AN EVALUATION OF VARIOUS MICROPROCESSOR IMPLEMENTATIONS OF AN ADAPTIVE DIGITAL PREDICTOR FOR INTRUSION DETECTION

by

DONOVAN J. NICKEL

B. S., Kansas State University, 1978

A MASTER'S REPORT

submitted in partial fulfillment of the

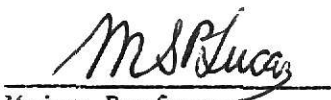
requirements for the degree

MASTER OF SCIENCE

Department of Electrical Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
1979

Approved by:



Major Professor

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

In communication theory, control systems theory, and other areas of electrical engineering which deal with signal detection, a common problem which is encountered deals with some type of a received signal which consists of the real or desired signal plus noise. In most instances it is useful to attempt to filter out as much of the noise component as possible and thus form a reasonable approximation to the desired signal. Often, the characteristics of the desired signal are known and conventional linear filters can be designed accordingly to produce a satisfactory solution to the stated problem. However, in the case of the problem of intrusion detection, the desired signal characteristics are not so easily determined and can vary according to many different circumstances. Therefore, a more inventive approach is needed to solve this signal detection problem.

Recent research in the area of digital signal processing by Ahmed [1] has led to a signal processing algorithm which is designed to deal with the problem stated above. The algorithm involves an Adaptive Digital Predictor and other more conventional digital filters. Its purpose is to effectively detect noise produced by an intruder in the presence of random ambient noise.

The emphasis of the work done by the author was in the development of working microprocessor implementations of the algorithm and subsequent

evaluations of the various microprocessors with respect to this task. Three microprocessors were considered. The first was the Zilog Z80. A minimal hardware system was designed and built, and complete software routines were written and tested to form a Z80 algorithm implementation. The second microprocessor considered was Intel's 8048 (8748) and the third was the RCA ATMAC. Since development systems were not available, the work on these two devices consisted of paper studies to evaluate the potential use of these microprocessors for this application. The studies included the writing of untested software routines required by the algorithm.

Chapter II of this report presents a brief review of the operation of the algorithm. After that, the various implementations are discussed beginning with the Z80 hardware and software developments in Chapters III and IV. Chapters V and VI present the implementation considerations relating to the 8748 and the ATMAC, respectively. Finally, some comparisons between the different implementations are made, and some conclusions are formed in Chapters VII and VIII.

CHAPTER II

THE WIDROW ALGORITHM

For purposes of intrusion detection, a typical input data signal is considered to have two components. They are the following:

1) random ambient noise

2) intruder-produced noise.

The objective in designing a signal processing algorithm is to be able to reliably detect the intruder-produced signal, if any, in the presence of random ambient noise.

Figure 2.1 shows a block diagram of this intrusion detection technique. The algorithm consists of two main parts. The first section is an Adaptive Digital Predictor (ADP) which is implemented using Widrow's Least-Mean-Square (LMS) algorithm [2]. The primary purpose of the ADP is to statistically decorrelate the noise component of the data signal. This causes the error sequence, E_M, to be less correlated and to have a smaller variance than the input sequence, F_M. Thus, the ADP reduces the detection problem to one of determining whether or not an intruder-produced signal is present in noise which is uncorrelated.

The second section of the algorithm is a conventional Moving Average Filter (MAF). Since the output of the ADP will tend to be band-limited white noise, and since the resulting uncorrelated noise samples will tend to average out in a MAF, the output of the MAF will consist primarily of

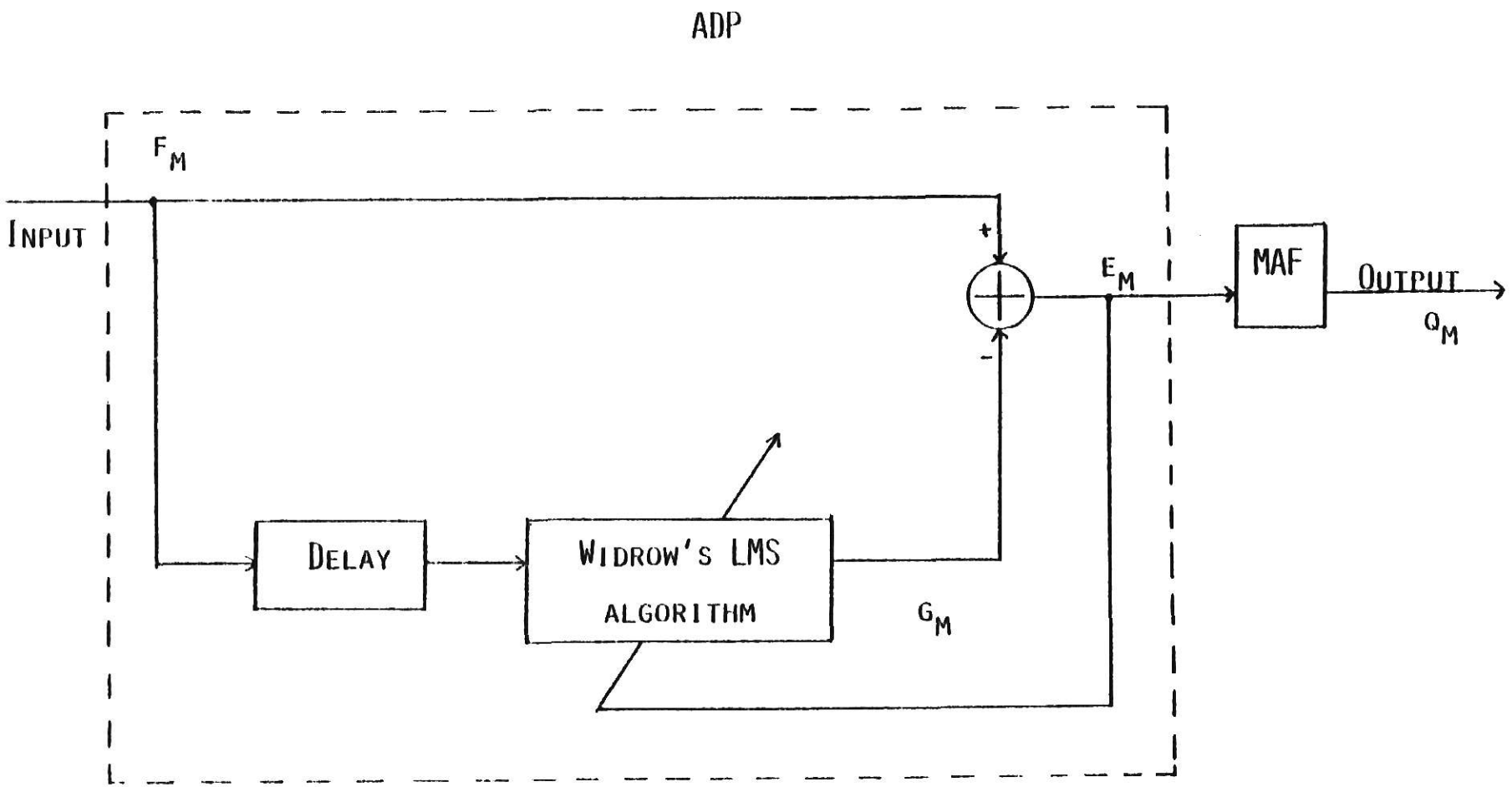


FIGURE 2.1. WIDROW ALGORITHM BLOCK DIAGRAM

intruder-produced noise, if any. Thus an output which reaches a certain predetermined magnitude threshold indicates the presence of an intruder.

It is important to realize that certain trade-offs exist between the accuracy of the algorithm with respect to correct detections versus false alarms, and the complexity of the implementation with respect to processor speed, system memory size, and fixed-length processor word size. For example, the random noise cannot be completely decorrelated by the ADP without using an unlimited number of samples in the estimate of input F_M. Yet the system constraints on memory size and operation speed prohibit such computations. However, if the noise component of the data signal is not sufficiently decorrelated by the ADP, the MAF will be less effective, and the false alarm rate will increase.

Another problem encountered in producing a working algorithm implementation was the difficulty in scaling the inputs to a filter with a varying transfer function. Since any overflow causes the system to break down and since the system must remain stable over long, continuous periods of time, the input sample scaling must be adequate to prevent overflow in the worst case. Due to these problems and others, it is obvious that reasonable approximations and trade-offs must be made to produce a satisfactory algorithm implementation.

There are many possible algorithm structures which could be effective in dealing with this intrusion detection problem. Therefore, as the work on the project progressed, some additions and changes were made to the algorithm. The Z80 implementation uses the algorithm as it is presented

in this chapter. The 8748 and the ATMAC implementations, however, used algorithms which are slightly different than the one described here. These differences are pointed out during the discussions of the implementations to which they apply.

CHAPTER III

THE MINIMAL Z80 MICROPROCESSOR

The objective in designing this minimal Z80 microprocessor was to develop a small, inexpensive system capable of implementing one channel of the algorithm described in Chapter II at a rate of at least 128 Hz. The Zilog Z80, which is an 8-bit, NMOS microprocessor, was chosen for this application for several reasons. Since most of the arithmetic required by the software program needed to be performed on 16-bit operands, the powerful instruction set of the Z80, which includes many 16-bit operations, was a very attractive feature. Another important constraint involved the program speed. The Z80, at 4 MHz, was comfortably able to meet the algorithm speed requirements. Although the Z80 used in this design was constructed with NMOS technology, the total system's power consumption was typically less than 250 milliamperes at 5 volts due to a mixture of CMOS and NMOS support components. This system was designed for a specific application, but imaginative users will find the system useful for a variety of tasks such as discrete time control systems and other digital signal processing operations.

Due to the possibility that the user, given a completed system, will desire only the knowledge necessary to make the system work without digging through the design and construction details, the following section will provide operation information which will hopefully be sufficient to allow the user to implement his application. Following the operation information,

the system configuration will be discussed in more detail and finally, the system construction will be presented including the design and construction of a printed circuit board. The assumption is made throughout this report that the user is familiar with the Z80 microprocessor and instruction set [3,4].

Minimal Z80 Operation

There are three main blocks of support devices used in this minimal Z80 system. They are program memory, data memory, and input/output devices. An understanding of the system's use of these support devices along with an understanding of the proper power supply and other edge board connections is all that is needed for proper operation of the system.

The program memory consists of a selected Intel 2716, 2k X 8, EPROM which is capable of 250 nanosecond access time operation. Thus the maximum total program length is 2048 bytes of code. The addresses which select the program memory begin at 0000H and end at 07FFH. Since a reset occurs when the system is turned on as well as when the reset switch is closed, the Z80 will begin executing the program starting at location 0000H when either of the two events takes place. It is important to remember that all program executions must begin at 0000H and that program execution can only be started by turning the power to the system from off to on or by pressing the reset switch. Thus when the 2716 EPROM is programmed, either the desired program or a branch to the desired program must begin in location 0000H.

The data memory consists of 128 bytes of static RAM. It can be read from or written to at address locations 4000H to 407FH.

One octal latch is available for use as an output port. Any output instruction such as OUT (01),A will enable the latch. The output data will be available until it is changed by another output statement. Since 74Cxx series CMOS is used, the output port is TTL compatible and is capable of driving one standard TTL load. The output data is available at the O/P socket on the board where socket pin 1 = least significant bit and socket pin 8 = most significant bit.

An eight bit A/D converter serves as the only means of input to the Z80. An input voltage in the range $\pm$10 volts is converted to an eight bit binary number for use by the Z80. Since the A/D converter is not free running, a "Start Conversion" pulse must precede a read statement by at least two milliseconds. A typical sequence of operations for obtaining an input byte from the A/D is listed below. The user is responsible to maintain the proper timing with software.

1. Send Start Conversion Pulse:

   This is achieved by an LD A, (8000) command or some other suitable command which puts 8000H on the address bus.

2. Pause:

   A pause of at least 2 msec. is necessary. Other routines may run during this time.

3. Input From A/D:

   The data word is input to the Z80 with the command LD A, (C000).

Data from the A/D is received in offset binary form. Two's complement numbers are obtained by inverting the most significant bit of the data word.

The power supply connections and other edge connections are listed in Table 3.1 for the standard 22-pin edge connector. A -5 volt source is needed only if the A/D converter is used. The system will typically require less than 250 milliamperes at 5 volts d.c. The connections listed are those for the PC board. The connections for the wire-wrapped board are given in the section on System Construction.

Table 3.1. PC Board Edge Connections

Pin #	Function
2	-5 volts
14	V_{IN} (input to the A/D converter = $\pm$10 volts)
20	ϕ (clock out)
22	+5 volts
D	GND.
T	$\overline{\text{WAIT}}$ signal in

Some software application examples are given on the following two pages.

Sample Program 1

This program will alternately output AAH and 55H to the output port. The bit patterns should be 1010 1010 and 0101 0101. This program tests the Z80 CPU, the 2716 EPROM, and the output port. The pattern should alternate at a rate of about 1/2 second.

location	instruction	code	comments
0000H	LD A, n	3E	A = AA
1		AA	
2	OUT (n), A	D3	output to port
3		00	
4	LD HL, nn	21	set up counter
5		FF	
6		FF	
7	DEC HL	2B	
8	LD A,n	3E	
9		00	delay
A	ADD A, H	84	
B	JR NZ,e	20	
C		FA	
D	LD A,n	3E	A = 55
E		55	
F	OUT (n), A	D3	output to port
10		00	
1	LD HL, nn	21	
2		FF	
3		FF	
4	DEC HL	2B	
5	LD A, n	3E	
6		00	delay
7	ADD A, H	84	
8	JR NZ, e	20	
9		FA	
A	JP nn	C3	go again
B		00	
C		00	

Sample Program 2

This program will input a data sample from the A/D converter, store it in the data memory, read it from the data memory, send it out to the output port, and repeat the process. The data word is changed from offset binary form to two's complement form prior to its output. The output port information can be used as an input to a D/A converter to form a wrap-around sampled test signal to aid in adjusting the A/D offset. This program tests all the components of the system.

location	instruction	code	comments
0000H	LD A, (nn)	3A	input sample
1		00	from A/D
2		C0	
3	LD B, A	47	B = sample
4	LD A, (nn)	3A	send "start
5		00	conversion" pulse
6		80	
7	LD A, n	3E	A = mask
8		80	
9	XOR B	A8	convert sample
A	LD (nn), A	32	to 2's comp.
B		00	store sample in
C		40	data memory
D	XOR A	AF	clear A
E	LD A, (nn)	3A	
F		00	A = sample
10		40	
1	OUT (n), A	D3	output sample
2		00	
3	LD B, n	06	
4		FF	
5	NOP	00	
6	NOP	00	delay for A/D
7	NOP	00	conversion time
8	NOP	00	
9	NOP	00	
A	DJNZ e	10	
B		F9	
C	JP nn	C3	go again
D		00	
E		00	

System Design

The block diagram in Figure 3.1 and the circuit diagram in Figure 3.2 illustrate the structure of this system. The system, which is based on the 4 MHz version of the powerful eight bit Z80 microprocessor, involves a minimum number of packages for maximum system simplicity. The total system is relatively low in power consumption due to a mixture of CMOS and NMOS components. The discrete components are listed in Table 3.2.

The oscillator is crystal controlled and runs at 4 MHz. It uses a 74C04 CMOS hex inverter chip to form ϕ for the Z80 as well as a copy of the clock signal which is available at the edge connector. It is TTL compatible and can drive one standard TTL load.

The output port is a 74C374 CMOS octal latch. It is enabled by any output instruction, regardless of the address. It is also TTL compatible and can drive one standard TTL load.

A Teledyne 8703 CMOS, eight bit, A/D converter with 3-state outputs is used for data input. It is a current integrating A/D converter and in this configuration yields an offset binary output. An input voltage in the range of $\pm$10 volts is passed through a resistor to form a current input. This current is shifted from $\pm$5 microamperes to 0 to 10 microamperes by a simple op amp circuit prior to conversion. R_{OFF} is used to trim this offset to its correct position. To set R_{OFF}, ground the input and adjust R_{OFF} until 80H is output from the A/D. The value of R_{OFF} can be calculated as follows:

$$R_{OFF} = \frac{V_o}{5(10^{-6})}$$, where V_o = output voltage of the op amp

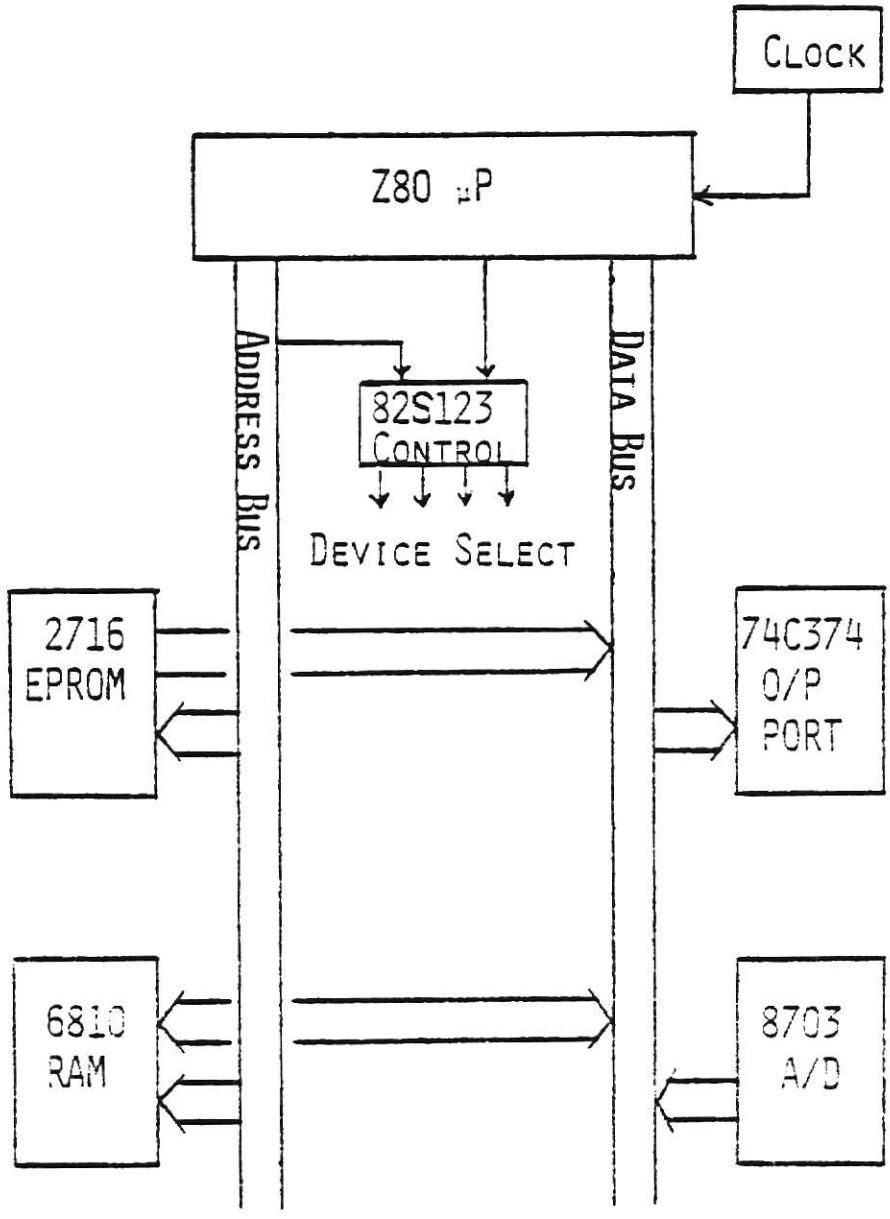


FIGURE 3.1. MINIMAL Z80 BLOCK DIAGRAM

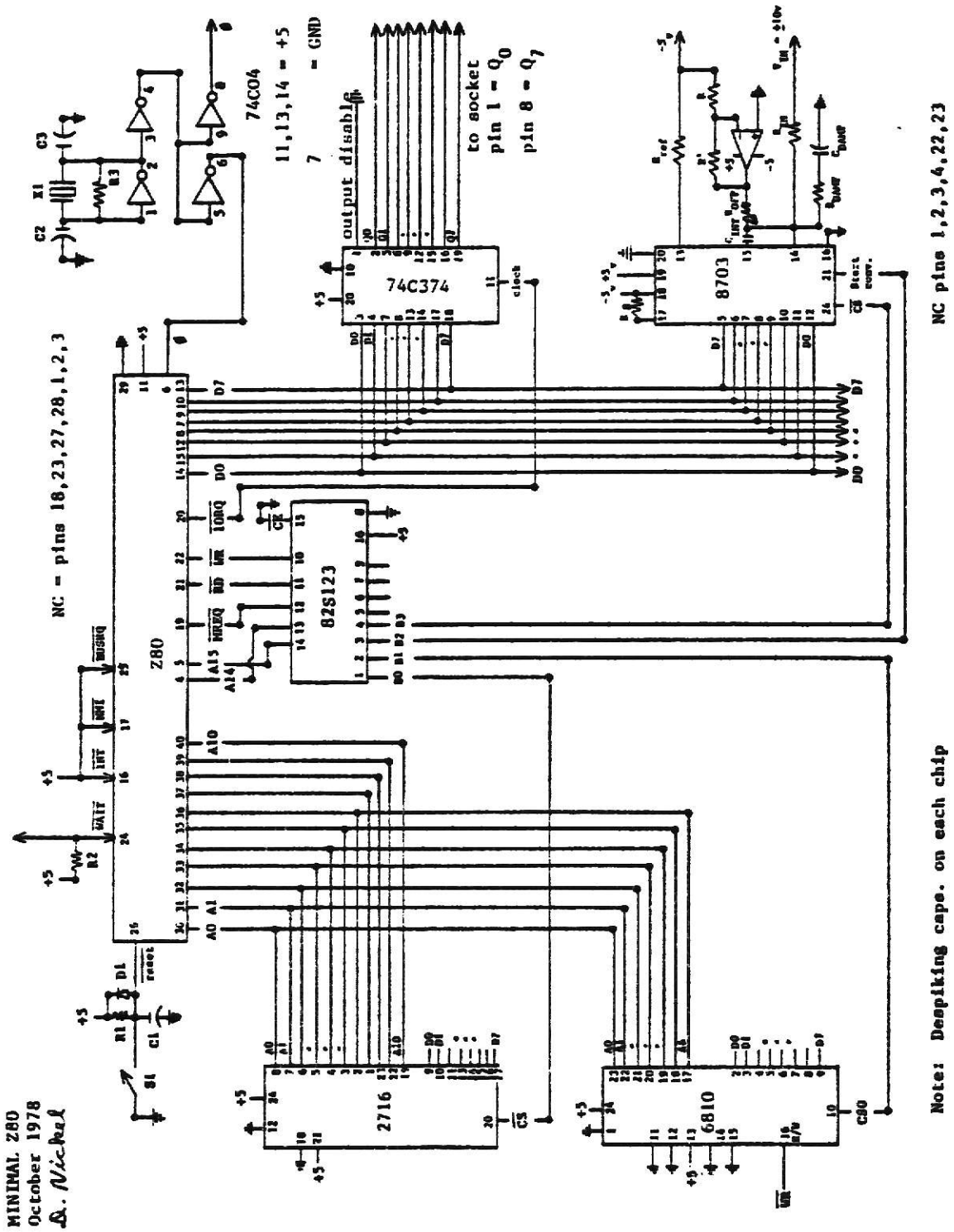


Figure 3.2. Minimal Z80 Circuit Schematic

Table 3.2. Minimal Z80 Discrete Components

Clock:

X1 = 4 MHz crystal

C2 = C3 = 33 pF

R3 = 10 MΩ

Z80:

S1 = momentary push button N.O. switch

R1 = 10 kΩ

R2 = 2 kΩ

C1 = 1.5 μF.

D1 = 914 diode

8703 A/D:

R_B = 100 kΩ

R_{ref} = 250 kΩ

R = R' = 4 kΩ

R_{OFF} = 1 MΩ trimmer potentiometer

R_{IN} = 2 MΩ

R_{DAMP} = 100 Ω

C_{DAMP} = 270 pF.

C_{INT} = 68 pF.

Power Supplies:

The system requires +5 volts, -5 volts, and GND.

Notes:

1. All resistors are 1/2 watt, ±5%, except R_{IN}, R, R', and R_{ref}, which are ±1%.
2. 0.1 μF capacitors were used to despike the power lines on all chips.

The operation of the A/D converter is completely controlled by the software program. The user is referred to Sample Program 2 for the correct operation procedure.

The system memory includes program memory and data memory. Program memory is realized in the form of an Intel 2716, 2k X 8, EPROM. The user is referred to an Intel Memory Data Book for the programming and erasing procedures. The 2716 must be programmed prior to insertion in its socket. Either the power on reset circuitry or the reset switch can be used to start the execution of the program stored at 0000H. Although 2716's are not specified to run at 250 nanoseconds, many of them will run that fast. However, it is possible that some will not work with this system, and therefore the specifications for this system call for "selected" 2716's.

Data memory consists of a Motorola 68B10P, 128 X 8, RAM. It can be used for data storage, buffer storage, and other scratch pad storage. Recall that this is a volatile device, and it can only remember data as long as power is supplied to it.

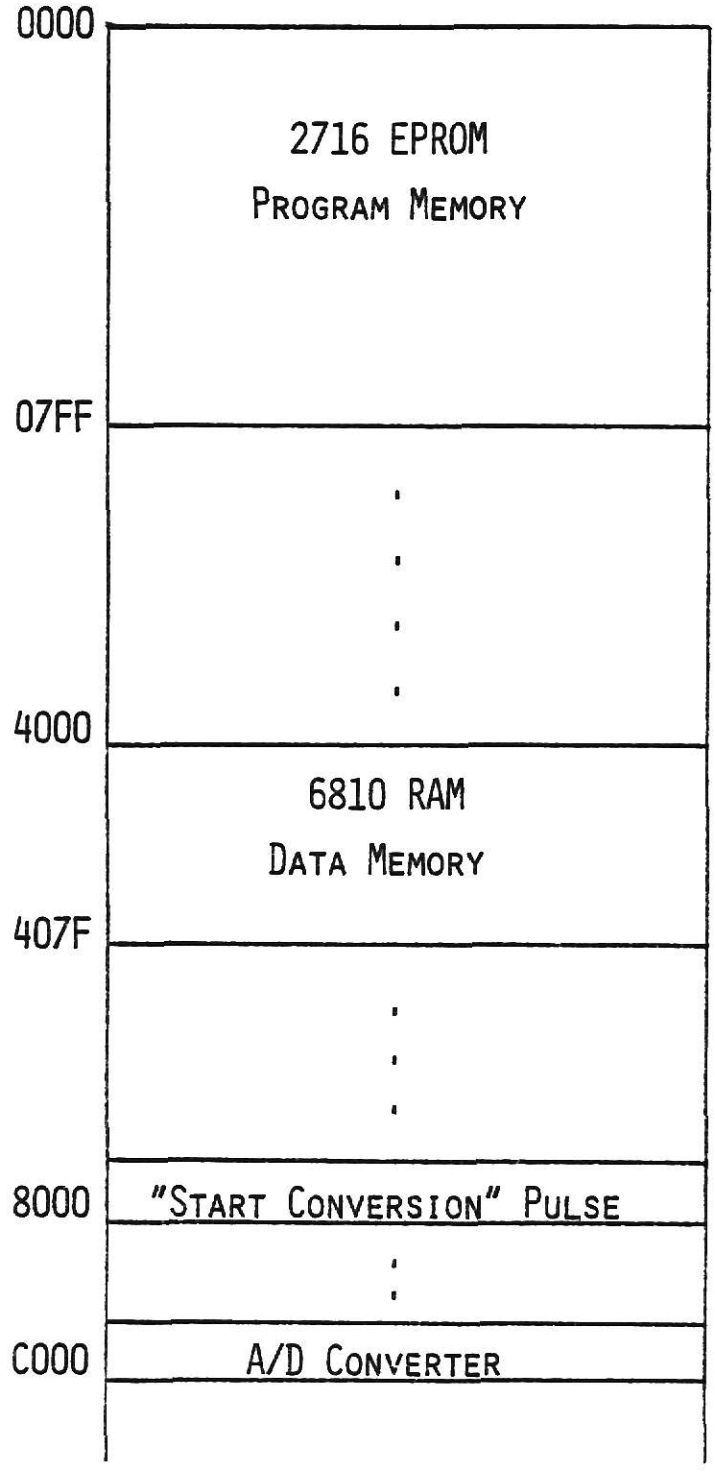
All of the device select control logic is performed by an Intel 82S123, 32 X 8, PROM. Inputs to the 82S123 which form its address lines are A15, A14, $\overline{\text{MREQ}}$, $\overline{\text{RD}}$, and $\overline{\text{WR}}$. Thus, a certain combination of two address lines and three control lines is used to select various support devices. The contents of the 82S123 are given in Table 3.3. A memory map of the system is given in Figure 3.3.

Table 3.3. Contents of the 82S123 PROM

The following bit pattern was burned into the PROM to achieve the correct device select logic.

Address Inputs					Output Data Word							
A_4	A_3	A_2	A_1	A_0	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	x	x	x	x	1	0	0	1
0	0	0	0	1					1	0	0	0
0	0	0	1	0					1	0	0	1
0	0	0	1	1					1	0	0	1
0	0	1	0	0					1	0	0	1
0	0	1	0	1					1	0	0	1
0	0	1	1	0					1	0	0	1
0	0	1	1	1					1	0	0	1
0	1	0	0	0					1	0	0	1
0	1	0	0	1					1	0	1	1
0	1	0	1	0					1	0	1	1
0	1	0	1	1					1	0	0	1
0	1	1	0	0					1	0	0	1
0	1	1	0	1					1	0	0	1
0	1	1	1	0					1	0	0	1
0	1	1	1	1					1	0	0	1
1	0	0	0	0					1	0	0	1
1	0	0	0	1					1	1	0	1
1	0	0	1	0					1	0	0	1
1	0	0	1	1					1	0	0	1
1	0	1	0	0					1	0	0	1
1	0	1	0	1					1	0	0	1
1	0	1	1	0					1	0	0	1
1	0	1	1	1					1	0	0	1
1	1	0	0	0					1	0	0	1
1	1	0	0	1					0	0	0	1
1	1	0	1	0					1	0	0	1
1	1	0	1	1					1	0	0	1
1	1	1	0	0					1	0	0	1
1	1	1	0	1					1	0	0	1
1	1	1	1	0					1	0	0	1
1	1	1	1	1					1	0	0	1

FIGURE 3.3. MINIMAL Z80 MEMORY MAP



System Construction

The prototype minimal Z80 system was originally put together on a 6-1/2" x 4-1/2" Vector board using wire-wrap construction. For this board, the edge connections are shown in Table 3.4.

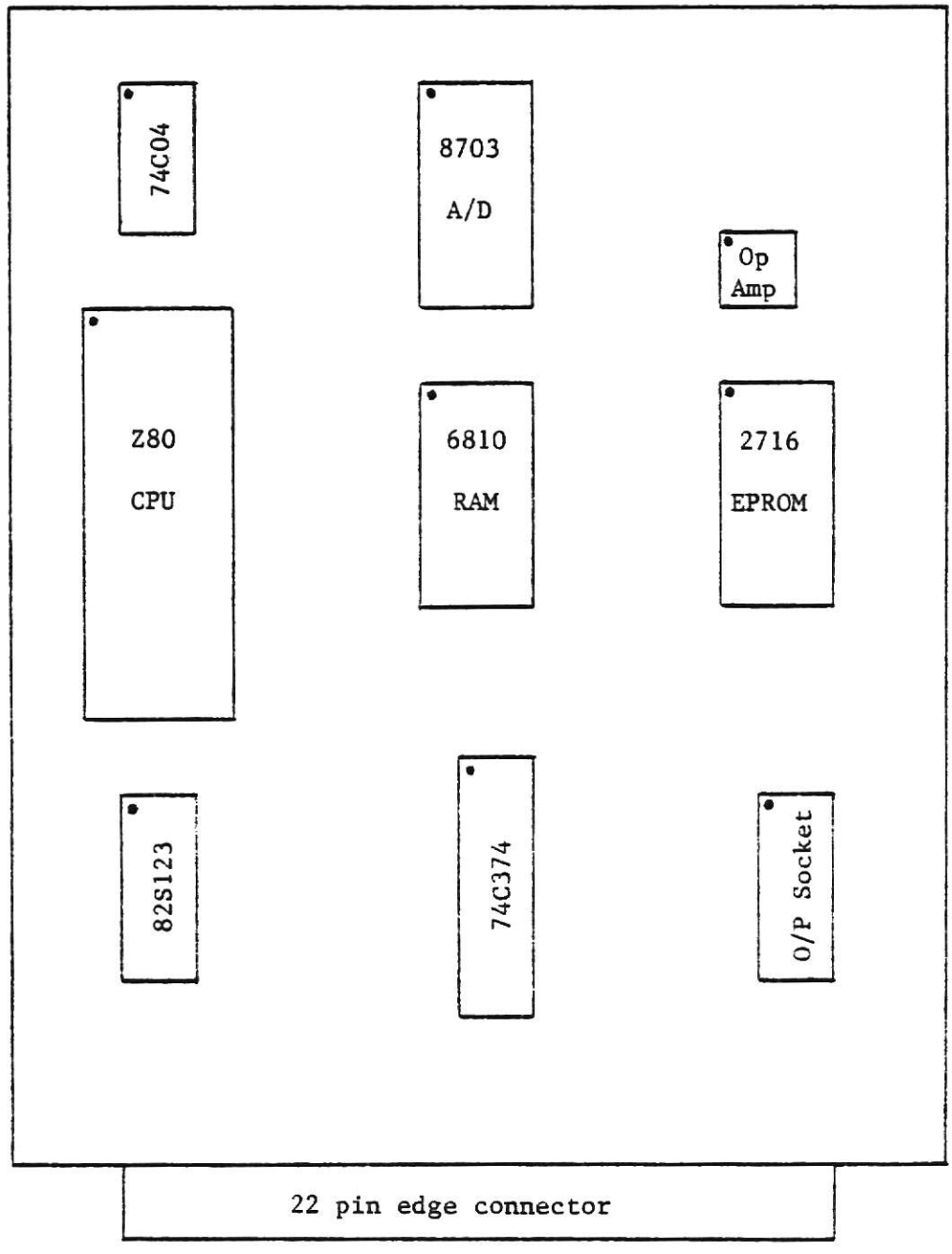
Table 3.4. Wire-Wrap Board Edge Connections

Pin #	Function
A	GND.
C	ϕ
E	$\overline{\text{WAIT}}$
V	V_{IN} = A/D Converter Input
X	-5 volts
Z	+5 volts

The board layout is shown in Figure 3.4. The output port information is available at pins 1 to 8 of the output socket. Pin 1 holds the least significant bit and pin 8 holds the most significant bit of the output byte.

After the prototype board had been tested, a printed-circuit board was designed for the minimal Z80 system. Although this board has the same dimensions as the Vector board, it has some important differences. In addition to a revised package layout, these differences include a changed edge connector scheme, a mounted reset switch, and a series of

Figure 3.4. Wire-Wrap Board Component Layout
(Vector 3677-2)



Top View Component Side

DIP pads for additional package placement on the board by the user. Room for two 24 pin packages or three 16 pin packages has been defined on the board and connections to the existing circuit can be made via wire-wrap techniques. A listing of the edge connections is given in Table 3.5.

Table 3.5. PC Board Edge Connections

Pin #	Function
2	-5 volts
14	V_{IN} = A/D Converter Input
20	ϕ
22	+5 volts
D	GND.
T	$\overline{WAIT}$

Since the capability to form plated-through holes was not available, wire-wrap sockets, placed about 1/4" above the component side of the board, were used to form the connections through the board by soldering the pins to both sides of the board. A step by step summary of the procedure followed in the production of the printed-circuit board and the construction of the minimal Z80 system is given below. For correct component placement and connections, the user is referred to the drawings and artwork which appear in Figures 3.5, 3.6, and 3.7 as well as the circuit schematic in Figure 3.2. Figure 3.8 shows a completed system.

Step 1:

From the circuit design, a trial and error process was used to obtain a reasonable package layout.

Step 2:

The printed circuit board artwork was then drawn at 2X.

