PURDUE UNIVERSITY

Graduate School

This is to certify that the thesis prepared

By David Randall Beering

Entitled

"The Use of the AT&T DSP32 Digital Signal Processing Development System: Applications and New Developments"

Complies with University regulations and meets the standards of the Graduate School for originality and quality

For the degree of Master of Science in Electrical Engineering

Signed by the final examining committee:

Edward J. Delp, chair

Leah H. Jamieson

O. K. Mitchell

Approved by the head of school or department:

_____ 19__ _____

This thesis ☐ is ☒ is not to be regarded as confidential

Edward J. Delp
Major professor

Grad. School
Form No. 9
Revised 9-85

THE USE OF THE AT&T DSP32 DIGITAL SIGNAL PROCESSING DEVELOPMENT SYSTEM: APPLICATIONS AND NEW DEVELOPMENTS

A Thesis

Submitted to the Faculty

of

Purdue University

by

David R. Beering

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical Engineering

December 1987

To my parents

ACKNOWLEDGMENTS

I would like to thank Professor Edward. J. Delp and his research group for their help and support over the last two years. I would also like to thank Professors O. R. Mitchell and L. H. Jamieson for their support. Also, Wayne Peart of AT&T Technology Systems for many meaningful discussions.

TABLE OF CONTENTS

Page

Page

APPENDICES:

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Beering, David R. M.S.E.E., Purdue University. December 1987. The Use of the AT&T DSP32 Digital Signal Processing Development System: Applications and New Developments. Major Professor: Edward J. Delp.

The AT&T Western Electric DSP32 digital signal processing Development System is a high-speed 32-bit floating-point processor designed to handle many digital signal processing applications at speeds up to 8 MIPS. The processor is equipped with a full complement of programmable inputs and outputs, including a high-speed 8-bit CODEC (A/D and D/A). In addition to the hardware environment, the DSP32 supports a software development environment which includes a software simulator which aids in algorithm development.

The DSP32 and its support functions are introduced. The hardware and software environment are discussed in detail, and some of the supplied routines are examined and tested. In addition, we present a new library of one-dimensional digital filtering algorithms, including rank-order operations, α trimmed-mean operations, and median filtering. Also presented is a group of image processing algorithms which includes morphological operations, window smoothing, Laplacian enhancement, and the median filter.

Finally, we present an experiment in real-time speech processing using the DSP32. Comparisons are made between the DSP32 and other devices in

its class, and timing comparisons are made with several minicomputers running similar algorithms.

CHAPTER 1
INTRODUCTION

The AT&T DSP32 Digital Signal Processing Development System is a 32-bit, 16 MHz, VLSI processor which is capable of operating at speeds up to 8 million floating-point operations per second. It is the third member of a family of single-chip digital signal processors developed by AT&T. The need for a fast, versatile processor migrated from telephone switching systems, where a variety of signal processing functions must be performed within the switch. These applications, among others, include filtering, high-speed modems, speech recognition, and multichannel signaling [Al86].

The DSP family of architectures provides a single device capable of handling any of these tasks at high speed. To fit the DSP to a certain application, the device is merely mask-programmed for the particular application. As the technology evolved, and VLSI techniques became a reality, the DSP family was strengthened with the DSP20 (the predecessor to the DSP32). The DSP20 was pin and architecture compatible with the original DSP, but offered twice the speed [BoH86].

Most recently, the DSP32 was released, capable of speeds of 4 million floating point operations per second, with the capability to run twice as fast if the internal architecture is utilized properly. Now the DSP32 is available in both the mask-programmable version as well as a part of the DSP32 Development System. The Development System consists of a DSP32 simulator which is used for program development. The simulator is supported by a development environment which contains a library of pre-programmed subroutines, a compiler for DSP32 programs, and an interface to both the simulated and actual DSP32. The interface, called "dsp3sim", provides access to all memory and registers, disassembly of instructions loaded in memory, breakpoint debugging, and user-defined conditional execution [BoG86], [BoH86].

CHAPTER 2
THE AT&T DSP32 DEVELOPMENT SYSTEM

2.1 INTRODUCTION

The DSP32 is a 100-pin array manufactured using 1.5 micrometer NMOS technology. The chip, with an area of approximately 81 square millimeters, has nearly 155,000 transistors. The device operates on a single external 5-volt power supply [BoH86]. The basic structure of the DSP32 is 2048 bytes of on-chip ROM memory, 4096 bytes of on-chip RAM memory, a control arithmetic unit (CAU), a data arithmetic unit (DAU), a serial I/O section, a parallel I/O section, a control section, and an on-board clock section (see Figure 2-1) [We86], [HoC86].

2.2 DATA ARITHMETIC UNIT (DAU)

The DAU is the main execution unit for signal processing algorithms. It contains four 40-bit floating-point accumulators, a floating-point multiplier and adder, and the data arithmetic unit control register (DAUC). The data arithmetic unit performs all multiply and accumulate operations on signal processing data and all data type conversion.

The DAU performs operations of the form (a = b + (c x d)) at 4 million instructions per second where a, b, c and d are real and floating point. The DAU accepts inputs from memory, I/O, registers, or from the 40-bit accumulators. Type conversions are also facilitated in the DAU and are of the type floating point to-and-from integer, and floating point to-and-from μ-law and A-law.

Arithmetic and logic functions are implemented in the Control Arithmetic Unit, since the architecture of the DAU does not lend itself to these implementations. The DAU multiplier and adder operate in parallel. Each requires one processor cycle. The pipeline of the CAU is not as deep as the pipeline of the DAU, making 16-bit integer arithmetic faster there.

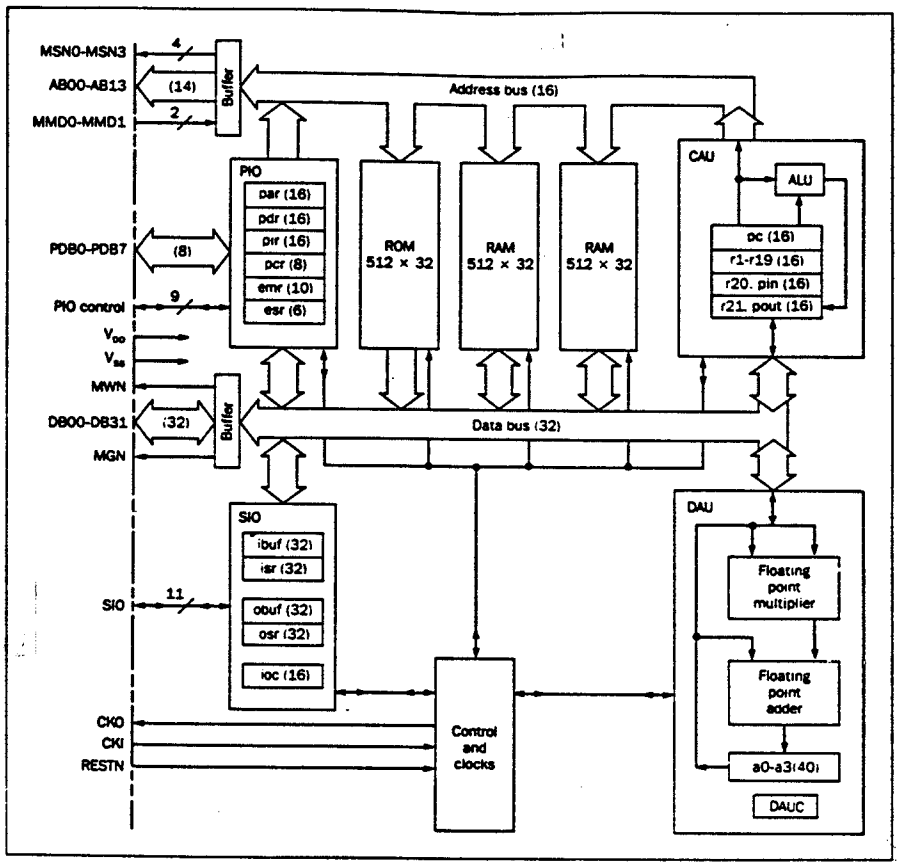

Figure 2-1. Block diagram of the DSP32 digital signal processor. The numbers in parentheses designate the size of a particular register or bus. The symbols at the left represent pins on the device package.

2.3 CONTROL ARITHMETIC UNIT (CAU)

Primary functions of the control arithmetic unit are generating addresses to memory and executing 16-bit integer instructions. The CAU has twenty-one 16-bit static general-purpose registers (r1 - r21), a 16-bit program counter, and an arithmetic logic unit (ALU). Registers r1 - r14 are used as general purpose registers in control arithmetic (CA) instructions and as memory pointers in data arithmetic (DA) instructions. Registers r15 - r19 are

used as general purpose registers in CA instructions and as increment registers in DA instructions. Registers r20 and r21 have other special purposes. Register r20 is the pointer in (PIN) and is used as the serial I/O DMA input pointer. Register r21 is used as the serial output DMA pointer. These two registers may be used as general purpose registers, however, their impact on DMA must be considered. The CAU generates one address every four processor clock cycles (see Figure 1-1 for the DSP32 block diagram).

The DSP32 processor cycle is divided into four stages. Due to the highly pipelined nature of the DSP32 processor, six different instructions are in some stage of maturity during each processor cycle. The stages of instruction execution are instruction fetch, two memory operand reads, instruction decode, and memory write.

2.4 MEMORY

The DSP32 provides 6 kilobytes of on-chip memory. This is allocated as 2048 bytes of ROM and 4096 bytes of RAM. Typically, fixed instructions and operands are stored in ROM and variable operands are stored in RAM. On-chip memory greatly increases processor efficiency, as off-chip delays due to memory access latency can negatively impact processor performance. On-chip RAM is dynamic and can be refreshed either automatically or by program control. All instructions are 32-bits and are fetched in one memory access. Three data types, 8, 16, or 32-bits, are provided. Memory is uniformly byte addressable.

Fifty-six Kbytes of directly addressable off-chip memory is also provided. Memory control signals from the processor are available at the off-chip memory making the interface transparent. The RAM and ROM can be arranged in four different configurations. These are selected on the device with the MMD front panel switches, and in software during program assembly (see Figure 2-2).

Memory in the DSP32 is divided into two banks, upper and lower. Although memory access can be made without regard to bank selection, effective memory access time can be cut in half if successive memory accesses are made to alternating memory banks. External memory can only be assigned to the lower bank of memory. The maximum amount of memory that can be addressed by the DSP32 is 64 Kbytes, however, the top 2 Kbytes of memory cannot be accessed.

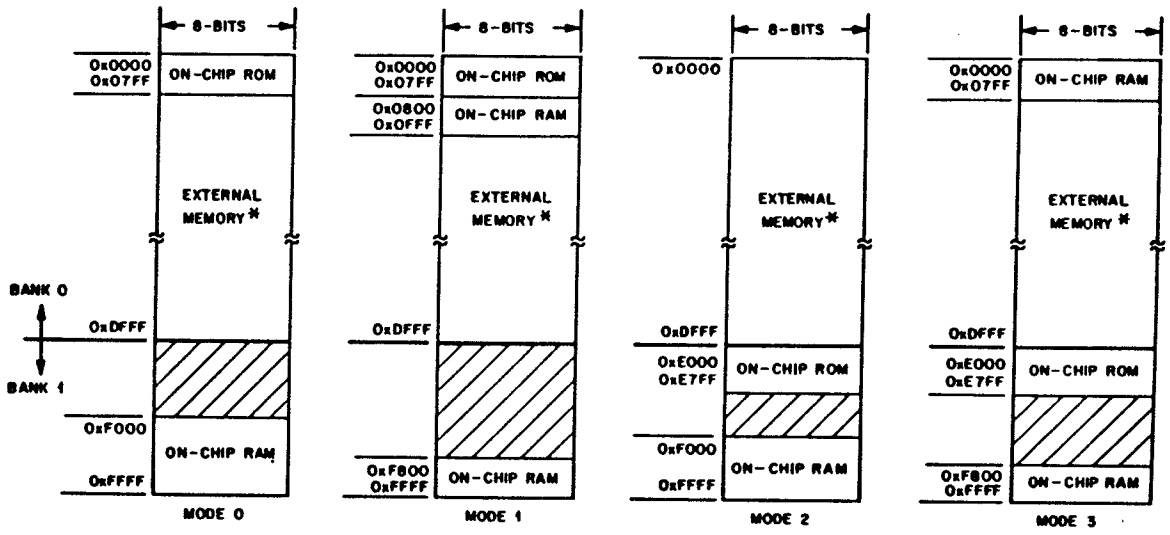

Figure 2-2. DSP32 memory address configuration.

2.5 SERIAL I/O

The serial I/O (SIO) unit allows the DSP32 to easily interface with other serial devices. The SIO uses a double buffering scheme to convert a serial input stream to parallel and parallel output data to serial. This is accomplished by loading input serial data into the input shift register in either 8-, 16-, or 32-bit widths, followed by loading the input buffer. This double buffering arrangement allows back-to-back transfers. Also, a second serial transmission can be started before the first has completed. External control signals allow a direct interface to a TDM line, a CODEC, and direct

DSP32-to-DSP32 interface for multi-processor applications.

Also provided is the facility for program-independent DMA using the input and output buffers. In this application, PIN and POUT are used as user-defined dedicated pointers for DMA transfers. This allows input and output buffers for entire frames of data to be established in memory.

2.6 PARALLEL I/O

Parallel I/O is supported by three 16-bit registers, PAR, PDR, and PIR; an 8-bit register, PCR; and a 6-bit register, ESR. These registers are used to interface to a microprocessor and to detect errors. Eight-bit transfers are accomplished between the parallel data bus and the the DSP32 using the parallel data register (PDR). These transfers may be controlled either by programs of DMA. If DMA mode is chosen, the microprocessor sets the starting address for the DMA transfer in the parallel address register (PAR). The control pins allow the microprocessor access to PDR and cause DMA access to proceed. This allows the loading of an application program or data with another program in progress. The parallel interrupt register (PIR) contains the DSP32 device output for the microprocessor. It also triggers the parallel interrupt (PINT) output pin in the PIO control group. This signal is also used by the error interrupt section. All registers and their functions are summarized in Table 2-1.

2.7 DATA TYPES AND ADDRESSING MODES

The DSP32 supports three data formats: floating point, integer, and companded data. Floating point representation is 32-bit format and is used for multiply and accumulate commands. Integer format is 16-bit and is used for addresses and fixed point data. Companded data format is 8-bit and is used with A-law and μ-law.

Floating point data is defined by a 24-bit mantissa and an 8-bit exponent. The most significant bit of the mantissa is the sign bit (see Figure 2-3). Each DAU multiply/accumulate instruction has three floating point operands. Two of these operands are multiplied together, and the result is added to the third. The result of the addition is stored in an accumulator. The resolution of multiply/accumulate operations is 40-bits. Therefore, the result of a multiply/accumulate operation must first be truncated to 32-bits

before it may be written to memory. As a result, tight loops with successive multiply/accumulate commands followed by memory writes are likely to produce truncation errors.

```
sffffffffffffffffffeeeeeeee
< -----24----- > < -8- >
```

Figure 2-3. DSP32 floating point format.

The integer data type is 8 or 16-bit two's complement integers. The 16-bit integers can either be signed (value between -32768 and 32768) or unsigned absolute addresses (range 0 to 65535). Companded data are 8-bit compressed PCM formats.

The four addressing modes supported by the DSP32 instruction set are immediate, memory direct, register direct, and register indirect. While these addressing modes lend themselves well to the DSP32 architecture, they also provide a set of primitive addressing modes upon which to build more complex addressing modes. The DSP32 addressing modes are summarized in Table 2-2.

2.8 CONNECTING THE DSP32 TO THE HOST COMPUTER

The DSP32 is designed to interface directly to a host computer running either the UNIX operating system or MS-DOS. Connections are supplied for either parallel or serial operation. The particular DSP32 Development System discussed here is connected to an AT&T 3B2/310 running UNIX System V. The DSP32 communicates with the 3B2 through a 9600-baud RS-232 serial line. The hardware configuration of the Development System and supporting peripherals is depicted in Figure 2-4.

The DSP32 is connected to the 3B2 on the console terminal port with an IBM 3151 terminal controlling I/O. For normal operations, the DSP32 passes all communications between the console terminal and the 3B2, essentially "listening" for special instructions. As a result, the 5620 terminal operates as /dev/tty11 rather than as the console, since the DSP32 is completely disabled (confused) when "layers" is invoked on its terminal. The

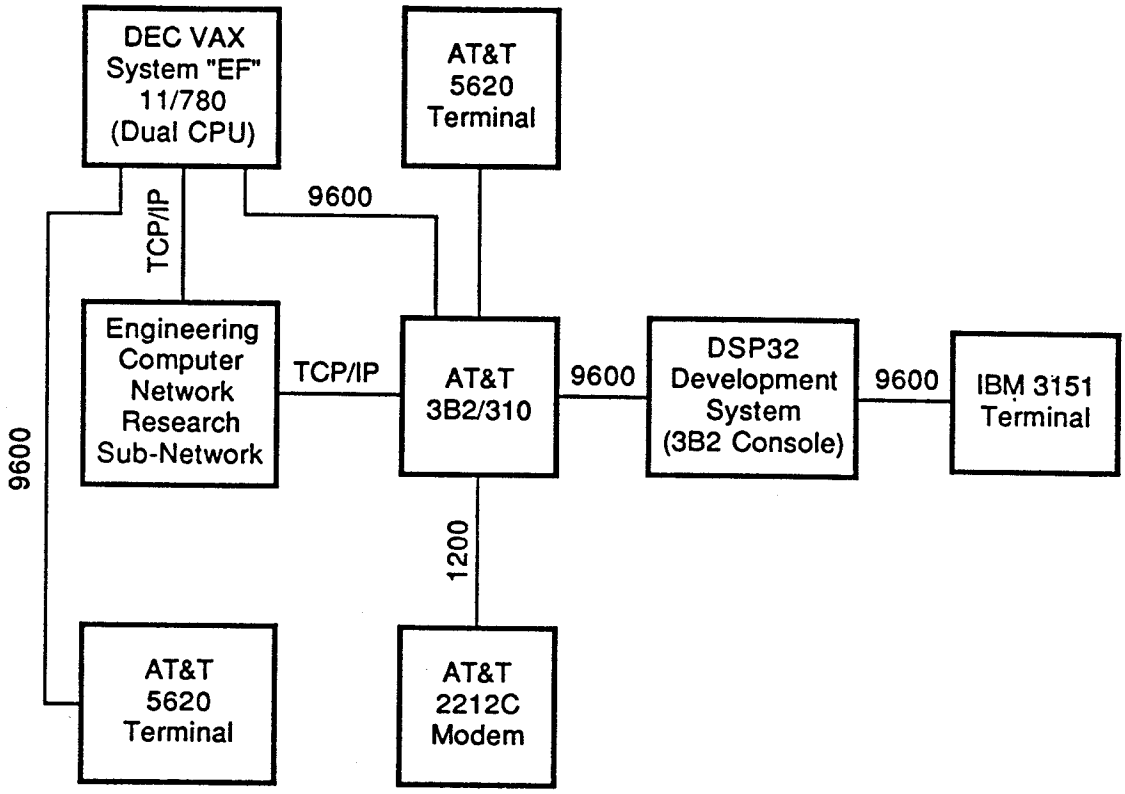

Figure 2-4. The DSP32 development environment.

