



Micromega Corporation

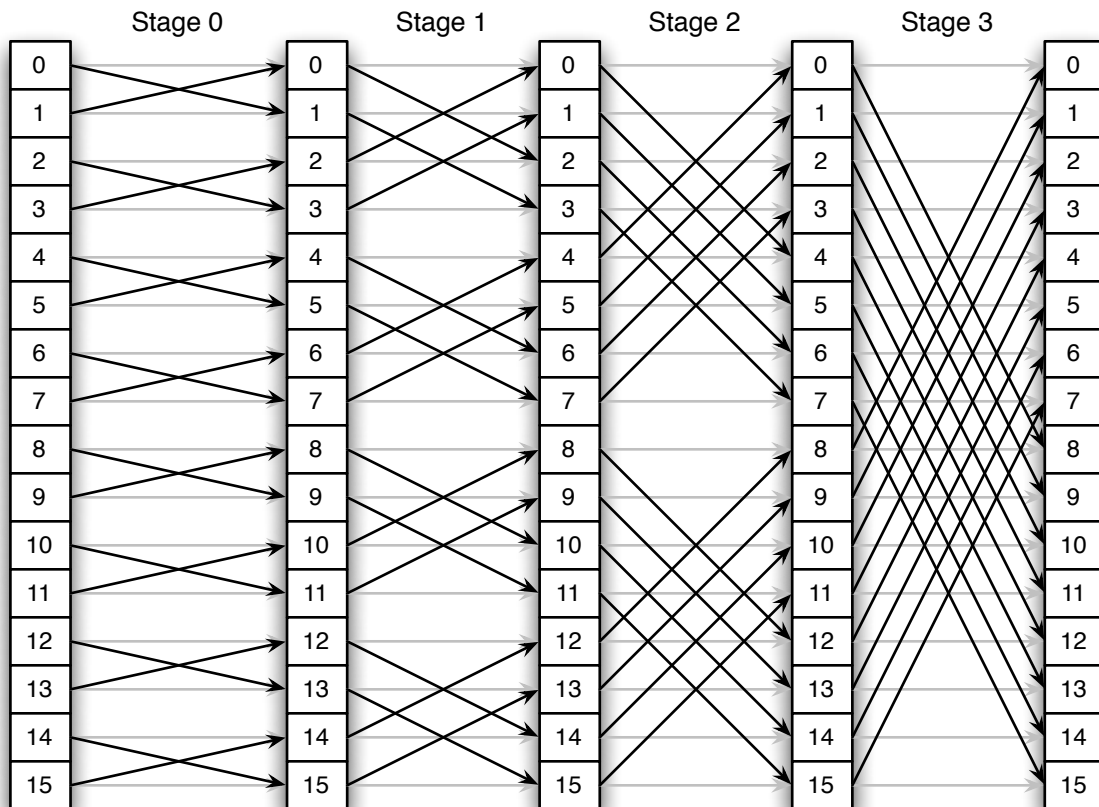
## Application Note 35

# Fast Fourier Transforms Using the FFT Instruction

The Fast Fourier Transform (FFT) is used in a variety of digital signal processing applications. This application note describes how to use the uM-FPU V3 floating point coprocessor's FFT instruction to calculate Fast Fourier Transforms. The FFT instruction provides support for both FFT and inverse FFT calculations. The mathematics and theory of FFTs is not discussed in this application note. The reader should be familiar with the general concepts and terminology associated with FFTs.

### Introduction

There are many different ways to calculate FFTs. The algorithm used by the FFT instruction first sorts the input data points in bit-reverse order, then performs a series of butterfly calculations. This is a decimation-in-time technique and is the method used by the Cooley-Tukey FFT algorithm. The following diagram shows the steps for calculating a FFT with 16 data points, after the bit-reverse sort has been done.



## Bit-Reverse Sort

The decimation-in-time approach requires the input data points to be sorted in bit-reverse order. The bit-reverse sort makes it easier to implement the butterfly calculations, and allows results to be stored in place, so additional storage space is not required. For data points stored in a matrix indexed from 0 to  $N$ , a bit-reverse sort is done by taking the binary value of the index, reversing the order of the bits, and exchanging the values at the initial index and the bit-reversed index. The following table shows an example of the initial index and bit-reverse index when  $N$  equals 8.

