CODEN: LUTFD2/(TFRT-5351)/1-148/(1986)

Implementation of Fast Recursive Parameter Estimation Algorithms on the Signal Processor TMS 320

Bo Bernhardsson

Department of Automatic Control
Lund Institute of Technology
May 1986

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
May 1986

Document Number
CODEN: LUTFD2/(TFRT-5351)/1-148/(1986)

Author(s)
Bo Bernhardsson

Supervisor
Sten Bergman, Per Persson

Sponsoring organisation
Asea Relays

Title and subtitle
Implementation of Fast Recursive Parameter Estimation Algorithms on the Signal Processor TMS 320

Abstract

Fault detection in connection with relay protection is usually related to measuring e.g. the impedance of a power system. Parameter estimation related to Black-box models of e.g. autoregressiv type gives parameters that do not have a direct physical interpretation but nevertheless can be valuable for fault detection.

This thesis discusses the use of fast recursive parameter estimation methods for fault detection in electrical power system. Both algorithmic and practical aspects are treated. For the implementation a signal processor, Texas TMS 320, is used. A code generator has been written which automatically produces assembly code for parameter estimation in an ARMAX-model. The performance has been simulated using VAX-software.

Key words

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
148

Recipient's notes

Security classification

CONTENTS

PREFACE

This thesis was written at the department of automatic control during the period of November 1984 to November 1985. I would like to thank all the people of the department for their support and enthusiasm. I am grateful to Per Persson, Karl Johan Åström and Rolf Johansson for suggestions on literature and practical hints concerning parameter estimation. I also thank Bengt Bengtsson at the department of tele transmission theory for hints concerning implementation on fast signal processors. Finally I am especially grateful to my advisorer at ASEA RELAYS, Sten Bergman, for an inspirational stay at ASEA in April 1985, and for his patience with this report.

Lund, November 1985

Bo Bernhardsson

"- So you claim that you can count, said the queen doubtfully. What is then one and one ?

- I dont know. I lost count, said Alice.

- Ha, she can't even do simple addition, said the queen laughing"

Lewis Caroll

1. Introduction

The purpose for this MS dissertation has been to explore the possibilities of using fast recursive parameter estimation methods for fault detection in electrical power system. This included the following tasks:

- Learn recursive parameter estimation methods.
- Study and select appropriate hardware.
- Develop software.
- Perform experiments with fault detection.

A power system is an electrical network. When short circuits or other faults occur the properties of the network changes. One possibility to detect a fault is thus to monitor the network parameters. The final goal was defined as follows : Determine the feasibility to estimate the R & L parameters from a given data record in less than a period i.e. 20 ms. This goal was successfully completed. Since rapid estimation is crucial a lot of effort was devoted to factors which influence convergence rates. This involves both algorithmic and hardware issues.

After several considerations it was decided to use a signal processor TMS 320. This was based partially on cost and availability. A drawback of the signal processor is that it has to be programmed in assembler. To make this in a convenient way a code generator was developed which generates TMS assembler based on a high level description of the estimation algorithm.

This report is organized as follows :

Parameter estimation with practical aspects on implementation is discussed in Section 2. The problem of detecting changes in the parameters ('faults') on-line is also discussed here. Different possibilities to implement the algorithms proposed in section 2 on-line have been investigated. The choice stood between using a floating point array processor to an IBM personal computer and a signal processor TMS 320 normally used for other applications.

The TMS 320 with development boards is described in Section 3. The new TMS 32020 processor with enhanced possibilities to floating point calculations is also discussed.

A drawback with today's signal processors is that they lack floating point numbers. Either all calculations must be done using fixed point numbers or floating point routines must be implemented in software. This is discussed in Section 4. A little unusual suggestion for the arithmetic format called the FOCUS number system with fractional exponent and no mantissa is also described here.

A procedure based system for automatically generating assembly code for the TMS 320 is described in Section 5. The code generator is written in Pascal, it is also shown how it could have been written in Prolog making an interactive programming environment possible.

The practical experiments performed are described in Section 6.

Section 7 contains a reference list and five appendices with programs etc..

2. Recursive Parameter Estimation

Models of physical, economical, social or biological systems can all be obtained in at least two different ways : by use of prior knowledge about the system or by experimentation and observation. Successful modelling is usually a combination of both. Often some parameters of the model have to be determined from experiments. Due to the development in computer technology and VLSI design the technics of parameter estimation can now be applied to new areas. If knowledge about the system is needed quickly the time for calculating good guesses of the parameters can be critical. One then has to settle with faster, less sofisticated algorithms. This is particularly the case for relay systems where decisions must be taken quickly. It is important that the algorithms be robust against disturbances. Reliability is especially important for relay systems that has to work, say once in ten years.

2.1 The ARMA Model

For a survey over the field of parameter estimation see [12]. We will here restrict us to one specific family of models often used for linear systems, the ARMAX-processes:

$$A(q)y(k) = B(q)u(k) + C(q)e(k) \quad k = ..-1,0,1,.. \qquad (2.1)$$

Here y is the output, u the input and e a disturbance often modelled to be white (often also Gaussian) noise. A, B, and C are polynomials in the forward-shift operator q, e.g. $qy(k) = y(k+1)$.

The coefficients of the polynomials A,B and C might be time-varying. The impedance of a power net-work will e.g. change drastically when short-circuit occurs. This is the idea behind the ASEA RELAYS interest in these problems. 'Fault detection' means to discover when these parameter changes occurs. For the moment we will however assume that A,B and C are time-independent.

The signals y and u are assumed to be known and the problem is to fit the polynomial-coefficients to experimental data as well as possible. To measure how well the model corresponds to data some criterion must be introduced. The resulting 'optimal' parameter estimates will be highly dependent on how this criteria is chosen. The least squares criterion

$$J(A,B,C,N) = \sum_{1}^{N} \hat{e}^2(k) \qquad (2.2)$$

is commonly used. Here $\hat{e}(k)$ is some kind of prediction error. The equation (2.1) can also be written more compactly as

$$y(k) = \Theta^T \varphi(k) + e(k) \qquad (2.3)$$

where

$$\Theta = (a_1 \ldots a_m, b_1 \ldots b_n, c_1 \ldots c_1)^T$$
$$\varphi = (-y(k-1) \ldots -y(k-m+1), u(k-1) \ldots u(k-n+1), e(k-1) \ldots e(k-j))^T$$

The signals e(k-1)...e(k-j) are not measurable. They are replaced by estimates calculated from prior measurements, see [1]. The parameter which minimizes the least square criterion is given by

$$\Theta = [\Phi^T \Phi]^{-1} \Phi^T Y \qquad (2.4)$$

where

$$\Phi = \begin{bmatrix} \varphi_1^T \\ \vdots \\ \varphi_N^T \end{bmatrix} \quad \text{and} \quad Y = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix}$$

It can be shown that if $C(q) = q^n$ and e(k) is white noise with zero mean and finite variance then the parameters Θ in (2.3) can be estimated unbiasedly by (2.4). With some restrictions on the input signal ("sufficient richness") the variance of the estimates will approach zero. This means that the parameters can be estimated arbitrary well if N is large enough. The 'convergence-rate' is determined through

$$\text{var } \hat{\Theta} = \sigma^2 (\Phi^T \Phi)^{-1}$$

If $C(q) <> q^n$ the estimates (2.4) will be biased using the least squares criterion. Unbiased estimates can be obtained if other estimation methods are used, see [1]. Neither of these methods are suitable for on-line computation because a large matrix has to be inverted every time new data is obtained.

criterion. Unbiased estimates can be obtained if other estimation methods are used, see [1]. Neither of these methods are suitable for on-line computation because a large matrix has to be inverted every time new data is obtained. This will make on-line computation practically impossible which is a major drawback if the results are needed quickly. The time for detection of a short circuit in a power system should e.g. be less than one period i.e. 20 ms. For a discussion of fault detection in power systems see [2].

The least squares algorithm (2.4) can be written as a recursive formula. This means that new guesses of the model parameters are obtained in terms of old guesses. This speeds up the updating. It allows higher sampling rates and uses less memory.

2.3 Recursive Computation

The recursive least squares algorithm can be written as

$$P(N+1) = [P(N)^{-1} + \varphi(N+1)\varphi(N+1)^T]^{-1} \qquad (2.5)$$

$$\hat{\Theta}(N+1) = \hat{\Theta}(N) + P(N+1)\varphi(N+1)[y(N+1) - \hat{\Theta}^T(N)\varphi(N+1)]$$

, see [10]. This can be interpreted as a steepest descent method with decreasing gain given by the Hessian matrix P. Similar algorithms can be found in optimization literature. The equations (2.5) can also be viewed as a Kalman filter for a system having constant parameters as state variables and white measurement noise e :

$$\begin{cases} \Theta(k+1) = \Theta(k) \\ y(k) = \Theta^T\varphi(k) + e(k) \end{cases}$$

The most time consuming part of (2.5) is the matrix inversion. By use of a matrix inversion lemma one can take advantage of the fact that $\varphi\varphi^T$ is a rank one matrix to rewrite the updating of the P matrix in (2.5). The new formula will require a computation time proportional to n^2 instead of n^3 where n is the number of parameters, see [10]. Even after this improvement the recursive formula (2.5) may be too demanding computationally. Different simplifications

have therefore been developed.

In stochastic approximation the matrix P is replaced by the scalar

$$P(k) = \left[\sum_{1}^{k} \varphi^T \varphi \right]^{-1}$$

The convergence of the estimates with this algorithm will generally be slower but with proper conditions on the input the estimates will still be unbiased. The properties of the parameter estimation depend of course on the model and on the nature of the disturbances. If there is no disturbances the estimates from (2.5) will converge to the correct values after a finite number of steps. For stochastic approximation we have instead the following sufficient conditions on P(k) for consistent estimates :

$$P(k) \geq 0$$
$$\Sigma\ P(k) = \infty$$
$$\Sigma\ P^2(k) < \infty$$

If these three conditions are fulfilled the estimates will eventually converge to the correct values. Nothing is however said about the convergence rate which usually is slower than with the Hessian matrix P. Kestin and Lapidus has suggested that P(k) only should be decreased if $\Delta\Theta_k$ changes sign. This, and similar tricks, can increase the convergence rate.

A third alternative is to have constant scalar gain P. This is called 'the MIT-rule' or just a 'constant gain algorithm'. The calculations can then be performed even faster,see [23]. A correct choice of P is important and may be troublesome. The choice is to be a trade-off between convergence rate and robustness against noise. A large P will give a fast but noise sensitive adaption. The choice of P is unfortunately also very sensitive to changes in the input signal. Too large a P will give an unstable algorithm as seen by the following calculation :

Introducing the prediction error $\tilde{\Theta} = \hat{\Theta} - \Theta$ equation (2.5) can be written as

$$\tilde{\Theta}_t = H_t \tilde{\Theta}_{t-1} + P\ \varphi_t e_t$$

where

$$H_t = I - P\ \varphi_t \varphi_t^T \qquad (2.6)$$

The matrix H_t has the eigenvalues 1, with multiplicity n-1, and $1-P\varphi_t^T\varphi_t$. The recursive equation (2.6) is thus unstable if P is negative or larger than

$$P_c = \frac{2}{\varphi_t^T\varphi_t}$$

If

$$P = \frac{1}{\varphi_t^T\varphi_t}$$

then H becomes a Householder projection-matrix. This will give the minimal prediction error. A proper choice of P therefore requires good knowledge about the magnitude of the input and output signals. This knowledge can be obtained e.g. through simulations of the system with 'typical' input signals.

With a constant P the variance of the estimates will not converge to zero. Still the method has the advantage of being very simple and can easily be implemented on a signal processor like the TMS 320. That the variance of the estimates not converges to zero should not worry to much. There is a general conflict between fast convergence and possibility to follow system variations. The constant gain algorithm has the advantage of being able to follow system variations fast.

Ex 2.1 Figure 2.1 shows some simulations of the constant gain algorithm with $y(k) = a\varphi(k) + e(k)$, a=1, e(k) is white noise with zero mean and variance $V(e) = 0.1$. $\varphi(k)$ is randomly generated with a Gaussian distribution and $V(\varphi) = 1$. P is a constant equal to (a) 0.1, (b) 0.5 and (c) 5.

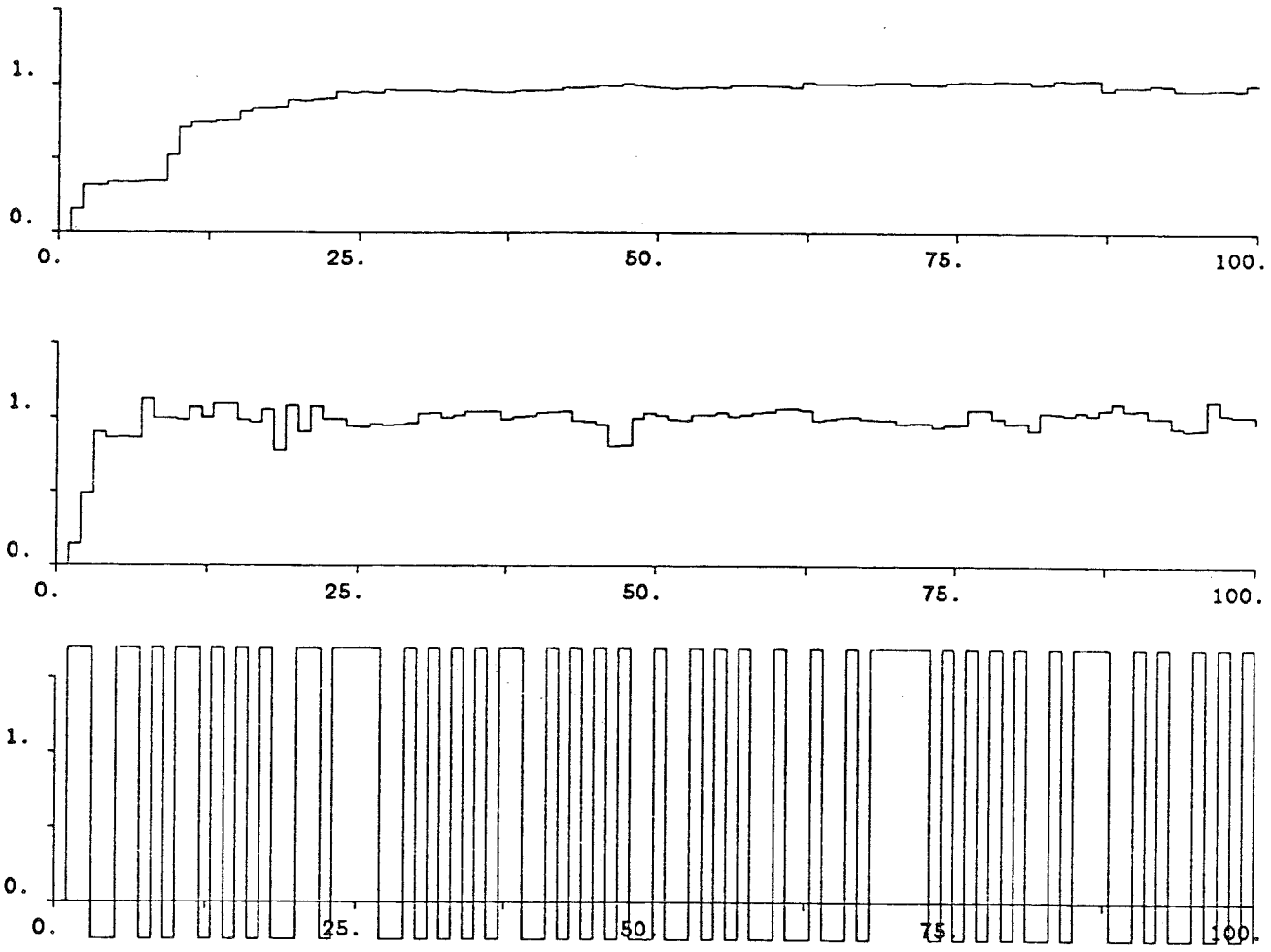


Fig 2.1

A Comparison between methods

Figure 2.2 shows how the parameter estimates typically change when using the different algorithms discussed up to this point.

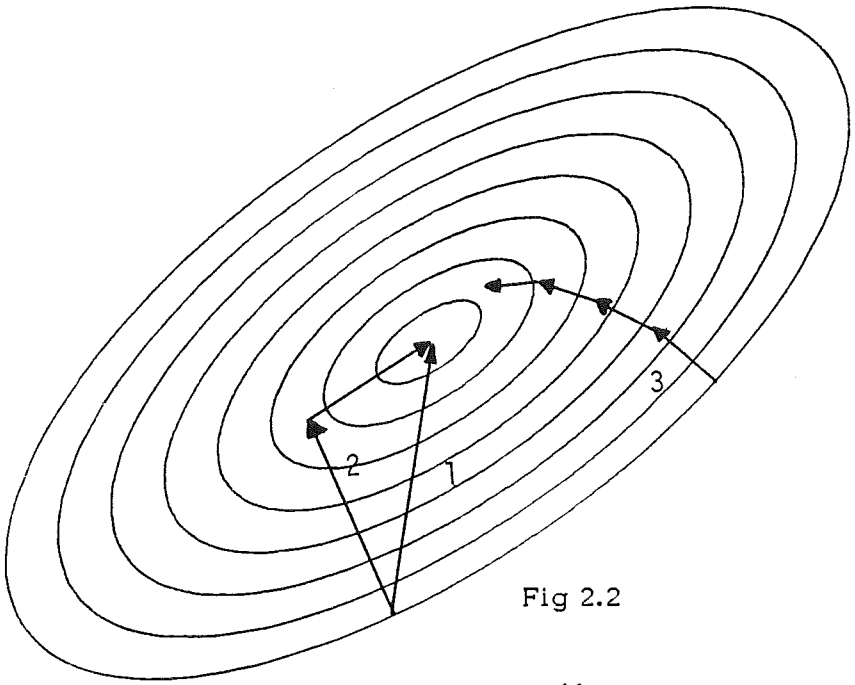


Fig 2.2

1) P = Hessian matrix, RLS algorithm.

2) P = Decreasing scalar gain, Stochastic approximation.

3) P = Constant scalar gain, MIT-rule.

Comparing the calculation time needed in 1,2 and 3 one sees that the following is needed

1) Inversion of n*n matrix (n= number of estimates).

2) Inversion of scalar, otherwise like 3.

3) Scalarproduct of n-vectors.

Comparing the convergence rate one sees that in the case of no noise the RLS method converges to the correct value in a finite number of steps and that algorithms with a scalar P converge exponentially instead (provided that the input u is 'sufficiently rich'). Generally the stochastic approximation algorithm gives slower convergence and using the MIT-rule the variance of the estimates will not converge to zero at all. This can be seen from the following example :

Ex 2.2 $y(k+1) = ay(k) + bu(k) + e(k)$

$V(e) = 0.1$

$a = 0.8$

$b = 1$

u = Random binary sequence with amplitude 1

1) RLS with $\Theta(0) = 0$ and $P(0) = 100\ I$

2) Stoch. approximation $P(k) = 1/\Sigma\varphi^T\varphi$

3) Constant gain $P = 0.05$

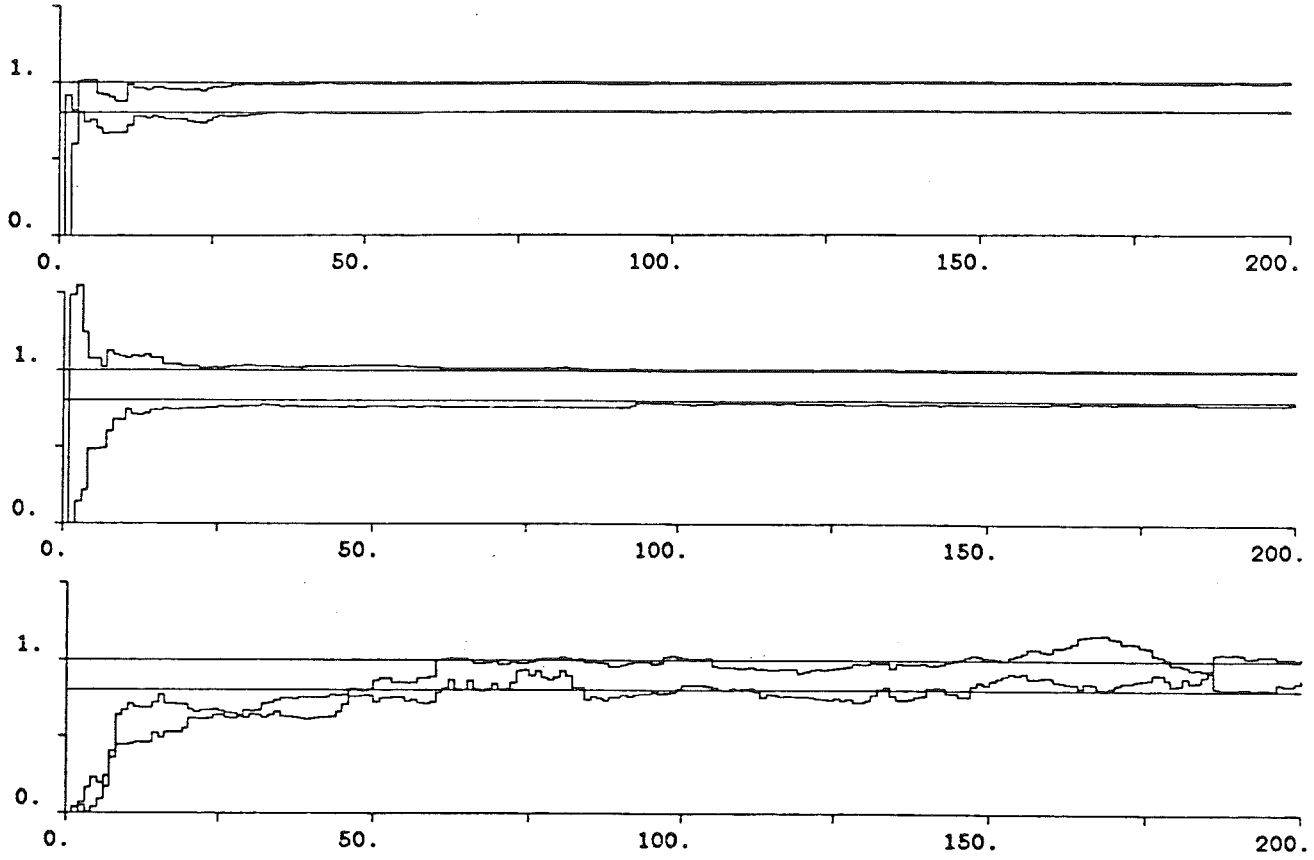


Fig 2.3

Numerical Problems, the UDU^T-Algorithm

The recursive least squares algorithm described above is poorly conditioned numerically because it requires inversion of the matrix $\Phi^T\Phi$ in (2.4) whose elements are products of measured signals. Algorithms which avoids squaring the signals can be obtained by writing P as U D U^T and derive recursive equations for U and D. These class of algorithms are called square root algorithms. They are discussed in debth in Biermann [3]. These algorithms are all rather computationally complicated.

To estimate the parameters unbiasedly and consistently from input - output data the input signal has to fulfil certain requirements. The more parameters we want to estimate the 'richer' the input signal has to be. The input signal should excitate all modes. Another way to say the same is that to find n unknowned one has to have n equations. There are a number of criterions on when it is possible to estimate n parameters consistently. The most useful is perhaps :

Theorem

Sufficient conditions for estimating n parameters consistently in an ARMAX-model is that

(1) $\bar{u} = \lim_{N\to\infty} \frac{1}{N} \sum_{1}^{N} u(k)$ exists.

(2) $R(i) = \lim_{N\to\infty} \frac{1}{N} \sum_{1}^{N} \{u(k+i)-\bar{u}\}\{u(k)-\bar{u}\}$ exists for all i.