IBM 3151 terminal emulates a Wyse50 terminal.

Two commands directly affect DSP32 operation; **dsp3sim** and **dsp3init.** When **dsp3sim** is used with the **-d1** option, the Development System is reset and readied for a run. If a file is supplied on the command line, the DSP32 is loaded with the file. **dsp3init** is used to initialize the DSP32 for a particular computer system. The two options used with **dsp3init** are the hardware port to which the DSP32 is connected and the number of DSP32 units connected to the system (up to seven Development Systems may be connected in series). For single-system implementation,

when the system is properly initialized, the LED display on the front panel of the unit will show ".1", with the "1" representing the logical unit number of the Development System, and the decimal point indicating that the Development System is selected for operation.

Table 2-1: The DSP32 registers.

Reg. Name	Bits	Description
Control Arithmetic Unit (CAU)		
pc	16	Program Counter
r1—r14	16	Pointers (Data Arithmetic) or General-Purpose (Control Arithmetic)
r15—r19	16	Increments (DA) or General-Purpose (CA)
r20	16	PIN, Serial DMA Input Pointer, or General-Purpose (CA)
r21	16	POUT, Serial DMA Output Pointer, or General-Purpose (CA)
Data Arithmetic Unit (DAU)		
a0—a3	40	Accumulators
DAUC	3	Data Arithmetic Control Register
Serial I/O (SIO)		
IOC	16	Serial I/O Control Register
IBUF	8/16/32	Serial Input Buffer
OBUF	8/16/32	Serial Output Buffer
Parallel I/O (PIO)		
PCR	8	PIO Control Register
PAR	16	PIO Address Register
PDR	16	PIO Data Register
PIR	16	PIO Interrupt Register
EMR	10	Error Mask Register
ESR	6	Error Source Register

Table 2-2: DSP32 addressing modes.

Mode	Notation	Description
Immediate	N	The operand is supplied by the instruction. Since all DSP32 instructions are a single 32-bit memory reference, this mode is not possible for DAU 32-bit operands.
Memory Direct	*N	In the direct mode, the instruction contains the address of an operand. That operand can be contained in any storage location, but if the operand is in memory, the mode is usually referred to as direct. A direct address is 16 bits and therefore cannot be applied to DAU operands.
Register Direct	*N	REG is the name of the register used as an operand. In various instructions the DSP32 allows REG = aN, rP, obuf, ibuf, etc., as register direct operands. Compared to memory direct addressing, register direct addressing is faster and requires fewer bits to encode; however, it offers access to fewer operands.
Register Indirect	*rP *rP++ *rP-- *rP++rI	For indirect modes, the instruction contains the name of a storage location that contains the address of the operand. Register indirect means that this storage location is a register. These registers are always post-modified, as indicated by the notation on the left: by 0, by +1/+2/+4 (depending on the data type: byte/integer/float), by −1/−2/−4 (depending on the data type: byte/integer/float), or by the contents of another register. This mode is useful for signal processing algorithms in which the operands are organized in tables.

CHAPTER 3
THE DSP32 PROGRAMMING ENVIRONMENT

3.1 THE DSP32 INSTRUCTION SET

There are two types of instructions implemented in the DSP32 instruction set. They are data arithmetic (DA) instructions and control arithmetic (CA) instructions. Data arithmetic instructions are handled by the data arithmetic unit (DAU), whereas control arithmetic instructions are handled by the control arithmetic unit (CAU).

The DAU performs two basic types of operations; multiply/accumulate functions and type conversions. The DAU has a 4-stage pipeline. The CAU instructions are 16-bit integer instructions. CA instructions perform arithmetic and logic operations and branch control. The CAU arithmetic is two's-complement fixed point. With few exceptions, the CAU will complete the current instruction before the next instruction begins execution, with the exception being loading registers from memory. The DSP32 flags are affected by both CAU and DAU instructions and certain I/O instructions. These flags are not directly accessible to the user, however, they may used for testing by conditional instructions. Flags are summarized in Table 3-1. A brief description of the instruction set and programming environment follows [We86].

3.2 DATA ARITHMETIC INSTRUCTIONS

Data arithmetic instructions are divided into two groups; multiply/accumulate instructions and special instructions mainly concerned with type conversion. If a multiply/accumulate instruction uses an accumulator as its input, the value of the accumulator is the value established three instructions earlier due to pipeline latency. Therefore, if an accumulator value is set in instruction I1, its value will not be accessible until instruction I4. Any instruction referencing the value of the accumulator during instructions I2 or I3 will see the previous value of the accumulator.

Data arithmetic Instructions are summarized in Table 3-2.

Data types supported by the Data Arithmetic Special Functions Group are integer, float, A-Law and μ-Law companded. The special functions group performs data type and format conversions and rounding. The data arithmetic special functions group instructions are summarized in Table 3-3.

3.3 CONTROL ARITHMETIC INSTRUCTIONS

3.3.1 The Control Group

The control group instructions allow the alteration of instruction flow from one instruction to the next. Included in these instructions are conditional and unconditional branching, loop counter testing, and subroutine call and return. The condition of branching can be based either on an CA, DA, or I/O events. Conditions are generated by flags set by either CA or DA group instructions. The Control Group instructions themselves have no effect on the flags. The CAU Control Group instructions are summarized in Table 3-4.

The integer N, referred to in CAU instructions, is a 16-bit integer which can be either a signed fixed point representation, or an unsigned immediate address. N used as an immediate address may be represented in decimal, as an unsigned octal number preceded by "0", or as an unsigned hexadecimal number preceded by "0x". N is assembled as an unsigned binary integer.

If it is necessary to insert no-ops in the code, the command "n*nop" will insert n no-ops in the instruction stream, where n is an integer. This is especially useful when implementing tight loops using accumulators. The default value of n is 1.

The instruction set supports six fundamental comparisons for signed arithmetic. Those are: not equal (ne), equal (eq), greater than or equal (ge), less than (ls), greater than (gt), and less than or equal to (le). Also implemented are carry clear (cc) and carry set (cs), which are used for multpile precision integer arithmetic. The sign of CAU register contents may be tested using plus (pl) and minus (mi). Replacement tables for CA Control Group Instructions covering both CA conditions and DA conditions are summarized in Table 3-5.

I/O events can be tested by the CA Control Group for the following: input buffer, output buffer, parallel data register, parallel interrupt register, sync signal and serial frame boundary. These conditions are summarized in Table 3-6.

3.3.2 The Arithmetic/Logic Group

These instructions perform all basic arithmetic and logical operations. Immediate integer or float data is not provided explicitly. Rather, immediate must be provided in the form of a CAU register. Therefore, operands in memory must first be moved to the CAU register array by using instructions from the data move group. The data move group allows data to be moved between memory and the CAU registers and between CAU registers and I/O registers.

Memory can be addressed either in 8-bit bytes or in 16-bit words. Likewise, data can be moved between CAU registers and memory in bytes or words. CAU registers may also be used as pointers to memory or as increments. Restrictions are that pointers are limited to r1 - r14, and increments are limited to r15 - r19 for DAU instructions. The arithmetic/logic and data move operations are summarized in Table 3-7. The entire instruction set is summarized in Appendix A.

Table 3-1: The DSP32 flags.

Flag	Test (State=1)	Meaning	Test (State=0)	Meaning
N	alt	Result is negative	age	Result not negative
Z	aeq	Result is zero	ane	Result not zero
V	avs	Result overflowed	avc	No overflow
U	aus	Result underflowed	auc	No underflow
CAU Flags				
n	mi	Result is negative	pl	Result not negative
z	eq	Result is zero	ne	Result not zero
v	vs	Result overflowed	vc	No overflow
c	cs	Carry or borrow out of MSB	cc	No carry or borrow
I/O Flags				
i	ibf	Input buffer full	ibe	Input buffer empty
o	obf	Output buffer full	obe	Output buffer empty
p	pdf	Parallel data register full	pde	PDR empty
P	pif	Parallel interrupt register full	pie	PIR empty
s	sys	SY (I/O sync pulse) set	syc	SY cleared
b	fbs	Serial I/O frame boundary	fbc	Not SIO frame boundary

Table 3-2: DSP32 data arithmetic (DA) instructions.

Instructions	DAU Flags Affected	Description
[Z =] aN = [−]aM {+,−} Y*X	NZVU	The product of the X and Y fields is added/subtracted to/from the accumulator (aM) and the result is stored in an accumulator (aN). The result can also be output according to the Z field.
aN = [−]aM {+,−} (Z=Y)*X	NZVU	The Y field operand is output according to the Z field. The product of the X and Y fields is added to the accumulator (aM) and the sum is stored in an accumulator (aN).
[Z =] aN = [−]Y {+,−} aM*X	NZVU	The product of the X field and the accumulator (aM) is added/subtracted to/from the Y field. The result is placed in an accumulator (aM) and can also be output according to the Z field.
[Z =] aN = [−]Y*X	NZVU	The product of the X and Y fields is added/subtracted to/from zero. The result is stored in aN and can also be output according to the Z field.
aN = [−](Z=Y)*X	NZVU	The value of the Y field is output according to the Z field. The product of the Y and X fields is stored in an accumulator (aN).
[Z =] aN = [−]Y {+,−} X	NZVU	The sum or difference of the Y and X fields is stored in an accumulator (aN) and the result output according to the Z field.
[Z =] aN = [−]Y	NZVU	The value of the Y field is placed in accumulator (aN) and also output according to the Z field.

Table 3-3: DSP32 data arithmetic, Special Function Group instructions.

Instruction	DAU Flags Affected	Description
[Z=] aN = ic(Y)	NZ00	Input Conversion μ-law, A-law to float
[Z=] aN = oc(Y)	—	Output Conversion float to μ-law, A-law
[Z=] aN = float(Y)	NZ00	integer to float
[Z=] aN = int(Y)	—	float to integer
[Z=] aN = round(Y)	NZVU	float(40) to float(32)
[Z=] aN = ifalt(Y)	—	if(alt) then [Z=] aN = Y else [Z=] aN

Table 3-4: Control Arithmetic Unit (CAU) Control Group instructions.

Instruction	Flags Affected	Description
if (CA COND) goto {rH, N, rH+N, rH−N}		Conditional branch
if (rM−−> =0) goto {rH, N, rH+N, rH−N}		Conditional branch
if (DA COND) goto {rH, N, rH+N, rH−N}		Conditional branch
if (IO COND) goto {rH, N, rH+N, rH−N}	None	Conditional branch
call {rH, N, rH+N, rH−N} (rM)		Call subroutine
return (rM)		Return from subroutine
goto {rH, N, rH+N, rH−N}		Unconditional branch
[L]*nop		No operation

Table 3-5: Replacement tables for CA Control Group instructions.

Value	CAU Flags*	Meaning
pl	n=0	Result is nonnegative (plus)
mi	n=1	Result is negative (minus)
ne	z=0	Result not equal to zero
eq	z=1	Result equal to zero
vc	v=0	Overflow clear, no overflow
vs	v=1	Overflow set, overflowed
cc	c=0	Carry clear, no carry
cs	c=1	Carry set, carry
ge	n^v=0	Greater than or equal to
lt	n^v=1	Less than
gt	z|(n^v)=0	Greater than
le	z|(n^v)=1	Less than or equal to
hi	c|z=0	Greater than (unsigned number)
ls	c|z=1	Less than or equal to (unsigned number)

Value	DAU Flags	Meaning
ane	Z=0	Not equal to zero
aeq	Z=1	Equal to zero
age	N=0	Greater than or equal to zero
alt	N=1	Less than zero
avc	V=0	Overflow clear, no overflow
avs	V=1	Overflow set, overflowed
auc	U=0	Underflow clear, no underflow
aus	U=1	Underflow set, underflowed
agt	N|Z=0	Greater than zero
ale	N|Z=1	Less than or equal to zero

Table 3-6: CA Control Group I/O event tests.

Mnemonic	Condition	Meaning
ibe	ibf=0	Input buffer empty
ibf	ibf=1	Input buffer full
obe	obe=1	Output buffer empty
obf	obe=0	Output buffer full
pde	pdf=0	Parallel data register empty
pdf	pdf=1	Parallel data register full
pie	pif=0	Parallel interrupt register empty
pif	pif=1	Parallel interrupt register full
syc	sy=0	Sync signal low
sys	sy=1	Sync signal high
fbc	fb=0	Serial frame boundary clear
fbs	fb=1	Serial frame boundary set

Table 3-7: Arithmetic/Logic and data move operations.

Instructions	CAU Flags Affected	Description
rD = rH+N	nzvc	Three operand add
rD = rD+rS	nzvc	Add
rD = rD−rS	nzvc	Right subtract
rD−{N, rS}	nzvc	Compare
rD = {N, rS}−rD	nzvc	Left subtract
rD = rD&{N, rS}	nz00	AND
rD&{N, rS}	nz00	Bit test
rD = rD|{N, rS}	nz00	OR
rD = rD^{N, rS}	nz00	XOR
rD = rS/2	nz0c	Arithmetic right shift
rD = rS>>1	0z0c	Logical right shift
rD = −rS	nzvc	Negate
rD = rS*2	nzvc	Arithmetic left shift

CHAPTER 4
THE DSP32 SIMULATOR

4.1 INTRODUCTION

The Western Electric DSP32 Digital Signal Processing Simulator, **dsp3sim,** is designed for real-time development support of digital signal processing applications. The programming environment provides a user interface to both the actual DSP32 Development System and a software-implemented virtual machine. Also provided is a means to manipulate both the real and virtual machines, including breakpointing, loading and examining registers and memory, and timing. This environment, as well as the program development process will be discussed below in detail [HoC86].

4.2 FEATURES AND FUNCTIONALITY OF THE DSP32 SIMULATOR

The simulator provides a "virtual chip," a software simulation of the actual DSP32 chip. The simulator provides 64K of memory (the maximum amount of memory the DSP32 can address). It should be noted, however, that the actual DSP32 chip cannot address a full 64K of memory, rather it is limited to 62K, since system functions reside in the upper 2K of memory.

The simulator provides user access to all registers, accumulators and memory. It supports user-defined functions and variables. The user can also define *exec* files which control the setup of a simulator session. *Exec* files can contain breakpoint definitions (breakpoints may be defined by assembly language labels, and several may be in effect at one time), lists of user commands, user-defined functions and labels, and control constructs.

For timing, a pseudo-real-time clock keeps track of CPU cycles, wait states, and DMA accesses. Wait states are caused when memory accesses are not interleaved and frequently during DMA operations. Limited emulation of I/O is also provided through the simulator. All serial I/O is synchronous with respect to the clock/counter above. It is also possible to set up the

simulator to flag conditions such as a write to ROM memory. Finally, an *escape to shell* feature, similar to the ":!" command in the *vi* editor, is provided so that **dsp3sim** need not be terminated to perform functions in the shell. See Appendix B for the complete DSP32 manual pages.

4.3 AN EXAMPLE USING dsp3sim

This example was extracted directly from the Western Electric DSP32-SL Support Software Library User Manual [We86]. In this discussion, numeric constants will often be represented as the C programming language would use them. They are used this way in the DSP32 assembly language and in interacting with **dsp3sim.** In particular this means that when a constant has a leading zero, as in 0010, it is octal, while to represent a hexadecimal number, we use a leading "0x", as in 0xa4.

When interactive sessions running **dsp3sim** are presented here, the users' input will be in bold type, while automatic responses from the computer will be in ordinary type. The following example, while extremely simple, will illustrate much of the potential of the simulator. This 2-line sequence will produce the absolute value of a floating point number.

In particular

```
a1 = -*r2
*r2 = a1 = ifalt(*r2)
```

has the effect of setting

$$*r2 = a1 = |*r2| \quad (|x| == \text{absolute value of } x)$$

i.e. it replaces whatever is at location r2 with its absolute value, and also leaves that same value in a1. Note that the action will not be complete until 4 instructions after the 2-line sequence (due to latency). Now consider the following short program contained in the file **abs.s.**

```
.global     abs_end, x
            r2 = x
            a1 = -*r2           /* 2-LINE */
            *r2 = a1 = ifalt(*r2)/* SEQUENCE */
```

```
                    4*nop
abs_end:            2*nop
x:                  float -11.0
```

Note that **.global** is required for any label to be referred to by **dsp3sim.** If we can run the program, get it to stop at location abs_end, and then display the contents of location x and of accumulator a1, then we can test the sequence for the single value -11.0. To assemble and load **abs.s** type:

```
dsp3make abs
    dsp3as -l abs.s
    dsp3ld abs.o -o abs
```

The simulator can now be used as follows:

```
$ dsp3sim abs
DSP32 Simulator Release 2.3
$im: bpset abs_end
bp -1 set at addr 0x1c
$im: run
$im: *x.f
*x = 11
$im: a1.f
a1 = 11
$im: quit
```

$im: is the simulator prompt.

The statement **bpset abs_end** set a breakpoint at **abs_end;** the program was run halting at location **abs_end,** and *x and a1 were examined (.f causes display to be a floating point number).

