AN1381
APPLICATION NOTE

Implementing the Radix-4 FFT Algorithm Using the ST120 DSP

By Marianne DELPHIN

INTRODUCTION

This application note for the 'Radix-4' implementation of the Fast Fourier Transform algorithm on the ST120 DSP shows how this algorithm well exploits the high parallelism of the 'ST100' superscalar architecture.

TABLE OF CONTENTS

1 - FFT PRINCIPLES

1.1 - Fast Fourier Transform

The Fourier transform converts signals from the time domain into the frequency domain and vice versa. It is described by the following equation:

Direct Fourier transform from time to frequency : (1)

$$\chi(f) = \int_{-\infty}^{\infty} x(t) e^{-j2\pi ft}\, dt$$

Inverse fourier transform from frequency to time: (2)

$$\chi(t) = \int_{-\infty}^{\infty} x(f) e^{j2\pi ft}\, df$$

To compute the Fourier transform using a fixed point DSP, the Discrete Fourier Transform which is the discrete version of the continuous Fourier transform is used.

It is expressed as: (3)

$$\chi(f) = \sum_{n=-\infty}^{\infty} x(nT) e^{-j2\pi nTf}$$

for a signal which is sampled every T (the sampling period) seconds.

As T is constant and assuming that the signal x(n) consists of N samples N-periodically repeated, we then have: (4)

$$\chi(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi k\frac{n}{N}}$$

that becomes, if the twiddle factor Wn is defined as:

$$W_N = e^{-j2\frac{\pi}{N}}$$

(5)

$$\chi(k) = \sum_{n=0}^{N-1} x(n) W_N^{nk}$$

One can notice the symmetry and periodicity properties of the twiddle factors as

$$W_N^{k} = -W_N^{k+\frac{N}{2}}$$

$$W_N^{k} = W_N^{k+N}$$

1.2 - Radix-4 FFT Algorithm

Direct computation would take N^2 complex multiplications and N(N-1) complex additions which would lead to a complexity of $O(N^2)$.

To reduce this complexity, a new algorithm derived from the Cooley and Turkey [3] algorithm was created. This new algorithm, assuming N is a power of 4, divides an N-point Discrete Fourier Transform (DFT) into four N/4-point DFTs, then into 16 N/16-point DFTs and so on, up to arrive to a 4-point Fourier Transform.This algorithm is known as the radix-4 Fast Fourier Transform (radix-4 FFT).

The complexity is then reduced to:

Table 1 : Number of Operations of radix-2 and radix-4 FFTs

N	radix-2 additions	radix-4 additions	radix-2 multiplications	radix-4 multiplications
16	152	148	24	20
64	1032	976	264	208
256	5896	5488	1800	1392
1024	30728	28336	10248	7856

This leads to the decomposition of equation (5) into

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

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

as

$$W_N^{\frac{N}{4}k} = (-j)^k$$

$$W_N^{\frac{N}{2}k} = (-1)^k$$

$$W_N^{\frac{3N}{4}k} = (j)^k$$

It leads to : (6)

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

To get a 4-point DFT decomposition, let's decompose again as: (7)

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

$$\chi(k+1) = \sum_{n=0}^{\frac{N}{4}-1} \left[x(n) - jx\left(n+\frac{N}{4}\right) - x\left(n+\frac{N}{2}\right) + jx\left(n+3\frac{N}{4}\right) \right] W_N^{n(k+1)}$$

$$\chi(k+2) = \sum_{n=0}^{\frac{N}{4}-1} \left[x(n) - x\left(n+\frac{N}{4}\right) + x\left(n+\frac{N}{2}\right) - x\left(n+3\frac{N}{4}\right) \right] W_N^{n(k+2)}$$

$$\chi(k+3) = \sum_{n=0}^{\frac{N}{4}-1} \left[x(n) + jx\left(n+\frac{N}{4}\right) - x\left(n+\frac{N}{2}\right) - jx\left(n+3\frac{N}{4}\right) \right] W_N^{n(k+3)}$$

This basic calculation is called a **butterfly** and is repeated for all the 4 point bundles.The butterfly pattern is shown graphically in Figure 1.