(3) The Toeplitz matrix $A_n = \{a_{ij} = R(i-j)\}$ $i=1,\ldots,n$

is positive definite. (2.7)

This implies consistent estimates for the least square algorithm, the maximum likelihood algorithm and the maximum likelihood method with white measurement noise, see [4]. Observe that since a matrix is positive definite if and only if its leading principal minors are positive, an input signal that do not satisfy the conditions for an order n do not satisfy them for any greater order.

For instance if the input is a pure sinusoidal we can see that the conditions are satisfied for order two (unless there is some aliasing effect) but not for order three or greater :

A pure sinusoidal is not persistently excitating of order three. The connection between the input and output is given by the transfer function at the frequency of the sinusoidal input. This transfer function is a complex number and therefore totally determined by two parameters. Yet another way to say this is that the sinusoidal output is determined by its amplitude and phase. In the same way if the input signal is a sum of n different sinusoidals at most 2n parameters can be estimated consistently.

A frequency domain variant of conditions for sufficient richness in the input signal can be found in [4].

2.5 Time Varying Systems

Up to this point only time invariant models have been discussed. That is, the parameters to be determined are constant but unknown. A little different problem arises when it is known that the system parameters can change and the goal is to detect or adapt to such changes. These changes are often called 'faults', although these 'faults' does not necessarily correspond to malfunctions of the system. When having time-varying systems the least squares algorithm above is not satisfactory since the decreasing gain will make parameter-changes hard to detect. The influence of old data therefore has to be decreased. This can be done by introducing so called forgetting-factors. The loss function can e.g be changed to give exponentially decreasing weights to old data :

$$J(\Theta) = \sum_{1}^{N} \lambda^{N-k} \hat{e}^2(k)$$

The stochastic approximation algorithm will then be

$$P(N+1) = [\, \lambda\, P(N)^{-1} + \varphi^T(N+1)\varphi(N+1)\,]^{-1} \qquad (2.8)$$

$$\hat{\Theta}(N+1) = \hat{\Theta}(N) + P(N+1)\varphi(N+1)[y_{N+1} - \hat{\Theta}^T(N)\varphi(N+1)]$$

The scalar P will now not approach zero. The forgetting factor, λ,will make it possible to adapt to system variations. The choice of λ will be a trade-off between fast adaption and robustness against noise, as can be seen from the following example.

The stochastic approximation algorithm will then be

$$P(N+1) = [\lambda P(N)^{-1} + \varphi^T(N+1)\varphi(N+1)]^{-1} \qquad (2.8)$$

$$\hat{\Theta}(N+1) = \hat{\Theta}(N) + P(N+1)\varphi(N+1)[y_{N+1} - \hat{\Theta}^T(N)\varphi(N+1)]$$

The scalar P will now not approach zero. The forgetting factor, λ,will make it possible to adapt to system variations. The choice of λ will be a trade-off between fast adaption and robustness against noise, as can be seen from the following example.

Ex 2.3 $y(k) = au(k) + e(k)$

$\lambda = 1, 0.99, 0.9$

$a = 0$ ($k<250$) and 1 ($k \geq 250$)

$V(e) = 0.5$

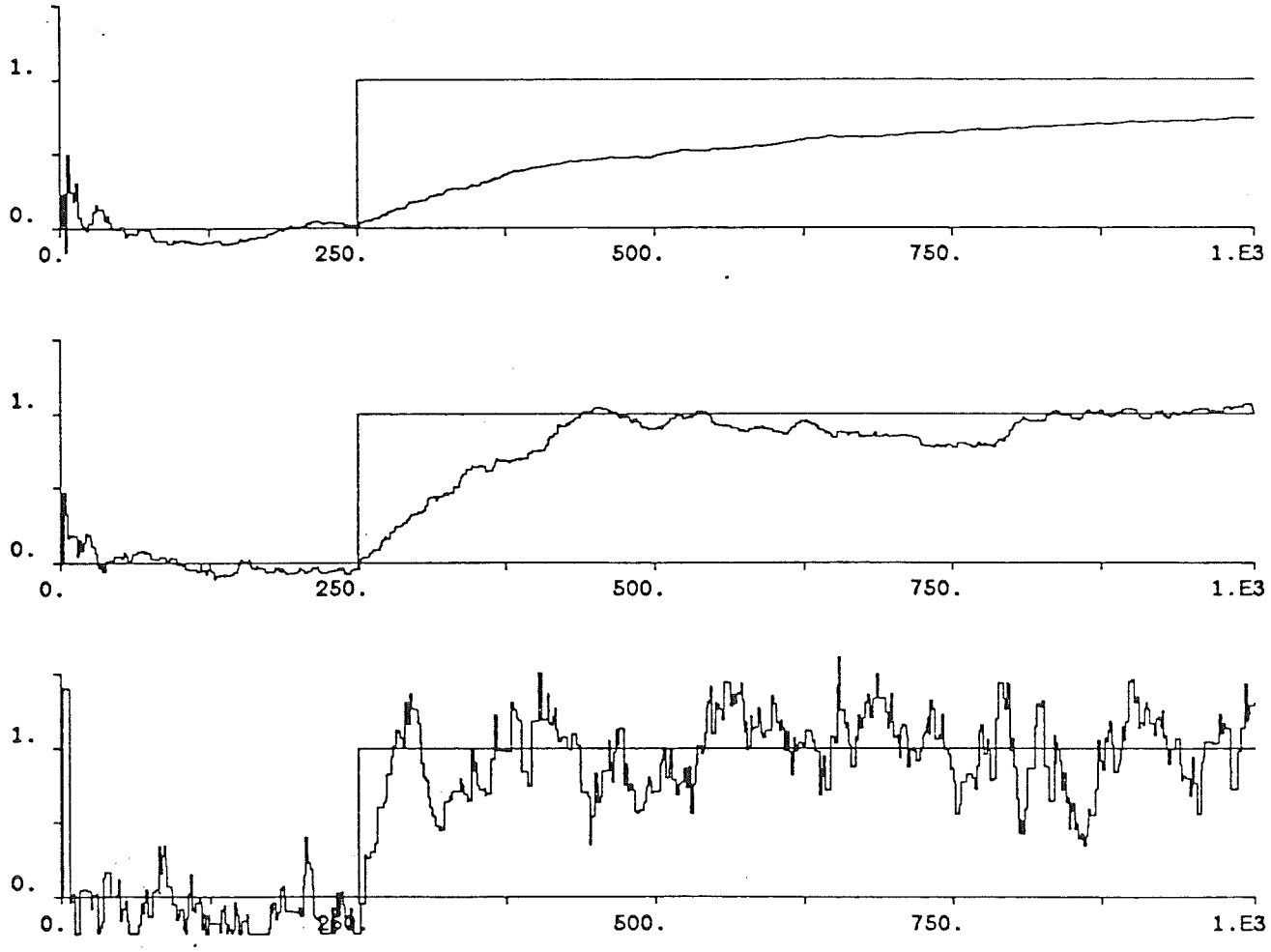


Fig 2.4

Detecting Sudden Parameter Changes

Different methods have been suggested for finding large changes in the parameters. For a summary see [5]. Most methods require good knowledge about the disturbances which are often modelled having Gaussian distributions. This can partly be motivated by the central limit theorem, however if this fails to be true the properties of the algorithms might change drastically. It is probably not true that the disturbances occuring in power systems have Gaussian distribution.

The most obvious method for fault detection is perhaps to look at the magnitude of the estimation errors $\hat{e}(k)$. This has also been suggested in the literature. Large estimation errors suggest that a change in the parameters of the system has occured. Different schemes for implementing the above idea has been proposed, see e.g. [5].

Drawbacks of the above suggested methods are easily found. Only faults that have a large influence on the output signal can be detected. Better fault detection methods can be obtained by looking at the parameter estimates $\hat{\Theta}$ directly and try to detect a change in these instead. One way of doing this is discussed in [5]. Introduce the difference between two consecutive estimates

$$\Delta\hat{\Theta}_t = \hat{\Theta}_t - \hat{\Theta}_{t-1}$$

If $\Delta\hat{\Theta}$ is large this suggests a change in the parameters. A way of fault detection would therefore be to look at the norm of $\Delta\hat{\Theta}_t$. If the norm is larger than some threshold it is concluded that a fault has occurred. The threshold must be a compromise between low false alarm frequency and high sensitivity for faults. The test can be improved if one looks at more than one consecutive samples, e.g. by setting $W_{t+1} = \lambda W_t + \Delta\hat{\Theta}_t$ where $0 \leq \lambda < 1$

Another possible fault detection method uses the direction of the parameter variations. In case of constant parameters the variation of the estimates has no trend and therefore

$$P\ (\Delta\hat{\Theta}_t^T W_t > 0) = P\ (\Delta\hat{\Theta}_t^T W_t < 0) = \frac{1}{2}$$

In case of a fault this will however no longer be true. The trend of the parameter estimates will then give.

$$P\ (\Delta\hat{\Theta}_t^T W_t > 0) > P\ (\Delta\hat{\Theta}_t^T W_t < 0)$$

This method has been suggested by Hägglund in [5]. It is a development of similar arguments used on the output estimates. For a discussion of this see [6].

Example 4 shows how the method of looking on the magnitude of the estimation error can be used to detect a change in the gain of a very simple system. For other examples, see [5].

Ex. 2.4 y(k) = au(k) + e(k)

a = 0 if k≤250 else 1

u = random signal with V(u)=1

V(e) = 0.1

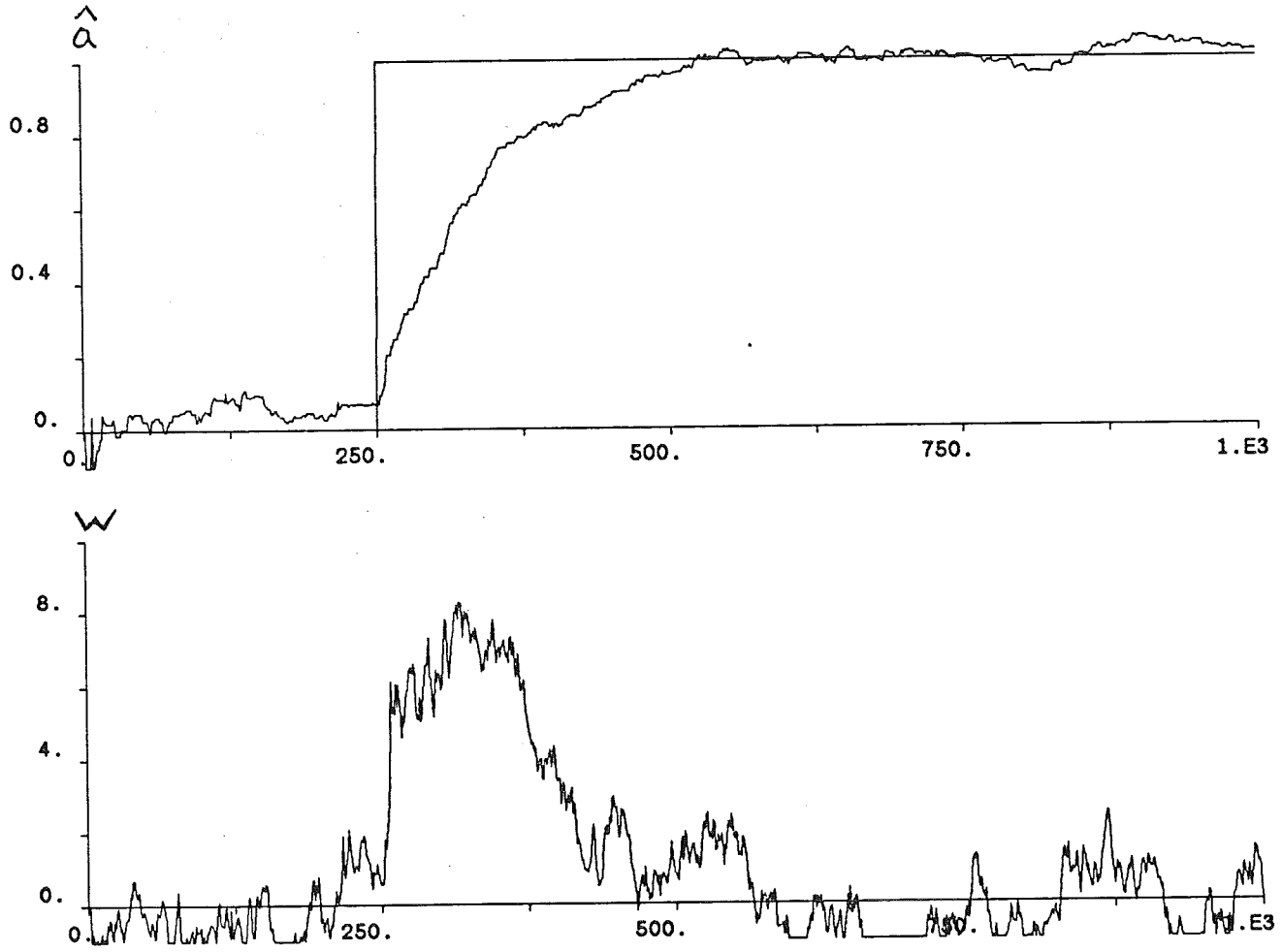


Fig 2.5

The success of all fault-detection algorithms described above will be rather dependent on the nature of the disturbances. In the theory these are often modelled to be Gaussian. If this fails to be true the fault detection might detoriate. The conclusion is that the applicability of the chosen method has to be carefully checked in practice on real input output data.

3. The TMS 32010 Processor with Development Boards

3.1 Introduction

The digital signal processor TMS 32010 is a member of Texas Instruments new digital signal processing family, designed especially for high-speed numeric-intensive applications. It is a 16/32-bit single-chip microcomputer with an on-chip array multiplier which offers an inexpensive alternative to multichip bit-slice processors.

The TMS320 is capable of performing 5 million instructions per second. This is due to the highly pipelined Harvard architecture and the reduction of the instruction set to incorporate only those operations normally needed for digital signal processing algorithms.

Table 3.1 lists some 'typical applications' of the TMS 320 as presented by TEXAS INSTRUMENT. It should be noted that those are possible applications, all have probably not been tried out in reality.

SIGNAL PROCESSING	TELECOMMUNICATIONS	IMAGE PROCESSING
• Digital filtering • Correlation • Hilbert transforms • Windowing • Fast Fourier transforms • Adaptive filtering • Waveform generation • Speech processing • Radar and sonar processing • Electronic counter measures • Seismic processing	• Adaptive equalizers • μ-A law conversion • Time generators • High speed modems • Multiple-bit-rate modems • Amplitude, frequency, and phase modulation/demodulation • Data encryption • Data scrambling • Digital filtering • Data compression • Spread-spectrum communications	• Pattern recognition • Image enhancement • Image compression • Homomorphic processing • Radar and sonar processing **HIGH-SPEED CONTROL** • Servo links • Position and rate control • Motor control • Missile guidance • Remote feedback control • Robotics
INSTRUMENTATION	**NUMERIC PROCESSING**	**SPEECH PROCESSING**
• Spectrum analysis • Digital filtering • Phase-locked loops • Averaging • Arbitrary waveform generation • Transient analysis	• Fast multiply/divide • Double-precision operations • Fast scaling • Non-linear function computation (i.e., sin x, e^x)	• Speech analysis • Speech synthesis • Speech recognition • Voice store and forward • Vocoders • Speaker authentification

Table 3.1

The key features of the TMS 320, as described by Texas Instrument, are collected in table 3.2.

- 200-ns instruction cycle
- 288-byte on-chip data RAM
- ROMless version – TMS32010
- 3K-byte on-chip program ROM – TMS320M10
- External program memory expansion to a total of 8K bytes at full speed
- 16-bit instruction/data word
- 32-bit ALU/accumulator
- 16 x 16-bit multiply in 200 ns
- 0 to 15-bit barrel shifter
- Eight input and eight output channels
- 16-bit bidirectional data bus with 40-megabits-per-second transfer rate
- Interrupt with full context save
- Signed two's complement fixed-point arithmetic
- 2.7-micron NMOS technology
- Single 5-V supply
- 40-pin DIP

Table 3.2

The processor costs 50 $ and the delivery time is said to be about 8 weeks (Texas Instrument, Stockholm). This has been verified (for the processor and evaluation board) by the author. The analog interface however took about 4 months to get. The evaluation module costs 995 $ and the analog interface board 850 $ (spring 1985).

The easiest way to become familiar to the TMS is probably to read [11], [19], [20] and [21]. These manuals are available from TEXAS INSTRUMENT. Another good introduction is presented in the collected papers [29] which

also contains a summary of digital signal processing techniques. It is advisable to do small experimental programs in parallel to reading and use the simulator to test and verify these. The architecture is described in section 3.3. The system development tools including a simulator on VAX, an emulator board and a analog interface board is described in section 3.4.

3.2 Why the TMS 320 Was Chosen, Some Alternatives

When this work was started the choice of implementation stood between using an array processor-floating point processor attached to an IBM personal computer and using a DSP-chip such as the TMS 320. The market for array processors was investigated by reading articles and advertisements. The best alternative found was a product available from SYSTOLIC SYSTEMS called the PC-100 and PC-1000 DESKTOP ARRAY PROCESSOR.

The PC-100 is a array processor that can increase the computational speed of the IBM PC over 100 times or a factor 3-10 if compared to a PC with 8087 coprocessor. The PC-100 features IEEE standard arithmetic, a 32-bit floating point processor, parallel I/O on the IBM PC bus, an extensive mathematical software library and a high-resolution graphics package.

The PC-100 is programmed in ANSI 77 FORTRAN via subroutine calls to the PC-100's math library. The math library consists of commonly used matrix/vector routines, linear equation solvers, numeric integration methods, simulation tools, optimization algorithms and least square curve fitting. Fortran callable routines are provided to evaluate trigonometric and transcendental functions.

A signal processing library is also available with the PC-100. It contains Fortran callable routines for fast Fourier transforms, digital filtering, correlation, convolution and spectrum analysis. The PC-100 costs 2995 $ (October 1984) plus about 2000 $ for software library routines for signal processing, matrix operations and high-resolution graphics.

A short summary of the math library is given in below.

LINEAR MATRIX/VECTOR OPERATIONS

- Matrix/Vector Clear
- Matrix/Vector Addition
- Matrix/Vector Subtraction
- Matrix/Vector Multiplication
- Matrix Inversion (Vector Division)
- Solution of Linear Equations
- Solution of Triangular Equations
- Least Square Curve Fitting

NONLINEAR MATRIX/VECTOR OPERATIONS

- Matrix/Vector Absolute Value
- Matrix/Vector Squared
- Matrix/Vector Square Root
- Matrix/Vector Logorithm
- Matrix/Vector Exponential
- Matrix/Vector Sine/Cosine
- Random Number Generator

SIGNAL PROCESSING LIBRARY

- Fast Fourier Transforms (FFTs)
- Digital Filtering
- Window Functions
- Auto/Cross Spectrum
- Convolution/Correlation
- Coherence Function

SIMULATION/INTEGRATION LIBRARY

- Trapezoidal Rule Integration
- Runge-Kutta Integration
- Predictor-Corrector Integration

UNCONSTRAINED OPTIMIZATION

- Steepest Descent Method
- Davidon-Fletcher-Powell (DFP) Method
- Broyden-Fletcher-Shanno (BFS) Method

PC-100 HARDWARE SPECIFICATIONS

- Wordlength (24-bit mantisa, 8-bit exponent) ..... 32 bits
- Multiply Time (32-bit floating point) ..... 500 nsec
- Addition Time (32-bit floating point) ..... 500 nsec
- Computational Speed ..... 1 MFLOP

Table 3.3 contains some benchmarks of the PC-100 as presented by SYSTOLIC SYSTEMS.

Calculation	IBM PC	IBM PC + 8087	IBM PC + PC 100
100*100 Matrix Multiply	38 min	9 min	21 sec
100*100 Matrix Addition	12 sec	4 sec	0.31 sec
1024 point Real FFT	4.1 sec	0.9 sec	0.06 sec

Table 3.3

The PC-1000

This is how the PC-1000 is described by the manufacturer, Systolic Systems :

> " The PC-1000 desktop array processor is the new industry standard for real-time data acquisition, estimation and control because now you can do it all for less than $20,000. Applications like classical control, digital filtering and adaptive control can all be solved in real-time on the IBM PC and PC 1000. Control system designs based on frequency domain and multivariable control methods can all be implemented on the PC 1000 in a matter of minutes to improve engineering productivity. Decision analysis and graphics are performed on the IBM PC based on data received from the PC 1000. The PC 1000 can also be used in the laboratory for evaluating several engineering designs in the same day. "

Application software to the PC-1000 can be written in high-level language such as FORTRAN or ADA on the IBM PC. The PC 1000 consists of a master processor, slave processor and data acquisition system. The is designed with a iAPX 8086 CPU, iAPX 8087 math coprocessor and 256K memory. The master processor performs numerical computations, coordinates each slave processor and communicates with the IBM PC. Two 32-bit slave processors can be installed in the PC 1000 to evaluate control functions at measurable throughput rates of up to 9.2 MFLOPs.

The data acquisition system provides 32 analog I/O channels with a maximum sampling frequency of (only) 2000 Hz.

The speedup of the PC-1000 system is 1000 times compared to the IBM PC (without coprocessor) when performing multiplication and addition.

The PC-1000 costs 20,000 $ (Sept. 1984). It was therefore not consider any further in this project.

The TMS 32020 Processor

The TMS 32020 has been introduced during the spring of 1985. Its main features are collected in fig 3.4. There will also be development boards available for the TMS 32020. The system cost will however be so much greater so that the TMS 32020 can not be regarded as an alternative to the TMS 32010.

* 170 ns instruction cycle time
* 544 words of on-chip RAM (288 words are always data memory, 256 words are programmable as either data or program memory)
* 128K words of total program/data space
* 16-bit data word with internal 32-bit operations
* 16-bit instruction word
* Block moves for efficient data/code management
* 32-bit ALU and accumulator
* Single-cycle 16x16-bit multiply / 32-bit accumulate instruction
* Floating point operation support through instruction set
* 0-16 bit input scaling shifter
* Fractional/integer arithmetic support
* Variety of bit-manipulation and logical instructions
* 5 auxiliary registers (ARs) for indirect addressing and temporary storage
* A register arithmetic unit (RAU) dedicated to the AR file for arithmetic operations on AR and for auto-indexing by 16-bit numbers
* 16-bit parallel multiprocessor interface
* A hardware global memory interface
* Serial port for multiprocessing and interfacing to codecs and serial A/D converters
* On-chip timer for control operations
* 3 external maskable user interrupts
* 1 non-maskable interrupt for test and emulation

On-chip clock generator

Fig 3.4

A very important factor when choosing the equipment to use is to check that the processor and development tools will be available with not too long delivery time. This can not be enough stressed.

Conclusions

The PC-100 (PC-1000) system needs an IBM PC. It is a suitable system for doing extensive simulations of different estimation algorithms off-line with simulated input data. As an component of a small intelligent relay system it is both too expensive with a system cost of 5-10,000 $ and too slow for on-line calculations. Another disadvantage was that those systems were, when this project was started in November 1985, rather new and untested.

Instead it was decided to look into signal processors. Those have until recently only been used for a restricted type of applications like digital filters, FFT calculations or voice recognition. They are now being used in new areas where computational speed is of importance.

This inheritance of technique has had some drawbacks. The algorithms for which signal processors normally have been used can be characterized by very simple calculations being performed very fast. The signal processors are therefore programmed in assembler and use fixpoint arithmetics. This is the best trade off between complexity and speed in traditional applications. In control applications a slower sampling rate is used and often much can be gained by using a more advanced, and therefore more computationally complicated, algorithm.

It is therefore desirable to program in a higher, signal-processing oriented, language and to use floating point arithmetics.

The question arising was therefore: Is it possible to implement e.g. a parameter estimation algorithm on todays signal processors? And if not, what features are lacking? Which components should a signal processor contain if it is going to be used for e.g. adaptive control algorithms? How should it be programmed?

Future Trends for Signal Processors

(The comments inside paranthesis stands for the TMS 320)

- Faster clock cycle. (170 ns instead of 200 ns)

- More RAM on-chip. (544 * 16 bit words)

- More address space for program,external data and I/O. (64 K + 64 K)

- Floating point support. (Instructions for normalize and computed shifts)

- Multi address machines.

