Implementing the Radix-4 Decimation in Frequency (DIF) Fast Fourier Transform (FFT) Algorithm Using a TMS320C80 DSP

APPLICATION REPORT: SPRA152

Author: Charles Wu
SC Sales & Marketing – TI Taiwan

Digital Signal Processing Solutions
January 1998

IMPORTANT NOTICE

Texas Instruments (TI) reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

TI warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are utilized to the extent TI deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain application using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

TI SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of TI products in such applications is understood to be fully at the risk of the customer. Use of TI products in such applications requires the written approval of an appropriate TI officer. Questions concerning potential risk applications should be directed to TI through a local SC sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

TI assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does TI warrant or represent that any license, either express or implied, is granted under any patent right, copyright, mask work right, or other intellectual property right of TI covering or relating to any combination, machine, or process in which such semiconductor products or services might be or are used.

TRADEMARKS

TI is a trademark of Texas Instruments Incorporated.

Other brands and names are the property of their respective owners.

CONTACT INFORMATION

US TMS320 HOTLINE	(281) 274-2320
US TMS320 FAX	(281) 274-2324
US TMS320 BBS	(281) 274-2323
US TMS320 email	dsph@ti.com

Contents

Figures

Tables

Implementing the Radix-4 Decimation in Frequency (DIF) Fast Fourier Transform (FFT) Algorithm Using a TMS320C80 DSP

Abstract

This application report describes the implementation of the radix-4 decimation in frequency (DIF) fast Fourier transform (FFT) algorithm using the Texas Instruments (TI™) TMS320C80 digital signal processor (DSP). The radix-4 DIF algorithm increases the execution speed of the FFT.

Each TMS320C80 DSP parallel processor (PP) contains four major units operating in parallel. This parallel operation allows each parallel processor to execute up to four instructions per cycle. The TMS320C80 parallel processor also contains many general-purpose registers. These registers are used for ALU status information and to configure features such as the zero overhead loop control units.

The units enable each parallel processor to execute a radix-4 DIF FFT three to four times faster than other devices. For example, a parallel processor can process a 256 complex FFT in approximately 5k instruction cycles.

Product Support on the World Wide Web

Our World Wide Web site at www.ti.com contains the most up to date product information, revisions, and additions. Users registering with TI&ME can build custom information pages and receive new product updates automatically via email.

Radix-4 FFT Algorithm

The butterfly of a radix-4 algorithm consists of four inputs and four outputs (see Figure 1). The FFT length is 4^M, where M is the number of stages. A stage is half of radix-2.

The radix-4 DIF FFT divides an N-point discrete Fourier transform (DFT) into four $N/4$ -point DFTs, then into 16 $N/16$ -point DFTs, and so on. In the radix-2 DIF FFT, the DFT equation is expressed as the sum of two calculations. One calculation sum for the first half and one calculation sum for the second half of the input sequence. Similarly, the radix-4 DIF fast Fourier transform (FFT) expresses the DFT equation as four summations, then divides it into four equations, each of which computes every fourth output sample. The following equations illustrate radix-4 decimation in frequency.

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

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

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

$$= \sum_{n=0}^{N/4-1} \left[\begin{array}{l} x(n) + x\left(n+N/4\right) W_N^{\left(n+N/4\right)k} + x\left(n+N/2\right) W_N^{\left(n+N/2\right)k} \\ + x\left(n+3N/4\right) W_N^{\left(n+N/2\right)k} \end{array} \right] W_N^{nk} \qquad (1)$$

The three twiddle factor coefficients can be expressed as follows:

$$W_N^{\left(N/4\right)k} = \left[\cos\left(\pi/2\right) - \mathrm{j}\sin\left(\pi/2\right)\right]^k = \left(-\mathrm{j}\right)^k$$

$$W_N^{\left(N/2\right)k} = \left[\cos(\pi) - \mathrm{j}\sin(\pi)\right]^k = \left(-1\right)^k$$

$$W_N^{\left(3N/4\right)k} = \left[\cos\left(3\pi/2\right) - \mathrm{j}\sin\left(3\pi/2\right)\right]^k = \mathrm{j}^k$$

Equation (1) can thus be expressed as

$$X(k) = \sum_{n=0}^{N/4-1} \begin{bmatrix} x(n) + (-\mathrm{j})^k x\left(n + N/4\right) + (-1)^k x\left(n + N/2\right) \\ +(\mathrm{j})^k x\left(n + 3N/4\right) W_N^{nk} \end{bmatrix} \quad (2)$$

To arrive at a four-point DFT decomposition, let $W_N^4 = W_{N/4}$. Equation (2) can then be written as four $N/4$ point DFTs, or

$$X(4k) = \sum_{n=0}^{N/4-1} \begin{bmatrix} x(n) + x\left(n + N/4\right) + x\left(n + N/2\right) \\ +x\left(n + 3N/4\right) \end{bmatrix} W_{N/4}^{nk} \quad (3)$$

$$X(4k+1) = \sum_{n=0}^{N/4-1} \begin{bmatrix} x(n) - \mathrm{j}x\left(n + N/4\right) - x\left(n + N/2\right) \\ +\mathrm{j}x\left(n + 3N/4\right) \end{bmatrix} W_N^n W_{N/4}^{nk} \quad (4)$$

$$X(4k+2) = \sum_{n=0}^{N/4-1} \begin{bmatrix} x(n) - x\left(n + N/4\right) - x\left(n + N/2\right) \\ -x\left(n + 3N/4\right) \end{bmatrix} W_N^{2n} W_{N/4}^{nk} \quad (5)$$

$$X(4k+3) = \sum_{n=0}^{N/4-1} \begin{bmatrix} x(n) + \mathrm{j}x\left(n + N/4\right) - x\left(n + N/2\right) \\ -\mathrm{j}x\left(n + 3N/4\right) \end{bmatrix} W_N^{3n} W_{N/4}^{nk} \quad (6)$$

for $k = 0$ to $N/4 - 1$

$X(4k)$, $X(4k+1)$, $X(4k+2)$ and $X(4k+3)$ are $N/4$ -point DFTs. Each of their $N/4$ points is a sum of four input samples $x(n)$, $x(n+ N/4)$, $x(n+ N/2)$ and $x(n+ 3N/4)$, each multiplied by either +1, -1, j, or -j. The sum is multiplied by a twiddle factor (W_N^0, W_N^n, W_N^{2n},or W_N^{3n}).