Figure 1 : 16-Point radix-4 DIF FFT Butterfly

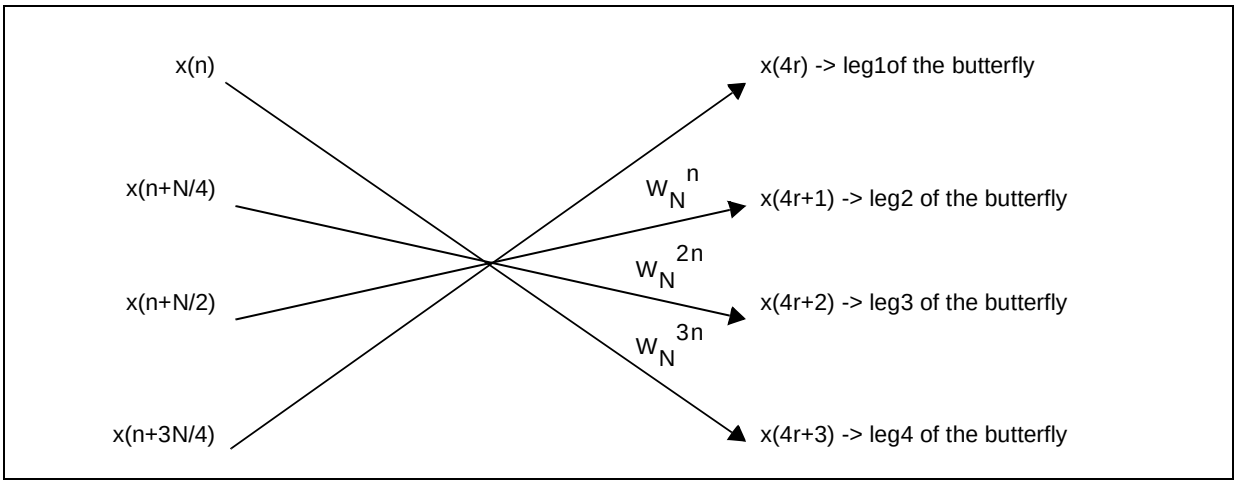
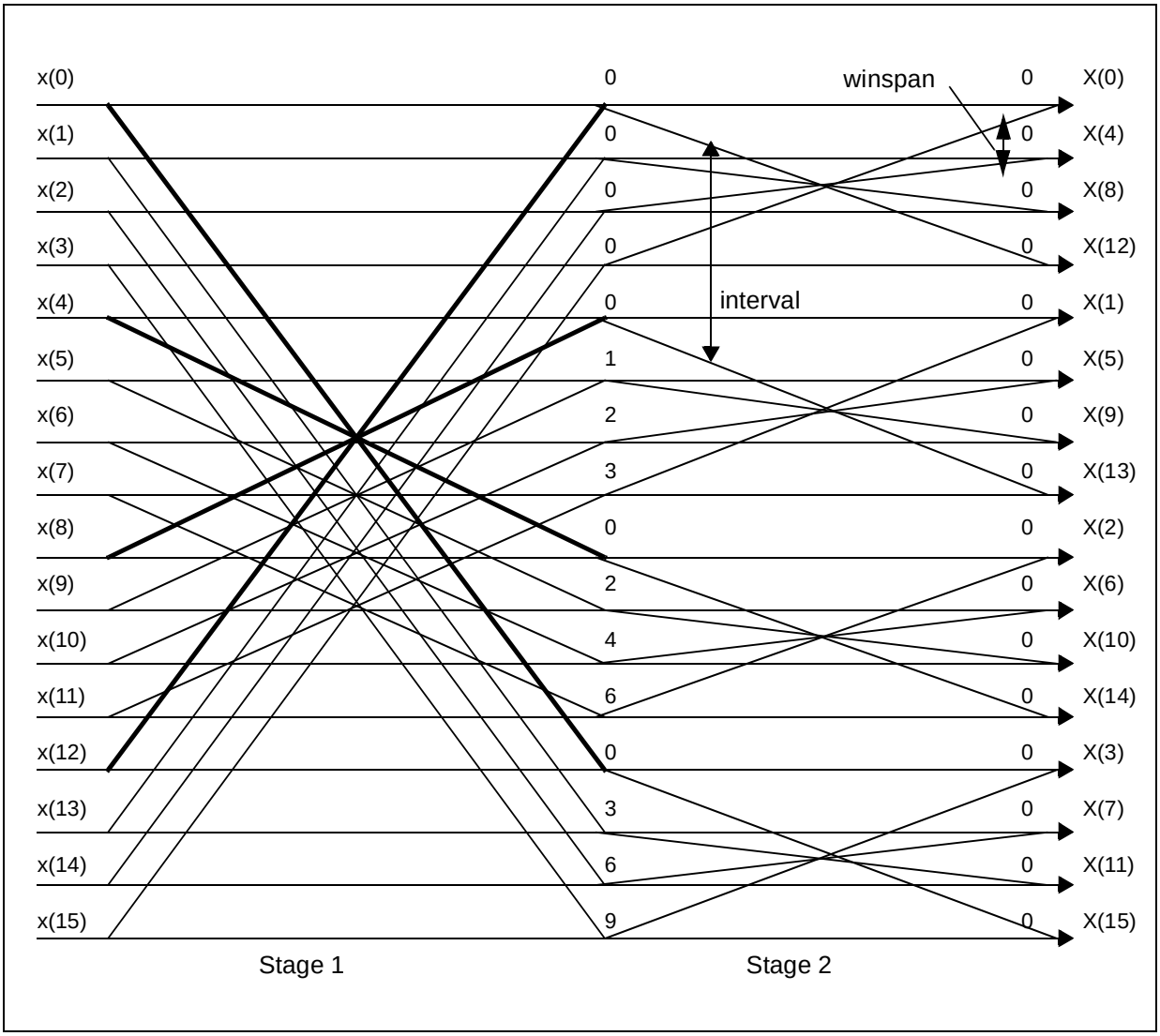


Figure 2 : 16-Point radix 4 DIF FFT



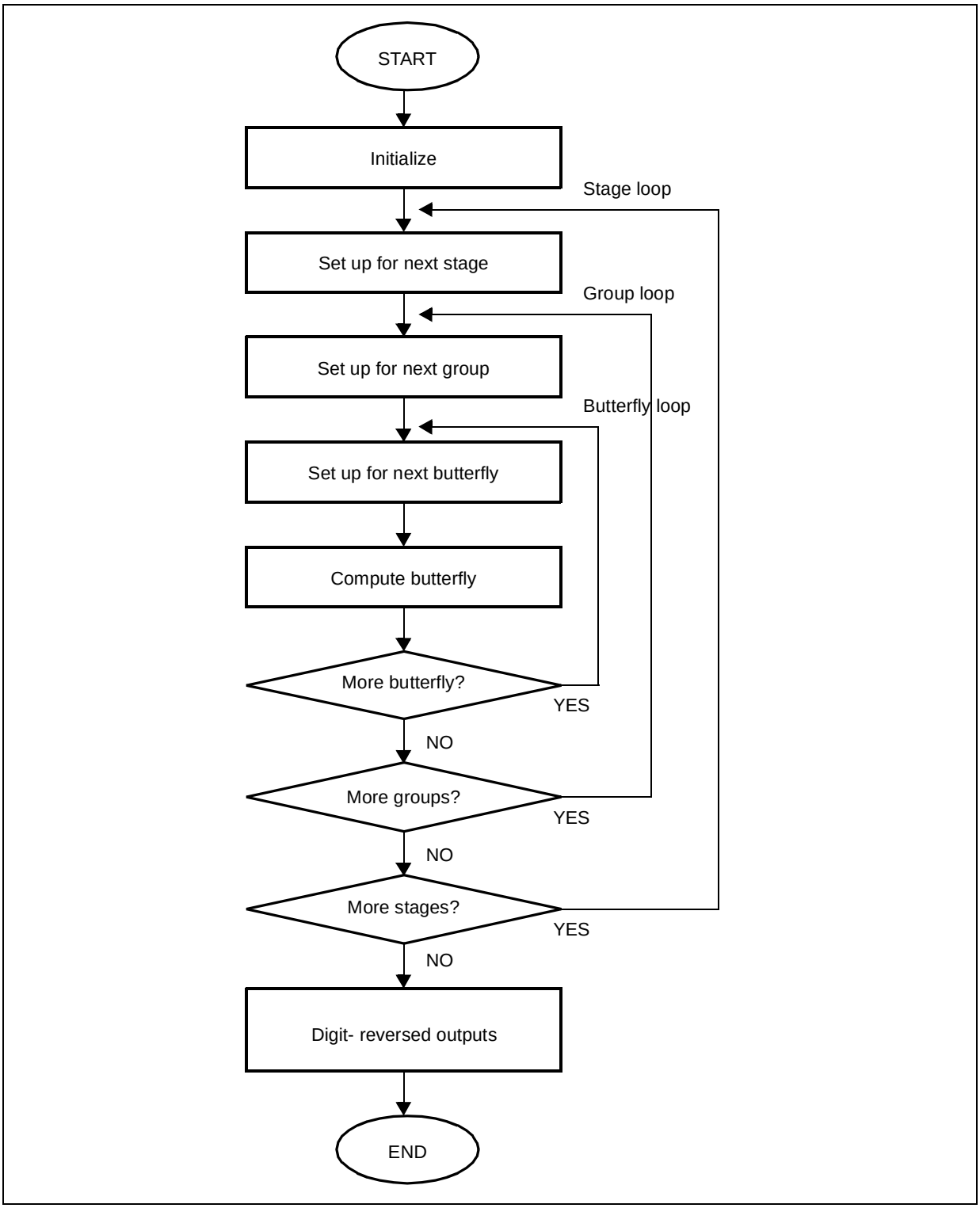
The number at the end of each branch of the butterfly corresponds to the twiddle factor exponent(i.e. 6 stands for W_N^6).