- More addressing modes. (Register ALU for e.g. auto-indexing)

- More functionality: timers,communication ports. (On chip timer and serial port).

- Loop and repeat hardware.(repeat counter)

- Better ALU.(logical operations on 32 bits etc)

- More interrupt support, context switch support. (3 external maskable)

- Lowering of external hardware speed requirements. (Wait states for slow memorys)

- Multi processor interfaces.

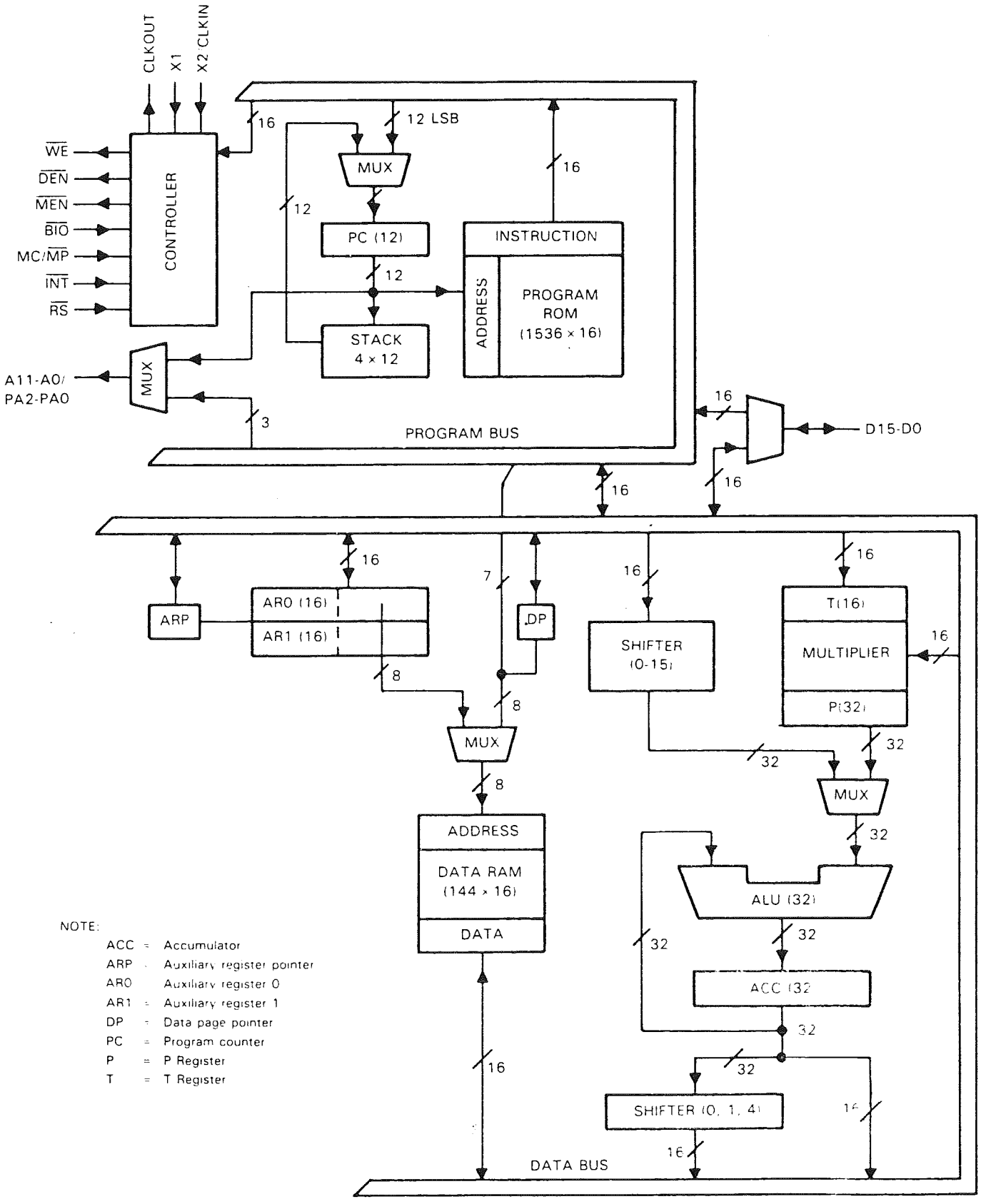
- Higher level programming languages.

- Higher degree of complexity. (More instructions)

In November 1985 the TMS 320 was believed to be the most promising signal processor. This opinion has just been strengthen during the project. There have appeared a lot of articles describing the use of TMS 320 in different applications, see [34], [35], [36], [37] and [38]. The TMS 320 has almost got a status of 'industrial standard' in the United States. Much of this is due to its simple architecture.

The Harvard Architecture

A block diagram over the TMS 320M10 is shown in figure 3.5.



NOTE:

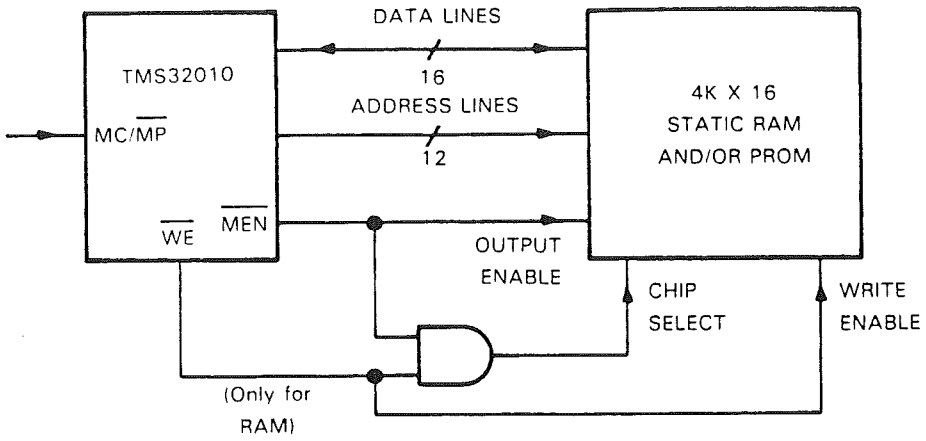
ACC	=	Accumulator
ARP	=	Auxiliary register pointer
AR0	=	Auxiliary register 0
AR1	=	Auxiliary register 1
DP	=	Data page pointer
PC	=	Program counter
P	=	P Register
T	=	T Register

The TMS 320 has a modified Harvard architecture. In a Harvard Architecture the data (RAM) and program memory (RAM or ROM) are separated to make concurrency between instruction fetch and instruction execution possible. When the processor fetches an instruction from program memory, it can fetch the data that it needs from data memory at the same time since each bus operates independently. This of course increases the speed.

The TMS modification allows for transfering of data between program and data memory. An almost necessary modification since the data memory is only 144 double-bytes long. As an example data tables longer than 144 words can reside within program space so that the designer can make tradeoffs between the amount of table space and program space needed for a specific application. This can be used e.g. when storing the filter coefficients for a digital filter. It takes three instruction cycles to execute each of these transfers of 16 bit data. Future TMS chip will probably have a larger amount of on-chip data-RAM as finer (e.g. 1-micron) technology becomes available.

With some extra hardware, the IN and OUT instructions can be used to read and write from an external RAM storage addressed as a peripherel. The price will of course be larger execution times since 'in' and 'out' takes at least two cycles to execute, more if slower memory have to be used. Except for the transfer-instructions, TBLR and TBLW, all instructions operate on the data memory.

Program memory can lie both on-chip (in a 1536 * 16-bit ROM) and off-chip. The TMS 32010 processor has no program ROM on chip. The ROM version, called TMS 32010M is used for large series (say more than 1000 ex) signal processors for the same application Since the program counter is 12 bits wide the maximum amount of program memory that can be addressed is 4096 * 16-bit words. Fast memories with access times of under 100 ns are required if instructions in off-chip memories should be executed at full speed. Figure 3.6 shows how off-chip memory is addressed.



EXTERNAL PROGRAM MEMORY EXPANSION EXAMPLE

Fig 3.6

The data memory consists of 144 * 16 bit RAM divided into two pages. Page 0 contains 128 double-bytes, page 1 contains 16 double-bytes. The reason for this unusual arrangement is probably that there was some space over on the chip after the preliminary layout. This is modified in the new TMS 32020 processor

Multiplier, ALU, Accumulator and Shifters

All arithmetic operations are performed in fixed-point two's complement arithmetic. Different kinds of arithmetic formats are discussed in the next section.

The 16 * 16-bit multiplier consists of : the T register (16 bit) which holds one of the factors, the P register (32 bit) that stores the product and the multiplier.

In order to use the multiplier the T-register must first be loaded with one of the operands. Then a MPY or a MPYK instruction is executed. The second factor is taken either from data memory (direct or indirect addressing) or defined by the instruction (immediate addressing). The product is in the third instruction cycle either added to or subtracted from the accumulator where from it can be loaded to data memory.

In this way a single multiplication takes 4 cycles = 800 ns. However when

several consecutive multiplication-add operations are to performed, e.g. when calculating scalar products, the operations can be pipelined in such a way that the overall multiplication-add time will be 400 ns. The architecture could be said to be optimized for calculations of scalar products which breaks down to the calculation of

```
ACC := ACC + X1 * X2
```

In TMS instructions this will become

```
LT   X1
MPY  X2
APAC
```

There is also a multiply immediate instruction which multiplies with 13-bit constants. For example :

```
LT   X1
MPYK 5192
APAC
```

will accomplish the multiplication

```
ACC := ACC + X1 * 5192
```

The pipelining of multiplications in scalar products works as follows:

```
ZAC          CLEAR ACCUMULATOR
LT   X1
MPY  Y1      FIRST MULTIPLICATION
LTA  X2
MPY  Y2      SECOND - " -
LTA  X3
MPY  Y3      THIRD - " -
.
.
.
APAC         STORE THE RESULT
```

The LTA instruction will load the T register and add the product register to the accumulator. In this way a scalar product of two vectors of length N can be accomplished in 2(N+1) instructions, (each taking 200 ns).

The ALU is 32 bit arithmetic logical unit which can do the operations add, subtract, abs, and, or and xor. The first operand is always the accumulator the second operand is fetched from data memory (16 bits possibly left shifted 0-15 bit, the rest are zerofilled) or from the product register.

Note that the logical operations and or and xor only operates on the rightmost 16 bits of the accumulator. This is of course an annoying drawback when operations on the full 32 bits are wanted. The only possible solution is then to do these operations in several steps and store intermediate results in data memory. This will be rather time consuming. The TMS 32020 has a more flexible ALU

The ALU can be put in overflow mode under program control. If an overflow occurs when set in this mode, the most positive or negative representable value of the ALU will be loaded to the accumulator. This models the saturation processes inherent in analog systems and increases the chance of getting useful results even if an overflow have occured. Note though that the multiplication of -1 * -1 will yield the answer -1 even in overflow mode ! The overflow flag is not affected. This is a documented error which is corrected in the new TMS 32020 processor. Better would in this case be to load the accumulator with the largest representable positive number and set the overflow flag.

The accumulator has a 32 bit word length. Instructions exist for storing the high or low-order bits in data memory with shift (SACH and SACL). However only 0,1 or 4 bit left shift have been implemented and this only on the high order part. SACL has no shift facility at all. This is changed in the new processor to enhance the flexibility. The shifting facility is needed in the scaling operations.

There are two shifters available. Both are 16 bit and can only do left shifts. A right shift by n can however be implemented by loading the accumulator with the operand shifted 32-n bits and storing the high order part. This takes 2 clock cycles.

Registers

These consist of two 16 bit auxiliary registers (AR0, AR1) one auxiliary register pointer (ARP) and the data memory pointer (DP).

The auxiliary registers are used for indirect addressing in data memory and loop control. Indirect addressing uses the least significant bits of the auxiliary registers. There is only one operations that can be used for loop control, BANZ, branch on accumulator not zero.

Stack

The stack is four levels deep and 12 bits wide. It can be used to store return addresses when using nested subroutines.

Status Register

The status register consist of five status bits containing information of the processor state.

OV - overflow has occurred

OVM - overflow mode is on

INTM - interrupt is enabled

ARP - current auxiliary pointer is AR1

DP - current data page is 1

The status register can be saved and reloaded by the SST and LST instructions.

Input/output

There are 8 parallel input ports and 8 parallel output ports on the TMS 32010 all 16 bits wide. Data is transfered to and from the data memory by the strobes DEN (data enable) and WE (write enable).

Technology

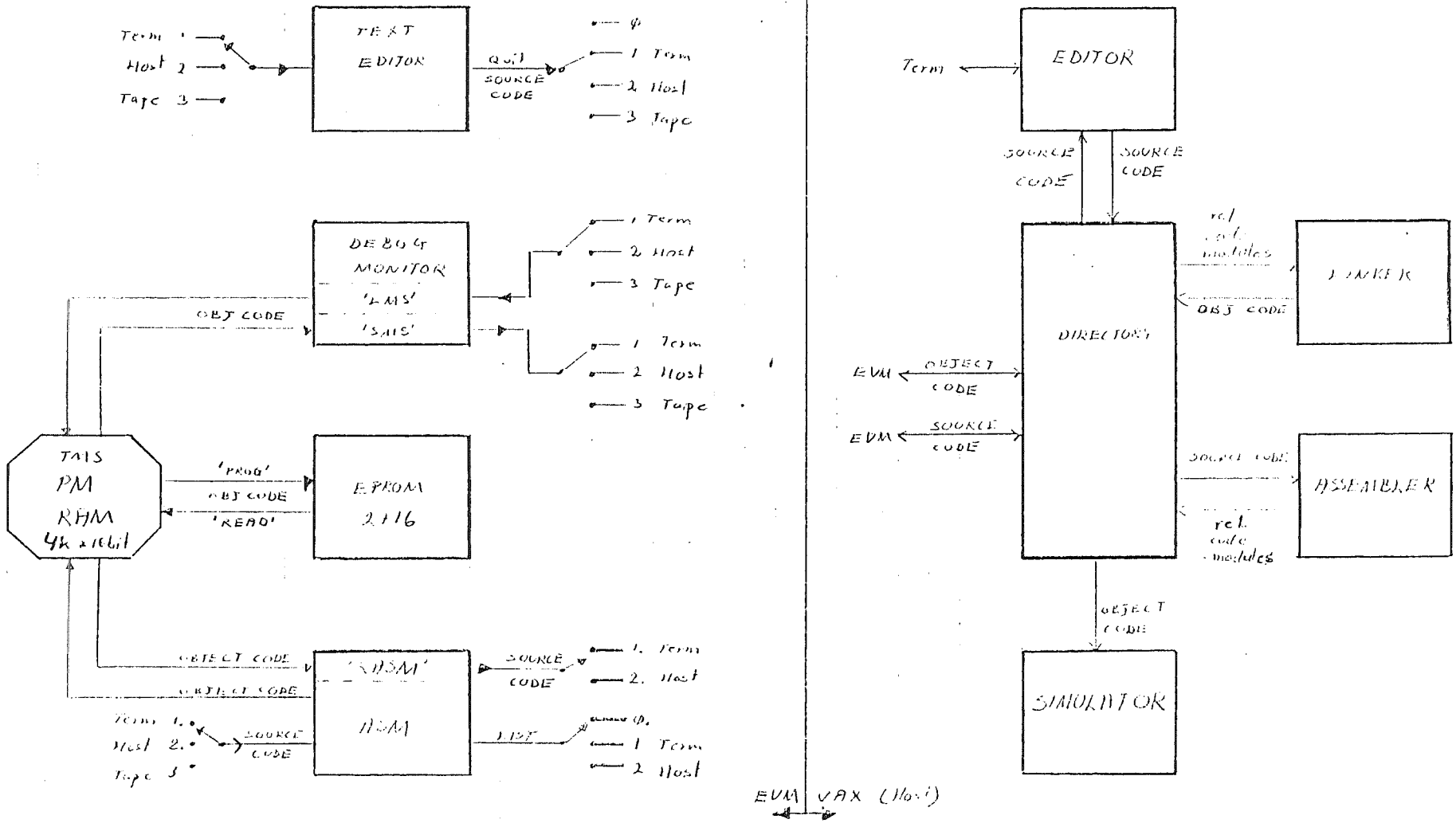
The TMS 32010 is made in a VLSI 2.7-micron NMOS technology. The next generation will probably be made in even finer technology making a larger number of on-chip registers and data memory possible.

To help the engineer in the design of the hardware and software TI have some development facilities :

- An assembler
- A linker
- A simulator.
- An evaluation module (EVM) with emulation support.
- An analog interface board (AIB) that is used with the EVM.

The assembler, linker and simulator reside on a host computer, VAX 11/780 or IBM PC. There is also an assembler on the EVM-board.

See also figure 3.7.



TEXHS TMS 320; EVM-bit & VAX soft support : TMS program code transfer

850201
B Bengtsson

The design process

The design of a complete system could go as follows,see figure 3.8

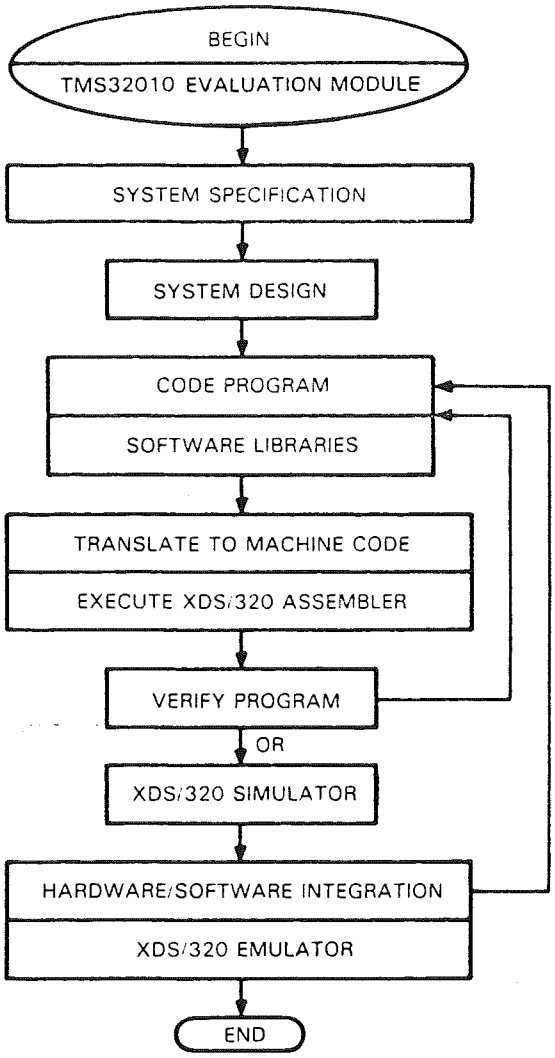


Fig 3.8

1) System specification/design.

The function of the system must be exactly defined and generally how the system will be implemented. It is probably advisable to do extensive simulations of the system with different solutions if the problem is complicated. In this stage a preliminary algorithm flow chart and hardware block scheme should be drawn. Based on the requirements and functionality an implementation approach is chosen. This will include the selection of processor and other hardware together with how software will be developed (e.g. programming language). In the design of signal processing systems the designer is unfortunately forced to

write in assembly language since there is no alternative available today. The use of an code generator, described in section 5, can however fasten up the process.

2) Hardware/software design.

Here the design engineer choose an implementation of the parts left out under point 1. A detailed schematics over the hardware together with a high level pseudo language description of the software is made.

3) Software coding/Hardware fabrication.

Due to the lack of suitable languages with compilers for signal processing one is often forced to produce the assembly code for the processor by one self. The code can be generated either 'by hand' which is the most used way, or by a code generator. (If a high-level language compiler is available the code generator will be part of the compiler.) The advantages of this and an example of a simple code generator is described in section 5. The output in this stage will be assembly language which is then inputted to an assembler which produces machine code. This has been done with the problem of fast recursive parameter estimation as an example. This is also found in section 5.

4) Software/Hardware test.

Before the final implementation of the system it is necessary to test the software in the real environment. The design up to this point has only been based on a, more or less accurate, model of the real world and it must be checked that not too much information has been lost in this process. This can be a time consuming part of the design process. A normally good tip is that the software and hardware first should be tested out separately if possible. For the TMS 320 there is a simulator available which make it possible to test the assembly program extensively before going to point 5.

5) Final implementation.

In the last step the hardware and software that now has been tested and verified are integrated into the application environment.

The following is a demonstration of how the software programs is run on a VAX 11/780 at the Department of Automatic Control, Lunds Institute of Technology.

The Assembler

For an extensive description see [20]. The files concerning the assemblation are collected in

```
[BOB.IMPORT.ASM3]
```

They are

```
README.DAT  - Installation and verification guide
XASM.COM    - Executes the assembler
LINKASM.COM- Re-links the assembler
PARSE.C25   - Parses pathnames for VAX 2.5 O.S.
PARSE.COM   - Used by the above procedures to parse
              pathnames for 3.0 O.S.
ASM320.EXE  - Executable code for the assembler.
ASM.OBJ     - Assembler objet code, used in re-linking
RUNTIME     - Source/ object library
TEST.ASM    - Source for the assembler test program
TEST.LIS    - Correct listing for the test program
TEST.MPO    - Correct object for the test program
```

The *.com files contains information that has to be changed if the files are copied to another directory. E.g. the command

```
$ @[BOB.IMPORT.XASM]PARSE
```

in XASM.COM should be changed to the new directory-name. If one puts

```
xasm == "@[BOB.IMPORT.ASM]XASM"
```

in the login-file the assembler can be started with the command

```
XASM FILENAME
```

FILENAME.ASM should then contain the assembler program in VMS-format. The

file should start with the line :

```
IDT    'FILENAME'
```

and end with

```
END
```

It is important that capital letters is used in FILENAME.ASM.

The assembler will produce two output files :

```
FILENAME.LIS - information about errors during assemblatic

FILENAME.MPO - relocatable module. Is used by the linker.
```

A four pages long assembly program takes only a few seconds to assemble.

The Linker

For an extensive description of the linker see [20].

The files concerning the linking are collected in

```
[BOB.IMPORT.LINKER]
```

They are

```
README.DAT  - Installation and verification guide
LINKER.COM  - Executes the linker
LINKLINK.COM - Re-links the assembler
PARSE.C25   - Parses patnames for VAX 2.5 O.S.
PARSE.COM   - Used by the above procedures to parse
              pathnames for 3.0 O.S.
LINKER.EXE  - Executable code for the assembler.
LINKER.OBJ  - Assembler objet code, used in re-linking
LINKRTS.OLB- Source/ object library
TEST1.ASM   - Source for the assembler test program
TEST2.ASM   - Source for the second test program
```

```
TEST1.LIS   - Correct listing for the test program
TEST2.LIS   - Correct listing for the test program
TEST.CON    - Control file for the test
TEST.MAP    - Correct map file
TEST.LOD    - Correct load module
```

The *.com files contains information that has to be changed if the files are copied to another directory. E.g. the command

```
$ @[BOB.IMPORT.LINKER]PARSE
```

in LINKER.COM should be changed to the new directory-name. If one puts

```
xlink == "@[BOB.IMPORT.LINKER]XLINK"
```

in the login-file the assembler can be started with the command

```
XLINK FILENAME
```

The linker takes relocatable units and links them together to an object file. To execute the linker one has to write a command file, see [19]. It could look like (capital letters) :

[BOB.EXJOBB.TMS]FILENAME.CON :

```
TASK KOD
PROGRAM >0000
DATA >0000
INCLUDE [BOB.EXJOBB.TMS]FILENAME
END
```

The linker produces two output files:

```
FILENAME.MAP - contains the errors during linking
FILENAME.LOD - object code
```

The object code can then be used as input to the simulator.

The linking takes less than ten seconds.

The Simulator.

The simulator is a software program that is available for VAX and IBM-PC. It can, off-line, simulate the behaviour of a real TMS 320 and is a very useful tool when testing out the software. A thorough description of the simulator can be found in [19].

