CHAPTER 3

A NOVEL SHIFT AND ADD ALGORITHM FOR LOW POWER AND AREA EFFICIENT FIR FILTER

3.1 INTRODUCTION

High-speed DSP systems are increasingly being implemented on FPGA hardware platforms. This trend is being fuelled by insurmountable ASIC project costs and the flexibility and reconfigurability advantages of FPGAs over traditional DSPs and ASICs, respectively. Very recently, Structured ASIC technology has yielded lower cost solutions to full custom ASIC by predefining several layers of silicon functionality that requires the definition of only a few fabrication layers to implement the required design.

However, the FPGA platform provides high performance and flexibility with the option to reconfigure (Macpherson et al 2002) and is the technology focused on for this thesis. There is a constant requirement for efficient use of FPGA resources where occupying less hardware for a given system can yield significant cost-related benefits (Bull and Horrocks 1991).

- Reduced power consumption
- Area for additional application functionality
- Potential to use a smaller and cheaper FPGA.

FIR digital filters (Meyer-Baese 2001) are common DSP functions and are widely used in FPGA implementations. If very high sampling rates are required, full-parallel hardware must be used where every clock edge feeds a new input sample and produces a new output sample (Dempster et al 2002). Such filters can be implemented on FPGAs using combinations of the general purpose logic fabric, on-board RAM and embedded arithmetic hardware.

Full-parallel filters cannot share hardware over multiple clock cycles and so tend to occupy large amounts of resource. Hence, efficient implementation of such filters is important to minimise hardware requirement. When implementing a DSP system on a platform containing dedicated arithmetic blocks, it is normal practice to utilise such blocks as far as possible in preference to any general purpose logic fabric. However, in some cases, there may not be enough blocks for the target application or layout/routing constraints may prohibit their use. In such cases, the general purpose logic fabric must be utilised.

Due to rapid increases in fabrication technology, the current generation of FPGAs contains a large number of Configurable Logic Blocks (CLBs) and are becoming more feasible for implementing a wide range of arithmetic applications (Dempster et al 1994). The high nonrecurring engineering costs and long development time for ASICs make FPGAs attractive for application specific DSP solutions (Thomas Poonnen and Adly Fam 2007).

In this chapter, two methods for implementing high speed FIR filter on FPGA are presented. FIR filter based shift and add method (Maskell 2007) is used for designing FIR. In the second method, a new algorithm considering reduced adder graph technique for a set of fixed coefficients is presented (Jin-Gyun et al 2002). Implementation of full-parallel filters using only the general purpose logic fabric is considered. Hence, the techniques presented here are also applicable to ASIC and Structured ASIC platforms.

Due to the rapid progress in the field of VLSI, improvements in speed, power and area are quite evident. Research and development in this field are motivated by growing markets of portable mobile devices such as personal multimedia players, cellular phones, digital camcorders and digital cameras.

In Section 3.2, basic multiplier concepts, multiplier architecture, Booth encoder and partial product generator, Booth algorithm and carry save adder are described. The first method - FIR Filter based Shift/Add multiplier is discussed in Section 3.3. Multiplier block synthesis for low FPGA area is discussed in Section 3.4. The second method - new RSG algorithm is presented in Section 3.5, followed by the presentation of results in Section 3.6. Conclusion is given in Section 3.7.

3.2 MULTIPLIERS

Multiplier is one of the most important components of many systems and should be designed in such a way that it consumes less power and less time while processing (Sheeran and Claessen 2000). In all signal processing applications and VLSI field, the multipliers are frequently used for correlations, Fourier transforms, Fast transforms, filtering, frequency analysis, Microprocessor, Convolution applications, DSP applications and some kinds of kernels for multimedia applications also (Rabaey 1996).

The accusative of a best multiplier is to provide a high speed, minimum level of power consumption and minimum level of physical area requirement in a System on Chip. Designing of adders and multipliers in

VLSI for the purpose of fixed functions is both physically and economically untempting. Multipliers are classified by the format in which data words are accessed in three formats: a) Serial form b) Parallel form c) Serial-Parallel form (Brent and Kung 1982).

The designers of these multipliers have to confirm either the speed of operation or the hardware cost (Bewick 1994). Depending upon the application requirements, if the multiplication operation is performed, it is at the faster rate and if it is low cost, then the speed of the hardware is slow. The speed and area of the multipliers are inversely proportional (Wallace 1964). Depending upon the performance of the multiplier, the system's performance is varied. That is, multipliers performances are directly proportional to the system performance. Generally there are various types, here some of them are classified as follows: array multiplier, baugh wooley multiplier, braun array multiplier, wallace tree multiplier, carry save adder multiplier from carry save adder and ripple carry adder multiplier from ripple carry adder (Gary Bewick et al 1992).

3.2.1 Binary Multiplication

In the binary number system, the digits, called bits are limited to the set [0, 1]. The result of multiplying any binary number by a single binary bit is either 0, or the original number. This makes forming the intermediate partial-products simple and efficient. Summing up these partial-products is the time consuming task for binary multipliers. One logical approach is to form the partial-products one at a time and sum up them as they are generated. Often implemented by software on processors that do not have a hardware multiplier, this technique works fine, but is slow because at least one machine cycle is required to sum each additional partial-product. For applications where this approach does not provide enough performance, multipliers can be implemented directly in hardware (Brent and Kung 1982).

3.2.2 Hardware Multipliers

Direct hardware implementations of shift and add multipliers can increase performance over software synthesis, but are still quite slow. The reason is that as each additional partial-product is summed, a carry must be propagated from the least significant bit (lsb) to the most significant bit (msb) This carry propagation is time consuming and must be repeated for each partial product to be summed.

One method to increase multiplier performance is by using encoding techniques to reduce the number of partial products to be summed (Bob Elkind et al 1987). The original Booth's algorithm ships over contiguous strings of l's by using the property that: $2^n + 2^{(n-1)} + 2^{(n-2)} + \ldots + 2^{(n-m)} = 2^{(n+l)} - 2^{(n-m)}$. Although Booth's algorithm produces at most N/2 encoded partial products from an N bit operand, the number of partial products produced varies. This has caused designers to use modified versions of Booth's algorithm for hardware multipliers.

Modified 2 bit Booth encoding halves the number of partial products to be summed. Since the resulting encoded partial-products can then be summed using any suitable method, modified 2 bit Booth encoding is used on most modern floating-point chips (Lu 1988, McAllister et al 1986). A few designers have even turned to modified 3 bit Booth encoding, which reduces the number of partial products to be summed by a factor of three (Benschneider 1989). The problem with 3 bit encoding is that the carry-propagate addition required to form the 3X multiples often overshadows the potential gains of 3 bit Booth encoding.

To achieve even higher performance, advanced hardware multiplier architectures search for faster and more efficient methods for summing the partial-products. Most multipliers increase performance by eliminating the time consuming carry propagate additions. To accomplish this, they sum the partial-products in a redundant number representation. The advantage of a redundant representation is that two numbers, or partial-products, can be added together without propagating a carry across the entire width of the number.

