# REAL TIME DIGITAL SIGNAL PROCESSING

# Frequency Analysis

## Fast Fourier Transform (FFT)

# Fast Fourier Transform

**DFT:** $$X[k]=\sum_{n=0}^{N-1}x[n]W_N^{kn},\qquad k=0,1,\dots,N-1,$$

$N^2$ complex multiplications

$N(N-1)$ complex aditions

$$X[k]=\sum_{n=0}^{N-1}\left[(\mathcal{R}e\{x[n]\}\mathcal{R}e\{W_N^{kn}\}-\mathcal{J}m\{x[n]\}\mathcal{J}m\{W_N^{kn}\})\right.$$
$$\left.+\,j(\mathcal{R}e\{x[n]\}\mathcal{J}m\{W_N^{kn}\}+\mathcal{J}m\{x[n]\}\mathcal{R}e\{W_N^{kn}\})\right],$$
$$k=0,1,\dots,N-1,$$

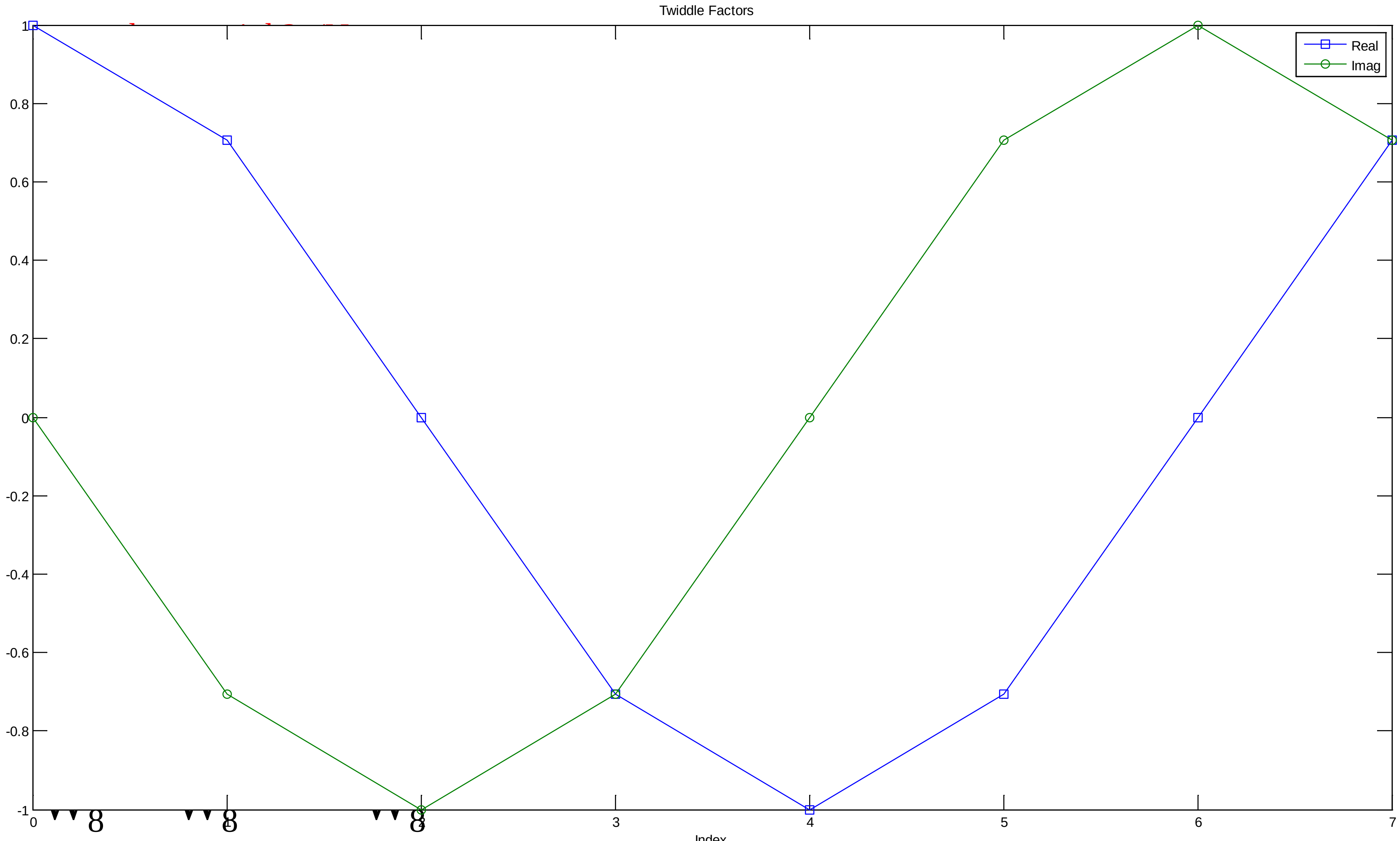
$4N^2$ real multiplications

$N(4N-1)$ real aditions

1. $W_N^{k[N-n]}=W_N^{-kn}=(W_N^{kn})^*$ (complex conjugate symmetry);
2. $W_N^{kn}=W_N^{k(n+N)}=W_N^{(k+N)n}$ (periodicity in $n$ and $k$).

