Design and Implementation of a Multiprocessor Architecture for Adaptive Digital Filters

by

R.L. Barnes

Center for Communications and Signal Processing
Electrical and Computer Engineering Department
North Carolina State University

July 1986

CCSP-TR-86/14
Abstract

BARNES, RUSSELL L. Design and Implementation of a Multiprocessor Architecture for Adaptive Digital Filters. (Under the direction of Sasan Houston Ardalan.)

The purpose of this project was to implement a multiprocessor architecture that can be used in an adaptive digital filtering development system.

Our architecture consists of a master processor, which controls data acquisition and performs scalar data operations, and a tightly coupled SIMD slave multiprocessor, which performs vector data operations. Interprocessor communication has been optimized for the vector inner product calculations encountered in transversal adaptive filters. The system was implemented with TMS32010 digital signal processing chips. Results are presented for real-time system identification, echo cancellation, adaptive equalization, and signal separation. The system's performance is analyzed for the LMS, the Fast Kalman and the normalized Fast Transversal Filter algorithms.

Our system demonstrates the efficiency of a tightly coupled network of processors optimized for a particular class of DSP algorithms. Extensive analysis of our system has revealed relatively minor modifications to the TMS32010 that could double the throughput of our adaptive filters. Furthermore, we propose a modular interconnection network that would allow large scale integration techniques to greatly reduce system hardware.
## TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Figures</td>
<td>iv</td>
</tr>
<tr>
<td>Chapter 1 - Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2 - Motivation for Implementing BRACELET</td>
<td>4</td>
</tr>
<tr>
<td>2.1. Review of Adaptive Filtering Theory</td>
<td>4</td>
</tr>
<tr>
<td>2.2. Adaptive Filtering Applications</td>
<td>7</td>
</tr>
<tr>
<td>System Identification</td>
<td>8</td>
</tr>
<tr>
<td>Adaptive Equalization</td>
<td>9</td>
</tr>
<tr>
<td>Noise Cancellation</td>
<td>10</td>
</tr>
<tr>
<td>Linear Prediction</td>
<td>11</td>
</tr>
<tr>
<td>Signal Separation</td>
<td>12</td>
</tr>
<tr>
<td>2.3. System for Developing Adaptive Digital Filters</td>
<td>13</td>
</tr>
<tr>
<td>Value of an Adaptive Digital Filter Development System</td>
<td>13</td>
</tr>
<tr>
<td>Proposed Adaptive Digital Filter Development System</td>
<td>15</td>
</tr>
<tr>
<td>Chapter 3 - Implementation Issues of Transversal Adaptive Filters</td>
<td>18</td>
</tr>
<tr>
<td>3.1. The LMS Algorithm</td>
<td>18</td>
</tr>
<tr>
<td>Data Storage Requirements</td>
<td>19</td>
</tr>
<tr>
<td>Transversal Filter Operations</td>
<td>19</td>
</tr>
<tr>
<td>Adaptive Filter Operations</td>
<td>22</td>
</tr>
<tr>
<td>3.2. Adaptive Digital Filtering with the TMS32010 DSP Chip</td>
<td>23</td>
</tr>
<tr>
<td>Overview of the TMS32010</td>
<td>23</td>
</tr>
<tr>
<td>Data Storage Requirements</td>
<td>27</td>
</tr>
<tr>
<td>Transversal Filter Operations</td>
<td>27</td>
</tr>
<tr>
<td>Adaptive Filter Operations of the LMS Algorithm</td>
<td>28</td>
</tr>
<tr>
<td>3.3. Factors Concerning a Multiprocessor Implementation</td>
<td>29</td>
</tr>
<tr>
<td>More Complex Algorithms</td>
<td>30</td>
</tr>
<tr>
<td>Larger Filter Sizes</td>
<td>30</td>
</tr>
<tr>
<td>Serial and Parallel Data Flow</td>
<td>31</td>
</tr>
</tbody>
</table>
List of Figures

Fig. 1 - Signal Flow Diagram of a Transversal Adaptive Filter ........................................... 5
Fig. 2 -- Generalized Adaptive Filter Signal Flow Diagram .................................................. 7
Fig. 3 -- Adaptive Filter Configured for System Identification ........................................... 8
Fig. 4 -- Adaptive Filter Configured for Adaptive Equalization ........................................... 9
Fig. 5 -- Adaptive Filter Configured for Noise Cancellation .............................................. 10
Fig. 6 -- Adaptive Filter Configured for Linear Prediction ................................................. 11
Fig. 7 -- Adaptive Filter Configured for Signal Separation ............................................... 12
Fig. 8 -- Proposed Adaptive Digital Filter Development System ......................................... 16
Fig. 9 -- Transversal Filter Operations .................................................................................. 20
Fig. 10 -- Parallel Addition Using a Binary Tree .................................................................. 22
Fig. 11 -- Adaptive Operations of the LMS Algorithm ......................................................... 24
Fig. 12 -- Internal Structure of the TMS32010 ..................................................................... 26
Fig. 13 -- Slave Multiprocessor ......................................................................................... 35
Fig. 14 -- Master Processor ................................................................................................. 36
Fig. 15 -- Global Input Bus .................................................................................................. 40
Fig. 16 -- Pipelined Interconnections .................................................................................. 40
Fig. 17 -- Ring and Pipelined Interconnections .................................................................... 40
Fig. 18 -- Accumulation of Partial Sums with a Ring Network ............................................ 42
Fig. 19 -- Expanded Binary Tree Structure ......................................................................... 43
Fig. 20 -- Accumulation of Partial Sums with BRACELET Network ..................................... 45
Fig. 21 -- BRACELET Interconnection Network ................................................................. 46
Fig. 22 -- Dedicated 16-Bit Latches for Links k=1 and k=2 ................................................. 46
Fig. 23 -- Global Memory Organization ............................................................................... 48
Fig. 24 -- Local Memory Organization ............................................................................... 48
Fig. 25 -- Synchronization Circuit ....................................................................................... 50
Fig. 26 -- Synchronization Timing Diagrams ...................................................................... 51
Fig. 27 -- System Identification Demonstration .................................................................... 62
Fig. 28 -- Adaptive Filter Configuration for Echo Cancellation ....................................... 63
Fig. 29 -- Echo Cancellation Demonstration ....................................................................... 64
Fig. 30 -- Adaptive Equalization Demonstration ............................................................... 65
Fig. 31 -- Signal Separation Demonstration ....................................................................... 67
Chapter 1 — Introduction

The advances made in microelectronics technology in the last two decades have allowed digital signal processing (DSP) research to mature from a mathematical curiosity to widespread practical applications. Exciting research continues to address the problems associated with today's telecommunication network in terms of signal transmission, signal detection, and speech processing. DSP applications in these areas include identifying unknown systems, equalizing the effects of transmission channels, separating narrowband and wideband signals, and cancelling noise and echo effects.

The adaptive digital filter has been gaining acceptance to perform these operations largely due to the simplicity of the least mean squares (LMS) algorithm. Many custom LSI and VLSI circuits have been proposed and fabricated to meet the speed and throughput requirements of real-time applications. Within the last three years, however, the increased availability of high speed programmable DSP chips has provided researchers with powerful tools for rapidly breadboarding real-time adaptive digital filters. Furthermore, much of today's research focuses on algorithms that perform better than LMS in terms of adaptation time and tracking ability at the expense of increased computational complexity. To meet the requirements of these large filters and more complex algorithms, the commercially available DSP chips may be used in innovative multiprocessor networks.

In this paper we present BRACELET — a fully operational multiprocessor currently being used in the research of adaptive digital filters. BRACELET was designed as the heart of a development system that would allow us to investigate parallel processing techniques for implementing adaptive digital filters and to evaluate the real-time performance of adaptive digital filtering algorithms. The basic architecture consists of a single master processor which controls an array of four slave processors. While the
BRACELET architecture is independent of the specific processor chosen, our implementation uses the Texas Instruments TMS32010 DSP chip. A common global program memory simultaneously provides the slave TMS32010s with identical instructions so they may operate in a single-instruction/multiple-data stream (SIMD) mode. On-chip data registers contain the filter coefficient vector and the delayed input sample vector. External local memory for each processor allows us to implement fast table-oriented algorithms or to dynamically configure the system in a multiple-instruction/multiple-data stream (MIMD) mode. Our interconnection network optimizes the interprocessor communications necessary for a multiprocessor to implement transversal adaptive filters. Through effective integration of hardware and software, we clearly demonstrate significant versatility with the BRACELET architecture while still achieving the high performance necessary for real-time operation.

Before we present our design, we briefly review transversal adaptive filters and describe some of their useful applications. We then discuss the value of having a development system that would allow us to design adaptive filtering algorithms and applications and to analyze their real-time performance. Next we present the BRACELET architecture and explain how we applied well-known parallel processing techniques to optimize its hardware and software for transversal adaptive filters. To validate the use of such a tightly coupled multiprocessor for adaptive filtering applications, we provide results of the real-time operation of BRACELET configured for system identification, adaptive equalization, and signal separation, and we analyze the real-time performance of the LMS algorithm, the Fast Transversal Filter algorithm, and the Fast Kalman algorithm mapped onto the BRACELET architecture. We use this experience with BRACELET to suggest modifications to the TMS32010 that would greatly enhance its performance of adaptive filtering algorithms. Finally, to encourage further research in this area, we propose an
improved interconnection network that promises modular expansion suitable for large scale integration techniques.
Chapter 2 – Motivation for Implementing BRACELET

In his survey of highly parallel architectures and algorithms for speech analysis, L. J. Siegel [17] emphasizes the practicality of designing parallel systems for a particular class of algorithms rather than designing a large scale general purpose parallel computer. Since real-time performance analysis is one of our main objectives, we wish to minimize the burden that our architecture places on the algorithms being tested. We therefore optimized our multiprocessor architecture for a single class of DSP algorithms (transversal adaptive filters), and simultaneously developed software to perform specific algorithms in a variety of applications.

To appreciate our motivation toward this design philosophy, the reader should be familiar with transversal adaptive filters and their applications. This paper does not treat adaptive filtering theory in detail, but the reader is encouraged to refer to the many papers that do expound on adaptive filtering theory and the derivation of particular adaptive algorithms [3,4,6,8,9]. McCool and Widrow [8] provide an excellent review of the principles and applications of adaptive filters, which we abbreviate here. We also examine several general configurations that utilize the same basic adaptive filter structure to produce such useful functions as system identification, adaptive equalization, noise cancellation, linear prediction, and signal separation. We then discuss the value of having a development system to help us design and test adaptive filtering algorithms.

2.1. Review of Adaptive Filtering Theory

Fig. 1 illustrates the signal flow diagram of a transversal adaptive filter. The filter's impulse response is defined by an array of N coefficients whose value at sampling time T is the column vector
Fig. 1 -- Signal Flow Diagram of a Transversal Adaptive Filter
A stream of digital input samples, denoted by \( x \), is fed to the adaptive filter from an A/D converter. At sampling time \( T \), the current sample and the previous \( N-1 \) samples are defined by the column vector

\[
\mathbf{X}(T) = \begin{bmatrix} x_0(T), x_1(T), \ldots, x_i(T), \ldots, x_{N-1}(T) \end{bmatrix} \tag{2.2}
\]

Using our notation, the current input sample is \( x_0(T) \), and the relation between each sample and its associated delay is \( x_i(T) = x_0(T-i) \). Vectors \( \mathbf{W}(T) \) and \( \mathbf{X}(T) \) are combined by multiplying their corresponding elements and then accumulating these \( N \) inner products. This process, yielding the filter output \( y(T) \), may be expressed by the relation

\[
y(T) = \sum_{i=0}^{N-1} w_i(T) x_i(T) \tag{2.3}
\]

or in vector notation as

\[
y(T) = \mathbf{W}^T(T) \mathbf{X}(T) \tag{2.4}
\]

An error value \( e(T) \) is derived by subtracting the filter output from a sampled desired response signal \( d(T) \)

\[
e(T) = d(T) - y(T) = d(T) - \mathbf{W}^T(T) \mathbf{X}(T) \tag{2.5}
\]

This error value is then used to readjust the filter coefficients so as to minimize future error values. In essence, the filter coefficients weight the delayed input samples to estimate, or predict, the desired response. The process of minimizing the error between this prediction and the desired response is the core of adaptive algorithms and varies with each algorithm.

As the algorithm minimizes the error, the filter coefficients converge to the
impulse response of the system whose stream of input samples is \( x \) and whose stream of output samples is \( d \). Thus, if the adaptive filter is long enough, it can approximate any impulse response and any frequency response. The penalty for greater length, in addition to greater complexity, is a longer adaptation time or greater misadjustment. For many applications an optimal length can be found.

2.2. Adaptive Filtering Applications

We have generalized the adaptive filter signal flow diagram in Fig. 2 to illustrate how different I/O connections can perform a variety of useful applications. In the following discussion and in the corresponding figures, the input to the adaptive filter is \( x \), the filter output is \( y \), the desired response is \( d \), and the prediction error is \( e \). The impulse response of the adaptive filter is the coefficient (or weight) vector \( W \). The desired information for a particular application may be the weight vector, the filter output, or the prediction error.

Fig. 2 -- Generalized Adaptive Filter Signal Flow Diagram
System Identification

Fig. 3 shows the adaptive filter configured for system identification. As stated above, the filter coefficients approximate the impulse response of the system whose input is \( x \) and whose output is \( d \). The additive noise \( n \) does contribute to the error in the adaptive weight vector but should not prevent convergence of the filter. While the weight vector \( W \) approximates the impulse response of the unknown system, some practical applications of system identification do not actually report this impulse response. For instance, in the cancellation of echoes in telephone circuits, the desired echo suppressed signal may be obtained directly from the prediction error \( e \) [9,18].

![Fig. 3 - Adaptive Filter Configured for System Identification](image-url)
Adaptive Equalization

By simply exchanging the connections for the input $x$ and the desired response $d$ of the filter in Fig. 3, we can model the inverse of the system, as shown in Fig. 4(a). Once the adaptive filter converges, the original signal $p$ passes first through a system and then through the inverse of the system, equalizing the effect of the first system. If a broadband pilot signal is sampled for $p$, a truncated but stable approximation of the optimal equalization filter will be realized. Once the equalizing filter has converged, the coefficients may be frozen and the information signal may then be transmitted. In practical applications, such as equalizing the effect of a transmission line, we cannot receive an uncorrupted pilot signal, so the filter must have a local copy of the sampled pilot signal. Redrawing Fig. 4(a) to reflect this practical consideration produces the filter configuration shown in Fig. 4(b).

![Diagram](attachment:fig4.png)

Fig. 4 -- Adaptive Filter Configured for Adaptive Equalization
Noise Cancellation

Fig. 5 shows an adaptive system sampling an information signal $s$ that has been corrupted by noise $n_0$. The adaptive filter can estimate noise $n_0$ using an available noise source $n_1$. By subtracting this estimate from the original information signal, the filter can cancel the contaminating noise. McCool and Widrow [8] show that the only requirements are that $n_0$ and $n_1$ are correlated and that $s$ is uncorrelated with $n_0$ and $n_1$. A typical application of an adaptive noise canceller is to eliminate the power line interference that obscures signals recorded by sensitive instruments such as electrocardiograms [4,8]. Another practical application is in mobile voice communications where speech signals can be corrupted by background noise from a helicopter or an engine [4].

![Fig. 5 -- Adaptive Filter Configured for Noise Cancellation](image-url)
Linear Prediction

Prediction filters provide estimates of statistically stationary signals at future times. As shown in Fig. 6, the input to the filter is a delayed copy of the original sampled signal $s$. The filter attempts to transform this delayed input stream into the original undelayed input stream received at the desired response input. In essence, the filter is simply equalizing the effect of the delay, which we regard as a prediction. The filter coefficients are then copied into a slave filter which operates on the undelayed input stream to predict the sampled signal's future value. Linear prediction is used to efficiently encode correlated signals, such as speech, with a fewer number of bits per sample to reduce the bit rate required for communication [4].

Fig. 6 – Adaptive Filter Configured for Linear Prediction
Signal Separation

Fig. 7 illustrates how the adaptive predictor of Fig. 6 can be used without the slave filter to separate a composite signal into its narrowband and broadband components. A broadband signal has a narrow autocorrelation function and is therefore hard to predict from past samples; a narrowband signal has a wide autocorrelation function and is therefore relatively easy to predict from past samples. Using these principles, a filter designer may choose a delay that allows the adaptive filter to predict only the narrowband component of the original composite signal. The broadband component will appear in the error stream \( e \) after the narrowband component has been cancelled by the subtraction. This technique can be used to resolve signals of small amplitude in the presence of a signal of large amplitude. Adaptive filter signal separation has been shown to be competitive with conventional Fourier analysis and even more practical in certain applications [8].

Fig. 7 – Adaptive Filter Configured for Signal Separation
2.3. System for Developing Adaptive Digital Filters

The variety of applications described in the previous section illustrates the versatility of the adaptive digital filter. By redefining the inputs and outputs, we have the ability to implement different functions while preserving the basic adaptive filter structure. Furthermore, the adaptive filter structure is completely independent of the specific algorithm chosen. This orthogonality permits the user of a flexible DSP system to either test a particular algorithm in a variety of applications or test an application using a variety of algorithms. In this sense, an application program can be written as a generic shell independently of any particular algorithm, and an algorithm can be written as a module independently of any particular application. A filter designer simply inserts the algorithm module into the application shell to build the desired test program. Any specific filter characteristic, such as the filter size, can be included as a parameter of the algorithm module thus allowing a quantitative measurement of its effect on the filter's performance.

We can capitalize on this versatility with a development system that allows us to design and test adaptive digital filters. We have already described the many applications capable with such a system; we now illustrate its value to adaptive digital filter design and outline how our multiprocessor will be used to implement a real-time adaptive digital filter development system.

