Application Report
ZHCA414 – Jan. 2012

基于 TMS320C64x+DSP 的 FFT 实现

冯华亮 DSP System

ABSTRACT

This application note introduces the implementation of FFT (Fast Fourier Transform) on TMS320C64x+ DSP family, the performance is also discussed.

摘要

本文介绍基于 TI TMS320C64x+ DSP 的 FFT（快速傅立叶变换）的实现，并讨论相关性能。

Contents

1 快速傅立叶变换介绍 3
1.1 基 2 FFT 3
1.2 基 4 FFT 6
1.3 混合基 4 和基 2 FFT 10
1.4 混合基 FFT 的常规 C 语言实现 13
1.5 快速傅立叶逆变换(IFFT) 17
2 TMS320C64x+ DSP 简介及 FFT 相关特性 17
3 基于 TMS320C64x+的 FFT 实现 21
3.1 C64x+ DSP 库里的 FFT 函数 21
3.2 对旋转因子访问的优化 22
3.3 蝶形运算的优化 25
3.4 Cache 冲突的优化 28
4 执行速度指标 29
5 数据缩放和精度 33
References 35

Figures

Figure 1. 基 2，N=8 点 FFT 6
Figure 2. 基 4，N=16 点 FFT 9
Figure 3. N=32 混合基 4 和基 2 FFT 12
Figure 4. TMS320C64x+ DSP 内核框图 18
Figure 5. C64x+ DSP 上 1024 点基 2 FFT 比特反转 20
Figure 6. C64x+ DSP 上 1024 点基 4 FFT 比特反转 21
Figure 7. FFT 函数命名规则 22
Figure 8. 优化实现中的基本蝶形运算 26
Figure 9. FFT 函数执行的指令周期数 30
Figure 10. IFFT 函数执行的指令周期数 32

Tables

Table 1. 基 2，8 点 FFT 序号比特反正 6
Table 2. 基 4，16 点 FFT 序号比特反转 9
Table 3. 旋转因子组和 FFT 级之间的关系 12
Table 4. 32 点混合基 FFT 序号比特反转 12
Table 5. C64x+ DSP 上用于 FFT 的指令 18
Table 6. C64x+ DSP 库里的 FFT 函数 21
Table 7. 可连续访问的旋转因子表 23
Table 8. FFT 函数执行的指令周期数 29
Table 9. IFFT 函数执行的指令周期数 31
Table 10. Cache “touch”对 DSP_fft16x16 性能的影响 32
Table 11. 数据动态缩放的 DSP_fft16x16 的性能 34

1 快速傅立叶变换介绍

序列 x(n), n = 0...N −1 的离散傅立叶变换 (Discrete Fourier Transform, DFT) X(k), k = 0...N −1 可定义为:

$$X(k) = DFT_N\left(x(n)\right) = \sum_{n=0}^{N-1} x(n) W_N^{kn}, \qquad k = 0,...., N-1$$

其中，$W_N^{kn} = e^{-j(2\pi/N)kn}$ 是旋转因子。

反离散傅立叶(Inverse DFT, IDFT)可定义为:

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k) W_N^{-kn}, \qquad n = 0,...., N-1$$

其中

$$W_N^{-kn} = e^{j(2\pi/N)kn}$$

它们的输入输出序列都是复数。如果我们定义一个基本的运算单位为两个复数相乘再相加，则直接根据以上公式进行 DFT 计算需要 N^2 次基本运算。当 N 比较大时，N^2 会变得非常大。这使得直接的 DFT 计算方法对很多应用来说都不现实。

1.1 基 2 FFT

然而，如果 N 是 2 的整数次方，一种快速傅立叶变换（Fast Fourier Transform, FFT）算法可显著的减少运算量。基 2 FFT 算法利用了旋转因子的以下周期性特性:

$$W_N^0 = e^{-j(2\pi/N)0} = e^{-j(0)} = \cos(0) + j\sin(0) = 1$$

$$W_N^{kN/2} = e^{-j(2\pi/N)kN/2} = e^{-j(\pi k)} = (\cos(-\pi) + j\sin(-\pi))^k = (-1)^k$$

$$W_N^{2kn} = e^{-j(2\pi/N)2kn} = e^{-j(2\pi/(N/2))kn} = W_{N/2}^{kn}$$

利用这些特性，可以把 N 点 DFT 分解为以下两个 N/2 点 DFT:

$$X(k)=\sum_{n=0}^{N-1}x(n)W_N^{nk}=\sum_{n=0}^{N/2-1}x(n)W_N^{nk}+\sum_{n=N/2}^{N-1}x(n)W_N^{nk}$$
$$=\sum_{n=0}^{N/2-1}[x(n)W_N^{nk}+x(n+N/2)W_N^{(n+N/2)k}]$$
$$=\sum_{n=0}^{N/2-1}[x(n)W_N^{nk}+x(n+N/2)W_N^{kN/2}W_N^{nk}]$$
$$=\sum_{n=0}^{N/2-1}[x(n)W_N^{nk}+(-1)^k x(n+N/2)W_N^{nk}]$$
$$=\sum_{n=0}^{N/2-1}[x(n)+(-1)^k x(n+N/2)]W_N^{nk}$$

让我们把输出序列 X(k), k= 0,…, N-1 分解成连个序列：

偶数序列: X(2r), r= 0,…,N/2-1

$$X(2r)=\sum_{n=0}^{N/2-1}[x(n)+(-1)^{2r}x(n+N/2)]W_N^{2nr}$$
$$=\sum_{n=0}^{N/2-1}[x(n)+x(n+N/2)]W_N^{2nr}$$
$$=\sum_{n=0}^{N/2-1}[x(n)+x(n+N/2)]W_{N/2}^{nr}$$
$$=DFT_{N/2}(x(n)+x(n+N/2))$$

奇数序列: X(2r+1), r= 0,…,N/2-1

$$X(2r+1)=\sum_{n=0}^{N/2-1}[x(n)+(-1)^{(2r+1)}x(n+N/2)]W_N^{n*(2r+1)}$$
$$=\sum_{n=0}^{N/2-1}[x(n)-x(n+N/2)]W_N^n W_N^{2nr}$$
$$=\sum_{n=0}^{N/2-1}\{[x(n)-x(n+N/2)]W_N^n\}W_{N/2}^{nr}$$
$$=DFT_{N/2}((x(n)-x(n+N/2))W_N^n)$$

按以上方法反复的分解 N 点 DFT 成 2/N 点 DFT 直到 N=2，使得 N 点傅立叶变换的运算复杂度由原来的 N^2 降到 $N\log_2 N$，这是非常显著的改进。这个算法也被称为频域抽取(Decimate-In-Frequency, DIF)FFT 算法。图 1 演示了 N=8 点 FFT 的分解运算过程。上面的两个箭头（每个箭头上都有一个数字“1”）表示两个数相加，下面的两个箭头（一个箭头上数字“1”，一个箭头上数字“-1”）表示两个数相减。后面的W_N^n表示加减的结果再与旋转因子W_N^n相乘。

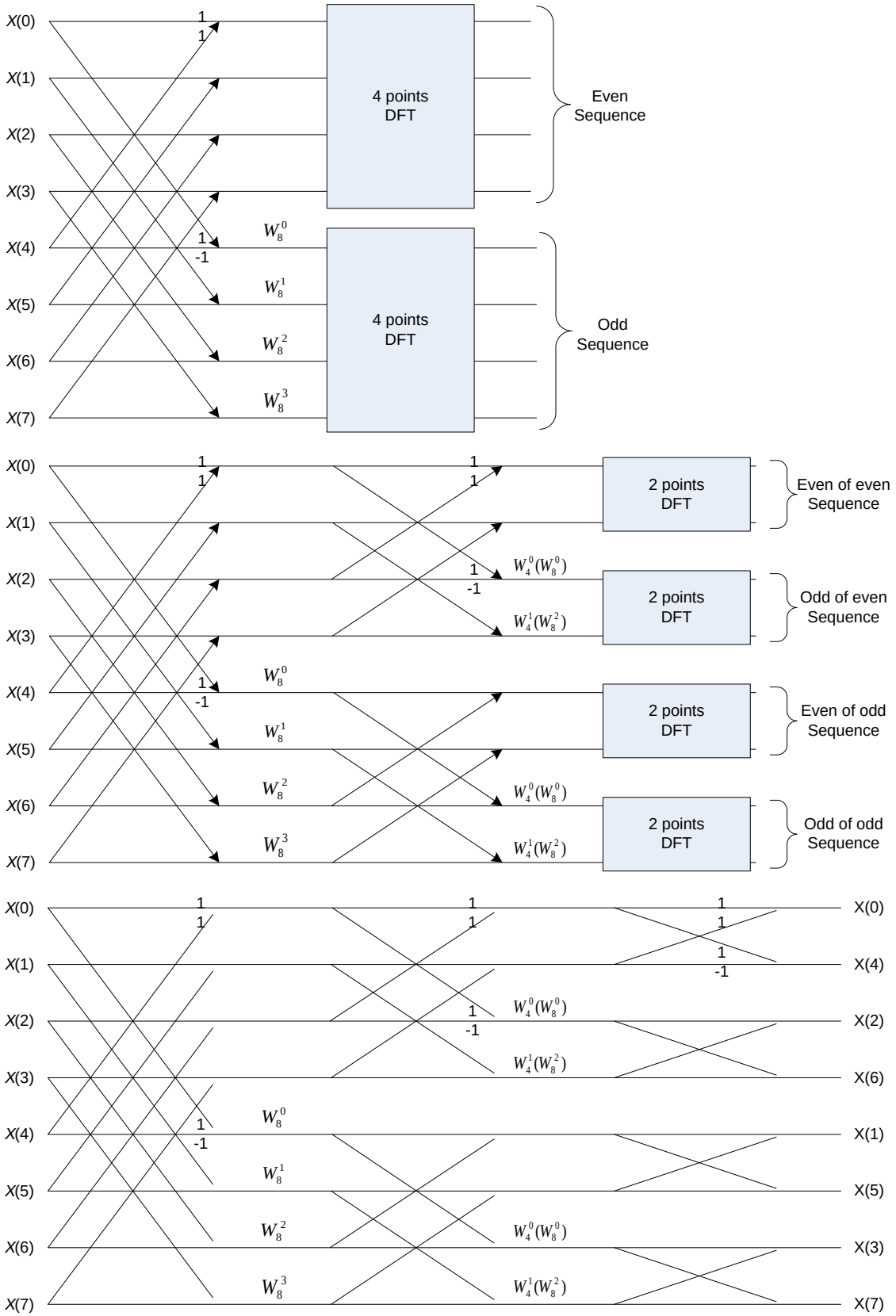

X(0)
X(1)
X(2)
X(3)
X(4)
X(5)
X(6)
X(7)
1
1
4 points
DFT
Even
Sequence
1
-1
W_8^0
W_8^1
W_8^2
W_8^3
4 points
DFT
Odd
Sequence
2 points
DFT
Even of even
Sequence
$W_4^0(W_8^0)$
$W_4^1(W_8^2)$
Odd of even
Sequence
Even of odd
Sequence
Odd of odd
Sequence
X(0)
X(4)
X(2)
X(6)
X(1)
X(5)
X(3)
X(7)

Figure 1. 基 2，N=8 点 FFT

从图 1 可以看出，输出数据的顺序被打乱了。这种乱序其实有规律，我们把顺序的序号用二进制数列在表 1 中的左边，把乱序的序号用二进制数列在表 1 中的右边。从表中我们可以看出，乱序序号的二进制码可由顺序序号的二进制码反转得到，所以，这种规律被叫做比特反转。

Table 1. 基 2，8 点 FFT 序号比特反正

Normal order of index n	Binary bits of index n	Reversed bits of index n	Bit-reversed of order index n
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

