Implementation of Time-Multiplexed Sparse Periodic FIR Filters for FRM on FPGAs

Syed Asad Alam and Oscar Gustafsson
Department of Electrical Engineering, Linköping University
SE-581 83 Linköping University, Sweden
E-mail: {asad, oscarg}@isy.liu.se

***Abstract*—Frequency-response masking (FRM) is a set of techniques for lowering the computational complexity of narrow transition band FIR filters. These FRM use a combination of sparse periodic filters and non-sparse filters. In this work we consider the implementation of these filters in a time-multiplexed manner on FPGAs. It is shown that the proposed architectures produce lower complexity realizations compared to the vendor provided IP blocks, which do not take the sparseness into consideration. The designs are implemented on a Virtex-6 device utilizing the built-in DSP blocks.**

I. Introduction

Finite-length impulse response (FIR) filters are digital filters whose impulse responses to a Kronecker delta input are finite, i.e., they settle to zero in a finite number of sample intervals [1]. The difference equation that defines the output of an Nth-tap (filter order: $N-1$) FIR filter in terms of its input is:

$$y(n) = \sum_{i=0}^{N-1} h(k)x(n-k) \tag{1}$$

where $y(n)$ is the output sequence, $x(n)$ is the input sequence and $h(k)$ are the coefficients.

The complexity of FIR filters mainly depends on the number of multiplications, which can be seen in (1) is proportional to the filter order. The filter order is in turn roughly inversely proportional to the width of the transition band, i.e., the difference between the passband edge and the stopband edge. Clearly, this will lead to high filter orders for narrow transition bands.

Linear-phase FIR filters exhibit symmetry or anti-symmetry of the filter coefficients. For ease of exposition we assume symmetry, i.e.

$$h(i) = h(N-1-i),\ i = 0, 1, \ldots, N-1. \tag{2}$$

this halves the total number of multiplications.

Further complexity reduction can be achieved, e.g., by employing so called frequency-response masking techniques where the key building block is a periodic filter. This leads to a periodic filter with period L, i.e., the transfer function can be written as $H(z^L)$, there are $L-1$ zeros between every non-zero coefficient [1]–[3]. We will refer to such a filter as a *periodic sparse* filter.

While the design of FRM and related structures have received considerable attention, only a few attempts of dedicated implementation have been reported [4]–[6]. Furthermore, there has been no attempt to study the relationship between resource utilization, time-multiplexing and periodic sparsity of filters. Reference [4] studies multiplierless narrow band frequency masking filters with a fixed tap count. The authors of [5] focused on reducing memory fetches between the FPGA and an external memory. In [6], the authors compare FRM filters with conventional, sharp FIR filters developed using Xilinx core generator tool. However, that work only considers fully parallel filters. Apart from FRM implementation, authors in [7] compare the impact of different sparsity factors and placement of zeros on FPGA utilization while implementing a 200-order fully parallel FIR filter.

Time-multiplexing is an efficient way to fully utilize FPGA resources for cases where the sample rate is lower than the maximum obtainable clock frequency. Especially, it will help in reducing the number of multipliers required. This, combined with sparseness helps to significantly reduce the hardware complexity. The current work proposes an architecture, incorporating both sparseness and time-multiplexing and compares it against the FIR core provided by Xilinx. Field-programmable gate array (FPGA) is used for implementation purposes.

This paper is arranged as follows: Section II briefly describes FRM FIR filters. Section III explains different aspects of FPGA for implementing signal processing algorithms. Section IV describes the proposed architectures for implementing time-multiplexed sparse periodic filters on FPGAs. Finally Sections V and VI present the results and conclude this paper, respectively.

II. Frequency-Response Masking FIR Filters

Frequency-response masking is a set of techniques for realizing filters with very narrow transition bands. Basically, there are two different structures that have been utilized. The first is for narrowband or wideband filters, i.e., where the passband edge for a lowpass filter is close to 0 rad or π rad. In this case it is possible to use a periodic filter, i.e., a filter with inserted zeros. This filter will have a periodic frequency response with multiple passbands, one which is the required. This filter is cascaded with a second filter that removes the unwanted passbands. The first filter is called a model filter or a band-edge shaping filter, while the second filter is called a masking filter. This structure is sometimes referred to as an interpolated FIR (IFIR) filter [3].

