Automated Design Exploration for FIR filters in FPGAs

HK Verma

1. Introduction

Today, Field Programmable Gate Arrays (FPGAs) are large enough to perform various DSP functions. Finite Impulse Response (FIR) filters are basic Digital Filters used widely in DSP systems. FIR filters for various requirements can be implemented nicely in typical FPGAs, which we explain further in this paper.
A general FIR Filter expression with Constant Coefficients is given by

$$y(n) = \sum_{i=0}^{n-1} \left(C_i \times x(i) \right)$$ **Equation 1. A FIR Filter with Constant Coefficients**

Let us assume that the bit width of the constant is c. Let the bit width of x be b. The constants and variables can then be represented in terms of the following equations.

$$x(i) = \sum_{j=0}^{b-1} 2^j \times x_{ij}$$ **Equation 2. An input variable x(i)**

$$C_i = \sum_{k=0}^{c-1} 2^k \times C_{ik}$$ **Equation 3. A constant C*i***

If we incorporate Equation 2 and Equation 3 in Equation 1, we get the following equation.

$$y(n) = \sum_{i=0}^{n-1} \left[\left\{ \sum_{k=0}^{c-1} \left(2^k \times C_{ik} \right) \right\} \times \left\{ \sum_{j=0}^{b-1} \left(2^j \times x_{ij} \right) \right\} \right]$$ **Equation 4. A FIR Filter**

A very promising approach for FIR filter realization in XC4000 series FPGAs is using Distributed Arithmetic (DA) techniques, which we describe in the next Section. As it turns out, there are certain design choices associated with implementation using DA techniques, which affect area significantly. Our contribution in this paper is to identify those choices and provide an automated way to make the optimal choice. We assume that there are no constraints on the latency of the output. So we can insert as many delay elements (i.e. registers) as we want in the data path. This will not affect the functionality, because there are no feedback paths. So, the fanout in various implementations do not differ a lot. Hence, a comparable clock speed for most of the implementations can be achieved by reducing the logic levels to a minimum by inserting suitable number of registers. For a register rich architecture like Xilinx FPGAs, these constraints do not increase the resources used.

2. Distributed Arithmetic

In a Distributed Arithmetic Implementation of a FIR filter , the memory is distributed in the Look Up Tables (LUTs) of a Xilinx FPGA Architecture. A typical LUT in Xilinx architecture can be used to realize any 4-input function. In this section we will show the common DA implementation that has been used in earlier implementations. In this implementation, we have a ROM which will store the information shown in the following equation as sda(j).

$$sda(j) = \sum_{i=0}^{n-1} \left[\left\{ \sum_{k=0}^{c-1} \left(2^k \times C_{ik} \right) \right\} \times x_{ij} \right]$$ **Equation 5. A common term for a DA implementation**

Each sda(j) for a given value of j can be determined from a ROM if the n values of the input x_{ij} are available. We need to implement this ROM in an FPGA. We can create different LUTs for each 4 inputs of x_{ij}. These different LUTs are then added together to get the value of the term sda(j).

The following equation can now be implemented either serially or in parallel.

$$y(n) = \sum_{j=0}^{b-1} 2^j \times sda(j)$$ **Equation 6. The parallel terms of a DA implementation**

In a serial implementation, there is a scaling accumulator at the end which sums the input with the weighted output to give the correct result. In a parallel implementation, the b results are added using an adder tree. If we assume that the System Speed is S in both serial and parallel case, the output will be available every b cycles in the serial case whereas in the parallel case it will be available every cycle. In other words, the throughput rate is (S/b) in serial case and S in the parallel case.

In the subsequent sections, we are going to show how we can design the most efficient DA implementations.

