Optimal FFT Implementation on the Carmel™ DSP Core

Gil Naveh, Yaniv Nana, Anat Hershko and Haim Granot, I.C.COM Ltd., Israel

Abstract - The Fast Fourier Transform (FFT) algorithm may be implemented efficiently when SW and HW efforts are combined. This article will discuss a very efficient SW implementation of the FFT algorithm on the Carmel DSP Core. The Carmel's flexible architecture (with efficient parallelism model) is coupled with its Configurable Long Instruction Word (CLIW™) technology to turn out the excellent performance figure of two cycles per butterfly. This is, additionally, an example of optimal implementation of a dual MAC (Multiply Accumulate) machine.

1. Introduction

FFT (Fast Furrier Transform) is an efficient algorithm that transforms discrete time domain signals into a frequency domain representation.

Because of it's complexity, the FFT algorithm has not been used in consumer real-time SW applications for many years. Recently, with the introduction of powerful DSP processors and new trends in digital communication, the FFT algorithm have entered the consumer arena. FFT is widely used today in applications like radars (the spectral analysis tool), Discrete Multi Tone – DMT - modems (modulator and demodulator), speech enhancement devices (filtering and spectral subtraction) , etc.

The FFT algorithm, which involves intensive computation, requires specific architectural features for real-time SW implementation. These features, which are provided by DSP manufacturers today, include : bit reversal addressing mode [1], [2], parallel addition and subtraction instructions [2], block floating point support [1] and special commands for FFT acceleration [1].

This article will discuss the implementation of Decimation In Time (DIT) FFT algorithm, on the Carmel DSP core. The Carmel's FFT support is presented as well as its support for block floating point and for quantization noise optimization.

2. The FFT Algorithm

The term "FFT" refers to the entire set of efficient Discrete Fourier Transform (DFT) algorithms. The general form of DFT is [3]:

$$X(k) = \sum_{n=0}^{N-1} x(n) w_N^{kn} \quad ; \ \text{k=0..N-1} \qquad (1.)$$

Where w_N^{kn}, the twiddle factor, is defined as:

$$w_N^{kn} = e^{-j2pnk/N} \qquad (2.)$$

The complexity of the direct computation is proportional to N^2 (the required number of complex multiplications). The FFT algorithms' main duty is complexity reduction by way of decomposing the DFTs into smaller DFTs in a recursive manner.

The most popular decomposition method is radix-2 DIT FFT. Here, the entire DFT sequence is decomposed into two smaller DFTs, which is further divided into two smaller DFTs, and so on. The recursion ends when the smallest size DFT is reached, a two-point-DFT called "butterfly". This method significantly reduces the complexity of DFT. The reduced complexity of the radix-2 FFT algorithms is proportional to $N*\log_2 N$, rather than N^2 of direct computation.

3. Decimation In Time FFT

In the DIT FFT, the DFT decomposition is performed on the sequence x(n) which is considered as the time domain sequence. Its flow graph for eight point sequences is shown in Figure 1.

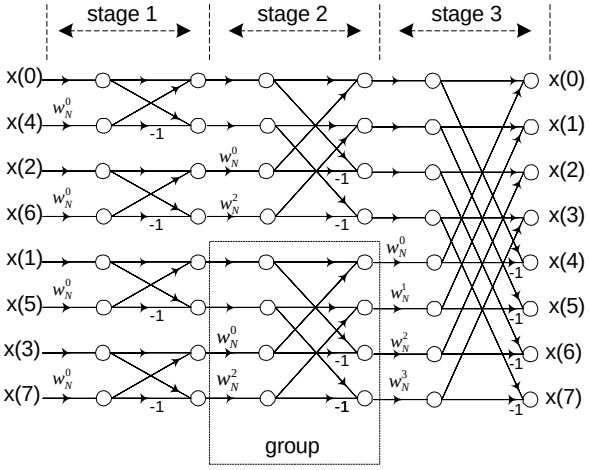

Figure 1 – DIT FFT flow graph

This flow graph immediately suggests the SW implementation of the algorithm. There are three nested loops:
(a) the stage loop (b) the group loop and (c) the butterfly loop.

