A COMPARATIVE STUDY OF AVAILABLE DIGITAL SIGNAL PROCESSOR CHIPS

by

Carl Thomas Hardin
B.S., Electrical Engineering, University of Kentucky, 1986

A MASTER'S THESIS

Submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Approved by

Donald H Lenhert
Major Professor

Contents

List of Figures

List of Tables

1.0 Introduction

As microprocessors became faster and more powerful, their use in the digital signal processing area became possible. Such tasks as echo cancellation and spectrum analysis for low bandwidth situations could be implemented with a simple microprocessor based dedicated system. Desires by the engineering community to apply these same algorithms to higher bandwidth situations has prompted many microprocessor manufacturers to develop and market a new type of microprocessor, the digital signal processor.

The question now is if an engineer has an algorithm he wishes to implement in a microprocessor based system, which of these new and basically unfamiliar set of digital signal processors should he use ? He has many considerations when it is time for him to choose as well as many products to choose from. Together these two factors point to a need for a general guide for choosing such a processor. It will be the purpose of this paper to provide an objective presentation and evaluation of these processors.

This paper outlines the features of these new processors. First, the situations in which these processors are mainly used is discussed. Then the Digital Signal Processor (hereafter referred to as DSP's) is defined. Next the varieties available are listed and features

tabulated. Then the most promising processors are compared directly by implementing them to perform a standard algorithm.

The Algorithm that is developed for those processors is the Widrow's Algorithm Adaptive Linear Predictor Filter. The system with sufficient memory to perform the algorithm is developed and the software for the routine is written. This information is given in appendix A. The performance of each of the processors (both for hardware and software considerations) in the development and execution of that algorithm is compared on the basis of speed, power consumption and complexity of the system.

Finally, the topic of floating point operations and these processors is discussed.

It should be understood that the evaluations and comparisons done in this paper are based on information given by the manufacturers at this time. Additional information may soon become available which will alter the interpretation of the results. Most of these products are new and some have not been released as of yet. Thus information released may be changed or appended.

1.1 Digital Signal Processing

Since Shannon's theorem proved that any continuous time signal could be accurately represented by sampling that signal at regular intervals (provided that it is sampled at a frequency greater than twice the highest frequency in that signal's spectrum)[1,2], means of analyzing and transforming that signal in the discrete time domain have been investigated. For instance, if it is desired to filter some signal to remove undesired components (such as the high frequency end of an audio signal), the question is, can this same operation be carried out by operating on the sequence of discrete points? The answer is yes and a whole lot more[3]. In fact, some processing algorithms in the discrete time domain have no counterpart in the continuous time domain.

The main advantage to performing the processing algorithms in the discrete time domain is that of reconfiguration. For example, suppose that it is desired to remove the low frequency components of the aforementioned audio signal instead of the high frequencies. In the analog domain the circuit components would have to be physically changed. In the digital domain all that must be changed is the vector of coefficients. This can easily be done with software. Additionally, analog component values are subject to drift due to temperature changes as well as time

degradation. Digital filter components (coefficients) are not subject to this. The main disadvantages of digital processors are the limit in bandwidth which is inevitably encountered and the limit in resolution of the data (number of bits in the representation).

The use of signal processing is widespread. Any time a signal is sent through a degrading system, that signal must be processed to undo the degradation. This encompasses a very large number of circumstances. The post-processing may be as simple as a filter or as complicated as a spectrum analysis. In either case, some means of performing this processing must be realized. In a situation where the processing is simple and not subject to change over time, then analog processing may be the best alternative. But in a more complicated processing scheme, where perhaps no analog method is directly applicable, a digital method must be employed. Such is the case for many video processing algorithms as well as a system which calls for a dynamic processing routine.

Some examples of applications of digital signal processing are image recognition, image enhancement (such as shading and smoothing), echo cancellation in communication systems, spectrum analysis, and noise reduction systems. All of these applications require numerically intensive processing.

1.1.1 Digital Signal Processing Computations

Indeed, most algorithms involve a good deal of calculation. A calculation that is common to many algorithms is the correlation calculation. The correlation of a set of data is the sum of the products of the data and the weight of that sample. The weighting coefficients may be anything from probabilities to filter coefficients. The concept is the same regardless of physical interpretation of the coefficients. In mathematical symbology, the correlation, G_M, is

$$G_M = \sum_K F_K * B_K$$

where F_K is the data sample, B_K is the weighting coefficient, and K is the index[4].

This calculation can be done by any microprocessor or computer, but the task of the Digital Signal Processor is to perform this and other tasks with much greater speed to increase the performance in real-time situations.

1.2 The Digital Signal Processor

Because the correlation function is used in so many signal processing algorithms, the DSP's are designed to minimize the execution time of this calculation[5]. The correlation calculation involves multiplying the present operands and accumulating the result of the previous operands to that of all the samples that preceded . Thus a digital signal processor's performance would be greatly enhanced if it has the capability to perform this task in

a single instruction. This reduces the execution time because a program fetch is eliminated and the fast multiplier makes possible execution in typically less than three clock cycles.

In traditional microprocessors, a separate multiply instruction may not even be available. That makes it necessary to perform multiplication by successive addition or software controlled shift and add. In other words, the multiplication is done in software by the programmer or by the microprocessor itself. Regardless of what controls the process, the multiply instruction is very slow and no accumulate function is a part of that instruction. A Digital Signal Processor performs the multiply in hardware. It is accomplished with a series of shift and adds. Even with the hardware,the multiply takes several clock cycles to execute. If the operands are already latched into some multiplier registers, then the multiplication process can start as soon as the operands become available. This is a common practice found in DSP's[6].

The multiplier can be imagined as a device that has two inputs (the operands) and one output (the result). These inputs and outputs are registers that are available for manipulation by every instruction. The result of the multiplication (the product register) is available for manipulation by the next instruction. This makes possible the multiply and accumulate instruction.

The way the multiply/accumulate instruction works is

to load two new operands and take the result of the previous two operands and add it to the accumulator. To make this possible, the DSP must have the capability to move operands and move the result simultaneously. This is only possible if the data transfer controller is independent of the ALU controller. This feature is very common among DSP's.

1.2.1 Indexed Addressing

If the data and coefficient sets have many elements, then the correlation calculation will require each pair of elements be brought in individually, the multiplication performed, the product accumulated and the next pair loaded. The most efficient way to access the operands then is by indexed addressing. This means that registers contain the addresses of the operands. These registers are updated after every operand fetch to contain the address of the next operand. This updating is the job of the Address Generation Unit. This unit works independently of the rest of the control circuitry. This makes the correlation function very easy to implement.

Another indexing feature common among DSP's is circular or modulo addressing. Circular indexing is the capability of an index register to allow post- and pre-updates (decrement, increment, etc.) on only some number of the least significant bits (i.e. modulo N) of an index register. For example, if only the four LSB's are affected, then if 16 increments are done on this index register, its

contents will be the same as it was when it started. This is convenient in that if operations are to be performed on a vector of length of some power of two, then after the operation has been performed on the entire vector, the index register is automatically pointing again to the first element. This alleviates the task of reloading the register with the address of the first element at the end of every vector operation.

1.3 DSP Architecture

1.3.1 Traditional Microprocessor Architecture

Traditionally, no distinction is made by microprocessors between program and data memory space[7]. The first fetch made is for the program word and successive fetches are for the operands of that instruction. The block diagram of this architecture is shown in figure 1-1:

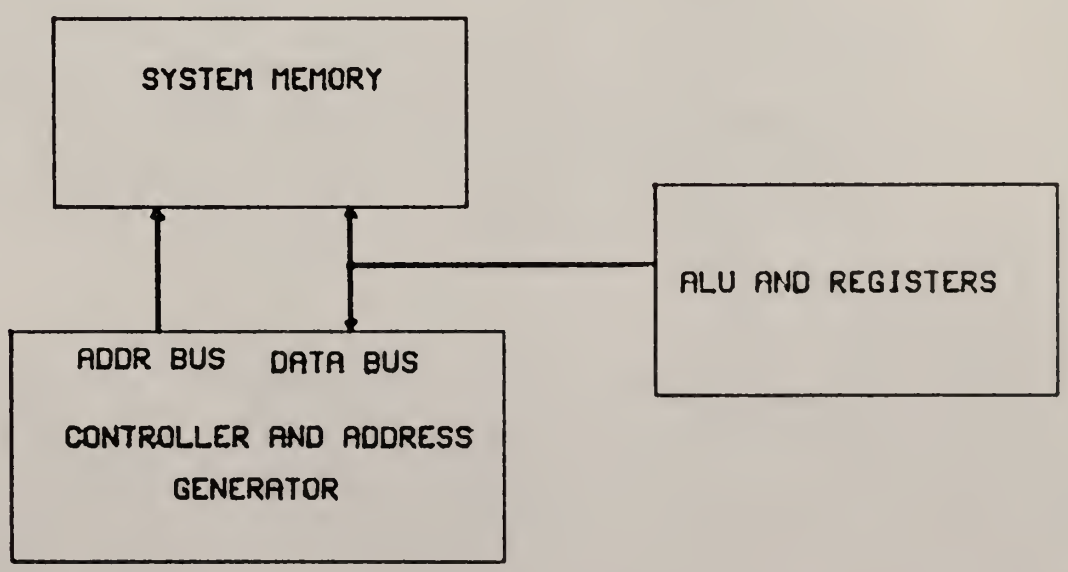

Figure 1-1 Traditional Architecture

There are two main factors which make this system inefficient. The first is that it makes it necessary that the program and data words must be of the same width, e.g. if the data word is 16 bits, then the program word must be 16 bits. A wider program word may be useful to facilitate immediate operands in the instruction and to increase the flexibility of the instruction set in general. The second factor is that after the fetch of the program word has been completed, the processor must take time to decode the instruction before it can perform the operand fetches or ALU operation.

1.3.2 The Harvard Architecture

An alternative to this is the Harvard Architecture[8]. This scheme allows a distinction to be made between the two memory spaces. The block diagram for this architecture is shown in figure 1-2.

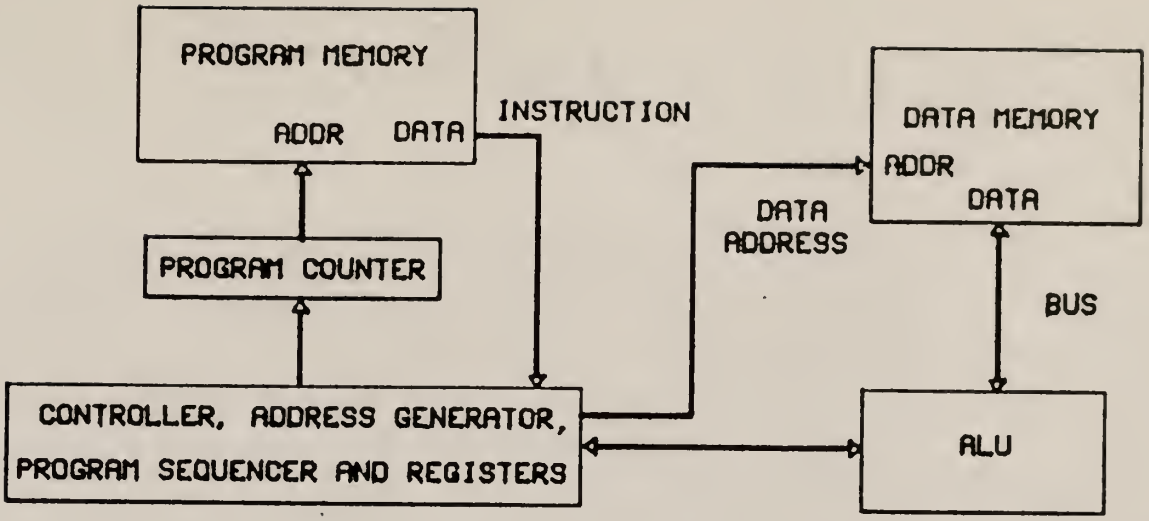

Figure 1-2 Harvard Architecture

This architecture allows for different data and program word sizes and facilitates efficient pipelined instruction execution. This scheme works as follows. First an instruction is fetched. Then the instruction is decoded. Next the actual execution of the instruction takes place. While the instruction is being decoded, the next instruction is being fetched in the program memory space. While the execution of the last instruction is taking place, the instruction fetched during the decode cycle of the last instruction is being decoded. This process is illustrated graphically in figure 1-3.

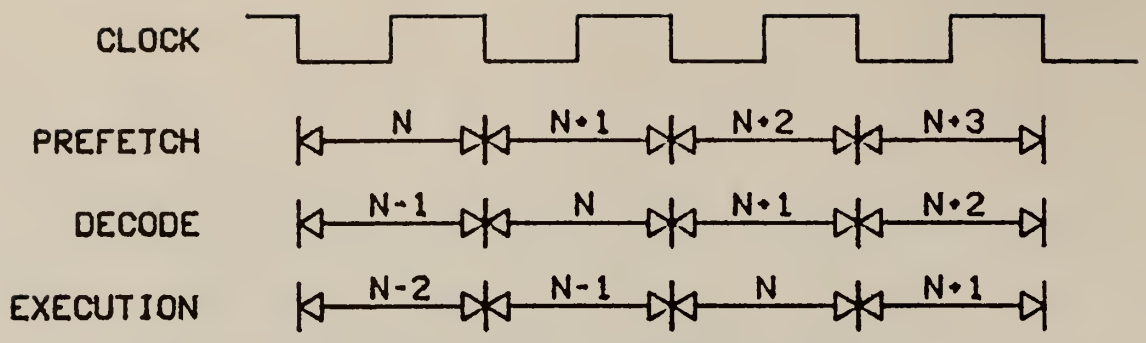

Figure 1-3 Pipelined Instruction Execution

The index N (or N $\pm$ n) denotes the instruction to which the particular operation being performed belongs.

This process is repeated continually. It is possible due to the fact that there is no contention between the data fetches and the program fetches. The buses are physically separate, thus no contention is possible.

1.3.3 Advanced Harvard Architecture

The memories referred to may be on-chip or off-chip. Many DSP's have internal program memory and/or data memory. These memories have separate buses to implement the Harvard Architecture. The external memory space may not be thusly separate. Most DSP's claim to use an Advanced or Modified Harvard Architecture. This can mean a couple of different things or possibly both. It may mean that a physical connection exists internally between the two buses to allow fetches of operands from program memory (see figure 1-4) or that the two buses are distinguished externally only by a single bit. In other words, if external memory is used,

only one set of buses serve both memory spaces. The distinction is made by a status bit(s). Execution will thus be slowed when external memory is used for either or both data and program data.

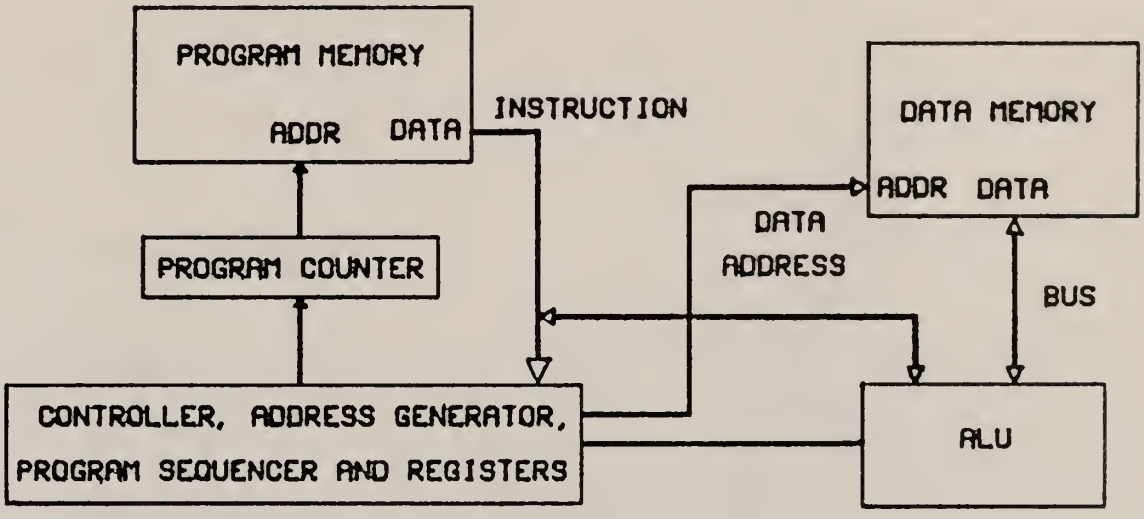

Figure 1-4 Advanced Harvard Architecture

1.4 What is on the Market

Several different classes of DSP's are currently available. These basically fall into three different categories: 1) traditional microprocessor-like systems, 2) digital filter processors which require a host to give it coefficients and data, and 3) dedicated high speed multiplier/accumulators. This paper will limit itself to consideration of the traditional microprocessor-like DSP's.

1.4.1 Types of DSP's

The multiply/accumulate feature is the heart of all DSP's. It is practically the only characteristic that they

all share. Many options are possible building up from the basic multiply/accumulate capability. For instance, a DSP could be designed to perform as a peripheral device, an in-line processor on a data stream, or as the basis of an image processing computer.

1.4.2 DSP Memory Arrangement

The type of design implemented greatly affects the way the processor retrieves and stores data and the way it handles I/O. For instance, a DSP that is designed to act as an in-line processor is more likely to have a limited memory space than a processor designed to be used as a video processor which will have to process a very large number of points which corresponds to having access to a large memory.

Another feature of DSP's is that they need fast access to their operands. It does no good to have a fast multiplier/accumulator if it takes several clock cycles to retrieve the operands from memory. The memory organization will vary greatly as well. Since the multiplier needs two operands, the data memory may be broken up into two separate banks, or maybe one of the operands is to come from a program memory bank, or perhaps the two operands are to come from a special set of registers.

1.4.3 DSP Arithmetic

Another issue to be addressed by a DSP designer is the type of arithmetic his processor will support. Different

applications require varying degrees of accuracy. For instance, an audio signal needs as least 12 bits of accuracy to maintain its quality through the processing. Some applications may require less, some more. Additionally, how many bits should be allowed to guard against overflow, and if an overflow occurs, should the result stand or should the result be largest number representable? If the algorithm to be implemented inherently involves calculations in the complex plane, then a processor that has dual ALU's would significantly decrease the time of execution of that algorithm.

The discussion that follows applies to DSP's which use fixed-point, signed two's complement arithmetic. Complex data may be used by these processors, but they may not have a built-in complex data handling capability.

2.0 Features of Available Fixed-Point DSP's

2.1 Processors to be Analyzed

The processors that will be discussed in this section are:

DSP56000 (Motorola, Inc.)[9]

TMS320C25 (Texas Instruments, Inc.)[10]

TMS32020 (Texas Instruments, Inc.)[11]

TMS320C10 (Texas Instruments, Inc.)[12]

LM32900 (National Semiconductor, Inc.)[13]

UDPI 01 (ITT Corporation)[14]

ADSP2100 (Analog Devices, Inc.)[15]

ZR34161 (Zoran Corporation)[16]

S7720 (AMI/Gould Industries)[17]

MSM7720 (OKI International)[18]

2920 (Intel Corporation)[19]

HD64180 (Hitachi International)[20]

DSP16 (AT&T)[21]

The information used to evaluate each of these was obtained from the source referenced for each processor.

2.2 DSP Feature Tabulation

The features tabulated are broken up into four separate tables. The first two and fourth tables are contain only yes/no entries. If the processor has the capability mentioned then it is so noted. The third table contains numerical entries.

The first table (table 2-1) is for hardware considerations (features that are important in the context of a developing a system around the DSP). The second table (table 2-2) displays software features (those concerned with algorithm development). The third table (table 2-3) is a quantitative display of both hardware and software features. The fourth table (table 2-4) lists support features that are available for each DSP. For ease of simultaneous viewing of the tables, they are all grouped together at the end of this section (2.2).

In all of the tables, if the information is not known, it is denoted by the entry "NA" (not available). If the entry does not apply to a particular DSP, then that entry is "NAP" (not applicable).

The intent of these tables is to provide the potential user with a quick reference to base a choice or simply an evaluation on. They indicate nothing about what it takes to get one into a running system or how it would perform in that system.

Next, an explanation of the contents and usefulness of each of these tables is given.

2.2.1 Hardware Considerations

The first table (table 2-1) concerns the hardware aspects of the processors. This table gives information concerning the memory (on-chip and off-chip program and data memory), I/O, physical aspects (fabrication

technology, TTL compatibility), capability to communicate with other devices, and analog signal interfacing.

Some other entries in the table include memory strobe bits, separate program and data buses, built-in clock circuitry, power down mode, bus grant to another processor, and requires host. A brief discussion of exactly what each of these mean follows.

Memory strobe bits are bits which are used as timing reference signals for the external memory. They signify that a transfer is about to take place (by the level of the pin) and when the transfer takes place (by the transition of the pin). This alleviates the responsibility of generating these signals with external logic.