The four $N/4$ -point DFTs together make up an N-point DFT. Each of these $N/4$ -point DFTs is divided into four $N/16$ -point DFTs. Each $N/16$ DFT is further divided into four $N/64$ -point DFTs, and so on, until the final decimation produces four-point DFTs. The four-point DFT equation makes up the butterfly calculation of the radix-4 FFT. A radix-4 butterfly is shown graphically in Figure 1.

$$X(4k) = \sum_{n=0}^{N/4-1} \begin{bmatrix} x(n) + x\left(n + N/4\right) + x\left(n + N/2\right) \\ + x\left(n + 3N/4\right) \end{bmatrix} W_{N/4}^{nk} \qquad (3)$$

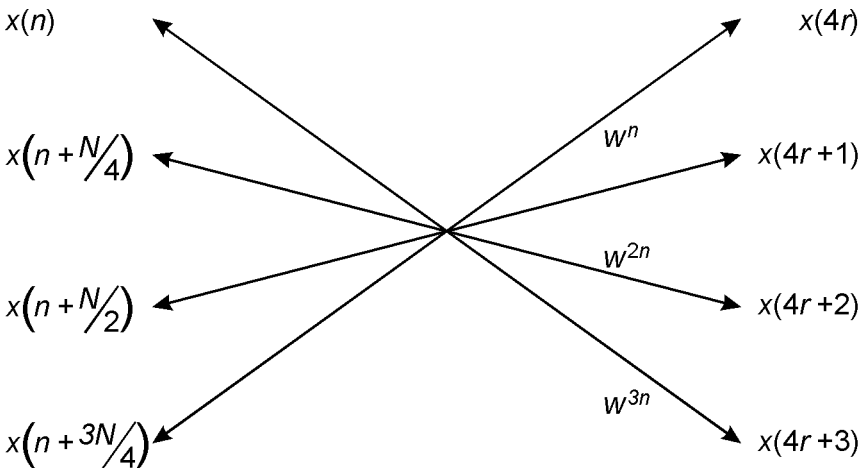
Let $x(n) + x\left(n + N/4\right) + x\left(n + N/2\right) + x\left(n + 3N/4\right) = g0(n)$

$$X(4k) = \sum g0(n) W_{N/4}^{nk} \qquad N/4 \text{-point FFT}$$

then $X(16K) = \sum \begin{bmatrix} g0(n) + g0\left(n + N/4\right) + g0\left(n + N/2\right) \\ + g0\left(n + 3N/4\right) \end{bmatrix} W_{N/16}^{nk}$

$N/16$ -point FFT

Figure 1. Radix-4 DIF FFT Butterfly



Based on Figure 1, assume the following:

$x(n)$ = xa+j ya
$x(n+N/4)$ = xb+j yb
$x(n+N/2)$ = xc+j yc
$x(n+3N/4)$ = xd+j yd

$x(4r)$ = xa'+j ya'
$x(4r+1)$ = xb'+j yb'
$x(4r+2)$ = xc'+j yc'
$x(4r+3)$ = xd'+j yd'

W^n = Wb = Cb+j (-Sb)
W^{2n} = Wc = Cc+j (-Sc)
W^{3n} = Wd = Cd+j (-Sd)

The real and imaginary output values for the radix-4 butterfly are given by equations (7) – (14).

xa' = xa+xb+xc+xd (7)

ya' = ya+yb+yc+yd (8)

xb' = (xa+yb–xc-yd)Cb - (ya–xb–yc+xd)(-Sb) (9)

yb' = (ya–xb–yc+xd)Cb + (xa+yb–xc-yd)(-Sb) (10)

xc' = (xa–xb+xc-xd)Cc - (ya–yb+yc-yd)(-Sc) (11)

yc' = (ya–yb+yc-yd)Cc + (xa–xb+xc-xd)(-Sc) (12)

xd' = (xa–yb–xc+yd)Cd - (ya+xb–yc-xd)(-Sd) (13)

yd' = (ya+xb–yc-xd)Cd + (xa–yb–xc+yd)(-Sd) (14)

Typically, more than one hundred operations are required to calculate this radix-4 butterfly. Due to PP high degree parallelism, the radix-4 DIF butterfly can be done within 22 machine cycles. It is three or four times faster than other devices.

Radix-4 DIF FFT Implementation on C80 PP

The Advantage of Parallel Processor

This section describes how to implement the radix-4 FFT using the TMS320C80 DSP parallel processor (PP). There are four PPs in each TMS320C80 DSP. Each contains the following four major units:

- Data unit
- Local address unit
- Global address unit
- Program flow control unit

These units operate in parallel. This parallel operation allows each PP to execute up to four instructions per cycle. The instructions that can be performed in one cycle include each of the following:

- Multiply (one 16-bit*16-bit or two 8-bit*8 bit) operation
- Three input arithmetic logic units (ALUs)

- Three extended ALU (EALU) operations (one 32-bit, two 16-bit, or four 8-bit)
- One barrel rotator operation
- One mask generator operation
- Two memory operations with address update

The TMS320C80 parallel processor also contains many general-purpose registers. These registers are used for ALU status information and to configure certain features such as the zero overhead loop control units. Condition and status protection functions are also provided to allow PP to process high level logical operations very efficiently.

These units and functions enable each parallel processor to execute a radix-4 DIF FFT three to four times faster than other devices. For example, a parallel processor can process a 256 complex FFT in approximately 5k instruction cycles.

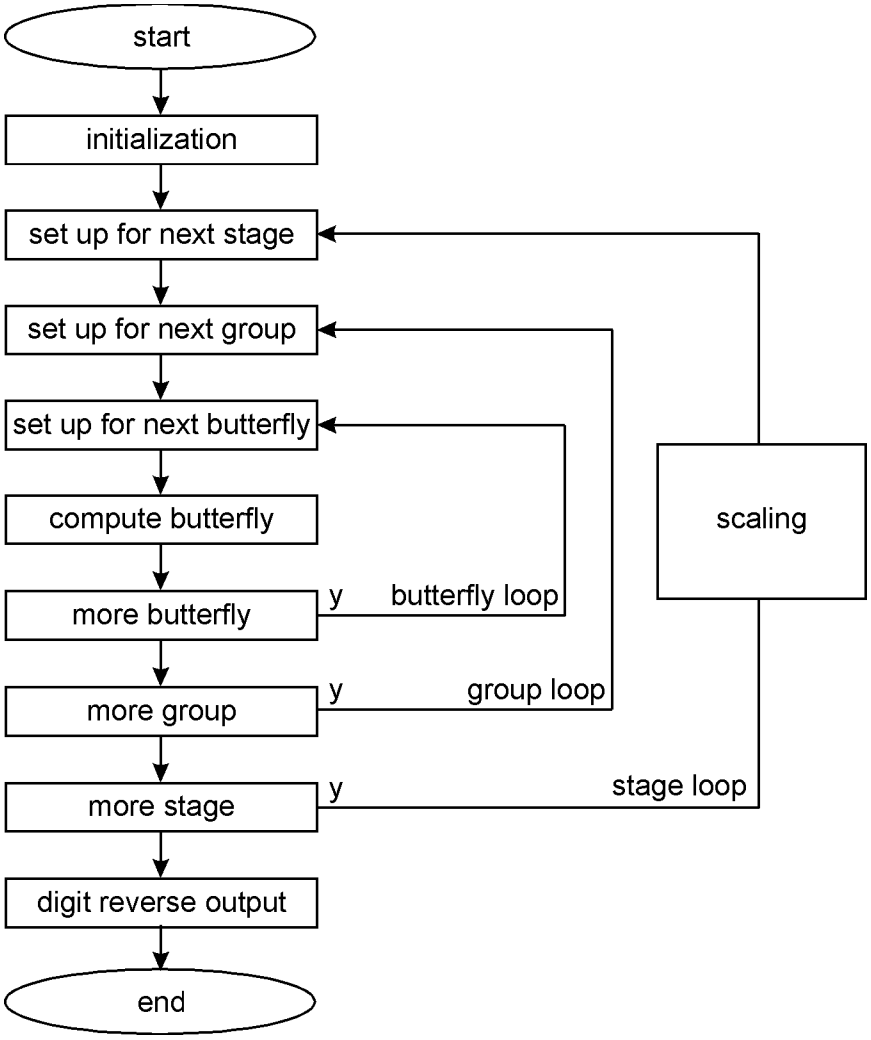
Radix-4 DIF FFT Implementation

Table 1. Summary of the Characteristics of an N-Point Radix-4 FFT

Stage		1	2	3	.....	$\log_2 N/2$
Butterfly Group		1	4	16	.....	$N/4$
Butterflies per Group		$N/4$	$N/16$	$N/64$	.....	1
Dual Node Spacing		$N/4$	$N/16$	$N/64$	.....	1
Twiddle	leg0	0	0	0	.....	0
Factor	leg2	n	$4n$	$16n$		$(N/4)n$
Exponent	leg3	$2n$	$8n$	$32n$		$(N/2)n$
	leg4	$3n$	$12n$	$48n$		$(3N/4)n$
		$n = 0 - N/4 - 1$	$n = 0 - N/16 - 1$	$n = 0 - N/64 - 1$	.....	n=0

Note: A 256 point FFT has four stages.

