MASTER

Real-time implementation of a low delay acoustic echo canceller on one TMS320C30 DSP

Alexiu, G.M.

Award date:
1995

Link to publication
Real-time implementation of a low delay acoustic echo canceller on one TMS320C30 DSP

Master's thesis of G. M. Alexiu

January 1995

Supervisor: Prof. Dr. Ir. W. M. G. van Bokhoven

Coaches: Dr. Ir. P. C. W. Sommen
Ir. G. P. M. Egelmeers

Period: project: December 93 - January 94, March 94 - July 94
        report: August 94 - September 94, December 94 - January 95

At: Eindhoven University of Technology
    Department of Electrical Engineering
    Electronic Circuit Design Group (EEB)
    Digital Signal Processing
Abstract

The area of the adaptive filters is one of the topics approached by the people from the Electronic Circuit Design Group who are doing research in Digital Signal Processing. The acoustic echo cancelling is an application field where the adaptive filters are the suitable solution.

The present report describes an implementation of the Decoupled Partitioned Block Frequency Domain Adaptive Filter algorithm [6] on one DSP chip.

This final degree-project is a development stage between the theoretical research, validated by simulation, then followed by the setup of some basic routines, and the final tests stage, meant to feedback into theory, in order to get the application ready to be used.

The implementation is realized in two main parts, the filter- and the update-part. They work with each other by means of the interrupt service routine, which controls the general program and data flow, including the necessary input/output. The input signal is convolved with the adaptive coefficients by the filter-part routine. To reach the necessary performance, the Direct Memory Access controller copies into the on-chip RAM the next partition to be processed, concurrently with the CPU computing over the current one.

The update-part routine estimates the gradient which modifies the adaptive coefficients and performs the update of the coefficients. This gradient results from the correlation between the input signal and the residual signal normalized with an inverse power estimate of the input signal.

The most relevant operations take place in the frequency domain. For the correspondence between the time and the frequency domain, Fast Fourier Transform routines were taken over from the former development stage.

The algorithm implementation is coded in assembly language. The code accepts any (valid) set of parameters to configure the adaptive filter.

As a practical example was chosen an adaptive filter with 2016 coefficients, processing signals sampled at 8kHz.

The filter-part routine works over blocks of 8 samples, while the update-part routine computes over blocks of 504.

For this case, the total answer delay is 1.750msec, and the cancelled echo is 250msec long.
Contents

1 Introduction ........................................................................................................... 1

1.1 Acoustic echo canceller ................................................................................... 1

1.2 Theoretical aspects .......................................................................................... 4

  1.2.1 Block adaptive filter algorithms ............................................................... 4
      1.2.1.1 Block Normalized Least Mean Square Algorithm ............................ 4
      1.2.1.2 Block Orthogonal Projection Method .............................................. 5

  1.2.2 Efficient implementations of block adaptive filters ................................. 7
      1.2.2.1 Overlap-save method for fixed filters ............................................. 7
      1.2.2.2 Efficient implementation of BNLMS ............................................. 10
      1.2.2.3 Efficient implementation of BOP .................................................. 11

  1.2.3 Decoupled Partitioned Block Frequency Domain Adaptive Filter .......... 13
      1.2.3.1 Partition of the filter-part .............................................................. 13
      1.2.3.2 Partition of the update-part ......................................................... 15
      1.2.3.3 Coupling of the partitioned filter- and update-part ....................... 16

  1.2.4 Simulation results and conclusions .......................................................... 18

1.3 The hardware used ......................................................................................... 20

2 The implementation ............................................................................................. 23

  2.1 General ideas behind the implementation .................................................... 23
  2.2 The filter-part routine .................................................................................... 27
  2.3 The update-part routine .................................................................................. 34
  2.4 The interrupt service routine .......................................................................... 41
  2.5 The main program .......................................................................................... 45

3 Conclusions and recommendations ..................................................................... 48
5 Appendixes .......................................................................................................................... 52

5.1 Other remarks; relevant parts of code ............................................................................. 52
5.2 Processor load and memory allocation example ............................................................... 65
5.3 Running with any valid parameter set ............................................................................... 68
  5.3.a The parameter set acceptance ...................................................................................... 68
  5.3.b The implementation code variables calculation ........................................................... 69
  5.3.c The buffers allocation within the available memory map ............................................. 70
5.4 Software development environment .................................................................................. 71

List of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.1</td>
<td>Acoustic echo canceller</td>
<td>1</td>
</tr>
<tr>
<td>1.1.2</td>
<td>Generic form of adaptive filter</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3</td>
<td>e for a single adaptive weight w</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Geometric interpretation of BOP algorithm</td>
<td>7</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Overlap-save method for fixed filters</td>
<td>8</td>
</tr>
<tr>
<td>1.2.3</td>
<td>Overlap-save implementation of BNLMS</td>
<td>10</td>
</tr>
<tr>
<td>1.2.4</td>
<td>BFDAF implementation</td>
<td>12</td>
</tr>
<tr>
<td>1.2.5</td>
<td>Partitioned filter-part</td>
<td>14</td>
</tr>
<tr>
<td>1.2.6</td>
<td>Partitioned update-part</td>
<td>15</td>
</tr>
<tr>
<td>1.2.7</td>
<td>Interface for the adaptive weights</td>
<td>17</td>
</tr>
<tr>
<td>1.2.8</td>
<td>Interface for the residual signal with ( A = \lambda \cdot B )</td>
<td>17</td>
</tr>
<tr>
<td>1.2.9</td>
<td>Complexity of different algorithms function of maximum delay</td>
<td>19</td>
</tr>
<tr>
<td>1.3.1</td>
<td>A simplified representation of the C30 CPU architecture</td>
<td>20</td>
</tr>
<tr>
<td>1.3.2</td>
<td>The total PC system board available memory map</td>
<td>22</td>
</tr>
<tr>
<td>2.1.1</td>
<td>The interrupts perform the general program synchronization</td>
<td>24</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Vectors elements disposal in the buffers</td>
<td>26</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Partitioned filter</td>
<td>28</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Correct (delayed) subtraction of the echo estimate from the near-end side signal</td>
<td>32</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Partitioned update</td>
<td>35</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Coupling the filter and the update - the adaptive coefficients</td>
<td>38</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Input/Output during the interrupt service routine</td>
<td>42</td>
</tr>
<tr>
<td>2.4.2</td>
<td>The accessing of the residual signal buffer by the filter routine and the interrupt service routine</td>
<td>43</td>
</tr>
<tr>
<td>2.5.1</td>
<td>The processing and the total delay definition</td>
<td>46</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Small and big frame splitting timings</td>
<td>66</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Acoustic echo canceller

Audio conferencing and hands-free telephony systems suffer from the feedback of the sound coming from the loudspeaker into the microphone. Besides the unpleasantness of hearing this echo, when it occurs at both ends of the system, a ringing effect can start (oscillations). To overcome this problem, the first solutions were based on frequency division multiplexing and time division multiplexing, having the disadvantage that they degrade the signal quality and use inefficiently the communication channel.

The solution adopted in our days is the adaptive echo canceller. That is an adaptive filter which models the room impulse response (the vector $W$ on figure 1.1.1), generating an estimate of the echo $\hat{e}$, to be subtracted from the microphone output $\tilde{e}$.

![Diagram of an acoustic echo canceller](image)

Figure 1.1.1: Acoustic echo canceller

The model must track the room impulse response. There is no unique solution to this. A variety of recursive algorithms is available, each one with its desirable features.
For this application it is assumed, based on the Wiener filter theory, that the unknown system function (represented by the vector \( \mathbf{w}_u \) in figure 1.1.2) can be modelled by a Finite Impulse Response (FIR) filter \( \mathbf{w} \).

One problem is the large impulse response length, typically 250msec (this means 2000 samples, for 8kHz sampling frequency): the filter will have a large number of coefficients. Other problems are the time-varying character of the echo path (this leaded in fact to the use of an adaptive filter), and the time-varying input signal statistical properties (thus an update algorithm insensitive to the input signal statistics has to be used).

The adaptation relies on a priori information - for instance the impulse response length; without it, the problem makes no sense.

For the moment being, the updating of the filter coefficients is inhibited when near-end talk is detected, because the adaptation algorithm has to have as inputs only the far-end input signal and its echo. Techniques to separate \( s \) out of \( r \) (figure 1.1.2) are being investigated.

For the considered FIR filter with stationary inputs, the mean-squared error \( \varepsilon = \mathbb{E}\{(e-\hat{e})^2\} \) is a quadratic function of the adaptive weight vector \( \mathbf{w} \). The paraboloid has an unique bottom at \( \varepsilon_{\text{min}} \), corresponding to the Wiener solution weight vector \( \mathbf{w}_{\text{opt}} = \mathbf{w}_u \). It can be shown [4] that the gradient vector \( \nabla \mathbf{w} = \partial \varepsilon / \partial \mathbf{w} \) always points away from this bottom, as depicted for a single adaptive weight in figure 1.1.3.

Using an update rule that makes every step in the negative direction of the gradient vector (steepest-descent update):

\[
\mathbf{w} = \mathbf{w} - \alpha \cdot \nabla \varepsilon,
\]

where \( \alpha > 0 \) being the adaptation constant,

the coefficient vector \( \mathbf{w} \) will finally reach, in average, the unknown optimal FIR Wiener solution \( \mathbf{w}_u \).

It can be shown [4] that the gradient vector \( \nabla \) is proportional with the negative value of the crosscorrelation between the input signal vector \( \mathbf{x} \) and the residual signal \( r \).
Thus the adopted solution involves two main operations: the calculation of the output signal \( \varepsilon \), that is a convolution of the adaptive weight vector \( \mathbf{w} \) with the input signal vector \( \mathbf{x} \), and the estimation of the gradient vector \( \mathbf{\nabla} \), for which is needed a crosscorrelation between the input signal vector \( \mathbf{x} \) and the residual signal \( r \).

From [3] it is known that, for large lengths (here in the range of 2000), the convolution and the correlation can be efficiently calculated on block basis (this however introduces a delay), in the frequency domain.

As starting point will be considered, in the next section, the Block Normalized Least Mean Square (BNLMS) algorithm. Being a gradient-based method, its convergence properties depend on the statistical properties of the input signal [4].

By using more past information of the input signal (a normalization with an estimate of the inverse autocorrelation matrix takes place), the dependance can be reduced. This is shown in a geometrical approach, further leading to the Block Orthogonal Projection (BOP) algorithm, and implemented as the Block Frequency Domain Adaptive Filter (BFDAF) [12]. The influence of the input signal statistics is reduced by normalizing each frequency bin by the corresponding power bin.

The BFDAF cannot solve the following contradiction: the block length has to be small, for a small processing delay, but yet large, for good convergence properties and small complexity. The number of multiplications to produce one output sample is a rough indication of the complexity.

A next step to decrease the processing delay is the Partitioned BFDAF (PBFDAF) [8-10], but this reduces also the decorrelation dimension.

The solution is to apply separate partitioning techniques to the filter- and update-part [6]. The result is the Decoupled PBFDAF (DPBFDAF), which realizes a minimal processing delay (when choosing the filter-part block length as small as possible) and good convergence properties (by choosing a large update-part block length).

The practical implementation complexity must fit the computational power of the chosen hardware.

Figure 1.1.3: \( \varepsilon \) for a single adaptive weight \( \mathbf{w} \)
1.2 Theoretical aspects

First is given a short review of two block adaptive algorithms that serve as starting point and then the deriving of the DPBFDAF algorithm.

1.2.1 Block adaptive filter algorithms

1.2.1.1 Block Normalized Least Mean Square Algorithm

With the block length $B$, the adaptive weight vector $w^N[kB]$ is updated every $B$ samples:

$$w^N[kB] = (w_{N-1}[kB], \ldots, w_1[kB], w_0[kB])'$ \[1.2.1\]

The sampling period is $T$, $k$ is the time index, and $(\cdot)'$ is the transpose of a vector.

Bold-underlined characters denote vectors while bold-not-underlined characters denote matrices, the dimensions being superscripted.

The $N \times B$ input signal matrix is defined as:

$$X^{N,B}[kB] = \begin{pmatrix}
(x^B[kB-N+1])' \\
. \\
. \\
(x^B[kB-1])'
\end{pmatrix}$$ \[1.2.2\]

with for $i = 0,1,\ldots,N-1$:

$$x^B[kB-i] = (x[kB-i-B+1], \ldots, x[kB-i])' \[1.2.3\]$$

The residual signal vector is defined as:

$$r^B[kB] = (r[kB-B+1], \ldots, r[kB])' \[1.2.4\]$$

It can be shown [8] that the gradient vector can be expressed as follows:

$$\nabla^N[kB] = -2 \epsilon \{ X^{N,B}[kB] r^B[kB] \} \[1.2.5\]$$

in which $\epsilon$ denotes the mathematical expectation.

This exact crosscorrelation cannot be used because it needs an infinite amount of time to do the averaging.

The Block Least Mean Square (BLMS) approach makes a rough estimate of the gradient by using only the present values of $X^{NB}[kB]$ and $r^B[kB]$. The adaptive filter will do the averaging itself. The update rule is given by:
\[ w^N[(k+1)B] = w^N[kB] + 2\alpha X^N.B[kB] x^B[kB]. \]  

Furthermore it follows from figure 1.1.2 that the residual signal samples in the vector \( x^B[kB] \) are composed of delayed input signal samples. This implies that, in average, the behaviour of the adaptive weights will depend on the quantity \( \epsilon \{(x[k])^2\} = \sigma_x^2 \). This dependency can be removed by normalizing the adaptation constant \( \alpha \) with \( \sigma_x^2 \), which results in the Block Normalized Least Mean Square (BNLMS) algorithm:

\[ w^N[(k+1)B] = w^N[kB] + \frac{2\alpha}{\sigma_x^2} X^N.B[kB] x^B[kB]. \]

The adaptive weights are fixed during \( B \) samples-time. The output of the adaptive filter can also be expressed on block basis. Every \( B \) samples, a \( B \)-dimensional vector

\[ \hat{a}^B[kB] = (e[kB-B+1], \ldots, e[kB])' \]

is produced as follows:

\[ \hat{a}^B[kB] = (X^N,B[kB])' w^N[kB]. \]

A possible recursive estimate for the variance \( \sigma_x^2 \) of the input signal is given by:

\[ \hat{\sigma}_x^2[(k+1)B] = (1-\beta) \cdot \hat{\sigma}_x^2[kB] + \frac{\beta}{B} (X^N,B[kB])' X^N,B[kB], \quad 0 < \beta < 1. \]

The inverse of this estimate is needed. An approximation without divisions is required (see also the note at the end of section 1.2.4). For small \( \beta \), this results in:

\[ (\hat{\sigma}_x^2[(k+1)B])^{-1} = (1+\beta)(\hat{\sigma}_x^2[kB])^{-1} - \frac{\beta}{B} ((\hat{\sigma}_x^2[kB])^{-1})^2 (X^N,B[kB])' X^N,B[kB]. \]

### 1.2.1.2 Block Orthogonal Projection Method

Here is given a technique that can decorrelate an input signal of an adaptive filter with \( N \) coefficients, by using a \( B \times B \) autocorrelation matrix.

A \( B \)-step Orthogonal Projection method makes a projection on a \( B \)-dimensional hyperplane \[13\], idea that can be implemented in two variants:

- the sliding procedure, that uses every iteration only one new input signal sample - this is the \( B \)-step Orthogonal Projection method (OP) \[13\], and
- the block approach, that uses \( B \) new samples, performing only one update of the adaptive weight vector every \( B \) samples - this is the Block Orthogonal Projection method (BOP) \[13\], and will be of further concern.

A geometrical interpretation is used in order to derive the BOP algorithm.

The inner product of two vectors \( \mathbf{a} \) and \( \mathbf{b} \) is defined as follows:

\[ \langle \mathbf{a}, \mathbf{b} \rangle = \mathbf{a}' \mathbf{b} = \mathbf{b}' \mathbf{a}. \]

The difference vector \( d^N[kB] \) is defined as the difference between the adaptive weight vector \( w^N[kB] \) and the unknown system weight vector \( w^N_0 \):

\[ d^N[kB] = w^N_0 - w^N[kB]. \]
First, a projection of \( \mathbf{d}^{N}[kB] \) is made on a B-dimensional plane, spanned by B N-dimensional vectors \( \mathbf{x}^{N}[kB-B+1],...\mathbf{x}^{N}[kB] \) (figure 1.2.1). The difference vector \( \mathbf{d}^{N}[kB] \) is written as a sum of two components, one perpendicular an the other one parallel to the plane:

\[
\mathbf{d}^{N}[kB] = \mathbf{d}_{\perp}^{N}[kB] + \mathbf{d}_{\parallel}^{N}[kB],
\]

with for i=0,1,...,B-1:

\[
< \mathbf{d}_{\perp}^{N}[kB], \mathbf{x}^{N}[kB-i] > = ( \mathbf{d}_{\perp}^{N}[kB] )^{t} \mathbf{x}^{N}[kB-i] = 0,
\]

and

\[
\mathbf{d}_{\perp}^{N}[kB] = \sum_{i=0}^{B-1} c_{i}[kB] \mathbf{x}^{N}[kB-i] = \mathbf{x}^{N}[kB] \mathbf{c}_{B}[kB],
\]

where the B-dimensional vector \( \mathbf{c}_{B}[kB] \) is defined as:

\[
\mathbf{c}_{B}[kB] = (c_{B-1}[kB],...,c_{0}[kB])^{t}.
\]

Without loss of generality, for simplicity, we assume \( s[k] = 0 \).

The B-dimensional residual signal vector \( \mathbf{r}^{B}[kB] \) can be written as:

\[
\mathbf{r}^{B}[kB] = (\mathbf{x}^{N,B}[kB])^{t} \mathbf{d}^{N}[kB].
\]

A procedure to calculate the coefficients of the vector \( \mathbf{c}_{B}[kB] \) is needed. This can be found by using the fact that \( \mathbf{d}_{\perp}^{N}[kB] \) is perpendicular to all vectors \( \mathbf{x}^{N}[kB-i] \), for \( i=0,...,B-1 \):

\[
\mathbf{r}^{B}[kB] = (\mathbf{x}^{N,B}[kB])^{t} \mathbf{d}^{N}[kB] = (\mathbf{x}^{N,B}[kB])^{t} (\mathbf{d}_{\perp}^{N}[kB] + \mathbf{d}_{\parallel}^{N}[kB]) =
\]

\[
= (\mathbf{x}^{N,B}[kB])^{t} \mathbf{d}_{\perp}^{N}[kB] + (\mathbf{x}^{N,B}[kB])^{t} \mathbf{x}^{N,B}[kB] \mathbf{c}_{B}[kB].
\]

The solution of these equations is given by:

\[
\mathbf{c}_{B}[kB] = \frac{1}{B} (\mathbf{R}^{B}[kB])^{-1} \mathbf{r}^{B}[kB],
\]

with the estimate of the BxB autocorrelation matrix defined as:

\[
\mathbf{R}^{B}[kB] = \frac{1}{B} (\mathbf{x}^{N,B}[kB])^{t} \mathbf{x}^{N,B}[kB].
\]

The \((p,q)\)th element of this matrix is:

\[
(\mathbf{R}^{B}[kB])_{p,q} = \frac{1}{N} (\mathbf{x}^{N}[kB-B+1+p])^{t} \mathbf{x}^{N}[kB-B+1+q] = \frac{1}{N} \sum_{i=0}^{N-1} x[kB-B+1+p+i] x[kB-B+1+q+i].
\]

When the input signal is (approximately) stationary, it follows that this quantity is an estimate of the autocorrelation function \( \rho \) of the input signal \( x[k] \), thus:

\[
(\mathbf{R}^{B}[kB])_{p,q} \approx \rho_{|p-q|}[kB].
\]

The vector \( \mathbf{d}^{N}[kB] \) can be rotated in such a way that it becomes more orthogonal in comparison with the previous one, so that its length is reduced by using the following update equation:

\[
\mathbf{d}^{N}[kB+1] = \mathbf{d}^{N}[kB] - 2\alpha B \mathbf{d}_{\perp}^{N}[kB] = \mathbf{d}^{N}[kB] - 2\alpha \mathbf{x}^{N,B}[kB] (\mathbf{R}^{B}[kB])^{-1} \mathbf{r}^{B}[kB].
\]

This equation leads, with the definition of the difference vector, to the BOP update equation:

\[
\mathbf{w}^{N}[kB+1] = \mathbf{w}^{N}[kB] + 2\alpha \mathbf{x}^{N,B}[kB] (\mathbf{R}^{B}[kB])^{-1} \mathbf{r}^{B}[kB].
\]

Comparing this algorithm with the BNMLS update equation shows that the BOP algorithm is a generalization of the BNLMS algorithm. The input signal is normalized (decorrelated) with a BxB autocorrelation matrix.
1.2.2 Efficient implementations of Block Adaptive Filters

The two main operations required by the adaptive algorithm, the convolution and the correlation, are efficiently performed, for large filter lengths, in the frequency domain. Overlap-save is a well known technique to convolve an infinite length input signal \((x[k])\) with a finite length impulse response \((w_i, i=0...N-1)\).