It appears that the radix-4 algorithm is organized in butterflies, which are organized in groups of butterflies themselves organized in stages.

In the example of a 16-point radix-4, two stages are available, four groups of one butterfly in the first stage and one group of four butterflies in the second stage.

Groups vary from N/4 to 1 and butterflies per group vary from 1 to N/4.

Figure 3 : radix-4 DIF FFT ALGORITHM



Data used to perform a butterfly calculation are called b0, b1, b2 and b3 (for example x(0),x(4),x(8) and x(12) in our previous example). They are separated by a distance called *wingspan* like bo from b1, b1 from b2 and so on.

For the first stage, Winspan = N/4 and is decreasing following the rule: wingspan = winspan>>2.

Data from two consecutive butterflies of the same groups are separated by a distance called *interval*.For the first stage interval = N as there is only one butterfly per group and at each stage, interval is decreasing following the same rule as winspan.

The relationship between wingspan and interval is interval = 4*winspan.

The algorithm can be described with flow chart in figure 3.

The characteristics of a N-point radix 4 FFT are summarized in table 2:

Table 2 : Characteristics of a N-Point radix-4

Stage		1	2	3	...	log4(N)
Butterfly pergroup		1	4	16	...	N/4
Butterfly group		N/4	N/16	N/64	...	1
Twiddle	leg1	0	0	0	...	0
Factor	leg2	n	4n	16n	...	(N/4)n
Exp.	leg3	2n	8n	32n	...	(N/2)n
	leg4	3n	12n	48n	...	(3N/4)n
		n=0 to N/4-1	n=0 to N/16-1	n=0 to N/64-1		n=0

A way of saving some memory to perform the FFT is to do" in place " calculation (.i.e. the input data for a stage M+1 are the outputs of stage M). That way N*2 memory locations (one for real complex input and one for imaginary complex input) are saved.

Then, due to the split at each stage of the $4k^{th}$, $4k+1^{th}$, $4k+2^{th}$, $4k+3^{th}$ outputs of the previous stage, final outputs are ordered in a digit-reversed mode.It means that a normal ordered input array will produce a digit-reversed output array. An example of a digit-reversed array is given in the following table:

Table 3 : Digit-Reverse Array

NORMAL ORDER			DIGIT-REVERSED ORDER		
Decimal	**Binary**	**Quaternary**	**Quaternary**	**Binary**	**Decimal**
0	0000	00	00	0000	0
1	0001	01	10	0100	4
2	0010	02	20	1000	8
3	0011	03	30	1100	12
4	0100	10	01	0001	1
5	0101	11	11	0101	5
6	0110	12	21	1001	9
7	0111	13	31	1101	13
8	1000	20	02	0010	2
9	1001	21	12	0110	6
10	1010	22	22	1010	10
11	1011	23	32	1110	14
12	1100	30	03	0011	3
13	1101	31	13	0111	7
14	1110	32	23	1011	11
15	1111	33	33	1111	15

2 - RADIX-4 FFT IMPLEMENTATION ON ST120

This section describes how to implement the radix-4 FFT algorithm on the ST120 and how to achieve a high performance program with low memory occupation thanks to the ST120 architecture.

2.1 - Data Organization

To take advantage of the ST120 memory architecture (i.e. to be able to perform two data memory accesses in the same cycle), input data are splitted on two different memory banks.

If NINPUTS is the number of points on which the FFT is performed then r[NINPUTS] is the real part array (in order) and i[NINPUTS]is the imaginary parts (in order) array.

Real and imaginary parts of the twiddles are in the same array of 2*NINPUTS length, interleaved as real/ imaginary/real/imaginary... and so on.

