

AN IMPLEMENTATION AND EVALUATION OF A
MINIMAL-COMPONENT MINIMAL-POWER MICROCOMPUTER
SYSTEM USING ROCKWELL'S AAMP

by

Eric Nelson

B. S. , Kansas State University, 1985
A. G. S., Colby Community Junior College, 1983

A MASTER'S THESIS

submitted in partial fulfillment of the
requirements for the degree

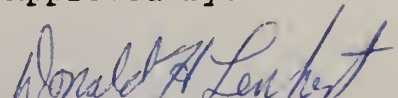
MASTER OF SCIENCE

Department of Electrical and Computer Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1987

Approved by:


Major Professor

Acknowledgements

This work was sponsored and funded by the Base and Installation Security Systems Program Office, Electronics System Division of the Air Force Systems Command, Hanscom Air Force Base, MA 01731 through the Systems Engineering Division, Organization 5238, Sandia National Laboratories, Kirtland Air Force Base, Albuquerque, New Mexico.

I would like to thank the professors who served on my committee, Dr.S. Dyer, and Dr. J. G. Thompson and especially, Dr. D. H. Lenhert who provided regular consultation on the project. I would also like to thank Ken Albin who provided information and insight into the workings of the AAMP.

Finally, thanks is extended to Gail Navinsky who did my laundry during a busy semester and provided brownies to my committee.

CONTENTS

	<u>page</u>
List of Figures	vi
List of Tables	vii
1. Introduction	1
2. Description of AAMP	3
2.1 Hardware	3
2.2 Packaging	4
2.3 Architecture	5
2.4 Instruction Set	9
3. Minimal Component AAMP Board	11
3.1 Objectives of Design	11
3.2 Memory	15
3.3 Address Decoding	16
3.4 I/O	18
3.5 Inverter/Oscillator	21
4. System Power Consumption	27
4.1 AAMP	29
4.2 Oscillator	29
4.3 Support	31
4.4 Wait State Operation	34
5. Software Evaluation	39
5.1 Verification of output	39
5.2 Speed optimization	40

5.3 Accuracy of estimates	42
5.4 Timing Estimates	42
6. Conclusions	45
References	47
APPENDIX A - Pin Assignment for AAMP	48
APPENDIX B - Instruction Set Timing Estimates	52
APPENDIX C - Printed Circuit Board Layout	57
APPENDIX D - Software Listings	67
D.1 Fractional 16-bit Precision -Loop Coding	68
D.2 Fractional 16-bit Precision -Inline Coding	78
D.3 Standard Precision Floating Point -loop Coding	83
D.4 Standard Precision Floating Point -Inline Coding	91
D.5 Extended Precision Floating Point -Loop Coding	94
D.6 Extended Precision Floating Point -Line Coding	103

LIST OF FIGURES

1. Addressing Using CENV and DENV Pointers	7
2. Block Diagram of Microcomputer System	13
3. Circuit Diagram of Microcomputer System	14
4. Memory Map	16
5. System Timing	17
6. Circuit for Software Confirmation	20
7. Transfer Function of Inverters	22
8. Phase Shift in Parallel Resonant Oscillator	23
9. Circuit Compensating for Gate Delay	24
10. Power Consumption of System	28
11. External Oscillator Circuit Used for System	30
12. Configuration for On-chip AAMP Oscillator	31
13. Power Consumption of Support Components	32
14. Effective Input Circuits of D Flip-flops	33
15. Wait State Circuitry Maintaining Two Alarm Outputs	34
16. Wait State Circuitry Converting Alarm Output to Controller	36
17. Power Consumption in High-impedance Mode	38
18. Half-Handshake Controller for GPIO Interface	40
A.1 Pin Assignment for 68 pin PGA AAMP	48
C.1 Assembly Drawing	58
C.2 Layer 1 -Component Side	59
C.3 Layer 2 -Ground Plane	60

C.4 Layer 3	61
C.5 Layer 4	62
C.5 Layer 5 -Ground Plane	63
C.5 Layer 6 -Circuit Side	64
C.6 Keepout Areas For Power and Ground Planes	65
C.7 Silkscreened Text on Component Side	66

LIST OF TABLES

1. Executive Entry Table	6
2. Opcode to Stack Depth Mapping	10
3. Algorithms used for Software Evaluation	43
4. Timing Estimates For Digital Filtering Sub-programs	44
A.1 AAMP Pin Assignments by Functions	49
B.1 Timing Estimates for Instruction Set	53

INTRODUCTION

In the past several years, Kansas State University has designed and developed several ultra-low power Analog to Digital Converters. These converters typically consumed far less power than the signal processing sections of integrated systems and provided better resolution than 16-bit single precision processors could maintain. The Advanced Architecture MicroProcessor (AAMP), Rockwell's CMOS floating point processor provides solutions to both of these problems. The following thesis describes an AAMP-based microcomputer system with a measured power consumption of under 25 milliwatts at the suggested operating frequency of 2.6 MHz. Use of the AAMP also supplies the system with a capacity for 32- and 48-bit floating point arithmetic, the dynamic range and extended precision of which should allow the processor to maintain precision from the A/D through several stages of multiplications without truncation errors.

Previous work at Kansas State University involving the AAMP is presented in a thesis by K. L. Albin¹ documenting the architecture and coding some typical signal processing algorithms, and a thesis by G. S. Mauersberger² detailing a large-scale AAMP-based microcomputer design which he built and tested. Mike Gaches also designed a board using

the AAMP which could be a direct replacement for 8086 boards presently in use. The system described in this thesis was also originally designed by Mike Gaches with several changes being made since that time.

This thesis begins with a brief description of the AAMP detailing hardware and architecture features which were important in the design and evaluation of this system. Details of the design are discussed with separate sections provided for each sub-system. Power consumption figures for the system and most components of the system are shown and discussed. And, finally, listing of several typical signal processing code segments are presented with estimated time of execution for each given.

2. Description of AAMP

The following section provides an overview of the hardware and software features of the AAMP with emphasis on those aspects utilized in the present design. Much more thorough descriptions of these are given in the documents provided by Rockwell³ and in the previous theses dealing with the AAMP.^{1 2}

2.1 Hardware

The AAMP is a 16-bit floating-point microprocessor built in either 2-micron CMOS or 2-micron CMOS/SOS. Data transfer is done through a 24-bit address, 16-bit data, non-multiplexed bus with the AAMP supporting either synchronous or asynchronous data transfer. When operated in synchronous mode, as in this design, the AAMP provides selection of bus timing parameters. By strapping the two setup select pins (S_0 , S_1) high or low, the user can select the total time required for a complete bus read cycle which may allow the use of slower memory devices while still maintaining processing speed.

The AAMP also has provisions which allow straightforward bus arbitration schemes to be used in large, shared bus, multiprocessor systems. When deselected by \overline{OE} being held high, all bus outputs from the AAMP are tri-stated

allowing other users complete access to a system bus. The Bus Request ($\overline{\text{BR}}$) pin is, however, still active and may be used to poll a master controller device.

Although the on-chip oscillator is rated for use from 4-20 MHz, use of an external oscillator can extend the bandwidth DC to 30 MHz. The on-chip oscillator was, however, found to operate at frequencies down to 1 MHz although power consumption at the low end was relatively high compared to the power consumption for an external oscillator.

2.2 Packaging

The AAMP is presently being packaged in two different forms, a chip carrier package, and pin grid array (PGA) package. The most common package, and the one used in this design is the 1.1" by 1.1" 68-pin PGA package. Early versions of the AAMP had a different pin assignment than the present version. Appendix A shows the pin assignment for an early CMOS/SOS version, the Bulk CMOS version used in the prototype, and for the new, top cavity device. Current PGA versions also have an alignment pin at the C-3 location.

Bulk CMOS AAMPs are currently being produced by Rockwell's Semiconductor Products Division (SPD) and by American Microsystems Incorporated (AMI). For radiation hardened applications, CMOS/SOS versions of the AAMP are

still being produced by Rockwell's Microelectronics Research and Development Center (MRDC).

Future plans by Rockwell for changes in the AAMP include a shrink to 1.2 microns which should allow a 50 MHz operating frequency. Rockwell also plans to begin in Dec. 1986, designing a new AAMP with a 48-bit ALU , a 6-deep stack cache, a block move instruction, and improved microcoding of the multiply and divide operations.⁴

2.3 Architecture

The stack architecture of the AAMP was designed to ease translation from the intermediate level output of high level language compilers to assembly language. The AAMP utilizes a stack architecture in which all logic and arithmetic operations are performed on the top members of the stack with the result being returned back to the top of the stack. To lessen the need for constant bus accesses to obtain each operand of an operation and to return the result, the AAMP maintains an on-chip cache of the top four elements of the stack.

Programming can be run in either Executive or User modes with Executive mode generally being used for initialization and control of program transfers to the various User programs. The first nine words of memory contain entries, called the Executive Entry Table which set the values for the various environment pointers and define

vectors for start-up, bus error, exception, trap, and interrupt routines. Upon invoking a program in executive mode, the first four of these entries are read while the rest are read only as needed. A similar table, called the User Processor State Descriptor (PSD) Table is used similarly upon a switch to User mode. All software evaluation for this system was done with programs written in the Executive mode. Table 1 shows the items listed in the Executive Entry Table.

Table 1. Executive Entry Table

\$0000	Continuation Status Pointer
\$0001	Initial Executive Stack limit
\$0002	Initial Executive Top of Stack
\$0003	Initial Executive Procedure Identifier
\$0004	Bus Error Procedure Identifier
\$0005	NMI Procedure Identifier
\$0006	INT Procedure Identifier
\$0007	Trap Procedure Identifier
\$0008	Exception Procedure Identifier

To aid in the access of a 24-address bit bus with 16-bit words, the AAMP provides environment pointers. Effective addresses are formed by using these pointers as the top eight or the top nine bits of the address and a normal 16-bit word for the lower address bits. (Figure 1) On data accesses, the Data Environment pointer (DENV) provides the upper eight bits for all but the Universal addressing mode in which all 24 bits are taken from the top

of the stack.

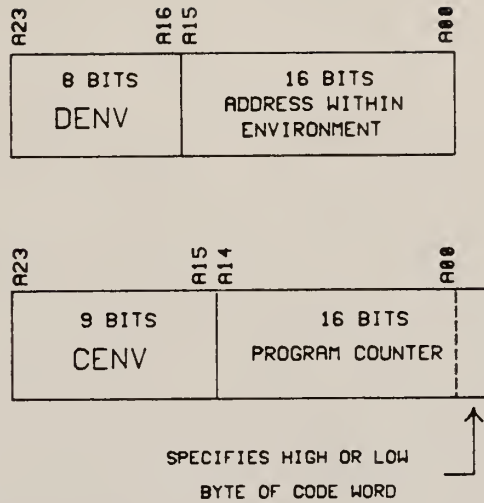


Figure 1. Addressing Using DENV and CENV Pointers

Code addressing requires 25 bits as each 16-bit code word often contains two separate single-byte opcodes. The Code Environment pointer (CENV) provides the upper nine bits of address. The 16-bit program counter provides the lower bits, the lowest of which specifies high or low byte within a 16-bit word of code. CENV and DENV are automatically set to zero in the Executive mode.

Also maintained within the processor is a Local Environment (LENV) pointer, a 16-bit word which in combination with DENV, points to the top of the Local Environment, a portion of memory at the bottom of the active stack frame set aside for quick access. Address calculation within the Local Environment is very efficient

requiring a minimum of bus accesses since only the offset into the environment must be specified. When using Local environment addressing, this offset is specified by the lower nibble within the opcode itself. While using Local Extended addressing, the offset is specified by a single byte following the opcode.

Another powerful feature of the AAMP's architecture is its dynamic memory allocation and parameter passage. Prior to a CALL to a new procedure, the calling procedure may copy into its stack a list of parameters to be passed to the called routine. Upon invocation of the CALL instruction, a user-specified number of arguments on the top of the stack become the bottom values of the new procedure's Local Environment and can be efficiently accessed as local variables. Upon return to the original procedure, a user-specified number of elements from the top of the called procedure's stack are copied back to the caller's stack. This makes parameter passage extremely efficient and nearly automatic.

2.4 Instruction Set

The AAMP comes with a very powerful instruction set. Included on-chip are 32- and 48-bit floating point add, subtract, multiply, and divide operations which make the AAMP a very capable single chip processor. These long operations, however, have execution rates which depend upon the data being processed, making exact timing prediction almost impossible. Appendix B provides a listing, generated by a program written by N. M. Mykris⁵, showing typical times of execution for the entire instruction set. Variable length instructions are indicated with an equation which shows dependence upon variables.

A major concern when estimating time of execution for programs is that of "stack thrashing". Upon the loading of an opcode into the micro-engine of the AAMP, a bit-mapping is performed which determines how many items must be on the stack in order for the impending operation to take place. This bit-mapping is optimal in most cases. In a few, unnecessary stack updating is performed during which data is read from or written to memory and the internal cache is rotated until the desired internal stack depth is achieved. These operations are very time consuming and can drastically slow down a non-optimized piece of code. By carefully selecting an appropriate order of execution and the proper opcodes, stack thrashing can, however, be minimized. Table 2, borrowed from K. L. Albin¹, shows the

bit-mapping and required stack depth for each instruction.

OpCodes	Stack depth allowed
-----	-----
00-1F	0-3
20-3F	0-2
40-5F	1-4
60-7F	2-2
80-9F	4-4
A0-BF	3-4
C0-DF	2-4
E0-FF	2-4

Table 2. Opcode to stack depth mapping
(from K. L. Albin¹)

3 Minimal Component AAMP Board

This section describes an AAMP-based microcomputer system designed and tested at Kansas State University for Sandia National Laboratories. The system described in this section is one used for power consumption measurements. For software testing, the system was modified to allow the output of data for verification of proper program execution.

3.1 Objectives of Design

The proposed use for the minimal component AAMP board is to provide the signal processing section of an ultra-low powered, "shirt pocket" sized helicopter noise detection system. The design was specified to minimize parts count and most importantly, minimize power usage.

Since the frequencies of interest for the detection of a helicopter are very low, 10 to 38 Hz⁶, sampling by the A/D portion of the system can be done at a low rate. Subsequently, data rates into the signal processing portion are relatively low. This allows the system clock frequency to be set low and aids in the minimization of power usage by CMOS parts whose power consumption is almost directly proportional to switching frequency.

The AAMP has a specified frequency range of from 4 to 20 MHz and is generally used as a high speed

microprocessor. Although the AAMP's speed is not fully utilized in this design, the powerful instruction set and its low power CMOS design make it the microprocessor of choice.

Sampled data is provided to the system through two 16-bit buffered inputs. The only outputs from the system are two flip-flops, one for each channel, to be set upon the detection of a helicopter. A block diagram of the system is shown in Figure 2 and a circuit diagram is shown in Figure 3. The system upon which all testing has been done was constructed using wire-wrap techniques.

Armando Corrales and Jim Heise have designed a printed circuit board which places the entire system, less the input buffers which may not be needed in a final system, on a 3" by 4" board. The design was done in six layers and is shown in Appendix C.

The original design and parts selection for the minimal component board were performed by Mr. Gaches in the Fall of 1985. Since the original design, changes have been made in the chip select/address decode circuit and in the approach taken to set the alarms.

The support components for the board can be classified into four subsystems: address decoding, memory, input/output, and clock. Each section, as assembled in the test system, is discussed separately below.

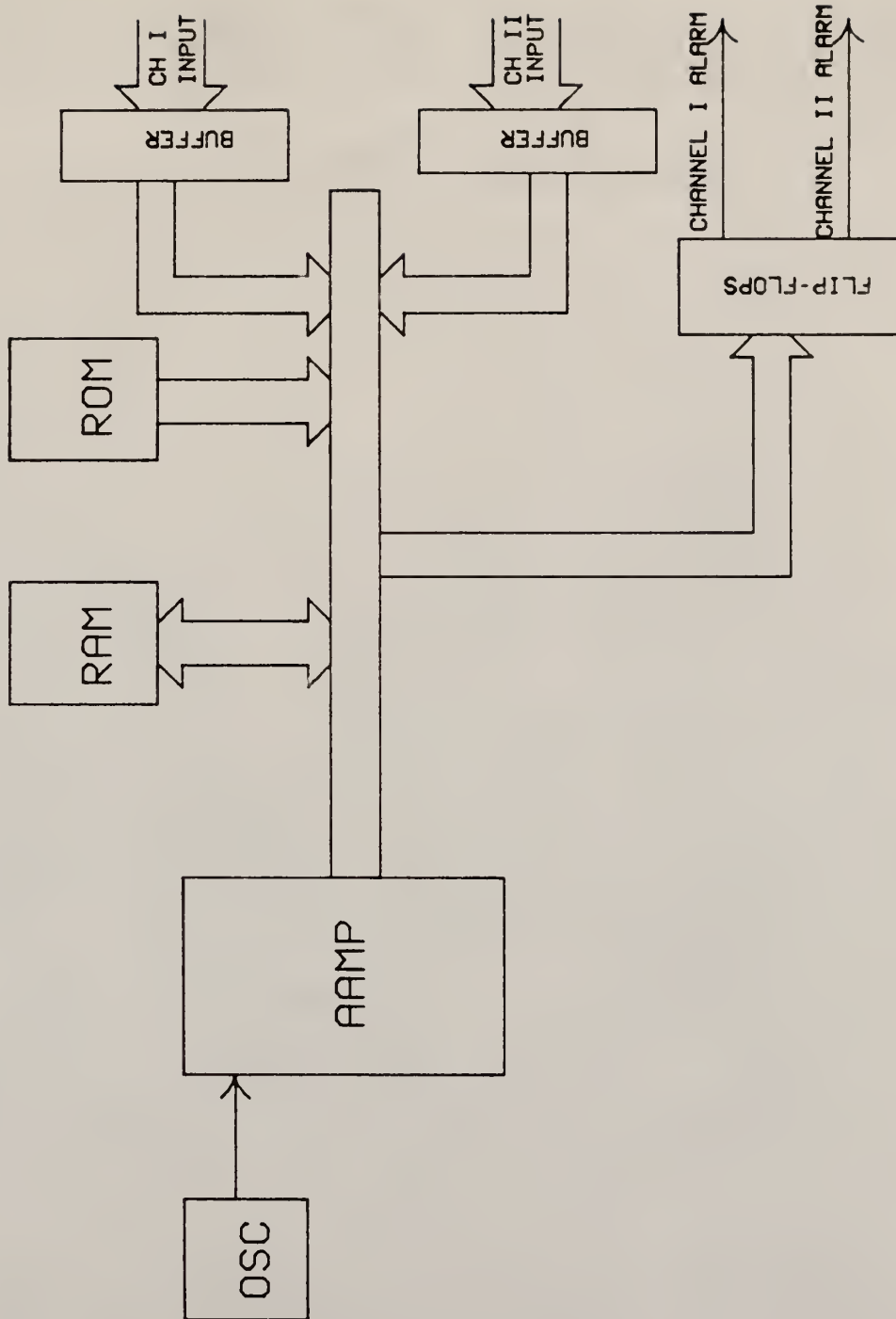


Figure 2. Block Diagram of Microcomputer System

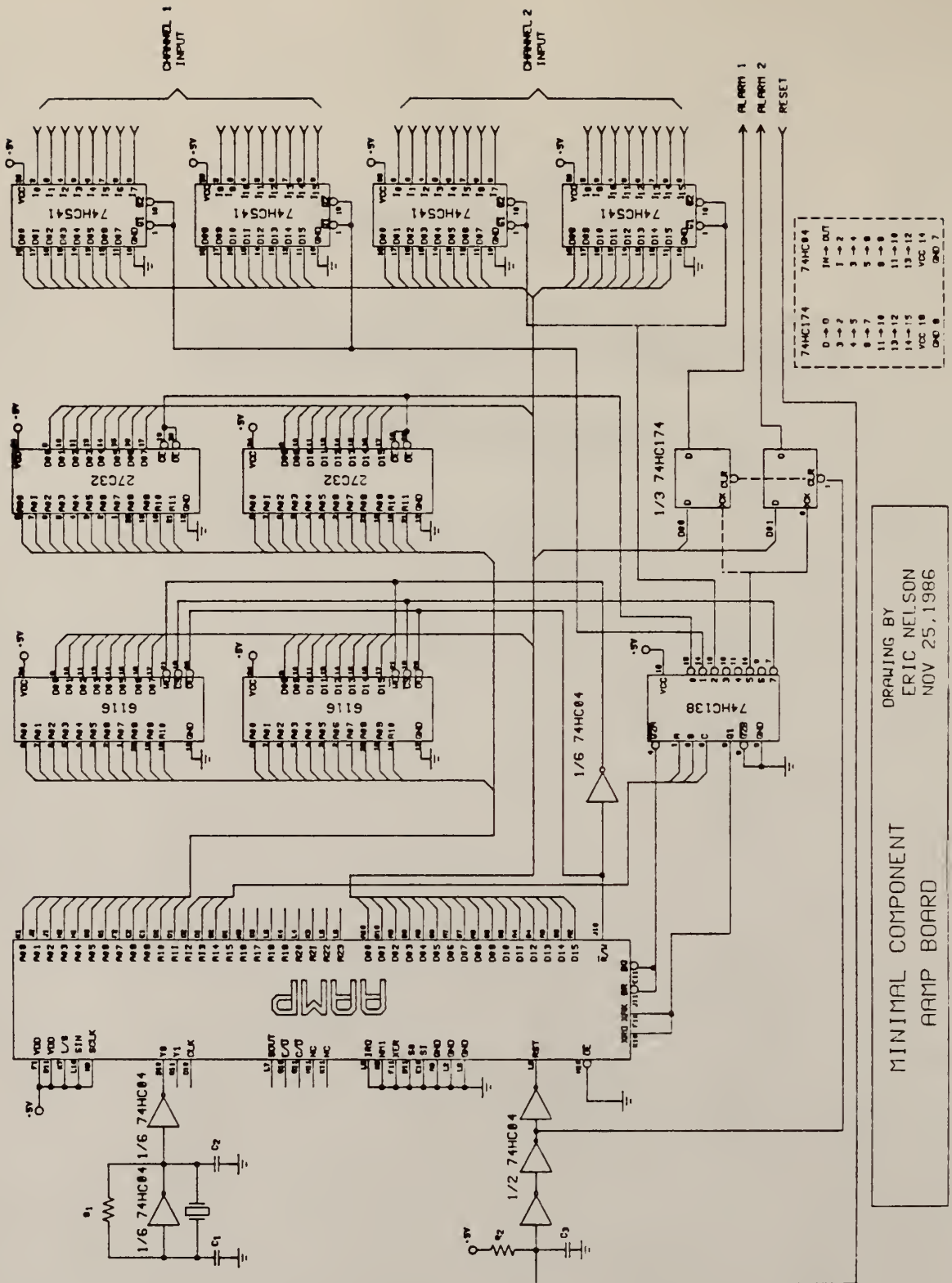


Figure 3. Circuit Diagram of Microcomputer System

3.2 Memory

Program memory space for the system is provided by two National Semiconductor NM27C32, 4096 X 8 bit UV erasable CMOS EPROMS connected side-by-side to provide a 16 bit wide data access. Upon assertion of RST low, occurring either on power-up or during a manual reset, the AAMP reads the Executive Entry Table from the lowest ten words of memory. Figure 4 shows the system memory map along with the location of these RST pointers. The function of these pointers is explained in the Architecture section and typical values are used in the Software section.