Many redundant number representations are possible. One commonly used representation is known as carry-save form. In this redundant representation two bits, known as the carry and sum, are used to represent each bit position. When two numbers in carry-save form are added together, any carry that results is never propagated more than one bit position. This makes adding two numbers in carry-save form much faster than adding two normal binary numbers where a carry may propagate (Stanion 1999). One common method that has been developed for summing rows of partial products using a carry-save representation is the array multiplier.

3.2.3 Multiplier Architecture

A multiplier has two stages. In the first stage, the partial products are generated by the Booth encoder and the Partial Product Generator (PPG), and are summed by compressors. In the second stage, the two final products are added to form the final product through a final adder.

The block diagram of traditional multiplier is depicted in Figure 3.1. It employs a booth encoder block, compression blocks, and an adder block. *X* and *Y* are the input buffers. *Y* is the multiplier which is recoded by the Booth encoder and *X* is the multiplicand. PPG module and compressor form the major part of the multiplier. Carry propagation Adder (CPA) is the final adder used to merge the sum and carry vector from the compressor module.

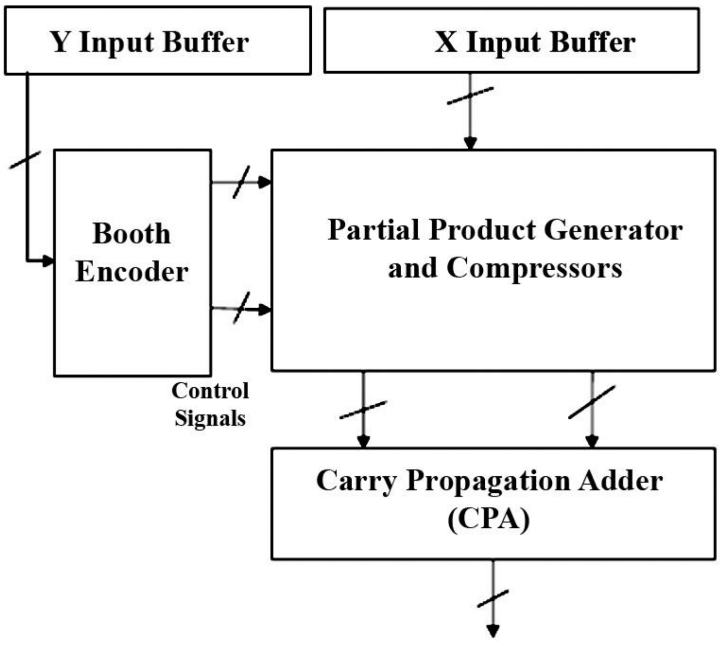

Y Input Buffer
X Input Buffer
Booth Encoder
Partial Product Generator and Compressors
Control Signals
Carry Propagation Adder (CPA)

Figure 3.1 Block diagram of multiplier architecture

Each block is further explained in detail as follows.

3.2.3.1 Booth encoder and partial product generator

Partial product generation is the very first step in binary multiplier. Partial product generators for a conventional multiplier consist of a series of logic AND gates as shown in Figure 3.2.

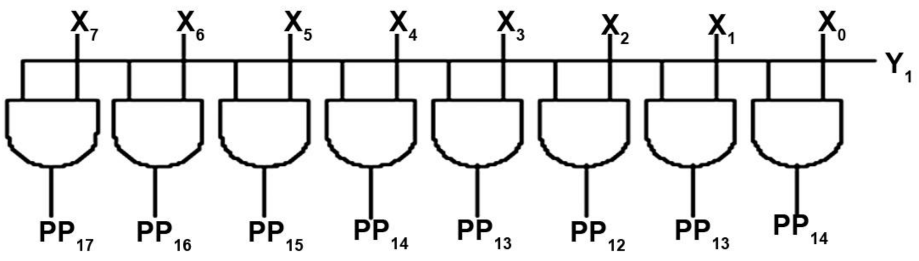


Figure 3.2 Partial product generator using AND gates (Israel Koren 2002)

If the multiplier bit is ‘0’, then partial product row is also zero, and if it is ‘1’, then the multiplicand is copied as it is. From the second bit multiplication onwards, each partial product row is shifted one unit to the left. In signed multiplication, the sign bit is also extended to the left.

3.2.3.2 Booth's algorithm

A.D. Booth proposed Booth encoding technique for the reduction of the number of partial products (Ki-seon Cho et al 2003). This algorithm is also called as Radix-2 Booth's Recoding Algorithm. Here the multiplier bits are recoded as Z_i for every ith bit Y_i with reference to Y_{i-1} .This is based on the fact that fewer partial products are generated for groups of consecutive zeros and ones. For a group of consecutive zeros in the multiplier there is no need to generate any new partial product. It is necessary to shift previously accumulated group partial product one bit position to the right for every 0 in the multiplier.

The radix-2 algorithm results in these observations (Israel Koren 2002) :

a) Booth observed that whenever there was a large number of consecutive ones, the corresponding additions could be replaced by a single addition and a subtraction

$$2^j + 2^{j-1} + \ldots\ldots\ldots\ldots\ldots + 2^{i+1} + 2^i = 2^{j+1} - 2^i \qquad (3.1)$$

b) The longer the sequence of ones, the greater the savings.

c) The effect of this translation is to change a binary number with digit set [0, 1] to a binary signed-digit number with digit set [-1, 1].

The Radix-2 Booth algorithm is given in Table 3.1.

Table 3.1 Radix-2 Booth recoding (Israel Koren 2002)

Y_i	Y_{i-1}	Z_i	Explanation
0	0	0	No string of 1s in sight
0	1	1	End of string of 1s in Y
1	0	1	Beginning of string of 1s in Y
1	1	0	Continuation of string of 1s in Y

In this algorithm the current bit is Y_i and the previous bit is Y_{i-1} of the multiplier $Y_{n-1}\ Y_{n-2}\ldots\ldots\ Y_1\ Y_0$ are examined in order to generate the i th bit Z_i of the recoded multiplier $Z_{n-1}\ Z_{n-2}\ \ldots\ldots. Z_1\ Z_2$. The previous bit Y_{i-1} serves only as the reference bit. The recoding of the multiplier bits need not be done in any predetermined order and can be even done in parallel for all bit positions. The observations obtained from the radix-2 Booth recoding are listed below:

- It reduces the number of partial products which in turn reduces the hardware and delay required to sum the partial products. It adds delay into the formation of the partial products.
- It works well for serial multiplication that can tolerate variable latency operations by reducing the number of serial additions required for the multiplication.
- The number of serial additions depends on the data (multiplicand)
- Worst case 8-bit multiplicand requires 8 additions
- Parallel systems generally are designed for worst case hardware and latency requirements. Booth-2 algorithm does

not significantly reduce the worst case number of partial products.

Radix-2 Booth recoding is not directly applied in modern arithmetic circuits; however, it does help in understanding the higher radix versions of Booth's recoding. It doesn't have consecutive 1s or -1s. The disadvantages of the radix-2 Booth algorithm can be overcome by using Modified Booth algorithm.