3. Efficient DA Implementations

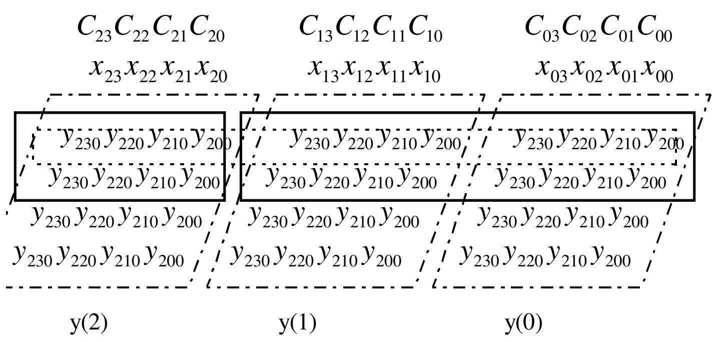
In this section, we explore in detail the usage of DA techniques for Xilinx Architecture with a view to forming the FIR filter groundwork for our design tool. Since we assumed that similar speeds can be obtained across different implementations by heavy pipelining, we would try to minimize area as an efficient implementation. The area is measured in terms of the following atomic resources available to us in an XC4000 architecture.

- 4-input Look-Up Table (LUT) which can be registered by choice.
- A 2 bit Full Adder (FA) with carry in and out which can also be registered by choice.

A LUT or an FA occupy half Configuration Logic Block (CLB) of an XC4000 architecture. In our formulations, we do not take carry initializations into consideration. We also assume that the FAs can be concatenated together to form a carry chain without any limitations. We neglect these because they have a small impact in the area calculations.

3.1 Design Choices

We first explore the fully parallel solution with the throughput being equal to the system speed S. In our approach we try to construct the serial term by decomposing the FIR filter implementation in different ways suitable for a DA implementation using 4-input LUTs. In Figure 1. Different Design Choices with different Coverings, it is illustrated with the help of a 3-tap filter with buswidth of input and constant bus width being equal to 4. In a Serial Term, we can include either one, two or four bits of consecutive input bits. These are shown as single-bit, two-bit and four-bit coverings. Each grouping can be obtained by 4 bits of input and therefore can be stored inside a LUT. These different groupings are then added together to get different design choices. Henceforth, we shall use Single Input-Bit for one-bit covering, Two Input-Bit for two-bit covering and Four Input-Bit for four-bit covering. These different implementations are explained in the subsequent sub-sections. In these formulations, we first give the general implementations with proper equations. These implementations are then explained with the design of a 10-tap FIR filter with the bus-width of input and constant coefficient being 16.


$C_{23}C_{22}C_{21}C_{20}$
$C_{13}C_{12}C_{11}C_{10}$
$C_{03}C_{02}C_{01}C_{00}$
$x_{23}x_{22}x_{21}x_{20}$
$x_{13}x_{12}x_{11}x_{10}$
$x_{03}x_{02}x_{01}x_{00}$
$y_{230}y_{220}y_{210}y_{200}$
y(2)
y(1)
y(0)

Bits enclosed by different rectangles show LUTs for a serial term.
Different lines indicates different coverings as shown below.

··········	Single-Bit Covering
———	Two-Bit Covering
–·–·–·–	Four-Bit Covering

Figure 1. Different Design Choices with different Coverings

3.1.1 Single Input-bit Implementation

For a single input bit implementation, we create LUTs for each j of input x_{ij}. This term has been previously shown in Equation 5. In the parallel implementation there are n such terms which are added using an adder tree. This was shown in Equation 6. As shown in this equation, this adder tree shifts higher serial terms by 1 and then sums it up.
The implementation in XC4000 for a 10-tap 16 buswidth filter is shown in Figure 2. Single-bit implementation of a FIR filter. This implementation is quite common.

.