System RAM space is provided by two Hitachi HM6116 ALP-20, 2048 X 8 bit, 200 nsec static CMOS RAMS also connected side-by-side to provide a 16-bit data path. Addressing for the RAM is also shown in Figure 4.

3.3 Address decoding

System address decoding and timing is performed by a National Semiconductor MM74HC138, 3 to 8 line decoder. A system timing diagram is shown in Figure 5. Chip enable for specific components occurs upon the combination of XRQ/XAK high, $\overline{BG}/\overline{BR}$ low and a corresponding address. Under this configuration, the AAMP has control of the bus at all times, chosen by tying BG to BR, data transfer is synchronous, chosen by connecting XAK to XRQ, and a

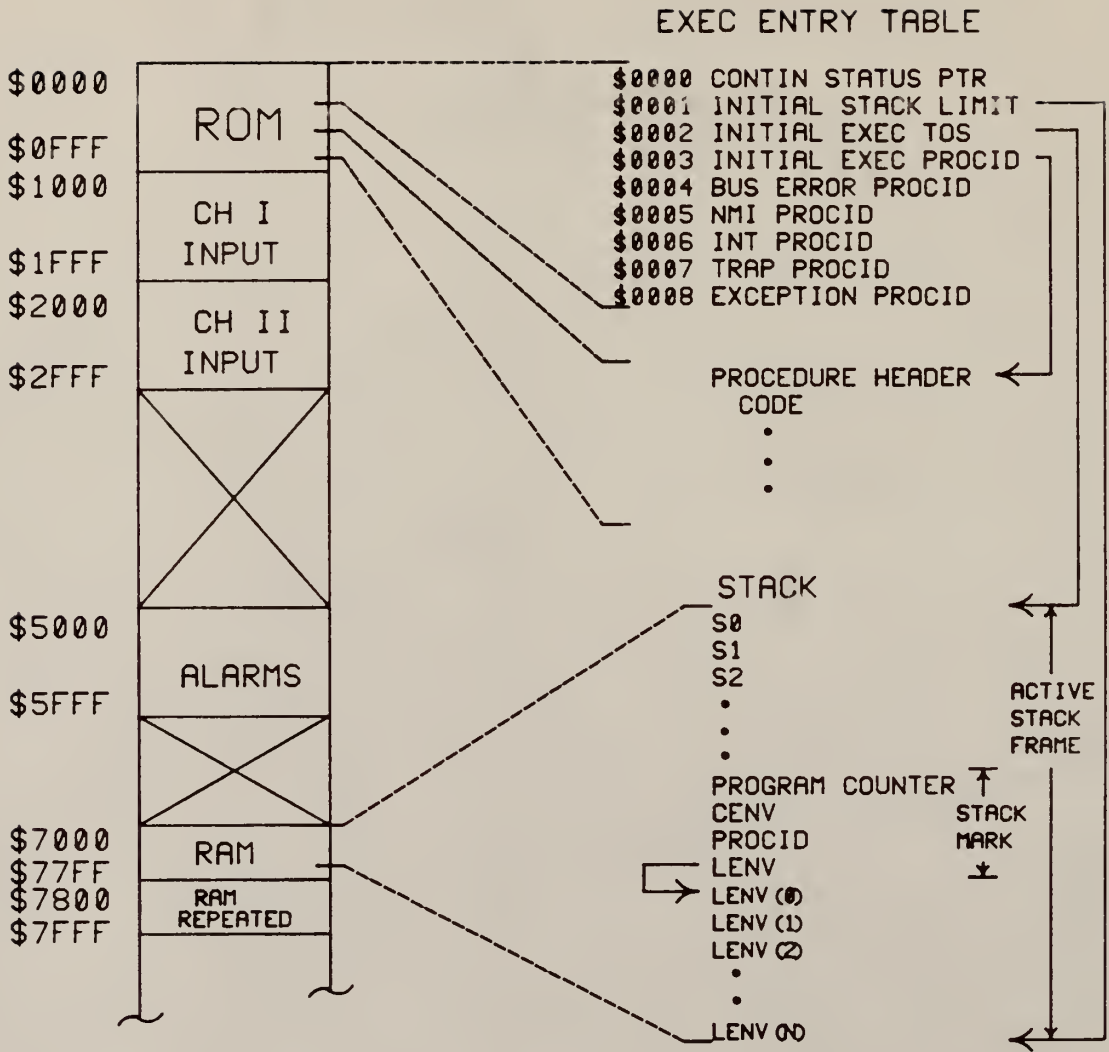


Figure 4. Memory Map

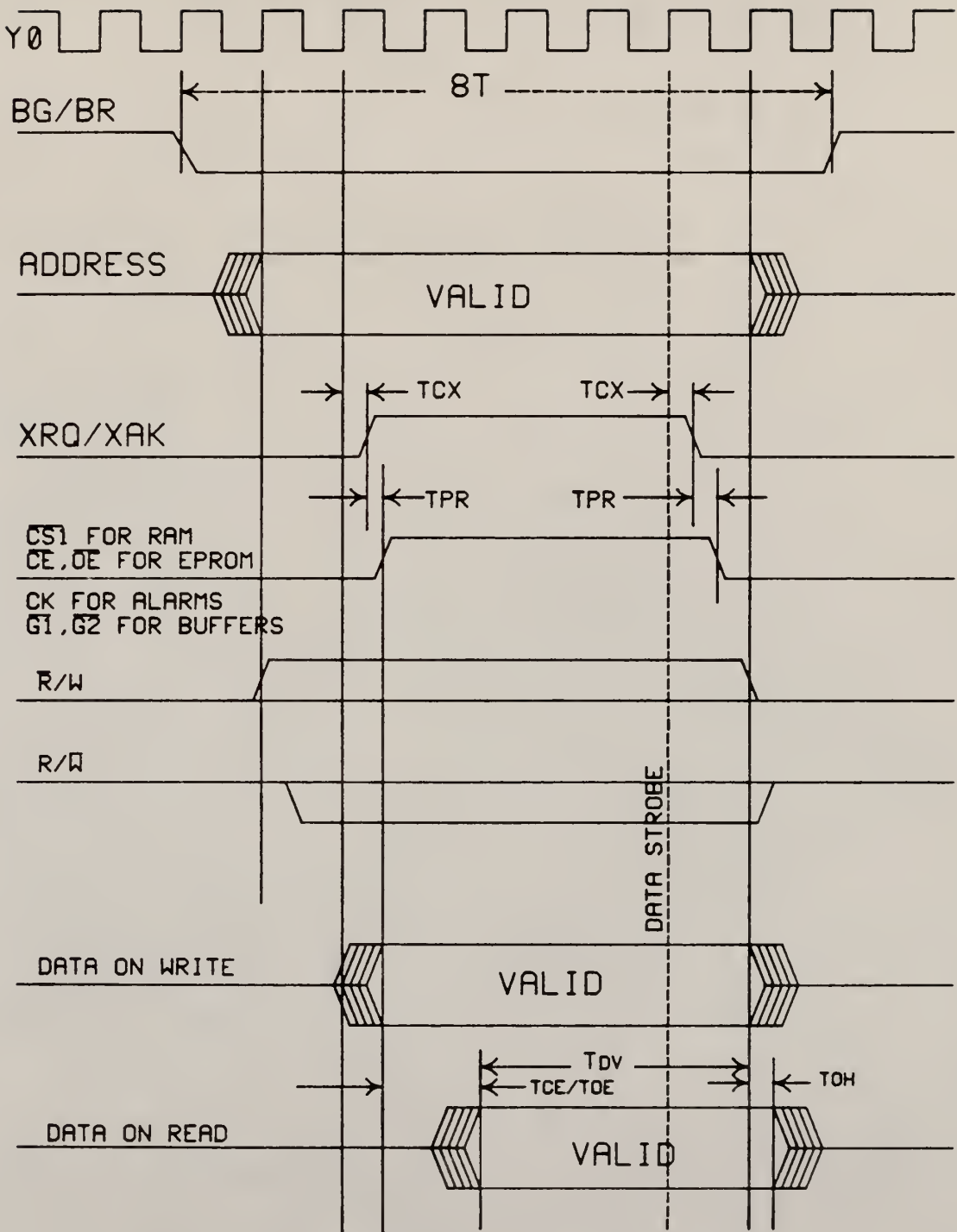


Figure 5. System Timing

complete bus transaction requires 8 clock cycles, set by strapping mode select pins S_1 , S_2 to ground. The most restrictive of the system timing equations pertains to a read from the EPROM. The equation is as follows:

$$T_{cxmax} + T_{prmax} + T_{cemax} + T_{dv} \leq 4T_{cyc}$$

Where T_{cyc} is the period at the oscillator input Y_0 .

T_{cx} is the time from a rising clock edge to the assertion of XRQ.

T_{pr} is propagation delay through decoder

t_{ce} is the time from chip enable until valid data is available from memory.

T_{dv} is the length of time that valid data must be held prior to a data read by the AAMP

Substitution of values from the AAMP Reference Manual³ and from the National Semiconductors CMOS Databook revealed that for a 450 nsec EPROM, maximum frequency is 6.78 MHz, while for a 350 nsec EPROM, maximum frequency is 8.16 MHz. All other system timing equations are less restrictive. National Semiconductor has promised a 200 nsec version, due to be available in January 1987, which speculatively could raise the maximum frequency to 11.8 MHz.

3.4 I/O

The two 16-bit input channels of the system are each provided by two National Semiconductor MM74HC541 Octal tri-

state buffers. Addressing for these is also shown in the memory map. For software testing, one of these input channels was changed to an output channel so that proper program execution could be confirmed. This was done by replacing two of the input buffers with MM74HC573 octal latches and inverting the existing address decoding to latch valid data from the system data bus. (Figure 6)

The original design of the system specified the alarm circuit to be two JK flip-flops in toggle mode using the address decoder for activation. This design was discarded since any noise in the system could have caused the flip-flops to toggle and since the microprocessor would not then know in which state the flip-flop had landed. Instead, two D flip-flops, activated by the address decoder and with inputs connected to the data bus were used. Under this configuration, the state of the alarm is controlled with a much greater confidence of outcome.

The type of D flip-flop was then chosen on the basis of power consumption. The choices were, using National Semiconductor parts, the MM74HC74, a 14-pin dual D flip-flop with separate preset and clear, or the MM74HC174, a 16-pin hex-package D flip-flop with single clear and no preset. On the basis of lower power consumption, the MM74HC174, although in a larger package and having extra components, was chosen. The reasons for this choice are explained in the Power Consumption section.

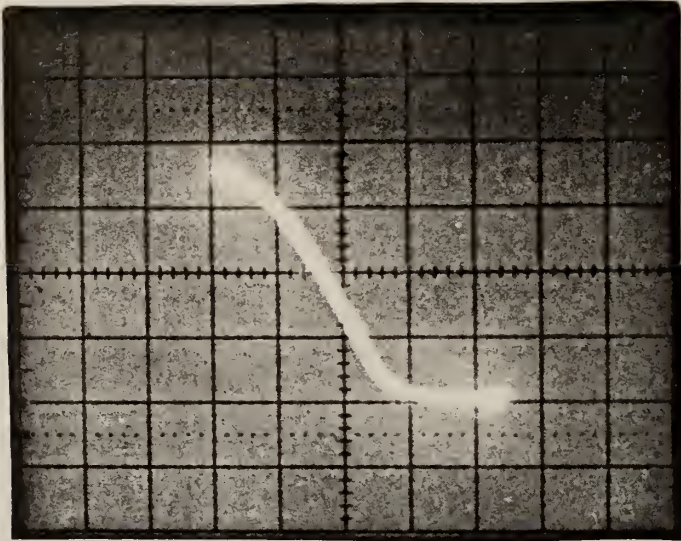
3.5 Inverter/Oscillator

Since the AAMP's on-chip oscillator is rated for operation from 4 to 20 MHz and since the system may be working at frequencies below this, an external oscillator circuit was used. The circuit used is a parallel resonant gate oscillator which utilizes a CMOS inverter as a driver and another inverter to "square up" the waveform. The particular inverter chip to use is dependant upon the system's frequency of operation.

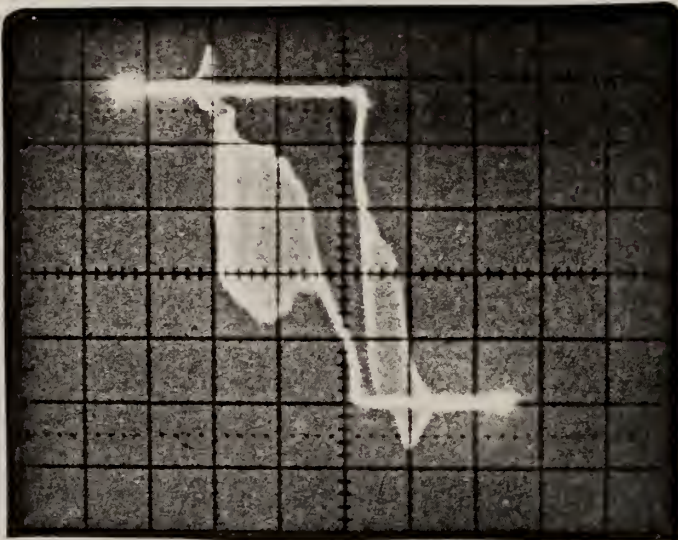
A standard (54C/74C family) CMOS inverter will use less power and make a more stable driver for an oscillator than will a High Speed (HC) CMOS part. This can be explained by examining the transfer characteristics of each device. Standard CMOS parts have a smooth linear region with constant although low gain whereas HC parts have very high gain with the switching region nonlinear and suffering from massive jitter. See Figure 7.

These non-linear portions of the waveform for the HC parts inject higher frequencies into the loop causing the oscillator to operate at an overtone of the crystal frequency. The overtone problems seemed to vanish when using crystal frequencies above 4 MHz making HC parts an acceptable choice for higher frequency oscillators.

Unbuffered high-speed CMOS (HCU) parts which are touted for use in gate oscillator circuits were also



Standard CMOS
x = 1 Volt/div y = 1 Volt/div



High-Speed CMOS
x = 1 Volt/div y = 0.1 Volt/div

Figure 7 Transfer Function of Inverters

tested. The transfer function for the HCU parts was smooth and linear with little jitter and the waveform output from an HCU oscillator showed a very square waveform with short rise and fall times. Power consumption for the HCU parts was, however, very high and further testing was not deemed useful.

In order for an oscillator to operate in the parallel resonant mode, there must exist 360° phase change through the loop, 180° of which must exist in each half of the circuit. See Figure 8.

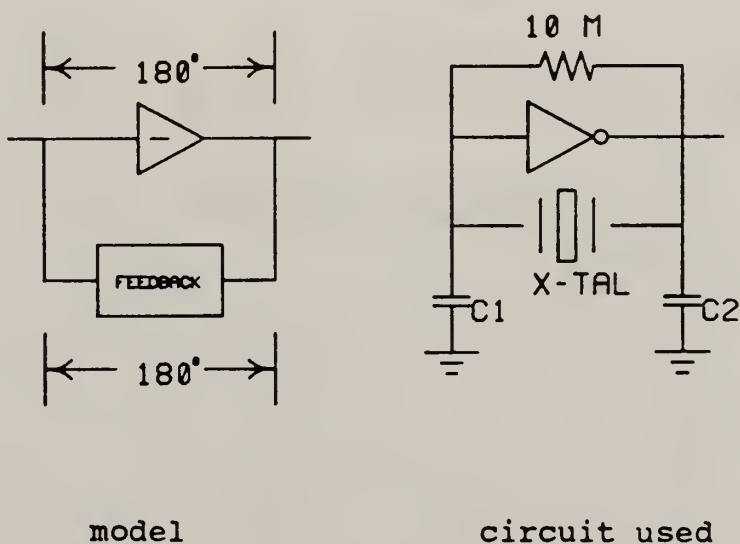


Figure 8. Phase Shift in Parallel Resonant Oscillator
Adapted from Holmbeck⁷

In this system, the feedback loop consists of a crystal connected in parallel with a large valued resistor. The tie-down capacitors at each terminal of the crystal are to match the effective load capacitance of the circuit

with that of the crystal while the resistor acts to force the gate into its linear conducting region. At high frequencies, the propagation delay through the inverter causes the following phase shift:

$$\theta = f * t_{pr} * 360^{\circ}$$

For a 3 MHz oscillator using the worst-case propagation delay of 90 nsecs for standard CMOS, this value is 97.2° . To compensate for this inductive phase change, (output current lags input voltage), a capacitor may be placed in the loop as shown in Figure 9.

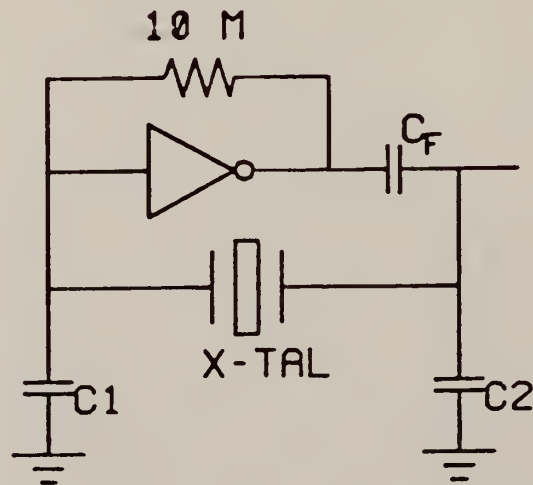


Figure 9. Circuit Compensating for Gate Delay.

A suggested value for C_f is $1/C_{eq}^8$, where C_{eq} is the input impedance viewed from the output of the gate into the

crystal feedback network. Adding this component to the circuit indeed increased the useful bandwidth of the system. However, power consumption was also increased. Instead, the value of C_2 was raised and the value of C_1 was decreased. This was done to increase C_{eq} and minimizing the value needed for the now missing C_f , yet maintaining a balanced load across the crystal.

Using this type of capacitive phase shifting, the standard CMOS inverter oscillator was found to be very reliable up to 3MHz. Above 3MHz, the slow rise time and large propagation delay of the standard CMOS became so dominant that the waveform no longer attained the AAMP's required external oscillator input voltage swing of from $0.6 V_{max}$ on the low cycle to $4.2V_{min}$ on the high cycle.

The inverters are also used to produce a delayed reset on power-up. Rockwell suggests that the reset should remain low for around 1000 clock cycles prior to the microprocessor being enabled which would dictate a reset time of around 1 millisecond for the lowest clock speed used. However, the power supplies used in the laboratory require nearly 4 milliseconds to reach 5 volts when warm and nearly 20 milliseconds to reach 5 volts when started cold. Taking this into account and designing the reset to be delayed by a sufficiently long time resulted in such a gradually rising output from the RC network that

upon switching from high to low the output of a single-buffered network chattered. The problem is solved by triple-buffering the reset signal from the RC network to the Reset input on the AAMP. This raises the total gain and causes the switching to be more abrupt, disallowing chattering at the output.

In conclusion, while operating at frequencies below 3MHz a National Semiconductor MM74C04 hex inverter should be used and while operating above 3MHz an MM74HC04 hex inverter should be used. These chips are pin-for-pin compatible and can easily be interchanged.

4 System Power Consumption

Power consumption testing for the system was performed by breaking into the power bus and connecting a Fluke 8010A digital multimeter in series with the supply voltage to the component under test and measuring the current. To obtain an instruction mix which should be typical for the proposed application, the Standard Widrow Adaptive Linear Predictor algorithm, originally coded for the AAMP by K. L. Albin¹, was used.

Data input to the system was provided by simply tying the inputs to ground. This prevented peripheral devices from affecting consumption readings but also allowed many flip-flops to remain unswitched and possibly caused a lower power reading than real data input should cause. For this reason and since power consumption for a particular component varies from chip to chip, the figures presented here should be used for comparisons and not as a guaranteed rating.

This section discusses the power usage of the system. Parts selections made on the basis of power consumption are discussed in more detail here and a type of oscillator is suggested for each portion of the system's frequency range.

Figure 10 shows the power supply current versus operating frequency for the total system, the AAMP and if used, the external oscillator circuit. All system

componets run on a five volt supply. Conversion from current to power consumption required a multiplication by 5. A straight line has been fit to the data set indicating the trend for the data in the reliable regions of each configuration. Although worst case timing showed that the system should only be expected to operate up to 8.16 MHz, testing of power consumption was continued up to 10 MHz.