Step 3:

Clear plastic was placed over the 2X artwork, and Bishop Graphics material, such as tape and DIP patterns, was placed over the plastic to trace all the connections and form all the component pads.

Step 4:

A negative reduction was made from the plastic sheet photographically to yield a 1X negative of the desired pattern.

Step 5:

A presensitized, double-sided, copper clad board, obtained from Kepro, was exposed through the negatives to a sun lamp for about 5 to 6 minutes.

Step 6:

The exposed board was developed in a Trichloroethylene bath for about 1-1/2 minutes.

Step 7:

The board was then baked at 180°F for 5 minutes to set the pattern.

Step 8:

The board was etched in an etching solution spray until the pattern was clear.

Step 9:

For a longer lasting finish, the board was tin plated.

Step 10:

The board was completed by cutting it to the proper size and drilling the holes.

Z80 Construction, Step 1:

The three connections through the board where no components are placed were made first using bare wire-wrap wire.

Step 2:

Wire-wrap sockets were placed in their proper position about 1/4" above the component side of the board and were soldered to both sides of the board.

Step 3:

The discrete components were placed in position and soldered to both board sides.

Step 4:

The reset switch was mounted to the board and was connected via short lengths of wire-wrap wire to the two sides of capacitor C1.

Step 5:

The chips were placed in the sockets with pin 1 of each chip placed as is indicated by the markings on the board.

Figure 3.5. Printed Circuit Board Component Layout
(Top View, Top Side)

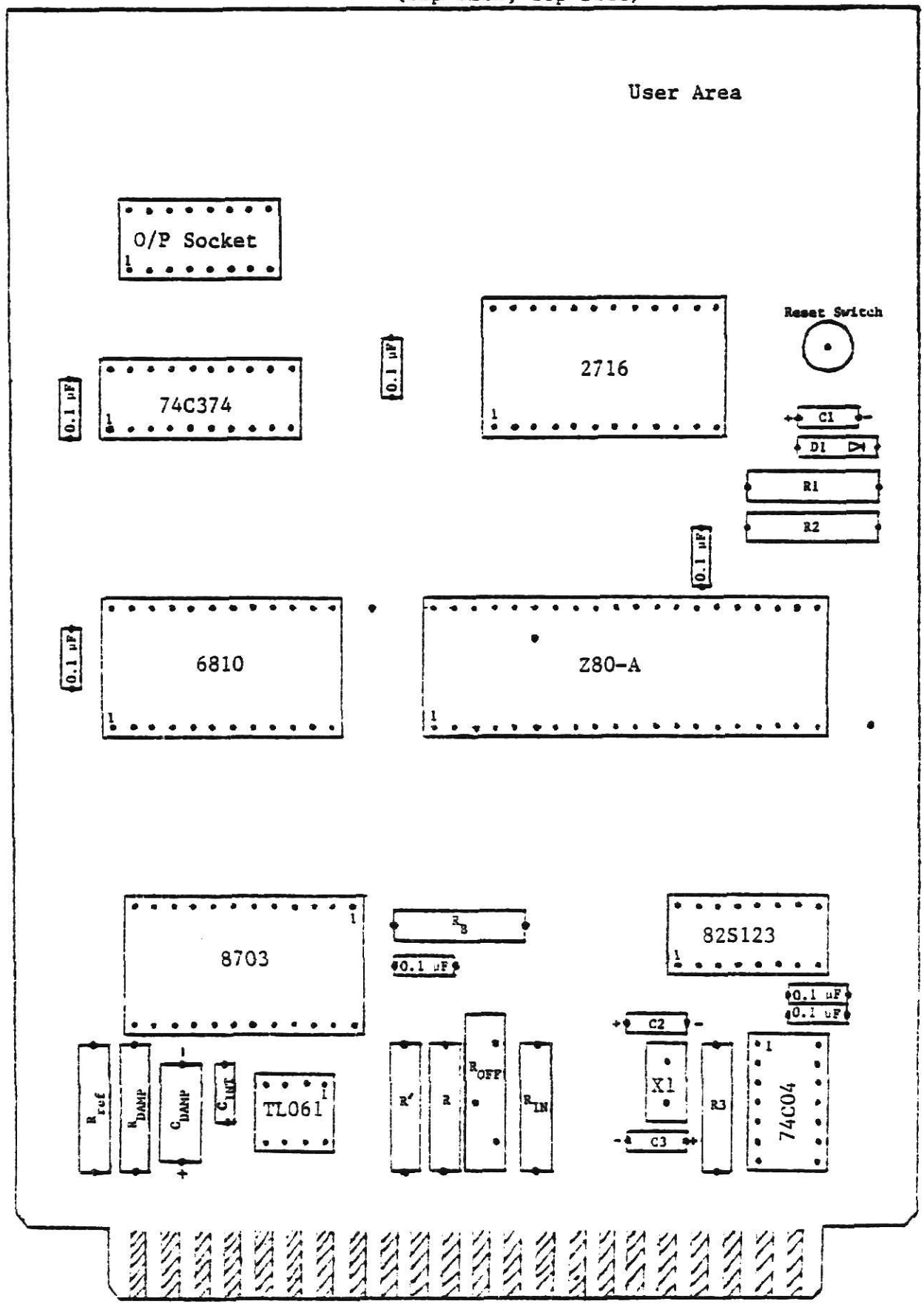


Figure 3.6. Printed Circuit Board Artwork (Top View, Top Side)

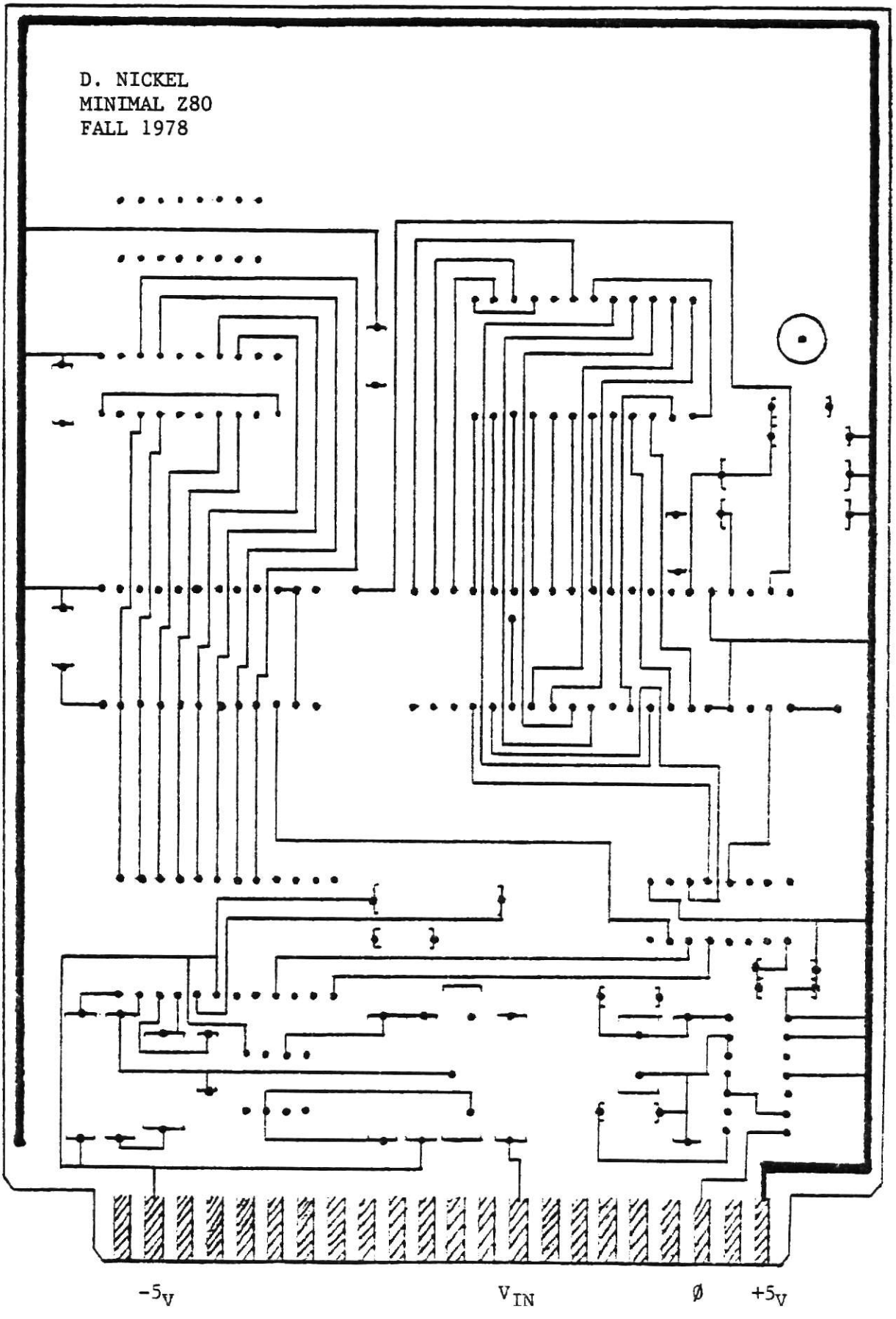
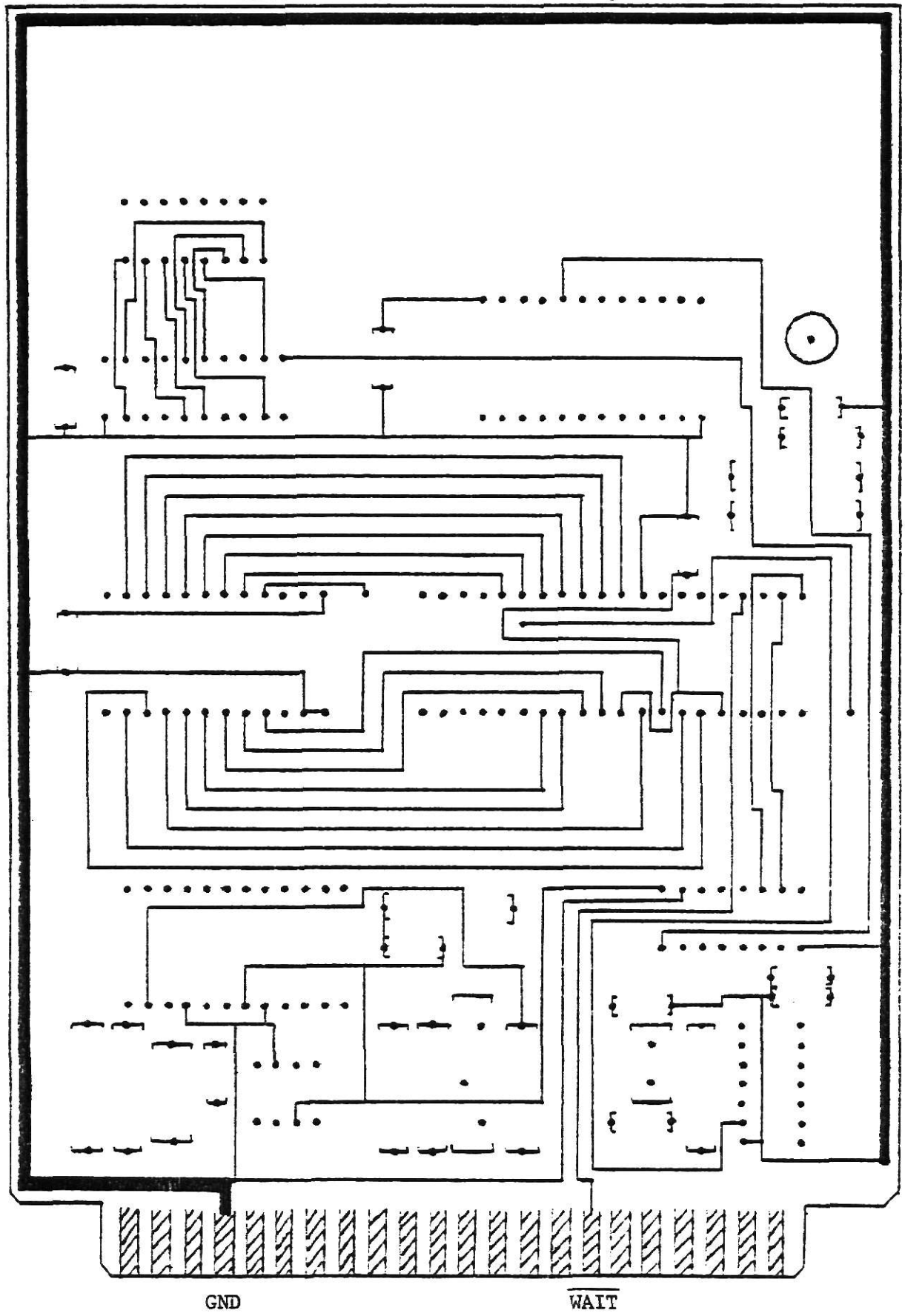


Figure 3.7. Printed Circuit Board Artwork (Top View, Bottom Side)



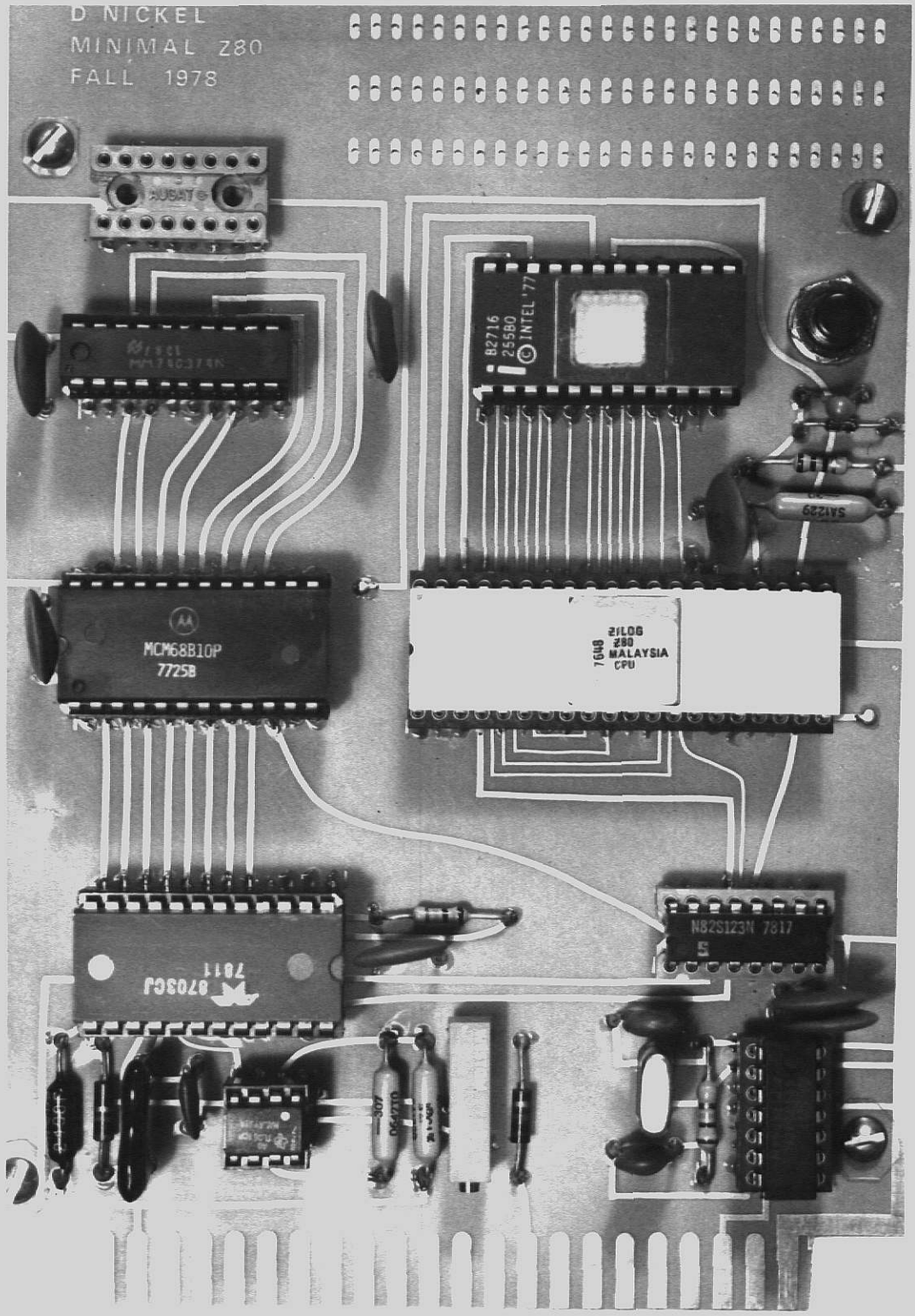


Figure 3.8. Completed Minimal Z80

CHAPTER IV

Z80 WIDROW ALGORITHM SOFTWARE

The algorithm which was implemented using the Z80 is the one described in Chapter II. At the time of the development of this program, the timing constraint required that the operation rate of the program needed to be at least 128 Hz.

The software implementation of the algorithm proved to be a rather demanding task for an 8-bit microprocessor. In order to maintain sufficient accuracy in the various filtering operations with a fixed-point implementation, 16-bit arguments and arithmetic operations were desired throughout the program. Included in the program were thirty-three signed multiplications and many other arithmetic operations. Therefore, the single biggest problem became that of developing a method of performing thirty-three 16-bit signed multiplies in less than seven milliseconds. Since no software method of doing this could be found, and since the use of an external hardware multiplier implied a significantly greater total system cost and power consumption, the use of signed 16-bit multiplies became impractical. Consequently, an 8-bit software signed multiply routine was used which operated on the most significant bytes of the two 16-bit arguments to form a reasonable approximation to the actual product.

As in any fixed-point digital filter implementation, the inputs and filter coefficients needed to be scaled to avoid the possibility of overflows. Thus the binary point was assumed to be to the left of the most significant bit of the 16-bit word and one arithmetic shift right was performed on the input samples to further limit the dynamic range of the arguments.

Figure 4.1 shows a software program flowchart which summarizes the necessary algorithm computations; the symbols are identified in Figure 2.1. Three sixteen member buffers, each containing 16-bit words and pointers to them must be maintained throughout the program.