As Figure 1 shows, there are $n = \log_2 N$ stages (the outer loop), each stage m, $m = 1..n$, has 2^{n-m} groups and each group has 2^{m-1} butterflies.

Other important considerations are:

(a) the input sequence is ordered in a bit reversal order and (b) the twiddle factors within each group are decimated version of the twiddle factors sequence $w_N^0, w_N^1, ..., w_N^{N/2-1}$.

By extracting a single butterfly from the FFT (Figure 2) and writing its input-output relations (equations 3 and 4), we can define its SW requirements.

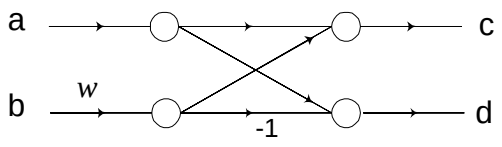

Figure 2 : General form of DIT butterfly

$$c = a + bw \qquad (3.)$$

$$d = a - bw \qquad (4.)$$

Note that all the numbers are complex. Eq. 3, therefore, requires one complex multiplication and one complex addition while eq. 4 requires one complex multiplication and one complex subtraction.

Note, additionally, that the multiplication terms (*bw*) in both equations are identical, leading to a single multiply operation common to both equations. Consequently, a single butterfly calculation requires a total of four multiply operations and four add/sub operations.

Bus capacity is an important factor in the implementation of DSP algorithms on DSP processors. The DIT butterfly in Figure 2, shows three complex memory read (a, b and w) and two complex memory write (c and d), a total of 10 memory access.

4. The Carmel

The Carmel is a fixed point DSP core. It has a parallel processing capabilities with a deep pipe and it is rated 120 MHz @ 2.5V.

The Carmel, which is a joint development of Siemens and I.C.Com, is a third generation DSP processor. Its architecture and instruction set are capable of executing 6 instructions and generate four addresses for data operands, all in one cycle. Although the Carmel has a 48-bit program memory, full parallelism is achieved by using CLIW™ — a special Configurable Long Instruction Word architecture. Each CLIW™ instruction may be composed of a parallel combination of up to four arithmetic instructions and two parallel load instructions. Additional advantage is that the Carmel is not typical load/store architecture. In the Carmel, memory is a valid source and/or destination of any instruction.

The Carmel architecture is shown in Figure 3. It has two parallel Execution Units (EU). Each of them has one MAC unit and one ALU unit. The left-hand side EU includes, in addition, an exponent unit and a barrel shifter. The Carmel

has 6x40 bit accumulators and an enhanced addressing unit (not shown in the figure), that supports a variety of addressing modes including bit-reversal for FFTs.

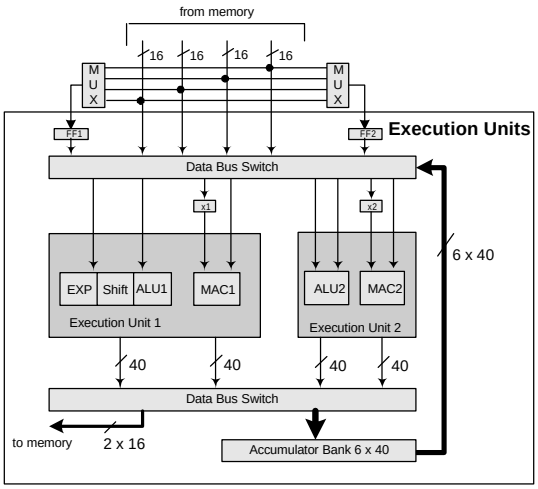

Figure 3 : Carmel architecture

The Carmel is capable of generating four addresses for memory data operands in one cycle. In order to increase the effective bus capacity, two latch registers (FF1 and FF2) have been introduced on the data buses. These registers can capture bus values for future use as EU sources. Additional register pair, x1 and x2, serve the MAC units. They latch the left-hand side input of each MAC unit until a new value is passed through. These two register pairs may be used as operands.