Bulk CMOS AAMP

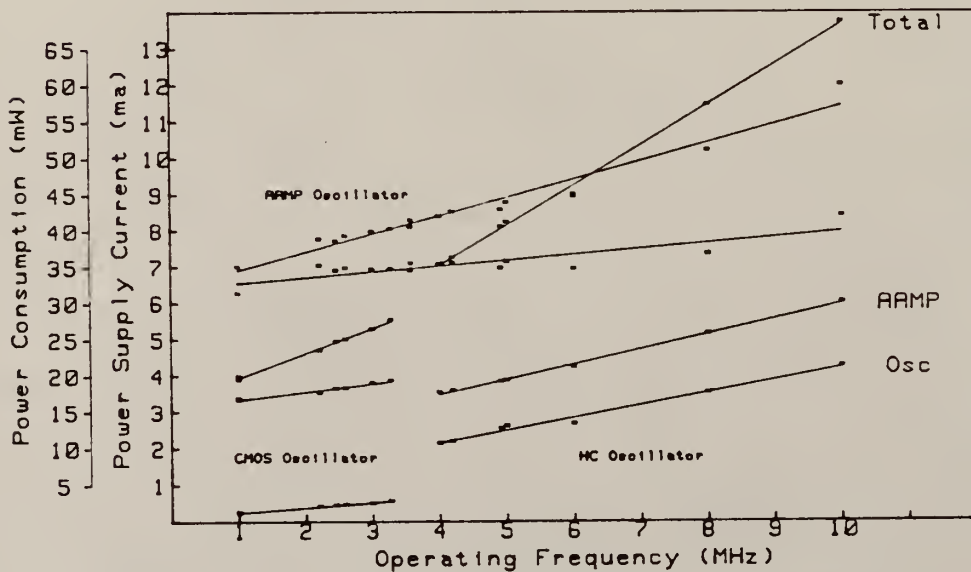


Figure 10. Power Consumption of System

Straight-line approximations for the external oscillators show a good fit. Consumption for the system using the on-chip oscillator however, showed a slight exponential increase but is still reasonably approximated with a straight-line fit.

4.1 AAMP

The AAMP itself shows a y-intercept value uncharacteristic of CMOS parts. The varying y-intercept values for the three oscillator configurations can be related to the amount of gain required by the AAMP's clock input gate. While using the AAMP's on-chip oscillator, the buffer at the Y_0 pin is required to act like a linear device. In this mode, rise times are long and the gate spends a large amount of time between the on and off states. Nearly all power consumption by CMOS gates can be attributed to the amount of time spent in this state. The sharp square wave output of the high-speed CMOS oscillator, on the other hand, causes the y-intercept of the AAMP to be lowest in this configuration. Standard CMOS waveforms have slower rise times and cause a slightly higher y-intercept.

4.2 Oscillator

As seen in the plot, the Standard CMOS oscillator was only useful below 3 MHz but in this region showed the lowest power usage. The oscillator circuit is shown in Figure 11.

Trimmer capacitors were used in the test system. The values were adjusted to minimize power consumption and maximize voltage swing of the output. These two conditions generally occurred simultaneously with the value of C_1 often around three fourths the value of C_2 and their series

combination being approximately equal to the crystal's specified load capacitance.

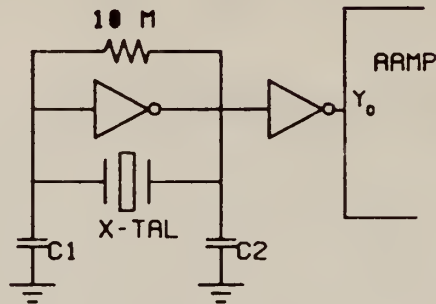


Figure 11. External Oscillator Circuit Used for System

The circuit of Figure 11 was also used for the HC oscillator resulting in about the same capacitor ratio but with the series combination now being equal to around twice the rated load capacitance. As seen in the plot, the HC external oscillator should be used in the frequency range from 4 to 6 MHz.

In the range of frequencies from 3 to 4 MHz and again above 6 MHz, the AAMP's on-chip oscillator should be used. Although its operation is reliable throughout the tested range of 1 to 10 MHz, power consumption is generally higher than that of the others outside this suggested range. The on-chip oscillator of the AAMP uses the same equivalent circuitry as the external oscillator in Figure 11. The suggested circuit for using the on-chip oscillator requires no load capacitors.³ Power consumption can however be

lessened by attaching a small load capacitor with a value near the specified load capacitance of the crystal to the Y_1 pin. (Figure 12)

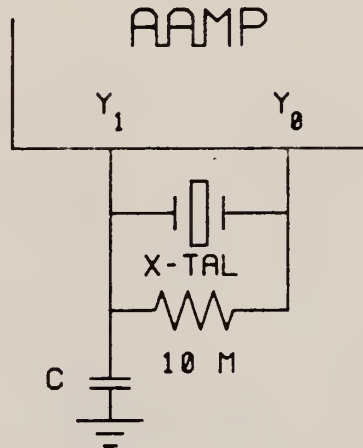


Figure 12. Configuration for On-chip AAMP Oscillator

4.3 Support Components

Power consumption for support components is shown in Figure 13. Values for power usage by support components are well fit by a straight line with no noticeable y-intercept. During power measurements, data was static therefore creating an artificially low amount of gate switching in the RAM. The slope for the RAM will, of course, change with the amount of gate switching on writes.

The type of flip-flop used for the alarm was chosen on the basis of power consumption. The configuration used calls for the flip-flop to load the values from the data bus upon a write from the microprocessor with the clock

input to the flip-flop held high except in the rare case of the detection of an intruder.

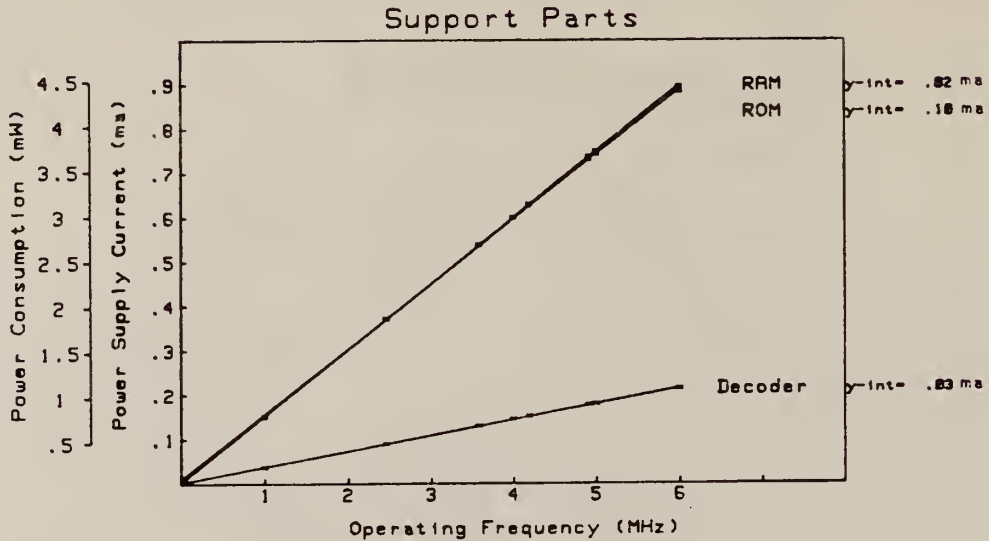


Figure 13. Power Consumption of Support Components

Under these conditions, and although no output switching was being performed, the MM74HC74 dual D Flip-Flop used around five times as much power as the MM74HC174 hex D Flip-Flop. A call to Larry Wakeman, Applications Engineer for National Semiconductors, confirmed that the MM74HC74 dual package was a poorly designed CMOS part, with flaws that make it non-ideal for ultra-low power systems. These problems are shown in Figure 14.

The dual package routes the input through two buffers and directly into two logic gates whereas the hex package runs the input through a single inverter then uses a

transmission gate to virtually isolate the single-buffered input from the rest of the circuit. The more densely packed hex inverter also requires smaller internal component size than the less space-restrictive design of the dual package. Figure 14 shows the effective circuitry of the two packages.

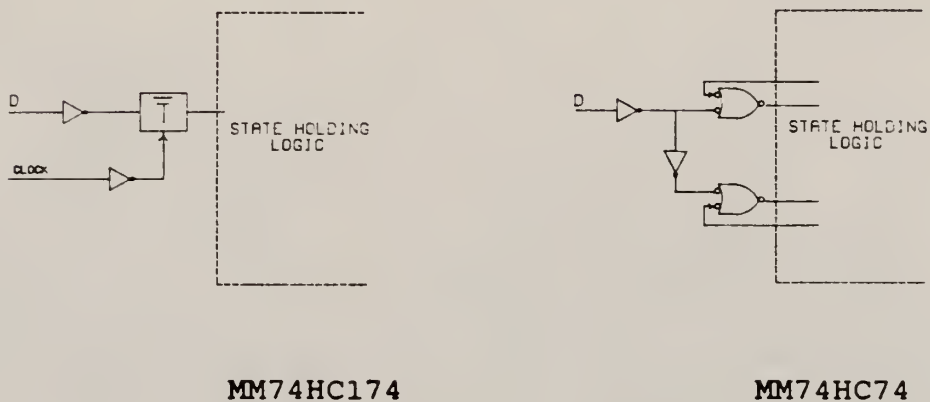


Figure 14. Effective Input Circuits of D Flip-Flops

Although power consumption for either component, while not being clocked, but under the highest bus rate possible is under one milliwatt, the MM74HC174 hex flip-flop was chosen for the design. The extra flip-flops may also be used to control a proposed wait state mode.

4.4 Wait State Operation

The following section describes a technique where, through hardware, the need for exact timing of algorithms

may be eliminated. The method would allow the AAMP to perform all necessary calculations for a sampling interval, then go to a high-impedance, lessened-power state until awakened by a signal from a controller upon the next completed sample conversion. The added hardware is shown in Figure 15. The flip-flop controller was taken from the spares of the MM74HC174 Hex package. Therefore, no active components are added to the design, with circuit additions being only resistors, diodes and added traces.

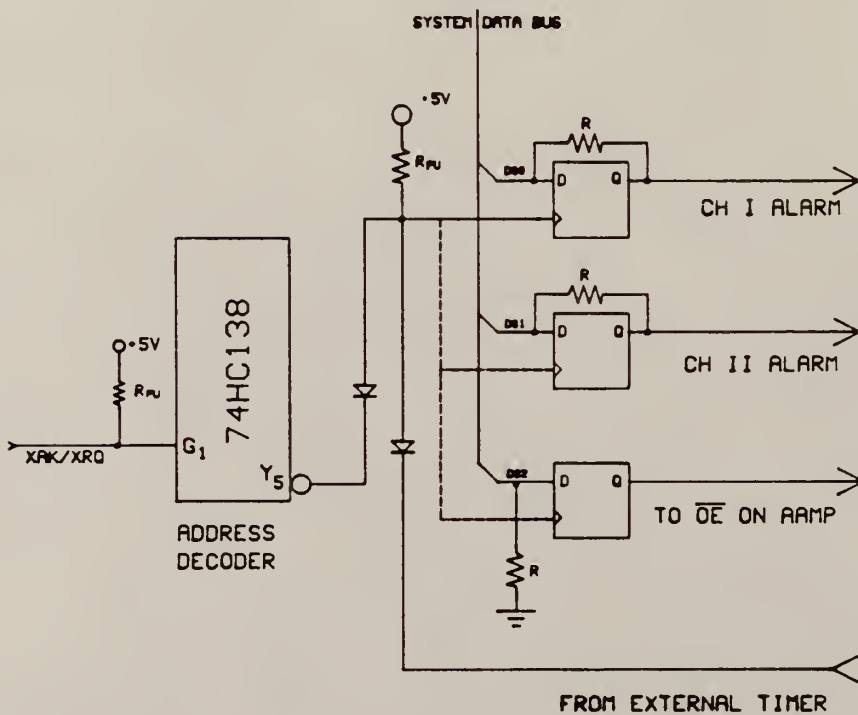


Figure 15. Wait State Circuitry Maintaining Two Alarm Outputs

During the high impedance state, the controller flip-flop will hold the \overline{OE} pin high until a new clock signal is

received from an external timer at which time the controller flip-flop will be cleared. Timer and decoder clocking of the flip-flops are wire ANDed to allow either to pull the clock low. During clocking of the flip-flops by the timer, the present states of the alarms are maintained by the identity feedback resistors while the input to the controller is pulled low by a high valued pull-down resistor.

Programming to reach the high impedance state would simply require that a hexadecimal four be ORed with the alarm state and written to the alarms as the final instruction of the routine. Upon being awakened from the wait mode, a dummy read should be performed and program execution may then be restarted.

The present version of the system has two alarm outputs. If it were possible to run the system with only one alarm for output, a more straight-forward wait controller could be built. Figure 16 shows the simplified controller which is made possible by using the MM74HC74 which has individual asynchronous clears. Software using this version would require a write to the alarms upon completing the processing of a set of samples and a dummy read upon being awakened. This method, although using the less efficient MM74HC74, would probably require less power since pull-down and identity feedback resistors would no

longer be needed.

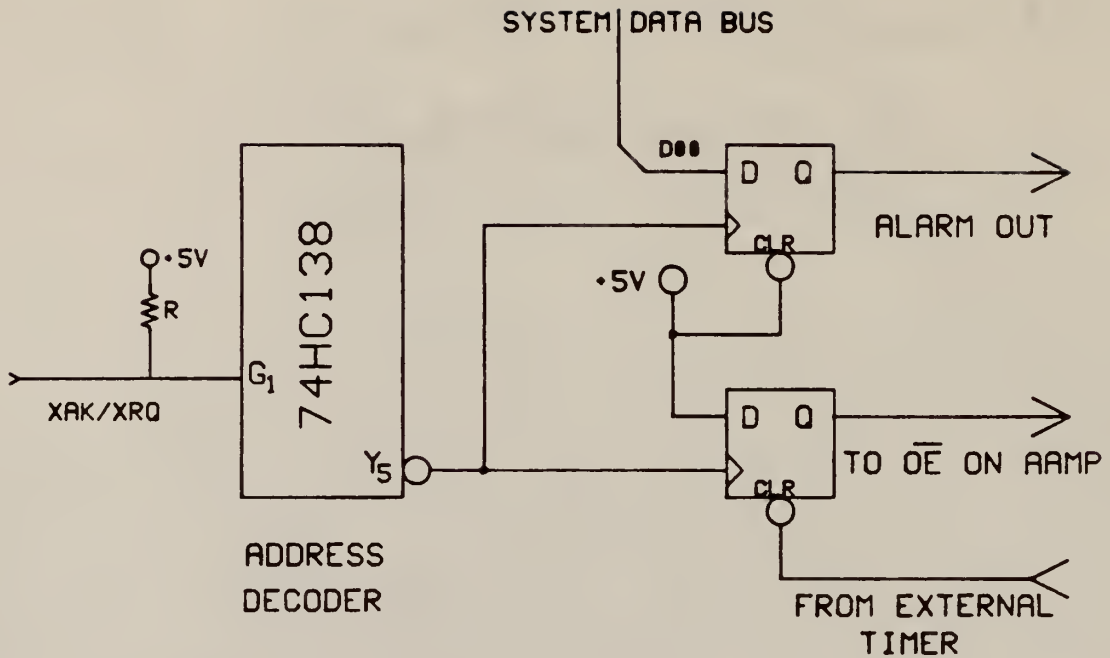


Figure 16. Wait State Circuitry Converting Alarm Output to Controller

The operation is the same when using either type of controller. With the \overline{OE} pin held high, the AAMP will try to assert a transfer request (XRQ) signal which would be read at the transfer acknowledge (XAK) pin and would allow program execution to continue. Instead XRQ is tri-stated and program execution is suspended. In this mode of suspension, all output pins except Bus Request (\overline{BR}) are tri-stated and only the oscillator portion of the processor is switching. The AAMP, however, buffers the clock signal

several times before using it throughout the processor and therefore power consumption does remain considerable during this high impedance state.

Power consumption for the total system in the high impedance state is shown in Figure 17. Dashed lines indicate total power for normal operation, while solid lines show a best-fit line for wait mode power consumption. While using the AAMP oscillator, power consumption stays a near-constant eight milliamps with even a slight decrease in power consumption as the frequency of operation approaches the frequency for which the oscillator was primarily designed. When using the on-chip oscillator, the break-even point for using the high impedance wait mode as compared to continuous operation occurs around 3.5 MHz while when using either of the external oscillators a slight decrease is seen at all frequencies.

Effective power consumption for the system while using wait state mode can be calculated by adding each phase multiplied by its duty cycle. Programming time overhead is not large, requiring only an additional 15.5 clock cycles for the masking of the data bit and a non bus-access function during wake-up.

This method allows accurate timing of sampling rate to be performed by an external device which should be possible if, like previous low-power A/D's built at KSU, the A/D

chosen for the system utilizes a microcontroller with fixed times for instruction execution.

Bulk CMOS AAMP

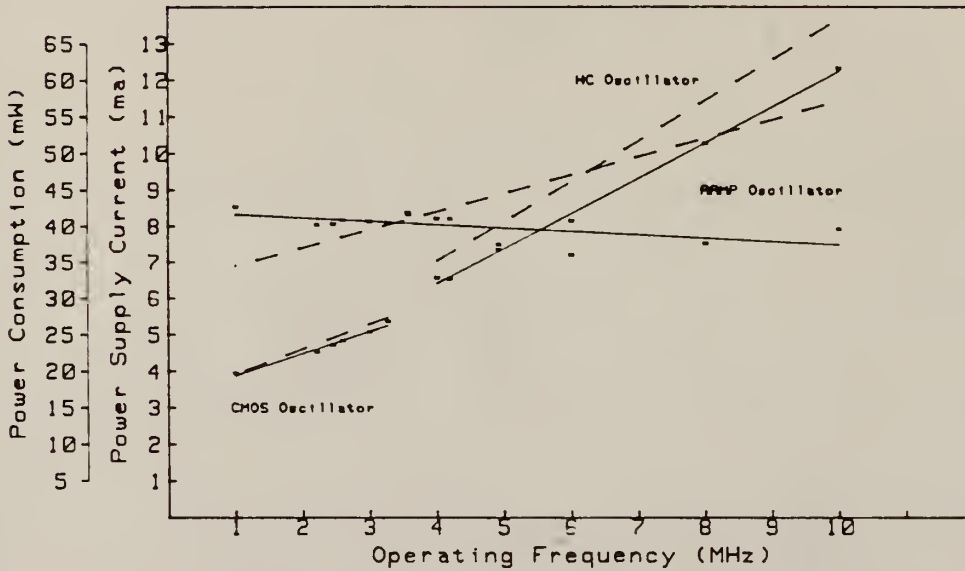


Figure 17. Power Consumption of High Impedance Mode -dashed lines represent normal operation

Software Evaluation

In order to properly evaluate the performance of the AAMP as a signal processor, various digital filtering sub-programs were coded. The time of execution figures listed here should provide rough estimates of processing speed and should provide a useful basis for comparing the AAMP to other signal processors.

5.1 Verification of output

Several of the algorithms have been tested on the proto-type board and the actual time of execution is shown for those. Program segments from which timing figures were taken are shown in Appendix D. Testing was performed with the system operating at 1 MHz which was chosen to allow easy conversion to the number of cycles and to allow the test equipment, an HP 9845B with GPIO, to keep up with data transfers while debugging and testing of program segments.

Time of execution for the segments tested on the proto-type system was measured using an HP 1611A Logic State Analyzer. Output from the filter was monitored for proper execution using an HP 9845B computer with data transfer exchanged through a half-handshake board-to-GPIO controller. (Figure 18)

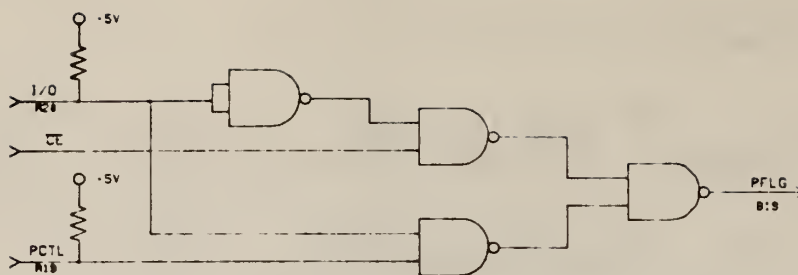


Figure 18. Half-handshake Controller for GPIO Interface

HP's GPIO provides a programmable control register which allows the user to determine mode of transfer and type of handshaking. This register is, however, programmed by installing the appropriate wire-wrap jumpers and cannot be changed during program execution. Therefore, a controller, which allows the AAMP system to run at full speed requiring only the host computer to poll for data transfer was used.

5.2 Speed optimization

Appendix D shows listings of the programs with time of execution for each block of code identified. Coding of the algorithms was done in the most straight-forward approach possible with speed optimization done only on a near-sighted basis.

In loop coding, speed optimization for storage and retrieval of counters and intermediate results was

accomplished by using the Local Environment storage area, which contains the 16 quickest access locations. To speed line coding, arrays of variables were placed in the Local Environment Extended. Segments were made interchangeable by maintaining these memory assignments in both forms of coding.

Addressing of array elements while loop coding is accomplished by calculating the base address of the LENV, adding a constant offset component to specify which array was being accessed, and adding I to find the I^{th} element of the array. The AAMP has quick and efficient commands for these three actions.

A method to increase speed of execution which was tried but with little success was that of leaving a copy of the count variable on the stack at the end of each iteration. Using this method, calculated times for execution were much lower than for the final versions, but observed times were much greater.

This difference from calculated to observed may be attributed to stack thrashing. The prime instruction for creating copies of the count variable is the DUP command which, as indicated by Table 2 of the Architecture section, is a non-optimal stack depth command. Using DUP causes the processor to read or write from its internal cache to RAM, each time transferring members within the internal cache

itself, until exactly two elements are left in the internal stack. This stack thrashing caused execution times to greatly increase. Therefore, the use of DUP should be avoided except when the stack depth is already two or when the time to calculate the data on the stack was more than that encountered during a stack penalty. (approx. 24 cycles for a one-item stack update).