Separate program and data buses means that the two memory spaces do not share any buses externally. This is the Harvard Architecture. This entry applies to its external bus connections only.

A built in clock implies that the DSP needs only the controlling crystal to produce the clock signal. The signal does not have be generated externally.

A power down mode is a state that the DSP can enter when it has nothing to do. It waits in a low power consumption state until it receives an interrupt or bus request.

Bus grant to host implies that the DSP has the capability to place its bus drivers into a high impedance state upon a request (via a request line) and allow an

external driver. The host can then have access to the DSP's memory.

Requires host is an entry that indicates that the processor cannot function as a stand alone unit. It needs a host to direct its program flow or possibly to give it instructions.

2.2.2 Software Considerations

The second table (table 2-2) demonstrates the features that of particular concern when programming the processor. These features include data format support (double precision, complex,floating point), important digital signal processing instructions (store backwards, multiply/accumulate, divide primitive, branch on I/O status), overflow handling, addressing, loop control, fetch of two operands simultaneously, and result justification (shifting).

To be classified as having complex data support a processor must have separate ALU's for each part of the data. To classify as supporting double precision the DSP must have a double precision accumulator and a means to perform ALU operations with this and another double-precision operand.

To be classified as offering floating point support, the DSP must have a normalization procedure and a counter to keep track of the number of shifts needed to perform the normalization.

Overflow handling consists of two parts. The first is the presence of guard bits on the accumulator to maintain the correct result on an overflow. The second is the capability of the DSP to automatically set the result to the most positive or most negative number representable in case of an overflow. In other words, if the operation causes a negative overflow (such as subtracting one from the most negative number representable) then the result is set to be the most negative number representable. This maintains the integrity of the sign of the result and most nearly represents the correct result.

The loop control entries are the capability to repeat an instruction N times (with index register update at the end of each repetition) and loop variables. The repeat capability is convenient for the implementation of the correlation calculation as well as adding a series of numbers. A loop variable is a register that gets decremented each time the loop is traversed. The program flow is then directed based on the content of that register. If it has expired (contains zero) then continue passed the loop. If it has not, then return to the top of the loop.

The circular indexing entry indicates that the DSP has the capability to perform modification on only the last N bits of the index register (see section 1.2.1).

A divide primitive is an instruction which aids in the divide calculation. To divide two integers in N-bit binary

arithmetic, one must perform a series of operations similar to that the human machine uses[22]. Basically, the technique for dividing two positive integers is as follows. First, shift the divisor left N-1 bits. Next subtract this from the dividend (extended with zeros to double precision). If the result is negative, then ignore the result of the subtraction, reload the old dividend and shift in a zero from the right to the accumulator (dividend). If the result of the subtraction is positive, then keep the result of the subtraction as the new dividend and shift in a one. In either case, it is the first bit of the result. Repeat this operation as many times as needed (for as many bits of result as desired). The result must then be justified according to the radix points of the operands. To carry out this procedure, a subtract conditionally with a left shift (with appropriate value) is extremely helpful. If the DSP has this or a similar capability, it is categorized as offering a divide primitive.

Store backwards is the ability to store a value at a location pointed to by some register with the last N bits of the pointer reversed. This is useful for FFT's.

The shifting entries are in-line shifting for accumulator access and a dedicated shifter (barrel shifter). An in-line shifter means that operands may optionally be shifted before moving to or from the accumulator. This is helpful for justification when storing the result of an arithmetic operation. A barrel shifter is

a device that has a very wide input field (maybe twice data word length). The input to the shifter is directed to some section of that input field. The output is always whatever resides in the special section that corresponds to a shift of zero. Access to this device is usually an instruction in itself.

The block move capability is the ability of the processor to move a set of data from one place in memory to another place in memory without the need to store away any of the registers. In other words, the procedure is all handled by the processor instead of the programmer hiding a register in memory somewhere, then loading the first value into that register, storing it to the new location, getting the next value,etc.

2.2.3 Quantitative Features

The third table (table 2-3) is a quantitative demonstration of each DSP's features both for hardware and software considerations. The table gives information on the size of on-chip memory, total memory space size, clock frequency limits, temperature range, power consumption, number of bits in the data word, number of index registers, number of I/O ports, stack depth, most recent information used for evaluation, and availability date where known.

Also listed is memory speed. This is the speed that any memory added to the DSP's bus must be to operate at the DSP's fastest clock frequency. No absolute number can be given (because of varying select logic delays, etc.) but

the number given is the time that the DSP requires on a read from external memory. The time is defined as the maximum allowable delay from the time the DSP's address bus goes valid to the time when the data on the output of the memory must be valid (to meet minimum set-up times). This gives an approximation as to how fast all memory must be (ROM and RAM since they are both readable).

2.2.4 Development Tools

The fourth table (table 2-4) is a compilation of available support tools for each DSP. The support tools listed are software simulator, hardware emulator, evaluation board, assembler, and high level language compiler where this information is known.

A simulator is a program that allows a designer to write programs in the appropriate assembly language and then synthesize memory contents to debug the software without transporting it to the system under development (which itself may not be debugged).

An emulator is a device that acts as the DSP in a system, but has the additional capability to show the values of memory locations (and possibly registers) on some sort of display. This is helpful in finding bugs in system hardware.

All of the preceding information is given in tables 2-1 through 2-4.

Table 2-1 Hardware Considerations for DSP's

	DSP 56000	TMS 32025	TMS 32020	TMS 32010	ZR 34161	MSM 7720	S7720	UDPI 1	ADSP 2100	LM 32900	HD 64180	2920	DSP16
Separate program and data buses	N	N	N	N	N	NAP	NAP	NAP	Y	Y	N	NAP	NAP
On-chip program ROM	Y	Y	N	Y	N	Y	Y	Y	N	N	N	Y	Y
On-chip program RAM	Y	Y	Y	N	N	N	N	N	N	N	N	N	N
On-chip data RAM	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	Y	Y
On-chip data ROM	Y	N	N	N	N	Y	Y	Y	N	N	N	N	N
On-chip EPROM	N	N	N	N	N	N	N	N	N	N	N	Y	N
On-chip A/D	N	N	N	N	N	N	N	N	N	N	N	Y	N
On-chip D/A	N	N	N	N	N	N	N	N	N	N	N	Y	N
Memory transfer strobe bits	N	Y	Y	Y	Y	NAP	NAP	Y	Y	Y	N	NAP	N
Dedicated I/O pins	Y	Y	Y	Y	N	Y	Y	N	N	Y	N	NAP	Y
Bus grant to host	Y	Y	Y	N	Y	NAP	NAP	N	Y	Y	Y	NAP	N
Requires host	N	N	N	N	Y	N	N	N	N	N	N	N	N
Accepts interrupts	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	Y	N	Y
CMOS technology	Y	Y	N	Y	Y	Y	N	Y	Y	Y	Y	NA	Y
TTL compatible	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	Y	NAP	Y
Low power mode	Y	N	N	N	N	N	N	N	N	Y	Y	N	N
Built-in clock	Y	Y	Y	Y	N	N	N	Y	N	N	Y	Y	N

NA denotes information regarding this entry unavailable
NAP denotes entry not applicable to this processor

Table 2-2 Software Considerations for DSP's

	DSP 56000	TMS 32025	TMS 32020	TMS 32010	ZR 34161	MSM 7720	S7720	UDPI 1	ADSP 2100	LM 32900	HD 64180	2920	DSP16
Multiply accumulate instruction	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	Y
Simultaneous fetch of two operands	Y	Y	Y	N	N	Y	Y	Y	Y	Y	Y	N	Y
Branch on I/O status	Y	Y	Y	Y	N	N	N	Y	N	Y	N	N	N
Block move	N	Y	Y	N	Y	N	N	N	Y	N	Y	N	N
Divide primitive	Y	Y	Y	Y	N	N	N	N	N	Y	N	N	N
FFT or store backwards	Y	Y	N	N	Y	N	N	N	Y	N	N	N	N
In-line shifting	Y	Y	Y	Y	Y	N	N	N	N	Y	N	N	N
Barrel shifter	N	Y	Y	Y	N	N	N	N	Y	Y	N	N	N
Guard bits on acc. for overflow	Y	N	N	N	Y	N	N	N	Y	N	N	N	Y
Round to infinity on overflow	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	N	N	Y
Removes extra sign bit of multiply	N	Y	Y	N	Y	Y	Y	Y	Y	Y	NAP	NAP	N
Repeat next instr.	Y	Y	Y	N	N	N	N	N	N	Y	Y	N	Y
Loop counters	Y	Y	Y	Y	N	N	N	Y	Y	Y	N	N	Y
Circular indexing	Y	N	N	N	N	Y	Y	N	Y	Y	N	N	Y
Double precision accumulator	Y	Y	Y	Y	N	Y	Y	Y	Y	Y	N	Y	Y
Double precision support	Y	Y	Y	Y	N	N	N	N	Y	N	N	N	Y
Floating-point support	Y	N	N	N	N	N	N	N	N	Y	N	N	N
Complex data support	N	N	N	N	Y	N	N	N	N	N	N	N	N

NA denotes information concerning entry not available
NAP denotes entry not applicable to processor

Table 2-3 Quantitative Features for DSP's

	DSP 56000	TMS 32025	TMS 32020	TMS 32010	ZR 34161	MSM 7720	S 7720	UDPI 1	ADSP 2100	LM 32900	HD 64180	INTEL 2920	DSP16
On-chip program ROM	2K	4K	0	1.5K	0	512	512	1K	0	0	0	192	2K
On-chip program RAM	0	256	256	0	0	0	0	0	0	0	0	0	0
On-chip data ROM	512	0	0	0	0	512	512	72	0	0	0	0	0
On-chip data RAM	512	544	544	144	128	128	128	440	0	0	0	40	512
Program memory space	192K	64K	64K	4K	64K	512	512	1K	4K	64K	512K	192	64K
Data memory space	192K	64K	64K	4K	64K	128	128	1K	4K	64K	64K	40	512
Data word length	24	16	16	16	16	16	16	16	16	16	8	25	16
External memory access * (ns)	65	40	76	55	45	NAP	NAP	NAP	50	45	750	NAP	50
Stack depth	15	8	4	4	0	4	4	4	16	64K	64K	0	1
Number of index registers	8	8	5	2	0	1	1	10	8	8	6	0	4
Clock cycle limits from	4	6.7	6.7	6.7	5	.5	.5	5	0	0	0	0	1
to (MHz)	20.5	40	20	25	20	8.2	8.2	20	32.8	20	8	10	36.4
Number of parallel ports	2	16	16	4	0	1	1	1	0	2	0	NAP	1
Parallel port width (bits)	9,15	16	16	16	NAP	8	8	16	NAP	16	NAP	NAP	16
Number of serial ports	2	1	1	1	0	2	2	2	0	2	2	NAP	1
Number of pins	88	68	68	40	48	28	28	40	100	172	64	NA	84
Maximum power usage (W)	1	1	1.5	.4	.4	1	.9	.9	.6	.5	.075	NA	1
Temperature range from	-40	0	0	0	-55	-10	-40	0	-55	-55	NA	NA	-40
to (C)	+85	70	70	70	125	70	85	65	125	125	NA	NA	120
Most recent information	'86	'86	'86	'86	'86	'86	'85	'86	'86	'86	'85	'85	'88
Availability date	-	NA	NA	-	NA	NA	NA	NA	NA	NA	-	-	NA

* Address valid to data valid at maximum clock frequency
NA denotes information concerning entry not available
NAP denotes entry not applicable to processor
- denotes currently available

	DSP 56000	TMS 32025	TMS 32020	TMS 32010	ZR 34161	MSM 7720	S7720	UDPI 1	ADSP 2100	LM 32900	HD 64180	INTEL 2920	DSP16
Assembler	Y	Y	Y	Y	Y	NA	NA	Y	Y	Y	Y	NA	Y
Simulator	Y	Y	Y	Y	Y	NA	NA	Y	Y	Y	Y	NA	Y
Emulator	NA	Y	Y	Y	Y	NA	NA	Y	Y	N	Y	NA	Y
High level language	Y	Y	Y	Y	NA	NA	NA	NA	NA	NA	N	N	Y
Evaluation board	Y	N	N	N	N	NAP	NAP	NAP	NA	NA	Y	NAP	NA

NA denotes information concerning this entry not available
NAP denotes entry not applicable to processor

Table 2-4 Support Tools for DSP's

2.3 Using the Tables

The question that arises at this point is, what good do the preceding tables do for the person who wishes to compare the prospective performances of those DSP's in his situation? The best way to illustrate their usefulness is to outline the procedure to use given some dominant consideration.

First consider software development. It is considered first because the size of the algorithm used will greatly influence the choice of processors. The first step is to determine the amount data memory that will be required. First, determine the number of fixed variables and constants that the routine to be implemented will use. These quantities will definitely need unique memory locations. Other variables (such as loop counters, temporary variables, etc.) may not need a separate memory location set aside for them. Next determine the number of nested loops and/or subroutine calls that the routine will make. Now check the chart to determine if the processor has a stack depth sufficient to perform these and if it has enough loop counters to perform all of the nested loops. If either of these conditions is not sufficient, then memory must be set aside to perform these tasks in software if possible. Next estimate the number of temporary memory locations that will be needed. Use these numbers to make a rough estimate of the size of data memory required. Eliminate all processors that do not have sufficient space

for the data.

Now determine the amount of program memory that will be required. This task is not simple because it is a function of the efficiency of the machine language of the processor and complicated by the fact that some processors require that some variables be stored in program memory to facilitate simultaneous fetch of two operands. At any rate some estimate must be made to determine which of the processors has sufficient program memory space to facilitate the algorithm.

Now that a list of processors that can implement the algorithm has been made, three basic elements must be addressed - speed of execution, complexity of implementation and power consumption. Any or possibly two or three of these may be of great concern to the user. These considerations must be prioritized before continuing. If power consumption is the major concern, then proceed to evaluating that feature. The same applies to the other topics. The point is that a primary list of candidates should be generated based on the most important aspect and further reduction in candidates should come come from subsequent evaluations.

2.3.1 Speed of Implementation

Many DSP features contribute to speed performance. Chief among these is instruction cycle time, I/O efficiency, data movement within memory, and unsupported data type calculations.

The first step is to define the type of data that the algorithm is to use. If the algorithm is to use 16-bit fixed point for all of its data, then an 8-bit processor would be inappropriate. If some data are inherently complex, then those processors that offer complex data support would be the most promising. If some calculations will be on double precision data, then check to find the processors which offer some support for double precision calculations. Also, if the algorithm requires a substantial number of divides, then a processor that supports divide calculations should be rated above those that do not. A simple divide can occupy a disproportionate amount of the execution time of an algorithm if some support for that calculation is not offered.

Additionally, if the operands of calculations are of mixed radix points, then some shifting will have to be done on the result to come up with the correct radix point. This may require a barrel shifter or simply an optional shift of the result on access to the accumulator.

After the remaining processors have been rated according to data representation and calculation efficiency, consider the I/O that will be required.

First determine how many channels will be required (both parallel and serial). Next check to see if the processor has the capability to branch on I/O status. If it does not have that capability it may be quite difficult to determine when data is available from or

required by peripheral devices.

Next consider the amount that data must be shuffled around in memory. If some of the data consists of a buffer of past values that is updated at the end of each iteration of the algorithm, then the ability of the processor to make block moves of data may be a great asset.

Now factor in the clock frequency. A processor can make up for some inefficiencies by just being faster.

This should leave the user with a prioritized list of candidates of processors. Further narrowing and rearranging of that list can come from the following considerations.

2.3.2 Power Consumption

Three factors contribute to the total power consumed by the system - the power consumed by the processor, the power consumed by peripheral devices and memory, and the speed at which those parts are operated.

Check the power consumption entry on the quantitative table to find those processors which consume less power than the constraint of the situation. It should be kept in mind that this figure will be lower for CMOS devices if they can be run at a lower clock frequency. CMOS devices consume little power except when they switch states. When the clock frequency is decreased, then number of switches in some unit of time will be reduced. Thus power consumption will be reduced. Thus if the algorithm is not required to execute extremely fast, then the power consumption can be cut.

The power consumed by peripheral devices will probably not vary a great deal from processor to processor unless that processor requires some sort of host or I/O processor. The power consumed by this device may be substantial.

The power consumed by the memory is very critical. If the DSP does not enough internal memory to hold the variables and the program, then memory will have to be added. Therefore the amount of memory and its associated power consumption must be figured.

Additionally, if a DSP is fabricated in CMOS, it may well be due for a shrink in size (due to the advancing technology in CMOS fabrication). This may lead to future reduced power consumption.

2.3.3 Complexity of Implementation

The main contributor to this consideration is the amount of program and data memory that must be added to the DSP's buses. The main factor to consider once the amount of memory to be added has been determined is the speed of that memory. The attempt to reduce the number of clock cycles needed to execute an instruction necessitates that external memory access time be reduced as well. Some of the access times (given in the quantitative table) are so short that if the processor is to be run at full speed, then the memory must be about as fast as present technology can offer (about 40 ns).

Even with the information given in these tables, it may still be difficult to determine the effort required to

implement the DSP's in a digital signal processing environment. It would be helpful to have a direct comparison of the DSP's on an unbiased algorithm.

The next section of this report will select a subset of the processors compared and perform a digital signal processing algorithm with them. The nature of the algorithm will exclude some of the DSP's mentioned from being evaluated.

3.0 Comparison On Standard Algorithm

The vast number of features listed in the tables of the preceding section of this report are too numerous to all be considered when evaluating the prospective performances of the DSP's in a specific task. A head to head comparison of the performance of each of the DSP's on an unbiased digital signal processing algorithm would give some insight into their usefulness. This would entail setting up a system for each processor to be able to execute the chosen algorithm (with consideration of initialization requirements) on some predetermined data flow. This would show a prospective user what all of that information tabulated previously boils down to when it is time to implement these processors.

3.1 The Standard Algorithm

The problem lies in developing an algorithm that is representative of the sort of task that these DSP's will be called upon to perform, but is not biased to one or more of the processors. For instance, a video processing algorithm would most likely require access to a large memory bank. The processors that do not have external connections to their buses would perform very marginally in such an environment. Because the area of digital signal processing is much wider in scope than just video processing, this would not be a fair comparison.

Many applications do not need access to an extremely large amount of memory. If this were not so, these type of

processors would never have been built. A more just comparison would be an algorithm that exhibits characteristics of most signal processing algorithms. It should contain the correlation calculation, have a definite data format that must be maintained throughout the execution, execute continuously (because these processors are mainly intended to perform real time processing), not have a large memory requirement, exhibit the ability to signal a fault, and require some minimal initialization.

A practical algorithm that exhibits these features is the adaptive linear predictor. This algorithm is commonly used for echo cancellation in communication systems and any place where additive deterministic signals are to be isolated.

3.2 The Adaptive Predictor Filter

The basis of the adaptive predictor filter (APF) is the finite impulse response (FIR) filter. An FIR filter is a digital filter whose output depends only the last N inputs and not at all on any of the previous outputs[23,24]. The output of the FIR filter is given by :

$$G(T) = \sum_{i=1}^{n} B(i)*F(T-i)$$

where G(T) is the output sample, B(i) is the i^{th} coefficient of the filter, and F(T-i) is the input at time T-i. It is quite apparent that this is identical to the correlation calculation mentioned earlier.

Indeed the name correlation is quite appropriate for this algorithm. The APF's function is to adapt the

coefficients of the FIR filter to decorrelate the output sequence (G). This is jumping the gun a little. It is necessary to describe the situation in which the APF is used.

3.2.1 Application of the APF

Suppose that an uncorrelated (white) sequence of data is to be sent through a system of unknown transfer function, H(z) (as shown in figure 3-1).

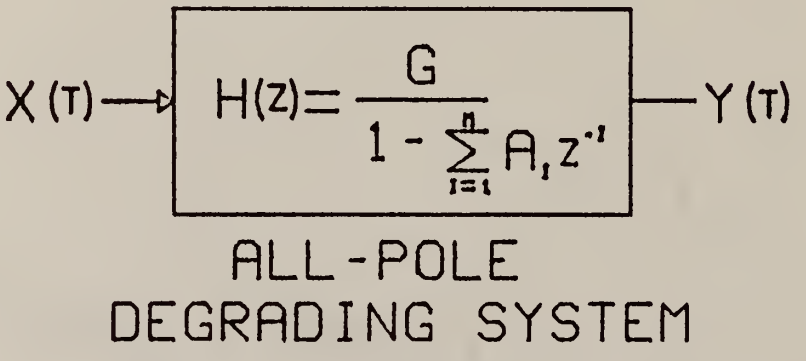

Figure 3-1 Corrupting System

The output of this degrading system is a valid model for a number of stochastic processes. The autocorrelation of any random process is the Fourier transform of the power spectral density (PSD) of that process. Any transformable PSD can be generated by passing a white PSD (flat spectrum) signal through a filter which has a magnitude response equal to the desired PSD. It can thus be noted that any stochastic process can be represented as a white process that has been passed through a causal filter.

