AN EVALUATION OF ROCKWELL'S ADVANCED ARCHITECTURE MICROPROCESSOR FOR DIGITAL SIGNAL PROCESSING APPLICATIONS

by

KENNETH LEE ALBIN

B. S., Kansas State University, 1981

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

1984

Approved by:
TABLE OF CONTENTS

Introduction 1

Features of the AAMP 3

Software environment .................................. 3
Process Stack ........................................... 9
Executive Process ........................................ 14
Event handling ........................................... 19

Evaluation procedure 21

Code used for evaluation ................................ 21
Widrow algorithm modification ............................ 28
Optimization for the AAMP ................................ 32
Compiler optimization ......................................... 43
Performance measurements .................................... 47

Results and Conclusions 63

Acknowledgements 68

References 69

Appendices 70

Appendix A : Hand compiled listings 70

Notes on hand compiled listings ......................... 70
Standard Widrow listing .................................. 73
Standard Lattice listing ................................... 79
Modified Widrow listing .................................... 84
Optimized Widrow listing .................................. 89
Optimized Lattice listing .................................. 94

Appendix B : Ada-subset listings 99

Notes on Ada-subset compiler ............................. 99
Widrow fixed-point listings ................................ 100
Widrow floating-point listings ............................... 107
Lattice fixed-point listings ................................ 111
Lattice floating-point listings ............................... 116
ADATESTS fixed-point listings .............................. 120
ADATESTS floating-point listings ........................... 127
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>AAMP addressing modes.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Process stack during procedure call</td>
<td>10</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Process stack and linkages</td>
<td>11</td>
</tr>
<tr>
<td>Figure 4</td>
<td>User PSD and process stack</td>
<td>17</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Executive and User data structures</td>
<td>18</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Widrow algorithm</td>
<td>22</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Lattice algorithm</td>
<td>23</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Weight updating process</td>
<td>30</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Sample and error array updating scheme</td>
<td>31</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Sample and weight pair updating</td>
<td>31</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Synchronization timing diagrams</td>
<td>54</td>
</tr>
<tr>
<td>Figure 12</td>
<td>AAMP Timing worksheet</td>
<td>59</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Execution rate quick estimate worksheet</td>
<td>60</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 1</td>
<td>AAMP data types.</td>
<td>7</td>
</tr>
<tr>
<td>Table 2</td>
<td>AAMP arithmetic operations</td>
<td>8</td>
</tr>
<tr>
<td>Table 3</td>
<td>AAMP Widrow algorithm execution rates</td>
<td>26</td>
</tr>
<tr>
<td>Table 4</td>
<td>AAMP Lattice algorithm execution rates</td>
<td>26</td>
</tr>
<tr>
<td>Table 5</td>
<td>Stack-updating vs other operations</td>
<td>33</td>
</tr>
<tr>
<td>Table 6</td>
<td>Opcode to stack update mapping</td>
<td>34</td>
</tr>
<tr>
<td>Table 7</td>
<td>Non-optimum stack-updating instructions</td>
<td>35</td>
</tr>
<tr>
<td>Table 8</td>
<td>Microcycles required by the AAMP</td>
<td>48</td>
</tr>
<tr>
<td>Table 9</td>
<td>Estimated vs measured execution rates</td>
<td>52</td>
</tr>
<tr>
<td>Table 10</td>
<td>Widrow compilation differences</td>
<td>62</td>
</tr>
<tr>
<td>Table 11</td>
<td>Lattice compilation differences</td>
<td>62</td>
</tr>
<tr>
<td>Table 12</td>
<td>Widrow implementation comparisons</td>
<td>64</td>
</tr>
<tr>
<td>Table 13</td>
<td>Lattice implementation comparisons</td>
<td>65</td>
</tr>
</tbody>
</table>
Introduction

With the advent of low-cost and relatively high performance microprocessors, digital signal processing has found application in a wide variety of fields. One such application is the use of adaptive linear prediction in intruder detection devices. These algorithms reduce false alarms by adapting to correlated background noise and passing only intruder signals. Many processors are capable of performing these algorithms in real-time, but few of these have the low power requirements desirable for field applications. The Electrical and Computer Engineering Department at Kansas State University, in conjunction with Sandia National Laboratories, has attempted to identify processors which are most appropriate for such use. The ideal processor would require very little power, be easy to interface, perform multiplications very quickly and use floating-point arithmetic. Processors which have been previously evaluated include the Zilog Z80, Intel 8748, RCA ATMAC [1] and National NSC800 [2,3]. These processors were successful to varying degrees, but still left much room for improvement.

In the winter of 1983-1984, a processor that satisfies the above criteria became available for evaluation. This microprocessor, the Advanced Architecture Microprocessor (AAMP), was designed by Rockwell-Collins in Cedar Rapids, Iowa and is produced by Rockwell in Anaheim, California. It is a CMOS/SOS
microprocessor that has a stack architecture with a 16-bit wide data path. Single and double precision integer and fractional as well as single and extended precision floating-point data types are supported on a single chip. It consumes approximately 50 mW at its rated 20 MHz clock rate and uses a single 5 volt supply.

The purpose of this thesis is to examine the architecture of the AAMP and attempt to estimate performance on signal processing algorithms. Special attention is paid to both strong points and bottlenecks of the processor. Relative efficiency that can be achieved with high-level languages is also investigated.

The remainder of this thesis consists of three parts. The first part is an introduction to the AAMP's architecture, instruction set and data structures. This description is not exhaustive but seeks to highlight the processor's properties which are significant to the evaluation at hand and to supplement the detailed treatment available from Rockwell-Collins. The second part details the investigation and findings from the evaluation. Included in this section is a discussion of ways to optimize the Widrow and Lattice algorithms for the processor's architecture. The third part contains the results and conclusions of the evaluation in a concise form.

Gary Mauersberger is currently completing a hardware oriented evaluation of the AAMP which includes the development of a minimal system. The hardware evaluation combined with this thesis should provide a comprehensive view of the AAMP and form a basis for future comparisons of microprocessors.
Features of the AAMP

The purpose of this chapter is to provide an introduction to the architecture and capabilities of the AAMP. The discussion is directed toward an Electrical Engineer with a limited knowledge of computer run-time structures. A concise but detailed description can also be found in the August 1982 issue of IEEE Micro [4]; a very detailed description is contained in a document from Collins-Rockwell titled AAMP, CAPS-7 and CAPS-10 INSTRUCTION SET ARCHITECTURE [5].

Software environment

The primary run-time structure found in the AAMP is the process stack. This process stack contains the environment of the currently active procedure and the status of procedures that were suspended in the calling process. This will be discussed in more detail below.

Because the AAMP has nearly a pure stack architecture, that is, it has no user-accessible registers, nearly all of its instructions fall into four main categories:

1) Memory reference instructions which place the contents of the specified memory location on the top of the stack. Also, literal instructions which place constants on the top of the stack.

2) Operators which perform actions on operands which reside on the top of the stack, deleting the operands and placing the result on the top of stack.

3) Memory assignment instructions which remove data from the
top of the stack and place them in the specified memory location.

4) Control instructions such as SKIP, CALL and RETURN which affect the sequence in which instructions are executed.

The AAMP uses a 24 bit address word to select 16 bit memory words. Since all AAMP opcodes are 8 bits long, the 16 bit word containing the opcode byte is read and a 25th bit is used internally to select the proper byte. Constructing the 24 bit address from concatenating the top two stack locations is known as the Universal addressing mode (see Figure 1a).

Because the data path is only 16 bits wide, it becomes awkward to specify the full address. In order to increase efficiency, the Global addressing form specifies the least significant 16 bits and automatically uses the upper address bits specified when the procedure started. The 8 most significant address bits for data constitute the Data Environment (DENV). The Code Environment (CENV) consists of the 9 most significant address bits for the area of memory containing the opcodes. The 16 least significant address bits are specified by the word on the top of the stack (see Figure 1b) or by two immediate bytes following the opcode. Note that the DENV and CENV can both refer to the same area if desired.

A third form of addressing is yet more efficient and is used to reference variables local to the current procedure (see Figure 1c). A reference or assignment using Local addressing can specify any of 16 locations in a single byte opcode or any of 256 locations using a one byte opcode with an immediate byte. Local addressing is also very useful because of the nature of block
structured languages and their emphasis on local variables. Instructions are available to provide the absolute address of a local storage location in the current procedure (LOCL) and in calling procedures (LOCNL).

Finally, memory may be accessed through an Indexed addressing mode where the index into the array is contained in the stack and the base address of the array is either on top of the stack or in an immediate word following the opcode. The array base and index are used to calculate the address of the element, taking into account the data type specified in the instruction (see Figure 1d). Another addressing mode is the Constant Offset form which is essentially the same as the Indexed immediate mode with the offset in an immediate byte and the array base on the top of the stack. The calculation of the element's address consists of adding the base and the offset together without taking into account the data type being accessed as the Indexed mode does.

Each addressing mode discussed above can be used to access single (16-bit) words and double (32-bit) or triple (48-bit) words stored in the form of consecutive 16-bit memory locations. Also, a byte indexed mode is available wherein a byte offset is added to a base (both of which must be on the stack) to access a byte.
a) REFSU: reference single word with Universal addressing mode.


before    after

REFS: reference single word with Global addressing mode.

TOS→[ BBBB ] TOS→[ 6666 ] CCBBBBB [ 6666 ]

before    after

c) REFSL.0: reference single Local from location 0.

TOS→[ 1000 ] DENV = xxCC
[ 0040 ] TOS→[ 4444 ] CC1040 [ 4444 ]

before    after

d) REFSX: reference single word indexed.

Figure 1. AAMP Addressing Modes
The three word-lengths correspond to the data types supported in the instruction set with arithmetic and conversion functions. The data types are shown in Table 1.

Table 1. AAMP data types

<table>
<thead>
<tr>
<th>Data type</th>
<th>Precision</th>
<th>Length</th>
<th>Notation used</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>-</td>
<td>16 bits</td>
<td>0=FALSE, else TRUE</td>
</tr>
<tr>
<td>Integer</td>
<td>Single</td>
<td>16 bits</td>
<td>Two's complement</td>
</tr>
<tr>
<td>Integer</td>
<td>Double</td>
<td>32 bits</td>
<td>Two's complement</td>
</tr>
<tr>
<td>Fractional</td>
<td>Single</td>
<td>16 bits</td>
<td>Two's complement, msb = $2^{-1}$</td>
</tr>
<tr>
<td>Fractional</td>
<td>Double</td>
<td>32 bits</td>
<td>Two's complement, msb = $2^{-1}$</td>
</tr>
<tr>
<td>Floating-point</td>
<td>Single</td>
<td>32 bits</td>
<td>Signed, hidden-bit, 8 bit XS128 exponent</td>
</tr>
<tr>
<td>Floating-point</td>
<td>Extended</td>
<td>48 bits</td>
<td>Signed, hidden-bit, 8 bit XS128 exponent</td>
</tr>
</tbody>
</table>

In the floating-point notation, the mantissa is represented in a positive normalized form with the sign bit and an assumed binary point to the left of the most significant bit. Since a properly aligned floating-point number (the AAMP automatically handles alignment) will have a one for the most significant bit (except in the case of zero), the bit can be omitted. The representation of zero is defined to be the case where the sign, mantissa and exponent are all zero. The exponent is represented in excess 128 form in the least significant byte. Extended
precision floating-point numbers are the same except for 16 additional bits of precision in the mantissa.

The six arithmetic operations available for each of the above non-boolean data types and their execution times are shown in Table 2. Other instructions perform boolean (AND, OR, NOT and XOR) and numeric data type conversion operations.

Table 2. Arithmetic operations and execution times.
(all times in microseconds)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Fixed-point</th>
<th></th>
<th>Floating-point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>single precision</td>
<td>double precision</td>
<td>single precision</td>
</tr>
<tr>
<td>Addition</td>
<td>0.55</td>
<td>0.75</td>
<td>7.55</td>
</tr>
<tr>
<td>Subtraction</td>
<td>0.55</td>
<td>0.75</td>
<td>8.65</td>
</tr>
<tr>
<td>Multiplication</td>
<td>4.75</td>
<td>14.95</td>
<td>19.15</td>
</tr>
<tr>
<td>Division</td>
<td>5.55</td>
<td>15.75</td>
<td>19.75</td>
</tr>
<tr>
<td>Negation</td>
<td>0.55</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>Absolute value</td>
<td>0.75</td>
<td>0.85</td>
<td>0.35</td>
</tr>
</tbody>
</table>

The preceding paragraphs have briefly described the primary data types available and the instructions to manipulate them. The procedures doing the manipulations are rooted in a process stack which is dedicated to a particular task. This means that a task's procedures, functions and subroutines and their associated local variables, accumulator stack and parameters are all contained in the stack. AAMP supports the concept of task concurrency, that is, having multiple independent process stacks. An executive stack initializes the system on reset and provides
the means for transferring control between tasks.

Process Stack

The process stack for a task has an active stack frame for the currently active procedure on the top and its calling procedures' stack frames below it in the calling order. When a procedure is called, a new stack frame is set up on top of the current one and becomes active. When the procedure ends, the new stack frame is deactivated and, in effect, becomes lost as the previous stack frame becomes active. This is illustrated in Figure 2. Each of these stack frames consists of three main areas: 1) the accumulator stack, 2) the local storage area and 3) the stack mark.

A procedure's accumulator stack is the area on the top of the stack where nearly all operations on data are performed. This area is the logical equivalent of registers in conventional architecture microprocessors. The accumulator stack is initially empty but grows as literal and reference instructions place data on it and shrinks as words of data are removed by operations and assignment instructions. If the current procedure calls another procedure, the accumulator stack is left unused under the new stack frame until the new stack frame is removed when the new procedure returns.
Figure 2. Process stack for Procedure A, after Procedure A has called Procedure B, and after Procedure B has returned.
Figure 3. Process stack and linkages after Procedure A calls Procedure B.
The Local storage area is the area of the stack frame below both the accumulator stack and stack mark. This area is used for variables needed for the procedure associated with the stack frame. This local variable area has four advantages: 1) the quickest access times, 2) freedom from side-effects from other procedures, 3) space is reclaimed automatically when the procedure ends and 4) independence from a particular location in memory or calling order. Local variable locations are created when the stack frame is set up by leaving unused a specified number of stack locations between the calling procedure's accumulator stack and the stack mark of the stack frame being created. This is shown in Figure 2.

The stack mark is the linkage between a procedure and its calling procedure as shown in Figure 3. Recorded in the stack mark is the calling procedure's Syllable (byte) Program Counter Register (SPCR), Code Environment (CENV), Procedure ID (PROCID) and Local Environment (LENV). The Code Environment is concatenated with the Syllable Program Counter Register to form the byte address of the instruction of the calling procedure which is to be executed upon return from the called procedure. The Procedure ID is an identification number for the calling procedure which happens to be the byte address of the header of its code body. The Local Environment is a pointer to the location of the first Local storage location of the calling procedure. These four words of data give the processor information it needs to restart the calling procedure when the called procedure ends.
At this point, it is appropriate to ask what makes up a procedure. A procedure is a body of code with a header at the location given by PROCID. This single word header defines the number of words of storage to allocate for Local variables between the calling procedure's accumulator stack and the new stack frame's stack mark. The least significant byte of the word following the header contains the first opcode to be executed in the new procedure. Each time a procedure is called, a stack frame is created to be associated with the procedure. Each time a procedure is exited, the stack frame associated with that activation is discarded. Thus, as long as Universal or Global references are not used, the procedure may be called by different procedures, by itself or even by different tasks and work well, free from unwanted side-effects. Procedures are therefore recursive with the above qualifications.

The calling sequence has been described above, but there is one more detail: argument passing. To pass arguments to a called procedure, the arguments are simply placed on top of the calling procedure's accumulator stack before the CALL instruction is executed. Since these arguments (and the rest of the calling procedure's accumulator stack) are just below the called procedure's Local variables, the called procedure can access them using the Local addressing mode. The number of Local variables and their relative locations are assigned and incorporated into the procedure's header and instructions at the time the code is
When a RETURN is executed, the top of the accumulator stack must contain a number. This number tells the processor how many storage locations below the stack mark to "deallocate." The locations deallocated can include the called procedure's Local storage, passed arguments and locations on the calling procedure's accumulator stack. The called procedure's stack mark is used to restore the processor state and is then discarded along with the indicated number of local variables and calling procedure arguments. Any locations between the called procedure's stack mark and the deallocation number are considered to be arguments to be returned and are copied onto the newly determined top of the calling procedure's stack. Note that parameters can also be returned if they reside in the local storage locations immediately adjacent to calling procedure's accumulator stack. The number of locations to be deallocated would simply be the total number of local storage locations less the number of locations to be left on the stack.

Executive process

In a system that may have multiple process stacks, the mechanism which organizes the transfer of control between processes is the Executive process. The Executive process begins execution on reset through use of the Executive Entry Table. The Executive Entry Table is located at memory addresses 0-8 and contains information in three categories: 1) a Continuation Status Pointer, 2) initialization information and 3) PROCIDs for procedures handling special events that might arise.
When the processor is reset, there must be some way for it to tell if it is starting cold or if it was executing a procedure which needs to continue. The Continuation Status Pointer at location 0 contains the address of a memory location. If the memory location pointed to contains zero, it indicates that initialization should take place upon reset. Nonzero contents indicate that the processor was interrupted in the middle of some process, the status of which has been preserved and may now be recovered to resume execution. Note that a pointer was used because the Executive Entry Table will nearly always be located in ROM and the indicated location in RAM. If the processor is always to be initialized on reset, a zero in location 0 will insure this.

The three pieces of data in the Executive Entry Table used in initialization are the Initial Stack Limit, Initial Top of Stack and the Initial PROCID. The first two elements define the location and extent of the Executive stack and the third element gives the location of the instructions needed to perform initialization. In addition, the processor automatically sets LENV = DENV = CENV = 0 and disables interrupts. The resulting processor state is known as the Initialization State.

If a suspended process is to be restarted, the conditions which existed before interruption must be recovered from the process's Processor State Descriptor (PSD). For the Executive process, this PSD is written out just before the processor halts. This halting can occur from executing the HALT instruction or
from any of a number of error conditions that have been trapped. Recorded in this Executive PSD are the contents of internal registers that make up the processor state: Stack Limit (SKLM), Top of Stack (TOS), LENV, DENV, SPCR and CENV. In addition, the interrupt enable flip-flop status and an error code giving the reason the processor was halted are provided. This dumping of the processor status happens just below the Initial Executive Top of Stack (the base of the Executive stack). The processor can be restarted only if the error code indicates that the stoppage was caused by the HALT instruction. No other errors can be corrected by the processor (bus failure, etc.) and all are considered fatal.

Once the Executive process has been started, it may then call other procedures and perform operations on the Executive stack. This single task system is the simplest configuration. If multi-tasking is to take place, the Executive task must take the responsibility for scheduling the tasks and initiating their execution. Each User task (any task except the Executive task) has its own PSD. This PSD contains the processor status (SKLM, TOS, LENV, DENV, SPCR and CENV) of the task when execution was stopped or the initial status if it has not yet been executed. In addition, the PROCID and CENV for both the task and exception handling routines are recorded. The User PSD and its relationship with its process stack is shown in Figure 4.
Figure 4. User PSD and Process stack.
**EXECUTIVE ENTRY TABLE**

<table>
<thead>
<tr>
<th></th>
<th>[CONT.STAT.PTR ]</th>
<th>[CONTINUE.STAT]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[INIT.EXEC.SKLM]</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>[INIT.EXEC.TOS ]</td>
<td>[INIT.PROCID ]</td>
</tr>
<tr>
<td>2</td>
<td>[BUS.ERROR.PROC]</td>
<td>[NMI.PROCID ]</td>
</tr>
<tr>
<td>3</td>
<td>[INT.PROCID ]</td>
<td>[TRAP.PROCID ]</td>
</tr>
<tr>
<td>4</td>
<td>[EXCEP.PROCID ]</td>
<td>[EXCEP.PROCID ]</td>
</tr>
<tr>
<td></td>
<td>EXEC.STACK</td>
<td>EXEC.STACK</td>
</tr>
<tr>
<td></td>
<td>-&gt; [</td>
<td>[</td>
</tr>
<tr>
<td></td>
<td>[</td>
<td>[</td>
</tr>
<tr>
<td></td>
<td>[</td>
<td>[</td>
</tr>
<tr>
<td></td>
<td>[</td>
<td>[</td>
</tr>
<tr>
<td></td>
<td>[</td>
<td>[</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th></th>
<th>[USER.PSD.PTR ]</th>
<th>[EXEC.SKLM ]</th>
<th>[EXEC.TOS ]</th>
<th>[EXEC.LENV ]</th>
<th>[EXEC.DENV ]</th>
<th>[EXEC.SPCR ]</th>
<th>[EXEC.CENV ]</th>
<th>[INT.ENABLE.FF ]</th>
<th>[EXEC.ERR.CODE ]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[</td>
<td>-</td>
<td>:</td>
<td>:</td>
<td>: EXEC.PSD</td>
<td>:</td>
<td>:</td>
<td>:</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>[</td>
<td>[</td>
<td>[</td>
<td>[</td>
<td>[</td>
<td>[</td>
<td>[</td>
<td>[</td>
<td>[</td>
</tr>
<tr>
<td></td>
<td>Task A</td>
<td>Task B</td>
<td>Task B</td>
<td>Task C</td>
<td>Task C</td>
<td>Task C</td>
<td>Task C</td>
<td>Task C</td>
<td>Task C</td>
</tr>
</tbody>
</table>

**Note:** Task B and Task C are inactive

**Figure 5. Executive and User data structures**
The Executive process initiates a User task by executing a RETURN from the outermost Executive procedure. A pointer to the PSD of the User task must be stored in the initial Executive TOS. The processor uses the information in the indicated User task PSD to set up the proper state in the processor and resume (or begin) execution of the task. This is called Outer Procedure Return Processing. The relationship between the Executive Entry Table, Executive PSD and User PSD is shown in Figure 5.

The other instance of Outer Procedure Return processing is when the User task execution is terminated. This could be due to either an interrupt or a trap. The most common type of trap is that which is generated when a procedure attempts to return with no previous procedures on the User task's process stack. In any case, the status of the processor is written into the task's PSD so that execution can resume in the future and a pointer to the User task's PSD remains in the initial top of the Executive process stack. If the process has terminated itself, the PSD is reset to its initial start-up state.

Event handling

There are three special kind of events that are handled by the processor: interrupts, traps and exceptions. Interrupts are generated externally by a reset, by a bus error condition or by an external device asking for service. Traps are essentially interrupts generated by the CPU itself. A trap can be caused by the TRAP instruction, by an illegal instruction or by data accessing problems such as stack overflow. Interrupts and traps
are handled on a system-wide basis by specific Executive procedures. The PROCID of the routine corresponding to the type of interrupt or trap can be found in the Executive Entry Table. If a User task is executing at the time a trap or interrupt occurs, the processor first performs an Outer Procedure Return to save the User task status and to return to the Executive mode where the proper servicing routine can be activated. In the case of a trap, the trap number is placed on the top of the Executive process stack so that it will be passed to the trap handling procedure. The trap handling procedure may then use the trap number to select and call a procedure appropriate to handle the trap.