The F_M buffer contains the sixteen most current input samples. The buffer is circulating in the sense that a given element, say F_{k-1}, becomes F_{k-2} in the succeeding program iteration. As a result, either the entire buffer must be repositioned in memory every time a new sample is obtained, or a pointer to a specific sample, such as the single most recent sample, must be maintained and updated each time a new sample is read.

Sixteen filter coefficients or weights are stored in the B_M buffer. The values of the weights are recalculated during each program iteration. This buffer is not circulating.

The third buffer contains the most recent sixteen values of the error, E_M, which are computed by the program. Like the F_M buffer, this is a circulating buffer.

The flowchart indicates one possible sequence of calculations for the algorithm. All the buffers are cleared during the initialization. This results in a slightly delayed output corresponding to a certain input, but this property of most digital filters is generally acceptable.

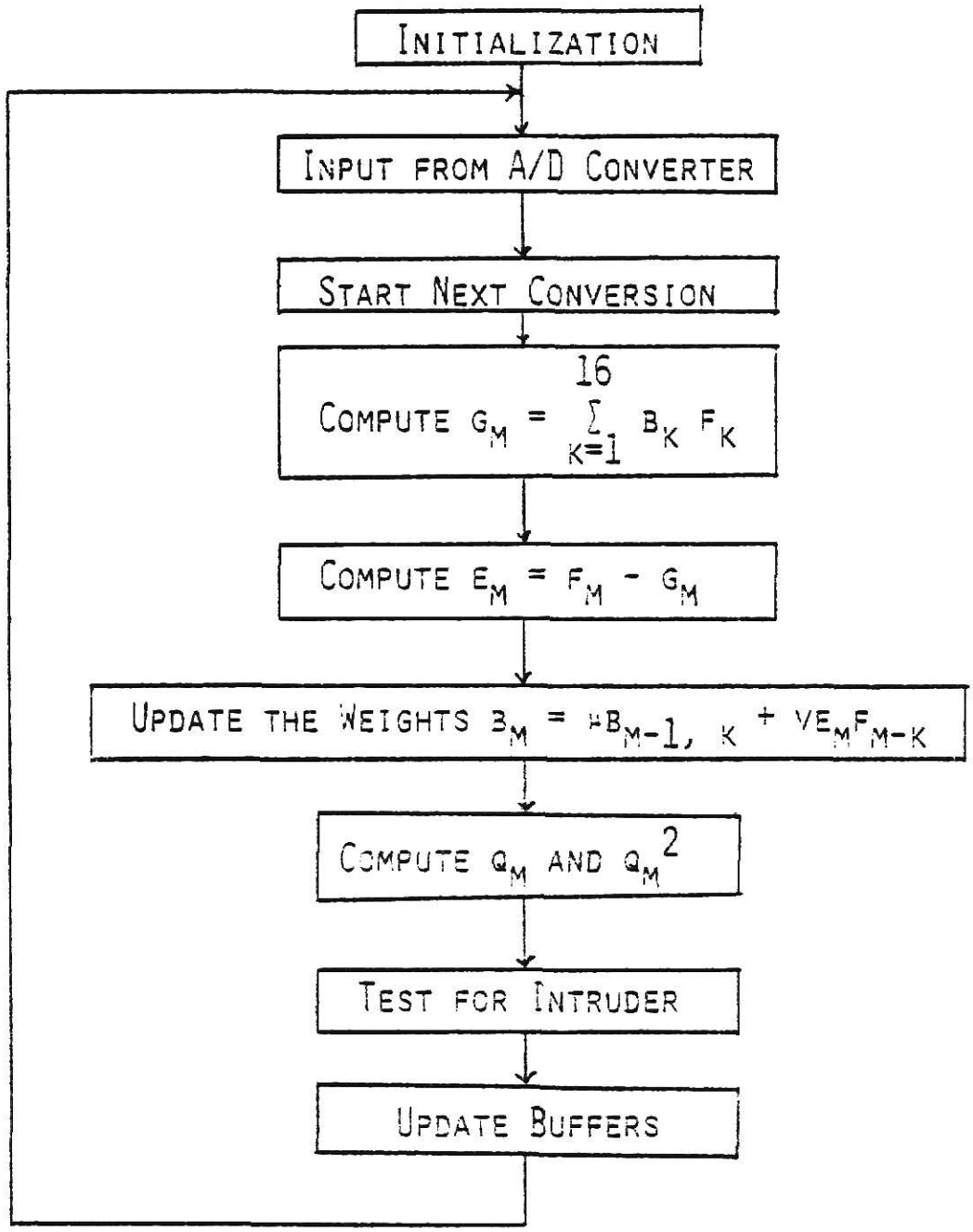


FIGURE 4.1. Z80 PROGRAM FLOWCHART
μ AND V ARE CONSTANTS,
K IS THE DELAY LENGTH,
B_K ARE THE WEIGHTS USED
IN THE LMS ALGORITHM.

A new sample is read from the A/D converter at the start of the program. Immediately thereafter, a "start conversion" pulse is sent to the A/D converter to allow it to compute the next sample while the rest of the program is being executed.

The calculation of G_M, which is the output of Widrow's LMS routine, and of the error, E_M, represent the operation of the ADP. It is then convenient to update the B_M buffer by computing the new set of weights.

The final stage of the algorithm is the MAF which operates on the elements of the E_M buffer to form the output Q_M. This output is squared to yield an idea of the magnitude of the output. This magnitude is compared to a predetermined alarm threshold. If the output magnitude exceeds the threshold, an alarm occurs. After updating the buffers and pointers, the routine returns to input a new sample.

The total program length was approximately 300 bytes of code and the necessary data memory length was about 110 bytes. The program execution rate was above 130 Hz, which was faster than the specified requirements. A complete assembler program listing is given in Appendix 1. The timing analysis appears in Table 4.1.

Results

Data files made available by Sandia Laboratories were used to test the system. These files were stored in digital form in a Data General NOVA 1200 minicomputer. The files were output through a D/A converter by the NOVA to create the analog input for the Z80 system's A/D converter. The output of the algorithm computed by the Z80 was sent back to the NOVA for evaluation. The graphs shown in Figure 4.2 compare the input

Table 4.1. Z80 Program Timing

Routine	Time in microseconds	+	Multiply Times in microseconds	=	Total Time in microseconds
Initialization	707.25	+	0.0	=	707.25
Input from A/D	21.5	+	0.0	=	21.5
Compute g	1107.25	+	16 x 147.25	=	3463.25
Compute e_m	29.75	+	0.0	=	29.75
Update the weights	1265.75	+	16 x 147.25	=	3621.75
Compute q_m	50.75	+	1 x 147.25	=	198.0
Block move of buffers	351.0	+	0.0	=	351.0
Total Times (excluding Initialization)	2826.0	+	4859.25	=	7685.25

Program execution rate $\cong$ 130 Hz.

Note: These times were calculated for a Z80 CPU clock rate of 4.0 MHz.

data signal, which in this case consisted of an intruder signal in the presence of noise generated by heavy equipment, with the Z80 algorithm output. The negative peaks result from the use of an inverting D/A converter at the output of the Z80. It is obvious that the algorithm effectively separated the input data signal into two components and eliminated the random noise generated by the equipment.

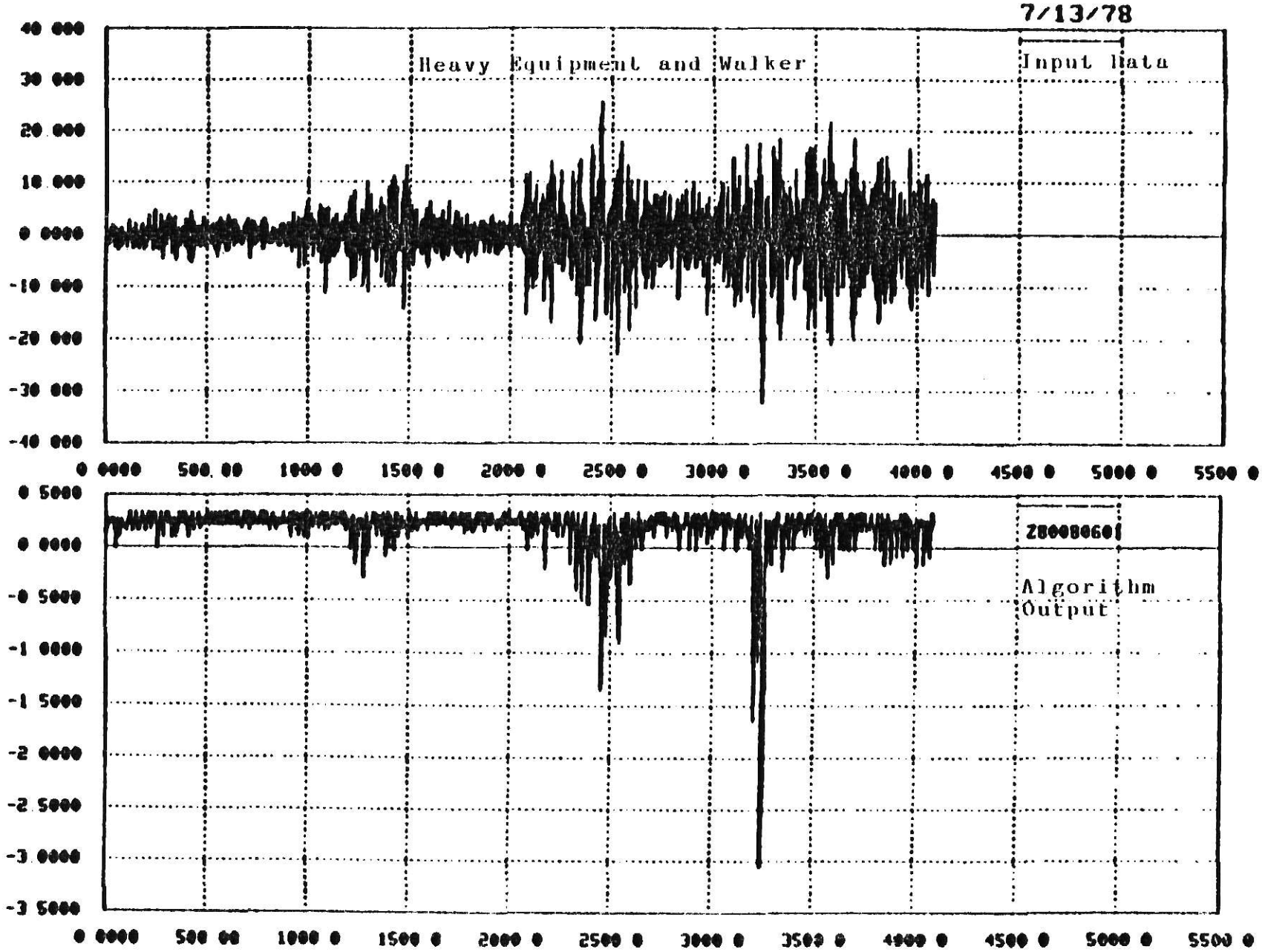

7/13/78
Heavy Equipment and Walker
Input Data
40 000
30 000
20 000
10 000
0 0000
-10 000
-20 000
-30 000
-40 000
0 0000
500 00
1000 0
1500 0
2000 0
2500 0
3000 0
3500 0
4000 0
4500 0
5000 0
5500 0
0 5000
0 0000
-0 5000
-1 0000
-1 5000
-2 0000
-2 5000
-3 0000
-3 5000
Z8008060
Algorithm
Output

Figure 4.2

CHAPTER V

THE 8748 MICROPROCESSOR IMPLEMENTATION

The 8748 is a member of Intel's single chip microprocessor family [5]. Included on the CPU chip are 1k x 8 words of EPROM program memory, 64 x 8 words of RAM data memory, 27 lines of input/output functions, and an 8-bit timer/counter. A production version, the 8048, has mask programmable ROM in place of the EPROM program memory of the 8748. Since a development system was not available for program testing, the evaluation of the 8748 consisted of a paper study which included the development of untested software routines used in the implementation of the algorithm. The assumed system was an NMOS version of the 8748 operating with a 6 MHz crystal to yield 2.5 microsecond cycle times.

Like those used in the Z80 implementation, the software routines maintain all the elements of the buffers as 16-bit words. All of the necessary arithmetic operations are performed on 16-bit arguments except the multiply subroutine, which performs an 8-bit x 8-bit signed multiply on the most significant bytes of the two arguments and returns a 16-bit result.

There appears one major difference between the routines presented here and those written for the Z80. It is due to the modification of the algorithm discussed in Chapter II to include an Adaptive Threshold

Detection Routine (ATD). Figure 5.1 shows a block diagram of the ATD technique. The output value of q_m^2 is compared with an average of previous outputs to test for an intruder.

Table 5.1 lists the software routine execution times. Obviously, the 80 Hz program operation rate is far short of the 128 Hz goal. However, selected CMOS devices at ten volts are capable of operating at more than twice their typical speed at five volts. Also, there is reason to believe that CMOS versions of the 8748 will be available in the near future. Thus a CMOS 8748 at ten volts could probably run one channel of the algorithm comfortably.

The instruction set of the 8748 lacks some important, often needed, instructions. Programming ease as well as program speed would be much improved if some of the following operations were added to the existing 8748 instruction set:

1. Binary or 2's complement multiply
2. Subtraction
3. Arithmetic shifts
4. Data transfers between registers
5. Any 16-bit operations.

A further inconvenience resulted from the need for external RAM to supplement the internal RAM data memory.

Complete software program listings are given in Appendix 2. Also included are some comments on a possible system structure.

For this version of the algorithm, the program length was about 500 bytes of code. 500 bytes of RAM were also required for data memory.

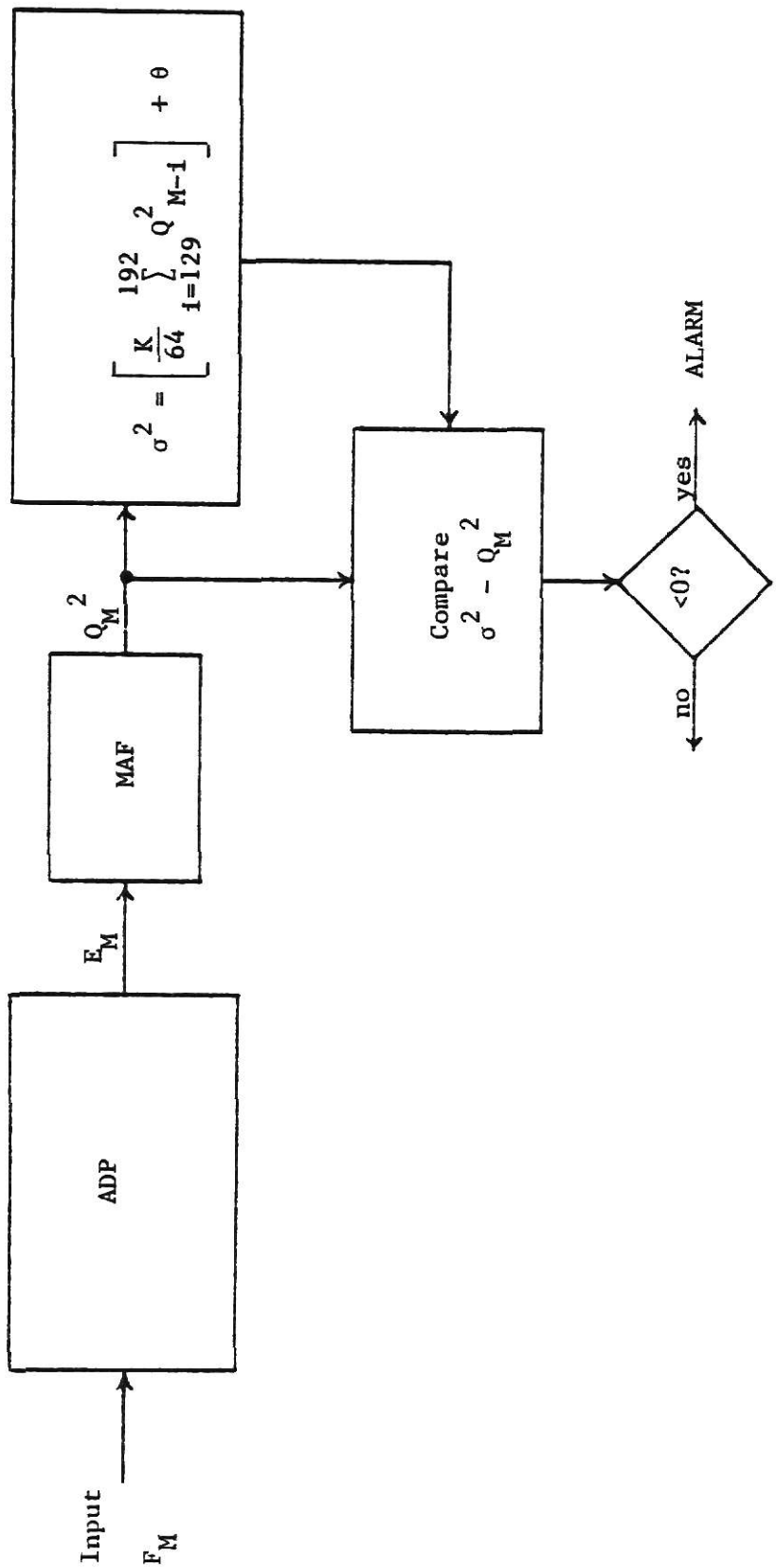


Figure 5.1. 8748 Algorithm Block Diagram

Table 5.1. 8748 Program Timing

Routine	Time in microseconds	+	Multiply Times in microseconds	=	Total Time in microseconds
Initialization	6082.5	+	0.0	=	6082.5
Input from A/D	77.5	+	0.0	=	77.5
Compute g	1235.0	+	16 x 252.5	=	5275.0
Compute e_m	87.5	+	0.0	=	87.5
Update the weights	2025.0	+	16 x 252.5	=	6065.0
Compute q and q^2	135.0	+	1 x 252.5	=	387.5
ATD	317.5	+	1 x 252.5	=	570.0
Update F and E buffers	102.5	+	0.0	=	102.5
Total	3980.0	+	8585.0	=	12565.0
Program Execution Rate ≅ 80 Hz					

All times are calculated for a 2.5 μsec cycle time. Totals do not include Initialization time.