1.2.2.1 Overlap-save method for fixed filters

The weight vector \(\mathbf{w}^N\) is fixed. The block processing approach is such that each step a block of B output samples must be calculated by a linear convolution:

\[
\mathbf{x}[kB] = (\mathbf{X}^M[B][kB])^T \mathbf{w}^N. \tag{1.2.26}
\]

The overlap-save method is based on the partial convolution of a length M segment of the input signal \(x[k]\) with a length N weight vector \(\mathbf{w}^N\), \(M \geq N-1\) and \(B\). It can be implemented as depicted in figure 1.2.2.

Each step, B new input signal samples \(x[k]\) are collected ("S/P") and split into overlapping length M vectors \(\mathbf{x}^M[kB]\) ("overlap"). The overlap must be at least \(N-1\) samples.

The vector \(\mathbf{x}^M[kB]\) is transformed to frequency domain ("DFT\_M"). Mathematically, this can be described with the following equation:

\[
\mathbf{x}^M[kB] = F^M \cdot \mathbf{x}^M[kB]. \tag{1.2.27}
\]

The \((k,l)^{th}\) element of the MxM Fourier matrix \(F^M\) is defined as:

\[
(F^M)_{k,l} = e^{-j\theta_M k l}, \quad \text{with} \quad \theta_M = \frac{2\pi}{M}, \quad \text{and} \quad k, l \in (0, 1, \ldots, M-1). \tag{1.2.28}
\]
On the other side, the length $N$ weight vector $w^N$ is mirrored ("$J^N$") and augmented with zeroes ("$0^{M-N}$") to a vector of length $M$, which is transformed to the frequency domain. With the $N \times N$ mirror matrix $J^N$, whose $(k,l)^{th}$ element is defined as:

$$(J^N)_{k,l} = \begin{cases} 1 & \text{if } k+I = N-1 \\ 0 & \text{elsewhere} \end{cases} \tag{1.2.29}$$

this can be mathematically described as follows:

$$W^M = F^M \begin{pmatrix} I^N \\ 0_{M-N,N} \end{pmatrix} J^N w^N, \tag{1.2.30}$$

where $I^N$ is the $N \times N$ identity matrix, and $0_{M-N,N}$ is an $(M-N) \times N$ zero matrix.

The (cyclic) convolution is carried out in frequency domain by an element-wise multiplication of the vectors $X^M[kB]$ and $W^M$ as follows:

$$E^M[kB] = X^M[kB] \otimes W^M. \tag{1.2.31}$$

The result $E^M[kB]$ is transformed back to the time domain ("IDFT$_M$"). Only the last $B$ samples from this cyclic convolution match the linear convolution result; $M-B$ samples have to be discarded, resulting in the length $B$ vector $\hat{e}^B[kB]$. Mathematically, this result can be described as:

$$\hat{e}^B[kB] = (0^{B,M-B} I^B \cdot F^M)^{-1} E^M[kB]. \tag{1.2.32}$$

The original sample rate is obtained by desegregating ("P/S") this vector into the samples $\hat{e}[k-B+1],...,\hat{e}[k]$.

Applying this block processing technique results in a processing delay of $B-1$ samples. It was thus described a way of implementing the linear convolution [1.2.26] in an efficient way [1.2.31] and [1.2.32], as depicted in figure 1.2.2. It is to be noted that the input signal matrix $X^{MB}[kB]$ and the input signal vector $x^M[kB]$ contain the same samples.
It can be shown that the same result can be derived directly from the original linear convolution expression \[1.2.26\].

Since the used Fourier transforms are circulant, we first have to make a circulant extension of the input signal matrix \(X^{NB}[kB]\); we construct the circulant MxM matrix \(\hat{X}^{M}[kB]\) as the circulant expansion of the NxB mirrored matrix \(J^N \cdot X^{NB}[kB]\) \((M \geq N + B - 1)\). The circulant expansion is created by putting the matrix \(J^N \cdot X^{NB}[kB]\) in the upper right corner of \(\hat{X}^{M}[kB]\) and filling in the missing elements in such a way that \(\hat{X}^{M}[kB]\) becomes circulant. The following relationship stands:

\[
\begin{pmatrix}
I^N & 0^{N,M-N}
\end{pmatrix} \cdot \hat{X}^{M}[kB] \cdot \begin{pmatrix}
0^{M-B,B} \\
I^B
\end{pmatrix} = J^N \cdot X^{N,B}[kB].
\] \[1.2.33\]

It is known from literature \[11\] that a circulant matrix can be diagonalized by the Fourier matrix as follows:

\[
F^M (\hat{X}^{M}[kB])' (F^M)^{-1} = X^{M}[kB].
\] \[1.2.34\]

The diagonal elements of the MxM diagonal matrix \(X^{M}[kB]\) are defined as the Fourier transformed of the input signal vector \(x^{M}[kB]\), thus

\[
X^{M}[kB] = \text{diag}(\hat{X}^{M}[kB]), \quad \text{with} \quad \hat{X}^{M}[kB] = F^M x^{M}[kB].
\] \[1.2.35\]

Using the fact that \(J^N \cdot J^N = I^N\), the overlap-save method for the linear convolution can be mathematically described as follows:

\[
\begin{align*}
\hat{e}^{kB} & = (X^{N,B}[kB])' w^N \\
& = (J^N X^{N,B}[kB])' J^N w^N \\
& = (0^B,M-B I^B) (F^M)^{-1} F^M (\hat{X}^{M}[kB])' (F^M)^{-1} F^M \begin{pmatrix}
I^N \\
0^{M-N,N}
\end{pmatrix} J^N w^N \quad \text{[1.2.36]}
\end{align*}
\]

It is to be mentioned that the above described efficient implementation of the linear convolution (figure 1.2.2) costs 3 DFTs (one being superfluous for fixed filters). When \(M\) is a power of two, these transforms can be implemented with FFTs. The complexity of the transformations will be denoted by the quantity \(f(M)\). Furthermore, two complex-valued length \(M\) vectors have to be multiplied (one complex multiplication being equivalent to four real multiplications plus two additions, but the additions for one element of the vector are executed in parallel with the multiplications for the next element). Since both the input signal \(x[k]\) and the weight vector \(w^N\) are real, the resulting vectors in the frequency domain have symmetry properties, and only half plus one of the frequency components have to be calculated. As a rough measure of complexity will be used the number of real multiplications needed to produce one output sample, as explained in section 1.1. Thus, for \(M>1\), the complexity of the implementation in figure 1.2.2 is roughly given by:

\[
MUL \approx \frac{3 \cdot f(M) + \frac{1}{2} \cdot 4 (M-1)}{B}.
\] \[1.2.37\]
1.2.2.2 Efficient implementation of BNLMS

The above described efficient implementation of the overlap-save method is used for the BNLMS algorithm. The result is depicted in figure 1.2.3.

The convolution, that is carried out at the right side of figure 1.2.3, is a straightforward replica of the implementation in figure 1.2.2.

Furthermore, the signal \( e[kB] \) is desegmented ("S/P") into a length B vector \( e^B[kB] \).

The length B residual signal vector is given by \( e^B[kB] = e^0[kB] - e^B[kB] \).

The return to the original sampling rate is carried out by desegmenting this residual signal vector ("P/S"). This results in the delayed residual signal \( r[k-B+1] \).

Furthermore, the gradient estimate \( X^{N,B}[kB] \cdot R^B[kB] \) from the BNLMS update equation [1.2.8] is computed at the left side of figure 1.2.3. This correlation operation is equivalent to the convolution operation, the difference being that the input signal needs to be mirrored.

The mathematical description of the efficient overlap-save implementation in the frequency domain of this correlation operation is equivalent to the one in [1.2.36]:

\[
X^{N,B}[kB] \cdot R^B[kB] = J^N (I^{N0N,M^N}) (F^M)^{-1} F^M X^M[kB] (F^M)^{-1} F^M \left( O^{M,B,B} \right) R^B[kB] \\
= J^N (I^{N0N,M^N}) (F^M)^{-1} (X^M[kB]) \ast R^M[kB] \\
= J^N (I^{N0N,M^N}) (F^M)^{-1} \{ (X^M[kB]) \ast \otimes R^M[kB] \} \quad \text{[1.2.38]}
\]

![Figure 1.2.3: Overlap-save implementation of BNLMS](image-url)
The Fourier transform of the residual signal vector is defined as follows:

\[ R^M[kB] = F^M \begin{pmatrix} 0^{M-B} & B \\ I^B \end{pmatrix} \mathbf{r}^B[kB]. \]  \[1.2.39\]

The residual signal vector \( \mathbf{r}^B[kB] \) is augmented with zeroes ("0^{M-B}") to a length \( M \) vector, and then transformed to frequency domain ("DFT\( _M^x \)).

Equation [1.2.38] can be implemented in an efficient way in the frequency domain. The main difference between correlation and convolution is reflected in the frequency domain by the fact that the correlation needs a complex conjugation ("\(^*\)) of the input signal vector \( X^M[kB] \).

As mentioned before, the BNLMS algorithm is used. The normalization is carried out in such a way that the number of multiplications is minimized. For this, each element of \( \mathbf{r}^B[kB] \) is first multiplied by the adaptation constant \( 2\alpha \sigma_s^{-2} \mathbf{1}^M \), where the vector \( \mathbf{1}^M \) is an \( M \)-dimensional vector with all elements equal to 1.

The complexity, for \( M > 1 \), for the efficient overlap-save implementation of the BNLMS algorithm with 5 FFTs, as depicted in figure 1.2.3, is as follows:

\[ \text{MUL}_{\text{BNLMS}} \approx 5f(M) + 4(M-1) + B. \]  \[1.2.40\]

### 1.2.2.3 Efficient implementation of BOP

The same approach is used to describe an efficient way to implement the BOP algorithm in the frequency domain.

The filter part of the BOP algorithm is equivalent to the filter part of the BNLMS algorithm. This is depicted at the right part of figure 1.2.4.

The update equation [1.2.25] of the BOP algorithm can also be implemented efficiently in the frequency domain. For this, the update equation is rewritten as follows:

\[ \mathbf{w}^N((k+1)B) = \mathbf{w}^N[kB] + \mathbf{X}^N, B[kB] \mathbf{r}^B[kB], \]  \[1.2.41\]

\[ \text{with : } \mathbf{r}^B[kB] = 2\alpha(\mathbf{R}^B[kB])^{-1}\mathbf{l}^B[kB]. \]  \[1.2.42\]

Since these equations have the same structure as the convolution operation in [1.2.36], the implementation in the frequency domain is efficient. The implementation of \( \mathbf{X}^N, B[kB] \mathbf{r}^B[kB] \) is equivalent to [1.2.38], with the only difference that \( \mathbf{r}^B[kB] \) must be replaced by the vector \( \mathbf{y}^B[kB] \), whose transformed vector \( \mathbf{Y}^M[kB] \) is defined as:

\[ \mathbf{Y}^M[kB] = F^M \begin{pmatrix} 0^{M-B} & B \\ I^B \end{pmatrix} \mathbf{y}^B[kB]. \]  \[1.2.43\]

This vector can be efficiently computed by using the same approximation [7] as follows:

\[ \mathbf{Y}^M[kB] \approx 2\alpha F^M(\mathbf{R}^M[kB])^{-1} \begin{pmatrix} 0^{M-B} & B \\ I^B \end{pmatrix} \mathbf{r}^B[kB]. \]  \[1.2.44\]

In this approximation (referring to the change of the information amount in the autocorrelation function), the first column of the \( M \cdot M \) circulant matrix \( \mathbf{R}^M[kB] \) equals the "pseudo-autocorrelation" vector \( \mathbf{p}^M = (\mathbf{p}_0[kB], \ldots, \mathbf{p}_{M-1}[kB])' \).

\[ \mathbf{p}^M \]  \[1.2.45\]
The inverse of a circulant matrix is a circulant matrix \([11]\), and from this it is obvious that the circulant matrix \((R^M[kB])^{-1}\) can be diagonalized with the Fourier matrix \(F\) as follows:

\[
F^M (R^M[kB])^{-1} F^M = (P^M[kB])^{-1}.
\]  

[1.2.47]

The diagonal elements of the diagonal matrix \(P^M[kB]\) are defined as the Fourier transform of the vector \(\hat{P}^M[kB]\), thus

\[
P^M[kB] = \text{diag} \{P^M[kB]\}, \text{ with } P^M[kB] = F^M \hat{P}^M[kB].
\]  

[1.2.48]

The result is that the efficient computation of \(Y^M[kB]\) is given by the following equation:

\[
Y^M[kB] = 2\alpha (P^M[kB])^{-1} \circ R^M[kB].
\]  

[1.2.49]

The operator \((\cdot)^{-1}\) denotes the elementwise inversion of a vector.

A smoother estimate for each separate frequency bin can be obtained by using an equivalent estimate as for the BNLMS normalization:

\[
(P^M[(k+1)B])_j = (1 - \beta)(P^M[kB])_j + \frac{\beta}{M} |(X^M[kB])_j|^2.
\]  

[1.2.50]

To avoid divisions, the same approximation (the note at the end of section 1.2.4) can be used:

\[
(P^M[(k+1)B])_j^{-1} = (1 + \beta)(P^M[kB])_j^{-1} - \frac{\beta}{M} (P^M[kB])_j^{-2} |(X^M[kB])_j|^2.
\]  

[1.2.51]
The vector $\mathbf{P}^{M}[kB]$ can be computed in the frequency domain as follows:

$$\mathbf{P}^{M}[kB] = \frac{1}{M} \left( \mathbf{X}^{M}[kB] \otimes (\mathbf{X}^{M}[kB])^{\ast} \right),$$  \hspace{1cm} \text{[1.2.52]}

and this implies that $\langle \mathbf{P}^{M}[kB] \rangle_{1}^{-1} = M/|\langle \mathbf{X}^{M}[kB] \rangle_{1}|^{2}$.

The result of the above described efficient implementation of the BOP algorithm in the frequency domain is the Block Frequency Domain Adaptive Filter (BFDAF) [12], whose implementation is depicted in figure 1.2.4.

For $M > 1$, the complexity for the BFDAF structure depicted in figure 1.2.4 is given by:

$$\text{MUL}_{\text{BFDAF}} = \frac{5f\{M\} + 7(M-1)}{B}. \hspace{1cm} \text{[1.2.53]}$$

### 1.2.3 Decoupled Partitioned Block Frequency Domain Adaptive Filter

For large filters, the Block Frequency Domain Adaptive Filter, as previously described, combines a relative low implementation complexity with a good performance [5]. On the other hand, the resulting processing delay is very large. For many practical applications, such as the acoustic echo canceller, this is a problem. Trying to overcome it by using a small block length $B$, increases the complexity in such a way that the approach becomes very inefficient. Next it is introduced a solution for this problem.

#### 1.2.3.1 Partition of the filter-part

As mentioned before, the block processing approach produces a $B$-dimensional vector of output samples, $\mathbf{y}^{B}[kB]$, that implies $B$ convolutions of $M$ input signal samples with a filter impulse response of length $N$.

This filter can be partitioned into smaller subfilters of length $Q$. For simplicity, $N/Q$ is assumed to be integer. The proposed partitioning can be described as follows:

$$\mathbf{X}^{N,B}[kB] = (\mathbf{X}_{N,Q-1}^{B}[kB])^{\prime} \cdot \mathbf{w}^{N}[kB] \hspace{1cm} \text{[1.2.54]}$$

$$= \sum_{i=0}^{N/Q-1} (\mathbf{X}_{i}^{Q,B}[kB])^{\prime} \cdot \mathbf{w}_{i}^{Q}[kB], \hspace{1cm} \text{[1.2.55]}$$

where the partitioning of the input signal matrix is defined as:

$$(\mathbf{X}^{N,B}[kB])^{\prime} = ((\mathbf{X}_{N/Q-1}^{Q,B}[kB])^{\prime} \ldots (\mathbf{X}_{0}^{Q,B}[kB])^{\prime})$$

$$(\mathbf{X}_{i}^{Q,B}[kB])^{\prime} = (\mathbf{x}_{B-iQ-Q+1}^{B}[kB-\ast \ast \ast \ast] \ldots \mathbf{x}^{B}[kB-iQ]). \hspace{1cm} \text{[1.2.56]} \hspace{1cm} \text{[1.2.57]}$$

The adaptive filter coefficients are partitioned in an equivalent way as follows:

$$\mathbf{w}^{N}[kB] = ((\mathbf{w}_{N/Q-1}^{Q}[kB])^{\prime} \ldots \mathbf{w}^{Q}[kB])^{\prime} \hspace{1cm} \text{[1.2.58]}$$

$$\mathbf{w}_{i}^{Q}[kB] = (w_{iQ-1}[kB] \ldots w_{iQ}[kB])^{\prime}. \hspace{1cm} \text{[1.2.59]}$$
The \( N/Q \) convolutions of [1.2.55] can be computed separately by using an overlap-save method, in the frequency domain, as previously described. With \( M \geq Q+B-1 \), the resulting partitioned convolution can be described as follows:

\[
X_j^{M}[kB] = F_M \cdot X_j^{M}[kB - i/Q]
\]

\[
W_j^{M}[kB] = F_M \cdot \left( I^Q \right) \cdot J^Q \cdot W_j^Q[kB]
\]

\[
E^{M}[kB] = \sum_{i=0}^{N/Q-1} X_j^{M}[kB] \otimes W_j^{M}[kB]
\]

\[
\hat{e}^B[kB] = (0^B, M-B \cdot I^B) \cdot (F_M)^{-1} \cdot E^{M}[kB].
\]

When assuming that \( Q/B = q \) is an integer, a simple delay line in the frequency domain can be used for the vectors \( X_j^{M}[kB] \), with \( i>0 \), since then:

\[
X_j^{M}[kB] = X_0^{M}[(k-iq)B].
\]

The realisation of [1.2.63] is depicted in figure 1.2.5. Each box "\( qT_B \)" represents \( q \) delay elements of \( T_B=B \cdot T \). The resulting processing delay of this partitioned filter is \( B-1 \) samples. When only a very small processing delay is allowed, it is shown in section 1.2.4 that, in comparison with the BFDAF approach, this partitioned approach can be realised with less complexity.

Figure 1.2.5: Partitioned filter-part
1.2.3.2 Partition of the update-part

As in the previous subsection, the BOP update equation [1.2.25] is partitioned in \( N/Z \) parts, and each of them is efficiently implemented in the frequency domain.

The block length for the update equation equals \( A \). With this, the partitioned BOP update equation becomes, for \( i=0,1,...,N/Z-1 \):

\[
\mathbf{w}_z[(k+1)A] = \mathbf{w}_z[kA] + X_z^A[kA] \cdot \mathbf{y}^A[kA],
\]

[1.2.65]

with \( \mathbf{y}^A[kA] = 2\alpha(\mathbf{R}^A[kA])^{-1} \mathbf{r}^A[kA] \).

