DSP101 Part 6 Implementing Fast Convolution

Introduction

The job is to do the Discrete Fourier Transform and its inverse. Also we need to do it fast and as efficiently as possible otherwise the operational upper frequency will be rather low for radio work! So, we must employ every trick in the book and for program description purposes we need a programming language that covers all the points. For this reason I shall use a special assembly code - you are welcome try to implement the algorithms in real world code, either high level or low level languages could be used but remember the speed imperative. Let's start on the tricks that are in the book! Doing it with the tricks make it a FAST Fourier Transform.

Radix-2 Decimation in Time (DIT)

Sounds bad, but actually is relatively simple. Direct calculation of all N frequencies in a given DFT needs N^2 complex multiplies and N^2 -N complex multiplies. Decimation in Time just means breaking down the series of calculations into parts. Some rather clever people realised that if you did this then you could re-use some of the calculations. Ultimately you can break down the whole thing into two point DFTs and then by re-ordering the data and sort of cross combining the outputs you can cut the computational cost by a factor of 2. Specifically, it then needs $N^2/2$ +N complex multiplies and N^2 complex adds. Well worth having! Two point FFTs are called 'butterflies' because of the shape of the flow graph that describes them. Larger data set sizes can be broken down in stages until they reach the 'butterfly' level. The whole set can first be broken down into 2 smaller sets, then each of those can be broken again into 2 smaller sets and so on until you are down to the 'butterfly' when you are dealing with two complex sample inputs resulting in two transformed complex sample outputs. To complete the overall calculation you have to have a combining stage followed by more butterflies, then combination, then more and so on. Note that the re-ordering spoken of means splitting the data set into odd and even halves a will be shown. Also, depending on which detailed algorithm is used either the input set or the output set will be in reverse order!

N-point FFT Butterflies and DIT combinations

The 2 point butterfly flow graph is shown in Fig1, whilst Fig2 shows the real goings on
Go to the next page...........

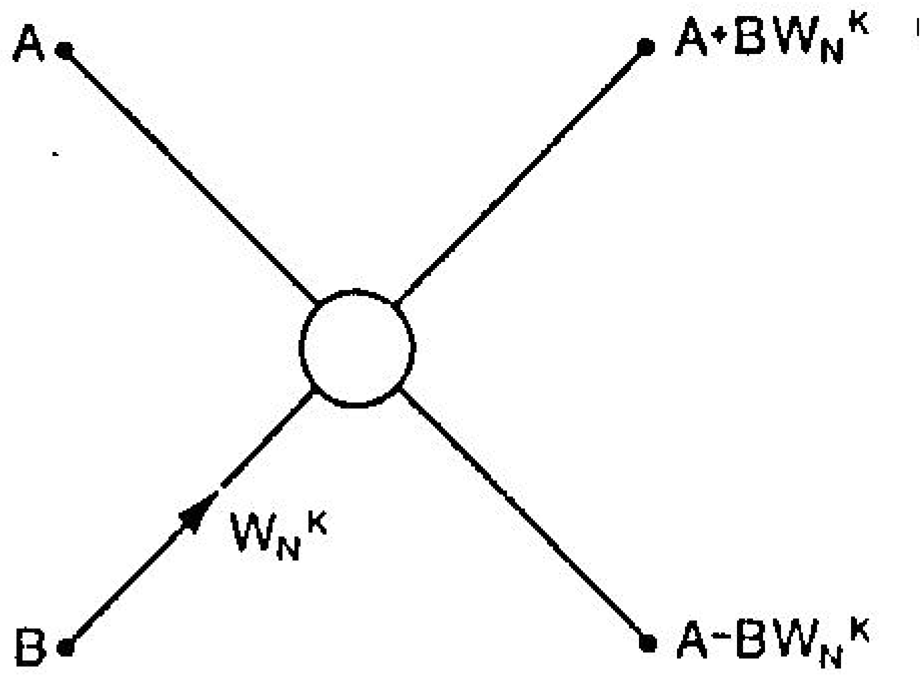

Fig 1. A Butterfly (Zoran)

