the conditions (i) & (ii) of step 6 are satisfied.

Step 8: $M = M + 1$. Get the reduced piterm tables for each variable, and find the $x_i$ for which the reduced piterm tables are single module implementations by repeating the steps 4 & 5.

Step 9: Get reduced piterm tables for each possible $1 \oplus x_i$ (by checking conditions (i) & (ii) of step 5), and find that $1 \oplus x_i$ for which the reduced
piterm tables are single module implementations by repeating the steps 4 & 5.

Step 10: Get the reduced piterm tables for each possible $x_i x_j$, and find the $x_i x_j$ for which the conditions (i) & (ii) of step 6 are satisfied.

Step 11: Get the reduced piterm tables for each possible $(x_i \oplus x_j)$, and find the $(x_i \oplus x_j)$ for which the conditions (i) & (ii) of step 6 are satisfied.

Step 12: $M = M + 1$. Get the reduced piterm tables for each possible $x_i x_j$, and find the $x_i x_j$ for which both reduced piterm tables are single module implementations by repeating the steps 4 & 5.

Step 13: Get the reduced piterm tables for each possible $(x_i \oplus x_j)$, and find the $(x_i \oplus x_j)$ for which both reduced piterm tables are single module implementations by repeating the steps 4 & 5.

The synthesis results obtained for various combinational logic functions with RM-ULMs using the exhaustive branching algorithm are given in Section 6.10.

5.4 Genetic Algorithm based Approach for Combinational Logic Synthesis

An evolutionary approach based on genetic algorithm (GA) is used as the main engine to synthesize logic functions. GA has been widely used as a search technique which mimics the natural process of Evolution and Darwin’s principle of “Survival of the Fittest”. In the last decade the use of GA for the design of digital circuits has led to a novel area of research in evolutionary design called evolvable hardware. Evolvable hardware is capable of realizing optimized circuits beyond those of conventional design of logic circuits.

The use of genetic algorithm for realization of logic functions has already been explored by researchers. John Koza has used genetic algorithm to
design combinational circuits using AND, OR and NOT logic gates; but his emphasis has been on generating functional circuits rather than optimizing them [Koza, 1992]. Coello et al. presented a computer program that automatically generates high quality circuit designs using five possible types of gates (AND, NOT, OR, XOR and WIRE) that reduces the use of gates other than WIRE [Coello et al., 1996]. Miller and Thompson applied GA to minimize FPRM expansions and ESOP expansions [Miller and Thompson, 1995]. Evolutionary algorithms applied for the design of arithmetic circuits is also presented [Fogarty et al., 1998]. Torresen presented an evolutionary method in order to solve complex problems by applying divide and conquer method [Torresen, 1998]. An evolutionary approach to synthesize combinational circuits using different sets of gates of varying complexity was done [Reis and Machado, 2003]. In all these cases different types of gates were used to synthesize the function. However, the use of the different types of gates may not be realistic in VLSI systems design where the emphasis is to reduce the manufacturing cost rather than the number of components used. VLSI implementations using single type of modular building blocks can reduce the system design and manufacturing cost. Aguirre and Coello presented genetic programming approach to synthesize Boolean functions using multiplexers [Aguirre et al., 1999], [Aguirre and Coello, 2004].

5.4.1 Universal Logic Modules (ULMs) for Logic Synthesis

The algorithm uses NAND gate, NOR gate, multiplexer and Reed-Muller module as ULMs for realizing any logic function specified as minterms. Multiplexer implements the function in AND-OR form and the behaviour of a multiplexer with \( n \)-select inputs, \( 2^n \) data inputs and a single output \( F \) is given by (5.15):
\[ F = \sum_{i=0}^{2^n-1} x_i m_i(s) \]  

(5.15)

where \( m_i(s) \) is the \( i^{th} \) minterm of the \( n \)-select variables and \( x_i \) is the \( i^{th} \) data input. Reed-Muller expressions are more advantageous than conventional expressions for XOR intensive applications such as error detection, arithmetic circuits etc. This AND-XOR representation has various advantages such as ease of complementing and testing. It may require lesser number of product terms leading to smaller circuits on-chip than the conventional implementations. RM functions can be implemented using discrete components or more conveniently by RM-ULMs. The behaviour of RM-ULM(c) with \( c \)-control inputs, \( 2^c \) data inputs and a single output \( f(c) \) is described as in (5.12). More specifically, the ULMs considered in this research are 3-input NOR gate, 3-input NAND gate, single control line multiplexer (2:1 MUX) and a single control line Reed-Muller ULM (RM-ULM(1)), each having 3-inputs and 1-output. A tree network is very suitable for VLSI realization because of the uniform interconnection structure and the repeated use of identical modules. Following this line of research, a GA-based evolutionary synthesis of combinational circuits using appropriate ULMs is presented here.