[1.2.66]

With \( L \cdot Z + A - 1 \), an equivalent procedure as used for the filter-part can be utilised to approximate this equation in such a way that it can be implemented efficiently in the frequency domain as follows:

\[
\mathbf{Y}^l[kA] = 2\alpha(\mathbf{P}^l[kA])^{-1} \otimes \mathbf{R}^l[kA].
\]

[1.2.67]

Furthermore, using the definitions [1.2.60] and [1.2.61], this yields:

\[
\mathbf{w}_z[(k+1)A] = \mathbf{w}_z[kA] + J^Z \cdot (\mathbf{I}^Z \cdot \mathbf{Z}^{Z-A}) \cdot (\mathbf{P}^l)^{-1} \cdot \mathbf{X}^l[kA] \otimes \mathbf{Y}^l[kA].
\]

[1.2.68]

When assuming that \( Z/A = z \) is an integer, a simple delay line in the frequency domain can be used, since then \( \mathbf{X}^l[kA] = \mathbf{X}_0^{l+[(k-z)A]} \).

The partitioned update-part is depicted in figure 1.2.6.

---

Figure 1.2.6: Partitioned update-part
1.2.3.3 Coupling of the partitioned filter- and update-part

**Partitioned BFDAF**

The easiest way to couple the partitioned filter- and update-part is to choose the different partition parameters as being equal, by making \( Z = Q \), \( A = B \), and \( L = M \). The result is the Partitioned BFDAF (PBFDAF) [14]. In this case the vectors \( \mathbf{X}^{L}_{0}[kA] \) until \( \mathbf{X}^{L}_{NZ-1}[kA] \) on figure 1.2.6 are equivalent to the vectors used in the delay line on figure 1.2.5, and thus they can be combined in one delay line.

For \( M > 1 \), the complexity of the PBFDAF is given by the following:

\[
MUL_{\text{PBFDAF}} = \frac{(3 + 2 \frac{N}{Q}) f(M) + (3 + 4 \frac{N}{Q})(M-1)}{B}.
\]

For \( Q = N \) (no partitioning) we have \( MUL_{\text{PBFDAF}} = MUL_{\text{BFDAF}} \).

**Decoupled PBFDAF**

Choosing different partition factors for the filter- and the update-part makes possible an efficient overall implementation that realizes a given minimal processing delay (by setting \( B \) small) and good convergence properties (by taking large \( Z \) and \( A \)). The result is the Decoupled Partitioned BFDAF (DPBFDAF) [6]. For this implementation, two interfaces between the filter- and the update-part have to be defined, one for the adaptive weight vector and the other one for the residual signal vector.

For the interface that couples the adaptive weight vector it is to be noted that it is useless to compute this vector more often than needed. For this reason \( A \geq B \). For simplicity reasons, \( A \) is taken integer multiple of \( B \), thus \( A = \lambda \cdot B \) with \( \lambda \in \{1, 2, \ldots\} \). The coupling is performed by first merging the separate vectors of the update-part. A hold function is needed before this vector is split in parts that are needed for the filter-part. This is depicted in figure 1.2.7 and can be expressed as follows:

\[
\begin{pmatrix}
\mathbf{w}^{Q}_{N/Q-1}[kB] \\
\vdots \\
\mathbf{w}^{Q}_{0}[kB]
\end{pmatrix}
= 
\begin{pmatrix}
\mathbf{w}^{Z}_{N/Z-1}[kA] \\
\vdots \\
\mathbf{w}^{Z}_{0}[kA]
\end{pmatrix}
\]

\[1.2.70\]

The partitioned vectors need to be transformed to the frequency domain. All operations are to be performed at the lowest possible rate. For this reason the hold function is performed after the DFTs. This is not shown on figure 1.2.7, but it is done within the implementation on the processor.

For an acoustic echo canceller, it is possible to control the hold function with information available from a double talk detector.
Figure 1.2.7: Interface for the adaptive weights

Thus, all partitioned vectors $w^Z_i[kA]$ of the update-part are merged, and then split to the vectors $w^Q_i[kB]$ that are needed for the filter-part. These vectors are mirrored ($J^0$) and transformed to the frequency domain at a rate $1/\lambda BT$ instead of $1/BT$.

For the interface of the residual signal vector it is to be noted that the filter-part generates vectors $r^B[kB]$. These vectors contain $B$ samples at a rate $1/BT$. Since $A/B = \lambda$ is assumed to be integer, $\lambda$ of these residual signal vectors can be put in a delay line, up to a length $A$ vector. This length $A$ vector can be read out at a rate $1/\lambda BT$, as depicted in figure 1.2.8.

Figure 1.2.8: Interface for the residual signal with $A = \lambda \cdot B$

Thus, the residual signal vector $r^B[kB]$ is put in a delay line at rate $1/BT$. This delay line contains $\lambda - 1$ delay elements $T_B = B \cdot T$ seconds. The contents of this delay line is put in the residual signal vector $r^A[kA]$ at a rate $1/\lambda BT$. The relation between the residual signal vectors is as follows:

$$r^A[kA] = ((r^B[(k-\lambda+1)\cdot B])') \ldots (r^B[k\cdot B]')'$$.  [1.2.71]
Defining $g_F = [N/Q]$ and $g_U = [N/Z]$, for $M > 1$, $L > 1$, the complexity of the efficiently implemented DPBFDAF structure is given by:

$$MUL_{DPBFDAF} = \frac{(2+\frac{1}{\lambda}g_F) \cdot f(M) + 2g_F(M-1)}{B} + \frac{(2+g_U) \cdot f(L) + (3+2g_U) \cdot (L-1)}{\lambda B}. \tag{1.2.72}$$

For $g_F = g_U$ and $A = B$ (thus $\lambda = 1$ and $L = M$), the DPBFDAF structure is reduced to the PBFDAF structure. In this case the above stated complexity does not equal $MUL_{PBFDAF}$. The reason for this is that the delay lines of the update- and filter-part are combined in the PBFDAF approach. In general for the DPBFDAF this combination cannot be performed.

### 1.2.4 Simulation results and conclusions

The results of the previous sections are compared. The number of real multiplications per sample for BFDAF, PBFDAF, DPBFDAF is plotted as a function of the allowable delay in figure 1.2.9. Note: an addition (or subtraction) takes as long as a multiplication (1 clock), but, within different parts of the algorithms, there are less additions (subtractions) to be performed, and they are coded to be executed in parallel with the multiplications. An input signal is used that is sampled at 8kHz. The echo canceller must be able to cancel echoes up to 250ms, thus $N \geq 2000$. The other parameters are chosen to minimize the number of real multiplications per sample for a given time delay $B \cdot T$. It is assumed that a real FFT/IFFT of length $M$ requires $f(M) = 3/4 \cdot \log_2(M/4)$ real multiplications.

Experiments made with an ARMA(10) far-end input signal $x[k]$, a white noise near-end input signal $s[k]$, a simulated room impulse response truncated at 250ms as echo $e[k]$, the sampling frequency 8kHz and the maximum allowable block-length 4, show [6] that PBFDAF's convergence behaviour is degraded in comparison with BFDAF, while DPBFDAF has a better convergence behaviour than BFDAF.

Thus, with the DPBFDAF it is possible to realize an efficient structure having both small processing delay and good convergence properties. These results can be used in the acoustic echo cancellation application. A reasonable number for the adaptive filter length in such an application is $N = 2000$. When a processing delay is allowed of 4 samples, the DPBFDAF structure can be realized with almost a factor 20 less complexity in comparison with an BFDAF having the same processing delay.
Figure 1.2.9: Complexity of different algorithms function of maximum delay

**Note**: power normalization without divisions

A division with the accuracy offered by the single-precision floating-point format \((2^{-23})\) lasts for instance 31 clocks, on the TMS320C30 processor.

The following equation:

\[
\hat{\sigma}_{x^2}[ ( k+1 ) B ] = \frac{1}{( 1 - \beta ) \hat{\sigma}_{x^2}[ kB ] + \beta \frac{1}{B} ( X^N, B[ kB ] )^t X^N, B[ kB ] }
\]

\[\text{[1.2.73]}\]

can be approximated, for small \( \beta \), by an equation without divisions:

\[
\hat{\sigma}_{x^2}[ ( k+1 ) B ] = \frac{\frac{1}{1 - \beta} \hat{\sigma}_{x^2}[ kB ]}{1 + \frac{\beta}{1 - \beta} \hat{\sigma}_{x^2}[ kB ] \frac{1}{B} ( X^N, B[ kB ] )^t X^N, B[ kB ]}
\]

\[
\approx \frac{\frac{1}{1 - \beta} \hat{\sigma}_{x^2}[ kB ]}{1 - \beta ( 1 - \beta )^2 \hat{\sigma}_{x^2}[ kB ]^2 \frac{1}{B} ( X^N, B[ kB ] )^t X^N, B[ kB ]}
\]

\[
\approx ( 1 + \beta ) \hat{\sigma}_{x^2}[ kB ] - \beta ( \hat{\sigma}_{x^2}[ kB ] )^2 \frac{1}{B} ( X^N, B[ kB ] )^t X^N, B[ kB ] \cdot X^N, B[ kB ].
\]

\[\text{[1.2.74]}\]

No divisions are required if \( 1/B \) is combined in one constant with \( \beta \), as \( \beta/B \).
1.3 The hardware used

To satisfy the computational load requirements, the TMS320C30 chip of Texas Instruments is a good choice. Working at a clock of 60ns, it provides 40 bits floating point arithmetic as well as 32 bits integer arithmetic, parallel instruction set (for parallel multiply and add/subtract sequence, the computational power is 33 MFLOPS), zero overhead loops and single clock branches, modified Harvard architecture, on-chip two RAM blocks of 1k words and one ROM block of 4k words, 64 words program cache, one Direct Memory Access (DMA) controller, two serial ports and two timers.

The C30's architecture has a pipelined structure, with multiple busses, allowing arithmetic operations and memory accesses, data and instruction address generation to be executed all of them in parallel.

As shown on figure 1.3.1, the main data manipulation registers, \( r0...r7 \), fed directly from the hardwired multiplier and Arithmetic-Logic Unit (ALU), have a direct path back into them, enabling a multiplier and ALU operation to be performed every clock.

The registers commonly used to hold addresses, \( ar0...ar7 \), are connected to a pair of integer ALUs, allowing two data addresses to be updated every clock.

---

![Diagram of C30 CPU architecture](image)

Figure 1.3.1: A simplified representation of the C30 CPU architecture
Like all other DSPs, the C30 has two dedicated addressing modes: the modulo (circular) addressing, to implement without overhead the shifting of data, and the bit-reversed addressing, to reduce the FFT computational load. Light restrictions concerning data location in the memory map have to be considered.

The program cache is useful when executing loops shorter than 64 words: the instructions are fetched from the cache, the resources being freed for other accesses.

The DMA controller reads and writes anywhere in the memory map, concurrently with the CPU accesses, decreasing the overhead caused by the I/O necessities, as well as being able to compensate the restriction introduced by the limited size on-chip RAM, where the peak performance is reached.

Apparently, instructions require four clocks for completion. The four-stage pipeline, Fetch-DecodetRead-Execute, each stage processing a different instruction, can execute them in one clock.

When a branch instruction is decoded, the CPU stops fetching, to avoid processing any stage of the following instructions, and, after the branch is taken, starts filling the pipeline again (this is a branch conflict).

The delayed branches eliminate this overhead, executing the following three instructions, before the branch is taken - therefore these must not affect the Program Counter. To use a delayed branch, write a standard branch, and then convert it, if there are clocks to gain, by carefully moving the instructions from before the branch to after it.

When executing a repeat-block-of-instructions, there are also no clocks spent to jump from the end to the beginning of the loop.

Other instructions that make use of the C30's internal concurrency are the parallel arithmetic and/or memory access.

Some addressing constraints, introduced by the particular hardwired implementation of the CPU and by the potential pipeline conflicts, show up especially when coding computational-intensive algorithms.

The pipeline conflicts arise when an instruction processing must pass to the next stage in the pipeline but that stage is busy.

Besides the already mentioned branch conflicts, register conflicts are encountered when the Decode unit accesses registers involved in the address generation, apparently (as the source code states) supposed to be earlier accessed by the Execute unit, but in fact (due to the pipelined instructions processing) being accessed later.

Finally, memory conflicts happen when the bandwith of a physical memory space is exceeded - each internal memory block is able to hold two accesses per clock, while the external bus interfaces support only one.

The interrupts, if enabled, affect the CPU activity and/or synchronize the DMA controller accesses.

The interrupt prioritization is fixed.

A delayed branch in the pipeline postpones an interrupt until after the branch is taken.

A repeat-single instruction is not interruptible.

Other remarks accompany the relevant parts of code, in section 5.1.
To develop an application until the final stage, people use a standard board, plugged into the PC, and the necessary software - simulator, monitor, debugger. The board available to test this application is the "TMS320C30 PC System Board", delivered with software by the Loughborough Sound Images Limited.

Besides the C30 operating from a 33.3MHz clock, the board is equipped with 192k words of zero wait-states and 64k words of one wait-state memory, a PC interface, and an analog interface with two 16-bit A/D and D/A converters. The conversion rate, set to be generated by the C30 timer1, can be as high as 200kHz. An interrupt1 request is issued to the C30 when the conversions end.

The one-wait state memory area is double-ported, being accessed from the PC by cycle stealing from the DSP if both try to access it simultaneously.

The board is suitable for the ATs and compatibles, as it uses the full 16-bit interface.

The communication speed between the DSP and the PC is limited by the speed of the AT and its software. An interrupt from the PC to the DSP and vice versa is supported.

The board is configured in the microprocessor mode, the 4k words internal ROM being disabled.

The timer can be programmed to a resolution of 120ns, for the mentioned clock rate.

The 16-bit words supplied by the PC are integers from the C30's point of view.

For each channel, the input for the D/A converter and the output of the A/D converter are mapped at the same (I/O) address, the difference being whether it is a read or a write. After a conversion, first must be the read and after that can follow the write, otherwise it is read what was just written, and not the result of the A/D conversion.

The total available memory map for this application is depicted in figure 1.3.2.

For further hardware details we refer to [15-17], while for the software to section 5.4.
2 The implementation

This chapter describes the way the DPBFDAF algorithm (section 1.2.3) is fit to work on one TMS320C30 processor. Deeper technical details can be found in chapter 5 and inside the source files.

At the first sight, the code is not easy readable, like any assembly program, and also not usable somewhere else (only the IFFT and the convolution routines can be easily used in another application).

Programming in assembler is very unproductive, but this was the only solution, because of the execution speed constraint. Even for this variant, very tight coded, the processor is loaded 92%, for the concrete example shown in section 5.2.

Very often, for instance, a register involved in address generation is loaded, to avoid a conflict, much earlier than its role begins. One conflict here, one conflict there... One inside a loop, which can be inside another loop... Step by step, the threshold of 100% load can be exceeded.

First are pointed out the prerequisites which lead to this type of implementation.

Then, each of the modules is discussed - namely the filter, the update, and the interrupt. Finally, the main program is described.

2.1 General ideas behind the implementation

This adaptive filter is an essentially computational-intensive algorithm, working on blocks of samples basis. The program must handle the computations, as well as the input/output, between these a synchronization being required.

As described in section 1.2.3, the filter-part (further named filter) processes over blocks of B samples, and the update-part computes over blocks of A samples. The coupling interfaces between these two parts are included in the last one (both coupling- and update-part are further referred to as update).

Example: with $B=8$ and $A=504$ ($A/B=63$), for each 504 samples passed there must be one update routine and 63 filter routines completed as well as the necessary input/output performed.

The problem is to synchronize all of these.
One solution is to let the CPU execute the filter routines and the split update routine, and to leave the input/output to be handled by the DMA controller. In this case, the CPU has to poll (before the right moments) for the proper blocks of samples available in the I/O buffers. This type of synchronization is difficult to realize when the purpose is to obtain the fastest variant of code. Besides this, it is complicated the input/output to be performed by the DMA controller, because of the missing circular addressing logic. Another bad reason is that the only DMA channel is much more needed by the computational parts of the algorithm. Thus, the CPU must take care about the input/output, by means of an interrupt service routine.

Once arrived at this point, the most natural way to organize the implementation shows up: the synchronization between the filter routine executions, the update routine execution, and the samples flow, must be carried out by the samples themselves. The first solution, which was supposing to split the large update routine execution into the right parts, is not a good solution, and requires too much work, as it has to be done for each possible set of parameters.

The chosen solution is simple and flexible: the interrupt service routine performs the input/output, and, each $B$ samples, causes the start of a filter routine (this is the synchronization). After the filter routine completion, the following piece of update routine is executed, until the $B^{th}$ interrupt arrives. Then, another filter routine is performed, and so on. The large update routine is split by the interrupt service routine, this fact being transparent for the programmer. The chosen solution is sketched in figure 2.1.1.

For a memory allocation example is referred section 5.2, while an allocation algorithm is suggested in section 5.3.c.

To maximize the execution speed, all the buffers excepting those holding the twiddle factors and the implementation code variables are located within the external memory. During the computational-intensive portions, the currently processed vectors are mirrored inside the (size restricted) on-chip memory. The C30 has only 2k on-chip RAM.

The "extra" buffers implied by this specific implementation are the filter and update spare vectors (size $2 \cdot \max(M)$, respectively $L$), plus the filter and update overlapped input signal vectors (both of size $L$). The other buffers are "intrinsic" to the algorithm. A list of the buffers can be found at the end of section 5.2.

There is no problem to spread all needed buffers within the available memory map on the board described in section 1.3.
Thus, each block of \( B \) input signal samples is filtered with the adaptive coefficients, which are updated each \( A \) samples. The samples input/output flow interrupts the computations and the short \textit{filter} routine executions "interrupt" the long \textit{update} routine execution. See also figure 5.2.1 (with the explanations) for a practical example.

"Not exceeding the chip computational power" means to keep the execution time of all \( A \) interrupt routines plus \( A/B \) \textit{filter} routines and one \textit{update} routine within \( A \) sampling periods. This was accomplished for the parameter set example considered in section 5.2, and has to be tested for which other sets of parameters stands.

To end this section, few details will be enumerated.

The DPBFDAF algorithm processes blocks of samples (vectors). The same computations take place on each element of a vector. This means the code will use loops. Performance is reached by implementing parallel instructions and working with on-chip resources.

The vectors are partitioned. The maximal advantage of the C30 resources is taken using the DMA controller to yield an on-chip copy of the next partition to be processed, while computations run over the current one.

This is the biggest problem behind a real-time application, namely how to fit into the available on-chip resources, where the necessary speed is reached.
The performance is perceptibly degraded with each moreover conflict on a loop, especially when it is placed inside another loop. To solve this, the loop has to be recoded until the fastest variant is found.

Although the DMA unit has the lowest priority assigned in the pipeline, it can generate conflicts, in this case difficult to observe. For instance, an one clock conflict occurs when the DMA controller initiates a two clocks access over an external bus interface which is needed during the next clock to be used by the CPU.

Because the implementation needs to work with interrupts, the \textit{rpts} (repeat single) instruction was avoided, its execution being uninterruptible. This means, for instance, that two of the samples coming each 834 clocks (at 20kHz sampling frequency) will be lost if the CPU is at that moments busy executing an 1024 clocks long \textit{rpts} instruction.

All the vectors used by the algorithm are of two types: complex symmetrical and real. This is due to the real-valued nature of the input data, as specified in section 1.2.2.1. The vectors elements disposal in their buffers is depicted on figure 2.1.2. All buffers are drawn with the lowest memory address upside. The size of each vector is specified in section 1.2.3 and inside the source files.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure2_1_2.png}
\caption{Vectors elements disposal in the buffers}
\end{figure}
2.2 The filter-part routine

The task of the filter routine is to convolve the input signal with the adaptive weights. As detailed in section 1.2.3.1, this partitioned convolution takes place in the frequency-domain. The general program and data flow of its overlap-save implementation on the C30 is sketched on figure 2.2.1.