Figure 2. Radix-4 DIF FFT Flow Chart

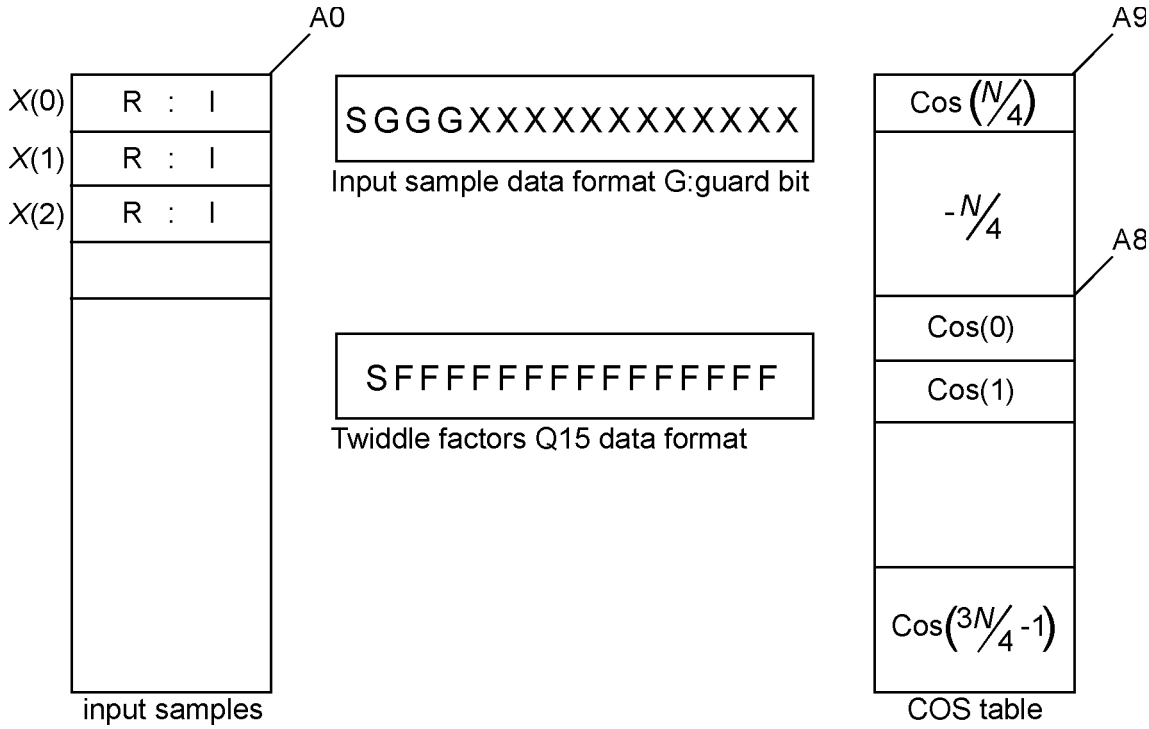


The core modules of the Radix-4 DIF FFT are the Butterfly loop, scaling, and digital reverse output.

Butterfly Loop

Assume input samples with three guard bits are stored in RAM0 of PP. The output data is stored in RAM1. The real and imaginary values are interleaved as shown in Figure 3. A0 is used to store input samples address pointers.

Figure 3. Memory Configuration and Data Format of Radix-4



To conserve the amount of data memory required to store twiddle factors, a single array of size N is used to store the real cosine value. Using an address modified value of $-N/4$ ($-X/2$ in term of angular displacement), the imaginary value (sin values) can be derived from the cosine table. This is based on the following trigonometric identity.

$$\sin(x) = \cos\left(x - \pi/2\right)$$

For example, $W_{1024}^{256} = \cos[256] - j\sin[256]$

$$= \cos[256] - j\cos[0]$$

$$= \cos[256] - j\cos[0]$$

where cos[x] is defined as $\cos\left[2\pi/N\right]$

The twiddle factor of cos is 16 bits Q15 data format as shown in Figure 3 and is stored in memory location RAM2.

ALU Multiple Arithmetic Modes

Multiple arithmetic mode is a special ALU mode that allows the ALU to operate in either a 4 byte, 2 halfword, or 1 word ALU mode. The radix-4 FFT uses the 2 halfword ALU mode to increase performance by a factor of 2.

For example, assume d3 = $x(n)$ = xa : ya
d4 = $x(n+N/4)$ = xb : yb
d5 = x(n+N/2) = xc : yc
d6 = x(n+3N/4) = xd : yd

Then use two halfword ALUs for $x(4r)$:

D7 = m D3 + D5 = xa+xc : ya+yc

where m stands for multiple arithmetic ALU.

D4 = m D4 + D6 = xb+xd : yb+yd

D6 = m D7 + D4 = (xa+xc+xb+xd) : (ya+yc+yb+yd)

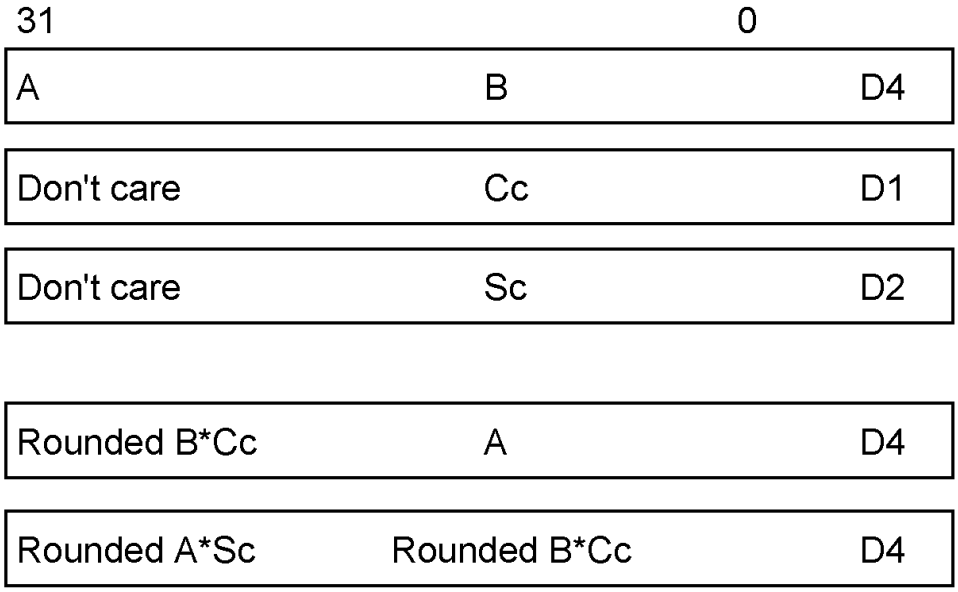
Only three steps $x(4r)$ = D6 = (xa+xc+xb+xd) : (ya+yc+yb+yd) can be obtained.