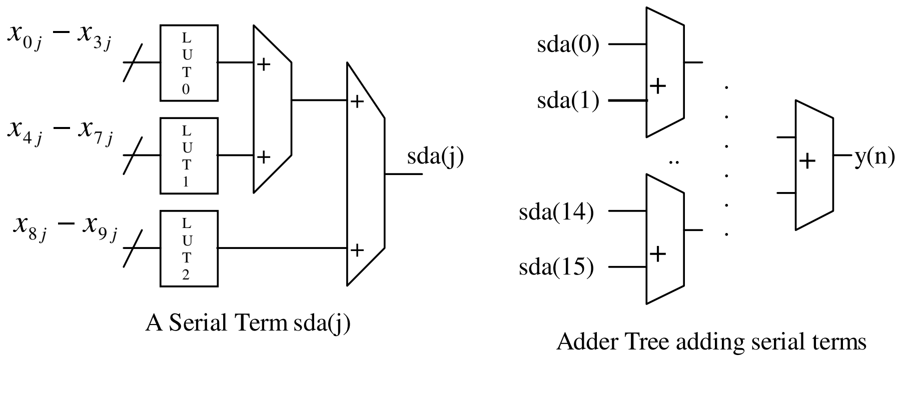


Figure 2. Single-bit implementation of a FIR filter

The contents of single LUT addressed by $x_{0j}, x_{1j}, x_{2j} \& x_{3j}$ are given in Table 1. The contents of other LUTs can be obtained similarly.

Table 1. LUT Contents for a single-bit implementation

LUT Address	LUT contents
0	0
1	C0
2	C1
3	C0+C1
4	C2
5	C2+C0
6	C2+C1
7	C2+C1+C0
8	C3
9	C3+C0
10	C3+C1
11	C3+C1+C0
12	C3+C2
13	C3+C2+C0
14	C3+C2+C1
15	C3+C2+C1+C0

3.1.2 Two Input-bit Implementation

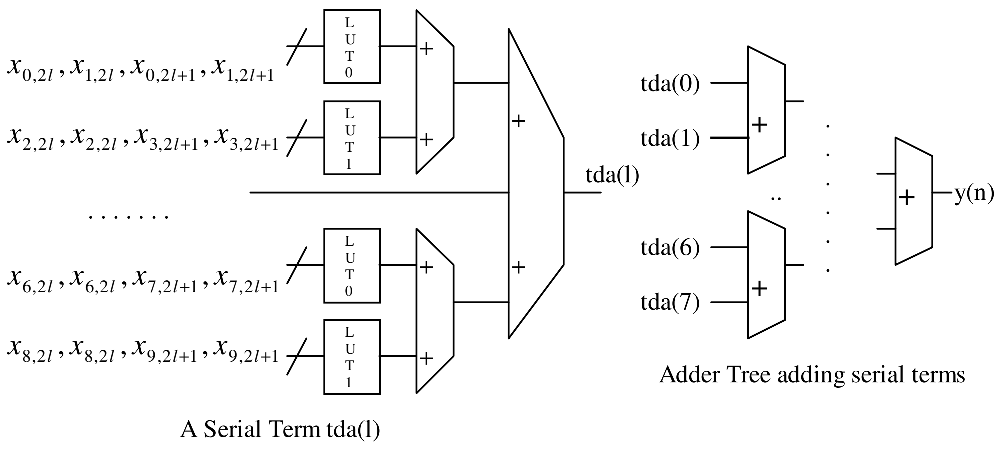
For a two-input bit implementation, we create LUTs for two consecutive j numbers of inputs such as x_{ij} and x_{ij+1}. The resulting serial term is shown in $tda(l) = sda(2l) + 2 \times sda(2l+1)$ Equation 7. Serial term for two-input bit implementation. sda(j) is the serial term for single bit implementation. Maximum value of l in tda(l) is equal to (b/2).

$$tda(l) = sda(2l) + 2 \times sda(2l+1)$$ **Equation 7. Serial term for two-input bit implementation**

For example, in our 10-tap 16-bit wide filter, a serial term will have 5 LUTs. These LUTs will be addressed by the following groups of addresses- ($x_{0,2l}, x_{0,2l+1}, x_{1,2l} \& x_{1,2l+1}$),($x_{2,2l}, x_{2,2l+1}, x_{3,2l} \& x_{3,2l+1}$) and so on. The outputs of these LUTs are added together using an adder tree to get the serial output tda(l). In a parallel implementation, these 8 serial term tda's are added using an adder tree. This is shown in the subsequent equation. As shown in this equation, this adder tree shifts higher serial terms by 2 and then sums it up.

$$y(n) = \sum_{l=0}^{b/2-1} 2^{j} \times tda(l)$$ **Equation 8. The parallel terms of a two-input bit implementation**

The implementation of a 10-tap 16-bit wide filter is shown in Figure 3. Two-bit Implementation of a FIR filter.