1.2 基 4 FFT

如果 FFT 点数 N 是 4 的整数次方，采用基 4FFT 算法可以进一步减少运算量。基 4 FFT 算法算法利用了旋转因子的以下周期性特性::

$$W_N^{kN/4} = e^{-j(2\pi/N)kN/4} = e^{-j(\pi k/2)} = (\cos(-\pi/2) + j\sin(-\pi/2))^k = (-j)^k$$

$$W_N^{2kN/4} = e^{-j(2\pi/N)2kN/4} = e^{-j(\pi k)} = (\cos(-\pi) + j\sin(-\pi))^k = (-1)^k$$

$$W_N^{3kN/4} = e^{-j(2\pi/N)3kN/4} = e^{-j(3\pi k/2)} = (\cos(-3\pi/2) + j\sin(-3\pi/2))^k = (j)^k$$

$$W_N^{4nk} = e^{-j(2\pi/N)4kn} = e^{-j(2\pi/(N/4))kn} = W_{N/4}^{nk}$$

利用这些特性，可以把 N 点 DFT 分解为以下 4 个 N/4 点 DFT:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_N^{nk} = \sum_{n=0}^{N/4-1} x(n)W_N^{nk} + \sum_{n=N/4}^{2N/4-1} x(n)W_N^{nk} + \sum_{n=2N/4}^{3N/4-1} x(n)W_N^{nk} + \sum_{n=3N/4}^{N-1} x(n)W_N^{nk}$$

$$= \sum_{n=0}^{N/4-1} [x(n)W_N^{nk} + x(n+N/4)W_N^{(n+N/4)k} + x(n+2N/4)W_N^{(n+2N/4)k} + x(n+3N/4)W_N^{(n+3N/4)k}]$$

$$= \sum_{n=0}^{N/4-1} [x(n) + x(n+N/4)W_N^{kN/4} + x(n+2N/4)W_N^{2kN/4} + x(n+3N/4)W_N^{3kN/4}]W_N^{nk}$$

$$= \sum_{n=0}^{N/4-1} [x(n) + (-j)^k x(n+N/4) + (-1)^k x(n+2N/4) + (j)^k x(n+3N/4)]W_N^{nk}$$

让我们把输出序列 X(k), k= 0,…, N-1 分解成四个序列, X(4r), X(4r+1), X(4r+2), X(4r+3), r= 0,…,N/4-1

$$X(4r) = \sum_{n=0}^{N/4-1}[x(n) + (-j)^{4r}x(n+N/4) + (-1)^{4r}x(n+2N/4) + (j)^{4r}x(n+3N/4)]W_N^{4nr}$$

$$= \sum_{n=0}^{N/4-1}[x(n) + x(n+N/4) + x(n+2N/4) + x(n+3N/4)]W_{N/4}^{nr}$$

$$= DFT_{N/4}[x(n) + x(n+N/4) + x(n+2N/4) + x(n+3N/4)]$$

$$= DFT_{N/4}\left([x(n) + x(n+2N/4)] + [x(n+N/4) + x(n+3N/4)]\right)$$

$$= DFT_{N/4}\left(A1(n) + B1(n)\right)$$

其中,

$$A1(n) = x(n) + x(n+2N/4)\ ; \quad B1(n) = x(n+N/4) + x(n+3N/4)$$

$$X(4r+1) = \sum_{n=0}^{N/4-1}[x(n) + (-j)^{(4r+1)}x(n+N/4) + (-1)^{(4r+1)}x(n+2N/4) + (j)^{(4r+1)}x(n+3N/4)]W_N^{(4r+1)n}$$

$$= \sum_{n=0}^{N/4-1}[x(n) - jx(n+N/4) - x(n+2N/4) + jx(n+3N/4)]W_N^n W_N^{4nr}$$

$$= \sum_{n=0}^{N/4-1}[x(n) - jx(n+N/4) - x(n+2N/4) + jx(n+3N/4)]W_N^n W_{N/4}^{nr}$$

$$= DFT_{N/4}\left([x(n) - jx(n+N/4) - x(n+2N/4) + jx(n+3N/4)]W_N^n\right)$$

$$= DFT_{N/4}\left(\{[x(n) - x(n+2N/4)] - j[x(n+N/4) - x(n+3N/4)]\}W_N^n\right)$$

$$= DFT_{N/4}\left([A2(n) - jB2(n)]W_N^n\right)$$

其中,

$$A2(n) = x(n) - x(n+2N/4)\ ; \quad B2(n) = x(n+N/4) - x(n+3N/4)$$

$$X(4r+2)=\sum_{n=0}^{N/4-1}[x(n)+(-j)^{(4r+2)}x(n+N/4)+(-1)^{(4r+2)}x(n+2N/4)+(j)^{(4r+2)}x(n+3N/4)]W_N^{(4r+2)n}$$

$$=\sum_{n=0}^{N/4-1}[x(n)-x(n+N/4)+x(n+2N/4)-x(n+3N/4)]W_N^{2n}W_N^{4nr}$$

$$=\sum_{n=0}^{N/4-1}[x(n)-x(n+N/4)+x(n+2N/4)-x(n+3N/4)]W_N^{2n}W_{N/4}^{nr}$$

$$=DFT_{N/4}\left([x(n)-x(n+N/4)+x(n+2N/4)-x(n+3N/4)]W_N^{2n}\right)$$

$$=DFT_{N/4}\left(\{[x(n)+x(n+2N/4)]-[x(n+N/4)+x(n+3N/4)]\}W_N^{2n}\right)$$

$$=DFT_{N/4}\left([A1(n)-B1(n)]W_N^{2n}\right)$$

其中,

$$A1(n)=x(n)+x(n+2N/4);\quad B1(n)=x(n+N/4)+x(n+3N/4)$$

$$X(4r+3)=\sum_{n=0}^{N/4-1}[x(n)+(-j)^{(4r+3)}x(n+N/4)+(-1)^{(4r+3)}x(n+2N/4)+(j)^{(4r+3)}x(n+3N/4)]W_N^{(4r+3)n}$$

$$=\sum_{n=0}^{N/4-1}[x(n)+jx(n+N/4)-x(n+2N/4)-jx(n+3N/4)]W_N^{3n}W_N^{4nr}$$

$$=\sum_{n=0}^{N/4-1}[x(n)+jx(n+N/4)-x(n+2N/4)-jx(n+3N/4)]W_N^{3n}W_{N/4}^{nr}$$

$$=DFT_{N/4}\left([x(n)+jx(n+N/4)-x(n+2N/4)-jx(n+3N/4)]W_N^{3n}\right)$$

$$=DFT_{N/4}\left(\{[x(n)-x(n+2N/4)]+j[x(n+N/4)-x(n+3N/4)]\}W_N^{3n}\right)$$

$$=DFT_{N/4}\left([A2(n)+jB2(n)]W_N^{3n}\right)$$

其中,

$$A2(n)=x(n)-x(n+2N/4);\quad B2(n)=x(n+N/4)-x(n+3N/4)$$

按以上方法反复的分解 N 点 DFT 成 N/4 点 DFT 直到 N=4，使得 N 点傅立叶变换的运算复杂度由原来的 N^2 降到 $N\log_4 N$。图 2 演示了 N=16 点 FFT 的分解运算过程。

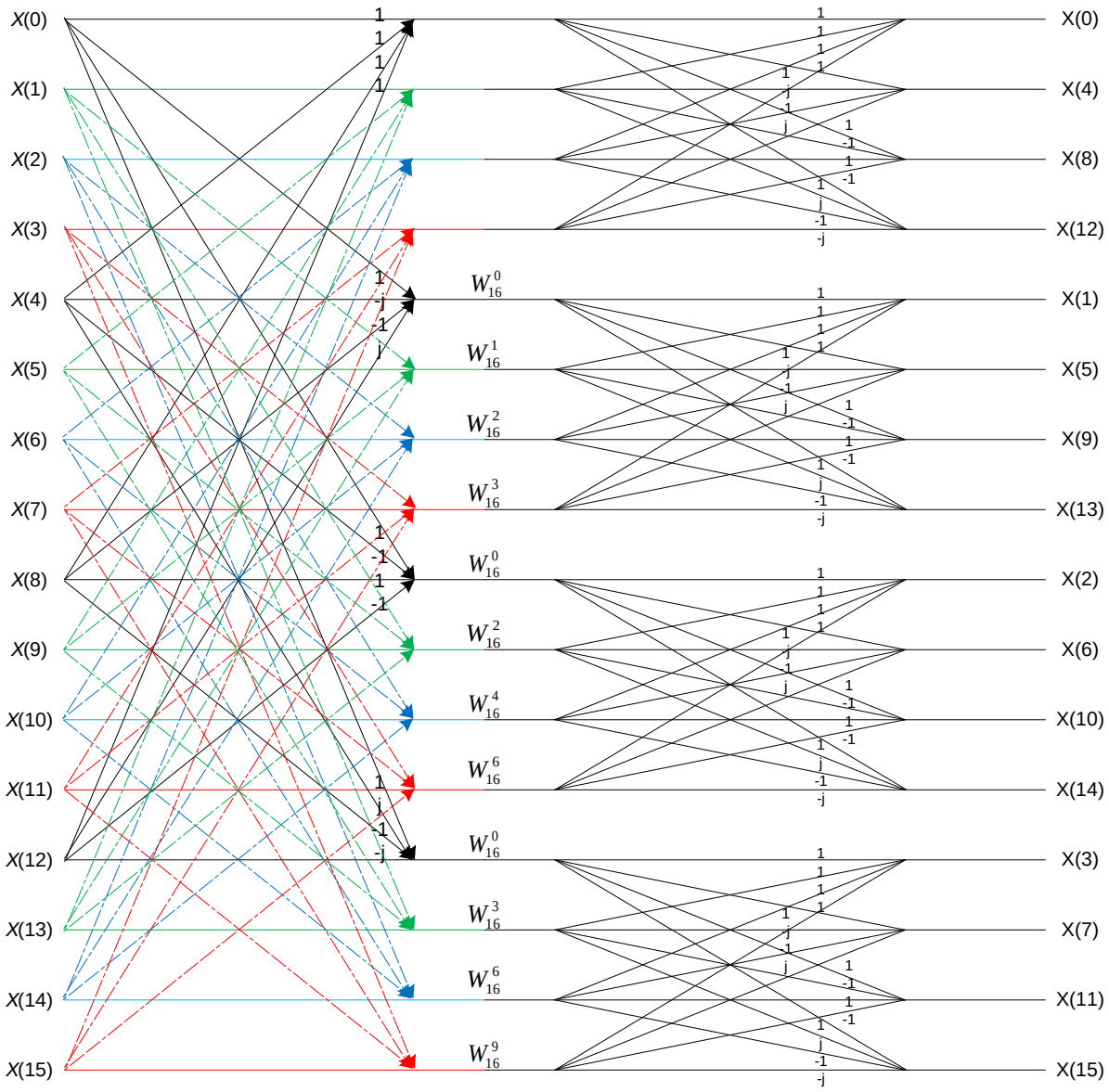


Figure 2.　基 4，N=16 点 FFT

从图 2 可以看出，输出数据的顺序也被打乱了。这种乱序其实有规律，我们把顺序的序号用二进制数列在表 1 中的左边，把乱序的序号用二进制数列在表 1 中的右边。从表中我们可以看出，乱序序号的二进制码可由顺序序号的二进制码以 2 比特为单位反转得到。

Table 2.　基 4，16 点 FFT 序号比特反转

Normal order of index n	Binary bits of index n	Reversed bits of index n	Bit-reversed of order index n
0	00 00	00 00	0
1	00 01	01 00	4

2	00 10	10 00	8
3	00 11	11 00	12
4	01 00	00 01	1
5	01 01	01 01	5
6	01 10	10 01	9
7	01 11	11 01	13
8	10 00	00 10	2
9	10 01	01 10	6
10	10 10	10 10	10
11	10 11	11 10	14
12	11 00	00 11	3
13	11 01	01 11	7
14	11 10	10 11	11
15	11 11	11 11	15

1.3 混合基 4 和基 2 FFT

如果 N= 4^M*2, 傅立叶变换可被分解成 $\log_4 N$= M 个基 4 FFT 和最后一级的一个基 2 FFT.