The other type of event, exceptions, are handled separately by each task. Exceptions occur when the data being processed cause arithmetic overflows, division by zero, etc. These events can be handled in a default manner if no exception handling procedures are specified (EXCEP.PROCID = 0). If an exception procedure is specified, it is handled as a normal procedure call on the currently active process stack with the exception type number passed as an argument.
This chapter discusses the research performed toward completion of this thesis. Two main evaluation areas were addressed. The first area was the identification of potential strengths and weaknesses in the processor's architecture and implementation. This was accomplished by first conducting a general study of the processor, followed by a specific analysis of its potential performance in the execution of signal processing algorithms. At the same time, an attempt was made to estimate how efficiently the AAMP can execute high-level compiled languages. The second area was the testing of the validity of the first analysis by running benchmark programs coded in the first step. This was accomplished through the cooperation of Collins-Rockwell in Cedar Rapids during a Sandia-sponsored visit. Also, the author assisted Gary Mauersberger this spring in the interfacing of a prototyping board supplied by Rockwell to the Electrical Engineering department's HP9845B testing system. This allowed the AAMP's initialization procedure and specific transfer sequences to be confirmed on a logic analyzer.

Code Used for Evaluation

To compare the AAMP to other processors in the execution of signal processing algorithms, it was necessary to use programs which are representative of the class of algorithms which would be ultimately run on the processor. Two adaptive linear prediction algorithms, the Lattice and Widrow, were selected
because they had been used for this purpose in previous evaluations. These algorithms are shown in Figures 6 and 7. These choices seem to be valid ones because even though the specific algorithms or implementations may not be used in future designs, the type of operations performed will be similar. Specifically, both algorithms involve large numbers of multiplications and array handling in a real-time environment. These factors were examined closely in addition to the general performance of the stack-architecture for this type of algorithm.

\[
\begin{align*}
(1) \quad g(m) &= \sum_{k=1}^{n} b(m,k) \cdot f(m-k) \\
(2) \quad e(m) &= f(m) - g(m) \\
(3) \quad b(m,k) &= u \cdot b(m-1,k) \quad k = 1,2,\ldots,n \\
&\quad \quad + v \cdot e(m) \cdot f(m-k) \\
(4) \quad q(m) &= \frac{1}{L} \sum_{k=1}^{L} e(m-k+1) \\
(5) \quad q2(m) &= q(m) \cdot q(m) \\
\end{align*}
\]

\[\text{delta} = 1 \text{ implied by subscripts} \]
\[n = 16 \text{ (# of weights)} \]
\[m = \text{iteration} \]
\[k = 1,2,\ldots,n \]
\[L = 16 \text{ (MAF window size)} \]
\[\text{output is squared for threshold detection} \]

Figure 6. Widrow adaptive linear prediction algorithm
\text{e}(l) = \text{adc\_input}
\text{wl}(l) = \text{e}(l)
\text{do } l = 1, n
\text{e}(l+1) = \text{e}(l) - k(l) \times \text{wl}(l)
\text{w}(l+1) = \text{wl}(l) - k(l) \times \text{e}(l)
\text{v}(l) = \text{beta} \times \text{v}(l) + \text{betal} \times (\text{e}(l) \times \text{e}(l) + \text{wl}(l) \times \text{wl}(l))
k(l) = k(l) + \text{alpha} \times (\text{e}(l+1) \times \text{wl}(l) + \text{e}(l) \times \text{w}(l+1))/\text{v}(l)
\text{wl}(l) = \text{w}(l)
\text{endo}
\text{wl}(n+1) = \text{w}(n+1)
\text{dac\_out} = \text{e}(n+1)
\text{loop back to the beginning}

Figure 7. Lattice adaptive linear prediction algorithm

Because of the potential speed of the processor and the unique architecture (among micros), two versions of both the Lattice and Widrow algorithms were coded. The listings of these programs can be found in the appendices. The first version was coded from a high-level representation to indicate what one would expect from a compiler. The second version takes advantage of assembly language "tricks" to optimize performance. The results of this comparison are discussed in detail in the following sections. After the Widrow algorithm had been coded, it was
discovered that previous evaluations had used what appeared to be a less efficient method of implementation. The fifth listing was written to correspond to the earlier implementations and see how much was gained from the algorithm modifications. The gain was 18% for floating-point and 35% for fixed-point calculations. This modification is discussed in detail in the section titled "Widrow Algorithm Modification".

The AAMP was designed as a stack-machine to enhance support of high-level language constructs. This efficiency coupled with the processor's speed allows high-level language implementation of algorithms which previously required assembly language programming. The ability to implement algorithms in high-level languages is a big advantage because it decreases required programmer time and increases program reliability, portability and quality of documentation.

Beginning with high-level pseudo-language, the Widrow and Lattice algorithms were coded and then converted into AAMP assembly language. Initialization was not included because it is quite language dependent and would not affect the performance of the algorithm once begun. It was assumed, however, that the most frequently used variables were declared as local variables and that the proper variable type declarations had been made. It was also assumed that the default exception handling (divide by zero, etc.) was used. An attempt was made to avoid restructuring the high-level language representations to take advantage of knowledge of the low-level structures except in the hand-optimized versions of the algorithms. Optimizations obvious at
the high-level were used, such as the Widrow modification, avoiding references to array members where local variables could be used (as in the Lattice) and performing a multiplication once outside a loop instead of every time through the loop. It was assumed that the compiler would correctly select the addressing modes, literal length, use the increment instruction, etc. and in general take advantage of the facilities offered by the processor. This turned out to be a reasonable assumption.

For the purposes of this evaluation, single-precision integer and single-precision floating-point versions of the algorithms were coded. Table 3 shows the execution times for the Widrow and Table 4 shows execution times for the Lattice. The AAMP has equivalent instructions available for each data format. Because of this, the two versions were coded side by side, each with the correct form of the arithmetic instructions and proper length memory reference instructions. Execution statistics for the fractional data format are identical to the integer version and was not coded again. Another possibility which offers a compromise between the fixed-point and floating-point is the double-precision fixed-point format. Because of the word length, double-precision fixed-point data transfers are the same as single-precision floating-point transfers. Double-precision fixed-point execution statistics have been estimated from single-precision floating-point execution estimates. Shorter execution times of the double-precision fixed-point arithmetic instructions were taken into account. Extended-precision floating-point implementations were not investigated because they would execute much more slowly and the added precision seemed unnecessary.
Table 3. AAMP Widrow Execution Times
(all times in microseconds)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Multiply</th>
<th>Add/</th>
<th>Stack</th>
<th>Other</th>
<th>Total</th>
<th>Samples /sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Subtract</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixed pt:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>237.50</td>
<td>19.25</td>
<td>507.60</td>
<td>588.20</td>
<td>1352.55</td>
<td>739</td>
</tr>
<tr>
<td>Modified</td>
<td>237.50</td>
<td>19.25</td>
<td>345.60</td>
<td>402.10</td>
<td>1004.45</td>
<td>996</td>
</tr>
<tr>
<td>Optimized</td>
<td>237.50</td>
<td>19.25</td>
<td>0.00</td>
<td>411.25</td>
<td>668.05</td>
<td>1497</td>
</tr>
<tr>
<td>Double-precision fixed pt:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>757.50</td>
<td>26.25</td>
<td>907.20</td>
<td>705.15</td>
<td>2396.10</td>
<td>417</td>
</tr>
<tr>
<td>Modified</td>
<td>757.50</td>
<td>26.25</td>
<td>691.20</td>
<td>486.40</td>
<td>1961.35</td>
<td>510</td>
</tr>
<tr>
<td>Optimized</td>
<td>757.50</td>
<td>26.25</td>
<td>0.00</td>
<td>566.25</td>
<td>1350.00</td>
<td>741</td>
</tr>
<tr>
<td>Floating-point:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>957.50</td>
<td>272.05</td>
<td>907.20</td>
<td>712.90</td>
<td>2849.65</td>
<td>351</td>
</tr>
<tr>
<td>Modified</td>
<td>957.50</td>
<td>272.05</td>
<td>691.20</td>
<td>494.15</td>
<td>2414.90</td>
<td>414</td>
</tr>
<tr>
<td>Optimized</td>
<td>957.50</td>
<td>272.05</td>
<td>0.00</td>
<td>574.00</td>
<td>1803.55</td>
<td>554</td>
</tr>
</tbody>
</table>

Table 4. AAMP 16-Stage Lattice Execution Times
(all times in microseconds)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Multiply/</th>
<th>Add/</th>
<th>Stack</th>
<th>Other</th>
<th>Total</th>
<th>Samples /sec</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Divide</td>
<td>Subtract</td>
<td></td>
<td></td>
<td>Total</td>
<td></td>
</tr>
<tr>
<td>Fixed pt:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>772.80</td>
<td>52.80</td>
<td>345.60</td>
<td>758.10</td>
<td>1929.30</td>
<td>518</td>
</tr>
<tr>
<td>Optimized</td>
<td>772.80</td>
<td>52.80</td>
<td>0.00</td>
<td>648.70</td>
<td>1474.30</td>
<td>678</td>
</tr>
<tr>
<td>Double-precision fixed pt:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>2436.80</td>
<td>72.00</td>
<td>1152.00</td>
<td>1001.25</td>
<td>4662.05</td>
<td>214</td>
</tr>
<tr>
<td>Optimized</td>
<td>2436.80</td>
<td>72.00</td>
<td>0.00</td>
<td>1134.75</td>
<td>3643.55</td>
<td>274</td>
</tr>
<tr>
<td>Floating-point:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard</td>
<td>3073.60</td>
<td>756.80</td>
<td>1152.00</td>
<td>1009.00</td>
<td>5991.40</td>
<td>167</td>
</tr>
<tr>
<td>Optimized</td>
<td>3073.60</td>
<td>756.80</td>
<td>0.00</td>
<td>1142.50</td>
<td>4972.90</td>
<td>201</td>
</tr>
</tbody>
</table>
It should be pointed out that the integer versions of both algorithms make no provision for scaling. If needed, scaling operations should not seriously affect the performance of the algorithm. It is possible that using fractional notation could take care of the scaling problem, but this has not been investigated in any detail. Also, the AAMP automatically invokes exception handling for overflows and division by zero. The default exception handling should be adequate for most uses and requires very little execution time overhead. If necessary, however, user-supplied exception handling routines can be used at the cost of the time needed to transfer to, execute and return from the routines.

After the programs described above were coded and execution rates were calculated, a trip to Collins-Rockwell at Cedar Rapids, Iowa was arranged. With the cooperation of the Rockwell personnel, the Lattice and Widrow algorithms were coded and executed on their test equipment. Originally, Rockwell's PL/I and Ada-subset were to be used, but due to lack of time and accessibility only the Ada-subset was used. The object code produced is discussed under the appropriate sections below and more detail is provided in the "Performance measurements" section.

The viability of using compiled output for time-critical real-time signal processing depends on the efficiency of the generated object code. Another program was written to test the
compiler's optimizations; it consists of a number of structures that are commonly optimized, particularly those which appear quite often in signal processing applications. The listings for this program, ADATESTS, can be found in the appendices.

Widrow Algorithm Modification

While most of the Widrow algorithm's execution time can be attributed to multiplications, a significant amount of time is spent aligning the weight (b), input (f) and error (e) arrays. This has been done either with block moves of the arrays or through maintaining circulating buffers. The following is a description of a method which is simpler and more efficient to implement. The author came across this method by examining the algorithm closely and has not found any previous use of this method.

A common form of the Widrow algorithm is shown in Figure 6. An important point to note is that the summation in step 1 will be correct as long as all of the corresponding weight and input pairs are multiplied and summed, regardless of order. The same weight-sample pairs as in step 1 are used in the weight updating in step 3, with one weight updated at a time. Again, the new weights will be correct regardless of the order the updating process uses. In fact, the only time a particular member of f is needed is when the oldest sample is replaced with the new sample (in a circular buffer).
Note that as long as a pointer is maintained to the oldest sample in $f$, we need only concern ourselves with providing the proper pairings of samples and weights for steps 1 and 3. Usually, the sample array is advanced by one to simulate passage of time. Instead, the weight array can be "moved back" by one to create the same pairings of samples and weights. This turns out to be convenient since newly calculated weight values must be written into the array anyway. The updated weights are simply written into the correct position for the next iteration's pairings. Figure 8 illustrates this process.

Because the order is irrelevant as long as the pairings are correct, steps 1 and 3 can be efficiently performed by proceeding from one end of the arrays to the other. In assembly language, it is most efficient to go from the largest to smallest buffer addresses, terminating when the array index equals zero. The weight updating then requires only one additional memory transfer to complete the circulation. This eliminates the overhead needed for circulating pointers.

The pointer to the oldest sample circulates and thus must be checked, but this occurs only once per sample. Also, the index to the oldest value of $e$ is the same as the index into $f$, allowing a single index to be used for both purposes. Figure 9 illustrates the updating of the sample ($f$) and error ($e$) arrays. Finally, Figure 10 shows that the pairs match correctly after updating.

The results of the comparison between the block-move updates and the modified version show an improvement of 35% for the fixed
point version and 18% for the floating point version. The timing
difference is actually greater in the floating-point version
because the buffers are twice as large, but the proportion of the
total is less. One would expect that the savings would be less
dramatic on processors that have block-move instructions using
register pointers. Dwight Gordon's NSC800/hardware multiplier
evaluation [3] showed that block-moves represented approximately
10% of the total execution time, which is what would be expected.

Figure 8. Weight Updating Process

Note: The prime indicates variables for the next iteration.
discard old  insert new value  discard old  insert new value

[ -f(16) ] --|-- --> [  f( 1)' ]  [ -e(16) ] --|-- --> [  e( 1)' ]
[  f(15) ]-------[->f(16)' ]  [  e(15) ]-------[->e(16)' ]
[  f(14) ]-------[ f(15)' ]  [  e(14) ]-------[ e(15)' ]
[  f(13) ]-------[ f(14)' ]  [  e(13) ]-------[ e(14)' ]
[  f(12) ]-------[ f(13)' ]  [  e(12) ]-------[ e(13)' ]
[  f(11) ]-------[ f(12)' ]  [  e(11) ]-------[ e(12)' ]
[  f(10) ]-------[ f(11)' ]  [  e(10) ]-------[ e(11)' ]
[  f( 9) ]-------[ f(10)' ]  [  e( 9) ]-------[ e(10)' ]
[  f( 8) ]-------[ f( 9)' ]  [  e( 8) ]-------[ e( 9)' ]
[  f( 7) ]-------[ f( 8)' ]  [  e( 7) ]-------[ e( 8)' ]
[  f( 6) ]-------[ f( 7)' ]  [  e( 6) ]-------[ e( 7)' ]
[  f( 5) ]-------[ f( 6)' ]  [  e( 5) ]-------[ e( 6)' ]
[  f( 4) ]-------[ f( 5)' ]  [  e( 4) ]-------[ e( 5)' ]
[  f( 3) ]-------[ f( 4)' ]  [  e( 3) ]-------[ e( 4)' ]
[  f( 2) ]-------[ f( 3)' ]  [  e( 2) ]-------[ e( 3)' ]
[  f( 1) ]-------[ f( 2)' ]  [  e( 1) ]-------[ e( 2)' ]

Figure 9. Sample and Error Array Updating Scheme

Note: Only the oldest sample and error values are physically replaced; the rest are left in the same place and merely relabeled.

-> = pointer to oldest, ' = for next iteration

[  f(16) ] <--- [  b(16) ]  [  f( 1)' ] <--- [  b( 1)' ]
[  f(15) ] <--- [  b(15) ]  [  f(16)' ] <--- [  b(16)' ]
[  f(14) ] <--- [  b(14) ]  [  f(15)' ] <--- [  b(15)' ]
[  f(13) ] <--- [  b(13) ]  [  f(14)' ] <--- [  b(14)' ]
[  f(12) ] <--- [  b(12) ]  [  f(13)' ] <--- [  b(13)' ]
[  f(11) ] <--- [  b(11) ]  [  f(12)' ] <--- [  b(12)' ]
[  f(10) ] <--- [  b(10) ]  [  f(11)' ] <--- [  b(11)' ]
[  f( 9) ] <--- [  b( 9) ]  [  f(10)' ] <--- [  b(10)' ]
[  f( 8) ] <--- [  b( 8) ]  [  f( 9)' ] <--- [  b( 9)' ]
[  f( 7) ] <--- [  b( 7) ]  [  f( 8)' ] <--- [  b( 8)' ]
[  f( 6) ] <--- [  b( 6) ]  [  f( 7)' ] <--- [  b( 7)' ]
[  f( 5) ] <--- [  b( 5) ]  [  f( 6)' ] <--- [  b( 6)' ]
[  f( 4) ] <--- [  b( 4) ]  [  f( 5)' ] <--- [  b( 5)' ]
[  f( 3) ] <--- [  b( 3) ]  [  f( 4)' ] <--- [  b( 4)' ]
[  f( 2) ] <--- [  b( 2) ]  [  f( 3)' ] <--- [  b( 3)' ]
[  f( 1) ] <--- [  b( 1) ]  [  f( 2)' ] <--- [  b( 2)' ]

Figure 10. Sample-Weight Pairs Before and After Updating

Note: The pairings remain the same between the arrays.

' = for next iteration
Optimization for the AAMP

The following section discusses a few of the features of the AAMP that significantly affect the execution rates of programs. These factors can be dealt with through coding style and compiler optimization.

As a stack machine, the AAMP performs operations on the top elements of the stack, popping the arguments off and pushing the result. To speed this process, there is a provision to hold up to four of the top values on the stack in registers inside the processor itself. These registers are transparent to the programmer; transfers into and out of these registers are handled automatically by the processor. As elements are placed on the stack, they are put in the processor registers. If a new value is to be pushed onto the top of the stack when all processor registers are full, the bottom element is moved out into memory to free a register. Later, when the top elements of the stack are removed by operations, the registers become empty and the values in memory must be brought back into the registers. Thus, each time the stack grows to more than four elements, stack updating must take place. This storage and later retrieval of stack elements is handled automatically by the processor but is very costly in terms of processing time. Table 5 compares the stack updating action with other operations. Each time a stack element is moved from the registers to memory it must later be returned to the registers. Therefore, the stack updating time in this report is the combined storage and retrieval times.
Table 5. Execution Times of Some Common Operations (all times in microseconds)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Execution time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point multiply</td>
<td>4.75</td>
</tr>
<tr>
<td>Floating-point multiply</td>
<td>19.15</td>
</tr>
<tr>
<td>Fixed-point addition</td>
<td>0.55</td>
</tr>
<tr>
<td>Floating-point addition</td>
<td>7.75</td>
</tr>
<tr>
<td>Local reference, 16-bit</td>
<td>0.85</td>
</tr>
<tr>
<td>Stack update</td>
<td>3.60</td>
</tr>
</tbody>
</table>

For the preliminary performance estimates, it was assumed that the processor displaced into memory only enough registers to make room for the element being pushed onto the stack. Also, the processor retrieved only enough elements from memory to perform the current instruction. This is the optimum approach since it avoids unnecessary transfers and seemed to be the logical way to implement the stack updating scheme. Tables 3 and 4 demonstrate the significance of the stack updating in the execution time. If any other scheme is used, it could degrade performance significantly.

Due to "real estate" problems encountered in the implementation of the AAMP on a single chip, it was not possible to have optimal stack-updating. Instead, Rockwell used the mapping of opcodes to internal stack status shown in Table 6. This mapping is nearly optimal with the non-optimal instructions listed in Table 7 and discussed below.
Table 6. Opcode to stack update mapping

There are three types of non-optimal stack-updates:

1) Unnecessary - a stack update which provides a range of stack elements that is more restrictive than required by the instruction. There is a 0.5 probability that this action must be reversed in subsequent instructions, causing inefficiency.

2) Inadequate - a stack update which provides a range of stack elements that is less restrictive than required by the instruction. The instruction must then continue the stack updating (if needed) to meet its more restrictive requirements. This type is harmless.

3) Destructive - a stack update which provides a condition that must be immediately corrected before the instruction can be executed.

Key to symbols:
( ) = a harmless state
<- = stack not used by instruction
! ? = stack-update which must be immediately undone
# = optimums are 0-0 for PROCID = CENV = 0
     0-4 for PROCID = CENV <> 0
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Mnemonic</th>
<th>Implemented</th>
<th>Optimum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1B</td>
<td>INTE</td>
<td>0-3</td>
<td>0-4</td>
</tr>
<tr>
<td>1D</td>
<td>SKIPI</td>
<td>0-3</td>
<td>0-4</td>
</tr>
<tr>
<td>(1F)</td>
<td>CALLPI</td>
<td>0-3</td>
<td>0-0</td>
</tr>
<tr>
<td>20</td>
<td>NOP</td>
<td>0-2</td>
<td>0-4&lt;-</td>
</tr>
<tr>
<td>(23)</td>
<td>CALLI</td>
<td>0-2</td>
<td>0-0</td>
</tr>
<tr>
<td>(26)</td>
<td>LIT48</td>
<td>0-2</td>
<td>0-1</td>
</tr>
<tr>
<td>28</td>
<td>LIT4B.8</td>
<td>0-2</td>
<td>0-3</td>
</tr>
<tr>
<td>29</td>
<td>LIT4B.9</td>
<td>0-2</td>
<td>0-3</td>
</tr>
<tr>
<td>2A</td>
<td>LIT4B.A</td>
<td>0-2</td>
<td>0-3</td>
</tr>
<tr>
<td>2B</td>
<td>LIT4B.B</td>
<td>0-2</td>
<td>0-3</td>
</tr>
<tr>
<td>2C</td>
<td>LIT4B.C</td>
<td>0-2</td>
<td>0-3</td>
</tr>
<tr>
<td>2D</td>
<td>LIT4B.D</td>
<td>0-2</td>
<td>0-3</td>
</tr>
<tr>
<td>2E</td>
<td>LIT4B.E</td>
<td>0-2</td>
<td>0-3</td>
</tr>
<tr>
<td>2F</td>
<td>LIT4B.F</td>
<td>0-2</td>
<td>0-3</td>
</tr>
<tr>
<td>(58)</td>
<td>TRAP</td>
<td>1-4</td>
<td>1-1</td>
</tr>
<tr>
<td>(5D)</td>
<td>CALL</td>
<td>1-4</td>
<td>1-1</td>
</tr>
<tr>
<td>(5E)</td>
<td>CALLP</td>
<td>1-4</td>
<td>1-1</td>
</tr>
<tr>
<td>65</td>
<td>CVTSD</td>
<td>2-2</td>
<td>1-3</td>
</tr>
<tr>
<td>66</td>
<td>LOCU</td>
<td>2-2</td>
<td>1-3</td>
</tr>
<tr>
<td>67</td>
<td>REFD</td>
<td>2-2</td>
<td>1-3</td>
</tr>
<tr>
<td>68</td>
<td>REFDC</td>
<td>2-2</td>
<td>1-3</td>
</tr>
<tr>
<td>69</td>
<td>REFDXI</td>
<td>2-2</td>
<td>1-3</td>
</tr>
<tr>
<td>6A</td>
<td>DUP</td>
<td>2-2</td>
<td>1-3</td>
</tr>
<tr>
<td>6C</td>
<td>CVTDJE</td>
<td>2-2</td>
<td>2-3</td>
</tr>
<tr>
<td>6D</td>
<td>CVTFFE</td>
<td>2-2</td>
<td>2-3</td>
</tr>
<tr>
<td>6E</td>
<td>REFTXI</td>
<td>2-2</td>
<td>1-2</td>
</tr>
<tr>
<td>6F</td>
<td>REFTX</td>
<td>2-2</td>
<td>2-3</td>
</tr>
<tr>
<td>74</td>
<td>REFTI</td>
<td>2-2</td>
<td>0-1     !?</td>
</tr>
<tr>
<td>75</td>
<td>REFT</td>
<td>2-2</td>
<td>1-2</td>
</tr>
<tr>
<td>76</td>
<td>REFTC</td>
<td>2-2</td>
<td>1-2</td>
</tr>
<tr>
<td>77</td>
<td>REFTLE</td>
<td>2-2</td>
<td>0-1     !?</td>
</tr>
<tr>
<td>78</td>
<td>REFTU</td>
<td>2-2</td>
<td>2-3</td>
</tr>
<tr>
<td>79</td>
<td>DUPT</td>
<td>2-2</td>
<td>3-4     !?</td>
</tr>
<tr>
<td>7A</td>
<td>INCSELE</td>
<td>2-2</td>
<td>0-4     &lt;-</td>
</tr>
<tr>
<td>7B</td>
<td>INCSEI</td>
<td>2-2</td>
<td>0-4     &lt;-</td>
</tr>
<tr>
<td>7C</td>
<td>INCSE</td>
<td>2-2</td>
<td>1-4</td>
</tr>
<tr>
<td>7D</td>
<td>DECSLE</td>
<td>2-2</td>
<td>0-4     &lt;-</td>
</tr>
<tr>
<td>7E</td>
<td>DECSI</td>
<td>2-2</td>
<td>0-4     &lt;-</td>
</tr>
<tr>
<td>7F</td>
<td>DECS</td>
<td>2-2</td>
<td>1-4</td>
</tr>
<tr>
<td>B7</td>
<td>POPD</td>
<td>3-4</td>
<td>2-4</td>
</tr>
<tr>
<td>B8</td>
<td>ARS</td>
<td>3-4</td>
<td>2-4</td>
</tr>
<tr>
<td>BD</td>
<td>EXCEPT0</td>
<td>3-4</td>
<td>#</td>
</tr>
<tr>
<td>BE</td>
<td>EXCEPT1</td>
<td>3-4</td>
<td>#</td>
</tr>
<tr>
<td>BF</td>
<td>EXCEPT2</td>
<td>3-4</td>
<td>#</td>
</tr>
<tr>
<td>F4</td>
<td>NOT</td>
<td>2-4</td>
<td>1-4</td>
</tr>
<tr>
<td>F8</td>
<td>CVTBIT</td>
<td>2-4</td>
<td>1-4</td>
</tr>
<tr>
<td>FE</td>
<td>HALT</td>
<td>2-4</td>
<td>0-4?&lt;-</td>
</tr>
</tbody>
</table>

Table 7. Instructions with non-optimum stack-updating
Examination of the assembly listings for the Widrow and Lattice algorithms led to the conclusion that the nonoptimum instructions did not have a significant effect on execution rate. The hand-compiled versions used the LIT4B, REFDXI, DUP and DECSL instructions while the Ada-subset compiler output used only the LIT4B instruction. Nonoptimal instructions would be used quite frequently, however, if triple words were being accessed but would not be very significant compared with the accompanying relatively slow extended floating-point operations.