The process can be reversed. In other words, a process may be "whitened" by passing it through a filter that has a

magnitude response equal the inverse of the PSD of that process. If a filter can be constructed with this magnitude response, then the process can be decorrelated[25]. This is demonstrated by figure 3-2.

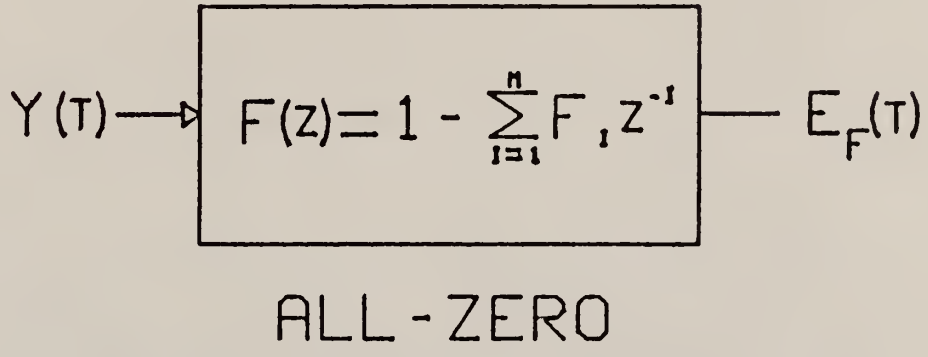

Figure 3-2 Whitening Filter

The APF adapts the coefficients of an FIR filter to accomplish this end. One potential problem with this approach is that the FIR filter is an all-zero filter. The only way the output of that filter can be made white is if the input has an all-pole spectrum. This turns out not to be a significant problem because of a mathematical relationship. Any zero can be expressed as an infinite product of poles as :

$$1 - AZ = \frac{1}{1 + AZ + A^2Z^2 + A^3Z^3 + \cdots}$$

If only the first few of these terms are kept, a reasonable approximation can be made. Thus the all zero filter can decorrelate any stochastic process provided that the order of the filter is sufficient. If the input process is not stochastic (if its mean and variance change with time) then the coefficients of the filter must be changed to meet

the changes in the signal.

3.2.2 The Linear Predictor Filter

The algorithm that is to be implemented decorrelates the sequence $F_M - G_M$. This signal is termed the error sequence (e_f). This adaptation does not strictly abide by the description given above. The purpose of the FIR filter is to make a prediction of the next input sample $F(T+1)$[26]. This is why it is given the name linear predictor filter (LPF). This corresponds to making the error process e_f a white process (uncorrelated). This is done by making the best choice for the next value based on the knowledge of the last N.

The way this is done is to change the coefficients of the FIR filter at each iteration in such a way that the error sequence is uncorrelated. This has the effect of picking out the correlation of the input process and using it to calculate the best guess for the next input. This is performed by means of adjusting the impulse response of the filter (which is equivalent to adjusting the magnitude and phase responses). This picks out the trend in the input process. Once that trend is known, then prediction is simply a linear prediction.

The problem is that the trend is not known. An unbiased estimate of it can be made by multiplying the error sample and the input sample. This is not a good estimate, but it has the correct sign and has the feature that when the error gets smaller, so does the estimate.

Thus it can be used as an update to the correlation function. If the estimate is negative for that sample, then decrease the coefficient. If the estimate is positive, then make the coefficient larger. This is termed a stochastic gradient algorithm because the adjustment to the coefficients takes a random path (because the estimate is based on a random process).

The particular stochastic gradient algorithm used is the Widrow's algorithm[27]. The updating algorithm is as follows:

$$B_{M,K} = B_{M-1,K}*U + V*E_M*F_{M-K}$$

where U and V are adaptation constants which determine the speed and accuracy with which the coefficients change. They are somewhat arbitrary but are based on some knowledge of the signals being processed.

3.3 The Adaptive Algorithm

3.3.1 Specifications

To implement this algorithm it is first necessary to know the exact requirements. The incoming signal is assumed to be analog. Thus provisions must be made to convert it to a digital sequence. A fault must be registered if the adaptation fails. The condition under which it is termed failing is if the square of the mean of the error sequence is greater than a specified value. Thus the error sequence must be averaged over the past N samples, then that value is squared. Finally, that value is compared against a known

threshold.

Additionally, the algorithm is to be 16-tap, which means that it acts only on the last 16 data samples and that the FIR filter has 16 coefficients.

To implement the algorithm it is necessary to perform the following tasks:

1) convert the incoming analog signal to a digital sequence
2) input the conversion to the system data memory
3) execute the Widrow's algorithm
4) compute the average (Q_M) of the error sequence
5) compute Q_M^2
6) compare Q_M^2 against THETA
7) output a bit indicating whether or not the threshold has been crossed

The analog to digital converter (A/D) is to meet the following criteria :

1) 12 bits of resolution
2) active low start conversion signal (input)
3) active low enable output of result signal (input)
4) active end of conversion flag (output)

The 12 bit resolution requirement is intended to give the most resolution at the high conversion rate which will be required by these algorithm executions. They are commercially available in fairly low power packages. Unfortunately, it makes implementation of the algorithm practical for 16-bit and wider processors. This is not a

great loss because it is difficult to compare 8-bit processors with the much more powerful 16-bit processors.

The algorithm itself is to assume that all variables and constants are of the 1/0/15 format. That is a 16-bit representation with 1 sign bit (the most significant bit), 0 bits left of the radix point and 15 bits to the right of the radix point. Since the converted data word is only 12 bits wide, the remaining bits must be zeroed. Also the most significant bit of the converted data will be incorrect. It must be inverted. This is because the A/D converts the signal that is equal to V_{ref}^{+} as all ones. In 1/0/15 format, all ones is the representation for - 1/32768 (the smallest negative number representable. If the MSB is inverted, it becomes 32767/32768 (the most positive number representable) which is correct.

An input value equal to V_{ref}^{-} converts to all zeros. If the MSB is inverted, then it becomes -1 in 1/0/15 format which is the appropriate value.

The block diagram of the complete algorithm is given in figure 3.3.

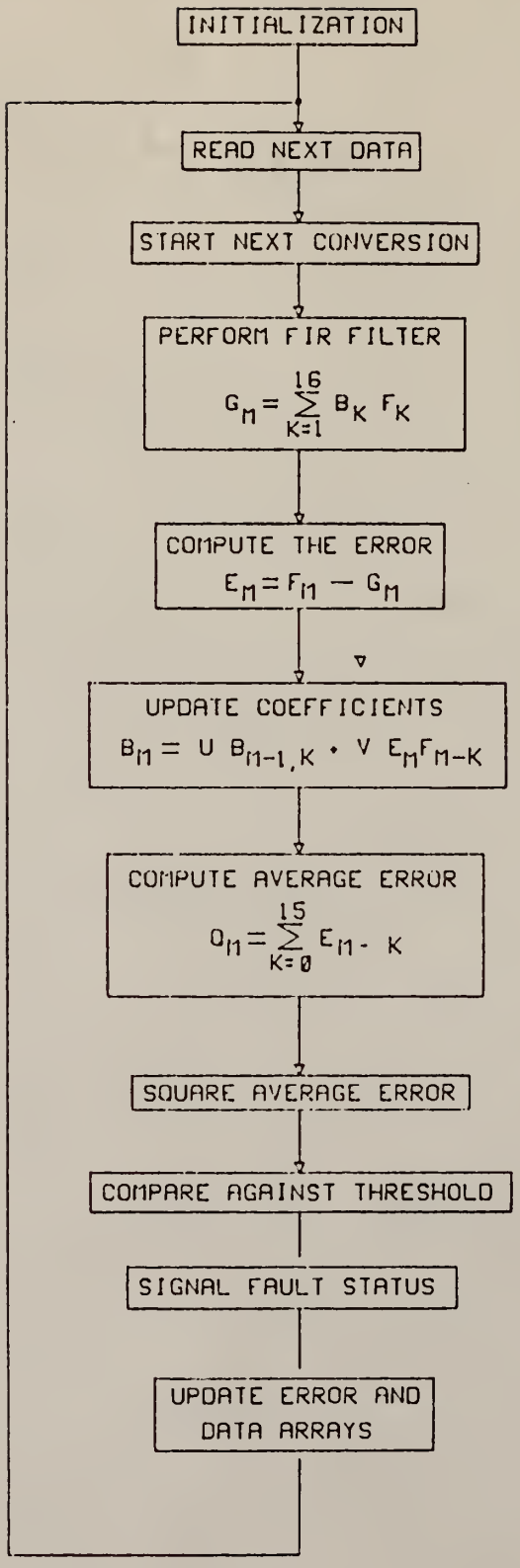

Figure 3-3 Algorithm Block Diagram

The initialization involves the following:

1) downloading the initial coefficients and the constants U, V and THETA (threshold) from ROM
2) start the first conversion of the incoming data
3) initializing the input sequence to all zeros

Because the evaluation is mainly interested in evaluating the performance of the DSP's execution of the main loop of the adaptation algorithm, no attempt is made to write the initialization routine. All that is done is to make provisions for it (allowing sufficient program memory and finding a source in ROM of the initial variables and constants).

The update buffers section involves getting rid of the last data and error sample after each iteration. Because the algorithm acts only upon the last sixteen samples, each time through the last sample must be discarded and the remaining samples pushed down one position in their respective arrays. In other words, the data sample with index 3 becomes the data sample with index 4. The same holds true for the error array. The data array has a special consideration in that the data sample obtained at the beginning of the loop should now be placed on the top of the data array.

The update function is essentially a pushdown stack. A pushdown stack is a sequence of memory locations that contains the last N samples of some input or output. Each iteration, a new sample is placed at the top of the buffer

and all previous samples are moved down one position in the buffer. The last element of the buffer is lost.

3.4 Implementations

Some of the processors presented earlier are not evaluated here. The reasons vary from processor to processor. The DSP16 is not implemented because not enough information concerning its I/O structure and instruction set details can be obtained. The HD64180 is excluded because it is an 8-bit processor with little double-precision support. That makes 16-bit computations quite difficult. The Intel 2920 has no built-in multiply/accumulate instruction. This makes the algorithm very slow in comparison to the other processors.

The DSP's which are evaluated are (in the same order as they appear in the appendix, which is the order in which they appear in the tables of chapter two):

1) DSP56000
2) TMS32025
3) TMS32020
4) TMS32010
5) ZR34161
6) S7720 and MSM7720
7) UDPI 1
8) ADSP2100
9) LM32900

The S7720 and MSM7720 are evaluated as one DSP because they are pin for pin compatible as well as software compatible. They also run at the same clock frequency which makes comparison of speed performance identical.

The system for each DSP is designed with the assumption that speed of execution is the most important factor. Thus if a reasonable amount of hardware can be used to replace a slower implementation in software, it is done. Thus some implementations have more hardware than is absolutely necessary to perform the task.

These systems are outlined in detail in appendix A. For each DSP it contains information concerning hardware implementation (including a block diagram), usage of memory and registers, initial conditions, software for the adaptive algorithm and evaluation of implementation.

The hardware implementation section for each processor outlines the logic of the system, i.e. it describes how the system reads data into the DSP and how the DSP signals an alarm. Details about select circuitry, memory arrangement, and I/O functions are addressed. It includes a hardware block diagram that shows the system circuitry at a high level. It is not a schematic, but part numbers may be cited to show what is necessary for implementation. It shows only necessary connections, all else is not shown. It is not necessarily a minimal system, only one that outlines the logic necessary. It might better be realized by a programmable logic array. If a block diagram does not show

a clock circuit, then that DSP has an internal oscillator.

The initial conditions section lists all assumptions made in addition to the assumptions that all of the systems face (downloading of constants, start conversion of first data, etc.).

The assembly code section is a listing of code that was written to perform the algorithm at hand. It must be understood that this code is not tested and not necessarily the most efficient possible. This test is designed only to give some idea as to performance. The code is commented for ease of reading. All algorithms follow the same format and use the same variable names.

The format is as follows:

1) input next sample from A/D
2) store that in a memory location called FM
3) perform an FIR filter on the COEFFS and DATA arrays, call the result FM
4) compute the error in this estimate as EM = FM - GM
5) store this value at the top of the ERROR array
6) update the COEFFS array according to the formula

   $B_{M,K} = B_{M-1,K}*U + V*E_M*F_{M-K}$,

   where V and U are constants and $B_{M,K}$ is a coefficient of the M^{th} iteration and index number K (equal to 0 through 15)
7) perform the moving average filter on the ERROR array, call the result QM
8) square QM, call it QM2

9) compare QM2 against THETA (a known, constant threshold)

10) set the alarm according to QM2 and THETA

11) update the ERROR and DATA arrays

DATA, ERROR and COEFFS are all arrays of 16 elements.

Finally, an evaluation of the system is given. This may include commentary on ease of comprehension of manufacturers information, complexity of programming, complexity of I/O and memory circuitry, results of the implementation and possibly areas that DSP might be better suited for use.

3.5 Results

Now an index as to how well each of the processors would perform this task must be formulated. Again a table of pertinent information is given. This saves the evaluator the effort of examining the code and system block diagrams in detail.

The table (table 3-1) gives information concerning the I/O circuitry, additional memory requirements, program length, and execution time.

The I/O circuitry entries describe what additional circuitry was needed to implement the A/D conversion as well as maintain a bit for a threshold detection.

The memory entries describe how many distinct (in function) memory banks are required in addition to any on-chip memory. Also the need for select circuitry for these banks is indicated. The magnitude of these banks is also

given.

The number of program words is listed. This gives an idea as to the efficiency of the code.

The execution time is the time that it takes the processor to perform one iteration of the adaptation routine. The frequency of execution is simply the reciprocal of the execution time.

Table 3-1 Results of Implementation

	DSP 56000	TMS 32025	TMS 32020	TMS 32010	ZR 34161	S,MSM 7720	UDPI 1	ADSP 2100	LM 32900
Needed I/O controller	N	N	N	N	Y	Y	Y	N	Y
Needed host	N	N	N	N	Y	N	N	N	N
Needed extra select logic	N	Y	Y	Y	Y	N	N	Y	N
Needed clock circuit	N	N	N	N	Y	Y	N	Y	Y
Needed latch for alarm	N	N	N	Y	Y	N	Y	Y	N
Number of memory banks	0	0	1	0	2	0	0	3	3
Number of prog words	50	55	58	86	114	99	NA	69	40
Words of external RAM	0	0	0	0	64	0	0	96	128
Words of external ROM	0	0	128	0	128	0	0	128	128
Execution time (us)	19.6	37	38.7	86	70.5	129	56.5	33.1	25.1
Frequency of exec. (kHz)	51	27	25.6	9.17	14.2	7.77	17.7	30.2	39.8
Maximum Power Consum. (W)	1	1	1.5	.4	.4	.9	.9	.6	.5

NA denotes information concerning this entry not available
NAP denotes entry not applicable to processor

Now these processor's performance can be evaluated directly. The performance is judged on the basis of speed, complexity and power consumption.

As far as power consumption is concerned, the processor that performed the best is the TMS32010. It requires very little in the area of external circuitry and its maximum power consumption is only .4 Watts. The LM32900 uses only .5 W (maximum) but requires much external circuitry (I/O processor and memory).

The least complex design was for the DSP56000. It had only the A\D externally. No additional control logic or memory was required. The TMS32025 also had limited external connections, but did require some extra select circuitry.

The fastest implementation is the DSP56000. It's execution time is only 19.6 us. Next was the LM32900 at 25.1 us. It should be noted that the LM32900's performance will not suffer as the memory requirement increases (because it has separate buses externally). The DSP56000's performance would be greatly slowed if it were necessary to use external memory instead of its internal banks.

The table indicates that the DSP56000 is the best suited for this algorithm. The LM32900 also has good performance, but the need to include external memory (with very short access time) adds complexity and cost to a relatively simple algorithm.

The TMS320 series performed well (especially with

regard to system simplicity). The 32010 is slower than the other two, but its power consumption is smaller. The 32020 needed external program memory, but executed quickly with relatively efficient code. The 32025 performed better than either of the other two.

The ADSP2100 also had a short execution time, but the system is quite complex. The UDPI 1 needed an I/O controller, but otherwise performed well for a processor with no external connection to its buses. The 7720's (MSM and S) had an implementation similar to the UDPI 1, but was much slower, facilitated only 8-bit I/O, but needed no latch for the alarm bit. The MSM7720 is of CMOS technology, thus its power consumption will be lower than the S7720 (NMOS fabrication).

The ZR43161 is a special case. It requires a host, which means that its performance cannot be judged independently of its host. It requires the host to direct program flow and start-up. This DSP is fairly efficient for operations on longer vectors (e.g. 128 taps instead of 16), but is not well suited to this algorithm (as the table indicates).

It should be noted that many aspects of some of these processors are not brought out by this algorithm. This is unavoidable in light of the wide variety of features available. It does give some indication as to the requirements for system development and efficiency of code.

4.0 Floating-Point Considerations

Many algorithms are developed in digital signal processing by humans with the assumption that the simple mathematical calculations involved are possible and can be carried out to the precision necessary to bring meaning to the result. After all, it is a simple matter to continue the calculation one more decimal place or include one more term in the series. The problem in using these algorithms is that accuracy of the calculations is fixed in hardware (or possibly in software but nonetheless fixed).

With a fixed-point word of 16 bits, only about four and one-half orders of magnitude can be represented. What if an algorithm calls for a number to be inverted or scaled by a factor of 100,000. Either of these cases may provoke an overflow. Maybe 16 bits is sufficient for accuracy (number of significant bits in a calculation) but it is often insufficient for the dynamic range (order of magnitude).

A system which allows for expanded range with the same number of bits is the floating-point representation[28]. This scheme calls for some bits to represent the mantissa and some bits the exponent (power of two). Additionally, this system can be implemented in such a way that all numbers are normalized (have the same radix point). This is very advantageous for ease of calculation because it eliminates the need for shifting the result of calculations to correct the radix point. Additionally, the result has the most

number of significant bits possible.

4.1 Floating-Point Standards

A point of contention is exactly how should numbers be represented. How many bits for each field and should the exponent field be signed or unsigned with an assumed offset or bias? The answer to these questions is provided by ANSI (American National Standards Institute) and IEEE (Institute of Electrical and Electronics Engineers)[29]. These organizations set standards that the industry can follow to introduce some conformity and transportability of code. The standard they have agreed upon is as follows.

For single precision numbers, the mantissa is 24 bits wide (signed) and the exponent is eight bits wide (unsigned) with a bias of +127 with -127 and 128 as reserved for special numbers. This means that to obtain the actual exponent, subtract 127 from the representation of the exponent. The maximum exponent is 127 and the minimum is -126. An exponent of 128 (represented as 255) is a special code for infinity or not-a-number. If the mantissa is $\pm$ 0, then it represents not-a-number, otherwise it represents $\pm$ infinity. An exponent of -127 (represented as 0) represents either 0 or an unnormalized number. As is expected, it represents 0 if the mantissa is 0.

Double precision is similarly defined except that the mantissa is 53 bits wide (signed) and the exponent is eleven bits (unsigned) with a bias of 1023.

4.2 Floating-Point Processors

The value of representing numbers this way may be very great in some digital signal processing algorithms, but so is its cost. Floating-point operations are very time consuming in traditional processors. For example, to add two floating-point numbers requires that the operand of greater magnitude first be shifted right until the two exponents match, then the addition can take place, then the result must be left shifted for normalization. Additionally, the shifting may have created an underflow in the operand or overflow in the result. Both of these conditions must be checked. This demonstrates the effort that must be exerted to perform one of these operations.

In a situation where time is of the essence, this may not be a practical way to represent numbers. The software involved is too time consuming. It is, however, possible to perform in hardware. A floating-point processor can be built that will perform all these tasks automatically without the need to fetch the instructions to do the same thing.

Such processors are currently being developed by at least two companies - AT&T and Motorola, Incorporated. These processors have an ALU which expects floating point operands exclusively and can perform the calculations much more efficiently than a software controlled calculation. They also support fixed-point calculations with a separate ALU (which the AT&T devices also use for address

generation).

These processors have many of the same characteristics that the fixed-point DSP's have but have the additional floating-point capability. The same tables that were used to compare DSP's are used to compare FPP's (Floating-Point Processors) with a few minor modifications to the software table (table 4-2). The round to ±infinity on overflow entry was eliminated because it is somewhat inherent to the representation. Similar reasons lead to the removal of the shifting entries and the removal of the extra sign bit in multiplication.