Because the filter routine "interrupts" the much longer update routine, the first of all it saves its context: the DMA controller status, the FFT routine calling parameters, and the temporary needed on-chip memory area. The CPU status (and with this, the update-in-progress point) is saved by the (Bth) interrupt routine which switches to filter - there has no importance who saves it, the purpose is to be restored when the filter routine ends.

Depending on which part of the update routine was in progress, the DMA controller could be been busy (the status must be saved) or not busy (nothing to save). For both possibilities, the time needed to traverse this portion of code is adjusted to be the same (see appendix 5.1.a) - the reason for it appears close to the end of this section.

This up-to-here "internal housekeeping" was not indicated on figure 2.2.1, being not relevant for the algorithm flow.

The temporary needed on-chip memory area is saved later by the DMA controller, during the FFT routine execution, until then the needed external bus interface being busy with program fetches or data moves (the data processed by the FFT routine resides on-chip, thus, during the loops execution, when the code is in the cache, the external bus interface is free).

Next, the FFT routine calling parameters (excepting the data buffer address) are initialized, as they can have another set of values within the update routine execution.

Then, the overlapped time-domain input signal vector \( x_M[kB] \), included in the upper part of buffer \( XLI \) on figure 2.2.1, is updated with the newest block of \( B \) samples, from buffer \( XL \) (where the interrupt service routine inserts the far-end input signal samples - see section 2.4).

The reason for keeping the buffer \( XLI \) is to have the filter and update input signal vectors \( x_M[kB] \) and \( x[kA] \) defined in respect with the same time reference (otherwise, \( x[kA] \) directly taken over from \( XL \), at the beginning of the update routine execution, will be shifted in respect with the last filtered \( x_M[kB] \)). Both buffers being located in the external memory, the code cannot take advantage of parallel instructions; moreover, a two clocks delay occurs on the loop (external read cannot take place simultaneously with external write - see appendix 5.1.c.1). See also note 1 at the end of this section.

Next, the vector vector \( x_M[kB] \) is bit-reversed copied to the internal RAM, to be processed by the FFT routine. Parallel instructions are used (external read and internal write - see appendix 5.1.c.2). Here it is first encountered the early loading of the registers involved in address generation, to avoid conflicts; this will not be further mentioned for each occurrence. Comments about the necessary circular and bit-reversed addressing are given in appendix 5.1.d, respectively 5.1.e.
Figure 2.2.1: Partitioned filter

clock rate = B-Ts
At this point, the FFT routine is called to compute in-place the frequency domain transformation $X[kB]$ of the vector $x[kB]$. The fastest code variant is obtained placing both twiddle factors and data within the on-chip memory. For further details concerning the IFFT routine implementation, [18] is referred.

Then, the transformed vector $X[kB]$ is inserted into the delay line $MO$ (figure 2.2.1) containing the rest of the partitions $X[kB-B], ..., X[kB-(N-Q)/B-B]$. Because of the two clocks external write, parallel instructions cannot be used - see appendix 5.1.c.3. With the same notations from section 1.2.3.1, there are totally $(N/Q-1)\times 1$ length M buffers; the value inside each box (buffer) gives the time reference (upside is the newest overlapped partition), in $B$ samples-time units $B \cdot T = T_B$.

The convolution which follows is the time-critical section of the code, loading around 51% the processor, for the parameter set example in section 5.2. It consists out of $N/Q$ element-wise products and summations $F[kB] = F[kB] + X[kB] \circ W[kB]$, with $i=0,...,N/Q-1$. The partitions $X[kB] = X[(k-iq)B]$ of the input signal vector are accessed by stepping each $q$ buffers inside the delay line $MO$, while the partitions $W[kB]$ of the frequency domain adaptive weights vector are successively contained within the buffer $F0$ on same figure (here the value inside each box shows which partition is - the same significance as on figure 1.2.5). The intermediate results are summed into buffer $E$, located in the internal RAM.

With both input signal and adaptive weights vectors placed in the off-chip memory, there is no chance to reach the necessary speed, therefore one of them must be brought into the internal memory. For the first product (here is nothing to sum), the input vector partition is already there, and, for the followings, the necessary weights vector partition will be already copied (while the previous product and sum computation is in progress) by the DMA controller into the internal RAM, in one of the two areas alternatively pointered (see appendix 5.1.f) for this purpose, denoted $int.ram01/01m$ on figure 2.2.1.

All CPU registers are used here (in fact one more index register would be needed), being involved in computations or holding pointers and other local housekeeping variables.

The convolution routine implementation has, besides the initializations at the entrance, three parts: the first product, the inner $N/Q-2$ products and summations, and the last product and summation.

The first product overwrites the buffer $E$, while during the last product and sum computation there is no DMA controller task to be performed; for the rest, they are identical to the inner $N/Q-2$ ones. These inner $N/Q-2$ are executed on a loop, shorter than 64 instruction words - thus, after the first pass, the code will be inside the cache, freeing the external bus interface.

On this loop we distinguish again some initializations (start the DMA controller job - see appendix 5.1.b, update the current pointers - see source file $convo.asm$, and son on) and then the product and sum itself. Due to the symmetry of the frequency domain vectors, there are to be computed only half plus one elements. The first and the last have only real-component, requiring thus a multiplication and an addition. The rest of the elements computation involves four multiplications and four additions, being executed on a loop.

This inner loop is the key-portion of the real-time implementation. Once the general program and data flow roughly established, the implementation up-to-here was almost straightforward (still, less critical conflicts had to be minimized). Finding the fastest code variant for this key-loop did imply a lot of attempts (totally ten), and few of them can be seen in appendix 5.1.g.
Correct understanding of pipeline operation (chapter 10 in [16]) and parallel instructions set utilization (chapter 7 in [15]) are the basic condition to succeed coding at the necessary speed such computational-intensive portions.

The worst solution found here (see appendix 5.1.g.1) did suppose two separate loops, one for the real- and the other for the imaginary-component of the result. The indirect addressing of all input vectors components plus the computations and the results storing took fourteen clocks; only this inner loop is a load of around 92% (for the section 5.2 parameter set example), leaving not enough time for all the rest of the algorithm execution.

A typical idea is to code into only one loop the multiplications for one result element to be executed in parallel with the additions for the previous result element. The actually chosen sequence (see appendix 5.1.g.2) executes the multiplications for the current real-component in parallel with one addition and the store for the previous imaginary-component, then the additions for the current real-component in parallel with the multiplications for the current imaginary-component, and finally the store of the current real-component in parallel with one addition for the current imaginary-component. Indirect addressing is used for both components of the on-chip vector element and for one component of the off-chip vector element, while the other off-chip vector element component is loaded into a register.

The loop has totally six instructions, but it is executed in seven clocks. It is to be taken into account that both CPU and DMA controller need the external bus interface, the CPU for three one clock accesses and the DMA controller for two two clocks accesses (DMA controller off-chip reads). Although the CPU has greater priority assigned in the pipeline, a DMA controller access in progress cannot be interrupted, thus, anyhow would be positioned the three instructions with the external CPU accesses among the six of the loop, the DMA controller will generate, in the best case, an one clock conflict. On the other side, puzzling with the C30 parallel instructions set, we find that, in the best case, the CPU will also generate an one clock conflict (three CPU data accesses, one write and two reads).

The point is to match these two conflicts to occur during the same clock, to have in fact a total delay of only one clock. Otherwise the loop would last eight clocks, and each moreover clock on this inner loop drastically increases the load (with 5.85%, for the section 5.2 parameter set example). The time matching of the two conflicts is possible, because in fact the two units do not compete for the same resource: the DMA controller keeps busy the external bus interface for one more clock, when the CPU would normally have needed it, but it does not need it, as three CPU data accesses cannot occur at once (the CPU has two data address busses) - two of them (the external one included) being one clock postponed by the internal pipeline logic.

Thus, a better (faster) solution than a seven clocks loop does not exist. Another variant of translating this solution into five instructions is mentioned in appendix 5.1.g.3.

The outer loop schedules each partition to be convolved like this. At the end of the outer loop there is a DMA controller job completion test performed, for the data flow safety. Pipeline conflicts that occur on the outer loop have been also minimized, although they are (N/Q-2 times) less critical. A certain parameter set might exist, for which one conflict overload here (0.21% for the parameter set example in section 5.2) can just exceed 100%.

The convolution routine implementation ends by restoring one of the two (update) on-chip memory areas that have been altered (the DMA controller handles this). All the convolution routine execution lasts exactly \( \left( \frac{M}{2} - 2 \right) \left( 6 + 7 \cdot \frac{N}{Q} - 1 \right) + (\frac{N}{Q} - 2) \cdot 25 + 114 \) clocks, which means 51% processor load for the section 5.2 parameter set example.
After the convolution, the IFFT routine is invoked to transform the result $\hat{f}^M[kB]$ to the time domain. The calling parameters - including the data buffer ($E$) address - are the same as earlier initialized. The IFFT routine output is bit-reversed ordered.

The result unscrambling cannot be performed in-place, therefore the data is first copied to the (update) on-chip memory area not yet restored (see also note 2 at the end of this section).

The CPU and the DMA controller cooperate to make it faster. This is possible, because, each clock, within the internal memory, the CPU is able to perform two data accesses, and the DMA controller one access (see appendix 5.1.h).

These totally three accesses per clock cycle do not exceed the bandwidth of the concerned physical memory spaces, as the source and the target of this data transfer are located within different internal RAM blocks.

The same approach (concurrently using the DMA controller and the CPU for the same job) cannot be used for the bit-reversal which follows (although it takes place within the internal memory), because the DMA controller misses the bit-reversed addressing logic. Parallel instructions are used - see appendix 5.1.c.4.

Next is restored the (update) on-chip memory area left not restored (see also note 2 at the end of this section). An external read “allows” parallel internal write - see appendix 5.1.c.2.

The last $B$ samples of this cyclic convolution result constitute the needed answer $\hat{e}^B[kB]$, which must be subtracted from the near-end input signal combined with far-end echo (the vector $e^B[kB]$, contained in buffer $D$ on figure 2.2.1), to produce the residual signal vector $r^B[kB]$.

Supposing that the unknown (room) impulse response function is non-zero valued beginning with the origin, a correct subtraction from the vector $e^B[kB]$ is ensured by first delaying $\hat{e}^B[kB]$ with the same amount (of samples) as the filter routine needs to yield the answer $\hat{e}^B[kB]$. Figure 2.2.2 sketches this situation for the practical case considered in section 5.2. The right size of the circular buffer $D$ causes the needed delay to be inserted on the near-end path. This delay is measured by the code itself during the initializations (see section 2.5), which is again a flexible solution, not dependent on the specific parameter set the code is running with. The delay is set to be the same for all the branches the execution can take within the filter routine, up to the concerned moment (this is the postponed explanation from the beginning of this section).

To avoid any uncalled for situation (depending on the parameter set), during this subtraction the interrupts are disabled - otherwise the actual delay could not match the supposed (measured) one. Thus, a possible pending interrupt is held (postponed) for a maximum of $21+4\cdot B$ clocks ($3.18\mu$s, for $B=8$ - this means a 2.5% shift, for 8kHz sampling frequency), the consequence being that one out of $B$ samples is shifted, if the parameter set that causes this situation exists. See also note 4 at the end of this section.

The subtraction itself is coded in two loops (see appendix 5.1.c.5). During the first one, the vector $\hat{e}^B[kB]$ is subtracted from the vector $e^B[kB]$ (delayed), with the $\hat{e}^B[kB]$ overwriting; because of the three CPU data accesses, the loop cannot consist of only one instruction. The second loop inserts the result $r^B[kB]$ into its circular buffer RB (figure 2.2.1); the loop execution cannot last only one clock, due to the two clocks external write.
This subtraction, computing the residual signal vector \( r^B[kB] \), could not be implemented in only one loop: because of the different circular addressing radius for the buffers \( D \) and \( RB \), two loops are faster.

After reenabling the interrupts, the same result \( r^B[kB] \) is inserted in another buffer, \( RBPC \) on same figure, for future development of the algorithm implementation. The buffer is located within the one wait-state memory area shared between the PC and the DSP, thus the loop execution lasts three clocks (see appendix 5.1.c.6). When this (liniar) buffer is filled, a certain value placed in the PC-DSP communication area orders the PC to read the buffer, and the pointer is switched to the other \( RBPC \) (see the source file \( filter.asm \)).

Next is accomplished the big frame-level (see figure 2.1.1) synchronization function: the small frames-counter is incremented. This counter is tested at the end of the update routine, in order to wait until all \( A/B \) filter routines are performed, then to interswitch the current adaptive coefficients with the new ones and to restart the update routine. The consecutive values of this counter are recorded (see the source file \( filter.asm \)) just for testing purposes. The concerned three instructions and the recording zone can be removed.

In the end, the update status (the \( N\)FFT routine calling parameters, the CPU status, and - if necessary - the DMA controller status) is restored, and, with this, the execution resumes from the saved update-in-progress point (see the source file \( filter.asm \)).

For the time left within the small frame (figure 2.1.1) the DSP executes another part of the update routine.
These last internal housekeeping actions (synchronization and restoring) were not indicated on figure 2.2.1, being not relevant for the algorithm flow.

The return to the original sampling rate (desegmenting $r^B[kB]$ into $r[k\text{-delay}]$), as well as the collecting of the input signals samples (segmenting into the input vectors), are carried out by the interrupt service routine (see section 2.4).

Note 1: as specified in appendix 5.1.c, the fastest external to external memory transfer is realized via the internal memory, but this has to be implemented in two loops, which is not a faster variant for $B \leq 8$.

Note 2: because anyway the external bus interface is permanently busy after the execution of the IFFT routine, the DMA controller cannot perform a "background" restoring of the (update) on-chip memory area left not restored; the CPU must do it, in "foreground". Thus, the copy sequence which follows to the IFFT call is superfluous. I realized this when the project development was being taken over by the next graduating student. The overload is $20+\% \cdot M$ clocks (around 0.37% for the mentioned parameter set example).

Note 3: the restoring of the (update) on-chip memory area left not restored can be postponed until after reenabling the interrupts - this is again something missed before the project development was taken over. In this case, the answer comes one sample faster, only if the (new) interrupts disabling moment falls just before receiving an interrupt (thus for some specific parameter sets). Therefore the processing delay can be decreased by one sample-time (125\,\mu s, for 8kHz sampling frequency) for certain parameter sets, by shifting this code portion of $10+M$ clocks long (4.44\,\mu s for the section 5.2 parameter set example). This is not the case for the above mentioned example, where the (old/new) moment falls close to the middle time distance between the fifth and the sixth sample.

Note 4: if this shifting of one sample is too large, in the case that such a parameter set exists, a better solution is to disable the interrupts only until the oldest sample subtraction takes place (the code must be modified). I noticed this during the update routine report writing. For this new solution, the possible maximum shifting would be 12 clocks (0.72\,\mu s - this means 0.67% shift, for 8kHz sampling frequency). Once again, this situation (the shifting) might occur only for certain parameter sets - but not for the one exemplified in section 5.2.
2.3 The update-part routine

This routine is in charge with computing the gradient estimate needed to update the adaptive weights. The update is performed when the double talk-flag is not set.

As mentioned in section 1.2.1, this gradient is proportional with the crosscorrelation between the residual signal and the input signal; the residual signal spectrum is first normalized, for each frequency bin, with an inverse power estimate (for the corresponding bin) of the input signal. As detailed in sections 1.2.2.3 and 1.2.3.2, the inverse power estimation, the spectrum normalization, and the partitioned crosscorrelation take place in the frequency domain, while the adaptive coefficients are updated in the time domain.

The general program and data flow of the update routine implementation on the C30 is sketched on figure 2.3.1. We distinguish: the above mentioned main operations, the transforms between the time and the frequency domain, and the "technical housekeeping" involved in the assembly language algorithm translation. The "clock" for this part "beats" each $A$ samples.

At the beginning, the (update) overlapped input and residual signal vectors $\mathbf{x}^L[kA]$ and $\mathbf{r}^A[kA]$ are saved into buffers $XL2$ and $RL$ (figure 2.3.1), since the buffers $XL1$ and $RB$ are meanwhile modified by the filter routine executions (see section 2.2). Because all these buffers are located within the external memory, as pointed out in appendix 5.1.c the saving must be done by the CPU, via the internal memory. With the same notations from section 1.2.3.2, the sizes of the circular buffers $XL1$ and $RB$ are $L$, respectively $A$. Again, the circular addressing radius being different ($A < L$), more than one loop had to be implemented.

The internal RAM receives firstly the vector $\mathbf{r}^A[kA]$ and secondly the oldest $A$ samples ($\mathbf{x}^A[kA]$) of the vector $\mathbf{x}^L[kA]$ (external read in parallel with internal write, $rpts$ avoided - as can be seen in appendix 5.1.c.7). Thirdly, the vector $\mathbf{r}^A[kA]$ is inserted into the buffer $RL$ and the rest ($\mathbf{x}^{L-A}[kA]$) of the vector $\mathbf{x}^L[kA]$ is saved into the internal RAM (one loop, two external and two internal accesses in three clocks - see appendix 5.1.c.8). It had to be taken into account that $\mathbf{r}^A[kA]$ and $\mathbf{x}^{L-A}[kA]$ could be of different lengths (see the source file update.asm).

After this, the vector $\mathbf{x}^L[kA]$ is moved to buffer $XL2$ (cannot use parallel instructions because of the two clocks external write - see appendix 5.1.c.9).

If the time constraint for this piece of code (saving $\mathbf{x}^L[kA]$ and $\mathbf{r}^A[kA]$) is tough (depending on the chosen parameter set, the filter routine execution can last close to $B-1$ samples, and thus each update routine slice can dispose of around one sample to go), it has to be replaced with the variant which saves $B$ by $B$ samples from $RB$ and $XL1$. The variant is globally slower, but meets the tough sliced time constraint requirements (see the source file variant.upd).

This up-to-here portion, although mentioned on figure 2.3.1, belongs to the coupling-part between the filter routine and the update routine.
Figure 2.3.1: Partitioned update
The update routine implementation itself starts by transforming the vector \( \mathbf{r}[kA] \) appended with \( 0^{L-A} \) to the frequency domain. There is a bit-reversal from the buffer \( RL \) to the internal RAM (the same loop as in appendix 5.1.c.2), followed by an FFT parameters initialization and routine call. The frequency domain residual signal vector \( \mathbf{R}[kA] \) is then saved into the buffer \( YL \) as shown on figure 2.3.1 (the loop in appendix 5.1.c.9 is identical). The same happens with the vector \( \mathbf{x}[kA] \): bit-reversal and transformation to the frequency domain (see the source file update.asm). The result \( \mathbf{X}_o[kA] \) has to be conjugated in order to be inserted in the delay line, as discussed in section 1.2.3.2, but this is done later on, before the loop which performs the partitioned crosscorrelation (actually it does not matter if now or then).

Next is computed in the buffer \( PL \) (figure 2.3.1) the inverse power estimate of the (finite length) input signal vector, according to the equation [1.2.51] in section 1.2.2.3, for \( i=0, ..., L-1 \):

\[
P_{L-1}[(k+1)A] \cdot L^{-1} = (1 + \beta) (P_{L}[kA])^{-1} - \frac{\beta}{2\alpha L} (P_{L}[kA])^{-2} (X_{L}[kA])^{-2}
\]

With \( c_1 = 1 + \beta \), \( c_2 = -\beta / 2\alpha L \), the vector \((P_{L}[kA])^{-1}\) inside the buffer \( PL \), finally overwritten by \((P_{L-1}[k+1A])^{-1}\), and the vector \( \mathbf{X}_o[kA] \) within the internal memory area denoted on same figure by \( ram01 \) (all the buffers names suggest only the lowest address), the above written equation comes to a form closer to the C30 implementation:

\[
PL_i = c_1 \cdot PL_i + c_2 \cdot (PL_i)^2 \cdot |ram01|^2.
\]

Due to the symmetry of the complex-valued vector \( \mathbf{X}_o[kA] \), only half plus one elements of the real-valued vector \((P_{L}[kA])^{-1}\) have to be computed. \( PL_0 \) and \( PL_{L/2} \) are calculated first, followed by a loop for \( PL_1, ..., PL_{L/2-1} \) (see appendix 5.1.i), where the six multiplications and two additions plus the involved loads/stores are coded in seven instructions and executed in seven clocks. No DMA controller job being in progress, it was not difficult to lay without conflicts the three external memory accesses (two of one clock and one of two clocks) among the other five internal ones, with or without computations. A better solution is not possible, because of the restricted C30 parallel instructions set possibilities.