3.2.3.3 Modified Booth algorithm

The radix-2 disadvantages can be eliminated by examining three bits of *Y* at a time rather than two. The modified Booth algorithm is performed with recoded multiplier which multiplies only ± *a* and ± *2a* of the multiplicand, which can be obtained easily by shifting and/or complementation. The truth table for modified Booth recoding is shown in Table 3.2.

Table 3.2 Radix-4 Booth recoding (Ohkubo 1995)

Y_{i+1}	Y_i	Y_{i-1}	Z_{i+1}	Z_i	$Z_{i/2}$	**Explanation**
0	0	0	0	0	0	No string of 1s in sight
0	0	1	0	1	1	End of strings of 1s
0	1	0	0	1	1	Isolated 1
0	1	1	1	0	2	End of string of 1s
1	0	0	-1	0	-2	Beginning of string of 1s
1	0	1	-1	1	-1	End a string, begin a new one
1	1	0	0	-1	-1	Beginning of string of 1s
1	1	1	0	0	0	Continuation of string of 1s

The main advantage of the modified Booth algorithm is that it reduces the partial products to *n/2*.

The following gives the algorithm for performing sign and unsigned multiplication operations by using radix-4 Booth recoding.

Algorithm: (for unsigned numbers)

- Pad the LSB with one zero
- Pad the MSB with two zeros if n is even and one zero if *n* is odd
- Divide the multiplier into overlapping groups of 3-bits
- Determine partial product scale factor from modified Booth-2 encoding table
- Compute the multiplicand multiplies
- Sum partial products

Algorithm: (for signed numbers)

- Pad the LSB with one zero
- If *n* is even don't pad the MSB (*n/2* PP's)
- Divide the multiplier into overlapping groups of 3-bits
- Determine partial product scale factor from modified Booth-2 encoding table
- Compute the multiplicand multiplies
- Sum partial products

Booth recoding is fully parallel and carry free. It can be applied to design a tree and array multiplier, where all the multiples are needed at once. Radix-4 Booth recoding system works perfectly for both signed and unsigned operations.

3.2.3.4 Compressors

A Carry-Save Adder (CSA) is a set of one-bit full adders, without any carry-chaining and is shown in Figure 3.3. Therefore, an *n*-bit CSA receives three n-bit operands, namely *a (n-1)..a (0), b (n-1)..b (0), and* c_{in} *(n-1)..*c_{in} *(0),* and generates two n-bit result values, *sum (n-1)..sum (0) and* c_{out} *(n-1)..* c_{out} *(0).*

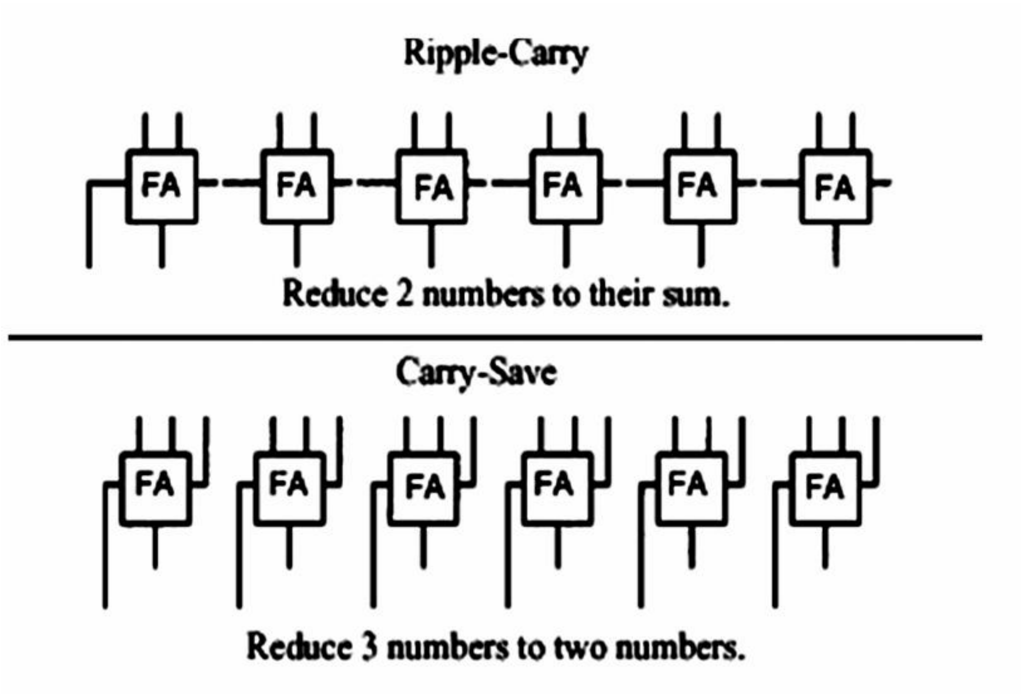

Ripple-Carry
FA
Reduce 2 numbers to their sum.
Carry-Save
FA
Reduce 3 numbers to two numbers.

Figure 3.3 Carry save adders (Israel Koren 2002)

A carry save adder tree can reduce *n* binary numbers to two numbers having the same sum in *O (log n)* levels. Carry save adder is also called a compressor and a Wallace Tree is constructed with CSAs. Wallace trees are CSAs in a tree structure used as a compressor. The most important application of a carry-save adder is to add the partial products in integer multiplication. From CSA separate sum and carry vector are obtained. In CSA, the output carry is not passed to the neighboring cell but is saved and passed to the cell one position down.

3.2.3.5 Carry propagation adder

The final step in completing the multiplication procedure is to add the final terms in the final adder. The Carry Propagation Adder, CPA, is a final adder used to add the final carry vector to the final sum vector partial products to give the final multiplication result. This is normally called a “Vector-merging” adder. The choice of the final adder depends on the structure of the accumulation array. Various fast adders can be used as CPA. Some of them are Carry look-ahead adder, Simple carry skip adder, Multi level carry skip adder, Carry-select adder, Conditional sum adder and Hybrid adder.

A Carry look-ahead Adder is an adder used in digital logic. All the carry outputs are calculated at once by specialized look-ahead logic. But it requires generate and propagate signals. Simple carry skip adders looks for the cases in which carry out of a set of bits are identical to carry in. Circuits for binary adders to efficiently skip a carry bit over two or more bit positions with two or more carry-skip paths is called multilevel carry skip adders.

In the 4-bit carry select adder there are two 4-bit adders each of which takes a different preset carry-in bit. The two sums and carry-out bits that are produced are then selected by the carry-out from the previous stage.

In conditional sum adder, sum and carry outputs at the first stage assume the previous carry to be zero and sum and carry outputs at the second stage assume the previous carry to be one. For CPA, any of these adders can be combined as a hybrid adder.

3.3 FIR FILTER BASED SHIFT/ADD MULTIPLIER

FIR filter is implemented using the shift and add method. All the optimizations are performed in the multiplier block. The constant multiplications are decomposed into additions and shifts and the multiplication complexity is reduced (Shahnam Mirzaei et al 2006). It is possible to implement the design in the two forms and the graph forms of design 1 and design 2 are shown in the Figure 3.4.