The second approach can be used for arbitrary bandwidths and is composed of a complementary pair of periodic filters. These are cascaded with two masking filters, each removing the unwanted periodic passbands in the resulting stopband. The outputs of these filters are then added to form the final passband and stopband [2].

While the filter order increases when using these types of filters, the arithmetic complexity decreases as many of the filter coefficients are zero. The arithmetic complexity varies with the period and can be minimized by a careful selection. In this work we focus on the implementation of the periodic filters since much work has already considered the implementation of regular FIR filters.

III. Implementing Filters on FPGAs

FPGAs have, in addition to the general purpose LUTs and registers, a number of dedicated blocks for different specialized functions. Current state-of-the-art FPGAs, like Xilinx Virtex-5 and Virtex-6, have dedicated blocks, called DSP blocks, for implementing multipliers, multiply-accumulates (MACs) and multiply-adds (MADs). These DSP blocks are very efficient in implementing the convolution operation which is at the center of filter operation.

The DSP blocks can also be deeply pipelined by using the available internal registers. Virtex-6 devices also have a pre-adder, which helps in realizing symmetric, linear phase FIR filters. The FPGAs also have resources for implementing memories. There are two types, dedicated memory blocks called Block RAMs (BRAMs) and memories provided by Look-Up Tables (LUTs) called Distributed RAMs (DRAMs).

This combination of DSP and memory blocks (BRAM or DRAM) provide an opportunity to efficiently map time-multiplexed FIR filter architectures. The addition of periodic sparsity to the complexity equation helps to reduce the number of these DSP blocks required to realize the whole filter with a slight overhead in the control circuitry.

IV. Proposed Architectures

The proposed architectures, both pipelined and non-pipelined, are explained in this section. Let N and L be defined as before, i.e., number of taps and period of sparsity. Let M denote the time-multiplexing factor such that a new sample arrives every Mth cycle, N_{dm} number of data memories and D_{dm} denote the depth of each data memory.

Since FIR filters are commonly linear phase, focus is on trying to utilize coefficient symmetry. However, this architecture can also be extended easily to implement non-linear, non-symmetric filters. The design methodology is as follows: an array of ROMs holds the non-zero coefficients and an array of RAMs is used to store data. The DSP blocks are used to implement the convolution function.

For time-multiplexed filters, there has $M-1$ cycles between each input. This indicates that current and previous inputs need to be saved in memories. Thus, each delay element in the path of inputs in direct form representation can be replaced by a memory of depth $D_{\text{dm}} = M$. With regards to

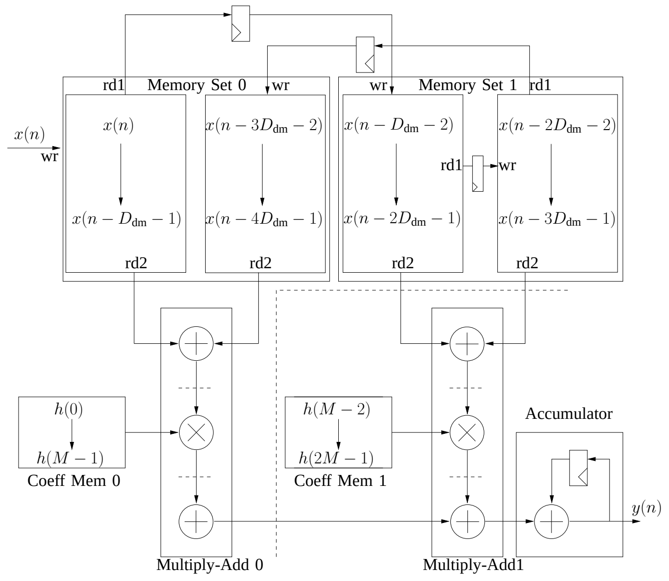
Memory Set 0
Memory Set 1
rd1
wr
rd2
$x(n)$
$x(n - D_{\text{dm}} - 1)$
$x(n - 3D_{\text{dm}} - 2)$
$x(n - 4D_{\text{dm}} - 1)$
$x(n - D_{\text{dm}} - 2)$
$x(n - 2D_{\text{dm}} - 1)$
$x(n - 2D_{\text{dm}} - 2)$
$x(n - 3D_{\text{dm}} - 1)$
$h(0)$
$h(M-1)$
Coeff Mem 0
$h(M-2)$
$h(2M-1)$
Coeff Mem 1
Accumulator
$y(n)$
Multiply-Add 0
Multiply-Add1

Fig. 1. Time-multiplexed architecture for FIR filters.

coefficients, instead of one coefficient per multiplier, there are M coefficients in a ROM. This architecture is shown in Fig. 1 for a $4M$-tap filter with symmetrical coefficients, where the arrows on the data memory show transfer of data between memories.