Then, it takes place the normalization of the residual signal spectrum (buffer \( YL \)) with this input signal inverse power estimate (buffer \( PL \)). Both vectors are located in the external memory, which globally is a faster variant, because the vector \( \mathbf{X}_o[kA] \) is later needed in the internal memory (its maximal length equals the internal RAM block size). The element-wise multiplication between the symmetrical complex-valued vector \( \mathbf{R}[kA] \) in \( YL \) and the real-valued vector \((P_{L}[kA])^{-1}\) in \( PL \) could be as well implemented in one two instructions loop (see appendix 5.1.j.1) executed in five clocks, but the chosen variant was a four clocks five instructions loop (see appendix 5.1.j.2).

The conjugation that follows is coded in a loop with parallel instructions (the on-chip memory can hold two accesses per clock) yielding \((\mathbf{X}_o[kA])^*\)- see appendix 5.1.c.10.

Next is implemented the correlation, divided in \( N/Z \) partitions (branches).

The first branch begins by (circularly) inserting the conjugated frequency domain input signal vector \((\mathbf{X}_o[kA])^*\) into the delay line, denoted \( LO \) on figure 2.3.1; the value inside each box (buffer) gives the time reference (upside is the most recent overlapped partition), in \( \Delta \) samples-time units \( \Delta \cdot T_s = T_A \). The loop shown in appendix 5.1.c.3 is identical.
The rest of the branches begin by copying into the on-chip memory the delayed vector 
\((X^I[kA])^* = (X^I_0(k-izA))^*\), \(i=1...NIZ-1\), accessed by stepping each \(z\) buffers within the delay line \(LO\) (the same notations from section 1.2.3.2). The loop is the same with the one mentioned in appendix 5.1.c.7. This data transfer task cannot be accomplished "in background" by the DMA controller during the previous partition processing, because there is not enough room within the on-chip memory to hold two partitions at once (the partitions maximal size \(L\) equals one internal RAM block size, 1024; the other internal RAM block is partially occupied by the 512 twiddle factors and 179 up-to-now declared variables).

Each branch continues by an element-wise product between two (symmetrical) complex-valued vectors, the frequency domain (delayed) input signal vector \((X^I[kA])^*\) (in \(\text{ram0I}\)) and the normalized residual signal spectrum vector \(Y^I[kA]\) (in \(YL\)), this being in fact the partitioned correlation. Due to the frequency domain vectors symmetry (caused by the time domain vectors real-valued nature), only half plus one elements have to be computed. The first and the last have only real-component, requiring thus one multiplication. The rest have real- and imaginary-component, implying four multiplications and two additions; these have been implemented in a five instructions loop, executed in six clocks (see appendix 5.1.k), less than six being impossible for the needed seven internal and two external memory accesses. Each moreover conflict on this correlation inner loop causes a \((L/M) \cdot (Q/Z) \cdot (B/A)\) times less critical overload than one on the convolution inner loop (see section 2.2), but nevertheless the best solution had to be found. Here the limitations proceed from the C30 restricted parallel instructions set, which fits closely the specific CPU hardwired architecture.

Next, the IFFT routine transforms to the time domain the gradient estimate vector partition. The calling parameters were earlier initialized; to obtain the best execution speed, both data and twiddle factors are inside the on-chip RAM; the computation is in-place; the output is bit-reversed ordered.

The bit-reversal sequence which follows stores in normal order the elements of the time domain gradient estimate vector partition, in the buffer \(LB\) on figure 2.3.1 (the bit-reversal cannot be performed in-place). The loop execution lasts two clocks, comprising one external write (see appendix 5.1.c.11).

In the end, the double talk-flag is tested, and, if it is not set, the adaptive coefficients partition on that branch (contained in the buffer \(WO\) on figure 2.3.1 - here the value inside each box points out which partition is) is mirrored added with the oldest \(Z\) samples from this cyclic correlation result. The loop execution lasts five clocks, because of the two clocks conflict between external read and external write (see appendix 5.1.1.1). See also note, at the end of this section.

The outer loop schedules each input signal vector partition to be like this correlated with the normalized residual signal vector, filling in the buffer \(WO\) with the (next) adaptive coefficients vector. For a correlation processor load example is referred section 5.2.

The implementation code is continued by transforming to the frequency domain the (next) adaptive coefficients vector partitions, with changing the size, as they are needed by the filter routine (the partitionings are decoupled).

The same style as in convolution is employed, with separate code portions for the first partition transformation, the middle \((N/Q-2)\) ones, and the last one. The resemblance is given by the DMA controller use to copy on-chip the next partition to be processed (see also note, at the end of this section).
To begin, the FFT routine calling parameters (excepting the data buffer address) and some technical staff (pointers and counters) are initialized; the DMA controller receives the job to copy on-chip the second partition (see the source file `update.asm`).

The first time domain adaptive coefficients vector partition (all the partitions are contained in the buffer $W_0$, figure 2.3.2 - the same $W_0$ from figure 2.3.1) is mirrored copied by the CPU into the upper half (denoted by $\text{intram01m/03m}$ on figure 2.3.2) of one of the two on-chip memory areas alternatively (from one partition to another) pointed for this purpose (see the source file `update.asm`). The coded loop execution timing is the same with the one presented in appendix 5.1.c.7 - external read in parallel with internal write.

It is now to be noticed that, the previous buffer $W_0$ filling in and this same buffer reading out are mentioned in section 1.2.3.3 under the name of “interface for the adaptive weights”, while the $r^B[kB]$ answer writing into the buffer $RB$ at the end of the filter routine execution plus the $r^A[kA]$ reading out from the same buffer at the beginning of the update routine execution are referred in the same section as “interface for the residual signal with $A=\lambda \cdot B$”.

Then, the last $M-Q$ locations in both areas $\text{intram01m}$ and $\text{intram03m}$ are zero-initialized (this is the appending with $0^{M-Q}$ - the parallel internal stores shown in appendix 5.1.c.12. It is enough to do this once, since these locations are not further altered within the update routine execution.

It follows a bit-reversed storing of the current time domain adaptive coefficients vector partition into the lower half (denoted $\text{intram01/03}$ on figure 2.3.2) of one of the mentioned alternating areas. The loop in appendix 5.1.c.4 is identical.

Figure 2.3.2: Coupling the filter and the update - the adaptive coefficients
After this, the FFT routine is invoked to perform the partition transformation to the frequency domain. To get the best performance, it is arranged to work only with internal resources. The transformation is in-place.

To conclude the first partition processing, the result (the first frequency domain adaptive coefficients vector partition) is stored into the buffer $S0$ (figure 2.3.2). The two clocks two instructions loop is identical with the one in appendix 5.1.c.9. The buffer $F0$ on same figure holds the current (big-frame) adaptive weights, used by convolution, while $S0$ collects the next ones. For these two buffers, the value inside each box shows which partition is.

Next is coded on a loop the transformation to the frequency domain of the middle ($N/Q-2$) adaptive weights vector partitions. There is almost the same sequence of actions as for the first partition transformation (see the source file update.asm): initializations of counters and pointers, a DMA controller job completion test (to ensure over the availability of the current partition), a new job assignment for the DMA controller (copy the next partition), then the bit-reversal followed by the FFT routine call, to obtain the frequency domain adaptive weights vector partition, and finally the result collecting into the buffer $S0$.

The loop misses, in respect with the actions performed for the first partition transformation, the I/FFT routine parameters initialization (they are unmodified even if a filter routine "interrupts"), the CPU on-chip current partition mirrored copying (it is done by the DMA controller), and the $0^{M-Q}$ appending (the locations are preserved).

The filter routine can "interrupt" the update routine execution in any portion of code, depending on the parameter set the adaptive filter is running with. For this reason the filter routine must save and restore not only the CPU status and the altered (on-chip) memory area, but also the DMA controller status.

The described loop is less than 64 instruction words long, but the code will not be within the cache because of the FFT routine call. Although the speed of this loop execution is not critical, the conflicts were minimized.

The code sequence that computes the last frequency domain adaptive coefficients vector partition is identical to the one inside the previous loop, excepting that there is no DMA job to reinitiate (see the source file update.asm).

For the adaptive coefficients transformation execution timing is referred section 5.2.

When the (next) frequency domain adaptive weights vector is available, the update routine execution becomes idle, waiting for the last small frame (the frame types were defined on figure 2.1.1).

The small frames-counter incremented at the end of each filter (see section 2.2) is now tested (see the source file update.asm). For the section 5.2 parameter set example, the C30 is idle around 83000 clocks, which means a zero-load of 8% (see figure 5.2.1). In this "gap" it is supposed to be inserted a double talk evaluator, for the future developments of the algorithm implementation.

After all the $A/B$ filter routines are performed, the pointers to the buffers holding the current ($F0$) and the next ($S0$) frequency domain adaptive weights vectors are interswitched (figure 2.3.2). See also figure 5.2.1.
In the end, it is read the memory location where the PC writes a certain stop-value if the adaptive filter execution must be halted (see the source file *update.asm*). If the stop-value is found, the execution branches outside the adaptive filter code. If the stop-value is not present, the small frames-counter is again tested, until it indicates the first small frame (within a new big frame), and then the execution resumes from the beginning of the *update* routine (see the source file *update.asm*).

The actions after the buffers (pointers) interswitching were not mentioned on figure 2.3.2, being transparent for the algorithm flow.

---

**Note 1:** a (Z clocks) faster variant can be obtained by implementing the coefficients partition update in two loops, via internal memory (see also appendix 5.1.1.2). I saw it during this report piece writing. The gain is around 0.05% for the section 5.2 parameter set example.

**Note 2:** the *update* speed loss caused by not using the DMA controller would be \((N-7\cdot(N/Q)+13)\) clocks (around 1800 clocks - or 0.18% overload - for the section 5.2 example). This remark is given because the graduating student who took over the project development expressed his doubts concerning the (*update*) DMA controller job interruption and resumption, caused by the *filter* routine occurrences. If this task is switched to the CPU, there is no DMA controller job to be accomplished during the *update* routine execution, and thus the DMA controller status saving and restoring within the *filter* routine execution become superfluous. Discarding this status saving-restoring leads to a *filter* speed gain of approximately (see appendix 5.1.m) \(34\cdot(A/B)\) clocks (around 2100 clocks - or 0.21% less load - for the mentioned parameter set example). Thus, depending on the chosen parameter set, the overall execution speed might lose or gain.
2.4 The interrupt service routine

As explained in section 2.1, the interrupt service routine must handle the I/O requests as well as carry out the global code synchronization (the *update* routine execution splitting among the *filter* routine executions - see figure 2.1.1).

Briefly, the things unfold like this: at a rate equal to the sampling period, the C30 *timer1* issues to the converters on the DSP board the Start-Of-Conversion signal (see section 1.3); this unleashes the quantization of the incoming analog samples to the 16-bit format (integer, for the C30) and the conversion of the outgoing sample to the analog format (in the range of -3V...+3V); when the conversions finish, it is generated the End-Of-Conversion signal, which goes to the C30 *interrupt1 request* pin; this does not affect the DMA controller activity (it was so initialized - see section 2.5); when the CPU can accept the interrupt (for instance not if there is a conflict in the pipeline), the execution branches to the interrupt service routine. The other interrupt on the board (*interrupt3*) services the PC-DSP communication. It will not be commented here. It is mentioned in section 5.4.

Since the interrupted sequence of instructions must resume without disturbance, this routine firstly saves (to restore at the end) any supposed to be modified item: the flags and a few registers. Because the interrupt intervenes very often during the adaptive filter execution (see figure 2.1.1), the fastest code variant had to be found. Due to the pipelined implementation of instructions execution, which can generate conflicts, the "fastest variant" does not necessarily imply the minimum number of registers used (although those used have to be saved/restored).

First it is to be established what actions must be carried out, and then to try the possible variants. Besides the mentioned saving/restoring, there are two samples to be taken over from the A/D converters and (circularly) inserted into the buffers, one sample to be (circularly) extracted from the buffer and written to one D/A converter, plus the synchronization function to be accomplished.

The saving/restoring of an *extended-precision* (40-bit) register takes twice longer than for the other ones, but at least one is needed for the data type conversions (the program works with floating-point data, while the converters manipulate integer data). To use a second one means a speed decrease: the input sample writing to the buffer takes three clocks (external write after fetch and followed by fetch), while the saving/restoring lasts four clocks.

The *block size* register and at least one *auxiliary* register are necessary to address the buffers as well as the converters. As shown in appendix 5.1.n.1, this formula is not the fastest one, since the loading (or the reading) of the *auxiliary* register comes too often in conflict with the indirect addressing (an address that should be earlier ready in the pipeline, is generated only after the loading/reading), because the instructions cannot be distanced among other ones (the whole routine comprising just a few instructions).
A faster solution uses two auxiliary registers, one to address the buffers and the other one for the converters (see appendix 5.1.n.2). Anyhow the instructions would be sequenced, the conflicts amount to minimum 17 clocks: two two waits external reads (three clocks delay, being over the expansion bus interface), two external writes after fetches ans followed by fetches (three clocks delay), two register conflicts (two clocks delay), and one external read/no fetch (one clock delay).

The variants using three or four auxiliary registers are slower, as they involve the same conflicts (because of the block size register modifying), plus the saving/restoring of each moreover altered register.

If absolute addressing is used for the converters (the buffers need compulsory indirect addressing), the execution time is increased by the data page register modifying generated conflicts and by the data page register saving/restoring.

Thus, the chosen solution makes use of two auxiliary registers and the block size register for the indirect addressing, plus an extended-precision register for the data manipulation.

As described in appendix 5.1.n.2, after the necessary savings, the far-end input signal sample $x[k]$ is read from the A/D converter, converted to floating-point format, inserted into the buffer $XL$, then the near-end input signal sample $e[k]$ is read from the A/D converter, converted to floating-point format, inserted into the buffer $D$, and next the residual signal sample $r[k\text{-delay}]$ is extracted from the buffer $RB$, converted to integer format, and sent to the D/A converter (figure 2.4.1).

![Figure 2.4.1: Input/Output during the interrupt service routine](image)
At this point, it is performed the synchronization function: the interrupt occurrences-counter is incremented and tested, in order to decide whether the execution resumes from the interruption point or branches to the beginning of the filter routine (as shown on figure 2.1.1). Two clocks are spared by implementing a delayed branch (see appendix 5.1.n.2 and also note, at the end of this section).

For the interrupts $1, ..., B-1$ (within a small frame) the altered registers are restored and the execution resumes from the top of the stack pointed instruction (see appendix 5.1.n.2).

The $B$th interrupt servicing ends by saving the (update) CPU registers (excepting those in-between unmodified by the filter routine and the interrupts), cancelling the repeat mode (if any), storing as the update-in-progress point the address from the top of the stack, and branching to the filter routine beginning (see appendix 5.1.n.4). The rest of the update routine context is saved at the beginning of the filter routine execution and the complete update routine context is restored at the end of the filter routine execution (see section 2.3). This is the way the nested on blocks of samples processing is implemented.

There is a detail to be mentioned, namely that the filter routine and the interrupt service routine keep separate pointers to address the residual signal buffer $RB$ (see the source files filter.asm and dpbfdaf.asm).

As depicted on figure 2.4.2, the difference between the two pointers is connected with the filter routine execution time, in samples.

![Figure 2.4.2: The accessing of the residual signal buffer by the filter routine and the interrupt service routine](image)

Figure 2.4.2: The accessing of the residual signal buffer by the filter routine and the interrupt service routine
The interrupt routine execution is slower with two or four clocks if there is a DMA controller job in progress implying external reads, respectively external writes. Without considering this possibility, the interrupt service routine causes a processor load of around 3.2% for the section 5.2 parameter set example: the interrupts 1,...,B-1 last 62 clocks each, while the Bth one lasts 99 clocks (as shown in appendix 5.1.n.2, respectively 5.1.n.4). One moreover clock delay means here around 0.05% overload, for the considered parameter set example.

The functioning of the on-board analog interface does not need the interrupt acknowledge signal, thus the iack instruction is not necessary, as specified in appendix 5.1.n.2.

The first implementation code versions included a "filter active" flag (set during the filter routine execution) tested by the interrupt service routine, to determine whether the filter routine or the update routine context is to be saved. After a closer look, this idea proved itself to be useless and leading to a slower implementation.

Note: another clock would be spared by first testing and then incrementing the interrupt occurrences-counter (see appendix 5.1.n.3). I realized this during the report writing preparations.
2.5 The main program

The implementation code execution starts by disabling the interrupts and performing some initializations which bring the system to the desired status (see the source file dpbfda.asm). The data page and the stack pointer registers are set to point the data page where are located the 179 (up to this development stage) implementation code variables, respectively the bottom of the stack reserved area.

The cache memory utilization and the overflow mode are enabled. The external bus interfaces are initialized to work with zero software wait states and external generated ready, while the DMA controller is reset.

The pointers to the circular buffers, as well as the pointers to and the limits of the linear buffers are given valid values. The buffer D size (figure 2.2.2) is for the time being set to cause the maximal processing delay (B samples), and accordingly is initialized the difference between the two RB buffer pointers (figure 2.4.2).

The C30 timer1 is started to trigger the converters at a rate equal to the sampling period, and the End-Of-Conversion signal is enabled to interrupt the CPU activity (no influence over the DMA controller activity). See also note1 at the end of this section.

The FFT routine calling parameters are given some valid values.

At this point, it is measured the processing delay encountered when filtering a block of samples with the adaptive filter being configured by that specific set of parameters. As sketched on figure 2.5.1, we distinguish, after the block of B samples collecting, near to the end of the filter routine running, a distinct moment when the residual signal vector $\mathbf{r}[kB]$ is available; the first answer sample $r[k\cdot\text{delay}]$ comes out with the next interrupt.

Up to here it is defined the (algorithm) processing delay. The filter routine execution might end before or after this interrupt (it makes no difference - see section 2.2).

This interrupt service routine writes the sample $r[k\cdot\text{delay}]$ into the D/A converter shift register, but the conversion itself is started by the next timer1 signalization, thus the sample release in the analog format happens just before the next interrupt.

Until this moment we have to do with the (physical) total delay, which is very close to one sampling period more than the processing delay (a few 60ns clocks approximation).

In order to get the processing delay measured, the C30 waits for an interrupt (see the source file dpbfda.asm). Because the interrupt occurrences-counter was previously set to indicate the end of a small frame, the execution will branch, after the interrupt routine, to a filter routine (see section 2.4). This is identical to a normal filter routine, up to the point where the interrupts are disabled and the residual signal vector $\mathbf{r}[kB]$ is computed (see the source file filter0.asm). Then the execution returns to the initializations, with the interrupts being still disabled, and the interrupt occurrences-counter carrying the information concerning the processing delay.
As specified in section 2.2, this delay is set to be the same for all filter routine executions running with a specific parameter set. For the section 5.2 parameter set example, the processing delay is 1.625msec (13 samples) and the total delay is 1.750msec (14 samples).

According to this information are reinitialized: the buffer \( D \) size, for a correct subtraction of the echo estimate from the echo combined with the near-end side signal (figure 2.2.2), and the difference between the pointers addressing the residual signal buffer \( RB \) (figure 2.4.2). Both the interrupt occurrences- and the small frames-counter are reinitialized to indicate the beginning of a big frame, while the return address at the end of the \( B \)th interrupt is switched from the filter0 routine start to the (normal) filter routine start. The C30 timer1 is reset and restarted, the End-Of-Conversion signal is reenabled to interrupt the CPU activity, and the \( \text{FFT} \) calling parameters are reinitialized with some valid values (see also note 2 at the end of this section).