For hand-compilation of algorithms for the performance estimates, it was assumed that the compiler was not "smart" enough to generate object code that would not cause stack updating. This turned out to be the case with the Ada-subset compiler. There are two methods of avoiding stack updating: 1) rearranging arguments and 2) storing intermediate results in temporary locations. These two methods are complementary, and both have been used in the hand-optimized versions of the algorithms.

Rearranging arguments is the most desirable way of avoiding stack updating because it does not require any extra instructions. This rearranging of arguments is commonly used by owners of RPN calculators when equations are entered beginning with the innermost parenthetical expression. Also, multiplication and division terms are evaluated before addition and subtraction terms whenever possible. This does not always work, as in the following case:

\[ F = A \times B + C \times D \]
In this case, both product terms must be evaluated before they can be summed. As the second multiplication is about to take place, the product A*B, C and D are on the stack. If being performed with floating-point numbers, each variable takes up two storage locations, forcing the stack to have six members (or more depending on previous actions). This causes at least two stack updates.

The second way of avoiding stack updates is to store intermediate answers in temporary locations. In the example above, the product A*B would be stored in a temporary location until C*D had been evaluated. Then the value would be retrieved and the summation could take place. This method is only economical if the temporary location can be efficiently accessed. The addressing mode must use immediate data (to avoid pushing more data on the stack!) or, preferably, the local addressing mode.

Both of the above methods were used in the optimized versions of the algorithms. Tables 3 and 4 show that the time spent on "other" operations increased when the stack updates were taken out. This was due to the added overhead from the temporary variables.

A second important feature of the AAMP is its addressing modes. The methods of addressing array elements are of particular interest for the evaluation of signal processing algorithms. The AAMP provides three addressing modes which can be used for accessing array members: 1) Indexed, 2) Indexed Immediate, and 3) Constant Offset.

The Indexed addressing mode adds the top two elements of the
stack to form an address. The top element of the stack represents the base address of the array while the element next to the top represents the index into the array. Before adding, the index is multiplied by two for double-length word accesses and by three for triple-word accesses. Through use of this addressing mode, an array element may be specified by index regardless of the element's word length.

The Indexed Immediate addressing mode is the same as the indexed mode except that the base address of the array is in immediate data in the instruction instead of on the top of the stack.

The third addressing mode is Constant Offset. This mode works essentially the same as Indexed Immediate except that the base and offset are added together without taking into account the word-length of the data being accessed. In other words, the two numbers are added together without any multiplication of the offset.

These are not the only methods of referencing arrays, but they are the most convenient. Other methods include more complicated calculation of offset and pre-calculating addresses when the index into the array being used for a particular reference is constant.

The AAMP's addressing modes are very convenient for referencing data in tables and other common structures, but there are some actions that are awkward at best for the AAMP to perform. In particular, block moves and other actions which require the use of one or more pointers fairly intensively are
awkward. The processor is fast enough to make these actions reasonable, but the processor will yield better performance when other programming techniques are used to perform these tasks. This was the factor that led to the modified version of the Widrow algorithm.

Because of the nature of array addressing in the AAMP, the array's base address must be obtained from either the stack or the accessing instruction's immediate data bytes. The array index must be taken from the top of the stack. To get the index onto the top of the stack, another memory access must have taken place. If an array member is to be accessed more than once, it becomes advantageous to store the member in a temporary local location during its first access. From the next reference until the last reference, the locally stored variable is accessed. Quite often, the last access involving a variable is to assign a new value to it before the next iteration is begun. This allows the new value to be written directly to the actual array storage location only.

Another possible optimization is in the case of a loop that references the k+1 element during the kth iteration. If k+1 is used more than once in the loop, k+1 might be calculated once and stored for future references, but a better solution at the assembly language level is to specify an offset array base so that when the Indexed address is calculated, it automatically includes the +1 offset. This turned out to be one of the few optimizations the Ada-subset compiler made.

The most efficient storage is in a 16 word Local area. These locations should be used for the most frequently used
variables. For example, during a block move, the pointers must be continually retrieved from memory. If the Local addressing mode is used to access the pointers, significant improvements in performance can be achieved.

Another significant factor in signal processing programs is how efficiently loop structures can be implemented. The AAMP has a pair of instructions, DO and ENDO, for that express purpose. Before DO can be executed, the information necessary for control of the loop must be put on the stack: loop variable address, initial value, final value and increment value. The DO instruction is then executed, initializing the variable and executing the loop. In the process, the initial loop value on the stack is replaced by the address of the beginning of the loop (the instruction following DO). At the end of the loop, the ENDO instruction performs the incrementing and comparison necessary to determine whether to execute the loop again or to exit. To do this, the four stack locations containing the information must all be brought into the registers. Note that the DO instruction is executed only once but the ENDO instruction is executed every time the loop is executed.

The assumption made when hand-compiling the algorithms was that the compiler would use these instructions to implement most if not all loop structures. This had a rather significant effect on the execution rates of the algorithms because of the large number of stack updates it caused. Since all four stack locations had to be brought into the processor for the ENDO instruction, any word placed onto the stack automatically led to
a stack-update. For large loops, this would not constitute a very significant part of the execution time, but for small loops the effect is quite significant. In the hand optimizations, the DO and ENDO instructions were not used but the actions were coded explicitly. This eliminates the need for four words of information to be stored on the stack during the loop and does not cause stack-updates. This change alone accounted for much of the improved performance in the hand-optimized versions of the algorithms. It was discovered that the Ada-subset compiler also discards the DO/ENDO instructions and codes the structures explicitly.

In the implementation of large signal processing programs, it is desirable if not necessary to partition the program into functions and subroutines or procedures. This partitioning offers the advantages of making the program easier to understand, maintain and modify. The following discusses the resources available in the AAMP to implement such partitioning.

The structures of functions, subroutines and procedures can all be implemented through use of the AAMP's CALL and RETURN instructions. These instructions are powerful because they allow the programmer to set up a local environment and to pass parameters using only a few instructions. Working with this mechanism is the instruction LOCNL (locate nonlocal) which will search through the process stacks until it finds the specified PROCID and locates the specified variable. This mechanism allows variables to be accessed from calling procedures without passing the variables as arguments or making the variables global. An example of where this would be useful is when a procedure which
has created a local array calls another procedure to manipulate the array. Using LOCNL, the called procedure can locate the array base and calculate the positions of the individual elements.

The chief advantages of the AAMP's procedure calling mechanism are the ease of programming and the flexibility it allows in the calling order of procedures. The alternative is to put a return address and arguments on the stack and SKIP to the subroutine. The subroutine would return by SKIPping to the return address left on the stack. The advantage of this alternative method is that it requires less execution time. The disadvantages are that more care must be taken in accessing variables and passing parameters and that new local storage is not set up for temporary variables used by the function. A break-even point can be calculated where the savings from local referencing set up by a procedure call become greater than the procedure call's overhead. Both single and double word references and assignments take one more microcycle and one half more instruction fetch for Local Extended than for Local addressing. The CALLI, LIT4A and RETURN instructions require 31 microcycles, 4.5 fetch cycles, 3 read cycles and 4 write cycles. In addition, 6 microcycles, 1 read cycle and 1 write cycle are required for each argument returned. Using these figures, 27 Local Extended accesses require the same amount of execution time as 27 Local accesses plus a procedure call with no arguments returned. Thus, 27 or more accesses make the procedure call economical. The other advantages, however, should encourage the
use of the CALL/RETURN mechanism more often than what is strictly economical.

Another possibility is that of doing away with the looping structures altogether and using "in-line" code. This is not a very graceful solution but should be considered, especially in light of the good code density characteristic of AAMP. Instead of a looping structure where each iteration processes a corresponding stage, in-line code would have a specific set of instructions for each stage. Instead of using the loop variable as an index into the arrays, sections of in-line code would contain the exact address of the element of interest, coded as bytes of immediate data.

Compiler optimizations

In the past, digital signal processing programs written for microprocessors have had to be hand-coded, with the utilization of as many assembly language "tricks" as possible. As a result, program efficiency was to a large degree a function of the programmer's cleverness. Unfortunately, there always seems to be a shortage of clever programmers and the cleverness must often later be unraveled. If feasible, the ability to program in high-level languages would greatly decrease the programming and maintenance time.

The efficiency of execution of compiled high-level languages seems to be dependent on three factors: 1) the ability of the compiler to manipulate the program without altering the semantics, 2) the mapping of compiled structures into machine
language instructions and 3) the execution speed of these instructions.

The first factor, the manipulation of the algorithm, is dependent upon the compiler. After the compiler has converted the program into an internal form, often an abstract syntax tree, this internal form can be rearranged and condensed. Rearrangement uses commutativity to produce a more efficient order of evaluation. The internal form can be condensed by the calculation of constant expressions, elimination of common subexpressions, etc. [6].

The second factor, the translation of compiled constructs into machine language instructions, is mostly dependent on the microprocessor's architecture. Most register-oriented microprocessors must use several machine language instructions to implement a complex operation such as a procedure call or a floating-point multiply. In particular, the allocation of their registers is critical to performance. Because of the AAMP's instruction set and stack architecture, high-level languages map quite directly into machine language instructions. Another way to state this is to say that AAMP programs exhibit good code densities. Stack machines have no register allocation problems but instead must strive to keep few arguments on the stack. This is somewhat less of an optimization problem than register allocation.

The third factor, the execution speed of the instructions, depends on the technology of the implementation and appears to be quite adequate in the case of the AAMP. The specifics of this
can be found elsewhere.[7]

The potential efficiency of compiled high-level languages was first assessed when the Widrow and Lattice algorithms were compiled. To examine the first two factors more closely, a program titled ADATESTS was coded in both integer and floating-point versions. This program was then compiled by the ICSC Ada-subset compiler at Collins-Rockwell. The results of this test show directly only what had been implemented on this compiler but should indicate problems other compiler implementations might encounter. Many common structures and commonly optimized expressions were placed in this program:

- while, for and loop structures
- function and procedure calls
- commutative rearrangement of expressions
- optimization between statements
- constant expression evaluation
- loop invariant expressions
- use of the increment instruction

Examination of the compiled object code revealed little optimization but did reveal an efficient translation of high-level structures into machine language instructions. Exceptions found were the DO/ENDO instructions discussed previously and the DUP and INC instructions. The compiler, however, did calculate a constant expression in the integer version. Also, in the Lattice and Widrow programs, Local Extended addressing was used to access specific array elements by computing the address of the element at compile time.

The optimizations not present in the Ada-subset compiler
are practical but had not yet been added due to lack of time. Another pass could be added to the compiler to take the intermediate code macros it produces and optimize them before the macro assembler generates object code. With this, the compiler's output would be very close to that produced by a skilled programmer. Optimization on the programmer's part would then be performed only by simplifying the high-level representation.
Performance measurements

Table 8 shows the total numbers of various types of cycles for each of the versions of the algorithms coded. These numbers were derived from the program listings and Rockwell's "AAMP Instruction Execution Statistics" document [7]. These totals are provided here to show how the data shown in earlier tables were arrived at and to allow performance calculations for various memory speeds and processor clock speeds. The following equations were supplied in Rockwell documents to calculate execution times. The following section will develop a variant of this equation which was used for the execution rate estimations. Briefly, the equation from Rockwell does not take into account synchronization.

\[ T_e = N_c \times T_c + N_f \times (T_f + (S+3) \times T_c/4) + N_r \times (T_r + (S+3) \times T_c/4) + N_w \times (T_w + (S+2) \times T_c/4) + N_b \times T_b \]

where:

- \( T = \) time
- \( N = \) number of actions
- \( S = \) set-up time configuration of the processor
- \( e = \) total execution
- \( c = \) internal cycles
- \( f = \) instruction fetches
- \( r = \) data reads
- \( w = \) data writes
- \( b = \) bus cycles
Table 8 Microcycles Required by the AAMP

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cycles</th>
<th>Fetches</th>
<th>Reads</th>
<th>Writes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed Point:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard Lattice</td>
<td>7311</td>
<td>735</td>
<td>594</td>
<td>228</td>
</tr>
<tr>
<td>Optimized Lattice</td>
<td>5531</td>
<td>581</td>
<td>482</td>
<td>164</td>
</tr>
<tr>
<td>Floating Point:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Standard Lattice</td>
<td>26270</td>
<td>785</td>
<td>1107</td>
<td>566</td>
</tr>
<tr>
<td>Optimized Lattice</td>
<td>21818</td>
<td>791</td>
<td>835</td>
<td>405</td>
</tr>
</tbody>
</table>

Since the AAMP is being evaluated for use on small systems, bus arbitration is unnecessary and $T_b = 0$, canceling the last term in the equation. Rockwell has done all of its specifications using a 20 MHz external clock and seems to be getting parts to run at that speed or better, so 20 MHz was used for this study. The microcycle clock frequency is the external
clock frequency divided by 4; thus, $T_c = 200$ ns. The fetch, read and write times depend on the memory used and on the method of generating control signals. To be conservative, 200 ns was allowed for each write and 250 ns for each fetch and read cycle. This could represent any one of the following conditions:

1) $T_w = T_r = T_f = 100$ ns, $S = 0$.
2) $T_w = T_r = T_f = 50$ ns, $S = 1$.
3) $T_w = T_r = T_f = 0$ ns, $S = 2$.

These seem to represent a wide variety of set-up times which should surely allow a common RAM to be used. It is possible that such a system will be able to run without wait-states ($S=0$, $T_f=0$, $T_r=0$ and $T_w=0$), but this needs to be investigated further.

To check the validity of the estimates, the programs were written in Ada and executed on a Rockwell development system at Collins-Rockwell in Cedar Rapids, Iowa. The output of the Ada-subset front-end is in the form of macro-instructions for a general stack machine, which is then translated into machine instructions for either the VAX, 8086 or in this case, the AAMP. The object code output was down-loaded into the AAMP test equipment and executed. A Hewlett-Packard logic analyzer was used to monitor execution rates of the programs.

Because there were no analog-to-digital or digital-to-analog converters available on the AAMP development system, memory locations (variables in the high-level notation) were read from and written to respectively. This should approximate memory-mapped real devices except for the lack of control signals and possible overflows and underflows due to non-varying input.
(unchanged memory contents) into the adaptive digital filter. The algorithms have been checked by several people and appear to be correct, but have not been run with actual data.

In the Ada-subset coding, a **pragma** was used to instruct the compiler to generate code which does not check for array bounds errors during execution. This checking would be very costly when the number of array references is taken into account. Including this pragma increases the execution rate significantly but puts the burden on the programmer of guaranteeing correctness of references. In these small programs, this represented no problem. In larger programs, the pragma could be omitted until the program is debugged and then inserted to increase execution speed.

Table 9 compares estimated and measured sampling rates for the Widrow and Lattice algorithms. The measured values were obtained during the April 16-17 visit to the Rockwell facility in Cedar Rapids, Iowa. These algorithms were coded in Ada and were executed on a test system.

The timing differences between the estimated and measured values are due primarily to coding differences. To check the timing estimates, the following major coding differences were taken into account. First, the Ada-subset compiler does virtually no optimization. In the Widrow, a multiplication that was moved outside of a loop in the estimated version was left inside in the measured version. The Ada-subset compiler did not use the DO/ENDO instructions, and therefore produced fewer stack-updates and faster execution. These differences are shown in
Tables 10 and 11. Some differences in execution time remain unaccounted for but probably would not be if all coding differences were reconciled. Also, floating-point instruction times vary according to the data used.
Table 9. Estimated vs measured execution rates

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Timing Parameters</th>
<th>Estimated (20 MHz)</th>
<th>Actual (20 MHz)</th>
<th>Actual (30 MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Widrow, integer</td>
<td>s=0</td>
<td>780</td>
<td>826</td>
<td>1156</td>
</tr>
<tr>
<td></td>
<td>s=1</td>
<td>773</td>
<td>766</td>
<td>1140</td>
</tr>
<tr>
<td></td>
<td>s=2</td>
<td>698</td>
<td>741</td>
<td>1042</td>
</tr>
<tr>
<td></td>
<td>s=3</td>
<td>693</td>
<td>694</td>
<td>1031</td>
</tr>
<tr>
<td>Widrow, floating</td>
<td>s=0</td>
<td>357</td>
<td>422</td>
<td>606</td>
</tr>
<tr>
<td></td>
<td>s=1</td>
<td>354</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s=2</td>
<td>335</td>
<td>392</td>
<td>565</td>
</tr>
<tr>
<td>Lattice, integer</td>
<td>s=0</td>
<td>488</td>
<td>481</td>
<td>625</td>
</tr>
<tr>
<td></td>
<td>s=1</td>
<td>483</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s=2</td>
<td>446</td>
<td>427</td>
<td>617</td>
</tr>
<tr>
<td>Lattice, floating</td>
<td>s=0</td>
<td>168</td>
<td>176</td>
<td>258</td>
</tr>
<tr>
<td></td>
<td>s=1</td>
<td>166</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s=2</td>
<td>161</td>
<td>168</td>
<td>245</td>
</tr>
</tbody>
</table>

All measured times use XAQ=XRQ, BG=BR and Tb<0 ns.
During this benchmarking, the use of an odd number of set-up cycles \((S)\) caused erratic measurements. This problem was due to the bus delays in the test system and a synchronization action of the AAMP which is discussed below.

Although the AAMP interfaces with external devices in an asynchronous manner, the signals are synchronized internally. Figures 11a and 11b contain timing diagrams showing this synchronization for read and write cycles respectively. A bus cycle begins in the middle of the 5 MHz microcycle clock and stops the microcycle clock until the bus cycle is complete. The microcycle clock is then restarted and continues the second half of the microcycle.

When the microcycle clock is stopped, the AAMP attempts to assert Bus Request. Bus Request can only become active when Bus Grant, Transfer Acknowledge and Transfer Error are inactive. In this manner, the AAMP will not disrupt other processors which might be using the bus or use a bus which has failed. External bus arbitration logic responds to Bus Request with a Bus Grant when appropriate. If there is only one processor on the bus, Bus Request and Bus Grant can be tied together, by-passing the bus arbitration logic. When received, however, Bus Grant is synchronized internally so that for a 20 MHz clock, tying Bus Grant to Bus Request \((T_b = 0)\) gives no time improvements over a bus acquisition time of slightly less than one clock cycle \((T_b < 50 \text{ ns})\).
Figure 11a. Read Cycle Synchronization

Figure 11b. Write Cycle Synchronization
When Bus Grant has been received, the address, data and status line drivers are enabled immediately. These signals are then given time to propagate through the bus interface before a Transfer Request is asserted. This time comes from the Bus Grant synchronization delay plus one clock cycle for reads, or plus two clock cycles for writes. The SI and SO pins provide a means of externally selecting an additional set-up time of from zero to three cycles. Thus, if Bus Request is tied to Bus Grant and $S0 = S1 = 0$, there will be nearly two clock cycles (100 ns) from Bus Request to Transfer Request for a read and three clock cycles (150 ns) for a write.

The device being accessed is responsible for generating a Transfer Acknowledge in response to a Transfer Request, allowing itself enough time to operate correctly. Transfer Acknowledge is, however, synchronized internally with the 10 MHz clock, probably to ensure the microcycle clock restarts correctly. Because of this synchronization, there must be an integer number of 10 MHz clock cycles and thus an even number of 20 MHz clock cycles between Bus Request and the internally synchronized Transfer Acknowledge.

The microcycle clock is restarted when the synchronized Transfer Acknowledge is received in the case of a write, or after an additional 10 MHz clock cycle in the case of read. Transfer Request and the other address, data and status lines remain active until the end of the microcycle (100 ns after the clock is restarted). Hold and Bus Request remain active until the middle of the next microcycle, the point where another bus transaction could begin. Because of this, the processor can make consecutive
bus transactions without relinquishing the bus, thus eliminating the bus arbitration logic delay. The minimum, however, is the same as the case where Bus Request is tied to Bus Grant due to synchronization.

The key to the previously mentioned problem with an odd number of set-up cycles is that because of the 10 MHz synchronization, the Transfer Request assertion is delayed without lengthening the total bus transaction time. The amount of time Transfer Request is active is thus shortened. Usually, Transfer Request is used to select the memory device and erratic operation may result if it is not selected for a sufficient amount of time.

The above analysis is summarized in Figure 12a in the form of a timing worksheet. Note that Transfer Request is active from the end of the set-up time until the end of the microcycle. Applying the worksheet to the conditions that existed for the benchmarking shows that at 20 MHz and S1,S0 = 0, Transfer Request is active six clock cycles (300 ns) for a read and three clock cycles (150 ns) for a write. With S1 = 0 and S0 = 1 (one set-up cycle), Transfer Request is shortened to five clock cycles (250 ns) for a read and lengthened to four cycles (200 ns) for a write. The shortened read select in combination with bus delays most probably caused the erratic operation.

Once the timing worksheet has been filled out, it is then possible to calculate the execution times of instructions. Figure 12b shows the numbers used for the performance estimates. Because of the predictable nature of translation from high-level
representations to AAMP instructions, it is possible to quickly estimate the execution rate of an algorithm from a high level language representation. This process is summarized in another worksheet provided in Figure 13. By counting the occurrences of various types of references, arithmetic operations and loops, the majority of operations have been accounted for and a reasonable estimate of the execution rate of an algorithm can be obtained. This quick estimate is for single and double precision fixed-point and single precision floating-point only. Less frequently used operations such as type conversion are not included. By far the most important operation omitted is the stack-updating. The resulting estimate assumes that the coding was such that no stack-updating occurred. As discussed earlier, this could be brought about by making the compiler optimize more or by hand-coding the program. Otherwise, a hand-optimized version will likely yield only small amounts of improvement unless high-level optimizations such as loop invariants are ignored.