For testing a number of different values of x, it is useful to create a file of commands that will execute when when the program is started. The file should be names abs.ex and contain the following:

```
bpset abs_end;
var v= 0.0;
go{*x = v; run; *x.f; *a1.f};
```

In the following, the **-e** option is used to cause **dsp3sim** to read **abs.ex.** The **-b** flag causes a message to be printed each time a breakpoint is hit. To use the program to find $|N|$, set v = N and type **go:**

```
$im: dsp3sim abs
DSP32 Simulator Release 2.3
bp -1 set at addr 0x1c
$im: v = -21.3
$im: go
breakpoint -1 at 0x001c decode:nop;
*x = 21.3, a1 = 21.3
$im: v = 4.4
$im: go
breakpoint -1 at 0x001c decode:nop;
*x = 4.4, a1 = 4.4
$im: quit
```

v is an example of a user variable. **go** is a user-defined function. The **-s** option and **abs.ex** fit together because every time the program is tested, it may be useful to set a breakpoint at **abs_end** and to have the **go** sequence available.

4.4 THE DEVELOPMENT PROCESS USING dsp3sim

The following describes the development process for digital signal processing applications on the DSP32. All programs for the DSP32 Digital Signal Processing Development System are written in a C-style assembly language. All variables and labels used in DSP32 programs must be declared and all constants must be defined. Care must be taken to avoid type clashes and problems arising from pipeline latency (especially when using accumulators).

All source programs written for the DSP32 must have a **.s** extension. Similarly, all command files (files executed upon invoking the simulator) must have an **.ex** extension. For instance, the source file for a sort routine might be called **sort.s,** and its command file called **sort.ex.**

Once the program has been written, it is compiled using the **dsp3make** command. The **dsp3make** command combines the functions of assembling and linking the program. The **dsp3make** utility automatically inserts the

appropriate library routines necessary for successful compilation. **dsp3make** has options for setting memory mode and breakpoint capability. These are described in detail in Appendix B.

If the assembler compiles the file successfully, it creates a file with a **.o** extension. Otherwise, it will output appropriate error messages and no object file will be written. Next, the linker will output a file with a .l extension. The executable will be output in a file with no extension. For the sort program above, the object file would be called **sort.o,** the link file called **sort.l,** and the executable called **sort.**

The program is now ready for testing on the simulator. The DSP32 simulator has two main modes: virtual and real. The virtual mode (Mode 0) is an environment identical to the actual DSP32 machine, but implemented entirely in software. The real, or live mode (Mode 1), is an interface to the actual DSP32 machine. The simulator is invoked by typing dsp3sim filename at the UNIX prompt. This will automatically load the file into the simulated DSP32 memory to be run. **dsp3sim** also has several options. These options are for memory mode, real or simulated DSP32, and whether or not a command file is to be included in the load.

To enter a virtual DSP32 session with a command file using memory mode 0 (MM0) with the sort program above, one would type:

dsp3sim -m0 -e -d0 sort

Similarly, to invoke the actual DSP32 machine with the same parameters, one would type:

dsp3sim -m0 -e -d1 sort

The options for **dsp3sim** described in detail in the DSP32 manual pages listed in Appendix B.

All program testing should be performed on the virtual DSP32. This environment supports debugging using multiple breakpoints, as well as the capability to single-step the virtual machine. Due to pipeline latency, the actual DSP32 cannot be single-stepped. A synopsis of the DSP32 simulator commands is provided in Appendix C.

CHAPTER 5
SOFTWARE TOOLS

5.1 INTRODUCTION

The DSP32 Digital Signal Processing Development System is provided with a library of digital signal processing routines for a variety of applications. These routines are divided into three major groups: the math group, the matrix group, and the filter group. A summary of these routines is provided in Appendix D. The math group provides routines for transcendental functions, floating point division and format conversion. The matrix group includes optimized matrix multiplication routines for several sizes of square matrices as well as a general matrix multiplication routine. The filter group includes several varieties of Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filters, and Fast Fourier Transforms. Three of these routines will be discussed in detail: the Finite Impulse Response Filter, the Infinite Impulse Response Filter, and the Fast Fourier Transform.

5.2 THE FINITE IMPULSE RESPONSE (FIR) FILTER

The FIR filter offers absolute control over the phase characteristics of the filter. Non-recursive FIR filters are always stable and allow one to take advantage of the computational speed of the FFT algorithm. One disadvantage of the FIR filter, however, is that FIR filter design techniques are not as advanced as infinite impulse response filter design techniques.

Implementation involves limiting the impulse response of the filter by using a window. The most straightforward approach utilizes a square window. However, this technique causes disturbances in the cutoff band (the Fourier Transform of a square wave is a sinc function). More common, and less straightforward implementation involves the use of more streamlined windows. These windows reduce the disturbance at the stopband at the expense of wider window width, making the cutoff less defined.

The FIR filter implemented in the DSP32 library is taken directly from Oppenheim and Schafer [OpS75]. The filter is defined by the convolution sum:

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

Where x(n) is the input sequence, y(n) the output sequence, and h(n) the impulse response. The code requires 64 bytes of memory. Four bytes of memory are required for each input coefficient. A complete program listing for the FIR filter routine can be found in [Hat86]. Figure 5-1 shows the input signal used for both IIR and FIR filter analyses, consisting of two additive cosine waves at 500 Hz and 3250 Hz. Figure 5-3 shows the impulse response of a lowpass FIR filter. The impulse response is 13 points long with zero crossings every two sample points and approximates a sinc function, and hence, an "ideal" lowpass filter. This will provide a cutoff frequency of 2 KHz, since the sampling frequency is 8 KHz, and the folding frequency of the system is 4 KHz. The magnitude of the sinc function was normalized to provide unity gain. Figure 5-2 shows the output of the 2 KHz lowpass FIR filter when the signal in Figure 5-1 is used as the input. Examining Figure 5-2 shows that the FIR filter effectively removed the higher frequency cosine signal. Program execution time is related to the input data size N by the following [HaT86]:

Number of instruction cycles = (12 + 2 x N)

5.3 THE INFINITE IMPULSE RESPONSE (IIR) FILTER

Much of the basis for digital IIR filter design comes from analog filter design. IIR filters provide a better amplitude response at the expense of nonlinear phase response [OpS75]. The philosophy behind the design of IIR filters involves the pole-zero placement of the transfer function in the z-domain. It is the placement of the poles and zeros (and, of course, the sampling rate) which determines the effective cutoff frequency.

Since the art of IIR filter design is so advanced, there are many ways to implement an IIR filter function. The most straightforward of these is a direct digital implementation of the associated analog filter function. The

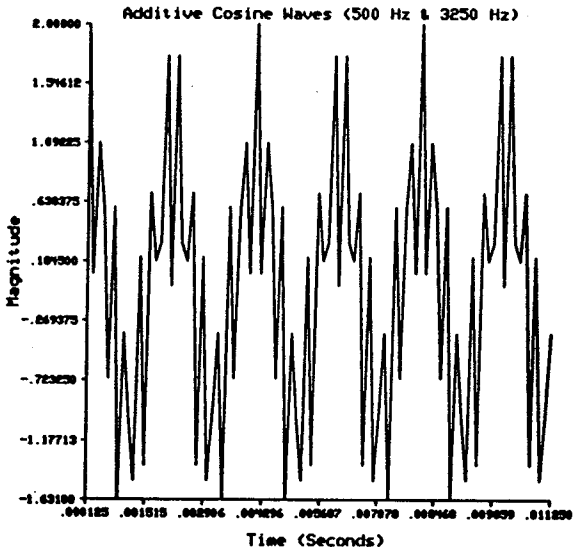

Figure 5-1. Input signal for the FIR and IIR filters consisting of two additive cosine waves with frequencies of 500 Hz and 3250 Hz.

IIR filter implementation in the DSP32 library is the direct form I (cascade second-order sections with four multiplies per section) taken from Oppenheim and Schafer [OpS75].

$$H(z) = A \prod_{k=1}^{[(N+1)/2]} \frac{1 + \beta_{1k} z^{-1} + \beta_{2k} z^{-2}}{1 - \alpha_{1k} z^{-1} - \alpha_{2k} z^{-2}}$$

The routine code occupies 68 bytes of DSP memory and the number of DSP instructions required to run the routine varies with data size by the following:

Number of instruction cycles = (14 + 5 x N)

The source code for the IIR filter can be found in [HaT86]. A normalized analog second-order Butterworth filter transfer function served as the basis for our example for generating the filter coefficients. Using a sampling rate of 8 KHz, the effective folding frequency of the system is 4 KHz. The signal consists of the two additive sinusoids as shown in Figure 5-1.

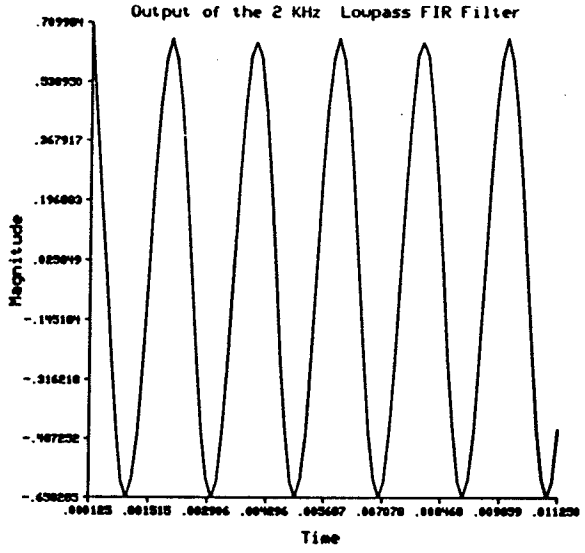

Figure 5-2. Output signal after being lowpass filtered at 2 KHz using a FIR filter. The input signal is shown in Figure 5-1 and the impulse response is shown in Figure 5-3.

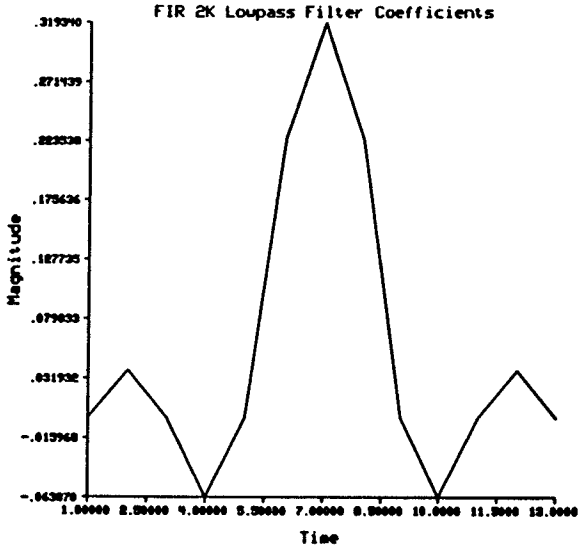

Figure 5-3. The impulse response of the 2 KHz lowpass FIR filter

The normalized analog transfer function for the second-order Butterworth filter is given by the following [StD84]:

$$G(s) = \frac{1}{1 + 1.4142136s + s^2}$$

Using the folding frequency f_0 of 4 KHz and a cutoff frequency f_r of 500 Hz, the constant multiplier C, which assures unity gain, can be determined by the following:

$$C = \cot \frac{\pi}{2} \left(\frac{f_r}{f_0}\right)$$

Now, by combining with:

$$s = C \frac{(1 - z^{-1})}{1 + z^{-1}}$$

the transfer function in terms of z can be established, with the filter coefficients being the constant multiplier C and the coefficients of the z^{-1} and z^{-2} terms. Figure 5-4 shows the output of the IIR filter when applied to the input signal of Figure 5-1. The 500 Hz cutoff frequency allowed the 500 Hz cosine wave to pass, while eliminating the cosine at 3250 Hz.

5.4 THE FAST FOURIER TRANSFORM (FFT)

It is often necessary to analyze digital signals in both the time domain and in the frequency domain. In digital systems, signals may be represented by the Discrete Fourier Transform. The Fast Fourier Transform is an algorithm which effectively reduces the Discrete Fourier Transform computation. When this reduction takes place in the time domain it is known as decimation-in-time. Reduction in the frequency domain is known as decimation-in-frequency.

The DSP32 library provides a decimation-in-time FFT algorithm which is capable of handling input data up to 1024-points in length. The algorithm was adapted from Rabiner and Gold [RaG75]. The routine occupies 348

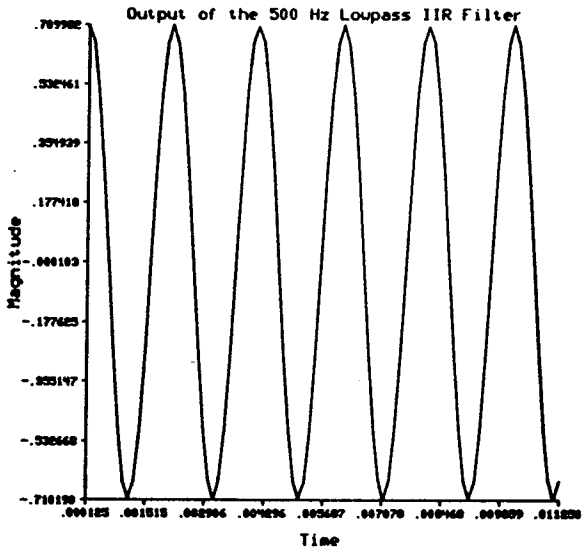

Figure 5-4. Output signal after being lowpass filtered at 500 Hz using the IIR filter.

bytes of DSP memory. The routine supports input data in complex format (i.e. each input data point is composed of a real part and an imaginary part). Therefore, each data point in the input data array occupies 8 bytes of memory. Some timing for the algorithm follows [We86]:

Data Size N	No. of Instructions	No. of Wait States	Time (ms) 16 MHz
64	3415	847	0.91
128	7271	2001	1.95
256	15467	4627	4.16
512	32711	10517	8.84
1024	69067	23575	18.74

Figure 5-5 shows the output of the FFT routine when it is applied to the test signal, shown in Figure 5-1, consisting of the additive sinusoids at 500 Hz and 3250 Hz. The graph has appropriate peaks at 500 Hz and 3250 Hz. Leakage into other frequencies is due to the signals not being exact integer multiples of each other.

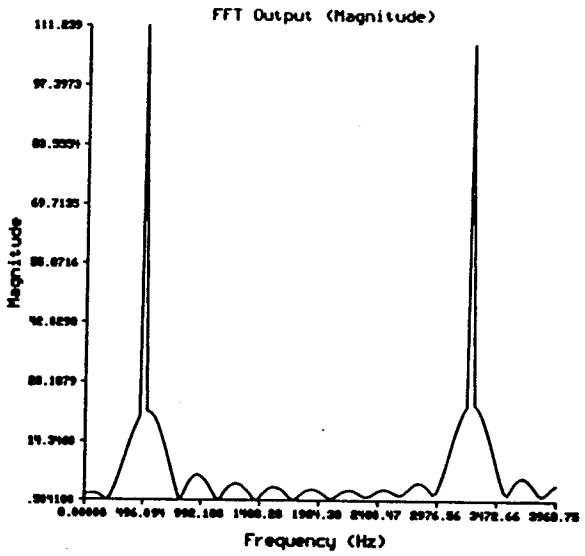

Figure 5-5. Magnitude of the Discrete Fourier Transform of the signals shown in Figure 5-1.

CHAPTER 6
CUSTOM TOOLS

6.1 INTRODUCTION

To investigate the utility of the DSP32 for specialized digitial signal processing tasks, a series of custom digital processing algorithms were developed. The custom program library consists of a one-dimensional rank-order filter and an α trimmed mean filter program. These two programs are discussed in detail below. The programs were first coded for testing using the DSP32 simulator. Once implementation and architecture problems were solved, the programs were implemented on the DSP32 Development System.

6.2 THE RANK-ORDER FILTER

The rank-order filter is a nonlinear that has been shown to be useful in various applications. Nonlinear techniques have shown to have promise, especially in environments where signals are corrupted by impulsive noise. Linear averaging filters are not as effective on impulsive data. The output of the filter can be chosen to be any ranked value within the window [ArG86]. For instance, if we choose the fourth largest element from a window of size seven, we have a fourth-ranked-order filter. These filters have also been shown to reject impulsive noise.

The version of the rank-order filter used here is a non-recursive implementation [ArG86]. The data is read into the input data space along with a header containing filter parameters. These parameters describe the window size, window center point, rank of the filter, and the size of the data to be filtered. The center point is provided to the program rather than calculated internally to eliminate a costly divide. The data space is limited by the data space definition to 96 data points.

The data is read into memory at location 1400 (hex). It is then copied to a temporary data space at 1800 (hex). The temporary data space maintains its integrity throughout the process, since it is the source for all

windows. Windows are extracted from the temporary data space and moved into a temporary sort space. In the temporary sort space, a modified bubble sort is performed on the data. Once the bubble sort has completed, the data is in rank sorted order from lowest to highest.

Once the data in the sort space has been sorted, the pointer to the sort space is incremented by the rank of the filter (given in input parameters). This pointer now points to the appropriate element to replace the middle of the original window. This data element is moved to the input data space at 1400 (hex) which now serves to accumulate output medians. The input data space is padded at the edges with the zeroes at the beginning and the end repeated up to ((2 x windowsize) - 1) elements. Using this method, the first several iterations of the window are ignored in the output data. This practice is inconsequential, since only the valid output elements are considered in the final analysis.

To test the operation of the filter routine, a known signal was filtered. First, the uncorrupted signal was filtered, followed by the same signal corrupted by guassian noise, then impulsive noise. The window size chosen for the test was five. The data size chosen for the test was 96 elements. Figure 6-1 shows the input signals used to test the filter routines. Figure 6-1 (a) shows the original cosine wave, corrupted by (b) gaussian noise with a SNR of 10 dB (discussed below) and (c) by impulsive noise. Figure 6-2 shows the output of the noise-corrupted signals after being filtered by the median filter. Figure 6-3 shows the output after filtering with the α trimmed-mean filter. The DSP32 ran the all of the one-dimensional filter routines in well under one second.