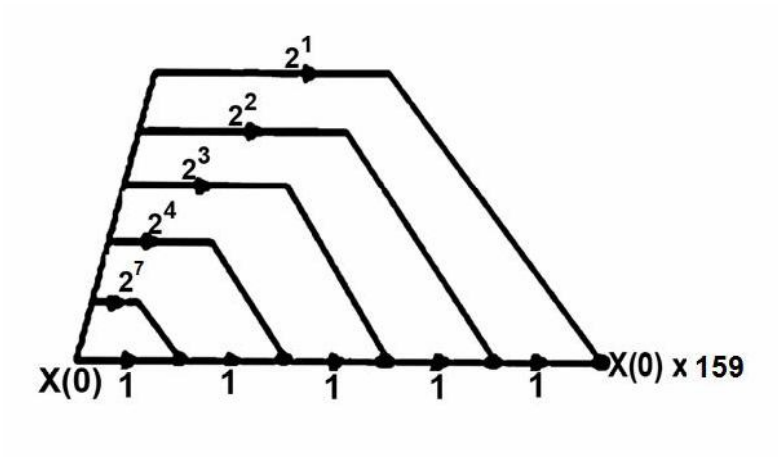
Design 1 :

The coefficients are changed to integer getting multiplied to a multiple power of 10, then these coefficients are arranged to the positive power of 2. This procedure is shown below in graph form. Graph branches stand for left shift and notches stand for sum.

$$c[0]= 0.159 ,c[1]= -.053 \xrightarrow{x1000} c[0]= 159, c[1]= -53 \tag{3.1}$$

$$c[0]= 128+16+8+2+1 = 2^7+2^4+2^3+2^2+2^1+2^0 \tag{3.2}$$

$$c[1]= -32-16-4-1 = -2^5-2^4-2^2-2^0 \tag{3.3}$$



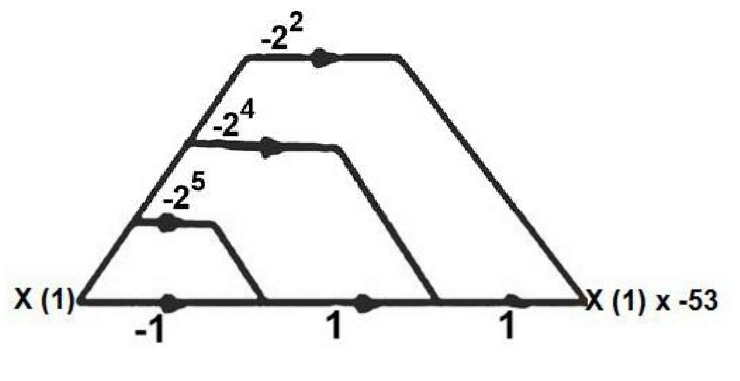


Figure 3.4 (a) Graph form of design 1

Design 2 :

First, decimal coefficients are arranged according to negative and positive power of 2 (no need to them into integer). So the filter hardware and power consumption are reduced.

c[0]=3.75

$$c[0] = 2^1+2^{-1}+2^{-2}+2^0 \tag{3.4}$$

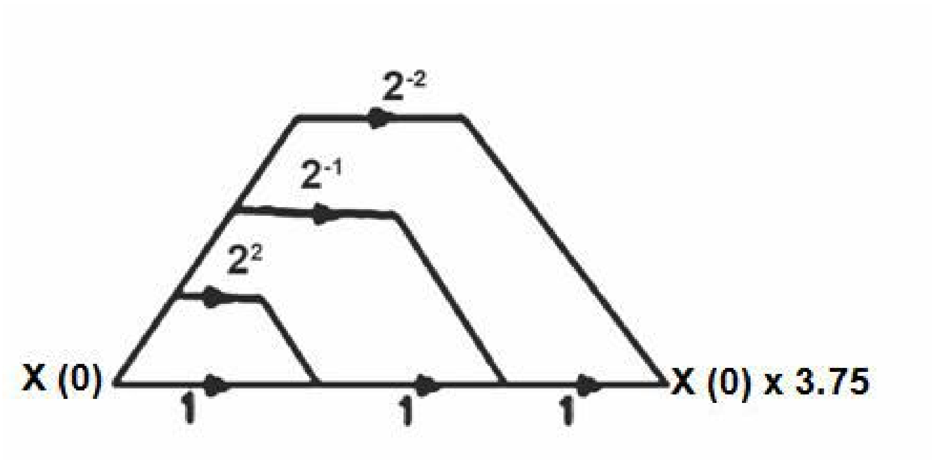


Figure 3.4 (b) Graph form of design 2

3.4 MULTIPLIER BLOCK SYNTHESIS FOR LOW FPGA AREA

3.4.1 Multiplication Hardware Operation and Area Estimation

Figure 3.5 shows an example multiplier block (also referred to as a graph) that multiplies the input by 3, 13, 21 and 37 in two clock cycles (the ‘logic depth’ of the block is also 2) Multiplication is achieved using only adds, subtracts and shifts which map very efficiently to FPGA architectures (Dempster et al 2002). As an example, the input is fed to the ‘3’ adder untouched and after being left shifted once (multiplied by two). Hence, the output of the ‘3’ adder is 2x + x = 3x as required. This product can then be used as a graph output to be fed to the filter summation chain and, if required, routed internally to generate further multiples of the input (Dempster et al 1994).

