An Efficient parallel design of FFTs in FPGAs

Sudip K. Nag and Hare K. Verma
Xilinx Inc.
2100, Logic Drive, San Jose. CA 95124

1. Introduction

Fourier Transform converts information in time domain to frequency domain. The digital equivalent of this using finite number of time samples at a time is known as DFT (Discrete Fourier Transform) and is represented as

$$y_k = \sum_{n=0}^{N-1} x(n) \times W_N^{kn} \quad \text{where } W_N = e^{-j(2\pi/N)}.$$

y is the frequency domain output and *x* is the time domain input. FFT (Fast Fourier Transform) is a method to calculate the frequency domain samples efficiently. FFT cleverly exploits the repetitive values of *W*. This paper presents a novel FFT design method for FPGA using radix-2 butterflies as a core. The method comprises of two parts:

- designing an optimal radix-2 butterfly
- Using multiple radix-2's to design full FFT to obtain the desired performance.

2. Design of a Radix-2 Butterfly

In this section we first describe a promising work done recently [2]. We then present our enhancement to achieve larger speeds with relatively small area overhead.

2.1 Previous Work

A radix-2 butterfly has two inputs x_m and x_n and two outputs y_m and y_n. For an *N* point FFT, one way of implementing radix-2 is illustrated by the following set of equations. In these equations suffixes *R* and *I* are for real and imaginary parts respectively.

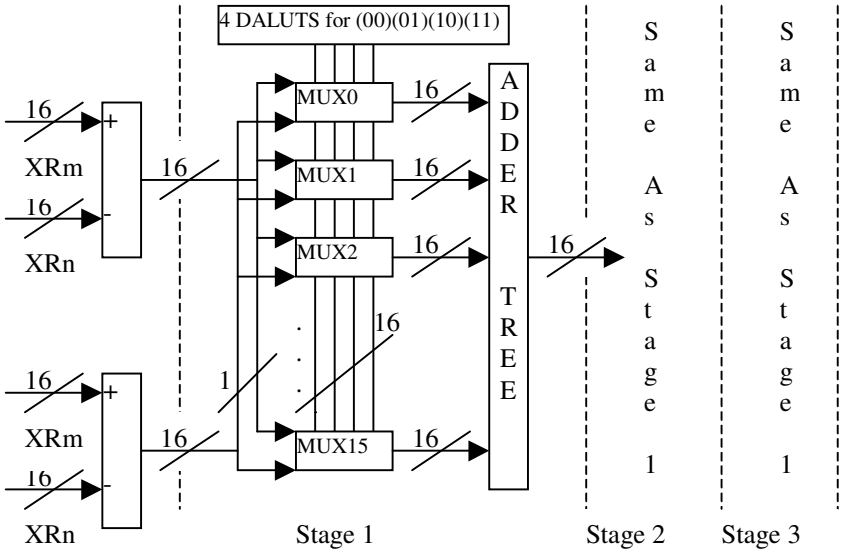
$$\begin{aligned} &= (x_{Rm} - x_{Rn}) \times \cos\theta_k + (x_{\mathrm{Im}} - x_{In}) \times \sin\theta_k \\ &+ j[(x_{Rm} - x_{Rn}) \times -\sin\theta_k + (x_{\mathrm{Im}} - x_{In}) \times \cos\theta_k] \end{aligned}$$