Value of an Adaptive Digital Filter Development System

In their discussion of adaptive systems, Claasen and Mecklenbrauker [4] emphasize the fact that today's adaptive systems are based on a great deal of prior knowledge about the signals being processed. Adaptive filters are designed for a specific type of input signal (binary data, speech, etc) for specific interferences (additive white
noise, sinusoidal signals, etc) and for a specific type of transmission channel (linear, finite impulse response). The only unknowns at the time of operation are some parameters that specify the actual signal or system characteristics within the given class. Serious performance degradation, however, can be expected if this prior knowledge proves to be incorrect. For example, the assumption that an interfering signal is periodic can be very helpful when designing an interference cancellation system; but if the interfering signal proves to be aperiodic, the algorithm will nevertheless find a periodic signal and try to cancel it. We must therefore carefully validate all assumptions underlying our decisions regarding the system design.

As we develop our system, we must also ensure that our design conforms to a reasonable set of performance specifications that achieves a balance between ideal results and technical feasibility. Since a human receiver is often the judge of the final output, perceptual factors should be considered when specifying this performance quality. Unfortunately, perceptual factors are usually impossible to optimize mathematically other than by minimizing spectral components of the error signal. Much of our research therefore attempts to derive and implement algorithms that optimize this elusive quality criterion while still retaining a certain robustness in the sense that small deviations from the optimum will not lead to substantial deterioration of the system's performance.

A development system would be instrumental during these two phases of the design process. A designer could repeatedly alter his filter and the input signals to test his assumptions of expected signal characteristics and to correlate perceived quality with statistical results. He could also judge more accurately the exact hardware and software required for a particular implementation through the fast development of working prototypes. A researcher would have the opportunity to evaluate the performance of real filters
operating on real signals so he could more completely determine the extent to which his simulations deviate from actual conditions. Indeed, such a system would be valuable to both university and industrial development work.

Important areas that could be studied effectively on an adaptive filter development system include the effects of finite register lengths on filter performance, table lookup structures for dealing with nonlinear channels, and systems for tracking nonstationary signals with relatively fast statistical variations. Other practical research deals with systems in which the algorithm or quality criterion themselves can be adapted to specific signal conditions, such as the double talker detector in telephone echo cancelling. More powerful and robust systems, such as lattice filters and frequency domain adaptive filters, are also becoming practical despite their greater complexity. Much work still must be done for us to reach an understanding of the performance of these algorithms comparable to our comprehension of the popular LMS algorithm.

Proposed Adaptive Digital Filter Development System

The BRACELET multiprocessor is shown in Fig. 8 incorporated as the heart of our proposed development system. Interacting through a host computer, the user can construct a library of test routines and test signals for the development of real-time adaptive filtering algorithms. A filter designer could specify an adaptive filtering application, a particular algorithm, a set of filter characteristics (e.g. the number of filter taps), and the information he wishes BRACELET to report to the host computer. The development system would then automatically generate the machine code required by BRACELET's processors and download this code to BRACELET for execution. The user could also specify whether the input signals are retrieved from the system library or input from A/D
converters and whether the desired information is stored in the system library or output through D/A converters to a speaker or to a test instrument such as a spectrum analyzer. Furthermore, the host computer could provide elaborate real-time graphical displays of the filter's performance. With a programmable system, an endless number of options are available to the user.

Even though we have stressed real-time analysis, the development system would also include a mode of operation in which the user could specify the exact number of samples to be run. In this simulation mode, the user could step through each line of code or through one or more sample periods while examining a particular filter's performance on a known signal. With this technique, the filter designer would find it easy to compare different algorithms using identical signals or to adjust filter characteristics to achieve maximum filter performance.

Fig. 8 -- Proposed Adaptive Digital Filter Development System
Such a development system would be a definite advantage to our participation in the exciting area of adaptive digital filter research. For the complex algorithms, high throughput, and variety of performance analysis routines we wish to implement, the BRACELET architecture must have considerable processing power and versatility. The presentation of the BRACELET architecture in the next chapter explains how our multiprocessor does in fact achieve the necessary power and flexibility to realize an effective adaptive digital filter development system.
Chapter 3 – Implementation Issues of Transversal Adaptive Filters

The adaptive filtering algorithms that we intend to implement ultimately dictate our design of the BRACELET architecture. Before we can explore the details of applying parallel processing techniques to this design, we must first identify those parallel and serial data operations typically found in adaptive filtering algorithms. We concentrate on the implementation issues of the widely-used LMS algorithm because many of the more complex adaptive filtering algorithms may be programmed using the same hardware and basic software structures defined by the much simpler LMS algorithm. We then briefly describe the TMS32010 DSP chip used in our design and show that it efficiently implements the structures required for high speed digital filtering. Finally, we explain our need for a multiprocessor design for BRACELET and outline the problems specific to a multiprocessor implementation of transversal adaptive filters.

3.1. The LMS Algorithm

As we discussed previously, an adaptive filter produces an output \( y(T) \) from a weighted sum of delayed input samples,

\[
y(T) = W^T(T) X(T)
\]

(3.1)

The filter output is an attempt to predict a desired response \( d(T) \) for which the prediction error \( e(T) \) is given by

\[
e(T) = d(T) - y(T)
\]

(3.2)

The manner in which the filter coefficient vector \( W(T) \) is updated to minimize this prediction error defines the particular algorithm being performed. The LMS algorithm, devised by Widrow and Hoff, is an approximation of the gradient method for minimizing the mean
square prediction error. The filter coefficients are updated each sample period by the vector relation

\[ W(T+1) = W(T) + \alpha e(T)x(T) \]  

(3.3)

where \( \alpha \) is the feedback gain constant that controls the stability and rate of convergence of the filter coefficients. For our work, we assume that \( \alpha \) is always properly chosen to meet the given performance specifications. Many papers have already been published regarding the theory and performance of LMS [3,4], but for now we are interested solely in the implementation issues.

**Data Storage Requirements**

The data storage requirements for the LMS algorithm are very straightforward and common to the direct form representation of any non-adaptive transversal filter [15]. The hardware must physically retain the filter coefficients and an equal number of delayed input samples as indicated by Eq. 2.1 and Eq. 2.2. Assuming the filter size is \( N \), we therefore require \( 2N \) data registers for vector storage. Extra registers contain the scalar values for the desired response, the prediction error, the filter output, the feedback gain constant, and other miscellaneous scaling factors. More complex algorithms may require more vector and scalar storage.

**Transversal Filter Operations**

The data operations of the transversal filter are illustrated by the signal flow diagram in Fig. 9. Before any calculations are performed, the filter must first shift the \( N \) previously input values through the delayed input sample vector,
Fig. 9 -- Transversal Filter Operations
\[ X(T) = \left[ x_0(T), x_0(T-1), \ldots, x_i(T-1), \ldots, x_{N-2}(T-1) \right]^T \] (3.4)

discarding the last sample \( x_{N-1}(T-1) \) and accepting the current input value \( x_0(T) \). We redefine this vector in terms of the current sampling time \( T \) as

\[ X(T) = \left[ x_0(T), x_i(T), \ldots, x_{i+1}(T), \ldots, x_{N-1}(T) \right]^T \] (3.5)

In a serial machine, \( N \) steps are required to delay the \( N \) input samples by individually shifting each vector element \( x_i(T-1) \) into the next higher vector position \( x_{i+1}(T) \). This time could be reduced to a single step with a circular queue implementation in which the contents of the registers are not actually shifted, but instead a pointer identifies the register containing \( x_{N-1}(T-1) \) so it may be discarded and replaced by the current input sample \( x_0(T) \). For maximum speed, a parallel machine can take advantage of the natural pipelined structure of the vector shift operation to shift all samples through \( N \) separate delay elements in only one step.

The dominating arithmetic operations of the transversal filter are multiplication and addition. Each delayed input sample \( x_i(T) \) is multiplied (or weighted) by the corresponding filter coefficient \( w_i(T) \). These inner products must then be accumulated to produce the filter output. Calculating the filter output on a serial machine therefore requires \( N \) multiplication and \( N-1 \) addition steps. While a parallel machine can perform the \( N \) multiplications in a single step, the addition is usually implemented as a binary operation (binary in the sense that two numbers are being added together in one operation) and requires a minimum of \( \log_2 N \) steps using a binary tree structure. The tree shown in Fig. 10 illustrates eight numbers being added in \( \log_2 8 = 3 \) addition steps.
Adaptive Filter Operations

Once the transversal filter operations have produced the filter output, the adaptive algorithm subtracts this prediction from the desired response to determine the prediction error (Eq. 3.2). Most algorithms then multiply this error value by a feedback gain constant that varies the time constant and the stability of the adaptation process [4]. Both of these operations are scalar and cannot be performed until all the inner products of the transversal filter have been accumulated. Because the normal form transversal filter contains no delay elements in the accumulation process and the next state of the filter coefficient vector is dependent upon this scaled error value, we cannot tolerate appreciable delays in the accumulation process. We find this transition from parallel to serial flow to be the major bottleneck in adaptive digital filter data flow.

Fig. 10 -- Parallel Addition Using a Binary Tree
In the LMS algorithm defined by Eq. 3.3 and represented graphically in Fig. 11, the scalar adjustment factor $\alpha e(T)$ is multiplied by each element of the delayed input sample vector, and these products are then added to the corresponding element in the filter coefficient vector. Once the transition from serial to parallel flow has been made by broadcasting the adjustment factor to each delayed input sample / filter coefficient pair, a parallel machine can take advantage of the inherent parallelism to perform the multiplication operations in one step and the addition operations in another step to adapt the $N$ filter coefficients simultaneously. A serial machine requires $N$ separate multiplication and $N$ separate addition steps to perform this same adaptation process.

3.2. Adaptive Digital Filtering with the TMS32010 DSP Chip

The Texas Instruments TMS32010 was introduced as the first member of TI's TMS320 family of high speed DSP chips. In this section, we briefly describe the internal structure of the TMS32010 and show how it has been optimized for digital filtering. We also relate the capabilities of this processor to the implementation issues of adaptive filters in terms of data storage, transversal filter operations, and adaptive filter operations. We then express the need for a multiprocessor architecture and outline the associated obstacles that must be overcome for a successful implementation.

Overview of the TMS32010

The structure of the TMS32010 takes advantage of the Harvard architecture, in which program and data memory are separate, to permit a full overlap of the fetch and execution of instructions. This organization is evident in the functional diagram of Fig.
12. TI further modified the Harvard architecture to allow transfers between program memory and data memory so that large tables could be stored in external program memory. The pipelined architecture of the TMS32010 enables it to execute five million operations per second (5 MOPS). These operations have been incorporated into an instruction set optimized for DSP applications allowing digital filtering at the rate of 2.5 million samples per second.

To achieve this high processing speed, the TMS32010 implements many functions in hardware that are usually performed by software. For example, the TMS32010 is capable of multiplying two 16-bit, two's complement integers to produce a 32-bit, two's complement product in one 200 ns instruction cycle. A 32-bit arithmetic logic unit (ALU) can manipulate this product with the 32-bit accumulator for fast double precision arithmetic. A hardware barrel shifter can shift data for scaling as it is sent to the ALU. Auxiliary registers can be automatically incremented and decremented during ALU operations for single cycle manipulation of internal data tables.

The TMS32010 can address 4K words of external program memory and 144 words of internal data memory (sectioned into a 128 register page and a 16 register page). The 16-bit data bus can channel I/O at a burst rate of 40 million bits per second (40 Mbps) through any one of eight input or eight output ports. A reset, a vectored interrupt, and an integrated polling input and jump instruction allow external synchronization of software routines for applications involving the acquisition of sampled data or for multiprocessor applications. Appendix A contains additional information concerning the instruction set and timing diagrams pertinent to our multiprocessor design. With this information, we can relate the capabilities of the TMS32010 to the general requirements of transversal adaptive filters and the specific requirements of the LMS algorithm.
Fig. 12 -- Internal Structure of the TMS32010
Data Storage Requirements

With the TMS32010, both vector and scalar values are stored in either internal data registers or external random access memory (RAM). The TMS32010 is capable of very fast access to 128 internal data registers. Each processor therefore has the internal storage capacity to implement a 64 tap filter. Even though the modified Harvard architecture provides for the storage of data in program memory, using the external program memory for this purpose would be extremely inefficient. The TMS32010 requires an extra 3 cycle instruction (TBLR or TBLW) for each data access to this external memory space. Furthermore, the accumulator must be reserved to address this memory space, destroying the pipelined operation of the TMS32010's internal data paths.

Transversal Filter Operations

The TMS32010 efficiently realizes the transversal filter operations of Fig. 9 with its pipelined multiplier, ALU, and accumulator. An inner product accumulation routine steps through the $X(T)$ vector from the $x_{N-1}(T)$ element to the $x_0(T)$ element so the vector shift function may be performed at the same time. As the vectors are traversed, the TMS32010 prepares to multiply elements $x_i(T)$ and $w_i(T)$ while adding to the accumulator the product of elements $x_{i+1}(T)$ and $w_{i+1}(T)$ computed during the previous instruction cycle. Referring to Fig. 12 and the instruction set listed in Appendix A, we find we can implement the shift, multiply, and accumulate operations in two instruction cycles for each delayed input sample / filter coefficient pair. The LTD instruction has three simultaneous effects: the previously computed product $x_{i+1}(T)w_{i+1}(T)$ in the P register is added to the running sum in the accumulator; the T register is loaded with $x_i(T)$ for the next multiply operation; and sample value $x_i(T)$ is shifted into the register.
that previously contained $x_{i+1}(T)$. The MPY instruction then multiplies $x_i(T)$ in the T register by $w_i(T)$ in its data register and places the product in the P register. For a filter with N coefficients, these two instructions are repeated N times, allowing the TMS32010 to perform the transversal filter operations in $O(2N)$ instruction cycles. As with any pipelined structure, additional cycles are needed to initialize or fill the pipe and to clear the pipe once the last set of values has been inserted.

At first glance, the process of accumulating N 32-bit products seems to require an accumulator wider than 32 bits. This conclusion would be true if we were designing our system to accommodate the maximum possible accumulated value. We instead use our knowledge of the signal statistics to design for the expected number of bits in the final sum. Since the filter output is attempting to predict another input, we only need to match the number of bits in the accumulator with the number of bits in the desired response sample (we are assuming any error is relatively small which is the case for proper filter operation). Should we expect to exceed the capacity of the 32-bit accumulator, we can use the data shifting abilities of the TMS32010 to scale our values for valid accumulation. The TMS32010 can also be set in an overflow mode with the SOVM instruction. In this mode, an overflow causes the accumulator to be loaded with the maximum positive or negative value rather than a grossly erroneous overflow value.

**Adaptive Filter Operations of the LMS Algorithm**

The adaptive filter operations illustrated by Fig. 11 first involve subtracting the accumulated inner product values from the desired response value. This prediction error is then multiplied by the feedback gain constant to produce the adjustment factor $\alpha e(T)$. In the TMS32010, the LT instruction is used to load the T register with $\alpha e(T)$. The LAC
instruction loads the accumulator with the filter coefficient \( w_i(T) \) that is to be adjusted. Next the MPY instruction multiplies \( \alpha e(T) \) in the T register by the corresponding delayed input sample \( x_i(T) \) stored in its data register and places the product in the P register. The APAC instruction then adds this product to the filter coefficient \( w_i(T) \) waiting in the accumulator to produce the new filter coefficient value \( w_i(T+1) \). Finally, this new coefficient is stored in its respective data register. Since the T register does not need to be reloaded with \( \alpha e(T) \) for each filter coefficient adjustment, the TMS32010 can perform the LMS adaptive operations on a filter with \( N \) coefficients in \( O(4N) \) instruction cycles.

The total time required for the TMS32010 to perform the LMS algorithm on an \( N \) tap filter is \( O(6N) \). With a 200 ns instruction cycle, the TMS32010 averages approximately 1.2 \( \mu \)s to perform these operations on a single filter tap. A single TMS32010 is capable of storing a 64 tap filter. Ignoring the overhead associated with data input, output, and the pipelined operations, a 64 tap filter can be performed within a sample period of approximately 77 \( \mu \)s. Many of our applications deal with speech signals that are sampled every 125 \( \mu \)s. Obviously, the LMS algorithm does not tax the computational power of the TMS32010 for these applications. In fact, the storage limitations of the TMS32010 are exceeded before its processing limitations for the LMS algorithm.

### 3.3. Factors Concerning a Multiprocessor Implementation

While the TMS32010 is capable of performing the LMS algorithm on a 64 tap filter with an 8 kHz sampling frequency, our development work involves even larger filters and more complex adaptive algorithms. We are therefore motivated to design the BRA-CELET architecture as an array of TMS32010s in a multiprocessor network. Our design
must take into account these more complex algorithms, the larger filter sizes, and the serial and parallel data flow necessary for an efficient and versatile system.

More Complex Algorithms

Our discussion has been emphasizing LMS algorithm implementations. More complex algorithms, such as the Fast Transversal Filter (FTF), promise substantial improvements in transient behavior in comparison with the LMS algorithm [3]. The FTF algorithm uses three additional transversal filters to achieve faster convergence of the adaptive filter's impulse response with better tracking stability. Not only does the FTF algorithm require 2.5 times the vector storage of the LMS algorithm, but the normalized FTF also requires two $O(2N)$ inner product accumulations, five $O(4N)$ filter adaptations, and six $O(3N)$ vector normalizations in which an entire vector is multiplied by a scalar value. The better performance of FTF is achieved by executing $O(42N)$ instruction cycles for an $N$ tap filter. The effect of this complexity becomes apparent when the FTF and LMS algorithms are compared -- an 8 kHz sampling frequency allows only 14 filter taps to be processed by FTF compared with 104 taps for LMS. An implementation based on the TMS32010 must therefore involve multiprocessing techniques to perform an FTF filter of useful size.

Larger Filter Sizes

Another motivating factor for a multiprocessor implementation is the important application of system identification to cancel the echo effects heard in telephone circuits. These filters typically require 128 or 256 coefficients [9]. Even if the TMS32010 were
capable of storing 128 coefficients and an equal number of delayed input samples in the internal data registers, it would require approximately 154 μs to perform an LMS adaptive filter. A 256 tap filter would require 308 μs. We find that a single TMS32010 is simply unable to execute the number of operations required by these large filters within the 125 μs sample period of speech signals. Fortunately, the data storage for these algorithms is mainly in the form of vectors that may be easily divided among the processors in a multiprocessor network.