To start the simulator write

```
RUN [BOB.IMPORT.SIM]SIM.EXE
```

(Or, better, you can define this as XSIM in your login-file).

On the prompt 0/1 ? answer 0 (return) and then strike return again. You will get a list of all the commands available. The list is slightly hierarchical, for example you can get all the I/O commands by writing IOH.

The TMS assembly program to be tested should now have been written with the host computer editor. It is loaded into the (artificial) TMS memory by the command :

```
L [DIRECTORY_NAME]FILENAME.MAP
```

Note that capital letters must be used everywhere (slightly irritating). Now suitable breakpoints can be set. Investigate this by writing :

```
BH
```

This will list all the commands for setting and resetting different kinds of breakpoints. For a thorough description see [19]. To start the simulation write

```
R
```

A simulation can be halted with Ctrl-c at any time. It is also possible to single step by writing

```
SS
```

To display the current status of the processor write

```
ST
```

This will display all registers, the current instruction and the number of clock pulses since the last reset.

To look at the data or program memory write

```
MH
```

which will display all the commands available for doing this. Input and output files can be 'connected' to the simulator. Write IOH to learn more about this. The input data should be in TMS arithmetic format, e.g. 7FFF is the largest representable positive number. More about the arithmetic format can be found in section 4.

To save your commands in a journal file write

```
JF
[DIRECTORY_NAME]JOURNAL_FILE
```

To execute the commands in such a file write

```
EX
[DIRECTORY_NAME]JOURNAL_FILE .
```

Finally to stop the simulation write

```
Q
```

The Evaluation Module

The Evaluation Module (EVM) enables a user to test if the TMS 320 meets the speed requirements of the application. A firmware package contains a debug monitor, editor, assembler, reverse assembler, EPROM programmer, communication software to talk to two EIA ports and an audio cassette interface. Either source code or object code can be downloaded into the EVM via the EIA ports provided on the board. These supports baud rates of 110-19200 baud. There was however some problems when using baud rates of over 600 baud when dumping an assembly program from the mass storage. The

board contains a Debug Monitor including over 60 commands with 8 breakpoints, single step and software trace.

The board can be configured either as a stand alone unit with just a cassette recorder or with a host CPU as mass storage, see figure 3.9.

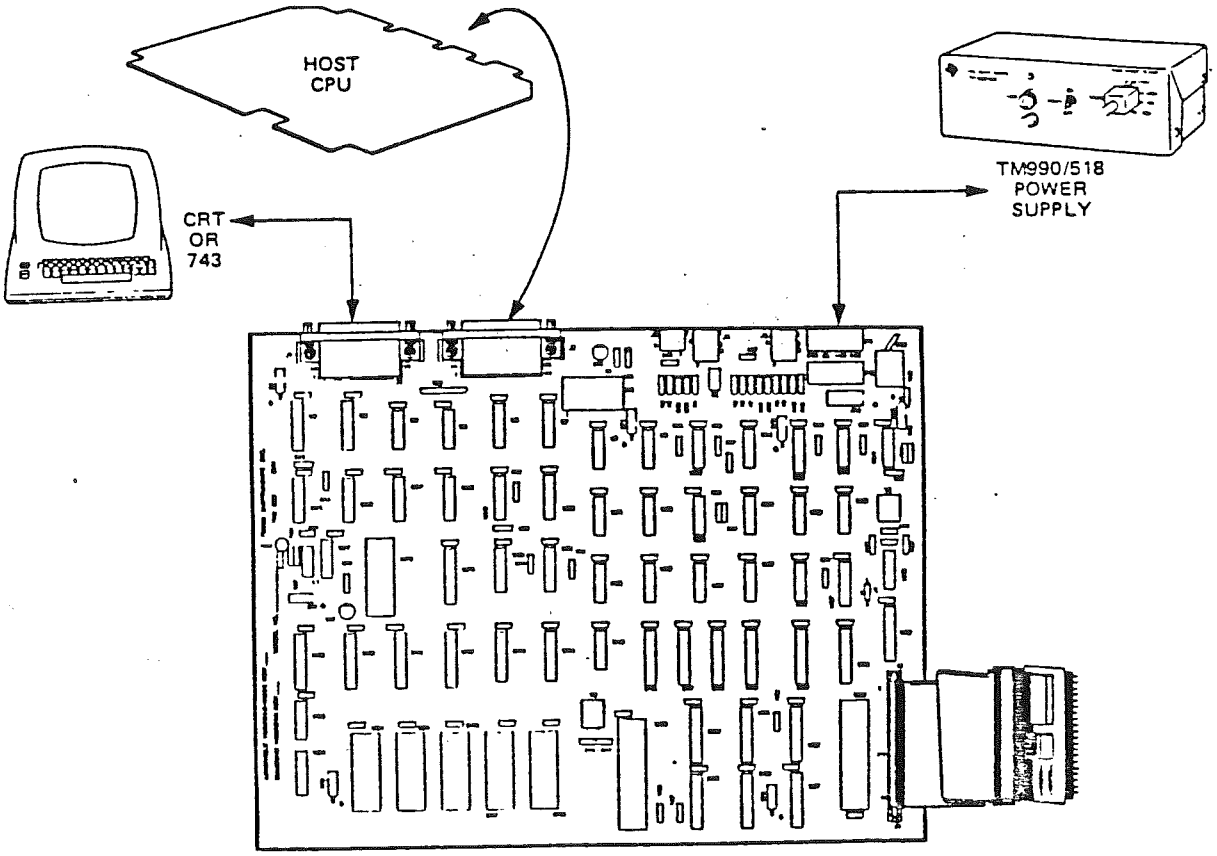


Fig 3.9

If a host computer is used there is of course no reason for using the on board editor or assembler.

For more information about the Evaluation Module, see [21].

The Analog Interface Board

The analog interface board has A/D and D/A converter (12 bit) together with an anti-aliasing lowpass filter. It is compatible with the evaluation module. The maximum sampling rate is limited by the A/D converter to 20,000 Hz.

4 Signal Processor Arithmetics

The TMS 320 has like most other signal processors fixed point arithmetics. No true floating point signal processor is yet available, although for example the Hitachi's HD 61810 has floating point support for a number representation with a 16 bit mantissa and 4 bit exponent. Floating point support normally only means that instructions are available for implementing floating point operations. The new TMS 32020 processor have e.g. floating point support in the form of instructions for normalization of the mantissa and calculation of computed shifts. (Shift the accumulator x bits where x is stored in a register or data memory.)

4.1 Floating Point Software Routines

An example of how floating point operations can be implemented on the TMS 32010 is given in appendix 1 in the form of some floating point macros. The lack of proper instructions for shift of the accumulator and normalizing of the mantissa makes this approach slow. Floating point multiplication will be about 100 times slower than fixed point multiplication. This is probably not acceptable. Another drawback with those routines is that IEEE standard for floating point representation on microprocessors is not used.

Software floating point procedures for the TMS 320 are further discussed in [7], where it is suggested that additional hardware should be cludged to the processor to speed up the normalization of the mantissa. The 7.8 microsec. multiply and 18.4 microsec. addition time mentioned there could be compared to the 24 and 25 microsec. of the 8086/8087 co-processor arrangement. IEEE standard is used. This is not a very attractive alternative either.

The lack of floating point operations is a major problem with signal processors. This problem can sometimes be avoided by choosing the algorithm in such a way that all data can be represented in the same way internally. The TMS 320 makes implicit use of the left point two's complement format which means that all 16 bit numbers represent real numbers in the interval [-1,1). The 16 bit of the TMS is used as one sign bit and a 15 bit mantissa. Therefore all numbers between -1 and 1 should be multiplied by a factor 2^{15} and rounded

to the nearest integer before conversion to internal binary form.

A program for conversion between 16 bit left point two's complement format and real numbers is shown in appendix 3.

4.2 Scaling Factors

Due to the fixed point number format it is up to the programmer to interpret the 16/32 bit data correctly. The binary pattern 0100.0000.0000.0000 might stand for the real number 0.5 as well as $0.5*2^8$ if a scaling factor of 2^8 is assumed for that number. The most practical would be to use the same scaling factor for all data. If a larger dynamical range is needed one must however use different scaling factors for different data. Another way to interpret fixed point format with scaling factors is that only mantissas are stored and the exponents are fixed for each data. Since they are not stored they must be remembered by the programmer. A way to handle this automatically is to use the automatic code generator proposed in section 5. The scaling factors for each data can then be supplied by the user and the correct code can be generated automatically using correct scaling factors for each data.

The problem of choosing appropriate scaling factors is often 'solved' by guessing or in the best case by performing simulations of the algorithm in 'typical situations' on a floating point processor. Procedures for simulating calculations in fixed point format can easily be written in Pascal e.g. by representing fixed point numbers by records :

```
type data = record
            begin
              case i:integer of
                1: array[0..31] of 0..1;
                2: r:real;
            end;
```

Another possibility is to use the 'chop(x)' instruction in the program package CNTLC, see [9].

The scaling factors are most often chosen as exponents of 2 since conversion

between different scaling factors then corresponds to simple binary shifting of data.

The performance of the algorithm can be very dependent on the scaling factors. If a to small factor is used i.e. the data x is represented internally by $x*2^n$ where n is smaller than required there will be loss of accuracy since the data are shifted out to the right. This loss of accuracy will build up during the computations and can eventually result in erroneous results.

On the other hand if the scaling factor is chosen to large there is an increased risk of overflow occuring i.e. the data are shifted out to the left. This is even worse since it is in this case the most significant bits that are lost. If one uses the TMS 320 the errors can be made smaller setting the processor in the saturation mode described in section 3.

Choosing appropriate scaling factors is a problem with fixed point arithmetics and this is why changeable scaling factors, i.e. floating point numbers, where invented.

When finally the suitable scaling factors have been chosen there must be ways of operating on pairs of data with different scaling factors. E.g. there should be instructions for addition,subtraction, multiplication and division of data represented in different forms. For instance when two numbers with different scaling factors should be added one of them must be shifted before performing the addition. Such instructions is not directly available in the TMS 320 but can be implemented in the automatic code generator as macro's. Such macros ADDSC (add scaled numbers) and MULSC (multiply scaled numbers) are shown below.

Here again the advantages with automatic code generation are appreciated. The scaling procedure when writing assembly code 'by hand' is a great burden on the programmer. Taking care of a number of different ways to represent data and an even greater way of combining different data with different representations is quite troublesome. This is easily taken care of by the automatic code generator.

Remember though that the problem only arises when different scaling factors must be used for different data. In e.g. applications with digital filters, (which is the normal use for this processor), this can usually be avoided by

scaling the filter coefficients by the same factor. (E.g. the largest coefficient or the sum of the absolute values of the coefficients, the later is called worst case scaling).

Macros for different scaling factors

ADDSC (A,B,C,EA,EB,EC)

$C := 2^{-EC}(A*2^{EA} + B*2^{EB})$

Adds two 16-bit numbers A and B stored in data ram with, possibly different, scaling factors EA and EB. The sum, C, is stored in data ram with a third scaling factor. WARNING: The overflow flag is not set if an overflow occurs.

Ex. ADDSC(A,B,C,-15,-14,-15) where A=4000H and B=1000H (hexa decimal numbers) yield the answer C=6000H. This corresponds to the fact that $4000H*2^{-15} + 1000H*2^{-14} = 6000H*2^{-15}$. That is 0.5 + 0.25 = 0.75

Since overflow and underflow has to be avoided in the intermediate steps the procedure has been divided into three cases. Case 2 requires that a macro for shifting a double word left and storing the upper 16 bits in data RAM has been written. (The SACH instructions normally only works with shifts of 0,1 and 4.)

1) $EA \leq EB \leq EC$

```
LAC  (A,EA-EC+16)
ADD  (B,EB-EC+16)
SACH (C)
```

2) $EA \leq EC \leq EB$

```
LAC  (A,EA-EB+16)
ADDH (B)
SACHE(C,EB-EC)
```

3) $EC \leq EA \leq EB$

```
LAC  (A,EA-EC)
ADD  (B,EB-EC)
SACL (C)
```

MULSC (A,B,C,EA,EB,EC)

$C := 2^{-EC}(A*2^{EA} * B*2^{EB})$

Multiplies two numbers A,B in data RAM with possibly different scaling factors and stores the product in data memory C. The procedure is divided into two cases. Case 1 requires the extended SACH macro. Case 2 requires that a macro for shifting the accumulator left and storing the lower 16 bits has been written.

1) $-16 \leq EA + EB - EC \leq 0$

```
LT   (A)
MPY  (B)
PAC
SACHE (C,EA+EB-EC+16)
```

2) $0 \leq EA + EB - EC < 16$

```
LT   (A)
MPY  (B)
PAC
SACLE(C,EA+EB-EC)
```

Summary of 4.2

It is not advisable to try to implement floating point routines on the TMS 32010. Better is to store only the mantissa and attach each data a constant exponent called scaling factor. The choice of scaling factor is important.

Remark

Another possible solution is to only store an exponent and have no mantissa. The exponent can be fractional not to give to large quantization steps. This is described in the next section, and is there called the FOCUS number system.

4.3 FOCUS Number System

FOCUS is a number system especially useful for computer control and signal processing applications on processors which lack a hardware multiplier / divider. It is based on a representation of numbers without any mantissa and with fractional exponent. In this number system multiplication and division becomes very fast. Addition and subtraction can be performed using a one-dimensional lookup table, see [15], [16] and [18].

The name FOCUS was chosen to emphasize that this representation concentrate available states near zero as an analogy to how the human eye concentrates near 'a center of focus'. This is however true also for a normal floating point system.

The choice of number representation in computer systems have been extensively debated, see [27]. The rich variety of different systems existing today proves the nonexistense of a universally superior solution. When implementing floating point numbers on a fixed point processor one can choose the number representation oneself. The IEEE standard format for 32 bit floating point numbers is

```
 32  31          22                        0
[ S | EEEEEEEE | MMMMMMMMMMMMMMMMMMMMMMM ]
```

Where S is a sign bit, the E:s eight exponent bits and the M:s 23 mantissa bits.

When choosing number representation for a digital control system or a signal processing system one must remember that many systems should have the possibility to represent both large and small numbers with high accuracy. The choice of number representation one makes will be a trade-off between a large number range and high accuracy i.e. possibility to represent numbers separated with small steps. The instruction set of the processor might also

influence the choice. The TMS 320 has possibilities to handle 16 bit quantities only and it is advisable to store the exponent and mantissa separately:

```
          15  14                0
Mantissa [ S | MMMMMMMMMMMMMMM ]

          15  14                0
Exponent [ S | EEEEEEEEEEEEEEE ]
```

The roundoff error introduced through the finite number length can be modelled by random noise. The FOCUS system has a constant quotient between two neighbouring representable numbers. This gives a logarithmically uniform distribution of representable numbers. The quantization noise introduced thereby is in some cases smaller than with a 'normal' number representation. This is true under some conditions of the probability distribution on the data to be quantizised, as we shall see in a later section.

The FOCUS representation

FOCUS is a logarithmic coding system in which no mantissa is given. This separates it from usual floating point systems. The numbers are represented with one sign bit and an offset binary for the integer part of the exponent.

Ex. A system with 8 bits is used and one decides upon using 3 fractional bits.

```
[ S ]   [ E | E | E | E ] . [ F | F | F ]
 sign        exponent          fraction
```

If an offset of 1000 for the integer part of the exponent is used one will have:

1000 = 0

1001 = 1

1010 = 2

1011 = 3

1100 = 4

1101 = 5

1110 = 6

1111 = 7

0111 = -1

0110 = -2

0101 = -3

0100 = -4

0011 = -5

0010 = -6

0001 = -7

The smallest representable number is thus

$$2^{-7} * 2^{0} = \frac{1}{128} = 0.0078$$

The largest representable number is

$$2^{7} * 2^{7/8} \approx 235$$

The dynamic range is thus $235 * 128 \approx 30.000$.

EX. 0 1000.000 = 1

1 1001.000 = -2

1 0101.010 = $-2^{-3+2/8} = -0.149$

0 1011.100 = $2^{3+4/8} = 11.3$

A-D and D-A decoding.

To translate analog measurements to digital FOCUS-form one can of course build special hardware devices. One possible solution would be to selectively

multiply instead of add a constant associated to each bit or to construct a converter compound of a number of linear segments approximating a logarithmic curve. Another proposed method is to use the exponential characteristics of semiconducters together with ordinary linear decoders. see [18]. It is however questionable if this gives sufficient accuracy for converters with more than 8-bits. Considerations for compensating for the temperature drift must be taken.

Arithmetic operations.

The main advantage with the FOCUS number system is that multiplication and division are performed by binary adders. Therefore those operations are almost trivial. So is calculations of square roots and exponentiation by 2,4,8... (They can now be performed as binary shifts.)

Addition and subtraction can be performed quickly with a (rather small) lookup table by use of the identity:

$$A + B = A (1 + B / A)$$

Or in the logarithmic domain

$$\log_2(A + B) = \log_2 A + \log_2(1 + B/A) = \log_2 A + F(\log_2 A - \log_2 B)$$

The multiplication and division is, as remarked, easily performed and the addition problem has been reduced to an addition with the constant one. In this way only a one dimensional lookup table for the function F has to be used!

The function F is defined by

$$F(x) = \log_2(1 + 2^{-x})$$

With an extra test one can assure that a > b so when tabulating F(x) one only has to consider x > 0. The signs are handled separately. When a>>b one will

have a + b = a within the accuracy of the number representation. This observation can save a large amount of the table but an extra test has to be performed.

The addition of two 8-bits numbers can in this way be carried out fast, requiring only two binary additions, a single memory reference and some overflow detection. The lookup table will be at most 256 bytes long.

Ex. To add 4 and 2 with FOCUS arithmetic :

```
X = 4        0 1010.000
Y = 2    -   0 1001.000
             ------------
             0 0001.000
```

F(0 0001.000) = 0 0000.101 (from table) ⇒

X + Y = 0 1010.000 + 0 0000.101 = 0 1010.101

$X + Y = 6 \approx 2^{2\ 5/8}$

Summary of 4.3

The FOCUS number system is an unusual way to solve the problem that multiplication and division are slow on normal processors. The proposition that FOCUS gives better accuracy to the calculations is overestimated. This is further discussed in section 4.4. The choice of bits representing mantissa respectively exponent will in general be a trade-off between dynamic range and step size. The choice of no mantissa and fractional exponent can be seen as just one choice of parameters in this trade-off and can of course be stated to be optimal using the 'right' criteria. To discuss this trade-off more seriously there must be made assumptions on the probability distribution of the numbers to be represented.

It is questionable if FOCUS will ever attract any attention, especially when fast on-chip multipliers (dividers ?) becomes available.

When representing real numbers with finite accuracy the numbers will necessarily be quantizised. This can have a drastic effect on the results. As a measure on the quantization effect one often takes the signal distorsion ratio, see [25].

$$SDR = 10 \log \left(\frac{E(x^2)}{E(x - x_q)^2} \right) \qquad (4.1)$$

Here x is the numbers to be quantizised, x_q the quantizised data and E mathematical expectation. This ratio should be as large as possible. It is questionable if this is an appropriate measure in all situations, e.g. there is no extra penalty for underflow.

To use the SDR (signal-disturbance ratio) criteria when choosing number representation one has to have knowledge about the probability function p(x). It is in the following assumed that p(x) is an even function, that the number of quantization steps is large and that p(x) is approximatively constant over each such step. Assuming this approximative formulas for the SDR can be found. For instance when using fixed point format with all numbers in [-1,1] (no overflow) and an uniform quantization, the following formula is valid :

$$SDR \approx 6r + 10 \log(3*E(x^2)) \quad (dB) \qquad (4.2)$$

where r is the number of bits used for representing x_q. From this it is seen that the SDR will increase with the number of bits and that one should try to use the available range as well as possible.

When using floating point numbers or the FOCUS system the quantization will not be uniform. If the number of quantization steps is large and there is no risc for overflow one can get the following approximative formula, see[25] :

$$SDR \approx 10 \log \left[\frac{3\,E(x^2)}{R^2} 2^{2r} \left[\int_{-R}^{R} \frac{p(x)}{(g'(x))^2}\, dx \right]^{-1} \right] \qquad (4.3)$$

Here R is the range, 2^r the number of steps, p(x) the probability function and

g(x) determines the step size through :

$$g(-R) = -R$$

$$g(R) = R$$

$$\Delta_i(x) = \frac{2R}{2^r g'(x)}$$

where Δ_i is the step size near x. A large g'(x) will thus give small steps. The problem of choosing g(x) to maximize (4.3) is solved by the following result:

The SDR is maximized for the g(x) satisfying

$$g'(x) = \text{const } p(x)^{1/3} \qquad (4.4)$$

A proof of this is given in appendix 2.

The particular choice of g'(x) given by the FOCUS number system where the numbers are logarithmically distributed is thus seen to be optimal for one special p(x). There is however no way to make statements about which number system that introduces the least noise without further knowledge about p(x), as done in [18] (at least not with the normally used SDR-criteria).

Different g(x) has been tried to minimize the quantization noise in e.g. A/D converters for signal applications. These g(x) has been motivated by knowledge about p(x), e.g. when x represents speech. The choice of

$$g(x) \approx \frac{\text{const.}}{|x|} \quad \text{(except near } x=0\text{)}$$

is called the μ-law and is standard in the USA for A/D converters.

Summary of 4.4

The major trade-off is between large dynamic range and small step-size. The FOCUS number system uses a constant quotient between adjacent representable numbers but there is no reason for favouring this from a SDR point of view. Usual floating point systems will also have a large dynamic range FOCUSing (using smaller step sizes) near zero.

5. Automatic Code Generation for the TMS 32010

5.1 Introduction

The TMS 32010 is programmed in an assembly language that is special for the processor. The instruction set consists of approximatively 60 different instructions including branches, subroutines and interrupts. All calculations are performed in two's complements fixed-point arithmetics. The assembly language is further described in [11], [20] and [29].

TMS assembly code can easily be created in two ways. The first alternative is to use a line oriented editor on the evaluation board called TMS EVALUATION MODULE (EVM). For a description of this see [21]. The second and more attractive alternative is to create the code on a host computer and dump the source code to the TMS processor. The dumping is easily performed with help of the EVM board, see [21].

When using a host computer the code can be generated 'by hand' with any available editor. This is a little faster than using the EVM-editor, since one then probably can use a screen oriented editor. Another possibility, and this is what will be described here, is to use a program written in some high-level language, e.g. Pascal or Prolog, which produces an assembly code file as output. This has several advantages. The number of parameters, the values of certain coefficients or the number of iterations in some loop in an algorithm can be determined automatically and changed when needed much faster than by rewriting the whole assembly program by hand. This of course shortens program development time considerably. Also small deterministic loops can be expanded and addresses that are knowned in advance can be precalculated. This shortens the execution time. Other advantages of automatic code generation are collected in the next section.

Automatic code generators have existed for some time mostly for digital filter applications e.g. :

- IBM for Real Time Signal Processor (IEEE ASS P Feb 83)
- Atlanta Software Digital Filter Design Package for TMS 320
- Helsinki University, for NEC 7720 (Eur. Conf. on Circ. Th. and Design, Stuttg art 9/83)
- Twente University, for NEC 7720 Special language for digital filter impl., simulator (LISP-based software) (EURASIP, 1983)
- Mechanical Engineering Automatic Control Group, University of Paderborn, West-Germany Code generation for digital controllers.

5.2. Advantages With Code Generators.

The following is a a citation from L.R. Morris, Automatic Generation of Time Efficient Digital Signal Processing Software,see [29] :

" The technique for generating correct in-line code is simple. An existing high-level language signal processing program which employs looping structures and involves arithmetic/logical computation to select data or effect loop control (and thus program sequencing) is modified to produce another related high-level language program which, at run time, automatically generates an in-line program.

The generated program is most often assembly language but may be high-level language for some applications. The modifications to the original program consists mainly of replacing the signal processing computational kernel with one or more WRITE statements. As suggested earlier, the code thus generated can effectively incorporate (and thus eliminate) all explicit deterministic runtime calculation. The generated program will then consist of a time-optimized linear instruction sequence, with the optimization occuring at program generation time rather than at run time.