Initial Index	Bit-Reversed Index
000	000
001	100
010	010
011	110
100	001
101	101
110	011
111	111

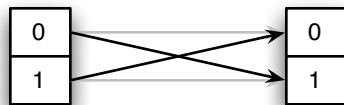
Sample C code for a bit-reverse sort is shown below. `FFT_SIZE` specifies the number of data points in the FFT. The real part of a data point is stored in array `xr` and the imaginary part is stored in array `xi`.

```
// bit reverse sort
j = FFT_SIZE / 2;
for (i = 1; i < FFT_SIZE-1; i++) {
    if (i < j) {
        tr = xr[j];
        ti = xi[j];
        xr[j] = xr[i];
        xi[j] = xi[i];
        xr[i] = tr;
        xi[i] = ti;
    }

    k = FFT_SIZE / 2;
    while (j >= k) {
        j = j - k;
        k = k / 2;
    }
    j = j + k;
}
```

## Butterfly Calculation

The butterfly calculation is the most basic calculation performed by the FFT and its basic form is shown in the following diagram:



Each butterfly calculation performs the follows:

```
temporary = real(0) + real(1)
real(1) = real(0) - real(1)
real(0) = temporary
temporary = imaginary(0) + imaginary(1)
imaginary(1) = imaginary(0) - imaginary(1)
imaginary(0) = temporary
```

There's a trigonometric component to the butterfly calculation (sine and cosine terms) that's not shown above. These are handled internally by the FFT instruction.

## Using the uM-FPU V3 FFT Instruction

The data points for the FFT instruction are stored in the registers associated with matrix A on the uM-FPU V3 chip. The matrix size is  $N$  rows by 2 columns, where  $N$  must be a power of two. The data points are specified as complex numbers, with the real part stored in the first column and the imaginary part stored in the second column. If all data points fit in the matrix, the FFT can be calculated with a single FFT instruction. If there are more data points than will fit in the matrix, the calculation must be done in a series of blocks using an algorithm that is described below. The uM-FPU V3 chip has 128 registers, and the number of data points must be a power of two, so the number of rows in the matrix must be 2, 4, 8, 16, 32, or 64. The size of the matrix, and the registers to use, are specified using the SELECTMA instruction.

The FFT is a complex calculation requiring  $N \log_2 N$  operations, and all data points must be available during the calculation. For microcontroller applications, the available data storage and the execution time will both be limiting factors in determining the size of the data set. The execution time is faster if more registers are allocated to the FFT calculation.

## Calculating the FFT for a Small Number of Data Points

If all data points can fit in matrix A, the FFT can be calculated with a single FFT instruction. This is particularly useful for microcontrollers that have limited data space available. Sample data can be transferred to the uM-FPU V3 chip (or captured directly using the on-chip uM-FPU A/D channels), transformed with the FFT instruction, and the results stored in the uM-FPU registers. The FFT instruction has an efficient bit-reverse sort option that can be used for single instruction FFT calculations. The inverse FFT can also be calculated with a single instruction. The single instructions are as follows:

### Single instruction FFT

```
FFT, 0x04    first stage
              pre-processing bit-reverse sort
```

### Single instruction inverse FFT

```
FFT, 0x1C    first stage
              pre-processing bit-reverse sort
              pre-processing for inverse FFT
              post-processing for inverse FFT
```

## Inverse FFT

The pre-processing for inverse FFT option changes the sign of the imaginary part prior to the butterfly calculations. The post-processing for inverse FFT option divides the real and imaginary parts of the result by the size of the FFT and changes the sign of the imaginary part. When working with a large number of data points, these operations can be performed at the first and last stage of a multistage calculation.

## Calculating the FFT for a Large Number of Data Points

When there are more data points than will fit in the matrix, the calculation must be done in a series of steps. The data must first be bit-reverse sorted, then processed block by block using the FFT instruction. The maximum FFT size is 32768 data points, provided there is sufficient data storage available on the microcontroller for all data points. Execution time will be a consideration when using a large number of data points since  $N \log_2 N$  operations are required.