CHAPTER VI

THE RCA ATMAC IMPLEMENTATION

Referring to the RCA ATMAC as a microprocessor stretches the meaning of the word considerably. The ATMAC is a CMOS/SOS, very low power device which is designed with an 8-bit-slice architecture that is meant to be used in high speed signal processing and array processing applications [6]. It has an excellent instruction set and, when combined with a hardware multiply and accumulate special function unit, is capable of very fast algorithm operation. Unfortunately, the capabilities of the ATMAC are currently more than matched by the system's cost and complexity.

The ATMAC CPU uses two 8-bit modular chip types, each of which is a 64-pin package. One is the Data Execution Unit. It contains eight general registers, performs the arithmetic and logic functions on the data, and supplies the operands for any Special Function Unit. The other chip is the Instruction and Operand Fetch Unit. It contains eight indirect address registers, the program sequence counter, and a four-word program counter stack. This study assumed the use of a 16-bit system which would require two of each of these CPU chips. Since four separate busses are provided for the program memory address and data lines and the data memory and input/output address and data lines, the ATMAC can execute one instruction while it is fetching the next. The result is a very fast processor with instruction execution times of 280 nanoseconds for short cycle instructions and 350 nanoseconds for long cycle instructions.

For machine configurations with data words of 16 bits or less, the minimum instruction word length is 24 bits [6]. Two types of instructions are used by the ATMAC. Type I instructions allow the contents of various registers to be used as operands, and Type II instructions specify an immediate operand. Since a total of four register operands as well as a processor mode control can be specified by one instruction, the ATMAC instruction set is very flexible and powerful. The disadvantage of this is that assembler program writing and reading is rather difficult and clearly understanding the operation of a program is not an easy task.

The algorithm used for the ATMAC implementation was somewhat different than those used in the previously discussed implementations. The ADP as was originally shown in Figure 2.1 was left unchanged. However, the MAF was eliminated, and a new version of the ATD was inserted. Figure 6.1 shows a block diagram of the revised algorithm used here. In addition to the algorithm components shown in Figure 6.1, a low pass digital filter was included at the point of sample input. This filter and the routine to input a sample from an A/D converter were omitted from this study. Also changed from the original algorithm was the required speed of operation. It was decided that program operation rates of 32 Hz or less were adequate.

Table 6.1 lists the timing of the software routines. The 11.5 kHz operation rate indicates that the ATMAC is indeed an extremely fast processor. Even faster times could probably be achieved, but their value would be minimal for this application. The ATMAC would be most valuable for applications requiring much faster speeds than this algorithm needs such as real time speech processing or array processing.

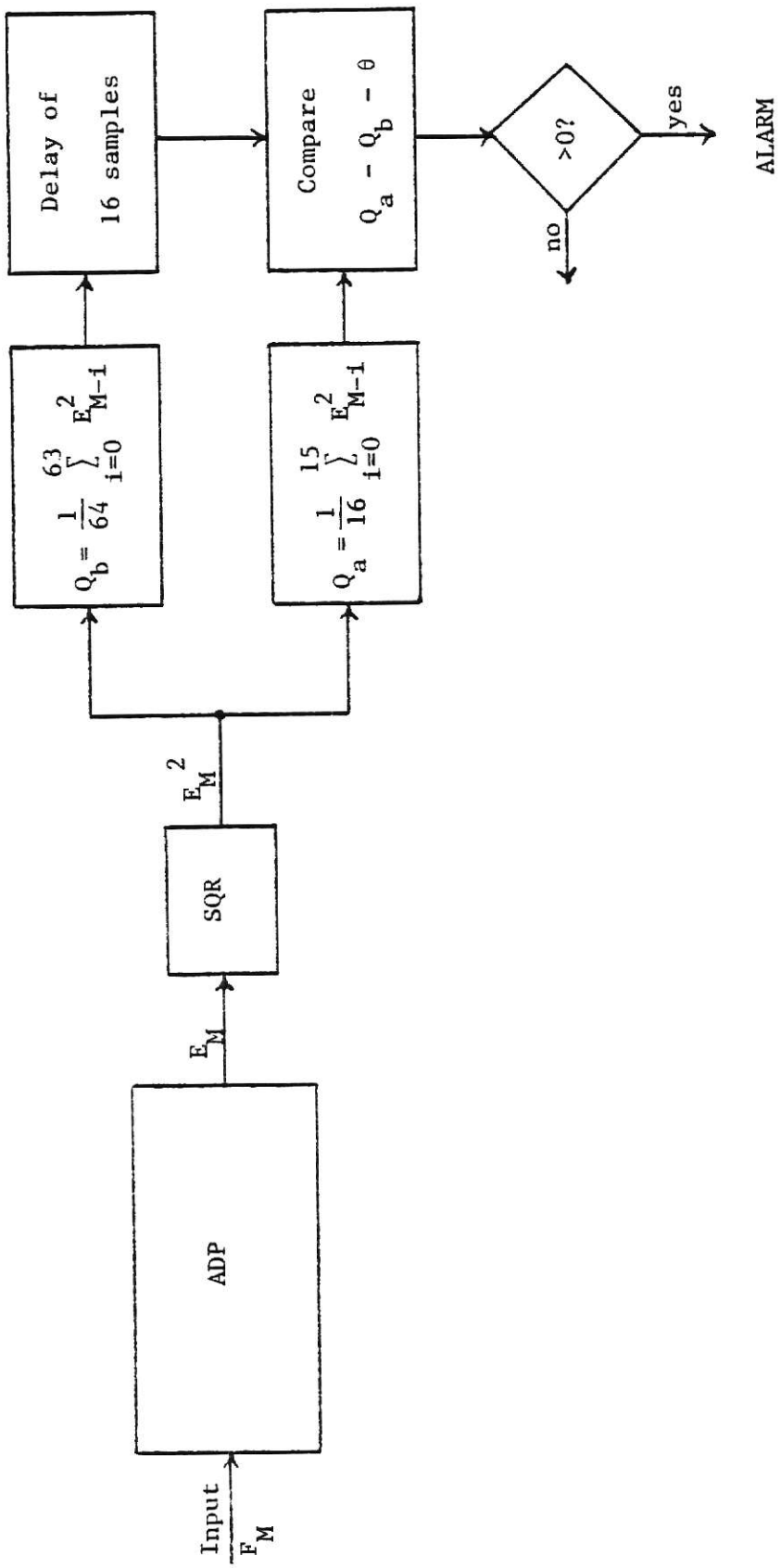


Figure 6.1. ATMAC Algorithm Block Diagram

The complete software routines and supporting documentation appear in Appendix 3. None of the routines have been tested. They were written only for the purpose of evaluating the capabilities of the ATMAC with respect to this application.

Table 6.1. ATMAC Program Timing

	Program Routine	Execution Time(microseconds)
1.	Initialization	42.77
2.	Input from A/D (estimate)	1.00
3.	Compute g	11.90
4.	Compute e_m	0.56
5.	Update the weights	39.69
6.	Adaptive Threshold Detection	6.02
7.	Update buffers	23.03
8.	Digital Filter and other I/O Routines not included	5.00

Total Program Time (without Initialization Routine) is,

$T = 87.20$ microseconds

Program Operation Rate is,

$1/T \cong 11.5$ kHz

Note: These times were calculated as follows:
short cycle instruction time = 280 nsec.
long cycle instruction time = 350 nsec.

CHAPTER VII

COMPARISON OF THE VARIOUS IMPLEMENTATIONS

Having thoroughly studied three possible algorithm implementations, it is useful to form some ideas about how they compare with each other. However, since the 8748, Z80, and ATMAC represent a full range of microprocessor sophistication levels, comparisons are hard to make. For instance, all three of the devices have many advantages with respect to the type of tasks they were designed to perform. Yet the ATMAC and 8748 fall on opposite ends of the spectrum of available microprocessor systems in terms of cost, complexity, and capability while the Z80 falls somewhere in the middle of that spectrum. In this chapter, some ideas regarding the merit of the respective implementations will be formed based on various attributes of the microprocessors. Some of the criteria used will be deterministic and fixed while others will be subjective in nature and will to a large extent reflect the opinions of the author.

The 8748 has several nice features. It was designed to simplify many microprocessor system designs by including program and data memory on the chip. A CMOS version would allow operation at ten volts and thus all of the advantages of CMOS, including very low power consumption and good noise immunity as well as the potential for sufficient program operation speeds, would add to the attractiveness of the 8748.

The 8748 suffers from a number of disadvantages. The intended system simplicity limits the ability of the device to perform programs which require more data or program memory than is available internally. The instruction set is not as complete or versatile as one would desire. This results in more lengthy programs which are not always easily understood. It also restricts the ease of data flow to and from external memory.

An implementation based on the 8748 would use a very simple hardware configuration. It would include a minimum number of chips and would be easy and relatively inexpensive to build. The software limitations, however, would become annoying if many program changes were necessary.

Initially, the major disadvantages of the Z80 implementation seemed to stem from the fact that an NMOS version of this 8-bit microprocessor at five volts lacked the necessary speed to perform an algorithm which required the use of 16-bit operands and that it consumed a significant amount of power. However, since National Semiconductor recently began to produce CMOS versions of the Z80, the problem of power consumption has been eliminated, and the speed of the device can probably be improved.

The Z80 is a very capable 8-bit microprocessor. It has the advantage of an excellent instruction set. The problem of implementing an algorithm which requires the use of 16-bit operands is greatly reduced by the many 16-bit operations included in the Z80's instruction set. The only serious problem remaining is due to the lack of a multiply instruction. The need to call a multiply subroutine repeatedly throughout the program limits the possible operation speed of the program, but sufficient speeds can still be achieved for this algorithm. In general, the software capabilities of the Z80 are much better than those of the 8748.

The minimal Z80 hardware system described in Chapter III illustrates the simplicity of a possible implementation as well as the relatively low cost of such a system. Although it is slightly more complex than a minimal 8748 system might be, the difference is insignificant compared to the advantage of the Z80's superior instruction set.

The ATMAC is in a class of its own at the top end of the microprocessor spectrum. Its CMOS/SOS construction yields extremely fast program execution speeds and requires only a small amount of power. In no way does it have difficulty meeting the requirements of the algorithm. The instruction set is excellent and is very well suited to signal processing applications.

Unfortunately, the few disadvantages that the ATMAC does have are very major. A fairly high degree of system complexity is expected for a device of this nature and in this case the system complexity is not out of line with the capabilities of the ATMAC. A more severe problem comes from the fact that the ATMAC is not a particularly friendly microprocessor. The long instruction word includes fields to specify several variations for any instruction and thus makes the interpretation of a program quite difficult. Again, this problem is not out of line with the capabilities of the ATMAC and to some extent it is an expected problem. The biggest disadvantage of the ATMAC is its cost, which currently is excessive even for a microprocessor with its ability.

CHAPTER VIII

CONCLUSIONS

It is the opinion of the author that none of the implementations described here make use of the available microprocessors which would be most closely matched to the task of implementing this algorithm. The excessive overkill capabilities of the ATMAC would be an advantage and would allow a great deal of flexibility in future algorithm modifications. The ATMAC's present cost, however, is too high to justify its use for this particular algorithm.

CMOS versions of the 8748 and of the Z80 would both be capable of implementing the algorithm. The hardware systems would take on slightly different appearances due to their architectures and would vary somewhat in terms of cost. These differences would probably not be significant. Therefore, the Z80, due to its superior instruction set and operation speed, would seem to be the most practical implementation of the three described in this report. At lower speeds, the Z80 could perform 16-bit x 16-bit multiplies in addition to performing all of the other necessary 16-bit arithmetic and logic operations. Although this would not be an ideal solution to the implementation of this algorithm, it would be a reasonable one.

ACKNOWLEDGEMENTS

This work was sponsored and funded by the Base and Installation Security Systems Programs Office, Electronics Systems Division of the Air Force Systems Command, Hanscom Air Force Base, MA 01731.

The author would like to thank Sandia Laboratories for their support of this project. Appreciation is also due to Dr. N. Ahmed and to Dr. D. H. Lenhert, who served as committee members. Finally, the author would like to express sincere thanks to Dr. M.S.P. Lucas who served as the author's major advisor and provided a great deal of assistance.

REFERENCES

1. N. Ahmed, On Intrusion-Detection via Adaptive Prediction, SAND77-1591, Sandia Laboratories, Albuquerque, NM, April 1978.

2. B. Widrow, et al., "Stationary and Nonstationary Learning Characteristics of the Adaptive LMS Filter," Proc. IEEE, Vol. 64, p. 1151, August 1976.

3. "Z80-Assembly Language Programming Manual," Zilog Corporation.

4. "Z80-CPU Technical Manual," Zilog Corporation.

5. "MCS-48 Microcomputer User's Manual," Intel Corporation.

6. "ATMAC Programmer's Reference Manual," RCA Advanced Technology Laboratories, Camden, NJ.

APPENDIX 1

Z80 WIDROW ALGORITHM ASSEMBLER PROGRAM

Memory Organization:

Location	Contents		Location	Contents	
4ØØØ HEX	f_{m-16}	L.O.	4Ø42	e_{m-16}	L.O.
4ØØ1	f_{m-16}	H.O.	4Ø43	e_{m-16}	H.O.
4ØØ2	f_{m-15}	L.O.	4Ø44	e_{m-15}	L.O.
4ØØ3	f_{m-15}	H.O.	4Ø45	e_{m-15}	H.O.
⋮	⋮		⋮	⋮	
4Ø1E	f_{m-1}	L.O.	4Ø6Ø	e_{m-1}	L.O.
4Ø1F	f_{m-1}	H.O.	4Ø61	e_{m-1}	H.O.
4Ø2Ø	f_{m}	L.O.	4Ø62	e_{m}	L.O.
4Ø21	f_{m}	H.O.	4Ø63	e_{m}	H.O.
4Ø22	$b_{m,16}$	L.O.	4Ø64	g(l.s.b.)	L.O.
4Ø23	$b_{m,16}$	H.O.	4Ø65	g(l.s.b.)	H.O.
4Ø24	$b_{m,15}$	L.O.	4Ø66	g(m.s.b.)	L.O.
4Ø25	$b_{m,15}$	H.O.	4Ø67	g(m.s.b.)	H.O.
⋮	⋮		4Ø68	q_{m}	L.O.
			4Ø69	q_{m}	H.O.
4Ø4Ø	$b_{m,1}$	L.O.	4Ø6A	Ve_{m}	L.O.
4Ø41	$b_{m,1}$	H.O.	4Ø6B	Ve_{m}	H.O.

Note: The F buffer and the E buffer are circulating buffers. However, rather than maintaining circulating pointers to these buffers, they are updated each program iteration by block moves.

Multiply Subroutine

Note: This subroutine performs an 8-bit x 8-bit signed multiply of the contents of registers D and H. The result is returned in register pair HL with DE = ØØØØ. This routine uses the FUDGE method.

	Location	Instruction	Code	Comments
	Ø2ØØH	LD E,H	5C	D=MPX ,E=MPY
	1	XOR A	AF	clear A
	2	LD H,A	67	clear H
	3	LD L,A	6F	clear L
	4	LD C,A	4F	clear C
	5	LD B,Ø8	Ø6	B=iteration counter
	6		Ø8	
	7	BIT 7,D	CB	test sign of MPX
	8		7A	
	9	JR NZ,ADJ1	2Ø	if negative, go to
	A		18	ADJ1, else, continue
RET1	B	BIT 7,E	CB	test sign of MPY
	C		7B	
	D	JR NZ,ADJ2	2Ø	if negative, go to
	E		18	ADJ2, else, continue
RET2	F	SRL D	CB	shift right logical
	1Ø		3A	MPX
	1	JR NC,SKIP	3Ø	test l.s.b. of MPX
	2		Ø3	if Ø, go to SKIP
	3	LD A,H	7C	else, add MPY to
	4	ADD E	83	high part of the
	5	LD H,A	67	result
SKIP	6	RR H	CB	shift right HL
	7		1C	
	8	RR L	CB	
	9		1D	
	A	DJNZ,e	1Ø	decrement iteration
	B		F3	counter and repeat if > Ø
	C	LD A,H	7C	subtract contents
	D	SUB C	91	of FUDGE register
	E	LD H,A	67	from high part of result
	F	XOR A	AF	
	2Ø	LD D,A	57	clear DE
	1	LD E,A	5F	
	2	RET	C9	return to calling program
ADJ1	3	LD A,E	7B	
	4	LD C,E	4B	if MPX < Ø, add
	5	JR RET1	18	MPY to FUDGE
	6		E4	
ADJ2	7	ADD D	82	if MPY < Ø, add
	8	LD C,A	4F	MPX to FUDGE
	9	JR RET2	18	
	Ø22A		E4	

Initialization

This routine clears the buffer contents in memory. The program execution begins here at location ØØØØ.

Location	Instruction	Code	Comments
ØØØØH	LD B,n	Ø6	set up iteration counter
1		6C	
2	XOR A	AF	clear A
3	LD HL,nn	21	load HL with
4		6B	memory pointer
5		4Ø	
6	LD (HL),A	77	clear memory byte
7	DEC HL	2B	decrement HL
8	DJNZ,e	1Ø	decrement iteration counter
9		FC	and repeat if > Ø

Input from A/D Converter

This routine inputs the new sample from the A/D converter as is explained in Chapter III. This input is scaled by an arithmetic shift right and is stored in memory.