Computational algorithms that exploit both the symmetry and the periodicity of the sequence $W_N^{kn}$ has come to be know as the fast Fourier transform, or FFT.

# Applying the properties of symmetry and periodicity to $W_N^r$ for N=8



# Decimation-In-Time FFT algorithms I

$$X[k] = \underbrace{\sum_{n \text{ even}} x[n]W_N^{nk}} + \underbrace{\sum_{n \text{ odd}} x[n]W_N^{nk}}$$

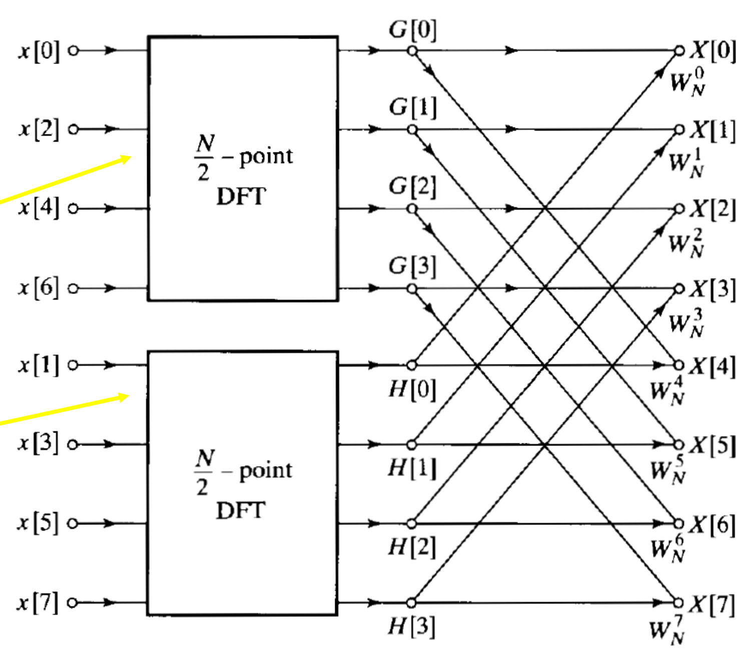
$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r]W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x[2r+1]W_N^{(2r+1)k}$$

$$= \sum_{r=0}^{(N/2)-1} x[2r](W_N^2)^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1](W_N^2)^{rk}$$

$$\boxed{W_N^2 = e^{-2j(2\pi/N)} = e^{-j2\pi/(N/2)} = W_{N/2}}$$

$$X[k] = \boxed{\sum_{r=0}^{(N/2)-1} x[2r]W_{N/2}^{rk}} + W_N^k \boxed{\sum_{r=0}^{(N/2)-1} x[2r+1]W_{N/2}^{rk}}$$

$$= G[k] + W_N^k H[k], \qquad k = 0, 1, \ldots, N-1.$$

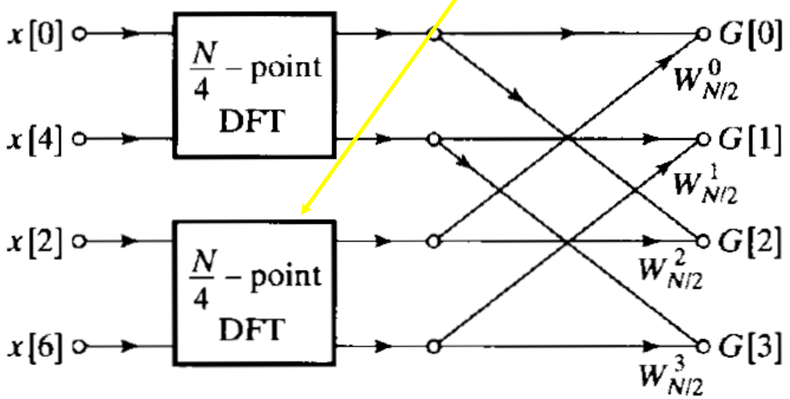


# Decimation-In-Time FFT algorithms

$$G[k]=\sum_{r=0}^{(N/2)-1} g[r]W_{N/2}^{rk}=\sum_{\ell=0}^{(N/4)-1} g[2\ell]W_{N/2}^{2\ell k}+\sum_{\ell=0}^{(N/4)-1} g[2\ell+1]W_{N/2}^{(2\ell+1)k},$$

or

$$G[k]=\boxed{\sum_{\ell=0}^{(N/4)-1} g[2\ell]W_{N/4}^{\ell k}}+W_{N/2}^{k}\boxed{\sum_{\ell=0}^{(N/4)-1} g[2\ell+1]W_{N/4}^{\ell k}}.$$