The original performance estimates used the equations supplied by Rockwell in [7] which failed to take into account the synchronization action. These estimates have since been recalculated to reflect the correct timing using the equations at the bottom of Figure 13.

Another undocumented feature of the AAMP is a limited prefetch feature. A single portion of the AAMP's microinstruction word controls either its shift registers or its bus cycle logic. During long instructions which are not performing any shifts, the instruction word containing the next opcode can be fetched if it is not already in the upper byte of
the word in the instruction latch. This prefetching does not appear to increase the execution rate because the opcode fetch microcycle must be performed by the instruction anyway. The only difference is that the time for the bus transaction is taken during a different microcycle.
### Figure 12a. AAMP Timing Worksheet

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Request to Bus Grant</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Set-up overhead</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Set-up cycles (0-3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transfer Request to Transfer Acknowledge</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SUBTOTAL**

**Add 1 if SUBTOTAL is odd**

**10 MHz cycle for read**

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**TOTAL CYCLES**

\[ Cf = Cr = \quad Cw = \]

Note: * indicates that the number should be rounded up to the next highest integer; the minimum is one cycle.

### Figure 12b. Timing parameters used for estimates

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bus Request to Bus Grant</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Set-up overhead</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Set-up cycles (0-3)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Transfer Request to Transfer Acknowledge</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**SUBTOTAL**

**Add 1 if SUBTOTAL is odd**

**10 MHz cycle for read**

<table>
<thead>
<tr>
<th></th>
<th>read</th>
<th>write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**TOTAL CYCLES**

\[ Cf = Cr = \quad Cw = \]

Note: * indicates that the number should be rounded up to the next highest integer; the minimum is one cycle.
Figure 13. Execution Rate Estimate Worksheet

<table>
<thead>
<tr>
<th></th>
<th>Nc</th>
<th>Nf</th>
<th>Nr</th>
<th>Nw</th>
<th>Instances</th>
<th>Operation totals</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Reference Variable (non-indexed)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. fixed</td>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>3</td>
<td>0.5</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>3</td>
<td>0.5</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reference Fixed-index variable</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. fixed</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Reference Indexed variable</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. fixed</td>
<td>7</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>9</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Assign Variable (not indexed)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. fixed</td>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>3</td>
<td>0.5</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>3</td>
<td>0.5</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Assign Fixed-index variable</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. fixed</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Assign Indexed variable</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. fixed</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>9</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Constants</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. fixed</td>
<td>1</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Addition</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. fixed</td>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>3</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>38</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Figure 13 (continued). Execution Rate Worksheet

<table>
<thead>
<tr>
<th>Subtraction</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s.p. fixed</td>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>3</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>38</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiplication</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s.p. fixed</td>
<td>23</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>74</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>94</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Division</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s.p. fixed</td>
<td>27</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>d.p. fixed</td>
<td>78</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>s.p. floating</td>
<td>98</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Procedure/function Call and Return</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>plus for N return parameters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6N</td>
<td>0</td>
<td>N</td>
<td>N</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Loop structure                |       |       |       |       |       |
| initial                        | 15.5  | 5.5   | 1     | 1     |       |
| plus for N iterations          | 24.5  | 9     | 2     | 2     |       |

| If...then...else               |       |       |       |       |       |
| If branching                   | 2     | 1.5   | 0     | 0     |       |
| else branching                 | 2     | 1.5   | 0     | 0     |       |

| Goto                             |       |       |       |       |       |
| 2                                | 1.5   | 0     | 0     |       |

TOTAL CYCLES = Nc * 4 + Nf * Cf + Nr * Cr + Nw * Cw

Note: Cf, Cr and Cw come from the AAMP Timing Worksheet

ITERATIONS/SECOND = CYCLES/SECOND

TOTAL CYCLES
Table 10. Widrow coding differences

<table>
<thead>
<tr>
<th></th>
<th>Fixed-point</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nc</td>
<td>Nf</td>
<td>Nr</td>
<td>Nw</td>
<td>Nc</td>
<td>Nf</td>
<td>Nr</td>
<td>Nw</td>
<td>Nc</td>
<td>Nf</td>
<td>Nr</td>
<td>Nw</td>
<td>Nw</td>
</tr>
<tr>
<td>Estimated</td>
<td>4953</td>
<td>482.5</td>
<td>463</td>
<td>261</td>
<td>11765</td>
<td>506.5</td>
<td>708</td>
<td>441</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invariant</td>
<td>-30</td>
<td>-3</td>
<td>-1</td>
<td>-1</td>
<td>-106</td>
<td>-4</td>
<td>-2</td>
<td>-2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiplication</td>
<td>+384</td>
<td>+16</td>
<td>0</td>
<td>0</td>
<td>+1584</td>
<td>+24</td>
<td>+32</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop structure</td>
<td>+724</td>
<td>+376</td>
<td>+50</td>
<td>0</td>
<td>+724</td>
<td>+376</td>
<td>+50</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack updates</td>
<td>-2115</td>
<td>0</td>
<td>-141</td>
<td>-141</td>
<td>-2820</td>
<td>0</td>
<td>+188</td>
<td>+188</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Index ref.s</td>
<td>+280</td>
<td>+70</td>
<td>0</td>
<td>0</td>
<td>+140</td>
<td>+70</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack updates</td>
<td>+140</td>
<td>+70</td>
<td>0</td>
<td>0</td>
<td>+140</td>
<td>+70</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop var refs</td>
<td>0</td>
<td>-16</td>
<td>+32</td>
<td>0</td>
<td>-32</td>
<td>-48</td>
<td>+64</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack updates</td>
<td>-720</td>
<td>0</td>
<td>-48</td>
<td>-48</td>
<td>-960</td>
<td>0</td>
<td>-64</td>
<td>-64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constants</td>
<td>0</td>
<td>-24</td>
<td>+48</td>
<td>0</td>
<td>-48</td>
<td>-72</td>
<td>+64</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DUP not used</td>
<td>+160</td>
<td>+48</td>
<td>+64</td>
<td>0</td>
<td>+160</td>
<td>+48</td>
<td>+48</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New estimate</td>
<td>4230</td>
<td>962.5</td>
<td>413</td>
<td>119</td>
<td>11299</td>
<td>959</td>
<td>667</td>
<td>251</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 11. Lattice coding differences

<table>
<thead>
<tr>
<th></th>
<th>Fixed-point</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nc</td>
<td>Nf</td>
<td>Nr</td>
<td>Nw</td>
<td>Nc</td>
<td>Nf</td>
<td>Nr</td>
<td>Nw</td>
<td>Nc</td>
<td>Nf</td>
<td>Nr</td>
<td>Nw</td>
<td>Nw</td>
</tr>
<tr>
<td>Estimated</td>
<td>7311</td>
<td>735</td>
<td>594</td>
<td>228</td>
<td>26270</td>
<td>785</td>
<td>1107</td>
<td>566</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Loop structure</td>
<td>+247</td>
<td>+128</td>
<td>+17</td>
<td>0</td>
<td>+247</td>
<td>128</td>
<td>17</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Stack updates</td>
<td>-720</td>
<td>0</td>
<td>-48</td>
<td>-48</td>
<td>-960</td>
<td>0</td>
<td>-64</td>
<td>-64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constants</td>
<td>0</td>
<td>-24</td>
<td>+48</td>
<td>0</td>
<td>-48</td>
<td>-72</td>
<td>+64</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DUP not used</td>
<td>+160</td>
<td>+48</td>
<td>+64</td>
<td>0</td>
<td>+160</td>
<td>+48</td>
<td>+48</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>New estimate</td>
<td>7494</td>
<td>1015</td>
<td>691</td>
<td>196</td>
<td>25957</td>
<td>1017</td>
<td>1188</td>
<td>518</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Results and Conclusions

The AAMP offers significant improvements over previously evaluated processors due to its powerful instruction set and low power consumption. The execution statistics for previously evaluated processors are shown in Tables 12 and 13 for the Widrow and Lattice algorithms respectively. Unfortunately, information on digital signal processing execution by 16 bit microprocessors was not available. Other microprocessors which might compete with the AAMP, such as the Motorola's 68000 or National's 16032, are not currently available in low-power versions. AAMP's closest rival would probably be Intel's 80C86, which has to use an external multiplier board or an external 8087 math chip, thus increasing power consumption. Execution data comparing AAMP and these other microprocessors for other types of programs have been published [4]. The AAMP appears to live up to published performance claims and (unlike other recently released advanced microprocessors) no bugs were observed. Undocumented features found were: a limited instruction prefetching, nonoptimal stack updating and synchronized timing. None of these features appear to significantly affect performance.
Table 12 Widrow Processor Evaluations  
(all times in microseconds)

<table>
<thead>
<tr>
<th>Processor</th>
<th>Samples/second</th>
<th>Clock Rate (MHz)</th>
<th>#bits</th>
<th>Multiply type</th>
<th>time</th>
<th>ADC</th>
<th>Compute</th>
<th>Compute</th>
<th>Update weights</th>
<th>Compute</th>
<th>Update buffers</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z80</td>
<td>130</td>
<td>4</td>
<td>8</td>
<td>SW, fixed</td>
<td>147.25</td>
<td>21.50</td>
<td>3463.25</td>
<td>29.75</td>
<td>3621.75</td>
<td>198.00</td>
<td>351.00</td>
<td>7685.25</td>
</tr>
<tr>
<td>Z80</td>
<td>47</td>
<td>4</td>
<td>16</td>
<td>SW, fixed</td>
<td>554.25</td>
<td>14.25</td>
<td>9979.25</td>
<td>13.75</td>
<td>10156.25</td>
<td>600.25</td>
<td>369.50</td>
<td>21144.50</td>
</tr>
<tr>
<td>NSC800</td>
<td>296</td>
<td>4</td>
<td>8</td>
<td>HW, fixed</td>
<td>1194.75</td>
<td>28.35</td>
<td>1667.75</td>
<td>33.50</td>
<td>1674.25</td>
<td>91.50</td>
<td>363.25</td>
<td>3379.00</td>
</tr>
<tr>
<td>NSC800</td>
<td>273</td>
<td>4</td>
<td>16</td>
<td>HW, fixed</td>
<td>1430.75</td>
<td>34.75</td>
<td>702.75</td>
<td>37.00</td>
<td>1704.75</td>
<td>97.25</td>
<td>363.25</td>
<td>3665.75</td>
</tr>
<tr>
<td>8748</td>
<td>80</td>
<td>6</td>
<td>8</td>
<td>SW, fixed</td>
<td>252.5</td>
<td>77.5</td>
<td>5275.0</td>
<td>87.5</td>
<td>6065.0</td>
<td>957.5*</td>
<td>102.5</td>
<td>12565.0</td>
</tr>
<tr>
<td>ATMAC</td>
<td>11500</td>
<td>**</td>
<td>16</td>
<td>HW, fixed</td>
<td></td>
<td>1.00</td>
<td>11.90</td>
<td>0.56</td>
<td>39.69</td>
<td>11.02*</td>
<td>23.03</td>
<td>87.20</td>
</tr>
<tr>
<td>AAMP (standard)</td>
<td>739</td>
<td>20</td>
<td>16</td>
<td>HW, fixed</td>
<td>4.75</td>
<td>4.05</td>
<td>411.35</td>
<td>3.10</td>
<td>535.25</td>
<td>14.55</td>
<td>385.75</td>
<td>1354.05</td>
</tr>
<tr>
<td>AAMP (modified)</td>
<td>996</td>
<td>20</td>
<td>16</td>
<td>HW, fixed</td>
<td>4.75</td>
<td>4.05</td>
<td>411.35</td>
<td>3.10</td>
<td>553.80</td>
<td>14.70</td>
<td>13.26</td>
<td>1004.45</td>
</tr>
<tr>
<td>AAMP (optimized)</td>
<td>1497</td>
<td>20</td>
<td>16</td>
<td>HW, fixed</td>
<td>4.75</td>
<td>4.05</td>
<td>251.55</td>
<td>3.45</td>
<td>382.40</td>
<td>14.35</td>
<td>12.20</td>
<td>668.00</td>
</tr>
<tr>
<td>AAMP (standard)</td>
<td>351</td>
<td>20</td>
<td>24</td>
<td>HW, float</td>
<td>19.15</td>
<td>9.45</td>
<td>962.45</td>
<td>12.20</td>
<td>1343.35</td>
<td>50.45</td>
<td>471.75</td>
<td>2849.65</td>
</tr>
<tr>
<td>AAMP (modified)</td>
<td>414</td>
<td>20</td>
<td>24</td>
<td>HW, float</td>
<td>19.15</td>
<td>9.45</td>
<td>962.45</td>
<td>12.20</td>
<td>1362.90</td>
<td>50.60</td>
<td>15.26</td>
<td>2412.86</td>
</tr>
<tr>
<td>AAMP (optimized)</td>
<td>555</td>
<td>20</td>
<td>24</td>
<td>HW, float</td>
<td>19.15</td>
<td>9.45</td>
<td>641.05</td>
<td>12.75</td>
<td>1074.20</td>
<td>50.60</td>
<td>14.90</td>
<td>1802.95</td>
</tr>
</tbody>
</table>

* = includes Adaptive Threshold Detection, etc.  
** = 280 nsec short cycle, 350 nsec long cycle
<table>
<thead>
<tr>
<th>Action</th>
<th>fixed-point</th>
<th>floating-point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NSC800 (8-Stage)</td>
<td>AAMP</td>
</tr>
<tr>
<td>Multiply type</td>
<td>HW</td>
<td>HW</td>
</tr>
<tr>
<td>Multiply time</td>
<td>74.5</td>
<td>4.75</td>
</tr>
<tr>
<td>Input from A/D and init. loop</td>
<td>32.25</td>
<td>11.65</td>
</tr>
<tr>
<td>Loop:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute e(l+1)</td>
<td>122.0</td>
<td>22.60</td>
</tr>
<tr>
<td>Compute w(l+1)</td>
<td>119.25</td>
<td>11.80</td>
</tr>
<tr>
<td>Compute v(l)</td>
<td>364.50</td>
<td>38.15</td>
</tr>
<tr>
<td>Update weights</td>
<td>403.25</td>
<td>37.70</td>
</tr>
<tr>
<td>w(l)=w(l)</td>
<td>27.50</td>
<td>3.25</td>
</tr>
<tr>
<td>Loop overhead</td>
<td>-</td>
<td>6.10</td>
</tr>
<tr>
<td>Output</td>
<td>10.50</td>
<td>4.05</td>
</tr>
</tbody>
</table>

Totals:
8-Stage                     | 8334.75     | 972.50 | 743.05         | 3009.00 | 2500.75 |
16-Stage                     |             | 1929.30 | 1474.30         | 5991.40 | 4972.90 |

Sampling rate (in Hz):
8-Stage                     | 121         | 1028  | 1346            | 332   | 400   |
16-Stage                     |             | 518   | 678             | 167   | 201   |
After a preliminary learning period, the AAMP instruction set was quite easy to use. Coding was easy because of the relative symmetry of the instruction set, which is demonstrated by the side-by-side listings of the fixed-point and floating-point versions.

The lack of registers eliminates register usage optimization but introduces other optimization problems. First, the number of arguments on the stack must be limited to avoid stack-updates. Secondly, the Local memory locations must be used wisely for most efficient operation. Both of these problems, especially the former, can probably be dealt with more easily than register optimizations on other microprocessors. The ease of optimization and high-level language support structures indicate that compilers could produce code very nearly as efficient as that of assembly language programmers. While not very important for this application, the high code density characteristic of the AAMP is an indicator of the instruction set's efficiency. To further ensure efficiency, compilers could be modified to optimize structures common to signal processing programs.

An important factor in using a microprocessor is the availability and completeness of documentation. With the exception of timing specifications, the documentation provided is as good if not better than that available for most commercially marketed microprocessors. The lack of timing specifications was due to Rockwell's use of the processor primarily on a CPU board with a bus interface. The bus timing information was supplied, however, and the Rockwell personnel were cooperative in answering questions.
Most signal processing algorithms rely heavily on arrays of values, which can be efficiently implemented with index registers. The AAMP's lack of index registers is compensated for by its speed, but best performance can be achieved by avoiding structures such as block-transfers which require pointers.
Acknowledgements

This work was sponsored and funded by the Systems Engineering Division, Organization 5238, Sandia National Laboratories, Kirtland Air Force Base, Albuquerque, New Mexico.

The author would like to thank Dr. M.S.P. Lucas, Dr. M. Van Swaay and Dr. D.H. Hummels for serving as committee members. Special thanks go to John Rasure and David Hardin for their careful proofreading and support. Appreciation is also extended to D.W. Best and C.E. Kress of Rockwell for their cooperation.
References


Appendix A: Notes on Widrow and Lattice Listings

The following are lists of variables used in the hand-compiled Widrow and Lattice algorithms. These lists may help to explain the use of certain addressing modes in the algorithms. It was assumed that the necessary declarations were made in the high-level language to establish the named variables as local. Not shown but common to all implementations are the digital-to-analog and analog-to-digital converters which are assumed to be memory-mapped and are referenced through use of the Universal addressing mode. The universal addressing mode allows a complete address to be specified, which might make upper address line decoding easier.

Standard Lattice:

Local variables Non-local variables
-------------- ---------------------
present_w k( )
present_e wl( )
next_w v( )
next_e
*1 (loop variable)

Optimized Lattice:

Local variables Non-local variables
-------------- ---------------------
present_w k( )
present_e wl( )
next_w v( )
next_e
*1 (loop variable)
w1_l
k_l
**temp

70
Traditional Widrow:

<table>
<thead>
<tr>
<th>Local variables</th>
<th>Non-local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>b( )</td>
</tr>
<tr>
<td>g</td>
<td>f( )</td>
</tr>
<tr>
<td>*k (loop variable)</td>
<td>e( )</td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td></td>
</tr>
</tbody>
</table>

Modified Widrow:

<table>
<thead>
<tr>
<th>Local variables</th>
<th>Non-local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>b( )</td>
</tr>
<tr>
<td>g</td>
<td>f( )</td>
</tr>
<tr>
<td>*k (loop variable)</td>
<td>e( )</td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td></td>
</tr>
<tr>
<td>*ptr</td>
<td></td>
</tr>
</tbody>
</table>

Optimized Widrow:

<table>
<thead>
<tr>
<th>Local variables</th>
<th>Non-local variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>b( )</td>
</tr>
<tr>
<td>g</td>
<td>f( )</td>
</tr>
<tr>
<td>*k (loop variable)</td>
<td>e( )</td>
</tr>
<tr>
<td>e</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
</tr>
<tr>
<td>q</td>
<td></td>
</tr>
<tr>
<td>*ptr</td>
<td></td>
</tr>
<tr>
<td>**temp</td>
<td></td>
</tr>
</tbody>
</table>

Since only the execution times were important, no attempt was made to write initialization or exception handling routines, assign actual memory addresses to variables or include opcodes.

The single asterisk denotes variables that are always integer. The rest of the variables and arrays vary according to the number system in use. The double asterisk denotes variables that are used only in the floating-point or double-precision arithmetic.
fixed-point implementations.

An important point is that there are only 16 local memory words. These 16-bit words can store either 16 single-precision numbers, 8 floating-point (or double-precision fixed-point) numbers or some combination of the two. Fifteen of the sixteen available locations were used in the optimized Lattice, but some programs could require more, thus calling for careful coding or a good compiler to make the most of these locations.

A point that might not be immediately obvious is that the five instructions generated from a "do" statement initialize the loop and are only executed once. On the other hand, the "endo" instruction is executed each time the loop is executed.

Unlike the first three listings, the two optimized listings use low-level manipulations. The majority of the changes come from the following optimizations:

- Using a series of instructions to replace the "do" and "endo" instructions to avoid stack penalties.
- Changing an incrementing loop to a decrementing loop to allow easier testing for the final value (0).
- Rearranging arguments to avoid stack penalties.
- Using DUP to leave an argument on the stack for the next operation.
- Storing frequently referenced indexed variables into local memory for more efficient access.
- Using REFS in place of REFSXI because both do the same thing but REFS has a shorter instruction (fewer instruction fetches).
Standard Widrow listing

This listing approximates the output of a non-optimizing compiler for the algorithm given below. The program was translated quite directly and few assembly language modifications were made.

```
loop:  f = adc_in
       g = 0
       do k = 1,16
           g = g + b(k) * f(k)
       endo
       e = f - g
       c = v * e
       do k = 1,16
           b(k) = u * b(k) + c * f(k)
       endo
       q = q - e(16) + e
       dac_out = q * q
       do k = 1,15
           e(k+1) = e(k)
           f(k+1) = f(k)
       endo
       e(1) = e
       f(1) = f
       goto loop
```

Note: + = indicates a stack update
"loop:  \( f = \text{adc\_in} \)"

\[
\begin{align*}
\text{read ADC} & : \quad \text{LIT32} & 5.0 & 2.5 & \cdots & \text{LIT32} & 5.0 & 2.5 & \cdots \\
\text{convert to f.p.} & : \quad \text{REFSU} & 5.0 & 0.5 & 1 & \cdots & \text{CVTSD} & 2.0 & 0.5 & \cdots \\
\text{store f} & : \quad \text{ASNSL} & 2.0 & 0.5 & \cdots & \text{ASNDL} & 3.0 & 0.5 & 2 & 12. & 3.5 & 1 & 1 & 36. & 4.5 & 1 & 2
\end{align*}
\]

"g = 0"

\[
\begin{align*}
\text{get 0} & : \quad \text{LIT4} & 1.0 & 0.5 & \cdots & \text{LITDO} & 2.0 & 0.5 & \cdots \\
\text{store g} & : \quad \text{ASNSL} & 2.0 & 0.5 & \cdots & \text{ASNDL} & 3.0 & 0.5 & 2 & 3. & 1. & - & 1 & 5. & 1. & - & 2
\end{align*}
\]

"do k = 1,16"

\[
\begin{align*}
\text{loop var. addr.} & : \quad \text{LIT16} & 3.0 & 1.5 & \cdots & \text{LIT16} & 3.0 & 1.5 & \cdots \\
\text{initial value} & : \quad \text{LIT4} & 1.0 & 0.5 & \cdots & \text{LIT4} & 1.0 & 0.5 & \cdots \\
\text{final value} & : \quad \text{LIT8} & 2.0 & 1.0 & \cdots & \text{LIT8} & 2.0 & 1.0 & \cdots \\
\text{increment} & : \quad \text{LIT4} & 1.0 & 0.5 & \cdots & \text{LIT4} & 1.0 & 0.5 & \cdots \\
\text{DO} & : \quad \text{DO} & 10.0 & 2.0 & \cdots & \text{DO} & 10.0 & 2.0 & 1 & 17. & 5.5 & - & 1 & 17. & 5.5 & - & 1
\end{align*}
\]