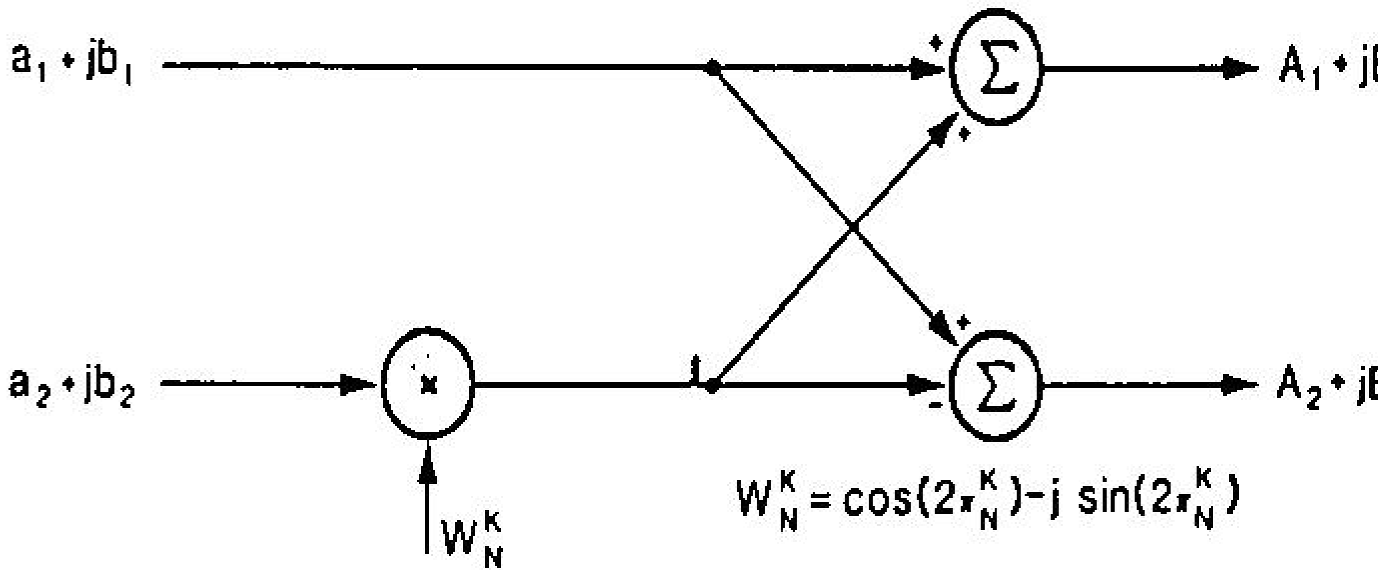

Figure 2 Operations of the 2 point butterfly (Zoran)

The DIT combinations are illustrated in Fig 3 below for N=8. You will see that there are three lots of combinations. This is called a 3 pass algorithm because each of the stages is done before the next, so that the processor, in this case, executes 3 passes on the data. Note the ordering of input data, first 4 are even samples, next 4 are odd. The output is normally ordered. If you are interested

there are two options here, the Cooley-Tukey and the Sande-Tukey. In the latter factorisation the output is a bit-reversed re-ordering, whist in the former the output suffers bit reversal. Looking at Fig 3 shows that the so called first pass separation is 1 (adjacent inputs of the already re-ordered data) and the separation between inputs on the last pass is 4. These are important parameters and must be specified in the algorithm. They are usually called the FPS and LPS (first and last pass separation of input samples to the butterflies)

Figure 3 FFT decimation in time (Strathclyde Univ.)

The three passes are characterised by the three sets of 'crossovers' going from left (input) to right (output).