Some entries concerning floating-point operations are added. These include IEEE standard adherence, fixed-point support, conversion to and from the two representations, single-extended precision (more accuracy), convert to single precision capability, and byte addressing capability. All of these are fairly self-explanatory except memory byte-addressable. This simply indicates that the processor may access a single 32-bit floating-point word as two or four smaller fixed-point words if desired. The processors evaluated are :

1) DSP32 (AT&T)[30]

2) DSP32C (AT&T)[31]

3) DSP96001 (Motorola, Inc.)[32]

4) DSP96002 (Motorola, Inc.)[33]

This information is given in tables 4-1 through 4-4.

	DSP 96001	DSP 96002	*DSP32	*DSP32C
Separate program and data buses	N	N	N	N
On-chip program ROM	Y	Y	Y	Y
On-chip program RAM	Y	Y	Y	Y
On-chip data RAM	Y	Y	Y	Y
On-chip data ROM	N	N	Y	Y
On-chip EPROM	N	N	N	N
On-chip A/D	N	N	N	N
On-chip D/A	N	N	N	N
Memory transfer strobe bits	N	N	N	N
Dedicated I/O pins	Y	N	Y	Y
Bus grant to host	Y	Y	N	Y
Requires host	N	N	N	N
Accepts interrupts	Y	Y	N	Y
CMOS technology	Y	Y	N	Y
TTL compatible	Y	Y	N	N
Low power mode	Y	Y	N	N
Built-in clock	Y	Y	N	N

NA denotes information regarding this entry unavailable
NAP denotes entry not applicable to this processor
* not Harvard Architecture

Table 4-1 Hardware Considerations of FPP's

	DSP 96001	DSP 96002	DSP32	DSP32C
Multiply accumulate in floating-point	Y	Y	Y	Y
Simultaneous fetch of two operands	Y	Y	N	N
Branch on I/O status	N	N	Y	Y
Block move	N	N	N	N
Divide primitive	Y	Y	N	N
FFT or store backwards	Y	Y	N	Y
Round to infinity on overflow	Y	Y	Y	Y
Repeat next instr.	Y	Y	N	N
Loop counters	Y	Y	Y	Y
Circular indexing	Y	Y	Y	Y
Double precision support	Y	Y	N	N
IEEE format	Y	Y	N	N
Fixed-point	Y	Y	Y	Y
Convert fixed-float	Y	Y	Y	Y
Convert float-fixed	Y	Y	Y	Y
Single extended prec.	Y	Y	Y	Y
Convert to single	Y	Y	Y	Y
Memory byte-addr.	N	N	Y	Y

NA denotes information concerning entry not available
NAP denotes entry not applicable to processor

Table 4-2 Software Considerations of FPP's

	DSP 96001	DSP 96002	*DSP32	*DSP32C
On-chip program ROM	32	32	*512	*512
On-chip program RAM	512	512	*1K	*1K
On-chip data ROM	1K	1K	*512	*512
On-chip data RAM	1K	1K	*1K	*1K
Program memory space	4G	8G	*16K	*4M
Data memory space	8G	16G	*16K	*4M
Data word length	32	32	32	32
External memory access + (ns)	NA	NA	50	15
Stack depth	15	15	1	1
Number of index registers	8	8	22	22
Clock cycle limits from	NA	NA	8	0
to (MHz)	NA	NA	25	50
Number of parallel ports	2	0	1	1
Number of serial ports	1	0	1	1
Number of pins	NA	NA	100	133
Maximum power usage (W)	NA	NA	2.3	1.9
Temperature range from	NA	NA	0	0
to (C)	NA	NA	115	70
Most recent information	'88	'88	'88	'88
Availability date	4Q'89	3Q'89	-	NA

+ Address valid to data valid
at maximum clock frequency
NA denotes information concerning entry not available
NAP denotes entry not applicable to processor
- denotes currently available
* not Harvard Architecture

Table 4-3 Quantitative Features of FPP's

	DSP 96001	DSP 96002	DSP32	DSP32C
Assembler	Y	Y	Y	Y
Simulator	Y	Y	Y	Y
Emulator	Y	Y	Y	Y
High level language	Y	Y	Y	Y
Evaluation board	NA	NA	NA	NA

NA denotes information concerning this entry not available
NAP denotes entry not applicable to processor

Table 4-4 Support Tools for FPP's

Because the products are not yet available, not enough information is at hand to perform an evaluation similar to the algorithm developed for the DSP's. The information in the tables stands as the only means of comparison and evaluation.

For example, the information available on the DSP96000's does not provide information concerning its clock frequency or the number of clock cycles necessary to perform each instruction. It does claim to be software compatible with the DSP56000 series, but this does not lend enough information to evaluate its performance even if a routine could be written.

The DSP32 information does not include any indication as to the number of clock cycles each instruction requires.

Enough information is available to write a routine with a fair degree of confidence in its integrity, but no execution time could be estimated.

5.0 Conclusions

The Harvard architecture really makes the execution time as short as is possible for a given clock frequency. A routine that requires a large memory will be more quickly executed with one of the processors that use the Harvard Architecture even for external memory accesses (such as the LM32900 and ADSP2100).

The results of the adaptive filter algorithm development showed that the DSP5600 is the most efficient for the task. It is a very versatile processor and will be the most desirable in terms of speed and complexity when the routine can be placed in the internal ROM. If that is not possible or practical, then perhaps it will not perform significantly better than some of the other processors.

As far as power consumption is concerned, the most promising DSP is the TMS32010. A CMOS version is available, and for low speed situations, its power consumption will be even less (because of its CMOS fabrication). It is a fairly efficient processor as far as coding is concerned and may well suit many low power applications. The MSM7720 is also a low-power CMOS device with approximately the same speed as the TMS32010, but is much less efficient in coding. Both processors have the ability to stand alone (with no external memory in limited situations), but the TMS32010 has the opportunity for expansion of its address space. The MSM7720 does not.

The available DSP's have been presented. Their features have been tabulated and their performance in executing an adaptive filter algorithm have been analyzed. The tables of features give the reader the chance to find a processor (or possibly several) which has the features that would be the most applicable to his situation. All that he needs to know are the constraints of the problem (memory requirements, speed requirements, power consumption requirements, and possibly complexity requirements) to use the table to find a list of processors. He may then study the system developed in this report to gain some insight into the complexity of the system that he will design.

Because these processors are so new, it is convenient to have them all presented together without bias to be able to evaluate them fairly. Not every feature imaginable has been covered, but ones that are common to most designs have been.

For future development, the tables need to be completed (which requires gathering new information from the manufacturers) and expanded as new processors become available. In this way, a means of judging a prospective processor can be evaluated before it is experimented with physically. It may do away with the need to experiment at all.

Finally, floating-point processing may become applicable to a wider variety of dedicated systems soon.

The performance of these processors may well be a natural extension of the system of evaluation developed in this report.

In conclusion, these processors offer a wide variety of possibilities for the designer whose scope is limited by the bandwidth of traditional microprocessors. It is the author's hope that this report will offer some assistance to him when he tries to evaluate them himself.

Acknowledgements

I would like to give thanks to Sandia National Laboratories, Albuquerque, New Mexico for providing partial funding for this project. I would also like to thank Dr. Donald H. Lenhert for his support and guidance on this project. Additionally, I would like to recognize my parents, Scott and Joyce, for their unending devotion and encouragement to me. Special thanks go to Clare Calman for providing inspiration in all facets of my life. Finally, I would like to thank all of my friends in the Department of Electrical and Computer Engineering and at K-State in general for their wonderful contribution to my life's education.

References

[1]H. Troy Nagle and Charles L. Phillips, Digital Control System Analysis and Design, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1984), p.78.

[2]Nasir Ahmed and T. Natarajan, Discrete Time Signals and Systems, (Reston, Va.: Reston Publishing company, Inc., 1983), pp. 121-3.

[3]K. Steiglitz, " Equivalence of Digital and Analog Signal Processing ", Digital Signal Processing, ed Lawrence R. Rabiner and Charles M. Rader, (New York: IEEE Press, 1972).

[4]Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), p.18.

[5]Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), pp 1-16.

[6]Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), pp 8-10.

[7]William F. Leahy, Microprocessor Architecture and Programming, (New York: John Wiley and Sons, 1977).

[8]Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), pp 128-9.

[9]DSP56000 Digital Signal Processor User's Manual, Motorola, Inc., 1986.

[10]TMS320C25 User's Guide, Texas Instruments Incorporated, 1986.

[11]TMS32020 User's Guide, Texas Instruments Incorporated, 1986.

[12]TMS32010 User's Guide, Texas Instruments Incorporated, 1985.

[13]LM32900 Digital Signal Processor Reference, National Semiconductor, 1986.

[14]UDPI 01 Universal Digital Signal Processor, ITT Semiconductors, 1985.

[15]DSP Microprocessor ADSP2100, Analog Devices, 1986

[16]ZR34161 Vector Signal Processor Engineering Data, Zoran Corporation, 1986.

[17]S7720 Signal Processing Interface Technical Manual, Gould AMI Semiconductors, 1985.

[18]MSM7720 General Purpose Digital Signal Processor, OKI Semiconductor, 1986.

[19]J. Rittenhouse, " The Intel 2920 ", Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), chapter 3.

[20]HD64180 8-Bit High Integration Microprocessor User's Manual, Hitachi America Limited, 1985.

[21]WE DSP16 Digital Signal Processor for Military Applications, AT&T, 1988.

[22]Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), pp 21-2.

[23]Nasir Ahmed and T. Natarajan, Discrete Time Signals and Systems, (Reston, Va.: Reston Publishing company, Inc., 1983), chapter 7.

[24]Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), pp 23-5.

[25]Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), p 38.

[26]Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), pp 37-42.

[27]C.F.N. Cowan and P.M. Grant, Adaptive Filters, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), p 44.

[28]Signal Processor Chips, ed David Quarmby, (Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1985), pp 18-20.

[29]" ANSI/IEEE Std. 488.2-1987 ", IEEE Standard Codes, Formats, Protocols, and Common Commands, (New York: IEEE Press, 1988), pp 93-95.

[30]WE DSP32 Digital Signal Processor, AT&T, 1988

[31]WE DSP32C Digital Signal Processor, AT&T, 1987

[32]DSP96001, 96-Bit General-Purpose Floating-Point Digital Signal Processor, Motorola, Inc., 1988.

[33]DSP96002, 96-Bit General-Purpose Floating-Point Digital Signal Processor, Motorola, Inc., 1988.

APPENDIX A

Implementations

A.0 Introduction

This appendix describes the system developed for each DSP to perform the Widrow's Adaptive Linear Predictor algorithm. The DSP's evaluated are (in the same order they appear in the tables of chapter 2) :

1) DSP56000
2) TMS32025
3) TMS32020
4) TMS32010
5) ZR34161
6) S7720 and MSM7720
7) UDPI 1
8) ADSP 2100
9) LM32900

The S7720 and the MSM7720 are considered together because they are software and pin-for-pin compatible. Also, they operate over the same range of clock frequencies.

For each DSP, this appendix contains information concerning hardware implementation (including a block diagram), usage of memory and registers, initial conditions, software and evaluation of implementation.

The hardware implementation section for each processor outlines the logic of the system, i.e. it describes how the system reads data into the DSP and how the DSP signals an

alarm. Details about select circuitry, memory arrangement, and I/O functions are addressed. It includes a hardware block diagram that shows the system circuitry at a high level. It is not a schematic, but part numbers may be cited to show what is necessary for implementation. It shows only necessary connections, all else is not shown. It is not necessarily the minimal system to accomplish that logic. A simple PLA may well replace all of the discrete logic in most of the implementations. If a block diagram does not show a clock circuit, then that DSP has an internal oscillator. The system for each DSP is designed with the assumption that speed of execution is the most important factor. Thus if a reasonable amount of hardware can be used to replace a slower implementation in software, it is done. Thus some implementations have more hardware than is absolutely necessary to perform the task.

The initial conditions section lists all assumptions made in addition to the assumptions that all of the systems face (downloading of constants, start conversion of first data, etc.).

The assembly code section is a listing of code that was written to perform the algorithm at hand. It must be understood that this code is not tested and not necessarily the most efficient possible. This test is designed only to give some idea as to performance. The code is commented for ease of reading. All algorithms follow the same format and use the same variable names.

The format is as follows:

1) input next sample from A/D
2) store that in a memory location called FM
3) perform an FIR filter on the COEFFS and DATA arrays, call the result FM
4) compute the error in this estimate as EM = FM - GM
5) store this value at the top of the ERROR array
6) update the COEFFS array according to the formula

$$B_{M,K} = B_{M-1,K}*U + V*E_M*F_{M-K} ,$$

where V and U are constants and $B_{M,K}$ is a coefficient of the M^{th} iteration and index number K (equal to 0 through 15)

7) perform the moving average filter on the ERROR array, call the result QM
8) square QM, call it QM2
9) compare QM2 against THETA (a known, constant threshold)
10) set the alarm according to QM2 and THETA
11) update the ERROR and DATA arrays

Finally, an evaluation of the system is given. This may include commentary on ease of comprehension of manufacturers information, complexity of programming, complexity of I/O and memory circuitry, results of the implementation and possibly areas that DSP might be better suited for use.

A.1 DSP56000

A.1.1 Hardware Implementation

This implementation is quite straightforward and needs little additional hardware (see figure A-1). Since the DSP56000 has adequate on-chip program ROM and data RAM and ROM, no external memory is required. All that is necessary is I/O hardware. Since the DSP has two independent I/O ports, one can be configured to control the A/D and the other to read the data from the conversion. Port C is configured to have 3 output pins and one input pin. The output pins are $\overline{OE}$, $\overline{SC}$, and the alarm bit. The input pin reads the status of the EOC pin of the A/D.

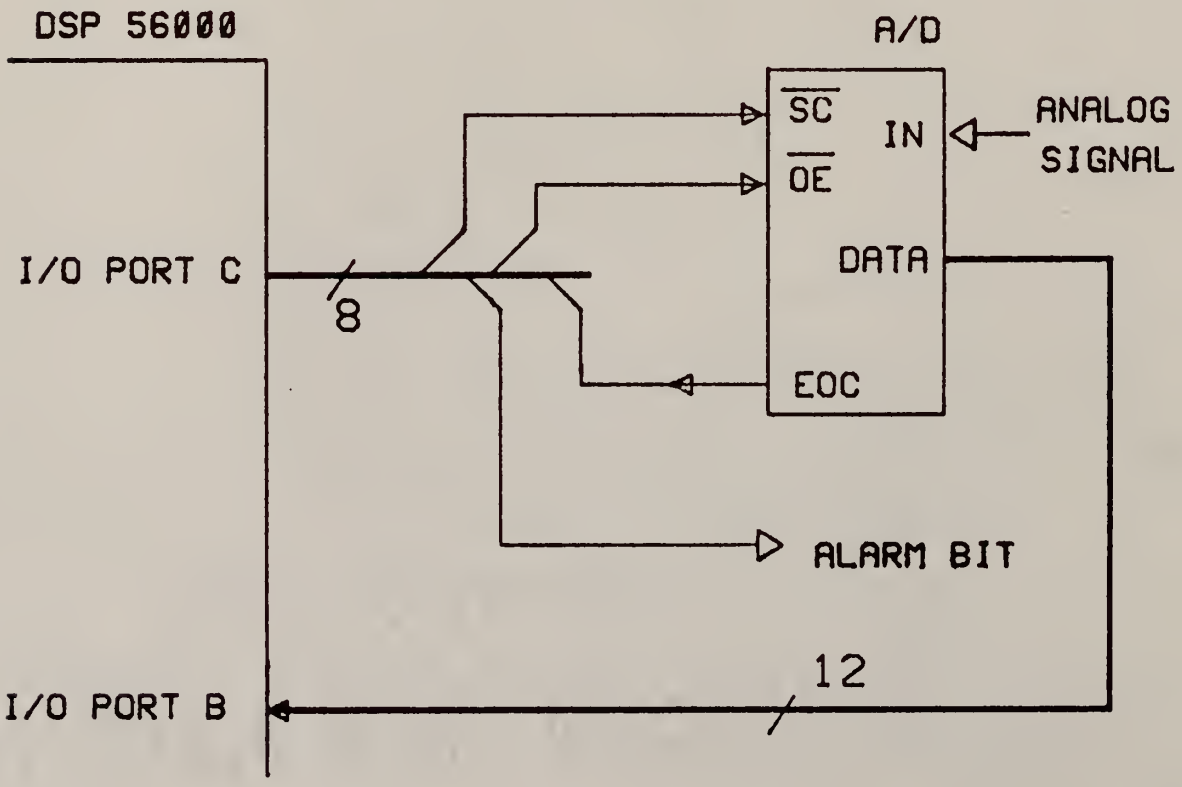

Figure A-1 Block Diagram for DSP56000 Implementation

A.1.2 Usage of Memory and Registers

Index Register	Value Pointed To
R0	COEFFS ARRAY X: (0 - 15)
R4	DATA ARRAY Y: (0 - 15)
R6	ERROR ARRAY X: (16 - 31)

Variable	Location
GM	X:32
QM2	Y:16
U	X:256 (ROM)
V	Y:256 (ROM)
THETA	X:257 (ROM)

A.1.4 Initial Conditions

Port C is set up as control I/O . Its physical address is in X memory at $FFE5. Bit 0 controls $\overline{SC}$, bit 1 controls $\overline{OE}$, bit 2 reads EOC, and bit three is the alarm bit.

Port B is set up to read the result of the conversion. Its physical address is X:$FFE4.

A.1.4 Assembly Listing for Adaptive Algorithm

```
MAIN    JSET 2,X:$FFE5,MAIN { wait for EOC = 1 }
        BCHG 1,X:$FFE5      { set OE = 0 }
        MOVE X:$FFE4, X1    { put result of conversion
                                    into X1 reg }
        BCHG 1,X:$FFE5 { set OE = 1 }
        BCHG 0,X:$FFE5 { set SC = 0 }
```

```
          BCHG 0,X:$FFE5 { set SC = 1 }
{ Now that the next sample has been read and the next
conversion started, perform an FIR filter using the
previous sixteen data samples and the current value of the
coefficient vector to predict what the present data value
should have been. }
FIR       MOVE R0,R1     { reg R1 points to coeffs }
          MOVE R4,R5     { reg R5 points to data }
          NOP            { wait for R5 to get data }
          CLR  A         { clear acc A }
               X:(R1)+,X0  { X0 gets first coeff }
               Y:(R5)+,Y0  { Y0 gets first data }
          REP  #$10      { do next 16 times }
          MAC  X0,Y0,A   { FIR filter }
               X:(R1)+,X0     { X0 gets next coeff }
               Y:(R5)+,Y0     { Y0 gets next data }
          ASL  A         { shift out extra sign bit }
          MOVE A1, X0    { save it temporarily }
{ Now calculate the error }
          MOVE X1,A0     { A0 gets FM }
               R7,R6     { R7 points to error array }
          SUB  X0,A0     { EM = FM - GM }
          MOVE A0,X:(R7)+ { store EM }
{ Now perform moving average filter on last 16 errors }
          MOVE X:(R7)+,Y1     { load next error sample }
          REP  #$0F      { do next fifteen times }
          ADD  Y1,A0     { add next error to sum }
```

```
            X:(R7)+,Y1  { load next error }
        REP  #$04       { divide by 16 ( ASR 4 times ) }
        ASR  A
{ The value in A0 is QM. Now compare its square against a
known threshold }
SQUARE  MOVE  A0,X0
        MPY  X0,X0,A    { A gets QM2 }
        ASL  A          { remove extra sign bit }
        MOVE A1,Y:QM2   { and save it }
{ Now update the weighting coefficients }
        MOVE R4,R5      { R5 points to data }
             X:(R0),Y1  { Y1 gets first coeff }
        MOVE R0,R1      { R1 points to coeffs }
             X:(R6),X0  { X0 gets EM }
        MOVE X:U,X2     { X2 gets U }
             Y:(R5)+,X1 { R5 points to second coeff }
        MOVE Y:V,Y2     { Y2 gets V }
        DO   #$10, ENDUP    { DO loop 16 times }
        MPY  Y2,X0,A    { A gets V*EM }
        ASL  A
        MOVE A1,Y0
        MPY  X2,Y1,A    { A gets U*(old coeff) }
        MAC  Y0,X1,A { A gets U*(old coeff)+V*EM*F(M-K)}
             Y:(R5)+,X1     { X1 gets next data }
        ASL  A
        MOVE A1,X(R1)+  { store this as the new coeff }
        MOVE X:(R1),Y1  { load up next coeff }
```

```
ENDUP          { end update DO loop }
{ Now compare QM2 against the threshold }

THRESH    MOVE X:THETA,A0      { A0 gets theta }
               Y:QM2,Y1         { Y1 gets QM2 }
          CMP  Y1,A0      { if QM2 > theta then set alarm }
          JGT  SETALM
NOALM     BCLR 3,X:$FFE5 { else reset alarm }
          JMP  ROLL
SETALM    BSET 3,X:$FFE5 { set alarm }
{ Now push every element of the data and error arrays down
one position ( which translates to moving each on up to the
next higher address ). Lose the last element and put the
latest data sample on top of the data array }
ROLL      MOVE X:(R7)-2,X2     { dummy moves to set the in-}
               Y:(R5)-2,X1     { regs to bottom of arrays }
          MOVE X:(R7)-1,X1     { put last usable elements }
               Y:(R5)-1,X2        { into regs }
          DO   #$08, END       { do loop 8 times }
          MOVE X:(R7)+2,X0     { load present element and }
               Y:(R5)+2,Y2     { and point to dest of last }
          MOVE X1,X:(R7)-1     { store these and point to }
               X2,Y:(R5)-1      { dest of other vals in regs}
          MOVE X0,X:(R7)-2     { store these and point to }
               Y2,Y:(R5)-2        { source of next elements }
          MOVE X:(R7)-1,X1        { and retrieve these }
Y:(R5)-1,X2        { elements }
END       { end of DO loop }
```

```
{ Now return for the next data sample }
          JMP  MAIN
```

A.1.5 Evaluation

This processor required 50, 24-bit program words to realize the main loop of the algorithm. Its execution time (with a clock frequency of 20.5 MHz) is 19.6 us. This corresponds to a sample rate of 51,000 samples per second.