"g = g + b(k) \times f(k)"

\[
\begin{align*}
\text{get g} & : \quad \text{+REFSL} & 2.0 & 0.5 & 1 & \cdots & \text{++REFDL} & 3.0 & 0.5 & 2 & - \\
\text{get k} & : \quad \text{+REFSL} & 2.0 & 0.5 & 1 & \cdots & \text{+REFSL} & 2.0 & 0.5 & 1 & - \\
\text{get b(k)} & : \quad \text{REFSXI} & 4.0 & 1.5 & 1 & \cdots & \text{+REFDXI} & 5.0 & 1.5 & 2 & - \\
\text{get k} & : \quad \text{+REFSL} & 2.0 & 0.5 & 1 & \cdots & \text{+REFSL} & 2.0 & 0.5 & 1 & - \\
\text{get f(k)} & : \quad \text{REFSXI} & 4.0 & 1.5 & 1 & \cdots & \text{+REFDXI} & 5.0 & 1.5 & 2 & - \\
\text{multiply} & : \quad \text{MPYI} & 23.0 & 0.5 & \cdots & \text{MPYF} & 95.0 & 0.5 & \cdots \\
\text{add} & : \quad \text{ADD} & 2.0 & 0.5 & \cdots & \text{ADDF} & 38.0 & 0.5 & \cdots \\
\text{store in g} & : \quad \text{ASNSL} & 2.0 & 0.5 & \cdots & \text{ASNDL} & 3.0 & 0.5 & 2 & 41. & 6. & 5 & 1 & 153. & 6. & 8 & 2
\end{align*}
\]

"endo"

\[
\begin{align*}
\text{end of loop} & : \quad \text{ENDO} & 9.0 & 1.0 & 1 & 1 & \cdots & \text{ENDO} & 9.0 & 1.0 & 1 & 1 \\
\text{9.} & : \quad \text{9.} & 1. & 1 & 1 & \cdots & \text{9.} & 1. & 1 & 1
\end{align*}
\]
iteration total | 50  | 7  | 6  | 2  | 162  | 7  | 9  | 3  |
loop total     | 800 | 112| 96 | 32 | 2592 | 112| 144| 48|

"e = f - g"
get f          | REFL | 2.0 | 0.5 | 1   | REFDL | 3.0 | 0.5 | 2   |
get g          | REFL | 2.0 | 0.5 | 1   | REFDL | 3.0 | 0.5 | 2   |
subtract       | SUB  | 2.0 | 0.5 | -   | SUBF  | 40.0| 0.5 | -   |
store e        | ASNSL| 2.0 | 0.5 | -   | ASNDL | 3.0 | 0.5 | - 2 |

----------

8.  2.  21

49.  2.  42

"c = v * e"
get v          | LIT16| 3.0 | 1.5 | -   | LIT32 | 5.0 | 2.5 | -   |
get e          | REFL | 2.0 | 0.5 | 1   | REFDL | 3.0 | 0.5 | 2   |
multiply       | MPYI | 23.0| 0.5 | -   | MPYF  | 95.0| 0.5 | -   |
store c        | ASNSL| 2.0 | 0.5 | -   | ASNDL | 3.0 | 0.5 | - 2 |

----------

30.  3.  11

106.  4.  22

"do k = 1,16"
loop var. addr. | LIT16| 3.0 | 1.5 | -   | LIT16 | 3.0 | 1.5 | -   |
initial value   | LIT4 | 1.0 | 0.5 | -   | LIT4  | 1.0 | 0.5 | -   |
final value     | LIT8 | 2.0 | 1.0 | -   | LIT8  | 2.0 | 1.0 | -   |
loop increment  | LIT4 | 1.0 | 0.5 | -   | LIT4  | 1.0 | 0.5 | -   |
DO              | 10.0| 2.0 | -   | 1    | DO    | 10.0| 2.0 | - 1 |

----------

17.  5.5 | 1

17.  5.5 | 1

"b(k) = u * b(k) + c * f(k)"
get u          | +LIT16| 3.0 | 1.5 | -   | ++LIT32| 5.0 | 2.5 | -   |
get k          | +REFL | 2.0 | 0.5 | 1   | +REFL  | 2.0 | 0.5 | 1   |
get b(k)       | REFSXI| 4.0 | 1.5 | 1   | +REFDXI| 5.0 | 1.5 | 2   |
multiply       | MPYI  | 23.0| 0.5 | -   | MPYF   | 95.0| 0.5 | -   |
get c          | REFL  | 2.0 | 0.5 | 1   | REFDL  | 3.0 | 0.5 | 2   |
get k          | +REFL | 2.0 | 0.5 | 1   | +REFL  | 2.0 | 0.5 | 1   |
get f(k)       | REFSXI| 4.0 | 1.5 | 1   | +REFDXI| 5.0 | 1.5 | 2   |
multiply       | MPYI  | 23.0| 0.5 | -   | MPYF   | 95.0| 0.5 | -   |
add            | ADD   | 2.0 | 0.5 | -   | ADDF   | 38.0| 0.5 | -   |
get k          | REFL  | 2.0 | 0.5 | 1   | REFL   | 2.0 | 0.5 | 1   |
store b(k+1)   | ASNSXI| 4.0 | 1.5 | - 1 | ASNDXI | 5.0 | 1.5 | 2   |

----------

71.  9.5 | 6 1

257. 10.5 | 9 2

"endo"
end the loop   ENDO  9.0 1.0 1 1  ENDO  9.0 1.0 1 1
               ---------   ---------
               9. 1. 1 1           9. 1. 1 1

iteration total  80  10.5 7 2
loop total      1280 168 112 32

"q = q - e(16) + e"

get q     REFSL  2.0 0.5 1 -
get 16    LIT8   2.0 1.0 -
get e(16) REFSXI 4.0 1.5 1 -
subtract  SUB    2.0 0.5 -
get e     REFSL  2.0 0.5 1 -
add       ADD    2.0 0.5 -
duplicate q DUP    1.0 0.5 -
store in q ASNSL  2.0 0.5 -
               ---------
               17. 5.5 3 1

"dac_out = q * q"

duplicate q DUP    1.0 0.5 -
square q     MPYI   23.0 0.5 -
convert from f.p. -  
write to DAC LIT32  5.0 2.5 -
               ---------
               34. 4. - 1

"do k = 1,15"

loop var. addr. LIT16  3.0 1.5 -
initial value  LIT4   1.0 0.5 -
final value    LIT8   2.0 1.0 -
loop increment LIT4   1.0 0.5 -
               ---------
               17. 5.5 - 1

"e(k+1) = e(k)"

get k      +REFSL  2.0 0.5 1 -
get e(k)   REFSXI  4.0 1.5 1 -
get k      +REFSL  2.0 0.5 1 -
get 1       +LIT4  1.0 0.5 -
               ---------
               76
ADD 2.0 0.5 -
store e(k+1) ASNSXI 4.0 1.5 - 1

ADD 2.0 0.5 -
store e(k+1) ASNSXI 4.0 1.5 - 1

f(k+1) = f(k)

get k REFSL 2.0 0.5 1 -
get f(k) REFSXI 4.0 1.5 1 -
get k REFSL 2.0 0.5 1 -
get 1 LIT4 1.0 0.5 - -
add ADD 2.0 0.5 - -
store f(k+1) ASNSXI 4.0 1.5 - 1

"endo"

end the loop ENDO 9.0 1.0 1 1

iteration total 39 11 7 3
loop total 585 165 105 45

"e(l) = e"

get e REFSL 2.0 0.5 1 -
get 1 LIT4 1.0 0.5 - -
store in e(l) ASNSXI 4.0 1.5 - 1

"f(l) = f"

get f REFSL 2.0 0.5 1 -
get 1 LIT4 1.0 0.5 - -
store in e(l) ASNSXI 4.0 1.5 - 1

"goto loop"

repeat LIT8N 2.0 1.0 - -
      SKIP 2.0 1.0 - -
<table>
<thead>
<tr>
<th></th>
<th>Total instr. cycles</th>
<th>Stack updates</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2838 482.5 322 120</td>
<td>2115 - 141 141</td>
<td>4953 482.5 463 261</td>
</tr>
<tr>
<td></td>
<td>7985 506.5 456 189</td>
<td>3780 - 252 252</td>
<td>11765 506.5 708 441</td>
</tr>
</tbody>
</table>
Standard Lattice listing

This listing approximates the output of a non-optimizing compiler for the algorithm given below. The program was translated quite directly and few assembly language modifications were made.

begin present_w = adc_input
present_e = present_w

loop do l = 0,15
    next_e = present_e - k(l) * wl(l)
    next_w = wl(l) - k(l) * present_e
    v(l) = beta * v(l) + betal * (present_e * present_e + wl(l) * wl(l))
    k(l) = k(l) + alpha * (next_e * wl(l) + present_e * next_w)/v(l)
    wl(l) = present_w
    present_w = next_w
    present_e = next_e
end

dac_out = present_e
goto begin

Note: + indicates a stack update.
"begin
  present_w = adc_input
  present_e = present_w"

| read ADC          | LIT32 5.0 2.5 - | LIT32 5.0 2.5 - |
| convert to f.p.   | REFSU 5.0 0.5 1-| REFSU 5.0 0.5 1-|
| duplicate data    | -              | CVTSD 2.0 0.5 -|
| store present_w   | DUPS 1.0 0.5 - | CVTDF 21.0 0.5 -|
| store present_e   | ASNSL 2.0 0.5 -1| DupD 2.0 0.5 -|

| present_w         | ASNSL 2.0 0.5 -1| ASNSL 3.0 0.5 -2|
| present_e         | ASNSL 2.0 0.5 -1| ASNSL 3.0 0.5 -2|
| 15.0 4.5 1 2      | 41.0 5.5 1 4    |

"loop
  do 1 = 0,15"

| loop var. addr.   | LIT16 3.0 1.5 - | LIT16 3.0 1.5 - |
| initial value     | LIT4 1.0 0.5 - | LIT4 1.0 0.5 - |
| final value       | LIT4 1.0 0.5 - | LIT4 1.0 0.5 - |
| increment         | LIT4 1.0 0.5 - | LIT4 1.0 0.5 - |
| DO                | 10.0 2.0 -1     | DO 10.0 2.0 -1  |

| 16.0 5.0 -1       | 16.0 5.0 -1     |

"next_e = present_e - k(l) * wl(l)"

| get present_e     | +REFSL 2.0 0.5 1- | ++REFDL 3.0 0.5 2 -|
| get 1             | +REFSL 2.0 0.5 1- | +REFSL 2.0 0.5 1- |
| get k(l)          | REFSXI 4.0 1.5 1-| +REFDXI 5.0 1.5 2-|
| get 1             | +REFSL 2.0 0.5 1- | +REFSL 2.0 0.5 1- |
| get wl(l)         | REFSXI 4.0 1.5 1-| +REFDXI 5.0 1.5 2-|
| multiply          | MPYI 23.0 0.5 - | MPYF 95.0 0.5 - |
| subtract          | SUB 2.0 0.5 - | SUBF 40.0 0.5 - |
| store next_e      | ASNSL 2.0 0.5 -1| ASNSL 3.0 0.5 -2|

| 41.0 6.0 5 1      | 155.0 6.0 8 2   |

"next_w = wl(l) - k(l) * present_e"

| get 1             | REFSL 2.0 0.5 1- | REFSL 2.0 0.5 1- |
| get wl(l)         | REFSXI 4.0 1.5 1-| REFDXI 5.0 1.5 2-|
| get 1             | REFSL 2.0 0.5 1- | REFSL 2.0 0.5 1- |
| get k(l)          | REFSXI 4.0 1.5 1-| REFDXI 5.0 1.5 2-|
| get present_e     | REFSL 2.0 0.5 1- | ++REFDL 3.0 0.5 2 -|
| multiply          | MPYI 23.0 0.5 - | MPYF 95.0 0.5 - |
| subtract          | SUB 2.0 0.5 - | SUBF 40.0 0.5 - |
| store next_w      | ASNSL 2.0 0.5 -1| ASNSL 3.0 0.5 -2|

80
\[ v(l) = \beta v(l) + \beta_1 (\text{present}_e \times \text{present}_e + w(l) \times w(l)) \]

\[ k(l) = k(l) + \alpha (\text{next}_e \times w(l) + \text{present}_e \times \text{next}_w) / v(l) \]
"wl(1) = present_w"

get present_w REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
get 1 REFSL 2.0 0.5 1 - REFSL 2.0 0.5 1 -
store wl(1) ASNSXI 4.0 1.5 - 1 ASNDXI 5.0 1.5 - 2

8. 2.5 2 1 10. 2.5 3 2

"present_w = next_w"

get next_w REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
store present_w ASNSL 2.0 0.5 - 1 ASNDL 3.0 0.5 - 2

4. 1. 1 1 6. 1. 2 2

"present_e = next_e"

get next_e REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
store present_e ASNSL 2.0 0.5 - 1 ASNDL 3.0 0.5 - 2

4. 1. 1 1 6. 1. 2 2

"endo"

end loop ENDO 9.0 1.0 1 1 ENDO 9.0 1.0 1 1

9. 1. 1 1 9. 1. 1 1

iteration total 364 45 31 8 1336 48 49 15
loop total 5824 720 496 128 21376 768 784 240

"dac_output = present_e"

get present_e REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
convert from f.p. - CVTFD 17.0 0.5 -
- CVTDS 3.0 0.5 -
store to DAC LIT32 5.0 2.5 -- LIT32 5.0 2.5 --
ASNSU 5.0 0.5 - 1 ASNSU 5.0 0.5 - 1

12. 3.5 1 1 33. 4.5 2 1

"goto begin"

goto 'begin' LIT8N 2.0 1.0 -- LIT8N 2.0 1.0 --
SKIP 2.0 1.0 -- SKIP 2.0 1.0 --

2.0 2.0 -- 2.0 2.0 --
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>total instr. cycles</td>
<td>5871</td>
<td>735</td>
<td>498</td>
<td>132</td>
<td>21470</td>
<td>785</td>
</tr>
<tr>
<td>stack updates</td>
<td>1440</td>
<td>-</td>
<td>96</td>
<td>96</td>
<td>4800</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td>7311</td>
<td>735</td>
<td>594</td>
<td>228</td>
<td>26268</td>
<td>785</td>
</tr>
</tbody>
</table>
Modified Widrow listing

This listing approximates the output of a non-optimizing compiler. Although the algorithm is modified, the translation was quite direct and few assembly language modifications were made.

loop:

\[
\begin{align*}
f &= \text{adc\_in} \\
g &= 0 \\
do \ k &= 1,16 \\
& \quad g = g + b(k) * f(k) \\
endo \\
e &= f - g \\
c &= v * e \\
do \ k &= 16,1 \\
& \quad b(k+1) = u * b(k) + c * f(k) \\
endo \\
b(1) &= b(17) \\
q &= q - e(ptr) + e \\
dac\_out &= q * q \\
e(ptr) &= e \\
f(ptr) &= f \\
if \ ptr &= 16 \\
& \quad then \ ptr = 1 \\
& \quad else \ ptr = ptr + 1 \\
goto \ loop
\end{align*}
\]
Note: + = indicates a stack update

<table>
<thead>
<tr>
<th>Comments</th>
<th>Fixed-Point</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>opcode</td>
<td>Nc Nf Nr Nw</td>
</tr>
<tr>
<td>&quot;loop: f = adc_in&quot;</td>
<td>LIT32 5.0 2.5 -</td>
<td>LIT32 5.0 2.5 -</td>
</tr>
<tr>
<td>read ADC</td>
<td>REFSU 5.0 0.5 1 -</td>
<td>REFSU 5.0 0.5 1 -</td>
</tr>
<tr>
<td>convert to f.p.</td>
<td>-</td>
<td>CVTSD 2.0 0.5 -</td>
</tr>
<tr>
<td>store f</td>
<td>ASNSL 2.0 0.5 - 1</td>
<td>ASNDL 3.0 0.5 - 2</td>
</tr>
<tr>
<td></td>
<td>12. 3.5 1 1</td>
<td>36. 4.5 1 2</td>
</tr>
<tr>
<td>&quot;g = 0&quot;</td>
<td>LIT4 1.0 0.5 -</td>
<td>LITD0 2.0 0.5 -</td>
</tr>
<tr>
<td>get 0</td>
<td>ASNSL 2.0 0.5 - 1</td>
<td>ASNDL 3.0 0.5 - 2</td>
</tr>
<tr>
<td>store g</td>
<td>3. 1. - 1</td>
<td>5. 1. - 2</td>
</tr>
<tr>
<td>&quot;do k = 1,16&quot;</td>
<td>LIT16 3.0 1.5 -</td>
<td>LIT16 3.0 1.5 -</td>
</tr>
<tr>
<td>loop var. addr.</td>
<td>LIT4 1.0 0.5 -</td>
<td>LIT4 1.0 0.5 -</td>
</tr>
<tr>
<td>initial value</td>
<td>LIT8 2.0 1.0 -</td>
<td>LIT8 2.0 1.0 -</td>
</tr>
<tr>
<td>final value</td>
<td>LIT4 1.0 0.5 -</td>
<td>LIT4 1.0 0.5 -</td>
</tr>
<tr>
<td>increment</td>
<td>DO 10.0 2.0 - 1</td>
<td>DO 10.0 2.0 - 1</td>
</tr>
<tr>
<td></td>
<td>17. 5.5 - 1</td>
<td>17. 5.5 - 1</td>
</tr>
<tr>
<td>&quot;g = g + b(k) * f(k)&quot;</td>
<td>+REFSL 2.0 0.5 1 -</td>
<td>+REFDL 3.0 0.5 2 -</td>
</tr>
<tr>
<td>get g</td>
<td>+REFSL 2.0 0.5 1 -</td>
<td>+REFSL 2.0 0.5 1 -</td>
</tr>
<tr>
<td>get k</td>
<td>REFSLXI 4.0 1.5 1 -</td>
<td>REFSLXI 5.0 1.5 2 -</td>
</tr>
<tr>
<td>get b(k)</td>
<td>+REFSL 2.0 0.5 1 -</td>
<td>+REFSL 2.0 0.5 1 -</td>
</tr>
<tr>
<td>get k</td>
<td>REFSLXI 4.0 1.5 1 -</td>
<td>REFSLXI 5.0 1.5 2 -</td>
</tr>
<tr>
<td>multiply</td>
<td>MPYI 23.0 0.5 -</td>
<td>MPYF 95.0 0.5 -</td>
</tr>
<tr>
<td>add</td>
<td>ADD 2.0 0.5 -</td>
<td>ADDF 38.0 0.5 -</td>
</tr>
<tr>
<td>store in g</td>
<td>ASNSL 2.0 0.5 - 1</td>
<td>ASNDL 3.0 0.5 - 2</td>
</tr>
<tr>
<td></td>
<td>41. 6. 5 1</td>
<td>153. 6. 8 2</td>
</tr>
<tr>
<td>&quot;endo&quot;</td>
<td>ENDO 9.0 1.0 1 1</td>
<td>ENDO 9.0 1.0 1 1</td>
</tr>
</tbody>
</table>

85
"e = f - g"

get f
REFL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
get g
REFL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
subtract
SUB 2.0 0.5 - - SUBF 40.0 0.5 - -
store e
ASNSL 2.0 0.5 - 1 ASNDL 3.0 0.5 - 2

8. 2. 2 1
49. 2. 4 2

"c = v * e"

get v
LIT16 3.0 1.5 - - LIT32 5.0 2.5 - -
get e
REFL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
multiply
MPYI 23.0 0.5 - - MPYF 95.0 0.5 - -
store c
ASNSL 2.0 0.5 - 1 ASNDL 3.0 0.5 - 2

30. 3. 1 1
106. 4. 2 2

"do k = 16,1"

loop var. addr.
LIT16 3.0 1.5 - - LIT16 3.0 1.5 - -
initial value
LIT8 2.0 1.0 - - LIT8 2.0 1.0 - -
final value
LIT4 1.0 0.5 - - LIT4 1.0 0.5 - -
loop increment
DO 10.0 2.0 - 1 DO 10.0 2.0 - 1

17. 5.5 - 1
17. 5.5 - 1

"b(k+1) = u * b(k) + c * f(k)"

genu
+LIT16 3.0 1.5 - - ++LIT32 5.0 2.5 - -
genud
+REFL 2.0 0.5 1 - +REFL 2.0 0.5 1 -
genux
REFSXI 4.0 1.5 1 - +REFDXI 5.0 1.5 2 -
multiply
MPYI 23.0 0.5 - - MPYF 95.0 0.5 - -
genu
REFL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
genu
+REFL 2.0 0.5 1 - +REFL 2.0 0.5 1 -
genu
REFSXI 4.0 1.5 1 - +REFDXI 5.0 1.5 2 -
multiply
MPYI 23.0 0.5 - - MPYF 95.0 0.5 - -
add
ADD 2.0 0.5 - - ADDF 38.0 0.5 - -
genu
REFL 2.0 0.5 1 - REFSL 2.0 0.5 1 -
genu
LIT4 1.0 0.5 - - LIT4 1.0 0.5 - -
add
ADD 2.0 0.5 - - ADD 2.0 0.5 - -
store b(k+1)
ASNSXI 4.0 1.5 - 1 ASNDXI 5.0 1.5 - 2

74. 10.5 6 1
260. 11.5 9 2

86
"endo"

end the loop

iteration total   83  11.5  7  2
loop total       1328  184  112  32

"b(l) = b(17)"

get 17
get b(17)
get 1
store in b(1)

"q = q - e(ptr) + e"

get q
get ptr
get e(ptr)
subtract
get e
add
duplicate q
store in q

"dac_out = q * q"

duplicate q
square q
convert from f.p.
write to DAC

"e(ptr) = e"

get e
get ptr
store in e(ptr)
"f(ptr) = f"

- get f: REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
- get ptr: REFSL 2.0 0.5 1 - REFSL 2.0 0.5 1 -
- store in e(ptr): ASNSXI 4.0 1.5 - 1 ASNDXI 5.0 1.5 - 2

"if ptr = 16"

- get ptr: REFSL 2.0 0.5 1 - REFSL 2.0 0.5 1 -
- get 16: LIT8 2.0 1.0 - - LIT8 2.0 1.0 - -
- equal?: EQ 3.0 0.5 - - EQ 3.0 0.5 - -
- if not go to else: SKIPZI 3.5 1.5 - - SKIPZI 3.5 1.5 - -

"then ptr = 1"

- get l: LIT4 1.0 0.5 - - LIT4 1.0 0.5 - -
- store in ptr: ASNSL 2.0 0.5 - 1 ASNSL 2.0 0.5 - 1
- jump past else: SKIPI 2.0 1.5 - - SKIPI 2.0 1.5 - -

"else ptr = ptr + 1"

- increment ptr: INCSLE 5.0 1.0 1 1 INCSLE 5.0 1.0 1 1

"goto loop"

- repeat: LIT8N 2.0 1.0 - - LIT8N 2.0 1.0 - -
- SKIP 2.0 1.0 - - SKIP 2.0 1.0 - -

Total instr. cycles: 2387.5 339.5 238 77 7404.5 363 328 117
Stack updates: 1440 - 96 96 2880 - 192 192

TOTAL: 3827.5 339.5 334 173 10286.5 363 520 309
Optimized Widrow listing

The following listing is a hand-optimized version of the Widrow algorithm given below. The program generally follows the equations below but uses some assembly language "tricks" to improve efficiency.

loop:  
\[
f = \text{adc}_\text{in}
\]
\[
g = 0
\]
do k = 16,1
\[
g = g + b(k) \times f(k)
\]
endo
e = f - g
c = v \times e
do k = 16,1
\[
b(k+1) = u \times b(k) + c \times f(k)
\]
endo
b(1) = b(17)
q = q - e(ptr) + e
dac_out = q \times q
e(ptr) = e
f(ptr) = f
if ptr = 16
\[
\text{then} \; ptr = 1
\]
else ptr = ptr + 1
goto loop

Note: + = indicates a stack update
"loop:  \( f = \text{adc}_\text{in} \)"