图 3 演示了 N=32 点 FFT 的分解运算过程。

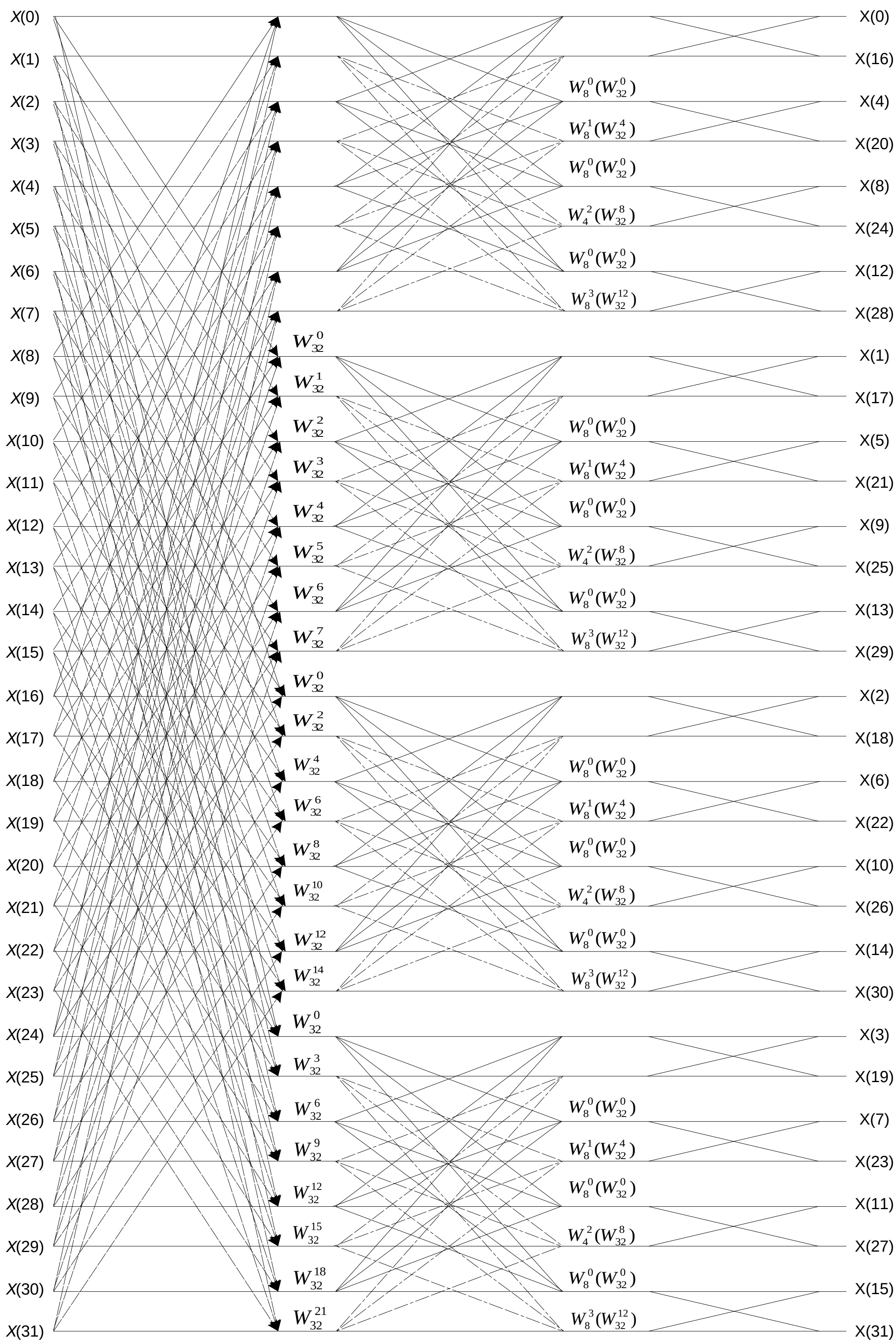

X(0)
X(1)
X(2)
X(3)
X(4)
X(5)
X(6)
X(7)
X(8)
X(9)
X(10)
X(11)
X(12)
X(13)
X(14)
X(15)
X(16)
X(17)
X(18)
X(19)
X(20)
X(21)
X(22)
X(23)
X(24)
X(25)
X(26)
X(27)
X(28)
X(29)
X(30)
X(31)
W_{32}^{0}
W_{32}^{1}
W_{32}^{2}
W_{32}^{3}
W_{32}^{4}
W_{32}^{5}
W_{32}^{6}
W_{32}^{7}
W_{32}^{0}
W_{32}^{2}
W_{32}^{4}
W_{32}^{6}
W_{32}^{8}
W_{32}^{10}
W_{32}^{12}
W_{32}^{14}
W_{32}^{0}
W_{32}^{3}
W_{32}^{6}
W_{32}^{9}
W_{32}^{12}
W_{32}^{15}
W_{32}^{18}
W_{32}^{21}
$W_8^0(W_{32}^0)$
$W_8^1(W_{32}^4)$
$W_8^0(W_{32}^0)$
$W_4^2(W_{32}^8)$
$W_8^0(W_{32}^0)$
$W_8^3(W_{32}^{12})$
$W_8^0(W_{32}^0)$
$W_8^1(W_{32}^4)$
$W_8^0(W_{32}^0)$
$W_4^2(W_{32}^8)$
$W_8^0(W_{32}^0)$
$W_8^3(W_{32}^{12})$
$W_8^0(W_{32}^0)$
$W_8^1(W_{32}^4)$
$W_8^0(W_{32}^0)$
$W_4^2(W_{32}^8)$
$W_8^0(W_{32}^0)$
$W_8^3(W_{32}^{12})$
$W_8^0(W_{32}^0)$
$W_8^1(W_{32}^4)$
$W_8^0(W_{32}^0)$
$W_4^2(W_{32}^8)$
$W_8^0(W_{32}^0)$
$W_8^3(W_{32}^{12})$
X(0)
X(16)
X(4)
X(20)
X(8)
X(24)
X(12)
X(28)
X(1)
X(17)
X(5)
X(21)
X(9)
X(25)
X(13)
X(29)
X(2)
X(18)
X(6)
X(22)
X(10)
X(26)
X(14)
X(30)
X(3)
X(19)
X(7)
X(23)
X(11)
X(27)
X(15)
X(31)

Figure 3. N=32 混合基 4 和基 2 FFT

值得注意的是，在第一级中，每个蝶形运算的旋转因子都不一样；在第二级，四组蝶形运算使用相同的旋转因子。下一级每一组中的不同蝶运算仍使用不同的旋转因子。重复以上规则，可得到下面的表：

Table 3. 旋转因子组和 FFT 级之间的关系

级数	每组中蝶的个数	蝶形组的个数	蝶的总数数
1	N/4	1	N/4
2	N/16	4	N/4
…	…	…	…
$\log_4 N$	1	N/4	N/4

FFT 的实现代码利用以上关系来减少旋转因子的访问，即每一级中相同的旋转因子只读取一次，然后给所有使用这个相同旋转因子的蝶形运算用。

由图 3 可看出，数据输出的顺序被打乱了。表 4 可帮我们理解这种乱序的规则。它将正常序号二进制编码以 2 比特为单位反转。因为总比特个数不是偶数，最后的一个比特单独反转。

Table 4. 32 点混合基 FFT 序号比特反转

Normal order of index n	Binary bits of index n	Reversed bits of index n	Bit-reversed of order index n
0	00 00 0	0 00 00	0
1	00 00 1	1 00 00	16
2	00 01 0	0 01 00	4
3	00 01 1	1 01 00	20
4	00 10 0	0 10 00	8
5	00 10 1	1 10 00	24
6	00 11 0	0 11 00	12
7	00 11 1	1 11 00	28
8	01 00 0	0 00 01	1
9	01 00 1	1 00 01	17
10	01 01 0	0 01 01	5
11	01 01 1	1 01 01	21
12	01 10 0	0 10 01	9
13	01 10 1	1 10 01	25
14	01 11 0	0 11 01	13
15	01 11 1	1 11 01	29
16	10 00 0	0 00 10	2
17	10 00 1	1 00 10	18
18	10 01 0	0 01 10	6
19	10 01 1	1 01 10	22

20	10 10 0	0 10 10	10
21	10 10 1	1 10 10	26
22	10 11 0	0 11 10	14
23	10 11 1	1 11 10	30
24	11 00 0	0 00 11	3
25	11 00 1	1 00 11	19
26	11 01 0	0 01 11	7
27	11 01 1	1 01 11	23
28	11 10 0	0 10 11	11
29	11 10 1	1 10 11	27
30	11 11 0	0 11 11	15
31	11 11 1	1 11 11	31

1.4 混合基 FFT 的常规 C 语言实现

下面的 C 代码是混合基 FFT 的常规 C 语言实现，它用 FFT 算法的主要发明人 Cooley 和 Turkey 命名。它要求正常顺序的输入 x[]，产生正常顺序的输出 y[]。输入，输出和旋转因子 wn[]都是复数，实部和虚部存在数组中相邻的单元，实部在偶数单元，虚部在奇数单元。所有数据数据都是 16 位的 Q.15 格式的小数。

```
void CooleyTukeyFft16x16(int N, short wn[], short x[], short y[])
{
  int    n1,  n2,  ie,   ia1,  ia2, ia3;
  int    i0,  i1,  i2,    i3,  j,   k;
  short  co1, co2, co3,  si1,  si2, si3;
  short  xt0, yt0, xt1,  yt1,  xt2, yt2;
  short  xh0, xh1, xh20, xh21, xl0, xl1,xl20,xl21;

  n2 = N;
  ie = 1;

  /*Performs log4(N)-1 stages of radix4 FFT */
  for (k = N; k > 4; k >>= 2)
  {
    n1 = n2;
    n2 >>= 2;
    ia1 = 0;

    /*loop through butterflies with different twiddle factors
    j * i0 = N/4 */
    for (j = 0; j < n2; j++)
    {
      ia2 = ia1 + ia1;
      ia3 = ia2 + ia1;

      co1 = wn[2 * ia1    ];
      si1 = wn[2 * ia1 + 1];
      co2 = wn[2 * ia2    ];
      si2 = wn[2 * ia2 + 1];
      co3 = wn[2 * ia3    ];
      si3 = wn[2 * ia3 + 1];
      ia1 = ia1 + ie;

      /*loop through the groups with same twiddle factors*/
      for (i0 = j; i0< N; i0 += n1)
      {
        i1 = i0 + n2;
        i2 = i1 + n2;
        i3 = i2 + n2;

        xh0  = x[2 * i0    ] + x[2 * i2    ]; //calculate real(A1)
        xh1  = x[2 * i0 + 1] + x[2 * i2 + 1]; //calculate imag(A1)
        xl0  = x[2 * i0    ] - x[2 * i2    ]; //calculate real(A2)
        xl1  = x[2 * i0 + 1] - x[2 * i2 + 1]; //calculate imag(A2)

        xh20 = x[2 * i1    ] + x[2 * i3    ]; //calculate real(B1)
        xh21 = x[2 * i1 + 1] + x[2 * i3 + 1]; //calculate imag(B1)
        xl20 = x[2 * i1    ] - x[2 * i3    ]; //calculate real(B2)
        xl21 = x[2 * i1 + 1] - x[2 * i3 + 1]; //calculate imag(B2)

        x[2 * i0    ] = (xh0 + xh20);         //calculate real(A1+B1)
        x[2 * i0 + 1] = (xh1 + xh21);         //calculate imag(A1+B1)

        xt0  = xh0 - xh20;                    //calculate real(A1-B1)
        yt0  = xh1 - xh21;                    //calculate imag(A1-B1)
        xt1  = xl0 + xl21;                    //calculate real(A2-jB2)
        yt1  = xl1 - xl20;                    //calculate imag(A2-jB2)
        yt2  = xl1 + xl20;                    //calculate imag(A2+jB2)
        xt2  = xl0 - xl21;                    //calculate real(A2+jB2)

        x[2 * i1    ] = (xt1 * co1 + yt1 * si1) >> 15;
```

```
      x[2 * i1 + 1] = (yt1 * co1 - xt1 * si1) >> 15;
      x[2 * i2    ] = (xt0 * co2 + yt0 * si2) >> 15;
      x[2 * i2 + 1] = (yt0 * co2 - xt0 * si2) >> 15;
      x[2 * i3    ] = (xt2 * co3 + yt2 * si3) >> 15;
      x[2 * i3 + 1] = (yt2 * co3 - xt2 * si3) >> 15;
    }
  }

  ie <<= 2;
}

/*performs either a radix2 or radix4 transform on the   */
/*last stage depending on "npoints". If "npoints" is a multiple of 4, */
/*then this last stage is also a radix4 transform, otherwise it is a  */
/*radix2 transform.*/
if(4==k)
{
  i0= 0;
  for(j= 0; j< N/4; j++)
  {
    i1= i0+ 1;
    i2= i1+ 1;
    i3= i2+ 1;

    xh0  = x[2 * i0    ] + x[2 * i2    ]; //calculate real(A1)
    xh1  = x[2 * i0 + 1] + x[2 * i2 + 1]; //calculate imag(A1)
    xl0  = x[2 * i0    ] - x[2 * i2    ]; //calculate real(A2)
    xl1  = x[2 * i0 + 1] - x[2 * i2 + 1]; //calculate imag(A2)

    xh20 = x[2 * i1    ] + x[2 * i3    ]; //calculate real(B1)
    xh21 = x[2 * i1 + 1] + x[2 * i3 + 1]; //calculate imag(B1)
    xl20 = x[2 * i1    ] - x[2 * i3    ]; //calculate real(B2)
    xl21 = x[2 * i1 + 1] - x[2 * i3 + 1]; //calculate imag(B2)

    x[2 * i0    ]  = xh0 + xh20;          //calculate real(A1+B1)
    x[2 * i0 + 1]  = xh1 + xh21;          //calculate imag(A1+B1)
    x[2 * i2    ]  = xh0 - xh20;          //calculate real(A1-B1)
    x[2 * i2 + 1]  = xh1 - xh21;          //calculate imag(A1-B1)
    x[2 * i1    ]  = xl0 + xl21;          //calculate real(A2-jB2)
    x[2 * i1 + 1]  = xl1 - xl20;          //calculate imag(A2-jB2)
    x[2 * i3 + 1]  = xl1 + xl20;          //calculate imag(A2+jB2)
    x[2 * i3    ]  = xl0 - xl21;          //calculate real(A2+jB2)

    i0+= 4;
  }
}
else if(2==k)
{
  i0= 0;
  for(j= 0; j< N/2; j++)
  {
    i1= i0+ 1;

    xh0  = x[2 * i0    ] + x[2 * i1    ];
    xh1  = x[2 * i0 + 1] + x[2 * i1 + 1];
    xl0  = x[2 * i0    ] - x[2 * i1    ];
    xl1  = x[2 * i0 + 1] - x[2 * i1 + 1];

    x[2*i0]=  xh0;
    x[2*i0+ 1]= xh1;
```

```
      x[2*i1]= xl0;
      x[2*i1+ 1]= xl1;

      i0+= 2;
    }
  }

  //reverse the output order
  for (k = 0; k < N; k++)
  {
    j= bitReverse(k);

    y[2*j]= x[2*k];
    y[2*j+1]= x[2*k+1];
  }
}
```