This processor performs quite well. It is easy to implement because of its vast I/O and it has enough on-chip data, program and constant memory to perform a much more complicated algorithm than the one implemented. It is configurable to suit many applications, perhaps more than one at once.

Its pipelined instruction architecture is easily followed but has an unexpected side-effect - if an index register is modified in some manner other than pre-decrement or post-increment, then that modification will not have completed by the execution cycle of the next instruction. In other words, an instruction which loads an index register with a new value or increments or decrements by more than one must not be used to reference memory in the next instruction. Other than that little aberration, the instruction format is quite well structured and programming quite simple.

A.2 TMS32025

A.2.1 Hardware Implementation

This implementation is quite simple (see figure A-2). Since the DSP has sufficient memory on-chip for the adaptive routine and start-up, no external memory is necessary. All that is needed is some control for the A/D. This is accomplished by the I/O select output pin ($\overline{IS}$). If an I/O operation is to be performed, then the state of the R/$\overline{W}$ determines if the operation performed is start the next conversion or read the result of the last conversion. The DSP waits for EOC to go active (high) by monitoring the $\overline{BIO}$ (branch on I/O status) input pin. The DSP has an external flag bit (XF) which serves as the alarm bit. This bit is fully software controlled. The DSP is set to the microcomputer mode (MC/$\overline{MP}$ input = 1) which makes the internal ROM usable.

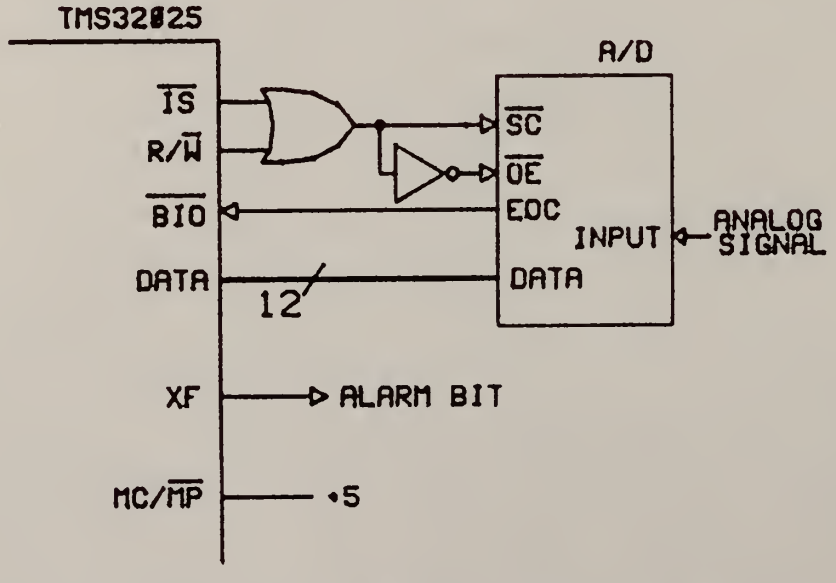

Figure A-2 Block Diagram for TMS32025 Implementation

A.3.2 Usage of Memory and Registers

ADDR	VALUE
96 - 111	COEFFS ARRAY
512	FM
513 - 528	DATA ARRAY
529 - 544	ERROR ARRAY
545	QM
546	QM2
547	U
548	V
549	THETA
550 - 565	TEMP1 ARRAY
569	TEMP
570	FIFTEEN (contains 15)
571	SIXTEEN (contains 16)
572	ERREND (ending addr of error array + 1)
573	ENDDAT (end addr of data + 1)
574	FOURTEEN (contains 14)

A.2.3 Initial Conditions

It is assumed that the initial coefficients have been downloaded from program ROM to data RAM. Also, auxiliary register one has been set to point at the data array and auxiliary register three has been set to point at the array of previous iteration errors. Block zero of the on-chip RAM

has been configured as data memory (it contains the COEFFS array). Also the shift mode of the product register has been set to left one bit. This is done because all of the multiplies are signed thus the redundant sign bit needs to be shifted out.

A.2.4 Assembly Listing for Adaptive Algorithm

```
INPUT     BIOZ      INPUT     { wait for EOC }
          IN    A/D,FM    { read result from A/D }
          OUT   A/D,FM    { dummy write to start conv }

{ Now that the latest data value has been found, compute
the output of the FIR filter with present coefficients }
          CNFP      {make coeffs program memory }
          LARP      { point to first aux reg }
          LAR   1, DATA   { load it with beg addr of data }
          ZAC       { zero the accumulator }
          RPT   FIFTEEN   { now perform FIR }
          MAC   COEFFS,*+
          APAC      { PUT RESULT IN ACC }
{ Now compute the error of the estimate of this sample }
          SUB   FM    { EM = FM - GM }
          NEG
          SACL  EM    { store at top of error array }

{ now update the weighting coefficients }
          LAR   0,FIFTEEN       { setup loop control }
          LAR   1,DATA    { aux reg 1 points to data array }
```

```
          CNFD      { make coeffs data memory }
          LAR  2,COEFFS  { aux reg 2 points to coeffs }
{ start the loop that performs the update :
          B(M,K) = U*B(M-1,K) + V*EM*F(M-K)  }
UPDATE    LT   EM
          MPY  *+,2 { mult EM by FM-K and make aux reg 2
                      the next default auxiliary reg }
          SPH  TEMP { store this result temporarily }
          LT   TEMP
          MPY  V    { multiply it by V }
          PAC       { and put it in accum }
          LT   U
          MPY  *    { multiply U and B(M-1,K) }
          APAC      { and add it to accum }
          SACL *+,0 { store it as new coeff, make aux reg 2
point to the next coeff and make AR0 the next aux reg }
          BANZ UPDATE,*-,1 { if AR0 <> 0 then return for
                             next coeff, else continue on }

{ now perform the moving average filter on the last 16
error samples }

          MAR  *,3  { make AR3 ( points to error array )
                      next aux reg }
          LAR  0,FIFTEEN      { setup loop control }
          ZAC
MAF       ADD *+,0  { add in next error and make AR0 AR }
```

```
          BANZ MAF,*-,3   { decr AR0 and point to AR3 }
          RPTK THREE      { now divide by 16 }
          SFR
          SACL QM    { store average as QM }
{ now compute QM * QM }
          SQRA QM
          SPH  QM2
{ now compare QM2 against a threshold }
          LAC QM2
          SUB THETA
          BGEZ NoALM      { if QM2 > threshold, then set
                                 alarm }
          SXF
          BRA  ROLL       {  and jump to buffer updating }
NoALM     RXF             { else reset alarm }

{ now update the data and error arrays }
ROLL      LAR  3,ERREND
          RPT  FOURTEEN
          DMOV *-
          LAR 3,ENDDAT
          RPT FIFTEEN
          DMOV *-

{ now return for next sample }
          BRA  INPUT
```

A.2.5 Evaluation

This processor required 55, 16-bit program words to realize the main loop of the algorithm. Its execution time (with a clock frequency of 40 MHz) is 37 us. This corresponds to a sample rate of 27,000 samples per second.

This implementation is quite simple and fairly efficient. Coding the TMS32025 is simple except for the fact that one must keep track of which section of memory the operands are. This DSP allows for many types of transfers between memories and redeclarations of memory types to increase throughput. If a programmer forgets exactly what his memory map looks like at any given time, it may lead to catastrophe. It is a very flexible and efficient system, but it requires some mindfulness on the part of the programmer.

The distinction between data, program, and I/O spaces makes accessing all of these types of memory over the same set of buses efficient. Efficiency can be markedly improved if the routine is masked into the internal ROM.

The abundance of addressing modes, the capability to simultaneously access two operands, and the data move functions combine to give a wide variety of processing applications.

A.3 TMS32020

A.3.1 Hardware Implementation

This implementation is identical to the TMS32025 except that this implementation includes a program ROM (see figure A-3). This DSP has sufficient internal RAM for the variables and constants needed to perform the adaptive routine but lacks any on-chip ROM to hold the start-up and filtering routines. Thus a 128 x 16 program ROM is connected to the external buses.

The external circuitry is controlled in a similar fashion to that of the TMS32025 except that the program ROM is selected by a low level on the $\overline{PS}$ output (program memory select line).

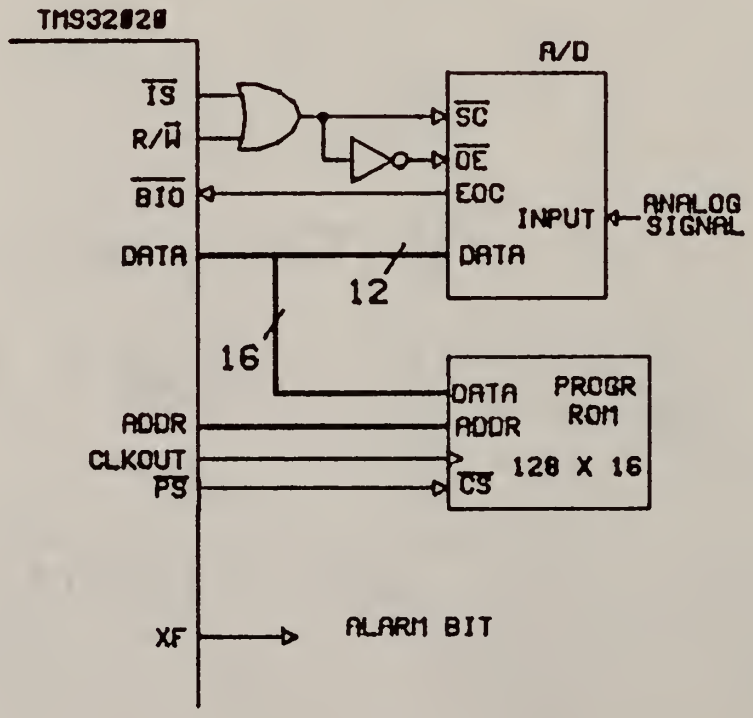

Figure A-3 Block Diagram for TMS32020 Implementation

A.3.2 Usage of Memory and Registers

ADDR	VALUE
96 - 111	COEFFS ARRAY
512	FM
513 - 528	DATA ARRAY
529 - 544	ERROR ARRAY
545	QM
546	QM2
547	U
548	V
549	THETA
550 - 565	TEMP1 ARRAY
569	TEMP
570	FIFTEEN (contains 15)
571	SIXTEEN (contains 16)
572	FOURTEEN (contains 14)
573	ERREND (end addr of error + 1)
574	ENDDAT (end addr of data + 1)

A.3.3 Initial Conditions

It is assumed that the initial coefficients have been downloaded from program ROM to data RAM. Also, auxiliary register one has been set to point at the data array and auxiliary register three has been set to point at the array of previous iteration errors. Block zero of the on-chip RAM has been configured as data memory (it contains the COEFFS

array). Also the shift mode of the product register has been set to left one bit. this is done because all of the multiplies are signed thus the redundant sign bit needs to be shifted out.

A.3.4 Assembly Listing for Adaptive Algorithm

```
INPUT     BIOZ      INPUT     { wait for EOC }
          IN   A/D,FM     { read result from A/D }
          OUT  A/D,FM     { dummy write to start conv }

{ Now that the latest data value has been found, compute
the output of the FIR filter with present coefficients }
          CNFP      {make coeffs program memory }
          LARP      { point to first aux reg }
          LAR  1, DATA   { load it with beg addr of data }
          ZAC       { zero the accumulator }
          RPT  FIFTEEN   { now perform FIR }
          MAC  COEFFS,*+
          APAC      { PUT RESULT IN ACC }
{ Now compute the error of the estimate of this sample }
          SUB  FM   { EM = FM - GM }
          NEG
          SACL EM   { store at top of error array }

{ now update the weighting coefficients }
          LAR  0,FIFTEEN      { setup loop control }
          LAR  1,DATA    { aux reg 1 points to data array }
```

```
          CNFD      { make coeffs data memory }
          LAR  2,COEFFS  { aux reg 2 points to coeffs }
{ start the loop that performs the update :
          B(M,K) = U*B(M-1,K) + V*EM*F(M-K)  }
UPDATE    LT   EM
          MPY  *+,2 { mult EM by FM-K and make aux reg 2
                      the next default auxiliary reg }
          PAC       { load result into acc then store }
          SACH TEMP { store this result temporarily }
          LT   TEMP
          MPY  V    { multiply it by V }
          PAC       { and put it in accum }
          LT   U
          MPY  *    { multiply U and B(M-1,K) }
          APAC      { and add it to accum }
          SACL *+,0 { store it as new coeff, make aux reg 2
point to the next coeff and make AR0 the next aux reg }
          BANZ UPDATE,*-,1 { if AR0 <> 0 then return for
                             next coeff, else continue on }

{ now perform the moving average filter on the last 16
error samples }

          MAR  *,3  { make AR3 ( points to error array )
                      next aux reg }
          LAR  0,FIFTEEN      { setup loop control }
          ZAC
```

```
MAF       ADD *+,0  { add in next error and make AR0 AR }
          BANZ MAF,*-,3  { decr AR0 and point to AR3 }
          RPTK THREE     { now divide by 16 }
          SFR
          SACL QM   { store average as QM }
{ now compute QM * QM }
          SQRA QM
          PAC
          SACH QM2
{ now compare QM2 against a threshold }
          LAC QM2
          SUB THETA
          BGEZ NoALM     { if QM2 > threshold, then set
                                alarm }
          SXF
          BRA  ROLL      {  and jump to buffer updating }
NoALM     RXF            { else reset alarm }

{ now update the data and error arrays }
ROLL      LAR  3,ERREND
          RPT  FOURTEEN
          DMOV *-
          LAR 3,ENDDAT
          RPT FIFTEEN
          DMOV *-

{ now return for next sample }
```

BRA INPUT

A.3.5 Evaluation

This processor required 58, 16-bit program words to realize the main loop of the algorithm. Its execution time (with a clock frequency of 20 MHz) is 77.4 us. This corresponds to a sample rate of 12,900 samples per second.

The implementation of this processor is practically identical to that of the TMS32025 except that the 32020 has no on-chip ROM, thus it needs some program memory. It shares all of the other qualities of the 32025 (good and bad).

A.4 TMS32010

A.4.1 Hardware Implementation

This DSP has sufficient on-chip memory to facilitate all of the storage necessary to start-up and execute the adaptive routine. The I/O (A/D and alarm latch) circuitry is based on the 74139 decoder (see figure A-4). This decoder is selected any time an IN or OUT instruction is executed. This is accomplished by ANDing the $\overline{WE}$ (OUT instruction strobe) and $\overline{DEN}$ (IN instruction strobe). The device selected by the decoder is determined by the two bit I/O address PA1 PA0. If an OUT to the A/D is performed, the $\overline{SC}$ pin is strobed. If an IN of the A/D is performed, the $\overline{OE}$ pin is asserted low causing the data to put on the bus. If

an OUT to the alarm latch is performed, the clock on the latch is strobed latching in the LSB of the data bus.

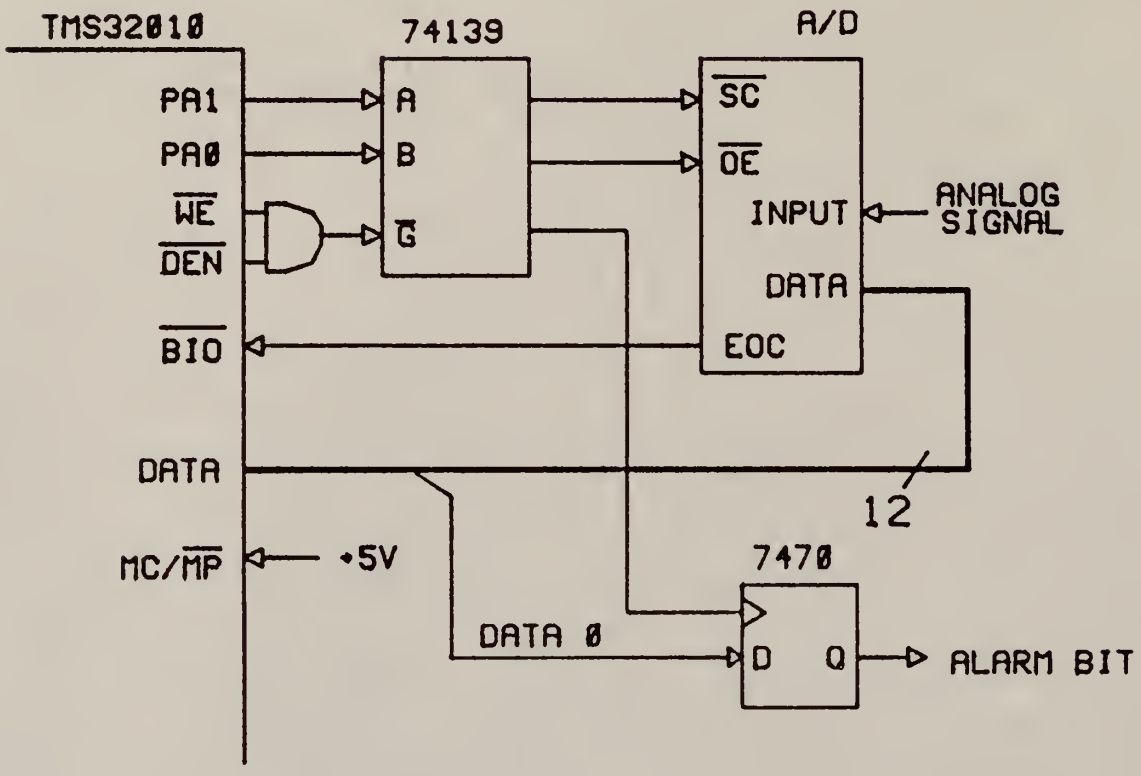

Figure A-4 Block Diagram for TMS32010 Implementation

A.4.2 Usage of Memory and Registers

ADDR	VALUE
0 - 15	COEFFS ARRAY
6	FM
7 - 22	DATA ARRAY
23	EM
23 - 38	ERROR ARRAY
39	QM
40	QM2
41	U
42	V
43	THETA
44	TEMP1
45	TEMP2
46	FIFTEEN (contains 15)
47	SIXTEEN (contains 16)
48	DATADDR (address of data array)
49	COEFADDR (address of coefficient array)
50	EMADDR (addr of EM)
51	ERREND (EMADDR + 14)
52	ENDDAT (DATADDR + 14)
53	GM

A.4.3 Initial Conditions

It is assumed that the initial coefficients have been downloaded from program ROM to data RAM. Also, the DP (data page pointer) register has been set to zero.