Scaling and Rounding Multiplication

A PP 16*16 multiply instruction produces a rounded 16-bit result. The instruction adds bit 15 to bits 31-16 of the multiplier output. The rounded result is written to bits 31-16 of the destination. Bits 15-0 of the destination are filled with bits 31-16 of scr3 as shown in Figure 4.

Scaling with rounding enables the PP to maintain a constant binary point while allowing constants to approach $\pm$ 4 for signed multiplies or $\pm$ 8 for unsigned multiplies.

Figure 4. Rounded Multiply



For example: To get a $x(4r+2)$ follow the steps below:

Assume: D1 = real(Wc) = Cc, D2 = image(Wc) = Sc

D7 = (xa+xc): (ya+yc), D4 = (xb+xd) : (yb+yd)

then:

D4 = m D7 - D4 = (xa+xc-xb-xd) : (ya+yc-yb-yd)

D6 = r (D4*D1) << 1 = ((ya+yc-yb-yd)Cc) : (xa+xc-xb-xd)

D6 = r (D6*D1) << 1 = ((xa+xc-xb-xd)Cc) : ((ya+yc-yb-yd)Cc)

(15)

D4 = r (D4*D2) << 1 = ((ya+yc-yb-yd)(Sc)) : (xa+xc-xb-xd)

D4 = r (D4*D2) << 1 = ((xa+xc-xb-xd)(Sc)) : ((ya+yc-yb-yd)(Sc))

(16)

D4 and D6 through *Extended ALU Operation*, $x(4r+2)$ can be obtained.

Extended ALU Operation

Extended ALU (EALU) operations configure the ALU operation in the D0 register instead of specifying it directly in the opcode. This allows more operands to be specified in the instruction and also supports an extended number of DATA Unit operations. EALU operations can execute halfword addition and halfword subtraction complicated arithmetic operations to get $x(4r+2)$.

For example:

From (15) and (16):

D6 = ((xa+xc–xb-xd)Cc) : ((ya+yc–yb-yd)Cc)
D4 = ((xa+xc–xb-xd)(Sc)) : ((ya+yc–yb-yd)(Sc))
D0 = HaddHsub

then

D4 = m EALU (HaddHsub: D6 + (D4\\16 & ~%%16 | - (D4\\16) & %%16))
= ((xa+xc–xb-xd)(Cc) + (ya+yc–yb-yd)(Sc)) : ((ya+yc–yb-yd)(Cc) - (xa+xc–xb-xd)(Sc))
= $x(4r+2)$

where D4 \\ 16 = ((ya+yc–yb-yd)(Sc)) : ((xa+xc–xb-xd)(Sc))

%%16 = 0000FFFF and ~%%16 = FFFF0000

(D4\\16 & !%%16 | -(D4\\16) &%%16)) =
(ya+yc–yb-yd)(Sc) : -(xa+xc–xb-xd)(Sc)

Local and Global Address Units

The parallel execution unit lets each PP execute up to four instructions in each cycle. A parallel execution unit consists of the following units:

- Multiplier
- One ALU or EALU
- One local access
- One global access

For example, assume the following:

D7 = xa_xc_ybydIya_yc_xb_xd = xa-xc+yb-yd : ya-yc-xb+xd
D1 = COS = Cb
D6 = tmp_xyb_c

then:

tmp_xyb_c = r (xa_xc_ybydIya_yc_xb_xd * COS) << 1 ⇒

D6 = r (D7 * D1) << 1 = (ya-yc-xb+xd) Cb : xa-xc+yb-yd (17)

Assume the following:

D5 = xa_xcIya_yc = (xa-xc) : (ya-yc)
D6 = xb_xdIyb_yd = (xb-xd) : (yb-yd)
D5 = xa_xc_ybydIya_ycxb_xd = (xa-xc-yb+yd) : (ya-yc+xb-xd)
D0 = HsubHadd

then:

xa_xc_ybydIya_ycxb_xd = m EALU (HsubHadd:D5 + (-(D6\\16) & ~%%16D4\\16 & %%16))

⇒ D5 = m EALU (HsubHadd: D5 + (-(D6\\16) &~%%16 | D6\\16 & %%16))

= (xa-xc-yb+yd) : (ya-yc+xb-xd) (18)

Equations (17) and (18) can be combined with local and global access into a parallel instruction that can be executed in one machine cycle.

```
D0 = mHsubHadd
D6 =r ( D7*D1) << 1    ;( ya-yc-xb+xd ) Cb : xa-
xc+yb-yd
   || D5 =m ealu( mHsubHadd: D5 + ( -(D6\\16) &
~%%16 | D6\\16&%%16 ) )
   || D0 = *( EALU_TABLE + [ rd_sht] )  ; load D0
for next EALU operation
   || D3 = *( A0++= [ X0] )    ; load next raidx_4
butterfly input sample
```

Where A0 ++= [X0] means using current A0 register value, but post-dec by index register X0, where X0 is the node space. The beginning and ending bracket symbols [ ], mean scale the index by data size (*4 for word, *2 for halfword, *1 for byte).

The local/global address units ADDer/Subtractor can be used to perform general-purpose addition or subtractions. That means we can have up to three 16-bit ADD/SUB in one machine cycle by using ALU and ADDer/Subtractor at the same time. For example, as shown in *Appendix A*,

```
XbxdIybyd =m xbIyb + xdIyd    ;    data unit ALU
operation
     || LEG_SPACE = &*( GZERO + L ) ; address unit
addition operation
```

where: LEG_SPACE=A11, GZERO=A15,L=X8

Zero Overhead Looping

The program flow control unit contains three compactors and associated loop control hardware that support up to three levels of zero-overhead looping. The three zero overhead control units can be used to control radix-4 FFT stage, group, and butterfly looping.

For example:

```
Let

ls0 = BUTTERFLY_LOOP_START
le0 = BUTTERFLY_LOOP_END
lr0= &*( BUTTERFLY_LOOP -1 )

lctl = 0XBA9

INSTRUCTION_1          ; Delayed slotted
instructions
INSTRUCTION_2

BUTTERFLY_LOOP_START:

    Radix-4 butterfly code segment

BUTTERFLY_LOOP_END:
```

Scaling: Using Condition and Status Protection Operation

Because the radix-4 butterfly can add three bits from input to output, the bit growth must be monitored to avoid data overflow. Three guard bits are left in input samples to avoid overflow as shown in Figure 3.

When a stage is finished, the number of output data bits to be shifted to the right depends on the maximum amount of bit growth in this stage. In conditional block scaling, data is only shifted if bit growth occurs. If one or more outputs grow, the entire block of data is shifted to the right and the block exponent is updated.