$$y_n = (x_m - x_n) \times W_N^{\,k}$$

$$y_m = x_m + x_n = x_{Rm} + x_{Rn} + j(x_{\mathrm{Im}} + x_{In})$$

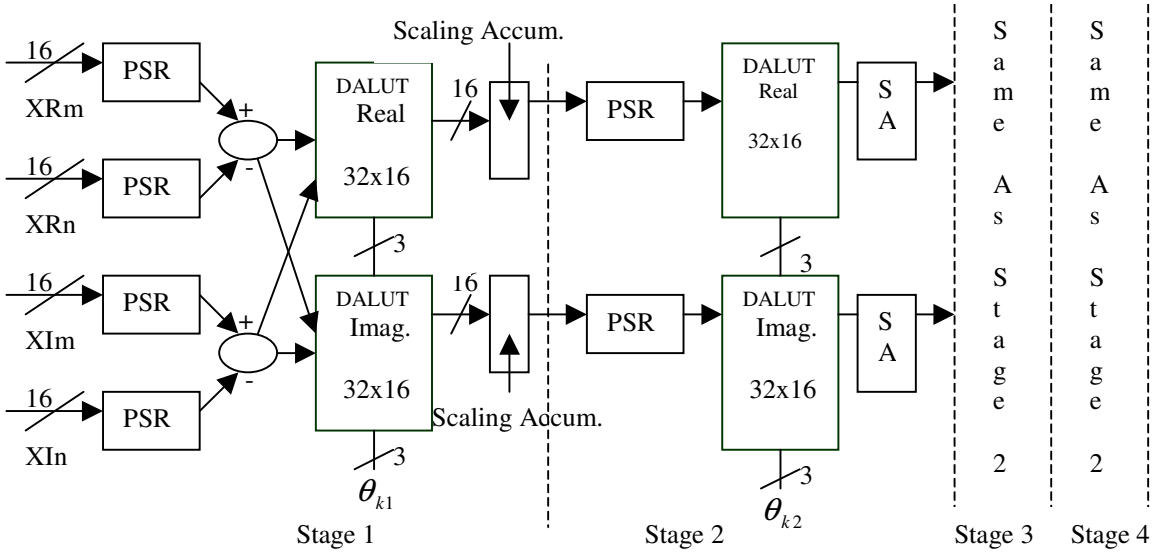
From the above equations we can see that y_m can be easily implemented using 2 adders. y_n can be implemented in FPGA using Distributed Arithmetic (DA) techniques, which are ideal for implementing constant coefficient filters in FPGAs. This technique utilizes DALUTs (Distributed Arithmetic LUT) which contain pre-computed sums of partial products for combinations of three variables: $(x_{Rm} - x_{Rn})$, $(x_{\mathrm{Im}} - x_{In})$ and θ_k. For an *N* point FFT, we need N/2 values of θ_k. In other words, θ_k has k bits where $k = \log_2(N/2)$. If we implement the real and imaginary parts of y_n in two DALUTs, the memory size becomes extremely large for large FFTs. For example, an 8192 point FFT for a sine-cosine accuracy of 16 bits, we will need two 16384 deep and 16 bit wide DALUT.

In [2], an elegant approach is used to decompose θ_k such that each part handles θ_k variation at different levels of coarseness. This allows replacing a very large DALUT by a set of smaller DALUTs (e.g. a 32 word deep) resulting in huge area savings. These stages are pipelined for the final result. Figure 1 shows an implementation for an 8192 point FFT. For



4 DALUTs shown in this diagram have pre calculated constants for the input patterns of (00) (01)(10)(11). These 4 DALUTs are repeated for the real and imaginary outputs for y_n . In the diagram the buswidth for data and constant is 16. The implemen- tation shown here is to be repeated twice, one for real and the other of imaginary outputs.

Figure 2. Efficient radix-2 implementation using DALUTs for N=8192



PSR is Parallel to Serial Register ; SA is Scaling Accum.

Figure 1. Previous radix-2 implementation using DALUTs for N=8192

more information on this technique, refer to [2]; this describes a serial-mode implementation of the same.

2.2 A New Approach for Speed-up

Let us analyze the design of a radix-2 butterfly for an *N*-point FFT. Assume the bus widths for the input and output data is *b*. Assume that we store *c* bits of accuracy for the DALUTs, which store the calculated constants. A parallel implementation of the previous technique (Figure 1) requires that we remove the PSR and have *b* times as many DALUTs in each stage. The outputs of these *b* DALUTs are added using an adder tree at the end of each stage. In this case, the speedup will be *b* times. However the extra area required, if we neglect PSR and adder

tree, will be *(b-1)a* DALUTs, where *a* is the number of stages in the DA implementation. When we use 32-word DALUTs, *a* will be equal to $\log_2(N/2)/3$. This is a simple parallel mode implementation.