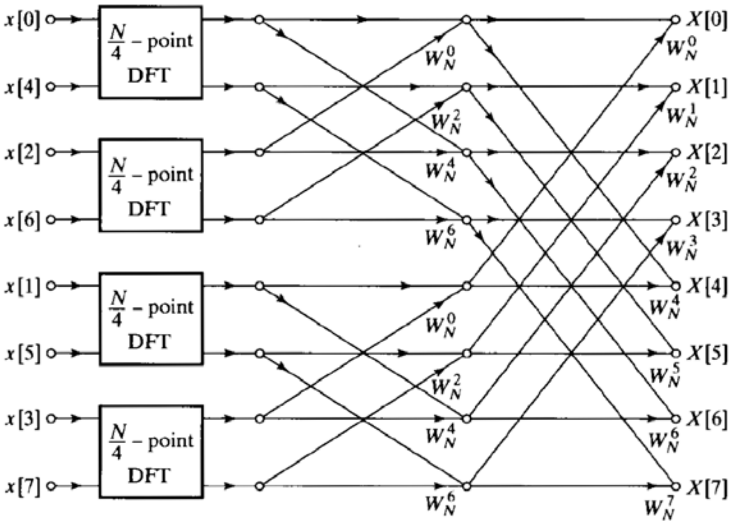


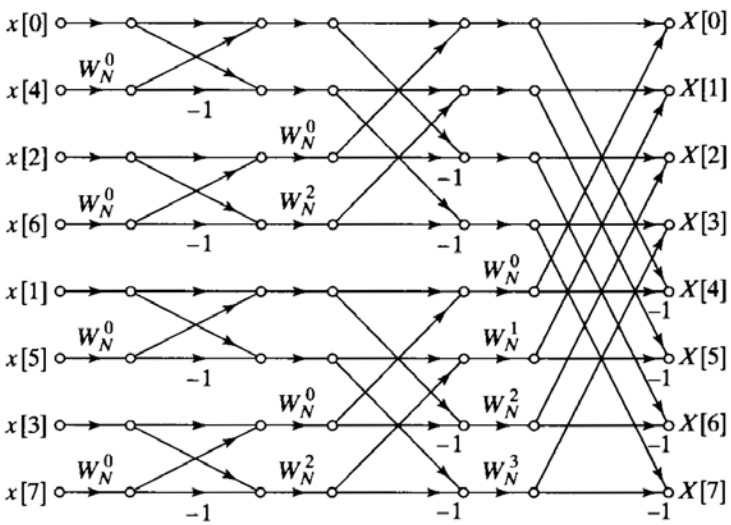

$$W_{N/2}^{k}=e^{-j2\pi k/(N/2)}=W_{N}^{2k}$$
$$W_{N/2}^{0}=W_{N}^{0}$$
$$W_{N/2}^{1}=W_{N}^{2}$$
$$W_{N/2}^{2}=W_{N}^{4}=-W_{N}^{0}$$
$$W_{N/2}^{3}=W_{N}^{6}=-W_{N}^{2}$$

Similarly, $H[k]$ would be represented as

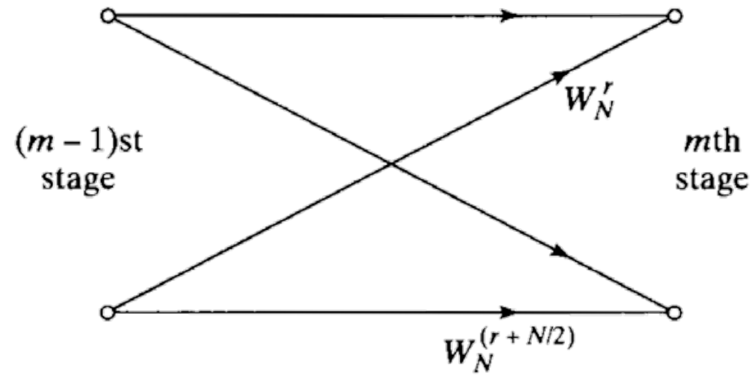
$$H[k]=\sum_{\ell=0}^{(N/4)-1} h[2\ell]W_{N/4}^{\ell k}+W_{N/2}^{k}\sum_{\ell=0}^{(N/4)-1} h[2\ell+1]W_{N/4}^{\ell k}.$$

# Decimation-In-Time FFT algorithms

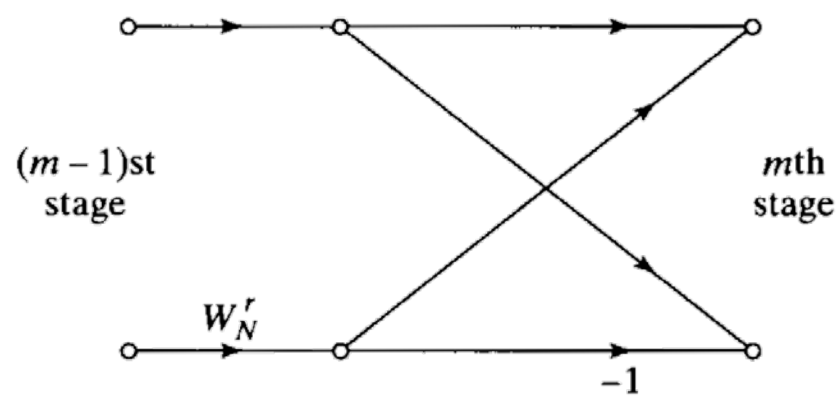



$N \log_2 N$ complex multiplications and complex aditions




$$\boxed{W_N^{r+N/2} = W_N^{N/2} W_N^r = -W_N^r.}$$




$N/2 \log_2 N$ complex multiplications and $N \log_2 N$ complex aditions

# FFT vs. DFT

- **The FFT is simply an algorithm for efficiently calculating the DFT**

- **Computational efficiency of an N-Point FFT:**
  - **DFT:** $N^2$ **Complex Multiplications**
  - **FFT:** $(N/2) \log_2(N)$ **Complex Multiplications**

| N | DFT Multiplications | FFT Multiplications | FFT Efficiency |
|---|---|---|---|
| 256 | 65,536 | 1,024 | 64 : 1 |
| 512 | 262,144 | 2,304 | 114 : 1 |
| 1,024 | 1,048,576 | 5,120 | 205 : 1 |
| 2,048 | 4,194,304 | 11,264 | 372 : 1 |
| 4,096 | 16,777,216 | 24,576 | 683 : 1 |