For example, if the original block exponent is 0 and the data is shifted three positions, the resulting block exponent is +3.

The following are code segments for conditional block scaling:

```
ls0 = find_max_bit_growth_start
le0 = find_max_bit_growth_end
lr0 = N - 1
sr = 0xAB
D0 = 0

lctl = 0x09
D5 = 0
D4 = *(Output_addr ++)

find_max_bit_growth_start:

   D4 =m D4  -  0

   D4 = m (-D4 & @mf | D4 & ~ @mf);(|real| : |image|)

find_max_bit_growth_end:

D5 = D5  |  D4                 ; record the max number of bit growth

 || D4 = *(Output_addr ++ )     ; of real and image part separately

D5 = D5  |  (D5\\16)           ; record the max number of bit growth

D5 = lmo(D5)


D7 = D5 – 0           ; if (D5 == 0  ||  D5 > 3) go to No_bit_growth
D7 = [u.z] 3 - D5
BR = [le] No_bit_growth
                      ; else bit growth
ls0 = Scaling_start
le0 = Scaling_end
lr0 = N-1

lctl = 0x09
D5 = D5 + 28          ; D5 = 32-max number of bit growth
D4 = *(Out_addr ++ )
```

```
Scaling_start:
  D3 = D4 >> -D5       ; shift to right with -D5 with sign extension
  D2 = D4 \\16         ; halfword swap (image: real)
  D6 = D2 >> -D5       ; shift to right with -d5 with sign extension
     || D4 = *(Output_addr --)
  D6 = D6\\16
  D2 = (D3 & ~%%16 | D2 & %%16) ;combine d3 and d2 = (real
                                          >> d5: image >> d5)
Scaling_end:
  *(Out_addr ++ [2]) = D2
```

Digital Reversed

Whereas bit reversal reverses the order of bits in binary (based 2) numbers, digital reversal reverses the order of digits in quaternary (based 4) numbers. Every two bits in the binary number system correspond to one digit in the quaternary number system. For example, binary 1011 = quaternary 23. The radix-4 DIF FFT successively divides a sequence into four subsequences, resulting in an output sequence in digital-reversal order. A digital-reversal sequence is unscrambled by digit-reversing the data positions.

In an *N*-point radix-4 FFT, the number of digits needed to represent *N* locations are reversed. Two digits are needed for a 16-point FFT, three digits for a 64-point FFT and five digits for a 1024-point FFT.

Table 2. Digit Reversal

Sample Sequential Order	Binary	Quaternary	Digit Reversed	Sample Digit-Reversed
x(0)	0000	00	00	x(0)
x(1)	0001	01	10	x(4)
x(2)	0010	02	20	x(8)
x(3)	0011	03	30	x(11)
x(4)	0100	10	01	x(1)
x(15)	1111	33	33	x(15)

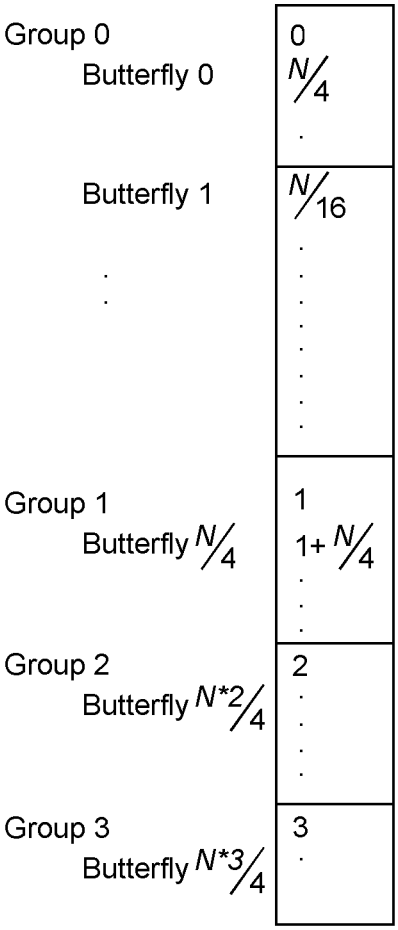
A look-up table method is suitable for digit-reversed operation.

The following algorithm will show that only a $N/16$ halfword is needed for a digit-reversal look-up table.

For *N* point FFT, the digit reversal of 0x01 is $N/4$. That means that the node space of the last stage is $N/4$.

The sequence of radix-4 FFT output is shown in Figure 5.

Figure 5. The Algorithm of Digit Reversal for Look-Up Table



	Group			
	0	1	2	3
	0	1	2	3
	N/16	N/16 +1	N/16 +2	N/16 +3
Butterfly	N*2/16	N*2/16+1	N*2/16+2	N*2/16+3
Start	N*3/16	N*3/16+1	N*3/16+2	N*3/16+3
address	.	.	.	.
	.	.	.	.
	.	.	.	

Output of Radix-4 FFT ,
Where 0, $N/16$, . are butterfly address

From Figure 5, we can group the output of radix-4 FFT into four sub groups. The starting addresses of these sub-groups are 0, 1, 2 and 3. In each sub-group, there are $N/16$ radix-4 butterflies and each radix-4 butterfly start address is $N/16$ apart. Therefore, the first subgroup radix-4 butterfly start address must be stored. In processing the first subgroup, the address of the first subgroup radix-4 butterfly will be updated by one. Using this technique, the second subgroup address can be obtained after the first subgroup is completed, and so on.

Further Improvement

In the last stage, all twiddle factors are 1. Therefore, the real and imaginary output values for the radix-4 butterfly are given by equations (19) through (26).

$$xa' = xa + xb + xc + xd \quad (19)$$

$$ya' = ya + yb + yc + yd \quad (20)$$

$$xb' = (xa+yb-xc-yd) \quad (21)$$

$$yb' = (ya-xb-yc+xd) \quad (22)$$

$$xc' = (xa-xb+xc-xd) \quad (23)$$

$$yc' = (ya-yb+yc-yd) \quad (24)$$

$$xd' = (xa-yb-xc+yd) \quad (25)$$

$$yd' = (ya+xb-yc-xd) \quad (26)$$

Two 16-bit split ALUs can be used to efficiently calculate this kind of operation.

Summary

The implementation of a radix-4 DIF FFT is based on the following PP features:

- Split ALU
- Rounded and scaling operation
- EALU and two address units
- Zero overhead looping
- Condition and status protection
- Write priority

Taking advantage of these features enables the PP to execute a 256 point complex radix-4 DIF FFT in 5k cycles. These features allow the PP to operate three or four time faster than other devices. The memory required to store input samples is N words, N words for output data, N halfword for COSIN table, and $N/16$ halfword for digit reversal look-up table.