5.3 Accuracy of estimates

Timing estimates, especially for loop coding, were not extremely accurate, generally falling only within +20% of the observed. Therefore, the timing figures listed should only be used as a rough comparison with other processors. Precise prediction of timing is not possible since some parameters are data-dependant while others are dependant upon the recent history of the processor. The large discrepancy between calculated and observed times for loop coding can be justified by considering that any error in the prediction of timing within the loop is multiplied by the number of times through the loop. Timing estimates for inline coding were consistent with the observed.

5.4 Timing Estimates

Table 3 shows time of execution for the various digital filtering sub-programs which have been coded. Each subprogram was coded in three data formats: Single Precision Fractional, Single Precision Floating (24-bit

mantissa, 8-bit exponent), and in Double Precision Floating (40-bit mantissa, 8-bit exponent). Where applicable, segments were written using both loop- and linear-coding. Overhead time for loop set-up is included in the total time for 16 iterations while time-per-N figures exclude this figure.

Table 3 shows the algorithms used for the software evaluation. In the fixed-point fractional data format of programming, alignment of data is performed in assembly language whereas in both floating point implementations, data is automatically aligned in the much more efficient microcoding. This explains why the fixed-point ratio calculation required more time than the floating-point versions. In other sections of code, the time of execution increases with precision.

Table 3. Algorithms used for Software Evaluation

FIR Filter
$$Y = \sum_{i=0}^N a(i) * x(i)$$

Filter Update

$$x(i + 1) = x(i) \quad i = 0 \text{ to } N-1$$

Ratio Calculation

$$r = x^2/y^2$$

Decision

$d = \begin{matrix} 1 & r \geq \text{THETA} \\ 0 & r < \text{THETA} \end{matrix}$

Weight Update

$a(i) = a(i) + da * y(i) \quad i = 0 \text{ to } n$

IIR Filter

$v(m) = (1 - \text{BETA}) * v(m-1) + \text{BETA} * x^2$

Table 4 Time of Execution of Several Digital Filters

Operation	Single Prec Fractional		Single Prec Floating		Double Prec Floating	
	loop	line	loop	line	loop	line
FIR Filter						
time/N	269.5	137	635.5	493	1104	936
16-coeff	4344	2213	10209.5	7918.5	17720.5	15016
Filter Update						
time/N	143.5	33	153.5	43	193	107.5
16-coeff	2163.5	495	2473	643	2911	1612.5
Ratio Calculation	----	1236	----	996	----	2048
Decision	----	49	----	103	----	195
Weight Update						
time/N	297	169	658.5	530.5	1146.5	1043
16-coeff	4768.5	2704	10552.5	8488	18360.5	16696
IIR Filter						
l = 2	----	378	----	1132.5	---	2255

CONCLUSIONS

Although the AAMP was designed largely for use in high-speed, large imbedded-processing systems, the AAMP makes a very efficient microprocessor for minimal component, minimal power systems. Its capacity for floating point arithmetic and the inherent low power consumption of CMOS make the AAMP a very attractive processor for applications in remote intruder detection systems. Bus protocol using the synchronous mode requires a minimum number of glue chips (one) while bus transaction time may be adjusted to allow the use of slow memories while maintaining a high ALU speed.

Once the change-over from register-oriented assembly language programming to stack-oriented programming is made, code can actually be written quicker using the AAMP's high level commands. Local variables, although slower than true registers, may be used for quick access and temporary storage. The various built-in data types of the AAMP make coding for various precisions of data simple and closely parallel.

For this thesis, a system intended to provide the digital signal processing portion of an ultra-low power intruder detection system was built and tested. Very few problems were encountered while using the AAMP and, as a

rule, items stated in the reference manual were found to be true. However, a problem was found in the intermediate Bulk CMOS packages as pull-up resistors were not placed on some test pins with no mention of this made in the AAMP manual³. Pull-ups were added to the circuit solving the problem and Rockwell has since changed packaging and included the pull-ups in the new design.

Excellent support for the project was provided by Rockwell through constant contact with K. L. Albin. Much of the information given for packaging and availability of the AAMP was gathered during a Sandia-sponsored trip to Cedar Rapids in October, 1986. Several meetings were arranged where, Dave Best and several other Rockwell personnel gave presentations on the AAMP.

With the type of support for the AAMP which was shown and with the outstanding characteristics of the AAMP itself, the AAMP should definitely be considered when designing any "modern" system.

REFERENCES

¹Kenneth L. Albin, "An Evaluation of Rockwell's Advanced Architecture Microprocessor for Digital Signal Processing Applications," (Master's Thesis, Kansas State University, 1984).

²Gary S. Mauersberger, "The Design and Hardware Evaluation of an Advanced 16-bit, Low-Power, High performance Microcomputer System for Digital Signal Processing", (Master's Thesis, Kansas State University, 1985).

³Advanced Architecture Microprocessor Reference Manual, (Avionics Group, Rockwell International Corporation, Cedar Rapids Iowa, 1985).

⁴Statement by Ken Albin, Technical Staff Member, Avionics Group, telephone interview, Rockwell International, Cedar Rapids IA 52498, Nov 21, 1986.

⁵N. M. Mykris, Avionics Group, a program to calculate Bulk CMOS AAMP Instruction Execution Times, Rockwell Internation Corporation Cedar Rapids IA).

⁶Nasir Ahmid, T. Natarajan, Discrete-Time Signals and Systems, (Reston Va: Reston Publishing, 1983).

⁷J. D. Holmbeck, "Frequency Tolerance Limitations With Logic Gate Clock Oscillators," (Proceedings of the thirty-first annual Frequency Control Symposium, Fort Manmouth, NJ, 1977, pp. 390-95).

⁸Thomas B. Mills, Application Note 340, Logic Databook Volume 1, National Semiconductor Corporation, pp. 2-138, 1984.

APPENDIX A

The latest version of the AAMP has a different pin-assignment than the previous versions tested. The following is a table of the pin-assignments for the three versions of the AAMP seen at Kansas State. The original CMOS/SOS version was used by G. S. Mauersberger² and by Mike Gaches in earlier designs. The implementation described in this thesis uses a Bulk CMOS AAMP from a transition stage in packaging. The AAMP is now packaged as a top-cavity (TC) device and the pin-out has again changed. The printed circuit board design in Appendix C uses this new design. Present and future versions of the AAMP will follow the newest assignment. Figure A-1, taken from the AAMP reference manual³ shows the pin assignment and physical dimensions for the 68 pin pin-grid-array package.

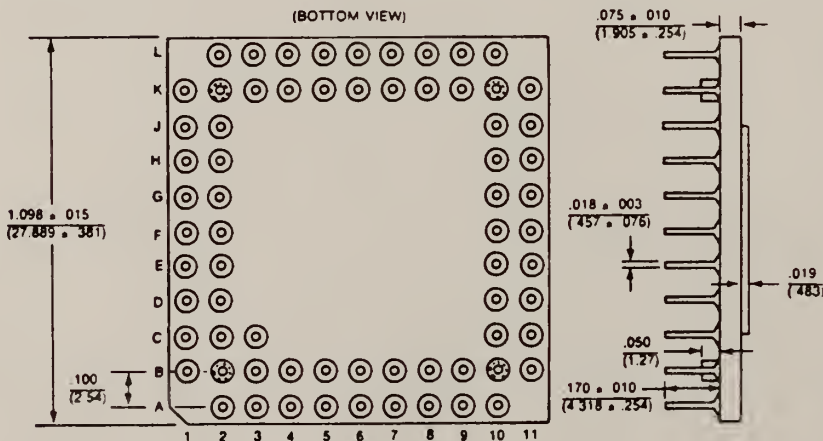


Figure A-1 Pin assignment for 68 pin PGA AAMP
(from AAMP Reference Manual³)

Table A-1 AAMP Pin Assignments by Functions

SIGNAL	CMOS/SOS	BULK CMOS	TC BULK CMOS
Supply			
VDD	B11	B11	B2
	---	F1	B1
	---	---	D1
	---	---	K6
	---	---	B10
	---	---	A10
GND	L6	L6	F2
	---	L2	F1
	---	A5	G10
	---	---	A2
Address			
A23	K2	K2	K1
A22	L3	L3	J1
A21	K3	K3	J2
A20	L4	L4	H1
A19	K4	K4	H2
A18	L5	L5	G1
A17	K5	K5	G2
A16	K6	K6	E1
A15	B1	B1	L10
A14	B2	B2	L9
A13	C1	C1	K9
A12	C2	C2	L8
A11	D1	D1	K8
A10	D2	D2	L7
A09	E1	E1	K7
A08	E2	E2	L6
A07	F2	F2	L5
A06	G1	G1	K5
A05	G2	G2	L4
A04	H1	H1	K4
A03	H2	H2	L3
A02	J1	J1	K3
A01	J2	J2	L2
A00	K1	K1	K2
Data			
D15	A2	A2	K10
D14	B3	B3	K11
D13	A3	A3	J10

D12	B4	B4	J11
D11	A4	A4	H10
D10	B5	B5	H11
D09	B6	B6	G1
D08	A6	A6	F10
D07	B7	B7	F11
D06	A7	A7	E10
D05	B8	B8	E11
D04	A8	A8	D10
D03	B9	B9	D11
D02	A9	A9	C10
D01	E10	B10	C11
D00	A10	A10	B11

Clock

Y0	D10	D10	B8
Y1	C11	C11	A9
CLK	C10	C10	B9

Interrupt
SIGNAL

CMOS/SOS

BULK CMOS

TC BULK CMOS

IRQ	L8	L8	D2
NMI	K8	K8	C1
RST	L9	L9	C2

Control

BR	J11	J11	B3
BG	E11	E11	A7
XRQ	K10	K10	A3
XAK	F10	F10	B6
XER	F11	F11	A6
R/W	J10	J10	A4
S0	D11	D11	A8
S1	E10	E10	B7
OE	---	H10	B4

Monitor

E/U	G10	G10	B5
C/D	G11	G11	A5

Test

SOUT	L7	L7	---
L/S	K7	K7	---
SCLK	K9	K9	---
SIN	---	L10	---
HLD	K11	---	---

HB	H11	---	---
LB	H10	---	---
No connects			
NC	F1	H11	E2
	L2	---	---
	L10	---	---
	A5	---	---
Alignment			
NC	---	---	C3

APPENDIX B

Instruction Set Timing Estimates

The following list, adapted from a listing created by a computer program written by N.M. Mykris⁵, shows the calculated time of execution for the AAMP's entire instruction set. Variable-length instructions are indicated by an equation showing how to calculate time of execution. For these variable-length instructions, the number of alignments (A) and normalizations (N) is dependant upon the data while the number of shifts (S) and the length (L) of the segment shifted is determined by the programmer. Typical values for these four variables are indicated with each listing. Time of execution for LOCNL is dependant upon the nesting level of called subroutines from the main routine. Time of execution for RETURN depends upon the number of arguments returned from the routine.

The left-hand column of the listing shows the percentage of each command used to attain a standard Gibson benchmark. Throughput per MHz for the specified mix is shown at the end of the list.

Table B.1 Timing Estimates for Instruction Set

	ABS	"50",	Time = 13.0 cyc	
	ABSD	"DC",	Time = 15.0 cyc	
	ABSF	"DE",	Time = 5.5 cyc	
	ABSFE	"AE",	Time = 9.0 cyc	
Mix = 6.10%,	ADD	"E4",	Time = 9.0 cyc	
	ADDD	"80",	Time = 13.0 cyc	
Mix = 6.90%,	ADDF	"84",	Time = 153.0 cyc	A=6, N=1
			Time = 125 + 4*(A + N) cyc	
	ADDFE	"92",	Time = 229.0 cyc	A=6, N=1
			Time = 173 + 8*(A + N) cyc	
Mix = 1.60%,	AND	"E8",	Time = 5.5 cyc	
	ARS	"B8",	Time = 49.0 cyc	S = 8
			Time = 17 + 4 * S cyc	
	ASNBX	"A4",	Time = 43.0 cyc	
	ASND	"A8",	Time = 14.0 cyc	
	ASNDC	"A9",	Time = 19.5 cyc	
	ASNDI	"F6",	Time = 25.0 cyc	
	ASNDL	" C",	Time = 14.0 cyc	
	ASNDLE	"F7",	Time = 19.5 cyc	
	ASNDU	"8B",	Time = 26.0 cyc	
	ASNDX	"8C",	Time = 22.0 cyc	
	ASNDXI	"AA",	Time = 25.0 cyc	
	ASNF	"9E",	Time = 108.5 cyc	L=8, S=8
			Time = 108.5 + 4*(S - L) cyc	
	ASNS	"D3",	Time = 10.0 cyc	
	ASNSC	"D4",	Time = 15.5 cyc	
	ASNSI	"54",	Time = 21.0 cyc	
	ASNSL	" 4",	Time = 10.0 cyc	
	ASNSLE	"5C",	Time = 15.5 cyc	
	ASNSU	"A7",	Time = 22.0 cyc	
	ASNSX	"A6",	Time = 14.0 cyc	
	ASNSXI	"D5",	Time = 21.0 cyc	
	ASNT	"98",	Time = 18.0 cyc	
	ASNTC	"99",	Time = 23.5 cyc	
	ASNTI	"B6",	Time = 29.0 cyc	
	ASNTLE	"B5",	Time = 23.5 cyc	
	ASNTU	"9A",	Time = 44.0 cyc	
	ASNTX	"9B",	Time = 44.0 cyc	
	ASNTXI	"9C",	Time = 33.0 cyc	
	CALL	"5D",	Time = 68.5 cyc	
	CALLI	"23",	Time = 75.5 cyc	
	CALLP	"5E",	Time = 81.0 cyc	
	CALLPI	"1F",	Time = 88.0 cyc	

	CALLU	"64",	Time = 68.5 cyc	
	CVTBIT	"F8",	Time = 13.0 cyc	
	CVTDF	"D9",	Time =137.0 cyc	N = 15
			Time = 77 + 4 * N cyc	
	CVTDFE	"6C",	Time =225.0 cyc	N = 15
			Time = 105 * 4 * N cyc	
	CVTDS	"DA",	Time = 13.5 cyc	
	CVTFD	"DB",	Time =113.5 cyc	A = 15
			Time = 53.5 + 4 * A cyc	
	CVTFED	"AF",	Time =113.5 cyc	A = 15
			Time = 53.5 + 4 * A cyc	
	CVTFEF	"B4",	Time = 49.0 cyc	
	CVTFE	"6D",	Time = 13.0 cyc	
	CVTSD	"65",	Time = 9.0 cyc	
	DECS	"7F",	Time = 20.0 cyc	
	DECSI	"7E",	Time = 31.0 cyc	
	DECSLE	"7D",	Time = 25.5 cyc	
Mix = 0.20%,	DIV	"FA",	Time =109.0 cyc	
	DIVD	"97",	Time =313.0 cyc	
Mix = 1.50%,	DIVF	"87",	Time =313.0 cyc	N = 1
			Time = 309 + 4 * N cyc	
	DIVFE	"95",	Time =706.0 cyc	N = 1
			Time = 698 + 8 * N cyc	
	DIVI	"E7",	Time =109.0 cyc	
	DIVID	"83",	Time =317.0 cyc	
	DO	"8F",	Time = 45.0 cyc	
	DUP	"6A",	Time = 5.5 cyc	
	DUPD	"6B",	Time = 9.0 cyc	
	DUPT	"79",	Time = 49.5 cyc	
	ENDO	"9F",	Time = 41.0 cyc	
	EQ	"EB",	Time = 13.0 cyc	
	EQD	"88",	Time = 15.0 cyc	
	EQT	"90",	Time = 29.5 cyc	
Mix = 5.30%,	EXCH	"ED",	Time = 13.0 cyc	
	EXCHD	"8D",	Time = 25.0 cyc	
	EXCHT	"9D",	Time = 47.0 cyc	
Mix = 3.80%,	GR	"EC",	Time = 13.0 cyc	
	GRD	"89",	Time = 17.0 cyc	
	GRF	"8A",	Time = 49.0 cyc	
	GRFE	"91",	Time = 53.0 cyc	
	HIGHER	"F5",	Time = 13.0 cyc	
	INCS	"7C",	Time = 20.0 cyc	
	INCSI	"7B",	Time = 31.0 cyc	
	INCSLE	"7A",	Time = 25.5 cyc	

	INSERT	"8E",	Time = 93.0 cyc	L=8,S=8
			Time = 93 + 4*(S - L) cyc	
	INTE	"1B",	Time = 5.5 cyc	
	LIT16	"1A",	Time = 16.5 cyc	
	LIT24	"24",	Time = 22.0 cyc	
	LIT32	"25",	Time = 27.5 cyc	
	LIT4A	" 1",	Time = 5.5 cyc	
	LIT4B	" 2",	Time = 5.5 cyc	
	LIT48	"26",	Time = 68.0 cyc	
	LIT8	"18",	Time = 11.0 cyc	
	LIT8N	"19",	Time = 11.0 cyc	
	LITD0	"27",	Time = 9.0 cyc	
	LOCL	"53",	Time = 5.5 cyc	
	LOCNL	"D2",	Time = 48.5 cyc	1 stack
			Time = 24.5 + 24*(Number of stacks) cyc	
	LOCU	"66",	Time = 9.0 cyc	
	LOCX	"FF",	Time = 5.5 cyc	
Mix = 0.60%,	MPY	"F9",	Time = 93.0 cyc	
	MPYD	"96",	Time = 301.0 cyc	
Mix = 3.80%,	MPYF	"86",	Time = 293.0 cyc	N = 1
			Time = 289 + 4 * N cyc	
	MPYFE	"94",	Time = 539.0 cyc	N = 1
			Time = 531 + 8 * N cyc	
	MPYI	"E6",	Time = 93.0 cyc	
	MPYID	"82",	Time = 301.0 cyc	
	NEG	"51",	Time = 9.0 cyc	
	NEGD	"DD",	Time = 13.0 cyc	
	NEGF	"DF",	Time = 13.0 cyc	
	NEGFE	"AD",	Time = 17.0 cyc	
	NOP	"20",	Time = 5.5 cyc	
	NOT	"F4",	Time = 5.5 cyc	
	OR	"E9",	Time = 5.5 cyc	
	POP	"52",	Time = 5.5 cyc	
	POPD	"B7",	Time = 9.0 cyc	
	REFBX	"D1",	Time = 42.5 cyc	
	REFD	"67",	Time = 18.0 cyc	
	REFDC	"68",	Time = 23.5 cyc	
	REFDI	"21",	Time = 29.0 cyc	
	REFDL	" 3",	Time = 18.0 cyc	
	REFDLE	"22",	Time = 23.5 cyc	
	REFDU	"D6",	Time = 29.0 cyc	
	REFDX	"D7",	Time = 26.0 cyc	
	REFDXI	"69",	Time = 29.0 cyc	
	REFS	"55",	Time = 12.0 cyc	
	REFSC	"56",	Time = 17.5 cyc	
	REFSI	"1C",	Time = 23.0 cyc	
Mix = 31.20%,	REFSL	" 0",	Time = 12.0 cyc	

	REFSLE	"1E",	Time = 17.5 cyc	
	REFSU	"D8",	Time = 23.0 cyc	
Mix = 18.00%,	REFSX	"D0",	Time = 16.0 cyc	
	REFSXI	"57",	Time = 23.0 cyc	
	REFT	"75",	Time = 57.5 cyc	
	REFTC	"76",	Time = 59.0 cyc	
	REFTI	"74",	Time = 93.5 cyc	
	REFTLE	"77",	Time = 84.0 cyc	
	REFTU	"78",	Time = 36.0 cyc	
	REFTX	"6F",	Time = 40.5 cyc	
	REFTXI	"6E",	Time = 79.5 cyc	
	RETURN	"5F",	Time = 87.0 cyc	1 Arg
			Time = 63 + 24*(Number of Args)	cyc
Mix = 4.40%,	SHIFT	"FB",	Time = 53.0 cyc	S = 8
			Time = 21 + 4 * S	cyc
	SHIFTL	"FD",	Time = 49.0 cyc	S = 8
			Time = 17 + 4 * S	cyc
	SHIFTR	"FC",	Time = 49.0 cyc	S = 8
			Time = 17 + 4 * s	cyc
Mix = 16.60%,	SKIP	"59",	Time = 10.0 cyc	
	SKIPI	"1D",	Time = 12.0 cyc	
	SKIPNZ	"EF",	Time = 16.0 cyc	
	SKIPNZI	"5B",	Time = 17.0 cyc	
	SKIPZ	"EE",	Time = 16.0 cyc	
	SKIPZI	"5A",	Time = 17.0 cyc	
	SUB	"E5",	Time = 9.0 cyc	
	SUBD	"81",	Time = 13.0 cyc	
	SUBF	"85",	Time = 161.0 cyc	A=6, N=1
			Time = 133 + 4*(A + N)	cyc
	SUBFE	"93",	Time = 237.0 cyc	A=6, N=1
			Time = 181 + 8*(A + N)	cyc
	SWAPSU	"AB",	Time = 28.0 cyc	
	XOR	"EA",	Time = 5.5 cyc	
	XTRACT	"AC",	Time = 85.0 cyc	L=8, S=8
			Time = 85 + 4*(S - L)	cyc