Finally, the C30 waits for an interrupt. After its routine execution, the complete adaptive filter code execution starts (see the source file \( dpbfda.fasm \)). First is executed a filter routine, then starts the update routine execution, "interrupted" by the subsequent filter routine executions (as depicted on figure 2.1.1). The adaptive filter code execution can be stopped at the end of a big frame (for the section 5.2 parameter set example a big frame execution lasts around 0.63sec) if a certain stop-value written by the PC is found in the PC-DSP communication area.
For an exhaustive adaptive filter code run timings example is referred section 5.2.

Concerning the parameter set acceptance (validation), the rest of the implementation code variables calculation, and the memory data/code buffers allocation (all of them depend on the specific parameter set the adaptive filter code must run with), is referred section 5.3. See also note_3 at the end of this section.

Note_1: while testing the real-time implementation with one of the simulators (see section 5.4), which cannot simulate the converters functioning, the timer_1 signal itself is enabled to interrupt the CPU activity. In this case the adaptive filter code execution timing is shifted in respect with the C30 timer signals. This has no significance.

Note_2: for instance a wrong set of FFT routine calling parameters causes the internal RAM block_1 to be entirely occupied by the FFT routine data buffer. In this case all the implementation code variables (counters, pointers, and so on) are overwritten, which disturbs the program execution.

Note_3: up to this stage, the complete implementation code comprises (without variants) 1059 assembly instructions and has 179 declared variables. A typical DMA controller job initiation mistake and few other disconnected mistakes (one in the convolution routine, three in the update routine, and one at the initializations) were observed and corrected by the next graduating student, during the period he took over the project development.
3 Conclusions and recommendations

An implementation of the Decoupled Partitioned Block Frequency Domain Adaptive Filter algorithm [6] on one 40 bits floating-point DSP operating from a 60ns clock has been conceived for any valid parameter set, and practically led up to a certain point (here subsequently shown) for the parameter set specified in section 5.2.

Because the algorithm is not simple, the code is written in assembly language, and the size of the on-chip RAM puts serious restrictions for the execution speed, I considered as first priority to see which is the arrangement for the data buffers within the different memory types together with the code execution flow, that yields the fastest correct implementation.
I estimated that, doing this at the beginning, makes the overall development duration shorter than just start coding and (possibly) modifying backwards for each new added piece of code.

The next step was to code the decided arrangement.
Firstly were piece by piece coded the filter-part and the update-part. Here were tested the execution speed (the pieces were recoded until the fastest variant was found) and the exactness of the accessing within the specified address bounds for each data buffer (most of the data accesses are of circular or bit-reversed types).
After the filter-part and the update-part were that far tested, the interrupt service routine was coded to link them and to ensure the communication with the outside world. Besides the execution speed, the problem tested here was the correct execution flow of the overall implementation, as we have to deal (in assembly language) with a larger routine, comprising loops and calls for subroutines, interrupted by a smaller routine, having loops and calls for the same subroutines (but possibly with other calling parameters), and both of them being interrupted by the samples flow. When the code of such a nested implementation, supposed to run with any valid parameter set, expands to 1059 assembly language instructions, it is not at hand to follow all the details. Because of the execution speed constraint, the way of working is not high level languages-like, where we do not have to care about altering the registers, the stack, and so on.
Finally, the main program was coded.
The correctness of the computations was not tested before I handed over the project development.

The FFT/IFFT routines were taken over from the former development stage.
The following graduating student took over my whole contribution, and, regarding it, he expressed as negative remark that the implementation code did not work correctly from the first moment, because of the mistakes mentioned in note 3 of section 2.5, tracked down and mended by himself, during the taking over period.
Technical details concerning my contribution were described in chapter 2. For the parameter set specified in section 5.2, the total answer delay is 1.750msec, framing for instance within the standards required by the standard telephony networks. nevertheless, the adaptive filter can run with any valid parameter set, the implementation being conceived not dependent on the parameter set. This means that for any valid values of the parameters, the code execution flow is correct, unfolding like depicted in figures 2.1.1, 2.2.1, 2.3.1, 2.3.2, 2.4.1, 2.4.2, and the echo estimate is subtracted from the echo (figure 2.2.2) with a delay measured by the code itself during the initializations (figure 2.5.1).

The implementation code lacks at this development stage some initialization parts, to be able to run easily with any valid set of parameters:
1. the conditions checking over the parameters configurating the adaptive filter execution.
   See appendix 5.3.a.
2. the calculation of the implementation code variables derived from these parameters.
   See appendix 5.3.b.
3. the buffers allocation within the available memory map. See appendix 5.3.c.

After the tests stage (this is an entire other work to do, needing also a DAT recorder-hard disk interface), meant to settle the final form of the update equation and to solve other problems possible to show up, the application (the C30 board with the program) becomes standalone (not dependent on a PC).

Because the code is written in assembly language, modifications (if necessary) should be done carefully (with patience).
He who needs to code in assembly language is advised, after becoming familiar with [16], to take a look at [15], where missing details are very well pointed out.
See also the notes at the end of this section.

**Note:** A C30 variant with on-chip E(2lpROM (TMS320E30), if available on the market (I did not get this information from Koning en Hartman or from Texas Instruments Holland BV), allows the total convolution partitions length to be chosen up to M = 512 and the update routine either to be speeded up by using both on-chip RAM blocks for computations or to be used in this variant with the correlation partitions length up to L = 2048 (concerning these restrictions see appendix 5.3.a).

**Note:** Running the adaptive filter code exactly in this version on a C40 leads to a better performance (the C40 operates from a 40ns clock). The total answer delay for the section 5.2 parameter set example would be only two samples less (this means 1.500msec), but it will be possible to enlarge the parameter sets lot for which the processor load is less than 100%. Yet the implementation can be improved only with minor changes, and the allowable parameters ranges can be extended, as the C40 has 8k words on-chip RAM. One DMA channel can be used by the filter routine and another one by the update routine.
4 References


[19] Texas Instruments, "TMS320 Floating point DSP assembly language tools user's guide", August 1988, literature number SPRU035A


[21] Texas Instruments, ftp-site DSP public domain files on ti.com in /mirrors/tms320bbs

[22] Texas Instruments, "TMS320C50 C source debugger user's guide", August 1990, 2547297-9721

5 Appendixes

5.1 Other remarks; relevant parts of code

In this section will be presented the traced execution of some code parts.
As mentioned in section 1.3, the C30 has a four-stages pipelined implementation of the
instructions execution, namely Fetch - Decode - Read - Execute. This is shown by the letters
F D R E above the pipeline maps. For each new entrance in the pipeline, the simulator's
trace function assigns another letter, a b c or d. An "v" denotes that the corresponding
pipeline stage performs no operation during the concerned clock (because of a conflict).
Other remarks will be given wherever are not enough explanations within the source files.
The off-chip memory has zero-wait states, unless stated otherwise.

5.1.1: timing equalization at the fibe routine entrance

5.1.1.1: for the case the DMA controller was not busy

<table>
<thead>
<tr>
<th>F</th>
<th>D</th>
<th>R</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: LDI</td>
</tr>
<tr>
<td>2</td>
<td>clock</td>
<td>pipeline</td>
<td>= [d:a:b:c: LDI</td>
</tr>
<tr>
<td>3</td>
<td>clock</td>
<td>pipeline</td>
<td>= [c:d:a:b: LDI</td>
</tr>
<tr>
<td>4</td>
<td>clock</td>
<td>pipeline</td>
<td>= [b:c:d:a: CMP</td>
</tr>
<tr>
<td>5</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMPI</td>
</tr>
<tr>
<td>6</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>7</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>8</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>9</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>10</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>11</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>12</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>13</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>14</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>15</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>16</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>17</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>18</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>19</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>20</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>21</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>22</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
</tbody>
</table>

5.1.1.2: for the case the DMA controller was busy

<table>
<thead>
<tr>
<th>F</th>
<th>D</th>
<th>R</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: LDI</td>
</tr>
<tr>
<td>2</td>
<td>clock</td>
<td>pipeline</td>
<td>= [d:a:b:c: LDI</td>
</tr>
<tr>
<td>3</td>
<td>clock</td>
<td>pipeline</td>
<td>= [c:d:a:b: LDI</td>
</tr>
<tr>
<td>4</td>
<td>clock</td>
<td>pipeline</td>
<td>= [b:c:d:a: CMP</td>
</tr>
<tr>
<td>5</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMPI</td>
</tr>
<tr>
<td>6</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>7</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>8</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>9</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>10</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>11</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
<tr>
<td>12</td>
<td>clock</td>
<td>pipeline</td>
<td>= [a:b:c:d: CMP</td>
</tr>
</tbody>
</table>

52
5.1.b: typical DMA controller job initiation sequence

A DMA controller transfer comprises a read from the location pointed out by the source address register and a write to the location indicated by the destination address register. The DMA controller global control register establishes whether or not:

- one or both accesses are synchronized by the enabled interrupts received by the DMA controller;
- these addresses are modified after each access;
- the transfers continue and/or an interrupt is issued upon the completion of a block transfer.

```plaintext
ldi @dma, r5 ; load the DMA controller global control register address
ldi @cntl3, r7 ; load the global control stop transfers value
ldi @cntl3, r3 ; load the number of transfers to be done
ldi @src,R5 ; load the transfers (base) source address
ldi @dst,R4 ; load the transfers (base) destination address
ldi @glblrd,R5 ; load the global control transfer mode value
sti r7,*r5 ; stop the current transfers (see also note, at the end of section 2.5)
sti r5,*r5(4) ; initialize the transfers (base) source address
sti r4,*r5(6) ; initialize the transfers (base) destination address
sti r3,*r5(8) ; initialize the transfer counter
sti r6,*r5 ; initialize the transfer mode and start the transfers
```

# For direct addressing the data page register must be properly loaded before each access - if all the variables are not located within the same data page.

5.1.c: different types of data transfers

# Clocking of the memory accesses

Two on-chip memory CPU accesses can be performed each clock. One off-chip zero-wait states memory CPU read can occur each clock, while an off-chip zero-wait states memory CPU write lasts for two clocks, unless it does not follow a read via the same bus interface, in which case it takes three clocks.

The DMA controller on-chip accesses are single-clock, but only one per clock (there is only one DMA address bus); the DMA controller off-chip zero-wait states reads last for two clocks, while the DMA controller off-chip write is identical to the CPU one.

When transferring between closed moments of time a block of data within the off-chip memory, use cannot be made of the DMA controller because its read is one clock slower than the CPU read, and the external bus interface has only one pair of busses. The fastest variant is the CPU transfers the data block via the on-chip memory. This takes three times the block size, in clocks, for zero-wait states memory.

The data transfer examples cited within the chapter 2 do follow:

5.1.c.1:

```plaintext
<table>
<thead>
<tr>
<th>clock</th>
<th>pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F D R E</td>
</tr>
<tr>
<td>11</td>
<td>pipeline = [b][e][c][d]</td>
</tr>
<tr>
<td>12</td>
<td>pipeline = [c][b][d][e]</td>
</tr>
<tr>
<td>13</td>
<td>pipeline = [a][b][c][d]</td>
</tr>
<tr>
<td>14</td>
<td>pipeline = [a][b][c][d]</td>
</tr>
<tr>
<td>15</td>
<td>pipeline = [d][c][b][a]</td>
</tr>
<tr>
<td>16</td>
<td>pipeline = [a][d][c][b]</td>
</tr>
<tr>
<td>17</td>
<td>pipeline = [a][b][c][d]</td>
</tr>
<tr>
<td>18</td>
<td>pipeline = [a][b][c][d]</td>
</tr>
<tr>
<td>19</td>
<td>pipeline = [b][a][d][c]</td>
</tr>
<tr>
<td>20</td>
<td>pipeline = [c][b][a][d]</td>
</tr>
<tr>
<td>21</td>
<td>pipeline = [a][b][c][d]</td>
</tr>
<tr>
<td>22</td>
<td>pipeline = [a][b][c][d]</td>
</tr>
</tbody>
</table>
```

*external read cannot take place simultaneously with external write, being postponed 2 clocks because the decode unit "informs" the internal pipeline logic that nothing will advance in the pipeline, during the 2 extra clocks there is no fetch attempted (although it is made from the cache)*
5.1.c.2:  
**F D R E**

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>clock-b1: RPTB b1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>clock-b2: RPTB b2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>clock-b3: RPTB b3</td>
</tr>
</tbody>
</table>

5.1.c.3:  
**F D R E**

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clock-b1: RPTB b1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>clock-b2: RPTB b2</td>
</tr>
</tbody>
</table>

5.1.c.4:  
**F D R E**

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clock-b1: RPTB b1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>clock-b2: RPTB b2</td>
</tr>
</tbody>
</table>

5.1.c.5:  
**F D R E**

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clock-b1: RPTB b1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>clock-b2: RPTB b2</td>
</tr>
</tbody>
</table>

5.1.c.6:  
**F D R E**

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>clock-b1: RPTB b1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Clock</th>
<th>Pipeline Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>clock-b2: RPTB b2</td>
</tr>
</tbody>
</table>
clock = 29  pipeline = |c|a*;b; STF R4,*AR3--(1)
clock = 30  pipeline = |v|c;a;b;
  // wait-state external write (3 clocks), followed by internal read
  // the third clock of the external write is transparent for the CPU
  // during the 1 clock wait there is no fetch, although it is made from the cache - the pipeline is "informed" that this is not necessary

5.1.c.7:

<table>
<thead>
<tr>
<th>PIPELINE</th>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 1</td>
<td>pipeline =</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIPELINE</th>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 8</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 9</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 10</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 11</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 12</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 13</td>
<td>pipeline =</td>
</tr>
</tbody>
</table>
  // external write in parallel with internal read

5.1.c.8:

<table>
<thead>
<tr>
<th>PIPELINE</th>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 1</td>
<td>pipeline =</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIPELINE</th>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 17</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 18</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 19</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 20</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 21</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 22</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 23</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 24</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 25</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 26</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 27</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 28</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 29</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 30</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 31</td>
<td>pipeline =</td>
</tr>
</tbody>
</table>
  // external write, followed by external read, followed by internal read in parallel with internal write
  // because the external write lasts 2 clocks, the external read which follows is prolonged with 1 clock
  // two accesses being impossible to occur at once over the external bus interface
  // the variant with external write after read would have leaded to the same 2 extra clocks

5.1.c.9:

<table>
<thead>
<tr>
<th>PIPELINE</th>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 14</td>
<td>pipeline =</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIPELINE</th>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 14</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 15</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 16</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 17</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 18</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 19</td>
<td>pipeline =</td>
</tr>
</tbody>
</table>
  // internal read followed by external write
  // the 2 clocks write is transparent for the CPU

5.1.c.9:

<table>
<thead>
<tr>
<th>PIPELINE</th>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 14</td>
<td>pipeline =</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PIPELINE</th>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 9</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 10</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 11</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 12</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 13</td>
<td>pipeline =</td>
</tr>
<tr>
<td>clock = 14</td>
<td>pipeline =</td>
</tr>
</tbody>
</table>
  // internal read external write
  // during the second clock of the write the bus interface is not requested for another access

55
5.1.c.10:  

The pipeline is:

```
5.1.c.10:
  RPTB  conj
```

The pipeline is:

```
clock = 5  pipeline = [a;|c;b|;] RPTB  conj
```

5.1.c.11:  

The pipeline is:

```
5.1.c.11:
  RPTB  br3
```

The pipeline is:

```
clock = 5  pipeline = [b;|c;d|;] LDF  *AR2**(1),R2
```

5.1.c.12:  

The pipeline is:

```
5.1.c.12:
  RPTB  mov5
```

The pipeline is:

```
clock = 5  pipeline = [a;|b;d|;] STF  R7,**AR3**(1)  conj  STF  R7,**AR1**(1)
```

5.1.d: about circular addressing

# The circular addressing over a buffer of length N needs:

a) a block of memory beginning at an address multiple of \(2^m\), if \(N\) is a power of 2, or

b) a block of memory beginning at an address multiple of \(2^{m+1}\), otherwise;

b) the block-size register initialized with \(\log_2 N\);

c) an auxiliary register initialized to point within the buffer.

The circular addressing can be used with post-modify increment or decrement.

5.1.e: about bit-reversed addressing

# The bit-reversed addressing over a buffer of length \(N = 2^n\) needs:

a) a block of memory beginning at an address multiple of \(N\);

b) the \(i0\) index register initialized with \(\frac{N}{2}\); [16] states wrong \(i0 = -N\); in [15] it is correct;

c) an auxiliary register initialized to point at the beginning of the buffer.

The only bit-reversed addressing possibility is with post-modify increment.

5.1.f: pointing two alternating areas

```
biglip
```

```
subli  jr0
```

```
ldi  *ar7(i0),ar1  ;switch (each partition) the switch between 0 and 1
```

```
;ar7 holds a pointer to 2 consecutive pointers
;ar1 will point to intram01[s]witch], which means to intram01 or to intram01m
```

When nested loops are needed, the decrement-and-branch instruction is useful.
5.1.g: the convolution key-product

5.1.g.1: two loops, 7 + 7 clocks

F D R E

clock = 1  pipeline = [d:c:b:a] RPTB  first_half_product_i

clock = 15  pipeline = [b:a:d:]* ADDF  *AR2,R1
clock = 16  pipeline = [c:b:a:d] SUBF3 R0,R1,R1
clock = 17  pipeline = [d:c:b:a] STF  R1,*AR2++(1)
clock = 18  pipeline = [a:d:c:b] MPYF3  *AR1++(1),*AR0++(1),R1
clock = 19  pipeline = [b:a:d:c] MPYF3  *AR4--(1),*AR3--(1),R0
clock = 20  pipeline = [t:*]  *AR4--(1),*AR0--(1),R0
clock = 21  pipeline = [t:*]  *AR4--(1),*AR0--(1),R0
clock = 22  pipeline = [c:b:a:]* ADDF  *AR2,R1
clock = 23  pipeline = [d:c:b:a] SUBF3 R0,R1,R1
clock = 24  pipeline = [a:d:c:b] STF  R1,*AR2++(1)

the DMA controller performs external to internal memory transfers

The time sequencing of the accesses performed by the CPU and by the DMA controller, is:

DMA

- external (read) - has 2 clocks
- external (read) - has 2 clocks
- internal (write) - has 2 clocks
- internal (write) - has 2 clocks
- no access (the CPU has priority)

CPU

- CPU
- CPU
- CPU
- CPU
- CPU

5.1.g.2: one loop, 6 instructions, 7 clocks

F D R E

clock = 1  pipeline = [d:c:b:a] RPTB  product_i

clock = 23  pipeline = [a:d:c:]* ADDF  *AR6,R0,R0
clock = 24  pipeline = [b:a:d:c] LDF  *AR3--(1),*AR5, R4
clock = 25  pipeline = [c:b:a:d] MPYF3  *AR0,*AR1,R0
clock = 26  pipeline = [d:c:b:a] MPYF3  *AR4, R4,R1
clock = 27  pipeline = [a:d:c:b] MPYF3  *AR0++(1),*AR4--(1),R0
clock = 28  pipeline = [b:a:d:c] MPYF3  *AR1++(1),R4,R1

the same conditions and events time unfolding bold for this second loop

57
clock = 32  pipeline = |a|d|c|b|; MPYF3 *AR0,*AR1,R0  || ADDF3 R0,R1,R2
clock = 33  pipeline = |b|a|d|c|; MPYF3 *AR4,R4,R1  || STF R2,*AR6--(1)
clock = 34  pipeline = |c|b|a|d; MPYF3 *AR0++(1),*AR4--(1),R0  || SUBF3 R1,R0,R2
clock = 35  pipeline = |d|c|b|a; MPYF3 *AR1++(1),R4,R1  || ADDF3 *AR2,R2,R2
clock = 36  pipeline = |*|+|*|+|b|  -
          ;a0, a3 - point to the external memory
          ;a1, a4, a2, a6 - point to the internal memory
          ;the DMA controller operates external to internal memory transfers

The time sequencing of the CPU and DMA controller accesses, is:

<table>
<thead>
<tr>
<th>DMA</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>no access (interface busy with a CPU access)</td>
<td>external + internal</td>
</tr>
<tr>
<td>no access (CPU is more prioritary)</td>
<td>external + internal</td>
</tr>
<tr>
<td>external (2 clocks)</td>
<td>internal</td>
</tr>
<tr>
<td>external (the conflict is anyway generated by the CPU)</td>
<td>internal + conflict (external + internal, postponed)</td>
</tr>
<tr>
<td>internal</td>
<td>external + internal</td>
</tr>
<tr>
<td>external (2 clocks)</td>
<td>internal</td>
</tr>
<tr>
<td>external</td>
<td>internal + conflict (external interface occupied with a DMA controller access)</td>
</tr>
</tbody>
</table>

5.1.g.3: one loop, 5 instructions, 7 clocks

<table>
<thead>
<tr>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 1 pipeline =</td>
</tr>
</tbody>
</table>

| clock = 14 pipeline = |a|c|b|c|; LDF *AR4--(1),R4  |
| clock = 15 pipeline = |d|a|c|b; MPYF3 *AR1,*AR0,R0  || ADDF3 R0,R1,R2
| clock = 16 pipeline = |*|+|*|+|c|  -
| clock = 17 pipeline = |b|d|a|c; MPYF3 *AR3,R4,R1  || STF R2,*AR6--(1)
| clock = 18 pipeline = |c|b|d|a; MPYF3 *AR1++(1),*AR3--(1),R0  || SUBF3 R1,R0,R2
| clock = 19 pipeline = |a|b|c|d; MPYF3 *AR0++(1),R4,R1  || STF R2,*AR2++(1)
| clock = 20 pipeline = |*|+|*|+|b|  -
| clock = 21 pipeline = |d|a|c|b; LDF *AR4--(1),R4  |
| clock = 22 pipeline = |b|d|a|c; MPYF3 *AR1,*AR0,R0  || ADDF3 R0,R1,R2
| clock = 23 pipeline = |*|+|*|+|a|  -
| clock = 24 pipeline = |c|b|d|a; MPYF3 *AR3,R4,R1  || STF R2,*AR6--(1)
| clock = 25 pipeline = |a|b|c|d; MPYF3 *AR1++(1),*AR3--(1),R0  || SUBF3 R1,R0,R2
| clock = 26 pipeline = |a|d|b|c; MPYF3 *AR0++(1),R4,R1  || STF R2,*AR2++(1)
| clock = 27 pipeline = |*|+|*|+|c|  -
          ;a1, a4 - point to the external memory
          ;a0, a3, a2, a6 - point to the internal memory
          ;the DMA controller transfers data from the external to the internal memory

The CPU and DMA controller memory accesses unfold as follows:

<table>
<thead>
<tr>
<th>DMA</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>no access (interface busy with a CPU access)</td>
<td>external</td>
</tr>
<tr>
<td>no access (CPU has greater priority)</td>
<td>external + internal</td>
</tr>
<tr>
<td>external (2 clocks)</td>
<td>internal</td>
</tr>
<tr>
<td>external (the conflict is anyway caused by the CPU)</td>
<td>internal + conflict (external + internal, postponed)</td>
</tr>
<tr>
<td>internal</td>
<td>external + internal</td>
</tr>
<tr>
<td>external (2 clocks)</td>
<td>internal</td>
</tr>
<tr>
<td>external</td>
<td>internal + conflict (external interface occupied with a DMA controller access)</td>
</tr>
</tbody>
</table>

# The memory conflicts together with the restricted addressing possibilities for the parallel instructions cause time consumption when coding computational-intensive parts of the algorithms.

5.1.h: data concurrently transferred by CPU and DMA controller

<table>
<thead>
<tr>
<th>F D R E</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock = 11 pipeline =</td>
</tr>
<tr>
<td>clock = 12 pipeline =</td>
</tr>
<tr>
<td>clock = 13 pipeline =</td>
</tr>
<tr>
<td>clock = 14 pipeline =</td>
</tr>
<tr>
<td>clock = 15 pipeline =</td>
</tr>
<tr>
<td>clock = 16 pipeline =</td>
</tr>
<tr>
<td>clock = 17 pipeline =</td>
</tr>
</tbody>
</table>
| clock = 18 pipeline = |*|+|d|b|a;  -
| clock = 19 pipeline = |*|+|d|b|a;  -
| clock = 20 pipeline = |*|+|d|b|a;  -
| clock = 21 pipeline = |*|+|*|+|d; LDF *AR3++(1),R1  || STF R1,*AR1++(1)  ;the CPU starts transferring now

58
clock = 22  pipeline = [b;1;*:1*: LDF *AR3++(1),R1; STF R1,*AR1++(1)

clock = 23  pipeline = [c;*:a;*: LDF *AR3++(1),R1; STF R1,*AR1++(1)

5.1.1: inverse power estimation

5.1.1.1: 2 instructions 5 clocks, loop twice longer

5.1.2: power normalization

5.1.2.1: there is no DMA controller job in progress

5.1.2.2: the CPU performs the last transfer

5.1.2.3: a test to check if the DMA controller has finished

5.1.2.4: branch not taken - the DMA controller has finished

5.1.2.5: within the loop, there are two CPU and one DMA controller data accesses performed each clock

5.1.2.6: the CPU accesses are within different RAM blocks, thus there will not be 3 accesses attempted within the same block

5.1.2.7: as explained at the beginning of 5.1.c, this is possible only within the internal memory

5.1.2.8: two external reads cannot occur at once

5.1.2.9: moreover, they must wait upon the (2 clocks) external write completion

5.1.2.10: a fetch (although from the cache) is not made until the pipeline does not "consider" that it is necessary
5.1.2: 5 instructions 8 clocks

FDRE

clock = 5 pipeline = [c:d:b:a] RPTB norm

--
clock = 21 pipeline = [a:b:d:*] MPYF3 *AR2,R0,R2
clock = 22 pipeline = [c:a:b:d] STF R1,*AR1++(1)
clock = 23 pipeline = [d:c:a:b] STF R2,*AR2--(1)
clock = 24 pipeline = [b:d:c:a] LDF *AR0++(1),R0
clock = 25 pipeline = [a:b:d;c] MPYF3 *AR1,R0,R1
clock = 26 pipeline = *[r]++[c] -
clock = 27 pipeline = *[r]++[d] -
clock = 28 pipeline = *[r]++[d] -
clock = 29 pipeline = [a:c:b] MPYF3 *AR2,R0,R2
clock = 30 pipeline = [d:c:a:b] STF R1,*AR1++(1)
clock = 31 pipeline = [b:d:c:a] STF R2,*AR2--(1)
clock = 32 pipeline = [a:b:d;c] LDF *AR0++(1),R0
clock = 33 pipeline = [c:a:b:d] MPYF3 *AR1,R0,R1
clock = 34 pipeline = *[r]++[d] -
clock = 35 pipeline = *[r]++[d] -
clock = 36 pipeline = *[r]++[d] -
;ar0, ar1, ar2 point to external memory
;one clock delay proceeds from the first external write (it lasts 2 clocks)
;the other two clocks delay are given by the second (2 clocks) external write, which postpones the next external read

5.1.1: correlation

FDRE

clock = 5 pipeline = [c:d:b:a] RPTB corr

--
clock = 15 pipeline = [d:b:a:*] LDF *AR6--(1),R4
clock = 16 pipeline = [c:d:b:a] MPYF3 *AR5,*AR2,R0
| ADDF3 R0,R1,R2
clock = 17 pipeline = [a:c:d:b] MPYF3 *AR3(1),R4,R1
| STF R2,*AR3--(1)
clock = 18 pipeline = [b:a:c:d] MPYF3 *AR5++(1),*AR3,R0
| SUBF3 R1,R0,R2
clock = 19 pipeline = [d:b:a:c] MPYF3 *AR2,R4,R1
| STF R2,*AR2++(1)
clock = 20 pipeline = *[r]++[a] -
clock = 21 pipeline = [c:d:b:*] LDF *AR6--(1),R4
clock = 22 pipeline = [a:c:d:b] MPYF3 *AR5,*AR2,R0
| ADDF3 R0,R1,R2
clock = 23 pipeline = [b:a:c:d] MPYF3 *AR3(1),R4,R1
| STF R2,*AR3--(1)
clock = 24 pipeline = [d:b:a:c] MPYF3 *AR5++(1),*AR3,R0
| SUBF3 R1,R0,R2
clock = 25 pipeline = [c:d:b:a] MPYF3 *AR2,R4,R1
| STF R2,*AR2++(1)
clock = 26 pipeline = *[r]++[b] -
;ar2, ar3 - point to internal memory
;ar5, ar6 - point to external memory
;three CPU data accesses at once generate a conflict

5.1.1: update coefficients

5.1.1.1: one loop, 5 clocks

FDRE

clock = 5 pipeline = [c:d:b:a] RPTB upd

--
clock = 10 pipeline = [c:d:b:*] STF R2,*AR4++(1)
clock = 11 pipeline = [a:c:d:b] LDF *AR4,R2
clock = 12 pipeline = [b:a:c:d] ADDF3 *AR2--(1),R2
clock = 13 pipeline = *[r]++[c] -
clock = 14 pipeline = *[r]++[c] -
clock = 15 pipeline = [d:b:a:*] STF R2,*AR4++(1)
clock = 16 pipeline = [c:d:b:a] LDF *AR4,R2
clock = 17 pipeline = [a:c:d:b] ADDF3 *AR2--(1),R2
clock = 18 pipeline = *[r]++[d] -
clock = 19 pipeline = *[r]++[d] -
;all auxiliary registers point to external memory
;external write takes 2 clocks, postponing thus with 2 clocks the next external read

60
5.1.1.2: two loops of 2 clocks

**FDRE**
clock = 5 pipeline = [c|d|b|a]; RFTB update

```
clock = 8 pipeline = [c|b|a|d]; LDF *AR4,R2 ; STF R2,*AR0+(1)
clock = 9 pipeline = [d|c|b|a]; ADDF *AR2–(1),R2
```

2 wait-states peripheral bus read, totally 3 extra clocks
5.1:m: auxiliary registers update in parallel instructions (connected with 5.1.k)

- The auxiliary register update for the case of using the same auxiliary register to point out two indirect operands in the parallel instructions, is:
  a) for parallel arithmetic or load and store, the destination operand of the store;
  b) for parallel load or parallel store, the second (as mentioned in the source file) indirect operand;
  c) for parallel arithmetic, the indirect operand of the addition/subtraction.

5.1:a: the interrupt service routine

5.1:a:1: using only one auxiliary register

**FDRE**
clock = 14 pipeline = [r|v|t|x|y|z]

```
clock = 15 pipeline = [r|v|t|x|y|z]
clock = 16 pipeline = [r|v|t|x|y|z]
clock = 17 pipeline = [r|v|t|x|y|z]
clock = 18 pipeline = [t|x|y|z|t] STI BK, @0809d8dH
```

6 extra clocks

1: The interrupt is being accepted, 4 extra clocks

- 2 clocks register conflict (cannot decode until ar2 is not loaded)
- External write after read (fetch) takes 3 clocks, no fetch
clock = 43  pipeline = [b|b|b|; STF  R0, *AR2=(1)%
          ...3 extra clocks due to the peripheral bus read

clock = 44  pipeline = [s|s|s|; R0

clock = 45  pipeline = [s|s|s|; R0

clock = 46  pipeline = [s|s|s|; R0

clock = 47  pipeline = [s|s|s|; R0

clock = 48  pipeline = [s|s|s|; R0

clock = 49  pipeline = [a|d|d|; LDI  @0809d02H, BK

clock = 50  pipeline = [b|a|d|; STI  AR2, @0809d03H
          ...the same 3 extra clocks write after read. no fetch

clock = 51  pipeline = [s|b|a|d|; R0

clock = 52  pipeline = [s|s|s|; R0

clock = 53  pipeline = [s|s|s|; R0

clock = 54  pipeline = [c|v|b|a; LDI  @0809d85H, AR2

clock = 55  pipeline = [v|d|c|b; LDF  *AR2=(1)%; R0
          ...address register conflict

clock = 56  pipeline = [s|s|s|; R0

clock = 57  pipeline = [s|s|s|; R0

clock = 58  pipeline = [a|d|d|; FIX  R0, R0
          ...primary bus) external read. no fetch (through the same bus interface)

clock = 59  pipeline = [s|s|s|; R0

clock = 60  pipeline = [c|v|b|a; LDI  @0809d44H, AR2

clock = 61  pipeline = [b|c|v|a; STI  R0, *AR2
          ...register conflict, 2 clocks

clock = 62  pipeline = [s|s|s|; R0

clock = 63  pipeline = [s|s|s|; R0

clock = 64  pipeline = [d|b|v|; LDI  @0809d2cH, R0

clock = 65  pipeline = [a|d|b|; CMP  @0809d23H, R0

clock = 66  pipeline = [a|d|d|; BGED  0025H

clock = 67  pipeline = [b|c|v|a|; JACC

clock = 68  pipeline = [b|c|v|a; STI  AR2, @0809d85H

clock = 69  pipeline = [s|s|s|; R0

clock = 70  pipeline = [a|b|v|; ADDI  1,R0

clock = 71  pipeline = [s|s|s|; R0

clock = 72  pipeline = [s|s|s|; R0

clock = 73  pipeline = [s|s|s|; R0

clock = 74  pipeline = [c|a|d|; STI  R0, @0809d2cH

clock = 75  pipeline = [b|c|v|a; LDI  @0809d0eH, AR2

clock = 76  pipeline = [b|c|v|a; LDI  @0809d0eH, BK

clock = 77  pipeline = [a|d|b|c; LDF  @0809d0eH, R0

clock = 78  pipeline = [c|a|d|; LDI  @0809d0eH, R0

clock = 79  pipeline = [b|c|v|a; LDIU  @0809d0eH, ST

clock = 80  pipeline = [d|b|c|a; RETIU  ...return to the interrupted sequence of instructions

clock = 81  pipeline = [s|s|s|; R0

clock = 82  pipeline = [s|s|s|; R0

clock = 83  pipeline = [s|s|s|; R0

# The only instructions that do not affect the flags are the parallel instructions, the pushes, the stores, and the conditional loads.

A conditional load was used before saving the status register - the auxiliary register was early loaded to avoid later conflicts.

5.1.n.2: using two auxiliary registers

      F D R E

clock = 402  pipeline = [s|s|s|; R0
          ...the interrupt is being accepted, 4 extra clocks

clock = 403  pipeline = [s|s|s|; R0

clock = 404  pipeline = [s|s|s|; R0

clock = 405  pipeline = [s|s|s|; R0

clock = 406  pipeline = [a|v|v|; STI  BK, @0809debfH

clock = 407  pipeline = [b|v|v|; STI  AR2, @0809decfH

clock = 408  pipeline = [c|b|v|a; STI  AR5, @0809dedfH

clock = 409  pipeline = [d|c|b|a; LDIU  LBK

clock = 410  pipeline = [a|d|c|b; LDIU  xpoint, AR2

clock = 411  pipeline = [b|a|d|c; LDIU  @0809de48H, AR5

clock = 412  pipeline = [c|b|a|d; STI  ST, @0809dedfH

clock = 413  pipeline = [d|b|a|c; STF  R0, @0809dedfH

clock = 414  pipeline = [a|d|c|b; STF  R0, @0809dedfH

clock = 415  pipeline = [b|a|d|c; FLOAT  *AR5, R0

clock = 416  pipeline = [c|b|a|d; STF  R0, *AR2=(1)%

clock = 417  pipeline = [d|b|a|d; STF  R0, *AR2=(1)%

clock = 418  pipeline = [s|b|b|; LDI  dsiz, BK
          ...2 waits peripheral bus read, 3 extra clocks

clock = 419  pipeline = [s|b|b|; LDI

clock = 420  pipeline = [s|b|b|; LDI

clock = 421  pipeline = [a|d|c|b; STI  AR2, xpoint
          ...primary bus write after read, totally 3 clocks. no fetch

clock = 422  pipeline = [s|s|s|; R0

clock = 423  pipeline = [s|s|s|; R0

clock = 424  pipeline = [s|s|s|; R0

clock = 425  pipeline = [b|v|a|d|; LDI  dpoin,AR2

62
clock = 426  pipeline = [c:b] [a] FLOAT **AR5(1), R0  
  an address cannot be generated earlier than the address register is loaded  
  (2 extra clocks)

  clock = 427  pipeline = [c:b] [a]  
  R0, *AR2–(1)%, R0

  clock = 428  pipeline = [c:b] [a]  
  a, BK

  clock = 429  pipeline = [c:b] [a]  
  the same 3 extra clocks peripheral bus read

  clock = 430  pipeline = [d:a] [c] LDI
  RO, *AR2–(1)%;

  clock = 431  pipeline = [d:a] [c]  
  (primary bus) write after fetch (read), 2 extra clocks

  clock = 432  pipeline = [d:a] [c]  
  no fetch through the same external bus interface (while writing)

  clock = 433  pipeline = [d:a] [c]  
  the same register conflict, 2 extra clocks

  clock = 434  pipeline = [b:d] [c] STI
  AR2, cpoin

  clock = 435  pipeline = [b:d] [c]  
  (2 extra clocks)

  clock = 436  pipeline = [b:d] [c]  
  read through the same bus interface, no fetch

  clock = 437  pipeline = [b:d] [c]  
  read through the same bus interface, no fetch

  clock = 438  pipeline = [a] [d] [c] LDI
  @0809d86H, AR2

  clock = 439  pipeline = [c:a] [b] LDF  
  *AR2–(1)%, R0

  clock = 440  pipeline = [c:a] [b]  
  the same register conflict, 2 extra clocks

  clock = 441  pipeline = [c:a] [b]  
  the same register conflict, 2 extra clocks

  clock = 442  pipeline = [d:c] [c] [a] FIX
  R0, R0

  clock = 443  pipeline = [d:c] [c] [a]  
  reads through the same bus interface, no fetch

  clock = 444  pipeline = [d:c] [c] [a]  
  reads through the same bus interface, no fetch

  clock = 445  pipeline = [b:a] [d] LDI
  cnt1, R0

  clock = 446  pipeline = [b:a] [d] ADDD
  1, R0

  clock = 447  pipeline = [d:c] [b] [a] CMPF
  b, R0

  clock = 448  pipeline = [a] [d] [c] BGEQ
  0b48eH

  clock = 449  pipeline = [a] [d] [c] [a] IACK
  *AR2

  clock = 450  pipeline = [c:b] [a] [d] STI
  AR2, @0809d86H

  clock = 451  pipeline = [c:b] [a] [d]  
  lock makes a dummy read. no fetch through the same external bus interface

  clock = 452  pipeline = [d:c] [b] [a] NOP
  only 2 clocks can be gained (in this variant) by using the delayed branch

  clock = 453  pipeline = [d:c] [b] [a] STI
  RO, cnt1

  clock = 454  pipeline = [b] [a] [d] [c] LDI
  @0809d86H, AR2

  clock = 455  pipeline = [b] [a] [d] [c] LDI
  @0809d86H, AR2

  clock = 456  pipeline = [b] [a] [d] [c] LDI
  @0809d86H, BK

  clock = 457  pipeline = [b] [a] [d] [c] LDF
  @0809d86H, R0

  clock = 458  pipeline = [b] [a] [d] [c] LDF
  @0809d86H, R0

  clock = 459  pipeline = [b] [a] [d] [c] LDH
  @0809d86H, ST

  clock = 460  pipeline = [d:c] [b] [a] RETI
  return to the interrupted program

  clock = 461  pipeline = [d:c] [b] [a]  
  total interruption time = 62 clocks

  clock = 462  pipeline = [d:c] [b] [a]  
  total interruption time = 62 clocks

  clock = 463  pipeline = [d:c] [b] [a]  
  total interruption time = 62 clocks

# Looking for the clocks when the primary bus interface is available, we infer that a DMA controller job in progress involving external reads would cause two more conflicts, while one involving external writes would generate four more conflicts.