The box around the pre-adder, multiplier and adder shows that all these elements can be combined in one DSP element of Virtex-6 [8]. The final accumulator will also be implemented in the DSP block due to its support for fast 48-bit addition [8]. For the data memories, distributed RAM is used because of short length of these. Each data memory should be able to concurrently support 1 write and 2 reads, where first read operation is for generating the taps and the second one is for copying the oldest data in each memory to the next memory. Each memory is implemented as a *circular buffer*, where *write* and *read* pointers control the two operations. To exploit symmetry, the data memory array is divided into two halves, where one memory from each half is combined to form one memory set. The total number of sets equal the number of coefficient memories and is given by:

$$\left\lceil \frac{N-1}{M \times 2} \right\rceil \tag{3}$$

All the memories in the first half are read in a top-down manner and all in second half are read in bottom-up manner. These two different read sequence are controlled by a *read counter*. This counter is then incremented (decremented for the other memory half) by one to read out all the data. To copy the oldest data in each memory to the next, the write pointer is updated, after every write operation, to the oldest data. This makes this data appear on the output port of the shared read/write port and is registered in the next clock cycle. This register is directly connected to the input port of the next memory. A write enable writes this data at the appropriate

TABLE I
EQUATIONS FOR TIME MULTIPLEXED, PERIODIC SPARSE FILTERS.

Description	Equation
New tap count (N_L)	$(N-1)\times L+1$
Data memory count (N_{dm})	$\left\lceil \frac{N_L-1}{M\times L} \right\rceil$
Coefficient memory/Data memory set count	$\left\lceil \frac{N_L-1}{M\times L\times 2} \right\rceil$
DSP count	$\left\lceil \frac{N_L-1}{M\times L\times 2} \right\rceil + 1$
Data memory depth (D_{dm})	$M\times L$
Middle memory depth	$\frac{N_L - L\times M\times(N_{\text{dm}}-2)}{2}$

time.

However, there are two issues when N is not an integer multiple of M. First, the depth of last memory was less than M. Second, quite a few indices of data memories that need to be pre-added to exploit symmetry ended up in the same memory with no regularity. The solution to these problems is to make the head and tail memories equal. This results in either one or two middle memories which were of shorter length. In the case of one middle memory, the memory was split into two small memories. This is done to make data available for pre-addition easily and caused the data memory count to increase by one. The depth of the middle memories is now given by (3)

$$\frac{N - M\times(N_{\text{dm}}-2)}{2} \tag{4}$$

The same architecture could also be mapped to sparse filters. There are a few key differences. First, the effective number of taps, denoted by N_L, increases because of interpolation by L. Secondly, the coefficient memory which would now hold only the non-zero coefficients. The size of the ROM would be the same as before. Finally, the data memory would now be larger and total number of data memories would be smaller. This is because with one coefficient memory holding only the non-zero coefficients, the range of coefficients stored in a ROM is larger. However, only the data indices corresponding to the non-zero coefficients are read out. This means, that read counter would be incremented/decremented by a factor of L. The equations showing the new filter order and those that govern different resources are given in Table I. The $+1$ in DSP count in the table indicates that the accumulator is implemented in a DSP slice.

This architecture can be pipelined deeply. The dotted lines in Fig. 1 show the points where pipelining registers can be added. The registers after the *pre-addition*, *multiplication* and *add* are available in the DSP blocks [8]. In fact, even the non-pipelined architecture can also use the registers after pre-add and multiplication. The effective pipelining register is only after the final adder in the DSP block. The other pipelining after the second memory could either be implemented as a separate register or included within the data memory. As shown in Fig. 2, the extra registers approach is selected to keep the read counter simple. This pipelining, when extended