One of Carmel's design objectives was high computational efficiency. As MAC units are used intensively by most DSP algorithms, they were selected as targets for the efficiency improvement efforts. The goal was to reduce the number of cycles per algorithm down to the order of half the number of its multiply operations (full utilization of the two MAC units). This goal, which is quite easy to achieve in simple algorithms (e.g., block FIR or correlation) is very challenging in complicated algorithms like FFT. We have achieved this goal, by the Carmel, for many algorithms including LMS, complex LMS, IIR, Vector Quantization and FFT.

5. FFT implementation on the Carmel

From a computational point of view, the DIT butterfly should be executed on the Carmel within two cycles. The reasons are: (a) The Carmel is capable of executing two multiply/MAC operations and two add/sub operations, all during a single cycle and (b) the DIT butterfly computation requirement is four multiply operations and four add/sub operations.

A closer look at the memory bus shows that the DIT topology of Figure 1 requires ten memory access per butterfly while the Carmel provides only eight (four per cycle).

We solved this problem by selecting another DIT FFT topology (Figure 4) where the input vector of Figure 1 is reordered linearly [3]:

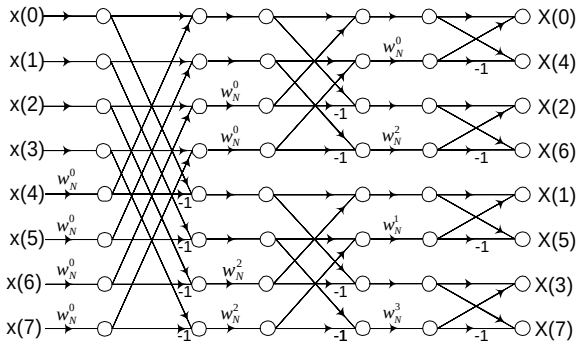

Figure 4 : Reordered input vector of DIT FFT

The reordered input vector called for additional changes in the flow diagram, but the important change was in the twiddle factors order. While they are ordered now in a bit reversal order, they are constant within each group (see Figure 4). Consequently, only a single twiddle factor should be fetched per group, next to the eight memory accesses per butterfly (read a, read b, write c, write d).

Now, that the DIT FFT topology is fitted to the Carmel's architecture, we can get into the implementation details. First, eq. 3 and eq. 4 will be rewriten in their real and imaginary forms:

$$c_r = a_r + w_r b_r - w_i b_i$$
$$c_i = a_i + w_r b_i + w_i b_r \qquad (5.)$$

$$d_r = a_r - w_r b_r + w_i b_i$$
$$d_i = a_i - w_r b_i - w_i b_r \qquad (6.)$$

Where the indexes r and i refer to the real and imaginary parts respectively.

Defining temporary variables as:

$$F_r \equiv w_r b_r - w_i b_i$$
$$F_i \equiv w_r b_i + w_i b_r \qquad (7.)$$

we get:

$$c_r = a_r + F_r$$
$$c_i = a_i + F_i \qquad (8.)$$

$$d_r = a_r - F_r$$
$$d_i = a_i - F_i \qquad (9.)$$

Equations 7-9 are implemented by the inner loop that is composed of two CLIW™ instructions.

Each CLIW™ instruction holds six CLIW™ slots (commands), five of which are used by the inner loop — four arithmetic operations (2 MAC and 2 ALU) and one move operation. The table below illustrates the implementation:

No	MAC1	ALU1	MAC2	ALU2	MOV1	MOV2
1	$A_4=w_r b_r$	$c_i=R\{FF1+A_5\}=a_i+F_i$	$A_5=w_i b_r$	$d_i=R\{FF1-A_5\}=a_i-F_i$		$FF2=a_r$
2	$A_2=A_4-w_i b_i=F_r$	$c_r=R\{FF2+A_2\}=a_r+F_r$	$A_5=A_5+w_r b_i=F_i$	$d_r=R\{FF2-A_2\}=a_r-F_r$	$FF1=a_i$	

- The command's destination is always on the left-hand side.
- A second equate sign is provided here (not in the SW) for clarification, using the above temporary variables.
- The sign R{.} stands for two's complement rounding.
- Uppercase letters represent CPU registers; lowercase letters represent memory variables (except '*w*').
- The twiddle factor '*w*' (fetched once per group) is placed in an accumulator.
- Memory variables are accessed using Addressing Unit pointers (not referred to in the table).
- Commands in *Italics* pertain to iteration k+1; other operations pertain to the k^{th} iteration (the current one).