$x_{0,2l}, x_{1,2l}, x_{0,2l+1}, x_{1,2l+1}$
$x_{2,2l}, x_{2,2l}, x_{3,2l+1}, x_{3,2l+1}$
....
$x_{6,2l}, x_{6,2l}, x_{7,2l+1}, x_{7,2l+1}$
$x_{8,2l}, x_{8,2l}, x_{9,2l+1}, x_{9,2l+1}$
LUT0
LUT1
LUT0
LUT1
tda(l)
A Serial Term tda(l)
tda(0)
tda(1)
tda(6)
tda(7)
y(n)
Adder Tree adding serial terms

Figure 3. Two-bit Implementation of a FIR filter

The contents of single LUT addressed by $x_{0,2l}, x_{1,2l}, x_{0,2l+1} \,\&\, x_{1,2l+1}$ are given in The contents of other LUTs can be obtained similarly.

Table 2. LUT Contents for a two-bit implementation

LUT Address	LUT contents
0	0
1	C0
2	2xC0
3	3xC0
4	C1
5	C1+C0
6	C1+2xC0
7	C1+3xC0
8	2xC1
9	2xC1+C0
10	2xC1+2xC0
11	2xC1+3xC0
12	3xC1
13	3xC1+C0
14	3xC1+2xC0
15	3xC1+3xC0

3.1.3 Four Input-bit Implementation

For a four-input bit implementation, we create LUTs for four consecutive j numbers of inputs such as x_{ij}, x_{ij+1} x_{ij+2} and x_{ij+3}. The resulting serial term is shown in $fda(m) = sda(4m) + 2 \times sda(4m+1) + 4 \times sda(4m+2) + 8 \times sda(4m+3)$ **Equation 9. Serial term for a four Input-bit Implementation**. fda(m) is the serial term for single bit implementation. Maximum value of m in fda(m) is equal to (b/4).

$fda(m) = sda(4m) + 2 \times sda(4m+1) + 4 \times sda(4m+2) + 8 \times sda(4m+3)$ **Equation 9. Serial term for a four Input-bit Implementation**

For example, in our 10-tap 16-bit wide filter, a serial term will have 10 LUTs. These LUTs will be addressed by the following groups of addresses- ($x_{0,4m}, x_{0,4m+1}, x_{0,4m+2}$ & $x_{0,4m+3}$), ($x_{1,4m}, x_{1,4m+1}, x_{1,4m+2}$ & $x_{1,4m+3}$) and so on. The outputs of these LUTs are added together using an adder tree to get the serial output fda(m). In a parallel implementation, these 4 serial term fda's are added using an adder tree. This is shown in the subsequent equation. As shown in this equation, this adder tree shifts higher serial terms by 4 and then sums it up.

$y(n) = \sum_{m=0}^{b/4-1} 4^j \times fda(m)$ **Equation 10. The parallel terms of a four-input bit implementation**

The implementation of a 10-tap filter is shown in Figure 4. Four-bit Implementation of a FIR filter.