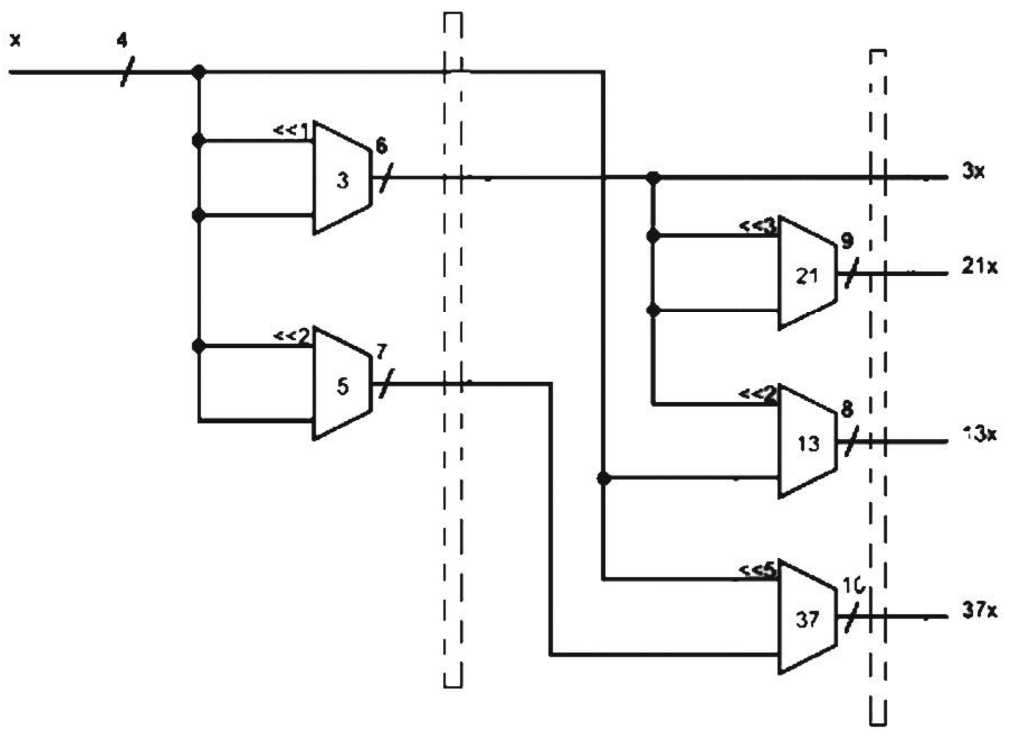


Figure 3.5 Multiplier Block: Five Adders, Two Pipeline Stages Costing 25 Slices

▷	Adder / Subtractor
<<	Shift signal left by p bits (multiply signal by 2^p)
–	Signal is subtracted from other entering adder / subtractor
—/—	Signal is 4 bits wide
⁞⁞	Crossing signals are pipelined

For efficiency, multiplier blocks need only generate positive, odd integers since negative filter coefficient weightings can be restored at the summation chain by subtracting the positive equivalent generated by the multiplier block. Odd valued block outputs can be left shifted en-route to the summation chain to generate even-valued coefficient multiplications. Pipelining multiplier blocks ensuring high clock rates are achieved when implemented on FPGA hardware. The multiplier blocks usually contain a mixture of adders and subtractors, but the ‘adder cost’ of a block refers to the number of adders and subtractors.

Hence, the adder cost of Figure 3.5 is 5. Note that adders may also be referred to as the graph ‘vertices’.

The multiplier block in Figure 3.5 is quoted as costing 25 slices. This is calculated by counting the number of flip-flops inferred by the multi-bit signals crossing pipeline boundaries and dividing by 2 since there are two flip-flops per slice. Equation (3.5) uses the set S which contains the bit-widths of all N multi-bit signal pipeline boundary crossings to obtain a slice estimate e

$$e = \sum_{a=1}^{N} \frac{S_a}{2} \qquad (3.5)$$

3.4.2 Multiplier Block Synthesis Optimisation Goals

Multiplier block synthesis has received a great deal of attention in recent decades. The majority of research has concentrated on producing algorithms to synthesise multiplier blocks with the optimization goal of minimum adder cost. Bull and Horrocks (1991) introduced the concept of representing multiplier blocks with graphs and defined several minimal adder synthesis algorithms. Dempster et al (1995) identified limitations in this work and defined the ‘ n-dimensional reduced adder graph’ (RAG-n) algorithm which is generally regarded as the primary reference for minimal adder multiplier block synthesis.

Additional techniques using CSD and subexpression sharing have also been proposed to minimize adder cost. Though the majority of research in this area has focused on full-parallel DSPs, recent work by Demirsoy et al (2003) incorporates multiplexers to allow efficient

FPGA implementation of time multiplexed filters and Direct Cosine Transform (DCT) processors.

Dempster et al (2002) defined the ‘C1’ synthesis algorithm with the optimization goal of minimizing multiplier block logic depth to reduce power consumption. C1 aims to minimize the power consumption by reducing the amount of logic transitions caused by long glitch paths through cascaded arithmetic logic. The logic toggling caused by glitch paths through more than one adder does not occur in fully pipelined multiplier blocks In general, from a hardware perspective, the motivation for minimum adders has been to reduce filter complexity for Very Large Scale Integration (VLSI) implementation (Uwe Meyer-Baese 2007) where adder cost dominates the area requirement.

However, FPGAs have a fixed architecture for implementing digital logic, not the ‘blank canvas’ of VLSI/ASIC design. Hence, algorithms synthesizing multiplier blocks for low FPGA area must operate with regard to FPGA architectures for best results. In this chapter, it shows that the classic multiplier block synthesis goal of minimizing adders does not minimize FPGA hardware cost. However, a new algorithm is defined that minimizes the hardware resources and reduces the power.

3.5 PROPOSED ALGORITHM

This section first describes the limitation of the previous algorithms, then elaborates the proposed algorithm. The previous dependence-graph algorithms such as the Bull-Horrocks Modified (BHM) and RAGn rely on heuristic approaches. Considering one coefficient at a time, the heuristic algorithms minimize the additional adder cost required to incorporate the coefficient into the current partial sum set. Though the adder graph generated for a coefficient is the best solution, it may not be the best if the whole coefficient set is considered because the remaining coefficients are not considered in generating the adder graph.

The proposed algorithm searches for the most frequently occurred common adder graph and then maximizes the reuse of the common adder graph and the partial products to generate entire set of coefficients. The previous dependence-graph algorithm generates an adder graph by inserting new partial sums which minimizes the difference to a coefficient and does not consider the remaining coefficients when synthesizing the coefficient. Fig 3.6 illustrates an example of the proposed adder graph generation process for a coefficient set {3, 13, 21, and 37} that considers the common adder graphs 3 and 5 to generate the remaining coefficients.

The proposed algorithm generates common candidates; a temporary partial sum set for each graph by tentatively inserting the partial sums contained in the graph into the current partial sum set, and then synthesizes the remaining coefficients by applying the conventional dependence-graph algorithm such as the BHM to the temporary set.