A valuable attribute of this technique is that future changes in the algorithm can first be implemented and tested by modification of the original high-level language program. The results of the modifications can then be rapidly propagated into fast operational code by simply changing the generator program to reflect the altered 'original' program, and then automatically regenerating a new time efficient in-line program."

To summarize the arguments for automatic code generation :

- The code generator can be the last step in a large system for automatic program development. This is the case for high-level language compilers.

- Different macro facilities becomes available. E.g. for scalar products, matrix products, matrix inversion, FFT-calculation or digital filter design. For an example of a macro for butterfly operations (used in FFT calculations) see figure 5.1.

- Arithmetic-logical possibilities to choose data or program flow completely automatically.

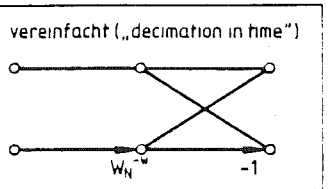
- Easier debugging and rewriting of programs.

- Easier to change parameters of the algorithm.

- In-line code can be produced with less effort. This means that all data independent operations are performed at code generation time instead of at run-time and that loops are broken up into several pieces of linear code. This is positive since branching is slow on the TMS 320, due to the high level of pipelining of instruction fetch and instruction execution which has to be broken up at branches. Therefore loops and subroutines should be avoided. An example is given in [29] where an in-line coded program is compared to a program with subroutines. The algorithm was a 64 - point FFT which run at 0.975 ms when in-line code was used and at 1.897 ms when coded with subroutines. It should however be remarked that in-line code gives longer programs. The choice of method has to be a trade-off between execution speed and program length. The new generation of signal processors will probably have more program space available making in-line code even more advantageous. In fact, in the last decade a number of new algorithms have been developed in which a reduction in data-dependent operations is achieved at the expense of relatively increased algorithm complexity. All of these algorithms will increase the advantage of in-line code. For a discussion of this see [29].

- Better documentation of programs is possible.

Bild 6. Elementaroperation „Butterfly", vereinfacht („Decimation in Time")




```
* UNTERPROGRAMM INPLACE-BUTTERFLY
*
* PARAMETER IM DATEN-RAM:
* WR                    REALTEIL/WURZEL
* WI                    IMAGINARTEIL/WURZEL
* ADUP                  DATENRAMADRESSE REALTEIL/OBERER AST  (RE1)
* ADDW                  DATENRAMADRESSE REALTEIL/UNTERER AST (RE2)
*
BUTT    LAR   1,ADUP            ADRESSE OBERER AST
        LAR   0,ADDW            ADRESSE UNTERER AST
        LT    *+                RE2
        MPY   WR                RE2 * WR
        PAC                     PRODUKT IN ACCU
        LT    *-                IM2
        MPY   WI                IM2 * WI
        SPAC                    (RE2 * WR - IM2 * WI)
        SACH  ZWR,1             ZWISCHENERGEBNIS IN ZWR (*2)
        MPY   WR                IM2 * WR
        PAC                     PRODUKT IN ACCU
        LT    *,1               RE2
        MPY   WI                RE2 * WI
        APAC                    (RE2 * WI + IM2 * WR)
        SACH  ZWI,1             ZWISCHENERGEBNIS IN ZWI (*2)
        LAC   *,15,0            RE1
        SUB   ZWR,15            RE1 - ZWR
        SACH  *+,0,1            ----- ERGEBNIS RE2  (/2)
        ADDH  ZWR               RE1 + ZWR
        SACH  *+                ----- ERGEBNIS RE1  (/2)
        LAC   *,15,0            IM1
        SUB   ZWI,15            IM1 - ZWI
        SACH  *,0,1             ----- ERGEBNIS IM2  (/2)
        ADDH  ZWI               IM1 + ZWI
        SACH  *,0,0             ----- ERGEBNIS IM1  (/2)
        RET                     RUECKSPRUNG
```

Bild 7. Butterfly-Code für TMS320

Fig 5.1

There are of course problems with automatic code generation as well :

- In-line code is longer.
- No code generator might be available.
- There is some extra work in learning how the code generator works. This makes the technique inefficient for short programs.
- The code will not be as optimized as when written by hand.

An example of a dialog when using a codegenerator for automatic digital filter design is shown below.

TMS CODE GENERATION PROGRAM

(C)COPYRIGHT (1984): ATLANTA SIGNAL PROCESSORS INC., VERSION 1.02
THIS PROGRAM GENERATES ASSEMBLER CODE FOR THE TMS 32010 SIGNAL PROCESSOR CHIP FROM DIGITAL FILTER DESIGNS GENERATED BY THE IIR, KFIR, AND PMFIR DIGITAL FILTER DESIGN MODULES OF THE Digital Filter Design Package (DFPD)

FILENAME OF FILTER FILE: LPF.FLT

DO YOU WISH TO SPECIFY THE FILTER I/O LOCATIONS: N

NEW SAMPLING FREQUENCY (KHZ): 10

.
.
Specifications of the Data Memory Organization
.
.
Specifications of the Program Memory Organization
.
.
.

After all the options have been selected and the appropriate data entered, CGEN puts together a TMS32010 program as specified, and writes it into the specified disk file. Then, a summary of program characteristics is displayed on the screen as in the following example

```
INFINITE IMPULSE RESPONSE (IIR)
ELLIPTIC LOWPASS FILTER
16-BIT QUANTIZED COEFFICIENTS

FILTER ORDER =      4
SAMPLiNG FREQUENCY =  10.000 KILOHERTZ

A(1,1)        A(1,2)        B(1,0)        B(1,1)        B(1,2)
-.594849      .204315       .165970       .277725       .165710

**  CHARTERISTICS OF DESIGNED FILTER  **

LOWER BAND EDGE       .00000
UPPER BAND EDGE      2.00000
NOMINAL GAIN         1.00000
NOMINAL RIPPLE        .01000
MAXIMUM RIPPLE        .00908
RIPPLE IN DB          .07849
```

The TMS program is completely automatically generated, with initialisation: routines for the A/D-D/A card etc:

```
>
FILIN   EQU     0
FILOUT  EQU     1
.
.
(4 pages)
.
.
        SACH    18
        RET
        END
<
```

5.3. The Code Generator CODEGEN.PAS

This is an example of how the code generation technique can be used to implement an identification algorithm on a TMS 320. The program produces TMS assembly code for recursive parameter estimation of the A and B polynomials in the ARMAX model

$$Ay = Bu + e$$

It is written in Pascal on a VAX 11/780. When reading this section compare with the Pascal code in appendix 4. The program can be partitioned into the sections:

Declarations.

The first part of the program defines tabulator positions and a default name, 'outfile.asm' on the outfile to which the assembly code will be produced.

Code generation procedures.

These procedures are used to produce assembly code on the output file 'outfile.asm'. They are used by the assembly instruction procedures and the main program as described later. Source statements in TMS assembler contain four ordered fields separated by one or more blanks. The source statement line may be as long as the source file format allows, however, the assembler will truncate after 60 characters without warning. Therefore nothing else than comments may extend past column 60. The function c converts an integer to a varying of char (special for VAX/VMS - Pascal). The comment procedure allows the programmer to write TMS comments which always should start with an asterisk (*) in the first character position. Comments have no effect on the assembler.

Assembly directive procedures

Each of these procedures creates when called from the main program an assembly directive, see [20]. Assembler directives supply the assembler with information of the following categories:

- Directives that affect the location counter.
- Directives that initialize constants.

- Directives that provide linkage between programs.

- Miscellaneous directives.

All directives accepted by the assembler are not included, since they was not needed for the application. New directives are however easily added.

Assembly instruction procedures

Each of these procedures creates an assembler instruction on the outfile when called from the main program. All instruction are included, however all combinations of instructions and operands have not been implemented (e.g. the operations for indirect addressing is not complete). The instruction set is however easily extended when noted that:

- All text should be written with capital letters for the TI assembler to accept it.

- All instructions are written at tab1 by the w procedure.

- All assembly instruction procedures end with a wln (write line).

- The function c converts an integer to an identifier

- No instruction should have a name that coincides with a reserved word in Pascal. Therefore e.g. 'and' is changed to 'and_'.

Main program

The main program consists of calls to assembler directives and instruction procedures which creates the assembly code. Here all the Pascal facilities can be used which is a large advantage. Only the main program has to be rewritten if another algorithm than the recursive parameter estimator is to be implemented.

The main program in appendix 4 asks for the number of parameters to be estimated in an ARMAX - model and the name of the assembly output file. The algorithm used for the parameter estimation is the MIT-rule and is described in [6] and [23].

As a test of how the code generator might have been written in Prolog the program codegen.pro was written as a 'direct' translation of the Pascal program to Prolog. The reason for this test was to see if a more interactive programming environment could be possible in Prolog.

There was no major problems when rewriting the program. One difference between the languages is that prolog lacks global variables. This can however be simulated by use of the predicates retract and asserta (or assertz) which adds or subtracts a rule from the database, see [30]. For instance putting the global parameter col to 13 one might write

```
retract(col())

asserta(col(13))
```

There was however some problems with stack overflow after a while using this solution and therefore the global parameter col was skipped. This is the reason why the assembly program has no straight left margins as seen in appendix 5. This is just an esthetic problem since the TEXAS assembler needs no straight left margins. Another small problem on the computer VAX 11/780 was that the prolog outfile was in UNIX format while the Texas assembler only accept VMS files as input. These UNIX files was therefore converted to VMS files by use of the 'unixtovms' command.

Since the code generation in it self is such an easy programming problem (it is really just some write commands), there is no large difference between using different programming languages.

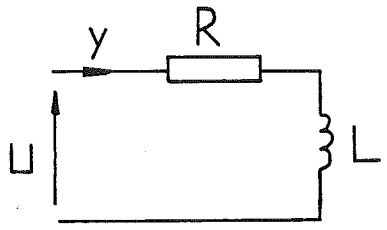
The advantage with Prolog is the interactive programming environment. The disadvantage is that Prolog is not a very well knowned language. For most code generation applications Pascal or even Fortran will do.

The Prolog version of the automatic code generator program and the assembly program generated by it is presented in appendix 5.

6 Practical Experiments
On-line Estimation of Two Parameters in a R-L Circuit

6.1 Theory

Consider the following network, a first degree R-L circuit:



The connection between the current y and the voltage u is given by

$$\dot{y} + \frac{R}{L} y = \frac{1}{L} u$$

where R and L are unknowned. The sampled version of this will be :

$$y_k = a y_{k-1} + b u_{k-1}$$

where

$$a = e^{-\frac{R}{L}h}$$

$$b = \frac{1}{R}(1-e^{-\frac{R}{L}h}) \qquad (6.1)$$

That is

$$R = \frac{1-a}{b}$$

$$L = -\frac{h}{\ln a}\frac{1-a}{b} \qquad (6.2)$$

The parameters a and b have no direct physical interpretation but can

nevertheless be used for fault-detection. There is a one-to-one correspondance between the physical parameters (R,L) and the parameters (a,b). Any domain of the (R,L) space can be translated to a corresponding domain in the (a,b) space.

ASEA RELAYS uses different domains of (R,L) space for signaling alarm when parameter changes occurs (the impedance decreases). For instance the following domains are used in different equipment :

1) $R \leq R_0$ and $L \leq L_0$

2) $R^2 + L^2 \leq A^2$

Here R_0 and L_0, or A can be set by the operator on a panel of control switches.

Figure 6.1 and 6.2 show the corresponding (a,b) domains. A disadvantage with the (a,b) domains are that they are harder to describe. Either (6.2) can be used or an approximation with e.g. polygons has to be made.

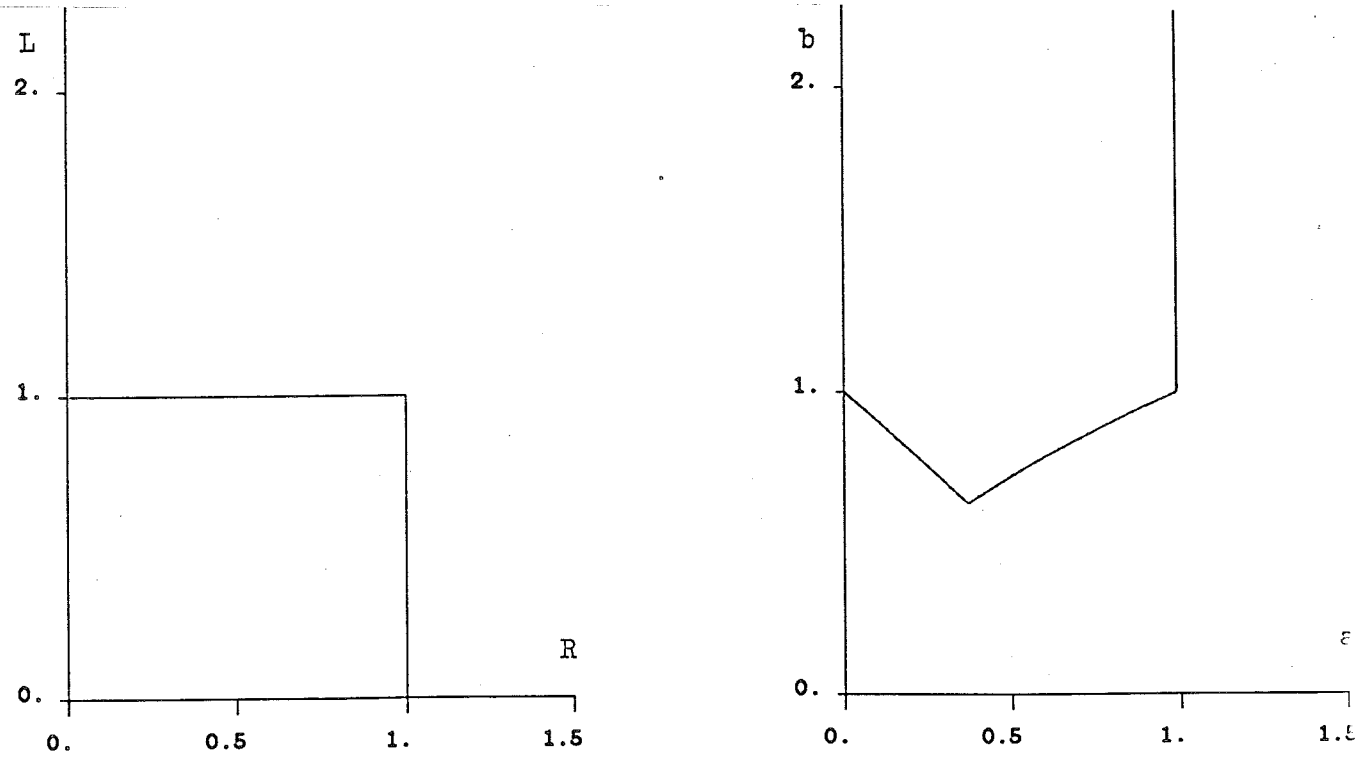


Fig 6.1

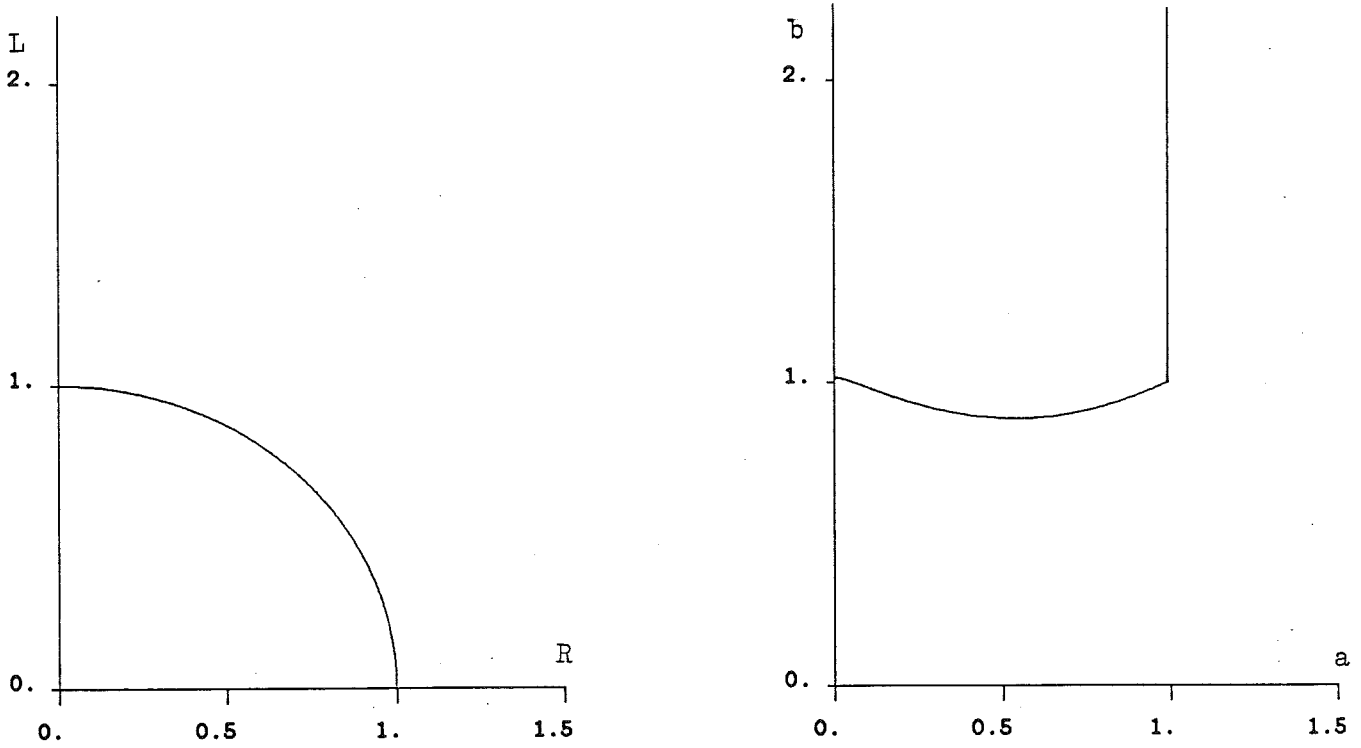


Fig 6.2

Sensitivity, Choice of Sampling Rate

Since

$$a = e^{-\frac{R}{L}h} = e^{-\frac{h}{T}} \quad \text{where} \quad T = \frac{L}{R}$$

Simple calculations show that

$$\frac{dT}{T} = -\frac{T}{h}\frac{da}{a}$$

For a given relative coefficient precision the equivalent precision of the time constant is thus inversely proportional to the sampling period. Therefore the sampling period should not be chosen to small. Moreover

$$\frac{dL}{L} = \left[-\frac{1}{1-a} - \frac{1}{a \ln a}\right] da - \frac{1}{b}\, db$$

$$\frac{dR}{R} = -\frac{1}{1-a}\, da - \frac{1}{b}\, db$$

For small h we have

$$\frac{1}{1-a} \approx \frac{T}{h}$$

This again shows that a too small sampling period should not be used.

A reasonable value on T/h is around 2-4, see [10].

6.2 Code Generation

The code generator is now used to generate code for fast recursive parameter estimation of the two parameters in the model:

$$Y(k) = aY(k-1) + bU(k-1)$$

The method with constant scalar gain is used, see section 2. A suitable constant P , see section 2, has been chosen by trial and error, P=0.8.

RUN CODEGEN
The user program should now be in userscode.pas
Outfile ([bob.exjobb.tms]outfile.asm) : (carriage return)
na = 1
nb = 1
Scalefactor for a and b = (-15) : -15
adaption factor p : 0.8
ok

The assembly program is presented in appendix 4.

6.3 A Simulation With the Assembly-Program on VAX 780

As a test of the code generation, the algorithm and the TMS-simulator on VAX 780, a file with input and output data from a first order circuit was generated.

Input and output data Y and U was obtained from IDPAC, see [31]. The a and b coefficients was chosen by the supervisor, Sten Bergman, and kept secret. The input signal was a Pseudo Random Binary Sequence with amplitude of 0.5. There was no noise in this example. The sequences was converted to ASCII format and used as input file to the simulator.

The assembly file obtained from the code generator was loaded into the simulator. When the simulator was started the estimates presented in fig 6.3 - 6.5 was produced on an output-file. The figures correspond to different choices of the scaling factor.

After 100 samples the estimates had converged to 3 decimal places:

$$a = 0.850 \quad \text{and} \quad b = 0.353$$

These were the correct values.

By calculating the number of operations performed, (this is done automatically by the simulator), one could see that a maximum sampling-rate of 100 kHz would have been possible in this example. The A/D converter would however had limited this to 20 kHz in practice.

If the matrix gain algorithm would have been used, see section 2, the estimates would in this case have been exactly correct after only two iterations. Such an algorithm would however have been much harder to implement making a much lower sampling rate necessary. The trade-off between algorithm complexity and sampling rate are further discussed in section 2.

Different scaling factors for the a and b estimates have been used. Here the necessity for choosing a proper scaling factor is shown again. If a too large scaling factor is used the calculations will overflow. On the other hand using a too small scaling factor will result in lost accuracy. This is what has happened in figure 6.5 where the estimates converged to

$$a = 0.812 \quad \text{and} \quad b = 0.359$$

The same estimation has also been done in Pascal with double precision. The result is shown in figure 6.6. It was found to agree perfectly with the output file from the simulator, see fig 6.3.

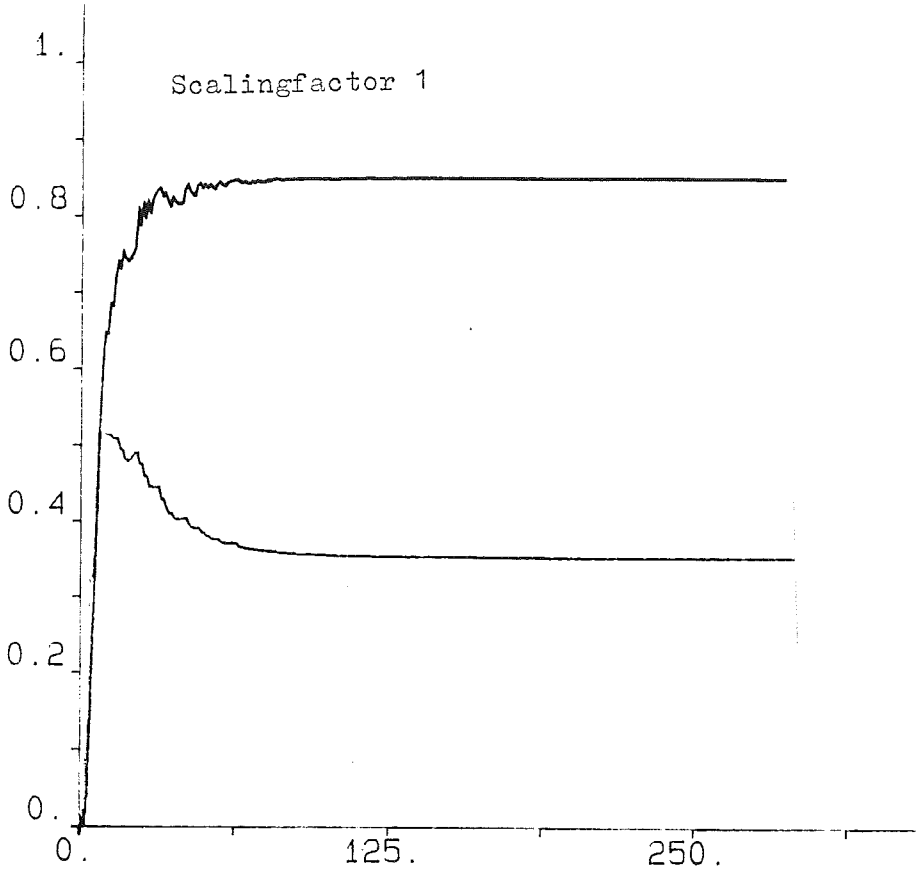

Scalingfactor 1
1.
0.8
0.6
0.4
0.2
0.
0.
125.
250.