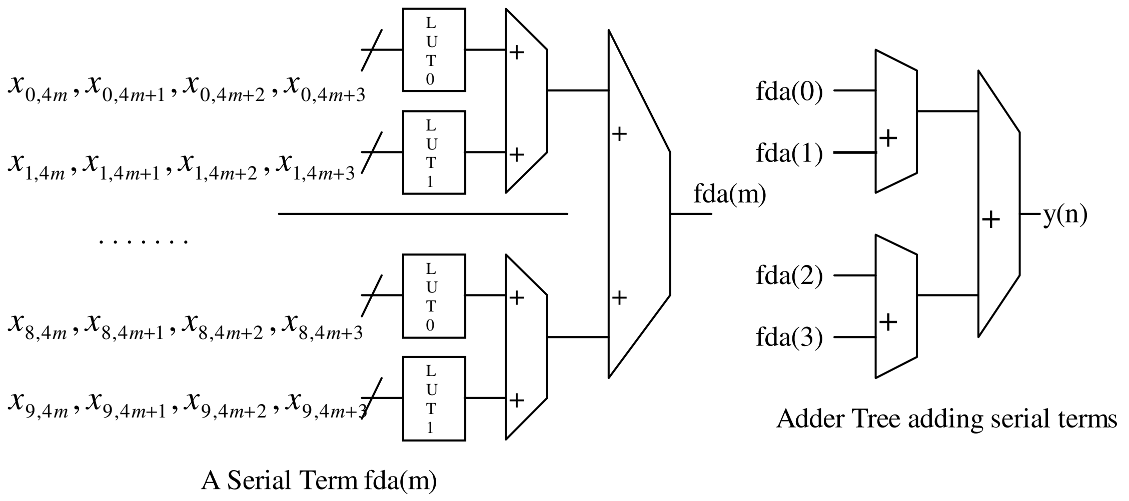

$x_{0,4m}, x_{0,4m+1}, x_{0,4m+2}, x_{0,4m+3}$
$x_{1,4m}, x_{1,4m+1}, x_{1,4m+2}, x_{1,4m+3}$
....
$x_{8,4m}, x_{8,4m+1}, x_{8,4m+2}, x_{8,4m+3}$
$x_{9,4m}, x_{9,4m+1}, x_{9,4m+2}, x_{9,4m+3}$
LUT 0
LUT 1
LUT 0
LUT 1
+
fda(m)
fda(0)
fda(1)
fda(2)
fda(3)
y(n)
A Serial Term fda(m)
Adder Tree adding serial terms

Figure 4. Four-bit Implementation of a FIR filter

The contents of single LUT addressed by ($x_{0,4m}, x_{0,4m+1}, x_{0,4m+2}$ & $x_{0,4m+3}$) are given in Table 3. LUT Contents for a four-bit implementation. The contents of other LUTs can be obtained similarly.

Table 3. LUT Contents for a four-bit implementation

LUT Address	LUT contents
0	0
1	C0
2	2xC0
3	3xC0
4	4xC0
5	5xC0
6	6xC0
7	7xC0
8	8xC0
9	9xC0
10	10xC0
11	11xC0
12	12xC0
13	13xC0
14	14xC0

15	15xC0

3.1.4 Impact of Design Choices

We chose the above-mentioned three ways of distributing memory, because they are best suited for a 4-input LUT architecture of Xilinx. We wrote a C-program to find out the number of LUTs required to implement one of the above mentioned implementations. The best implementation is chosen using this program. The pseudo code of this program is given below:

```
calculate_best_implementation_in_total_luts(taps, input_bus_width, output_bus_width) {
  minimum_of(total_luts_in_one_bit_implementation, total_luts_in_two_bit_implementation,
total_luts_in_four_bit_implementation);
}
total_luts_in_one_bit_implementation :=
total_luts_in_serial_terms+total_luts_for_adder_tree_in_serial_term+total_luts_for_adder_tree_shifted_b
y_1_for_all_serial_terms;
total_luts_in_two_bit_implementation :=
total_luts_in_serial_terms+total_luts_for_adder_tree_in_serial_term+total_luts_for_adder_tree_shifted_b
y_2_for_all_serial_terms;
total_luts_in_four_bit_implementation :=
total_luts_in_serial_terms+total_luts_for_adder_tree_in_serial_term+total_luts_for_adder_tree_shifted_b
y_4_for_all_serial_terms;
```

Some of the calculations done from that program are shown in Table 4. LUTs required to implement FIR in different methods. buswid. in the Table is the width of input and the constant. one, two and four are abbreviations for One-bit, Two-bit and Four-bit implementations respectively. We see that the for different combinations we have different methods which are best. The savings in some typical cases can go up to 20% if one makes the right design choice, which our tool does.