6.3 THE MEDIAN FILTER

The median filter involves moving an odd-sized window across the data in the time domain, with the filter output, Y(A) related to the filter input, X(A) by the following [ArG86]:

$$y(A) = \text{the median values of } [x(A - N),..., x(A - 1), x(A), x(A + 1),..., x(A + N)]$$

where $2N + 1$ is the window size.

Median filters have been shown to be very effective in preserving edges while removing impulses and rapid oscillations [ArG86]. The amount of

smoothing performed by a median filtering operation depends on window size. A larger window will perform more smoothing than a smaller window.

The median filter implemented on the DSP32 is merely a special case of the rank-order filter described above, where the rank is the middle value of the window. The input signal is shown in Figure 6-1, along with the signal corrupted by additive gaussian noise, and impulse noise. Figure 6-2 shows the outputs of the median filter for the two noise-corrupted input cases, using a window five data points wide. The gaussian noise was generated using the central limit theorem, where twelve random numbers were combined for each noise sample point. The SNR was set at 10 dB determined by the variance of the noise, calculated by the following:

$$\mathrm{SNR} = 10 \log_{10} \left\{ \frac{\frac{1}{N} \sum_{k=0}^{N-1} x_i^2}{\sigma^2} \right\}$$

Which represents the power of the signal divided by the power of the noise. Impulsive noise was generated using Bernoulli trials on each bit, where the probability of a bit being flipped was set at 5 percent (P_e=0.05). As can be seen in Figure 6-1 (d) and (e), the median filter was more effective removing impulsive noise than gaussian noise. This is due to the impulse-rejecting abilities of the median filter.

6.4 THE α TRIMMED-MEAN FILTER

The implementation of the α trimmed-mean filter using the DSP32 is very similar to the rank-order operator in that the α trimmed-mean filter performs ranking within the window. The α trimmed-mean operation is similar to an olympic scoring system. The data within the window are sorted from lowest to highest. The highest and lowest values (top and bottom elements) are then removed, and the remaining elements are averaged. The α trimmed-mean filter performs very well on signals corrupted by both impulsive and additive gaussian noise, since it strongly resembles both the median filter and linear averaging filters. The α trimmed-mean operation involves division, which is not directly supported on the DSP32.

As the operation proceeds, the input data is read into DSP32 memory at location 0x1400 (hex). The data is then copied to location 0x1800 (hex) for windowing and sorting. Each window is moved to a temporary sort space where the sort is performed. The sorting algorithm is a modified bubblesort. Once the data has been ranked within the window, the top and bottom elements are removed, and the remaining elements summed. The sum is then input into the divide routine as the numerator. The (window size - 2) is input as the denominator. The quotient from the divide routine is then placed at the output data location.

Figure 6-3 shows the output of the α trimmed-mean filter when applied to the input signals of Figure 6-1. The filter performed very well on both types of noise. Since the filter performs both ranking and averaging, it smooths impulse noise much more effectively than the median filter (see Figure 6-2).

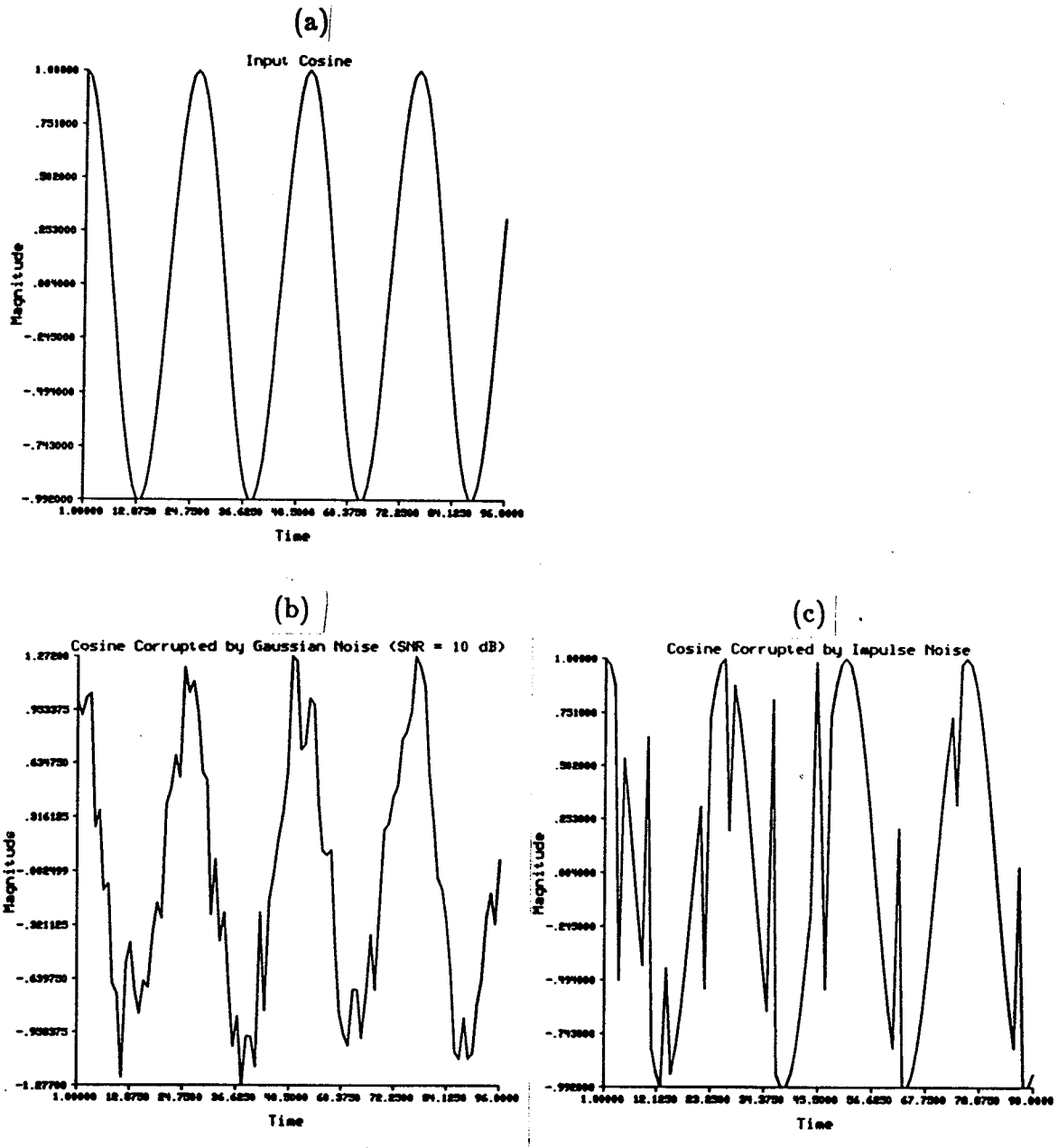

Figure 6-1. The input cosine wave (a) corrupted by additive gaussian noise (b) and impulse noise (c).

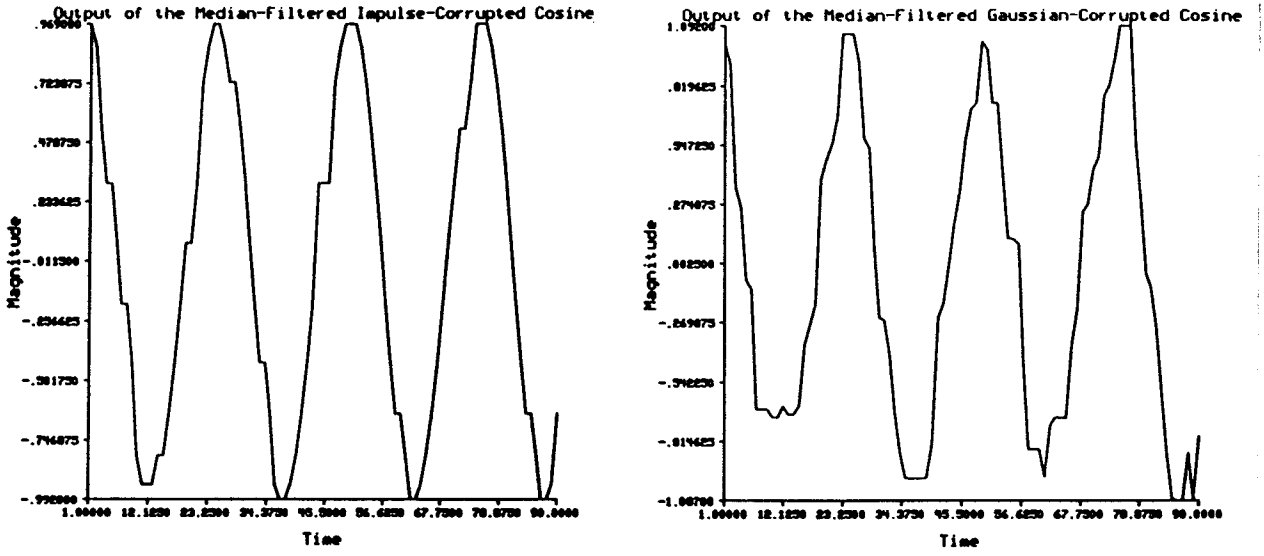

Figure 6-2. Output of the median filter when applied to the signals of Figure 6-1 (b) and (c).

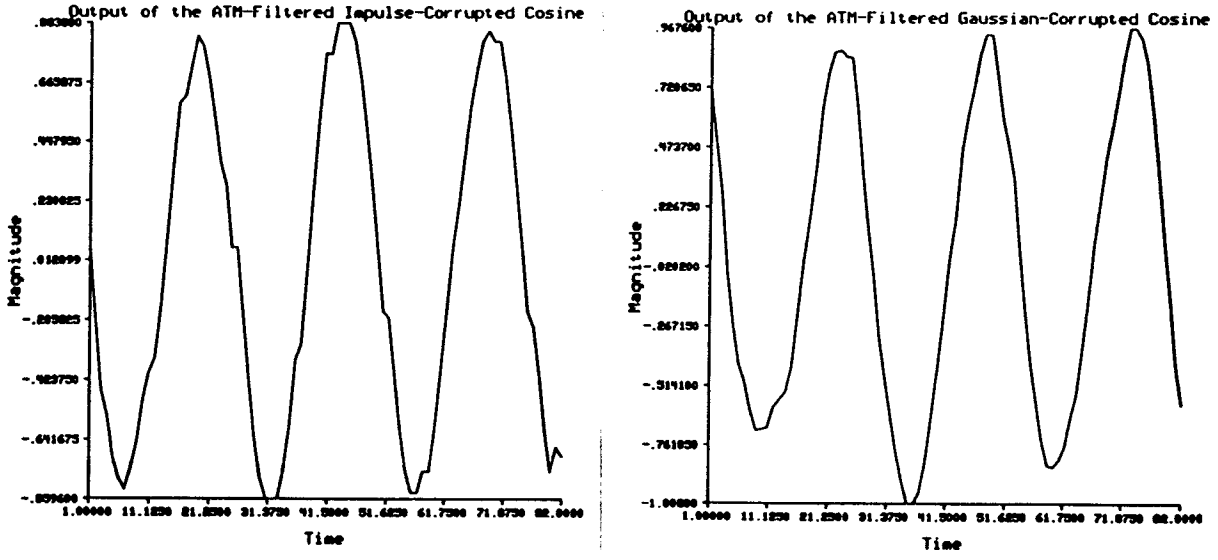

Figure 6-3. Output of the α trimmed-mean filter when applied to the signals of Figure 6-1 (b) and (c).

CHAPTER 7
IMAGE PROCESSING USING THE DSP32

7.1 INTRODUCTION

As an extension of the work discussed in the previous chapter for one-dimensional signals, we now focus our attention on the two-dimensional operations used in image processing. Several two-dimensional window programs were developed focusing on image morphology, smoothing, median filtering, and Laplacian operations [Ha86].

7.2 MORPHOLOGICAL OPERATIONS

Mathematical morphology uses window masks which, when operated on image data, emphasize desired elements of shape. For our purposes, we are concerned with binary images (pixels may either have a value of zero or 255). In order to completely describe a binary image, we need only define the set of black pixels, or the set of white pixels. Image morphology consists of the operations of dilation, erosion, opening, and closing. To perform the operations of opening and closing on the image, we utilize the primitive operations of dilation and erosion. The following sections will establish the necessary theoretical background for an understanding of the erosion, dilation, opening, and closing as implemented on the DSP32.

7.2.1 Dilation

Dilation combines the image data and a window in an element-by-element addition, where elements are represented by their indices. Physically, dilation is seen as an expansion of the set A by the set B. The dilation of set A by set B is denoted by $A \oplus \check{B}$ and is defined by [Se82]:

$$C = A \oplus \check{B} = \{z: B_z \cap X \neq \varnothing\} = \bigcup_{b \in B} X_{-b}$$

where A and B are subsets of the N-dimensional Euclidean space E^N and $\check{B}$ is the symmetric set obtained by rotating B 180 degrees. B_z is the translation of set B by z.

A simple example of dilation is given in [HaS86]. If the set A contains the elements (0,1),(1,1),(2,1),(2,2),(3,0) and the set B contains the elements (0,0),(0,1), the result of the dilation of A by B is the set C = {(0,1),(0,2),(1,1),(1,2),(2,1),(2,2),(2,3),(3,0),(3,1)}. The operation is shown in Figure 7-1.

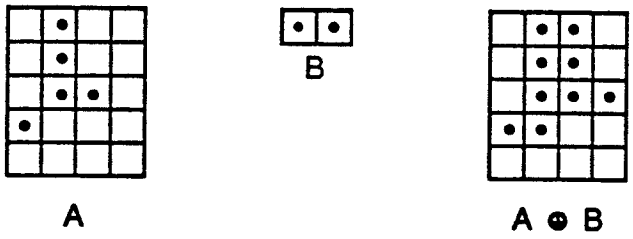

Figure 7-1 The result of dilating set A by set B.

Since the dilation operation is one of set addition, it is commutative and associative, since addition is commutative and associative. Even though the set A and B are symmetric with respect to the operation, the set A is considered to be the set being operated on and B is called the structuring element. The structuring element can be thought of as a probe that extracts shape information from the set A.

The dilation of the set A by the set B is most easily seen as a translation operation on the set A by the set B. The set C is then formed as a result of the arithmetic "OR" of all of the translations of A.

7.2.2 Erosion

Mathematically, erosion is the morphological dual of dilation. Physically, the erosion process shrinks the set A by the set B. The erosion of A by B is denoted by $A \ominus \check{B}$ and is defined by [Se82]:

$$C = A \ominus \check{B} = \{z: B_z \subseteq X\} = \bigcap_{b \in B} X_{-b}$$

Again, if we look at the erosion process as a translation operation on the set A by the set B, we can see that the resultant set C is formed by the arithmetic "AND" of all of the translations of the set A. For this process, B_z intersects the original set A. To demonstrate this, consider the set A = {(0,1),(1,1),(2,1),(2,2),(3,0)} which is to be eroded by the set B = {(0,0),(0,1)}. If we look first at the translation of the set A by the set B, we have the intermediate set which contains {(0,1),(1,1),(2,1),(2,2),(3,0)}, which is the translation of the set A by the element (0,0), and also the set {(0,2),(1,2),(2,2),(2,3),(3,1)} which is the translation of the set A by the element (0,1). If we "AND" the elements of the two intermediate sets together, we find the resultant eroded set C = {(2,2)}. It has been shown [HaS86] that the erosion will be more severe if the eroding element is a subset of the set A. It is also possible that, if the structuring element does not contain the origin, the erosion can result in a set which has nothing in common with the original set. This operation is shown in Figure 7-2.

7.2.3 Opening and Closing

The opening and closing operations are usually used in pairs, employing the primitive dilation and erosion operations described above. If dilations and erosions are applied iteratively, the result is the elimination of specific image detail which is smaller than the structuring element, while leaving other features geometrically undistorted.

The opening of image B by structuring element K is denoted by $B \bigcirc K$ [HaS86] and is defined as $B \bigcirc K = (B \ominus \check{K}) \oplus K$.

The closing of image B by structuring element K is denoted by $B \bullet K$ [HaS86] and is defined by $B \bullet K = (B \oplus \check{K}) \ominus K$.

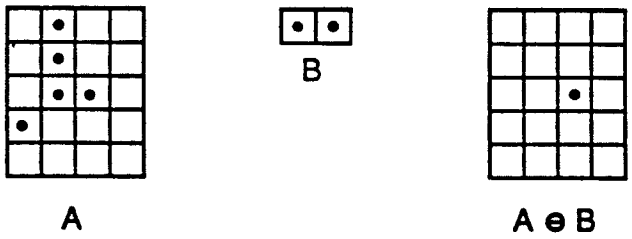

Figure 7-2. The result of eroding set A by set B.

Serra [Se82] describes the result of the opening on an image with a disk-shaped structuring element as smoothing the contour, breaking narrow isthmuses, and eliminating small islands and sharp peaks or capes. The closing on the image with the same structuring element then smooths the contours, fuses narrow breaks and long thin gulfs, eliminates holes, and fills the gaps on the contours.

These operations are idempotent. That is, successive iterations of the same operations will not alter the result. Simply defined, the opening operation is an erosion of an image by the structuring element, followed by a dilation of the result by the same structuring element. Likewise, the closing operation is defined as a dilation on the image by the structuring element, followed by an erosion of the result by that structuring element [HaS86].

7.2.4 Implementations on the DSP32

Since all of the image processing algorithms written for the DSP32 involve window operations, a generalized window program was developed. This program reads in the input data from a UNIX file which has a header containing information about data size and window size, followed by the actual data. After the data is read in, it is copied to a temporary location where the window operations on it can take place. The results of these operations are written back to the input data space. Since the input data cannot be placed in arrays, several pointers are necessary to make sure that the two-dimensional window is extracted properly from the one-dimensional data space. Each window is moved to a temporary window space. All of

the DSP32 image processing programs are identical in the respect that they move windows to the temporary window space from the one-dimensional data space. The morphological programs perform set operations on the window, the median filter sorts and ranks the window, the α trimmed-mean filter program sums over the window and averages, and the Laplacian program multiplies and sums across the window.