Fig 6.3

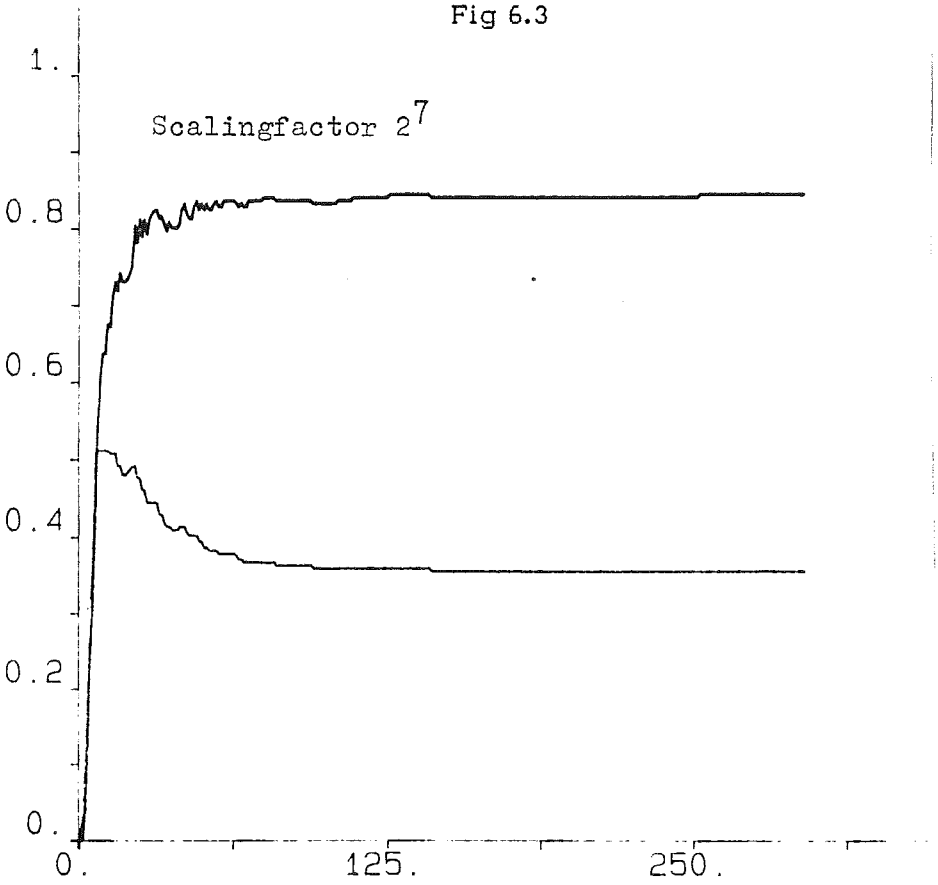

Scalingfactor 2^7
1.
0.8
0.6
0.4
0.2
0.
0.
125.
250.

Fig 6.4

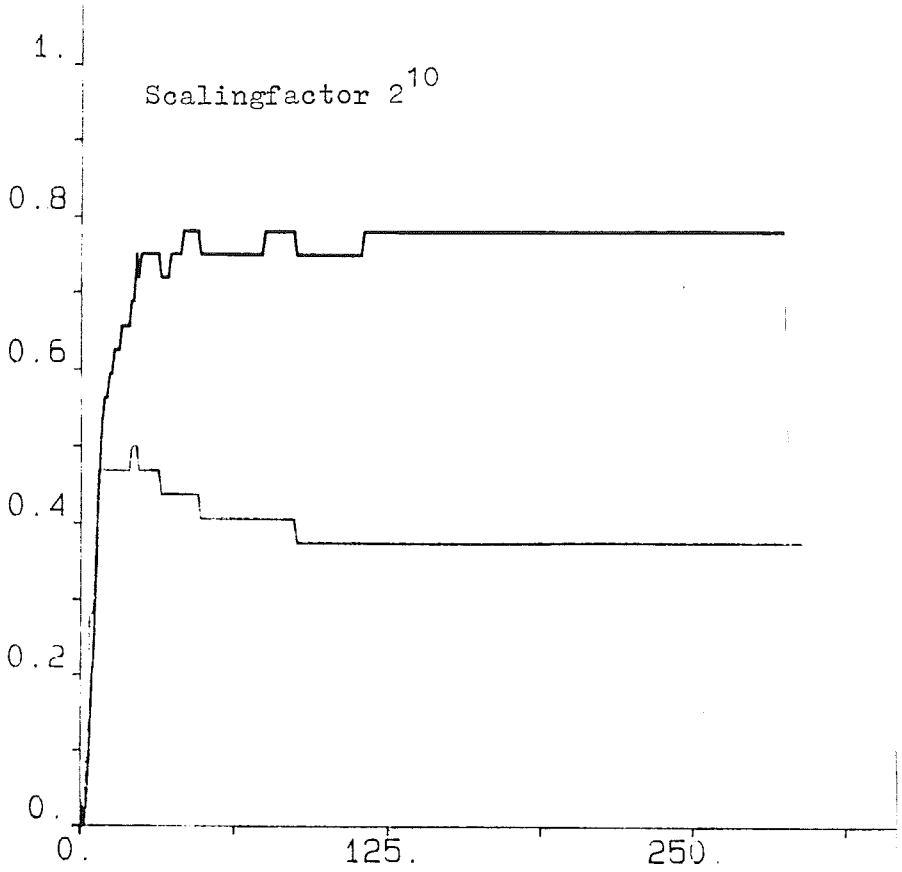

1.
Scalingfactor 2^{10}
0.8
0.6
0.4
0.2
0.
0.
125.
250.

Fig 6.5

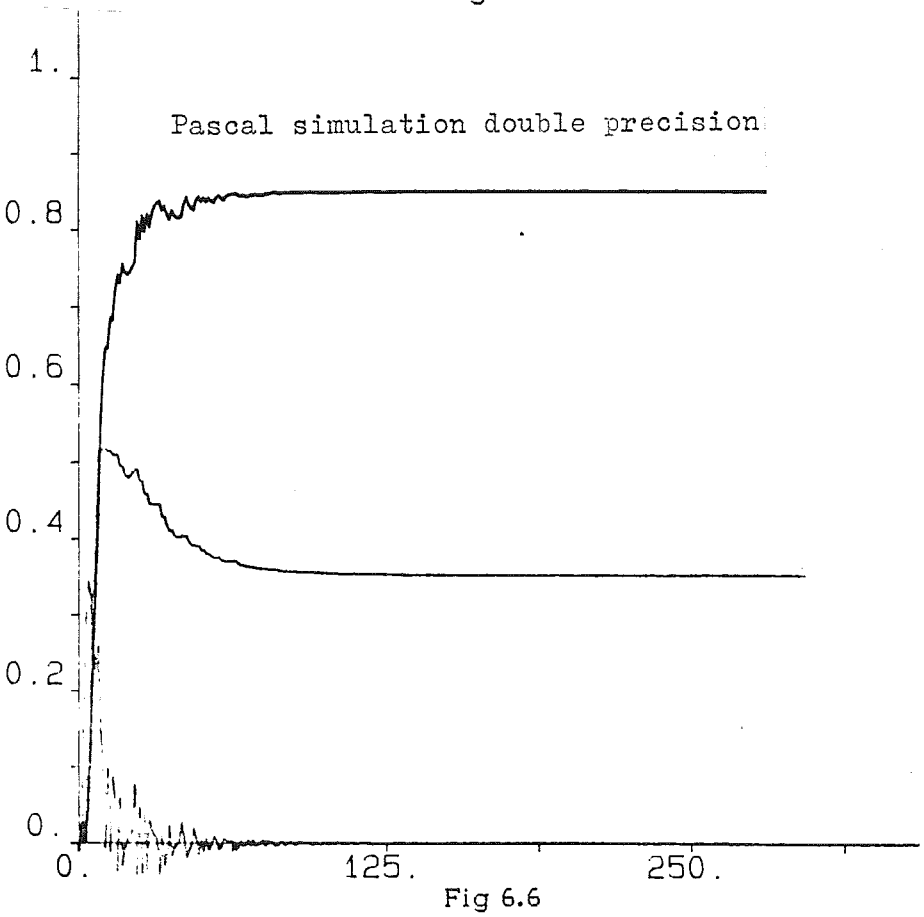

1.
Pascal simulation double precision
0.8
0.6
0.4
0.2
0.
0.
125.
250.

Fig 6.6

Appendix 1 Floating Point Routines

Appendix 1 contains software routines for floating point operations on the TMS 320. They have been tested and found to work. The routines are taken from the master thesis from Karin Sjöholm and Anders Tosteberg, LTH. It was written at ASEA ROBOTICS, Västerås in July 1984.

In the program some macros like $ASG, $MACRO or $VAR have been used. For a description of these see [20].

The routines was unfortunately found to be rather slow. The multiplication of two floating point numbers took around 100 times longer than the fixed point multiplication. The routines are also rather hard to check for bugs since they are long. Another drawback is that they take memory space from the user.

```
* MULTIPLY FLOTING-POINT WORD E AND G,
* STORE THE RESULT IN I
*
FMUL        $MACRO E,G,I
            $VAR J,JE,L
            $ASG 'TMP1' TO J
            $ASG 'TMP2' TO JE
            $ASG '$$LAB' TO L.S
            $ASG L.SV+1 TO L.SV
            UNL                     STOPPAR LISTNINGEN
            LT :G:
            MPY :E:
            PAC
            SACL :J:+1
            SACH :J:
            BZ SL$:L.SV:            MANTISSA C NOLL?
            SUBH C000
            BZ KR$:L.SV:
            LAC :G:+1
            ADD :E:+1               ADDERA EXPONENTERNA
            SACL :JE:               LAGRA NY EXPONENT
            LAC :J:
            AND OCH
            ADD :J:,2
            AND OCH
            BZ ZO$:L.SV:            LEADING ZERO
            LDAX :J:                NORMALISERAT
            ADD ONE,14              AVRUNDING OCH -
OV$:L.SV:   SACH :J:,1              SHIFT UT AV -
            LAC :JE:                EXTRA NOLLA ELLER ETTA
            B SL$:L.SV:
BA$:L.SV:   LDAX :J:
            B OV$:L.SV:
KR$:L.SV:   LAC ONE,14
            SACL :J:
            LAC :G:+1
            ADD :E:+1
            ADD ONE
            B SL$:L.SV:
ZO$:L.SV:   LDAX :J:
            ADD ONE,13              AVRUNDNING   LEADING ZERO
            SACH :J:
            SACL :J:+1
            LAC :J:
            AND OCH
            ADD :J:,2
            AND OCH
            BNZ BA$:L.SV:
            LDAX :J:
            SACH :J:,1
            SACL :J:+1
            LAC :J:+1,1
            AND MINUS
            ADDH :J:
            SACH :J:,1
```

```
            LAC :JE:
            SUB ONE
SL$:L.SV:   SACL :I:+1
            LAC :J:
            SACL :I:                    STARTAR LISTNINGEN IGEN
            LIST
            $END
```

```
*ADDERAR FLYTTALEN F OCH H
*RESULTATET LAGRAS I I (MANTISSAN)
*EXPONENTEN LAGRAS I I+1
*
FADD      $MACRO F,H,I
          $VAR B,D,TE,L,L1,E,G
          $ASG '$$LAB' TO L.S
          $ASG '$NSUB' TO L1.S
          $ASG 'TMP1' TO B
          $ASG 'TMP2' TO D
          $ASG 'TMP3' TO TE
          $ASG 'TAL1' TO E
          $ASG 'TAL2' TO G
          $ASG L.SV+1 TO L.SV
          UNL                       STOPPAR LISTNING
          LST CLOV                  NOLLSTALLER OV
          LAC :F:+1,2
          SUB :H:+1,2
          SACL :B:
          BGEZ OT$:L.SV:            HOPPAR OM EXP-F >= EXP-H
          LAC :H:
          SACL :E:
          LAC :F:
          SACL :G:
          LAC :H:+1
          SACL :I:+1
          B OK$:L.SV:
OT$:L.SV: LAC :F:
          SACL :E:
          LAC :H:
          SACL :G:
          LAC :F:+1
          SACL :I:+1
OK$:L.SV: LAC :B:
          ABS
          SACL :B:
          SUB ONE,6
          BLEZ PL$:L.SV:            HOPP OM ADDITIONEN SKA SKE
          LAC :E:
          SACL :I:
          B SL$:L.SV:
OV$:L.SV: LAC :I:+1                 KORRIGERA EXPONENTEN
          ADD ONE
          SACL :I:+1
          LAC :I:
          AND NOCH
          SACL :I:
          ZALH :TE:                 LADDA IN TECKENBIT
          ADD :I:,15                OVERFLOW AVKLARAT
          B OR$:L.SV:
          $IF L1.SV=0
          $ASG 5 TO L1.SV
SUB0      ZALH :E:                  SUBRUTINER FOR BER. AV-
          ADDH :G:                  ADDITIONEN I-
          RET                       KORREKT POSITION
```

```
NOP
ZALH :E:
ADD :G:,15
RET
NOP
ZALH :E:
ADD :G:,14
RET
NOP
ZALH :E:
ADD :G:,13
RET
NOP
ZALH :E:
ADD :G:,12
RET
NOP
ZALH :E:
ADD :G:,11
RET
NOP
ZALH :E:
ADD :G:,10
RET
NOP
ZALH :E:
ADD :G:,9
RET
NOP
ZALH :E:
ADD :G:,8
RET
NOP
ZALH :E:
ADD :G:,7
RET
NOP
ZALH :E:
ADD :G:,6
RET
NOP
ZALH :E:
ADD :G:,5
RET
NOP
ZALH :E:
ADD :G:,4
RET
NOP
ZALH :E:
ADD :G:,3
RET
NOP
ZALH :E:
ADD :G:,2
```

```
            RET
            NOP
            ZALH :E:
            ADD :G:,1
            RET
            NOP
            ZALH :E:
            ADD :G:,0
            RET
            NOP
            $ENDIF
PL$:L.SV:   LAC :E:                 TAR FRAM TECKENBITEN -
            AND COOO                (+1BIT) LAGRAR DEN I TE
            SACL :TE:
            LAC :B:
            LT ONE
            MPYK SUBO
            APAC
            CALA                    KALLA PA SUBRUTINEN-
            SACH :I:                FOR ADDITION
            BV OV$:L.SV:            HOPP VID OVERFLOW
            SACL :D:
            BZ SL$:L.SV:
            LAC :I:                 TEST OM TALET-
            AND OCH                 NORMALISERAT
            ADD :I:,1
            AND OCH
            BNZ RO$:L.SV:           HOPP OM TALET-
            FNORM :I:,:D:,:I:+1
RO$:L.SV:   ZALH :I:                AVRUNDNING
            ADDS :D:
OR$:L.SV:   ADD ONE,15
            BV OV$:L.SV:            HOPP VID OVERFLOW
            SACH :I:                LAGRA RESULTAT AV-
SL$:L.SV:   EQU $                   ADDITIONEN
            LIST                    STARTAR LISTNING IGEN
            $END
```

```
*LAGRAR DET TILL BELOPPET MINSTA FLYTTALET
*(AV 2 STYCKEN) I :G:
*
FAMIN     $MACRO E,F,G
          $VAR A,B,L
          $ASG 'TAL1' TO A
          $ASG 'TAL2' TO B
          $ASG '$$LAB' TO L.S
          $ASG L.SV+1 TO L.SV
          UNL                    STOPPAR LISTNINGEN
          SOVM
          LAC :F:
          BZ FM$:L.SV:
          LAC :E:
          BZ EM$:L.SV:
          LAC :E:+1
          SUB :F:+1
          SACL :G:+1
          ABS
          SUB ONE
          BGEZ TE$:L.SV:         :E:EXP-:F:EXP>=1
          ZALH :E:               EXP. LIKA STORA
          ABS
          SACH :A:
          ZALH :F:
          ABS
          SACH :B:
          LAC :B:
          SUB :A:                ABS:F:-ABS:E:
          BGZ EM$:L.SV:          HOPP OM ABS:F:>ABS:E:
          B FM$:L.SV:
TE$:L.SV: LAC :G:+1
          BLZ EM$:L.SV:          HOPP OM :E:EXP<:F:EXP
FM$:L.SV: LAC :F:
          SACL :G:
          LAC :F:+1
          SACL :G:+1
          LAC ONE
          SACL XY
          B SL$:L.SV:
EM$:L.SV: LAC :E:
          SACL :G:
          LAC :E:+1
          SACL :G:+1
          ZAC
          SACL XY
SL$:L.SV: ROVM
          LIST                   STARTAR LISTNINGEN IGEN
          $END
```

```
*NEGERAR ETT FLYTTAL
*RESULTATET LAGRAS I H (MANTISSAN)
*OCH H+1 (EXPONENTEN)
*
FINVER    $MACRO F,H
          $VAR L
          $ASG '$$LAB' TO L.S
          $ASG L.SV+1 TO L.SV
          UNL                          STOPPAR LISTNINGEN
          LAC :F:+1
          SACL :H:+1
          ZAC
          SUB :F:,0
          SACL :H:
          BGZ NE$:L.SV:
          LAC :H:,1
          AND OCH
          BZ KL$:L.SV:
          LAC :H:+1
          SUB ONE
          SACL :H:+1
          ZALH :H:
          SACH :H:,1
          B KL$:L.SV:
NE$:L.SV: LAC :H:
          SUB OCH
          BNZ KL$:L.SV:
          LAC :H:+1
          ADD ONE
          SACL :H:+1
          LAC ONE,14
          SACL :H:
KL$:L.SV: EQU $                        STARTAR LISTNINGEN IGEN
          LIST
          $END
```

```
*PLOCKAR BORT INLEDANDE NOLLER RESP ETTOR
*DVS NORMALISERAR TALET
*
FNORM     $MACRO I,J,K
          LAC :I:
          SACL TMP4
          LAC :J:
          SACL TMP4+1
          LAC :K:
          SACL TMPEXP
          CALL SHIFT
          LAC TMPEXP
          SACL :K:
          LAC TMP4+1
          SACL :J:
          LAC TMP4
          SACL :I:
          $END
```

```
        MLIB 'DUAD:ACONSULT.KSJOHOLM.MACROA'
        OPTION SYMLST,XREF
        TITL '                                MATSYSTEM TMS32010'
        PROG  SHIFT
        PSEG
        DEF SHIFT
        REF ONE,F,THREE,TMP4,TMPEXP,MINUS,TMP5
        REF FIVE,NINE,SEVEN,B,D
        PAGE
SHIFT1  LASX TMP4,TMP4,1
        LAC TMPEXP
        SUB ONE
        SACL TMPEXP
        RET
SHIFT2  LASX TMP4,TMP4,2
        LAC TMPEXP
        SUB ONE,1
        SACL TMPEXP
        RET
SHIFT3  LASX TMP4,TMP4,3
        LAC TMPEXP
        SUB THREE
        SACL TMPEXP
        RET
SHIFT4  LASX TMP4,TMP4,4
        LAC TMPEXP
        SUB ONE,2
        SACL TMPEXP
        RET
SHIFT5  LASX TMP4,TMP4,5
        LAC TMPEXP
        SUB FIVE
        SACL TMPEXP
        RET
SHIFT6  LASX TMP4,TMP4,6
        LAC TMPEXP
        SUB THREE,1
        SACL TMPEXP
        RET
SHIFT7  LASX TMP4,TMP4,7
        LAC TMPEXP
        SUB SEVEN
        SACL TMPEXP
        RET
SHIFT8  LASX TMP4,TMP4,8
        LAC TMPEXP
        SUB ONE,3
        SACL TMPEXP
        RET
SHIFT9  LASX TMP4,TMP4,9
        LAC TMPEXP
        SUB NINE
        SACL TMPEXP
        RET
SHIFTA  LASX TMP4,TMP4,>A
```

```
        LAC TMPEXP
        SUB FIVE,1
        SACL TMPEXP
        RET
SHIFTB  LASX TMP4,TMP4,>B
        LAC TMPEXP
        SUB B
        SACL TMPEXP
        RET
SHIFTC  LASX TMP4,TMP4,>C
        LAC TMPEXP
        SUB THREE,2
        SACL TMPEXP
        RET
SHIFTD  LASX TMP4,TMP4,>D
        LAC TMPEXP
        SUB D
        SACL TMPEXP
        RET
SHIFTE  LASX TMP4,TMP4,>E
        LAC TMPEXP
        SUB SEVEN,1
        SACL TMPEXP
        RET
SHIFTF  LASX TMP4,TMP4,>F
        LAC TMPEXP
        SUB F
        SACL TMPEXP
        RET
SHIFT   LAC TMP4                NORMALISERAR TALET I TMP4,
        BLZ NEG                 TMP4+1 OCH TMPEXP
        SUB ONE,7
        BLZ L0080
        SUB F,7
        BLZ L0800
        SUB THREE,>B
        BLZ L2000
        B SHIFT1
L2000   ADD ONE,>C
        BLZ SHIFT3
        B SHIFT2
L0800   ADD THREE,9
        BLZ L0200
        SUB ONE,9
        BLZ SHIFT5
        B SHIFT4
L0200   ADD ONE,8
        BLZ SHIFT7
        B SHIFT6
L0080   ADD F,3
        BLZ L0008
        SUB THREE,3
        BLZ L0020
        SUB ONE,5
        BLZ SHIFT9
```

```
        B SHIFT8
L0020   ADD ONE,4
        BLZ SHIFTB
        B SHIFTA
L0008   ADD THREE,1
        BLZ SHIFTE
        SUB ONE,1
        BLZ SHIFTD
        B SHIFTC
NEG     ADD ONE,7
        BLZ LFF80
        SUB F,3
        BLZ LFFF8
        SUB THREE,1
        BLZ LFFFE
        SUB ONE
        BLZ SHIFTE
        B SHIFTF
LFFFE   ADD ONE,1
        BLZ SHIFTC
        B SHIFTD
LFFF8   ADD THREE,3
        BLZ LFFE0
        SUB ONE,4
        BLZ SHIFTA
        B SHIFTB
LFFE0   ADD ONE,5
        BLZ SHIFT8
        B SHIFT9
LFF80   ADD F,7
        BLZ LF800
        SUB THREE,9
        BLZ LFE00
        SUB ONE,8
        BLZ SHIFT6
        B SHIFT7
LFE00   ADD ONE,9
        BLZ SHIFT4
        B SHIFT5
LF800   ADD THREE,>8
        BLZ SHIFT1
        SUB ONE,>C
        BLZ SHIFT2
        B SHIFT3
        PAGE
```

```
*OMVANDLAR HELTALET I A TILL ETT
*FLYTTAL I B OCH B+1
*A INLÄST FRÅN ADC OCH SHIFTAT EN BIT REDAN
*
FLT         $MACRO A,B
            $VAR L,I,J
            $ASG 'TMP1' TO I
            $ASG 'TMP2' TO J
            $ASG '$$LAB' TO L.S
            $ASG L.SV+1 TO L.SV
            UNL                     STOPPAR LISTNINGEN
            LACK >6                 EXPONENTEN MAX 6
            SACL :J:
            LT :A:                  BINÄRJUSTERING AV
            MPY BINAR               TALET FRÅN ADC
            PAC
            SACH :I:
            BZ NO$:L.SV:
            SACL :I:+1
            FNORM :I:,:I:+1,:J:
NO$:L.SV:   LAC :I:
            SACL :B:
            LAC :J:
            SACL :B:+1
            LIST                    STARTAR LISTNINGEN IGEN
            $END
```

Appendix 2 Quantization Effects

Lemma (The extended Hölders inequality)

Define $\|f\|_p = \left(\int |f|^p \, dx \right)^{1/p}$

If $1/p + 1/q = 1/r$ then

if sign(pqr) = 1 then for all functions f and g we have

$$\|fg\|_r \leq \|f\|_p \, \|g\|_q$$

if sign(pqr) = -1 we instead have

$$\|fg\|_r \geq \|f\|_p \, \|g\|_q$$

Equality occurs if $f^p \equiv \text{const} * g^q$

Proof Case 0 p, q, r all positive

Since the exponential function is convex* we have

$$e^{\frac{r}{p} \ln \frac{|f|^p}{\|f\|_p^p} + \frac{r}{q} \ln \frac{|g|^q}{\|g\|_q^q}} \leq \frac{r}{p} \frac{|f|^p}{\|f\|_p^p} + \frac{r}{q} \frac{|g|}{\|g\|_q^q}$$

Integrating both sides from $-\infty$ to ∞ and using $\int |f|^p dx = \|f\|_p^p$

$$\frac{\int |fg|^r dx}{\|f\|_p^r \, \|g\|_q^r} \leq \frac{1}{p} + \frac{1}{q} = 1$$

$$\|fg\|_r \leq \|f\|_p \|g\|_q$$

Case 1 q < 0, p and r > 0

Put $f' = fg$, $g' = g^{-1}$, $p' = r$, $q' = -q$, $r' = p$ and apply 0

$$\|f'g'\|_{r'} \leq \|f'\|_{p'} \|g'\|_{q'}$$

$$\|f\|_p \leq \|fg\|_r \|g^{-1}\|_{-q}$$

$$\|fg\|_r \geq \|f\|_p \|g\|_q$$

(*) A function f is said to be convex iff for every x,y and p,q > 0 with p+q=1 it is true that $f(px + qy) \leq p\, f(x) + q\, f(y)$.