<table>
<thead>
<tr>
<th>comments</th>
<th>fixed-point opcode</th>
<th>fixed-point Nc</th>
<th>fixed-point Nf</th>
<th>fixed-point Nr</th>
<th>fixed-point Nw</th>
<th>floating-point opcode</th>
<th>floating-point Nc</th>
<th>floating-point Nf</th>
<th>floating-point Nr</th>
<th>floating-point Nw</th>
</tr>
</thead>
<tbody>
<tr>
<td>read ADC</td>
<td>LIT32 5.0 2.5 -</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>LIT32 5.0 2.5 -</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>convert to f.p.</td>
<td>REFSU 5.0 0.5 1 -</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>REFSU 5.0 0.5 1 -</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>store f</td>
<td>ASNSL 2.0 0.5 -</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td>ASNSL 2.0 0.5 -</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

|                       |-------------------|----------------|----------------|----------------|-----------------|
|                       | 12. 3.5 1 1       |                |                |                |                 |

"g = 0"

| get 0                | LIT4 1.0 0.5 -    | -              |                |                |                | LITD0 2.0 0.5 -      | -                 | -                | -                | -                |
| store g              | ASNSL 2.0 0.5 -   | -              |                |                |                | ASNL 3.0 0.5 -       | -                 | -                | -                | -                |

|                       |-------------------|----------------|----------------|----------------|-----------------|
|                       | 3. 1. - 1         |                |                |                |                 |

"do k = 16,1"

| initialize count     | LIT8 2.0 1.0 -    | -              |                |                |                | LIT8 2.0 1.0 -      | -                 | -                | -                | -                |
| store count          | ASNSL 2.0 0.5 -   | -              |                |                |                | ASNSL 2.0 0.5 -     | -                 | -                | -                | -                |

|                       |-------------------|----------------|----------------|----------------|-----------------|
|                       | 4. 1.5 - 1        |                |                |                |                 |

"g = g + b(k) * f(k)"

| get g                | REFSL 2.0 0.5 -   | -              |                |                |                | REFSL 2.0 0.5 -     | -                 | -                | -                | -                |
| get k                | REFSL 2.0 0.5 -   | -              |                |                |                | REFSL 2.0 0.5 -     | -                 | -                | -                | -                |
| get b(k)             | REFSC 3.0 1.0 -   | -              |                |                |                | REFSC 3.0 1.0 -     | -                 | -                | -                | -                |
| get k                | REFSL 2.0 0.5 -   | -              |                |                |                | REFSL 2.0 0.5 -     | -                 | -                | -                | -                |
| get f(k)             | REFSC 3.0 1.0 -   | -              |                |                |                | REFSC 3.0 1.0 -     | -                 | -                | -                | -                |
| multiply             | MPYI 23.0 0.5 -   | -              |                |                |                | MPYF 95.0 0.5 -     | -                 | -                | -                | -                |
| get g                | -                 | REFDL 3.0 0.5 - | -              |                |                | REFDL 3.0 0.5 -     | -                 | -                | -                | -                |
| add                  | ADD 2.0 0.5 -     | -              |                |                |                | ADDF 38.0 0.5 -     | -                 | -                | -                | -                |
| store in g           | ASNSL 2.0 0.5 -   | -              |                |                |                | ASNL 3.0 0.5 -      | -                 | -                | -                | -                |

|                       |-------------------|----------------|----------------|----------------|-----------------|
|                       | 39. 5. 5 1       |                |                |                |                 |

"endo"

| decrement count      | DECSLE 5.0 1.0 1 1| -              |                |                |                | DECSLE 5.0 1.0 1 1 | -                 | -                | -                | -                |
| get count            | REFSL 2.0 0.5 1 1| -              |                |                |                | REFSL 2.0 0.5 1 1 | -                 | -                | -                | -                |
| loop if count<>0     | LIT8N 2.0 1.0 -   | -              |                |                |                | LIT8N 2.0 1.0 -    | -                 | -                | -                | -                |
|                      | SKIPNZ 3.5 1.0 -  | -              |                |                |                | SKIPNZ 3.5 1.0 -   | -                 | -                | -                | -                |

|                       |-------------------|----------------|----------------|----------------|-----------------|
|                       | 12. 5 3.5 2 1     |                |                |                |                 |

```
```

90
iteration total 51.5 8.5 7 2 165.5 9.5 10 3
loop total 824 136 112 32 2648 152 160 48

"e = f - g"
get f REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
get g REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
subtract SUB 2.0 0.5 - - SUBF 40.0 0.5 - -
duplicate e DUP 1.0 0.5 - - DUPD 2.0 0.5 - -
store e ASNSL 2.0 0.5 - 1 ASNDL 3.0 0.5 - 2

9. 2.5 2 1 51. 2.5 4 2

"c = v * e"
get v REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
multiply MPYI 23.0 0.5 - - MPYF 95.0 0.5 - -
store c ASNSL 2.0 0.5 - 1 ASNDL 3.0 0.5 - 2

27. 1.5 1 1 101. 1.5 2 2

"do k = 16,1"
initialize count LIT8 2.0 1.0 - - LIT8 2.0 1.0 - -
store count ASNSL 2.0 0.5 - 1 ASNSL 2.0 0.5 - 1

4. 1.5 - 1 4. 1.5 - 1

"b(k+1) = u * b(k) + c * f(k)"
get u REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
get k REFSL 2.0 0.5 1 - REFSL 2.0 0.5 1 -
get b(k) REPSL 3.0 1.0 1 - REFDXI 5.0 1.5 2 -
multiply MPYI 23.0 0.5 - - MPYF 95.0 0.5 - -
store in temp - ASNDL 3.0 0.5 - 2
get c REFSL 2.0 0.5 1 - REFDL 3.0 0.5 2 -
get k REFSL 2.0 0.5 1 - REFSL 2.0 0.5 1 -
get f(k) REPSL 3.0 1.0 1 - REFDXI 5.0 1.5 2 -
multiply MPYI 23.0 0.5 - - MPYF 95.0 0.5 - -
retrieve temp - REFDL 3.0 0.5 2 -
add ADD 2.0 0.5 - - ADDF 38.0 0.5 - -
get k REFSL 2.0 0.5 1 - REFSL 2.0 0.5 1 -
get l LIT4 1.0 0.5 - - LIT4 1.0 0.5 - -
add ADD 2.0 0.5 - - ADD 2.0 0.5 - -
store b(k+1) ASNSC 3.0 1.0 - 1 ASNDXI 5.0 1.5 - 2

70. 8. 7 1 264. 10.5 13 4
"endo"

decrement count  DECSLE  5.0  1.0  1  1  DECSLE  5.0  1.0  1  1
get count       REFSLE  2.0  0.5  1 -  REFSL  2.0  0.5  1 -
loop if count<>0 LIT0N  2.0  1.0 - -  LIT0N  2.0  1.0 - -
                SKIPNZ  3.5  1.0 - -  SKIPNZ  3.5  1.0 - -
                12.5 3.5 2 1  12.5 3.5 2 1
iteration total  82.5 11.5 9 2  276.5 14 15 5
loop total      1320 184 144 32  4424 224 240 60

"b(1) = b(17)"

get b(17)      REFSLE  3.0  1.0  1 -  REFDL  4.0  1.0  2 -
store b(1)     ASNSLE  3.0  1.0 - 1  ASNDLE  4.0  1.0 - 2
                6. 2. 1 1  8. 2. 2 2

"q = q - e(ptr) + e"

get q          REFSL  2.0  0.5  1 -  REFDL  3.0  0.5  2 -
get ptr        REFSL  2.0  0.5  1 -  REFSL  2.0  0.5  1 -
get e(ptr)     REFSC  3.0  1.0  1 -  REFDXI  5.0  1.5  2 -
subtract       SUB  2.0  0.5 - -  SUBF  40.0  0.5 - -
get e          REFSL  2.0  0.5  1 -  REFDL  3.0  0.5  2 -
add            ADD  2.0  0.5 - -  ADDF  38.0  0.5 - -
duplicate q    DUP  1.0  0.5 - -  DUPD  2.0  0.5 - -
store in q     ASNSL  2.0  0.5 - 1  ASNDL  3.0  0.5 - 2
                16. 4.5 4 1  96. 5. 7 2

"dac_out = q * q"

duplicate q    DUP  1.0  0.5 - -  DUPD  2.0  0.5 - -
square q       MPYI  23.0  0.5 - -  MPYF  95.0  0.5 - -
convert from f.p. - - CVTFD  17.0  0.5 - -
write to DAC   LIT32  5.0  2.5 - -  LIT32  5.0  2.5 - -
                5.0  0.5 - 1  ASNSU  5.0  0.5 - 1
                34. 4. - 1  127. 5. - 1

"e(ptr) = e"

get e          REFSL  2.0  0.5  1 -  REFDL  3.0  0.5  2 -
get ptr        REFSL  2.0  0.5  1 -  REFSL  2.0  0.5  1 -
store in e(ptr) ASNSC  3.0  1.0 - 1  ASNDXI  5.0  1.5 - 2
                7. 2. 2 1  10. 2.5 3 2

92
"f(ptr) = f"

get f
REFSL 2.0 0.5 1
get ptr
REFSL 2.0 0.5 1
store in e(ptr)
ASNSC 3.0 1.0 - 1

---

7. 2. 2 1

"if ptr = 16"
"then ptr = 1"
"else ptr = ptr + 1"

increment ptr
INCSLE 5.0 1.0 1 1
get ptr
REFSL 2.0 0.5 1 -
load mask pattern LIT16 3.0 1.5 --
mask ptr
AND 1.0 0.5 --
store ptr
ASNSL 2.0 0.5 - 1

---

13. 4. 2 2

"goto loop"

repeat
LIT8N 2.0 1.0 --
SKIP 2.0 1.0 --

---

4. 2.0 --

program total
22290 352 271 77
7541 411.5 424 149

(no stack updates)
Optimized Lattice listing

The following listing is a hand-optimized version of the Lattice algorithm presented below. The program generally follows the equations below but uses some assembly language "tricks" to improve efficiency.

begin
  present_w = adc_input
  present_e = present_w
loop
  do l = 0,15
    next_e = present_e - k(l) * wl(l)
    next_w = wl(l) - k(l) * present_e
    v(l) = beta * v(l) + betal * (present_e * present_e + present_e + wl(l) * wl(l))
    k(l) = k(l) + alpha * (next_e * wl(l) + present_e * next_w)/v(l)
    wl(l) = present_w
    present_w = next_w
    present_e = next_e
  endo
  dac_out = present_e
  goto begin

Note: + = indicates a stack update
"begin  
  present_w = adc_input
  present_e = present_w"

read ADC  
  LIT32  5.0 2.5  -  
  REFSU  5.0 0.5 1  -
convert to f.p.  
  -
  -
duplicate data  
  DUP  1.0 0.5  -  
  -
store present_w  
  ASNSL 2.0 0.5  - 1
store present_e  
  ASNSL 2.0 0.5  - 1

"loop  
  do l = 0,15"

init loop count  
  LIT8  2.0 1.0  -  
  ASNSL 2.0 0.5  - 1

"next_e = present_e - k(l) * wl(l)"

ger 1  
  -
duplicate 1  
  -
ger wl(1)  
  -
store wl_l  
  -
ger k(l)  
  -
duplicate k(l)  
  -
store k_l  
  -
ger present_e  
  REFSL 2.0 0.5 1  -  
  REFSL 2.0 0.5 1  -
ger 1  
  REFSL 2.0 0.5 1  -
ger k(l)  
  REFSCL 3.0 1.0 1  -
duplicate k_l  DUP  1.0 0.5 - -
store k_l  ASNSL  2.0 0.5 - -
get l  REFSL  2.0 0.5 1 -
get wl(1)  REFSC  3.0 1.0 1 -
duplicate wl_l  DUP  1.0 0.5 - -
store wl_l  ASNSL  2.0 0.5 - -
multiply  MPYI  23.0 0.5 - -
get present_e  REFSL  2.0 0.5 1 -
multiply  MPYI  23.0 0.5 - -
get k_l  REFSL  2.0 0.5 1 -
subtract  SUB  2.0 0.5 - -
store next_e  ASNSL  2.0 0.5 - 1

"next_w = wl(l) - k(l) * present_e"

get wl_l  REFSL  2.0 0.5 1 -
get k_l  REFSL  2.0 0.5 1 -
multiply  MPYI  23.0 0.5 - -
get k_l  REFSL  2.0 0.5 1 -
subtract  SUB  2.0 0.5 - -
store next_w  ASNSL  2.0 0.5 - 1

"v(l) = beta * v(l) + betal * (present_e * present_e + wl(l) * wl(l))"

get wl_l  REFSL  2.0 0.5 1 -
square wl_l  DUP  1.0 0.5 - -
multiply  MPYI  23.0 0.5 - -
store in temp  -
get present_e  REFSL  2.0 0.5 1 -
square present_e  DUP  1.0 0.5 - -
multiply  MPYI  23.0 0.5 - -
retrieve temp  -
sum squares  ADD  2.0 0.5 - -
multiply  MPYI  23.0 0.5 - -
store in temp  -
get v(l)  REFSL  2.0 0.5 1 -
get beta  REFSL  2.0 0.5 1 -
multiply  MPYI  23.0 0.5 - -
retrieve temp  -
sum expressions  ADD  2.0 0.5 - -
get l  REFSL  2.0 0.5 1 -
store in v(l)  ASNSC  3.0 1.0 - 1

116. 9. 7 1 502. 16. 12 6
"k(l) = k(l) + alpha * (next_e * wl(l) + present_e * next_w) / v(l)"

get present_e  REFSL  2.0  0.5  1  REFDL  3.0  0.5  2  
get next_w    REFSL  2.0  0.5  1  REFDL  3.0  0.5  2  
multiply      MPYI   23.0  0.5  -  MPYF   95.0  0.5  -  
store in temp  -       ASNDL  3.0  0.5  -  2  
get next_e    REFSL  2.0  0.5  1  REFDL  3.0  0.5  2  
get wl_l      REFSL  2.0  0.5  1  REFDL  3.0  0.5  2  
multiply      MPYI   23.0  0.5  -  MPYF   95.0  0.5  -  
retrieve temp  -       REFDL  3.0  0.5  2  -  
sum products  ADD    2.0  0.5  -  ADDF   38.0  0.5  -  
get alpha     REFSL  2.0  0.5  1  LIT32  5.0  2.5  -  
multiply      MPYI   23.0  0.5  -  MPYF   95.0  0.5  -  
get 1         REFSL  2.0  0.5  1  REFDL  2.0  0.5  1  -  
get v(l)      REFSC  3.0  1.0  1  REFDXI 5.0  1.5  2  -  
divide        DIVI   27.0  0.5  -  DIVF   98.0  0.5  -  
get k_l       REFSL  2.0  0.5  1  REFDL  3.0  0.5  2  -  
sum expression ADD    2.0  0.5  -  ADDF   38.0  0.5  -  
get 1         REFSL  2.0  0.5  1  REFDL  2.0  0.5  1  -  
store k(l)    ASNSC  3.0  1.0  -  1 ASNDXI 5.0  1.5  -  2

122. 9. 9 1  499. 13. 16 4

"wl(l) = present_w"

get present_w REFSL  2.0  0.5  1  REFDL  3.0  0.5  2  
get 1          REFSL  2.0  0.5  1  REFDL  2.0  0.5  1  
store wl(l)    ASNSC  3.0  1.0  -  1 ASNDXI 5.0  1.5  -  2

7. 2. 2 1 10. 2.5 3 2

"present_w = next_w"

get next_w    REFSL  2.0  0.5  1  REFDL  3.0  0.5  2  
store present_w ASNSL 2.0  0.5  -  1 ASNDL  3.0  0.5  -  2

4. 1. 1 1 6. 1. 2 2

"present_e = next_e"

get next_e    REFSL  2.0  0.5  1  REFDL  3.0  0.5  2  
store present_e ASNSL 2.0  0.5  -  1 ASNDL  3.0  0.5  -  2

4. 1. 1 1 6. 1. 2 2

"endo"
decrement count DECSLE 5.0 1.0 1 1 DECSLE 5.0 1.0 1 1
get count REFSL 2.0 0.5 1 - REFSL 2.0 0.5 1 -
loop if count<>0 LIT8N 2.0 1.0 - - LIT8N 2.0 1.0 - -
  SKIPNZ 3.5 1.0 - - SKIPNZ 3.5 1.0 - -
----------------------- -----------------------
   12.5  3.5  2  1       12.5  3.5  2  1

iteration total 343.5 35.5 30  10 1359.5  49  52  25
loop total 5496  568  480  160 21752  784  832  400

"dac_output = present_e"

get present_e REFSL 2.0  0.5 1 - REFDL 3.0 0.5 2 -
convert from f.p. - CVTFD 17.0 0.5 - -
                      - CVTDS  3.0  0.5 - -
store to DAC LIT32  5.0  2.5 - - LIT32  5.0  2.5 - -
              ASNSU  5.0  0.5 - 1 ASNSU  5.0  0.5 - 1
----------------------- -----------------------
         12.  3.5  1  1          33.  4.5  2  1

"goto begin"

repeat LIT8N  2.0  1.0 - - LIT8N  2.0 1.0 - -
        SKIP  2.0  1.0 - - SKIP  2.0 1.0 - -
----------------------- -----------------------
      4.  2.0 - -            4.  2.0 - -

program total 5499  571.5  482  164 21786  781.5  835  405
(no stack updates)
Appendix B: Ada-subset listings

Notes on the Ada-subset compiler

The Ada-subset compiler used is resident on a VAXll-780 at
the Rockwell Collins facility in Cedar Rapids, IA. The output of
the compiler front-end is in the form of macro instructions for a
stack machine. These macro instructions are then translated into
instructions for a particular machine, in this case the AAMP. In
order for the compiler to produce object code with Local
variables, the code must be inside a procedure within the
package. If the code is placed in the package without a
procedure, the variables will be addressed using the global
addressing mode, resulting in a considerable loss in efficiency.

Loop variables created in a program are assigned after
declared variables, thus causing them to often reside in the
Local Extended memory area. To use the more efficient Local
memory area, declare an integer variable with a different name at
the beginning of the loop and immediately assign the loop
variable's value to this new variable. This new variable is then
referenced during the rest of the loop. This is economical in
long loops which contain many loop variable references such as
the Lattice.

For each of the following three programs there is an integer
version source listing and both integer and floating-point
versions of the object listings. The reason for this is that the
source listings for integer and floating-point versions were the
same except for the type of number_system and the constants.
Standard Widrow Source listing

with TEXT_IO, portpack;
use TEXT_IO, portpack;

-- Standard Widrow Algorithm
-- April 17, 1984
-- Ken Albin, Dept. of Electrical and Computer Engineering
-- Kansas State University, Manhattan, KS 66506
-- This program is based on the Standard Widrow coded in the
-- AAMP preliminary report.

package WIDROWI is

    procedure WIDROW;

end WIDROWI;

package body WIDROWI is

procedure WIDRWO is

    pragma SUPPRESS(INDEX_CHECK);
    pragma SUPPRESS(RANGE_CHECK);

    -- The loop variable k is always an integer.
    -- Other variables will reflect the number system.

    subtype number_system is integer;

    k : integer;           -- loop variable
    f : number_system;    -- current input value
    g : number_system;    -- summation value
    e : number_system;    -- current error value
                          -- (filter output)
    q : number_system;    -- "alarm" output

    -- c is left out to test the compiler's optimization
    -- c represents an expression which is constant within a loop

    type values is array (1..16) of number_system;

    b_array : values;      -- weight array
    f_array : values;      -- sample array
    e_array : values;      -- error array

    u : constant := 1;
    v : constant := 0;
begin
-- initialization sequence goes here
  for k in 1..16 loop
    b_array(k) := 1;
    f_array(k) := 1;
    e_array(k) := 1;
  end loop;
-- end initialization
-- begin main loop
  loop
    f := adc_in;
    g := 0;
    for k in 1..16 loop
      g := g + b_array(k) * f_array(k);
    end loop;
    e := f - g;
    for k in 1..16 loop
      b_array(k) := u * b_array(k) + v * e * f_array(k);
    end loop;
    q := q - e_array(16) + e;
    dac_out := q * q;
    for k in 1..15 loop
      e_array(k+1) := e_array(k);
      f_array(k+1) := f_array(k);
    end loop;
    e_array(1) := e;
    f_array(1) := f;
  end loop;
end loop;
end WIDROW;
begin
  null;
end WIDROWI;
Integer Standard Widrow Object listing

Macro/Instruction Definitions will be read from module
[TDJ.AAMP16]AAMP16.MLB

Program Size For Counter 1 = 102 Words Decimal.