The results of applying the morphological operations defined above to a 64 x 64 binary image containing text are shown in Figures 7-3 and 7-4. The structuring element chosen for the example was a lower case "e", defined in a 5 x 5 window. The process of opening the image by the structuring element performed essentially the task of detecting occurrences of "e" in the text. The erosion reduced the text to several dots; one at the median location wherever an "e" had occurred ion the original text. The dilation which followed restored the dots to "e" in their original positions. This process demonstrates how the structuring element can be used to extract specific shapes from an image.

The results of the closing were not as dramatic. The dilation on the original text expanded the text by the structuring element, creating an image that hardly resembled the original. The following erosion reduced the image back to a "superset" of the input image. Many of the gaps inside letters and between letters were filled. Each morphological program ran in less than two seconds on the DSP32.

7.3 WINDOW SMOOTHING

Window smoothing is most often the simple averaging of pixels in the neighborhood around a pixel of interest. This straightforward averaging results in noise reduction at the expense of image blurring. Rosenfeld and Kak [RoK82] propose several improvements in the averaging scheme which deliver noise reduction with less image blurring. One scheme involves examination of the data in the Fourier domain, removing the noise, and taking the inverse transform to recover the original signal without noise. Another approach involves the assignment of probabilities to the signal and noise. The weighted average of the neighboring points is assigned to the point of interest.

The method of image smoothing chosen for the DSP32 smoothing program is a two-dimensional α trimmed-mean filter. The motivation for this

Figure 7-3. The original binary image (upper left) and the resulting images after eroding (upper right) and dilating (lower left) with the 5 x 5 "e"-shaped structuring element (lower right).

implementation is the similarity of the α trimmed-mean filter to the median filter. The α trimmed-mean filter ranks pixels within the window, removes the highest and lowest points, and averages the rest. If the window size is chosen to be small enough (3 x 3 in our case), high frequency and impulse noise can be effectively reduced, while retaining image sharpness. The input image used to test the filter was contaminated with zero-mean gaussian noise. The signal-to-noise ratio was 10dB, calculated using the method discussed in Chapter 6. Results of running the filter algorithm on the image are shown Due to the arithmetic intensity (namely, the division subroutine) of this algorithm, the DSP32 took nearly six seconds to filter the input data. in Figure 7-5. The image size is 64 x 64 pixels.

Figure 7-4. The original binary image (upper left), the opened image (upper right), and the closed image (lower right).

7.4 IMAGE SHARPENING USING THE LAPLACIAN

The Laplacian is another neighborhood operator which performs image sharpening by computing the second derivative of the image. Mathematically, for discrete signals, the Laplacian has the following form [RoK82]:

$$\nabla^2 f(i,j) = [f(i+1,j) + f(i-1,j) + f(i,j+1) + f(i,j-1)] - 4f(i,j)$$

where f(i,j) is the gray-level value at the i^{th} row and j^{th} column of the image.

When the Laplacian is combined with the original image, we arrive at the following:

$$5f(i,j) - [f(i+1,j) + f(i-1,j) + f(i,j+1) + f(i,j-1)]$$

Figure 7-5. The original image (upper left) contaminated by gaussian noise (upper right), and the α trimmed-mean-filtered image (lower right).

This is known as unsharped masking and tends to enhance edge structures in an image [RoK82]. As a window operator, the Laplacian looks like the following:

$$\begin{matrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{matrix}$$

and the unsharpened mask looks like:

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$$

The DSP32 Laplacian program works similarly to the previous image processing programs. The input data is read into DSP32 memory along with the integer header information. The windows are extracted from the data using the same window extraction routine. The extracted 3 x 3 window is multiplied by the Laplacian window and the sum of the products is placed at the label for the input data. The source code for the Laplacian operator can be found in Appendix I. To test the Laplacian operator, a blurry image was input to the DSP32. The input image and the enhanced output are shown in Figure 7-6. The operation took less than one second to run on the DSP32.

Figure 7-6. The original image (upper left), the blurry original image (upper right), the Laplacian-filtered blurry (lower left), and the unsharp mask-filtered blurry image (lower right).

7.5 THE 2-D MEDIAN FILTER

The two-dimensional implementation of the median filter on the DSP32 is similar in construction to the one-dimensional version of the program. The program, however, uses the window algorithm described previously to define the sort space. Again, the maximum data size supported by the program is 64 x 64 pixels. Once the window is extracted, it is sorted using a modified bubblesort. After the window elements have been sorted, the median value of the window is extracted and placed at the output data space. The window size used was 3 x 3.

Figure 7-7 shows a 64 x 64 pixel image which has been corrupted by impulse noise (probability of bit error = 5 %), and the resultant images after the median filter operation. The 2-D median filter ran in under one second on the DSP32.

Figure 7-7. The original image (upper left) contaminated by impulsive noise (upper right), the median-filtered original image (lower left), and the median-filtered noisy image (lower right).

CHAPTER 8
REAL-TIME SPEECH PROCESSING

8.1 INTRODUCTION

Since the DSP32 has its roots in telephone communications, it is not a surprise to find it useful for digital speech processing applications. Many of the routines easily implemented on the DSP32 are telephone/voice-related. Some of these include, real-time digital filtering, speech detection, tone detection, and pulse detection [Da84]. The DSP32 is capable of doing real-time speech processing with bandwidth up to 8 KHz. Most applications, however, do not demand bandwidth any greater than 4 KHz.

8.2 THE DSP32 CODEC

A CODEC (Coder/Decoder) is a device which converts analog signals to digital and digital signals back to analog (A/D and D/A conversion), by sampling, quantizing, and companding. The DSP32 is equipped with an on-board 8-bit CODEC to provide access to signals in real-time.

When a CODEC samples an analog signal, it converts the signal into a Pulse Amplitude Modulated (PAM) signal. The PAM signal is the actual value of the original analog signal at the time of sampling. This signal is then companded and quantized. Since the signal will spend more time near zero, a linearly quantized signal will suffer from more quantization error. Hence, the need for companding.

The most commonly used companding processes are A-law and μ-law. A-law companding is used in the United States primarily with digital telephone communications systems. The A-law compander uses a logarithmic transformation on the PAM signal. A-law quantization intervals are very closely spaced near zero, and spread farther apart as amplitude increases. This allows a better definition of signals at low levels which reduces quantization error and increases dynamic range. μ-law companding is the CCITT (Consultative Committee of International Telephone and Telegraph)

standard of companding. This standard is used primarily in Europe and Japan. μ-law companding also quantizes logarithmically. The differences are noted below. The formula for μ-law companding is [Sh79]:

$$|y| = \frac{\log(1 + \mu|x/x_{max}|)}{\log(1 + \mu)}$$

The formula for A-law companding is [Sh79]:

$$|y| = \begin{cases} \dfrac{A|x/x_{max}|}{1 + \log(A)}, & 0 \leq |x/x_{max}| \leq 1/A \\ \dfrac{1 + \log(A|x/x_{max}|)}{1 + \log(A)}, & 1/A \leq |x/x_{max}| \leq 1 \end{cases}$$

Where x is the input, y is the companded output, and x_{max} is the maximum input value. Typical values for A and μ are 100. The DSP32 A-law compander uses 100 as the value for A.

For our demonstration, we connected the DSP32 as shown in Figure 8-1. The input to the DSP32 came from an audio amplifier with a microphone as its input. The outputs of the amplifier were connected to the J4 BNC input terminal on the back of the DSP32 enclosure. The J5 output BNC port on the DSP32 was connected directly to an audio spectrum analyzer, and to a speaker [At86], [To86].

Three routines were run on the device. The first simply read data into a buffer and output it again without operating on it at all [Da84]. The second routine read data into the buffer, ran it through a 1 KHz lowpass IIR filter, and sent it to the output port. The third routine implemented the same lowpass IIR filter with a cutoff frequency of 500 Hz.

A-law companding was chosen for the speech processing demonstration. In the DSP32, all of the functions of sampling and companding are performed by the CODEC. The filter routines are designed to work with the PCM signal. The CODEC then expands the filter output to a PAM signal and smooths the output to an analog approximation of the filtered digital signal, i.e. D/A conversion.

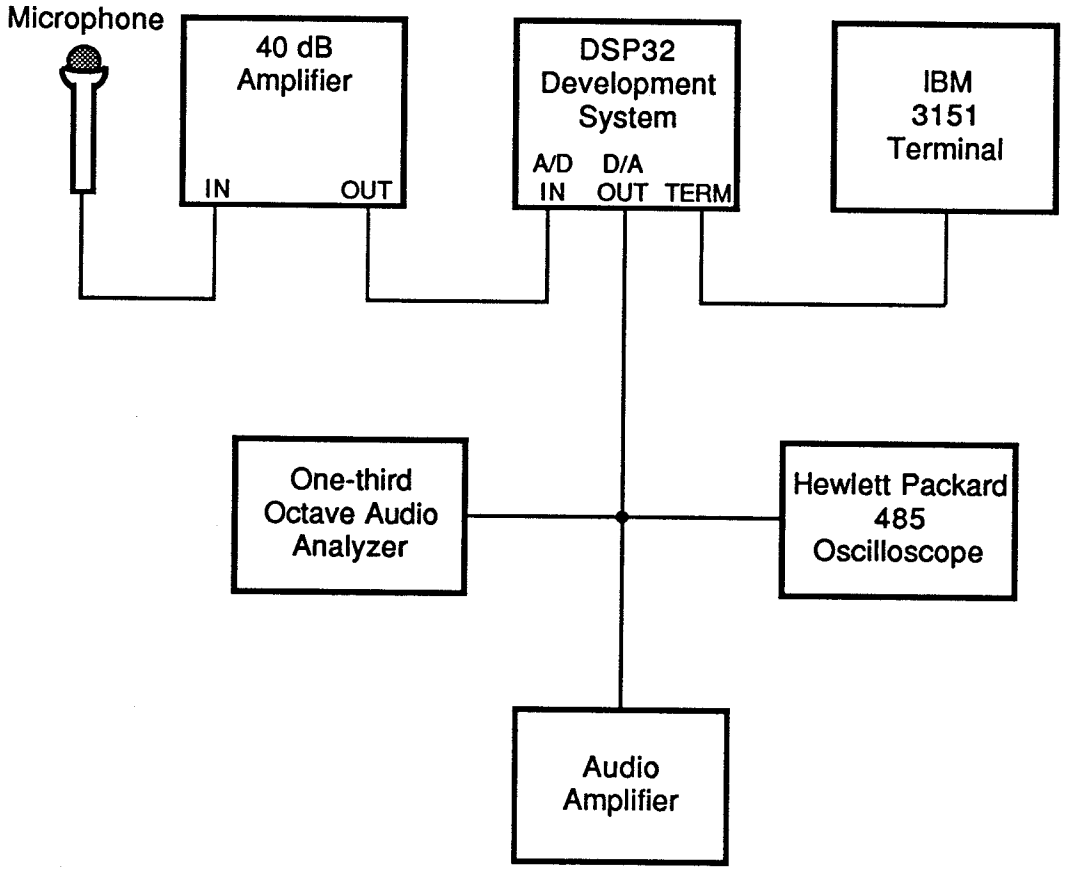

Figure 8-1. Diagram of the DSP32 real-time speech processing demonstration.

CHAPTER 9
COMPARISONS AND CONCLUDING REMARKS

9.1 COMPARING THE DSP32 TO OTHER SINGLE-CHIP DSPs

Today, several vendors offer single-chip digital signal processors. These devices greatly reduce the overhead associated with computationally intensive digital signal processing routines. These chips are either stand-alone (as with the DSP32) or are offered as a co-processor in a larger system [AlF87]. The DSP32 can be used either way due to extensive interface hardware. The DSP32 can communicate with other devices by means of serial or parallel interfaces. The DSP32 has also been migrated into a multiprocessor configuration which specializes in image processing operations. This is known as the Pixel Machine [At87], [Tu87].

As single-chip digital signal processing has evolved, emphasis has shifted from fixed-point arithmetic implementations to floating-point implementations. The predecessor of the DSP32, the DSP16, supported only 16-bit fixed-point arithmetic. As a result, any algorithm designed for the system had to be converted to fixed-point representation before it could be used. Often, as with the IIR filter, this conversion took a great deal of time to implement and reduced the accuracy of the algorithm.

The DSP32 is one of a very few DSP chips offering 32-bit floating-point capability. Others are the TMS320C30 from Texas Instruments and the μpd77230 from NEC. Since the TI TMS320C30 is not yet available, the DSP32 shares only the company of the NEC μpd77230 in floating point capability. The DSP32 accumulators have a resolution of 40 bits as opposed to the μpd77230's 32 bits. The TMS320C30, however, will also have 40-bit floating-point accumulator precision.

Most commercial DSP chips offer optimized arithmetic hardware to speed operations often found in digital signal processing applications, such as repeated sums and multiplications. Many DSP chips also utilize pipelined architectures and multiple address and data buses. The DSP32 uses a pipelined architecture with separate arithmetic and control processors. The

TMS320C30 uses a single processor with a DMA controller which supports concurrent I/O. Both the DSP32 and the TMS320C30 are clocked at 16 MHz, providing a cycle time of 60 ns.

The evolution of DSP chips is now utilizing integration in the 1 μm range. The TI TMS320C30 is leading the next generation with greatly expanded memory addressing capability (16 M 32-bit words as opposed to the DSP32's 14 K 32-bit words) and compatibility with high-level languages, such as C. The DSP32 is moving to higher levels of programmability with a C interface in the works, as well.

The availability of higher level languages to program digital signal processors will increase their portability and decrease development time. It is important to note, however, that compiled high level language interfaces may not provide the flexibility to optimize program execution on a specialized architecture [PoD82]. Table 9-1 shows a summary of DSP chips currently commercially available.

9.2 COMPARING DSP32 PERFORMANCE WITH MINICOMPUTERS

In order to generate a speed comparison, we compare the DSP32 with two high-performance minicomputers: a dual-VAX 11/780, which runs approximately 4 million instructions per second (MIPS), and a Gould NP1 minisupercomputer, which runs approximately 10 MIPS. Both computers operate under Berkeley UNIX 4.3 and are connected to the Purdue University Engineering Computer Network. The optimized DSP32 architecture is capable of 8 million floating point operations per second (MFLOPS)

The algorithm chosen for the comparison was the well-known Fast Fourier Transform (FFT). The FFT has been optimized in C and in the DSP32 assembly language. The performance of the DSP32 would be expected to be better than the higher level machines since it is, by definition, a single-user machine. Therefore, the comparison was based on CPU time, and actual clock time for the program run.

The input data for the FFT algorithm was a 1024-point cosine signal. Both the DSP32 FFT routine and the C Language FFT routine work with complex input and output. Since the DSP32 routine returns only complex numbers, the C Language FFT routine was limited to returning complex

numbers. The magnitude calculation section of the program was turned off.

Results of the comparison follow:

DSP32 FFT routine -- CPU time -- 19 ms
DSP32 FFT routine -- clock time -- under 1 second

Dual VAX 11/780 FFT -- CPU time -- 2.5 ms
Dual VAZ 11/780 FFT -- clock time -- 32 seconds

Gould NP1 FFT -- CPU time -- 1.9 ms
Gould NP1 FFT -- clock time -- 14 seconds

Results would indicate that the DSP32 did not perform competitively with the VAX or the NP1. One possible reason for this difference is that the DSP32 provides on-board timing in hardware, as opposed to the UNIX-based machines' software-based timing. As a result, the DSP32 can generate more accurate timing results. Clock time for the minicomputers is dependent on user load, so the DSP32's superior clock time performance was no surprise.

9.3 CONCLUSIONS

The DSP32 Digital Signal Processing Development System provides a powerful development environment for digital signal processing applications. The software interface to the machine provides access to many useful system functions. Very few problems were ever found with the development system, which is rare for a such a new product. The history of the DSP32 in AT&T telephone switches obviously played an important role in speeding the development of the DSP32 into a scientific product.

One aspect of the machine which hindered the development process was the lack of high-speed I/O with the turnkey system. The serial port which provides access to the machine makes the loading of large programs and static data a time consuming process. The lack of a large amount of on-board, off-chip memory is an indication that the DSP32 was designed with real-time engineer in mind. Programs which ran the real-time speech processing demonstration took little time to run and, since input came from the CODEC in real-time, we experienced no bottleneck with the serial port. However, when image processing applications and data were loaded into the machine, the shortcomings of the DSP32 became evident. The image

processing routines were much larger in size than the filter routines, and the data was larger, as well. Typical time to load a two-dimensional application into the development system was in excess of five minutes (the routines typically ran in under one second).

In the future, the DSP32 will have a better development environment under a C-like programming language. This will speed the development of programs, like our two-dimensional programs, which rely heavily on data structures. The DSP32 will also see use in image processing, where arrays of DSP32s will be combined to handle image data at high rates of throughput (in excess of 700 MIPS) [At87]. A welcome enhancement would be a larger on-board off-chip addressable memory with the DSP32. Some applications require large amounts of static data space for optimal operation. Our two-dimensional applications, for example, were limited to 4096-element pictures.

Table 9-1. Feature comparison of the leading single-chip digital signal processors [AlF87].