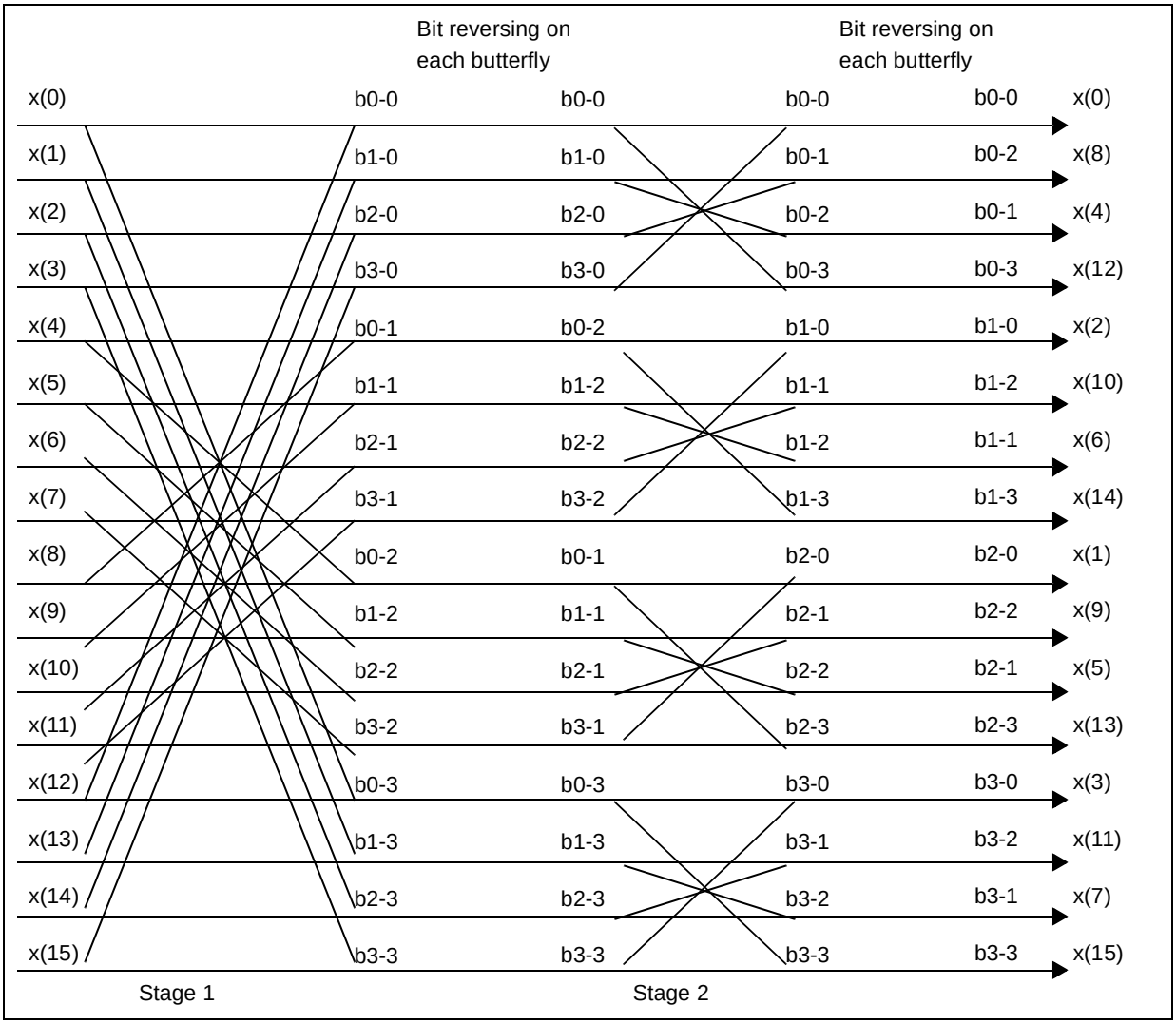
2.2 - Avoiding Digit Reversing

It can be shown that a radix-4 FFT with in order inputs and in order outputs can be performed using bit-reversing operations only.

For that, at each stage of the FFT's calculation, the outputs of each butterfly have to be bit-reversed, i.e., if the outputs of a butterfly are numbered 0,1,2 and 3 the outputs 1 and 2 have to be swapped.

For a 16-point FFT Figure 4 illustrates bit reversing operation.

Figure 4 : 16-point FFT bit reversing



After the last stage data are then in bit-reversed order.

A bit-reversing operation on the whole array is then needed to have in order outputs. As the ST120 has a bit-reverse addressing mode, this can be done very efficiently in the last stage of the calculation.

2.3 - C Code Implementation

The following fixed point C code takes into consideration the features of the ST120 architecture to have a similar structure to the optimized assembly code.

As data and twiddles are 16 bits variables they have been declared in "short " type.

```
void r4_full(short r[], short i[], short n, short w[])
{
   int stage, group, butterfly;
   int Stages, Groups, BpG;
   int WingSpan;
   int Wi, WStep;
   int Ri, Ii;
   short c0_1,s0_1, c0_2, s0_2, c0_3, s0_3;
   short c1_1,s1_1, c1_2, s1_2, c1_3, s1_3;
   short g0_0r, g0_0i, g0_1r, g0_1i, g0_2r, g0_2i, g0_3r, g0_3i;
   short g1_0r, g1_0i, g1_1r, g1_1i, g1_2r, g1_2i, g1_3r, g1_3i;
   short t0_0r, t0_0i, t0_1r, t0_1i, t0_2r, t0_2i, t0_3r, t0_3i;
   short t1_0r, t1_0i, t1_1r, t1_1i, t1_2r, t1_2i, t1_3r, t1_3i;
   short tt0_0r, tt0_0i, tt0_1r, tt0_1i, tt0_2r, tt0_2i, tt0_3r, tt0_3i;
   short tt1_0r, tt1_0i, tt1_1r, tt1_1i, tt1_2r, tt1_2i, tt1_3r, tt1_3i;
   Stages = 4;
```

Calculation for a butterfly gives:

```
(*)   for(butterfly=0; butterfly < BpG; butterfly++) {
            g0_0i = i[Ii];                        // Img(A0)
            g1_0i = i[Ii+1];                      // Img(A1)
            g0_0r = r[Ri];                        // Real(A0)
            g1_0r = r[Ri+1];                      // Real(A1)

            g0_1i = i[Ii+WingSpan];               // Img(B0)
            g1_1i = i[Ii+WingSpan+1];             // Img(B1)
            g0_1r = r[Ri+WingSpan];               // Real(B0)
            g1_1r = r[Ri+WingSpan+1];             // Real(B1)

            g0_2i = i[Ii+2*WingSpan];             // Img(C0)
            g1_2i = i[Ii+2*WingSpan+1];           // Img(C1)
            g0_2r = r[Ri+2*WingSpan];             // Real(C0)
            g1_2r = r[Ri+2*WingSpan+1];           // Real(C1)

            g0_3i = i[Ii+3*WingSpan];             // Img(D0)
            g1_3i = i[Ii+3*WingSpan+1];           // Img(D1)
            g0_3r = r[Ri+3*WingSpan];             // Real(D0)
            g1_3r = r[Ri+3*WingSpan+1];           // Real(D1)

            t0_0i = (short)(g0_0i + g0_2i);       // Img(A0 + C0)
            t1_0i = (short)(g1_0i + g1_2i);       // Img(A1 + C1)
            t0_0r = (short)(g0_0r + g0_2r);       // Real(A0 + C0)
            t1_0r = (short)(g1_0r + g1_2r);       // Real(A1 + C1)

            t0_1i = (short)(g0_0i - g0_2i);       // Img(A0 - C0)
            t1_1i = (short)(g1_0i - g1_2i);       // Img(A1 - C1)
            t0_1r = (short)(g0_0r - g0_2r);       // Real(A0 - C0)
            t1_1r = (short)(g1_0r - g1_2r);       // Real(A1 - C1)

            t0_2i = (short)(g0_1i + g0_3i);       // Img(B0 + D0)
            t1_2i = (short)(g1_1i + g1_3i);       // Img(B1 + D1)
            t0_2r = (short)(g0_1r + g0_3r);       // Real(B0 + D0)
            t1_2r = (short)(g1_1r + g1_3r);       // Real(B1 + D1)

            t0_3i = (short)(g0_1i - g0_3i);       // Img(B0 - D0)
            t1_3i = (short)(g1_1i - g1_3i);       // Img(B1 - D1)
            t0_3r = (short)(g0_1r - g0_3r);       // Real(B0 - D0)
            t1_3r = (short)(g1_1r - g1_3r);       // Real(B1 - D1)

            tt0_0r = (short)(t0_0r + t0_2r);   tt0_1r = (short)(t0_1r + t0_3i);
            tt0_0i = (short)(t0_0i + t0_2i);   tt0_1i = (short)(t0_1i - t0_3r);
            tt1_0r = (short)(t1_0r + t1_2r);   tt1_1r = (short)(t1_1r + t1_3i);
            tt1_0i = (short)(t1_0i + t1_2i);   tt1_1i = (short)(t1_1i - t1_3r);

            tt0_2r = (short)(t0_0r - t0_2r);   tt0_3r = (short)(t0_1r - t0_3i);
            tt0_2i = (short)(t0_0i - t0_2i);   tt0_3i = (short)(t0_1i + t0_3r);
            tt1_2r = (short)(t1_0r - t1_2r);   tt1_3r = (short)(t1_1r - t1_3i);
            tt1_2i = (short)(t1_0i - t1_2i);   tt1_3i = (short)(t1_1i + t1_3r);
```