CAPS Macro Assembler listing for module WIDROWI.OBJ

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instruction</th>
<th>Macro</th>
<th>Macro args.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0036</td>
<td>{ procedure header }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT.</td>
<td>1,1;</td>
</tr>
<tr>
<td>35 5C</td>
<td>ASNSLE</td>
<td>ASNL.</td>
<td>1,53;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L#1002;</td>
</tr>
<tr>
<td>35 1E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,53;</td>
</tr>
<tr>
<td>10 18</td>
<td>LIT8</td>
<td>LIT.</td>
<td>1,16;</td>
</tr>
<tr>
<td>EC</td>
<td>GR</td>
<td>GRT.</td>
<td>1;</td>
</tr>
<tr>
<td>1D5B</td>
<td>SKIPNZI</td>
<td>JUMPT.</td>
<td>L#1001;</td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT.</td>
<td>1,1;</td>
</tr>
<tr>
<td>35 1E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,53;</td>
</tr>
<tr>
<td>14 1D4A</td>
<td>ASNLX</td>
<td></td>
<td>1,4;</td>
</tr>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>ASNSX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT.</td>
<td>1,1;</td>
</tr>
<tr>
<td>35 1E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,53;</td>
</tr>
<tr>
<td>14 18</td>
<td>LIT8</td>
<td>ASNLX.</td>
<td>1,20;</td>
</tr>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>ASNSX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT.</td>
<td>1,1;</td>
</tr>
<tr>
<td>351E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,53;</td>
</tr>
<tr>
<td>2418</td>
<td>LIT8</td>
<td>ASNLX.</td>
<td>1,36;</td>
</tr>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>ASNSX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>351E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,53;</td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT.</td>
<td>1,1;</td>
</tr>
<tr>
<td>E4</td>
<td>ADD</td>
<td>ADD.</td>
<td>1;</td>
</tr>
<tr>
<td>355C</td>
<td>ASNSLE</td>
<td>ASNL.</td>
<td>1,53;</td>
</tr>
</tbody>
</table>
LIT8N
JUMP. L#1002; {go to loop check}

L#1001;
L#1003;
L#1004;

0000 1C REFSI REFS. 1,0, portpack; {f := adc_in}
 41 ASNSL.1 ASNL. 1,1;
10 LIT4A.0 LIT. 1,0; {g := 0}
 42 ASNSL.2 ASNL. 1,2;
11 LIT4A.1 LIT. 1,1; {init loop variable k}
 355C ASNSLE ASNL. 1,53;
L#1007;
351E REFSLE REFL. 1,53;
1018 LIT8 LIT. 1,16;
EC GR GRT. 1;
18 5B SKIPNZI JUMPT. L#1006;
02 REFSL.2 REFL. 1,2; {g := g + b_array(k) * f_array(k)}
351E REFSLE REFL. 1,53; {increment k}
14 LIT4A.4 REFLX. 1,4;
53 LOCL D0 REFSX
35 1E REFSLE REFL. 1,53;
14 18 LIT8 REFLX. 1,20;
53 LOCL D0 REFSX
 E6 MPYI MPY. 1;
 E4 ADD ADD. 1;
 42 ASNSL.2 ASNL. 1,2;
L#1006;
L#1008;
01 REFSL.1 REFL. 1,1; {e := f - g}
02 REFSL.2 REFL. 1,2;
E5 SUB SUB. 1;
43 ASNSL.3 ASNL. 1,3;
11 LIT4A.1 LIT. 1,1; {init loop variable k}
 355C ASNSLE ASNL. 1,53;
L#1010;
351E REFSLE REFL. 1,53; {check loop variable k}
1018 LIT8 LIT. 1,16;
EC GR GRT. 1;
20 5B SKIPNZI JUMPT. L#1009;
If $1;\{b_{array}(k):=u*b_{array}(k)+v*e*f_{array}(k)\}$

1,53; \{increment k\}

11 LIT4A.1 LIT. 1,1; \{init loop variable k\}

351E REFSLE REFL. 1,53; \{check loop variable k\}

14 LIT4A.4 REFLX. 1,4;

53 LOCL

D0 REFSX

E6 MPYI MPY. 1;

10 LIT4A.0 LIT. 1,0;

03 REFSL.3 REFL. 1,3;

E6 MPYI MPY. 1;

35 1E REFSLE REFL. 1,53;

14 18 LIT8 REFLX. 1,20;

53 LOCL

D0 REFSX

E6 MPYI MPY. 1;

E4 ADD ADD. 1;

35 1E REFSLE REFL. 1,53;

14 LIT4A.4 ASNLX. 1,4;

53 LOCL

A6 ASNSX

351E REFSLE REFL. 1,53; \{increment k\}

11 LIT4A.1 LIT. 1,1;

E4 ADD ADD. 1;

355C ASNSLE ASNL. 1,53;

2619 LIT8N JUMP. L#1010; \{go to loop check\}

59 SKIP

L#1009;;

L#1011;;

04 REFSL.4 REFL. 1,4; \{q:=q-e_{array}(16)+e\}

341E REFSLE REFL. 1,52;

E5 SUB SUB. 1;

03 REFSL.3 REFL. 1,3;

E4 ADD ADD. 1;

44 ASNSL.4 ASNL. 1,4;

04 REFSL.4 REFL. 1,4; \{dac_out := q * q\}

04 REFSL.4 REFL. 1,4;

E6 MPYI MPY. 1;

0001 54 ASNXI ASNS. 1,1, portpack;

11 LIT4A.1 LIT. 1,1; \{init loop variable k\}

35 5C ASNSLE ASNL. 1,53;

L#1013;;

35 1E REFSLE REFL. 1,53; \{increment k\}

2F LIT4B.F LIT. 1,15;

EC GR GRT. 1;

21 5B SKIPNZI JUMPT. L#1012;

35 1E REFSLE REFL. 1,53; \{e_{array}(k+1) := \}

24 18 LIT8 REFLX. 1,36; \{e_{array}(k)\}

53 LOCL

D0 REFSX

35 1E REFSLE REFL. 1,53; \{note: an optimization! \}

25 18 LIT8 ASNLX. 1,37; \{base+1 is calculated\}

105
53  LOCL
A6  ASNSX

35 1E  REFSLE  REFL.  1,53; {f_array(k+1) :=
14 18  LIT8    REFLX. 1,20;   f_array(k)}
53  LOCL
   D0  REFSX
35 1E  REFSLE  REFL.  1,53;
15 18  LIT8    ASNLX. 1,21; {note: an optimization! base+1 is calculated}
53  LOCL
   A6  ASNSX

35 1E  REFSLE  REFL.  1,53; {increment k}
11  LIT4A.1  LIT.   1,1;
   E4  ADD     ADD.   1;
35  5C  ASNSLE  ASNL.  1,53;

26 19  LIT8N   JUMP.  L#1013; {go to loop check}
59  SKIP
    L#1012;  L#1014;
03  REFSL.3  REFL.  1,3; {e_array(1) := e}
25  5C  ASNSLE  ASNL.  1,37;
01  REFSL.1  REFL.  1,3; {f_array(1) := f}
155C ASNSLE  ASNL.  1,21;

9519  LIT8N  JUMP.  L#1004; {go to beginning}
59  SKIP
    L#1005;  L#1000;

36 18  LIT8    PROCEND. 54,0; {procedure return}
5F   RETURN

PKGDEF. $init.widrowi.0000,12;
0000 {procedure header for package body}
00 0023 CALLI CALLGS. $init.textio.0000,textio;
0000 23 CALLI CALLGS. $init.portpack.0000,portpack;
  L#2000;  L#3000;
10  LIT4A.0  PKGEND.  0;
5F   RETURN

FINI
Widrow Floating-point Object listing

Macro/Instruction Definitions will be read from module
[TDJ.AAMP16]AAMP16.MLB

Program Size For Counter 1 = 120 Words Decimal.

CAPS Macro Assembler listing for module WIDROWF.OBJ

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>Macro</th>
<th>Macro args.</th>
</tr>
</thead>
<tbody>
<tr>
<td>006E</td>
<td>{ procedure header }</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>8125 LIT32</td>
<td>LIT.</td>
<td>2,1.00000000;</td>
</tr>
<tr>
<td>69 F7</td>
<td>ASNDLE</td>
<td>ASNL.</td>
<td>2,105;</td>
</tr>
<tr>
<td>0000</td>
<td>25 LIT32</td>
<td>LIT.</td>
<td>2,0,00000000;</td>
</tr>
<tr>
<td>6BF7</td>
<td>ASNDLE</td>
<td>ASNL.</td>
<td>2,107;</td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT.</td>
<td>1,1; {init loop variable k}</td>
</tr>
<tr>
<td>6D 5C</td>
<td>ASNSLE</td>
<td>ASNL.</td>
<td>1,109;</td>
</tr>
<tr>
<td></td>
<td>L#1002:;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6D 1E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,109; {check loop variable k}</td>
</tr>
<tr>
<td>10 18</td>
<td>LIT8</td>
<td>LIT.</td>
<td>1,16;</td>
</tr>
<tr>
<td>EC</td>
<td>GR</td>
<td>GRT.</td>
<td>1;</td>
</tr>
<tr>
<td>295B</td>
<td>SKIPNZI</td>
<td>JUMPT.</td>
<td>L#1001;</td>
</tr>
<tr>
<td>00</td>
<td>8125 LIT32</td>
<td>LIT.</td>
<td>2,1.00000000;</td>
</tr>
<tr>
<td>6D 1E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,109; {b_array(k) := 1}</td>
</tr>
<tr>
<td>17</td>
<td>LIT4A.7</td>
<td>ASNLX.</td>
<td>2,7;</td>
</tr>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8C</td>
<td>ASNDX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>8125 LIT32</td>
<td>LIT.</td>
<td>2,1.00000000;</td>
</tr>
<tr>
<td>6D 1E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,109; {f_array(k) := 1}</td>
</tr>
<tr>
<td>27 18</td>
<td>LIT8</td>
<td>ASNLX.</td>
<td>1,39;</td>
</tr>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8C</td>
<td>ASNDX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>25 LIT32</td>
<td>LIT.</td>
<td>2,1.00000000;</td>
</tr>
</tbody>
</table>
$\text{increment } k$

$$f := \text{adc_in} \cdot b_{array}(k) \cdot f_{array}(k)$$

$$g := g + f_{array}(k)$$
31 REFDL.1 REFL. 2,1; {e := f - g}
33 REFDL.3 REFL. 2,3;
85 SUBF SUB. 2;
C5 ASNDL.5 ASNL. 2,5;
11 LIT4A.1 LIT. 1,1; {init loop variable k}
6D5C ASNSLE ASNSL. 1,109;
L#1010:;
6D1E REFSLE REFL. 1,109; {check loop variable k}
1018 LIT8 LIT. 1,16;
EC GR
22 5B SKIPNZI JUMPT. L#1009;
69 22 REFDLE REFL. 2,105; {b_array(k):=u*b_array(k) + v*e*f_array(k)}
6D 1E REFSLE REFL. 1,109;
17 LIT4A.7 REFLX. 2,7;
53 LOCL
D7 REFDX
86 MPYF MPY. 5;
6B 22 REFDLE REFL. 2,107;
35 REFDL.5 REFL. 2,5;
86 MPYF MPY. 5;
6D 1E REFSLE REFL. 1,109;
27 18 LIT8 REFLX. 2,39;
53 LOCL
D7 REFDX
86 MPYF MPY. 5;
84 ADDF ADD. 5;
6D 1E REFSLE REFL. 1,109;
17 LIT4A.7 ASNLX. 2,7;
8C ASNDX
6D1E REFSLE REFL. 1,109; {increment k}
11 LIT4A.1 LIT. 1,1;
E4 ADD ADD. 1;
6D5C ASNSLE ASNSL. 1,109;
2819 LIT8N JUMP. L#1010; {go to loop check}
59 SKIP
L#1009:;
L#1011:;
37 REFDL.7 REFL. 2,7; {q:=q-e_array(16)+e}
6722 REFDLE REFL. 2,103;
85 SUBF SUB. 5;
35 REFDL.5 REFL. 2,5;
84 ADDF ADD. 5;
C7 ASNDL.7 ASNL. 2,7;
37 REFDL.7 REFL. 2,7; {dac_out := q * q}
37 REFDL.7 REFL. 2,7;
86 MPYF MPY. 5;
0001 54 ASNXI ASNS. 1,1, portpack;
11 LIT4A.1 LIT. 1,1; {init loop variable k}
6D 5C  ASNSLE  ASNL.  1,109;
    L#1013;
6D 1E  REFSLE  REFL.  1,109;  \{check loop variable k\}
  2F  LIT4B.F  LIT.  1,15;
  EC  GR  GRT.  1;
21  5B  SKIPNZI  JUMPT.  L#1012;
6D 1E  REFSLE  REFL.  1,109;  \{e_array(k+1) :=
  47 18  LIT8  REFLX.  2,71;  \ e_array(k)\}
  53  LOCL
  D7  REFDX
6D 1E  REFSLE  REFL.  1,109;
  49 18  LIT8  ASNLX.  2,73;  \base+1 is calculated\}
  53  LOCL
  8C  ASNDX
6D 1E  REFSLE  REFL.  1,109;  \{f_array(k+1) :=
  27 18  LIT8  REFLX.  1,39;  \ f_array(k)\}
  53  LOCL
  D7  REFDX
6D 1E  REFSLE  REFL.  1,109;
  29 18  LIT8  ASNLX.  2,41;  \{note: an optimization!\}
  53  LOCL
  8C  ASNDX
  6D 1E  REFSLE  REFL.  1,109;  \{increment k\}
  11  LIT4A.1  LIT.  1,1;
  E4  ADD  ADD.  1;
6D 5C  ASNSLE  ASNL.  1,109;
26 19  LIT8N  JUMP.  L#1013;  \{go to loop check\}
59  SKIP
    L#1012;
    L#1017;
  35  REFDL.5  REFL.  2,5;  \{e_array(1) := e\}
  49 F7  ASNDLE  ASNL.  2,73;
  31  REFDL.1  REFL.  2,1;  \{f_array(1) := f\}
  29F7  ASNDLE  ASNL.  2,41;
  9F19  LIT8N  JUMP.  L#1004;  \{go to beginning\}
  59  SKIP
    L#1005;
    L#1000;
  6E 18  LIT8  PROCEND.  110,0;  \{procedure return\}
  5F  RETURN
PKGDEF.  $init.widrowf.0000,12;
0000  \{procedure header for package body\}
00 0023  CALLI  CALLGS.  $init.textio.0000,textio;
0000  23  CALLI  CALLGS.  $init.portpack.0000,portpack;
10  LIT4A.0  PKGEND.  0;
5F  RETURN
FINI
with portpack;
use portpack;

-- Standard Lattice Algorithm
-- April 18, 1984
-- Ken Albin, Dept. of Electrical and Computer Engineering
-- Kansas State University, Manhattan, KS 66506
-- This program is based on the Standard Lattice coded in the
-- AAMP preliminary report.

package LATTICEI is
    procedure LATTICE;
end LATTICEI;

package body LATTICEI is

procedure LATTICE is

    pragma suppress(index_check);
    pragma suppress(range_check);

    stages: constant integer := 16;

    type number_system is new integer;
    type values is array (1..stages) of number_system;

    loop_count: integer;
    present_w: number_system;
    present_e: number_system;
    next_w: number_system;
    next_e: number_system;

    k: values;
    w1: values;
    v: values;

    beta: constant := 1;
    betal: constant := 2;
    alpha: constant := 0;

begin

    loop

        present_w := number_system(adc_in);

        present_e := k(e);
        next_w := w1;
        next_e := v;

        beta := next_w - present_w;
        betal := next_e - present_e;

        alpha := (present_w * beta) / betal;

        loop_count := loop_count + 1;

        present_w := present_w + alpha;
present_e := present_w;

for i in 1..stages loop

  loop_count := i;

  next_e := present_e - 
             k(loop_count) * wl(loop_count);

  next_w := wl(loop_count) - 
             k(loop_count) * present_e;

  v(loop_count) := beta * v(loop_count) + 
                  betal * (present_e * present_e + 
                          wl(loop_count) * wl(loop_count));

  k(loop_count) := k(loop_count) + alpha * 
                  (next_e * wl(loop_count) + 
                   present_e * next_w) / v(loop_count);

  present_w := next_w;

  present_e := next_e;

end loop;

dac_out := integer(present_e);

end loop;

end lattice;

begin

  null;

end latticei;
### Integer Lattice Object listing

Macro/Instruction Definitions will be read from module [TDJ.AAMP16]AAMP16.MLB

Program Size For Counter 1 = 71 Words Decimal.

CAPS Macro Assembler listing for module LATTICEI.OBJ

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instruction</th>
<th>Macro args.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0036</td>
<td>{procedure header}</td>
<td></td>
</tr>
<tr>
<td>0001C</td>
<td>REFSI</td>
<td>1,0, portpack; {present_w := 1,1; number_system(adc_in)}</td>
</tr>
<tr>
<td>41</td>
<td>ASNSL.1</td>
<td>1,1; {present_e := present_w}</td>
</tr>
<tr>
<td>01</td>
<td>REFL.</td>
<td>1,2;</td>
</tr>
<tr>
<td>42</td>
<td>ASNSL.2</td>
<td>1,1; {init loop variable i}</td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>1,53;</td>
</tr>
<tr>
<td>35 5C</td>
<td>ASNSLE</td>
<td></td>
</tr>
<tr>
<td>35 1E</td>
<td>REFL.</td>
<td>1,53; {loop count check}</td>
</tr>
<tr>
<td>10 18</td>
<td>LIT8</td>
<td>1,16;</td>
</tr>
<tr>
<td>EC</td>
<td>GR</td>
<td>1;</td>
</tr>
<tr>
<td>6A5B</td>
<td>SKIPNZI</td>
<td>L#1003;</td>
</tr>
<tr>
<td>351E</td>
<td>REFL.</td>
<td>1,53; {loop_count := i}</td>
</tr>
<tr>
<td>40</td>
<td>ASNL.0</td>
<td>1,0;</td>
</tr>
<tr>
<td>02</td>
<td>REFL.</td>
<td>1,2; {next_e := present_e}</td>
</tr>
<tr>
<td>00</td>
<td>REFL.</td>
<td>1,0; {- k(loop_count) * }</td>
</tr>
<tr>
<td>14 44</td>
<td>REFLX.</td>
<td>1,4; {wl(loop_count)}</td>
</tr>
<tr>
<td>53 1B</td>
<td>LOCL</td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>REFSX</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>REFL.</td>
<td>1,0;</td>
</tr>
<tr>
<td>14 1B</td>
<td>LIT8</td>
<td>1,20;</td>
</tr>
<tr>
<td>53 1B</td>
<td>LOCL</td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>REFSX</td>
<td></td>
</tr>
<tr>
<td>E6</td>
<td>MPYI</td>
<td>1;</td>
</tr>
<tr>
<td>E5</td>
<td>SUB</td>
<td>1;</td>
</tr>
<tr>
<td>44</td>
<td>ASNSL.4</td>
<td>1,4;</td>
</tr>
<tr>
<td>00</td>
<td>REFL.</td>
<td>1,0; {next_w := wl(loop_count)}</td>
</tr>
</tbody>
</table>

113
<table>
<thead>
<tr>
<th>14 18</th>
<th>LIT8</th>
<th>REFLX.</th>
<th>1,20;</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>REFSX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>REFSL.0</td>
<td>REFL.</td>
<td>1,0;</td>
</tr>
<tr>
<td>14</td>
<td>LIT4A.4</td>
<td>REFLX.</td>
<td>1,4;</td>
</tr>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>REFSX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td>REFSL.2</td>
<td>REFL.</td>
<td>1,2;</td>
</tr>
<tr>
<td>E6</td>
<td>MPYI</td>
<td>MPY.</td>
<td>1;</td>
</tr>
<tr>
<td>E5</td>
<td>SUB</td>
<td>SUB.</td>
<td>1;</td>
</tr>
<tr>
<td>43</td>
<td>ASNSL.3</td>
<td>ASNL.</td>
<td>1,3;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>11</th>
<th>LIT4A.4</th>
<th>LIT.</th>
<th>1,1;</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>REFSL.0</td>
<td>REFL.</td>
<td>1,0;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>24 18</th>
<th>LIT8</th>
<th>REFLX.</th>
<th>1,36;</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>REFSX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E6</td>
<td>MPYI</td>
<td>MPY.</td>
<td>1;</td>
</tr>
<tr>
<td>12</td>
<td>LIT4A.2</td>
<td>LIT.</td>
<td>1,2;</td>
</tr>
<tr>
<td>02</td>
<td>REFSL.2</td>
<td>REFL.</td>
<td>1,2;</td>
</tr>
<tr>
<td>02</td>
<td>REFSL.2</td>
<td>REFL.</td>
<td>1,2;</td>
</tr>
<tr>
<td>E6</td>
<td>MPYI</td>
<td>MPY.</td>
<td>1;</td>
</tr>
<tr>
<td>00</td>
<td>REFSL.0</td>
<td>REFL.</td>
<td>1,0;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>14 18</th>
<th>LIT8</th>
<th>REFLX.</th>
<th>1,20;</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>REFSX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E6</td>
<td>MPYI</td>
<td>MPY.</td>
<td>1;</td>
</tr>
<tr>
<td>E4</td>
<td>ADD</td>
<td>ADD.</td>
<td>1;</td>
</tr>
<tr>
<td>E6</td>
<td>MPYI</td>
<td>MPY.</td>
<td>1;</td>
</tr>
<tr>
<td>E4</td>
<td>ADD</td>
<td>ADD.</td>
<td>1;</td>
</tr>
<tr>
<td>00</td>
<td>REFSL.0</td>
<td>REFL.</td>
<td>1,0;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>24 18</th>
<th>LIT8</th>
<th>ASNLX.</th>
<th>1,36;</th>
</tr>
</thead>
<tbody>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A6</td>
<td>ASNSX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>00</th>
<th>REFSL.0</th>
<th>REFL.</th>
<th>1,0;</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>LIT4A.4</td>
<td>REFLX.</td>
<td>1,4;</td>
</tr>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>REFSX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LIT4A.0</td>
<td>LIT.</td>
<td>1,0;</td>
</tr>
<tr>
<td>04</td>
<td>REFSL.4</td>
<td>REFL.</td>
<td>1,4;</td>
</tr>
<tr>
<td>00</td>
<td>REFSL.0</td>
<td>REFL.</td>
<td>1,0;</td>
</tr>
<tr>
<td>1418</td>
<td>LIT8</td>
<td>REFLX.</td>
<td>1,20;</td>
</tr>
<tr>
<td>53</td>
<td>LOCL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D0</td>
<td>REFSX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E6</td>
<td>MPYI</td>
<td>MPY.</td>
<td>1;</td>
</tr>
<tr>
<td>02</td>
<td>REFSL.2</td>
<td>REFL.</td>
<td>1,2;</td>
</tr>
<tr>
<td>03</td>
<td>REFSL.3</td>
<td>REFL.</td>
<td>1,2;</td>
</tr>
<tr>
<td>E6</td>
<td>MPYI</td>
<td>MPY.</td>
<td>1;</td>
</tr>
<tr>
<td>E4</td>
<td>ADD</td>
<td>ADD.</td>
<td>1;</td>
</tr>
<tr>
<td>E6</td>
<td>MPYI</td>
<td>MPY.</td>
<td>1;</td>
</tr>
</tbody>
</table>
00 REFSL.0 REFL. 1,0;
24 18 LIT8 REFLX. 1,36;
53 LOCL
D0 REFSX
E7 DIVI DIV. 1;
E4 ADD ADD. 1;
00 REFSL.0 REFL. 1,0;
14 LIT4A.4 ASNLX. 1,4;
53 LOCL
A6 ASNSX
01 REFSL.1 REFL. 1,1; {wl(loop_count) := present_w}
00 REFSL.0 REFL. 1,0; present_w
14 18 LIT8 ASNLX. 1,20;
53 LOCL
A6 ASNSX
03 REFSL.3 REFL. 1,3; {present_w := next_w}
41 ASNSL.1 ASNL. 1,1;
04 REFSL.4 REFL. 1,4; {present_e := next_e}
42 ASNSL.2 ASNL. 1,2;
35 1E REFSLE REFL. 1,53; {increment i}
11 LIT4A.1 LIT. 1,1;
E4 ADD ADD. 1;
35 5C ASNSLE ASNL. 1,53;
70 19 LIT8N JUMP. L#1004; {go to loop check}
59 SKIP
L#1003;
L#1005; {dac_out := integer(present_e) }
02 REFSL.2 REFL. 1,2; present_e
0001 54 ASNSI ASNS. 1,1, portpack;
8019 LIT8N JUMP. L#1001; {go to beginning}
59 SKIP
L#1002;
L#1000;
36 18 LIT8 PROCEND. 54,0; {procedure return}
5F RETURN

PKGDEF. $init.latticei.0000,12;
0000 {procedure header}
00 0023 CALLI CALLGS. $init.portpack.0000,portpack;
10 LIT4A.0 PKGEN. 0;
20 NOP FINI
Floating-point Lattice Object listing

Macro/Instruction Definitions will be read from module [TDJ.AAMP16]AAMP16.MLB

Program Size For Counter 1 = 85 Words Decimal.