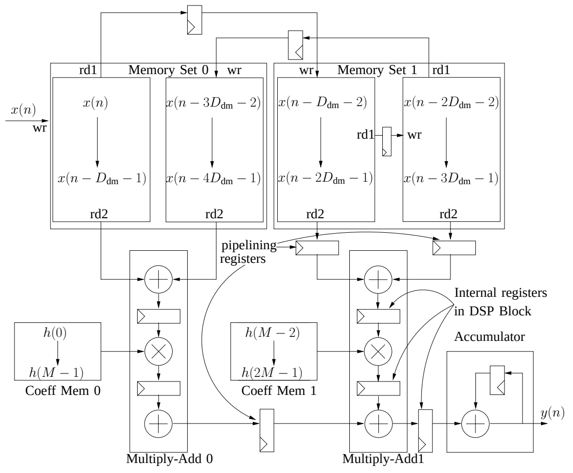

Fig. 2. Pipelined time-multiplexed architecture for sparse periodic FIR filters.

for larger filter orders, would result in two pipeline register in the memory set M_2, three registers in M_3 and k pipeline registers in the memory set M_k.

Pipelining could also have been implemented by retiming the architecture. That involves changing the number of delays between the memories. This, in turn, would have needed different reading scheme to copy data to the next memory, something which would require more specialized memory like quad-port distributed memories or block RAMS provided by Xilinx [9], [10].

There is one exception to the above design methodology. One extra middle tap is introduced when N is odd. This was handled by one extra flip flop between the two halves of data memory array. This middle tap along with the middle coefficient is fed to the multiplier of the DSP block used as an accumulator. The output of this multiplier is used, instead of a zero, as the initializing value of the accumulator, thus saving one extra DSP slice. This arrangement is shown in Fig. 3. To properly pipeline this tap, m registers were added after the middle tap in the pipelined architecture with m coefficient memories.

V. RESULTS

The results are based around a 41-tap lowpass filter. This is then implemented with varying time-multiplexing factors as well as periods. The data and coefficient word length are fixed at 18 bits which fit the 25×18 multiplier of the DSP block [8]. All the VHDL code is generated by MATLAB and synthesized by Xilinx ISE to a Virtex-6 XC6VLX75T-2 FPGA. Both the straight forward architecture in Fig. 1 as well as the pipelined in Fig. 2 are implemented with the middle tap extension. For comparison reasons we generated time-multiplexed FIR filters using Xilinx core generator. All results reported are obtained after placement and routing. Timing results were obtained by running static timing analysis on the placed and routed design.

Figure 4 shows the results for varying time-multiplexing factor with $L = 5$. It is clear that the proposed architectures result in a significantly reduced use of DSP blocks. The

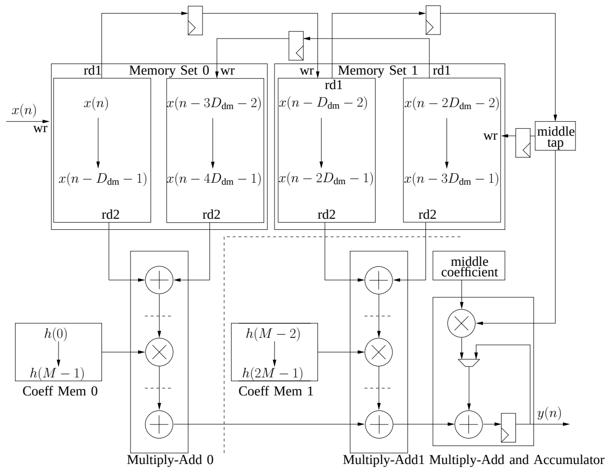

Fig. 3. Time-multiplexed architecture using the final accumulator DSP block for the middle tap.

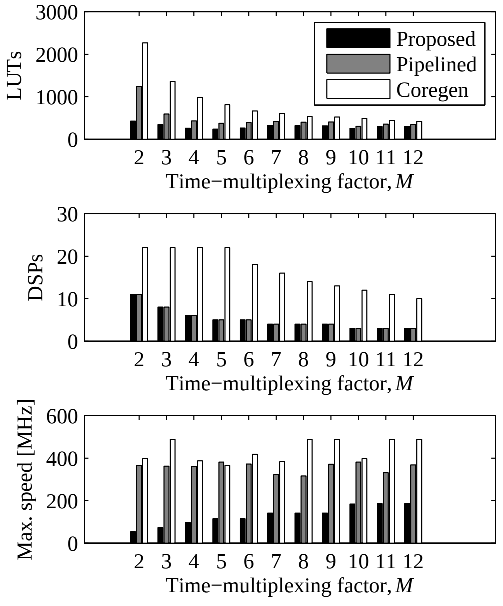

Fig. 4. Synthesis results for varying time-multiplexing factor: (above) number of LUTs, (middle) number of DSPs, and (below) maximum clock frequency.

graphs show significant improvement of speed between the two proposed architectures due to pipelining. However, the speeds achieved are, for most cases, slightly lower than the one achieved by Xilinx core generator. This could be improved by pipelining the control signals which are currently in the critical path.