Serial and Parallel Data Flow

A versatile multiprocessor architecture must provide for any anticipated path for the flow of data through the system. In a versatile system, software should enable or disable particular paths as they are needed by a specific algorithm or application. While defining these data paths for a one processor system is rather trivial, for a system with four or more processors, more consideration should be given to the interconnection network since it can have a profound effect on the overall system performance.

To implement any transversal filter, the multiprocessor must have a means of shifting the delayed input samples. This data flow implies a pipelined structure in which each storage register can pass its contents to the next storage register. We have described the ability of the TMS32010 to perform this operation internally; a multiprocessor system, however, requires an interconnection network that reflects this pipelined structure so the elements of the vector may pass successively through each processor in the link.

The accumulation of inner products within a single processor is another trivial matter for the TMS32010 that gains importance in a multiprocessor system. Once a TMS32010 has accumulated its own inner products, it must combine its sum with the sum
of all other processors in the network so the filter output can be computed. As we have already emphasized, this transition from parallel flow to serial flow represents the major bottleneck in digital filter processing. Furthermore, the network must also accommodate the transition from serial flow back to parallel flow that occurs when a scalar value is broadcast to all processors for the adjustment of their respective filter coefficients. Fig. 11 shows that after each processor possesses this global adjustment factor, the adaptive operations for each filter coefficient are independent of all other coefficients permitting an efficient mapping to an array of processors.

The very nature of identical operations being performed on arrays of data define an SIMD approach to implementing transversal adaptive filters. However, the transitions between parallel and serial data flow of the LMS algorithm imply a corresponding shift between vector and scalar operations. More complex normalized algorithms require the scalar division of the prediction error by the average signal power of the input stream to eliminate an undesirable dependence of adaptation time on the input signal power [9]. Some of these algorithms, including the normalized FTF, also require time consuming square root operations [3]. Since a multiprocessor is not used effectively when performing these scalar calculations, our design must therefore consider modifications to a strict SIMD architecture to sustain processor efficiency.
Chapter 4 – Presentation of the BRACELET Architecture

Many researchers are exploring multiprocessor architectures that efficiently implement DSP applications. Some designs call for custom VLSI circuits [2,9] while other designs use commercially available components [12]. A development system requires components that are easily reconfigured for a variety of adaptive algorithms, applications, and filter sizes. For our applications, these components also need to meet the real-time throughput requirements for complex adaptive algorithms operating at high bandwidths. Furthermore, we wish to demonstrate rapid, low-risk development of adaptive systems for use in either a university or an industrial environment. All of these capabilities can be realized by an architecture based on an array of TMS32010s configured in a multiprocessor network.

4.1. Hardware Design

A preliminary study of this system by Faber and Miller [5] was presented by Miller and Alexander [10] at the 1985 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP 85). Their architecture features a number of TMS32010 DSP chips accessing a common global program memory and an interconnection network optimized for the pipelined and parallel segments of digital filtering algorithms.

The architecture that evolved from this early design consists of three main components: a slave multiprocessor that performs vector operations, a master processor that performs scalar arithmetic and control operations, and an interconnection network that provides for interprocessor communication. Other major design issues include the organi-
zation of the memory, the synchronization of the processors, and the use of a status port to increase BRACELET's versatility. In this section, we describe our design of these components and show how well-known parallel processing techniques have been applied to produce an effective multiprocessor architecture.

**Slave Multiprocessor**

The slave multiprocessor is configured in an SIMD structure to take advantage of the parallelism inherently found in digital filtering algorithms. Each processor operates on an equal portion of the delayed input sample and filter coefficient vectors. These subvectors consist of consecutive elements so each TMS32010 can efficiently delay the input samples while accumulating its share of the inner products as described in Section 3.2. A filter with $N$ coefficients mapped onto a system that consists of $P$ slave processors and a single master processor requires $M = \frac{N}{P}$ coefficients to be handled by each slave processor. Once each processor has accumulated its own inner products, the $P$ partial sums are combined using the interconnection network to produce the filter output. Each processor then independently performs the particular adaptive algorithm on its subvector of $M$ coefficients. Ignoring interprocessor communication for now, the slave multiprocessor can perform the LMS algorithm in $O(6M)$ instruction cycles.

The SIMD configuration requires each processor to obtain a global instruction and operate on local data. In the BRACELET architecture, the single instruction stream is implemented by having every slave processor execute source code that is stored in a single global program memory rather than having each slave processor simply execute a local copy of a global program. Fig. 13 shows that the address and control lines of only one slave processor actually interface with the global memory. Despite criticism that a
TMS32010-based multiprocessor could not operate from a single global memory [2], we have been able to synchronize the slave processors in our system with the master processor so they may simultaneously execute an instruction fetch and therefore accept identical instructions via the global input bus. Local data memory associated with each slave provides the multiple data stream consisting of the filter coefficients, the delayed input samples, and intermediate results of calculations.

While the BRACELET architecture is independent of the number of slave processors, our multiprocessor consists of four TMS32010s and is known as a four-processor system. In their feasibility study, Faber and Miller [5] wrote several untested programs that would implement adaptive digital filters using the LMS algorithm. One program distributed 32 filter taps on a four processor system to achieve a maximum sampling rate of 40 kHz. The same four processor system was able to be reprogrammed to accommodate a total of 232 taps with an 8 kHz sampling rate. The results of their study indicate a simple yet powerful system consisting of four slave TMS32010s is capable of providing the throughput and flexibility required by our proposed adaptive digital filter development system.

![Diagram of Slave Multiprocessor](image-url)

Fig. 13 -- Slave Multiprocessor
Master Processor

The adaptive filter development system we described in Section 2.3. features a host computer that provides a high-level interface to the human operator in an already familiar operating system. The controlling master processor isolates the slave multiprocessor from this host computer as shown in Fig. 14. The master relieves the host of any real-time control of the slaves, such as downloading programs to the slaves, initializing vector and scalar values, and starting and stopping the slaves. BRACELET can therefore function independently of the host computer once the necessary programs and data have been downloaded from the host to the master. Our implementation uses a Multibus-compatible prototyping board to further standardize the link between BRACELET and the host.

![Diagram of Master Processor](image-url)
The master also provides an interface between the slave multiprocessor and the external data ports. I/O can be channeled directly between the slaves and the data ports, or the master may intercept the data for processing before or after the slaves process the data. Fig. 14 illustrates the control the master has over the global input port, the filter input port, and the filter output port (since the interpretation of the actual data is determined by the software, these names serve mainly for convenience).

As we have already discussed, adaptive filtering algorithms are not entirely suited for parallel processing. To avoid redundant calculations on a multiprocessor, we partition the filtering algorithms into a stream of vector calculations and a stream of scalar calculations. The vector stream is mapped efficiently to the array of slave processors, and the scalar stream is mapped to the single master processor. For many algorithms, the scalar and vector operations can overlap each other for simultaneous processing to reduce the time required for data dependencies to be satisfied. The same master/slave interface meant for intercepting input and output data between the slaves and the data ports is used to pass data between the master and the slaves for this additional co-processing. Since few instruction cycles are actually needed to provide real-time control of the slaves, the master has ample time available to perform these scalar operations.

If used solely as a controller or as a memory management unit, the master processor is not restricted by the same processing qualities desirable in the slaves. Motorola's MC68000 is a very popular processor commonly used in multiprocessor architectures for these purposes [12]. However, by implementing the master with a fifth TMS32010, we can easily process scalar feedback values or larger filters with a DSP chip identical to the slaves. Furthermore, all of BRACELET's processors may have a single clock source, and
the slave processors may be synchronized to the controlling master processor's clock. By tracking a routine through each instruction cycle, a programmer can know exactly how the master program and the slaves' program parallel each other. Similarly, automatic software generating routines on the host computer can more easily synchronize the master program with the slaves' program for the most efficient mapping of data dependencies between vector and scalar co-processing.

Special purpose hardware needed to route data for any one particular algorithm is eliminated since the master processor can be programmed to implement the global broadcasting and pipelined data flow required by adaptive filters. The master also has the ability to process I/O and to effectively act as a fifth vector processor to perform additional adaptive filtering operations on its own subvectors. Additional uses of the master processor include monitoring the performance of the adaptive filter to verify filter convergence, to freeze the adaptive process when a programmed condition is met, or to rescue a diverging adaptive process.

Interconnection Network

The interconnection network must optimize the pipelined and parallel data transfers required by adaptive filters to delay input samples, combine accumulated inner products, and broadcast scalar values in our multiprocessor architecture. We consider the broadcasting of scalar values first, since the hardware required for this operation has already been defined. The global input bus in Fig. 13 and the global input port in Fig. 14 can be connected as in Fig. 15 to provide a path for broadcasting a scalar data value to the slave processors. When valid data is present at the global bus (from the master or directly from the port), the slaves need only execute an IN instruction to accept the data
in two instruction cycles. No handshaking signals are necessary to realize the synchronized transfer of our tightly coupled network since the master processor has full control of the multiprocessor and can ensure that valid data exists at the proper port and at the critical times.

Fig. 16 illustrates our four processor system with the pipelined connections that most efficiently effect the delay line. The input to Slave 1 is the same filter input port that the master arbitrates in Fig. 14. Slave 4 does not need an output path since the last sample in the delay is discarded. SIMD operation, however, forces all slaves to execute an OUT instruction followed by an IN instruction to realize a data transfer in four instruction cycles. As with the global data input port, valid data must appear at the filter input port when the slaves perform the common IN instruction.

An efficient method is needed to combine the accumulated inner products of all processors and then to broadcast the resulting filter output value to every slave processor. The simple ring network of Fig. 17 may be constructed from the pipeline of Fig. 16 by connecting Slave 4 to Slave 1. We have introduced a switch SW which represents logically the hardware control for the pipe and the ring. Miller [11] derives the name BRACELET from the ability to unclasp Slave 4 from Slave 1 to convert BRACELET's network from a ring structure for circular data flow to a pipelined structure with an input and an output. The connection between Slave 4 and the master is the same filter data output port arbitrated by the master in Fig. 14. Since the master and the slaves are all TMS32010s, the required instructions still consist of an OUT followed by an IN. Notice that with this network, the master processor can indeed be considered a fifth processor in the ring network. A filter designer can take advantage of this extra processing and storage resource by programming the master to perform inner product calculations for
Fig. 15 - Global Input Bus

Fig. 16 - Pipelined Interconnections

Fig. 17 -- Ring and Pipelined Interconnections
even larger filters.

A ring is an adequate network to accumulate inner products since each processor can place its value on the ring and then sum the values as they are shifted through the network. This process is depicted in Fig. 18. In 18(a), each processor outputs its own value to the next processor in the ring. Each processor then inputs the corresponding value from the ring and adds it to its own value to update the running sum. In 18(b), each processor outputs this new sum to the next processor in the ring but does not add the corresponding input value to the sum. Instead, the value is merely passed through the ring one more time before it is added to the running sum in 18(c). Broadcasting this final sum in 18(d) is not necessary, because all processors in a SIMD structure simultaneously compute the filter output value. Unfortunately, while only \( \log_2 N \) addition operations are required, the number of I/O operations increases linearly with the number of slaves and the number of inner product accumulations in a particular algorithm. The effect may be negligible for a small number of slave processors, but for larger systems and more complex algorithms, a larger portion of the sample period would be spent shifting data through the network.

The network proposed by Faber and Miller [5] connects an output port of each slave to the input ports of all slaves whose clockwise distance around the ring is \( 2^{k-1} \) for \( k = 1, 2, 3, \ldots, \log_2 P \) (where \( P \) is the number of slave processors). Notice the ring network corresponds to \( k = 1 \). Essentially, the tree structure for the parallel addition operation shown in Fig. 10 has been expanded to achieve the network shown in Fig. 19. The reader may recognize this structure as a subset of the PM2I network [16]. With this improved network, only \( \log_2 P \) communication stages are necessary for \( P \) slaves to accumulate an inner product. For a four processor network, the two transfer and addition stages are
Fig. 18 -- Accumulation of Partial Sums with a Ring Network
Fig. 19 -- Expanded Binary Tree Structure
illustrated in Fig. 20(a) and 20(b). As in the ring network, all processors contain the final value so no extra communication is required to broadcast a scalar value to every slave processor in Fig 20(c).

The entire interconnection network for the BRACELET multiprocessor is represented in Fig. 21. Each 16-bit communication path actually consists of two dedicated 8-bit latches with high impedance output states. The latches serving as I/O ports for links \( k=1 \) and \( k=2 \) are shown in Fig. 22. The common clock signal reduces the communication procedure to an OUT instruction to the appropriate set of latches followed by an IN instruction from the same set of latches. This procedure results in very little degradation to system throughput due to interprocessor communication.

**Memory Organization**

As previously mentioned, one slave addresses the global memory. The master is also able to control the global memory to download programs to the slaves. To prevent any contention for the memory, the master uses a multiplexer to select the source of the global program memory address, as shown in Fig. 23. Our design includes a synchronous counter that the master uses to address the global memory. The master can load the counter with an initial address, specify whether the address counter is to automatically increment or decrement, and then simply output a block of data using an OUT instruction without having to update the memory address. Slaves do not have the ability to write to global memory, hence the master does not need to read from global memory. Nevertheless, our implementation does give the master the ability to read from global memory, permitting the master to verify the validity of programs and data downloaded to the slaves before the master restarts the slaves.
(a) Transfer with $k = 1$ Link and Add

(b) Transfer with $k = 2$ Link and Add

(c) All Processors Contain $SUM = A + B + C + D$

Fig. 20 -- Accumulation of Partial Sums with BRACELET Network
Fig. 21 -- BRACELET Interconnection Network

Fig. 22 -- Dedicated 16-Bit Latches for Links $k = 1$ and $k = 2$
Our system also provides for the desirable ability to use table-oriented algorithms to calculate time consuming functions (such as trigonometric functions). Since a global data table design must handle the possible contention associated with multiple slave processors accessing different table locations in SIMD mode, our design instead provides each slave processor with enough private memory space to implement adequate look-up tables. BRACELET is thus capable of realizing a considerable improvement in throughput simply by representing a complex function as a table of values.

Our implementation of this local memory deserves a closer look. Each TMS32010 has only 144 words of on-chip data RAM (most of which is used by filter coefficients and delayed input samples). However, each TMS32010 does have access to 4K words of external program memory. Since the memory chips used in our design are each organized into $2K \times 8$ bits, we can easily designate the lower 2K memory space to be global memory and the upper 2K memory space to be local memory by using the most significant address bit to enable and disable the appropriate memory chips. Thus each slave is associated with a reasonable amount of memory space for private look-up tables. This organization, shown in Fig. 24, leaves only 2K words of global program memory; however, this memory space is still adequate for processing speech signals with an 8 kHz sampling frequency since the TMS32010 is only capable of executing a maximum of 625 instructions between samples. Furthermore, the upper 2K of table memory is actually in program memory, allowing extra instructions to be stored in and executed from local memory. If every slave has identical instructions in local memory, the multiprocessor is essentially configured with a 4K global program. In fact, while the slaves are executing a program from their local memories, the master can download new programs (or data) to the global memory allowing programs of virtually unlimited length.
Fig. 23 – Global Memory Organization

Fig. 24 – Local Memory Organization
An interesting side-effect results from the ability to store programs in this local memory originally meant for data tables. If each slave has a different set of instructions in local memory, we can dynamically configure our system in a synchronized MIMD mode of operation. To switch from SIMD to MIMD, the slaves merely branch from low (global) memory to high (local) memory. Switching from MIMD to SIMD is slightly more complicated since the slaves must be resynchronized, but the common clock signal makes this task easy. The shorter routines can be padded with NOPs so every routine executes in exactly the same number of instruction cycles. Simultaneous jumps into low memory then cause all slaves to be resynchronized. Other methods could use an interrupt or the integrated polling input and jump instruction BIOZ. We should emphasize that the system is not optimized for multiconfigurable SIMD / MIMD operation, but we should not ignore the possible uses of a MIMD mode of operation in multitasking DSP algorithms [12].

Synchronization

Essential to proper operation of our SIMD architecture is the synchronization of the master and slave processors. Each instruction cycle of the TMS32010 consists of four phases. We must assume that at least one slave processor's output clock will not be completely in phase with the master processor's output clock upon power-up. Since only Slave 1 actually interfaces with the global memory, the consequence of any timing discrepancy could be a slave processor that is unable to accept the program instructions from the global input bus when Slave 1 makes the instructions valid.

We avoid these problems by providing each slave processor with the synchronization circuit shown in Fig. 25. The synchronization circuit is enabled by the master
processor after power-up to force all slaves to match their output clock phases with the master’s output clock phase. Essentially, the circuit holds the slave’s input clock signal at a high state whenever it detects that the slave’s output clock signal is at a low state while the master’s output clock signal is at a high state. Fig. 26 shows examples of the synchronization circuit in operation for each possible out-of-phase power-up situation. Fig. 26(a) shows the master’s 20 MHz input clock and 5 MHz output clock. The slave in 26(b) requires the maximum time to synchronize its output clock to the master’s output clock, but this time represents only two instruction cycles. In 26(c) and 26(d), the slaves’ output clocks are synchronized after only one instruction cycle. The program used to synchronize the slaves is listed in Appendix B. Once the processors are synchronized, the circuit may be disabled. The slave processors should not require re-synchronizing unless the power is interrupted.

Fig. 25 -- Synchronization Circuit
Fig. 26 -- Synchronization Timing Diagrams
Status Port

To increase the versatility of our proposed development system, our design also includes a status port that the master uses to control BRACELET's operation. Status lines are connected to the slaves' reset pins, the data port multiplexers (Fig. 14), the global memory address multiplexer (Fig. 23), and the synchronization circuit (Fig. 25). The programming examples in Appendix B demonstrate the use of these ports to start the slaves, to stop the slaves, to designate the source of the global memory address, and to route data through the appropriate I/O ports. Instead of exactly synchronizing the master's program with the slaves' program to implement master/slave co-processing of scalar and vector calculations, handshaking signals could be used to force the processors to honor data dependencies. The master and the slaves could trigger each other to continue processing or to hold processing using these status bits in conjunction with an interrupt or polling protocol.