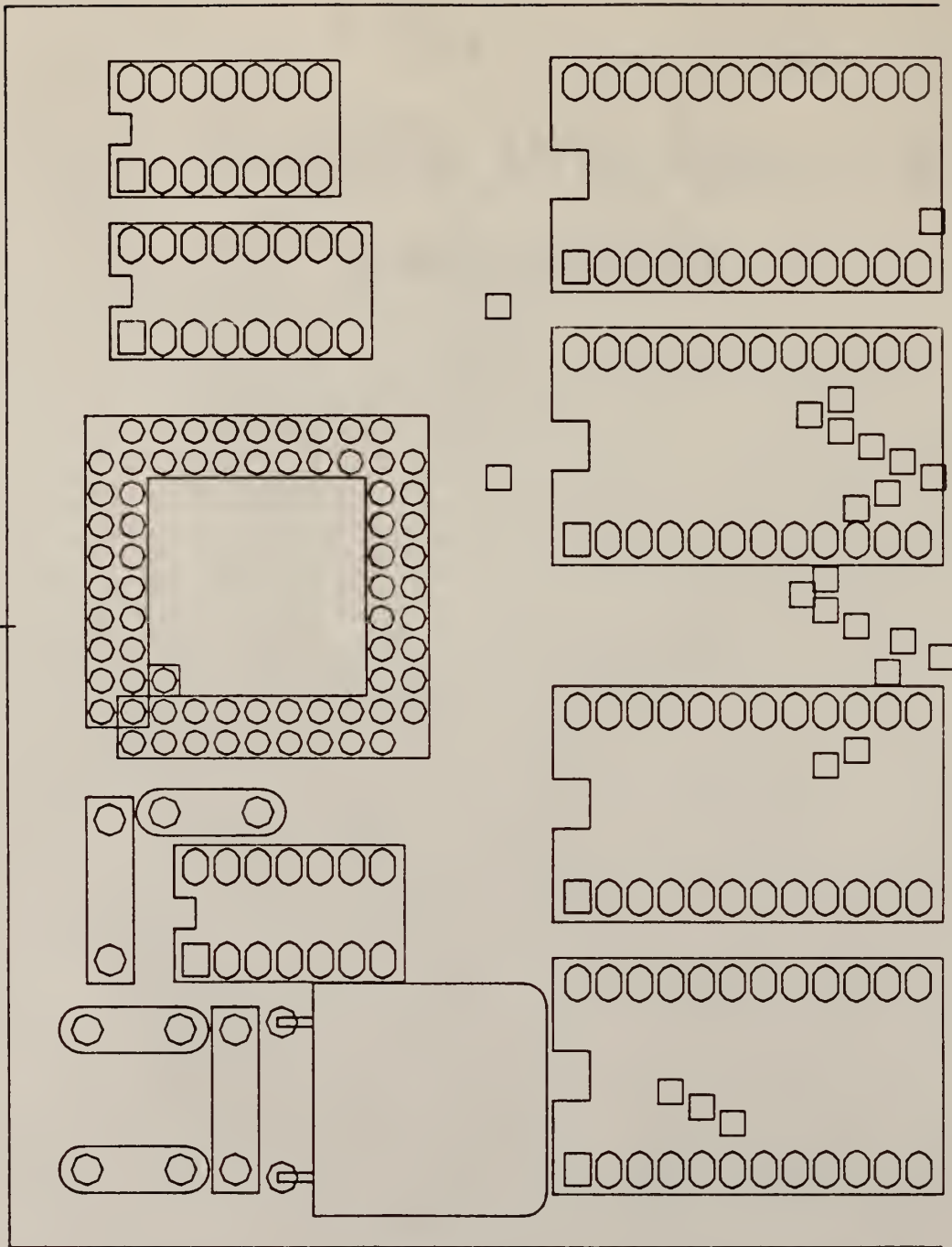
Instruction throughput based on the given percentages:

25.254 KOPS

Appendix C

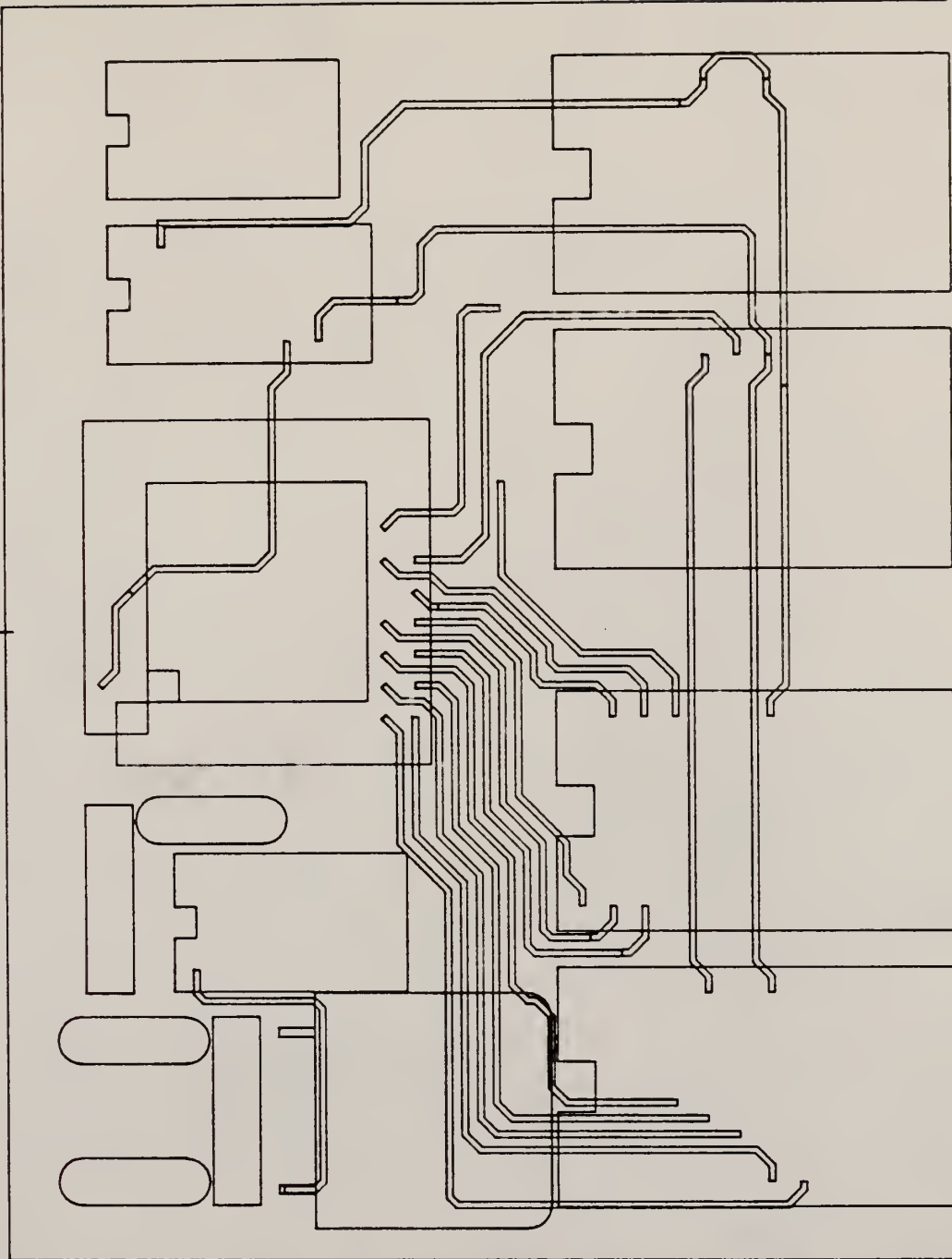
The printed circuit layout shown in this section was created by Armando Corrales and Jim Heise on an HP9845C computer using the Engineering Graphics System (EGS). Specifications for board connections were not known at the time the board was laid out so traces simply lead to the edge of the board and stop. The design uses the new Top-Cavity PGA pin assignment for the AAMP.

Not included on this board is the input buffer section which may or may not be needed in a final system. This design also differs from the prototype circuit in that the Flip-Flop used is the MM74HC74 which, as pointed out in the power consumption section, was found to use slightly more power than the MM74HC174. Room is, however, available for replacing this 14-pin chip with the 16-pin chip if deemed necessary.



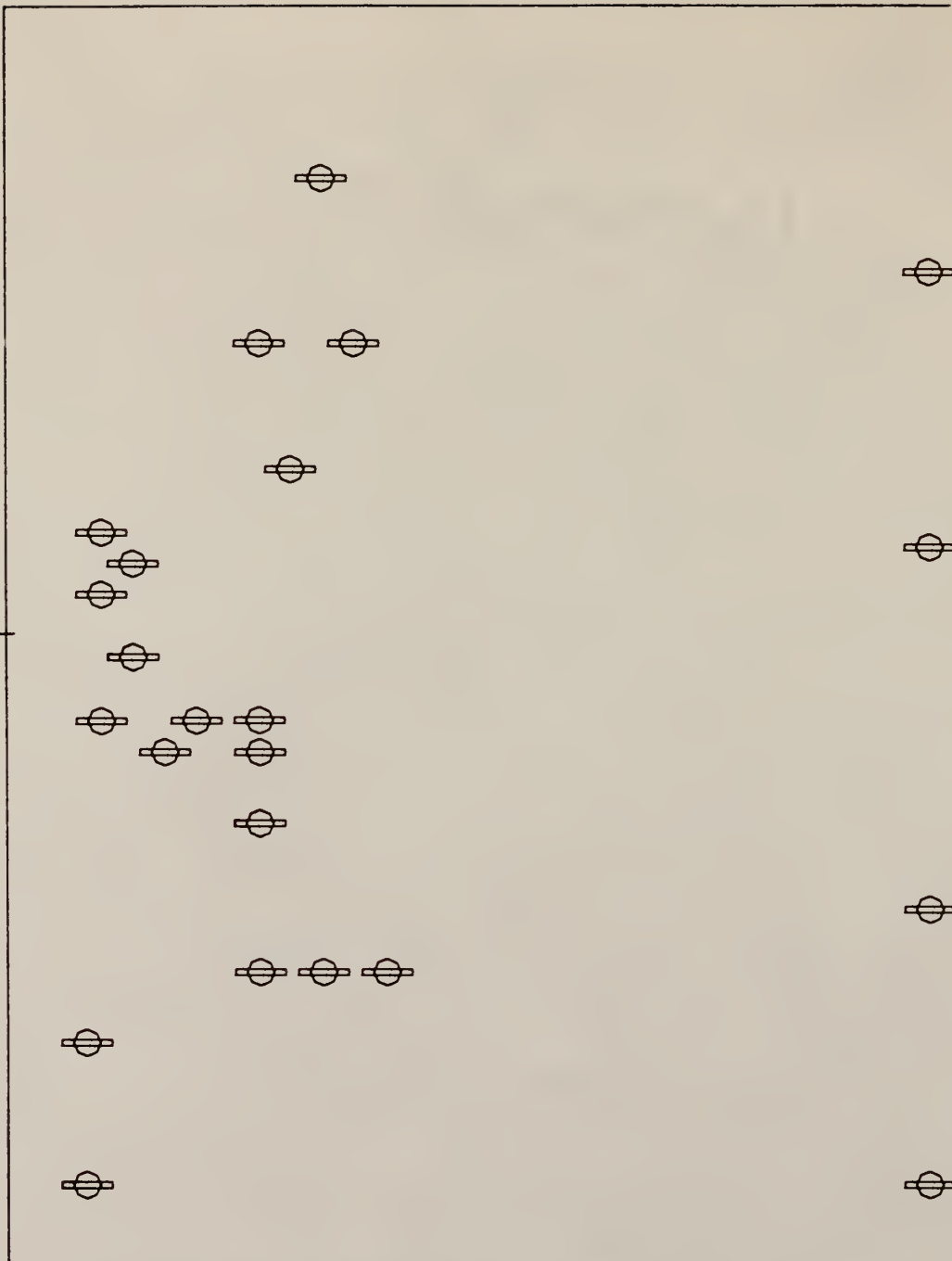
KANSAS STATE UNIVERSITY
 DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
 MINIMAL COMPONENT AMP BOARD
 ARMANDO CORRALES SEPTEMBER 8, 1986 SCALE 1.75/1

Figure C.1 Assembly Drawing



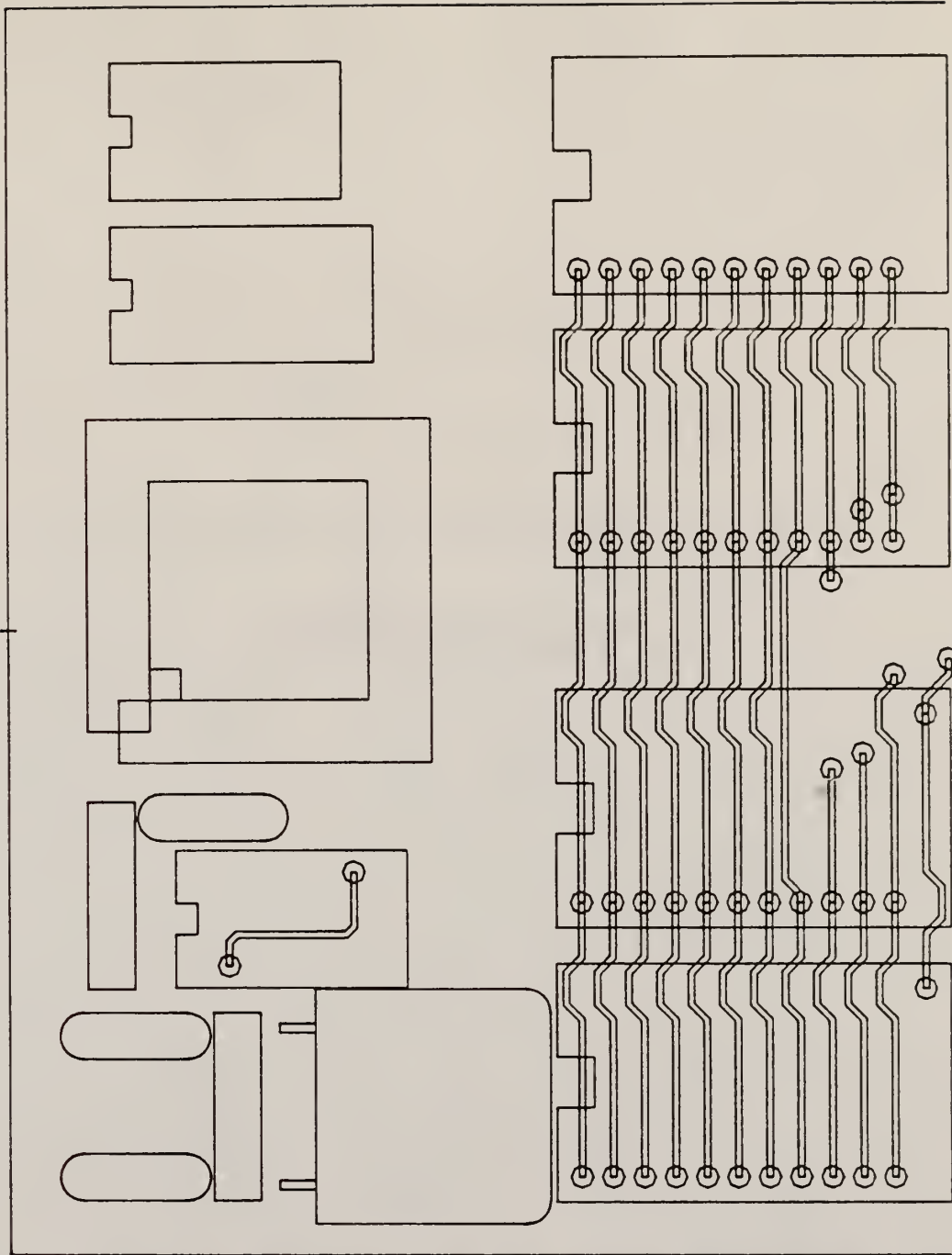
KANSAS STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
MINIMAL COMPONENT AMP BOARD
ARMANDO CORRALES SEPTEMBER 8, 1986 SCALE 1.75/1

Figure C.2 Layer 1 -Component Side



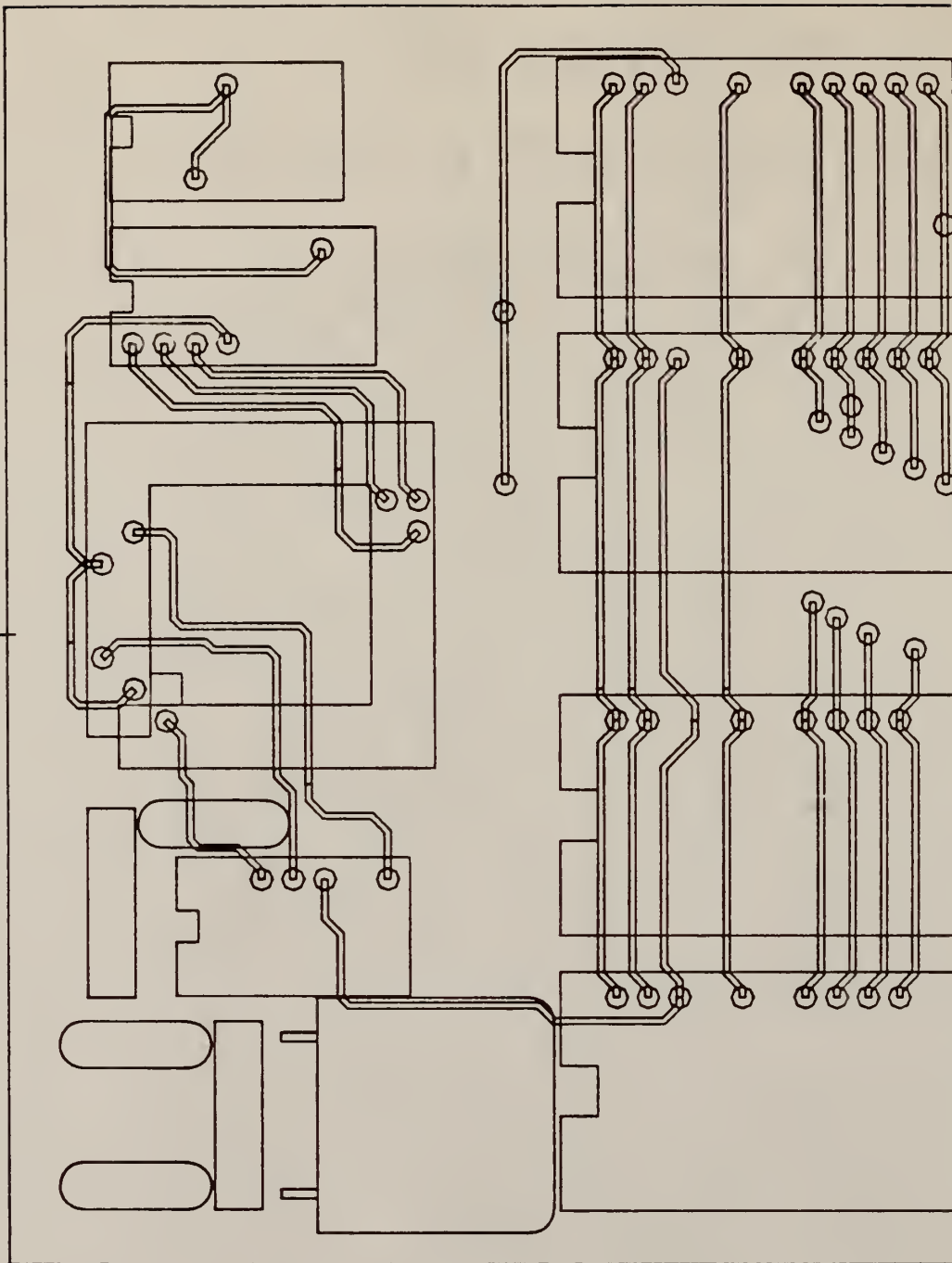
KANSAS STATE UNIVERSITY
 DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
 MINIMAL COMPONENT AMP BOARD
 ARMANDO CORRALES SEPTEMBER 8, 1986 SCALE 1.75/1

Figure C.3 Layer 2 -Ground Plane



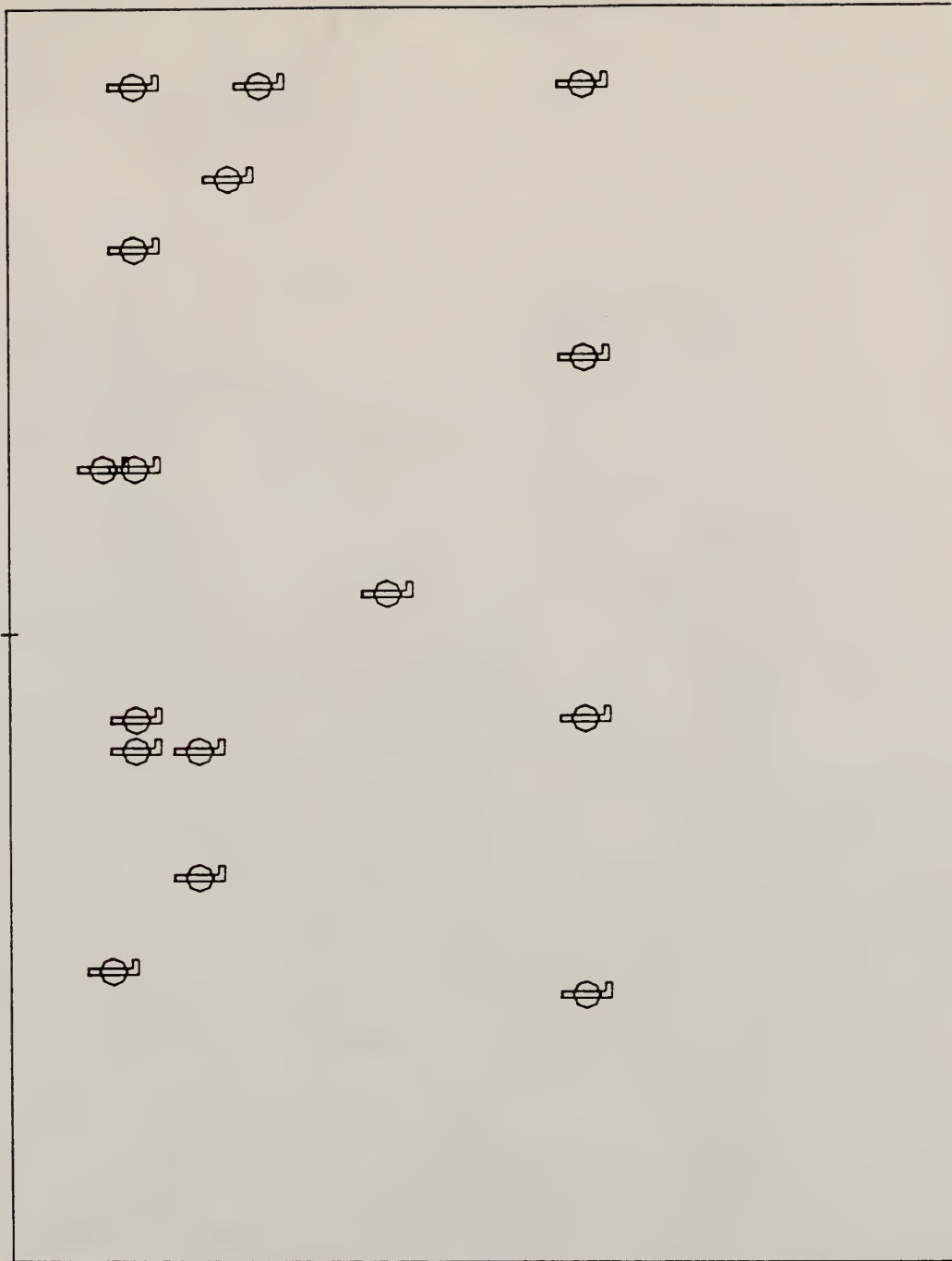
KANSAS STATE UNIVERSITY
 DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
 MINIMAL COMPONENT AMP BOARD
 ARMANDO CORRALES SEPTEMBER 8, 1986 SCALE 1.75/1