前面的基 4 FFT 由三个主循环实现。最外层的"k"代表 FFT 的级，第二层的"j"代表不同的旋转因子，最内的"i0"代表使用相同旋转因子的不同的蝶。最后一级可能是基 4 或基 2，放在最后单独实现

根据Table 3的关系我们可以得出以下规律:

a) 最内层循环次数从 1 到 N/4 变化，每级扩大 4 倍。

b) 第二层循环次数从 N/4 到 1 变化，每级缩小 1/4。

因此，规律 a)和 b)正好相反。

c) 如果最内层循环和第二层循环合并成一层循环，则循环次数固定为 N/4。在 DSP 上的优化实现正是这样做的。

FFT 的旋转因子按如下公式产生：

$$W_N^n = e^{-j2\pi * n/N} = \cos(-2\pi * n/N) + j\sin(-2\pi * n/N) = \cos(2\pi * n/N) - j\sin(2\pi * n/N)$$

生成旋转因子 wn[]的代码如下：

```
void gen_wn_16(short *w, int N)
{
    int n;
    float M = 32767;
    float PI = 3.14159265358979323846;

    for (n = 0; n < N; n++) {
        w[2*n] =  M * cos(2.0 * PI * n / N);
        w[2*n+1] =  M * sin (2.0 * PI * n / N);
    }
}
```

请注意，保存在旋转因子表中的旋转因子的虚部是 sine 而不是-sine，“sine”前面的负号在 FFT 复数乘法运算时被补进去。

上面 FFT 实现中，每个蝶形运算所用的三个旋转因子在存储空间上是不连续的，这会降低访问的效率。在 DSP 的优化实现中，旋转因子表被重新组织，从而可以连续访问。

1.5 快速傅立叶逆变换(IFFT)

逆向 DFT 也可以类似地按前面介绍的方法来实现。IFFT 和 FFT 实现上的主要区别是：

1, 输出结果除 N

2, 旋转因子W_N^{kn} 变成W_N^{-kn}

另外，IFFT 也可以用 FFT 函数来计算：

$$x(n) = \frac{1}{N}\sum_{k=0}^{N-1} X(k)W_N^{-kn} = \frac{1}{N}[\sum_{k=0}^{N-1} X^*(k)W_N^{kn}]^* = \frac{1}{N}\{DFT[X^*(k)]\}^*$$

实现方法是，先把输入数据取复共轭，执行 FFT，再对结果取复共轭并除 N。但，如果输入和输出数据的复共轭计算不方便的话，就需要用专门的代码来实现 IFFT。

本文档仅专注于 FFT 的优化实现。 相同的方法可应用到 IFFT 的实现上。

2 TMS320C64x+ DSP 简介及 FFT 相关特性

本节简介 TMS320C64x+ DSP，探讨 DSP 内为 FFT 实现而做的增强功能。
TMS320C64x+ DSP 是 Texas Instruments 最新的 DSP 系列。它运行速度可达到 1GHz，有 8 个运算单元，因此每个周期可执行 8 条指令，每秒钟最高可执行 8G 个 16bitsx16bits 的乘累加运算。下图是 C64x+ DSP 核对框图，更详细的信息请参阅”*TMS320C64x C64x+ DSP CPU and Instruction Set Reference Guide (spru732)”.*

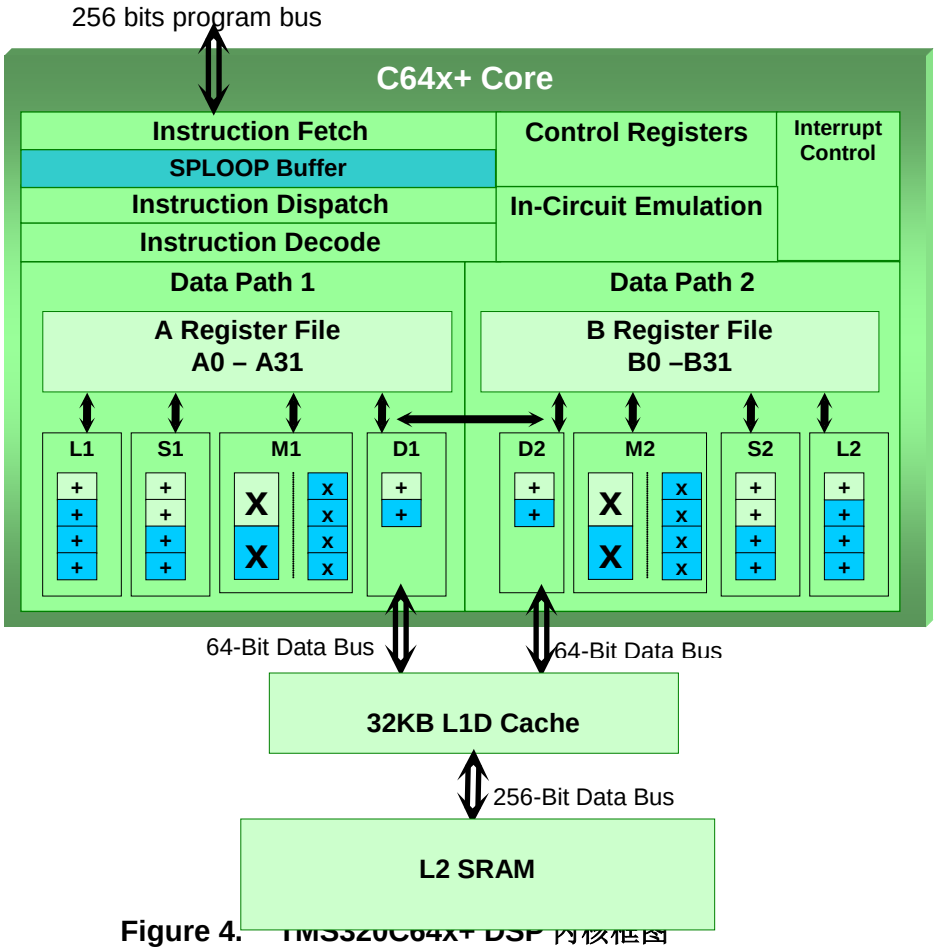


Figure 4.　TMS320C64x+ DSP 内核框图

C64x+有两组 64-bit 数据总线，可直接访问 32KB 一级数据存储器(L1D)。L1D 中的 0～32KB 可以被配置为 cache，其它部分可被用作普通 RAM。一级存储器的速度和内核一样，二级(L2)存储器的速度是 DSP 核的一半。

为了充分利用 64-bit 数据总线，算法的数据访问要被优化成每次访问 64-bit(double word，双字)；为了充分利用 L1D cache，数据应该尽量的被连续访问。

C64x+ DSP 针对 FFT 做了专门的增强，下表总结了主要的对 FFT 运算有帮助的指令：

Table 5.　C64x+ DSP 上用于 FFT 的指令

FFT 运算	C64x+ 指令	说明
蝶形运算中的加法和减法运算	ADD2	两个 16-Bit 整数加。用于 16 bit 复数加： (a+jb) + (c+jd)= (a+c)+ j(b+d)
	SUB2	两个 16 Bit 整数减。用于 16 bit 复数减：

		(a+jb) - (c+jd)= (a-c)+ j(b-d)
	ADDSUB2	对相同输入数据的并行的 ADD2 和 SUB2 运算。用于 16 bit complex 加和减： (a+jb) + (c+jd)= (a+c)+ j(b+d) (a+jb) - (c+jd)= (a-c)+ j(b-d)
	ADDSUB	并行的 ADD 和 SUB 运算，用于 32-bit FFT。
	AVG2	两个 16-Bit 整数取平均。用于蝶形运算第一个加法输出的结果除 2： (a+jb) + (c+jd)= (a+c)/2+ j(b+d)/2
加减运算的结果乘旋转因子	CMPYR	16-bit 复数乘法并对结果四舍五入和饱和: (a+jb) * (c+jd)= (ac-bd)>>16+ j(ad+bc)>>16
	MPY2IR	两个 16-Bit x 32-Bit 运算，结果右移 16 位并四舍五入。用于 32-bit 数据， 16-bit 旋转因子的 FFT
	SMPY32	32-Bit x 32-Bit 乘法。用于 32-bit FFT.
输出数据顺序比特反转	BITR	32-bit word 比特顺序反转。
	DEAL	DEAL 对 32-bit 数据做比特交织。SHFL 是 DEAL 的逆运算。
	SHFL	
	NORM	确定数据的最高有效位。也就是计算 $\log_2(N)$.

关于这些指令的细节，请参阅"*TMS320C64x C64x+ DSP CPU and Instruction Set Reference Guide* (SPRU732)"。

基 2FFT 的比特反转可由以下代码实现：

```
power2= 30- _norm(N);      //power2= log2(N)
j= _bitr(k)>>(32-power2);
```

这里的 k 是原始序号，j 是重排序的序号，N 是 FFT 点数。

上面代码中的"_bitr"和"_norm"被称作指令 BITR 和 NORM 的 intrinsic，这是在 DSP C 语言程序中指定使用某条 DSP 指令的方法。DSP 编译器并不把它们当作函数调用，而是把它们编译成相应的指令。几乎每条 DSP 指令都有相应的 intrinsic，这使得 C 语言程序的优化更便捷。

下图演示了在 C64x+ DSP 上 1024 点基 2FFT 比特反转的实现，图中每个字母代表一个比特。

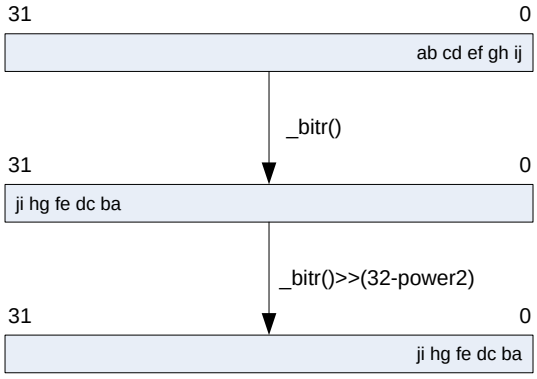


Figure 5. C64x+ DSP 上 1024 点基 2 FFT 比特反转

基 4 FFT 的比特反转可用以下代码实现：

```
power2= 30- _norm(N);      //power2= log2(N)
j= _shfl(_rotl(_deal(_bitr(k)>>(32-power2)),16))
```