Implementation Details

Our hardware implementation of BRACELET mounts the four slave TMS32010s, the master TMS32010, the data latches, the external RAM, and the synchronization circuit on a double-high Multibus-compatible protoboard in a neat and organized layout. The integrated circuits were interconnected using 3M's Scotchflex wiring system with color-coded 30 gauge wire. The only power required for this board is 5 volts. A complete system that includes a 4K×16-bit private memory space for the single master processor, a 2K×16-bit global memory space, and a 2K×16-bit local memory space for each of the four slave processors draws approximately 4 amperes.

The master processor is actually implemented using TI's XDS Development
System and TMS32010 Emulator. The emulator provided us with a valuable tool early in the implementation stages to set breakpoints in programs and to allow us to examine the contents of the master processor's internal data registers. Data values local to the slave processors were passed through the interconnection network to the emulator for examination. We are currently in the process of replacing the emulator with a fifth TMS32010 by providing the Multibus circuitry necessary to download information from a host computer to BRACELET without the RS-232 serial port supplied by the emulator. This addition is a major step in transforming the operational multiprocessor into a convenient development system.

We designed our external data ports (global input, filter input, and filter output) to interface with A/D and D/A converters built for a previous project. These converters use a standard asynchronous protocol of ready and acknowledge signals derived from the input and output strobe signals for the appropriate ports. The quantizers have 14 bit precision with a binary offset representation (i.e. the most negative number is represented by $0000_{16}$ and the most positive number is represented by $3FFF_{16}$). Software must convert data between binary offset notation for I/O and two’s complement notation for arithmetic operations.

A Digital Equipment Corporation Professional 350 (Pro 350) business computer serves as our host computer. TMS32010 assembly language routines are developed on the Pro 350. We use DEC's Macro 11 assembly directives and Digital Signal Processing Software's TMS320 assembly routines [13] to produce our own BRACELET macro routines. A filter program then calls our macros whenever appropriate. Once a program is written, it may be compiled on the Pro 350 and downloaded to the TMS32010 emulator for execution on BRACELET. This procedure is detailed in the following section.
4.2. Software Design

By designing BRACELET’s hardware and software simultaneously, we were able to ensure efficient and versatile programming of adaptive filtering algorithms and applications. The software considerations that we describe in this section include executing vector operations with in-line code generated by macro routines, controlling the slaves with the master processor, and writing programs to implement adaptive filters.

In-Line Code

For our real-time applications, in-line code is used instead of loop structures to eliminate the instructions associated with loop control. In addition to the extra time required to initialize the loop control variable before the loop is entered, a significant time penalty is caused by the test for loop termination and the branch to the beginning of the loop during each pass through the loop. The TMS32010’s test and branch instruction BANZ executes in two instruction cycles. A filter that consists of 128 coefficients implemented on a four processor system would require 64 instruction cycles to simply control each loop structure. For a speech signal, only 625 instruction cycles can be executed during each sample period, and 128 of those cycles would be consumed by loop control for the LMS algorithm. More complex algorithms with an increased number of vector operations would require even more of these inefficient loop structures. Since BRACELET has sufficient program memory, we instead develop macro routines that expand into in-line code when assembled for optimal execution.
Macro Routines

We have developed an extensive file of macro routines that provide us with many of the master's basic control operations and the slaves' basic vector operations. With an INCLUDE or LIBRARY command, each filter program can access this separate macro file when assembling the filter program on the Pro 350. Since these macro routines are documented in Appendix B, we will only highlight some of the routines here.

INITSTATUS is used to place the entire BRACELET multiprocessor into a known state at the beginning of each program. STOPSLAVES and STARTSLAVES control the global program memory address multiplexer and the slaves' RESET lines to stop and start the slave multiprocessor.

DOWNLOAD transfers a block of code (program or data) from the master's program memory to the global program memory. UPLOAD transfers a block of code from the global program memory to the master's program memory. Since the slaves cannot write to global memory, UPLOAD is used mainly to verify the correctness of any downloaded code.

MAVECSHIFT performs a vector shift operation on an N element vector in the master by repeating the DMOV instruction. SLAVECSHIFT performs the shift operation on a vector stored among the slaves by repeating the DMOV operation and using the pipelined interconnection network to transfer values through the slaves.

PRODUCT multiplies corresponding values of two vectors and accumulates the inner products within each slave. TRANSVERSAL combines the operations of PRODUCT and SLAVECSHIFT to perform the basic transversal filter operations. ACCUMULATE adds the partial sums contained within each slave to produce a final sum in every slave.
Various ADAPT routines are employed by the LMSADAPT, FKADAPT, and FTFADAPT macros which perform the vector adaptation operations required by the LMS, Fast Kalman, and normalized Fast Transversal Filter algorithms. The NORM routines multiply a scalar by an entire vector and are used in normalized algorithms to scale vectors by an average signal power.

Values can be reported from the slave multiprocessor to the master using the pipelined interconnection network. To do this, the master executes MASHIN (master shift in) while the slaves execute SLASHOUT (slave shift out). The master can also transfer different values to each slave using the MASHOUT routine while the slaves execute the SLASHIN routine. These four routines are employed by BMASHIN, BMASHOUT, BSLSHIN, and BSLSHOUT to transfer entire blocks of data in the same manner. To execute different code in MIMD mode, each slave must first use these macros to shift in and then store each instruction from the master's memory to its own local program memory.

Controlling the Slave Multiprocessor

Few control operations are necessary once a filter program begins processing signals. The control operations mainly deal with synchronizing the slaves, downloading the global program into the global memory, defining whether the data ports are connected directly to the slave multiprocessor or through the master, and starting and stopping the slaves.

Software control of the synchronization circuit is very straightforward. As listed in Appendix B, the program SYNC.MAC enables the synchronization circuit by latching a logical 1 in bit six of the status port. A minimum delay of two instruction cycles allows
the output clock of the slave processors to synchronize with the output clock of the master processor. A second OUT instruction then disables the synchronization circuit by latching a logical 0 in bit six of the status port.

Once the hardware has been synchronized, several methods could be used to synchronize the master's program with the slaves' program. Our real-time operation of BRACELET verifies that two instruction cycles after the master latches a logical 1 on the slaves' RESET lines, the master can begin executing a program in concert with the slaves. With such precise control over the slaves, the master could therefore start and stop the slaves using the RESET lines whenever a data dependency must be satisfied. Another method would permit the master and the slaves to use interrupts or polling routines to test for these data dependencies. Still another method would require the master program and the slaves' program to be exactly synchronized at all times so the master would only need to start the slaves once.

In our filter programs, the master starts and stops the slaves only between major program segments. For instance, the master starts the slaves, vectors them to the program segment that initializes their data values, and then stops the slaves. Once the master accepts an input sample, it again starts the slaves, vectors them to the program segment that processes the input sample, and then stops the slaves after each sample. In this way, as long as the processing time for a single input sample does not exceed the sampling period, the sampling period can be changed without altering the filter program. Within each segment, NOPs are inserted to ensure exact synchronization between data dependencies. Careful division of the master's scalar stream and the slave multiprocessor's vector stream minimizes the delays associated with satisfying these dependencies.
Adaptive Filter Routines

The adaptive filter programs LMSMODEL.MAC, FKMODEL.MAC, and FTFMODEL.MAC listed in Appendix B demonstrate how a particular algorithm can be constructed from our macro routines. The equations that perform each algorithm are summarized in Appendix C. Since our main concern is to map these algorithms to the BRACELET architecture, we do not provide derivations or detailed explanations of these algorithms. The interested reader is encouraged to study more rigorous analyses [3,6,8,11]. In some cases, the order of the operations has been changed to increase the utility of the master and to minimize delays associated with data dependencies. Our documentation emphasizes these dependencies with comments denoting synchronization points so the reader can determine how the master and slave routines parallel each other.

The filter programs in Appendix B all perform system identification, but they can be easily modified to perform adaptive equalization, signal separation, and the other applications described in Section 2.2. The instructions that specifically perform the LMS algorithm would comprise the algorithm module, and the application shell would consist of the code unique to a particular application. For instance, regardless of the particular algorithm, the signal separation application would require a delay queue, which could be implemented by the slave multiprocessor's vector storage and pipelined interconnection network. To increase the amount of parallelism in the system, the master could implement the delay queue by performing a MAVECESHIFT on its own data memory while the slaves are performing other vector operations.

Just as different applications can be automatically generated by macro routines, variations to a particular algorithm can be included as well. The LMS algorithm modified for echo cancellation scales the error value by the average signal power of the delayed
input samples. One method used to implement this normalization process has the slaves accumulate the product $X^T(T)X(T)$ during each sample period $T$ and then pass the sum to the master for the division. Another method allows the master to square each sample as it is input and shift the entire vector of squared samples through the slaves' pipelined interconnection network. The master only has to add the current squared sample $x^2(T)$ to a running sum of squares and then subtract $x^2(T-N)$ after the slaves pass it to the master. Finally, the master could implement the entire delay queue within its own data memory and sum the squares while the slave are performing other vector calculations.

To generalize these filter programs, the number of coefficients in the filter is defined as a macro and can be easily changed before assembling. The program inserts sufficient NOPs to synchronize the master and slave routines for the desired filter size. The number of processors in the system could also be defined in this manner to program the proper number of interprocessor communication stages and to automatically adjust the number of coefficients stored within each slave processor.
Chapter 5 — Real-Time Operation of BRACELET

To verify proper operation of our hardware, we used software test routines to assert known signals on the 16-bit data busses, the 12-bit address busses, and the various control lines while viewing these signals on a Tektronix 7403N Oscilloscope. We also used the development facilities of the XDS emulator to examine data and program memory within the master processor and the slave multiprocessor. Satisfied that these tests proved proper operation of BRACELET in tightly coupled SIMD and MIMD modes, we then demonstrated the use of our multiprocessor architecture in several adaptive filtering applications of the LMS algorithm. We briefly describe this real-time operation and include spectral plots of various I/O signals that illustrate these applications. We then discuss BRACELET's real-time performance and provide plots and tabulated data that graphically depict our analysis. Based on our analysis of the LMS algorithm applications, we are able to project the system's performance of our untested implementations of the Fast Kalman algorithm and the normalized Fast Transversal Filter algorithm.

5.1. Adaptive Filtering Demonstrations

While the filter programs were operating on our sampled signals, we were able to view the frequency components of the signals on a Hewlett Packard 3538A Spectrum Analyzer. In this way, we were able to compare the input signal's spectrum, the prediction error signal's spectrum, and output signal's spectrum for a qualitative verification of proper filter operation. The Pro 350, the spectrum analyzer, and a Hewlett Packard 7470A Plotter are interconnected through an HP-IB interface bus (IEEE standard 488-1978) to obtain plots of the spectrum analyzer's display. The instrument settings of the
spectrum analyzer -- e.g. Resolution Band Width (RBW), Video Band Width (VBW), Sweep Time (ST) -- greatly affect the resolution of the displayed spectrum, so the absolute signal characteristics in these examples are not as important as the characteristics of one signal relative to the characteristics of another signal. Since the master processor is actually implemented by the XDS emulator, we could stop the adaptive filter program and easily change the feedback gain constant $\alpha$ to adjust the rate of convergence. We were therefore able to alter the cut-off frequency of the low pass filter and the frequencies of the sine waves to watch BRACELET adapt to the new system characteristics. A more rigorous analysis of the performance of a particular application or algorithm would examine the sampled input, output, and error signals in more detail; but for our demonstrations of BRACELET, the following examples are sufficient.

System Identification

System identification (Fig. 3) provides an excellent demonstration of adaptive filtering. The system that BRACELET identified was a Wavetek/Rockland 852 Dual Hi/Lo Filter set for a low pass filter with a cut-off frequency at 1300 Hz. A General Radio 1383 20Hz-20MHz Random Noise Generator produced a white noise input signal which was band-limited to 4 kHz by the Rockland filter. Since the input to the filter was white noise, the spectrum of the output signal represented the frequency response of the filter, as shown in Fig. 27(a). BRACELET input the white noise through the filter input port and the desired response signal through the global input port. The program executed by BRACELET was a standard LMS algorithm with 128 filter coefficients and is listed in Appendix B. Fig. 27(b) shows the output of the adaptive filter, which does indeed model the output of the low pass filter.
Fig. 27 - System Identification Demonstration

(a) Spectrum of Analog Filter Output

(b) Spectrum of Adaptive Filter Output
Echo Cancellation

In echo cancellation, an important application of adaptive system identification, the adaptive filter models the leakage path that exists through a telephone hybrid (Fig. 28). Rather than returning this leakage signal to the original talker as an echo, the adaptive filter subtracts an estimate of the leakage from the actual leakage to produce an echo residual. For our demonstration, the filter input and therefore the hybrid leakage was a frequency-swept sine wave produced by a Tektronix FG504 Function Generator (Fig. 29(a)). A normalized LMS algorithm with 128 filter coefficients was implemented by BRACELET. Fig. 29(b) shows that the spectrum of the echo residual was greatly reduced and de-correlated from the actual leakage. In a real application, a double talker detector would sense the near end speech and freeze the adaptation to prevent the error from suddenly becoming large.

Fig. 28 - Adaptive Filter Configuration for Echo Cancellation
(a) Spectrum of Telephone Hybrid Leakage

(b) Spectrum of Echo Residual

Fig. 29 - Echo Cancellation Demonstration
Adaptive Equalization

In the adaptive equalization demonstration (Fig. 4(a)), a broadband input signal was shaped by the same 1300 Hz low pass filter used in the system identification example. Fig. 30(a) shows the spectrum of the original broadband signal (band-limited to 4 kHz) and Fig. 30(b) shows the spectrum of the signal after it passed through the analog filter. The program executed by BRACELET was a standard LMS algorithm with 128 filter coefficients. Fig. 30(c) shows that the adaptive filter did equalize the effect of the 1300 Hz low pass filter. With the filter coefficients frozen, a 4 kHz information signal (such as speech) could be transmitted through the low pass filter and the adaptive filter would equalize its effect.

(a) Spectrum of Original Broadband Signal

Fig. 30 - Adaptive Equalization Demonstration
Fig. 30 - (continued) Adaptive Equalization Demonstration
Signal Separation

In the signal separation demonstration (Fig. 7), a broadband signal was adaptively separated from narrowband signals. The delay queue de-correlated the broadband input from the reference input allowing the adaptive filter to predict only the highly correlated narrowband signals. The adaptive filter input (Fig. 31(a)) consisted of four sine waves with decreasing amplitudes at 800, 1600, 2400, and 3200 Hz embedded in Gaussian noise band-limited to 4 kHz. A standard LMS algorithm with 64 filter coefficients and a delay queue of 8 samples were implemented by BRACELET. The filter output (Fig. 31(b)) shows the broadband signal was reduced to reveal all four sine waves. The prediction error (Fig. 31(c)) shows the narrowband sine waves were successfully eliminated.

Fig. 31 - Signal Separation Demonstration
Fig. 31 - (continued) Signal Separation Demonstration
5.2. Performance Analysis of the BRACELET Architecture

When expressing the real-time performance capabilities of BRACELET, it is important to avoid vague statistics. For example, each TMS32010 is capable of executing 5 MOPS, so the BRACELET architecture configured with a master processor and four slave processors executing simultaneously performs 25 MOPS. A data transfer through the interconnection network requires an output instruction followed by an input instruction (each requiring two 200 ns instruction cycles). Since the master and four slaves can execute these instructions simultaneously, five 16-bit words are transferred in 800 ns. Expressed another way, BRACELET's interconnection network is capable of transferring information at a rate of 100 Mbps. While both of these representations are impressive, they are vague and misleading because they do not reflect the system's performance under normal operating conditions.

A more meaningful measure of system performance states the maximum possible sampling frequency for a given algorithm, application, and filter size. We can compare our multiprocessor architecture with uniprocessors or other multiprocessors by considering how the master processor and the number of slave processors affects system performance. We can also compare BRACELET's interconnection network with other networks by examining the number of instruction cycles required for interprocessor communication. Our analysis emphasizes BRACELET's performance of system identification and compares our LMS algorithm implementation with our untested implementations of the Fast Kalman algorithm and the normalized Fast Transversal Filter algorithm.
LMS Algorithm

Using the LMSMODEL.MAC program listed in Appendix B, we can express the number of instruction cycles executed during each sample period as a function of the number of filter taps \( N \) and the number of slave processors \( P \) in our system,

\[
\text{cycles} = 6 \frac{N}{P} + 7 \log_2 P + 26 \quad \text{(5.1)}
\]

The ratio \( \frac{N}{P} \) defines the number of filter taps stored in each of the slave processors, and \( \log_2 P \) defines the number of interprocessor communication stages required to combine the partial sums of the \( P \) slave processors in the BRACELET network. We do not include the time required to accept and scale the input sample, to scale and latch the filter output, or to start and stop the slave processors in this equation. Since we are calculating maximum sampling frequency, we assume the slaves are never stopped -- as soon as they calculate one sample, the next sample is ready to be processed. We can also stagger the master/slave programs so the master is doing I/O and scaling while the slaves are doing vector calculations.