Our new approach achieves the same *b* times speed up with much less area overhead. The key observation is that the various stages of the pipeline in the above approach have the same θ_k bits in all the DALUTs. Also the *b* DALUTs in each stage can have only one out of four possible input values of (x_m, x_n): (00), (01), (10) and (11). Therefore, instead of having *b* DALUTs each one of which gets addressed by two data inputs and three angle bits, we can have four DALUTs, one for each possible values of (00), (01), (10) and (11) for the two bits of inputs, x_m and x_n. Each of these new DALUTs gets addressed by θ_k alone. For each output we will need a multiplexer, which selects the correct output using the two input bits from the four DALUTs. This produces significant area savings without sacrificing the timing. In each stage of the new implementation, *c* LUTs, each 16 words deep, constitute a DALUT. There are four DALUTs in each stage. The number of stages used in the new approach becomes $\log_2(N/2)/4$. Such an implementation has been shown in Figure 2. This is an efficient parallel mode implementation.

2.3 Area Savings Calculation

In this section, we compare the area required to implement the three implementations: serial mode, simple parallel mode and efficient parallel mode. The implementations are discussed only for the output y_n.

2.3.1 Serial Mode

A 32 deep DALUT is used in this mode. The number of DALUTs required for real and imaginary part will be *2*(k/3)*, each *c* bits wide. Two *c* bits wide scaling accumulators will use *c* CLBs. Four *b* wide PSRs will use *2*b* CLBs. The serial adders in the first stage use 4 CLBs. Therefore the total required CLBs will be *(2ck/3+c+2b+4).*

2.3.2 Simple Parallel Mode

DALUTs in this case will use *(2bck/3)* CLBs, *b* times the CLBs used in serial mode. Two adder trees will use *(b-1)*c* CLBs. The input stage adders will now use *2b* CLBs. Therefore the total number of CLBs required will be *(2bck/3+bc-c+2b).* We get a speed up factor of *b*.

2.3.3 Efficient Parallel Mode

For efficiency sake, the DALUT size is 16. Therefore the number of DALUTs required for real and imaginary part will be *4*(k/4)* with the width being *c*, which is a total of *ck* CLBs. Muxes need *bc* CLBs. Adder tree uses *(b-1)*c* CLBs. Therefore the total number of CLBs required is *(ck+bc+(b-1)c+2b).* We get a speed up factor of *b*.

For the example, *N*=8192, *k*=12, *b*=16 and *c*=16, we get 180 CLBs for serial mode, 2320 for simple parallel mode and 717 for efficient parallel mode. We can see a speed up of 16 in efficient mode, whereas area increases by a factor of 4.

3. FFT Design with efficient radix-2 units

Let us consider an FFT with *N*=1024, *k*=8, *b*=*c*=16. For our efficient radix-2 implementation, there are two stages, which take 2 cycles to compute the answer. For the full FFT calculation, we will therefore need to do a total of 5120 radix-2 operations. If only one radix-2 is available, it will take 10240 cycles to do a full FFT.

A key observation is that to speed up FFT execution further, an increase in the number of FFTs alone is not enough since the bottleneck is elsewhere. Let us assume that the coefficients are stored in two dual port memory blocks, one for real and one for imaginary part. Also assume it takes 2 cycles to read and write each data. In this case, the access to memory itself becomes the bottleneck.

3.1 Memory partitioning based solution

We can achieve speed up with multiple radix-2s if the memory is partitioned. This is especially practical when on-chip memory is used, which can be partitioned without any penalty. In this section we present a smart method to partition the memory and use multiple radix-2s in an optimal fashion.

In an FFT structure, the more advanced the stage is, the smaller is the size of memory that a particular radix-2 interacts with. For a particular partitioning of the memory (say *p* partitions), there exists a *critical* stage, at and beyond which *p* radix-2 structures can operate independently. In other words, after this critical stage (say *cstage*), data is never required from multiple partitions by a single radix-2. Mathematically, $p = 2^{cstage}$. Till the critical stage, only *p/2* radix-2s can operate in parallel. Figure 3 illustrates this further.