Company	Product[2]	Package,[3] number of pins	Power dissipation, W	Instruction cycle time, ns	Internal memory size[4]		External memory size, bits	Word length,[5] bits	Multiply-accumulate parameters	
					RAM, bits	ROM, bits			Accumulation precision, bits	Time per operation,[6] ns
Analog Devices Inc., Norwood, Mass	ADSP2100	PGA, 100	0.6	125	16 × 24 C	None	16K × 24 P 32K × 16 D	16	40	125
AT&T Bell Laboratories, Murray Hill, N.J.	WEDSP16	PLCC, 84; CLCC, 84	0.4	60	512 × 16 D,P	2K × 16 D,P	64K × 16 P	16	36	60
	WEDSP32	DIP, 40; PGA, 100	2.6	244	1K × 32 D,P	512 × 32 P	14K × 32 D,P	8 E, 24 M	8 E, 32 M	244
Fujitsu Ltd., Tokyo, Japan	MB8764	PGA, 88; PLCC, 84	0.3	100	256 × 16 D	1K × 24 P	1K × 16 P,D	16	26	100
Inmos Corp., Colorado Springs, Colo.	IMSA100	PGA, 84	<1.5	NA	NA	NA	NA	16	36	400
Motorola Inc., Phoenix, Ariz.	DSP56000	PGA, 88; PLCC, 88	NA	98	512 × 24 D	512 × 24 D, 2K × 24 P	64K × 24 P 128K × 24 D	24	56	98
	DSP56200	DIP, 28	NA	NA	256 × 24 D 256 × 16 D	NA	NA	16 D, 24 C	40	98
NEC Electronics Inc., Mountain View, Calif.	μpd77X20X	DIP, 28	0.6	250	128 × 16 D	512 × 23 P, 510 × 23 D (EPROM)	None	16	16	500
	μpd77230	PGA, 68	1.7	150	1K × 32 D	1K × 32 D, 2K × 32 P	4K × 32 P 4K × 32 D	8 E, 24 M	8 E, 47 M	150
Oki Electric Industry Co. Ltd., Tokyo, Japan	MSM6992	PGA, 132	0.4	100	1K × 32 P	1K × 32 P	64K × 32 D 64K × 32 P	6 E, 16 M	6 E, 16 M	100
Signetics Corp., Sunnyvale, Calif.	PCB5010	PLCC, 68	0.5	125	256 × 16 D 32 × 40 P	512 × 16 D 992 × 40 P	64K × 16 D,P	16	40	125
	PCB5011	PGA, 144	0.5	125	256 × 16 D	None	512 × 16 D 64K × 16 D,P	16	40	125
Thomson-CSF Components Corp., Canoga Park, Calif.	TS68930	DIP, 48	1.5	160	256 × 16 D	512 × 16 D 1K × 32 P	4K × 16 D	16, 32	32	160 R, 360 C
	TS68931	PGA, 84; CLCC, 84	1.8	160	256 × 16 D	512 × 16 D	4K × 16 D, 64K × 32 P	16, 32	32	160 R, 360 C
Texas Instruments Inc., Houston, Texas	TMS320C1X	DIP, 40; PLCC, 44	0.2	160; 200	256 × 16 D	4K × 16 P (EPROM)	4K × 16 P	16	32	320, 400
	TMS32020	PGA, 68	1.2	200	288 × 16 D 256 × 16 D,P	None	64K × 16 D 64K × 16 P	16	32	200
	TMS320C25	PLCC, 68; PGA, 68	0.5	100	288 × 16 D 256 × 16 D,P	4K × 16 P	64K × 16 D 64K × 16 P	16	32	100
	TMS320C30	QFP, 100; PGA, 176	NA	60	2K × 32 D,P 64 × 32 C	4K × 32 D,P	16M × 32 D,P	8 E, 24 M	8 E, 32 M	60
Zoran Corp., Santa Clara, Calif.	ZR33881	LCC, 68	1.0	NA	NA	NA	NA	8	25	50
	ZR34161	DIP, 48	0.5	100	256 × 17 D	256 × 17 D	64K × 16 D,P	16	25	100

1. Based on an extensive table available from DSP Associates, 18 Peregrine Rd., Newton, Mass. 02159; except for the Motorola DSP56000 and the Texas Instruments TMS320C30, all ICs are currently available; NA—data not available or not applicable
2. Except for the digital filters IMSA100, DSP56200, and ZR33881 and the vector signal processor ZR34161, all the ICs are general-purpose digital signal processors; most devices handle integer-type data; the WEDSP32, μpd77230, and MSM 6992 are floating point devices; the TMS320C30 is designed to operate in either an integer or a floating-point mode; and the ZR34161 is a block floating-point device; μpd77X20X represents three devices: the μpd7720A, 77C20, and 77P20; TMS 320C1X represents six devices: the TMS320C10, C15, E15, C17, E17, and TMS32011
3. Legend: PGA—pin-grid array; PLCC—plastic, leadless chip carrier; CLCC—ceramic, leadless chip carrier; DIP—dual in-line package; QFP—quad flat package; LCC—leadless chip carrier
4. Legend: D—data; P—program; C—cache, EPROM—electrically programmable read-only memory; K—1024 bits; M—1 048 576 bits
5. Legend: E—exponent; M—mantissa; C—coefficient
6. Legend: R—real numbers; C—complex numbers; for DSP56200, the time is per filter tap

LIST OF REFERENCES

LIST OF REFERENCES

[Al86] H. Alrutz, "Implementation of a multi-pulse coder on a single chip floating-point signal processor," *Proceedings of the International Conference of Acoustics, Speech and Signal Processing,* pp. 2367-2370, Tokyo, April 1986.

[AlF87] A. Aliphas and J. A. Feldman, "The versatility of digital signal processing chips," *IEEE Spectrum,* Vol. 24, No. 6, pp. 40-45, June 1987.

[ArG86] G. R. Arce, N. C. Gallagher, and T. A. Nodes, "Median filters: theory for one- and two-dimensional filters," in *Advances in Computer Vision and Image Processing,* vol. 2, T. S. Huang, Ed. Greenwich, CT: JAI Press, 1986.

[At86] T7520 high precision PCM codec with filters, AT&T Preliminary Data Sheet, May 1986.

[At87] "AT&T Pixel Machines, PXM 900 Series," Product Specification, AT&T New Jersey, July 1987.

[BoG86] J. R. Boddie, R. N. Gadenz, R. N. Kershaw, W. P. Hays and J. Tow, "The DSP32 digital signal processor and its application development tools," *AT&T Technical Journal,* Vol. 65, Issue 5, pp. 89-104, September/October 1986.

[BoH86] J. R. Boddie, W. P. Hays, and J. Tow, "The architecture, instruction set and development support for the western electric DSP32 digital signal processor," *Proceedings of the International Conference of Acoustics, Speech and Signal Processing,* pp. 421-424, Tokyo, April 1986.

[Da84] J. W. Daugherty, "Using the WE DSP32 digital signal processor in speech processing applications," AT&T Technical Memorandum, September 1984.

[Ha86] R. M. Haralick, "Glossary of Computer Vision," *Machine Vision International,* Ann Arbor, MI, May 1986.

[HaS86] R. M. Haralick, S. R. Sternberg, and X. Zhuang, "Image analysis using mathematical morphology," *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol PAMI-9, No. 4, pp. 532-550, July 1987

[HaT86] J. Hartung and J. Tow, *AT&T DSP32 Library Routines Manual,* Release v0.0, September 1986.

[HoC86] T. D. Hopmann, R. J. Canniff, M. A. Derrenberger, and P. A. Stiling, "A System Design for Real Time Signal Processing," *Proceedings of the International Conference of Acoustics, Speech and Signal Processing,* pp. 1341-1344, Tokyo, April 1986.

[OpS75] A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing,* Prentice Hall, 1975.

[PoD82] J. F. Poje and E. J. Delp, "A survey of computer architecture used in image processing," University of Michigan, Center for Robotics and Integrated Manufacturing Technical Report, RSD-TR-9-82, April, 1982.

[RoK82] A. Rosenfeld and A. C. Kak, *Digital Picture Processing,* Second Edition, Academic Press, 1982.

[Se82] J. Serra, *Image Analysis and Mathematical Morphology,* Academic Press, 1982.

[Sh79] K. S. Shanmugam, *Digital and Analog Communication Systems,* John Wiley & Sons, 1979.

[StD84] W.D. Stanley, G. R. Dougherty and R. Dougherty, *Digital Signal Processing,* Second Edition, Reston, 1984.

[To86] J. Tow, "Interfacing the DSP32 and AT&T PCM codecs, AT&T Technology Systems Technical Memorandum, TM86-54332-860212-01IM, December 1986.

[Tu87] D. Tunick, "Powerful display system revs up image and graphics processing," *Electronics Design,* pp. 58-62, July 23, 1987.

[We86] *Western Electric DSP32-SL Support Software Library User Manual,* AT&T Document Management Organization, July 1986.

APPENDICES

APPENDIX A: THE DSP32 INSTRUCTION SET

Instructions	Format	CAU Flags Affected			
Control Group:					
if (CA COND) goto {rH, N, rH+N, rH−N}	0	—	—	—	—
if (rM− −> =0) goto {rH, N, rH+N, rH−N}	3	—	—	—	—
if (DA COND) goto {rH, N, rH+N, rH−N}	0	—	—	—	—
if (IO COND) goto {rH, N, rH+N, rH−N}	1	—	—	—	—
call {rH, N, rH+N, rH−N} (rM)	4	—	—	—	—
return (rM)	0	—	—	—	—
goto {rH, N, rH+N, rH−N}	0	—	—	—	—
[L]*nop	0	—	—	—	—
Arithmetic Logical Group:					
rD = rH+N	5	n	z	v	c
rD = rD+rS	6a	n	z	v	c
rD = rD−{N, rS}	6a	n	z	v	c
rD−{N, rS}	6a,b	n	z	v	c
rD = {N, rS}−rD	6a,b	n	z	v	c
rD = rD&{N, rS}	6a,b	n	z	0	0
rD&{N, rS}	6a,b	n	z	0	0
rD = rD|{N, rS}	6a,b	n	z	0	0
rD = rD^{N, rS}	6a,b	n	z	0	0
rD = rS/2	6a	n	z	0	c
rD = rS> >1	6a	0	z	0	c
rD = −rS	6a	n	z	v	c
rD = rS*2	6a	n	z	v	c
Data Move Group:					
rD = N	5	n	z	0	0
{ioc, dauc} = VALUE	5	—	—	—	—
{MEM, *N, obuf} = {rSh, rSl}	7a,b,c	—	—	—	—
{MEM, *N, obuf, pdr, pir} = rS	7a,b,c	—	—	—	—
rD = {MEM, *N, ibuf, pdr}	7a,b,c	n	z	0	0
{rDh, rDl} = {MEM, *N, ibuf}	7a,b,c	n	z	0	0

Instructions	Format	DAU Flags Affected			
Multiply/Accumulate:					
[Z =] aN = [−]aM {+,−} Y*X	3	N	Z	V	U
aN = [−]aM {+,−} (Z=Y)*X	2	N	Z	V	U
[Z =] aN = [−]Y {+,−} aM*X	1	N	Z	V	U
[Z =] aN = [−]Y*X	3	N	Z	V	U
aN = [−](Z=Y)*X	2	N	Z	V	U
[Z =] aN = [−]Y {+,−} X	1	N	Z	V	U
[Z =] aN = [−]Y	1	N	Z	V	U
Special Functions:					
[Z=] aN = ic(Y)	4	N	Z	0	0
[Z=] aN = oc(Y)	4	—	—	—	—
[Z=] aN = float(Y)	4	N	Z	0	0
[Z=] aN = int(Y)	4	—	—	—	—
[Z=] aN = round(Y)	4	N	Z	V	U
[Z=] aN = ifalt(Y)	4	—	—	—	—

APPENDIX B: MANUAL PAGES

NAME

dsp3as – DSP3 assembler

SYNOPSIS

dsp3as [–V] [–P] [–C] [–D*n*=*v*] [–D*n*] [–U*n*] [–I*dir*] [–l] *filename* [...*file*...]

DESCRIPTION

Dsp3as is the DSP3 assembler that runs on the UNIX System. It accepts two types of arguments. The arguments beginning with an initial minus sign, as described below, must appear first in the command string.

Arguments whose names end with **.s** or **.i** are taken to be DSP3 assembly source programs. They are assembled, and each object program is left on the file whose name is that of the source with **.o** substituted for the **.s** or **.i**.

Flag arguments are interpreted by **dsp3as** as follows.

–V	Print the version number of the assembler when it is invoked.
–P	Run only the macro preprocessor on the named DSP3 programs, and leave the result on corresponding files suffixed **.i**.
–C	Include all comments in the listing file.
–D*n*=*v*	Define *n*, an identifier, to the preprocessor, as if by **#define**, and give it value *v*.
–D*n*	Define *n* to the preprocessor as 1.
–U*n*	Remove any initial definition of *n*. *n* is undefined.
–I*dir*	Change the algorithm for searching for **#include** files whose names do not begin with / to look in *dir* before looking in the directories on the standard list. Thus, **#include** files whose names are enclosed in "" will be searched for first in the directory of the *filename* argument, then in directories named in –I options, and last in directories on a standard list. For **#include** files whose names are enclosed in <>, the directory of the *filename* argument is not searched.
–l	(lower-case L). Produce an assembly listing.

EXAMPLES

dsp3as –D*value1*=036 –l *filename*.s

The above line causes file *filename.s* to be preprocessed and assembled into file *filename.o*. Whenever the identifier **value1** occurs, it is replaced with an octal 36. A listing is produced.

The command line

dsp3as −Dvax *file1*.**s** *file2*.**i**

causes identifier *vax* to have value 1 when the files are assembled. The preprocessor will be called for *file1*.**s**. The files will be assembled and output into *file***1.o** and *file***2.o** respectively.

FILES

file.**s**	input source file
file.**i**	preprocessed source file
file.**o**	assembly output - object file

NAME

dsp3ckfile – check the contents of a object file

SYNOPSIS

dsp3ckfile [–d] [–l] [–r] [–s] [–N *nn*] [–V] files

DESCRIPTION

The **dsp3ckfile** command is a common object file auditor. It cross-checks the contents of an object file to detect all possible errors.

The **dsp3ckfile** command only checks the non-executable portions of the object file. It performs no checking on internal raw text and data.

The following options are interpreted by dsp3ckfile.

–d	Set a user debug flag, used to print the contents of file header, section headers and additional information.
–l	Suppress the line number check.
–r	Suppress the relocation check.
–s	Suppress the symbol table check.
–N *nn*	Set the maximum number of messages to be printed to nn. The argument "nn" is a numeric constant. If the "–N" flag is not used, the maximum number of messages will be 50,000.
–V	Version of dsp3ckfile command executing.

FILES

SDPage? temporary file

DIAGNOSTICS

Diagnostics are intended to be self-explanatory. Some knowledge of the common object file format is needed.

NAME

dsp3conv – DSP3 SGS object file converter

SYNOPSIS

dsp3conv [–s] –t *target* [–V] *files*

DESCRIPTION

The **dsp3conv** command converts DSP3 object files from their current format to the format of the *target* machine. The converted *file* is written to *file*.v.

Command line options are:

–s Causes dsp3conv to function exactly as 3bswab, i.e., to "preswab" all characters in the object file. This is useful only for 3B20 object files which are to be "swab-dumped" from a DEC machine to a 3B20.

–t *target* Indicates the machine (target) to which the object file is being shipped. This may be another host or a target machine. Legal values for *target* are: pdp, vax, ibm, i80, x86, b16, n3b and m80.

–V Version of **dsp3conv** command executing.

DSP3conv can be used to convert all object files in common object file format, not only DSP3 object files. It can be used on either the source ("sending") or target ("receiving") machine.

DSP3conv is meant to ease the problems created by a multihost cross-compilation development environment. DSP3conv is best used within a procedure for shipping object files from one machine to another.

EXAMPLE

```
* ship object files from pdp11 to ibm
Secho *.out | dsp3conv –t ibm –$OFC/foo.o
Suucp *.v my370! /rje/
```

DIAGNOSTICS

All intended to be self-explanatory. Fatal diagnostics on the command lines cause termination. Fatal diagnostics on an input file cause the program to continue to the next input file.

BUGS

Special applications, such as *DMERT*, must compile dsp3conv differently if it is to convert special object files, e.g., products of ldp, correctly. Archives created on another architecture and converted on the target machine may have members with incorrect permissions (this has no effect so long as they remain members of the archive).

NAME

dsp3dump – dump selected parts of an object file

SYNOPSIS

dsp3dump [**–a**] [**–f**] [**–o**] [**–h**] [**–s**] [**–r**] [**–l**] [**–t**] [**–z** *name*] [**–V**] *files*

DESCRIPTION

The **dsp3dump** command dumps selected parts of each of its object *file* arguments.

This command will accept both object files and archives of object files. It processes each file argument according to one or more of the following options:

–a	Dump the archive header of each member of each archive file argument.
–f	Dump each file header.
–o	Dump each optional header.
–h	Dump section headers.
–s	Dump section contents.
–r	Dump relocation information.
–l	Dump line number information.
–t	Dump symbol table entries.
–z *name*	Dump line number entries for the named function.
–V	Version of dsp3dump command executing.

The following modifiers are used in conjunction with the options listed above to modify their capabilities.

–d *number*	Dump the section number or range of sections starting at *number* and ending either at the last section number or *number* specified by **+d**.
+d *number*	Dump sections in the range either beginning with first section or beginning with section specified by **–d**.
–n *name*	Dump information pertaining only to the named entity. This modifier applies to **–h**, **–s**, **–r**, **–l**, and **-t**.
–t *index*	Dump only the indexed symbol table entry. The **–t** used in conjunction with **+t**, specifies a range of symbol table entries.
+t *index*	Dump the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the **–t** option.
–v	Dump information in symbolic representation rather than numeric (e.g., C_STATIC instead of 0X02). This modifier can be used with all the above options except **–s** and **–o** options of **dsp3dump**.

−z *name,number* Dump line number entry or range of line numbers starting at *number* for the named function.

+z *number* Dump line numbers starting at either function *name* or *number* specified by **−z**, up to *number* specified by **+z**.

Blanks separating an option and its modifier are optional. The comma separating the name from the number modifying the **−z** option may be replaced by a blank.

The **dsp3dump** command attempts to format the information it dumps in a meaningful way, printing certain information in character, hex, octal or decimal representation as appropriate.

NAME

dsp3ld – link editor for DSP3 object files

SYNOPSIS

dsp3ld [**–a**] [**–e** *epsym*] [**–f** *fill*] [**–i**] [**–lx**] [**–m**] [**–r**] [**–s**] [**–o** *outfile*] [**–u** *symname*] [**–L** *dir*] [**–V**] [**–X**] *filenames*

DESCRIPTION

The **dsp3ld** command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given and **dsp3ld** combines them, producing an object module that can either be executed or used as input for a subsequent **dsp3ld** run. The output of **dsp3ld** is left in dsp3a.out. This file is executable, if no errors occurred during the load. If any input file, *filename*, is not an object file, **dsp3ld** assumes it is either an ASCII file containing link editor directives or an archive library.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. If a routine from a library references another routine in the library, the referenced routine must appear after the referencing routine in the library. Thus, the order of programs within libraries is important.

The following options are recognized by **dsp3ld**.