If we wish to find the maximum sample frequency \( f_{\text{max}} \) as a function of \( N \) and \( P \), we include the instruction cycle time \( T_c \) in the following equation

\[
f_{\text{max}} = \frac{1}{T_c \left(6 \frac{N}{P} + 7 \log_2 P + 26 \right)} \quad \text{(5.2)}
\]

Table 1 and the corresponding graph in Fig. 32 show the results of these calculations for \( T_c = 200 \text{ ns} \), \( N \) ranging from 16 to 512, and \( P \) ranging from 1 to 64. The table entry \( P > N \) reflects the fact that the number of slave processors cannot exceed the number of filter coefficients. We ignore for now the TMS32010's restraint of having only 128 internal data registers in its first page of memory so that we may easily extrapolate our results.
### Table 1 - Maximum Sampling Frequency (Hz)
LMS Algorithm - BRACELET Network

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>40984</td>
<td>61728</td>
<td>78125</td>
<td>84746</td>
<td>83333</td>
<td>P &gt; N</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>32</td>
<td>22936</td>
<td>38760</td>
<td>56818</td>
<td>70423</td>
<td>75758</td>
<td>74627</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>64</td>
<td>12195</td>
<td>22222</td>
<td>36765</td>
<td>52632</td>
<td>64103</td>
<td>68493</td>
<td>67568</td>
</tr>
<tr>
<td>128</td>
<td>6297</td>
<td>11990</td>
<td>21552</td>
<td>34965</td>
<td>49020</td>
<td>58824</td>
<td>62500</td>
</tr>
<tr>
<td>256</td>
<td>3201</td>
<td>6242</td>
<td>11792</td>
<td>20921</td>
<td>33333</td>
<td>45872</td>
<td>54348</td>
</tr>
<tr>
<td>512</td>
<td>1614</td>
<td>3187</td>
<td>6188</td>
<td>11601</td>
<td>20325</td>
<td>31847</td>
<td>43103</td>
</tr>
</tbody>
</table>

Fig. 32 - Performance Analysis of LMSMODEL.MAC on BRACELET Network
to a TMS32010 with more internal data storage. Actually, we could use the second page of 16 internal registers or the slaves' external memory to provide more storage, but the program would be less efficient. The master processor could also store part of the filter taps and delayed input samples for more parallel storage and processing, but our analysis is based on the assumption that the vectors are stored in the slave processors and the scalars in the master processor. Furthermore, since our interconnection network is optimized for the binary tree structure of the parallel addition, we only consider the cases in which the number of processors is a power of two.

As we have already shown, for speech signals with a sampling frequency of 8 kHz, the LMS algorithm causes the storage capacity of the TMS32010 to be exceeded before its processing ability. For \( \frac{N}{P} = 64 \) coefficients per processor over the tabulated range of \( N \) and \( P \), the minimum \( f_{\text{max}} \) is 11601 Hz. Since BRACELET was designed to perform more complex algorithms, this overdesign for the LMS algorithm is welcome.

Fig. 32 and Table 1 can help us design specific multiprocessor configurations and applications. Given the filter size and the number of slave processors, we can determine the maximum sampling frequency of our adaptive filter. For instance, an LMS algorithm implementing system identification with \( P=4 \) and \( N=128 \) can model a system with a bandwidth of 10776 Hz. Given the number of slave processors and a specified sampling frequency, we can determine the maximum filter size. For \( f_{\text{max}}=40 \text{ kHz} \) and \( P=4 \), BRACELET can implement approximately 58 taps. Solving Eq. 5.2 for \( M=\frac{N}{P} \) to determine this value exactly produces

\[
M = \frac{N}{P} = \frac{1}{6} \left( \frac{1}{T_{\text{c}} f_{\text{max}}} - 7 \log_2 P - 26 \right)
\]

(5.3)

Since the slave processors execute in SIMD mode, the maximum number of filter taps is
proportional to the largest integer less than \( M \), i.e.

\[
N_{\text{max}} = P \left\lfloor M \right\rfloor
\]  

For our example, \( N_{\text{max}} = 60 \). Finally, we can determine the number of processors required to implement a particular application. For \( f_{\text{max}} = 20 \text{ kHz} \) and \( N = 256 \), we determine that \( \text{BRACELET} \) must consist of eight slave processors to execute the \( \text{LMSMODEL.M.A..C} \) program within the given sample period.

The master processor has little effect on \( \text{BRACELET}'s \) performance of the LMS algorithm due to the inherent simplicity of LMS. The weight vector adaptation is dependent on the scalar prediction error, which is in turn dependent on the transversal filter output. The result is very little master/slave co-processing. The master could, however, perform its I/O and conversion operations while the slaves perform vector operations. Furthermore, the normalized LMS algorithm could be implemented with no degradation in performance if the normalization process (including the time-consuming division) occurs in the master.

General trends in Table 1 worth noting are most easily observed at the extreme ranges of the tabulated values. When the number of processors in the system is relatively small (\( P \leq 8 \)) and the number of filter taps is large (\( N \geq 128 \)), the instruction cycles devoted to interprocessor communication represents a small percentage of the total processing time. The resulting effects is that \( f_{\text{max}} \) is proportional to \( \frac{P}{N} \), which gives the filter designer the ability to double \( f_{\text{max}} \) by doubling \( P \) and keeping \( N \) constant and to keep \( f_{\text{max}} \) constant by doubling both \( P \) and \( N \).

At the opposite extreme, when \( N < 128 \) and \( P > 8 \), the interprocessor communication represents a large percentage of the total processing time. While keeping \( N \) con-
stant and doubling $P$, $f_{\text{max}}$ reaches a maximum and then begins to decline. Very little degradation to performance occurs, however, when $N$ is doubled while $P$ remains constant. Finally, when $N$ is doubled, doubling $P$ adds another communication stage with the net effect of decreasing $f_{\text{max}}$.

Since BRACELET’s interconnection network is optimized for adaptive filter operations, any other network could only equal BRACELET at best and could instead seriously degrade performance. Fig. 33 and Table 2 demonstrate this effect for the simple ring network. Modifying Eq. 5.1 to reflect the ring’s performance produces a term that is proportional to the number of processors $P$,

$$\text{cycles}_{\text{LMS,ring}} = 6 \frac{N}{P} + 4P + 3 \log_2 P + 34 \quad (5.5)$$

The maximum sampling frequencies reach a maximum at a smaller value of $P$ than for BRACELET, and these maximum frequencies are themselves smaller values. Fig. 33 graphically depicts the degradation in performance due to the inefficiencies of shifting values through a long ring. Table 1 with Table 2 do reveal, however, that the hardware cost of a small system, such as our four processor system or even an eight processor system, could be greatly reduced with very little performance degradation by implementing a simple ring network rather than the entire BRACELET structure.
Table 2 -- Maximum Sampling Frequency (Hz)
LMS Algorithm - Ring Network

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>40984</td>
<td>61728</td>
<td>73529</td>
<td>66667</td>
<td>48077</td>
<td>P&gt;N</td>
<td>P&gt;N</td>
</tr>
<tr>
<td>32</td>
<td>22936</td>
<td>38760</td>
<td>54348</td>
<td>57471</td>
<td>45455</td>
<td>29240</td>
<td>P&gt;N</td>
</tr>
<tr>
<td>64</td>
<td>12195</td>
<td>22222</td>
<td>35714</td>
<td>45045</td>
<td>40984</td>
<td>28219</td>
<td>16556</td>
</tr>
<tr>
<td>128</td>
<td>6297</td>
<td>11990</td>
<td>21186</td>
<td>31447</td>
<td>34247</td>
<td>26455</td>
<td>16234</td>
</tr>
<tr>
<td>256</td>
<td>3201</td>
<td>6242</td>
<td>11682</td>
<td>19608</td>
<td>25773</td>
<td>23474</td>
<td>15625</td>
</tr>
<tr>
<td>512</td>
<td>1614</td>
<td>3187</td>
<td>6158</td>
<td>11186</td>
<td>17241</td>
<td>19157</td>
<td>14535</td>
</tr>
</tbody>
</table>

Fig. 33 - Performance Analysis of LMSMODEL.MAC on RING Network
Fast Kalman Algorithm

The evaluations made for the LMS algorithm can also be applied to our untested FKMODEL.MAC coding of the Fast Kalman algorithm [6,11]. The number of instruction cycles executed per sample is given by

$$\text{cycles}_{FK} = 32 \frac{N}{P} + 28 \log_2 P + 46$$ (5.6)

The tabulated values of $f_{\text{max}}$ for the previous range of $N$ and $P$ are given in Table 3 and depicted in Fig. 34. Again we assume that the TMS32010 has sufficient internal data memory for this study.

The same general trends observed with BRACELET's implementation of the LMS algorithm apply to the Fast Kalman implementation, although the greater complexity of the Fast Kalman algorithm causes the values of $f_{\text{max}}$ to be proportionately lower than the values for LMS. For example, our four processor system executing the Fast Kalman algorithm can only operate on a speech signal with 64 filter coefficients, and an eight processor system is unable to execute a 128-tap filter at 8 kHz. As opposed to the relatively simple LMS algorithm, the Fast Kalman implementation validates our use of a multiprocessor architecture to achieve high bandwidth processing of large, complex filters in real-time.

The Fast Kalman also affords the BRACELET architecture the opportunity for more master/slave co-processing of scalar/vector operations. Unfortunately, even for this more complex algorithm, the master only co-processes approximately 70 instruction cycles of scalar and control operations, which corresponds to an increased filter size of two coefficients per processor. To maximize the utility of the master, either the ratio $\frac{N}{P}$ should be kept very small or the master should be used as a fifth vector processor.
Table 3 -- Maximum Sampling Frequency (Hz)
Fast Kalman Algorithm - BRACELET Network

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>8961</td>
<td>15152</td>
<td>21739</td>
<td>25773</td>
<td>26316</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>4673</td>
<td>8532</td>
<td>13966</td>
<td>19380</td>
<td>22523</td>
<td>22936</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>2388</td>
<td>4554</td>
<td>8143</td>
<td>12953</td>
<td>17483</td>
<td>20000</td>
<td>20325</td>
</tr>
<tr>
<td>128</td>
<td>1207</td>
<td>2356</td>
<td>4440</td>
<td>7788</td>
<td>12077</td>
<td>15924</td>
<td>17986</td>
</tr>
<tr>
<td>256</td>
<td>607</td>
<td>1199</td>
<td>2326</td>
<td>4333</td>
<td>7463</td>
<td>11312</td>
<td>14620</td>
</tr>
<tr>
<td>512</td>
<td>304</td>
<td>605</td>
<td>1191</td>
<td>2296</td>
<td>4230</td>
<td>7163</td>
<td>10638</td>
</tr>
</tbody>
</table>

Fig. 34 -- Performance Analysis of FKMODEL.MAC on BRACELET Network
Finally, to emphasize the need for an efficient interconnection network for BRACELET's implementation of the Fast Kalman algorithm, we modify Eq. 5.6. to reflect the performance of the simple ring network

\[
\text{cycles}_{FK,\text{ring}} = 32 \frac{N}{P} + 16P + 12 \log_2 P + 30
\]

(5.7)

Table 4 and Fig. 35 depict the tabulated values of the Fast Kalman algorithm performed on the ring network, which is seriously degraded except on a very small system such as our four processor architecture. The increased number of inner product accumulations amplifies the inefficiency of the ring network for large systems.

**Fast Transversal Filter Algorithm**

The last adaptive filtering program we evaluate is our untested implementation of the normalized Fast Transversal Filter [3], listed as FTFMODEL.MAC in Appendix B. The number of instruction cycles executed per sample is

\[
\text{cycles}_{FTF} = 42 \frac{N}{P} + 14 \log_2 P + 43
\]

(5.8)

The tabulated maximum sampling frequencies for FTF is given in Table 5 and graphed in Fig. 36.

The Fast Transversal Filter program provides an interesting comparison with the Fast Kalman program because the FTF is more heavily dependent on the number of filter taps per processor \( \left( 42 \frac{N}{P} \right. \) vs. \( \left. 32 \frac{N}{P} \right) \) but less dependent on the number of communication stages \( (14 \log_2 P \) vs. \( 28 \log_2 P) \). The resulting effect is that for \( N < \frac{P}{10} (14 \log_2 P + 3) \), the Fast Kalman algorithm achieves a higher maximum sampling
Table 4 -- Maximum Sampling Frequency (Hz)
Fast Kalman Algorithm - Ring Network

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>8961</td>
<td>15152</td>
<td>20325</td>
<td>19330</td>
<td>13661</td>
<td>P &gt; N</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>32</td>
<td>4673</td>
<td>8532</td>
<td>13369</td>
<td>15528</td>
<td>12563</td>
<td>7886</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>64</td>
<td>2388</td>
<td>4554</td>
<td>7937</td>
<td>11111</td>
<td>10823</td>
<td>7508</td>
<td>4318</td>
</tr>
<tr>
<td>128</td>
<td>1207</td>
<td>2356</td>
<td>4378</td>
<td>7032</td>
<td>8475</td>
<td>6849</td>
<td>4202</td>
</tr>
<tr>
<td>256</td>
<td>607</td>
<td>1199</td>
<td>2308</td>
<td>4105</td>
<td>5910</td>
<td>5828</td>
<td>3987</td>
</tr>
<tr>
<td>512</td>
<td>304</td>
<td>605</td>
<td>1187</td>
<td>2230</td>
<td>3692</td>
<td>4498</td>
<td>3618</td>
</tr>
</tbody>
</table>

Fig. 35 – Performance Analysis of FKMODEL.MAC on RING Network
Table 5 -- Maximum Sampling Frequency (Hz)
Normalized FTF Algorithm - BRACELET Network

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>16</td>
<td>6993</td>
<td>12723</td>
<td>20921</td>
<td>29586</td>
<td>35461</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>32</td>
<td>32</td>
<td>3605</td>
<td>6859</td>
<td>12285</td>
<td>19763</td>
<td>27322</td>
<td>32258</td>
</tr>
<tr>
<td>64</td>
<td>64</td>
<td>1831</td>
<td>3569</td>
<td>6729</td>
<td>11876</td>
<td>18727</td>
<td>25381</td>
</tr>
<tr>
<td>128</td>
<td>128</td>
<td>923</td>
<td>1821</td>
<td>3534</td>
<td>6605</td>
<td>11494</td>
<td>17794</td>
</tr>
<tr>
<td>256</td>
<td>256</td>
<td>463</td>
<td>920</td>
<td>1812</td>
<td>3499</td>
<td>6485</td>
<td>11136</td>
</tr>
<tr>
<td>512</td>
<td>512</td>
<td>232</td>
<td>463</td>
<td>918</td>
<td>1803</td>
<td>3465</td>
<td>6369</td>
</tr>
</tbody>
</table>

Fig. 36 -- Performance Analysis of FTFMODEL.MAC on BRACELET Network
frequency but greater values of $N$ cause the Fast Transversal Filter to achieve a higher $f_{\text{max}}$. The size of the filter and the number of processors in the system therefore join the inherent adaptive properties of each algorithm to complicate the filter designer's use of a particular algorithm.

Eq. 5.8. is modified in Eq. 5.9. to show that the Fast Transversal Filter is also burdened by inner product accumulations that cause a ring network to be inefficient for large values of $P$

$$\text{cycles}_{\text{FTF,ring}} = 42\frac{N}{P} + 8P + 6\log_2P + 35 \quad (5.9)$$

Table 8 and the corresponding plot in Fig. 37 illustrate Eq. 5.9. for various values of $N$ and $P$.

Finally, we examine the utility of the master as a scalar co-processor. In our FTFMODEL.MAC routine, the master only co-processes about 80 instruction cycles of scalar and control operations, again corresponding to about two taps per processor. The utility of the master would be significantly higher if FTFMODEL.MAC were to perform the two square roots and two divisions summarized in Appendix C and would, in fact, give the master almost 100\% utility without processing any filter taps. Instead, our programs use look-up tables to eliminate these time-consuming calculations.
Table 6 -- Maximum Sampling Frequency (Hz)
Normalized FTF Algorithm - Ring Network

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>6993</td>
<td>12723</td>
<td>20243</td>
<td>24876</td>
<td>21834</td>
<td>P &gt; N</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>32</td>
<td>3605</td>
<td>6859</td>
<td>12048</td>
<td>17544</td>
<td>18450</td>
<td>13774</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>64</td>
<td>1831</td>
<td>3569</td>
<td>6658</td>
<td>11038</td>
<td>14085</td>
<td>12346</td>
<td>8000</td>
</tr>
<tr>
<td>128</td>
<td>923</td>
<td>1821</td>
<td>3514</td>
<td>6337</td>
<td>9560</td>
<td>10225</td>
<td>7496</td>
</tr>
<tr>
<td>256</td>
<td>463</td>
<td>920</td>
<td>1807</td>
<td>3422</td>
<td>5821</td>
<td>7610</td>
<td>6658</td>
</tr>
<tr>
<td>512</td>
<td>232</td>
<td>463</td>
<td>917</td>
<td>1783</td>
<td>3266</td>
<td>5035</td>
<td>5441</td>
</tr>
</tbody>
</table>

Fig. 37 -- Performance Analysis of FTFMODEL.MAC on RING Network
Chapter 6 – Improvements to the BRACELET Architecture

Throughout the entire process of designing, implementing, and testing BRACELET's hardware and adaptive filter routines, we have been able to analyze the TMS32010 and the BRACELET multiprocessor to identify the advantages as well as possible improvements to the system. Since the basic architecture is independent of the particular processor used, DSP chips that have recently appeared on the market (e.g. TI's TMS32020 [20], NEC's μPD77230 [14], INMOS's IMS T424 Transputer [7]) could replace the TMS32010 used in our design to improve BRACELET's overall performance. Rather than discussing these newer processors in detail, we continue to concentrate on the TMS32010 by describing relatively simple modifications to the TMS32010 that would significantly improve BRACELET's performance. We then address the issue of expandability and propose a modular interconnection scheme that can take advantage of large scale integration techniques to provide a compact network for interprocessor communications.

6.1. Improvements to the TMS32010 DSP Chip

One method of improving BRACELET's performance is to improve the performance of the slaves used in the multiprocessor. This section explores four general areas of improvements that we have discovered through our experience with the TMS32010. First we examine the effect of increasing the clock rate. Then we describe several internal data path modifications that would speed the processor's execution of adaptive filtering algorithms. The improved performance of our modified TMS32010s leads us next to consider greater internal memory capacity. Finally we briefly examine the advantages of floating
point arithmetic as opposed to the integer arithmetic of the TMS32010.