Location	Instruction	Code	Comments
ØØØAH	XOR A	AF	clear A
B	LD L,A	6F	clear L
C	LD A,(nn)	3A	input sample
D		ØØ	from A/D
E		CØ	
F	LD B,A	47	B = sample
1Ø	LD A,(nn)	3A	start the next
1		ØØ	conversion
2		8Ø	
3	LD A,n	3E	A = mask
4		8Ø	convert sample to
5	XOR B	A8	2's complement form
6	LD H,A	67	H = sample
7	SRA H	CB	scale sample
8		2C	by arithmetically
9	RR L	CB	shifting right HL
A		1D	
B	LD (nn),HL	22	store f_m in memory
C		2Ø	
ØØ1D		4Ø	

Compute $g = \sum_{k=1}^{16} f_{m-k}\, b_{m,k}$

Location	Instruction	Code	Comments
ØØ1E	LD HL,nn	21	clear HL
F		ØØ	
2Ø		ØØ	
1	LD (nn),HL	22	clear low order 16 bits
2		64	of g
3		4Ø	
4	LD (nn),HL	22	clear high order 16 bits
5		66	of g
6		4Ø	
7	LD IX,nn	DD	load index register X
8		21	with address of L.O.
9		22	byte of $b_{m,16}$
A		4Ø	
B	LD IY,nn	FD	load index register Y
C		21	with address of L.O.
D		ØØ	byte of f_{m-16}
E		4Ø	
F	LD B,n	Ø6	set up iteration counter
3Ø		1Ø	
1	LD L,(IX+D)	DD	
2		6E	HL = $b_{m,k}$
3		ØØ	
4	LD H,(IX+D)	DD	
5		66	
6		Ø1	
7	LD E,(IY+D)	FD	
8		5E	
9		ØØ	DE = f_{m-k}
A	LD D,(IY+D)	FD	
B		56	
C		Ø1	
D	PUSH BC	C5	save the counter
E	CALL	CD	call multiply
F		ØØ	subroutine
4Ø		Ø2	
1	INC IX	DD	increment buffer
2		23	pointer
3	INC IX	DD	
4		23	
5	INC IY	FD	increment buffer
6		23	pointer
7	INC IY	FD	
ØØ48		23	

Location	Instruction	Code	Comments
ØØ49	LD BC,(nn)	ED	load BC with L.O.
A		4B	16 bits of g
B		64	
C		4Ø	
D	EX DE,HL	EB	update L.O. 16 bits
E	ADD HL,BC	Ø9	of g and store
F	LD (nn),HL	22	in memory
5Ø		64	
1		4Ø	
2	EX DE,HL	EB	
3	LD BC,(nn)	ED	load BC with H.O.
4		4B	16 bits of g
5		66	
6		4Ø	
7	ADC HL,BC	ED	update H.O. 16 bits
8		4A	of g and store
9	LD (nn),HL	22	in memory
A		66	
B		4Ø	
C	POP BC	C1	retrieve counter
D	DJNZ,e	1Ø	decrement counter and
ØØ5E		D2	repeat if > Ø

Compute $e_m = f_m - g$

Note: Enter this routine with HL = g (H.O. 16 bits).

Location	Instruction	Code	Comments
ØØ5F	EX DE,HL	EB	DE = g (H.O. 16 bits)
6Ø	LD HL,(nn)	2A	
1		2Ø	HL = f_m
2		4Ø	
3	XOR A	AF	clear A, clear carry
4	SBC HL,DE	ED	HL = $f_m - g = e_m$
5		52	
6	SRA H	CB	
7		2C	
8	RR L	CB	
9		1D	
A	SRA H	CB	
B		2C	divide by 2^4
C	RR L	CB	to form
D		1D	HL = $e_m/2^4$
E	SRA H	CB	
F		2C	
7Ø	RR L	CB	
1		1D	
2	SRA H	CB	
3		2C	
4	RR L	CB	
5		1D	
6	LD (nn),HL	22	store $e_m/2^4$ in memory
7		62	
ØØ78		4Ø	

Update the Weights $b_{m+1,k} = ub_{m,k} + Ve_m f_{m-k}$

Note: Enter this routine with $Ve_m = e_m/2^4$ in HL.

Location	Instruction	Code	Comments
ØØ79	LD IX,nn	DD	load index register X
A		21	with the address of
B		22	the L.O. byte of $b_{m,16}$
C		4Ø	
D	LD IY,nn	FD	load index register Y
E		21	with the address of
F		ØØ	the L.O. byte of f_{m-16}
8Ø		4Ø	
1	LD (nn),HL	22	store Ve_m in memory
2		6A	
3		4Ø	
4	LD B,n	Ø6	set up iteration counter
5		1Ø	
6	PUSH BC	C5	save counter
7	LD HL,(nn)	2A	HL = Ve_m
8		6A	
9		4Ø	
A	LD E,(IY+D)	FD	
B		5E	DE = f_{m-k}
C		ØØ	
D	LD D,(IY+D)	FD	
E		56	
F		Ø1	
9Ø	CALL	CD	call multiply subroutine
1		ØØ	
2		Ø2	
3	LD E,(IX+D)	DD	
4		5E	
5		ØØ	DE = $b_{m,k}$
6	LD D,(IX+D)	DD	
7		56	
8		Ø1	
9	EX DE,HL	EB	
A	LD B,H	44	form $b_{m,k}/2^{10}$
B	LD C,H	4C	
C	SRA C	CB	
D		29	
E	SRA C	CB	
ØØ9F		29	

Location	Instruction	Code	Comments
ØØAØ	XOR A	AF	clear A
1	RL B	CB	rotate m.s.b. into
2		1Ø	carry
3	JR NC,e	3Ø	fill B with sign
4		Ø1	bit
5	CPL A	2F	
6	LD B,A	47	
7	XOR A	AF	clear A
8	SBC HL,BC	ED	HL = $ub_{m,k}$
9		42	$= b_{m,k} (1 - 2^{10})$
A	ADD HL,DE	ED	HL = $b_{m+1,k}$
B		5A	
C	LD (IX+D),L	DD	store updated
D		75	$b_{m+1,k}$ in memory
E		ØØ	
F	LD (IX+D),H	DD	
BØ		74	
1		Ø1	
2	INC IX	DD	increment pointers
3		23	
4	INC IX	DD	
5		23	
6	INC IY	FD	
7		23	
8	INC IY	FD	
9		23	
A	POP BC	C1	retrieve counter
B	DJNZ,e	1Ø	decrement counter and
ØØBC		C9	repeat if > Ø

Compute $q_m = \frac{1}{16} \sum_{k=1}^{16} e_{m-k}$

Location	Instruction	Code	Comments
ØØBD	LD HL,(nn)	2A	HL = $e_{m-16}/16$
E		42	
F		4Ø	
CØ	EX DE,HL	EB	exchange DE and HL
1	LD HL,(nn)	2A	
2		62	HL = $e_m/16$
3		4Ø	
4	XOR A	AF	clear A, clear carry
5	SBC HL,DE	ED	HL = $(e_m - e_{m-16})/16$
6		52	
7	EX DE,HL	EB	exchange DE and HL
8	LD HL,(nn)	2A	
9		68	HL = old q_m
A		4Ø	
B	ADD HL,DE	19	HL = new q_m
C	LD (nn),HL	22	store new value of
D		68	q_m in memory
E		4Ø	
F	LD D,H	54	DE = HL = q_m
DØ	LD E,L	5D	
1	CALL	CD	call multiply subroutine
2		ØØ	form $q_m^2/16^2$
3		Ø2	
4	SLA L	CB	
5		25	
6	RL H	CB	
7		14	
8	SLA L	CB	
9		25	multiply by 16
A	RL H	CB	to form
B		14	HL = $q_m^2/16$
C	SLA L	CB	
D		25	
E	RL H	CB	
F		14	
EØ	SLA L	CB	
1		25	
2	RL H	CB	
3		14	
4	LD A,H	7C	output result to
5	OUT [n],A	D3	output port
ØØE6		ØØ	

NOTE: The information at the output port can be used as an input to a D/A converter to obtain the algorithm output. This is the quantity which was plotted in the results given in Chapter IV.

Block move of E and F buffers

Location	Instruction	Code	Comments
ØØE7	LD DE,nn	11	load target address
8		42	
9		4Ø	
A	LD HL,nn	21	load source address
B		44	
C		4Ø	
D	LD BC,nn	Ø1	load number of bytes
E		2Ø	to be moved
F		ØØ	
FØ	LDIR	ED	block move of
1		BØ	E buffer
2	LD DE,nn	11	load target address
3		ØØ	
4		4Ø	
5	LD HL,nn	21	load source address
6		Ø2	
7		4Ø	
8	LD BC,nn	Ø1	load number of bytes
9		2Ø	to be moved
A		ØØ	
B	LDIR	ED	block move of
C		BØ	F buffer
D	JP nn	C3	jump to Input from
E		ØA	A/D Routine
ØØFF		ØØ	

APPENDIX 2

8748 WIDROW ALGORITHM SYSTEM ORGANIZATION

The 1k x 8 internal ROM program memory is more than enough for this program. Some care must be taken to place the routines in memory so that the conditional jumps are to a location on the same page.

The internal 64 bytes of RAM are not enough to contain the buffers. 512 bytes of external RAM are organized as two 256 byte pages. A bit of Port 1 is used to select the page and the 8 bit BUS is used for address and data transfer.

Only the T buffer uses page 1, so user flags F0 and F1 are used to remember the page that T buffer pointers TLPTR and TLPTR+64 are currently pointing to. All other buffer pointers refer to page 0.

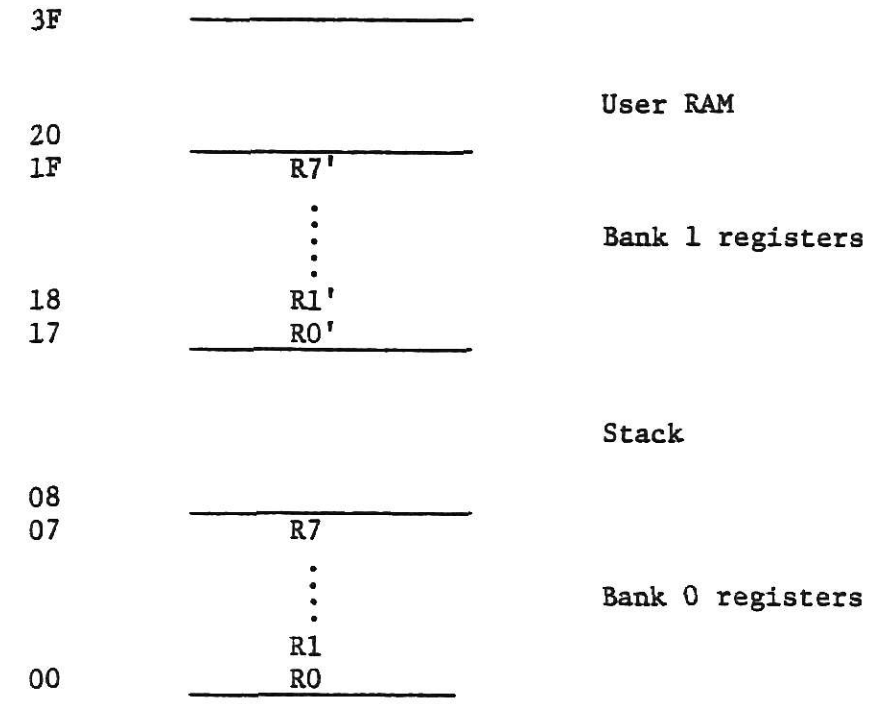
I/O Port 2 is used to input from the A/D converter, and I/O Port 1 is used for control.

I/O Port 1	Bit 0 - external RAM page select
	Bit 1 - ALARM
	Bit 2 - control to A/D, D/A
	Bit 3 - control to A/D, D/A

The internal timer is used to maintain constant program speed.

Internal RAM

Bank 0 registers are selected on reset.



Location	Contents	
20 H	FLPTR	points to $f_{(m-16)}$
21 H	ELPTR	points to $e_{(m-16)}$
22 H	$f_{(m)}$ (H.O.)	new sample from A/D
23 H	$f_{(m)}$ (L.O.)	
24 H	$e_{(m)}$ (H.O.)	$f_{(m)} - g$
25 H	$e_{(m)}$ (L.O.)	
26 H	g (H.O.)	
27 H	g (L.O.)	
28 H	$q/2^4$ (H.O.)	sum of elements of
29 H	$q/2^4$ (L.O.)	E buffer
2A H	$q^2/2^4$ (H.O.)	
2B H	$q^2/2^4$ (L.O.)	
2C H	TLPTR	points to $t_{(m-192)}$
2D H	TLPTR+64	points to $t_{(m-128)}$
2E H	SUM (H.O.)	sum of oldest 64 elements
2F H	SUM (L.O.)	of T buffer

External RAM

Page 0

00	$f_{(m-16)}$	(H.O.)	
01	$f_{(m-16)}$	(L.O.)	
	⋮		F buffer circulating
1E	$f_{(m-1)}$	(H.O.)	
1F	$f_{(m-1)}$	(L.O.)	
20	$b_{(m,16)}$	(H.O.)	
21	$b_{(m,16)}$	(L.O.)	
	⋮		B buffer non-circulating
3E	$b_{(m,1)}$	(H.O.)	
3F	$b_{(m,1)}$	(L.O.)	
40	$e_{(m-16)}$	(H.O.)	
41	$e_{(m-16)}$	(L.O.)	
	⋮		E buffer circulating
5E	$e_{(m-1)}$	(H.O.)	
5F	$e_{(m-1)}$	(L.O.)	
60	$t_{(m-192)}$	(H.O.)	
61	$t_{(m-192)}$	(L.O.)	
	⋮		T buffer circulating
FE	$t_{(m-113)}$	(H.O.)	
FF	$t_{(m-113)}$	(L.O.)	

Page 1

00	$t_{(m-112)}$	(H.O.)	
01	$t_{(m-112)}$	(L.O.)	
	⋮		T buffer (cont.)
DE	$t_{(m-1)}$	(H.O.)	
DF	$t_{(m-1)}$	(L.O.)	

8748 Widrow Algorithm Program

8-bit x 8-bit signed multiply

This routine is called by the standard subroutine CALL command.
Call this subroutine with R2 = MPX and R3 = MPY.
The result is returned in A (H.O.) and R2 (L.O.).

R4 = counter
R5 = FUDGE

Ø5Ø	MLT	MOV	R5,#ØØ	BD	clear FUDGE
1				ØØ	
2		MOV	A,R2	FA	A = MPX
3		JB7	FIX1	F2	test sign, if neg.
4				5A	go to FIX1
5	TEST	MOV	A,R3	FB	A = MPY
6		JB7	FIX2	F2	test sign, if neg.
7				5E	go to FIX2
8		JMP	GO	Ø4	if pos., go to GO
9				61	
A	FIX1	MOV	A,R3	FB	if MPX is neg.,
B		MOV	R5,A	AD	add MPY to FUDGE
C		JMP	TEST	Ø4	
D				55	
E	FIX2	MOV	A,R5	FD	if MPY is neg.,
F		ADD	A,R2	6A	add MPX to FUDGE
6Ø		MOV	R5,A	AD	
1	GO	MOV	R4,#Ø8	BC	initialize counter
2				Ø8	
3		CLR	A	27	clear A
4		CLR	C	97	clear carry bit
5	MLOOP	RRC	A	67	rotate right A
6		XCH	A,R2	2A	and R2
7		RRC	A	67	carry = bit shifted
8		XCH	A,R2	2A	out of MPX
9		JNC	MCONT	E6	if no carry, go to
A				6C	MCONT
B		ADD	A,R3	6B	if carry, add MPY
C	MCONT	DJNZ	R4,MLOOP	EC	to A
D				65	decrement and repeat until counter = 0
E		RRC	A	67	rotate right
F		XCH	A,R2	2A	A and R2
7Ø		RRC	A	67	
1		XCH	A,R2	2A	
2		XCH	A,R5	2D	subtract FUDGE
3		CPL	A	37	from high
4		INC	A	17	part of
5		ADD	A,R5	6D	result
Ø76		RET		83	return to calling program

Total number of cycles = 101
Time at 2.5 µsec/cycle = 252.5 µsec.

Initialization

This routine clears all external RAM buffer locations and initializes all pointers and flags used by the program.

```
ØD8   INIT     MOV    RØ,#2Ø      B8          RØ = location of
  9                                 2Ø            FLPTR
  A            CLR    A           27
  B            MOV    @RØ,A       AØ          set FLPTR = Ø
  C            INC    RØ          18          RØ = location of
  D            MOV    A,#4Ø       23            ELPTR
  E                                 4Ø
  F            MOV    @RØ,A       AØ          set ELPTR = 4Ø
 EØ            MOV    RØ,#2C      B8          RØ = location of
  1                                 2C            TLPTR
  2            MOV    A,#6Ø       23
  3                                 6Ø
  4            MOV    @RØ,A       AØ          set TLPTR = 6Ø
  5            INC    RØ          18          RØ = location of
  6            MOV    A,#9E       23            TLPTR+64
  7                                 9E
  8            MOV    @RØ,A       AØ          set TLPTR+64 = 9E
  9            CLR    FØ          85          clear FØ
  A            CLR    F1          A5          clear F1
  B            CLR    A           27          clear A
  C            MOV    RØ,A        A8          clear RØ
  D            MOV    R7,#ØØ      BF          set counter = ØØ
  E                                 ØØ
  F            ANL    P1,#FE      99          select external
 FØ                                FE            RAM page Ø
  1   ICONT    MOVX   @RØ,A       9Ø          clear all of
  2            INC    RØ          18            external RAM
  3            DJNZ   R7,ICONT    EF            page Ø
  4                                 F1
  5            ORL    P1,#Ø1      89          select external
  6                                 Ø1            RAM page 1
  7            CLR    A           27          clear A
  8            MOV    RØ,A        A8          clear RØ
  9            MOV    R7,#EØ      BF          set counter = EØ
  A                                 EØ
  B   ICONT2   MOVX   @RØ,A       9Ø          clear locations
  C            INC    RØ          18            ØØ to DF in
  D            DJNZ   R7,ICONT2   EF            external RAM
  E                                 FB            page 1
ØFF            OUTL   P1,A        39          output ØØ to page 1,
                                                 sets ALARM = Ø, selects
                                                 external RAM page Ø
```

Total number of cycles = 2433
Time at 2.5 µsec/cycle = 6082.5 µsec

Input from A/D

This routine controls the internal timer and inputs the samples from the A/D converter. It also stores the new sample at a temporary location in internal RAM. This routine was written for the 8-bit Teledyne 8700 A/D converter.

```
1ØØ   AD       STOP   T          65          stop timer
  1             MOV    A,#__      23          load timer with
  2                                __            the desired
  3             MOV    T,A        62            constant
  4             STRT   T          45          start timer
  5             MOV    R2,#ØØ     BA          clear R2
  6                                ØØ
  7             IN     A,P2       ØA          input sample
  8             ORL    P1,#Ø4     89          output control
  9                                Ø4            pulse to start
  A             XRL    A,#8Ø      D3            next conversion
  B                                8Ø          convert offset
  C             RRC    A          67            binary to 2's comp.
  D             XCH    A,R2       2A
  E             RRC    A          67
  F             XCH    A,R2       2A          arithmetic shift
 1Ø             JB6    SET7       D2            right A
  1                                16            and R2
  2             ANL    A,#7F      53
  3                                7F
  4             JMP    NCONT      24
  5                                18
  6   SET7      ORL    A,#8Ø      43
  7                                8Ø
  8   NCONT     MOV    RØ,#22     B8          store sample
  9                                22            at internal
  A             MOV    @RØ,A      AØ            locations 22
  B             INC    RØ         18            and 23
  C             XCH    A,R2       2A
  D             MOV    @RØ,A      AØ
  E             ANL    P1,#FB     99          output control
11F                                FB            pulse to A/D
```

Total number of cycles = 31
Time at 2.5 µsec/cycle = 77.5 µsec

Calculation of $g = \sum_{k=1}^{16} b_{(m,k)} \, f_{(m-k)}$

This routine calculates g. The b's and f's are in external RAM page 0. The result is stored in internal RAM.