Appendix A

```
;*****************************************************************
;*   Title:    Radix-4 DIF FFT implementation using PP of C80    *
;*   date:     06/30/1996                                        *
;*   Author:   Charles Wu                                        *
;*   RADIX_4 FFT Butterfly:                                      *
;*                                                               *
;*   (1)    xa' = xa + xb + xc + xd                              *
;*   (2)    ya' = ya + yb + yc + yd                              *
;*   (3)    xb' = (xa+yb-xc-yd)Cb + (ya-xb-yc+xd)(Sb)            *
;*   (6)    yb' = (ya-xb-yc+xd)Cb - (xa+yb-xc-yd)(Sb)            *
;*   (7)    xc' = (xa-xb+xc-xd)Cc + (ya-yb+yc-yd)(Sc)            *
;*   (8)    yc' = (ya-yb+yc-yd)Cc - (xa-xb+xc-xd)(Sc)            *
;*   (9)    xd' = (xa-yb-xc+yd)Cd + (ya+xb-yc-xd)(Sd)            *
;*   (10)   yd' = (ya+xb-yc-xd)Cd - (xa-yb-xc+yd)(Sd)            *
;*                                                               *
;*   Assume: input samples are stored in PP0 RAM0 and            *
;*           INPUT_SAMPLE_ADD is the address point. NODE_SPACE   *
;*           is the node-space. COS_TABLE and SIN_TABLE are the  *
;*           address point of cos and sin of twiddle factor.     *
;*                                                               *
;*           The following example is for 16 point FFT. For high *
;*           points FFT, N and stage_loop need to be modified.   *
;*****************************************************************
           -
          .ptext

          .global  _RADIX_4_DIF_FFT

N                                 .SET       16
SPACE                             .SET       N/4
STAGE_LOOP                        .SET       0
INPUT_SAMPLE_ADD                  .SET       A0
OUTPUT_ADDR                       .SET       A10
OUTPUT_BASED_ADDR                 .SET       A9
DIGIT_TABLE                       .SET       A3
BUTTERFLY_LOOP                    .SET       A1
GROUP_START_ADD                   .SET       A3
GROUP_LOOP                        .SET       A4
EALU_TABLE                        .SET       A8
COS_TABLE                         .SET       A9
SIN_TABLE                         .SET       A10
```

```
N_SPACE_TABLE                .SET      A2
N_SPACE                      .SET      A12
LEG_SPACE                    .SET      A11
LZERO                        .SET      A7
GZERO                        .SET      A15

NODE_SPACE                   .SET      X0
NODE_SPACE_3                 .SET      X1
N_NODE_SPACE                 .SET      X2

LEG1                         .SET      X8
LEG2                         .SET      X9
LEG3                         .SET      X10
L                            .SET      X8
IDX_SPACE                    .SET      X8


COS                          .SET      D1
SIN                          .SET      D2
xaIya                        .SET      D3
xbIyb                        .SET      D4
xcIyc                        .SET      D5
xdIyd                        .SET      D6
xaxcIyayc                    .SET      D7
xbxdIybyd                    .SET      D4
o_xaya                       .SET      D6
tmp_xyc_                     .SET      D4
tmp_xyc_c                    .SET      D6
xyc_c                        .SET      D6
tmp_xyc_s                    .SET      D4
xyc_s                        .SET      D4
o_xcyc                       .SET      D4
xxbIyyb                      .SET      D7
xxdIyyd                      .SET      D6
xa_xcIya_yc                  .SET      D5
xb_xdIyb_yd                  .SET      D6
xa_xcyb_ydIya_yc_xbxd        .SET      D7
tmp_xyb_c                    .SET      D6
xa_xc_ybydIya_ycxb_xd        .SET      D5
xyb_c                        .SET      D6
tmp_xyb_s                    .SET      D7
xyb_s                        .SET      D7
tmp_xyd_c                    .SET      D6
xyd_c                        .SET      D6
```

```
tmp_xyd_s                       .SET    D7
xyd_s                           .SET    D7
o_xbyb                          .SET    D7
o_xdyd                          .SET    D7
xaxcIyayc_1                     .SET    D1
xbxdIybyd_2                     .SET    D2
o_xaya_7                        .SET    D7
o_xcyc_7                        .SET    D7
xa_xcIya_yc_1                   .SET    D1
xb_xdIyb_yd_2                   .SET    D2
o_xbyb_7                        .SET    D7
o_xdyd_7                        .SET    D7


N_DIV_4                         .SET    N/2
RAM0                            .SET    0X00000000
RAM2                            .SET    0X00008000
RAM1                            .SET    RAM0+800H
RAM2_400H                       .SET    RAM2+400H
RAM2_100H                       .SET    RAM2+100H
EALU                            .SET    400H
DIGIT                           .SET    EALU+40H
DIGIT_LOOP                      .SET    N/16-1
BASE                            .SET    N/16
rd_sht                          .SET    0
addsub                          .SET    1
subadd                          .SET    2
maddsub                         .SET    3
msubadd                         .SET    4

;*********************************
;   test
;*********************************
        .data

cosin   .half   0
        .half   32768*3826/10000
        .half   32768*707/1000
        .half   32768*9238/10000
        .half   32768*9999/10000
        .half   32768*9238/10000
        .half   32768*707/1000
        .half   32768*3826/10000