# Bit Reversal

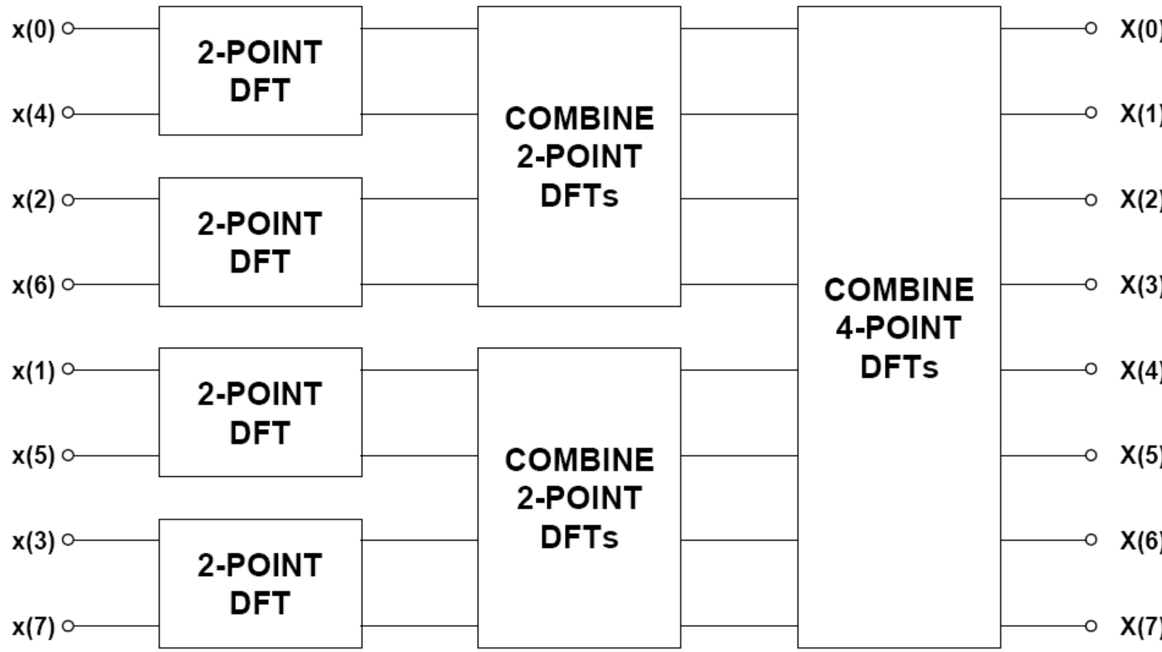
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Decimal Number : | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Binary Equivalent : | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
| | | | | | | | | |
| Bit-Reversed Binary : | 000 | 100 | 010 | 110 | 001 | 101 | 011 | 111 |
| Decimal Equivalent : | 0 | 4 | 2 | 6 | 1 | 5 | 3 | 7 |

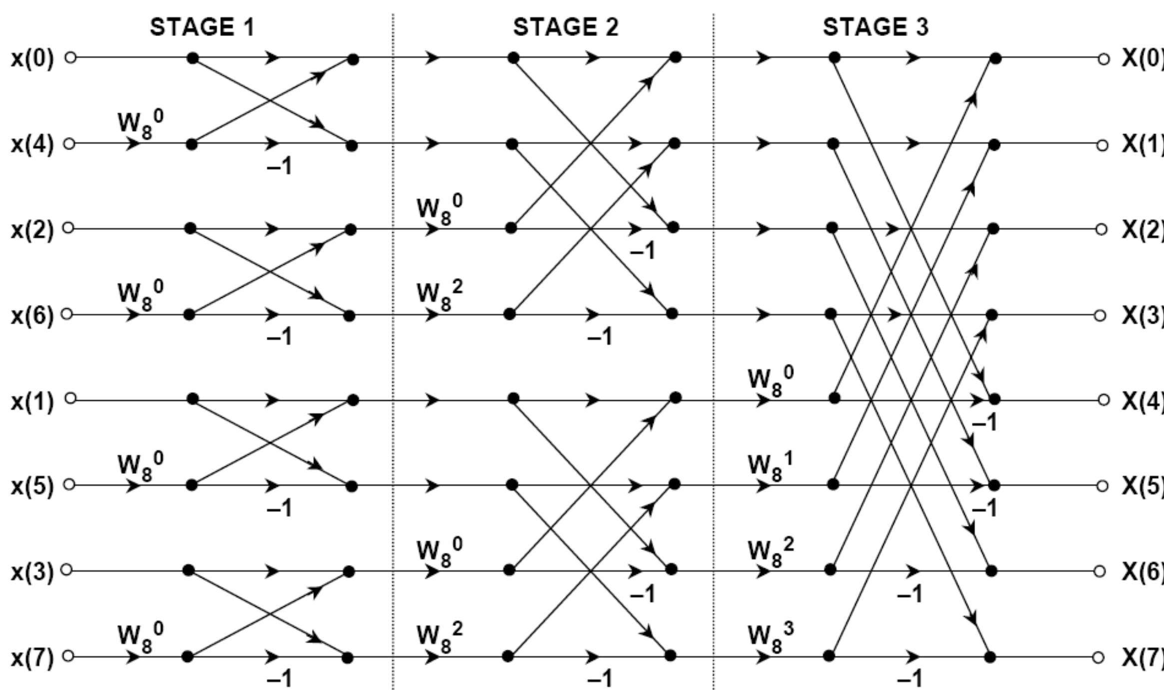
- The bit reversal algorithm used to perform the re-ordering of signals.
- The decimal index, n, is converted to its binary equivalent.
- The binary bits are then placed in reverse order, and converted back to a decimal number.
- Bit reversing is often performed in DSP hardware in the data address generator (DAG).