–a Produce an absolute file; give warnings for undefined references. Relocation information is stripped from the output object file unless the **–r** option is given. The **–r** option is needed only when an absolute file should retain its relocation information (not the normal case). If neither **–a** nor **–r** is given, **–a** is assumed.

–e *epsym* Set the default entry point address for the output file to be that of the symbol *epsym*. This option forces the **–X** option to be set.

–f *fill* This option sets the default fill pattern for "holes" within an output section as well as initialized bss sections. The argument *fill* is a two-byte constant.

–i This option specifies that separate "I" and "D" space are to be generated. This allows 64K of instructions and 64K of data.

–l *x* This option specifies a library named *x*. It stands for the standard library for DSP3 assembly language programs, **liba.a**. It stands for **lib***x***.a** where *x* is up to seven characters. A library is searched when its name is encountered, so the placement of a **–l** is significant. By default, libraries are located in LIBDIR.

–m This option causes a map or listing of the input/output sections to be produced on the standard output.

–o *outfile* This option produces an output object file by the name *outfile*. The name of the default object file is **dsp3a.out**.

-r This option causes relocation entries to be retained in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent **dsp3ld** run. Unless **-a** is also given, the link editor will not complain about unresolved references.

-s This option causes line number entries and symbol table information to be stripped from the output object file.

-u *symname* Takes the argument *symname* as a symbol and enters it as undefined in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

-L *dir* Change the algorithm of searching for **lib***x***.a** to look in *dir* before looking in LIBDIR.

-V Output a message giving information about the version of **dsp3ld** being used.

-X Generate a standard *UNIX* System file header within the "optional header" field in the output file.

FILES

LIBDIR/lib_.a	libraries
dsp3a.out	output file

NAME

dsp3make – Assemble/load Simple DSP3 Programs

SYNOPSIS

dsp3make [–s] [–{m | M}{0123}] *filename*

DESCRIPTION

Given that the user has created a DSP3 assembler file called *filename*.s, this command creates the file which can be run on dsp3sim, or on a DSP32. It can only be used with a program consisting of one file, but the ifiles which it uses (see the **FILES** section) can be used as models in more advanced applications. **dsp3make** causes other commands to be executed which invoke dsp3as and dsp3ld; these commands are printed as it goes along.

FEATURES WHICH IT ADDS TO DSP3AS/DSP3LD

The –s option should be used when assembling a program to run on the DSP23-DS DSP Development System. It links in special code to handle breakpoints: although this is not generally detrimental when in simulation mode, it does use memory locations 0xFE00–0xFFFF. These memory locations are reserved by **dsp3make** regardless of whether the –s option is invoked.

The following options are used to describe the memory arrangements to the link editor. For more information on memory, consult the DSP32 Digital Signal Processor Information Manual. Chapter 4, in this manual, also provides more detail on memory. If no memory mode is specified on the command line, then **.bank0** and **.bank1** are the only **.rsects** provided, and specify memory ranges 0x0000–0xDFFF and 0xE000–0xFFFF respectively. When **–m{0123}** is used, then **.rsects** are provided that are appropriate to specific memory modes, providing more precision in memory placement. Each of the 4 modes provides **.bank0** consisting of locations 0–DFFF(hex) or 800–DFFF(hex), depending on whether ROM is from 0–7FF. Additionally, they provide the following **.rsects** mapped into specific segments of bank 1:

–m0: .rom is 0–7FF; **.hi_ram** is F000–FDFF

–m1: .rom is 0–7FF; **.hi_ram** is F800–FDFF

–m2: .rom is E000–E7FF; **.hi_ram** is F000–FDFF

–m3: .rom is E000–E7FF; **.hi_ram** is F800–FDFF

–M{0–3} provides a further distinction between the on-chip RAM section in bank 0 and external memory. This is particularly useful if no external memory will be available, in which case memory modes 1 or 3 will probably be preferred, and one should explicitly select whether memory goes into **.lo_ram, .rom, or .hi_ram**. (–M0 and –M2 are identical to their lower-case counterparts and are provided for convenience).

–M1: **.rom** is 0–7FF; **.lo_ram** is 800–1000; **.hi_ram** is F800–FDFF

–M3: **.lo ram** is 0–7FF; **.rom** is E000–E7FF; **.hi_ram** is F800–FDFF

In each case, locations FE00—FFFF are reserved (as .rsect ".sbds") for code used by the **SBDS** for breakpoint processing.

EXAMPLE

The following program, when assembled and loaded using **dsp3make** −M3 will put the machine instructions and the block of constants in rom (bank 1), and the block of variable data in low ram (bank 0). The program as it is can be simulated, but to run it on a DSP32, one would have to order chips with special ROM masks. Note however that if **.hi_ram** is substituted for **.rom**, one can test the program with the same number of wait states as in the final product.

```
/*  biquad.s - Low Pass Filter. */
.global       loop, wait, SoS, coef, s, main
.rsect ".rom"
main:
      ioc = 0xff0
      dauc = 0
      r16 = 8

loop:  r1 = s
      r4 = coef
wait:  if(ibe) goto wait
      r3 = 0
      a0 = ibuf
SoS:          a0 = a0 − *r1++ * *r4++
      *r1 =   a0 = a0 − *r1−− * *r4++
            nop
            a1 = (*r1 = *r1++) * *r4++
            nop
            a1 = a1 + *r1++ * *r4++
      if(r3−− >= 0) goto SoS
      a0 = a1 + a0 * *r4++
      goto loop
      obuf = a0 = a0

.rsect ".lo_ram"
        goto main
        nop
s:    4*float 0.0

.rsect ".rom"
coef:  2 * float −0.4, 0.2, 0.4, 0.2, 0.2
```

dsp3make −M3 biquad

(Make runnable file biquad from biquad.s −
dsp3make will print two lines)

```
dsp3sim biquad
coef
coef = 0xe048
coef[0:40].f
coef [ 0 : 40 ] =
0x0000:  −0.4 0.2 0.4 0.2
0x0010:  0.2 −0.4 0.2 0.4
0x0020:  0.2 0.2 0
coef = "new_coef"
```

The dsp3sim commands verify that **coef** has address 0xe048, in bank 1, and contains the expected data. The last line reads in new coefficients from a file.

FILES

The following may be a good source of models for doing programs involving several files, which **dsp3make** cannot do. This requires some understanding of the *UNIX* System **make** utility, and of **dsp3ld**, which resembles other *UNIX* System loaders.

```
$DSP3BIN/dsp3make
$DSP3LIB/dsp3.makefile
$DSP3LIB/dsp3.ifile
$DSP3LIB/m[0−3]_ifile
$DSP3LIB/M[0−3]_ifile
```

NAME

dsp3nm – print name list of DSP3 object file

SYNOPSIS

dsp3nm [–{ o d x}] [–v] [–n] [–e] [–f] [–u] [–V] *filename . . .*

DESCRIPTION

The **dsp3nm** command displays the symbol table of each DSP3 object file *filename*. *Filename* may be a relocatable or absolute DSP3 object file or it may be an archive of such DSP3 object files. For each symbol, the following information will be printed:

Name The name of the symbol.

Value Its value expressed as an offset or an an address depending on its storage class.

Class Its storage class.

Type Its type and derived type. If the symbol is an instance of a structure or of a union then the structure or union tag will be given following the type (e.g., struct-tag). If the symbol is an array, then the array dimensions will be given following the type (e.g., **char**[*n*][*m*]).

Size Its size in bytes, if available.

Line The source line number at which it is defined, if available.

Section For storage classes static and external, the object file section containing the symbol.

The output of **dsp3nm** may be controlled using the following flags:

–o A symbol's value and size will be printed in octal instead of hexadecimal.

–d A symbol's value and size will be printed in decimal instead of hexadecimal.

–v External symbols will be sorted by value before they are printed.

–n External symbols will be sorted by name before they are printed.

–e Only static and external symbols are printed.

–f 'Fancy' output is produced; that is, the symbol table information is post-processed to reflect the block structure of the source code.

–u Only undefined symbols are printed.

–V Version of dsp3nm command executing.

Flags may be used in any order, either singly or in combination, and may appear anywhere in the command line. Therefore, both **dsp3nm name –e –v** and **dsp3nm –ve name** print the static and external symbols in *name*, with external symbols sorted by value.

FILES

/usr/tmp/nm??????

SEE ALSO

dsp3as(1), dsp3ld(1).

DIAGNOSTICS

'dsp3nm: name: cannot open'
if *name* cannot be read.

'dsp3nm: name: bad magic'
if *name* is not a DSP3 object file.

'dsp3nm: name: no symbols'
if the symbols have been stripped from *name*.

NAME

dsp3sim – DSP3 simulator

SYNOPSIS

dsp3sim [–Vel] [–m*n*] ***filename***

DESCRIPTION

dsp3sim, the DSP3 simulator/debugger, is a program development tool which runs on the *UNIX* system. It accepts a standard DSP3 load file in the Common Object File Format (COFF), along with simulator execution commands. The COFF is created by the DSP3AS (assembler) or the DSP3LD (loader).

–V Prints the version number of the simulator when invoked. The optional **–m** argument, if supplied, must appear before the file name in the command string.

–e Says to read the file ***filename*.ex** as a list of commands. This can serve to set up commonly used functions and breakpoint settings.

–l Says to enable keep a record of user commands in a file **log.cmds**, or, if input is from a file, to echo the commands from the file as they are executed.

–m Specifies the memory mode. The value of *n* can be zero or one. In memory mode 0, the on-chip ROM is located from address 0 through 0x7ff. In memory mode 1, the on-chip ROM is located from address 0xe000 through 0xe7ff. If no mode is specified, then rom[x] coincides with mem[x] and all memory is writable.

dsp3sim execution commands allow manipulation of simulated memory and registers. In addition, breakpoints may be set to detect conditions which may occur during simulation of the program.

dsp3sim closely simulates the DSP3 architecture and functionality. The simulator provides for instruction latency effects, serial data transfer delays and program preformance information; approximate execution time and number of wait states.

After the simulated DSP3 memory has been loaded from the object file, the simulator execution commands are read from the standard input. All nonerror output produced by the simulator is directed to the standard output. All error messages are directed to the standard error output.

Multiple commands may appear on a line but must be separated by a semicolon. Commands may be continued over multiple lines by breaking the line at any white space and inserting a backslash (continuation character) before the newline.

NAME

dsp3size – print section sizes of DSP3 object files

SYNOPSIS

dsp3size [**–o**] [**–d**] [**–V**] *files*

DESCRIPTION

The **dsp3size** command produces section size information for each section in the DSP3 object files. For each section in the object file, the name of the section is printed followed by its size in bytes, its physical address, and its virtual address.

Numbers will be printed in hexadecimal unless either the **–o** or the **–d** option is used, in which case they will be printed in octal or in decimal, respectively.

The **–V** flag will supply the version information on the **dsp3size** command.

SEE ALSO

dsp3as(1), dsp3ld(1).

DIAGNOSTICS

'dsp3size: name: cannot open'
if *name* cannot be read.

'dsp3size: name: bad magic'
if *name* is not a DSP3 object file.

NAME

dsp3strip – strip symbol and line number information from DSP3 object file

SYNOPSIS

dsp3strip [–l] [–x] [–r] [–V] *filenames*

DESCRIPTION

The **dsp3strip** command strips the symbol table and line number information from DSP3 object files, including archives. Once this has been done, no symbolic debugging access will be available for that file; therefore, this command is normally run only on production modules that have been debugged and tested.

The amount of information stripped from the symbol table can be controlled by using the following options:

–l Strip line number information only; do not strip any symbol table information.

–x Do not strip static or external symbol information.

–r Reset the relocation indices into the symbol table.

–V Version of dsp3strip command executing.

If there are any relocation entries in the object file and any symbol table information is to be stripped, **dsp3strip** will complain and terminate without stripping *filename* unless the –r flag is used.

The purpose of this command is to reduce the file storage overhead taken by the object file.

FILES

/usr/tmp/dsp3str??????

SEE ALSO

dsp3as(1), dsp3ld(1).

DIAGNOSTICS

'dsp3strip: name: cannot open'
if *name* cannot be read.

'dsp3strip: name: bad magic'
if *name* is not a DSP3 object file.

'dsp3strip: name: relocation entries present; cannot strip'
if *name* contains relocation entries, the -r
flag not used, and any symbol table information was
to be stripped.

APPENDIX C: DSP32 SIMULATOR COMMAND SUMMARY

Keywords and Predefined Variables Accessed by the HELP command.

FALSE	OUTFMT	PDRIS	PDROS	PIROS
SOS	TRUE	a	a0-a3	aflt
bp	bpset	break	cfp	clock
cmplt	cont	dauc	decode	dis
emr	enb	es	esr	exec
fltmu	fp	ibuf	ioc	ld
mlp	muflt	next	nin	nopdr
nout	npir	nwait	obuf	pc
pdr	pick	pild	pin	pir
pold	pout	psw	psync	quit
r1—r21	r1h—r21h	r1l—r21l	reset	run
save	stat	step	t	trace
var	waits	wh	when	whenever

Type **'help "all"'** to see all the help entries for the above or for descriptions of bit-mapped variables, type **'help var'** or **'help var value'** where var is one of pcr, esr, psw, ioc, mm, or dauc and value is a possible value of that variable.

Results of the **HELP "ALL"** Command

FALSE — constant equaling 0

OUTFMT — Output format (for expression display)
OUTFMT = {"oct"|"hex"|"dec"}

PDRIS — Parallel Data Register Input STACK

PDROS — Parallel Data Register Output STACK

PIROS — Parallel Interrupt Register Output STACK

SIS — Serial Input STACK

SOS — Serial Output STACK

TRUE — constant equaling 1

a — a0 through a3 (display all at once)

a0—a3 — 32-bit DAU accumulators

aflt — Convert a-law to (host) floating point format
E.g., aflt 0 = −688.0

alp — Print a-law in step-chord format
E.g., alp 0 = 0a−5:5

bp — 'clr bp' - clear all 'bpset' breakpoints;
'clr bp arg1,arg2,...' − clear specified ones;
'lst bp' or 'lst bp arg1,arg2,...' list all or list specified etc.

bpset	Set breakpoints. Format – 'bpset label' or 'bpset label{function}'
break	Halt DSP3 run; get back to command level.
cfp	Convert host floating point to DSP32 floating point. E.g., cfp 1.0 = 0x80
clock	Display rate of passive I/O clocks.
clr	See 'bp'; also 'clr wh' clears all when/whenever conditions; 'clr es' clears all 'event stepping' commands
cmplt	Complete the execution of instructions in progress.
cont	Resume DSP3 execution where left off after a halt
dauc	2-bit DAU control register
decode	Disassemble machine code. E.g., 'decode(*pc)' prints source that could generate the current instruction
dis	e.g., 'dis nins' – stop incrementing nins when inputs occur.
do	'do{cmd-list} while cond' – repeat cmd-list at least once and as long as cond != 0 (see while)
emr	16-bit error mask register
enb	Undoes the affect of 'dis'
es	'es expr {cmd-list}' – During DSP3 run, perform cmd-list after each instruction cycle that value of expr changes
esr	8-bit error source register
exec	Execute command file: 'exec "file"' OR 'dsp3sim –e'.
flta	Floating point format to a-law conversion. E.g., flta 1.0 = 0xab
fltmu	Floating point format to mu-law conversion. E.g., fltmu 1.0 = 0x7f
fp	Convert DSP3 floating point format to host format. fp 0x60000080 = 1.75 Also format for output stack SOS; E.g., 'SOS = "fp"'
ibuf	serial input buffer
ioc	serial IO control register
ld	'ld' – reload memory with last program loaded 'ld mem "programfile" – load memory with program 'ld SIS "file" – start using "file" for serial input stack
lst	See 'bp'; also 'lst wh' list 'when/whenever' conditions; 'lst es' list es conditions; 'lst SOS' show contents of SOS
mlp	Display mu-law number in step/chord format. E.g., mlp 0 = 0m–7:15

muflt	mu-law to floating point conversion. E.g., muflt 0 = −4014.4
next	'pdr = next' gets next item from PDRIS, puts it into pdr.
nin	number of serial data inputs
nopdr	number of parallel data outputs
nops	number of instructions fetched
nout	number of serial data outputs
npir	number of interrupt data outputs
nwait	number of wait states
obuf	serial output buffer
pc	16-bit program counter
pcr	8-bit parallel IO control register
pdr	16-bit parallel IO data register
pick	passive input (bit) clock
pild	passive input load clock
pin	alias for r20; input pointer for serial DMA
pir	16-bit parallel IO interrupt vector register
pock	passive output (bit) clock
pold	passive output load clock
pout	alias for 21; output pointer for serial DMA
psw	processor status bits (Read Only)
psync	passive sync signal clock
quit	return to *UNIX* System
r	r1 through r21 (display all at once)
r1−r21	16 bit general purpose registers
r1h−r21h	high byte of register
r1l−r21l	low byte of register
reset	'reset SIS|SOS|...' Roll back to beginning of a stack. 'reset' − simulate a chip reset
run	Simulate a reset and execute from loc 0.
runmode	'runmode cont' − breakpoints don't halt a 'run' unless the command list has 'break' in it. 'runmode break' (default) − any breakpoint cause automatic halt of 'run'
save	'save SOS "file.sos"' − causes "file.sos" to be used for serial output stack

stat	'stat' – displays 'status' information.
step	Execute 1 instruction; step N Execute N instructions. step N{cmd-list}: N instructions & do cmd-list after each one.
t	time counter in states
trace	Display the last 256 branches. trace N: display last N
until	'do {commands} until cond' – see 'do'
var	Create a user variable. e.g., var x; OR var y=7,z=1.1;
waits	Display last 256 wait states. waits N: display last N
wh	'lst wh' OR 'clr wh';
when	'when cond {cmd-list}' – during a 'run', perform cmd-list each time cond goes from 0 to nonzero
whenever	'whenever cond {cmd-list}' – during a 'run', perform cmd-list after each instruction cycle in which 'cond' is nonzero.
while	'while cond {cmd-list}' – repeat cmd-list while cond != 0 In contrast to 'do{..}while', 'while x {..}' will do nothing if x is 0.(see do)