Calculation for a butterfly gives (continued)

```
    // 3 Complex Multiplies per butterfly
    // do a bitrev on the outputs.
    i[Ii]   = tt0_0i;                                          // Img(A0)
    i[Ii+1] = tt1_0i;                                          // Img(A1)
    r[Ri]   = tt0_0r;                                          // Real(A0)
    r[Ri+1] = tt1_0r;                                          // Real(A1)
    Ri += WingSpan; Ii += WingSpan;
    i[Ii]   = (short)((c0_2*tt0_2i + s0_2*tt0_2r)>>15);        // Img(B0)
    i[Ii+1] = (short)((c1_2*tt1_2i + s1_2*tt1_2r)>>15);        // Img(B1)
    r[Ri]   = (short)((c0_2*tt0_2r - s0_2*tt0_2i)>>15);        // Real(B0)
    r[Ri+1] = (short)((c1_2*tt1_2r - s1_2*tt1_2i)>>15);        // Real(B1)
    Ri += WingSpan; Ii += WingSpan;
    i[Ii]   = (short)((c0_1*tt0_1i + s0_1*tt0_1r)>>15);        // Img(C0)
    i[Ii+1] = (short)((c1_1*tt1_1i + s1_1*tt1_1r)>>15);        // Img(C1)
    r[Ri]   = (short)((c0_1*tt0_1r - s0_1*tt0_1i)>>15);        // Real(C0)
    r[Ri+1] = (short)((c1_1*tt1_1r - s1_1*tt1_1i)>>15);        // Real(C1)
    Ri += WingSpan; Ii += WingSpan;
    i[Ii]   = (short)((c0_3*tt0_3i + s0_3*tt0_3r)>>15);        // Img(D0)
    i[Ii+1] = (short)((c1_3*tt1_3i + s1_3*tt1_3r)>>15);        // Img(D1)
    r[Ri]   = (short)((c0_3*tt0_3r - s0_3*tt0_3i)>>15);        // Real(D0)
    r[Ri+1] = (short)((c1_3*tt1_3r - s1_3*tt1_3i)>>15);        // Real(D1)
    Ri += WingSpan; Ii += WingSpan;
}
```

The first optimization is to use the **packed arithmetic** feature of the ST120.

As the data for two consecutive butterflies in the same groups are consecutive in the data array, two half word data can be loaded at the same time in a word and two butterflies can be calculated at the same time with packed arithmetic. Therefore, the number of iteration of the group loop can be divided by 2.

Some other simplifications in the C code can be derived from the structure of the radix-4 algorithm.

First we can see from table-2 (Characteristics of an N-point FFT), that the different stages have different numbers of iteration for the group and butterfly loops.

For the first stage, there is only one butterfly loop per group loop iteration so that it can be skipped.Then there is only one inner loop (the group's one).

```
Groups = n;
    BpG = 1;                  // ButterflysPerGroup
    Wi = 0;

    Groups = Groups >> 2;
    WingSpan = Groups;
    Ri = 0; Ii = 0;

    for(group=0; group < (Groups>>1); group++) {
       c0_1 = w[2*Wi];
       s0_1 = w[2*Wi+1];
       c1_1 = w[2*(Wi+1)];
       s1_1 = w[2*(Wi+1)+1];

       c0_2 = w[2*2*Wi];     // loading the
       s0_2 = w[2*2*Wi+1];   // the twiddles
       c1_2 = w[2*2*(Wi+1)];
       s1_2 = w[2*2*(Wi+1)+1];

       c0_3 = w[3*2*Wi];
       s0_3 = w[3*2*Wi+1];
       c1_3 = w[3*2*(Wi+1)];
       s1_3 = w[3*2*(Wi+1)+1];
       Wi + = 2;

      BUTTERFLY CALCULTATION (*)

        } //for(group). 1st stage loop
```

For the last stage, two simplifications can be performed.

First, as for the first stage, one inner loop can be skipped as there is only one iteration per group loop for the whole stage.