Figure C.4 Layer 3



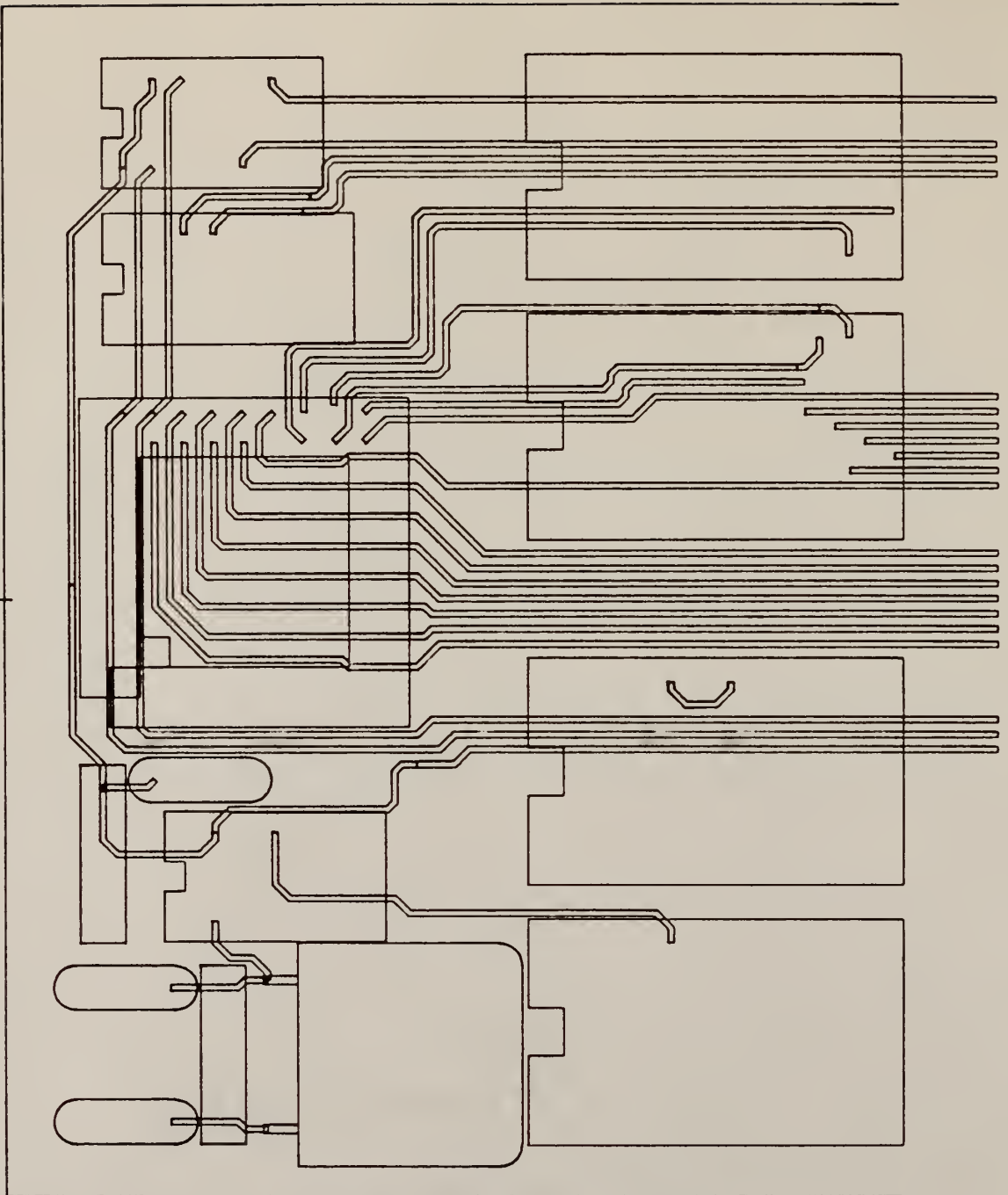
KANSAS STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
MINIMAL COMPONENT AMP BOARD
ARMANDO CORRALES SEPTEMBER 8, 1986 SCALE 1.75/1

Figure C.5 Layer 4



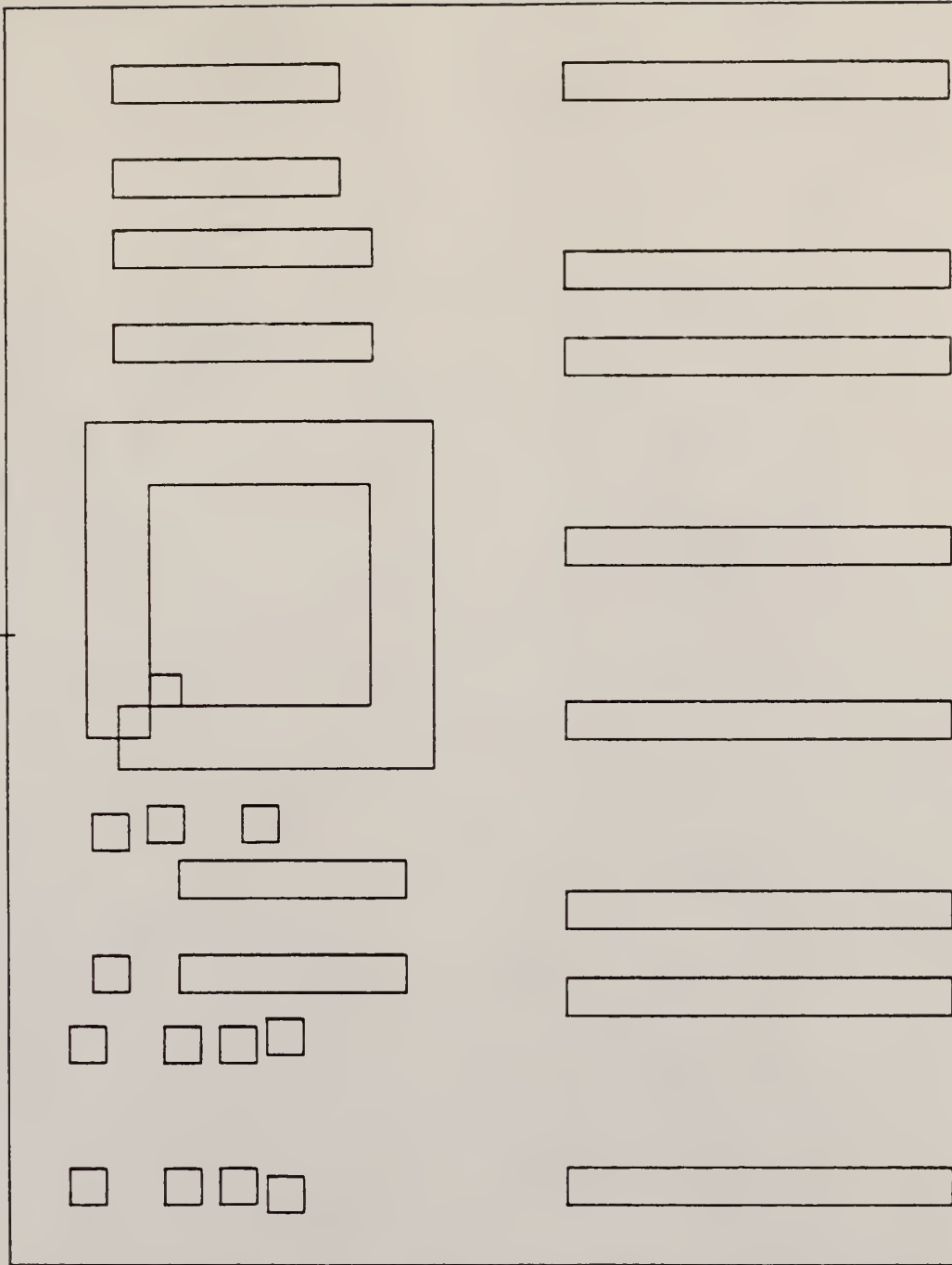
KANSAS STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
MINIMAL COMPONENT AMP BOARD
ARMANDO CORRALES SEPTEMBER 8, 1986 SCALE 1.75/1

Figure C.6 Layer 5 -Ground Plane



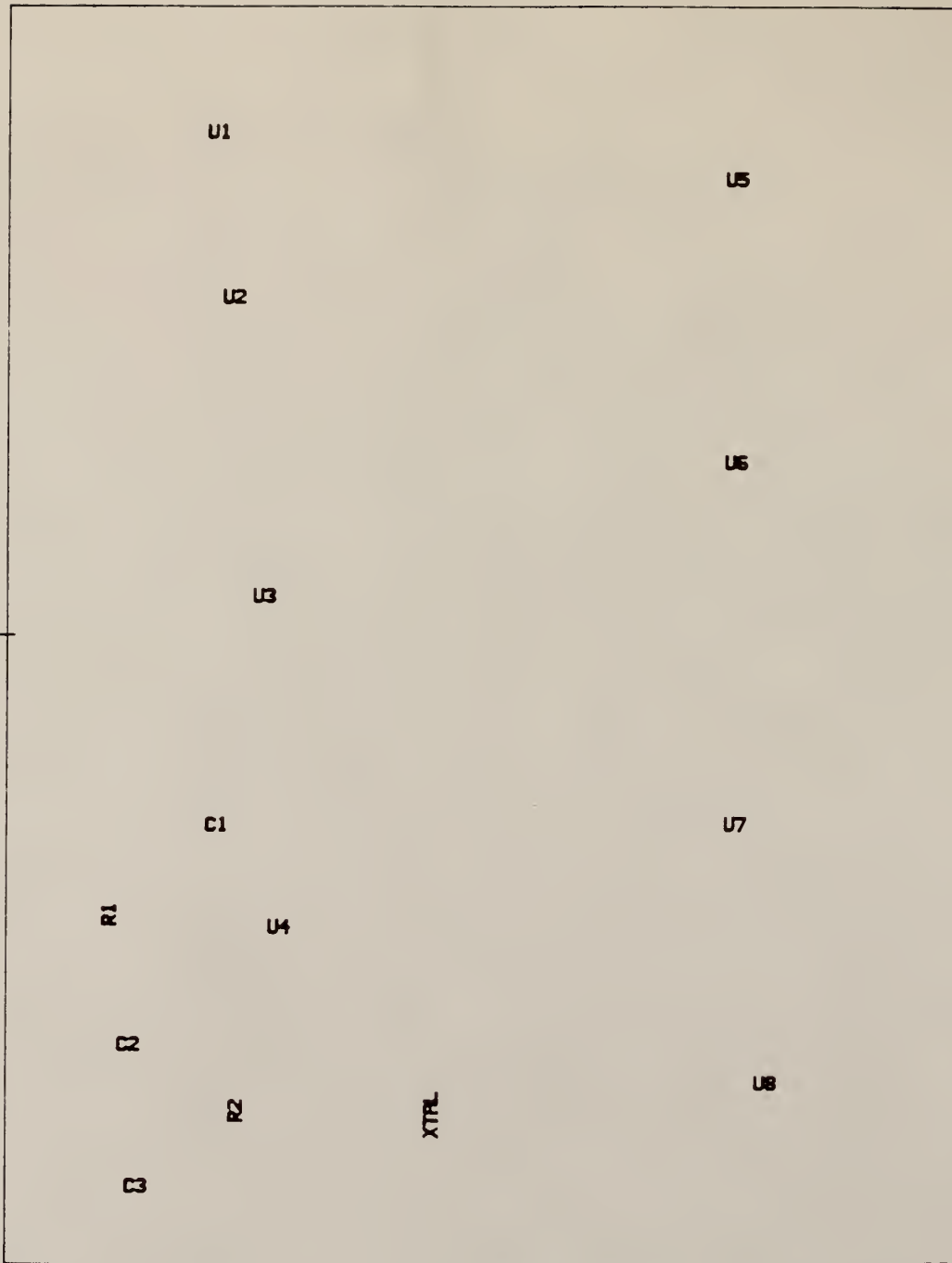
KANSAS STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
MINIMAL COMPONENT AMP BOARD
ARMANDO CORRALES SEPTEMBER 8, 1986 SCALE 1.75/1

Layer C.7 Layer 6 -Circuit Side



KANSAS STATE UNIVERSITY
 DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
 MINIMAL COMPONENT RAMP BOARD
 ARMANDO CORRALES SEPTEMBER 8, 1986 SCALE 1.75/1

Figure C.8 Keep-out Areas For Power and Ground Planes



KANSAS STATE UNIVERSITY
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
MINIMAL COMPONENT AMP BOARD
ARMANDO CORRALES SEPTEMBER 8, 1986 SCALE 1.75/1

Figure C.9 Silkscreened Text on Component Side

APPENDIX D

The following sub-programs were combined in an order which allowed results to be monitored to confirm proper program execution. The fixed precision listings show the program as it was written for testing. Other tested code was substituted into this original order of execution.

To make subprograms interchangeable, a filler opcode was placed in the final high byte of code when the sub-program consisted of an odd number of bytes. This also made it possible to differentiate between blocks of code during time measurements. For the same reason mentioned with the DUP command in the Architecture section, the use of NOP's was avoided and instead the INTE command, which does not cause stack thrashing, was used. Timing estimate and observation figures were altered to exclude these filler commands.

Data types for fixed-point implementations are shown in program headers. The values of S, L, and R in the notation (S / L / R) represent the presence or absence of a sign bit, the number of bits left of the decimal point and the number of bits right of the decimal respectively.

D.1 Fractional 16-bit precision - loop coding

Executive Entry Table

\$0000	00	00	Cont. Status pointer		
\$0001	71	00	Init. Exec Stack limit		
\$0002	74	FF	Init. Exec Top of Stack		
\$0003	00	40	Init. Exec PROCID		
\$0004	00	00	bus error PROCID	none	used
\$0005	00	00	NMI PROCID	"	"
\$0006	00	00	INT PROCID	"	"
\$0007	00	00	Trap PROCID	"	"
\$0008	00	00	Exception PROCID	"	"

Local variables

I	- Lenv(1)	
Yquick	- Lenv(2)	
x^2	- Lenv(3,4)	
THETA	- Lenv(5)	
$v^{(m-1)}$	- Lenv(6)	
y^2	- Lenv(7,8)	
$x(0)$	- Lenv(10)	input buffer 16 long
.	.	
.	.	
$x(F)$	- Lenv(1F)	
$a(0)$	- Lenv(20)	coefficient table
.	.	
.	.	
$a(F)$	- Lenv(2F)	
$y(0)$	- Lenv(30)	output buffer 16 long
.	.	
.	.	
$y(F)$	- Lenv(3F)	

Initial Coefficient Table
[Band-pass filter]

\$0010	01	7B	a(0)	=	.01155124
\$0011	09	3F	a(1)	=	.07222172
\$0012	09	9D	a(2)	=	.07476273
\$0013	FA	1B	a(3)	=-	.04603866
\$0014	E8	54	a(4)	=-	.18493122
\$0015	EA	2F	a(5)	=-	.17043359
\$0016	03	00	a(6)	=	.02344914
\$0017	1C	16	a(7)	=	.21941864
\$0018	E3	EA	a(8)	=-	.21941864
\$0019	FD	00	a(9)	=-	.02344914
\$001A	15	D1	a(A)	=	.17043359
\$001B	17	A1	a(B)	=	.18493122
\$001C	05	EA	a(C)	=	.04603866
\$001D	F6	6E	a(D)	=-	.07476273
\$001E	F6	C2	a(E)	=-	.07222172
\$001F	FE	86	a(F)	=-	.01155124

This block of code copies the initial coefficients into the Local Environment for efficient access in the FIR Filter subprogram. Data is assumed to be in the data format used in the Filter routine

This is the first executable code of the program, therefore, immediately after invocation of the program, the executive stack mark, consisting of the program counter (PC), the Code environment (CENV), the Procedure identifier (PROCID), and the Local Environment pointer (LENV), is copied into the four memory locations immediately above the start of the Local Environment.

\$0020 00 32

procedure header (# local vars)

Block Move

address						# cycles
\$0021	10 18	1810	LIT8	16	I = 16	11
L1		11	LIT4A.1			5.5
\$0022	E5 11	E5	SUB		I = I - 1	9
		41	ASNSL.1		save I	10
\$0023	01 41	01	REFSL.1			12
\$0024	10 56	56 10	REFSC	\$10	get table(I)	17.5
		01	REFSL.1			12
\$0025	53 01	53	LOCL			5.5
\$0026	20 D4	D4 20	ASNSC	\$20	store a(I)	15.5
		01	REFSL.1			12
\$0027	01 01	01	REFSL.1			12
\$0028	0E 19	19 0E	LIT8N	0E		11
		EF	SKIPNZ	L1	I = 0 ?	16
\$0029	52 EF	52	POP		kill old counter	5.5

!* Time of execution *!

	Calculated	Observed	Error %
Inside loop	138	144	- 4%
Set-up	16.5		
Total	2225	2624	-15%

input x(0)

This section of code inputs data from input channel one and stores it to x(0)

					#cycles	
\$002A	00 1C	1C 1000	REFSI	\$1000	get input	23
\$002B	5C 10	5C 10	ASNSLE	\$10	store to x(0)	15.5
\$002C	1B 10	1B	INTE			---

!* Time of execution *!

	Calculated	Observed	Error %
Time for input	38.5	38.5	0%

FIR Filter

This block of code performs an FIR filter on the array of input data and writes the result into the Local Environment. Data and coefficients are in the (1 / 0 / 15) data format.

$$Y = \text{SUM } a(i) * x(i) \quad i = 1 \dots 16$$

address						#cycles
		10	LIT4A.0	Yquick = 0		5.5
\$002D	42 10	42	ASNSL.2			10
\$002E	10 18	18 10	LIT8 16	I = 16		11
L2		11	LIT4A.1			5.5
\$002F	E5 11	E5	SUB	I = I - 1		9
		41	ASNSL.1			10
\$0030	01 41	01	REFSL.1			12
		53	LOCL			5.5
\$0031	56 53	56 10	REFSC \$10	get x(I)		17.5
\$0032	01 10	01	REFSL.1			12
		53	LOCL			5.5
\$0033	56 53	56 20	REFSC \$20	get a(I)		17.5
\$0034	F9 20	F9	MPY	a(I) * x(I)		93
		02	REFSL.2			12
\$0035	E4 02	E4	ADD			9
		42	ASNSL.2	add to y		10
\$0036	01 42	01	REFSL.1			12
		01	REFSL.1			12
\$0037	19 01	19 13	LIT8N 13			11
\$0038	EF 13	EF	SKIPNZ L2	I = 0 ?		16
		52	POP	kill old counter		5.5
\$0039	1B 52	1B	INTE			---

!* Time of execution *

	Calculated	Observed	Error %
Inside loop	269.5	224	20%
Set-up	32		
Total	4344	3701	17%

Filter Update

This block of code moves the array of time delayed input data to one sample greater delay and leaves x(0) empty for the next input of data. Data is maintained in the (1 / 0 / 15) format.

x(i + 1) = x(i) i = 1 . . 15

							#cycles
			2F		LIT4B.F	I = 15	5.5
LU	\$003A	11	2F	11	LIT4A.1		5.5
				E5	SUB	I = I - 1	9
	\$003B	41	E5	41	ASNSL.1	save I	10
				01	REFSL.1		12
	\$003C	53	01	53	LOCL		5.5
	\$003D	10	56	56	REFSC \$10	get x(i)	17.5
				01	REFSL.1		12
	\$003E	53	01	53	LOCL		5.5
	\$003F	11	D4	D4	ASNSC \$11	save x(i + 1)	15.5
				01	REFSL.1		12
	\$0040	01	01	01	REFSL.1		12
	\$0041	0F	19	19	OF	LIT8N OF	11
				EF	SKIPNZ LU	I = 0 ?	16
	\$0042	52	EF	52	POP	kill old counter	5.5

!* Time of execution *!