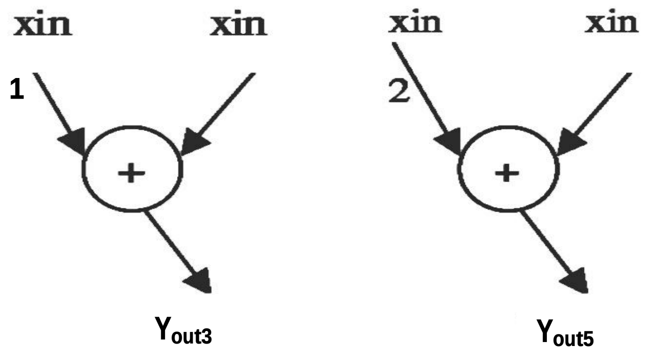


Figure 3.6 (a) Common adder graphs for the set of coefficients {3,13,21 and 27}

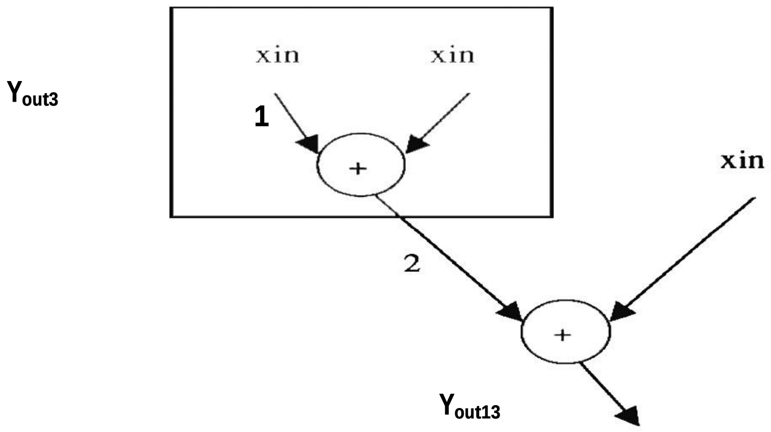


Figure 3.6 (b) Adder Graph for coefficient 13

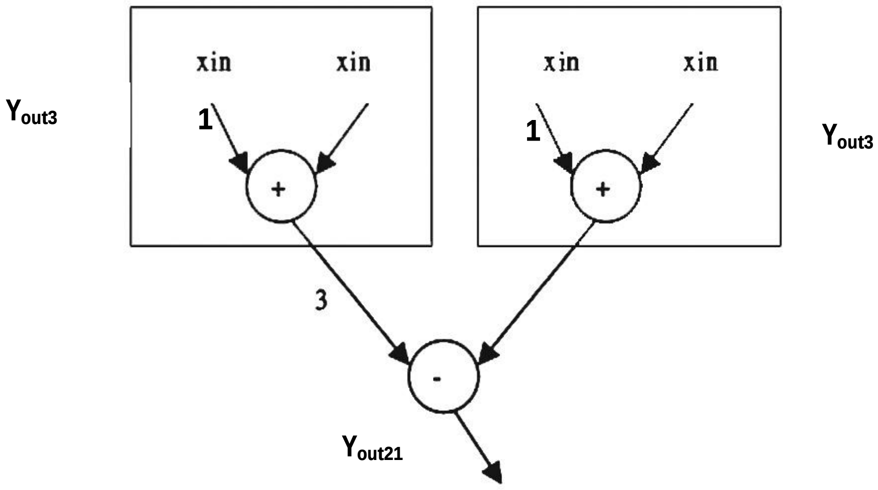


Figure 3.6 (c) Adder Graph for coefficient 21

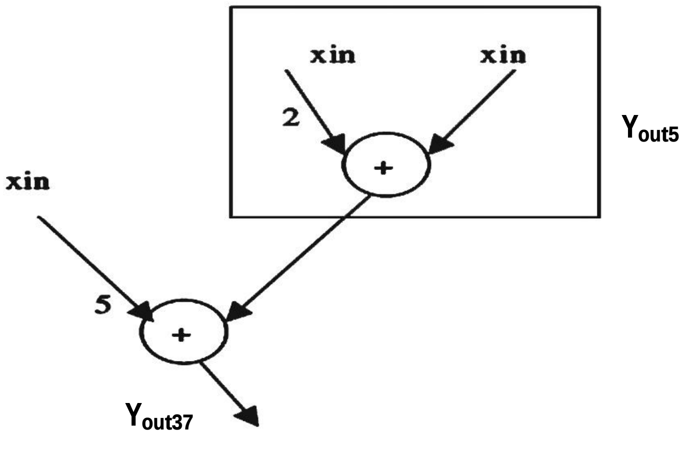


Figure 3.6 (d) Adder Graph for coefficient 37

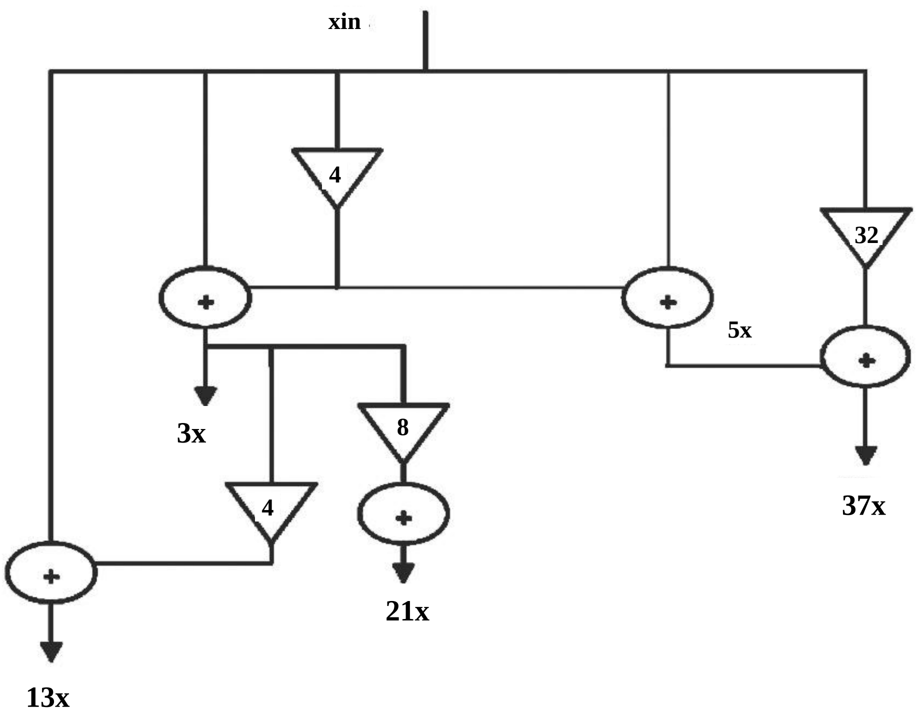


Figure 3.7 Multiplier blocks for coefficients {3,13,21,37} synthesized by the proposed adder graph method

The partial sums of Figure 3.6 (a) are inserted to the current partial sum set of Figure 3.6 (b) In this case, 13 can be synthesized by using one additional adder and the total adder cost becomes 2. Considering the effect of sharing on the remaining coefficients while synthesizing a coefficient, the total adder cost can be reduced. It is observed in the above example.

3.5.1 Generating Coefficient Adder Graphs

In order to reduce the complexity of the multiplier block for a set of fixed coefficient FIR filters (Yao et al 2005), an efficient method is proposed in this subsection to generate coefficient adder graphs from common seed adder graph. In the proposed method, common adder graphs are first generated considering all the coefficients to be synthesized by applying one of the previous dependence-graph algorithms

(Jeong-Ho Han et al 2008) and then using the common adder graph, the remaining coefficient adder graphs are generated. Figure 3.6 illustrates that three additional adder graphs (b),(c) and (d), are generated from a seed adder graph in Figure 3.6 (a). In Figure 3.6 (c), 21 can be synthesized by using two common adder graph, one additional subtractor and the total adder cost becomes 3 and in Figure 3.6 (d), 37 can be synthesized by using a common adder graph, one additional adder and the total adder cost becomes 2. Figure 3.7 illustrates an example how to expand the common adder graph to obtain the entire coefficient adder graph.

The seed graph can be constructed by using a cost-0 partial sum more than once. Considering the common adder graph, the number of edges coming out from a node is always one which is shown in Figure 3.6 (a). The common adder graph can be transformed into a tree by duplicating such a node. In this case, the shift amount and the sign of an outgoing edge to the input edges can be moved, as there is only one outgoing edge in a tree.