Secondly, if we look at the twiddle values for this stage, we can see that it is always (1,-1) so the multiplication by the twiddles are in fact only additions and subtractions.

```
Ri = 0; Ii = 0;
BpG = BpG >> 1;
for(butterfly=0; butterfly < BpG;  butterfly++) {
    g0_0i = i[Ii];                              // Img(A0)
    g0_1i = i[Ii+1];                            // Img(B0)
    g0_0r = r[Ri];                              // Real(A0)
    g0_1r = r[Ri+1];                            // Real(B0)

    g0_2i = i[Ii+2];                            // Img(C0)
    g0_3i = i[Ii+3];                            // Img(D0)
    g0_2r = r[Ri+2];                            // Real(C0)
    g0_3r = r[Ri+3];                            // Real(D0)

    g1_0i = i[Ii+4];                            // Img(A1)
    g1_1i = i[Ii+4+1];                          // Img(B1)
    g1_0r = r[Ri+4];                            // Real(A1)
    g1_1r = r[Ri+4+1];                          // Real(B1)

    g1_2i = i[Ii+4+2];                          // Img(C1)
    g1_3i = i[Ii+4+3];                          // Img(D1)
    g1_2r = r[Ri+4+2];                          // Real(C1)
    g1_3r = r[Ri+4+3];                          // Real(D1)

    t0_0i = (short)(g0_0i + g0_2i);             // Img(A0 + C0)
    t1_0i = (short)(g1_0i + g1_2i);             // Img(A1 + C1)
    t0_0r = (short)(g0_0r + g0_2r);             // Real(A0 + C0)
    t1_0r = (short)(g1_0r + g1_2r);             // Real(A1 + C1)

    t0_1i = (short)(g0_0i - g0_2i);             // Img(A0 - C0)
    t1_1i = (short)(g1_0i - g1_2i);             // Img(A1 - C1)
    t0_1r = (short)(g0_0r - g0_2r);             // Real(A0 - C0)
    t1_1r = (short)(g1_0r - g1_2r);             // Real(A1 - C1)
```

```
  t0_2i = (short)(g0_1i + g0_3i);             // Img(B0 + D0)
  t1_2i = (short)(g1_1i + g1_3i);             // Img(B1 + D1)
  t0_2r = (short)(g0_1r + g0_3r);             // Real(B0 + D0)
  t1_2r = (short)(g1_1r + g1_3r);             // Real(B1 + D1)

  t0_3i = (short)(g0_1i - g0_3i);             // Img(B0 - D0)
  t1_3i = (short)(g1_1i - g1_3i);             // Img(B1 - D1)
  t0_3r = (short)(g0_1r - g0_3r);             // Real(B0 - D0)
  t1_3r = (short)(g1_1r - g1_3r);             // Real(B1 - D1)

  tt0_0r = (short)(t0_0r + t0_2r);            tt0_1r = (short)(t0_1r + t0_3i);
  tt0_0i = (short)(t0_0i + t0_2i);            tt0_1i = (short)(t0_1i - t0_3r);
  tt1_0r = (short)(t1_0r + t1_2r);            tt1_1r = (short)(t1_1r + t1_3i);
  tt1_0i = (short)(t1_0i + t1_2i);            tt1_1i = (short)(t1_1i - t1_3r);

  tt0_2r = (short)(t0_0r - t0_2r);            tt0_3r = (short)(t0_1r - t0_3i);
  tt0_2i = (short)(t0_0i - t0_2i);            tt0_3i = (short)(t0_1i + t0_3r);
  tt1_2r = (short)(t1_0r - t1_2r);            tt1_3r = (short)(t1_1r - t1_3i);
  tt1_2i = (short)(t1_0i - t1_2i);            tt1_3i = (short)(t1_1i + t1_3r);

  // do a bitrev on the outputs.
  i[Ii]   = tt0_0i;                           // Img(A0)
  i[Ii+1] = tt0_2i;                           // Img(B0)
  r[Ri]   = tt0_0r;                           // Real(A0)
  r[Ri+1] = tt0_2r;                           // Real(B0)
  Ii +=2; Ri+=2;
  i[Ii]   = tt0_1i;                           // Img(C0)
  i[Ii+1] = tt0_3i;                           // Img(D0)
  r[Ri]   = tt0_1r;                           // Real(C0)
  r[Ri+1] = tt0_3r;                           // Real(D0)
  Ii +=2; Ri+=2;
  i[Ii]   = tt1_0i;                           // Img(A1)
  i[Ii+1] = tt1_2i;                           // Img(B1)
  r[Ri]   = tt1_0r;                           // Real(A1)
  r[Ri+1] = tt1_2r;                           // Real(B1)
  Ii +=2; Ri+=2;
  i[Ii]   = tt1_1i;                           // Img(C1)
  i[Ii+1] = tt1_3i;                           // Img(D1)
  r[Ri]   = tt1_1r;                           // Real(C1)
  r[Ri+1] = tt1_3r;                           // Real(D1)
  Ii +=2; Ri+=2;
}
```

2.4 - Assembly Implementation

The C implementation already contains some optimizations to exploit the ST120 architecture, like separating the different stages of the calculation and processing two butterflies at the same time.

2.4.1 - Packed Arithmetic

This last optimization was done to be able to use the packed arithmetic feature of the ST120.

The ST120 can perform 2 Data Unit (DU) and 2 Address Unit (AU) operations on 32-bit words in the same cycle.

With packed arithmetic, it can do up to 4 DU operations and 4 AU operations on 16-bit half-words in the same cycle if data are consecutive in memory.

As previously described, data of 2 butterflies belonging to consecutive groups are consecutive in memory and can be then loaded, added or subtracted in the same packed instruction.

Radix-4 butterfly takes 22 real additions and 12 real multiplications. Two butterflies calculations represent (22+12)*2 = 68 DU operations and thanks to the Multiply Accumulate(MAC) and Multiply Subtract (MSUB) operations and to packed arithmetic they can be performed in 20 cycles [2].

Similarly all the load operations are done in 32-bit format and since the MAC's result is a 32-bit data (16 msb needed only) store operations are done in 16-bit format.

2.4.2 - Software Pipelining

Software pipelining is a code optimization technique where instruction level parallelism is exploited by overlapping the execution of successive iterations from an inner loop. It means that a prologue of the loop code is executed before the start of the loop and an epilog of the loop code is executed after the end of the loop.This allows to improve the combination of AU and DU instructions and to reduce the number of iteration by 1.

No pure software pipelining is performed in the assembly code. Two packed loads are performed before the start of the loop to allow earlier start of the calculation in the first "sliw" group [1] of the asm code. Those two packed loads are then performed again near the end of the most inner loop (sliw group 15).

2.4.3 - Reload Counters

Reload counters allow to reload a loop counter value without any waiting penalty (zero overhead) when two loops are overlapped.

In our case, it is used in the intermediate stages to reload the butterfly and group loop counter value. It is also used in the first stage to precalculate the stage loop counter value for intermediate stages and in all the intermediate stages to precalculate the stage loop counter value for the last stage

2.4.4 - Bit Reversing

Assembly version differs from the C in the fact that in assembly, the bit-reverse operation is done at the same time as the last stage calculation, thanks to the bit-reverse addressing mode of the ST120.

Until the last stage, calculations are done" in place "i.e. data are loaded and stored in the same array. In the last stage, data are stored in order in another array just after they are calculated

2.4.5 - Data Management

2.4.5.1 - Data Index Management

Instructions in the loop group are performed to calculate the data pointer address used in the butterfly loop and to reload twiddles values.

Thanks to auto increment pointer of the ST120, the calculation of the data pointer address can be simplified.

For the radix-4 FFT the following relationship is always true:

4*wingspan*butterflies-per-group = NINPUTS.

The butterfly loop is performed butterflies-per-group times and the interval between two consecutive butterflies is 4*wingspan. Therefore, to calculate the next group data pointer address, only the subtraction of a constant is needed. If two groups are calculated at the same time, the subtraction of NINPUTS-2 is needed.