CAPS Macro Assembler listing for module LATTICEF.OBJ

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instruction</th>
<th>Macro</th>
<th>Macro args.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0070</td>
<td>{procedure header}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 8125</td>
<td>LIT32</td>
<td>LIT.</td>
<td>2,1.00000000;</td>
</tr>
<tr>
<td>00</td>
<td>69 F7</td>
<td>ASNDLE</td>
<td>2,105;</td>
</tr>
<tr>
<td>0082 25</td>
<td>LIT32</td>
<td>LIT.</td>
<td>2,2.00000000;</td>
</tr>
<tr>
<td>0000</td>
<td>6BF7 ASNDLE</td>
<td>ASNL.</td>
<td>2,107;</td>
</tr>
<tr>
<td>0000 0025</td>
<td>LIT32</td>
<td>LIT.</td>
<td>2,0.00000000;</td>
</tr>
<tr>
<td>00</td>
<td>6D F7</td>
<td>ASNDLE</td>
<td>2,109;</td>
</tr>
<tr>
<td></td>
<td>L#1001:;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000 1C</td>
<td>REFSI</td>
<td>REFS.</td>
<td>1,0, portpack;</td>
</tr>
<tr>
<td>65</td>
<td>CVTSD</td>
<td>CONVERT.</td>
<td>1,5,0,0; {present_w := number_system(adc_in)}</td>
</tr>
<tr>
<td>D9</td>
<td>CVTDF</td>
<td>ASNL.</td>
<td>2,1;</td>
</tr>
<tr>
<td>C1</td>
<td>ASNDL.1</td>
<td>ASNL.</td>
<td>2,3;</td>
</tr>
<tr>
<td>31</td>
<td>REFDL.1</td>
<td>REFL.</td>
<td>2,1; {present_e := present_w}</td>
</tr>
<tr>
<td>C3</td>
<td>ASNDL.3</td>
<td>ASNL.</td>
<td>2,3;</td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT.</td>
<td>1,1; {init loop variable i}</td>
</tr>
<tr>
<td>6F5C</td>
<td>ASNSLE</td>
<td>ASNL.</td>
<td>1,111;</td>
</tr>
<tr>
<td></td>
<td>L#1004:;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6F1E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,111; {loop_count check}</td>
</tr>
<tr>
<td>1018</td>
<td>LIT8</td>
<td>LIT.</td>
<td>1,16;</td>
</tr>
<tr>
<td>EC</td>
<td>GR</td>
<td>GRT.</td>
<td>1;</td>
</tr>
<tr>
<td>6D 5B</td>
<td>SKIPNIZI</td>
<td>JUMPT.</td>
<td>L#1003;</td>
</tr>
<tr>
<td>6F 1E</td>
<td>REFSLE</td>
<td>REFL.</td>
<td>1,111; {loop_count := i}</td>
</tr>
<tr>
<td>40</td>
<td>ASNSL.0</td>
<td>ASNL.</td>
<td>1,0;</td>
</tr>
<tr>
<td>33</td>
<td>REFDL.3</td>
<td>REFL.</td>
<td>2,3; {next_e := present_e}</td>
</tr>
<tr>
<td>00</td>
<td>REFSL.0</td>
<td>REFL.</td>
<td>1,0; - k(loop_count)</td>
</tr>
</tbody>
</table>
17 LIT4A.7 REFLX. 2,7;  * wl(loop_count) }
53 LOCL
D7 REFDBX
00 REFS6L.0 REFL. 1,0;
2718 LIT8 REFLX. 2,39;
53 LOCL
D7 REFDBX
86 MPYF MPY. 5;
85 SUBF SUB. 5;
C7 ASNDL.7 ASNLS 2,7;
00 REFS6L.0 REFL. 1,0; {next_w := wl(loop_count) }
2718 LIT8 REFLX. 2,39;  * k(loop_count)
53 LOCL
D7 REFDBX
00 REFS6L.0 REFL. 1,0;
17 LIT4A.7 REFLX. 2,7;
53 LOCL
D7 REFDBX
33 REFLD.3 REFL. 2,3;
86 MPYF MPY. 5;
85 SUBF SUB. 5;
C5 ASNDL.5 ASNLS 2,5;
6922 REFDBLE REFL. 2,105; {v(loop_count) := betaальнv(loop_count) + betal }
00 REFS6L.0 REFL. 1,0;  *v(loop_count) + betal }
47 18 LIT8 REFLX. 2,71; (present_e*present_e
53 LOCL
D7 REFDBX
86 MPYF MPY. 5;
6B22 REFDBLE REFL. 2,107;
33 REFLD.3 REFL. 2,3;
33 REFLD.3 REFL. 2,3;
86 MPYF MPY. 5;
00 REFS6L.0 REFL. 1,0;
2718 LIT8 REFLX. 2,39;
53 LOCL
D7 REFDBX
00 REFS6L.0 REFL. 1,0;
27 18 LIT8 REFLX. 2,39;
53 LOCL
D7 REFDBX
86 MPYF MPY. 5;
84 ADDF ADD. 5;
86 MPYF MPY. 5;
84 ADDF ADD. 5;
00 REFS6L.0 REFL. 1,0;
4718 LIT8 ASNLS 2,71;
53 LOCL
8C ASNDX
00 REFS6L.0 REFL. 1,0; {k(loop_count) :=
17 LIT4A.7 REFLX. 2,7;  * (next_e*wl(loop_count) + present_e*next_w) /
86 MPYF MPY. 5; \( v(loo\_c\_\_\text{count}) \}
33 REFDL.3 REPL. 2,3;
35 REFDL.5 REPL. 2,5;
86 MPYF MPY. 5;
84 ADDF ADD. 5;
86 MPYF MPY. 5;
00 REFSFL.0 REPL. 1,0;
47 18 LIT8 REPLX. 2,71;
53 LOCL D7 REF DX
87 DIVF DIV. 5;
84 ADDF ADD. 5;
00 REFSFL.0 REPL. 1,0;
17 LIT4A.7 ASNLX. 2,7;
53 LOCL 8C ASNDX
31 REFDL.1 REPL. 2,1; \{wl(loo\_c\_\_\text{count}) := \}
00 REFSFL.0 REPL. 1,0; \ \text{ present\_w} \}
27 18 LIT8 ASNLX. 2,39;
53 LOCL 8C ASNDX
35 REFDL.5 REPL. 2,5; \{ \text{present\_w := next\_w} \}
C1 ASNDL.1 ASNL. 2,1;
37 REFDL.7 REPL. 2,7; \{ \text{present\_e := next\_e} \}
C3 ASNDL.3 ASNL. 2,3;
6F 1E REFSLE REPL. 1,111; \{ \text{increment i} \}
11 LIT4A.1 LIT. 1,1;
E4 ADD ADD. 1;
6F 5C ASNSLE ASNL. 1,111;
73 19 LIT8N JUMP. L#1004; \{ \text{go to loop check} \}
59 SKIP
  L#1003:;
  L#1005:;
33 REFDL.3 REPL. 2,3; \{ \text{dac\_out :=} \}
DB CVTFD CONVERT. 5,1,0,0; \text{integer(present\_e)} \}
DA CVTDS
0001 54 ASNSI ASNS. 1,1,portpack;
8719 LIT8N JUMP. L#1001; \{ \text{go to beginning} \}
59 SKIP
  L#1002:;
  L#1000:;
70 18 LIT8 PROCEND. 112,0; \{ \text{procedure return} \}
  \{ \text{never used} \}
  PKGDEF. $-init.latticelf.0000,12;
0000 \{ \text{procedure header for package} \}
00 0023 CALLI CALLGS. $-init.portpack.0000,portpack;
  L#2000:;
10 LIT4A.0 PKGEND. 0;
5F  RETURN
20   NOP

FINI
Integer ADATESTS Source listing

-- Loop Structure Test
-- April 18, 1984
-- Ken Albin, Dept. of Electrical and Computer Engineering
-- Kansas State University, Manhattan, KS 66506
-- This program attempts to test the efficiency of various
-- compiled structures available in Ada.

package adatests is
  procedure dummy;
  procedure stuff;
end adatests;

package body adatests is
  procedure dummy is
    Nothing goes on here - this is just to look at calling code.
    begin
      null;
    end;

  procedure stuff is
    This section test various control structures found in Ada.
    type number_system is new integer;
    done: boolean := false;
    A,B,C,D,E,F,G: number_system;
    function add_seven(junk_in: number_system) return number_system is
      begin
        return junk_in + 7;
      end add_seven;
    begin
while not done loop
    done := true;
end loop;
for count in 1..5 loop
    null;
end loop;
loop
    null;
    exit;
end loop;
-- The following is a test to see the reordering (if any) performed.
A := B + C * (D + E * (F + G));
-- The following is a test to see if an optimization is made to avoid storing and then immediately retrieving a variable.
-- First argument matches last assigned (A).
A := B + C;
D := A + G;
-- Second argument matches last assigned (B).
C := D + E;
B := F + C;
-- Common subexpression elimination test.
A := (B + C) * D - (B + C);
-- Duplicate argument instead of fetch again.
A := D * D;
A := (D + 5) * (D + 5);
-- Removal of loop invariant expressions.
for count in 1..5 loop
A := 1;
E := 1 + 3;
B := C + D;
end loop;

-- Test to see if the increment instruction is used.
A := A + 1;

-- Sample procedure call.
dummy.

-- Sample function call.
B := add_seven(A);
end stuff;

begin
  null:
end adatests;
Integer ADATESTS Object listing

Macro/Instruction Definitions will be read from module [TDJ.AAMP16]AAMP16.MLB

Program Size For Counter 1 = 68 Words Decimal.

CAPS Macro Assembler listing for module ADATESTS.OBJ

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instruction</th>
<th>Macro</th>
<th>Macro args.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>{procedure header for dummy}</td>
<td>L#1000:</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LIT4A.0</td>
<td>PROCEND.</td>
<td>0,0; {null procedure body}</td>
</tr>
<tr>
<td>5F</td>
<td>RETURN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>{procedure header for add_seven}</td>
<td>L#2000:</td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>REFSL.0</td>
<td>REFL.</td>
<td>1,0; {return junk_in + 7}</td>
</tr>
<tr>
<td>17</td>
<td>LIT4A.7</td>
<td>LIIT.</td>
<td>1,7;</td>
</tr>
<tr>
<td>E4</td>
<td>ADD</td>
<td>ADD.</td>
<td>1;</td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>RETURN.</td>
<td>1;</td>
</tr>
<tr>
<td>5F</td>
<td>RETURN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>PROCEND.</td>
<td>0,1;</td>
</tr>
<tr>
<td>5F</td>
<td>RETURN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>NOP</td>
<td>PROCDEF.</td>
<td>stuff.adatests.0002,9,12;</td>
</tr>
<tr>
<td>0009</td>
<td>{procedure header for stuff}</td>
<td>L#3001:</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LIT4A.0</td>
<td>LIT.</td>
<td>1,0;</td>
</tr>
<tr>
<td>40</td>
<td>ASNSL.0</td>
<td>ASNL.</td>
<td>1,0;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>REFSL.0</td>
<td>REFL.</td>
<td>1,0; {initialize done:=false}</td>
</tr>
<tr>
<td>05</td>
<td>SKIPNZI</td>
<td>JUMPT.</td>
<td>L#3002;</td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT.</td>
<td>1,1;</td>
</tr>
<tr>
<td>40</td>
<td>ASNSL.0</td>
<td>ASNL.</td>
<td>1,0;</td>
</tr>
<tr>
<td>07</td>
<td>LIT8N</td>
<td>JUMP.</td>
<td>L#3001; {end loop}</td>
</tr>
<tr>
<td>59</td>
<td>SKIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
15 LIT4A.5 LIT. 1,5;
EC GR GRT. 1;
07 5B SKIPNZI JUMPT. L#3004;
  {null loop body}
08 REFSL.8 REFL. 1,8;  {increment count}
11 LIT4A.1 LIT. 1,1;
E4 ADD ADD. 1;
48 ASNSL.8 ASNL. 1,8;
0B 19 LIT8N JUMP. L#3005;  {go to loop check}
59 SKIP
  L#3004:;
  L#3006:;
  L#3007:;
031D SKIPI JUMP. L#3008;  {begin loop}
  L#3009:;
0419 LIT8N JUMP. L#3007;  {end loop}
59 SKIP
  L#3008:;
  L#3008:;
02 REFSL.2 REFL. 1,2;  { A := B + C * (D + E * (F + G)) } 
03 REFSL.3 REFL. 1,3;
04 REFSL.4 REFL. 1,4;
05 REFSL.5 REFL. 1,5;
06 REFSL.6 REFL. 1,6;
07 REFSL.7 REFL. 1,7;
E4 ADD ADD. 1;
E6 MPYI MPY. 1;
E4 ADD ADD. 1;
E6 MPYI MPY. 1;
E4 ADD ADD. 1;
41 ASNSL.1 ASNL. 1,1;
02 REFSL.2 REFL. 1,2;  { A := B + C } 
03 REFSL.3 REFL. 1,3;
E4 ADD ADD. 1;
41 ASNSL.1 ASNL. 1,1;
01 REFSL.1 REFL. 1,1;  { D := A + G } 
07 REFSL.7 REFL. 1,7;
E4 ADD ADD. 1;
44 ASNSL.4 ASNL. 1,4;
04 REFSL.4 REFL. 1,4;  { C := D + E } 
05 REFSL.5 REFL. 1,5;
E4 ADD ADD. 1;
43 ASNSL.3 ASNL. 1,3;
06 REFSL.6 REFL. 1,6;  { B := F + C } 
03 REFSL.3 REFL. 1,3;
E4 ADD ADD. 1;
42 ASNSL.2 ASNL. 1,2;
02 REFSL.2 REFL. 1,2;  { A := (B + C) * D - (B + C) } 
03 REFSL.3 REFL. 1,3;
E4 ADD ADD. 1;
04 REFSL.4   REFL.  1,4; { A := D * D }
E6 MPYI   MPY.  1;
02 REFSL.2   REFL.  1,2;
03 REFSL.3   REFL.  1,3;
E4 ADD   ADD.  1;
E5 SUB   SUB.  1;
41 ASNSL.1   ASNL.  1,1;

04 REFSL.4   REFL.  1,4; {A:=(D+5)*(D+5)}
15 LIT4A.5   LIT.  1,5;
E4 ADD   ADD.  1;
04 REFSL.4   REFL.  1,4;
15 LIT4A.5   LIT.  1,5;
E4 ADD   ADD.  1;
E6 MPYI   MPY.  1;
41 ASNSL.1   ASNL.  1,1;

11 LIT4A.1   LIT.  1,1; {init count := 1}
48 ASNSL.8   ASNL.  1,8;

08 REFSL.8   REFL.  1,8; {check count}
15 LIT4A.5   LIT.  1,5;
EC GR   GRT.  1;
0F5B SKIPNZI   JUMPT. L#3010;

11 LIT4A.1   LIT.  1,1; {A := 1}
41 ASNSL.1   ASNL.  1,1;

14 LIT4A.4   LIT.  1,4; {E := 1 + 3}
45 ASNSL.5   ASNL.  1,5;{note: an optimization!}

03 REFSL.3   REFL.  1,3; {B := C + D}
04 REFSL.4   REFL.  1,4;
E4 ADD   ADD.  1;
42 ASNSL.2   ASNL.  1,2;

08 REFSL.8   REFL.  1,8; {increment count}
11 LIT4A.1   LIT.  1,1;
E4 ADD   ADD.  1;
48 ASNSL.8   ASNL.  1,8;

1319 LIT8N   JUMP. L#3011; {go to check count}
59 SKIP

L#3010;
L#3012;

01 REFSL.1   REFL.  1,1; {A := A + 1}
11 LIT4A.1   LIT.  1,1;
E4 ADD   ADD.  1;
41 ASNSL.1   ASNL.  1,1;
0000 23 CALLI CALLG. dummy.adatests.0001;
01 REFSL.1 REFL. 1,1; { B := add_seven(A) }
0004 23 CALLI CALLL. add_seven.adatests.0000;
42 ASNSL.2 ASNSL. 1,2;
L#3000:;
29 LIT4B.9 PROCEND. 9,0;
5F RETURN
20 NOP PKGDEF. $init.adatests.0000,12;
0000 {procedure header}
L#4000:;
10 LIT4A.0 PKGEND. 0;
5F RETURN FINI
Floating-point ADATESTS Object listing

Macro/Instruction Definitions will be read from module [TDJ.AAMP16]AAMP16.MLB

Program Size for Counter 1 = 83 Words Decimal.

CAPS Macro Assembler listing for module ADATESTS.F.OBJ

IDENT. 'adatestsf',' AAMP/ACAPS Code
Generator Version 1.6';
XREF. standard;
PACKAGE. adatestsf;
XDEF. $init.adatestsf.0000;
XDEF. dummy.adatestsf.0001;
XDEF. stuff.adatestsf.0002;
PROCDEF. dummy.adatestsf.0001,0,12;

<table>
<thead>
<tr>
<th>Opcodes</th>
<th>Instruction</th>
<th>Macro</th>
<th>Macro args.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>{procedure header for dummy}</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>L#1000:;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LIT4A.4</td>
<td>PROCEND. 0,0; {null body of dummy}</td>
<td></td>
</tr>
<tr>
<td>5F</td>
<td>RETURN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0000</td>
<td>{procedure header for function add_seven}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>REFDL.O</td>
<td>REFL. 2,0; {arg passed on stack}</td>
<td></td>
</tr>
<tr>
<td>0083</td>
<td>LIT32</td>
<td>LIT. 2,7.00000000;</td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>ADDF</td>
<td>ADD. 5; {return junk_in + 7}</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>LIT4A.2</td>
<td>RETURN. 2;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>LIT4A.2</td>
<td>PROCEND. 0,2;</td>
<td></td>
</tr>
<tr>
<td>5F</td>
<td>RETURN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>NOP</td>
<td>PROCDEF. stuff.adatestsf.0002,16,12;</td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>{procedure header for stuff}</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>LIT4A.0</td>
<td>LIT. 1,0; {init done:=false}</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>ASNSL.O</td>
<td>ASNL. 1,0;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>REFL.O</td>
<td>REFL. 1,0; {while test}</td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>5B</td>
<td>JUMPT. L#3002;</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT. 1,1; {set done:=true}</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>ASNSL.O</td>
<td>ASNL. 1,0;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>07</td>
<td>19</td>
<td>JUMP. L#3001; {go to while test}</td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>SKIP</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>LIT4A.1</td>
<td>LIT. 1,1; {init loop variable}</td>
<td></td>
</tr>
<tr>
<td>4F</td>
<td>ASNSL.F</td>
<td>ASNL. 1,15;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0F</td>
<td>REFL.F</td>
<td>REFL. 1,15; {test loop variable}</td>
<td></td>
</tr>
</tbody>
</table>

127
07 5B  SKIPNZI  JUMPT.  L#3004;  
   {null body of loop}
0F  REFSL.F  REFL.  1,15;  
11  LIT4A.1  LIT.  1,1;
E4   ADD   ADD.  1;
4F  ASNSL.F  ASNL.  1,15;

0B 19  LIT8N  JUMP.  L#3005;  
   {go to loop test}
59   SKIP

  L#3004;:
  L#3006;:
  L#3007;:
  {beginning of loop}
031D  SKIPI  JUMP.  L#3008;  
   {exit loop}
0419  LIT8N  JUMP.  L#3007;  
   {go to loop beginning}
59   SKIP

  L#3008;:
33   REFDL.3  REFL.  2,3;  
35   REFDL.5  REFL.  2,5;
37   REFDL.7  REFL.  2,7;
39   REFDL.9  REFL.  2,9;
3B   REFDL.B  REFL.  2,11;
3D   REFDL.D  REFL.  2,13;
84   ADDF  ADD.  5;
86   MPYF  MPY.  5;
84   ADDF  ADD.  5;
86   MPYF  MPY.  5;
84   ADDF  ADD.  5;
  C1  ASNDL.1  ASNL.  2,1;
33   REFDL.3  REFL.  2,3;  
35   REFDL.5  REFL.  2,5;
84   ADDF  ADD.  5;
  C1  ASNDL.1  ASNL.  2,1;
31   REFDL.1  REFL.  2,1;  
3D   REFDL.D  REFL.  2,13;
84   ADDF  ADD.  5;
  C7  ASNDL.7  ASNL.  2,7;
37   REFDL.7  REFL.  2,7;  
39   REFDL.9  REFL.  2,9;
84   ADDF  ADD.  5;
  C5  ASNDL.5  ASNL.  2,5;
3B   REFDL.B  REFL.  2,B;  
35   REFDL.5  REFL.  2,5;
84   ADDF  ADD.  5;
  C3  ASNDL.3  ASNL.  2,3;
33   REFDL.3  REFL.  2,3;  
35   REFDL.5  REFL.  2,5;
84   ADDF  ADD.  5;

128
37 REF DL.7 REF. 2,7;  
86 M P Y F MPY. 5;  
33 REF DL.3 REF. 2,3;  
35 REF DL.5 REF. 2,5;  
84 ADD F ADD. 5;  
85 SUB F SUB. 5;  
C1 AS ND L.1 ASNL. 2,1;  

37 REF DL.7 REF. 2,7; \{A:=D*D\}  
37 REF DL.7 REF. 2,7;  
86 M P Y F MPY. 5;  
C1 AS ND L.1 ASNL. 2,1;  

0000 8325 LIT32 LIT. 2,5.000000000;  
20 84 ADD F ADD. 5;  
0083 25 LIT32 LIT32 2,5.000000000;  
2000 84 ADD F ADD. 5;  
86 M P Y F MPY. 5;  
C1 AS ND L.1 ASNL. 2,1;  
11 LIT4A.1 LIT. 1,1; \{init count:=1\}  
4F AS NSL.F ASNL. 1,15;  
L#3011:;  
0F RE FS L.F REF. 1,15; \{loop test\}  
15 LIT4A.5 LIT. 1,5;  
EC GR GRT. 1;  
1D5B SK IP N ZI JUMPT. L#3010;  

0000 8125 LIT32 LIT. 2,1.000000000; \{A:=1.0\}  
00 C1 AS ND L.1 ASNL. 2,1;  
0000 8125 LIT32 LIT. 2,1.000000000; \{E:=l+3\}  
0000 8125 LIT32 LIT. 2,3.000000000;  
4000 84 ADD F ADD. 5;  
C9 AS ND L.9 ASNL. 2,9;  
35 REF DL.5 REF. 2,5; \{B:=C+D\}  
37 REF DL.7 REF. 2,7;  
84 ADD F ADD. 5;  
C3 AS ND L.3 ASNL. 2,3;  
0F RE FS L.F REF. 1,15; \{increment count\}  
11 LIT4A.1 LIT. 1,1;  
E4 ADD ADD. 1;  
4F AS NSL.F ASNL. 1,15;  
2119 LIT8N JUMPT. L#3011; \{go to loop test\}  
59 SKIP
\texttt{L\#3010:;}  \\  \texttt{0000 \ 8125 \ LIT32 \ LIT. \ 2,1.00000000;}  \\  \texttt{00 \ 84 \ ADDF \ ADD. \ 5;}  \\  \texttt{C1 \ ASNDL.1 \ ASNL. \ 2,1;}  \\  \texttt{L\#3000:;}  \\  \texttt{0000 \ 23 \ CALLI \ CALLG. \ dummy.adatestsf.0001;}  \\  \texttt{31 \ REFDL.1 \ REFL. \ 2,1;}  \\  \texttt{L\#3000:;}  \\  \texttt{0004 \ 23 \ CALLI \ CALLL. \ add\_seven.adatestsf.0000;}  \\  \texttt{C3 \ ASNDL.3 \ ASNL. \ 2,3;}  \\  \texttt{10 \ 18 \ LIT8 \ PROCEND. \ 16,0;}  \\  \texttt{5F \ RETURN \ PKGDEF. \ $init.adatestsf.0000,12;}  \\  \texttt{0000 \ {procedure \ header} \ L\#4000:;}  \\  \texttt{10 \ LIT4A.0 \ PKGEN. \ 0; \ {null \ procedure \ body} \ L\#4000:;}  \\  \texttt{5F \ RETURN \ FINI}
AN EVALUATION OF ROCKWELL'S ADVANCED ARCHITECTURE MICROPROCESSOR FOR DIGITAL SIGNAL PROCESSING APPLICATIONS

by

KENNETH LEE ALBIN

B. S., Kansas State University, 1981

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

1984
This thesis examines the architecture of Rockwell's Advanced Architecture Microprocessor (AAMP) and predicts performance on signal processing algorithms. Performance that can be achieved with high-level languages is also investigated.

The Electrical and Computer Engineering Department at Kansas State University, in conjunction with Sandia National Laboratories, has attempted to identify processors which are most appropriate for implementation of real-time adaptive linear prediction in intruder detection devices. The ideal processor would require very little power, be easy to interface, perform multiplications very quickly and use floating-point arithmetic. The AAMP is a CMOS/SOS microprocessor that has a stack architecture with a 16-bit wide data path. Single and double precision integer and fractional as well as single and extended precision floating-point data types are supported on a single chip. It consumes approximately 50 mW at its rated 20 MHz clock rate and uses a single 5 volt supply.

This thesis consists of three parts. The first part is an introduction to the AAMP's architecture, instruction set and data structures. The second part details the investigation and findings from the evaluation. Included in this section is a discussion of ways to optimize the Widrow and Lattice algorithms for the processor's architecture. The third part contains the results and conclusions of the evaluation in a concise form.