An appropriate assembly code program

This is meant to illustrate the necessary steps only. The instruction set is quite high level and is derived from that of a specialised DSP processor. If you are programming a more mundane device it would be necessary to decompose some instructions, necessitating a detailed knowledge. Here I will

assume a 16 point operation. Also the algorithm is Sande-Tukey.

```
Begin
/* Load the 16 data points into internal RAM EVEN points first*/
LD NMPT:8,MBS:8,MSS:2,MBA:SIGNAL
/* Store consecutively /*
ST NMPT:8,MBS:8,MSS:8, MBA:REORDER
```

In the above two lines, LD means load, NBPT, is number of points, MBS is Memory block size,MSS is Memory step size and SIGNAL is at the MBA (Memory block address) where is was left by the A/D converter. These lines are picking up the signal samples starting with sample 2 and then getting the even numbered samples(MSS:2). The LD instruction is loading these samples into the CPU internal RAM.
The ST instruction (store), stores the points consecutively e.g. Samples 0,2,4,6,8,10.12.14. The internal memory that they are held in is called REORDER because that is what is going on!

```
/*Read in the odd sample points and reverse order them */
LD NMPT:8,MBS:8;MSS:2,MBA:SIGNAL+1
```

Load the other half beginning with sample 1, e.g. 1,3,5,7,9,11,13,15. RV=1 means reverse their order.

```
STB NMPT:8,MBS:8,MSS:8, MBAB:REORDER+8
```

So, now we should have loaded the data as samples 0,2,4,6,8,10,12,14,15,13,11,9,7,5,3,1. Further detailed re-ordering might be necessary depending on the exact algorithm.

```
/* Load the re-ordered data back into the processor */

LD NMPT:16,MBS:16,MSS:16,MBA:REORDER

/* Do the FFT DIT in 4 passes */

FFT NMBT:8, R:1,FPS:1,LPS:8,I:0,RS:1
```

Here, NMBT is the number of butterflies per pass. R:1 specifies that the data is in reversed order, I=0 specifies the FORWARD FFT. If I=1 then the inverse fft is executed. So the same instruction applies, with slightly different field parameters. RS:1 means use CPU RAM section 1.

It only remains to output the results from the processor to the output memory block.

```
/* Output to memory for use by other routines/methods etc */

ST NMPT:16,MBS:16,MSS:16,RS:0,MBA:RESULT
```

RS:0 means use processor RAM section 0. There is an assumption here that data to be operated on is stored in processor RAM section 1, whilst the result should be dumped in processor RAM section 0. What is in processor RAM section 0 needs to be stored in external RAM at memory base address 'RESULT' for use by other routines.

Conclusion

The assembly code used here is an adaption of the Zoran Vector Signal Processor assembler. This is a very high performance CPU with multiple ALUs capable of doing parallel butterflies. This article is intended to give an flavour of what it takes to do fast convolution. Remember from Article 4 that two FFTs, a multiply and and one IFFT is necessary for fast convolution. So, the software implementation is non-trivial, especially when dealing with much larger arrays than used here as examples. Clearly, a processor should be chosen that allows a higher level language to be used because anyone would be daunted by trying to define the individual assembly language steps on an 8 bit micro-controller! A lot of radio based simpler functions are achieved in hardware via the used of electrically programmed gate arrays, but Amateur TV is now moving into digital modes and MPEG2 demands high rate DCTs (Discrete Cosine Transforms - very similar in structure to FFTs) so ATV adherents will, if they want to build their own boxes, have to get to grips with all this. It makes the home TV digibox look like good value doesn't it? And it is!
I think the objective of this series of articles has been achieved i.e. To give a feel for the magnitude of DSP tasks as background to devising simple DSP projects for amateurs to attempt using cheap and accessible processors. My own view is that simple audio frequency filters might be possible and also EPLD devices could be employed for some simple rf tasks but the where-with-all is just not accessible or cheap for ambitious rf applications through the software route.

I hope you enjoyed these introductions to a fascinating area of interest.

This copyright of this article is owned by G3XSD and is assigned to the Itchen Valley Amateur Radio Club. Permission is needed for copying or any other re-use apart from individual study.