The two key design issues are: what should be the value of *p* and what should be the memory-partition/radix-2 interaction. Based on the previous discussion, we conclude that the optimal number of partitions is equal to the number of radix-2s available. The intent is to allow the radix-2s to independently access different memory partitions at and beyond the critical stage.

We represent memory partitions as R_V and radix-2s as X_V, where ***V*** is their numbers in binary. In Figure 3, we show an example where *p*=8. *cstage* is therefore 3. In any pre-critical stage *k (k<cstage),* we use those radix-2s (R_V) for which the *kth* bit of ***V*** is 0. Also R_V interacts with memory-partitions X_{V1} and X_{V2} where ***V1*** is equal to ***V*** and ***V2*** is same as ***V*** with the *kth* bit inverted. If we consider all the FFT stages, each X_{V1} interacts with those R_V partitions such that ***V*** is equal to ***V1*** or ***V*** can be derived from ***V1*** by inverting only 1 bit of ***V1***. For example in Figure 3, X*001* interacts with R*101*, R*010* and R*001*.

These well-defined interactions are used to design multiplexers to read and write data to the memory-partitions. These multiplexers use the stage as the select lines.

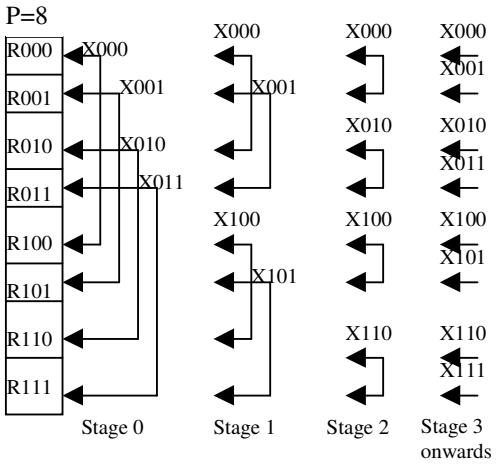


Figure 3. Radix-2-Memory Interaction

3.2 A Design for a maximal radix-2 utilization

Let us consider a case where *p*=4. The memory-partitions are R*00*, R*01*, R*10* and R*11*. The Radix-2 units are X*00*, X*01*, X*10* and X*11*. The interactions in different stages are as follows:

Stage 00:

X00<->(R00,R10), X01<->(R01,R11)
Stage 01:
X00<->(R00,R01), X10<->(R10,R11)
Stage 10 and Stage 11:
X00<->R00, X01<->R01,
X10<->R10, X11<->R11.

A generalized form of these interactions is used to design the multiplexers for accessing data from the memory partitions and writing into them.

3.3 Area Speed Calculations

In this section we explore the effect of our approach using multiple radix-2 optimally concurrent with memory partitions. We concentrate on the effect from area/speed viewpoint.

The results, given in Table 1 and Table 2, illustrate the optimality of our approach. The speed increases almost linearly with number of radix-2 used, thereby overcoming the memory access bottleneck.

Table1. Area Speed Trade-Off for 8192 FFT

No of Radix-2 Units	Area (No of CLBs)	Total no of cycles taken
1	723(1X)	106496(F)
2	1446(2X)	57344(1.9F)
4	2892(4X)	30720(3.5F)
8	5784(8X)	16384(6.5F)

Table2. Area Speed Trade-Off for 1024 FFT

No of Radix-2 Units	Area (No of CLBs)	Total no of cycles taken
1	640(1X)	10240(F)
2	1280(2X)	5632(1.9F)
4	2560(4X)	3072(3.5F)
8	5120(8X)	1664(6.5F)

4. Summary

Our contribution is two fold. Firstly we present a new approach to efficiently design radix-2 cores. Secondly we present a new method to utilize multiple radix-2s in an optimal fashion by distributing memories. This provides a mechanism to exploit the area speed tradeoff for FFTs as shown in sections 3.2 and 3.3. Using this method a system can generate an FFT to suit the user's area speed requirement.

References

[1] Alan Oppenheim and Ronald Schafer, "Digital Signal Processing", Prentice-Hall.

[2] Les Mintzer, "Large FFTs in a single FPGA", ICSPAT, 1996, pp. 895-500.