NxtDataIndex = LastDataIndex - (NINPUTS-1(or2))

2.4.5.2 - Data Register Management

Data register use is quite intensive in the 'sliw' groups of the butterflies calculation. As two butterflies are calculated at the same time, it multiplies by two the number of operands and thus the number of data registers needed.

An efficient way to save some registers is to put operand and destination data in the same register.Writing after each 'sliw' group which registers are used and which registers are free allows the programer to optimize the use of all the available data registers.

2.4.6 - ASM Code

Asm code example for two butterflies calculation is:

```
ldp R0, @(P0 !+ P2)          // R0H = Real(A1), R0L = Real(A0)
ldp R2, @(P6 !+ P2)          // R2H = Imag(A1), R2L = Imag(A0)
nop
nop
// Rused = 2;  Used: R0,R2 ; Avail: R1,R3-R15

_fft_group_loop_st:

_fft_bfly_loop_st:

// add network 1st level
ldp R4, @(P0 + P2)           // R4H = Real(C1), R4L = Real(C0)
ldp R6, @(P6 + P2)           // R6H = Imag(C1), R6L = Imag(C0)
addcp R0, R0, R4             // t1_0r=R(A1+C1), t0_0r=R(A0+C0)
subcp R4, R0, R4             // t1_1r=R(A1-C1), t0_1r=R(A0-C0)
// Rused = 4;  Used: R0,R2,R4,R6 ; Avail: R1,R3,R5,R7-R15

// add network 1st level
ldp R1, @(P0 !+ P10)         // R1H = Real(B1), R1L = Real(B0)
ldp R3, @(P6 !+ P10)         // R3H = Imag(B1), R3L = Imag(B0)
addcp R2, R2, R6             // t1_0i=I(A1+C1), t0_0i=I(A0+C0)
subcp R6, R2, R6             // t1_1i=I(A1-C1), t0_1i=I(A0-C0)
// Rused = 6;  Used: R0,R1,R2,R3,R4,R6 ; Avail: R5,R7-R15

// add network 1st level
ldp R5, @(P0 !- P10)         // R5H = Real(D1), R5L = Real(D0)
ldp R7, @(P6 !- P10)         // R7H = Imag(D1), R7L = Imag(D0)
addcp R1, R1, R5             // t1_2r=R(B1+D1), t0_2r=R(B0+D0)
subcp R5, R1, R5             // t1_3r=R(B1-D1), t0_3r=R(B0-D0)
// Rused = 8;  Used: R0-R7 ; Avail: R8-R15

// add network 2nd level
ldw   R8, @(P1 - P4)         // W0_1=R8 H,L: s0_1, c0_1
addcp R0, R0, R1             // tt1_0r=(t1_0r+t1_2r),tt0_0r=(t0_0r+t0_2r)
subcp R1, R0, R1             // tt1_2r=(t1_0r-t1_2r),tt0_2r=(t0_0r-t0_2r)
sdp @(P0 - P2), R0           // tt0_0r, tt1_0r
// Rused = 8;  Used: R1-R8 ; Avail: R0,R9-R15
```

Asm code (continued)

```
// add network 1st level
   ldw   R10, @(P1 + P4)    // W0_3=R10 H,L: s0_3, c0_3
   addcp R3, R3, R7         // t1_2i=I(B1+D1), t0_2i=I(B0+D0)
   subcp R7, R3, R7         // t1_3i=I(B1-D1), t0_3i=I(B0-D0)
   addba P4, P4, P3
   // Rused = 8;  Used: R1,R3-R8,R10 ; Avail: R0,R2,R9, R11-R15

   // add network 2nd level
   ldw   R9, @(P1 !+ P5)    // W0_2=R9 H,L: s0_2, c0_2
   addcp R2, R2, R3         // tt1_0i=(t1_0i+t1_2i),tt0_0i=(t0_0i+t0_2i)
   subcp R3, R2, R3         // tt1_2i=(t1_0i-t1_2i),tt0_2i=(t0_0i-t0_2i)
   sdp @(P6 - P2), R2       // tt0_0i, tt1_0i
   // Rused = 9;  Used: R1,R3-R10 ; Avail: R0,R2,R11-R15

   // add network 2nd level
   nop
   ldw   R11, @(P1 - P4)    // W1_1=R11 H,L: s1_1, c1_1
   addcp R4, R4, R7         // tt1_1r=(t1_1r+t1_3i),tt0_1r=(t0_1r+t0_3i)
   subcp R7, R4, R7         // tt1_3r=(t1_1r-t1_3i),tt0_3r=(t0_1r-t0_3i)
   // Rused = 10;  Used: R1,R3-R11 ; Avail: R0,R2,R12-R15

   // add network 2nd level
   ldw   R13, @(P1 + P4)    // W1_3=R13 H,L: s1_3, c1_3
   addcp R5, R6, R5         // tt1_3i=(t1_1i+t1_3r),tt0_3i=(t0_1i+t0_3r)
   subcp R6, R6, R5         // tt1_1i=(t1_1i-t1_3r),tt0_1i=(t0_1i-t0_3r)
   subba P4, P4, P3
   // Rused = 11;  Used: R1,R3-R11,R13 ; Avail: R0,R2,R12,R14-R15

   // Multiplies
   // group 1, index 1
   ldw     R12, @(P1 !- P5) // W1_2=R12 H,L: s1_2, c1_2
   nop
   mpfchl  R0, R1, R12      // Res1_2r = tt1_2r * c1_2
   mpfhh   R2, R1, R12      // Res1_2i = tt1_2r * s1_2
   // Rused = 14;  Used: R0-R13 ; Avail: R14-R15
```