12Ø		CLR	A	27	clear A
1		MOV	RØ,#26	B8	RØ = location of
2				26	g (H.O.)
3		MOV	#RØ,A	AØ	clear g (H.O.)
4		INC	RØ	18	
5		MOV	@RØ,A	AØ	clear g (L.O.)
6		MOV	RØ,#2Ø	B8	RØ = location of
7				2Ø	FLPTR
8		MOV	A,@RØ	FØ	A = FLPTR
9		MOV	RØ,A	A8	RØ = location of $f_{(m-16)}$
A		MOV	R1,#2Ø	B9	(H.O.)
B				2Ø	R1 = location of $b_{(m,16)}$
C		MOV	R7,#1Ø	BF	(H.O.)
D				1Ø	R7 = loop counter
E	GLOOP	MOVX	A,@RØ	8Ø	A = $f_{(m-k)}$ (H.O.)
F		MOV	R2,A	AA	R2 = $f_{(m-k)}$ (H.O.)
3Ø		MOVX	A,@R1	81	A = $b_{(m,k)}$ (H.O.)
1		MOV	R3,A	AB	R3 = $b_{(m,k)}$ (H.O.)
2		CALL	MLT	14	call multiply
3				5Ø	subroutine
4		MOV	R3,A	AB	R3 = result (H.O.)
5		MOV	A,RØ	F8	A = location of $f_{(m-k)}$
6		INC	A	17	(H.O.)
7		INC	A	17	update $f_{(m-k)}$ pointer
8		ANL	A,#1F	53	
9				1F	
A		MOV	R4,A	AC	save pointer
B		INC	R1	19	update $b_{(m,k)}$ pointer
C		INC	R1	19	
D		MOV	RØ,#27	B8	RØ = location of
E				27	g (L.O.)
F		MOV	A,@RØ	FØ	
4Ø		ADD	A,R2	6A	update g (L.O.)
1		MOV	@RØ,A	AØ	
2		DEC	RØ	C8	RØ = location of g (H.O.)
3		MOV	A,@RØ	FØ	update g (H.O.)
4		ADDC	A,R3	7B	
5		MOV	@RØ,A	AØ	
6		MOV	A,R4	FC	restore $f_{(m-k)}$ pointer
7		MOV	RØ,A	A8	to RØ
8		DJNZ	R7,GLOOP	EF	decrement R7 and branch to
149				2E	GLOOP if R7 ≠ Ø

Total number of cycles = 494
Time at 2.5 µsec/cycle = 1235 µsec (without multiplies)

Calculation of $e_{(m)} = f_{(m)} - g$

This routine calculates $e_{(m)}$. To prevent overflows, $e_{(m)}$ is shifted right arithmetically 4 times to form $e_{(m)}/2^4$. The value $e_{(m)}/2^4$ is stored in internal RAM. Note that the elements of the E buffer in external RAM are of the form $e_{(m-k)}/2^4$. This is to prevent overflows when the sum q is formed. Also note that in updating the weights, $V = 2^{-4}$. This factor is taken care of by using $e_{(m)}/2^4$.

Exit this routine with A = $e_{(m)}/2^4$ (L.O.) and R2 = $e_{(m)}/2^4$ (H.O.).

14A		MOV	RØ,#22	B8	RØ = location of $f_{(m)}$
B				22	(H.O.)
C		MOV	A,@RØ	FØ	A = $f_{(m)}$ (H.O.)
D		MOV	R2,A	AA	R2 = $f_{(m)}$ (H.O.)
E		INC	RØ	18	increment RØ
F		MOV	A,@RØ	FØ	A = $f_{(m)}$ (L.O.)
5Ø		MOV	RØ,#27	B8	RØ = location of
1				27	g (L.O.)
2		CPL	A	37	
3		ADD	A,@RØ	6Ø	16 bit subtract
4		CPL	A	37	to form
5		DEC	RØ	C8	$f_{(m)} - g$
6		XCH	A,R2	2A	
7		CPL	A	37	
8		ADDC	A,@RØ	7Ø	A = $e_{(m)}$ (H.O.)
9		CPL	A	37	R2 = $e_{(m)}$ (L.O.)
A		MOV	R1,#Ø2	B9	R1 = pointer to R2
B				Ø2	exchange L.O. nibbles
C		XCHD	A,@R1	31	of A and R2
D		XCH	A,R2	2A	exchange of A and R2
E		SWAP	A	47	swap nibbles of A
F		XCH	A,R2	2A	exchange A and R2
6Ø		SWAP	A	47	swap nibbles of A
1		JB3	NEG	72	test sign bit
2				67	
3		ANL	A,#ØF	53	
4				ØF	fill top nibble
5		JMP	SKIP	24	with sign bit
6				69	
7	NEG	ORL	A,#FØ	43	A = $e_{(m)}/2^4$ (H.O.)
8				FØ	R2 = $e_{(m)}/2^4$ (L.O.)
9	SKIP	MOV	RØ,#24	B8	RØ = location of $e_{(m)}$
A				24	(H.O.)
B		MOV	@RØ,A	AØ	store $e_{(m)}/2^4$ (H.O.)
C		INC	RØ	18	
D		XCH	A,R2	2A	store $e_{(m)}/2^4$ (L.O.)
16E		MOV	@RØ,A	AØ	

Total number of cycles = 35
Time at 2.5 μsec/cycle = 87.5 μsec

Update the Weights: $b_{(m,k)} = \mu b_{(m-1,k)} + ve_{(m)}\ f_{(m-k)}$

This routine updates the weights by forming the sum,

$$b_{(m,k)} = b_{(m-1,k)} - (2^{-10})\ b_{(m-1,k)} + ve_{(m)}\ f_{(m-k)}$$

The constants are $\mu = (1 - 2^{-10})$ and $v = 2^{-4}$. Enter this routine with $A = ve_{(m)}$ (L.O.) and $R2 = ve_{(m)}$ (H.O.).

16F		MOV	RØ,#2Ø	B8	RØ = location of
7Ø				2Ø	FLPTR
1		MOV	A,@RØ	FØ	A = FLPTR
2		MOV	RØ,A	A8	RØ = location of $f_{(m-16)}$
3		MOV	R1,#2Ø	B9	R1 = location of $b_{(m,16)}$
4				2Ø	
5		MOV	A,R2	FA	A = $ve_{(m)}$ (H.O.)
6		MOV	R6,A	AE	R6 = $ve_{(m)}$ (H.O.)
7		MOV	R7,#1Ø	BF	R7 = loop counter
8				1Ø	
9	BLOOP	MOV	A,R6	FE	A = $ve_{(m)}$ (H.O.)
A		MOV	R2,A	AA	R2 = $ve_{(m)}$ (H.O.)
B		MOVX	A,@RØ	8Ø	A = $f_{(m-k)}$ (H.O.)
C		MOV	R3,A	AB	R3 = $f_{(m-k)}$ (H.O.)
D		CALL	MLT	14	call multiply
E				5Ø	subroutine
F		MOV	R3,A	AB	R3 = $ve_{(m)}\ f_{(m-k)}$ (H.O.)
8Ø		INC	R1	19	R1 = location of $b_{(m,k)}$ (L.O.)
1		MOVX	A,@R1	81	A = $b_{(m,k)}$ (L.O.)
2		ADD	A,R2	6A	A = $ve_{(m)}\ f_{(m-k)}$
3		MOV	R2,A	AA	+ $b_{(m,k)}$ (L.O.)
4		DEC	R1	C9	
5		MOVX	A,@R1	81	A = $b_{(m,k)}$ (H.O.)
6		MOV	R4,A	AC	save $b_{(m,k)}$ (H.O.)
7		ADDC	A,R3	7B	in R4
8		MOV	R3,A	AB	R3 = $ve_{(m)} f_{(m-k)} + b_{(m,k)}$
9		MOV	A,R4	FC	(H.O.)
A		RR	A	77	A = $b_{(m,k)}$ (H.O.)
18B		RR	A	77	rotate right twice

Update the Weights: (cont.)

```
18C             JZ      BCONT       C6          if A = Ø, go to
  D                                 9F            BCONT
  E             CPL     A           37          complement A
  F             JB5     BSKIP       B2          test sign bit
 9Ø                                 98
  1             ANL     A,#3F       53
  2                                 3F
  3             INC     A           17          form 2's comp.
  4             ADD     A,R2        6A            of (2^-10) b(m,k)
  5             MOV     R2,A        AA            and add to
  6             JMP     BCONT       14            previous sum
  7                                 9F            to form the
  8   BSKIP     INC     A           17            updated b(m,k)
  9             ADD     A,R2        6A
  A             MOV     R2,A        AA
  B             MOV     A,R3        FB
  C             ADDC    A,#FF       13          R3 = new b(m,k) (H.O.)
  D                                 FF          R2 = new b(m,k) (L.O.)
  E             MOV     R3,A        AB
  F   BCONT     MOV     A,R3        FB          store the new b(m,k)
 AØ             MOVX    @R1,A       91            in the
  1             INC     R1          19            B buffer
  2             MOV     A,R2        FA
  3             MOVX    @R1,A       91
  4             INC     R1          19          update b(m,k) pointer
  5             MOV     A,RØ        F8
  6             INC     A           17          update f(m-k) pointer
  7             INC     A           17
  8             ANL     A,#1F       53
  9                                 1F
  A             MOV     RØ,A        A8
  B             DJNZ    R7,BLOOP    EF          decrement counter and
1AC                                 79            repeat if ≠ Ø
```

Total number of cycles = 810
Time at 2.5 μsec/cycle = 2025 μsec (without multiplies)

Calculation of $q = q_{(old)} + e_{(m)} - e_{(m-16)}$

This routine calculates the new value of q which is the sum of the elements of the E buffer. To do this, the oldest value of e, $e_{(m-16)}$, is subtracted from the old q and the newest value of e, $e_{(m)}$, is added to form the new value of q. Since the values in the E buffer are stored as $e/2^4$, the sum formed is actually $q/2^4$. This value is squared and scaled by 2^4 to form $q^2/2^4$ which is sent to the D/A converter for testing.

1AD	MOV	R1,#21	B9	R1 = location of
E			21	ELPTR
F	MOV	A,@R1	F1	R1 = location of $e_{(m-16)}$
BØ	MOV	R1,A	A9	(H.O.)
1	MOVX	A,@R1	81	A = $e_{(m-16)}$ (H.O.)
2	MOV	R3,A	AB	R3 = $e_{(m-16)}$ (H.O.)
3	INC	R1	19	
4	MOVX	A,@R1	81	A = $e_{(m-16)}$ (L.O.)
5	MOV	R4,A	AC	R4 = $e_{(m-16)}$ (L.O.)
6	MOV	RØ,#24	B8	RØ = location of $e_{(m)}$
7			24	(H.O.)
8	MOV	A,@RØ	FØ	A = $e_{(m)}$ (H.O.)
9	MOV	R2,A	AA	R2 = $e_{(m)}$ (H.O.)
A	INC	RØ	18	
B	MOV	A,@RØ	FØ	A = $e_{(m)}$ (L.O.)
C	CPL	A	37	
D	ADD	A,R4	6C	16 bit subtract to form
E	CPL	A	37	$e_{(m)} - e_{(m-16)}$
F	XCH	A,R2	2A	
CØ	CPL	A	37	
1	ADDC	A,R3	7B	R2 = $e_{(m)} - e_{(m-16)}$
2	CPL	A	37	(H.O.)
3	XCH	A,R2	2A	A = $e_{(m)} - e_{(m-16)}$ (L.O.)
4	MOV	RØ,#29	B8	RØ = location of
5			29	q (L.O.)
6	ADD	A,@RØ	6Ø	update q (L.O.)
7	MOV	@RØ,A	AØ	and store
8	XCH	A,R2	2A	
9	DEC	RØ	C8	update q (H.O.)
A	ADDC	A,@RØ	7Ø	and store
B	MOV	@RØ,A	AØ	
C	MOV	R2,A	AA	R2 = q (H.O.)
D	MOV	R3,A	AB	R3 = q (H.O.)
E	CALL	MLT	14	form q^2
F			5Ø	(actually $q^2/2^8$)
1DØ	SWAP	A	47	

Calculation of q (cont.)

1D1	XCH	A,R2	2A	shift right
2	SWAP	A	47	4 times
3	MOV	RØ,#Ø2	B8	arithmetically
4			Ø2	
5	XCHD	A,@RØ	3Ø	
6	ANL	A,#FØ	53	
7			FØ	$A = q^2/2^4$ (H.O.)
8	XCH	A,R2	2A	$R2 = q^2/2^4$ (L.O.)
9	ORL	P1,#Ø8	89	output control
A			Ø8	bit to select
B	OUTL	P2,A	3A	D/A
C	ANL	P1,#F7	99	output to D/A
1DD			F7	output control bit to select A/D

Exit this routine with $A = q^2/2^4$ (H.O.)
$R2 = q^2/2^4$ (L.O.)

Since almost all jump commands are to locations within the same 256 byte page of memory, we now go to page 3 for the ATD routine.

1DE	JMP	ATD	44	jump to location
F			ØØ	2ØØ

Total number of cycles = 54
Time at 2.5 μsec/cycle = 135 μsec

Adaptive Threshold Detection

This routine maintains the T buffer which contains 192 sixteen bit words. The routine forms the sum of the oldest 64 elements in the buffer. A constant σ^2 is formed as follows:

$$\sigma^2 = K(SUM) + \theta$$

where SUM = the sum of the 64 oldest buffer elements. σ^2 is compared with $q^2/2^4$ to determine ALARM status.

Enter this routine with A = $q^2/2^4$ (H.O.)
R2 = $q^2/2^4$ (L.O.)

2ØØ		MOV	RØ,#2A	B8	RØ = location of $q^2/2^4$
1				2A	(H.O.)
2		MOV	@RØ,A	AØ	store $q^2/2^4$ (H.O.)
3		INC	RØ	18	
4		XCH	A,R2	2A	A = $q^2/2^4$ (L.O.)
5		MOV	@RØ,A	AØ	store $q^2/2^4$ (L.O.)
6		RLC	A	F7	
7		XCH	A,R2	2A	
8		RCL	A	F7	form $t_{(m)} = (q^2/2^4)\ 2^{-7}$
9		XCH	A,R2	2A	
A		CLR	A	27	
B		JNC	ACONT	E6	
C				ØE	
D		CPL	A	37	R3 = $t_{(m)}$ (H.O.)
E	ACONT	MOV	R3,A	AB	R2 = $t_{(m)}$ (L.O.)
F		INC	RØ	18	RØ = location of TLPTR
1Ø		MOV	A,@RØ	FØ	A = TLPTR
1		MOV	RØ,A	A8	RØ = location of $t_{(m-192)}$
2		JFO	SWTCH	B6	if FØ=1, select
3				16	external RAM
4		JMP	ACONT2	44	page 1
5				18	else, continue
6	SWTCH	ORL	P1,#Ø1	89	
7				Ø1	
8	ACONT2	MOVX	A,@RØ	8Ø	A = $t_{(m-192)}$ (H.O.)
9		MOV	R4,A	AC	R4 = $t_{(m-192)}$ (H.O.)
A		INC	RØ	18	
B		MOVX	A,@RØ	8Ø	A = $t_{(m-192)}$ (L.O.)
C		MOV	R5,A	AD	R5 = $t_{(m-192)}$ (L.O.)
D		MOV	A,R2	FA	A = $t_{(m)}$ (L.O.)
21E		MOVX	@RØ,A	9Ø	store $t_{(m)}$ (L.O.)

ATD (cont.)

```
21F             DEC     RØ          C8
 2Ø             MOV     A,R3        FB      A = t(m)  (H.O.)
  1             MOVX    @RØ,A       9Ø      store t(m)  (H.O.)
  2             INC     RØ          18
  3             INC     RØ          18
  4             MOV     A,RØ        F8      update T buffer
  5             JFO     ONE         B6        by storing t(m) at old
  6                                 2B        location of
  7             JZ      SWTCH2      C6        of t(m-192)
  8                                 34
  9             JMP     ACONT3      44      update TLPTR
  A                                 35
  B   ONE       ADD     A,#2Ø       Ø3      update FØ
  C                                 2Ø        to point to
  D             JZ      RESTART     C6        correct page
  E                                 32        in external
  F             MOV     A,RØ        F8        RAM
 3Ø             JMP     ACONT3      44
  1                                 35
  2   RESTART   MOV     A,#6Ø       23
  3                                 6Ø
  4   SWTCH2    CPL     FØ          95
  5   ACONT3    MOV     RØ,#2C      B8      store updated
  6                                 2C        TLPTR
  7             MOV     @RØ,A       AØ
  8             MOV     R1,#2E      B9      R1 = location of
  9                                 2E        SUM  (H.O.)
  A             MOV     A,@R1       F1      A = SUM  (H.O.)
  B             MOV     R2,A        AA      R2 = SUM (H.O.)
  C             INC     R1          19
  D             MOV     A,@R1       F1      A = SUM  (L.O.)
  E             CPL     A           37      16 bit subtract
  F             ADD     A,R5        6D        to form
 4Ø             CPL     A           37        SUM - t(m-192)
  1             XCH     A,R2        2A
  2             CPL     A           37
  3             ADDC    A,R4        7C      R2 = SUM - t(m-192)  (L.O.)
  4             CPL     A           37      R3 = SUM - t(m-192)  (H.O.)
  5             MOV     R3,A        AB
  6             INC     RØ          18      RØ = location of
  7             MOV     A,@RØ       FØ        TLPTR + 64
  8             MOV     RØ,A        A8      RØ = location of
  9             JF1     SETB        76        t(m-128)
  A                                 4F
  B             ANL     P1,#FE      99      select proper
  C                                 FE        page of
  D             JMP     ACONT4      44        external RAM
24E                                 51
```

ATD (cont.)