        .half   0
```

```
        .half   32768*(-3826)/10000
        .half   32768*(-707)/1000
        .half   32768*(-9238)/10000
        .half   32768*(-9999)/10000
        .half   32768*(-9238)/10000
        .half   32768*(-707)/1000
        .half   32768*(-3826)/10000

        .ptext
;**********************************
_RADIX_4_DIF_FFT:
;**********************************
;
;             radix_4 dif fft loop initalization

    SR = 0x05

    EALU_TABLE = &*(PBA+EALU)         ;table for various EALU operation

    COS_TABLE = &*(DBA+RAM2_100H)     ;table for cos of twiddle factor

    SIN_TABLE = &*(COS_TABLE-N_DIV_4) ;table for sin of twiddle factor

    le0 = move_end
    ls0 = move_start
    lr0 =  N
    lctl = 0x09

    a0 = cosin
    nop
move_start:

       d0 =h *( a0 ++ )
move_end:   *(SIN_TABLE ++ ) = h d0


    SIN_TABLE=&*(COS_TABLE-N_DIV_4)   ;table for sin of twiddle factor


    le0 = fill_digit_reversal_start
    ls0 = fill_digit_reversal_end
    lr0 = DIGIT_LOOP

    DIGIT_TABLE = &*(PBA + DIGIT)     ;DIGIT REVERSAL LOOK-UP TABLE
```

```
    lctl = 0x09

    D0 = 0   ;;;;;;;;;;;;;;;
    D1 = BASE  ;;;;;;;;;;;;;;;

fill_digit_reversal_start:
                        *( DIGIT_TABLE ++ ) =h  D0
fill_digit_reversal_end:      D0 = D0+D1

    DIGIT_TABLE = &*(PBA + DIGIT)   ;

    D0 = ROUND_SHIFT                ;
    *(EALU_TABLE + [rd_sht]) = D ;store ROUND_SHIFT --> EALU_TABLE + [0]
    D0 = HaddHsub


;**********************************
;     RADIX_4 FFT STAGE LOOP MODULE
;**********************************

    ls2 = STAGE_LOOP_START          ;
    le2 = STAGE_LOOP_END            ; set stage loop start/end address
    lr2 = STAGE_LOOP                ; set loop count

    *(EALU_TABLE + [addsub]) = D0   ;store HaddHsub -> EALU_TABLE + [1]

    D0 = HsubHadd
    *(EALU_TABLE + [subadd]) = D0   ;store HsubHadd -> EALU_TABLE + [2]


    D0 = mHaddHsub
    *(EALU_TABLE + [maddsub]) = D0 ;store mHaddHsub -> EALU_TABLE + [3]

    D0 = mHsubHadd
    *(EALU_TABLE + [msubadd]) = D0 ;store mHsunHadd -> EALU_TABLE + [4]



    NODE_SPACE = N                  ; initial NODE_SPACE = N
    GROUP_LOOP = 1

    lctl = 0x0b00
```

```
    BUTTERFLY_LOOP  = N
    N_SPACE = 1

;**********************************
STAGE_LOOP_START:
;**********************************

    D0 = NODE_SPACE
    D0 = D0 >> 2
    NODE_SPACE = D0           ; update node_space --> node_space/4
    D1 = D0 << 1
    N_NODE_SPACE = D0 + 1
    NODE_SPACE_3 = D1 + D0

    ls1 = GROUP_LOOP_START
    le1 = GROUP_LOOP_END       ; set up group loop start/end address
    lr1 = h&*(GROUP_LOOP -1)        ; set group loop count
    GROUP_LOOP = GROUP_LOOP << 2

    lctl = 0x0ba0

    GROUP_START_ADDR = &*(LZERO)    ;set group start address=0

    BUTTERFLY_LOOP=BUTTERFLY_LOOP>>2




;**********************************
GROUP_LOOP_START:
;**********************************

    le0 = BUTTERFLY_LOOP_END

    INPUT_SAMPLE_ADDR = &*(GROUP_START_ADDR)    ;input_sample_add =
group
    ls0 = BUTTERFLY_LOOP_START      ;set up butterfly loop start/end
    xaIya = *(INPUT_SAMPLE_ADDR++=[NODE_SPACE]) ;load first value of
radix_4
                                    ;butterfly (xa:ya)
    lr0 = h &*(BUTTERFLY_LOOP -1)   ;

    lctl = 0xba9
```

```
    xbIyb = *(INPUT_SAMPLE_ADDR++=[NODE_SPACE]) ;load second value of
radix-4
                                        || LEG2 = GZERO    ;butterfly loop
(xb:yb). zero leg1
    xcIyc=*(INPUT_SAMPLE_ADDR++=[NODE_SPACE])   ;load third value of
radix-4
                                        || L = GZERO ;butterfly loop
(xc:yc). zero L

;**********************************
BUTTERFLY_LOOP_START:
;**********************************

    xdIyd = *(INPUT_SAMPLE_ADDR--=[NODE_SPACE_3]) ;load last value of
radix-4
       || xaxcIyayc = m xaIya + xcIyc ;first + third using 16-bit ALU
       || COS = h *(COS_TABLE + [LEG2]) ; load COS(2L)


    xbxdIybyd = m xbIyb + xdIyd     ; second + last using 16-bit ALU
       || LEG_SPACE = &*(GZERO + L)  ; LEG_SPACE = 0+L


    o_xaya = m xaxcIyayc + xbxdIybyd ; xa'=xa+xb+xc+xd;ya'=ya+yb+yc+yd
       || SIN = h*(SIN_TABLE+[LEG2]) ; Load sin (2L)


    tmp_xyc_ = m xaxcIyayc-xbxdIybyd ; temp_xyc_= xa+xc-xb-xd : ya+yc-
yb-yd
       || *(INPUT_SAMPLE_ADDR ++= [NODE_SPACE]) = o_xaya
       || D0 = *(EALU_TABLE + rd_sht )     ; do=round_shift


    tmp_xyc_c = r (tmp_xyc_ * COS)<<1 ;temp_xyc_c = (ya+yc-yb-yd) cos :
xa+xc-xb-xd
       || ealu(ROUND_SHIFT)
       || LEG3 = &*( LEG_SPACE + LEG2 ) ; LEG3 = L + 2L = 3L


    xyc_c = r (tmp_xyc_c * COS ) << 1 ;xyc_c = (xa+xc-xb-xd)cos :
(ya+yc-yb-yd)cos
       || ealu(ROUND_SHIFT)
       || COS = h*(COS_TABLE + [LEG1]); Load cos(L)
```

```
    tmp_xyc_s = r (tmp_xyc_ * SIN)<<1;temp_xyc_s = (ya+yc-yb-yd) sin :
xa+xc-xb-xd
    || ealu(ROUND_SHIFT)
    || xxbIyyb=*(INPUT_SAMPLE_ADDR++ =[NODE_SPACE]);xxbIyyb = xbIyb

    xyc_s = r (tmp_xyc_s * SIN) << 1;xyc_s = (xa+xc-xb-xd) sin:(ya+yc-
yb-yd)sin
    || ealu(ROUND_SHIFT)
    || D0 = *(EALU_TABLE + [addsub]) ; do = HaddHsub
    || D0 = *(INPUT_SAMPLE_ADDR ++ =[NODE_SPACE];input_sample +=
node_space


    o_xcyc=m ealu(HaddHsub: xyc_c+(xyc_s\\16&~%%16 | -(xyc_s\\16) &
%%16))
    || xxdIyyd = *(INPUT_SAMPLE_ADDR--= [NODE_SPACE]);xxdIyyd = xdIyd
    || SIN =h *( SIN_TABLE + [LEG1]);sin =sin(2l)
;; xc':yc' = (xa+xc-xb-xd)cos+(ya+yc-yb-yd)sin : (ya+yc-yb-yd)cos-
(xa+xc-xb-xd)sin


    xa_xcIya_yc = m xaIya – xcIyc   ; xa-xc:ya-yc
    || *(INPUT_SAMPLE_ADDR-- = [NODE_SPACE ] ) = o_xcyc