# DIT FFT

- Input signal must be properly re-ordered using a *bit reversal* algorithm
- In-place computation
- Number of stages: $log_2 N$
- Stage 1: all the twiddle factors are 1
- Last Stage: the twiddle factors are in sequential order

| | Stage 1 | Stage 2 | Stage 3 | Stage $Log_2N$ |
|---|---|---|---|---|
| Number of Groups | N/2 | N/4 | N/8 | 1 |
| Butterflies per Group | 1 | 2 | 4 | N/2 |
| Dual-Node Spacing | 1 | 2 | 4 | N/2 |
| Twiddle Factor Exponents | (N/2)k, k=0 | (N/4)k, k=0,1 | (N/8)k, k=0,1, 2,3 | k, k=0 to N/2–1 |

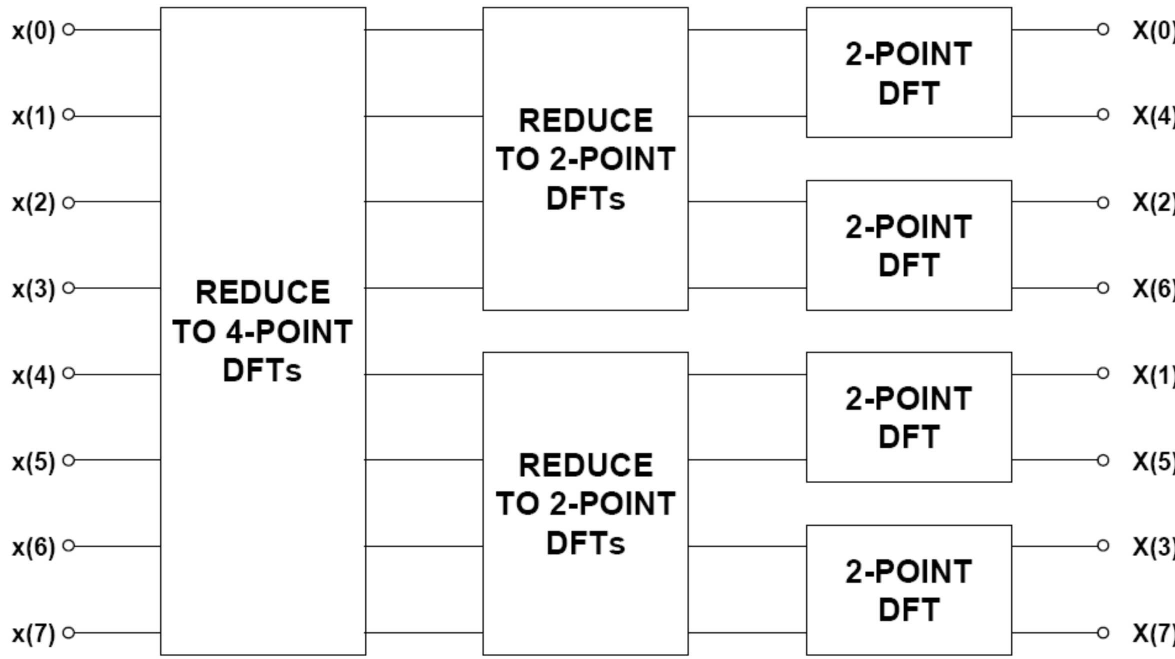


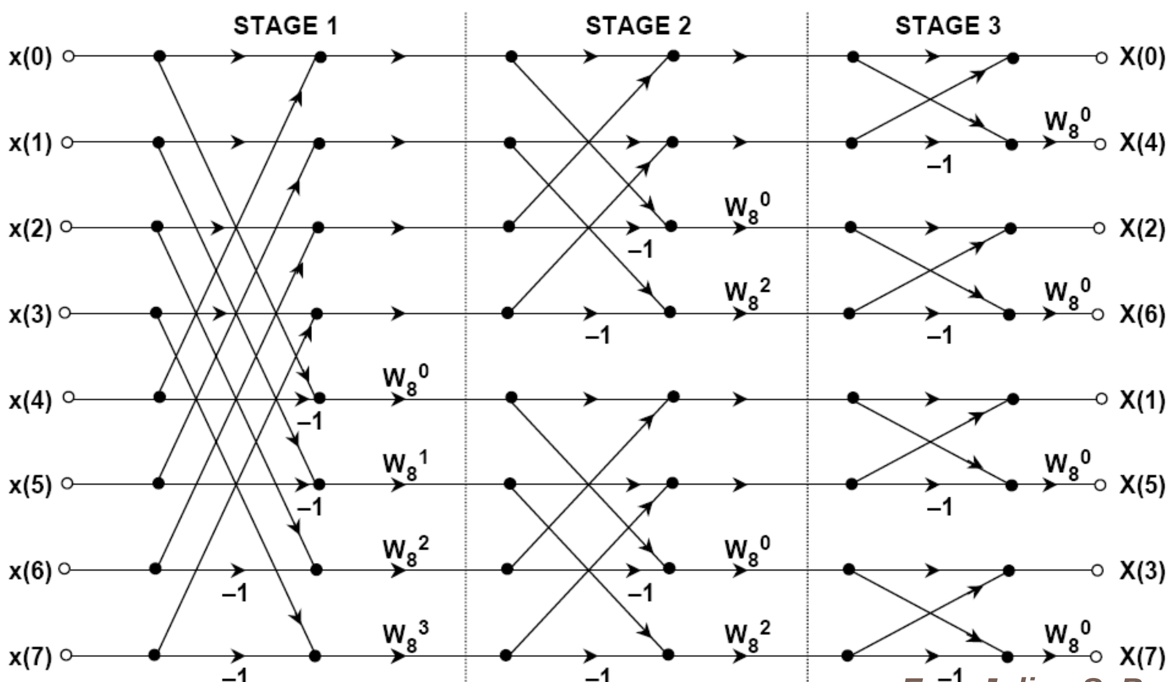



