A Flexible and Fast Software Implementation of the FFT on the BPE Platform

Teo Cupaiuolo and Daniele Lo Iacono
Advanced System Technology
STMicroelectronics Italy
{teo.cupaiuolo,daniele.loiacono}@st.com

Abstract—**The importance of having an efficient Fast Fourier Transform (FFT) implementation is universally recognized as one of the key enablers for the development of new and more powerful signal processing algorithms. In the field of telecommunications, one of its most recent applications is the Orthogonal Frequency Division Multiplexing (OFDM) modulation technique, whose superiority is recognized and endorsed by several standards. However, the horizon of standards is so wide and heterogeneous that a single FFT implementation hardly satisfies them all. In order to have a re-usable, easily extensible and reconfigurable solution, most of the baseband processing is moving towards a software implementation: to this end several new Digital Signal Processor (DSP) architectures are emerging, each with its own set of differentiating properties. Within this context, we propose a software implementation of the FFT on the Block Processing Engine (BPE) platform. Several implementations have been investigated, ranging from a single instruction based approach, to others employing several instructions either in parallel or in pipeline. The outcome is a flexible set of solutions that leaves degrees of freedom in terms of computational load, achievable throughput and power consumption. The proposed implementations closely approach the theoretical clock cycles expected by dedicated hardware counterpart, thus making it a concrete alternative.**

Keywords-component; software Fast Fourier Transform (FFT); Software Defined Radio (SDR); vector processors, SIMD, VLIW architectures

I. INTRODUCTION

In the recent years a new approach is taking its way in the development of radio communication systems, known as software defined radio (SDR): in its widest meaning most parts of the modem (if not the entire) should be performed in software rather than in hardware. Application Specific Integrated Circuits (ASIC) based solutions face a severe drawback: with every new standard a design re-spin is needed, incurring in costly development phases and most important eventually affecting the time to market. This is especially true for wireless communication, where several standards are emerging, each targeting different needs and often competing in covering a market share. Each standard brings along different specifications which can be hardly met by a single implementation. Within this context, several new Digital Signal Processors (DSP) architectures have appeared, both from the academic and the industrial world. To be an effective replacement of their ASIC counterpart, these processors must retain the flexibility of a programmable approach and efficiently execute current and future standards, normally at a cost of a certain increase of area and power consumption. The Fast Fourier Transform (FFT) is one of the most common algorithms in a modem and represents a computationally intensive task. In this work the mapping of the FFT on the Block Processor Engine (BPE) [1] has been investigated. Starting from a revised instruction set architecture (ISA) that includes, among the others, a dedicated instruction for the radix-2 butterfly computation, we developed a variety of solutions based on the classic Cooley-Tukey algorithm: along the flexibility of supported FFT sizes, each implementation has different requirements in terms of computational load, achievable throughput and power consumption, thus enabling to cover different needs.

The remaining of the paper is organized as follows: Sec. II reviews the FFT algorithm and introduces the conventions used in the paper; Sec. III gives a brief overview of the BPE; Sec. IV details the profiling of the FFT on the BPE and Sec. V evaluates the results; lastly, Sec. VI concludes this paper.

II. THE FFT ALGORITHM

The FFT is an efficient algorithm for computing the Discrete Fourier Transform (DFT) of a complex sequence *x(n)*. Starting from

$$X(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}, k = 0,1,\ldots,N-1, \quad (1)$$

the *N*-points data sequence can be split in two *N*/2 data sequences corresponding to the even- and odd-numbered samples of *x(n)*, that is (after some rearrangements):

$$\begin{aligned} X(2r) &= \sum_{n=0}^{N/2-1} \left(x(n) + x\left(n+\frac{N}{2}\right) \right) W_{N/2}^{2nr} \\ X(2r+1) &= \sum_{n=0}^{N/2-1} \left(x(n) - x\left(n+\frac{N}{2}\right) \right) W_N^n W_{N/2}^{2nr} \end{aligned}, \quad (2)$$

where $r = 0,\ldots,1,N/2-1$. The terms $W_N^n = e^{-j2\pi n/N}$ are called twiddle factors. The decomposition can be applied recursively until each DFT is reduced to 2-point DFT,