24F	SETB	ORL	P1,#Ø1	89	
5Ø				Ø1	
1	ACONT4	MOVX	A,@RØ	8Ø	A = $t_{(m-128)}$ (H.O.)
2		MOV	R4,A	AC	R4 = $t_{(m-128)}$ (H.O.)
3		INC	RØ	18	
4		MOX	A,@RØ	8Ø	A = $t_{(m-128)}$ (L.O.)
5		ADD	A,R2	6A	form SUM - $t_{(m-192)}$
6		XCH	A,R4	2C	+ $t_{(m-128)}$
7		ADDC	A,R3	7B	R3 = new SUM (H.O.)
8		MOV	R3,A	AB	R4 = new SUM (L.O.)
9		INC	RØ	18	
A		MOV	A,RØ	F8	
B		JF1	ONE2	76	
C				61	update TLPTR+64
D		JZ	SWTCH3	C6	
E				6A	update F1 to
F		JMP	ACONT5	44	point to the
6Ø				6B	correct page
1	ONE2	ADD	A,#2Ø	Ø3	in external
2				2Ø	RAM
3		JZ	RST2	C6	
4				68	
5		MOV	A,RØ	F8	
6		JMP	ACONT5	44	
7				6B	
8	RST2	MOV	A,#6Ø	23	
9				6Ø	
A	SWTCH3	CPL	F1	B5	store updated
B	ACONT5	MOV	RØ,#2D	B8	TLPTR+64
C				2D	
D		MOV	@RØ,A	AØ	
E		MOV	R2,#k	BA	R2 = K
F					
7Ø		CALL	MLT	$\overline{14}$	form K(SUM)
1				50	
2		XCH	A,R2	2A	
3		ADD	A,#θ(L.O.)	Ø3	form K(SUM) + θ = σ^2
4					
5		XCH	A,R2	$\overline{2A}$	
6		ADDC	A,#θ(H.O.)	13	
7					A = K(SUM) + θ (L.O.)
8		XCH	A,R2	$\overline{2A}$	R2 = K(SUM) + θ (H.O.)
9		MOV	R1,#2B	B9	R1 = location of
A				2B	$q^2/2^4$ (L.O.)
B		CPL	A	37	
C		ADD	A,@R1	61	16 bit subtract to form
D		CPL	A	37	$\sigma^2 - q^2/2^4$
27E		XCH	A,R2	2A	

ATD (cont.)

```
27F             DEC     R1          C9
 8Ø             CPL     A           37
  1             ADDC    A,@R1       71          A = σ² - q²/2⁴  (H.O.)
  2             CPL     A           37          R2 = σ² - q²/2⁴ (L.O.)
  3             JB7     ALARM       F2          if result is < Ø,
  4                                 89            sound alarm
  5             ANL     P1,#FD      99          else, clear
  6                                 FD            alarm bit
  7             JMP     DONE        44
  8                                 8B
  9  ALARM      ORL     P1,#Ø2      89          set alarm bit
  A                                 Ø2
  B  DONE       ANL     P1,#FE      99          select external
28C                                 FE            RAM page Ø
```

Total number of cycles = 127
Time at 2.5 μsec/cycle = 317.5 μsec

Update F and E buffers

This routine replaces $f_{(m-16)}$ with $f_{(m)}$ and $e_{(m-16)}$ with $e_{(m)}$. Pointers FLPTR and ELPTR are updated to point to the new $f_{(m-16)}$ and $e_{(m-16)}$.

```
28D             MOV     RØ,#2Ø      B8      RØ = location of
  E                                 2Ø        FLPTR
  F             MOV     A,@RØ       FØ
 9Ø             MOV     RØ,A        A8      RØ = location of f(m-16)
  1             MOV     R1,#22      B9      R1 = location of f(m)
  2                                 22
  3             MOV     A,@R1       F1      A = f(m)  (H.O.)
  4             MOVX    @RØ,A       9Ø      store in buffer
  5             INC     RØ          18
  6             INC     R1          19
  7             MOV     A,@R1       F1      A = f(m)  (L.O.)
  8             MOVX    @RØ,A       9Ø      store in buffer
  9             MOV     A,RØ        F8
  A             INC     A           17      update FLPTR
  B             ANL     A,#1F       53
  C                                 1F
  D             MOV     RØ,#2Ø      B8      store new
  E                                 2Ø        FLPTR
  F             MOV     @RØ,A       AØ
 AØ             INC     RØ          18      RØ = location of ELPTR
  1             MOV     A,@RØ       FØ
  2             MOV     RØ,A        A8      RØ = location of e(m-16)
  3             INC     R1          19      R1 = location of e(m)
  4             MOV     A,@R1       F1      A = e(m)  (H.O.)
  5             MOVX    @RØ,A       9Ø      store in buffer
  6             INC     RØ          18
  7             INC     R1          19
  8             MOV     A,@R1       F1      A = e(m)  (L.O.)
  9             MOVX    @RØ,A       9Ø      store in buffer
  A             MOV     A,RØ        F8
  B             INC     A           17
  C             ANL     A,#5F       53      update ELPTR
  D                                 5F
  E             MOV     RØ,#21      B8      store new
  F                                 21        ELPTR
 BØ             MOV     @RØ,A       AØ
  1   HERE      JMP     HERE        44      wait for timer
2B2                                 B1        interrupt
```

Total number of cycles = 41
Time at 2.5 μsec/cycle = 102.5 μsec

The timer, which was preset in the Input from A/D routine, will count continuously until it is stopped. When the count changes from FF to ØØ, a timer interrupt will set the PC = ØØ7. From here, the program branches back to the Input from A/D routine.

A reset will set the PC = ØØØ. From here, the program branches to the Initialization routine.

ØØØ	EN	TI	25	enable timer
1	JMP	INIT	Ø4	interrupt
2			D8	jump to initialize
ØØ7	JMP	AD	24	jump to input
8			ØØ	from A/D

APPENDIX 3

ATMAC WIDROW ALGORITHM PROGRAM

Memory: Assume a data memory structure as is shown below.

Pointers		Contents	Address
B1		$b_{m,1}$	B1 = ØØØØ H
		$b_{m,2}$	
		⋮	
		$b_{m,16}$	B16 = ØØØF H
FØ		f_m	FØ ⋮
F1		f_{m-1}	F1
		f_{m-2}	
		⋮	
		f_{m-16}	F16
E1		$e^2_{m-1}/16$	ETOP
		$e^2_{m-2}/16$	
		⋮	
E16	Circu-lating Buffer	$e^2_{m-16}/16$	ETOP+16
		$e^2_{m-17}/64$	
		⋮	
EØØ		$e^2_{m-8Ø}/64$	EBOTTOM
		Q_a	QA
		Q_b	QB

Note: E1, E16, and EØØ are maintained as pointers to a circulating buffer. The F buffer is updated by a block move. Thus the pointers to the F buffer as well as to the B buffer are stationary.

Registers:

General Registers

GRØ - general purpose

GR1 - general purpose

GR2 - general purpose

GR3 - general purpose

GR4 - g

GR5 - e_m

GR6 - general purpose

GR7 - general purpose

Indirect Address Registers

IARØ - general purpose

IAR1 - B1 address

IAR2 - FØ address

IAR3 - F1 address

IAR4 - E1 address

IAR5 - E16 address

IAR6 - E8Ø address

IAR7 - loop counter and initialization

ATMAC Widrow Program

Initialization:

This routine clears all buffers in data memory and sets up the buffer pointers.

	Instruction	Comments
	ILDI Ø; B1	IARØ ← location of the first element in data memory to be cleared.
	ZR Ø, ; Ø, ; MNOP	Clear GRØ
	ILDI 7; = 114	Set loop counter = 114 D and identify the next instruction as the first instruction in the loop.
LOOP1	NOP 1, ; ST Ø,Ø; RNBP1	Store the contents of GRØ at the address contained in IARØ. IARØ ← IARØ + 1. Decrement loop counter and branch to LOOP1 if > 0, else continue.
	ILDI 1; B1	IAR1 ← B1
	ILDI 2; FØ	IAR2 ← FØ
	ILDI 3; F1	IAR3 ← F1
	ILDI 4: ETOP	IAR4 ← E1
	ILDI 5; ETOP + 16	IAR5 ← E16
	ILDI 6: EBOTTOM	IAR6 ← E8Ø

Input from A/D:

This routine obtains the new f_m sample and stores it at FØ. It also starts the next conversion. The routine begins at program location START.

Compute $g = \sum_{k=1}^{16} b_{m,k} f_{m-k}$

Instruction	Comments
MDCH1 Ø,2; LD 1,1; AUTP1	Clear SFU accumulator 1 and set SFU to multiply and accumulate mode. GR1 ← $b_{m,1}$ IAR1 ← IAR1 + 1
NOP Ø, ; LD 3,3; AUTP1	GR3 ← f_{m-1} IAR3 ← IAR3 + 1
ILDI 7; = 15	Set loop counter = 15 D and identify next instruction as beginning of loop.
LOOP2 FMUL1 1,3; LD 1,1: AUTP1	accumulator = accumulator + $(b_{m,k})(f_{m-k})$ GR1 ← $b_{m,k}$ IAR1 ← IAR1 + 1
NOP , ; LD 3,3; RNBP1	GR3 ← f_{m-k} IAR3 ← IAR3 + 1 decrement loop counter and branch to LOOP 2 if > Ø, else continue
RDWD1 Ø,4; 4, ; MNOP	GR4 ← g (rounded)
ILDI 1; B1	restore pointer
ILDI 3; F1	restore pointer

Compute $e_m = f_m - g$

Instruction	Comments
NOP Ø, ; LD 5,2; MNOP	GR5 ← f_m
SUB 5,4; 5, ; MNOP	GR5 ← $e_m = f_m - g$

Update the Weights: $b_{m+1,k} = ub_{m,k} + ve_m f_{m-k}$

	Instructions	Comments
	LDI 1; V	GR1 ← V
	MDCH1 Ø,1; Ø, ; MNOP	Clear SFU accumulator and set to non-accumulate mode.
	FMUL1 1,5; 1, ; MNOP	acc. ← ve_m
	RDWD1 Ø,4; 3, ; MNOP	GR3 ← ve_m (rounded)
	LDI 1; u	GR1 ← u
	NOP 1, ; LD Ø,3; AUTP1	GRØ ← f_{m-k} IAR3 ← IAR3 + 1
	ILDI 7; = 15	Set loop counter = 15 D and identify next instruction as the first in loop.
LOOP3	FMUL1 Ø,3; LD 2,1; MNOP	acc. ← $ve_m f_{m-k}$ GR2 ← $b_{m,k}$
	RDWD1 Ø,4; 6, ; MNOP	GR6 ← $ve_m f_{m-k}$ (rounded)
	FMUL1 1,2; 1, ; MNOP	acc. ← $ub_{m,k}$
	RDWD1 Ø,4; 7, ; MNOP	GR7 ← $ub_{m,k}$ (rounded)
	ADD 6,7; LD Ø,3; AUTP1	GR6 ← $b_{m+1,k}$ GRØ ← next f_{m-k} IAR3 ← IAR3 + 1
	NOP Ø, ; ST 6,1; AUTP1	store $b_{m+1,k}$ at B buffer IAR1 ← IAR1 + 1
	NOP Ø, ; Ø, ; RNBP1	decrement loop counter. If > 0, go to LOOP3, else continue.
	ILDI 1; B1	restore pointer
	ILDI 3; F1	restore pointer

Adaptive Threshold Detection

This routine does the following.

1. Form $e_m^2/16$
2. Substract $e_{m-16}^2/16$ from Q_a
3. Add $e_m^2/16$ to Q_a to form the new Q_a
4. Store the result
5. Form $e_{m-16}^2/64$
6. Subtract $e_{m-80}^2/64$ from Q_b
7. Add $e_{m-16}^2/64$ to Q_b to form the new Q_b
8. Store the result
9. Form $Q_a - Q_b - \theta$
10. Test for alarm, i.e., $Q_a - Q_b - \theta > 0$

Instruction	Comments
CPGR 1,5; 1, ; MNOP	GR1 ← GR5 = e_m
FMUL1 1,5; 1, ; MNOP	acc. ← e_m^2
RDWD1 Ø,4; 1, ; MNOP	GR1 ← e_m^2 (rounded)
SHRA 1,1; 1, ; MNOP	divide by 2
SHRA 1,1; 1, ; MNOP	divide by 2
SHRA 1,1; 1, ; MNOP	divide by 2
SHRA 1,1; 1, ; MNOP	divide by 2 GR1 ← $e_m^2/16$
ILDI Ø; QA	IARØ ← QA
NOP Ø, ; LD 2,Ø; AUTP1	GR2 ← Q_a IARØ ← IARØ + 1 = QB
NOP Ø, ; LD 3,5; MNOP	GR3 ← $e_{m-16}^2/16$

Instruction	Comments
SUB 2,3; 2, ; MNOP	GR2 ← $Q_a - e^2_{m-16}/16$
ADD 2,1; 2, ; MNOP	GR2 ← new Q_a
SHRA 3,3; LD 6,Ø; AUTM1	GR3 ← $e^2_{m-16}/32$ GR6 ← Q_b IARØ ← QA
SHRA 3,3; LD Ø,6; MNOP	GR3 ← $e^2_{m-16}/64$ GRØ ← $e^2_{m-80}/64$
SUB 6,Ø; ST 2,Ø; AUTP1	GR6 ← $Q_b - e^2_{m-80}/64$ store new Q_a at QA IARØ ← QB
ADD 6,3; 6, ; MNOP	GR6 ← new Q_b
SUB 2,6; ST 6,Ø; AUTM1	GR2 ← $Q_a - Q_b$ store Q_b at QB IARØ ← QA
SUBI 2; = θ	GR2 ← $Q_a - Q_b - \theta$
BOT 3; NALARM	if result $\leq$ 0, branch to NALARM
BOT Ø; ALARM	if result > 0, branch to ALARM

Note: NALARM is a routine to output a "Ø" to signal no alarm.
ALARM is a routine to output a "1" to signal an alarm.

Control from both of these routines, which are not included here, would be to the routine which updates the buffers.

Note: Leave this routine with,

GR1 = $e^2_m/16$

GR3 = $e^2_{m-16}/64$

Update the Buffers:

This routine does the following.

1. Block move of the F buffer.
2. Update pointers E1, E16, and E8Ø.

	Instruction	Comments
	ILDI Ø; F16-1	IARØ ← address of f_{m-15}
	ILDI 1; F16	IAR1 ← F16
	ILDI 7; = 15	set loop counter = 15 D and identify the next instruction as the first in the loop.
LOOP4	NOP 2, ; LD Ø,Ø; AUTM1	GRØ ← f_{m-k} IARØ ← IARØ - 1
	NOP 2, ; ST Ø,1; AUTM1	store f_{m-k} at location of f_{m-k-1} IAR1 ← IAR1 - 1
	NOP Ø, ; Ø, ; RNBP1	decrement loop counter. If > 0, go to LOOP4, else continue.
	ILDI 1; B1	restore pointer
	NOP Ø, ; ST 3,5; MNOP	store $e^2_{m-16}/64$ at E16
	NOP Ø, ; ST 1,6; MNOP	store $e^2_m/16$ at E8Ø
	CPIG 4, ; 1, ; MNOP	GR1 ← IAR4
	DECR 1,1; 1, ; MNOP	GR1 ← GR1 - 1
	SUBI 1; F16	Test if pointer has gone beyond the end of the E buffer.
	BOT 1; FIX1	If so, go to FIX1, else continue.
	CPGI Ø,1; Ø,4; MNOP	IAR4 ← GR1
	BOT Ø; NEXT1	go to NEXT1
FIX1	ILDI 4; EBOTTOM	IAR4 ← EBOTTOM
NEXT1	CPIG 5, ; 1, ; MNOP	GR1 ← IAR5
	DEC 1,1; 1, ; MNOP	GR1 ← GR1 - 1

	Instruction	Comments
	SUBI 1; F16	Test if pointer has gone beyond the end of the E buffer.
	BOT 1; FIX2	If so, go to FIX2, else continue.
	CPGI Ø,1; Ø,5; MNOP	IAR5 ← GR1
	BOT Ø; NEXT2	go to NEXT2
FIX2	ILDI 5; EBOTTOM	IAR5 ← EBOTTOM
NEXT2	CPIG 6, ; 1, ; MNOP	GR1 ← IAR6
	DEC 1,1; 1, ; MNOP	GR1 ← GR1 - 1
	SUBI 1; F16	Test if pointer has gone beyond the end of the E buffer.
	BOT 1; FIX3	If so, go to FIX3, else continue.
	CPGI Ø,1; Ø,6; MNOP	IAR6 ← GR1
	BOT Ø; NEXT3	go to NEXT3
FIX3	ILDI 6; EBOTTOM	IAR6 ← EBOTTOM
NEXT3	BOT Ø; START	go to START

AN EVALUATION OF VARIOUS MICROPROCESSOR IMPLEMENTATIONS OF AN ADAPTIVE DIGITAL PREDICTOR FOR INTRUSION DETECTION

by

DONOVAN J. NICKEL

B. S., Kansas State University, 1978

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Electrical Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
1979

ABSTRACT

Recent research by Ahmed in the area of digital signal processing has led to an interesting new application for microprocessors. The signal processing algorithm which was developed for Sandia Laboratories involves the use of an Adaptive Digital Predictor and other conventional digital filters to detect the presence of noise produced by an intruder in a random ambient noise environment.

This report describes the implementation of the algorithm using three different microprocessors. First, a Zilog Z80 implementation is presented and includes both the software development and a minimal hardware configuration designed for the algorithm. Secondly, a possible implementation using Intel's 8048 (8748) microprocessor is discussed, and, finally, a similar study is presented on a possible implementation using the RCA ATMAC.

Included in this report is a review of the operation of the algorithm and a comparison of the various methods of implementation. Some test results are also presented.