Clock Rate

Perhaps the most obvious modification that would improve the performance of the TMS32010 and consequently the BRACELET architecture is an increase in the processor's basic clock rate. This step has, in fact, already been taken by TI. The instruction cycle of a newer version of the TMS32010 has been decreased from 200 ns to 150 ns. For BRACELET, this improvement corresponds to a direct 33% increase in the maximum sampling frequency possible for a given filter size, regardless of any particular algorithm or application. Table 5 is updated in Table 7 to show the maximum sampling frequencies for the normalized FTF algorithm performing system identification with the BRACELET architecture consisting of four of these faster TMS32010s. One particular case worth emphasizing is the 128 tap filter on an eight processor system. The 200 ns TMS32010s allow a maximum frequency of 6605 Hz while the 150 ns TMS32010s allow 8807 Hz. In this case, the 33% increase in $f_{\text{max}}$ spans the 8 kHz sampling frequency, which would permit the faster BRACELET configuration to process speech signals.

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>9324</td>
<td>16964</td>
<td>27894</td>
<td>39418</td>
<td>47281</td>
<td>P&gt;N</td>
<td>P&gt;N</td>
</tr>
<tr>
<td>32</td>
<td>4807</td>
<td>9145</td>
<td>16380</td>
<td>26350</td>
<td>36430</td>
<td>43011</td>
<td>P&gt;N</td>
</tr>
<tr>
<td>64</td>
<td>2441</td>
<td>4759</td>
<td>8973</td>
<td>15835</td>
<td>24969</td>
<td>33841</td>
<td>39448</td>
</tr>
<tr>
<td>128</td>
<td>1230</td>
<td>2429</td>
<td>4711</td>
<td>8807</td>
<td>15326</td>
<td>23725</td>
<td>31596</td>
</tr>
<tr>
<td>256</td>
<td>618</td>
<td>1227</td>
<td>2416</td>
<td>4665</td>
<td>8647</td>
<td>14848</td>
<td>22599</td>
</tr>
<tr>
<td>512</td>
<td>309</td>
<td>617</td>
<td>1224</td>
<td>2404</td>
<td>4620</td>
<td>8493</td>
<td>14399</td>
</tr>
</tbody>
</table>
Internal Data Paths

The TMS32010's pipelined multiply, accumulate, and register shift operations are already very efficient for performing transversal filters. However, our work has revealed several relatively simple changes to this internal pipeline, redrawn in Fig. 38, that would significantly improve the execution time of the array operations found in our filter programs. A necessary result of the TMS32010's fast hardware multiplier, these modifications allow the TMS32010 to load and store data in a more transparent fashion.

The first modification is used when the TMS32010 performs a multiply operation without an accumulate (e.g. when the error value \( e(T) \) is multiplied by the feedback gain constant \( \alpha \) in the LMS algorithm). The product is temporarily stored in the P register (Fig. 38) and must be loaded into the accumulator before it can be stored in a data register. If a path existed directly from the P register to the data registers, as in Fig. 39, this extra instruction cycle could be eliminated. While saving one cycle appears trivial, the normalized FTF algorithm performs this operation six times on vectors consisting of \( N \) elements, which would allow \( 6 \cdot \frac{N}{P} \) instruction cycles to be eliminated. The execution time of the Fast Kalman algorithm would be decreased by \( \frac{N}{P} \) instruction cycles and the LMS algorithm by only 1 instruction cycle.

Another data path modification would find use whenever filter coefficients are updated. Referring to the LMS algorithm, the old value of a filter coefficient \( w_j(T-1) \) is first loaded into the accumulator. The error factor \( \alpha e(T) \) in the T register is multiplied by the delayed input sample \( x_j(T) \) and is then either added to or subtracted from the value in the accumulator. By allowing the P register and any one of the 128 data registers to be ALU operands simultaneously, as shown in Fig. 40, the instruction cycle
Fig. 38 – Pipelined Data Path of the TMS32010

Fig. 39 – Direct Path from P-register to Data Registers
required to load the accumulator initially with $w_i(T-1)$ could be eliminated. Since the normalized FTF algorithm performs five vector update operations, the resulting improvement would be approximately $5 \frac{N}{P}$ instruction cycles. This modification would improve any algorithm that performs the filter update operation, including $5 \frac{N}{P}$ instruction cycles for the Fast Kalman algorithm and $\frac{N}{P}$ instruction cycles for the LMS algorithm.

The last data path modification that we note here would further improve the two previous modifications. By allowing the 128 data registers to be accessed simultaneously as the source operand and as the destination operand of an ALU operation, a filter

---

Fig. 40 -- ALU Operations Between P-register and Data Registers
coefficient may be scaled (Fig. 41(a)) or adjusted (Fig. 41(b)) in one instruction cycle. As in the previous modifications, the purpose is to eliminate instruction cycles used only for data transfers. If this modification would significantly increase the instruction cycle time, the store operation could be pipelined with the multiplication or ALU operation, allowing the result of the previous calculation to be stored while the current calculation is taking place. The normalized FTF algorithm would benefit by another $11 \frac{N}{P}$ reduction in instruction cycles with this modification; the Fast Kalman algorithm would benefit by another $6 \frac{N}{P}$ reduction; and the LMS algorithm would benefit by another $\frac{N}{P}$ reduction.

Tables 1, 3, and 5 have been updated in Tables 8, 9, and 10 to reflect the combined effects of these internal structural modifications and the 150 ns instruction cycle on BRACELET's performance. The number of instruction cycles processed during each sample period is expressed for each algorithm by the following equations,

$$\text{cycles}_{\text{LMS}} = 4 \frac{N}{P} + 7\log_2 P + 25 \quad (6.1)$$

$$\text{cycles}_{\text{FK}} = 20 \frac{N}{P} + 28\log_2 P + 46 \quad (6.2)$$

$$\text{cycles}_{\text{FTF}} = 20 \frac{N}{P} + 14\log_2 P + 43 \quad (6.3)$$

To perform the normalized FTF algorithm on a filter with 128 coefficients and a sampling frequency of 8 kHz, the BRACELET architecture with TMS32010s modified as we have described would require only four slave processors (Table 10). As a result of significantly improving the performance of the processor used in our design, we have thereby lessened the need for a large multiprocessor system and in turn lessened the need for an elaborate interconnection network.
(a) Vector Scaling Operation

(b) Vector Adjustment Operation

Fig. 41 – Dual Port Registers for Transparent Data Access
For this reason, we have devised a variation of BRACELET's interconnection network based on the inherent binary tree structure of the parallel addition operation. Just as in the present network, the processors are still linked by 16-bit parallel lines to form the global input and the pipe/ring structure required by adaptive filters. Unlike the present network, this new interconnection scheme is ideally suited for LSI implementation for greatly reducing the hardware and increasing its expandability.

Overview of Improved Network

Our improved network is shown in Fig. 42 interconnecting eight slave processors. Each node has a bidirectional link to each of its three neighbors, one higher level node and two lower level nodes. The lowest level connects to the slave processors, which are situated at the leaves of the tree structure. Each slave requires only one connection to the

Fig. 42 -- Improved Interconnection Network
entire network. The highest level node connects to the global memory, the global input port, the filter input port, and the filter output port. A system with $P$ processors requires $P-1$ nodes distributed among $\log_2 P$ levels. Except for the data port connections at the highest level node, the system may be easily expanded without altering already existing links.

Fig. 43 illustrates the circuitry that constitutes each node. For simplicity, the components are represented only by unidirectional buffers and latches with high impedance outputs. The three 16-bit buffers and the two 16-bit latches are capable of performing the pipe, ring, and global connections for a multiprocessor consisting of two slaves.

Fig. 43 -- Detail of a Single Interconnection Node
Theory of Operation

For the multiprocessor to execute code from a single global memory in SIMD mode, a global instruction must be broadcast to all slaves. Fig. 44 shows the paths activated to perform a global input (or broadcast) operation. The information could be an instruction from the global program memory or a data sample from the master processor. Identical paths are enabled through each node which simplifies the necessary control signals.

The interconnection network must also provide the pipelined data path to shift the delayed input sample vector. For a tree structure, this path is not at all obvious. As in the present network, the pipelined communication consists of an output followed by an

Fig. 44 -- Active Paths for Broadcast
input. Fig. 45 shows the active paths for the output stage. Each node stores the value from its left child in one latch and passes the value from its right child to the next higher level. This value does not need to be latched, but we will see later that latching this value allows this same output stage to be used for the output stage of the ring transfer.

Fig. 46 shows the connections that complete input stage of the pipeline. Before or during the output operation of Fig. 45, the master should output a new value to its own latch that connects to the root. As shown in Fig. 46, this value cascades down the leftmost child of each level to the leftmost processor of the pipe. Each node transfers the value latched from its left child down the right child path, where it then cascades down the successive left nodes until it reaches a processor at the lowest level.

As stated above, the active paths of Fig. 45 can double as the output stage for the ring transfer. Fig. 47 and Fig. 48 show the connections that complete the input stage of the ring transfers. Partial sums contained within each processor are shared in this manner to accumulate the final sum in log₂P input/output stages. In Fig. 47, the nodes at level 1 pass the value accepted from its left child to its right child, and the value accepted from its right child is passed to its left child. All nodes above level 1 have paths that have been placed in a high impedance state. In Fig. 48, the node at level 2 performs the exchange operation while all nodes at lower levels perform the broadcast operation. Unlike the broadcast and the pipelined communication paths, the specific operation performed by a node of a given level is dependent on that node's level and the level at which the ring transfer is taking place.

Disadvantage of Improved Network

The major drawback of this interconnection scheme is the accumulated delay as
Fig. 45 – Active Paths for Pipelined and Ring Output

Fig. 46 – Active Paths for Pipelined Input
Fig. 47 -- Active Paths for Level 1 Ring Input

Fig. 48 -- Active Paths for Level 2 Ring Input
data is transferred through the buffers of the tree network. One way to reduce this delay is to build the tree structure with four children rather than two, making the tree wider but with fewer levels. A corresponding reduction in performance would accompany this modification since the complete binary tree structure is not available to optimize the parallel addition. Further research is required in this area to determine the maximum number of levels attainable with today’s DSP chips.

Large Scale Integration of Nodes

A valid claim of reducing hardware by implementing this interconnection scheme assumes the buffers and latches of each node are fabricated on a single integrated circuit. In addition to the three 16-bit ports, the IC package must allow for various read, write, and level address control lines. Another organization method would allow each chip to accommodate 8-bit ports; each node thereby requiring two chips. Since today’s IC packages allow a large number of pin connections, two or more levels of nodes could be fabricated on a single chip. This technique has the additional advantage of decreasing the delay associated with the cascaded buffers. The number of chips in a large interconnection network would also be decreased significantly.
Chapter 7 — Conclusions

In this paper, we have presented a multiprocessor architecture that is to be used in a development system for the applied research of adaptive filtering algorithms and applications. A multiprocessor architecture was necessary so that commercially available DSP chips could be used to obtain the throughput required for processing complex algorithms on speech signals. To minimize unnecessary delays resulting from interprocessor communications, all processors were exactly synchronized and the interconnection network was optimized for adaptive filtering operations. We have demonstrated BRACELET’s successful execution of the LMS algorithm on real-time signals in applications that include system identification, echo cancellation, adaptive equalization, and signal separation. Furthermore, we have mapped the more complex Fast Kalman and normalized Fast Transversal Filter algorithms to the BRACELET architecture and projected the system’s performance of these algorithms.

We believe we can measure the success of this project in several aspects. First, we have a greater appreciation for the versatility of the adaptive filter in performing such applications as echo cancellation, noise cancellation, and adaptive equalization. We have been able to explore various adaptive algorithms so that we could more fully understand their operation and could therefore optimize a system for their implementation. Through the implementation of our BRACELET architecture, we have been able to prove the feasibility of a tightly coupled SIMD network of TMS32010 DSP chips executing a global program from a common external memory space. We have also shown how the master/slave relationship permits BRACELET to perform time consuming scalar operations (square root, division) and vector operations simultaneously. As a result of the considerable experience we gained from actually programming adaptive filtering algorithms on
BRACELET, we were able to suggest improvements to the TMS32010's internal data paths that would double the maximum throughput of our adaptive filters. While these improvements have the effect of reducing the need for a large multiprocessor system, more complex algorithms and our desire for higher throughput motivated our design of a modular interconnection network suitable for large scale integration to achieve a significant reduction in hardware and a corresponding increase in expandability.

Perhaps the most rewarding aspect of the design and implementation of BRACELET is the operational hardware and the library of software routines that we now have available for future research. This research could include testing algorithms and applications on real-time signals, analyzing the effects of finite register lengths, developing more intelligent decision-making abilities for the master processor, judging actual filter performance based on human perceptual factors, or simply using BRACELET as an array processor to achieve fast simulations. We have expounded on the versatility of a development system built around the adaptive filter. Adaptive techniques continue to win widespread approval as the theoretical and hardware complexities are overcome by a better understanding by communication engineers of the underlying theories, by the introduction of efficient and robust adaptive algorithms, and by the availability of high speed programmable DSP chips with floating point arithmetic. We feel the modifications to the TMS32010 and the modular interconnection network discussed in this paper are important results that can significantly improve the implementation of adaptive digital filtering applications. Since BRACELET is not just a theoretical study, but is instead a practical system for developing adaptive filters, we can continue to expand our knowledge and advance the use of adaptive digital filters in solving problems associated with today's telecommunications network.
REFERENCES


# TMS32010 Instruction Set Summary

## Accumulator Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>No Cycles</th>
<th>No Words</th>
<th>Opcode Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>Absolute value of accumulator</td>
<td>1</td>
<td>1</td>
<td>0111111111001000</td>
</tr>
<tr>
<td>ADD</td>
<td>Add to accumulator with shift</td>
<td>1</td>
<td>1</td>
<td>0000000101000000</td>
</tr>
<tr>
<td>ADDH</td>
<td>Add to high order accumulator bits</td>
<td>1</td>
<td>1</td>
<td>0010000001000000</td>
</tr>
<tr>
<td>ADDS</td>
<td>Add to accumulator with no sign extension</td>
<td>1</td>
<td>1</td>
<td>0110000011000000</td>
</tr>
<tr>
<td>AND</td>
<td>AND with accumulator</td>
<td>1</td>
<td>1</td>
<td>0111110111010101</td>
</tr>
<tr>
<td>LAC</td>
<td>Load accumulator with shift</td>
<td>1</td>
<td>1</td>
<td>0100000001000000</td>
</tr>
<tr>
<td>LACX</td>
<td>Load accumulator immediate</td>
<td>1</td>
<td>1</td>
<td>0111111111000000</td>
</tr>
<tr>
<td>OR</td>
<td>OR with accumulator</td>
<td>1</td>
<td>1</td>
<td>0111111010101010</td>
</tr>
<tr>
<td>SACH</td>
<td>Store high-order accumulator bits with shift</td>
<td>1</td>
<td>1</td>
<td>0010110101010101</td>
</tr>
<tr>
<td>SACL</td>
<td>Store low-order accumulator bits</td>
<td>1</td>
<td>1</td>
<td>0101000001000000</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtract from accumulator with shift</td>
<td>1</td>
<td>1</td>
<td>0001010000000100</td>
</tr>
<tr>
<td>SUBC</td>
<td>Conditional subtract for divide</td>
<td>1</td>
<td>1</td>
<td>0110010010001010</td>
</tr>
<tr>
<td>SUBH</td>
<td>Subtract from high order accumulator bits</td>
<td>1</td>
<td>1</td>
<td>0111000010101010</td>
</tr>
<tr>
<td>SUBS</td>
<td>Subtract from accumulator with no sign extension</td>
<td>1</td>
<td>1</td>
<td>0110000111000000</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive OR with accumulator</td>
<td>1</td>
<td>1</td>
<td>0111111000101010</td>
</tr>
<tr>
<td>ZAC</td>
<td>Zero accumulator</td>
<td>1</td>
<td>1</td>
<td>0111111111111001</td>
</tr>
<tr>
<td>ZALH</td>
<td>Zero accumulator and load high-order bits</td>
<td>1</td>
<td>1</td>
<td>0110010111111111</td>
</tr>
<tr>
<td>ZALS</td>
<td>Zero accumulator and low-order bits</td>
<td>1</td>
<td>1</td>
<td>0110011110111111</td>
</tr>
</tbody>
</table>

## Auxiliary Register and Data Page Pointer Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>No Cycles</th>
<th>No Words</th>
<th>Opcode Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAR</td>
<td>Load auxiliary register</td>
<td>1</td>
<td>1</td>
<td>0001111000101010</td>
</tr>
<tr>
<td>LAPK</td>
<td>Load auxiliary register immediate</td>
<td>1</td>
<td>1</td>
<td>0111000010000000</td>
</tr>
<tr>
<td>LAPP</td>
<td>Load auxiliary register pointer immediate</td>
<td>1</td>
<td>1</td>
<td>0110100010001000</td>
</tr>
<tr>
<td>LPDP</td>
<td>Load data memory page pointer</td>
<td>1</td>
<td>1</td>
<td>0110111110101010</td>
</tr>
<tr>
<td>LPDK</td>
<td>Load data memory page pointer immediate</td>
<td>1</td>
<td>1</td>
<td>0111011100000000</td>
</tr>
<tr>
<td>MAR</td>
<td>Modify auxiliary register and pointer</td>
<td>1</td>
<td>1</td>
<td>0110100110110111</td>
</tr>
<tr>
<td>SAR</td>
<td>Store auxiliary register</td>
<td>1</td>
<td>1</td>
<td>0011000010001010</td>
</tr>
<tr>
<td><strong>Mnemonic</strong></td>
<td><strong>Description</strong></td>
<td><strong>No. of Cycles</strong></td>
<td><strong>No. Within</strong></td>
<td><strong>Opcode</strong></td>
</tr>
<tr>
<td>--------------</td>
<td>----------------</td>
<td>------------------</td>
<td>---------------</td>
<td>-----------</td>
</tr>
<tr>
<td>B</td>
<td>Branch unconditionally</td>
<td>2</td>
<td>1</td>
<td>0 1 1 1 0 0 1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>BANZ</td>
<td>Branch on auxiliary register not zero</td>
<td>2</td>
<td>1</td>
<td>0 1 1 1 0 1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>BEZ</td>
<td>Branch if accumulator = 0</td>
<td>2</td>
<td>1</td>
<td>0 1 1 1 1 1 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>BRZ</td>
<td>Branch if accumulator &lt; 0</td>
<td>2</td>
<td>1</td>
<td>0 1 1 1 1 1 1 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>BNZ</td>
<td>Branch if accumulator &gt; 0</td>
<td>2</td>
<td>1</td>
<td>0 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>R</td>
<td>Branch on overflow</td>
<td>2</td>
<td>1</td>
<td>0 1 1 1 1 1 0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>C</td>
<td>Branch if accumulator = A</td>
<td>2</td>
<td>1</td>
<td>0 1 1 1 1 0 1 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>CALL</td>
<td>Call subroutine from accumulator</td>
<td>2</td>
<td>1</td>
<td>0 1 1 1 1 1 1 1 1 0 0 0 0 0</td>
</tr>
<tr>
<td>RET</td>
<td>Return from subroutine or interrupt routine</td>
<td>2</td>
<td>1</td>
<td>0 1 1 1 1 1 1 1 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