Case 2 p and r < 0, q > 0

Put $f' = g$, $g' = (fg)^{-1}$, $p' = q$, $q' = -r$, $r' = -p$ and apply

$\|f'g'\|_{r'} \leq \|f'\|_{p'} \|g'\|_{q'}$

$\|f^{-1}\|_{-p} \leq \|g\|_q \|(fg)^{-1}\|_{-r}$

$\|fg\|_r \leq \|f\|_p \|g\|_q$

Case 3 p,q,r all negative

Put $f' = f^{-1}$, $g' = g^{-1}$, $p' = -p$, $q' = -q$, $r' = -r$ and apply O

$\|f'g'\|_{r'} \leq \|f'\|_{p'} \|g'\|_{q'}$

$\|(fg)^{-1}\|_{-r} \leq \|f^{-1}\|_{-p} \|g^{-1}\|_{-q}$

$\|fg\|_r \geq \|f\|_p \|g\|_q$

In 1,2 and 3 we have used the fact that

$\|f^{-1}\|_{-p} = \|f\|_p^{-1}$

The equality statement is easily verified by direct calculat

Proof of (4.3)

To maximize the SDR measure the following integral should be minimized

$$\int_{-\infty}^{\infty} p(x)\, g'(x)^{-2} dx$$

under the condition

$$\int_{-\infty}^{\infty} g'(x)\, dx = g(\infty) - g(-\infty) = 2R$$

one elegantly uses the Hölder inequality with

$f = p(x)$, $g = g'(x)^{-2}$, $p=1/3$, $q=-1/2$ and $r=1$:

$$\int p(x)\, g'(x)^{-2}\, dx \geq \left(\int p^{1/3} dx \right)^3 * \left(\int g'\, dx \right)^{-2} =$$

$$\left(\int p^{1/3} dx \right)^3 (2R)^{-2}$$

The SDR will be maximized when equality occurs here, that is

$g'(x) = \text{const} * p(x)^{1/3}$

Appendix 3

The Program Convert

```
PROGRAM CONVERT(input,output);
{
Author:  Bo Bernhardsson, ASEA RELAYS, April 1985

This program will convert numbers between the TMS format and
16 bits real numbers. The TMS arithmetic format should be left
point two's complementary. E.g. (E000)Hex = -0.75
Before conversion to the TMS form the real numbers will all
be scaled by a constant factor (an exponent of 2) to get into
the interval [-1,1).
Another way to see this is that all the TMS numbers
should be interpreted as mantissas in [-1,1) having the same
(not stored) exponent.
The numbers should be stored one at a line in ASCII-format.

answer = 0:
Infile contains real number
Outfile will contain hexadecimal numbers in TMS format

answer = 1:
infile contains hexadecimal numbers in TMS format
Outfile will contain real numbers
}

type string = varying [80] of char;

var infile,outfile: string;
    answer : integer;
    exp: real;

procedure conv(infile,outfile:string; exp:real; answer:integer);
const konst = 2**15;
var r:real;
    i : integer;
```

```
    ch : char;
    f,g : text;

function value(ch:char):integer;
begin
   if (ord('0')<= ord(ch)) and (ord(ch) <= ord('9')) then
      value := ord(ch)-ord('0')
   else
      value := ord(ch)-ord('A')+10
end;

begin {procedure conv}
  open (f,infile,old);
  open (g,outfile,new);
  reset(f);
  rewrite(g);

  if answer = 0 then {real -> tms}
  begin
    while not ( eof(f) ) do
    begin
       readln(f,r);
       r := r*2**(-exp);
       if r<-1 then r:= -1;
       if r>=1  then r:= 1-1/konst;
       if r<0  then r := 2+r;
       i := round(r*konst);
       writeln(g,hex(i,4));
    end;
  end

  else if answer = 1 then {tms -> real}
  begin
    while not ( eof(f) ) do
    begin
       r:=0;
       repeat
          read(f,ch);
       until ch<>' ';
```

```
        while ch<>' ' do
        begin
          r:=16*r+value(ch);
          read(f,ch);
        end;
        r:=r/konst;
        if r>=1 then r:=r-2;
        if (r<-1) or (r>=1) then write('****** error : r = ',r);
        r:=r*2**exp;
        writeln(g,r);
      end;
    end; {if}
    close(f);
    close(g);
end; {conv}

begin {main}
    writeln('************************************************************
    writeln('This program converts between tms format and real numbe
    repeat
       writeln;
       writeln('real->tms  : 0');
       writeln('tms ->real : 1');
       writeln('quit       : 2');
       writeln;
       write('> ');
       readln(answer);
       if answer in [0,1] then
       begin
          writeln;
          write('infile :'); readln(infile);
          write('outfile :'); readln(outfile);
          write('exponent(0)'); readln(exp);
          conv(infile,outfile,exp,answer);
          writeln('ok');
       end;
    until answer=2;
end.
```

Appendix 4

The Program Codegen.pas

```
program CODEGEN (kod,input,output);

{ This program creates assembly code for the TMS 32010 }
{ VAX/VMS version 3.4 pascal is used.                   }
{ The main program should reside in userscode.pas       }

const t0 = 1;                          {tabulator positions  }
      t1 = 10;
      t2 = 16;
      t3 = 30;
      defaultname = '[bob.exjobb.demo]outfile.asm';

type identifier = varying [80] of char;

var  outfile: identifier;          {name of assembly output file}
     kod    : text;                {text file for assembly code }
     col    : integer;             {column pointer              }

%include 'proc.pas'                {includes procedures needed  }
                                   {for the code generation     }

%include 'userscode.pas'           {this should be a file       }
                                   {containing the users code-  }
                                   {generator. The file should  }
                                   {start: procedure userscode; }

{ ---------------------------------------------------------- }
{ ---------------------------------------------------------- }

begin {main}
   writeln('The user program should be in userscode.pas');
   write('Outfile ('+defaultname+') : '); readln(outfile);
   if outfile.length=0 then outfile := defaultname;
   open (kod,outfile,new);
```

```
    rewrite(kod);
    col := 1;
    userscode;                  { The code generator  for the }
                                { desired algorithm should be }
                                { in userscode.pas in the form}
                                { of a procedure.             }
    close(kod);
    write('ok');
end.
```

Proc.pas

```
{ ------------------------------------------------------------}
{ This section contains                                       }
{ procedures to create an assembler file for the TMS 32010.   }
{ It should be included in the codegen.pas program by         }
{ %include 'proc.pas'. Note that capital letters must be used }

procedure w(a:identifier);
begin
   write(kod,a);
   col:=col+a.length;
end;

procedure wln;
begin
   writeln(kod);
   col:=t0;
end;

procedure tab1;
var i:integer;
begin
   for i:=1 to t1-col do w(' ');
   col :=t1;
end;

procedure tab2;
var i:integer;
begin
   for i:=1 to t2-col do w(' ');
   col := t2;
end;

procedure tab3;
var i:integer;
begin
   for i:=1 to t3-col do w(' ');
```

```
   col := t3;
end;

function c(i:integer) : identifier;
(converts the integer i to an identifier
 ex 'SAMPEL'+c(13) = SAMPEL13    )
var a:identifier ;
begin
   a := '';
   if i>=10 then
   begin
      a:=a + c(i div 10);
      a:=a + c(i mod 10);
   end
   else if i>=0 then
      a:=a + chr(i + ord('0'));
   c := a;
end;

procedure comment ( a:identifier := ''; tab:integer := t3);
var i:integer;
begin
   w('* ');
   for i:=1 to tab-col do w(' ');
   w(a);
   wln;
end;

{-------------------------------------------------------------}
{ This section contains                                       }
{ procedures to create assembler directives                   }

procedure bss( a:identifier ; i:integer := 1);
begin
   w(a);
   tab1;
   w('BSS');
   tab2;
   w(c(i));
```

```
   wln;
end;

procedure data (a:identifier );
begin
   tab1;
   w('DATA');
   tab2;
   w('>');
   w(a);
   wln;
end;

procedure def(a:identifier);
begin
   tab1;
   w('DEF');
   tab2;
   w(a);
   wln;
end;

procedure dend;
begin
   tab1;
   w('DEND');
   wln;
end;

procedure dseg;
begin
   tab1;
   w('DSEG');
   wln;
end;

procedure end;
begin
   tab1;
```

```
    w('END');
    wln;
end;

procedure idt(a:identifier);
begin
    tab1;
    w('IDT');
    tab2;
    w('''');
    w(a);
    w('''');
    wln;
end;

procedure pseg;
begin
    tab1;
    w('PSEG');
    wln;
end;



{-----------------------------------------------------------------
{ This section contains procedures for creating assembler
{ instructions for the TMS 32010

procedure abs ;
begin
    tab1;
    w('ABS');
    wln;
end;

procedure add (a:identifier;i:integer:=0);
begin
    tab1;
    w('ADD');
```

```
    tab2;
    w(a);
    w(',');
    w(c(i));
    wln;
end;

procedure addh (a:identifier);
begin
    tab1;
    w('ADDH');
    tab2;
    w(a);
    wln;
end;

procedure adds (a:identifier);
begin
    tab1;
    w('ADDS');
    tab2;
    w(a);
    wln;
end;

procedure and (a:identifier);
begin
    tab1;
    w('AND');
    tab2;
    w(a);
    wln;
end;

procedure apac ;
begin
    tab1;
    w('APAC');
    wln;
```

```
end;


procedure b (a:identifier);
begin
   tab1;
   w('B');
   tab2;
   w(a);
   wln;
end;

procedure banz (a:identifier);
begin
   tab1;
   w('BANZ');
   tab2;
   w(a);
   wln;
end;

procedure bgez (a:identifier);
begin
   tab1;
   w('BGEZ');
   tab2;
   w(a);
   wln;
end;

procedure bgz (a:identifier);
begin
   tab1;
   w('BGZ');
   tab2;
   w(a);
   wln;
end;
```

```
procedure bioz (a:identifier);
begin
   tab1;
   w('BIOZ');
   tab2;
   w(a);
   wln;
end;

procedure blez (a:identifier);
begin
   tab1;
   w('BLEZ');
   tab2;
   w(a);
   wln;
end;

procedure blz (a:identifier);
begin
   tab1;
   w('BLZ');
   tab2;
   w(a);
   wln;
end;

procedure bnz (a:identifier);
begin
   tab1;
   w('BNZ');
   tab2;
   w(a);
   wln;
end;

procedure bv (a:identifier);
begin
   tab1;
```

```
    w('BV');
    tab2;
    w(a);
    wln;
end;

procedure bz (a:identifier);
begin
    tab1;
    w('BZ');
    tab2;
    w(a);
    wln;
end;

procedure cala ;
begin
    tab1;
    w('CALA');
    wln;
end;

procedure call (a:identifier);
begin
    tab1;
    w('CALL');
    tab2;
    w(a);
    wln;
end;

procedure dint ;
begin
    tab1;
    w('DINT');
    wln;
end;

procedure dmov (a:identifier);
```

```
begin
   tab1;
   w('DMOV');
   tab2;
   w(a);
   wln;
end;

procedure eint ;
begin
   tab1;
   w('EINT');
   wln;
end;

procedure in (a,b:identifier);
begin
   tab1;
   w('IN');
   tab2;
   w(a);
   w(',');
   w(b);
   wln;
end;

procedure lac (a:identifier ;i:integer:=0);
begin
   tab1;
   w('LAC');
   tab2;
   w(a);
   w(',');
   w(c(i));
   wln;
end;

procedure lack (i:integer);
begin
```

```
    tab1;
    w('LACK');
    tab2;
    w(c(i));
    wln;
end;

procedure lar (a,b:identifier);
begin
    tab1;
    w('LAR');
    tab2;
    w(a);
    w(',');
    w(b);
    wln;
end;

procedure lark (a,b:identifier);
begin
    tab1;
    w('LARK');
    tab2;
    w(a);
    w(',');
    w(b);
    wln;
end;

procedure larp (i:integer);
begin
    tab1;
    w('LARP');
    tab2;
    w(c(i));
    wln;
end;

procedure ldp (a:identifier );
```

```
begin
   tab1;
   w('LDP');
   tab2;
   w(a);
   wln;
end;

procedure ldpk (i:integer);
begin
   tab1;
   w('LDPK');
   tab2;
   w(c(i));
   wln;
end;

procedure lst (a:identifier );
begin
   tab1;
   w('LST');
   tab2;
   w(a);
   wln;
end;

procedure lt (a:identifier );
begin
   tab1;
   w('LT');
   tab2;
   w(a);
   wln;
end;

procedure lta (a:identifier );
begin
   tab1;
   w('LTA');
```

```
    tab2;
    w(a);
    wln;
end;

procedure ltd (a:identifier );
begin
    tab1;
    w('LTD');
    tab2;
    w(a);
    wln;
end;

procedure mar (a:identifier;i:integer);
begin
    tab1;
    w('MAR');
    tab2;
    w(a);
    w(',');
    w(c(i));
    wln;
end;


procedure mpy (a:identifier);
begin
    tab1;
    w('MPY');
    tab2;
    w(a);
    wln;
end;

procedure mpyk (i:integer);
begin
    tab1;
    w('MPYK');
```

```
    tab2;
    w(c(i));
    wln;
end;

procedure nop;
begin
    tab1;
    w('NOP');
    wln;
end;

procedure or (a:identifier);
begin
    tab1;
    w('OR');
    tab2;
    w(a);
    wln;
end;

procedure out (a,b:identifier);
begin
    tab1;
    w('OUT');
    tab2;
    w(a);
    w(',');
    w(b);
    wln;
end;

procedure pac ;
begin
    tab1;
    w('PAC');
    wln;
end;
```

```
procedure pop ;
begin
   tab1;
   w('POP');
   wln;
end;

procedure push ;
begin
   tab1;
   w('PUSH');
   wln;
end;

procedure ret ;
begin
   tab1;
   w('RET');
   wln;
end;

procedure rovm ;
begin
   tab1;
   w('ROVM');
   wln;
end;

procedure sach (a:identifier ; i:integer :=0);
begin
   if not (i in [0,1,4]) then
      comment('error sach '+c(i));
   tab1;
   w('SACH');
   tab2;
   w(a);
   w(',');
   w(c(i));
   wln;
```

```
end;

procedure sacl (a:identifier);
begin
   tab1;
   w('SACL');
   tab2;
   w(a);
   wln;
end;

procedure sar (a,b:identifier);
begin
   tab1;
   w('SAR');
   w(a);
   w(',');
   w(b);
   wln;
end;

procedure sovm ;
begin
   tab1;
   w('SOVM');
   wln;
end;

procedure spac ;
begin
   tab1;
   w('SPAC');
   wln;
end;

procedure sst (a:identifier);
begin
   tab1;
   w('SST');
```

```
   tab2;
   w(a);
   wln;
end;

procedure sub (a:identifier;i:integer:=0);
begin
   tab1;
   w('SUB');
   tab2;
   w(a);
   w(',');
   w(c(i));
   wln;
end;

procedure subc (a:identifier);
begin
   tab1;
   w('SUBC');
   tab2;
   w(a);
   wln;
end;

procedure subh (a:identifier);
begin
   tab1;
   w('SUBH');
   tab2;
   w(a);
   wln;
end;

procedure subs (a:identifier);
begin
   tab1;
   w('SUBS');
   tab2;
```

```
    w(a);
    wln;
end;

procedure tblr (a:identifier);
begin
    tab1;
    w('TBLR');
    tab2;
    w(a);
    wln;
end;

procedure tblw (a:identifier);
begin
    tab1;
    w('TBLW');
    tab2;
    w(a);
    wln;
end;

procedure xor (a:identifier);
begin
    tab1;
    w('XOR');
    tab2;
    w(a);
    wln;
end;

procedure zac ;
begin
    tab1;
    w('ZAC');
    wln;
end;

procedure zalh (a:identifier);
```

```
begin
   tab1;
   w('ZALH');
   tab2;
   w(a);
   wln;
end;

procedure zals (a:identifier);
begin
   tab1;
   w('ZALS');
   tab2;
   w(a);
   wln;
end;

{-------------------------------------------------------------}
{ procedures for (extended) assembler macros                  }
{ when using macros the main program has to be started with   }
{ the assembler directive 'main'.                             }

procedure macrouse;
begin
   pseg;
   lack(1);
   sacl('ONE');
   zac;
   sub('ONE');
   sacl('MINUS');
   dseg;
   bss('ONE');
   bss('MINUS');
   bss('XR0');
   bss('XR1');
   bss('XR2');
   bss('XR3');
   def('ONE');
   def('MINUS');
```

```
   def('XR0');
   def('XR1');
   def('XR2');
   def('XR3');
   dend;
end;

procedure initdata(dma:identifier; i:integer);
{loads dataram dma with the decimal constant i}
begin
   lack(i div 256);
   sacl('XR0');
   lack(i mod 256);
   add('XR0',8);
   sacl(dma);
end;

procedure mov(a,b:identifier);
{moves a to b in data memory}
begin
   lac(a);
   sacl(b);
end;

procedure negx;
{negates accumulators 32 bits}
begin
   sach ('XR0',0);
   sacl ('XR1');
   zac;
   subh ('XR0');
   subs ('XR1');
end;

procedure not;
{inverts lower 16 bits of accumulator}
begin
   xor('MINUS');
end;
```

```
procedure lasx(a,b:identifier; i:integer);
{ arithmetic left shift (0 - 15) of a and a+1 }
{ and store in b and b+1 }
begin
   lac(a+'+1',i);
   sacl (b+'+1');
   sach(b,0);
   lac ('MINUS',i);
   not;
   and (b);
   add (a,i);
   sacl(b);
end;

procedure rlsh(a,b:identifier; i:integer);
{logical left shift (0-15) of accumulators 32 bits}
begin
   lac (a,16-i);
   sach (b,0);
   lac ('MINUS',16-i);
   not;
   and (b);
   sacl(b);
end;

procedure rasx(a,b:identifier; i:integer);
{Arithmetic right shift (0-15) of accumulators 32 bits}
begin
   rlsh (a+'+1',b+'+1',i);
   lac (a,16-i);
   sach (b,0);
   or (b+'+1');
   sacl (b+'+1');
end;

procedure sache (a:identifier; i:integer);
{Extended SACH; left shifts of -16 to 31 possible}
{The accumulator will be changed }
begin
```

```
   if (i<-16) or (i>31) then
      comment(' ERROR : SACH WITH SHIFT OF '+c(i)+' ATTEMPTED')
   else if i<0 then
   begin
      sach('XR0',0);
      lac('XR0',16+i);
      sach(a);
   end
   else if i in [0,1,4] then
      sach (a,i)
   else  if i<16 then
   begin
      sach('XR0',0);
      sacl('XR1');
      lasx('XR0','XR2',i);
      zalh('XR2');
      adds('XR3');
      sach(a);
   end
   else if i=16 then
      sacl(a)
   else
   begin
      sacl('XR1');
      lac('XR1',i-16);
      sacl(a);
   end;
end;

procedure scalarproduct(a,b:identifier; na:integer);
{ acc:= a[1]*b[1] + ... a[na]*b[na] }
var j:integer;
begin
   zac;
   lt(a+c(1));
   mpy(b+c(1));
   for j:=2 to na do
   begin
      lta(a+c(j));
```

```
      mpy(b+c(j));
   end;
   apac;
end;

procedure mpyv(a,b:identifier; prod:identifier :=''; na:integer :=1)
{ prod[i]:= a[i]*b[i] ...  i:=1 to na; all scaling = -15 }
var j:integer;
begin
   for j:=1 to na do
   begin
      lt (a+c(j));
      mpy(b+c(j));
      pac;
      sach(prod+c(j),1);
   end;
end;

procedure bssv(a:identifier; na:integer);
{ Declares a vector a[1]...a[na] }
{ na = number of vector elements }
var j:integer;
begin
   for j:=1 to na do
       bss(a+c(j));
end;

procedure dmovv(a:identifier; na:integer);
{ Moves the vector a[1]...a[na] one step }
var j : integer;
begin
   for j:=na-1 downto 1 do
      dmov(a+c(j));
end;

procedure addv(a,b,sum: identifier; na:integer:=1);
{ sum[i] := a[i] + b[i] for i:=1 to na       }
var j : integer;
begin
```

```
   for j:=1 to na do
   begin
      lac(a+c(j));
      add(b+c(j));
      sach(sum+c(j));
   end;
end;


procedure addsc(a:identifier;
                ea:integer;
                 b:identifier;
                eb:integer;
                 c:identifier;
                ec:integer    );

{adds two numbers with different scaling factors ea and eb  }
{the result is in c and is scaled ec                        }
{eb >= ea otherwise swap }
var dum1 :identifier;
    dum2 :integer;
begin
   if eb<ea then
   begin
      dum1:=a;
      a:=b;
      b:=dum1;
      dum2:=ea;
      ea:=eb;
      eb:=dum2;
   end;
   if (ec-ea>16) or (eb-ec>16) then
      comment('error in addsc');
   if eb<=ec then
   begin
      lac(a,16-(ec-ea));
      add(b,16-(ec-eb));
      sach(c);
   end
```

```
   else if ec<=ea then
   begin
      lac(a,ea-ec);
      add(b,eb-ec);
      sacl(c);
   end
   else
   begin
      lac(a,ea-eb+16);
      addh(b);
      sache(c,eb-ec);  {this required extended sach}
   end;
end;

procedure mulsc(a,b,c:identifier; ea,eb,ec:integer);
{multiplies scaled numbers}
begin
   if (-16<= ea+eb-ec) and (ea+eb-ec <=0) then
   begin
      lt(a);
      mpy(b);
      pac;
      sache(c,16+ea+eb-ec);
   end;
   {note the restriction on the scalingfactors}
end;
```

```
procedure userscode;
{ This main program will generate code for parameter estimation
  using the MIT-rule for an arbitrary number of coefficients in
  an ARMAX-model. It uses CODEGEN.PAS and PROC.PAS.              }

const  inportY = 0;
       inportU = 1;
       outport = 0;
       scale   = 2**15;

var    na      : integer;
       nb      : integer;
       i       : integer;
       theexp  : integer;
       pin     : integer;
       preal   : real;
begin
   writeln('*****************************************************');
   writeln('This program will generate code for the TMS processor');
   writeln('for fast parameter estimation of an arbitrary number');
   writeln('of parameters in an ARMAX-model.');
   writeln('The MIT-rule is used and the scalar P should be ')
   writeln('given by the user');
   writeln('*****************************************************');
   write ('na = '); readln (na);
   write ('nb = '); readln (nb);
   write ('scalefactor for a and b =(-15): '); readln (theexp);
   write ('adaption factor p [0,1) : '); readln (preal);
   pin := trunc(preal*scale);
   writeln(pin);

   idt('OUTFILE');
   comment('*******************************************',t0);
   comment(' IDENTIFIES A SYSTEM AY = BU + E ',t0);
   comment(' WITH '+c(na)+' A- AND '+c(nb)+' B-PARAMETERS',t0);
   comment('*******************************************',t0);
```

```
macrouse;
comment('MACRO FACILITIES AVAILABLE');
comment();

dseg;
bssv('FI',na+nb);
bssv('THE',na+nb);
bss ('P');
bss ('EPS');
bss ('PEPS');
bss('Y');
dend;
comment();

initdata('P',pin);

w('SAMPEL');
in('Y',c(inportY));
in('FI'+c(na+1),c(inportU));
comment('READ Y AND U');

scalarproduct('THE','FI',na+nb);
sub('Y',-theexp);
negx;
comment('Y-THE*FI');
sache('EPS',16+theexp);
comment('EPS := Y - THE*FI');

mulsc ('P','EPS','PEPS',-15,-15,theexp);
comment('PEPS := P*EPS');

lt('PEPS');
for i:=1 to na+nb do
begin
   mpy('FI'+c(i));
   pac;
   add('THE'+c(i),15);
   sach('THE'+c(i),1);
   comment('THE'+c(i)+' := THE'+c(i)+' + PEPS*FI'+c(i));
```

```
    end;

    for i:= 1 to na+nb do
       out('THE'+c(i),c(outport));
    comment('OUTPUT ESTIMATED PARAMETERS');

    dmovv('FI',na+nb);
    if na>0 then
       mov('Y','FI1');
    comment('DELAY MOVE THE FI-VECTOR');

    b('SAMPEL');
    comment('NEW SAMPEL');
    end;
end;
```

```
          IDT   'OUTFILE'
* ***********************************************
*  IDENTIFIES A SYSTEM AY = BU + E
*  WITH 1 A- AND 1 B-PARAMETERS
* ***********************************************
          PSEG
          LACK  1
          SACL  ONE
          ZAC
          SUB   ONE,0
          SACL  MINUS
          DSEG
ONE       BSS   1
MINUS     BSS   1
XR0       BSS   1
XR1       BSS   1
XR2       BSS   1
XR3       BSS   1
          DEF   ONE
          DEF   MINUS
          DEF   XR0
          DEF   XR1
          DEF   XR2
          DEF   XR3
          DEND
*                              MACRO FACILITIES AVAILABLE
*
          DSEG
FI1       BSS   1
FI2       BSS   1
THE1      BSS   1
THE2      BSS   1
P         BSS   1
EPS       BSS   1
PEPS      BSS   1
Y         BSS   1
```

```
          DEND
*
          LACK   102
          SACL   XRO
          LACK   102
          ADD    XRO,8
          SACL   P
SAMPEL    IN     Y,0
          IN     FI2,1
*                                READ Y AND U
          ZAC
          LT     THE1
          MPY    FI1
          LTA    THE2
          MPY    FI2
          APAC
          SUB    Y,15
          SACH   XRO,0
          SACL   XR1
          ZAC
          SUBH   XRO
          SUBS   XR1
*                                Y-THE*FI
          SACH   EPS,1
*                                EPS := Y - THE*FI
          LT     P
          MPY    EPS
          PAC
          SACH   PEPS,1
*                                PEPS := P*EPS
          LT     PEPS
          MPY    FI1
          PAC
          ADD    THE1,15
          SACH   THE1,1
*                                THE1 := THE1 + PEPS*FI1
          MPY    FI2
          PAC
          ADD    THE2,15
```

```
        SACH   THE2,1
*                              THE2 := THE2 + PEPS*FI2
        OUT    THE1,0
        OUT    THE2,0
*                              OUTPUT ESTIMATED PARAMETERS
        DMOV   FI1
        LAC    Y,0
        SACL   FI1
*                              DELAY MOVE THE FI-VECTOR
        B      SAMPEL
*                              NEW SAMPEL
        END
```