# DIF FFT

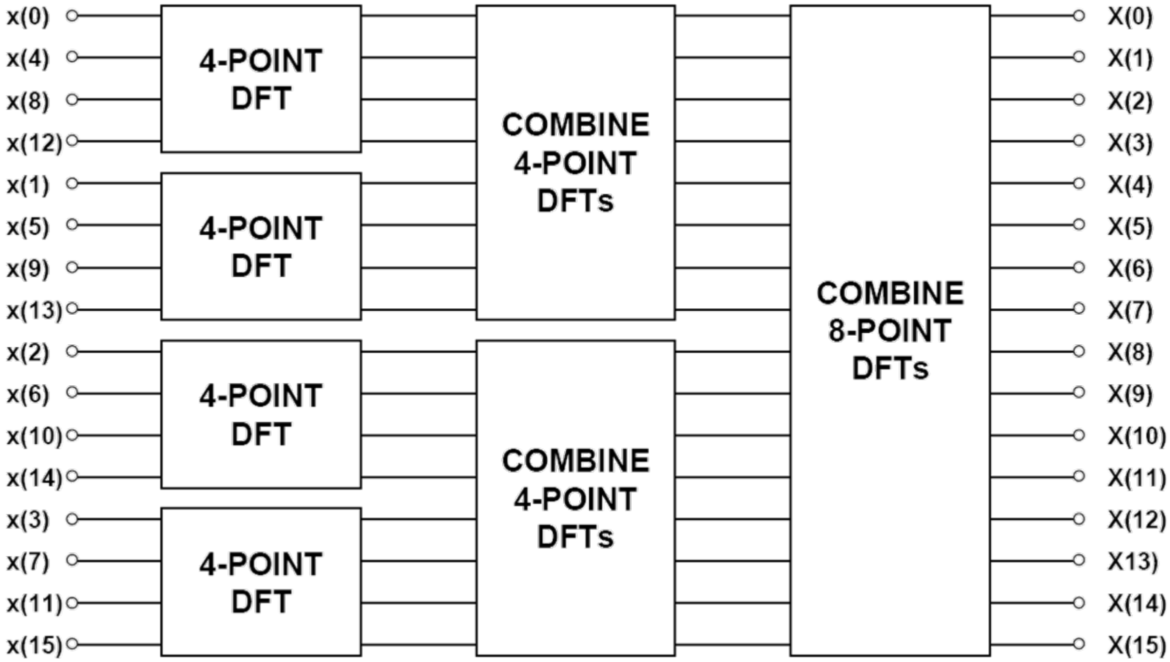
- Output signal must be properly re-ordered using a *bit reversal* algorithm
- In-place computation
- Number of stages: $log_2 N$
- Stage 1: the twiddle factors are in sequential order
- Last Stage: all the twiddle factors are 1

| | Stage 1 | Stage 2 | Stage 3 | Stage $Log_2N$ |
|---|---|---|---|---|
| Number of Groups | 1 | 2 | 4 | N/2 |
| Butterflies per Group | N/2 | N/4 | N/8 | 1 |
| Dual-Node Spacing | N/2 | N/4 | N/8 | 1 |
| Twiddle Factor Exponents | n, n=0 to N/2 - 1 | 2n, n=0 to N/4 - 1 | 4n, n=0 to N/8 - 1 | (N/2)n, n=0 |