Asm code (continued)

```
msfchh  R0, R0, R3, R12    // Res1_2r = Res1_2r - tt1_2i * s1_2
mafchl  R2, R2, R3, R12    // Res1_2i = Res1_2i + tt1_2i * c1_2
sdf @(P0 + 2), R0          // Res1_2r ,Store to index 1 for bitrev
sdf @(P6 + 2), R2          // Res1_2i
// Rused = 11;  Used: R1,R3-R11,R13 ; Avail: R0,R2,R12,R14-R15

// group 0, index 1
nop
nop
mpfcll  R0, R1, R9         // Res0_2r = tt0_2r * c0_2
mpflh   R2, R1, R9         // Res0_2i = tt0_2r * s0_2
// Rused = 13;  Used: R0-R11,R13 ; Avail: R12,R14-R15

msfclh  R0, R0, R3, R9     // Res0_2r = Res0_2r - tt0_2i * s0_2
mafcll  R2, R2, R3, R9     // Res0_2i = Res0_2i + tt0_2i * c0_2
sdf @(P0 !+ P2), R0        // Res0_2r, Store to index 1 for bitrev
sdf @(P6 !+ P2), R2        // Res0_2i
// Rused = 8;  Used: R4-R8,R10-R11,R13 ; Avail: R0-R3,R9,R12,R14-R15

// group 1, index 2
nop
nop
mpfchl  R9, R4, R11        // Res1_1r = tt1_1r * c1_1
mpfhh   R12, R4, R11       // Res1_1i = tt1_1r * s1_1

msfchh  R9, R9, R6, R11    // Res1_1r = Res1_1r - tt1_1i * s1_1
mafchl  R12, R12, R6, R11  // Res1_1i = Res1_1i + tt1_1i * c1_1
sdf @(P0 + 2), R9          // Res1_1r, Store to index 2 for bitrev
sdf @(P6 + 2), R12         // Res1_1i

// group 0, index 2
// Load Ar, Ai for next 2 groups
ldp R0, @(P0 + P10)        // R0H = Real(A1), R0L = Real(A0)
ldp R2, @(P6 + P10)        // R2H = Imag(A1), R2L = Imag(A0)
mpfcll  R9, R4, R8         // Res0_1r = tt0_1r * c0_1
mpflh   R12, R4, R8        // Res0_1i = tt0_1r * s0_1
```

Asm code (continued)

```
msfclh  R9, R9, R6, R8       // Res0_1r = Res0_1r - tt0_1i * s0_1
mafcll  R12, R12, R6, R8     // Res0_1i = Res0_1i + tt0_1i * c0_1
sdf @(P0 !+ P2), R9          // Res0_1r , Store to index 2 for bitrev
sdf @(P6 !+ P2), R12         // Res0_1i

// group 1, index 3
nop
nop
mpfchl  R9, R7, R13          // Res1_3r = tt1_3r * c1_3
mpfhh   R12, R7, R13         // Res1_3i = tt1_3r * s1_3

msfchh  R9, R9, R5, R13      // Res1_3r = Res1_3r - tt1_3i * s1_3
mafchl  R12, R12, R5, R13    // Res1_3i = Res1_3i + tt1_3i * c1_3
sdf @(P0 + 2), R9            // Res1_3r
sdf @(P6 + 2), R12           // Res1_3i

// group 0, index 3
nop
nop
mpfcll  R9, R7, R10          // Res0_3r = tt0_3r * c0_3
mpflh   R12, R7, R10         // Res0_3i = tt0_3r * s0_3

msfclh  R9, R9, R5, R10      // Res0_3r = Res0_3r - tt0_3i * s0_3
mafcll  R12, R12, R5, R10    // Res0_3i = Res0_3i + tt0_3i * c0_3
sdf @(P0 !+ P10), R9         // Res0_3r
sdf @(P6 !+ P10), R12        // Res0_3i

_fft_bfly_loop_en:

ldp R0, @(P0 - P14)          // R0H = Real(A1), R0L = Real(A0)
ldp R2, @(P6 - P14)          // R2H = Imag(A1), R2L = Imag(A0)
nop
nop
// Rused = 2;  Used: R0,R2 ; Avail: R1,R3-R15

 nop
 nop
 subha P0, P0, P8
 subha P6, P6, P8

 nop
 nop
 addha P4, P4, P3
 addwa P1, P1, P7

_fft_group_loop_en:

 nop
 nop
 shra2 P2, P2                // WingSpan=Groups
 GP32md
```

3 - COMPLEXITY AND PERFORMANCES

3.1 - Performances

To evaluate the theoretical performances of a NINPUTS points FFT, first the complexity of each separate loop has to be evaluated and then the complexity of each different stage (first, intermediate, last).

– Stage loop instruction (for intermediate stages): each instruction in this loop is performed [log4(NINPUTS)-2] times.

– Group loop instructions: each instruction in this loop is performed (NINPUTS/4-4)/3 times.

– Butterfly loop instructions: each instruction in this loop is performed [(log4(NINPUTS)-2)*NINPUTS/4] times.

The complexity of each different stage of our assembly code is calculated as follows:

– The complexity of the first stage:
20 cycles*1/2 * N/4 groups + constant

– The complexity of the last stage:
7 cycles * N/4 butterflies + constant

– The complexity of the intermediate stages:
20 cycles * 1/2 * [(log4(NINPUTS)-2)*NINPUTS/4] (butterfly loop instructions)
+ 3 cycles * (NINPUTS/4-4)/6 (group loop instructions)
+ 9 cycles * (log4(NINPUTS)-2) (stage loop instructions)
+ constant

The complexity of the whole algorithm is the sum of the complexity of each stage:

cycles count = 20*1/2*NINPUTS/4 + 3*(NINPUTS/4-4)/6 + 20*1/2 * [(log4(NINPUTS)-2)*NINPUTS/4] + 9*(log4(NINPUTS)-2) + 7*NINPUTS/4 + cst

Cycle Count =35*NINPUTS/8+(5*NINPUTS/2+9)*(log4(NINPUTS)-2)-2

That is to say

256 complex points radix-4 FFT	2416 + (constant=35) = 2451 cycles
1024 complex points radix-4 FFT	12185 + constant

Cycle count for radix-4

The average number of 32-bit instructions performed at each cycle by the ST120 is 3.51.

A ST120 EVA development ship has been used for measurement.

3.2 - Memory Use

The code size for the radix-4 function is of 1132 bytes.

For data memory, it has been explained before that to allow access decoupling on ST120, input data had to be separated on two different memory banks. Each of them occupies NINPUTS half-words so for 256 points input data take 2*256*2 = 1 Kbyte of memory.

Twiddles array is composed of 2*NINPUTS half-words so it also occupies 1 Kbyte of memory.

FFT calculation can be done "in place"(i.e. data is written at the same memory place where it has been read). After the last stage, data have to be bit-reversed to be in natural order. 2*NINPUTS additional memory locations will be required to store data in natural order.

Table 4 shows the overall memory requirement.

Table 4 : Overall Memory requirement

Program memory	RAM	ROM
1132 Bytes	2048 Bytes	1024 Bytes

4 - REFERENCES

[1] ST120 DSP-MCU Core Reference Guide available on www.st.com/st100/.

[2] ST120 DSP-MCU Instruction Set Reference Guide available on www.st.com/st100/.

[3] J.W. Cooley and J.W. Turkey "An algorithm for the machine calculation of complex Fourier series", Math.Comp., vol 19 April 1965 pp 297-301.