Appendix 5

The Program Codegen.pro

```
/*-------------------------------------------------------------*/
/* This is a prolog version of the codegenerator               */
/* which implements recursive parameter estimation.            */
/* The outfile is an assembly program for the TMS 32020        */


appendstring(X,Y,Z) :-
/* appends X and Y to Z */
   name(X,L),
   name(Y,M),
   append(L,M,N),
   name(Z,N).

append([], X, X).
append([A|B], C, [A|D]):-append(B, C, D).

w(X) :-
   write(X).

wln :-
/* new line */
   nl,
   put(13).

tab1 :-
/* prints some blanks */
   tab(10).

tab2 :-
   tab(4).

comment(Comm) :-
/* to write comments in the assembly file */
   w('*                                  '),
```

```
    w(Comm),
    wln.

/*-------------------------------------------------------------*/
/* This section contains                                       */
/* procedures to create assembler directives                   */

bss(X,N) :-
    w(X),
    tab1,
    w('BSS '),
    tab2,
    w(N),
    wln.

data(A) :-
    tab1,
    w('DATA'),
    tab2,
    w('>'),
    w(A),
    wln.


def(A) :-
    tab1,
    w('DEF '),
    tab2,
    w(A),
    wln.

dend :-
    tab1,
    w('DEND'),
    wln.

dseg :-
    tab1,
    w('DSEG'),
```

```
    wln.

end_ :-
    tab1,
    w('END '),
    wln.

idt(A) :-
    tab1,
    w('IDT '),
    tab2,
    w(''''),
    w(A),
    w(''''),
    wln.

pseg :-
    tab1,
    w('PSEG'),
    wln.


/*-------------------------------------------------------------------
/* This section contains procedures for creating assembler
/* instructions for the TMS 32010
/* All instructions are implemented. Indirect addressing (which is
/* done using AR0 and AR1) is not implemented since it was not neede

abs  :-
    tab1,
    w('ABS '),
    wln.


add(A,N) :-
    tab1,
    w('ADD '),
    tab2,
    w(A),
```

```
    w(','),
    w(N),
    wln.

addh(A) :-
    tab1,
    w('ADDH'),
    tab2,
    w(A),
    wln.

adds(A) :-
    tab1,
    w('ADDS'),
    tab2,
    w(A),
    wln.

and_(A) :-
    tab1,
    w('AND '),
    tab2,
    w(A),
    wln.

apac  :-
    tab1,
    w('APAC'),
    wln.

b(A) :-
    tab1,
    w('B   '),
    tab2,
    w(A),
    wln.

banz(A) :-
    tab1,
```

```
    w('BANZ'),
    tab2,
    w(A),
    wln.

bgez(A) :-
    tab1,
    w('BGEZ'),
    tab2,
    w(A),
    wln.

bgz(A) :-
    tab1,
    w('BGZ '),
    tab2,
    w(A),
    wln.

bioz(A) :-
    tab1,
    w('BIOZ'),
    tab2,
    w(A),
    wln.

blez(A) :-
    tab1,
    w('BLEZ'),
    tab2,
    w(A),
    wln.

blz(A) :-
    tab1,
    w('BLZ '),
    tab2,
    w(A),
    wln.
```

```
bnz(A) :-
    tab1,
    w('BNZ '),
    tab2,
    w(A),
    wln.

bv(A) :-
    tab1,
    w('BV  '),
    tab2,
    w(A),
    wln.

bz(A) :-
    tab1,
    w('BZ  '),
    tab2,
    w(A),
    wln.

cala  :-
    tab1,
    w('CALA'),
    wln.

call_(A) :-
    tab1,
    w('CALL'),
    tab2,
    w(A),
    wln.

dint  :-
    tab1,
    w('DINT'),
    wln.

dmov(A) :-
```

```
    tab1,
    w('DMOV'),
    tab2,
    w(A),
    wln.

eint  :-
    tab1,
    w('EINT'),
    wln.

in_(A,B) :-
    tab1,
    w('IN  '),
    tab2,
    w(A),
    w(','),
    w(B),
    wln.

lac(A,N) :-
    tab1,
    w('LAC '),
    tab2,
    w(A),
    w(','),
    w(N),
    wln.

lack(N):-
    tab1,
    w('LACK'),
    tab2,
    w(N),
    wln.

lar(A,B) :-
    tab1,
    w('LAR '),
```

```
    tab2,
    w(A),
    w(', '),
    w(B),
    wln.

lark(A,B) :-
    tab1,
    w('LARK'),
    tab2,
    w(A),
    w(', '),
    w(B),
    wln.

larp(N):-
    tab1,
    w('LARP'),
    tab2,
    w(N),
    wln.

ldp(A) :-
    tab1,
    w('LDP '),
    tab2,
    w(A),
    wln.

ldpk(N):-
    tab1,
    w('LDPK'),
    tab2,
    w(N),
    wln.

lst(A) :-
    tab1,
    w('LST '),
```

```
    tab2,
    w(A),
    wln.

lt(A) :-
    tab1,
    w('LT   '),
    tab2,
    w(A),
    wln.

lta(A) :-
    tab1,
    w('LTA  '),
    tab2,
    w(A),
    wln.

ltd(A) :-
    tab1,
    w('LTD  '),
    tab2,
    w(A),
    wln.

mar(A,N) :-
    tab1,
    w('MAR  '),
    tab2,
    w(A),
    w(','),
    w(N),
    wln.

mpy(A) :-
    tab1,
    w('MPY  '),
    tab2,
    w(A),
```

```
    wln.

mpyk(N):-
    tab1,
    w('MPYK'),
    tab2,
    w(N),
    wln.

nop :-
    tab1,
    w('NOP '),
    wln.

or_(A) :-
    tab1,
    w('OR  '),
    tab2,
    w(A),
    wln.

out_(A,B) :-
    tab1,
    w('OUT '),
    tab2,
    w(A),
    w(','),
    w(B),
    wln.

pac  :-
    tab1,
    w('PAC '),
    wln.

pop  :-
    tab1,
    w('POP '),
    wln.
```

```
push  :-
   tab1,
   w('PUSH'),
   wln.

ret  :-
   tab1,
   w('RET '),
   wln.

rovm  :-
   tab1,
   w('ROVM'),
   wln.

sach(A,N) :-
   (N=0 ; N=1 ; N=4),
   tab1,
   w('SACH'),
   tab2,
   w(A),
   w(','),
   w(N),
   wln.

sach(_,_) :-
   comment('ERROR IN SACH').

sacl(A) :-
   tab1,
   w('SACL'),
   tab2,
   w(A),
   wln.

sar(A,B) :-
   tab1,
   w('SAR '),
   w(A),
```

```
    w(', '),
    w(B),
    wln.

sovm  :-
    tab1,
    w('SOVM'),
    wln.

spac  :-
    tab1,
    w('SPAC'),
    wln.

sst(A) :-
    tab1,
    w('SST '),
    tab2,
    w(A),
    wln.

sub(A,N) :-
    tab1,
    w('SUB '),
    tab2,
    w(A),
    w(', '),
    w(N),
    wln.

subc(A) :-
    tab1,
    w('SUBC'),
    tab2,
    w(A),
    wln.

subh(A) :-
    tab1,
```

```
    w('SUBH'),
    tab2,
    w(A),
    wln.

subs(A) :-
    tab1,
    w('SUBS'),
    tab2,
    w(A),
    wln.

tblr(A) :-
    tab1,
    w('TBLR'),
    tab2,
    w(A),
    wln.

tblw(A) :-
    tab1,
    w('TBLW'),
    tab2,
    w(A),
    wln.

xor(A) :-
    tab1,
    w('XOR '),
    tab2,
    w(A),
    wln.

zac :-
    tab1,
    w('ZAC '),
    wln.

zalh(A) :-
```

```
    tab1,
    w('ZALH'),
    tab2,
    w(A),
    wln.

zals(A) :-
    tab1,
    w('ZALS'),
    tab2,
    w(A),
    wln.

/*-------------------------------------------------------------*/
/* Procedures for (extended) assembler macros                  */
/* When using macros the main program has to be started with   */
/* the macro macrouse.                                         */

macrouse :-
    pseg,
    lack(1),
    sacl('ONE'),
    zac,
    sub('ONE',0),
    sacl('MINUS'),
    dseg,
    bss('ONE',1),
    bss('MINUS',1),
    bss('XR0',1),
    bss('XR1',1),
    bss('XR2',1),
    bss('XR3',1),
    def('ONE'),
    def('MINUS'),
    def('XR0'),
    def('XR1'),
    def('XR2'),
    def('XR3'),
    dend.
```

```
initdata(Dma,N) :-
/*loads dataram Dma with the decimal constant N*/
   N1 is N // 256,
   lack(N1),
   sacl('XRO'),
   N2 is (N mod 256),
   lack(N2),
   add('XRO',8),
   sacl(Dma).

mov(A,B) :-
/*moves a to b in data memory*/
   lac(A,0),
   sacl(B).

negx :-
/*negates accumulators 32 bits*/
   sach('XRO',0),
   sacl('XR1'),
   zac,
   subh('XRO'),
   subs('XR1').

not_ :-
/*inverts lower 16 bits of accumulator*/
   xor('MINUS').

lasx(A,B,N) :-
/*arithmetic left shift (0 - 15) of accumulators 32 bits*/
   appendstring(A,'+1',A1),
   lac(A1,N),
   appendstring(B,'+1',B1),
   sacl(B1),
   sach(B,0),
   lac('MINUS',N),
   not_,
   and_(B),
   add(A,N),
   sacl(B).
```

```
rlsh(A,B,N) :-
/*logical left shift (0-15) of accumulators 32 bits*/
   N1 is 16-N,
   lac(A,N1),
   sach(B,0),
   lac('MINUS',N1),
   not_,
   and_(B),
   sacl(B).

rasx(A,B,N) :-
/*Arithmetic right shift (0-15) of accumulators 32 bits*/
   appendstring(A,'+1',A1),
   appendstring(B,'+1',B1),
   rlsh(A1,B1,N),
   N1 is 16-N,
   lac(A,N1),
   sach(B,0),
   or_(B1),
   sacl(B1).

sache(A,I) :-
/* extended sach. shift of -16 to 31 possible */
   (I < -16 ; I > 31),
   comment(' ERROR : SACH WITH ILLEGAL SHIFT ').

sache(A,I) :-
   I < 0,
   sach('XR0',0),
   I1 is 16+I,
   lac('XR0',I1),
   sach(A,0).

sache(A,I) :-
   (I=0 ; I=1 ; I=4 ),
   sach(A,I).

sache(A,I) :-
   I < 16,
```

```
    sach('XR0',0),
    sacl('XR1'),
    lasx('XR0','XR2',I),
    zalh('XR2'),
    adds('XR3'),
    sach(A).

sache(A,I) :-
    I = 16,
    sacl(A).

sache(A,I) :- /* I>16 */
    sacl('XR1'),
    I1 is I-16,
    lac('XR1',I1),
    sacl(A).


scalarproduct(A,B,N) :-
/* acc:= a[1]*b[1] + ... a[n]*b[n] */
    zac,
    appendstring(A,'1',A1),
    appendstring(B,'1',B1),
    lt(A1),
    mpy(B1),
    loop1(A,B,N),
    apac.

loop1(_,_,1) :- !.
loop1(A,B,N) :-
    N1 is N-1,
    loop1(A,B,N1),
    appendstring(A,N,A1),
    appendstring(B,N,B1),
    lta(A1),
    mpy(B1).


bssv(A,N) :-
```

```
/* Declares a vector a[1]...a[N] */
/* N = number of vector elements */
   loop2(A,N).

loop2(_,0) :- !.
loop2(A,N) :-
   N1 is N-1,
   loop2(A,N1),
   appendstring(A,N,A1),
   bss(A1,1).

dmovv(A,N) :-
/* Moves the vector a[1]...a[N] one step */
/* a[1] := a[2] ...; a[N] is not changed */
   N1 is N-1,
   loop3(A,N1).

loop3(_,0) :- !.
loop3(A,N) :-
   appendstring(A,N,A1),
   dmov(A1),
   N1 is N-1,
   loop3(A,N1).

mulsc(A,EA,B,EB,C,EC) :-
/*multiplies scaled numbers*/
/* C := A*B ; EA,EB and EC */
/* is the number of bits   */
/* that A,B and C should be*/
/* scaled                  */
   lt(A),
   mpy(B),
   pac,
   N is 16+EA+EB-EC,
   sache(C,N).

updatetheta(N) :-
/* THE := THE + PEPS*FI */
   lt('PEPS'),
```

```
    loop4(N).

loop4(0) .

loop4(N) :-
    N1 is N-1,
    loop4(N1),
    appendstring('FI',N,FI1),
    mpy(FI1),
    pac,
    appendstring('THE',N,THE1),
    add(THE1,15),
    sach(THE1,1).

outv(_,0,_) .
outv(A,N,PORT) :-
/* The vector A is send to */
/* the outputport PORT     */
    N1 is N-1,
    outv(A,N1,PORT),
    appendstring(A,N,A1),
    out(A1,PORT).

movy(NA) :-
/* FI1 := Y */
    NA > 0,
    mov('Y','FI1').
movy(_).

/*-------------------------------------------------------------*/
/* Here follows Userscode                                      */
/*                                                             */

codegen(Outfile) :-
    write('NA = '),
    read(NA),
    nl,
    write('NB = '),
    read(NB),
```

```
nl,
write('Scalefactor for a and b (e.g. -15) = '),
read(Theexp),
nl,
write('Adaptionfactor (e.g. 0.8*2**15) = '),
read(Pin),
nl,

tell(Outfile),
idt('OUTFILE'),
macrouse,
dseg,
NAB is NA + NB,
bssv('FI',NAB),
bssv('THE',NAB),
bss('P',1),
bss('EPS',1),
bss('PEPS',1),
bss('Y',1),
dend,
initdata('P',Pin),

w('SAMPEL'),
    in_('Y',0),
    NA1 is NA + 1,
    appendstring('FI',NA1,FI1),
    in_(FI1,1),

    scalarproduct('THE','FI',NAB),
    Minustheexp is 0 - Theexp,
    sub('Y',Minustheexp),
    negx,
    Theexp1 is 16 + Theexp,
    sache('EPS',Theexp1),
    comment('EPS := Y - THE * FI'),
    mulsc('P',-15,'EPS',-15,'PEPS',Theexp),
    updatetheta(NAB),
    comment('THETA VECTOR UPDATED'),
```

```
    outv('THE',NAB,2),
    dmovv('FI',NAB),
    movy(NA),
b('SAMPEL'),
end_,
tell(user),
write('ok'),
told.
```

```
          IDT     'OUTFILE'
          PSEG
          LACK    1
          SACL    ONE
          ZAC
          SUB     ONE,0
          SACL    MINUS
          DSEG
ONE          BSS     1
MINUS          BSS     1
XR0          BSS     1
XR1          BSS     1
XR2          BSS     1
XR3          BSS     1
          DEF     ONE
          DEF     MINUS
          DEF     XR0
          DEF     XR1
          DEF     XR2
          DEF     XR3
          DEND
          DSEG
FI1          BSS     1
FI2          BSS     1
THE1          BSS     1
THE2          BSS     1
P           BSS     1
EPS          BSS     1
PEPS          BSS     1
Y           BSS     1
          DEND
          LACK    102
          SACL    XR0
          LACK    102
          ADD     XR0,8
          SACL    P
```

```
SAMPEL          IN      Y,0
                IN      FI2,1
                ZAC
                LT      THE1
                MPY     FI1
                LTA     THE2
                MPY     FI2
                APAC
                SUB     Y,15
                SACH    XR0,0
                SACL    XR1
                ZAC
                SUBH    XR0
                SUBS    XR1
                SACH    EPS,1
*                                   EPS := Y - THE * FI
                LT      P
                MPY     EPS
                PAC
                SACH    PEPS,1
                LT      PEPS
                MPY     FI1
                PAC
                ADD     THE1,15
                SACH    THE1,1
                MPY     FI2
                PAC
                ADD     THE2,15
                SACH    THE2,1
*                                   THETA VECTOR UPDATED
                OUT     THE1,2
                OUT     THE2,2
                DMOV    FI1
                LAC     Y,0
                SACL    FI1
                B       SAMPEL
                END
```

[1] Olbjer L.(1983): Tidsserieanalys. Dept. of Math. Statistics, LTH.

[2] Bergman S.(1984): Protective Relays in Power system control. Research and Development group, Asea Relays, Västerås, Sweden.

[3] Biermann G J (1977): Factorization Methods for Discrete Sequential Estimation. Academic Press, New York.

[4] Aström K.J., Eykhoff P. (1971): System Identification-A Survey. Wiley.

[5] Hägglund T. (1983): New Estimation Techniques for Adaptive control. CODEN:LUTFD2/(TFRT-1025)/1-120/(1983).

[6] Young (1984): Recursive Estimation and Time-series analysis.

[7] Texas Instrument, Information about the TMS 320.

[8] VAX-11 PASCAL Language Reference Manual Digital Equipment Corp. (1982):

[9] CTRL-C (1983): A language for the Computer-Aided Design of Multivariable Control Systems. System Control Technology, Palo Alto, USA.

[10] Aström K.J., Wittenmark B. Computer Controlled Systems, Prentice-Hall.

[11] Texas Instrument, TMS32010 User's Guide.

[12] Ljung L, Söderström T (1983): Theory and Practice of Recursive Identification. MIT Press, Cambridge.

[13] Texas Instruments (1984): TMS 32010 Analog Interface Board User's Guide (SPDR015).

[14] Oppenheim A.V., Schafer R.W. (1975): Digital Signal Processing. Prentice-Hall.

[15] Swartzlander E.E. Jr., Alexopoulos A.G. (1975): The sign/ logarithm number system. IEEE Trans. Comptrs, p. 1238-1242.

[16] Lee S.C., Edgar A.D. (1977): The FOCUS number system. IEEE Trans. Comptrs, p. 1167-1170.

[17] Åström K.J., Eykhoff P. (1971): System Identification-A Survey.

[18] Lee S.C., Edgar A.D. (1979): FOCUS Microcomputer Number System. Communications of the ACM, p. 166-177.

[19] Texas Instrument (1984): TMS 32010 Simulator Manual.

[20] Texas Instrument (1984): TMS 32010 Assembler Manual.

[21] Texas Instrument (1984): TMS32010 Evaluation Module User's Guide (SPRU005).

[22] Texas Instrument (1984): Link Editor, User's Guide.

[23] Åström, K.J. (1983). Theory and Applications of Adaptive Control - A Survey. Automatica, Vol. 19, No. 5.pp.471-486.

[24] Master Thesis (1984): Tosteberg A., Sjöholm K. Asea Robotics, Västerås, Sweden.

[25] Eriksson P., Salomonsson G. (1984): Kompendium i Digital Signalbehandling. Lunds Institute of Technology, Sweden.

[26] C-Prolog User's Manual,Version 1.5 (1984) Edinburgh Computer Aided Architectural Design

[27] Kuck : Structure of computers and computations.

[28] Atlanta Software (1984): DFPD User's Guide.

[29] Morris L.R. (1985): Digital Signal Processing Software, (Collected papers). DSPS Inc., Ottawa, Canada.

[30] Clocksin-Mellish: Programming in Prolog, Springer Verlag.

[31] Wieslander (1980): IDPAC Commands - User's Guide. CODEN: LUTFD2/(TFRT-3157)/1-108/(1980).

[32] Wittenmark B. (1984): Analysis and Design of Control Systems Using CTRL-C CODEN: LUTFD2/(TFRT-7272)/1-022/(1984)

[33] Åström K.J. (1982): A Simnon Tutorial CODEN: LUTFD2/(TFRT-3168)/1-52/(1982)

[34] Volkers, H. (1984). Schnelle Fouriertransformation mit TMS 320 als Coprozessor. Elektronik 11.

[35] Teja, E. (1984). High-speed signal processors. Computers & Electronics, April.

[36] Bursky, D. (1984) Digital signal-processing chips move off the designers wish list and into everyday use. Electronic Design, May 17.

[37] Loges, W. (1983). Schneller Digitaler Regler mit Signalprozessor. Elektronik 19.

[38] Gauda, U. (1984). Einfach und kostengunstig: Lösungen mit den TMS320-Peripherie-Baustein TMS32050/51. Elektronik 11.

[39] Deshpande et.al. (1973). Adaptive Control of Linear Stochastic Systems. Automatica pp 107-115.