The order of processing must proceed in a hierarchical manner, stage by stage, level by level, and block by block. The order of processing is important in order to ensure the internal trigonometric calculations are correct. Sample C code to implement a multistage FFT is shown below. A bit-reverse sort (as shown above) would be done before this code is executed.

```
nLevels = 1;
blockSize = BLOCK_SIZE;

// repeat for each stage of FFT calculation
for (stage = 0; stage < FFT_SIZE_LOG2; ) {

    // determine maximum block size
    while ((nLevels * blockSize) > FFT_SIZE) {
        blockSize = blockSize / 2;
    }

    // select matrix A
    fpu_Write4(SELECTMA, DATA_ARRAY, blockSize, 2);

    // repeat for each level of the current stage
    for (level = 0; level < nLevels; level++) {

        // repeat for each block of the current level
        for (n = 0; n < FFT_SIZE; n = n + (blockSize * nLevels)) {

            // write block of data
            fpu_Write2(Fpu.SELECTX, DATA_ARRAY);
            index = level + n;
            for (i = 0; i < blockSize; i++) {
                fpu_write(FWRITEX);
                fpu_writeFloat(xr[index]);
                fpu_write(FWRITEX);
                fpu_writeFloat(xi[index]);
                index = index + nLevels;
            } // next i

            // perform FFT calculation on block of data
            if (stage == 0)
                type = 0;
            else if (level == 0 && n == 0)
                type = 1;
            else if (n == 0)
                type = 2;
            else
                type = 3;
            fpu_write2(FFT, type);
        }
    }
}
```

```

// read block of data
fpu_write2(SELECTX, DATA_ARRAY);
index = level + n;
for (i = 0; i < blockSize; i++) {
    fpu_write(FREADX);
    fpu_readDelay();
    xr[index] = fpu_readFloat();
    fpu_write(FREADX);
    fpu_readDelay();
    xi[index] = fpu_readFloat();
    index = index + nLevels;
}

} // next block

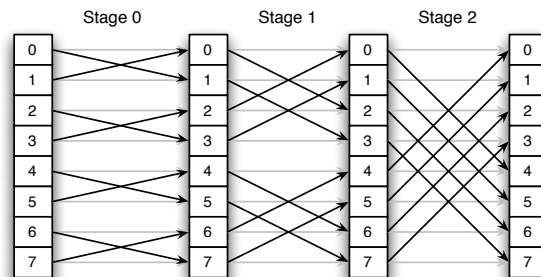
} // next level

if (stage == 0) {
    stage = stage + BLOCK_SIZE_LOG2;
    nLevels = BLOCK_SIZE;
}
else {
    stage = stage + 1;
    nLevels = nLevels * 2;
}
} // next stage

```

## Multistage FFT Example

To better understand the algorithm shown above for a multistage FFT, we'll walk through an example. This algorithm is typically used with a large number of data points and the largest matrix available. To simplify the example, an FFT with 8 data points and a 4x2 matrix (block size of 4) is used. The logic is the same for larger examples. The following diagram shows the complete FFT calculation for 8 data points after the initial bit-reverse sort has been done.



Refer to the code shown above as you follow this example.

### Initial Values

```

FFT_SIZE = 8
FFT_SIZE_LOG2 = 3
BLOCK_SIZE = 4
BLOCK_SIZE_LOG2 = 2
nLevels = 1
blockSize = 4

```

### Step 1

The `FFT, 0` instruction (first stage) always calculates as many stages of the FFT as possible. The number of stages calculated is equal to  $\log_2$  of the block size (i.e.  $\log_2$  of the number of rows in the matrix). In this example, the block size is four, so two stages are calculated.

next stage:

`stage = 0`

`SELECTMA` selects a 4x2 matrix

next level:

`level = 0`

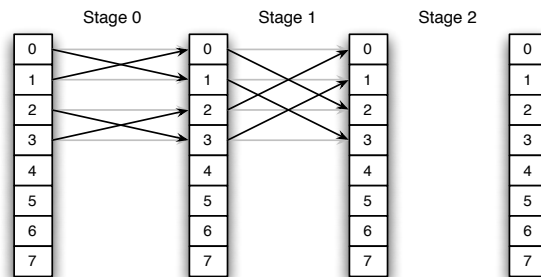
next block:

`n = 0`

write `xr[0]`, `xi[0]`, `xr[1]`, `xi[1]`, `xr[2]`, `xi[2]`, `xr[3]`, `xi[3]` to matrix A

execute `FFT, 0` instruction

read matrix A and store `xr[0]`, `xi[0]`, `xr[1]`, `xi[1]`, `xr[2]`, `xi[2]`, `xr[3]`, `xi[3]`



### Step 2

This step completes the second part of stage 0.