978-3-9810801-8-6/DATE12/©2012 EDAA

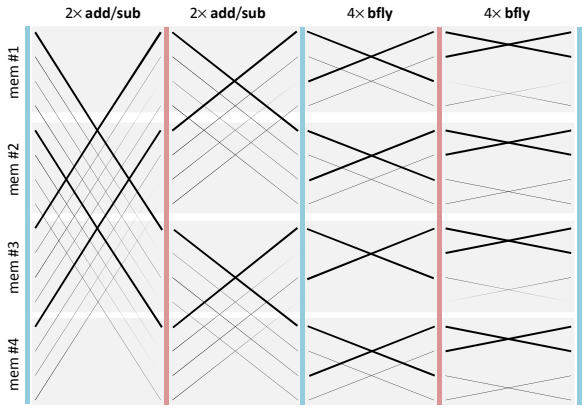


Fig. 1. Radix-2 FFT SFG (N = 16 samples)

commonly known as radix-2 butterfly. The signal flow graph (SFG) of the radix-2 FFT transform is shown in Fig. 1 (N = 16). The FFT provides the same result as the DFT, but the computation complexity is reduced from $O(N^2)$ to $O(\log_2 N)$. Higher radix decompositions can be applied, e.g. radix-4, with the only limitation that the FFT size N in this case must be a power of the radix. In the remaining of the paper the term butterfly will be more generally referred to as a generic radix-r based computation, unless otherwise specified.

III. The Block Processing Engine

The BPE is a mixed-grain vector processor: the template architecture is shown in Fig. 2. Its architectural characteristics have been discussed recently in [1]; here we will recall only the main concepts.

A. *The instruction set architecture*

The BPE ISA has two types of instructions: 1) the basic instructions (b-instruction) are devoted to flow control and are locally executed; 2) the dedicated instructions (d-instruction) are devoted to vector processing and are executed on the customizable dedicated unit bank (d-unit). When executing d-instructions, data vectors are allocated on the dedicated memory (d-memory) bank. The single instruction multiple data (SIMD) parallelism level can be tuned instruction-by-instruction: it is a degree of freedom left to the programmer to choose the number of units to eventually execute in parallel on a set of data. Such approach solves the resources underutilization faced by conventional SIMD architectures, with the further notable advantage of simplifying intra-vector manipulation and avoiding costly shuffling network. The BPE can be statically configured with a variable number of d-instructions, according to the need of a target application. The d-instructions belong to four families (in bold the related assembly instruction), each corresponding to one or more dedicated hardware units in the unit bank: 1) *arithmetic* instructions (**arith**), such as mac, add/sub, radix-2 butterfly and similar; 2) *vector* manipulation instructions, intra-vector operations, logic and bit operations (**vect**); 3) *communication* instructions (**comm**), to perform typical coarse-grain telecom operations (code generation, filtering and convolution); and 4) *math* instructions (**math**, based on the CORDIC operator) to compute 1/x, square root,