5.4.2 GA based Approach for Logic Synthesis

GA-based synthesis of combinational circuits specified by minterms using universal logic modules is considered here. The implementation is in the form of a tree-structured exhaustive branching network using single type of ULMs. The measure of circuit optimality is defined in terms of total number of ULMs used and the number of levels required. The algorithm searches
for the type of ULM to be used for realizing the circuit with minimum number of modules and levels. Thus the resulting implementation will have a reduced delay and/or power compared to those using other ULMs.

GA is a population based approach in which solution to a problem is encoded in the form of string of characters called chromosome [Goldberg, 1989]. The first aspect of this problem is encoding of solutions. Each circuit is encoded in the form $<input1><input2><input3>$ where ‘input i’ represents the input to each of the 3-input ULM. A chromosome is formed with as many triplets of this kind as needed for realizing a function. The number of bits of the chromosome depends on the total number of input combinations possible for each module. The initial population of circuits is generated at random and the algorithm searches for a solution among them. “Fitness” value of a chromosome tells how “good” the chromosome is. These initial chromosomes are evaluated using a fitness function, and a fitness value $f_i$ is assigned for each chromosome.

The three different genetic operators used are reproduction, crossover and mutation. Reproduction is a process in which individual strings are copied from the old population to the new population according to their fitness function values $f_i$. The reproduction operator may be implemented in algorithmic form in a number of ways such as, Roulette wheel selection, Boltzman selection, Tournament selection, Rank selection etc. This research uses a biased Roulette Wheel with slots sized in proportion to its fitness value. The better the fitness value of the chromosome, the greater the chances that it will be selected. However it is not guaranteed that the fittest member goes to the next generation. For the crossover operator, the strings in the new
population are grouped together into pairs at random. A crossover point is randomly selected, and then a single point crossover is performed among pairs. After a crossover is performed, the resulting solution may fall into a local optimum. Hence some genes of the child chromosome are randomly changed by *Mutation*. It is done by random alteration of the value of a string position. In binary coding, this simply means changing a 1 to a 0 and vice versa. When creating a new population by crossover and mutation, the best chromosome may be lost. Hence Elitism, which is a method for copying the best chromosome to the new population prior to crossover and mutation, increases the performance of GA.

The new algorithm works on the principle of genetic algorithm and is implemented level by level. The steps involved are as follows:

*Step 1*: Get the minterms and number of variables, and convert it to a truth table. Set level $L = 1$, and number of modules $M = 1$.

*Step 2*: Select a suitable encoding for the chromosome and generate an initial random population.

*Step 3*: Select a particular type of ULM. Set the number of iterations to $N$.

*Step 4*: Compute the fitness function for each chromosome. Assign $N = N - 1$.

*Step 5*: If fitness value $f_i = 1$, the objective is fulfilled and terminate. Else, generate an intermediate population using Roulette wheel, and apply crossover and mutation.

*Step 6*: If $N \neq 0$, repeat from step 4, for the same ULM, else repeat from step 3 until all the ULMs are considered.

*Step 7*: Assign $M = M + 1$, and check whether all possible branching options are considered at the current level. If No, repeat from step 2, else assign $L = L + 1$, and search for a solution at the next higher level.
The search at a particular level continues for an optimum number of
generations, and if no solution is found another type of ULM is considered. If
the objective is not attained with any type of ULM, the search moves on to the
next higher modular level. After getting a solution with fitness value 1, further
optimization is done by checking whether there are modules that implement
the same sub-functions in the network. The synthesis results obtained for
various functions using GA based approach are demonstrated in Section 6.11.

5.5 Summary

The Boolean functions implemented in the form of Reed-Muller
expressions provide ease of testability. Adders of different sizes are
implemented in both RM form and conventional AND-OR form. The test
patterns and the fault free responses for detecting single stuck-at faults are
found out using the test tool ATALANTA. The test results show that the test
set size for 100% fault coverage in RM circuit is less than that for AND-OR
circuit as shown in Section 6.9. The MAC units described basically consist of
adders and ROM modules. Hence, easily testable MAC units for a filter
structure can be obtained by implementing adders in RM form.

An exhaustive branching algorithm for the synthesis of RM-ULM
tree network is presented. The delivered network has reduction in delay and
complexity in terms of number of modules, compared to the existing
implementations. The logic synthesis results obtained for various functions
using the exhaustive branched algorithm are given in Section 6.10. By
suitable selection of variables, its complements or functions as control inputs,
the number of modules and delay are reduced. Theoretically, the algorithm
can handle any number of variables for any completely specified logic
function. The computation time is not always directly proportional to the number of variables, but this increases with the complexity of the function to be realized. Since the topology of the delivered network is that of a tree, VLSI implementation of this network requires very few extra works in routing algorithms to redesign or for circuit layout.

A genetic algorithm-based logic synthesis using exhaustive branched ULM network is also presented. The algorithm searches and finds an appropriate ULM so that the evolved network has minimum number of modules and levels. This optimization in turn results in reduction of power consumption and delay. The synthesis results obtained using the GA based approach for various functions are given in Section 6.11.