下图演示了在 C64x+ DSP 上 1024 点基 4 FFT 比特反转的实现。

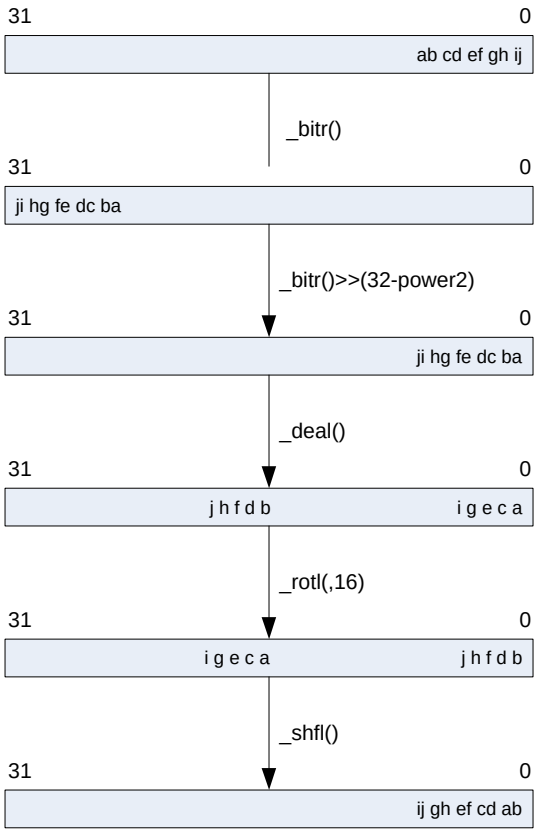


Figure 6. C64x+ DSP 上 1024 点基 4 FFT 比特反转

混合基 FFT 的比特反转可用以下代码实现：

```
power2= 30- _norm(N);           //power2= log2(N)
j= _shfl(_rotl(_deal(_bitr(k)>>(32-power2)),16))
if(last stage is radix-2)       //swap the last single bit
  j= ((k&1)<<(power2-1))|(j&((1<<power2)-1));
```

3 基于 TMS320C64x+的 FFT 实现

TI 提供的 C64x+ DSP 算法库中包含多种 FFT 的优化实现。所以这些库函数的可在 C 语言中调用。通过调用这些优化的函数，可以获得比标准 C 语言程序快得多的执行速度。TI 同时也提供这些库函数的原代码，以方便客户在此基础上修改以得满足特殊要求的 FFT。

3.1 C64x+ DSP 库里的 FFT 函数

下表总结了 C64x+ DSP 库里的 FFT 函数：

Table 6. C64x+ DSP 库里的 FFT 函数

函数	说明
void DSP_fft16x16(short *w, int nx, short *x, short *y)	16 bits 复输入输出数据，复数按先实部/后虚部交织存放 除最后一级外，每级的结果右移 1 并四舍五入
void DSP_fft16x16_**imre**(short *w, int nx, short *x, short *y)	16 bits 复输入输出数据，复数按先**虚部/后实部**交织存放 除最后一级外，每级的结果右移 1 并四舍五入
void DSP_fft16x16**r**(int nx, short *x, short *w, short *y, int radix, int offset, int n_max)	16 bits 复输入输出数据，复数按先实部/后虚部交织存放 除最后一级外，每级的结果右移 1 并四舍五入 **针对比较小的 Cache 优化**
void DSP_fft16x**32**(short *w, int nx, int *x, int *y)	**32** bits 复输入输出数据，复数按先实部/后虚部交织存放
void DSP_fft**32x32**(int *w, int nx, int *x, int *y)	**32** bits 复输入输出数据，复数按先实部/后虚部交织存放 **32** bits 旋转因子
void DSP_fft**32x32s**(int *w, int nx, int *x, int *y)	**32** bits 复输入输出数据，复数按先实部/后虚部交织存放 **32** bits 旋转因子 除最后一级外，每级的结果右移 1
void DSP_**i**fft16x16(	16 bits 复输入输出数据，复数按先实部/后虚部交织存放

short *w, int nx, short *x, short *y)	除最后一级外，每级的结果右移 1 并四舍五入
Void DSP_**ifft**16x16_**imre**(short *w, int nx, short *x, short *y)	16 bits 复输入输出数据，复数按先**虚部/后实部**交织存放 除最后一级外，每级的结果右移 1 并四舍五入
void DSP_**ifft**16x**32**(short *w, int nx, int *x, int *y)	**32** bits 复输入输出数据，复数按先实部/后虚部交织存放
void DSP_**ifft32x32**(int *w, int nx, int *x, int *y)	**32** bits 复输入输出数据，复数按先实部/后虚部交织存放 **32** bits 旋转因子

Figure 7 说明的 FFT 函数的命名规则：

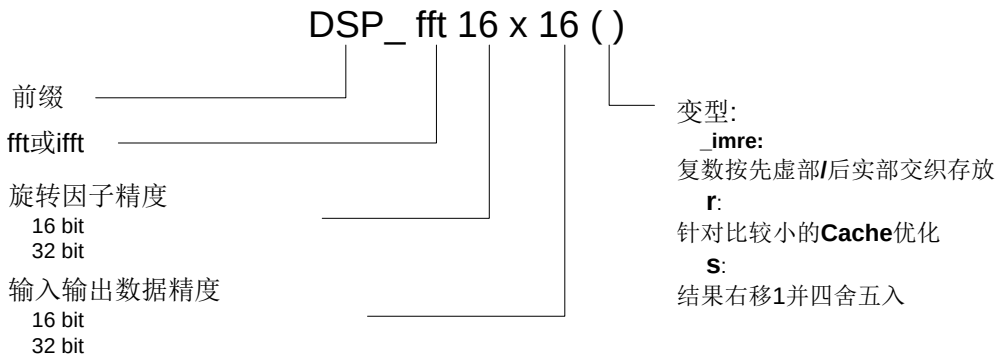


Figure 7. FFT 函数命名规则

所以函数都按混合基 FFT 实现。除最后一级外，都是基 4 运算，最后一级可能是基 2 或基 4。FFT 点数要求是 2 或 4 的整数次方，且 16 ≤ nx ≤ 65536。

所以数据都是 Q.15 或 Q.31 的复小数，实部虚部交织存放。

输出数据是经过排序后的数据。FFT 蝶形运算可以直接在输入数组 x[]上执行，但比特反转的结果必须存放的另一数组，所以最终输出结果要求存放的另一个数组 y[]。

关于 DSP 库的详细信息，请参阅”*TMS320C64x+ DSP Library Programmer's Reference* (SPRUEB8)”.

下面章节介绍 DSP 库里面的 FFT 函数实现的优化细节。

3.2 对旋转因子访问的优化

在第一部分介绍的普通 FFT 实现里，对旋转因子表的访问不是连续的。为了利用 cache 的特点，旋转因子表被按如下所示的方式重排，以使它能被连续访问：

Table 7.　可连续访问的旋转因子表

第一级的 3N/4 个旋转因子
第二级的 3N/16 个旋转因子
第三级的 3N/64 个旋转因子
……

对每个基 4 蝶，第一个旋转因子都是$W_N^0 = 1$，它不需要被存储，只有其它三个需要存储。所以，第一级只用 3N/4 个旋转因子。在下一级，运算被分成四个小 FFT，每个小 FFT 使用相同的旋转因子，所以第二级只用 3N/16 个旋转因子。以此类推，总共需要的旋转因子的个数为：

3N/4+ 3N/16+ 3N/64+ … = 3N(1/4+ 1/16+ 1/64+ …) <= 3N*(1/4)/(1-(1/4))= N

所以，旋转因子表的大小也是 N。

下面的代码产生用于 DSP_fft16x16 函数的优化的旋转因子表:

```
int gen_twiddle_fft16x16(short *w, int n)
{
  int i, j, k;
  float M = 32767;
  float PI = 3.14159265358979323846;

  /*loops through the stages, loop count is log4(N)-1,
  Twiddle factor in last stage is always 1 , is not necessary to generate*/
  for (j = 1, k = 0; j < n >> 2; j = j << 2)
  {
    /*generate twiddle factors for two butterflies in one loop,
    W[0-1, 4-5, 8-9] for the first butterfly,
    the others for the second butterfly,
    loop count= (N/4)/(2j)*/
    for (i = 0; i < n >> 2; i += j << 1)
    {
      /*twiddle factor for the second output of first butterfly*/
      w[k +  1] =  M * cos(2.0 * PI * (i    ) / n);
      w[k +  0] =  M * sin(2.0 * PI * (i    ) / n);

      /*twiddle factor for the second output of second butterfly*/
      w[k +  3] =  M * cos(2.0 * PI * (i + j) / n);
      w[k +  2] =  M * sin(2.0 * PI * (i + j) / n);

      /*twiddle factor for the third output of first butterfly*/
      w[k +  5] = -M * cos(4.0 * PI * (i    ) / n);
      w[k +  4] = -M * sin(4.0 * PI * (i    ) / n);

      /*twiddle factor for the third output of second butterfly*/
      w[k +  7] = -M * cos(4.0 * PI * (i + j) / n);
      w[k +  6] = -M * sin(4.0 * PI * (i + j) / n);

      /*twiddle factor for the fourth output of first butterfly*/
      w[k +  9] =  M * cos(6.0 * PI * (i    ) / n);
      w[k +  8] =  M * sin(6.0 * PI * (i    ) / n);

      /*twiddle factor for the fourth output of second butterfly*/
      w[k + 11] =  M * cos(6.0 * PI * (i + j) / n);
      w[k + 10] =  M * sin(6.0 * PI * (i + j) / n);
      k += 12;
    }
  }
  return k;
}
```

请注意，上面的代码里，为了 FFT 乘法的方便，有些旋转因子的符号被调整。FFT 实现在做复数乘法运算时都巧妙的补偿了这些调整的符号。

其它 FFT 函数需要的旋转因子表可能略有不同，在 DSP 库里面，每个 FFT 函数都提供了一个对应的旋转因子生成函数。

通常，旋转因子生成函数在软件初始化阶段调用，生成的旋转因子表被存到数组里。对于不同的 FFT 点数，要生成不同的旋转因子表。FFT 函数则在运行的过程中反复调用，旋转因子表也被反复使用。

3.3 蝶形运算的优化

C64x+ DSP library 里 FFT 的实现利用了Table 3中的旋转因子组和 FFT 级之间的关系，标准 C 代码里的两个内层循环"i0" 和 "j"被合成一个内层循环。

为了进一步利用 C64x+ DSP 的 8 个并行功能单元和两条 64bit 数据总线，内层循环被展开一倍，每次循环处理两个蝶形，下面的伪代码描述了基本的处理过程：

```
for(loop through the stages)
  for(i=0; i< N; i+=8)
  {
    //every loop calculate two butterflies, 8 points
    ......
  }
```