5.1.n.3: sparing one clock

F D R E

clock = 56  pipeline = [a] [c:b] [d] STI
  R0, *AR5

clock = 57  pipeline = [d:a] [c] LDI
  @0809d2cH, R0

  until here the same with 5.1.n.2

clock = 58  pipeline = [b:d] [a] [c] CMPF
  @0809d33H, R0

  until here the same with 5.1.n.2

clock = 59  pipeline = [b:d] [a] [c] BGEQ
  0b02SH

  this is not necessary when working with the board described in [17]

clock = 60  pipeline = [a:c:b] [d] IACK
  *AR2

  this is not required by the board described in [17]

clock = 61  pipeline = [a:c:b] STI
  AR2, @0809d85H

  this is not required by the board described in [17]

clock = 62  pipeline = [a:c] [b] IST
  lock dummy reads through the primary bus interface. no fetch

clock = 63  pipeline = [b] [d] [a] ADDI
  1, R0

  from here the same with 5.1.n.2

clock = 64  pipeline = [c:b] [a] [d] STI
  R0, @0809d2cH

  from here the same with 5.1.n.2

clock = 65  pipeline = [c:b] [a] [d] LDI
  @0809d9eH, AR2

# When converting standard branches to delayed branches:

  a) instructions cannot be moved before those using their result;

  b) if the branch is conditional, the condition must not be affected by the new inserted instructions.

5.1.n.4: the F4 interrupt routine

F D R E

clock = 823  pipeline = [a:c:b] [d] BGEQ
  0b48eH

  until here it is identical with 5.1.n.2

clock = 824  pipeline = [a:c:b] [d] IACK
  *AR2

  this is not required by the board described in [17]

clock = 825  pipeline = [b:d] [a] [c] STI
  AR2, @0809d86H

  this is not required by the board described in [17]

clock = 826  pipeline = [b:d] [a] [c]  
  dummy read, no fetch through the same bus interface

  clock = 827  pipeline = [b:d] [a]  
  dummy read, no fetch through the same bus interface

  clock = 828  pipeline = [c:b] [d] LDI
  0, R0

  clock = 829  pipeline = [d:a] [c] [b] LDI
  R0, cnt1
Note: the above presented traced executions are not 100% the original versions generated by the simulator.
As mentioned in section 5.4, the simulator has some bugs.
5.2 Processor load and memory allocation example

A C30 load of 92% is obtained for the next set of parameters:

- \( N = 2016 \) - the adaptive filter length;
- \( f_s = 7997 \) - the sampling frequency (Hz);
- \( B = 8 \) - the filter-part block size;
- \( A = 504 \) - the update-part block size;
- \( Q = 56 \) - the subfilters length;
- \( Z = 504 \) - the update equation partitions length;
- \( M = 64 \) - the total convolution partitions length, including the overlap (\( M - Q \));
- \( L = 1024 \) - the total correlation partitions length, including the overlap (\( L - Z \)).

For this set of parameters:

- one interrupt comes each 2084 clocks (=125\( \mu \)s);
- one small frame execution lasts 16672 clocks (=1msec);
- one big frame execution is completed in 1050336 clocks (=0.63sec).

The filter routine execution lasts around 11000 clocks (10994), or 5.25 samples (including 5 interrupt routine executions), being split as follows:

- the FFT routine execution - around 850 clocks (843) [=5\% load];
- the convolution routine execution - around 8500 clocks (8494) [=50.96\% load];
- the IFFT routine execution - around 1000 clocks (1007) [=6\% load], and
- the rest (the technical housekeeping) - 650 clocks [=3.9\% load].

The interrupt service routine execution takes 62 clocks and 99 clocks, for the interrupts 1, ..., \( B - 1 \), respectively \( B \).

The update routine execution is around 241000 clocks long (240964), being composed as follows:

- around 5600 clocks (5665) to save the input signals buffers;
- around 50000 clocks (50186) to transform them to the frequency domain;
- around 3600 clocks (3627) for the inverse power estimation;
- around 4100 clocks (4116) for the spectrum normalization;
- around 500 clocks (528) for the conjugation;
- around 137000 clocks (136804) for the partitioned correlation followed by transformation to the time domain:
  - around 35000 clocks (34972) for the first partition:
    - \( \approx 2000 \) clocks to insert \((X^H_t[kA])^*\) into the delay line,
=3000 clocks for the correlation,
=25000 clocks for the IFFT routine execution,
=2000 clocks for the bit-reversal, and
=3000 clocks for the first partition update;
around 34000 (33944) clocks for each of the rest of the partitions:
=1000 clocks to copy the partition from the delay line, and
- the rest lasts the same;
around 40000 clocks (40018) to transform the new coefficients to the freq. domain:
around 1200 clocks (1197) for the first partition:
=128 clocks to copy and bit-reverse the first partition to the internal RAM,
=850 clocks to transform the first partition to the frequency domain,
=128 clocks to store the first partition, and
=91 clocks for the rest of the technical housekeeping;
around 1100 clocks to transform each of the rest of the partitions to the freq. domain (1110 for the 1...N/Q-2 ones and 1084 for the last one):
=64 clocks for the bit-reversal, and
- the rest lasts the same;
around 33 clocks for the synchronization performed at the end.

The time unfolding of different routines executions is depicted for a clearer impression on figure 5.2.1 (explanations follow).

---

Figure 5.2.1: Small and big frame splitting timings
As explained in section 2.1 (see also figure 2.1.1), during the time frame when the new adaptive coefficients are computed, the input signal vector is 63 \((A/B)\) times filtered.

Firstly we refer to the lower side of figure 5.2.1. The update slices (denoted \(U_{1/63}...U_{63/63}\)) are performed between the filter executions (denoted \(F_{1}...F_{63}\)). Inside the 50th update slice the code completes the computation of the new frequency domain adaptive coefficients; during the last update slice performance they are interchanged with the old coefficients.

The code parts executed for each (filter) block of samples are depicted in the upper side of figure 5.2.1. Every interrupt request causes the start of an interrupt service routine execution. As stated in section 2.4, the first one \((I_{0})\) is longer than the others \((I_{1}...I_{5})\). They divide the execution of the filter and update routines into the slices denoted \(F_{1/6}...F_{6/6}\), respectively \(U_{1/3}...U_{3/3}\). Above the time axis are mentioned the execution durations for each piece of code.

After the new coefficients are available, the C30 is idle during the update routine execution around 83000 clocks (83171). The figure of 92% processor load refers to these idle clocks.

The overall load is 65.86% due to the filter routine executions, 22.94% due to the update routine execution, and 3.2% due to the interrupt service routine executions.

One moreover conflict within the time-critical part of the code (the convolution inner loop) would have caused a 5.85% processor overload.

One complete coefficients update lasts around 0.63sec.

The processing delay is 1.625msec and the total delay is 1.750msec.

For this parameter set, the buffers are allocated in memory as follows:

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Lowest Address</th>
<th>Length</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>init</td>
<td>000000h</td>
<td>001000h (fixed)</td>
<td>interrupt vectors and reserved locations</td>
</tr>
<tr>
<td>rbpc1</td>
<td>030000h</td>
<td>000500h (arbitrary)</td>
<td>buffer1 with the PC</td>
</tr>
<tr>
<td>pccd1</td>
<td>031000h</td>
<td>00000ah (arbitrary)</td>
<td>commands from the PC</td>
</tr>
<tr>
<td>rbpc2</td>
<td>032000h</td>
<td>000600h (arbitrary)</td>
<td>buffer2 with the PC</td>
</tr>
<tr>
<td>.data</td>
<td>809d400h</td>
<td>000100h</td>
<td>implementation code variables and stack</td>
</tr>
<tr>
<td>twid</td>
<td>809e400h</td>
<td>000200h ((max L)/2)</td>
<td>twiddle factors</td>
</tr>
<tr>
<td>lb</td>
<td>001000h</td>
<td>000400h (L)</td>
<td>update spare vector</td>
</tr>
<tr>
<td>w0</td>
<td>001400h</td>
<td>0007e0h (N)</td>
<td>time domain adaptive weights vector</td>
</tr>
<tr>
<td>l0</td>
<td>002000h</td>
<td>001000h ((1+(N-Z)/A)-L)</td>
<td>update delay line</td>
</tr>
<tr>
<td>xl2</td>
<td>003000h</td>
<td>000400h (L)</td>
<td>update overlapped input signal vector</td>
</tr>
<tr>
<td>yl</td>
<td>003400h</td>
<td>000400h (L)</td>
<td>residual signal vector normalized spectrum</td>
</tr>
<tr>
<td>rl</td>
<td>003800h</td>
<td>000400h (L)</td>
<td>((update)) overlapped residual signal vector</td>
</tr>
<tr>
<td>m0</td>
<td>004000h</td>
<td>003d80h ((1+(N-Q)/B)-M)</td>
<td>filter delay line</td>
</tr>
<tr>
<td>xl</td>
<td>008000h</td>
<td>000400h (L)</td>
<td>interrupt service routine overlapped input signal vector</td>
</tr>
<tr>
<td>pl</td>
<td>008400h</td>
<td>000201h (1+L/2)</td>
<td>input signal vector inverse power estimate</td>
</tr>
<tr>
<td>rb</td>
<td>008800h</td>
<td>001f8h (A)</td>
<td>residual signal vector</td>
</tr>
<tr>
<td>d</td>
<td>008b00h</td>
<td>000010h (2-B)</td>
<td>far-end signal delayed echo vector</td>
</tr>
<tr>
<td>xl1</td>
<td>009000h</td>
<td>000400h (L)</td>
<td>filter overlapped input signal vector</td>
</tr>
<tr>
<td>f0</td>
<td>009500h</td>
<td>000900h ((N/Q)-M)</td>
<td>current frequency domain adaptive weights vector</td>
</tr>
<tr>
<td>uv</td>
<td>00a000h</td>
<td>002000h (2(\times)max M))</td>
<td>filter spare vector</td>
</tr>
<tr>
<td>.text</td>
<td>00b000h</td>
<td>00600h</td>
<td>adaptive filter implementation code</td>
</tr>
<tr>
<td>so</td>
<td>00c000h</td>
<td>000900h ((N/Q)-M)</td>
<td>next frequency domain adaptive weights vector</td>
</tr>
<tr>
<td>cnt2</td>
<td>00d000h</td>
<td>000800h</td>
<td>record zone for small frames counter</td>
</tr>
<tr>
<td>adda</td>
<td>804000h</td>
<td>00020h</td>
<td>analog I/O</td>
</tr>
<tr>
<td>mnr</td>
<td>808000h</td>
<td>000420h</td>
<td>memory mapped registers and reserved locations</td>
</tr>
<tr>
<td>ram0</td>
<td>809800h</td>
<td>000500h</td>
<td>internal RAM available for computations</td>
</tr>
</tbody>
</table>
5.3 Running with any valid parameter set

5.3.a The parameter set acceptance

Due to the DPBFDAF algorithm, on one hand, and to the C30 implementation, on the other hand, there are some conditions to be checked over the specific parameter set the run is intended with.

The algorithm supposes restrictions like $A/B$ integer.

Due to the implementation for instance if the counter of a loop is derived from a parameter by some arithmetic rule, it is to be avoided the situation when it is computed as zero, and the loop will thus be executed a number of $2^{31} - 1$ times.

Another example concerns the sampling frequency, which can be as high as 160kHz, if the matter is only to complete the interrupt service routine execution until the next interrupt request comes, but in this case the C30 is more than 100% loaded even for the "lightest" parameter set and thus the DPBFDAF algorithm cannot run.

Another type of restrictions that have to be considered proceed from physical reasons:

- $N$ (the adaptive filter length): must be at least the (room) impulse response function length;
- $B$ (the filter routine block size) [decides the answer delay]: is limited ($B \leq B_{\text{max}}$) by the standards (the rules) the application must run under (for instance CCITT - for telephony);
- $A$ (the update routine block size) [gives the decorrelation dimension]: it is not necessary to track the input signal statistical properties faster than they vary ($A \geq A_{\text{min}}$);
- $L$ (the correlation overlapped partition size): is limited by the on-chip RAM block size, $L \leq 1024$ (see also chapter 3).

Putting together all these restrictions we find the following conditions to be checked over the parameter set:

$L$ must be: $2^x$ (power of two), $\geq 16$, $\leq 1024$, $\geq Z + A - 1$, $\geq 2A - 6$ or $\leq 2A - 2$;

$M$ must be: $2^x$ (power of two), $\geq 16$, $\leq 256$, $\geq Q + B - 1$;

$A$ must be: $2x$ (multiple of two), $\geq 6$, $\leq 512$;

$B$ must be: $2x$ (multiple of two), $\geq 6$, $\leq 256$, $A/B$ integer, $(L/B$ integer for $\text{variant.upd})$;

$Z$ must be: $\geq 2$, $\leq 1024$, $Z/A$ integer, $N/Z$ integer;

$Q$ must be: $2x$ (multiple of two), $\geq 6$, $\leq 256$, $Q/B$ integer, $N/Q$ integer;

$N$ must be: impulse response function length;

$f_s$ must be: $\leq$ maximal sampling frequency for which the load is below 100%. 


5.3.b The implementation code variables calculation

The implementation code needs to derive some variables from the parameter set \((N, L, A, M, B, Z, Q, f, \alpha, \beta)\) which are used as loops counters, computations constants, \texttt{fft} routine calling parameters, and so on (see the source file \texttt{dpbfdaf.asm}).

The list of the needed calculations is:

\[
\begin{align*}
11 & := L - 1; \\
hl & := L / 2; \\
hl1 & := hl - 1; \\
hl2 & := hl1 - 1; \\
hl2a & := hl2; \\
hl3 & := hl2 - 1; \\
hhl2 & := hl2 / 2 - 1; \\
z1 & := Z - 1; \\
a1 & := A - 1; \\
a2 & := a1 - 1; \\
ah2 & := A / 2 - 2; \\
l2a & := L - 2A; \\
l2a2 & := l2a / 2 - 2; \\
h1q2 & := (Q - 2) / 2 - 1; \\
m1 & := M - 1; \\
dm & := 2M; \\
hm & := M / 2; \\
hm2 & := hm - 2; \\
hm3 & := hm2 - 1; \\
hmq1 & := (M - Q) / 2 - 1; \\
q1 & := Q - 1; \\
b1 & := B - 1; \\
b2 & := b1 - 1; \\
hb2 & := B / 2 - 2; \\
stp & := M \cdot Q / B - hm; \\
lza1 & := L \cdot (Z / A + 1); \\
nrsf3 & := N / Q - 3; \\
fsiz & := M \cdot (N - Q + B) / B; \\
usiz & := L \cdot (N - Z + A) / A; \\
save & := (1 - A) / B - 1; \\
mstg & := \log_2 M; \\
lstg & := \log_2 L; \\
nfts & := A / B - 1; \\
ncors1 & := N / Z - 1; \\
period & := \text{round} \left[ 1 / (f \cdot 120) \right]; \\
close & := \text{period} - 100; \\
c1 & := 1 + \beta; \\
\text{the following three ones require a floating-point division routine:} \\
c2 & := - \beta / 2\alpha L; \\
msc & := 1 / M; \\
lsc & := 1 / L; \\
\text{the next two ones are experimentally established:} \\
mdma & := 86, 42, 22, 10, 6 - \text{for } M = 256, 128, 64, 32, 16; \\
mrc & := 170, 86, 42, 22, 10 - \text{for } M = 256, 128, 64, 32, 16.
\end{align*}
\]
After performing the computations described in section 5.3.b, all buffers sizes are known.

The circular buffers allocations have restrictions as follows:

<table>
<thead>
<tr>
<th>Buffer</th>
<th>Size</th>
<th>The lowest address must be a multiple of</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0</td>
<td>fsiz</td>
<td>(2^{\text{trunc} \left( \log_{2} \text{fsiz} \right)} + 1)</td>
</tr>
<tr>
<td>L0</td>
<td>usiz</td>
<td>(2^{\text{trunc} \left( \log_{2} \text{usiz} \right)} + 1)</td>
</tr>
<tr>
<td>XL, XL1</td>
<td>l</td>
<td>2L</td>
</tr>
<tr>
<td>RB</td>
<td>a</td>
<td>2A</td>
</tr>
<tr>
<td>D</td>
<td>dsiz</td>
<td>2B</td>
</tr>
</tbody>
</table>

Considering the total available memory map (figure 1.3.2), it has to be done a routine which allocates all the buffers in the area A (avoiding the 192 reserved locations), excepting the buffers for the communication with the PC (in the area B), and the twiddle factors buffer (in the internal RAM block 1), as well as the implementation code variables buffer (in the internal RAM block 1).

A proposition for the memory allocation algorithm can be roughly described as follows:

1. ordonate from large to small the circular buffers sizes;
2. ordonate from large to small the linear buffers sizes;
3. allocate the circular buffers, beginning with the largest one, from low memory addresses to high memory addresses; while allocating, keep track of the available memory map left;
4. begin with the largest linear buffer, and continue with the other ones, in decreasing order of the sizes:
   - allocate the linear buffer within the smallest gap where it fits;
   - compute the available memory map left;
   - if there are more buffers left to be allocated, loop back ("4.").
5.4 Software development environment

For the assembler and the linker, [19] is referred.

The simulators are very useful software tools to develop such an application without needing a DSP board. They make possible to simulate the processor functioning as well as the needed memory plus I/O map presence, and to debug the programs, supplying all the necessary informations. Further details can be found in [22-23].

The simulation speed is typically on the order of hundreds of instructions per second, for IBM PC. One complete coefficients update simulation for the 5.2 parameter set example (0.63sec) lasts on a 386 around one hour with the TMS320C3x simulator version 1.20, and two and a half hours with the version 2.10 under Windows.

A version 1.20 minor disadvantage is the non-windows oriented information presentation, while the (discovered) major disadvantages are: the bug concerning the execution from a journal file (the simulator cannot go on), and the bug that shows up when trying to simulate simultaneous interrupts (they are not simulated). For the first bug the solution has been found by "hacking" the MS-DOS service that opens a file handle (INT21h, function 3dh). The simulator was trying to open a (journal) file with a non-DOS name. See the source file sim.asm for further details. The second bug can be solved only by the (simulator) program writers. Besides the fact that it runs faster, other minor advantages are the on-line help availability, and the possibility to simulate the memory wait-states. The major advantage of this old version is the trace capability (see the examples in section 5.1). This allows the pipeline conflicts detection and minimization, absolutely necessary along the time-critical parts of code.

The version 2.10 has the advantage to run under Windows, but does not offer the trace capability, the memory wait-states simulation possibility, and the on-line help (unfortunately only a documentation for a C50 simulator [22] was available).

A detail that must be exploited, at both simulators, is the memory map declaring to cover exactly the buffers needed by the program. The simulation halts displaying the message "illegal memory access" when the tested code tries to address outside the declared memory map. This type of mistakes can thus be very easy discovered.
A last remark concerns the statement "non-real-time program verification", which is very oftenly met in relation with the simulators. In my opinion the correct wording is "real-time program verification at another time scale". "Non-real-time" implies that the events time sequencing is different, but in this case each action receives the same reaction (answer) as at the "real-" time scale.

The other software tools available to develop applications, the debug monitors, need a DSP board to run on.

Loughborough Sound Images Ltd. delivers the TMS320C30 PC System Board together with the MON30 (command-line debug monitor) and the SDS30 (windows-based debug monitor). These programs have a PC resident part, which is only the user interface, and a DSP board resident part, composed out of the monitor code and a copy of the DSP registers and internal memory.

Because the monitor code is interposed between the C30 and the user's code needed to be run, for instance an instruction which disables all the interrupts will not be "completely" executed (at least the communication with the PC must be awake). The communication with the PC (mentioned in section 1.3) is carried out by means of a lower priority interrupt line (interrupt3) for the DSP, because the I/O data (manipulated by the interruptI) can be lost, while the PC can wait until the DSP will answer.

For other details, [17] is referred.

To conclude this section, [21] is referred as being useful to avoid redescovering the wheel during the project development. Nevertheless, what is intended to be used must be firstly tested.

A few useful ideas can be as well found in [20]. Here it is described a PC operating system, which is a much more complex environment.