    xb_xdIyb_yd = m xxbIyyb – xxdIyyd     ; xb-xd : yb-yd
    || D3 = *(INPUT_SAMPLE_ADDR--= [NODE_SPACE]) ;input_sample - =node
space
    || D0 = *(EALU_TABLE + [addsub])


    xa_xcyb_ydIya_yc_xbxd= m ealu(HaddHsub:xa_xcIya_yc +(xb_xdIyb_yd\\16
& ~%%16 | -(xb_xdIyb_yd\\16) & %%16))
    || D0 = *(INPUT_SAMPLE_ADDR ++= [1]);just to update input_sample_add
    || D0 = *(EALU_TABLE + [msubadd])
;; result = xa_xcyb_yd-ya_yc_xbxd = xa-xc+yb-yd : ya-yb-xb+xd


    tmp_xyb_c =r (xa_xcyb_ydIya_yc_xbxd * COS)<<1
    || xa_xc_ybydIya_ycxb_xd = m ealu(mHsubHadd: xa_xcIya_yc+(-
(xb_xdIyb_yd\\16) & ~%%16 | xb_xdIyb_yd\\16 & %%16))
    || d0 = *(EALU_TABLE + [rd_sht])
    || xaIya = *(INPUT_SAMPLE_ADDR ++=[NODE_SPACE])
; temp_xyb_c = (ya-yc-xb_xd)cos : xa-xc+yb-yd
```

```
; xa_xc_ybyd-ya_ycxb_xd = xa-xc-yb+yd : ya-yc+xb-xd


    xyb_c = r (tmp_xyb_c * COS)<<1;xyb_c = (xa-xc+yb-yd)cos : (ya-yc-
xb_xd)cos
    || ealu(ROUND_SHIFT)


    tmp_xyb_s = r (xa_xcyb_ydIya_yc_xbxd * SIN)<<1
    || ealu(ROUND_SHIFT)
    || COS = h*(COS_TABLE + [LEG3])  ; cos = cos(3l)
    || xbIyb = *(INPUT_SAMPLE_ADDR)
;;temp_xyb_s = (ya-yc-xb+xd)sin : xa-xc+yb-yd


    xyb_s = r (tmp_xyb_s * SIN)<<1
    || ealu(ROUND_SHIFT)                  ; using write priority
    || D0 = *(EALU_TABLE + [maddsub])     ; do = mHaddHsub
    || D0 = *(INPUT_SAMPLE_ADDR --= [1])   ;just to update
input_sample_add
;;xyb_s = (xa-xc+yb-yd)sin:(ya-yc-xb+xd)sin

    tmp_xyd_c=r (xa_xc_ybydIya_ycxb_xd * COS) << 1
    || o_xbyb = m ealu(mHaddHsub:xyb_c+(xyb_s\\16 & ~%%16 | (-xyb_s\\16)
& %%16))
    || D0 = *(EALU_TABLE + [rd_sht]) ; D0 = round_shift
;;  tmp_xyd_c =  (  ya-yc+xb-xd ) cos : xa-xc-yb+yd
;;  xb'yb' = (xa-xc+yb-yd)cos + (ya-yc-xb+xd)sin:(ya-yc-xb_xd)cos -(xa-
xc+yb-yd)sin

    xyd_c =r(tmp_xyd_c * COS) << 1 ;(xa-xc-yb+yd) cos: (ya-yc+xb-xd) cos
    || ealu(ROUND_SHIFT)
    || SIN = h *(SIN_TABLE + [LEG3]) ; sin = sin(3l)
    || *( INPUT_SAMPLE_ADDR ++= [N_NODE_SPACE]) = o_xbyb


    tmp_xyd_s=r(xa_xc_ybydIya_ycxb_xd*SIN)<<1 ;ya-yc+xb-xd)sin : xa-xc-
yb+xd
    || ealu(ROUND_SHIFT)
    || L = &*(N_SPACE + L)           ; l = leg_space+l

    xyd_s = r (tmp_xyd_s * SIN) << 1; (xa-xc-yb+xd) sin : ( ya-yc+xb-
xd)sin
    || ealu(ROUND_SHIFT)
```

```
    || D0 = *(EALU_TABLE + [addsub]) ; D0 = 16
    || xcIyc = *(INPUT_SAMPLE_ADDR ++= [NODE_SPACE])

    o_xdyd = m ealu(HaddHsub: xyd_c + ((xyd_s\\16) & ~%%16 | -
(xyd_s\\16) & %%16))
    || D0 = &*(INPUT_SAMPLE_ADDR --)
;;xd'yd' = (xa-xc-yb+yd)cos + (ya-yc+xb-xd)sin : (ya-yc+xb-xd)cos -(xa-
xc-yb+xd) sin


;*********************************
BUTTERFLY_LOOP_END:

    *(INPUT_SAMPLE_ADDR ++= [1]) = o_xdyd
       || LEG2 = L << 1

;*********************************

    D0 = NODE_SPACE
    D0 =D0 << 2
    D1 = GROUP_START_ADD

;*********************************
GROUP_LOOP_END:

    GROUP_START_ADD = D0 + D1    ; group_start_add += node_space * 4

STAGE_LOOP_END:

    N_SPACE = N_SPACE << 2

;*********************************


;*********************************
;   LAST STAGE LOOP
;*********************************

    ls1 = L_GROUP_LOOP_START
    le1 = L_GROUP_LOOP_END      ; set up group loop start/end address
    lr1 = 3                     ; set group loop count

    INPUT_SAMPLE_ADDR = &*(DBA);input_sample_add = ram0
    OUTPUT_BASED_ADDR = &*(DBA + RAM1);
```

```
    xaIya = *(INPUT_SAMPLE_ADDR ++= [1]); load first value of radix_4

    xbIyb = *(INPUT_SAMPLE_ADDR ++= [1]);load second value of radix-4

    lctl = 0x0a0

    xcIyc = *(INPUT_SAMPLE_ADDR ++= [1]);load third value of radix-4

    xdIyd = *(INPUT_SAMPLE_ADDR ++= [1]);load last value of radix-4


;*********************************
L_GROUP_LOOP_START:
;*********************************

    ls0 = DIGIT_LOOP_START
    le0 = DIGIT_LOOP_END        ; set up group loop start/end address
    lr0 = DIGIT_LOOP            ; set group loop count
    lctl = 0x0a9

    DIGIT_TABLE = &*(PBA + DIGIT)
    A12 = 1

DIGIT_LOOP_START:

    IDX_SPACE =h *( DIGIT_TABLE )

    xaxcIyayc_1 = m xaIya + xcIyc   ;  first + third  using 16-bit ALU


    xbxdIybyd_2 = m xbIyb + xdIyd   ; second + last using 16-bit ALU
       || OUTPUT_ADDR = &*(OUTPUT_BASED_ADDR + [IDX_SPACE])

    o_xaya_7 = m xaxcIyayc_1 + xbxdIybyd_2;;xa' = xa+xb+xc+xd ; ya'=
ya+yb+yc+yd;;
       || D0 = h &* (A12 + IDX_SPACE)

    o_xcyc_7 = m xaxcIyayc_1 - xbxdIybyd_2
       || *(OUTPUT_ADDR ++= [SPACE]) = o_xaya_7
       || (DIGIT_TABLE ++) =h D0
;;;;;;;;; temp_xyc_ = xa+xc-xb-xd : ya+yc-yb-yd ;;;;;;;;;;;;;;;;

    xa_xcIya_yc_1 = m xaIya - xcIyc ; xa-xc : ya-yc
```

```
       || xaIya = *(INPUT_SAMPLE_ADDR ++=[1])
       || d0 = *(EALU_TABLE + [addsub])

    xb_xdIyb_yd_2 =m xbIyb - xdIyd  ;;;;; xb-xd : yb-yd ;;;;;;;;;;
       || *(OUTPUT_ADDR ++= [SPACE]) = o_xcyc_7
       || xbIyb = *(INPUT_SAMPLE_ADDR ++=[1])

    o_xbyb_7 = m ealu(HaddHsub: xa_xcIya_yc_1 + (xb_xdIyb_yd_2\\16 &
~%%16 | -(xb_xdIyb_yd_2\\16) & %%16))
       || xcIyc=*(INPUT_SAMPLE_ADDR ++=[1])
       || d0 = *(EALU_TABLE +  [ subadd] )
;;;;;;;;; result = xb'yb' =  xa-xc+yb-yd : ya-yb-xb+xd ;;;;;;;;;


    o_xdyd_7 = m ealuf(HsubHadd:xa_xcIya_yc_1+(-(xb_xdIyb_yd_2 \\16) &
~%%16 | xb_xdIyb_yd_2\\16 & %%16))
       || *( OUTPUT_ADDR ++= [ SPACE ] ) = o_xbyb_7
;;;;;;;;;; result = xb'yb' = xa-xc+yb-yd : ya-yb-xb+xd ;;;;;;;;;

;*********************************
DIGIT_LOOP_END:
L_GROUP_LOOP_END:
    xdIyd = *(INPUT_SAMPLE_ADDR ++=[1])
       || *(OUTPUT_ADDR ++=[SPACE]) = o_xdyd_7
;*********************************

       .end
```