A.4.3 Assembly Listing for Adaptive Algorithm

```
INPUT   BIOZ    INPUT     { wait for EOC }
        IN      A/D,FM    { read result from A/D }
        OUT  A/D,FM    { dummy write to start next
                  conversion of incoming analog signal }
{ Now that the latest data value has been found, compute
the output of the FIR filter with present coefficients }
        LAR  1,DATADDR { AR1 points to data array }
        LAC  COEFADDR  { put the address of the top of
the coeffs array in a temporary memory location }
        SACL TEMP2
        ZAC       { zero the accumulator }
        LAR 0,SIXTEEN,0     { run loop sixteen times and
                              make AR1 current aux reg }
FIR     MAR  *-,1 { decrement AR0 and point to AR1 }
        LT   *+   { load T with next data element }
        SAR  AR1,TEMP1 { store pointer to data array }
        LAR  AR1,TEMP2 { load pointer to coeff array }
        MPY  *+,0      { multiply data and coeff and
                      increment pointer to coeff array }
        SAR  AR1,TEMP2 { store coeff pointer }
        LAR  AR1,TEMP1 { load data pointer }
```

```
          APAC      { accumulate result of multiply }
          BANZ FIR  { if not done, return for next }
          SACH GM,1 { store result and shift out redundant
                                    sign bit }
{ now calculate error in estimate as EM = FM - GM }
          LAC  FM
          SUB  GM   { EM = FM - GM }
          SACL EM   { store at top of error array }

{ now update the weighting coefficients by the algorithm :
    B(M,K) = B(M-1,K)*U + EM*F(M-K)*V }
          LAR  0,SIXTEEN      { setup loop control }
          LAR  1,DATADDR    { aux reg 1 points
                                    to data array }
          LAC  COEFADDR
          SACL TEMP1     { store coeff pointer }
          MAR  *,0  { make AR0 current aux reg }
UPDATE    MAR  *-,1 { decr loop counter and point to AR1 }
          LT   EM
          MPY  *+   { multiply EM and F(M-K) ; incr data
                                pointer }
          PAC
          SACH TEMP2,1   { and store it }
          LT   TEMP2     { to multiply by V }
          MPY  V
          PAC       { put it into acc for future use }
          SAR  1,TEMP2   { store data pointer }
```

```
          LAR  1,TEMP1   { load coeff pointer }
          LT   U
          MPY  *     { mult current coeff and U }
          APAC       { add that to previous product }
          SACH *+,0  { store it as new coeff and make AR0
                               current aux reg }
          SAR  1,TEMP1   { store coeff pointer }
          LAR  1,TEMP2   { reload data pointer }
          BANZ UPDATE    { do until all coeffs updated }

{ Now perform a moving average on the error samples. Call
the output QM. Next, square that and compare it against a
given threshold to determine if the input sequence is
white. If it is, reset the alarm bit, else set the alarm }

          LAR  0,SIXTEEN      { loop variable }
          LAR  1,EMADDR  { AR1 points to error array }
          MAR  *,0   { make AR0 current aux reg }
          ZAC        { clear summation }
MAF       MAR  *-,1  { decr loop counter ; point to error }
          ADD  *+,0  { add next error sample and point to
                               loop counter }
          BANZ MAF
{ Now divide sum by sixteen to obtain average }
          SFR
          SFR
          SFR
```

```
          SFR
          SACL QM    { and store it }
          LT   QM
          MPY  QM    { square it }
          SACH QM2,1      { save it discarding MSbit }
          LAC  QM2
          SUB  THETA      { and compare against thresh }
          BGEZ NOALM
          LAC  ONE
          SACL ALARM      { set alarm for thresh crossing }
          OUT  LATCH,ALARM     { send to latch }
          BRA  ROLL       { branch to array update }
NOALM     ZAC             { else reset alarm }
          SACL ALARM
          OUT  LATCH,ALARM
{ now update the data and error arrays }
ROLL      LAR  0,FIFTEEN
          LAR  1,ERREND
          MAR  *,0
{ First update error array }
ROLLEM    MAR  *-,1  { decr loop counter ; point to error }
          DMOV *-,0  { push last element and decr error
                       pointer ; point to loop counter }
          BANZ ROLLEM     { continue until top element }
{ Next update data array being careful to push down the
latest data sample as well (FM) }
          LAR  0,SIXTEEN       { loop counter }
```

```
          LAR  1,ENDDAT    { point to end of data array }
ROLLFM    MAR  *-,1     { decr loop count ; point to data }
          DMOV *-,0
          BANZ ROLLFM

{ now return for next sample }
          BRA  INPUT
```

A.4.5 Evaluation

This processor required 86, 16-bit program words to realize the main loop of the algorithm. Its execution time (with a clock frequency of 25 MHz) is 86 us. This corresponds to a sample rate of 11,600 samples per second.

This processor is quite simple to master. Its I/O format is somewhat confusing, but overcomable. It offers a variety of addressing schemes, but is limited by the fact that it has only two index registers. Couple that with the fact that one of these is also used as loop control and it becomes apparent that an inadequacy in this area exists. In general, the code is easy to write and to follow.

The presence of internal memory is quite attractive and is large enough to perform simple algorithms. It does have a fairly large address space, lending itself to expansion. Its advantages over its successors (TMS32020 and TMS32025) are few and far between.

A.5 ZR34161

A.5.1 Hardware Implementation

Since the ZR34161 has no reset vector or sufficient program flow instructions, it requires the aid of a simple host processor (see figure A-5). The duty of this processor is to provide the starting address of the adaptive routine in the ZR34161's external memory to the processor. Upon a reset, control over the DSP's buses is given to the host. The host then can write instructions directly into the DSP's instruction FIFO (memory mapped to 306 h) . The host should give the DSP the instruction JMPI START, where START is the beginning address of the routine.

After that, the DSP will act independently until it must make a decision as to what state it should put the alarm latch. At this point, the DSP will request the host to take control over its buses, retrieve the information necessary to make the decision, and finally send the appropriate branching instruction. The DSP will perform the setting of the latch and continue from there.

The host directs the DSP by writing instruction words into the instruction FIFO. Once it has control over the buses, it waits for the DSP to signal a request for next instruction word (asserting $\overline{RD}$ low) then it outputs the word and sends the DSP a data strobe signal ($\overline{DSTB}$). This continues until the DSP de-asserts the bus request signal. At this point the host will de-assert the bus acknowledge signal and give full control of the buses to the DSP.

The remainder of the circuitry is dedicated to performing initialization and the adaptive routine. The DSP's internal memory is for operands only, therefore external data memory is needed in addition to the program ROM. The C/$\overline{D}$ pin distinguishes between program and data memory fetches. When the buses are controlled by the host, this pin is tristated. To ensure that no unintentional writes are made to the data bus by either of the memories, the select line of each memory is pulled up to disable them during host control.

If C/$\overline{D}$ is high, then a program fetch is done. If it is low, then it enables a decoder which selects the appropriate device (data RAM, A/D functions, or alarm latch). The decision is based on the two most significant bits of the address bus used (8 and 9) and the $\overline{WR}$ signal. The function of the outputs of this decoder are pretty straightforward except for the alarm latch and the output enable of the A/D. If the alarm latch is selected, that pin of the decoder serves as a clock or the latch. It will latch in the LSB of the data bus. If the output enable is selected, then wait states are generated until EOC goes high. This is accomplished by ORing $\overline{OE}$ and the EOC together. If the EOC is low and $\overline{OE}$ is low, then the $\overline{SUS}$ input to the DSP will be low. This causes the DSP to wait another clock cycle and check again. If it is now high, then it will perform the read of the data. Thus when EOC goes high, the processor will read the data.

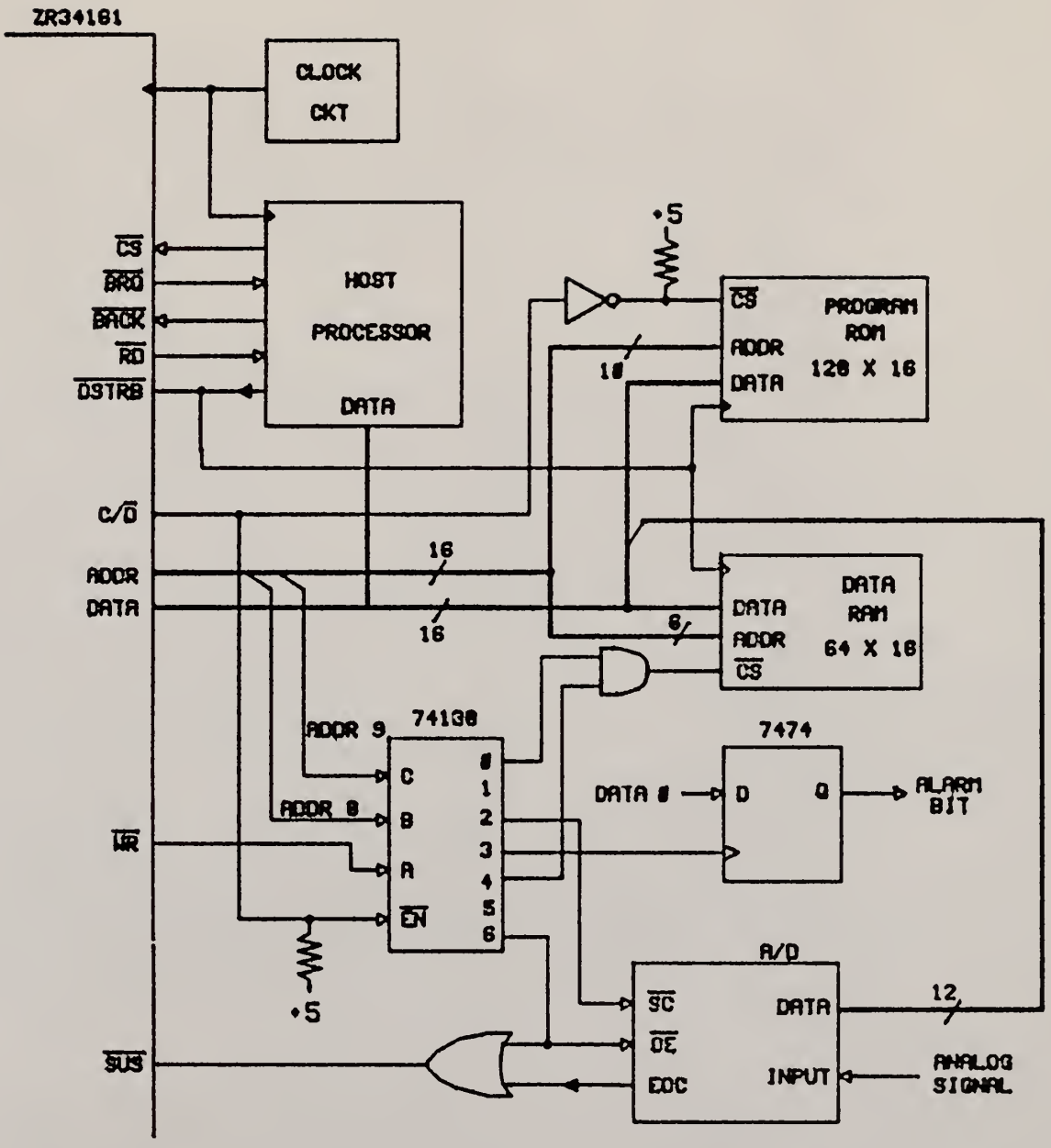

Figure A-5 Block Diagram for ZR34161 Implementation

A.5.2 Usage of Memory and Registers

Address	Contents
0	0
1	1
2	-1
3 - 18	COEFF array
19	FM
20 - 35	DATA array
36	EM
36 - 51	ERROR array
52	QM2
53	V
54	U
55	THETA
128	A/D
192	alarm latch

A.5.3 Initial Conditions

It is assumed that the host has given the DSP the starting address, the DSP has setup memory to contain the necessary constants and has downloaded the initial coefficients into the DATA memory (RAM). This can be done by manipulating the SIN/COS look-up table. Also, the first conversion has been started.

A.5.4 Assembly Listing for Adaptive Algorithm

```
LOOP      LD   NMPT = 1, MDF = 2   A/D  { input data }
```

```
          ST   NMPT = 1, MDF = 2   FM   { store it }
          ST   NMPT = 1, MDF = 2   A/D  { dummy write to
                                          start next conversion }
{ Now clear accumulator and perform the FIR filter on the
last 16 data samples }
          LD   NMPT = 1, MDF = 2,  ZERO
          ACCR NMPT = 1        { clear accumulator }
          LD   NMPT = 16, ADF = 2, DATA { load data array }
          MLTR NMPT = 16, ADF = 2, COEFFS    { FIR }
          STI  NMPT = 1, STR = 1, GM    { store acc at GM }
{ Now calculate the error in this sample }
          LD   NMPT = 1, MDF = 2, GM    { put GM in RAM }
          MLTR NMPT = 1, ADF = 2, NEGONE { negate it }
          ADDR NMPT = 1, ADF =2, FM     { EM = FM - GM }
          ST   NMPT = 1, MDF = 2, EM  { put on top of error
                                        array }
{ Now update the coefficients }
          LD   NMPT = 16,MDF = 2, COEFFS  { load coeffs }
          MULTR NMPT = 16, CN = 1, U  { multiply all coeffs
                                        by U }
          ST   NMPT = 16, MDF = 2, COEFFS  { store them }
          LD   NMPT = 16, MDF = 2, DATA { load data array }
          MLTR NMPT = 16, CN = 1, EM   { scale data by EM }
          MLTR NMPT = 16, CN = 1, V    { scale that by V }
          ADDR NMPT = 16,MDF = 2, COEFFS {  and add this to
                                 COEFFS*U to get new COEFFS }
          ST   NMPT = 16, MDF = 2, COEFFS
```

```
{ Now perform moving average filter on past 16 error
samples to get QM. }
          LD   NMPT = 16, MDF = 2, ERROR   { load errors }
          ACCR NMPT = 16       { acc gets sum of errors }
          STI  NMPT = 1, STR = 1, QM2
{ divide by 16 to average sum }
          LD   NMPT = 1, MDF = 2, QM2
          SCLT NMPT = 1, ADF = 2, SHF = 4    { /16 }
          ST   NMPT = 1,MDF = 2, QM2
{ Now square the average and compare it against the known
threshold THETA }
          MLTR NMPT = 1, MDF = 2, QM2   { square it }
{ Now multiply QM2 by -1 and add THETA. The DSP cannot
decide where to branch to, thus it will halt and wait for
the host to instruct it. }
          MLTR NMPT = 1, MDF = 2, NEGONE
          ADDR NMPT = 1, MDF = 2, THETA
          HLT
ALM       LD   NMPT = 1, MDF = 2, ONE   { set alarm }
          ST   NMPT = 1, MDF = 2, ALARM
          JMPI ROLL      { proceed to buffer update }
NOALM     LD   NMPT = 1, MDF = 2, ZERO   { set alarm }
          ST   NMPT = 1, MDF = 2, ALARM
{ Now push each of the elements of the data and error
arrays down one position. Accomplish this by copying the
arrays into internal RAM locations then copying them back
offset by one position. }
```

```
ROLL      LD   NMPT = 15, MDF = 2, EM  { copy error array }
          ST   NMPT = 15, MDF = 2, EM + 1  { store it }
          LD   NMPT = 16, MDF = 2, FM   { load data
                                          ( including FM ) }
          ST   NMPT = 16, MDF = 2, DATA
{ Now return for next data sample }
          JMPI LOOP
```

A.5.5 Evaluation

This processor required 114, 16-bit program words to realize the main loop of the algorithm. Its execution time (with a clock frequency of 20 MHz) is 70.5 us. This corresponds to a sample rate of 14,200 samples per second. These statistics are somewhat misleading because of the uncertainty in time and program memory requirement of the host processor.

This processor is really ill-suited for stand alone operation. Its real value is that of a peripheral to a controlling processor. In this situation, it would have been more efficient for the host and the VSP to share a common memory space and for the host to continually write instructions into the VSP's instruction FIFO. The host could have controlled the conversion of the data and signaled the VSP when it needed to perform vector operations. The results of that system would not have comparable to the results of the other DSP's. Nonetheless, due to factors such as its inability to branch

conditionally, that is the most efficient system for this DSP.

The vector operations themselves are very efficient. The fact that it can perform the same operation on the entire vector data in its memory greatly reduces execution time. The more operations performed on the same vector, the more efficient it becomes because there is no need to move the data back and forth between on-chip and off-chip memory. This implementation does not lend itself to utilizationof this processor's capabilities. It is the only processor available that directly supports complex arithmetic. In a situation where complex arithmetic is necessary (which are numerous in the digital signal processing area), this processor would most likely outshine all others.

For this example, however, it leaves much to be desired. The need to use a host to direct program flow makes it difficult to evaluate its performance. Its external memory, while not large in scope, must be extremely fast (45 ns access time at full speed). It will not be a low power or simple system.

A.6 S7720 and MSM7720

A.6.1 Hardware Implementation

This DSP has no external connection to either its address or data buses. It does have adequate memory on-chip to perform the initialization and the adaptive routine

which makes external memory unnecessary. Unfortunately, the only I/O the processor has is an 8-bit bi-directional register and two software controlled output pins. One of these pins serves as the alarm bit (see figure A-6). The other acts as a control signal to an external I/O control unit.

This unit accepts the control bit and performs the I/O control operations (strobe $\overline{SC}$ and strobe $\overline{OE}$). If P1=0, then the controller starts the next conversion. It then waits for EOC to go high before it enables the output of the A/D. At this point the data is latched into the three 4-bit registers. The controller then sends a data strobe signal ($\overline{WR}$) to the DSP and the DSP reads the 8 most significant bits of the conversion. The controller then waits for P1 to return to 1. That is a signal to write the remaining 4 bits into the I/O register. When the controller gets this signal, it de-asserts $\overline{OE}$. This causes the registers to read a new value. The most significant register will read the contents of the least significant register. The middle register will read all zeros and the least significant register will not load (it has been disabled by the de-assertion of $\overline{OE}$). Then the controller strobes the DSP once again to signal data valid.

The controller then waits for the DSP to set P1 = 0. At this point it will start the process over again.

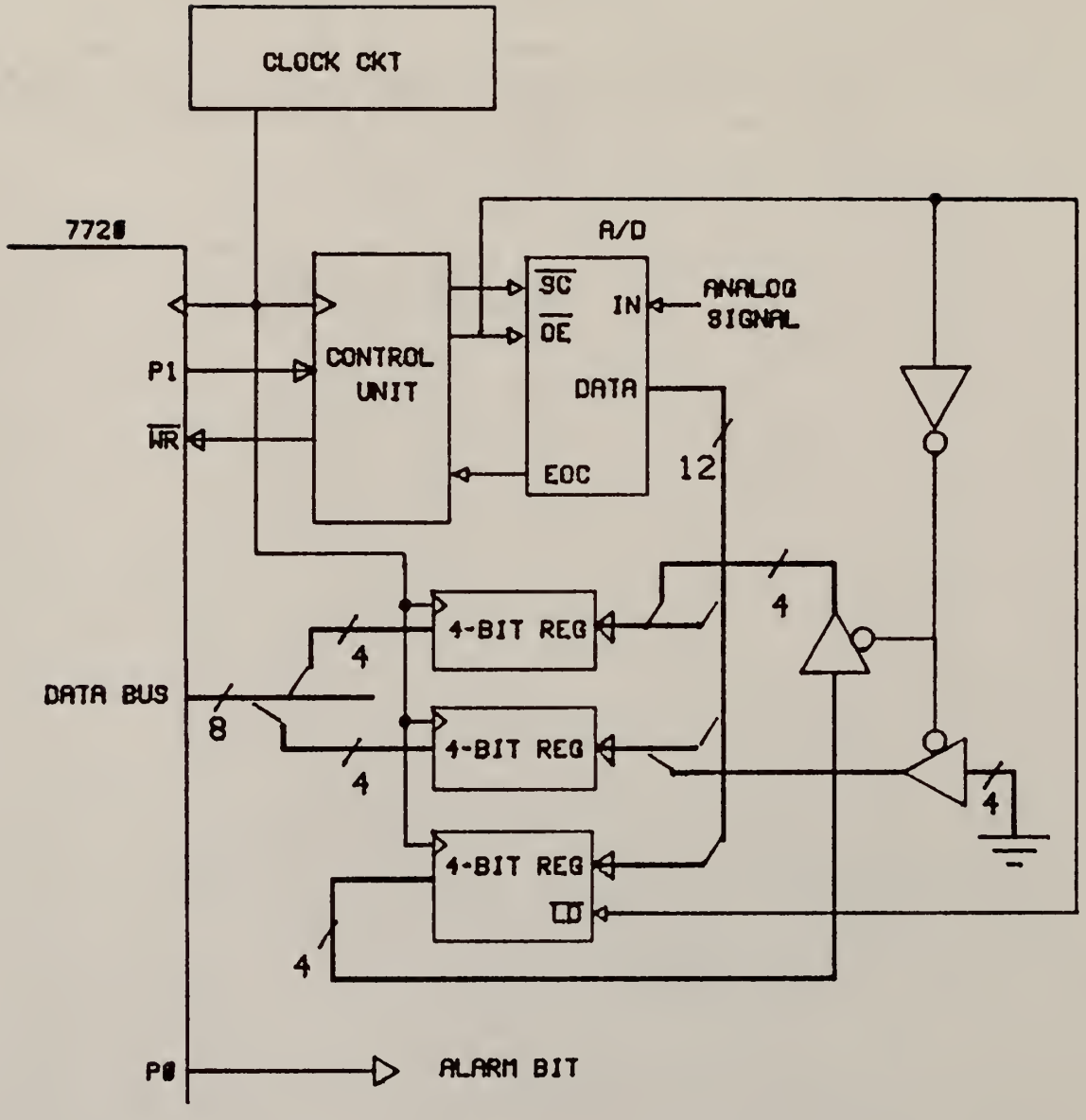

Figure A-6 Block Diagram for 7720 Implementations

A.6.2 Usage of Memory and Registers

Address	Contents
0 - 15	data array
16	FM
17	U
32 - 47	V(16)
64 - 79	coeffs array
80	EM
80 - 95	error array
96	QM2
97	THETA

A.6.3 Initial Conditions

It is assumed that the original coefficients as well as the constants U and V have been downloaded from data ROM into the dual bank RAM. Also, P1 has been set to zero (which starts the first conversion of the incoming data) and P0 has been set to zero (to indicate a " no alarm " condition. Also, locations 32-47 have all been set to V.