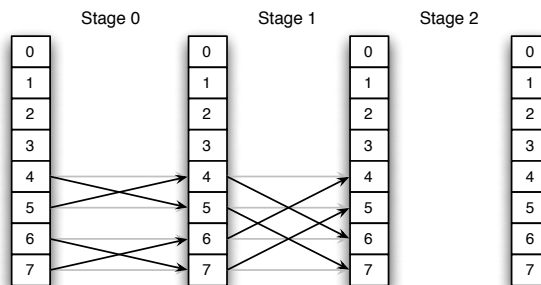
next block:

`n = 4`

write `xr[4]`, `xi[4]`, `xr[5]`, `xi[5]`, `xr[6]`, `xi[6]`, `xr[7]`, `xi[7]` to matrix A

execute `FFT, 0` instruction

read matrix A and store `xr[4]`, `xi[4]`, `xr[5]`, `xi[5]`, `xr[6]`, `xi[6]`, `xr[7]`, `xi[7]`



### Step 3

The FFT, 1 instruction (next stage) start stage 2.

next stage:

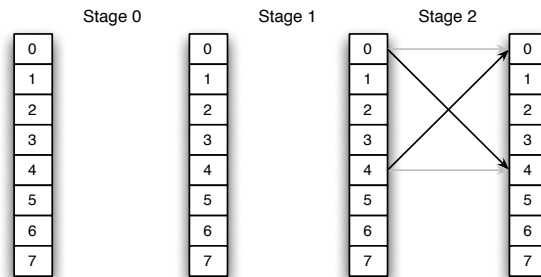
```
stage = 2
nLevels = 4
blockSize = 2
SELECTMA selects a 2x2 matrix
```

next level:

```
level = 0
```

next block:

```
n = 0
write xr[0], xi[0], xr[4], xi[4] to matrix A
execute FFT, 1 instruction
read matrix A and store xr[0], xi[0], xr[4], xi[4]
```



### Step 4

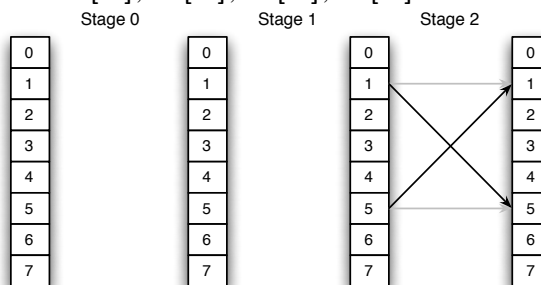
The FFT, 2 instruction (next level) starts the next level of stage 2.

next level:

```
level = 1
```

next block:

```
n = 0
write xr[1], xi[1], xr[5], xi[5] to matrix A
execute FFT, 2 instruction
read matrix A and store xr[1], xi[1], xr[5], xi[5]
```



### Step 5

The FFT, 2 instruction (next level) starts the next level of stage 2. This example only requires one block per level, but large FFTs can have several blocks at the same level.

next level:

level = 2

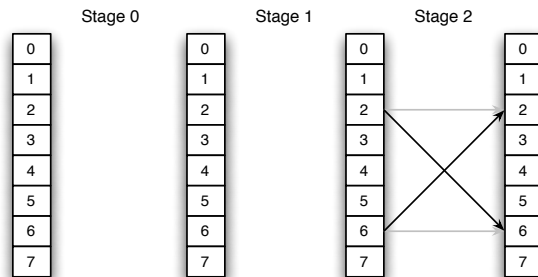
next block:

n = 0

write xr[2], xi[2], xr[6], xi[6] to matrix A

execute FFT, 2 instruction

read matrix A and store xr[2], xi[2], xr[6], xi[6]



### Step 6

The FFT, 2 instruction (next level) starts the last level of stage 2, and the FFT is complete.

next level:

level = 3

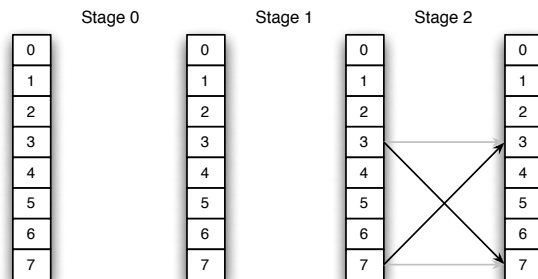
next block:

n = 0

write xr[3], xi[3], xr[7], xi[7] to matrix A

execute FFT, 2 instruction

read matrix A and store xr[1], xi[1], xr[5], xi[5]



### Further Information

See the Micromega website (<http://www.micromegacorp.com>) for additional information regarding the uM-FPU V3 floating point coprocessor, including:

*uM-FPU V3 Datasheet*

*uM-FPU V3 Instruction Set*