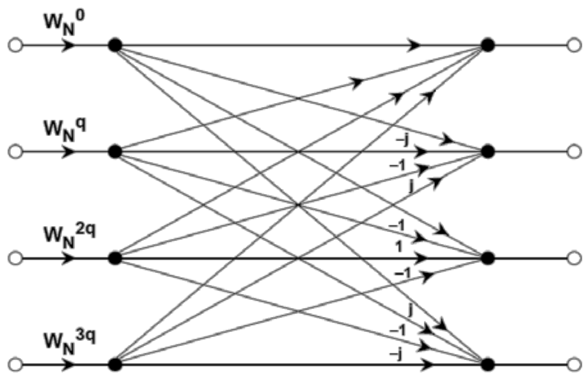


# Radix-4 Decimation-In-Time FFT Algorithm




- A radix-4 FFT combines two stages of a radix-2 FFT into one, so that half as many stages are required.
- The radix-4 butterfly is consequently larger and more complicated than a radix-2 butterfly.
- N/4 butterflies are used in each of $(\log_2 N)/2$ stages, which is one quarter the number of butterflies in a radix-2 FFT.
- Addressing of data and twiddle factors is more complex, a radix-4 FFT requires fewer calculations than a radix-2 FFT.
- It can compute a radix-4 FFT significantly faster than a radix-2 FFT



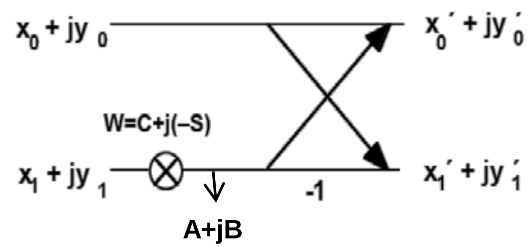
# Hardware benchmark comparisons

- **ADSP-2189M, 16-bit, Fixed-Point @ 75MHz**
  - **453μs (1024-Point)**
- **ADSP-21160 SHARC™, 32-bit, Floating-Point @ 100MHz**
  - **180μs (1024-Point), 2 channels, SIMD Mode**
  - **115μs (1024-Point), 1 channel, SIMD Mode**
- **ADSP-TS001 TigerSHARC™ @ 150MHz,**
  - **16-bit, Fixed-Point Mode**
    - **7.3μs (256-Point FFT)**
  - **32-bit, Floating-Point Mode**
    - **69μs (1024-Point)**

# Real Time FFT considerations

- **Signal Bandwidth**
- **Sampling Frequency, fs**
- **Number of Points in FFT, N**
- **Frequency Resolution = fs/N**
- **Maximum Time to Calculate N-Point FFT = N/fs**
- **Fixed-Point vs. Floating Point DSP**
- **Radix-2 vs. Radix-4 Execution Time**
- **Windowing Requirements**

# Implementation DIT FFT in ADSP 2181 First Stage




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

$W_N = C + j(-S)$

$A = (C)\, x_1 - (-S)\, y_1$

$B = (C)\, y_1 + (-S)\, x_1$

$x_0' = x_0 + A \quad y_0' = y_0 + B$

$x_1' = x_0 - A \quad y_1' = y_0 - B$

```
        i0 = inplacereal;
        i1 = inplacereal + 1;
        i2 = inplaceimag;
        i3 = inplaceimag + 1;
        m2 = 2;

        cntr = nover2;                                  //cntr=N/2
        ax0 = dm(i0,m0);                                //ax0=x0
        ay0 = dm(i1,m0);                                //ay0=x1
        ay1 = dm(i3,m0);                                //ay1=y1

        do group_lp until ce;
              ar=ax0+ay0, ax1=dm(i2,m0);                //ar=x0+x1, ax1=y0
              sb=expadj ar, dm(i0,m2)=ar;               //Check for bit growth, x0'=x0+x1
              ar=ax0-ay0;                               //ar=x0-x1
              sb=expadj ar;                             //Check for bit growth
              dm(i1,m2)=ar, ar=ax1+ay1;                 //x1'=x0-x1, ar=y0+y1
              sb=expadj ar, dm(i2,m2)=ar;               //Check for bit growth, y0'=y0+y1
              ar=ax1-ay1, ax0=dm(i0,m0);                //ar=y0-y1, ax0= next x0
              sb=expadj ar, dm(i3,m2)=ar;               //Check for bit growth, y1'=y0-y1
              ay0=dm(i1,m0);                            //ay0= next x1
group_lp:     ay1=dm(i3,m0);                            //ay0= next y1

        call bfp_adj;
```