**T-Register, Predecesor and Multiply Instructions**

<table>
<thead>
<tr>
<th><strong>Mnemonic</strong></th>
<th><strong>Description</strong></th>
<th><strong>No. of Cycles</strong></th>
<th><strong>No. Words</strong></th>
<th><strong>Opcode</strong></th>
<th><strong>Instruction Register</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>APAC</td>
<td>Add P register to accumulator</td>
<td>1</td>
<td>1</td>
<td>0 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>LT</td>
<td>Load T-register</td>
<td>1</td>
<td>1</td>
<td>0 1 1 1 0 1 0 1 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>LTAD</td>
<td>Load T-register</td>
<td>1</td>
<td>1</td>
<td>0 1 1 1 0 1 0 1 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>LTAM</td>
<td>Load T-register and APAC into one instruction</td>
<td>1</td>
<td>1</td>
<td>0 1 1 1 0 1 0 1 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>LTD</td>
<td>Load T-register and APAC into one instruction</td>
<td>1</td>
<td>1</td>
<td>0 1 1 1 0 1 0 1 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>MPY</td>
<td>Multiply T register, store product in P register</td>
<td>1</td>
<td>1</td>
<td>0 1 1 0 1 0 1 1 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>MPYE</td>
<td>Multiply T register, store product in P register</td>
<td>1</td>
<td>1</td>
<td>0 1 1 0 1 0 1 1 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>LAC</td>
<td>Load accumulator from P register</td>
<td>1</td>
<td>1</td>
<td>0 1 1 1 1 1 1 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>GPAC</td>
<td>Subtract P register from accumulator</td>
<td>1</td>
<td>1</td>
<td>0 1 1 1 1 1 1 1 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>
clock timing

NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

MEMORY AND PERIPHERAL INTERFACE TIMING

switching characteristics over recommended operating conditions

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TEST CONDITIONS</th>
<th>MIN</th>
<th>TYP</th>
<th>MAX</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>t_{d1}</em></td>
<td>Delay time CLKOUT to address bus valid (See Not5)</td>
<td>10\text{ns}</td>
<td>20</td>
<td>50</td>
<td>ns</td>
</tr>
<tr>
<td><em>t_{d7}</em></td>
<td>Delay time CLKOUT to D7</td>
<td>10\text{ns}</td>
<td>15</td>
<td>50</td>
<td>ns</td>
</tr>
<tr>
<td><em>t_{d8}</em></td>
<td>Delay time CLKOUT to WE</td>
<td>10\text{ns}</td>
<td>15</td>
<td>50</td>
<td>ns</td>
</tr>
<tr>
<td><em>t_{d9}</em></td>
<td>Delay time CLKOUT to WE</td>
<td>10\text{ns}</td>
<td>15</td>
<td>50</td>
<td>ns</td>
</tr>
<tr>
<td><em>t_{d10}</em></td>
<td>Delay time CLKOUT to data bus OUT valid</td>
<td>10\text{ns}</td>
<td>15</td>
<td>50</td>
<td>ns</td>
</tr>
<tr>
<td><em>t_{d11}</em></td>
<td>Time after CLKOUT that data bus starts to be driven</td>
<td>10\text{ns}</td>
<td>15</td>
<td>50</td>
<td>ns</td>
</tr>
<tr>
<td><em>t_{d12}</em></td>
<td>Time after CLKOUT that data bus stops being driven</td>
<td>10\text{ns}</td>
<td>15</td>
<td>50</td>
<td>ns</td>
</tr>
<tr>
<td><em>t_{d13}</em></td>
<td>Data bus OUT valid after CLKOUT</td>
<td>10\text{ns}</td>
<td>15</td>
<td>50</td>
<td>ns</td>
</tr>
</tbody>
</table>

NOTE: Address bus will be valid upon WE or DEN = 0. Value of *t_{d1}* is worst case

These values were derived from characterization data and are not tested.

reset timing

NOTE: *R* can occur anytime during a clock cycle. *T* is a minimum to ensure synchronous operation.

These values were derived from characterization data and are not tested.

reset requirements over recommended operating conditions

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>TEST CONDITIONS</th>
<th>MIN</th>
<th>TYP</th>
<th>MAX</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>t_{d14}</em></td>
<td>Setup data bus valid prior to CLKOUT</td>
<td>50</td>
<td>na</td>
<td>na</td>
<td>ns</td>
</tr>
<tr>
<td><em>t_{d10}</em></td>
<td>Hold data bus valid after CLKOUT</td>
<td>0</td>
<td>na</td>
<td>na</td>
<td>ns</td>
</tr>
</tbody>
</table>

NOTE: Data may be removed from the data bus upon setup or DEN preceding CLKOUT.

NOTE: 1 *R* forces DEN and DEN high and interface data bus DO through D15. AB outputs (and program counter) must be synchronously cleared in each clock cycle.

2 Program must be maintained to a maximum of two clock cycles.

3 Program must be maintained to a maximum of two clock cycles.

4 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

5 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

6 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

7 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

8 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

9 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

10 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

11 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

12 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

13 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

14 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

15 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

16 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

17 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

18 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

19 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

20 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

21 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

22 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.

23 Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 3.0 volts, unless otherwise noted.
INTERRUPT (INT) TIMING

timing requirements over recommended operating conditions

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>MIN</th>
<th>TYP</th>
<th>MAX</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT (H)</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>us</td>
</tr>
<tr>
<td>t(HI)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>us</td>
</tr>
<tr>
<td>t(HI) Setup time INT</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>us</td>
</tr>
</tbody>
</table>

NOTE: INT is an asynchronous input and can occur anytime during a clock cycle. Time given for t(HI) is minimum to ensure asynchronous operation.

Interrupt timing

CLKOUT

INT

t(HI)

NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

I/O (BDI) TIMING

timing requirements over recommended operating conditions

<table>
<thead>
<tr>
<th>PARAMETER</th>
<th>MIN</th>
<th>TYP</th>
<th>MAX</th>
<th>UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>BD (L)</td>
<td>15</td>
<td>15</td>
<td>15</td>
<td>us</td>
</tr>
<tr>
<td>t(BDI)</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>us</td>
</tr>
<tr>
<td>t(BDI) Setup time BDI</td>
<td>50</td>
<td>50</td>
<td>50</td>
<td>us</td>
</tr>
</tbody>
</table>

NOTE: BDI is an asynchronous input. It can occur anytime within a clock cycle but should be maintained for a minimum of one clock cycle to ensure correct operation and correct evaluation.

BDI timing

CLKOUT

BDI

t(BDI)

NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.
LEGEND:
1. INSTRUCTION PREFETCH
2. NEXT INSTRUCTION PREFETCH
3. ADDRESS BUS VALID
4. PERIPHERAL ADDRESS VALID
5. ADDRESS BUS VALID
6. INSTRUCTION IN VALID
7. DATA IN VALID
8. INSTRUCTION IN VALID

NOTE: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.
SYNC.MAC is executed by BRACELET's master processor to synchronize the slave processors upon power-up.

The slaves are synchronized by asserting BIT6 of the STATUS WORD to a high state and sending the STATUS WORD to the STATUS PORT, thereby enabling the sync circuit.

Two instruction cycles after the sync circuit is enabled, the slaves will be synchronized and the sync circuit may be disabled by asserting BIT6 of the STATUS WORD to a low state and sending this new STATUS WORD to the STATUS PORT.

```
.LIST MC
.TITLE SYNCMAC
.GLOBAL START END
.RADIX 10.

; data and I/O port definitions

STATUS=7 ; status word register
PORTSTATUS=PA7 ; status output port address

START: B GO ; PC=000
       B INT ; PC=002 SET SOFTWARE BREAKPOINT HERE

GO:   LACK 207 ; = 1100 1111 ; BIT0=1 GLOBAL DATA INPUT from MASTER
SAFL STATUS ; BIT1=1 SAMPLE DATA INPUT from MASTER
OUT STATUS,PORTSTATUS ; BIT2=1 FILTER DATA OUTPUT from MASTER
LACK 143 ; = 1000 1111 ; BIT3=1 GLOBAL MEMORY ADDRESS from MASTER
SAFL STATUS ; BIT4=0 SLAVE RESET = 0
OUT STATUS,PORTSTATUS ; BIT5=0 unused at present
   B INT ; BIT6=0 disable SYNC CIRCUIT
          ; =1 enable SYNC CIRCUIT
       B INT ; BIT7=1 GLOBAL MEMORY ADDRESS autocount up

END:   .END
```
B.2. – Listing of BRACELET Macro Routines

**BRACELET.MAC**

Macro routines for DSP algorithms on BRACELET with four slaves

**Russell L. Barnes**

CCSP – NCSU

September 1988

By using .INCLUDE macro, these routines may be included in source code of a particular program.

---

**INITZ**: Initialize register ZERO to be 0 and register ONE to be 1

- ONE = ZERO + 1

**ZERO** – register to contain 0000 hex

**ONE** – register to contain 0001 hex

4 words 4 cycles

.MACRO INITZ ZERO, ONE

SACL ZERO ; Initialize ZERO = 0000 hex
SACL ONE ; Initialize ONE = 0001 hex
.
.ENDM

---

**INITSTATUS**: Initialize the master’s STATUS PORT

- To be used at the beginning of a program to RESET slaves and to set STATUS in a known state

**STATUS** – register containing STATUS WORD

**PORTSTATUS** – port address used to load STATUS with STATUS WORD

3 words 4 cycles

.MACRO INITSTATUS STATUS, PORTSTATUS

LACK 143 ; STATUS WORD = 1111 0111
AND STATUS ; read BIT0
SACL STATUS ; reset BIT0
LACK 18 ; STATUS WORD = 0001 0000
OR STATUS ; set BIT4
SACL STATUS ; set BIT4
OUT STATUS, PORTSTATUS ; start slaves with GLOBAL MEMORY ADDRESS from MASTER
.
.ENDM

---

**STOPSLAVES** halts the slaves by bringing their RESET lines low and then switches the source of the GLOBAL MEMORY ADDRESS to be the master

- Only the RESET and GLOBAL MEMORY ADDRESS SOURCE bits in the STATUS WORD are changed

**STATUS** – register containing STATUS WORD

**PORTSTATUS** – port address used to load STATUS with STATUS WORD

8 words 10 cycles

.MACRO STOPSLAVES STATUS, PORTSTATUS

LACK 238 ; STATUS WORD = 1110 1111
AND STATUS ; reset BIT4
SACL STATUS ; reset BIT4
OUT STATUS, PORTSTATUS ; stop SLAVES
LACK 0B ; STATUS WORD = 0000 1000
OR STATUS ; set BIT3
SACL STATUS ; set BIT3
OUT STATUS, PORTSTATUS ; GLOBAL MEMORY ADDRESS from MASTER
.
.ENDM

---

**STARTSLAVES** switches the source of the GLOBAL MEMORY ADDRESS to be slave and starts the slaves by bringing their RESET lines high

- Only the RESET and GLOBAL MEMORY ADDRESS SOURCE bits in the STATUS WORD are changed

**STATUS** – register containing STATUS WORD

**PORTSTATUS** – port address used to load STATUS with STATUS WORD

7 words 8 cycles

.MACRO STARTSLAVES STATUS, PORTSTATUS

LACK 243 ; STATUS WORD = 1111 0111
AND STATUS ; read BIT0
SACL STATUS ; read BIT0
LACK 16 ; STATUS WORD = 0001 0000
OR STATUS ; set BIT4
SACL STATUS ; set BIT4
OUT STATUS, PORTSTATUS ; start slaves with GLOBAL MEMORY ADDRESS from SLAVE
.
.ENDM

---

**DOWNLOAD** transfers a block of code in the master’s program memory to the slaves’ global memory

**SOURCE** – address in master’s memory of first word to be downloaded
MACRO WAVESHIFT DATA,N
REG=DATA,N REPT N-1
REG=REG-1 DNDO REG ; shift contents of register REG
END ; into register NREG+1
.ENDW

SLAVECSHIFT performs a vector shift operation on a 4M element vector
distributed among four slave processors (M elements per slave).
DATA - address in master's data memory of first vector element
PORT - port address used to shift last vector element of one slave
SFL - number of elements in vector to be shifted
M = number of elements in vector.

MACRO SLAVECSHIFT DATA,PORT,M
REG=DATA,M REPT M-1
REG=REG-1 DNDO REG ; shift contents of register REG
END ; into first vector element from PORT
.ENDW

TRANSVERSAL multiplies the M corresponding elements
of vectors XDATA and WDATA and accumulates these M inner products
in register INNPROD.
DATA - address of register containing the first element of vector XDATA
WDATA - address of register containing the first element of vector WDATA
INNPROD - register containing the final accumulation of the inner products
SHIFT - scaling factor of final accumulation when starting in INNPROD
PORT - port address used to shift last vector element of one slave
SFL = number of elements in vector XDATA and WDATA
M = number of elements in vectors XDATA and WDATA
2MN-6 words
3MN cycles

MACRO TRANSVERSAL XDATA,WDATA,INNPROD,XSHIFT,PORT,M
XREG+DATA,M=0 ; clear R register
WREG=DATA,M=1 ; clear running accumulation
OUT XREG,PORT
.REPT M-1
XREG=XREG-1 ; (XREG) = (XREG) + previous inner product to accumulator and shift
MPV WREG ; WREG = multiply content of XREG by contents of WREG
WREG=WREG-1 ; load T register with contents of XREG
WDATA=DATA,M=1 ; save final accumulation with shift
MPV WREG ; WREG = multiply contents of XDATA by contents of WREG
WREG=WREG-1 ; load T register with contents of XREG
WDATA=DATA,M=1 ; add previous inner product to accumulator
MPV WREG ; WREG = multiply contents of XREG by contents of WREG
APAC; INNPROD,XSHIFT ; and last inner product to accumulator
SACH INNPROD,XSHIFT ; save final accumulation with shift
.ENDW

PRODUCT performs the multiplication of the corresponding M elements
of vectors XDATA and WDATA and accumulates these M inner products
in register INNPROD.
XDATA - address of register containing the first element of vector XDATA
WDATA - address of register containing the first element of vector WDATA
INNPROD - register containing the final accumulation of the inner products
SHIFT - scaling factor of final accumulation when starting in INNPROD
PORT - port address used to shift last vector element of one slave
SFL = number of elements in vectors XDATA and WDATA
M = number of elements in vectors XDATA and WDATA
2MN-3 words
2MN-3 cycles

MACRO PRODUCT XDATA,WDATA,INNPROD,XSHIFT,PORT,M
ZAC ; clear running accumulation
IT XDATA ; load T register with contents of XDATA
MPV WDATA ; multiply contents of XDATA by contents of WDATA
XREG=XDATA=M=1 ; clear R register
WREG=DATA,M=1 ; add previous inner product to accumulator
MPV WREG ; WREG = multiply content of XDATA by contents of WREG
WREG=WREG-1 ; load T register with contents of XDATA
WDATA=DATA,M=1 ; add previous inner product to accumulator
MPV WREG ; WREG = multiply contents of XDATA by contents of WREG
APAC; INNPROD,XSHIFT ; and last inner product to accumulator
SACH INNPROD,XSHIFT ; save final accumulation with shift
.ENDW

ACCUMULATE adds the contents of the data register DATA in all of the
slave processors connected by PORT1 and then adds this sum in all of
the slave processors connected by PORT2 (used to accumulate all of the
inner products of a 4 processor system producing the final sum in DATA).
114
DATA - address of register containing data to be accumulated
PORT1 - port address of first slave interconnection (eg CPU)
PORT2 - port address of second slave interconnection (eg DMA/HI)
Listing of slave routines which master must download to global memory

located in global program memory location 000, SLAVES is the first
instruction executed when slaves' RESETs go high

SLAVES: IN PROGM,GLOBAL ; Input slave program address from master
  LAC PRL,A ; load accumulator with program address
  CALL CALR ; call routine at program address

PROG1 initializes the slave processors

PROG1: IDPR 0
  SOVM
  CLEARBLOCK WMS,SLATAPS
  CLEARBLOCK WM,SLATAPS
  B WAIT-SLAVES+START

SYNC 0 -- SLAVES begin LMS algorithm
1. Instruction cycles

SYNC 1 -- MASTER does error calculation for LMS

SYNC 2 -- SLAVES get adjustment factor from MASTER and adapt weight vector

SYNC 3

SYNC 0 -- SLAVES begin LMS adaptive filter
1. Output DATA to SLAVES

SLAVES start with next instruction

SYNC 0 -- SLAVES begin LMS algorithm
1. Instruction cycles

SYNC 1 -- error calculation for LMS

SYNC 2 -- broadcast adjustment factor to slaves

SYNC 3

SYNC 0 -- SLAVES begin LMS algorithm
1. Instruction cycles

SYNC 1 -- error calculation for LMS

SYNC 2 -- broadcast adjustment factor to slaves

SYNC 3

END

STOP-SLAVES STATUS,POW STATUS

B I O L O O P

10

2
FMODEL MAC performs system identification application using the
FAST KALMAN algorithm on the BRACELET multiprocessor

This program has not been tested and may require initialization of
some variables before processing signals and scaling of some
variables while processing signals.

clist MAC
Listing finish

include /bracelet/mac/

define number of taps stored in master (MATAPS) and in each slave (SLATAPS)