Starting from the root node of 3x and 5x the shift amount and the sign of the output edge are propagated to each input edge, i.e. the sign and the shift amount are added to and multiplied by each input edge. Propagation applied recursively until it reaches the cost-0 partial sum nodes. In the adder graph expanding procedure, a partial sum table T is created that stores the shift amount and the sign of each cost-0 partial sum node contained in the adder graph tree.

The procedure to generate an adder graph is initially to select two partial sums, t_i and t_j from partial sum table T, and by adding the shifted values of p_i and p_j a new entry t_{new} is formed. After this, T is updated by removing t_i and t_j and inserting t_{new}. The sharing effect is maximizing of partial sums and the value of p_{new} is restricted to a positive odd number by adjusting its shift amount and sign value when inserting the new partial sum into T. This process event is recursively applied until only a partial sum remains in the table. An adder graph is generated that results in the target coefficient to update the table sequence.

The set of coefficient adder graphs can be obtained by changing the combination of two partial sums to be selected from T. The total number of adder graphs, *N*, that can be generated from a common adder graph is expressed in Equation (3.6) where *n* is |T|. As indicated in Equation (3.6), the number of adder graphs grows exponentially as *n* increases. For example, the number of possible adder graphs is 3 for *n*=3, 18 for *n*=4, and 180 for *n*=5.

$$N\prod_{i=0}^{n=3} {}_{n-i}C_2, \quad (n \geq 3)$$

$$N = \frac{n!}{2!(n-2)!} * \frac{(n-1)!}{2!(n-2)!} * ... * \frac{3!}{2!1!} = \frac{n!*(n-1)!}{2^{n-1}} \tag{3.6}$$

3.5.2 Description of Proposed N-RSG Algorithm

The proposed algorithm is divided into two parts. The first part is to select a coefficient to be synthesized from a set of coefficients S, the set of coefficients not synthesized yet. In the second part, selected coefficient adder graphs are generated from the seed adder graph .The seed adder graphs seems to be maximally shared with the remaining coefficients. The proposed algorithm is as follows:

- Initialisation process

  Form a coefficient set S from the filter coefficients.

- Check and remove the negative coefficients.
- Verify that all coefficients are positive.
- Reduce all selected positive coefficients to 2^k coefficients.
- Choose a coefficient for synthesis process.
- Get an adder graph for each $s \in S$ using the seed adder graph.
- Remove 2 K factors from the coefficient.
- Find the common cost from all coefficients.
- Calculate the remaining coefficients from the common cost.
- **MaxAdderCost** = maximum adder cost of **IncompleteSet** members
- for **CurrentCost** = **MaxAdderCost** down to 1
- {
- **mult** = next multiplier from **IncompleteSet** whose cost is **CurrentCost**
- if (**mult** is not in **GraphSet**)
- {
- **mult_graph** = **best graph** for **mult** from RAG implementation
- Add graph fundamentals of **mult_graph** to **GraphSet**
- Add **mult** to **GraphSet**
- Store graph topology data for **mult**
- }
- Remove **mult** from **IncompleteSet**
- }

3.6 SYNTHESIS RESULTS

In this section, the synthesis and design results of the proposed algorithm for adder graph generation process for a set of coefficients that considers the common candidates to generate the remaining coefficients are presented. Xilinx 12.3i ISE has been used for the purpose of synthesizing the design. The RTL schematic and simulation result of a FIR Filter for a set of fixed coefficients using proposed algorithm are shown in Figures 3.8 and 3.9.

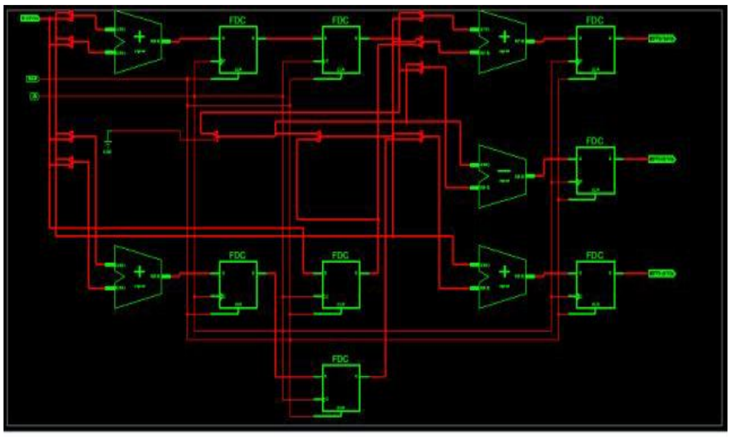


Figure 3.8 RTL schematic of the proposed FIR filter

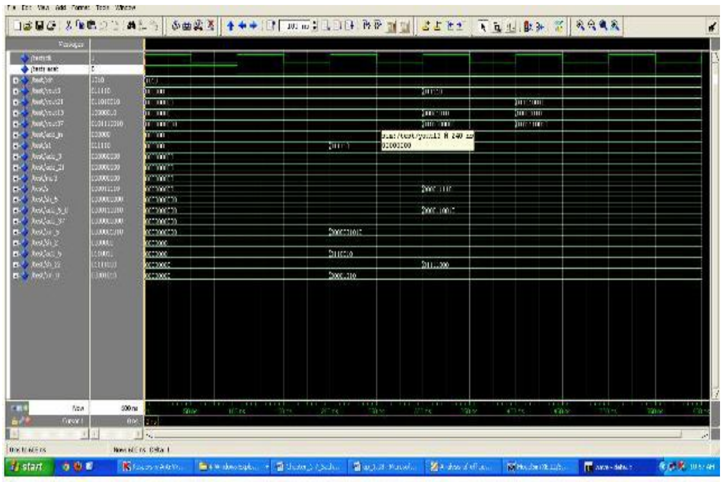


Figure 3.9 Simulation result of the proposed FIR filter

Comparison of the performances of proposed FIR Digital Filter Synthesis algorithm with the existing algorithms are shown in Table 3.3, Figures 3.10 to 3.12.

Table 3.3 Comparison of performances

Parameters	**Proposed N-RSG Algorithm**	**RSG**	**FIR Filter Shift/ Add Method**
Slices	124	156	188
LUT	112	142	163
Power (mW)	316	347	418