我们以 DSP_fft16x16 的优化实现为例，来说明实现方法。下图描述了每次循环内两个蝶形的运算过程。

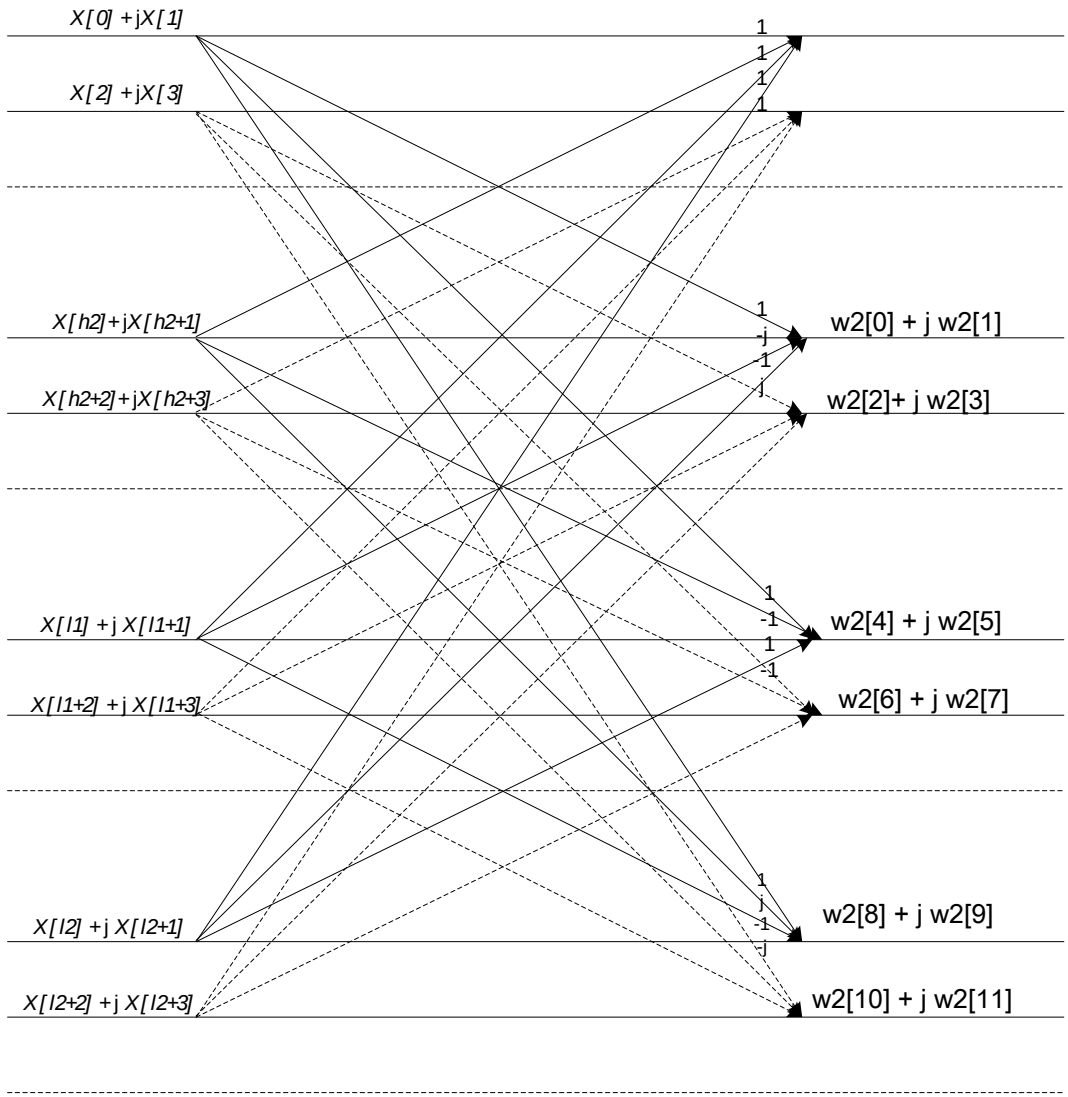


Figure 8.　优化实现中的基本蝶形运算

上图中，输入数组定义为：

```
Int16   x[2*N];
```

请注意，每个复数的实部和虚部存储在相邻的两个 16bit 单元，实部在偶数单元，虚部在奇数单元。如上图所示，第一个输入复数为 x[0]+jx[1].

两个蝶形的 8 个输入点可以用 4 个 64bit (double word) load 指令读取：

```
            x_3210 = _amemd8(&x[0]);
            x_h2_32_x_h2_10 = _amemd8(&x[h2]);
            x_l1_32_x_l1_10 = _amemd8(&x[l1]);
            x_l2_32_x_l2_10 = _amemd8(&x[l2]);
```

_amemd8 intrinsic 代表 double word load 指令. _amemd8 前面的"a"说明读取的 8 字节是按 8 字节对齐(align)的。

旋转因子的定义如下：

```
Int16   w2[2*N];
```

两个蝶形运算用到的 6 个旋转因子用 64bit (double word) load 指令读取:

```
            co11si11_co10si10 = _amemd8_const(w2);
            co21si21_co20si20 = _amemd8_const(&w2[4]);
            co31si31_co30si30 = _amemd8_const(&w2[8]);
```

_amemd8_const intrinsic 也执行 double word wide load 指令, 它的后缀“const”代表被访问的数据不会被修改。

请注意，蝶形运算的第一个输出用到旋转因子始终是 1，所以不必存取和计算。

由于两个蝶的运算相同，我们以第一个蝶为例。蝶形内的运算包括加法，减法和乘 j。具体运算分解如下：

First output= x + x_h2 + x_l1 + x_l2= (x + x_l1) + (x_h2 + x_l2)

Third output= x - x_h2 + x_l1 - x_l2= (x + x_l1) - (x_h2 + x_l2)

Second output= x - j*x_h2 - x_l1 + j*x_l2= (x - x_l1) - j*(x_h2 - x_l2)

Fourth output= x + j*x_h2 - x_l1 - j*x_l2= (x - x_l1) + j*(x_h2 - x_l2)

下面的代码计算 (x + x_l1) 和 (x - x_l1)， (x_h2 + x_l2) 和 (x_h2 - x_l2)

```
xh1_0_xh0_0_xl1_0_xl0_0 = _addsub2((_lo(x_3210)), (_lo(x_l1_32_x_l1_10)));
xh1_0_xh0_0 = (_hill(xh1_0_xh0_0_xl1_0_xl0_0));
xl1_0_xl0_0 = (_loll(xh1_0_xh0_0_xl1_0_xl0_0));

xh21_0_xh20_0_xl21_0_xl20_0 = _addsub2((_lo(x_h2_32_x_h2_10)), (_lo(x_l2_32_x_l2_10)));
xh21_0_xh20_0 = (_hill(xh21_0_xh20_0_xl21_0_xl20_0));
xl21_0_xl20_0 = (_loll(xh21_0_xh20_0_xl21_0_xl20_0));
```

上面的 _lo 和 _loll intrinsic 取 64 bit 数据的低 32 bit。每 64 bits 输入数据中，低 32 bit 是第一个蝶的数据。类似地，_hi 和 _hill intrinsic 取 64 bit 数据的高 32 bit。

那么，蝶形运算第一个输出的计算如下：

```
x_1o_x_0o = _avg2(xh21_0_xh20_0, xh1_0_xh0_0);
```

上面的_avg2 把两个数高低 16 位分别相加，然后右移一位 (>>1)并四舍五入。

蝶形运算第三个输出的计算如下：

```
myt0_0_mxt0_0 = _sub2(xh21_0_xh20_0, xh1_0_xh0_0);
xl1_1_0 = _cmpyr((_lo(co21si21_co20si20)), myt0_0_mxt0_0);
```

_cmpyr 把两个 16bits Q.15 的复数 (蝶形运算的第三个输出和其旋转因子)相乘，得到 32bits Q.30 的数据，加上 0x8000 (四舍五入)，然后取高 16 位，得到 16bit Q.14 数据，这实际等效于把 Q.15 格式的数据右移一位 2 (>>1)并做四舍五入。

蝶形运算的第二个和第四个输出的计算因为包含 j*(x_h2 - x_l2)而有一点复杂。它可以用复数乘法指令实现，但效率较低。

对于一个 16 比特复数(a+jb)，实部 a 存储在低 16bit，虚部 b 存储在高 16bit，j*(a+jb)= -b+ja，所以，乘 j 可以由高低 16bit 互换得到 b+ja，然后与旋转因子相乘。注意，b 前面没有负号，而前面我们提到，生成旋转因子时某些旋转因子数值前面的符号也做了特殊的变化，这就正好补偿了 b 前面的负号。

蝶形运算的第二个和第四个输出的计算如下：

```
xl0_0_xl1_0 = _rotl(xl1_0_xl0_0, 16);          //swap the real/imaginary part

xt1_0_yt2_0_xt2_0_yt1_0 = _addsub2(xl0_0_xl1_0, xl21_0_xl20_0);

xt1_0_yt2_0 = (_hill(xt1_0_yt2_0_xt2_0_yt1_0));
xt2_0_yt1_0 = (_loll(xt1_0_yt2_0_xt2_0_yt1_0));
yt1_0_xt1_0_yt2_0_xt2_0 = _dpackx2(xt1_0_yt2_0, xt2_0_yt1_0);

yt1_0_xt1_0 = (_hill(yt1_0_xt1_0_yt2_0_xt2_0));
yt2_0_xt2_0 = (_loll(yt1_0_xt1_0_yt2_0_xt2_0));
xh2_1_0 = _cmpyr((_lo(co11si11_co10si10)), yt1_0_xt1_0);
xl2_1_0 = _cmpyr((_lo(co31si31_co30si30)), yt2_0_xt2_0);
```

请在 DSPLib 的*dsplib_v210\src\DSP_fft16x16* 目录查看完整的源代码来研究进一步的细节。其它 FFT 函数的实现代码也可以从相应的目录找到。

3.4 Cache 冲突的优化

由于 FFT 运算过程中有三个数组需同时访问，在基于 Cache 访问的系统，如 C64x+，这可能会导致 Cache 冲突，从而明显的降低效率。

按上面介绍的实现方法，FFT 是按下面的方式逐级计算的：

```
Process N data in stage 1
Process N data in stage 2
……
Process N data in last stage
```

由于每级运算都会用到大部分数据，如果所有用到的数据合起来大于 Cache 的容量，则每级处理都会出现 Cache 冲突。例如，2048 点 16x16 FFT 需要 2K*4B=8KB 输入数据，8KB 旋转因子，和 8KB 输出数据。如果 DSP 的 L1D Cache 大小为 32KB，则刚好能满足 2048 点 FFT，处理大于 2048 点的 FFT 则会引起 L1D Cache 的冲突。

我们前面讨论过，对于 N 点基 4 FFT，第一级之后，它被分解成 4 个 N/4 点 FFT。所以，FFT 可按以下方式计算：

```
Perform first stage of N point FFT
Perform N/4 point FFT
Perform N/4 point FFT
Perform N/4 point FFT
Perform N/4 point FFT
```

在第一步，所有的数据都被用到；但在后面，只有 N/4 的数据被用到。所以，Cache 冲突可能只发生在第一步。这样就减轻了 Cache 冲突的影响。

DSP_fft16x16r 函数就是基于以上考虑来实现的。它的定义是：

```
void DSP_fft16x16r(int nx, short * restrict x, short * restrict w, short * restrict y,
                   int radix, int offset, int nmax)
```

这个函数执行在 nx 点数据上执行分解后的部分的 FFT 运算，分解前的主 FFT 的点数是 nmax。如我们前面讨论的，FFT 是逐级分解的，本函数一直将 FFT 分解到点数为 radix。如果要函数执行的最后一级，则 radix 为 4 或 2，nx 点个输出会按偏移 offset 存储到输出数组。例如：

```
DSP_fft16x16r (8192, &x[0],     &w[0],      y, 4,    0,   8192);
```

上面的函数调用执行完整的 8192 点 FFT 直到 radix=4。基于 32KB 的 Cache，每一级都会发生 Cache 冲突。

同样的功能可以分解成以下函数调用：

```
DSP_fft16x16r(8192, &x[0],     &w[0]    ,  y, 2048, 0,      8192);
DSP_fft16x16r(2048, &x[0],     &w[2*6144], y, 4,    0,      8192);
DSP_fft16x16r(2048, &x[2*2048],&w[2*6144], y, 4,    1*2048, 8192);
DSP_fft16x16r(2048, &x[4*2048],&w[2*6144], y, 4,    2*2048, 8192);
DSP_fft16x16r(2048, &x[6*2048],&w[2*6144], y, 4,    3*2048, 8192);
```

第一个函数将 8192 点 FFT 分解到 2048 点，也就是，处理基 4 FFT 的第一级。其它的四个函数执行 4 个完整的 2048 点 FFT（直到 radix=4）并且根据偏移 offset 将结果存到输出数组。请注意，如前面3.2节讨论的，第二级运算用到的旋转因子从旋转因子表的 3*N/4 位置开始。

基于 32KB Cache，Cache 冲突只在执行第一个函数时发生，后 4 个函数执行时不会发生 Cache 冲突。

按上面的分解方法，点数为 N，最后一级为基"rad"的 FFT，可以被分解为：

```
DSP_fft16x16r(N,  &x[0],          &w[0],          y, N/4, 0,     N);
DSP_fft16x16r(N/4,&x[0],          &w[2*(3*N/4)], y, rad, 0,     N);
DSP_fft16x16r(N/4,&x[2*(N/4)],    &w[2*(3*N/4)], y, rad, N/4,   N);
DSP_fft16x16r(N/4,&x[2*(N/2)],    &w[2*(3*N/4)], y, rad, N/2,   N);
DSP_fft16x16r(N/4,&x[2*(3*N/4)], &w[2*(3*N/4)],  y, rad, 3*N/4, N);
```

4 执行速度指标

文档"*TMS320C64x+ DSP Library Programmer's Reference* (SPRUEB8)" 提供了每个 FFT 函数理论上的执行速度指标，它没有考虑实际系统中 cache miss 的开销。本节提供在 DSP 硬件上的实测数据供参考。

Table 8 和 Figure 9 是在 TMS320TCI6484 EVM 板上测试的各种 FFT 执行指令周期数。L1D cache 大小设为 32KB。每个测试之前 Cache 都被清空。