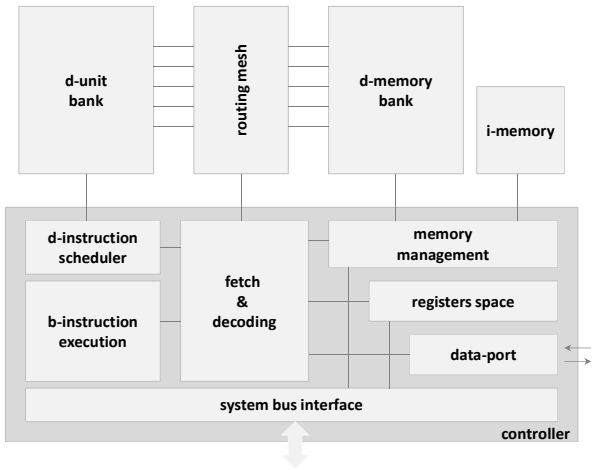


Fig. 2. The BPE template architecture

hyperbolic and trigonometric functions. The controller fetches and schedules instructions one after another until one of them requires resources that have been already allocated, as can be the case for a d-memory or another d-unit; it then waits until the execution of the instructions using those resources has been completed. This significantly reduces the control overhead of typical very long instruction word (VLIW) architectures while still allowing a similar parallelism degree. Such mechanism has the major benefit of being agnostic with respect to the latency of each d-unit, requiring the controller to be notified only when a resource has been released. A side benefit of such policy is that b-instructions executed right after the scheduling of d-instructions do not cause additional delay. The latter consideration inherently suggests that maximum efficiency can be reached only using vectors long enough to absorb b-instructions execution. The d-instructions can be concatenated through a run-time mesh as to build a block-diagram of pipelined instructions (i.e. the output of one instruction is the input of the subsequent one) that can bypass the d-memory bank, with consistent benefits in terms of throughput and power.

IV. FFT Implementation

A common approach to implement the FFT, is to have a single butterfly (or several butterflies in parallel) time-shared at each FFT stage, wherein a stage identifies a column of butterflies: for example, the radix-2 FFT of Fig. 1 has $\log_2 N = 4$ stages. Referring to the same figure, at odd stages the input memory (color blue) provides samples to be transformed and written to the output memory (color red); at even samples, the input and output memory swaps (ping-pong memory configuration). The BPE has native support for ping-pong buffering (see [1] for more details); further, the butterfly is simply an addition/subtraction and scaling operation that can easily mapped on the ISA. Thus, a radix-2 memory based FFT has been chosen as solution. Fixed-point simulations confirm that a 16-bit width resolution (for the real and imaginary part of the complex sample, respectively) combined with proper dynamic scaling at every stage, guarantees good system performance.

A. *The butterfly dedicated instruction*

Apparently, the most obvious choice for the dedicated butterfly instruction is a parallel architecture i.e. that processes two samples per clock cycle. Such architecture implies some critical drawbacks: 1) radix higher than two would not be supported, because the ISA format has a fixed number of inputs, specifically equal to three; 2) the d-memory bank requires being equipped with dual-port memories, which are both larger in terms of power and area. Overall, implementations based on several butterflies in parallel (as to increase the throughput), would exacerbate the memory requirements, making it rapidly unfeasible. Indeed, it is well know that when the number of butterflies increases, the FFT exhibits a memory conflict access [2]: to solve it, each in/output of the butterfly is (de)multiplexed. As discussed in Sec. III.A, the controller schedules one instruction per clock cycle and especially for small FFT sizes, the instruction scheduling and data propagation latency (between d-instructions) may become comparable to the vector length, thus heavily reducing the maximum achievable throughput. Therefore, a serial architecture for the butterfly d-instruction (**bfly**) has been selected: it elaborates one input per cycle and after an initial latency, every cycle one of the r outputs transform is computed. By this way, the butterfly computation (including the multiplication by the twiddle factor) can be described in assembler by cascading two dedicated instructions, the **bfly** and the **mul**.

B. *The FFT Signal Flow Graph*

The FFT SFG can be described by three nested loops: 1) one over the stages; 2) one over the butterflies sub-set; 3) and one that computes the butterflies belonging to the sub-set under evaluation. Here, by stage we identify a column of butterflies; the butterfly sub-set identifies the set of M-points FFT, where the last stage is made out of 2-points FFTs. At each stage the same computation needs to be performed (namely the butterfly): what changes is the way the samples are re-ordered and combined. The SFG is replicated by properly combining the set of b-instructions dedicated to memory pattern access management.

C. *Twiddle factors management*

The twiddle factors can be either computed on the fly (using **math**) or pre-stored in a memory and then later retrieved. Clearly, the second approach reduces the instruction load and computation overhead and is thus preferred. Similarly to the butterfly samples, twiddle can be accessed setting the proper memory access pattern with related b-instructions.

D. *Bit-reverse ordering the output*

The FFT generates scrambled outputs at every stage: for the radix-2 case, these are *bit-reverse*. For example, given $N = 4$, the input sequence $x_i = \{0,1,2,3\}_{10} = \{00,01,10,11\}_2$ is transformed, after the first stage, in the output sequence $X_i = \{0,2,1,3\}_{10} = \{00,10,01,11\}_2$. Bit-reverse addressing is handled by the BPE through a specific b-instruction, which is applied directly to the output memory: properly ordering the output does not require a dedicated phase as commonly happens, but can be done on the fly during the computation of the last stage.