MATAPS=12
SLATAPS=8

master I/O port definitions

GLOBAL+PA0 1 global data I/O port - input from A/D, output to slaves
SAMPLE+PA1 1 sample data I/O port - input from A/D, output to slaves
FILTER+PA2 1 filter data I/O port - input from slaves, output to A/D
COMMUTER+PA4 1 output slave clears the global memory address counter
COMMUTER+PA4 1 global memory address counter initialization port
DATA+PA5 1 global data I/O port
PORTSTATUS+PA7 1 status output port

slave I/O port definitions

GLOBAL+PA0 1 global data I/O port - input from A/D, output to slaves
SAMPLE+PA1 1 sample data I/O port - input from A/D, output to slaves
FILTER+PA2 1 filter data I/O port - input from slaves, output to A/D
COMMUTER+PA4 1 output slave clears the global memory address counter
COMMUTER+PA4 1 global memory address counter initialization port
DATA+PA5 1 global data I/O port
PORTSTATUS+PA7 1 status output port

master data definitions

DEFINITIONS FOR DATA RAM ONLY

ZER0=0 1 constant zero
ONE=1 1 constant one
FERR=M-3 1 forward prediction error
RESIDUA=3 1 forward residual multiplied by residual power
RRED=M-4 1 backward prediction error
VSCALE=5 1 last element of extended KALMAN gain vector
RSCALE=5 1 current error variance
LVE=5 1 current delayed response input sample
RVE=5 1 current filter input sample
V=1 1 current filter output value
FORWARD+11 1 forward prediction
RESIDUA=12 1 residual power
RRED+13 1 forward residual
N=15 1 filter input delayed N samples
BACKWARD+15 1 backward prediction
AN=16 1 exponential smoothing constant (forgetting factor)
LSCALE+18 1 last source address in master memory
DS+19 1 destination address in master memory
DATA+20 1 data to transfer
COMMUTER+21 1 loop control variable
STATUS+22 1 status word
MAX+23 1 max value for A/D conversion
OFF+24 1 offset value for A/D and D/A conversion
PRG+25 1 vector address for slave routine PRG+2
PRG+26 1 vector address for slave routine PRG+2

slave data definitions

DEFINITIONS FOR DATA RAM ONLY

PRG+24 1 contains vector address
FROM=1 1 temporary variables for storing accumulated inner products
FOREBR+2 1 as defined for master
RESSIGMA+3 1 as defined for master
BARN+4 1 as defined for master
ALPHA+5 1 as defined for master
VECTOR+6 1 as defined for master
KVE=7 1 as defined for master
VVE+8 1 delayed input sample vector
VVE+9 1 adaptive weight vector
VVE+10 1 forward prediction vector
VVE+11 1 backward prediction vector
VVE+12 1 extended KALMAN gain vector
VVE+13 1 first N elements of extended KALMAN gain vector

121
Listing of slave routines which master must download to global memory

Locate in global program memory location 000, SLAVES is the address
of the first instruction executed when slaves' RESETS go high.

SLAVES: IN PROGRAM GLOBAL
Input slave program memory from master
SLAVES: IN LAC PROM
Input slave program memory from master
SLAVES: IN CALA
Input slave program address from master
SLAVES: IN SLAVES
Input slave program address from master

SYNC 0 -- SLAVES begin PAST KALMAN algorithm to model an unknown filter

SYNC 1 -- calculate filter output Y

SYNC 2 -- update forward prediction vector

SYNC 3 -- shift delayed input sample vector

SYNC 4 -- update VVEC

SYNC 5 -- shift extended KALMAN gain vector

SYNC 6 -- normalize backward prediction vector with ALPHA

SYNC 7 -- update KALMAN gain vector and update adaptive weight vector W

SYNC 8 -- update forward prediction vector

END: END
B.5. System Identification using
Normalized Fast Transversal Filter Algorithm
```
; Listing of master routines

; Get next sample

; Initialize master processor

; Download slaves' program to global memory

; Initialize vector addresses for restarting slaves

; Initialize slaves

; Start slaves

; Initialize INVLAEMDA value in slaves
```
SYNC 1 -- forward prediction error calculation

IN X, FILTER
LAC ETAN
ADH
SAC ETAN
NORMS ETAN, GAMMA, R4, 1
LAC ETAN
THEB FILTER ; TABLE LOOK UP
OUT ETAN, GLOBAL
Cycles R + 1
END CYCLES

SYNC 2 -- update nth element of forward prediction vector

IN X, FILTER
OUT ZERO SAMPLE
IN CH, FILTER
NORMS AN, INV AMDA, R4, 1
LT ETAN
LAC AN, S12
MPY CH
ADD
MPY AN
CATTR AN, X4
LAC CH, S12
SUB X4, S12
NORM2 AN, IN, ECOS, R4, 1
OUT ECOS, GLOBAL
Cycles R + 1
END CYCLES

SYNC 3 -- use nth element of KAMAN gain vector to calculate RSE

ZAC
SUB CH
CALS RSE
NORMS RSE, GAMMA, X4, 1
NORMS RSE, BETA, X4, 1
OUT RSE, GLOBAL
Cycles R + 1
END CYCLES

SYNC 4 -- use RSE in determining RSE by table look-up

LAC RSE
TDR RSEC
TABLE LOOK UP
OUT RSEC, GLOBAL
Cycles R + 1
END CYCLES

SYNC 5 -- determine current error value and scale by GAMMA

NORMS GAMMA, ECOS, R4, 1
NORMS GAMMA, SEC, R4, 1
LAC V
AND 0
SAC EPSGAM
NORM EPSGAM, GAMMA, R4, 1
OUT EPSGAM, GLOBAL
Cycles R + 1
END CYCLES

SYNC 6 -- scale RSE with ASIG and LAMDA

NORMS BETA, RSEC, X4, 1
NORMS BETA, LAMDA, X4, 1
Cycles R + 1
END CYCLES

SYNC 7 -- use RSE with ASIG and LAMDA

STOPSLAVE STATUS, POURS, STATUS
B ENDDUP

SYNC 8 --

Cycles R + 1
END CYCLES

SYNC 9 --

Cycles R + 1
END CYCLES

SYNC 10 --

Cycles R + 1
END CYCLES
LISTING OF SLAVE ROUTINES WHICH MASTER MUST DOWNLOAD TO GLOBAL MEMORY

1. Instruction executed when slaves RESET by high

SLAVES: IN  PROG,GLOBAL | Input slave program address from master
ECAC      | Link accumulator with program address
CALA      | Call routine at program address

PROG1 initializes the slave processors

PROG1: LDPRO 0
        SWIVM
        CLEARBUFFER FFT,SLATAPS
        CLEARBUFFER FFT,SLATAPS
        CLEARBUFFER FFT,SLATAPS
        CLEARBUFFER FFT,SLATAPS
        BEGIN 1 IN SLAVE1 ONLY
        IN  INV Leadership,GLOBAL

0  WAIT-SLAVE'SSTART

SYNC 0  -- SLAVES use FFT algorithm to model an unknown filter
          calculate forward prediction

SYNC 1  -- calculate filter output

SYNC 2  -- update forward prediction vector and RALMAN gain vector

SYNC 3  -- normalize RALMAN gain vector with EOS

SYNC 4  -- update backward prediction vector and RALMAN gain vector

SYNC 5  -- calculate output

SYNC 6  -- normalize backward prediction vector and RALMAN gain vector
          with vector

SYNC 7  -- update adaptive weight vector

SYNC 8  -- update backward prediction vector and RALMAN gain vector

LASTNRD: NOP

END 1. END
C.1. – Summary of Least Mean Squares Operations

input \( x(T) \)

input \( d(T) \)

\[
\begin{bmatrix}
X(T) \\
0
\end{bmatrix} = \begin{bmatrix}
x(T) \\
x(T-1)
\end{bmatrix}
\]

\( y(T) = X^T(T)W(T-1) \)

\( e(T) = y(T) - d(T) \)

\( W(T) = W(T-1) - \alpha e(T)X(T) \)

output \( y(T) \)
C.2. – Summary of Fast Kalman Operations

input \( x(T) \)

input \( d(T) \)

\[ e_{for}(T) = x(T) + X^T(T-1) \Delta(T-1) \]

\[ \Delta(T) = \Delta(T-1) - e_{for}(T)K(T-1) \]

\[ e_{res}(T) = x(T) + X^T(T-1) \Delta(T) \]

\[
\begin{bmatrix}
X(T) \\
x_N(T)
\end{bmatrix} = \begin{bmatrix}
x(T) \\
X(T-1)
\end{bmatrix}
\]

\[ v(T) = x_N(T) + X^T(T)D(T-1) \]

\[ \Sigma(T) = \gamma \Sigma(T-1) + e_{for}(T)e_{res}(T) \]

\[ M^+(T) = K(T-1) + \frac{e_{res}(T)}{\Sigma(T)} \Delta(T) \]

\[
\begin{bmatrix}
M(T) \\
\mu(T)
\end{bmatrix} = \begin{bmatrix}
e_{res}(T) \\
\Sigma(T)
\end{bmatrix} | M^-(T)
\]

\[ D(T) = \frac{D(T-1) - v(T)M(T)}{1 - v(T)\mu(T)} \]

\[ K(T) = M(T) - \mu(T)D(T) \]

\[ y(T) = X^T(T)W(T-1) \]

\[ e(T) = d(T) - y(T) \]

\[ W(T) = W(T-1) + e(T)K(T) \]

output \( y(T) \)
C.3. – Summary of Fast Transversal Filter Operations

input $x(T)$

input $d(T)$

$$\begin{bmatrix} X(T) | x_N(T) \end{bmatrix} = \begin{bmatrix} x(T) & X(T-1) \end{bmatrix}$$

$$\tan\theta = \lambda^{-h} \gamma^\eta(T-1) \left\{ X^T(T)A(T-1) + x_N(T)a_N(T-1) \right\}$$

$$\cos\theta = \left\{ 1 + \tan^2\theta \right\}^{-h}$$

$$\begin{bmatrix} C^-(T) | c_N(T) \end{bmatrix} = \cos\theta \left\{ \begin{bmatrix} 0 & C(T-1) \end{bmatrix} - \lambda^{-h} \tan\theta \begin{bmatrix} A(T-1) | a_N(T-1) \end{bmatrix} \right\}$$

$$\begin{bmatrix} \Delta(T) | a_N(T) \end{bmatrix} = \cos\theta \left\{ \lambda^{-h} \begin{bmatrix} \Delta(T-1) | a_N(T-1) \end{bmatrix} + \tan\theta \begin{bmatrix} 0 & C(T-1) \end{bmatrix} \right\}$$

$$\sin\phi = -\lambda^h \beta^\varsigma(T-1) c_N(T)$$

$$\sec\phi = \left\{ 1 - \sin^2\phi \right\}^{-h}$$

$$\beta^\varsigma(T) = \lambda^h \beta^\varsigma(T-1) \sec\phi$$

$$B(T) = \sec\phi \left\{ \lambda^{-h} B(T-1) + \sin\phi C^-(T) \right\}$$

$$C(T) = \sec\phi \left\{ C^-(T) - \lambda^{-h} \sin\phi B(T-1) \right\}$$

$$y(T) = X^T(T)W(T-1)$$

$$\gamma^\eta(T) = \gamma^\eta(T-1) \cos\theta \sec\phi$$

$$e(T) = y(T) + d(T)$$

$$W(T) = W(T-1) + \gamma^\eta(T)e(T)C(T)$$

output $y(T)$
Table 8 - Maximum Sampling Frequency (Hz)
Least Mean Squares Algorithm - BRACELET Network
Modified TMS32010 Data Path - 150 ns Instruction Cycle

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>74906</td>
<td>104167</td>
<td>121212</td>
<td>123457</td>
<td>116959</td>
<td>P &gt; N</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>32</td>
<td>43573</td>
<td>69444</td>
<td>93897</td>
<td>107527</td>
<td>109290</td>
<td>104167</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>64</td>
<td>23725</td>
<td>41667</td>
<td>64725</td>
<td>85470</td>
<td>96618</td>
<td>98039</td>
<td>93897</td>
</tr>
<tr>
<td>128</td>
<td>12415</td>
<td>23148</td>
<td>39920</td>
<td>60606</td>
<td>78431</td>
<td>87719</td>
<td>88889</td>
</tr>
<tr>
<td>256</td>
<td>6355</td>
<td>12255</td>
<td>22599</td>
<td>38314</td>
<td>56980</td>
<td>72464</td>
<td>80321</td>
</tr>
<tr>
<td>512</td>
<td>3216</td>
<td>6133</td>
<td>12099</td>
<td>22075</td>
<td>36832</td>
<td>53763</td>
<td>67340</td>
</tr>
</tbody>
</table>

Table 9 - Maximum Sampling Frequency (Hz)
Fast Kalman Algorithm - BRACELET Network
Modified TMS32010 Data Path - 150 ns Instruction Cycle

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>18215</td>
<td>28490</td>
<td>36630</td>
<td>39216</td>
<td>37453</td>
<td>P &gt; N</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>32</td>
<td>9718</td>
<td>16920</td>
<td>25445</td>
<td>31746</td>
<td>33670</td>
<td>32362</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>64</td>
<td>5028</td>
<td>9337</td>
<td>15798</td>
<td>22989</td>
<td>28011</td>
<td>29499</td>
<td>28490</td>
</tr>
<tr>
<td>128</td>
<td>2558</td>
<td>4924</td>
<td>8985</td>
<td>14815</td>
<td>20964</td>
<td>25063</td>
<td>26247</td>
</tr>
<tr>
<td>256</td>
<td>1290</td>
<td>2531</td>
<td>4824</td>
<td>8658</td>
<td>13947</td>
<td>19268</td>
<td>22676</td>
</tr>
<tr>
<td>512</td>
<td>648</td>
<td>1284</td>
<td>2504</td>
<td>4728</td>
<td>8354</td>
<td>13175</td>
<td>17825</td>
</tr>
</tbody>
</table>

Table 10 - Maximum Sampling Frequency (Hz)
Normalized FTF Algorithm - BRACELET Network
Modified TMS32010 Data Path - 150 ns Instruction Cycle

<table>
<thead>
<tr>
<th>Taps (N)</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
<th>32</th>
<th>64</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>18365</td>
<td>30722</td>
<td>44150</td>
<td>53333</td>
<td>56022</td>
<td>P &gt; N</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>32</td>
<td>9761</td>
<td>17683</td>
<td>28860</td>
<td>40404</td>
<td>47962</td>
<td>50125</td>
<td>P &gt; N</td>
</tr>
<tr>
<td>64</td>
<td>5039</td>
<td>9565</td>
<td>17050</td>
<td>27211</td>
<td>37244</td>
<td>43573</td>
<td>45351</td>
</tr>
<tr>
<td>128</td>
<td>2561</td>
<td>4986</td>
<td>9376</td>
<td>16461</td>
<td>25740</td>
<td>34542</td>
<td>39920</td>
</tr>
<tr>
<td>256</td>
<td>1291</td>
<td>2547</td>
<td>4935</td>
<td>9195</td>
<td>15911</td>
<td>24420</td>
<td>32206</td>
</tr>
<tr>
<td>512</td>
<td>618</td>
<td>1288</td>
<td>2534</td>
<td>4884</td>
<td>9021</td>
<td>15396</td>
<td>23229</td>
</tr>
</tbody>
</table>
Internal Data Storage Capacity

We have shown that improving the performance of the TMS32010 allows a multiprocessor with fewer slaves to perform a given application. A consequence of decreasing the number of processors in the system is to increase the number of filter coefficients contained within each processor. Each slave must then be able to accommodate more vector data storage.

Table 1 illustrates the fact that the TMS32010 used in our multiprocessor does not have sufficient storage for the LMS algorithm. This problem is only compounded by our modified TMS32010s (Table 8). Although the normalized FTF algorithm requires five filters with \( N \) coefficients distributed among the slave processors, a greater number of instruction cycles are performed by the more complex algorithm. The result is a system that has sufficient data memory for the currently attainable throughput. However, the modified TMS32010s cause the Fast Transversal Filter to exceed data storage limitations as well (Table 10). Designers of more recent DSP chips have recognized the need for a large internal memory space. One example is the NEC \( \mu \text{PD77230} [14] \) which contains 1 K of 32-bit data RAM.

Floating Point Arithmetic

A major advancement to programming adaptive filtering algorithms on fast DSP chips is the availability of equally fast floating point operations. Programmers of NEC's \( \mu \text{PD77230} \) can take advantage of \( 32 \times 32 \)-bit floating point arithmetic operations that execute in a single 150 ns instruction cycle (including a multiply with a 55-bit product). For the relatively simple LMS algorithm programmed in LMSMODEL.MAC, integer arithmetic appears to be adequate, but a more complex algorithm, such as the Fast
Transversal Filter, requires divisions, normalizations, and square roots that complicate a filter designer's numerical analysis based on expected signal statistics. Floating point would greatly simplify programming by eliminating the shift operations required by the TMS32010's integer arithmetic. Since our FTFMODEL.MAC program is untested, we do not analyze the numerical processes to determine the optimum shift operations. However, we do note where these shift operations would occur.

6.2. Improvements to the Interconnection Network

The dedicated latches of BRACELET's interconnection network undoubtedly provide a fast method for interprocessor communication. The almost completely interconnected processors are able to use very simple software routines to share data for accumulating partial sums without having to broadcast the final result. Unfortunately, the hardware cost and board space is prohibitive should this system be expanded to 16 or more processors. Newer DSP chips, such as the INMOS Transputer, have multiple high speed serial links to reduce hardware external to the processors. However, the programmer must allow extra time for the serial communication to be completed, which could adversely affect the delays associated with data dependencies.

Our results demonstrate that by optimizing commercially available DSP chips for adaptive filters, we can reduce the necessity of large multiprocessing systems. We can then use simple interconnection schemes, such as a ring network, to provide interprocessor communications with either very little or no degradation to performance. However, some applications require higher throughput, more complex algorithms, or larger filters for real-time digital signal processing, so we must continue to explore more extensive interconnection networks for these larger multiprocessing systems.