Table 8. FFT 函数执行的指令周期数

Point	64	128	256	512	1024	2048	4096	8192	16384
CooleyTukey fft16x16	2557	5788	11501	26529	52937	120141	240149	617637	2626845
DSP_fft16x16	383	837	1595	3559	6883	15459	30267	79140	339383
DSP_fft16x16_imre	389	843	1601	3565	6889	15465	30273	79142	339385
full DSP_fft16x16r	478	1060	2071	4613	8897	19779	38427	96511	403297
decomposed DSP_fft16x16r	722	1344	2362	4950	9242	20170	38838	94490	317907
DSP_fft16x32	775	1499	2882	6276	12316	27362	61100	291540	597642
DSP_fft32x32	831	1759	3307	7259	14283	31679	74681	305807	625685
DSP_fft32x32s	763	1698	3204	7106	13988	31090	72582	352544	716664

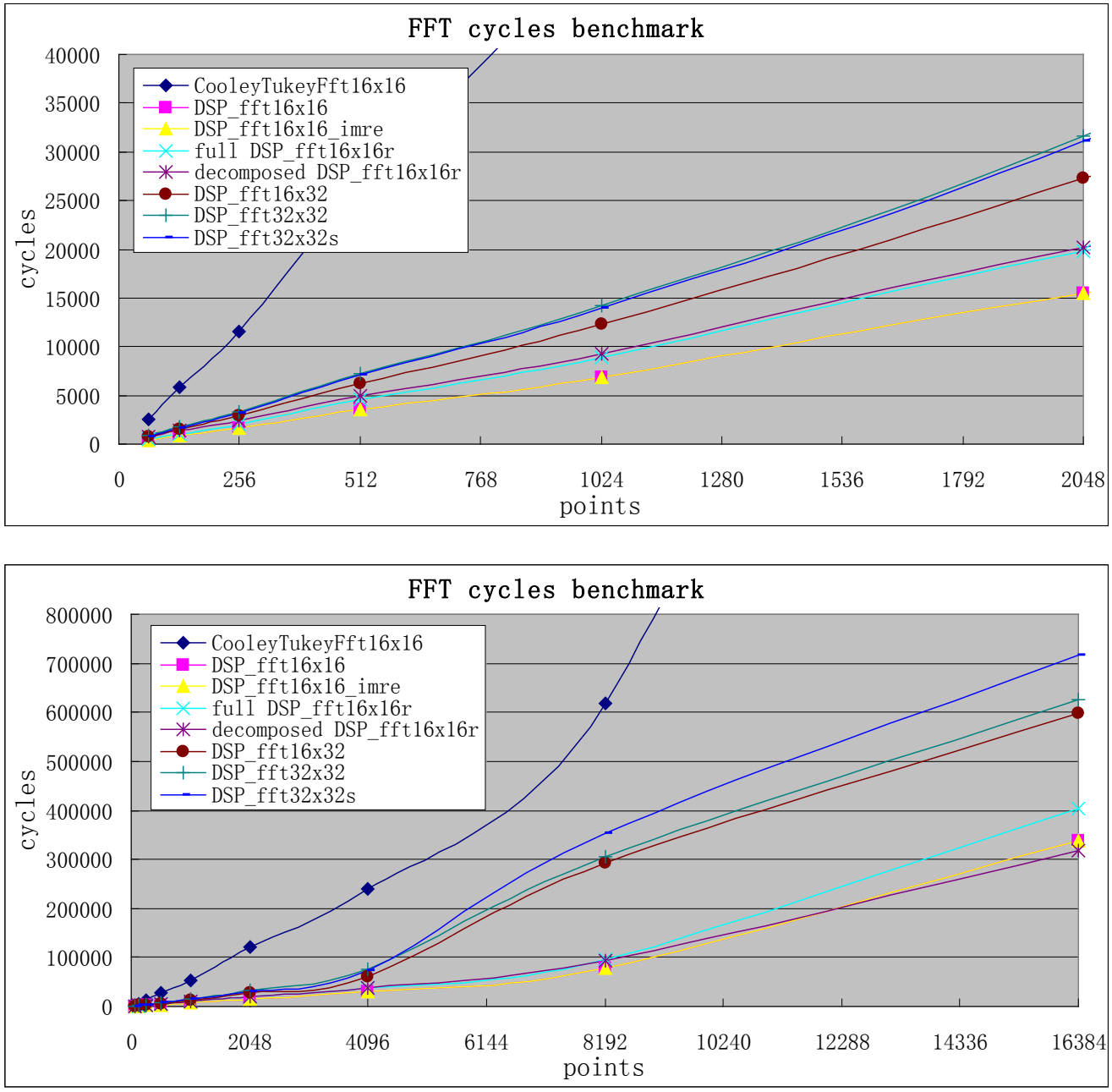


Figure 9. FFT 函数执行的指令周期数

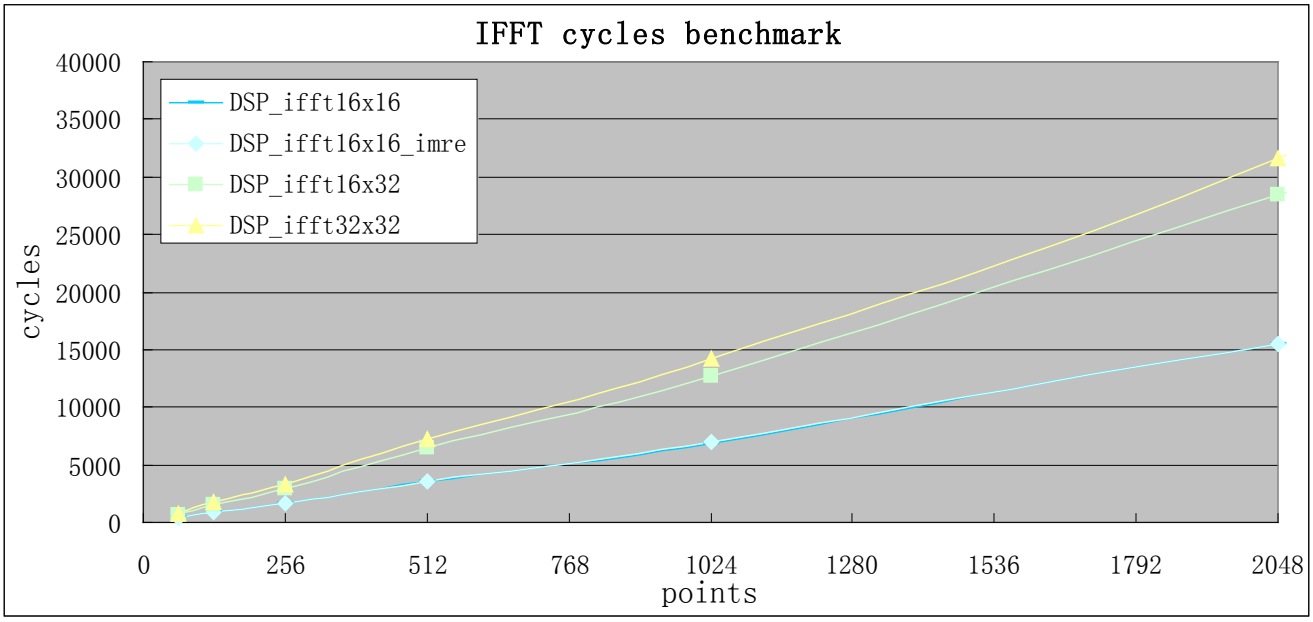
在上面的表和图里，“full DSP_fft16x16r”是一次执行完 FFT 运算；“decomposed DSP_fft16x16r”是按前面3.4节介绍的 Cache 优化方法，将 N 点 FFT 分解为 4 个 N/4 点 FFT。

从上面的数据可以看出，标准 C 实现的 CooleyTukey FFT 比 DSP Library 里优化的 FFT 实现消耗的指令周期多很多。对于少于 4096 点的 FFT，消耗的指令周期和点数几乎成正比；而当点数大于 4096 点时，指令周期数增长加快，这是由于 Cache 冲突造成的。可以看出“decomposed DSP_fft16x16r”在点数较大时消耗的指令周期少于” full DSP_fft16x16r”。

Table 9 和 Figure 10是在 TMS320TCI6484 EVM 板上测试的各种 IFFT 执行指令周期数。

Table 9. IFFT 函数执行的指令周期数

Point	64	128	256	512	1024	2048	4096	8192	16384
DSP_ifft16x16	402	856	1611	3575	6899	15475	30283	79151	339397
DSP_ifft16x16_imre	439	900	1647	3615	6941	15521	30331	79207	339449
DSP_ifft16x32	683	1461	2872	6424	12680	28412	60450	323618	668534
DSP_ifft32x32	783	1740	3288	7238	14262	31656	74648	304706	623634



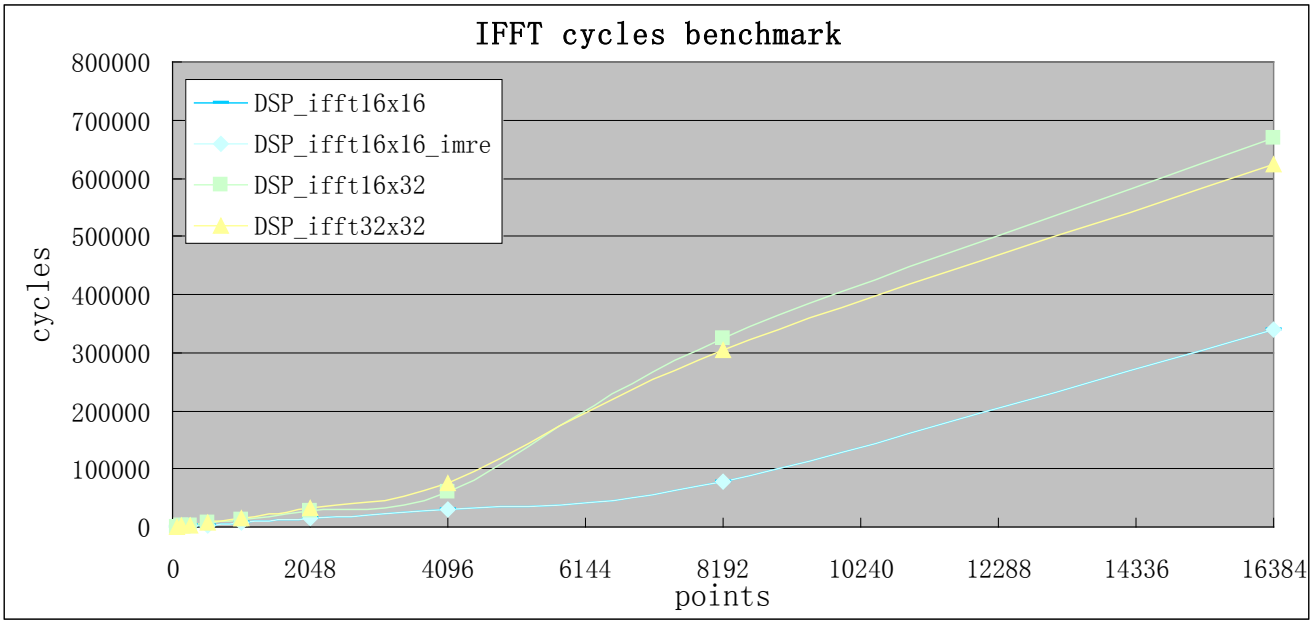


Figure 10. IFFT 函数执行的指令周期数

用 TI 的指令周期消耗分析工具，我们发现上面的数据中大概有 1/3 浪费在 L1D cache miss 上。主要由访问输入数组，旋转因子和输出数组引起，它发生在这些数据从 L2 搬到 L1D 时。

由于相同的旋转因子在多次调用 FFT 函数时被反复使用，所以，一个好的解决旋转因子 L1D miss 的方法是，把 L1D 的一部分当作普通 RAM 使用，直接把旋转因子表放到 L1D RAM 中。

为了解决输入输出数据引起的 L1D miss，我们也可以在 L1D 中为它们划分专门的空间。由于输入输出数据是不断变化的，可以在 DSP 执行当前的 FFT 时，同时用 DMA 将下一组数据搬到 L1D 中。但需要注意的是，在 L1D 中分配固定的数据块会减少 L1D Cache 的大小，这可能会降低其它需要较大 L1D Cache 的算法或函数的性能。所以，这需要在系统设计上做折中。

另外一个方法可减少 Cache miss 的影响，它把需要用到的数据在访问之前"touch"到 Cache 中，关于"touch"方法的更多信息，请参阅"*TMS320C64x+ DSP Cache User's Guide* (SPRU862)"。以上所以测试都没有用"touch"方法。

Take DSP_fft16x16 as example, Table 10 shows the impact of touching different data before perform DSP_fft16x16.