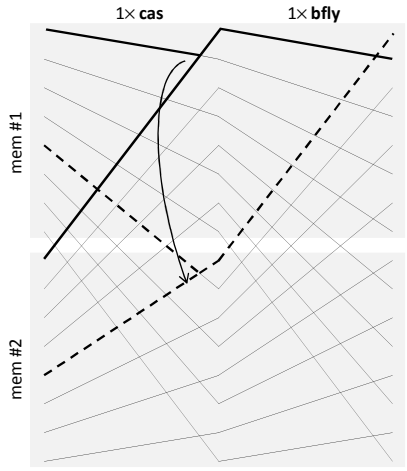


Fig. 3. Constant geometry FFT SFG: pipelined d-instructions

E. *Adding parallelism in the FFT computation*

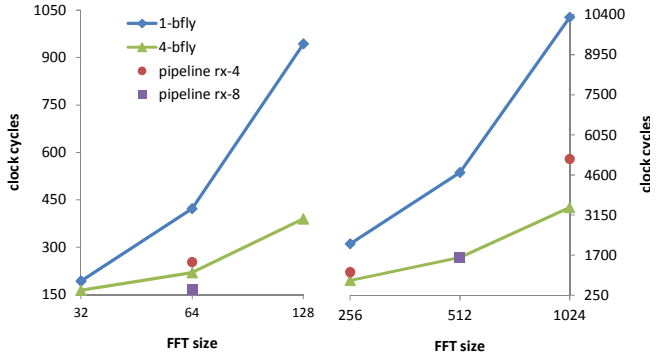
The FFT implementation discussed so far is based on a single butterfly instruction. In order to increase the achievable throughput, two more approaches have been investigated.

1) *Scheduling more butterfly d-instructions in parallel*

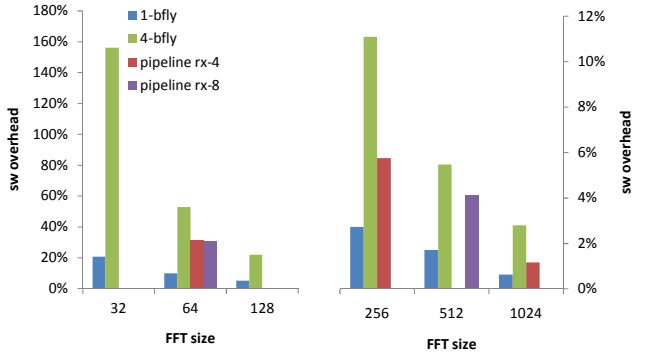
As described in Sec. IV.A, more butterflies computed in parallel can cause a data memory conflict. However, the serial architecture of the butterfly simplifies the memory requirements and management. Due to the FFT recursive decomposition, at every stage the number of butterflies that can be computed in parallel (p) without conflict equals $p = r^i$, $i = 0,1,\ldots,\log_r N-1$: in other words, the first stage can be processed by one butterfly, the second by two butterflies and so on. Accordingly, samples need to be stored in p memories in N/p consecutive locations. Such observation is highlighted in Fig. 1 for the case of four butterflies; note that in order to retain the same parallelism degree (four) for the stages preceding the 3-rd stage, two parallel butterflies are computed (rather than four serial ones) based on the combination of **add/sub** d-instructions. A further advantage of such approach is that a single twiddle memory can be shared along and within the stages by p butterflies and the code scales with the FFT size.

2) *Scheduling more butterfly d-instructions in pipeline*

The previous approach requires several basic instructions (both for the memory pattern management and the FFT SFG), which for small FFTs (less than 32 points) become comparable with the vector length, thus making it inefficient in terms of instructions scheduling and execution. Another way of parallelizing is along the horizontal axis by combining several d-instructions in pipeline: this is equivalent to a higher (serial) radix butterfly. Such approach is shown in Fig. 3, for two pipelined d-instructions: in this case a constant geometry SFG has been chosen (as opposite to the variable geometry discussed so far), which is equivalent to the previous with the advantage that it repeats