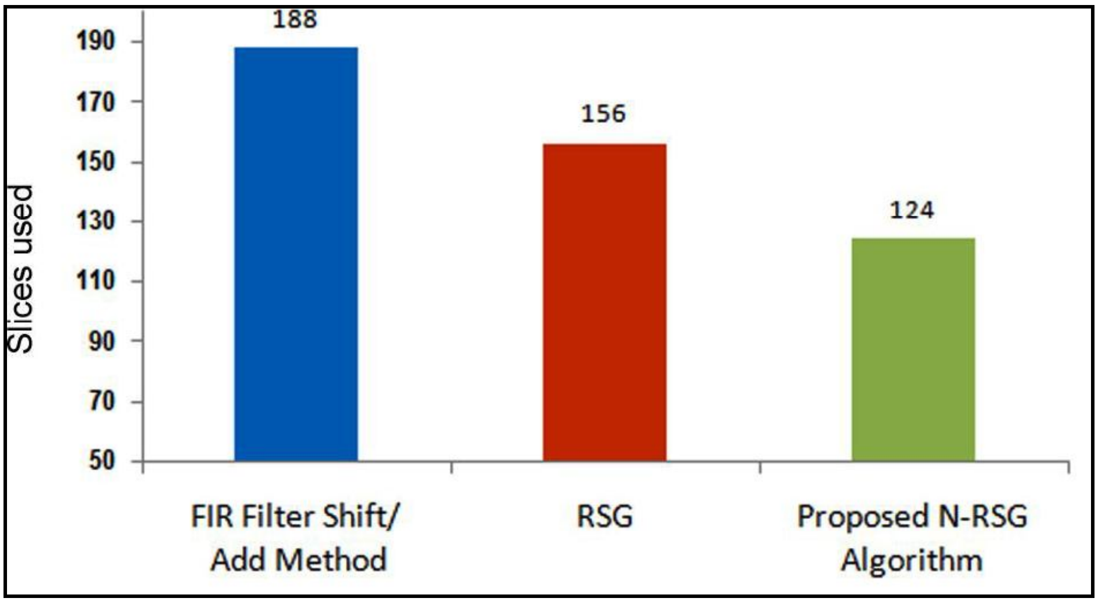


Figure 3.10 Comparison of slices used

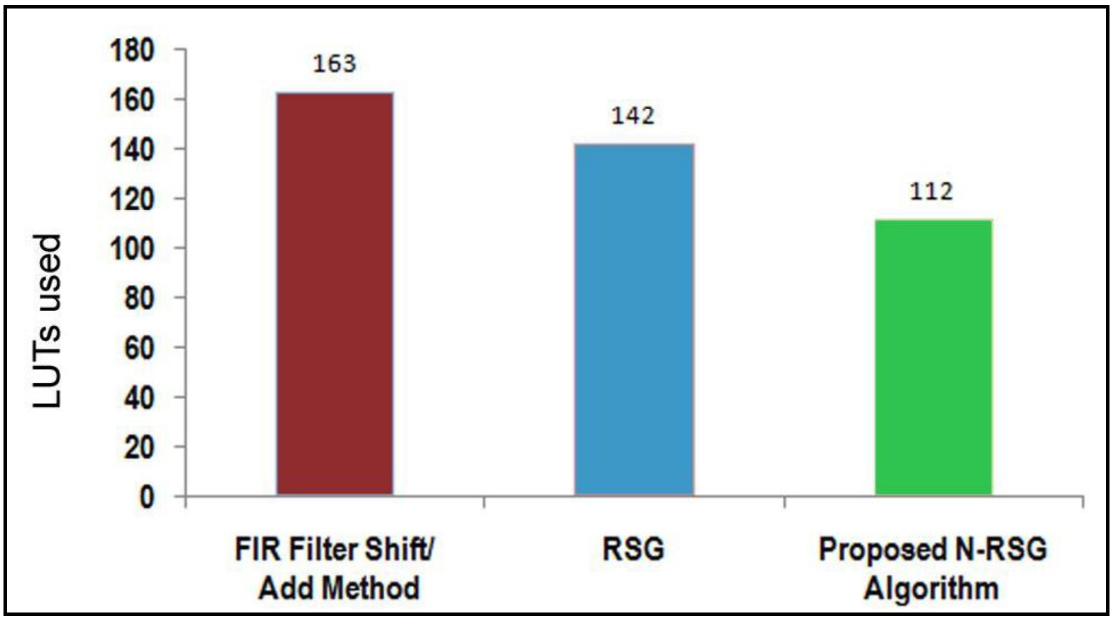


Figure 3.11 Comparison of LUTs utilisation

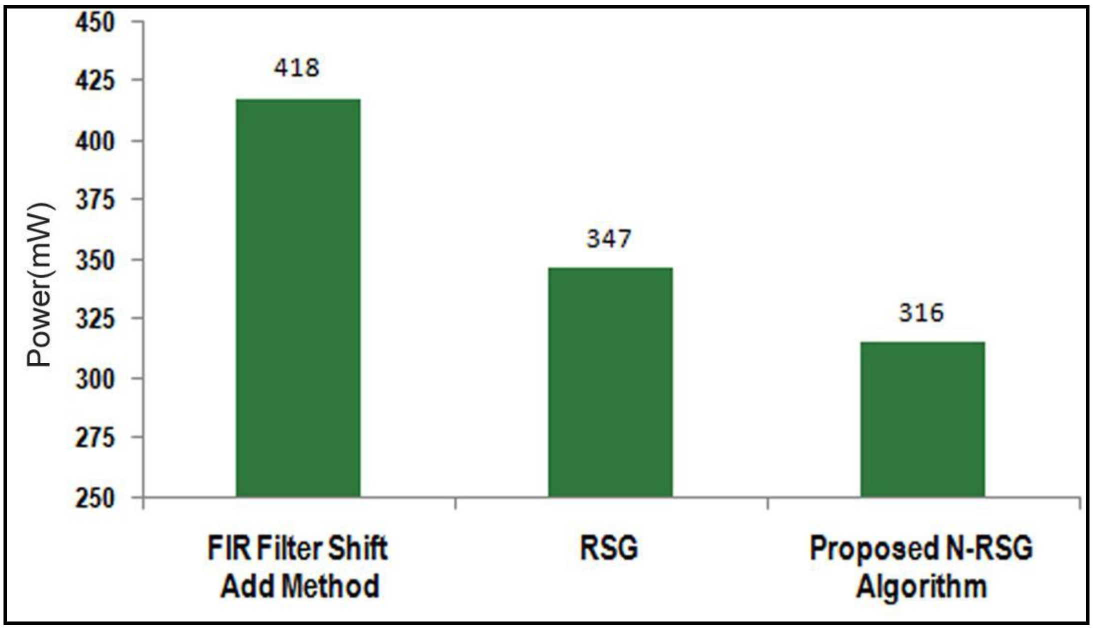


Figure 3.12 Comparison of power

3.7 CONCLUSION

In this research, the proposed new algorithm for FIR digital filter synthesis for a set of fixed coefficients has produced better reduction in the number of slices and to minimize logic delay and wire delay. The advantage of this method is that common adder graphs can be utilized that can be maximally shared with the remaining coefficients and hence results in reduction in adders cost and the adder step.

The proposed algorithm considers multiple adder graphs for a set of coefficients to be synthesized, whereas previous dependence-graph algorithms consider only one adder graph when implementing a coefficient. Seed adder graph that can be shared maximally with the remaining coefficients is selected first. Starting from a common adder graph, adder graphs for the entire set of coefficients have been derived. Comparing the proposed algorithm with the previous RSG algorithm, there is a 20% reduction in slices and 21% reduction in LUTs and power is reduced by 9%.

Experimental results show that the proposed algorithm reduces the hardware cost of the FIR filter for a set of fixed coefficients compared with the FIR filter based shift and add multiplier. The proposed algorithm provides 34% reduction in slices, 31% reduction in LUTs and power reduction by 24%.