The butterfly computation takes four cycles long, but due to SW pipelining it takes effectively only two cycles. A step by step description of butterfly computation is necessary for full understanding of the inner loop. It will require two passes over the loop.

(a) Starting with the *Italic* operations, that begin the butterfly, we assume that the current iteration is the k^{th}.

(b) In the first cycle, slot 1 and 3 relate to butterfly k+1 (the other slots are still busy with the k^{th} butterfly) and the products $w_r b_r$ and $w_i b_r$ are calculated and stored in the accumulators A_4 and A_5 respectively.

(c) In the second cycle, slots 1,3 and 5 relates to iteration k+1. Slots 1 and 3 complete the calculation of the temporary variables (F_r and F_i in A_2 and A_5 respectively). In order to avoid a memory bus overload situation, Slot 5 loads the real part of the input a (e.g., a_r) into the FF1 register for the next cycle.

(d) The k^{th} iteration is complete now. On the first cycle of iteration k+1, slots 2, 4 and 6 will relate to butterfly k+1. Slot 2 calculates the imaginary part of equation 8 and completes calculation of the output's imaginary part 'c'.

Note the efficiency of slot 2. First, it adds the 16-bit register FF1 (imaginary part of 'a') to the high part (bits 16 to 31) of the 40-bit accumulator A5. Then it performs a two's complement rounding of the sum to 16-bit, and

finally saves the 16-bit result in memory – all in single CLIW™ slot!

Similarly, slot 4 calculates the imaginary part of the output 'd' according to equation 9. In slot 6 the real part of 'a' is loaded into FF2 register (again, to avoid memory bus overload situation on the next cycle). On the second cycle, slot 2 and 4 complete calculation of the real parts of outputs 'c' and 'd' (according to equations 8 and 9). Now the butterfly computation is complete.

Due to the pipeline overlapping, the first iteration of the current group completes the last butterfly of the previous group at the same stage (in slots 2, 4 and 6). In the first butterfly iteration of the first group of each stage, slots 2 and 4 should be disabled since no relevant data is available (yet) to store in memory. Carmel's conditional execution mechanism is used to disable these slots.

6. Improved Block Floating Point FFT

As can be seen from Figure 4 and equations 3 and 4, each output node of each stage is a sum of two numbers. The dynamic range[1] of the output is growing by a factor of two.

Increasing the dynamic range of numbers, in fixed point DSP processors, may cause overflows during memory write. Few methods are available to cope with this problem in real time applications.

The most effective one is the Block Floating Point (BFP) technique. All the outputs of a certain stage, in this technique, are scaled down by a factor of two in case that the overflow probability is bigger than zero. In practice, the absolute value of all the outputs of each stage are compared (by HW) to a constant. If one of the outputs is greater than the constant, all the outputs of the next stage will be scaled down.

The Carmel provides a programmable register that holds the constant value. This register (patent pending) enables the user to optimize a "constant" value per application. A useful utilization of this tuning method can be to minimize[2] the number of scale-down stages, and consequently reduce the quantization noise[3] that is added during the FFT.

7. Summary

The DIT FFT implementation on the Carmel DSP core has been presented. It has been shown that the Carmel, by utilizing its CLIW™ technology, is capable of reaching the optimum performance for a dual MAC machine — two cycles per butterfly. This fact is reflected in the formula below, that calculates the Carmel's N-point FFT cycle count.

$$\text{Cycle count} = N*log_2N+5N/4+10*log_2N+4$$

A careful examination of this formula, immediately reveals Carmel's advantage over the competition.

References

[1] Motorola, *DSP56300 24-Bit Digital Signal Processor Family Manual*, Motorola, Inc. Semiconductor Products Sector, DSP Division, 1995.

[2] Analog Devices, *ADSP-21000 Family Application Handbook*, Vol. 1, 1995.

[3] A.V. Oppenheim, R. W. Schafer, *Digital Signal Processing*, Prentice-Hall International Editions, 1975.

[1] The dynamic range is the range between the maximum possible number and the minimum possible number

[2] By using signal statistics and the FFT size.

[3] Finite word length effects.