A.6.4 Assembly Listing for Adaptive Algorithm

```
{ Once the system has found EOC = 1, it will read the
result 8 bits at a time. Thus wait for the parallel port to
signal it has read the first 8 bits ( DRS bit in status
register = 1 ) and then wait for it to read the remaining
bits ( DRS bit = 0 ). }
          LDI   @TR,#$1001
INPUT     OP    MOV  SR,@ACCA
```

```
        OP   MOV  TR,@NON
             AND  ACCA,IDB  {is DRS =1 ? if not recheck}
        OP   MOV  TR,@NON
             SUB  ACCA,IDB
        JNZA INPUT
{ Now set P1 = 1 to signal that the first 8 bits have been
read and that DSP is waiting for the remainder of the
result }
        OP   MOV  @ACCA,SR  { acc gets status reg }
        LDI  @TR,1          { set for 16 bit I/O and
                              P1 = 1 for wait for }
        OP   OR   ACCA,TR
        OP   MOV  ACCA,@SR  { store new status reg }

{ Now wait for DRS to return to 0 }
WAIT1   OP   MOV  SR,@ACCA
        OP   MOV  TR,@NON
             AND  ACCA,IDB
        OP   MOV  TR,@NON
             SUB  ACCA,IDB
        JZA  WAIT1
{ Store the new data and reset P1 to signal the controller
to start the next conversion }

GETDATA LDI  @DP,#$10  { point to FM }
        OP   MOV  @RAM,DR   { store input }
        OP   MOV  @ACCA,SR  { acc gets status reg }
        LDI  @TR,0          { reset P1 = 0 to start }
```

```
          OP   OR   ACCA,TR          {  conversion }
          OP   MOV  ACCA,@SR   { store new status reg }
{ Now perform the FIR filter on the last sixteen data
samples with the current coefficients }
FIRINIT   OP   MOV  @NON,A
               XOR  ACCA,IDB   { clear acc A }
          OP   MOV  ACCB,@NON
               XOR  ACCB,IDB   { clear acc B }
{ Do the first tap of the FIR filter }
          LDI  @DP,#$40    { point to data and coeff arrays
                             and set DP6 = 1 }
          OP   MOV  @KLM,MEM  { load first data and coeff }
          OP   ADD  ACCB,N    { put result in acc }
          OP   ADC  ACCA,M
               DPINC          { point to next operands }
{ Now do the rest of the taps. A bit in the status reg
tells when 16 increments have been done to the data
pointer, so use that as terminating condition of the loop }
FIR       OP   MOV  @KLM,MEM  { load next operands }
          OP   ADD  ACCB,N    { accumulate result }
          OP   ADC  ACCA,M
               DPINC          { point to next operands }
          JDPL0     FIR       { do next tap }
{ Now compute the error in this estimate }
          OP   MOV  @TR,MEM   { pull FM out of TR }
               SUB  ACCA,IDB  { acc A gets GM - FM
               M2       { modify DP to point to first data }
```

```
        OP   CMP  A     { negate result of subtraction }
        OP   INC  A
             M5           { point to EM }
        OP   MOV  @MEM,A  { store EM }
{ Now update the weighting coefficients }
        OP   MOV  @DR,A   { store EM for future use }
             DPINC, M7    { point to U }
        OP   MOV    @TR,MEM   { store U in TR }
             DPDEC, M2    { point to coeffs array }
        OP   MOV  ACCA,@NON
             XOR  ACCA,IDB  { clear accumulators }
        OP   MOV  ACCB,@NON
             XOR  ACCB,IDB
{ update the first coefficient outside of the loop }
        OP   MOV  @K,TR    { K gets U }
        OP   MOV  @L,MEM   { L gets coeff }
        OP   ADD  ACCB,N   { acc = U*BM }
        OP   ADC  ACCA,M
        OP   MOV  @K,DR    { K gets EM }
             M6            { point to V }
        OP   MOV  @L,MEM   { L gets V }
             M2            { point to data }
        OP   MOV  @L,M     { L gets EM*V }
        OP   MOV  @K,MEM   { K gets data ( F(M-K) ) }
             M6            { point to coeff }
        OP   ADD  ACCB,N   { acc = U*BM + EM*V*F(M-K) }
        OP   ADC  ACCA,M
```

```
          OP   MOV  @MEM,A     { store as new first coeff }
               DPINC           { point to next coeff }
{ Now do the remaining coefficients }
UPDATE    OP   MOV  ACCA,@NON
               XOR  ACCA,IDB   { clear accumulators }
          OP   MOV  ACCB,@NON
               XOR  ACCB,IDB
          OP   MOV  @K,TR      { K gets U }
          OP   MOV  @L,MEM     { L gets coeff }
          OP   ADD  ACCB,N     { acc = U*BM }
          OP   ADC  ACCA,M
          OP   MOV  @K,DR      { K gets EM }
               M6              { point to V }
          OP   MOV  @L,MEM     { L gets V }
               M2              { point to data }
          OP   MOV  @L,M       { L gets EM*V }
          OP   MOV  @K,MEM     { K gets data ( F(M-K) ) }
               M6              { point to coeff }
          OP   ADD  ACCB,N     { acc = U*BM + EM*V*F(M-K) }
          OP   ADC  ACCA,M
          OP   MOV  @MEM,A     { store as new first coeff }
               DPINC           { point to next coeff }
          JMP  DPL0 UPDATE
{ Now compute the average of the last 16 error samples }
{ The DP reg already points to the error array }
          OP   MOV  ACCA,@NON
               XOR  A,IDB      { clear acc A }
```

```
          OP   ADD  A,MEM      { start sum with EM }
               DPINC      { point to next error }
SUM       OP   ADD  A,MEM      { add in next error }
               DPINC
          JMP  DPL0 SUM
{ Now divide the sum by 16 }
          OP   SHRl A
          OP   SHRl A
          OP   SHRl A
          OP   SHRl A
          OP   MOV  @MEM,A     { store as QM }
{ Now square the average and compare against a known
threshold }
          OP   MOV  @KLM,MEM   { K gets QM ; L gets QM }
          OP   MOV  @A,M       { A gets QM2 }
               DPINC           { point to THETA }
          OP   SUB  A,MEM      { subtract theta from QM2 }
          JMP  SA0,SETALM      { if QM2 > THETA then set an
                                     alarm }
          OP   MOV  @A,SR      { else reset alarm }
          LDI  TR,#$FE
          OP   MOV  @NON,TR    { clear P0 ( alarm bit ) }
               AND  A,IDB
          OP   MOV  @SR,A
          JMP  ROLL { proceed to buffer update }
SETALM    OP   MOV  @A,SR
          LDI  TR,1
```

```
          OP   MOV  @NON,TR
               OR   A,IDB      { set P0 ( alarm bit )  }
          OP   MOV  @SR,A
{ Now push all of the elements of the data and error arrays
down one position }
ROLL      LDI  DP,0       { point to data array }
          OP   MOV  @TR,MEM   { put first data int TR }
               DPINC          { point to next data }
LOOP2     OP   MOV  @DR,MEM   { put it into DR }
          OP   MOV  @MEM,TR  { replace it with newer data }
               DPINC          { point to next }
          OP   MOV  @TR,MEM
          OP   MOV  @MEM,DR   { store previous sample }
               DPINC          { point to next }
          JMP  LDP0 LOOP2      { do until done }
 { Put the latest sample (FM) on top of the array }
          OP   MOV  @TR,MEM   { get FM }
          LDI  DP,0           { point to top of data }
          OP   MOV  @MEM,TR
{Now do the same for the error array }
          LDI  DP,#$50      { point to error array }
          OP   MOV  @TR,MEM   { put first error int TR }
               DPINC          { point to next error }
LOOP3     OP   MOV  @DR,MEM   { put it into DR }
          OP   MOV  @MEM,TR  { replace it with new error }
               DPINC          { point to next }
          OP   MOV  @TR,MEM
```

```
        OP    MOV  @MEM,DR    { store previous sample }
              DPINC           { point to next }
        JMP   LDP0 LOOP3       { do until done }
{Now return for next data sample }
        JMP   INPUT
```

A.6.5 Evaluation

This processor required 99, 23-bit program words to realize the main loop of the algorithm. Its execution time (with a clock frequency of 8.2 MHz) is 129 us. This corresponds to a sample rate of 7,940 samples per second.

This processor is quite difficult to deal with. Its assembly language is extremely cryptic (even as assembly languages go). Code is very hard to follow (even for the author) and difficult to write. Once the structure of the processor is fully understood, it becomes easy to build instructions. The addressing is particularly difficult to understand. After its mysteries have been exposed, it becomes quite an efficient means of simultaneous access of two operands with data pointer modification. The pipelined structure makes possible some very interesting instructions (such as loading an operand on to a bus then having the ALU act upon the bus).

The I/O is too limited for a processor that has no access to external memory. This processor would be better suited for a situation in which a host continually supplied

it with data. This is still a hindrance because of the 8-bit I/O port and no hardware controlled protocol lines. In general, there simply is not enough allocation made for I/O.

This processor does allow for minimal systems. It has enough on-chip memory to perform many tasks and a CMOS version would allow for a low power system at low frequency. While its power consumption specifications are not extremely low, it is practically the only power drain in the system.

A.7 UDPI 1

A.7.1 Hardware Implementation

This implementation is complicated by the lack of I/O signals on the processor. Since no external connection to the UDPIl's address bus exists, a means must be used to externally determine which function to perform. Since the DSP has no external connection to its address bus, no external memory is applicable or necessary. The internal memory will serve adequately for this algorithm.

The external circuitry will use the fact that the order of the I/O functions for the adaptive routine is always the same each time through the main loop. Thus a counter is used to keep track of the next function to perform (see figure A-7). It must be in sync with the processor. Upon a reset, the DSP must, in its initialization routine, perform a start next conversion.

This is due to the fact that the counter is loaded with the value that corresponds to that function on a reset. From then on, the external circuitry is expecting an output enable, followed by a alarm latch clock, followed by a start next conversion, etc. The DSP does have input pins that it can read. Thus one of these pins is fed by the EOC output of the A/D. The DSP can read the state of this input until it becomes active (high) and then read the result.

To ensure that the output on the A/D has time to become valid before it is read by the DSP, a wait state is generated by the D flip-flop. This effectively delays the read function by one clock cycle.

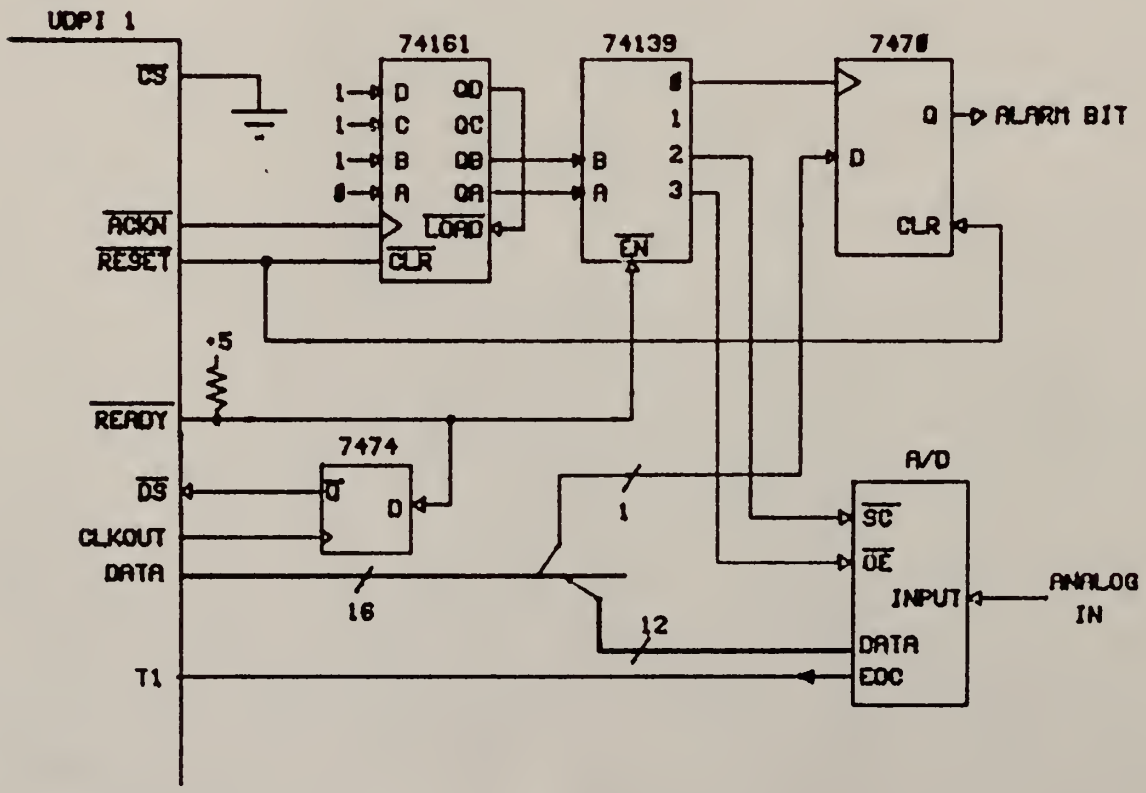

Figure A-7 Block Diagram for UDPI 1 Implementation

A.7.2 Usage of Memory and Registers

Address Counter	Contents
AC10	address of FM
AC11	address of present data
AC12	address of error array
AC13	address of threshold
AC14	general purpose
AC20	general purpose
AC21	current coefficient address
AC22	address of coeff array
AC23	address of constant V
AC24	general purpose

Bank 1

Address	Contents
0	FM
1 - 16	data array
17	EM
17 - 32	error array
220 (ROM)	threshold
221 (ROM)	1
222 (ROM)	2^{12}
223 (ROM)	2^{15}
224 (ROM)	U

Bank 2

Address	Contents
0 - 15	coefficient array
220 (ROM)	V
221 (ROM)	1
222 (ROM)	2^{12}
223 (ROM)	2^{15}

A.7.3 Initial Conditions

It is assumed that the system has been reset and a write to the alarm latch has been done. This is done to ensure that the external circuitry is in the correct state. Additionally, it is assumed that the first conversion has been started. Also, all of the constants used have been calculated and stored in the appropriate memory location.

A.7.4 Assembly Listing for Adaptive Algorithm

```
MAIN    JT   T1,NEXTIN      { wait for EOC = 1 }
        JMP  MAIN
NEXTIN  SRDY      { enable the output of the A/D }
        HCIP  { wait for parallel port to load result }
        MOVP AC10,PDBF      { store result as FM }
        CRDY      { disable output of A/D }
        SACK      { incr control counter }
{ Now do a dummy write to the alarm latch to make the
control counter contain the state for the start next
conversion. Then enable the decoder to send the S to the
A/D. }
```

```
          CLRA
          SRDY
          HCOP      { dummy write to latch }
          CACK      { clock control counter }
          SACK      { to start next conversion }
          CRDY      { disable decoder }
{ Now perform the FIR filter on the last 16 data samples }
          MOV  #AC11,INPUT    { load AC11 with addr of top
                                of data array }
          MOV  #AC21,COEFF    { load AC21 with addr of top
                                of error array }
          MOV  #LC0,#$10      { load loop counter with 16 }
FIR       MOVMR IAC11,IAC21   { load multipliers with
                                values and incr index regs}
          MAA  A              { acc gets data * coeff }
          DJNZ LC0,FIR        { do until loop counter = 0 ;
                               loop counter pre- decr every
                               time this instr. executed }
{ Now compute the error in this estimate }
          MOV  #AC14,ONE      { AC14 gets addr of const=1 }
          MOVMR AC10,AC14     { get FM into adder by
                                multiplying it by 1 }
          MSA  A    { acc gets FM - GM }
          MOVH A,AC12    { store it  as EM }
{ Now update the coefficient vector }
          MOV  #AC11,DATA     { point to top of data }
          MOV  #AC21,COEFFS   { point to first coeff }
```

```
          MOV   #AC14,U          { load AC14 with addr of U }
          MOV   #AC24,V          { load AC24 with addr of V }
          MOV   #LC0,#$10        { load loop counter with 16 }
UPDATE    MOVMR AC12,AC24        { load EM and V }
          MUL   B                { multiply them put in acc B}
          MOVMR B,IAC11          { reload result of previous }
               { product and get next data; incr pointer }
          MUL   B          { B = V*EM*F(M-K) }
          MOVMR AC14,AC21        { load coeff and U }
          MAA   B     { B = V*EM*F(M-K) + BM*U }
          MOVH  B,IAC21    { store new coeff and incr point }
          DJNZ  LC0,UPDATE       { do all coeffs }
{ Now perform moving average filter on error samples to
obtain QM }
          MOV   #AC24,#AC12    { AC24 points to error array }
          CLR   B
          MOV #AC14,FACTOR  { FACTOR is addr of the scaling
factor which will do the division by sixteen as the sum is
being calculated. That factor is 2^12 }
          MOV   #LC0,#$16
MAF       MOVMR AC14,IAC24     { load factor and error and
                                 incr error pointer }
          MAA   B     { perform sum }
          DJNZ  LC0,MAF
{ Now square this and compare against a known threshold }
          MOVH  B,AC20
          MOVMR AC20
```

```
        MUL  B          { B = QM*QM }
        MOVE #AC24,FACTOR1  { FACTOR1 is addr of location
containing constant 2^15 }
        MOVMR AC13,AC24     { load threshold }
        MCL  B     { compare QM2 against threshold }
        JS   NOALM      { if threshold >QM2 then NOALM}
        CLR  A
        INC  A
        SRDY       { enable decoder ( latch clock = 0 ) }
        HCOP
        MOVPH PDBF,A      { output a one to set alarm  }
        CRDY            { clock latch }
        SACK
        CACK            { clock control counter }
        JMP  ROLL       { proceed to buffer update }
NOALM   CLR  A
        SRDY       { enable decoder ( latch clock = 0 ) }
        HCOP
        MOVPH PDBF,A      { output a 0 to clear alarm  }
        CRDY            { clock latch }
        SACK
        CACK            { clock control counter }
{ Now push all of the elements of the data and error arrays
down one position }
ROLL    MOV  #AC11,#AC10    { AC11 points to FM }
        MOV  #AC14,#AC12    { AC14 points to EM }
        MOV  #AC24,ONE
```

```
{ First do the data array }
          MOVMR IAC11,AC24    { load first data ( FM ) }
          MUL  A
          MOV  #LC0,#$0F
LOOP1     MOVMR AC11,AC24     { load next data }
          MOVL A,IAC11 { store previous and point to next }
          NOP            { wait for acc transfer }
          MUL  A         { load last data into acc A }
          DJNZ LC0,LOOP1
{ Now do the error array }
          MOVMR IAC14,AC24    { load first error ( EM ) }
          MUL  A    { move it to acc }
          MOV  #LC0,#$0F
LOOP2     MOVMR AC14,AC24     { load next error }
          MOVL A,IAC14   { store previous and incr point }
          NOP            { wait for transfer }
          MUL  A         { move last error to acc }
          DJNZ LC0,LOOP2
{ Now return for next data sample }
          JMP  MAIN
```

A.7.5 Evaluation

This processor required an unknown number of 16-bit program words to realize the main loop of the algorithm. It is unknown because the information given in the data book used is insufficient to determine this statistic. The execution time is known and is (with a clock frequency of 20 MHz) is 56.5 us. This corresponds to a sample rate of

17,700 samples per second.

This processor does not lend itself to stand alone implementations. The lack of external memory space was most likely decided upon to facilitate the multiple index register addressing scheme. Larger data and program spaces would make that scheme much more difficult to implement. The addressing works well for vector entities (such as the data and coefficient arrays) but is clumsy when applied to constants and temporary locations.

The I/O is adequate if the type of peripherals is limited. The I/O port is not extremely "smart" and the need for additional software controlled protocol will severely limit the throughput of this DSP for data transfer intensive algorithms.

Once data are inside the processor, operating on them is quite straightforward and efficient. The lack of simultaneous ALU operation and data fetch limits the throughput of this DSP. The dual accumulators and multiple loop counters make it rather pleasant to deal with. Better shifting capabilities would enhance performance.