# Implementation DIT FFT in ADSP 2181 Butterfly Loop

```
        I4=twid_real;                                   //I4 --> C of W0
        I5=twid_imag;                                   //I5 --> (-S) of W0
        CNTR=DM(bflys_per_group);                       //CNTR = butterfly counter
        MY0=PM(I4,M4),MX0=DM(I1,M0);                    //MY0=C,MX0=x1
        MY1=PM(I5,M4),MX1=DM(I3,M0);                    //MY1=-S,MX1=y1

        DO bfly_loop UNTIL CE;
            MR=MX0*MY1(SS),AX0=DM(I0,M0);               //MR=x1(-S),AX0=x0
            MR=MR+MX1*MY0(RND),AX1=DM(I2,M0);           //MR=(y1(C)+x1(-S)),AX1=y0
            AY1=MR1,MR=MX0*MY0(SS);                     //AY1=y1(C)+x1(-S),MR=x1(C)
            MR=MR-MX1*MY1(RND);                         //MR=x1(C)-y1(-S)
            AY0=MR1,AR=AX1-AY1;                         //AY0=x1(C)-y1(-S),
                                                        //AR=y0-[y1(C)+x1(-S)]
            SB=EXPADJ AR,DM(I3,M1)=AR;                  //Check for bit growth,
                                                        //y1'=y0-[y1(C)+x1(-S)]
            AR=AX0-AY0,MX1=DM(I3,M0),MY1=PM(I5,M4);     //AR=x0-[x1(C)-y1(-S)],
                                                        //MX1=next y1,MY1=next (-S)
            SB=EXPADJ AR,DM(I1,M1)=AR;                  //Check for bit growth,
                                                        //x1'=x0-[x1(C)-y1(-S)]
            AR=AX0+AY0,MX0=DM(I1,M0),MY0=PM(I4,M4);     //AR=x0+[x1(C)-y1(-S)],
                                                        //MX0=next x1,MY0=next C
            SB=EXPADJ AR,DM(I0,M1)=AR;                  //Check for bit growth,
                                                        //x0'=x0+[x1(C)-y1(-S)]
            AR=AX1+AY1;                                 //AR=y0+[y1(C)+x1(-S)]
bfly_loop:  SB=EXPADJ AR,DM(I2,M1)=AR;                  //Check for bit growth,
                                                        //y0'=y0+[y1(C)+x1(-S)]
```

# Implementation DIT FFT in ADSP 2181
# Block Floating-Point Scaling Routine

```
bfp_adj:

        AY0=CNTR;                                   //{Check for last stage}
        AR=AY0-1;
        IF EQ RTS;                                  //{If last stage, return}
        AY0=-2;
        AX0=SB;
        AR=AX0-AY0;                                 //{Check for SB=-2}
        IF EQ RTS;                                  //{IF SB=-2, no bit growth, return}
        I0=inplacereal;                             //{I0=read pointer}
        I1=inplacereal;                             //{I1=write pointer}
        AY0=-1;
        MY0=0x4000;                                 //{Set MY0 to shift 1 bit right}
        AR=AX0-AY0,MX0=DM(I0,M1);                   //{Check if SB=-1; Get first sample}
        IF EQ JUMP strt_shift;                      //{If SB=-1, shift block data 1 bit}
        AY0=-2;                                     //{Set AY0 for block exponent update}
        MY0=0x2000;                                 //{Set MY0 to shift 2 bits right}

    strt_shift:

        CNTR=2047;                                  //{initialize loop counter}
        DO shift_loop UNTIL CE;                     //{Shift block of data}
              MR=MX0*MY0(RND),MX0=DM(I0,M1);        //{MR=shifted data,MX0=next value}
  shift_loop: DM(I1,M1)=MR1;                        //{Unshifted data=shifted data}

        MR=MX0*MY0(RND);                            //{Shift last data word}
        AY0=DM(blk_exponent);                       //{Update block exponent and}
        DM(I1,M1)=MR1,AR=AY0-AX0;                   //{store last shifted sample}
        DM(blk_exponent)=AR;
        sb = -2;
        RTS;

bfp_adj.end:
```

$A = (C)\, x_1 - (-S\,)y_1$

$B = (C)\, y_1 + (-S)\, x_1$

$x_0' = x_0 + A$

$y_0' = y_0 + B$

$x_1' = x_0 - A$

$y_1' = y_0 - B$

$x_0' < 1 \;,\; y_0' < 1$

$|C_{max}| = 1 \;,\; |S_{max}| = 1$

$x_0' = x_0 + x_1 + y_1 < 1$

$x_0 < 0.33 \;,\; x_1 < 0.33 \;,\; y_1 < 0.33$

$y_0' = y_0 + y_1 - x_1 < 1$

**0.33 = 0x2A3D**

**0.25**

| $-2^{0}$ | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ | $2^{-5}$ | $2^{-6}$ | $2^{-7}$ | $2^{-8}$ | $2^{-9}$ | $2^{-10}$ | $2^{-11}$ | $2^{-12}$ | $2^{-13}$ | $2^{-14}$ | $2^{-15}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | X | X | X | X | X | X | X | X | X | X | X | X |

# Implementation DIT FFT in ADSP 2181
# Scramble Routine

```
scramble:

            I4=inputreal;                   //{I4-->sequentially ordered data}
            I0=inplacereal;                 //{I0-->scrambled data}
            M4=1;
            M0=mod_value;                   //{M0=modifier for reversing N bits}
            L4=0;
            L0=0;
            cntr = N;
            ENA BIT_REV;                    //{Enable bit-reversed outputs on DAG1}
            DO brev UNTIL CE;
                  AY1=DM(I4,M4);            //{Read sequentially ordered data}
brev:             DM(I0,M0)=AY1;            //{Write data in bit-reversed location}
            DIS BIT_REV;                    //{Disable bit-reverse}
            RTS;                            //{Return to calling program}
scramble.end:
```