	Calculated	Observed	Error %
Inside loop	143.5	156	- 8%
Set-up	11		
Total	2163.5	2585	-16%

Output buffer Update

This section of code shifts the array of time delayed output data to one sample greater delay. Data type is maintained.

$$y(i + 1) = y(i) \quad y(0) = Y_{quick} \quad i = 1 \dots 15$$

						#cycles
		02		REFSL.2		12
\$0043	5C	02	5C 30	ASNSLE \$30	y(0) = Yquick	15.5
\$0044	2F	30	2F	LIT4B.F	I = 15	5.5
LO						
		11		LIT4A.1		5.5
\$0045	E5	11	E5	SUB	I = I - 1	9
			41	ASNSL.1	save I	10
\$0046	01	41	01	REFSL.1		12
			53	LOCL		5.5
\$0047	56	53	56 30	REFSC \$30	get y(i)	17.5
\$0048	01	30	01	REFSL.1		12
			53	LOCL		5.5
\$0049	D4	53	D4 31	ASNSC \$31	save y(i + 1)	15.5
\$004A	01	11	01	REFSL.1		12
			01	REFSL.1		12
\$004B	19	01	19 OF	LIT8N OF		11
\$004C	EF	OF	EF	SKIPNZ LO	I = 0 ?	16
			52	POP	kill old counter	5.5
\$004D	1B	52	1B	INTE		---

!* Time of execution *!

	Calculated
Inside loop	143.5
Set-up	38.5
Total	2334.5

Ratio Calculation

$$r = x^2/y^2$$

To make r comparable to THETA (0/8/8) in the decision section, x is shifted 4 right prior to squaring. The full 32 bits of the intermediate result are maintained through divide with a single precision value left on the stack upon exit.

Overflow will occur if outside range

$$x^2/256 \leq y^2 \leq 256 x^2$$

				(0/8/8) set-up			
address						#cycles	
						ls word	
\$004E	10	1E	1E	10	REFSLE \$10	get x(0)	17.5
				10	LIT4A.0		5.5
\$004F	2C	10	2C		LIT4B.C	abcd.....mnop	5.5
\$0050	0C	18	18	0C	LIT8 0C		11
			8E		INSERT	mnop000...000	93
						ms word	
\$0051	1E	8E	1E	10	REFSLE \$10	get x(0)	17.5
\$0052	14	10	14		LIT4A.4	abcd.....mnop	5.5
			B8		ARS	ssssabcd..jkl	49
\$0053	ED	B8	ED		EXCH	ls-ms order	13

End of (0/8/8) set-up

			6B		DUPD		9
\$0054	96	6B	96		MPYD	x ²	301
			C3		ASNDL.3	store temp	14
\$0055	1E	C3	1E	30	REFSLE \$30	get y(0)	17.5
\$0056	10	30	10		LIT4A.0	fractional CVTSD	5.5
			6B		DUPD	y ²	9
\$0057	96	6B	96		MPYD		301
			33		REFDL.3	get x ² back	18
\$0058	8D	33	8D		EXCHD		25
			97		DIVD		313
\$0059	52	97	52		POP	fractional CVTDS	5.5

	!* Time of execution	!* Error %
	Calculated	Observed
(0/8/8) set-up	217.5	220
time of execution	1236	1132
		1%
		-10%

The same (0/8/8) set-up could be used to shift THETA in the decision with the shift value then equal to 8 instead of 4.

Decision

The following section compares the result of the ratio calculation with a threshold, placing a boolean flag indicating T or F on the top of the stack.

$$d = \begin{cases} 1 & r > \text{THETA} \\ 0 & r < \text{THETA} \end{cases}$$

			#cycles	
	05	REFSL.5	get THETA	12
\$005A	ED 05	ED	EXCH	13
	EC	GR	T if THETA > r	13
\$005B	11 EC	11	LIT4A.1	5.5
	EA	XOR	T if THETA ≤ r	5.5
\$005C	1B EA	1B	INTE	---

	!* Time of execution	!* Error %
	Calculated	Observed
time of execution	49	54.5
		-10%

Weight Update

The following section of code uses past output values to adapt the coefficients of an FIR filter. In this case, $a(i)$ is assumed $(1 / 4 / 11)$

$$a(i) = a(i) + da * y(i)$$

							#cycles
	\$005D	10 18	18 10	LIT8	16	I = 16	11
LW			11	LIT4A.1			5.5
	\$005E	E5 11	E5	SUB		I = I - 1	9
			41	ASNSL.1			10
	\$005F	01 41	01	REFSL.1		get y(i)	12
			53	LOCL			5.5
	\$0060	56 53	56 30	REFSC	\$30		17.5
	\$0061	1A 30	1A	dada	LIT16	da	16.5
	\$0062	da da	F9	MPY			93
	\$0063	01 F9	01	REFSL.1		get a(i)	12
			53	LOCL			5.5
	\$0064	56 53	56 20	REFSC	\$20		17.5
	\$0065	E4 20	E4	ADD			9
			01	REFSL.1			12
	\$0066	53 01	53	LOCL		store	5.5
	\$0067	20 D4	D4 20	ASNSC	\$20	a(i)+da*y(i)	15.5
			01	REFSL.1			12
	\$0068	01 01	01	REFSL.1			12
	\$0069	13 19	19 13	LIT8N	13		11
			EF	SKIPNZ	LW		16
	\$006A	52 EF	52	POP			5.5

!* Time of execution *!

	Calculated	Observed	Error %
Inside loop	297	256	13%
Set-up	16.5		
Total	4768.5	4483.5	6%

IIR Filter

$$v(m) = (1-BETA) * v(m-1) + BETA * x * x$$

Assume BETA fixed and can be referenced immediate.
 The temporary value x^2 was calculated in RATIO CALCULATION section. 116 microseconds could be saved by recalling the value and not recalculating.

						#cycles	
\$0069	10	1E	1E	10	REFSLE \$10	get x(0)	17.5
\$006A	10	1E	1E	10	REFSLE \$10		17.5
			F9		MPY	x^2	93

or replacing above code

			04		REFSL.4	get top 16 bits of 32-bit x^2	12
\$006B	1A	F9	1A	3E68	LIT16 BETA	BETA = .98	16.5
\$006C	3E	68	F9		MPY		93
\$006D	06	F9	06		REFSL.6	get v(m-1)	12
\$006E	47	1A	1A	0147	LIT16 (1-BETA)	= .02	16.5
\$006F	F9	01	F9		MPY		93
			E4		ADD		9
\$0070	46	E4	46		ASNSL.6	store v(m)	10

!* Time of execution *!

	Calculated	Observed	Error %
normal data	378	372	2%
input = 0		220	
overflow		401	

D.2 Fractional 16-bit Precision -Inline coding

FIR Filter

This section of code performs an FIR filter on the array of input data and writes the result into the Local Environment. Data and coefficients are in the (1 / 0 / 15) data format.

$$Y = \text{SUM } a(i) * x(i) \quad i = 1 \dots 16$$

							#cycles
\$002D	00 18	18 0	LIT8 00	y = 0			11
\$002E	20 1E	1E 20	REFSLE \$20	get a(0)			17.5
\$002F	10 1E	1E 10	REFSLE \$10	get x(0)			17.5
		F9	MPY				93
\$0030	E4 F9	E4	ADD	y = y + *			9
\$0031	21 1E	1E 21	REFSLE \$21	get a(1)			17.5
\$0032	11 1E	1E 11	REFSLE \$11	get x(1)			17.5
		F9	MPY				93
\$0033	E4 F9	E4	ADD	y = y + *			9

\$005B	2F 1E	1E 2F	REFSLE \$2F	get a(F)			17.5
\$005C	1F 1E	1E 1F	REFSLE \$1F	get x(F)			17.5
		F9	MPY				93
\$005D	E4 F9	E4	ADD	y = y + *			9
\$005E	30 5C	5C 30	ASNSLE \$30	store to y(0)			15.5

	!* Time of execution *		
	Calculated	Observed	Error %
Set-up	26.5	--	---
time per N	137	--	---
16 coefficients	2213	2316	-4%

Filter Update

This block of code moves the array of time delayed input data to one sample greater delay and leaves x(0) empty for the next input of data. Data is maintained in the (1 / 0 / 15) format.

$$x(i + 1) = x(i) \quad i = 1 \dots 15$$

							#cycles
\$005F	1E	1E	1E	1E	REFSLE \$1E	get x(E)	17.5
\$0060	1F	5C	5C	1F	ASNSLE \$1F	store to x(F)	15.5
\$0061	1D	1E	1E	1D	REFSLE \$1D	get x(D)	17.5
\$0062	1E	5C	5C	1E	ASNSLE \$1E	store to x(E)	15.5
		
		
\$0071	10	1E	1E	10	REFSLE \$00	get x(0)	17.5
\$0072	11	5C	5C	11	ASNSLE \$01	store to x(1)	15.5

!* Time of execution *!	Calculated	Observed	Error %
time per N	33	--	---
16 coefficients	495	--	---
(15 iterations)			

Ratio Calculation

$$r = (x * x) / (y * y)$$

This routine retains the full 32 bits after multiplies. Numerator and denominator are then normalized in 8-bit increments until the leading byte is non-zero. The final divide is a full 32 bit by 32 bit fractional divide. Finally, the LSB is discarded leaving the result, a 16-bit unsigned fraction. (0/0/16)

offset						#cycles	
\$0000	10	1E	1E	10	REFSLE \$10	get x	17.5
			10		LIT4A.0	CVTSD fractional	5.5
\$0001	6B	10	6B		DUPD		9
			96		MPYD	x ²	301
\$0002	C3	96	C3		ASNDL.3		14

\$0003	10	1E	1E	10	REFSLE \$30		17.5
				10	LIT4A.0		5.5
\$0004	6B	10	6B		DUPD		9
				96	MPYD	y ²	301
\$0005	C7	96	C7		ASNDL.7		14
				04	REFSL.4	If top word zero	12
\$0006	08	04	08		REFSL.8		12
				E9	OR		5.5
\$0007	5B	E9	5B	1A	SKIPNZI	Over	17

!* initial set-up takes 740.5 cycles *!

\$0008	03	1A	03		REFSL.3	move full word	12
				44	ASNSL.4		10
\$0009	07	44	07		REFSL.7		12
				48	ASNSL.8		10
\$000A	10	48	10		LIT4A.0	clear LSB's	5.5
				43	ASNSL.3		10
\$000B	10	43	10		LIT4A.0		5.5
				47	ASNSL.7		10

!* double shift takes 75 cycles *!

Over

\$000C	04	47	04		REFSL.4	test top byte	12
				08	REFSL.8		12
\$000D	E9	08	E9		OR		5.5
\$000E	00	1A	1A	FF00	LIT16 FF00	mask out bottom	16.5
\$000F	E8	FF	E8		AND		5.5
\$0010	1A	5B	5B	1A	SKIPNZI	Out	17

!* single shift test takes 68.5 cycles *!

\$0011	28	07	28		REFSL.4		12
				FD	LIT4B.8	shift MSB	5.5
					SHIFTL		49
\$0012	13	FD	13		LIT4A.3		5.5
				53	LOCL	get top 8 bits	5.5
\$0013	11	53	11		LIT4A.1	from LSB	5.5
				D1	REFBX		42.5
\$0014	D1	E9	E9		OR	store	5.5
				44	ASNSL.4		10
\$0015	03	44	03		REFSL.3		12
				28	LIT4B.8	shift LSB	5.5

\$0016	FD	28	FD	SHIFTL		49
			43	ASNSL.3		10
\$0017	08	43	08	REFSL.8		12
			28	LIT4B.8	shift MSB	5.5
\$0018	FD	28	FD	SHIFTL		49
			17	LIT4A.7		5.5
\$0019	53	17	53	LOCL	get top 8 bits	5.5
			11	LIT4A.1	from LSB	5.5
\$001A	D1	11	D1	REFBX		42.5
			E9	OR	store	5.5
\$001B	48	E9	48	ASNSL.8		10
			07	REFSL.7		12
\$001C	28	07	28	LIT4B.8	shift LSB	5.5
			FD	SHIFTL		49
\$001D	47	FD	47	ASNSL.7		10

!* single shift takes 435 cycles *!

OUT			33	REFDL.3	get x^2	18
\$001E	37	33	37	REFDL.7	get y^2	18
			97	DIVD	fractional divide	313
\$001F	52	97	52	POP	save top 16 bits	5.5

!* 32-bit divide and memory accesses take 354.4 cycles *!

Calculated time of execution:

Zero Shifts	1163.5
One Shift	1598.5
Two Shifts	1238.5
Three Shifts	1673.5

Weight Update

The following section of code uses past output values to adapt the coefficients of an FIR filter. In this case, $a(i)$ is assumed (1 / 4 / 11)

$$a(i) = a(i) + da * y(i)$$

offset							#cycles
\$0000	30	1E	1E	30	REFSLE \$30	get y(0)	17.5
\$0001	da	1A	1A	dada	LIT16 da		16.5
\$0002	F9	da	F9		MPY	y(0) * da	93
\$0003	20	1E	1E	20	REFSLE \$20	get a(0)	17.5
			E4		ADD	y(0) + da * y(0)	9
\$0004	5C	E4	5C	20	ASNSLE \$20		15.5
\$0005	1E	20	1E	21	REFSLE \$21	get y(1)	17.5
\$0006	1A	21	1A	dada	LIT16 da		16.5
\$0007	da	da	F9		MPY	y(1) * da	93
\$0008	1E	F9	1E	21	REFSLE \$21		17.5
\$0009	E4	21	E4		ADD	y(1) + da * y(1)	9
\$000A	21	5C	5C	21	ASNSLE \$21		15.5
			:		:	:	
			:		:	:	
			:		:	:	
\$004B	1E	2E	1E	2F	REFSLE \$2F	get y(F)	17.5
\$004C	1A	2F	1A	dada	LIT16 da		16.5
\$004D	da	da	F9		MPY	y(F) * da	93
\$004E	1E	F9	1E	2F	REFSLE \$2F		17.5
\$004F	E4	2F	E4		ADD	y(F) + da * y(F)	9
\$0050	2F	5C	5C	2F	ASNSLE \$2F		15.5

!* Time of execution *!			
	Calculated	Observed	Error %
time per N	169	--	---
16 coefficients	2704	--	---

D.3 Standard Precision Floating Point -Loop Coding

Executive Entry Table

\$0000	00	00	Cont. Status pointer		
\$0001	71	00	Init. Exec Stack limit		
\$0002	74	FF	Init. Exec Top of Stack		
\$0003	00	60	Init. Exec PROCID		
\$0004	00	00	bus error PROCID	none	used
\$0005	00	00	NMI PROCID	"	"
\$0006	00	00	INT PROCID	"	"
\$0007	00	00	Trap PROCID	"	"
\$0008	00	00	Exception PROCID	"	"

Local variables

I - Lenv(1)

Yquick - Lenv(2,3)

y² - Lenv(4,5)

THETA - Lenv(6,7)

v(m-1) - Lenv(8,9)

da - Lenv(A,B)

BETA - Lenv(C,D)

1-BETA - Lenv(E,F)

x(0) - Lenv(10,11) input buffer 16 long
.
.
x(F) - Lenv(2E,2F)

```

a(0) - Lenv(30,31)   coefficient table
.
.
a(F) - Lenv(4E,4F)

y(0) - Lenv(50,51)   output buffer 16 long
.
.
y(F) - Lenv(6E,6F)

```

Initial Coefficient Table
[Band-pass filter]

```

$0010 82 D3      a(0) = .01155124
$0011 01 7A

$0012 8F 60      a(1) = .07222172
$0013 09 3E

$0014 D3 3E      a(2) = .07476273
$0015 09 91

$0016 67 B7      a(3) = -.04603866
$0017 FA 1B

$0018 2C 7E      a(4) = -.18493122
$0019 E8 54

$001A 3B 6F      a(5) = -.17043359
$001B EA 2F

$001C 61 A0      a(6) = .02344914
$001D 03 00

$001E E8 F6      a(7) = .21941864
$001F 1C 15

$0020 17 05      a(8) = -.21941864
$0021 E3 EA

$0022 9E 60      a(9) = -.02344914
$0023 FC FF

$0024 C4 91      a(A) = .17043359
$0025 15 D0

```

\$0026	D3	82	a(B) = .18493122
\$0027	17	A6	
\$0028	98	49	a(C) = .04603866
\$0029	05	E4	
\$002A	2C	C2	a(D) = -.07476273
\$002B	F6	6E	
\$002C	70	50	a(E) = -.07222172
\$002D	F6	C1	
\$002E	7D	2D	a(F) = -.01155124
\$002F	FE	85	

This block of code copies the initial coefficients into the Local Environment for efficient access in the FIR Filter subprogram. Data is initially stored in ROM in extended precision Fractional data format and is converted to floating by this routine.

This is the first executable code of the program, therefore, immediately after invocation of the program, the executive stack mark, consisting of the program counter (PC), the Code environment (CENV), the Procedure identifier (PROCID), and the Local Environment pointer (LENV), is copied into the four memory locations immediately above the start of the Local Environment.

\$0030 00 70 procedure header #local variables

Block Move

Address						#cycles
\$0031	20 18	18 20	LIT8 32	I = 32		11
L1		12	LIT4A.2			5.5
\$0032	E5 12	E5	SUB	I = I - 2		9
		41	ASNSL.1	save I		10
\$0033	01 41	01	REFSL.1			12
\$0034	10 68	68 10	REFDC \$10	get table(I)		23.5
		D9	CVTDF			137
\$0035	01 D9	01	REFSL.1			12
		53	LOCL			5.5
\$0036	A9 53	A9 30	ASNDC \$30	store a(I)		19.5
\$0037	01 30	01	REFSL.1			12
		.01	REFSL.1			12
\$0038	19 01	19 0F	LIT8N 0F			11
\$0039	EF 0F	EF	SKIPNZ L1	I = 0 ?		16
		52	POP	kill old counter		5.5
\$003A	1B 52	1B	INTE			---

!* Time of execution *!

	Calculated	Observed	Error %
Inside loop	285	296	- 4%
Set-up	16.5	--	---
Total	4576.5	4444	2%

FIR Filter

This block of code performs an FIR filter on the array of input data and writes the result into the Local Environment. Data and coefficients are in floating-point data format.

Y = SUM a(i) * x(i) i = 1 . . 16

							#cycles
			10		LIT4A.0	Yquick = 0	5.5
\$003B	10	10	10		LIT4A.0		5.5
			C2		ASNDL.2		14
\$003C	18	C2	18	20	LIT8 20	I = 32	11
L2 \$003D	12	20	12		LIT4A.2		5.5
			E5		SUB	I = I - 2	9
\$003E	41	E5	41		ASNSL.1		10
			01		REFSL.1		12
\$003F	53	01	53		LOCL		5.5
\$0040	10	68	68	10	REFDC \$10	get x(I)	23.5
			01		REFSL.1		12
\$0041	53	01	53		LOCL		5.5
\$0042	30	68	68	30	REFDC \$30	get a(I)	23.5
			86		MPYF	a(I) * x(I)	293
\$0043	32	86	32		REFDL.2		18
			84		ADDF		153
\$0044	C2	84	C2		ASNDL.2	add to y	14
			01		REFSL.1		12
\$0045	01	01	01		REFSL.1		12
\$0046	13	19	19	13	LIT8N 13		11
			EF		SKIPNZ L2	I = 0 ?	16
\$0047	52	EF	52		POP	kill old counter	5.5

	!* Time of execution	*!
	Calculated	Observed
Inside loop	635.5	748
Set-up	41.5	
Total	10209.5	12533
		Error %
		-15%
		-19%

Filter Update

This block of code moves the array of time delayed input data to one sample greater delay and leaves x(0) empty for the next input of data.

$$x(i + 1) = x(i) \quad i = 1 \dots 15$$

address						#cycles
\$0048	1E 18	18 1E	LIT8 30	I = 30		11
LU		12	LIT4A.2			5.5
\$0049	E5 12	E5	SUB	I = I - 2		9
		41	ASNSL.1	save I		10
\$004A	01 41	01	REFSL.1			12
		53	LOCL			5.5
\$004B	68 53	68 10	REFDC \$10	get x(i)		23.5
\$004C	01 10	01	REFSL.1			12
		53	LOCL			5.5
\$004D	A9 53	A9 12	ASNDC \$12	save x(i + 2)		19.5
\$004E	01 12	01	REFSL.1			12
		01	REFSL.1			12
\$004F	19 01	19 0F	LIT8N 0F			11
\$0050	EF 0F	EF	SKIPNZ LU	I = 0 ?		16
		52	POP	kill old counter		5.5

Input

The following segment inputs data from the input channel, converts to floating, and stores it as x(0) in the Local Environment Extended.

\$0051	1C 52	1C 1000	REFSI \$1000	get input		23
\$0052	10 00	10	LIT4A.0	fractional CVTSD		5.5
\$0053	D9 10	D9	CVTDF			137
\$0054	10 F7	F7 10	ASNDLE \$10			19.5

!* Time of execution *!