Similar advantage is visible in Fig. 5, where the results for varying period is shown with $M = 5$. Also, the number of DSP blocks required by the proposed design is constant. This is because, as L increases, so does N_L, keeping the DSP count constant. The only effect is the increase in data memory depth. It is also worthy to note that the number of DSP blocks required by Xilinx core generator flattens out at a certain point, as a number of multipliers end up with only zero coefficients and are removed.

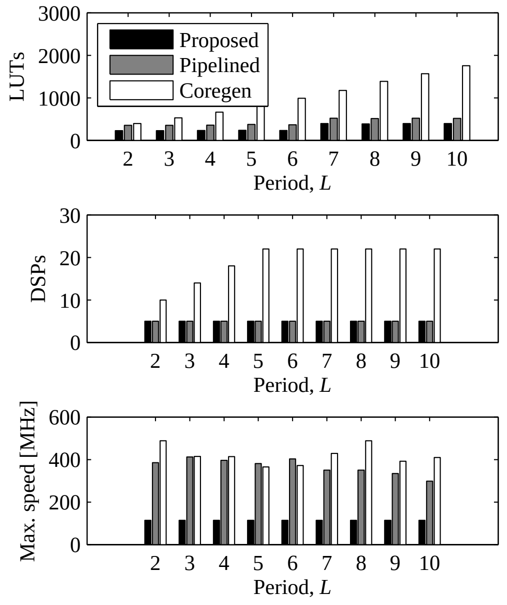

Fig. 5. Synthesis results for varying period: (above) number of LUTs, (middle) number of DSPs, and (below) maximum clock frequency.

VI. Conclusion

In this work we have presented architectures for time-multiplexed sparse periodic filters, found in frequency-response masking filters. The different design trade-offs were discussed and results show that the complexity is significantly lower compared to FIR filter generators not optimized for sparse periodic filters.

References

[1] L. Wanhammar and H. Johansson, *Digital Filters*. Department of Electrical Engineering, Linköping University: Linköping University, 2007.

[2] Y.-C. Lim, "Frequency-response masking approach for the synthesis of sharp linear phase digital filters," *IEEE Trans. Circuits Syst.*, vol. 33, no. 4, pp. 357–364, 1996.

[3] T. Saramäki, T. Neuvo, and S. Mitra, "Design of computationally efficient interpolated FIR filters," *IEEE Trans. Circuits Syst.*, vol. 35, no. 1, pp. 70–88, 1998.

[4] Y. Lian, "FPGA implementation of high speed multiplierless frequency response masking FIR filters," in *Proc. IEEE Workshop Signal Processing Syst.*, Lafayette, LA, 2000, pp. 317–325.

[5] Y. C. Lim, Y. J. Yu, H. Q. Zheng, and S. W. Foo, "FPGA implementation of digital filters synthesized using the FRM technique," *Circuits Syst. Signal Processing*, vol. 22, no. 2, pp. 211–218, 2003.

[6] S. Li and J. Zhang, "Efficient FPGA implementation of sharp FIR filters using the FRM technique," *IEICE Electronics Express*, vol. 6, no. 23, pp. 1656–1662, Dec. 2009.

[7] S. G. Patronis and L. S. DeBrunner, "Sparse FIR filters and the impact on FPGA area usage," in *Proc. Asilomar Conf. Signals Syst. Comput.*, Pacific Grove, CA, Oct. 2008, pp. 1862–1866.

[8] Xilinx, *Virtex-6 FPGA DSP48E1 Slice User Guide*, Sep. 2009. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-6.htm

[9] ——, *Virtex-6 FPGA Memory Resources User Guide*, Jan. 2010. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-6.htm

[10] ——, *Virtex-6 FPGA Configurable Logic Block User Guide*, Sep. 2009. [Online]. Available: http://www.xilinx.com/support/documentation/virtex-6.htm