(a) FFT size vs clock cycles



(b) FFT size vs sw overhead

Fig. 4. Implementation results for various FFT sizes

itself at every stage (with the positive side effect of reducing the number of b-instructions). When considering a generic stage, the inputs of a radix-r butterfly are the outputs of r butterflies branches whose computation is equivalent to an add/sub operation conditioned by a signal selector. The approach can be extended to more stages in pipeline. It has to be noted that, compared to the parallel version, this has the disadvantage of requiring a different twiddle memory for each stage; further, the number of required d-instructions doubles with the number of stages in pipeline and it does not scale with the FFT size (meaning that different FFT sizes require different programs). However, for small FFTs, the data vector becomes long enough to entirely absorb instructions control overhead (b-instructions), recovering the clock cycles loss of the parallel implementation.

V. RESULTS

Fig. 4a details the required clock cycles for various FFT sizes (N = 16,…,512) for the parallel (p = 1,2,4) and pipeline (radix-4/8) version: it highlights that different implementations perform better depending on the FFT size, thus leaving freedom to choose the most appropriate solution for a given target application. As a performance metric, the overhead of the software implementation compared to the theoretical lower bound has been evaluated: the latter is given by the number of clock cycles required to compute an N-size FFT with p (serial) radix-r butterflies in parallel, that is $(N/p)\log_r N$ (a similar relation holds for the pipeline version). The outcome is shown in Fig. 4b: as expected the longer the vector length, the smaller the gap; further, for small sizes, the pipeline version effectively performs better, while for larger FFTs the overhead of the various implementations reduces to about a few percentage points.

TABLE I
COMPARISON WITH OTHER SIMILAR WORKS

ref.	architecture	FFT size	cycles (N = 1024)	power [µW/MHz]
[3]	Sandblaster	64 – 2048	2198	n.a.
[4]	Tensilica	512 – 8192	1812	n.a.
[5]	ASIP	128 – 1024	4526	190
this work	BPE [a]	16 – 1024	10305/3421/5180	108/129/272

[a] Implementation results for 1-bfly, 4-bfly and pipeline rx-4, respectively

VI. CONCLUSIONS

Similar to this work, several software implementations have been presented in the literature, either based on a baseband vector processor, or on an Application Specific Integrated Processor (ASIP). Among them, TABLE I shows the FFT implementation results on the vector DSPs Sandblaster [3] and Tensilica [4] (both 65 nm technology) and on the recent ASIP by Guan et.al. [5] (130 nm technology). The BPE has been synthesized with 65 nm STMicroelectronics CMOS technology and the power consumption has been estimated through post-synthesis netlist simulation.

It is difficult to make a straight and fair comparison simply based on clock cycles, especially among DPSs: each has its own set of properties that might stand out with certain algorithms rather than other. The described implementations have both from the throughput and power consumption point of view performances comparable with that of the literature. Indeed the BPE offers several degrees of flexibility in the FFT computation and the required clock cycles approaches in several cases the theoretical lower bound of a hardware counterpart, thus making it an appealing alternative.

REFERENCES

[1] T. Cupaiuolo and D. Lo Iacono, “Software Implementation of Near-ML Soft-Output MIMO Detection,” Washington, DC, USA, 30 November - 3 December, 2010, Software Defined Radio Forum 2010 (SDR'10)

[2] J. Baek and K. Choi, "New address generation scheme for memory-based FFT processor using multiple radix-2 butterflies," SoC Design Conference, 2008. ISOCC '08. International, vol.01, no., pp.I-273-I-276, 24-25 Nov. 2008

[3] B. Beheshti, "On Performance of LTE UE DFT and FFT Implementations in Flexible Software Based Baseband Processors", Proceedings of 2009 IEEE Long Island Systems, Applications and Technology Conference (LISAT2009), May 1, 2009, Farmingdale, New York.

[4] C. Rowen,. P. Nuth and S. Fiske, "A DSP architecture optimized for wireless baseband," *System-on-Chip, 2009. SOC 2009. International Symposium on* , vol., no., pp.151-156, 5-7 Oct. 2009

[5] X. Guan, Y.Fei and H. Lin, "Hierarchical Design of an Application-Specific Instruction Set Processor for High-Throughput and Scalable FFT Processing," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* , vol.PP, no.99, pp.1-13, 2011