	Calculated	Observed	Error %
Inside loop	153.5	--	---
Set-up	16.5	--	---
Total	2473	2800	-12%
input	185	201	- 8%

Ratio Calculation

$$r = x^2/y^2$$

Assume x and y are stored in Local Environment Extended.
r is left on stack upon completion of segment.

offset						#cycles	
\$0000	50	22	22	50	REFDLE \$50	get y(0)	23.5
					DUPD		9
\$0001	86	6B	86		MPYF		293
					ASNDL.4	store y ² temp	14
\$0002	22	C4	22	10	REFDLE \$10	get y(0)	23.5
\$0003	6B	10	6B		DUPD		9
					MPYF	x ²	293
\$0004	34	86	34		REFDL.4	get y ² back	18
					DIVF		313

Calculated time of execution = 996 cycles

Decision

$$d = \begin{cases} 1 & r > \text{THETA} \\ 0 & r < \text{THETA} \end{cases}$$

d is boolean

Assume THETA stored in Lenv(6,7)
Assume that r was left on stack from RATIO CALCULATION segment.

offset						#cycles	
\$0006	36	87	36		REFDL.6	18	
					EXCHD	25	
\$0007	8A	8D	8A		GRF	49	
					LIT4A.1	toggle flag	5.5
\$0008	EA	11	EA		XOR	5.5	

Calculated time of execution = 103 cycles

Weight Update

The following section of code uses past output values to adapt the coefficients of an FIR filter. Data is in floating-point notation.

$$a(i) = a(i) + da * y(i)$$

Assume da stored in Lenv(A,B)

offset						#cycles
	\$0000	20 18	18 20	LIT8 32	I = 32	11
LW			12	LIT4A.2		5.5
	\$0001	E5 12	E5	SUB	I = I - 2	9
			41	ASNSL.1		10
	\$0002	01 41	01	REFSL.1		12
			53	LOCL		5.5
	\$0003	68 53	68 50	REFDC \$50	get y(i)	23.5
	\$0004	3A 50	3A	REFDL.A		18
			86	MPYF		293
	\$0005	01 F9	01	REFSL.1		12
			53	LOCL		5.5
	\$0006	68 53	68 30	REFDC \$30	get a(i)	23.5
	\$0007	84 30	84	ADDF		153
			01	REFSL.1		12
	\$0008	53 01	53	LOCL	store	5.5
	\$0009	30 A9	A9 30	ASNDC \$30	a(i)+da*y(i)	19.5
			01	REFSL.1		12
	\$000A	01 01	01	REFSL.1		12
	\$000B	16 19	19 16	LIT8N 16		11
			EF	SKIPNZ LW		16
	\$000C	52 EF	52	POP		5.5

!* Time of execution *

	Calculated	Observed	Error %
Inside loop	658.5	--	---
Set-up	16.5	--	---
Total	10552.5	--	---

IIR Filter

$$v(m) = (1-BETA) * v(m-1) + BETA * x * x$$

Assume BETA and 1-BETA stored in Lenv

offset							#cycles
\$0000	10	22	22	10	REFDLE \$10	get x(0)	23.5
			6B		DUPD		9
\$0001	86	6B	86		MPYF	x ²	293
			3C		REFDL.C	get BETA	18
\$0002	86	3C	86		MPYF		293
			38		REFDL.8	get v(m-1)	18
\$0003	3E	38	3E		REFDL.E	get (1-BETA)	18
			86		MPYF		293
\$0004	84	86	84		ADDF		153
			C8		ASNDL.8	store v(m)	14
\$0005	1B	C8	1B		INTE		---

Calculated time of execution = 1132.5 cycles

D.4 Standard Precision Floating Point -Inline Coding

FIR Filter

This block of code performs an FIR filter on the array of input data and writes the result into the Local Environment. Data and coefficients are in floating-point data format.

$$Y = \text{SUM } a(i) * x(i) \quad i = 1 \dots 16$$

offset							#cycles
			10		LIT4A.0		5.5
\$0000	10	10	10		LIT4A.0	y = 0	5.5
\$0001	30	22	22	30	REFDLE \$30	get a(0)	23.5
\$0002	10	22	22	10	REFDLE \$10	get x(0)	23.5
			86		MPYF		293
\$0003	84	86	84		ADDF	y = y + *	153

```

$0004 32 22 22 32  REFDLE $32  get a(1)      23.5
$0005 12 22 22 12  REFDLE $12  get x(1)      23.5
      86          MPYF          293
$0006 84 86 84     ADDF          y = y + *      153
      .          .          .
      .          .          .
      .          .          .

$002E 3E 22 22 4E  REFDLE $4E  get a(F)      23.5
$002F 2E 22 22 2E  REFDLE $2E  get x(F)      23.5
      86          MPYF          293
$0030 84 86 84     ADDF          y = y + *      153

$0031 50 F7 F7 50  ASNDLE $50  store to y(0) 19.5

```

```

!*          Time of execution          *!
          Calculated      Observed      Error %
Set-up          30.5          --          ---
time per N          493          --          ---
16 coefficients   7918.5          --          ---

```

Filter Update

This block of code moves the array of time delayed input data to one sample greater delay and leaves x(0) empty for the next input of data.

```
x(i + 1) = x(i)          i = 1 . . 15
```

```

offset                                          #cycles
$0000 2C 22 22 2C  REFDLE $2C  get x(E)      23.5
$0001 2E F7 F7 2E  ASNDLE $2E  store to x(F) 19.5

$0002 2A 22 22 2A  REFDLE $2A  get x(D)      23.5
$0003 2C F7 F7 2C  ASNDLE $2C  store to x(E) 19.5
      .          .          .
      .          .          .
      .          .          .
$001E 10 22 22 10  REFDLE $10  get x(0)      23.5
$001F 12 F7 F7 12  ASNDLE $12  store to x(1) 19.5

```

!* Time of execution *	Calculated	Observed	Error %
time per N	43	--	---
16 coefficients (15 iterations)	646	--	---

Weight Update

The following section of code uses past output values to adapt the coefficients of an FIR filter. Data is in floating-point notation.

$$a(i) = a(i) + da * y(i)$$

Assume da stored in Lenv(A,B)

offset							#cycles
\$0000	50	22	22	50	REFDLE \$50	get y(0)	23.5
			3A		REFDL.A	get da	18
\$0001	86	3A	86		MPYF	y(0) * da	293
\$0002	30	22	22	30	REFDLE \$30	get a(0)	23.5
			84		ADDF		153
\$0003	F7	84	F7	30	ASNDLE \$30	save new a(0)	19.5
\$0004	22	50	22	52	REFDLE \$52	get y(1)	23.5
\$0005	3A	52	3A		REFDL.A	get da	18
			86		MPYF	y(1) * da	293
\$0006	22	86	22	32	REFDLE \$32		23.5
\$0007	84	32	84		ADDF		153
\$0008	32	F7	F7	32	ASNDLE \$32	save new a(1)	19.5
			.		.	.	
			.		.	.	
			.		.	.	
\$003A	22	6C	22	6E	REFDLE \$6E	get y(F)	23.5
\$003B	3A	52	3A		REFDL.A	get da	18
			86		MPYF	y(F) * da	293
\$003C	22	86	22	4E	REFDLE \$4E		23.5
\$003D	84	4E	84		ADDF	y(F) + da * y(F)	153
\$003E	4E	F7	F7	4E	ASNDLE \$4E		19.5

!* Time of execution *	Calculated	Observed	Error %
time per N	530.5	--	---
16 coefficients	8488	--	---

D.5 Extended Precision Floating Point -Loop Coding

Executive Entry Table

\$0000	00	00	Cont. Status pointer		
\$0001	71	00	Init. Exec Stack limit		
\$0002	74	FF	Init. Exec Top of Stack		
\$0003	00	80	Init. Exec PROCID		
\$0004	00	00	bus error PROCID	none	used
\$0005	00	00	NMI PROCID	"	"
\$0006	00	00	INT PROCID	"	"
\$0007	00	00	Trap PROCID	"	"
\$0008	00	00	Exception PROCID	"	"

Local variables

I	-	Lenv(1)	
Yquick	-	Lenv(2,3,4)	
y ²	-	Lenv(5,6,7)	
THETA	-	Lenv(8,9,A)	
v(m-1)	-	Lenv(B,C,D)	
da	-	Lenv(A0,A1,A2)	
BETA	-	Lenv(A3,A4,A5)	
1-BETA	-	Lenv(A6,A7,A8)	
x(0)	-	Lenv(10,11,12)	input buffer 16 long
.	.	.	
.	.	.	
x(F)	-	Lenv(3D,3E,3F)	
a(0)	-	Lenv(40,41,42)	coefficient table
.	.	.	
.	.	.	
a(F)	-	Lenv(6D,6E,6F)	

$y(0)$ - Lenv(70,71,72) output buffer 16 long
 ·
 ·
 $y(F)$ - Lenv(9D,9E,9F)

Initial Coefficient Table
 [Band-pass filter]

\$0010 82 D3	$a(0) = .01155124$
\$0011 01 7A	
\$0013 8F 60	$a(1) = .07222172$
\$0014 09 3E	
\$0016 D3 3E	$a(2) = .07476273$
\$0017 09 91	
\$0019 67 B7	$a(3) = -.04603866$
\$001A FA 1B	
\$001C 2C 7E	$a(4) = -.18493122$
\$001D E8 54	
\$001F 3B 6F	$a(5) = -.17043359$
\$0020 EA 2F	
\$0022 61 A0	$a(6) = .02344914$
\$0023 03 00	
\$0025 E8 F6	$a(7) = .21941864$
\$0026 1C 15	
\$0028 17 05	$a(8) = -.21941864$
\$0029 E3 EA	
\$002B 9E 60	$a(9) = -.02344914$
\$002C FC FF	
\$002E C4 91	$a(A) = .17043359$
\$002F 15 D0	
\$0031 D3 82	$a(B) = .18493122$
\$0032 17 A6	
\$0034 98 49	$a(C) = .04603866$
\$0035 05 E4	
\$0037 2C C2	$a(D) = -.07476273$
\$0038 F6 6E	

```
$003A 70 50      a(E) =-.07222172
$003B F6 C1

$003D 7D 2D      a(F) =-.01155124
$003E FE 85
```

This block of code copies the initial coefficients into the Local Environment for efficient access in the FIR Filter subprogram. Data is initially stored in ROM in extended precision Fractional data format and is converted to floating-point extended by this routine.

This is the first executable code of the program, therefore, immediately after invocation of the program, the executive stack mark, consisting of the program counter (PC), the Code environment (CENV), the Procedure identifier (PROCID), and the Local Environment pointer (LENV), is copied into the four memory locations immediately above the start of the Local Environment.

```
$0040 00 A8  procedure header      #local variables
```

Block Move

Address						#cycles
	\$0041	30 18	18 30	LIT8 48	I = 48	11
L1			13	LIT4A.3		5.5
	\$0042	E5 13	E5	SUB	I = I - 3	9
			41	ASNSL.1	save I	10
	\$0043	01 41	01	REFSL.1		12
	\$0044	10 68	68 10	REFDC \$10	get table(I)	23.5
			C6	CVTDFE		225
	\$0045	01 D9	01	REFSL.1		12
			53	LOCL		5.5
	\$0046	99 53	99 40	ASNDC \$40	store a(I)	23.5
	\$0047	01 40	01	REFSL.1		12
			01	REFSL.1		12
	\$0048	19 01	19 0F	LIT8N 0F		11
	\$0049	EF 0F	EF	SKIPNZ L1	I = 0 ?	16
			52	POP	kill old counter	5.5
	\$004A	1B 52	1B	INTE		---

!* Time of execution *!

	Calculated	Observed	Error %
Inside loop	377	--	---
Set-up	16.5	--	---
Total	6048.5	--	---

FIR Filter

This block of code performs an FIR filter on the array of input data and writes the result into the Local Environment. Data and coefficients are in floating-point data format.

$$Y = \text{SUM } a(i) * x(i) \quad i = 1 \dots 16$$

address						#cycles
		10		LIT4A.0	Yquick = 0	5.5
\$004B	10 10	10		LIT4A.0		5.5
		10		LIT4A.0		5.5
\$004C	B5 10	B5 02		ASNTLE 02		23.5
\$004D	18 02	18 02		LIT8 30	I = 48	11
L2 \$004E	13 30	13		LIT4A.3		5.5
		E5		SUB	I = I - 3	9
\$004F	41 E5	41		ASNSL.1		10
		01		REFSL.1		12
\$0050	53 01	53		LOCL		5.5
\$0051	10 76	76 10		REFTC \$10	get x(I)	59
		01		REFSL.1		12
\$0052	53 01	53		LOCL		5.5
\$0053	40 76	76 20		REFTC \$30	get a(I)	59
		94		MPYFE	a(I) * x(I)	539
\$0054	77 94	77 02		REFTLE 02		84
\$0055	92 02	92		ADDFE		229
\$0056	02 B5	B5 02		ASNTLE 02	add to y	23.5
		01		REFSL.1		12
\$0057	01 01	01		REFSL.1		12
\$0058	16 19	19 16		LIT8N 16		11
		EF		SKIPNZ L2	I = 0 ?	16
\$0059	52 EF	52		POP	kill old counter	5.5

!* Time of execution *!

	Calculated	Observed	Error %
Inside loop	1104	--	---
Set-up	56.5	--	---
Total	17720.5	--	---

Filter Update

This block of code moves the array of time delayed input data to one sample greater delay and leaves x(0) empty for the next input of data.

$$x(i + 1) = x(i) \quad i = 1 \dots 15$$

							#cycles
	\$005A	2D 18	18 2D	LIT8 45	I = 45		11
LU			13	LIT4A.3			5.5
	\$005B	E5 13	E5	SUB	I = I - 3		9
			41	ASNSL.1	save I		10
	\$005C	01 41	01	REFSL.1			12
			53	LOCL			5.5
	\$005D	76 53	76 10	REFTC \$10	get x(i)		59
	\$005E	01 10	01	REFSL.1			12
			53	LOCL			5.5
	\$005F	99 53	99 13	ASNTC \$13	save x(i + 2)		23.5
	\$0060	01 13	01	REFSL.1			12
			01	REFSL.1			12
	\$0061	19 01	19 OF	LIT8N OF			11
	\$0062	EF OF	EF	SKIPNZ LU	I = 0 ?		16
			52	POP	kill old counter		5.5

Input

The following segment inputs data from the input channel, converts it to floating-point extended, and stores it as x(0) in the Local Environment Extended.

\$0063	1C 52	1C 1000	REFSI \$1000	get input	23
\$0064	10 00	10	LIT4A.0	fractional CVTSD	5.5
\$0065	6C 10	6C	CVTDFE		225
\$0066	10 B5	B5 10	ASNTLE \$10		23.5

	!* Time of execution *!		
	Calculated	Observed	Error %
Inside loop	193	--	---
Set-up	16.5	--	---
Total	2911	--	---
input	277	--	---

Ratio Calculation

$$r = x^2/y^2$$

Assume x and y are stored in Local Environment Extended.
r is left on stack upon completion of segment.

offset							#cycles
\$0000	70	77	77	70	REFTLE \$70	get y(0)	84
			79		DUPT		49.5
\$0001	94	79	94		MPYFE		539
\$0002	05	B5	B5	05	ASNTLE 05	store y ² temp	23.5
\$0003	77	C4	77	10	REFTLE \$10	get y(0)	84
\$0004	79	10	79		DUPT		49.5
			94		MPYFE	x ²	539
\$0005	77	94	77	05	REFTLE 05	get y ² back	23.5
\$0006	95	05	95		DIVFE		706

Calculated time of execution = 2098 cycles

Decision

$$d = \begin{cases} 1 & r \geq \text{THETA} \\ 0 & r < \text{THETA} \end{cases}$$

d is boolean

Assume THETA stored in Lenv(8,9,A)

Assume that r was left on stack from RATIO CALCULATION segment.

							#cycles
\$0007	08	77	77	08	REFTLE 08		84
			9D		EXCHT		47
\$0008	91	9D	91		GRFE		53
			11		LIT4A.1	toggle flag	5.5
\$0009	EA	11	EA		XOR		5.5

Calculated time of execution = 195 cycles

Weight Update

The following section of code uses past output values to adapt the coefficients of an FIR filter. Coefficients are in floating-point extended format.

$$a(i) = a(i) + da * y(i)$$

Assume da stored in Lenv(A0,A1,A2

offset							#cycles
\$0000	30	18	18	30	LIT8 48	I = 48	11
LW	\$0001	E5	13	13	LIT4A.3	I = I - 3	5.5
				E5	SUB		9
				41	ASNSL.1		10
\$0002	01	41	01		REFSL.1		12
				53	LOCL		5.5
\$0003	76	53	76	70	REFTC \$70	get y(i)	59
\$0004	77	70	77	A0	REFTLE \$A0		84
\$0005	94	A0	94		MPYFE		539
				01	REFSL.1		12
\$0006	53	01	53		LOCL		5.5
\$0007	40	76	76	40	REFTC \$40	get a(i)	84
				92	ADDFE		229
\$0008	01	92	01		REFSL.1		12
				53	LOCL	store	5.5
\$0009	99	53	99	40	ASNTC \$40	a(i)+da*y(i)	23.5
\$000A	01	40	01		REFSL.1		12
				01	REFSL.1		12
\$000B	19	01	19	17	LIT8N 17		11
				EF	SKIPNZ LW		16
\$000C	EF	17	52		POP		5.5
\$000D	1B	52	1B		INTE		---

!* Time of execution *!

	Calculated	Observed	Error %
Inside loop	1146.5	--	---
Set-up	16.5	--	---
Total	18360.5	--	---

IIR Filter

$$v(m) = (1-BETA) * v(m-1) + BETA * x * x$$

Assume BETA and 1-BETA stored in Lenv

offset							#cycles
\$0000	10	77	77	10	REFTLE \$10	get x(0)	84
			79		DUPT		49.5
\$0001	94	79	94		MPYFE	x ²	539
\$0002	A3	77	77		REFTLE A3	get BETA	84
			94		MPYFE		539
\$0003	77	94	77	0B	REFTLE \$0B	get v(m-1)	84
\$0004	77	0B	77	A6	REFTLE \$A6	get 1-BETA	84
\$0005	94-A6		94		MPYFE		539
			92		ADDFE		229
\$0006	B5	92	B5	0B	ASNTLE \$0B	store v(m)	23.5
\$0007	1B	0B	1B		INTE		---

Calculated time of execution = 2255 cycles

D.6 Extended Precision Floating Point -Line coding

FIR Filter

This block of code performs an FIR filter on the array of input data and writes the result into the Local Environment extended. Data and coefficients are in floating-point data format.

$$Y = \text{SUM } a(i) * x(i) \quad i = 1 \dots 16$$

offset							#cycles
		10		LIT4A.0			5.5
\$0000	10	10	10	LIT4A.0			5.5
			10	LIT4A.0	y = 0		5.5
\$0001	77	10	77	40	REFTLE \$40	get a(0)	84
\$0002	77	40	77	10	REFTLE \$10	get x(0)	84
\$0003	94	10	94		MPYFE		539
			92		ADDFE	y = y + *	229
\$0004	77	92	77	43	REFTLE \$43	get a(1)	84
\$0005	77	43	77	13	REFTLE \$13	get x(1)	84
\$0006	94	13	94		MPYFE		539
			84		ADDFE	y = y + *	229
			
			
			
\$002E	77	92	77	6D	REFTLE \$6D	get a(F)	84
\$002F	77	6D	77	3D	REFTLE \$3D	get x(F)	84
\$0030	94	3D	94		MPYFE		539
			92		ADDFE	y = y + *	229
\$0031	B5	92	B5	70	ASNTLE \$70	store to y(0)	23.5
\$0032	1B	70	1B		INTE		---

	!* Time of execution *		
	Calculated	Observed	Error %
Set-up	40	--	---
time per N	936	--	---
16 coefficients	15016	--	---

Filter Update

This block of code moves the array of time delayed input data to one sample greater delay and leaves x(0) empty for the next input of data.

$$x(i + 1) = x(i) \quad i = 1 \dots 15$$

offset							#cycles	
\$0000	3A	77	77	3A	REFTLE	\$3A	get x(E)	84
\$0001	3D	B5	B5	3D	ASNTLE	\$3D	store to x(F)	23.5
\$0002	37	77	77	37	REFTLE	\$37	get x(D)	84
\$0003	3A	B5	B5	3A	ASNTLE	\$3A	store to x(E)	23.5
			:		:		:	
			:		:		:	
\$001E	10	77	77	10	REFTLE	\$10	get x(0)	84
\$001F	13	B5	B5	13	ASNTLE	\$13	store to x(1)	23.5

	!* Time of execution *		
	Calculated	Observed	Error %
time per N	107.5	--	---
16 coefficients	1612.5	--	---
(15 iterations)			

Weight Update

The following section of code uses past output values to adapt the coefficients of an FIR filter. Data is in floating-point notation.

$$a(i) = a(i) + da * y(i)$$

Assume da stored in Lenv(A0,A1,A2)

offset							#cycles	
\$0000	70	77	77	70	REFTLE	\$70	get y(0)	84
\$0001	A0	77	77	A0	REFTLE	\$A0	get da	84
			94		MPYFE			539
\$0002	77	94	77	40	REFTLE	\$40	get a(0)	84
\$0003	92	40	92		ADDFE			229
\$0004	40	B5	B5	40	ASNTLE	\$40	store a(0)	23.5

\$0005	73	77	77	73	REFTLE \$73	get y(1)	84
\$0006	A0	77	77	A0	REFTLE \$A0	get da	84
			94		MPYFE		539
\$0007	77	94	77	43	REFDLE \$43	get a(1)	84
\$0008	92	43	92		ADDFE		229
\$0009	40	B5	B5	40	ASNTLE \$43	store a(1)	23.5
			.		.	.	
			.		.	.	
			.		.	.	
\$0044	9D	77	77	9D	REFTLE \$9D	get y(F)	84
\$0045	A0	77	77	A0	REFTLE \$A0	get a(F)	84
			94		MPYFE		539
\$0046	77	94	77	6D	REFTLE \$6D	get a(F)	84
\$0047	92	6D	92		ADDFE		229
\$0048	6D	B5	B5	6D	ASNTLE \$6D	store a(F)	23.5

!* Time of execution *!

 Calculated Observed Error %

time per N	1043	--	---
16 coefficients	16696	--	---

AN IMPLEMENTATION AND EVALUATION OF A
MINIMAL-COMPONENT MINIMAL-POWER MICROCOMPUTER
SYSTEM USING ROCKWELL'S AAMP

by

Eric Nelson

B. S. , Kansas State University, 1985
A. G. S., Colby Community Junior College, 1983

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

1987

ABSTRACT

This thesis describes a hardware implementation of a minimal component, minimal power, microcomputer system using Rockwell's Advanced Architecture MicroProcessor (AAMP), a high-performance 16-bit floating-point CMOS microprocessor.

The research was funded through a contract between Kansas State University and Sandia National Laboratories which has traditionally supported ultra-low power designs of A/D converters and microcomputer systems.

The system described is intended for use as the signal processing portion of an ultra-low power, highly portable, intruder detection system, therefore low power consumption and minimal parts count were the main objectives in design.

The thesis briefly describes the AAMP with more emphasis placed on items used in the design. Power consumption is shown for the system and for each component individually, with parts selections based on these figures explained. Performance of the AAMP as a signal processor is tested using several signal processing code segments with time of execution shown for various types of data.

The final product is a floating-point capable microcomputer system with power consumption at a proposed 2.6 Mhz operation of under 25 mW.

07-04-11
08-12