Table 10. Cache "touch"对 DSP_fft16x16 性能的影响

	64	128	256	512	1024	2048	4096	8192	16384
No touch	413	857	1591	3555	6875	15447	30279	79160	339391
Touch twiddle factor	412	834	1500	3334	6392	14442	28258	79132	344064
Touch twiddle factor and input data	411	793	1389	3087	5873	13379	26115	81416	348701
Touch twiddle factor and input, output data	431	768	1272	2785	5204	11997	31795	86208	353363
Best improvement	0.48%	10.39%	20.05%	21.66%	24.31%	22.33%	13.75%	0.04%	N/A

上表中，第二行数据是下面两个函数总共消耗的指令周期数：

```
touch(w16, nx*4);
DSP_fft16x16(w16, nx, x16, y16);
```

上面的代码在调用 DSP_fft16x16 之前先把旋转因子"touch"到 Cache 中。

上表中，第三行数据是下面三个函数总共消耗的指令周期数：

```
touch(w16, nx*4);
touch(x16, nx*4);
DSP_fft16x16(w16, nx, x16, y16);
```

上面的代码在调用 DSP_fft16x16 之前先把旋转因子和输入数据"touch"到 Cache 中。

上表中，第四行数据是下面四个函数总共消耗的指令周期数：

```
        touch(w16, nx*4);
        touch(x16, nx*4);
        touch(y16, nx*4);
        DSP_fft16x16(w16, nx, x16, y16);
```

上面的代码在调用 DSP_fft16x16 之前先把旋转因子，输入数据和输出数据”touch”到 Cache 中。

最后一行表示最好情况和没有”touch”的情况相比性能改进的百分比。最后的性能改进可达到 24%。

通常，在 FFT 函数调用之前”touch”数据可减少 FFT 函数的执行周期数。但调用”touch”函数会消耗额外的指令周期。所以，只有当 FFT 函数上节省的周期数大于”touch”函数消耗的周期数才能带来整体性能的改进。

通常，被”touch”的数组越大，节省的周期数会越多。但总共被”touch”的数据不能超过 cache 的大小。这就是上表中点数大于 4096 的情况，Cache 冲突会在”touch”时发生，性能不但不能改善，反而有可能恶化，所以”touch”不应该在这种情况下使用。

在实际的应用中，数据结构可能比这里的测试更复杂，设计者应该尝试不同的数据”touch”的组合来得到最好的结果。另外，在实际应用中，FFT 函数的输入可能是它之前的另一个函数的输出，这时，FFT 的输入数据可能已经在 Cache 之中，那么就没有必要再去”touch”这些数据。如果 FFT 的输出会被另一个函数使用，那么执行 FFT 之前”touch” FFT 的输出数据数组可能更有意义。

5 数据缩放和精度

FFT 计算中的乘法和加法可能会导致溢出。为了解决乘法溢出，所有的数据都被表示为小数，并按小数相乘，因为两个小数相乘的结果总是小于或等于 1。

第二种溢出来自于加法。根据前面讲到的基 4 FFT 算法，每个蝶形输出由四个数相加得到，它最多带来 2bit 的进位。对于基 2 运算(混合基最后一级)，每级可能产生 1bit 进位。解决这个问题最直接的方法是每级的输入预先右移 2bit 或 1bit。

每级都对输入数据右移 2bit 或 1bit 会消耗额外的指令周期。一个简化的方法是一次性对输入数据右移足够的 bit，以保证后面所有运算都不会溢出。N 点 FFT 整个运算可能带来的进位总数为 $\log_2(N)$，所以一次性把输入数据右移 $\log_2(N)$ bit 就可防止 FFT 运算溢出。

然而，右移防止溢出是以损失精度为代价的。例如，1024 点 FFT，为防止溢出，输入数据必须被右移 10bit。如果输入数据是 16bit 的，那么，只有 6bit 有效精度。

在 DSP library 里，大部分 FFT 函数在每级(最后一级除外)FFT 运算后都对输出右移一位并四舍五入。这由使用带右移功能的加法和乘法指令实现，如 AVG2 和 CMPYR, 这并不增加额外的指令周期。对于这种实现，要防止 FFT 运算溢出，只需要对输入数据右移:

$$\log_2(N)- (\#stages-1)= \log_2(N)- ceil[\log_4(nx)-1]$$

FFT 点数越多，移位所导致的精度损失越大。如果 16bit FFT 不能满足精度要求，可以用 32bit 的 FFT。通常，32bit FFT 消耗的指令周期是 16bit FFT 的两倍，占用的存储器也是两倍。

另外一个解决的办法是检查每级输出的进位，然后确定右移几位来防止溢出。这会使精度损失最小，但它增加了实现的复杂度。TI DSP library 不直接支持这种实现。由于 TI DSP Library 提供了所有函数的源代码，客户可以在这些源代码的基础上做一些修改来实现特殊功能。让我们以 DSP_fft16x16 为基础，实现一个叫 DSP_fft16x16_scale 的带动态缩放的 FFT 函数。

我们假设输入数据是 16-bit Q.15 格式的小数，表示为：

S.ABC DEFG HIJK LMNO,

上面，每个字母代表一个 bit，最高位，S，是符号位。A…O 是数据位。为了防止基 4 FFT 加法溢出，我们需要两个保护 bit，也就是，输入数据的 bit 14 (A) 和 bit 13 (B)应该等于符号位(S)。如果它们不等于符号位，我们就需要右移来得到两个保护 bit。

为了确定 bit 13 和 14 是否和符号位相同，我们把输入数据先右移两位，得到如下新数据：

S.SSA BCDE FGHI JKLM

把它和原始数据异或(XOR)，如果原始 bit 14 或 13 与符号位不同则结果的 bit 14 或 13 就是 1。

对每级的所有输出数据进行这种检测，代码如下：

```
            next_scale|= ((x0)^_shr2(x0,2))|;
                         ((x1)^_shr2(x1,2))|
                         ((x2)^_shr2(x2,2))|
                         ((x3)^_shr2(x3,2));
```

上面的 x0, x1, x2, x3 是蝶形运算的四个输出。请注意，它们是 32 位的，低 16 位是复数的实部，高 16 位是复数的虚部。所以，我们用 intrinsic _shr2()对高低 16 位同时右移。在每个数据上的检测结果被或(OR)在一起，因此，如果任何一个数据的 bit14 或 13 和符号位不同，则最终结果的 bit14 或 13 就为 1。在每一级运算的最后，我们可以确定要右移多少位，代码如下：

```
next_scale= next_scale|(next_scale>>16); //OR higher 16 bit with lower 16 bit
if(next_scale&0x4000)             //if bit 14 is 1, we should scale 2 bit
       scale= 2;
else if(next_scale&0x2000)        //if bit 13 is 1, we should scale 1 bit
       scale= 1;
else
       scale= 0;
```

在每个数据被下一级使用前，我们对它右移“scale”，这样就可以保证下一级运算不溢出，代码如下：

```
x_3210 = _amemd8(&x[0]);

x_3210_l= _shr2(_lo(x_3210),scale);
x_3210_h= _shr2(_hi(x_3210),scale);
```

这个实现使得精度损失最小化，但它增加了指令周期数。下表比较了 DSP_fft16x16_scale 和 DSP_fft16x16 的指令周期数。这里两种情况都使用 Cache “touch”。

Table 11. 数据动态缩放的 DSP_fft16x16 的性能

	64	128	256	512	1024	2048	4096	8192
DSP_fft16x16	407	744	1250	2763	5182	11964	31753	86186

DSP_fft16x16_scale	813	1266	2662	4840	11522	21886	56056	131958

从上面的数据可看出，DSP_fft16x16_scale 消耗的周期数大概是 DSP_fft16x16 的两倍。

References

1. *TMS320C64x+ DSP Library Programmer's Reference* (SPRUEB8)
2. *TMS320C64x C64x+ DSP CPU and Instruction Set Reference Guide* (SPRU732)
3. *TMS320C64x+ DSP Cache User's Guide* (SPRU862)
4. *Extended-Precision Complex Radix-2 FFT_IFFT Implemented on TMS320C62x* (SPRA696)
5. *Auto-scaling Radix-4 FFT for TMS320C6000 DSP* (SPRA654)
6. *Understanding Digital Signal Processing, Richard G. Lyons, 2004*

重要声明

德州仪器(TI) 及其下属子公司有权在不事先通知的情况下, 随时对所提供的产品和服务进行更正、修改、增强、改进或其它更改，并有权随时中止提供任何产品和服务。客户在下订单前应获取最新的相关信息，并验证这些信息是否完整且是最新的。所有产品的销售都遵循在订单确认时所提供的TI 销售条款与条件。

TI 保证其所销售的硬件产品的性能符合TI 标准保修的适用规范。仅在TI 保证的范围内，且TI 认为有必要时才会使用测试或其它质量控制技术。除非政府做出了硬性规定，否则没有必要对每种产品的所有参数进行测试。

TI 对应用帮助或客户产品设计不承担任何义务。客户应对其使用TI 组件的产品和应用自行负责。为尽量减小与客户产品和应用相关的风险，客户应提供充分的设计与操作安全措施。

TI 不对任何TI 专利权、版权、屏蔽作品权或其它与使用了TI 产品或服务的组合设备、机器、流程相关的TI 知识产权中授予的直接或隐含权限作出任何保证或解释。TI 所发布的与第三方产品或服务有关的信息，不能构成从TI 获得使用这些产品或服务的许可、授权、或认可。使用此类信息可能需要获得第三方的专利权或其它知识产权方面的许可，或是TI 的专利权或其它知识产权方面的许可。

对于TI 的产品手册或数据表，仅在没有对内容进行任何篡改且带有相关授权、条件、限制和声明的情况下才允许进行复制。在复制信息的过程中对内容的篡改属于非法的、欺诈性商业行为。TI 对此类篡改过的文件不承担任何责任。

在转售TI 产品或服务时，如果存在对产品或服务参数的虚假陈述，则会失去相关TI 产品或服务的明示或暗示授权，且这是非法的、欺诈性商业行为。TI 对此类虚假陈述不承担任何责任。

TI 产品未获得用于关键的安全应用中的授权，例如生命支持应用（在该类应用中一旦TI 产品故障将预计造成重大的人员伤亡），除非各方官员已经达成了专门管控此类使用的协议。购买者的购买行为即表示，他们具备有关其应用安全以及规章衍生所需的所有专业技术和知识，并且认可和同意，尽管任何应用相关信息或支持仍可能由TI 提供，但他们将独力负责满足在关键安全应用中使用其产 品及TI 产品所需的所有法律、法规和安全相关要求。此外，购买者必须全额赔偿因在此类关键安全应用中使用TI 产品而对TI 及其 代表造成的损失。

TI 产品并非设计或专门用于军事/航空应用，以及环境方面的产品，除非TI 特别注明该产品属于“军用”或“增强型塑料”产品。只 有TI 指定的军用产品才满足军用规格。购买者认可并同意，对TI 未指定军用的产品进行军事方面的应用，风险由购买者单独承担，并且独力负责在此类相关使用中满足所有法律和法规要求。

TI 产品并非设计或专门用于汽车应用以及环境方面的产品，除非TI 特别注明该产品符合ISO/TS 16949 要求。购买者认可并同意，如果他们在汽车应用中使用任何未被指定的产品，TI 对未能满足应用所需要求不承担任何责任。

可访问以下URL 地址以获取有关其它TI 产品和应用解决方案的信息：

	产品		应用
数字音频	www.ti.com.cn/audio	通信与电信	www.ti.com.cn/telecom
放大器和线性器件	www.ti.com.cn/amplifiers	计算机及周边	www.ti.com.cn/computer
数据转换器	www.ti.com.cn/dataconverters	消费电子	www.ti.com/consumer-apps
DLP® 产品	www.dlp.com	能源	www.ti.com/energy
DSP - 数字信号处理器	www.ti.com.cn/dsp	工业应用	www.ti.com.cn/industrial
时钟和计时器	www.ti.com.cn/clockandtimers	医疗电子	www.ti.com.cn/medical
接口	www.ti.com.cn/interface	安防应用	www.ti.com.cn/security
逻辑	www.ti.com.cn/logic	汽车电子	www.ti.com.cn/automotive
电源管理	www.ti.com.cn/power	视频和影像	www.ti.com.cn/video
微控制器 (MCU)	www.ti.com.cn/microcontrollers		
RFID 系统	www.ti.com.cn/rfidsys		
OMAP 机动性处理器	www.ti.com/omap		
无线连通性	www.ti.com.cn/wirelessconnectivity		
	德州仪器在线技术支持社区	www.deyisupport.com	

邮寄地址： 上海市浦东新区世纪大道 1568 号，中建大厦 32 楼 邮政编码： 200122