A.8 ADSP2100

A.8.1 Hardware Implementation

This implementation takes advantage of the fact that the ADSP2100 has separate program and data buses. Because it has no internal memory, external memory must hold the

boot-up and the adaptive routine as well as the data memory.

Because the processor is capable of simultaneous fetches of operands on the data and program buses, a RAM is connected to the program buses. Its selection is distinguished from the ROM by the PMDA active high output pin of the processor (see figure A-8).

The I/O is controlled by the output of a 74138. Bits 6 and 7 of the data memory address bus and the data read signal determine which function is selected - RAM, $\overline{OE}$, $\overline{SC}$, or alarm latch. The decoder is selected by the data strobe ($\overline{DMS}$). If the alarm latch is selected, the value on the least significant bit of the data data bus will be latched in.

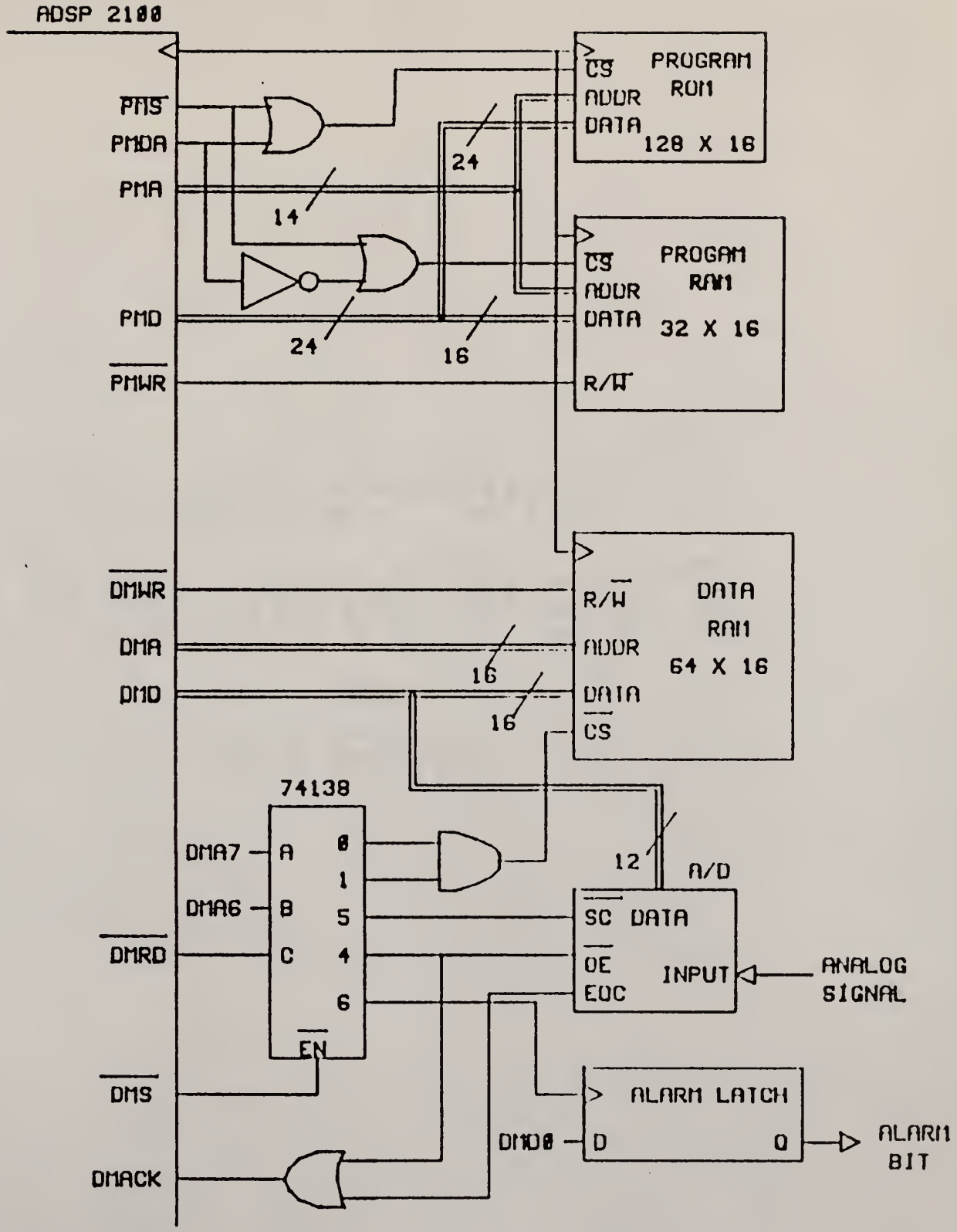

Figure A-8 Block Diagram for ADSP2100 Implementation

A.8.2 Usage of Memory and Registers

Index Registers

I0: points to top of data array ;
M0 = 1, L0 = 16

I1: points to top of error array ;
M1 = 1, L1 = 16

I2: points constant U ; M2 = 0, L2 = 0

I4: points to coeffs array in program memory ; M4 = 1 , L4 = 16

I5: points to constant V in program memory; M5 = 0, L5 = 0

Program Data Memory

Address	**Contents**
0 - 15	coeffs array
16	V

Data Memory

Address	**Contents**
0	FM
1 - 16	data array
17	EM
17 - 32	error array
128	A/D
192	alarm latch

A.8.3 Initial Conditions

The index registers have been setup as was outlined in the software map. The first data conversion has been started and all coefficients and constants have been

downloaded. Also, L2 and L3 have been set to 16.

A.8.4 Assembly Listing for Adaptive Algorithm

```
{ Input the latest data sample and store it in memory }
LOOP      AX0 = DM( A/D )
          DM( FM ) = AX0
          DM( A/D ) = AX0     {  dummy write to start next
                              conversion of the incoming data }
{ Now perform the FIR filter using the past sixteen data
samples and the present coefficients }
          MX0 = DM( I0 ),     { load first elements of }
               MY0 = PM( I4 ) { both arrays }
          MR = 0    { clear accumulator }
          CNTR = 15 { set loop counter }
          DO FIR UNTIL NOT CE
FIR            MR = MR + MX0*MY0,  { FIR algorithm }
                    MX0 = DM( I0 ), { MX0 gets next data }
                    MY0 = PM( I4 )  { MY0 gets next coeff }
{ Now compute the error in this estimate of the next data
sample . EM = FM - GM }
          AY0 = DM( FM ) { load FM into accumulator }
          AR = AY0 - R1  { EM = FM - GM }
          DM( EM ) = AR  { store it }
{ Now update the weighting coefficients }
          CNTR = 15
          DO ENDUP UNTIL NOT CE    { do loop 16 times }
          M4 = 0    { disable incr of coeff pointer }
```

```
          MX0 = DM( I2 ),       { load U }
               MY0 = PM( I4 )       { load present coeff }
          MR = 0     { clear acc }
          MR = MR + MX0*MY0,   { mult U and coeff }
               MX0 = DM( I0 ), { load data element }
               MY0 = PM( I5 )  { load V }
          MF = MX0 * MY0, { multiply and feedback }
               MX0 = DM ( EM )
          MR = MR + MX0*MF     { and mult by EM and add to
                    previous product to obtain new coeff }
          M4 = 1     { re-enable incr of coeff pointer }
ENDUP     PM( I4 ) = R1  { store new coeff and point to the
                           next for next loop transgression }
{ Now compute the average error for the past sixteen
samples }
          AX0 = DM( I1 )       { load first error sample }
          AF = PASS AX0,       { make it first partial sum }
               AX0 = DM( I1 ) { and load next error }
          CNTR = 14
          DO ENDSUM UNTIL NOT CE    { do add loop 15 times }
ENDSUM    AF = AX0 + AF
          AR = PASS AF         { AR gets sum }
          SR = ASHIFT AR BY -4 { and divide by sixteen }
{ Now compute the square of this average and compare that
against a known threshold }
          MX0 = SR
          MY0 = SR
```

```
          MR = MX0 * MY0


          AY0 = DM( THETA )
          AR = R0 - AY0  { AR = QM2 - THETA )
          IF LE JUMP NOALM  { reset alarm if QM2 < theta }
          DM( ALARM ) = 1     { else set alarm }
          JUMP ROLL      { and proceed to roll routine }
NOALM     DM( ALARM ) = 0     { clear alarm }
{ Now push all the elements in the data and error arrays
down one position ( a unit delay )}
ROLL      I2 = ENDDAT    { set an index reg to count
                           backwards from the bottom of
                           the data array }
          M2 = -1
          I3 = ERREND    { same for error array }
          M3 = -1
          I0 = ENDDAT + 1  { and other index regs to point
                     to the position directly above those }
          M0 = -1
          I1 = ERREND + 1
          M1 = -1
          CNTR = 15
          DO ENDRL  UNTIL NOT CE  { do loop 16 times }
          AX0 = DM( I0 )   { push each element down then }
          DM( I2 ) = AX0   { do the next until the top }
          AX0 = DM( I1 )   { of each array has been }
ENDRL     DM( I3 ) = AX0   { replaced. }
```

```
          I2 = V            { reset index regs }
          M2 = 0
          I0 = DATA
          M0 = 1
          M1 = 1
{ Now return for next data sample }
          JUMP LOOP
```

A.8.5 Evaluation

This processor required 69, 24-bit program words to realize the main loop of the algorithm. Its execution time (with a clock frequency of 32.8 MHz) is 33.1 us. This corresponds to a sample rate of 30,200 samples per second.

The lack of dedicated I/O and on-chip memory detract from an otherwise efficient and easy to master DSP. The " high level " assembly language is very easy to learn and to follow. The ability to simultaneously access program and data memory is very efficient, but necessitates having separate program and data buses adding to external complexity.

The indexed addressing is particularly built for vector manipulation. The ability to set, by default, the increment, decrement, and circular buffer length make it very convenient for accessing these type of data. It makes it awkward if one wishes to make a particular index register point to an array of different dimension and

different modification. Changing back and forth is somewhat clumsy but not extremely detracting.

The I/O and external circuitry as a whole is somewhat disproportionate to the task at hand. It is not unreasonable, but it is disheartening to imagine that a little on-chip memory would greatly diminish the magnitude of the external circuitry. The extra memory will have to be very fast (50 ns access time) to operate this system at full speed. This will add to the expense as well as the complexity, size and power consumption of this system.

A.9 LM32900

A.9.1 Hardware Implementation

This DSP has no on-chip memory thus all program and data memory will have to be added externally (see figure A-9). The LM32900 has simultaneous fetch of operands in two separate memory spaces. It has the capability to use immediate data in the program which eliminates the need for any data ROM. All that is necessary is to connect a program ROM and two banks of RAM. The DSP has sufficient control signals to select the memory with no glue logic.

The I/O is more complicated. Because only one I/O port is of any use (it has an input and an output port but only one may be used at any given time) an additional control circuit must be implemented to control the I/O function to be performed. The DSP does have programmable output pins and one of these will be used for the alarm bit.

As for the A/D, the control circuit will continually start the next conversion, wait for the end of conversion, then wait until the FIFO is not full to read the result and send a data strobe signal to the FIFO. This makes it almost independent of the routine which will serve to increase the speed of the routine at the cost of additional hardware.

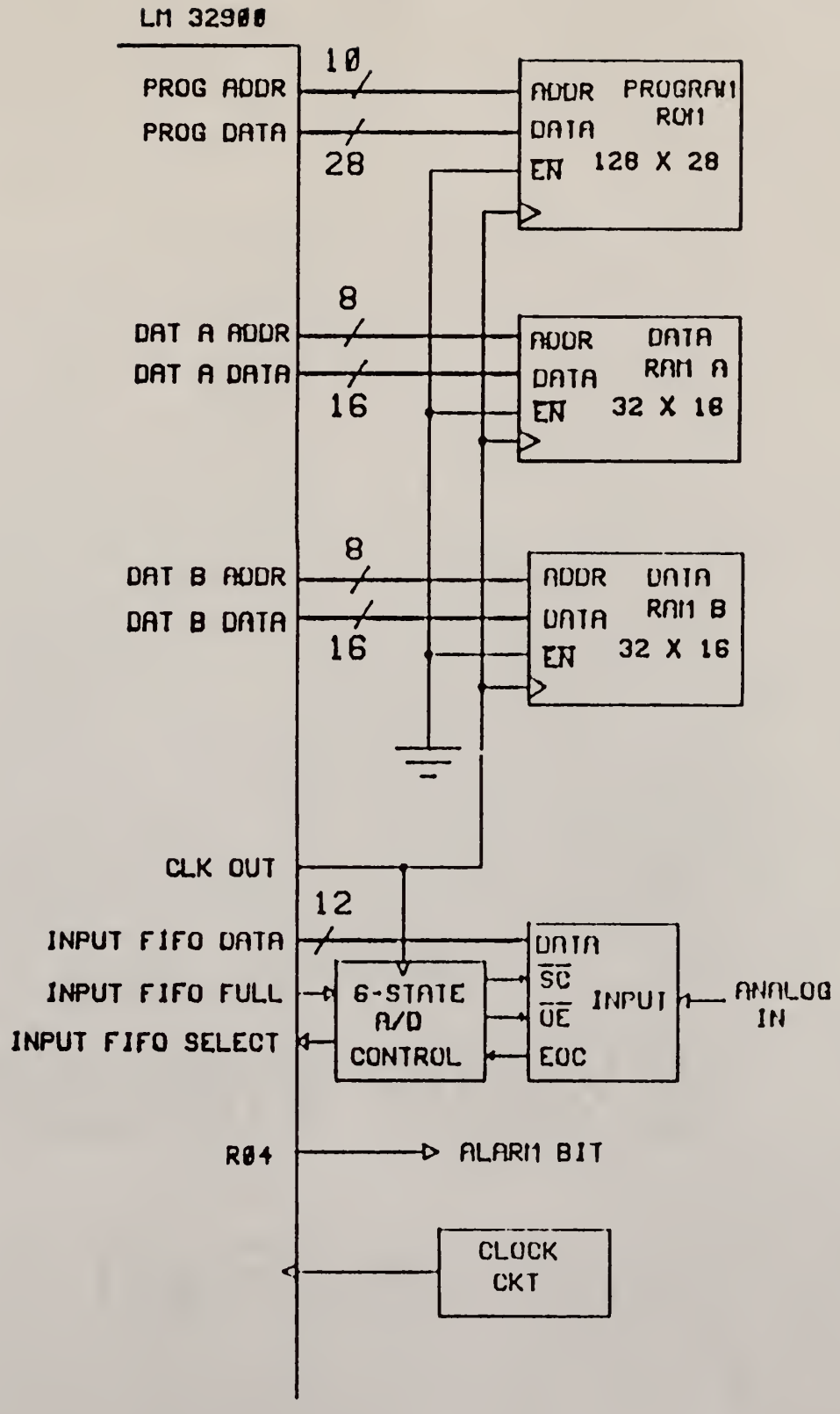

Figure A-9 Block Diagram for LM32900 Implementation

A.9.2 Usage of Memory and Registers

Buffer Registers

WA = WB = 4 (circular indexing of 16 elements for each reg)

These buffer registers make it possible for the index registers to increment or decrement 16 times and then roll over to the starting address. This is convenient for vector operations such as the FIR filter.

Index Registers

R1 : points to coeffs array in MEMA (16 -31)

R2 : points to data array in MEMB (48 - 63)

R3 : points to FM in MEMB (47)

R4 : points to error array in MEMA (0 -15)

R5 : points to U in MEMB (46)

R6 : points to V in MEMB (45)

R7 : holds shift count (= left 16 bits)

R0 : used as loop counter register (lower 4 bits) , bit 4 holds the state of the alarm bit. Also used to point to temporary storage for variables thus its contents should point to a real memory location.

All index registers for MEMA (except R0) should not point between $00 and $1F. Additionally, since copies are made of data and error arrays from one memory bank to another, these index registers (R2 and R4) should have

rights to the same addresses in both memory banks.

A.9.3 Initial Conditions

No conditions must be met other than those common to all of the processors.

A.9.4 Assembly Listing for Adaptive Algorithm

```
{ Input Data from FIFO }
MAIN      IN   R3,FIFO   { input from MEMA and transfer }
          MOVEAB    R3,R3 { to MEMB }
{ Now compute the output of the FIR filter }
          CLR       { clear accumulator }
          LD RC,#$10     { perform 16 mult & accum's }
          MULA R1+,R2+,1   { FIR filter }
{ The output of the FIR filter is GM. Compare this with the
input sample just obtained ( FM ) to find the error EM }
          SUB  R3,R7     { subtract FM ( shifted left 15
                 times ) from GM. R7 contains shift on FM }
          NEG       { negate result for proper sign }
          ST   AH,R4     { store at top of error array }
{ Now update the weighting coefficients }
{ To do this, a loop must be traversed 16 times. R0
contains the value 15 and will be used as a loop counter.
when its value is zero, then the looping will terminate. }
UPDATE    CLR
          MULA R1,R5,1   { mult old coeff and U with shift
                           of one to remove extra sign bit }
          MUL  R4,R6,1    { mult V and EM }
```

```
          ST PH,R0          { store that temporarily }
          MULA R0,R2+,1     { mult EM*V by F(M-K) and add to
                                B(M-1,K)*U }
          ST   AH,R1+       { store as new coeff }
          LD   AH,R0-       { dummy load to decr loop cntr }
          BNZA UPDATE

{ Now perform the moving average filter on the error
samples ( sum the last sixteen errors and divide by 16 }
          CLR
          LD   RC,#$10
          ADD  R4+          { sum errors }
          SHIFTA -4         { divide by 16 }
{ Now square this number and compare against threshold }
          ST   AL,R0        { store it in both MEMA and }
          MOVEAB R0,R0          { MEMB temporarily }
          MUL  R0,R0,1      { square it }
          CLR
          MOVE PH,AL        { and move it to accum }
{ is this larger than the threshold ? If so, set alarm.
Else reset alarm }
          SUB  THETA
          BGE  ALM
          LD   R0,#00       { reset alarm }
          BR   ROLL         { proceed to buffer update }
ALM       LD   R0,#10       {  set alarm }
{ Now push the data and error elements down one position in
```

```
their respective arrays }
ROLL      LD   RC,#0F     { use top 15 samples }
          MOVEBA R2+,R2+  { make a copy of data array in
                          MEMA and copy it to the new }
          LD   RC,#0F     { shifted position in MEMB }
          MOVEAB R2-,R2-
          MOVEBA R3,R2    { put FM on top of data array }
          MOVEAB R2,R2
          LD   RC,#0F     { move top 15 error samples }
          MOVEAB R4+,R4+
          LD   RC,#0F
          MOVEBA R4-,R4-
{ Now R2 should point to top of data array and R4 to the
top of the error array ( EM ). Return for the next data
sample }
          BR   MAIN
```

A.9.5 Evaluation

This processor required 40, 28-bit program words to realize the main loop of the algorithm. Its execution time (with a clock frequency of 20 MHz) is 25.1 us. This corresponds to a sample rate of 39,800 samples per second.

This DSP has a very time efficient architecture. The availability of both extended and multiply indexed addressing is a very convenient method. The circular buffers are a very convenient way to address vectors of whose length is a power of two (which is often the case in

digital signal processing algorithms). The coding of the algorithm was very straightforward.

The I/O would probably work more efficiently memory mapped into one of the dual banks. It did not have sufficient I/O capability for this circumstance. It requires the help of an external controller. This is not an unreasonable scheme to implement because many of its applications will probably involve accepting data from another system. As a stand alone system it lacks many virtues.

The lack of internal memory, unfortunately, necessitates a substantial amount of very expensive fast memory (approximately 45 ns at full speed with no wait states). This will add to the cost as well as the power consumption and physical size of the system. The need for a 28-bit wide instruction word may also cause some problems physically for some applications. Indeed, the shear number of connections for address and data is astounding (108 for all of the memory spaces combined).

A COMPARATIVE STUDY OF AVAILABLE DIGITAL SIGNAL PROCESSOR CHIPS

by

Carl Thomas Hardin
B.S., Electrical Engineering, University of Kentucky, 1986

AN ABSTRACT OF A MASTER'S THESIS

Submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

Abstract

This paper describes and compares a class of microprocessors known as Digital Signal Processors. The motivation for such a project is to allow designers of microprocessor based systems to be able to pick a processor from the set of currently available processors to perform a specific task.

The features of the Digital Signal Processors are tabulated and then a subset is chosen to implement Widrow's Adaptive Linear Predictor Algorithm. The implementation includes the development of a stand alone system and the assembly language code for the adaptive algorithm. Their performances in executing this algorithm on the basis of speed, power consumption and system complexity are evaluated and compared. All information is as per manufacturer's data and no actual testing is performed or discussed. Finally, some Floating-Point Processors are listed and discussed briefly.