Table 4. LUTs required to implement FIR in different methods

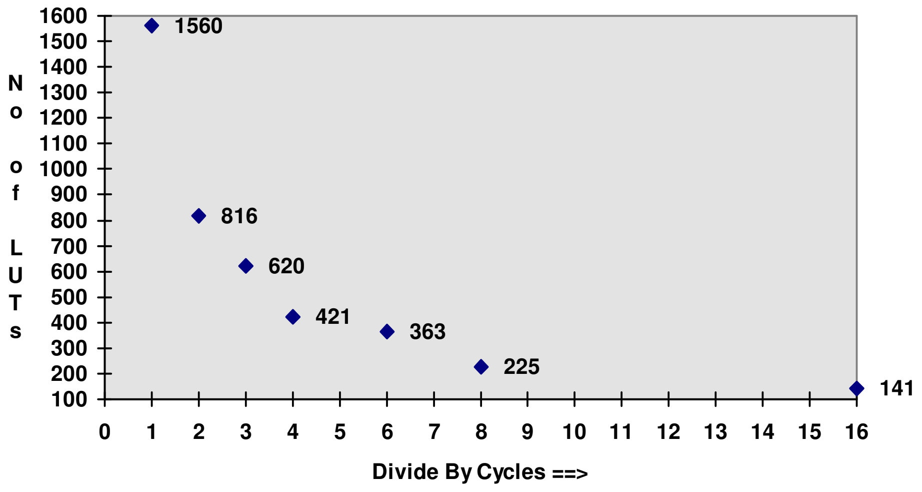
buswid->	8			12			16			32		
taps	one	two	four	one	two	four	one	two	four	one	two	four
6	318	266	285	678	552	580	1161	929	966	4392	3416	3489
7	326	350	335	690	726	679	1177	1225	1130	4424	4520	4073
8	326	354	383	690	732	775	1177	1233	1290	4424	4536	4649
10	493	453	486	1037	929	978	1768	1560	1625	6631	5703	5832

4. Designing for a given throughput

Assume our system speed is S for a fully parallel and pipelined implementation. If a user needs a lower data throughput, we need not implement it fully in parallel thus saving area. We can use a scaling accumulator at the output of a serial implementation to get the correct output. If *divide-by* is the number of cycles, one has to wait for each output, the data throughput speed will be (S/divide-by). We implement the minimum number of serial terms in parallel which obtains the given throughput. This is done for different Design Choices such as One-bit, Two-bit and Four-bit implementations. The best implementation is then chosen and then presented in a graphical form to the user for him to make a correct choice. As an example, we show the LUTs required to implement a 10-tap 16-bit wide filter for different throughputs using different Design Choices in Table 5. The best implementations are presented in a graphical form to the user as shown in Graph 1. By examining the graph, user can make an appropriate selection and choose the implementation which best suits his criteria.

Table 5. 10-Tap 16 bit wide filter with different throughputs

Design Choices=> Divide-by Cycle	One	Two	Four
1	1768	1560	1625
2	920	816	849
3	698	620	663
4	473	421	438
6	363	366	----
8	252	225	----
16	141	----	----



Graph 1. 10 Tap 16 bit-wide FIR filter

5. Summary

In this paper, we presented a method to efficiently design a FIR filter for Xilinx FPGAs. We used different Distributed Arithmetic techniques to do the implementation. The different design choices were then explained with the help of a design. Further., an algorithm was shown which can be used by the user to get an optimal design for the maximum throughput. For lower data throughputs, another algorithm was presented. It uses the same design techniques and scaling accumulators to find an optimal solution. Finally we presented results from our tool for FIR filter design which is based on the above mentioned algorithms. The results demonstrate that our tool essentially allows a user to pick a point in the area/speed curve and generate a design accordingly for FPGAs.

Reference 1. Mintzer L., "FIR filters with the Xilinx FPGAs", FPGA'92, ACM/SIGDA Workshop on FPGAs, pp 129-134.