APPENDIX D: SUMMARY OF AT&T-SUPPLIED DSP32 ROUTINES

SCIENTIFIC/MATH GROUP --

Program	Description
Zdiv.s	Calculates Q = N / D
Zcon1.s	Converts an array of numbers in the IEEE floating point format to the internal DSP32 format
Zcon2.s	Converts an array of numbers in the DSP32 floating point format to the IEEE floating point format
Zsin.s	Calculates the sin function of x for -pi/2 <= x <= pi/2
Zcos.s	Calculates the cosine function of x for -pi/2 <= x <= pi/2
Ztan.s	Calculates the tangent function of x for -pi/4 <= x <= pi/4
Zasin.s	Calculates the arcsine of x for |x| < 0.966
Zacos.s	Calculates the arccosine of x for |x| < 0.966
Zatan.s	Calculates the arctangent of x for |x| > 0
Zlog2.s	Calculates log(x) [base 2] for x > 0
Zloge.s	Calculates log(x) [base e] for x > 0
Zlog10.s	Calculates log(x) [base 10] for x > 0
Zalog2.s	Calculates the antilog(x) [base2] for 0.0 <= x <= 1.0
Zetox.s	Calculates e**x for x >= 0
Zxtoy.s	Calculates x**y for x > 0
Zsqrt.s	Calculates x**0.5 for x > 0

MATRIX GROUP --

Zmatmul.s	Multiplies two general matrices A and B and stores the result in C. A is m x n, B is n x p, C is m x p
Zmat2x2.s	Multiplies two 2 x 2 square matrices A and B and stores the result in C

Zmat3x3.s	Multiplies two 3 x 3 square matrices A and B and stores the result in C
Zmat4x4.s	Multiplies two 4 x 4 square matrices A and B and stores the result in C
Zmat5x5.s	Multiplies two 5 x 5 square matrices A and B and stores the result in C
Zmat4x1.s	Multiplies a 4 x 4 square matrix A and a 4 x 1 matrix B and stores the result in C
Zmat4x1F.s	Multiplies a 4 x 4 square matrix A and a 4 x 1 matrix B and stores the result in C

FILTER GROUP --

Zfir.s	Implements the finite impulse response (FIR) filter, y(n) = SUM [h(k) * x(n - k)] for k=0 to N-1
Zfir5.s	Implements the finite impulse response (FIR) filter, y(n) = [h(k) * x(n - k)] for k=0 to N-1
Ziir.s	Implements the infinite impulse response (IIR) filter represented by cascade second-order sections with 4 multiplications per section
Ziir2.s	Implements a 2-section and 4-multiplier per section infinite impulse response (IIR) filter represented by cascade second-order sections with 4 multiplications per section
Ziir3.s	Implements a 3-section and 4-multiplier per section infinite impulse response filter represented by cascade second-order sections with 4 multiplications per section
Ziir4.s	Implements a 4-section and 4-multiplier per section infinite impulse response (IIR) filter represented by cascade second-order sections with 4 multiplications per section
Ziird.s	Implements the infinite impulse response (IIR) filter represented by cascade second-order sections with 5 multiplications per section
Ziirt.s	Implements the infinite impulse response (IIR) filter represented by cascade second-order sections with 5 multiplications per section
Ziirt1.s	Implements a single 5-multiplier section infinite impulse response (IIR) filter represented by a single second-order section with 5 multiplications
Ziirt2.s	Implements a 2-section and 5-multiplier per section infinite impulse response (IIR) filter represented by cascade second-order sections with 5 multiplications per section
Ziirt3.s	Implements a 3-section and 5-multiplier per section infinite impulse response (IIR) filter represented by cascade second-order sections with 5 multiplications per section

Ziirt4.s	Implements a 4-section and 5-multiplier per section infinite impulse response (IIR) filter represented by cascade second-order sections with 5 multiplications per section
Zlms.s	Implements a least-mean-square (LMS) algorithm as an adaptive IIR filter. W(k + 1) = W(k) + 2 * u * E(k) * X(k) is the algorithm to update the wieghts if the FIR
Zfft.s	Implements the Fast Fourier Transform (FFT) algorithm. The complex FFT sixe is N = 2**M with the maximum size of N = 1024

APPENDIX E: SOURCE CODE: RANK-ORDER FILTER

```
/*************************************************************************
 * This program computes a vector rank order filter on given
 * data. The data comes from a UNIX file, "input_data," and the output
 * is saved at label "out_data" on the Development System.
 *
 * The input data has the following format:
 *
 *              1st element:            window size
 *              2nd element:            relative median location
 *              3rd element:            rank of filter
 *              4th element:            size of data (N)
 *              5th element:            data point 1
 *                   .                       .
 *                   .                       .
 *                   .                       .
 *              (N + 4)th element:      data point N
 *
 * Where:
 *              The window size must be an odd number >= 3
 *              Rank is from lowest to highest (1 is lowest)
 *************************************************************************/

#define IN_DATA         0x1400
#define TMP_DATA        0x1800
#define count           r9

.global window, compar, swap, noswap, pause, loop1, loop2, loop3, in_data
.global out_data, rep1, win1, loop3i, comp1, temp

int     4
.align  4

pause:  r1 = IN_DATA    /* location of input data */
        r2 = TMP_DATA   /* location of output data */
        r3 = temp       /* temporary array for sorting */
        r4 = 4          /* offset */

        r5 = *r1++r4    /* window size */
        r6 = *r1++r4    /* window median location */
        r7 = *r1++r4    /* rank of filter (1 is min) */
        r8 = *r1++r4    /* size of data */

        r5 = r5 + (-2)  /* adjust for compare to zero */
        count = r5      /* window size for countdown */
        r10 = r6        /* relative median location for countdown */
        r6 = r6 + (-1)  /* adjust to use as offset */
        r11 = r8        /* size of data for countdown */
        r7 = r7 + (-1)  /* adjust for use as offset */
```

```
        r7 = r7 * 2             /* calculate offset */
        r7 = r7 * 2             /* calculate offset */
        r8 = r8 + (-2)  /* adjust size of data for filter */

/*************************************************************************
 * Set up the data space ... this includes padding the input with zeroes
 * so that the window can convolve across the entire data space
 *************************************************************************/

        r10 = r10 + (-2)
        nop
        r10 = r10 * 2

loop1:  r2 = r2 + 4             /* add the pad */
        if (r10-- >= 0) goto loop1
        nop

/*************************************************************************
 * the following code transfers the input data to the temporary data space
 * for sorting and eventual replacement back into the input data space ...
 *************************************************************************/

        r11 = r11 + (-2)        /* adjust for countdown */
        nop
loop2:  *r2++ = a3 = *r1++      /* transfer the data to TMP_DATA space */
        if (r11-- >= 0) goto loop2
        nop

/*************************************************************************
 * the following code loads a window into the temporary data space for
 * sorting and rank replacement ...
 *************************************************************************/

        r13 = TMP_DATA          /* initialize for window scrolling */
        r1 = IN_DATA            /* re-initialize r1 for end replacement */
        r12 = r13
        goto loop3              /* call the loop ... skip loop3i the */
        nop                     /* first time through */

loop3i: r13 = r13 + 4           /* increment pointer for window */
        r12 = r13               /* pointer for window */
        count = r5              /* reset the count */
        r3 = temp               /* reset the pointer to the window space */

loop3:  *r3++ = a3 = *r12++     /* load the sort space with the window */
        if(count-- >= 0) goto loop3
        nop

/*************************************************************************
 * Code for the bubblesort routine follows ...
 *************************************************************************/

window: count = r5                      /* restore the count value */
win1:   r17 = count + (-1)
        if (count-- >= 0) goto compar   /* not finished yet, do the compare */
        nop
        goto repl                       /* the sort is completed */
        nop

/*************************************************************************
```

```
 * this section of code compares two adjacent elements to determine if they
 * should be swapped ...
 ************************************************************************/

compar: r10 = temp              /* r10 set equal to the first element in the */
        r14 = temp + 4          /* window, r14 assigned to r10's neighbor */
comp1:  a0 = *r14 - *r10        /* do the comparison ... if they need to be */
        3*nop
        if (alt) goto swap      /* swapped, swap 'em ... otherwise, don't */
        nop
        goto noswap
        nop

/************************************************************************
 * if the elements have been determined to be out of order, the exchange
 * is performed ...
 ************************************************************************/

swap:   a3 = *r14               /* swap routine ... performs flip-flop */
        4*nop
        *r14 = a0 = *r10        /* of data passed in upper and lower   */
        4*nop
        *r10 = a1 = a3
        4*nop
        r10 = r10 + 4                   /* increment the pointers */
        r14 = r14 + 4
        if (r17-- >= 0) goto comp1      /* if not end of window */
        nop
        goto win1                       /* reached the end of window */
        nop

/************************************************************************
 * even if the elements are in the proper order, pointers still must be
 * incremented and checks made to determine if sort should return ...
 ************************************************************************/

noswap: r14 = r14 + 4                   /* the elements were in the correct */
        r10 = r10 + 4                   /* order ... increment the pointers */
        if (r17-- >= 0) goto comp1      /* and check the done condition     */
        nop
        goto win1                       /* they were the last two elements  */
        nop                             /* in the window ... go back        */

/************************************************************************
 * the following code performs element replacement taking the sorted data from
 * the temporary sorting space and placing it back in the data space pointed
 * to by IN_DATA
 ************************************************************************/

rep1:   r10 = temp      /* reuse r10 as pointer to sorted window space */
        r10 = r10 + r7  /* r10 now points to the correct element in the */
                        /* sorted window */

        *r1++ = a0 = *r10       /* Replace the original data space with the */
        if(r8-- >= 0) goto loop3i       /* filtered data */
        nop

        goto pause
        nop
```

```
/*****************************************************************************
 * data space declaration ... this assures that the memory holding the data
 * is initialized to zero ...
 *****************************************************************************/

. =0x1400

in_data:        64*float 0.0

. =0x1800

out_data:       96*float 0.0

. =0x1980

temp:           16*float 0.0
```

APPENDIX F: SOURCE CODE: α TRIMMED-MEAN FILTER

```
/*************************************************************************
 * This program computes an alpha-trimed-mean filter on given
 * data.  The data comes from a UNIX file, "input_data," and the
 * output is saved at label "out_data" on the Development System.
 *
 * The input data has the following format:
 *
 *              1st element:            window size
 *              2nd element:            relative median location
 *              3rd element:            data size (N)
 *              4th element:            data point 1
 *              5th element:            data point 2
 *                   .                       .
 *                   .                       .
 *                   .                       .
 *              (N + 3)th element:      data point N
 *
 * Where:
 *              The window size must be an odd number >= 3
 *              Rank is from lowest to highest (1 is lowest)
 *************************************************************************/

#define IN_DATA         0x1400
#define TMP_DATA        0x1800
#define count           r9
#define zero            0

.global check, window, compar, swap, noswap, pause, loop1, loop2
.global out_data, rep1, win1, loop3i, comp1, temp, denom, numer
.global loop3, in_data, quot, total

int     4
.align  4

pause:  r1 = IN_DATA    /* location of input data */
        r2 = TMP_DATA   /* location of output data */
        r3 = temp       /* temporary array for sorting */
        r4 = 4          /* offset */

        r5 = *r1++r4    /* window size */
        r6 = *r1++r4    /* window median location (relative) */
        r8 = *r1++r4    /* size of data */

        r5 = r5 + (-2)  /* adjust for compare to zero */
        count = r5      /* window size for countdown */
        r10 = r6        /* relative median location for countdown */
        r6 = r6 + (-1)  /* adjust to use as offset */
        r11 = r8        /* size of data for countdown */
```

```
        r8 = r8 + (-2)  /* adjust size of data for filter */

/*************************************************************************
 * Set up the data space ... this includes padding the input with zeroes
 * so that the window can convolve across the entire data space
 *************************************************************************/

        r7 = in_data
        r10 = r10 + (-2)
        nop
        r10 = r10 * 2

loop1:  r2 = r2 + 4             /* add the pad */
        if (r10-- >= 0) goto loop1
        nop

/*************************************************************************
 * the following code transfers the input data to the temporary data space
 * for sorting and eventual replacement back into the input data space ...
 *************************************************************************/

        r11 = r11 + (-2)        /* adjust for countdown */
        nop
loop2:  *r2++ = a3 = *r1++      /* transfer the data to TMP_DATA space */
        if (r11-- >= 0) goto loop2
        nop

/*************************************************************************
 * the following code loads a window into the temporary data space for
 * sorting and rank replacement ...
 *************************************************************************/

        r13 = TMP_DATA          /* initialize for window scrolling */
        r1 = IN_DATA            /* re-initialize r1 for end replacement */
        r12 = r13
        goto loop3              /* call the loop ... skip loop3i the */
        nop                     /* first time through */

loop3i: r13 = r13 + 4           /* increment pointer for window */
        r12 = r13               /* pointer for window */
        count = r5              /* reset the count */
        r3 = temp               /* reset the pointer to the window space */

loop3:  *r3++ = a3 = *r12++     /* load the sort space with the window */
        if(count-- >= 0) goto loop3
        nop

/*************************************************************************
 * Code for the bubblesort routine follows ...
 *************************************************************************/

window: count = r5                      /* restore the count value */
win1:   r17 = count + (-1)
        if (count-- >= 0) goto compar   /* not finished yet, do the compare */
        nop
        goto repl                       /* the sort is completed */
        nop

/*************************************************************************
 * this section of code compares two adjacent elements to determine if they
```

```
 * should be swapped ...
 ************************************************************************/

compar: r10 = temp              /* r10 set equal to the first element in the */
        r14 = temp + 4          /* window, r14 assigned to r10's neighbor */
comp1:  a0 = *r14 - *r10        /* do the comparison ... if they need to be */
        3*nop
        if (alt) goto swap      /* swapped, swap 'em ... otherwise, don't */
        nop
        goto noswap
        nop

/************************************************************************
 * if the elements have been determined to be out of order, the exchange
 * is performed ...
 ************************************************************************/

swap:   a3 = *r14               /* swap routine ... performs flip-flop */
        3*nop
        *r14 = a0 = *r10        /* of data passed in upper and lower   */
        3*nop
        *r10 = a1 = a3
        3*nop
        r10 = r10 + 4                   /* increment the pointers */
        r14 = r14 + 4
        if (r17-- >= 0) goto comp1      /* if not end of window */
        nop
        goto win1                       /* reached the end of window */
        nop

/************************************************************************
 * even if the elements are in the proper order, pointers still must be
 * incremented and checks made to determine if sort should return ...
 ************************************************************************/

noswap: r14 = r14 + 4                   /* the elements were in the correct */
        r10 = r10 + 4                   /* order ... increment the pointers */
        if (r17-- >= 0) goto comp1      /* and check the done condition     */
        nop
        goto win1                       /* they were the last two elements  */
        nop                             /* in the window ... go back        */

/************************************************************************
 * At this point, the data has been sorted and is ready to be processed.
 * To accomplish this, the hihest and lowest elements of the window are
 * tossed and the average value of those points remaining is calculated.
 ************************************************************************/

rep1:   r2 = numer
        r4 = 0
        r10 = temp + 4
        *r2 = r4
        r19 = r5 + (-2)
        *r2 = a2 = float(*r2)

total:  a3 = *r10++
        3*nop
        *r2 = a2 = *r2 + a3
        3*nop
        if (r19-- >= 0) goto total
```

```
        nop
        r14 = numer
        *r14 = a2 = *r2
        nop
        r14 = denom
        r10 = quot
        nop
        *r10 = r5
        nop
        *r14 = a2 = float(*r10)
        nop

        call Zdiv(r14)
        nop
int     quot, denom, numer
int
        *r7++ = a0 = *r11                /* save the output */
check:  if (r8-- >= 0) goto loop3i       /* is the main loop done? */
        nop

        goto pause                       /* yes it is ... */
        nop

/*****************************************************************************
 * Divide routine ... enables division of two real numbers. Used by atmean
 * to calculate the average value of the data points in the window.
 *****************************************************************************/

Zdiv:   r11 = *r14++        /* r11 points to location for output */
        r1 = *r14++         /* r1 points to operand, D */
        r2 = *r14++         /* r2 points to operand, N */
        *r11 = a0 = *r1     /* move D to temp location */
        r4 = ZdivB

        r6 = 0x80;
        r15l = *r1           /* exponent(D) moved to r15h */
        *r11 = r6l;          /* set exponent to 0 */
        a2 = - *r2           /* a2=N*sgn(D) and a0=abs(mantissa(D)) */
        a0 = - *r11

        a0 = ifalt(*r11)
        a2 = ifalt(*r2)
        r15 = -1-r15         /* -exponent(D) - 1 */
        *r11 = a1 = *r4++    /* zero temp location */
        a0 = a0 * *r4++      /* D(0) = abs(mantissa(D)) * (2/3) */

        a1 = -a0 + *r4++     /* f(1) = (-abs(mantissa(D)) * (2/3)) + 2 */
        a2 = a2 * *r4--      /* N(0) = N * sgn(D) * (4/3) */
        r6 = 2               /* loop counter */
ZdivA:  a0 = a1 * a0;        /* D(i) = f(i) * D(i-1) */
        a1 = -a0 + *r4       /* f(i+1) = -D(i) + 2 */

        if(r6-- >= 0) goto ZdivA
        a2 = a1 * a2;        /* D(i) = f(i) * N(i-1) */
        nop                  /* required for latent a2 (2 instrs. below) */
        *r11 = r15l
        *r11 = a2 = *r11 * a2

        r14 = r14 + 2
        return(r14)
```

```
        nop

ZdivB:  float 0.0, 0.6666667, 2.0, 1.3333333

/*****************************************************************************
 * data space declaration ... this assures that the memory holding the data
 * is initialized to zero ...
 *****************************************************************************/

.=0x1400

in_data:        64*float 0.0

.=0x1800

out_data:       96*float 0.0

.=0x1980

temp:           16*float 0.0
quot:           float 0.0
denom:          float 0.0
numer:          float 0.0
outptr:         float 0.0
```

APPENDIX G: SOURCE CODE: 2-D MORPHOLOGICAL OPERATIONS

```
/*************************************************************************
 * This program performs an EROSION on a 64 x 64 image using a 5 x 5
 * structuring element.
 *
 * Input data has the following form:
 *
 *      1st element -- data size (64 for a 64 x 64 image)
 *      2nd element -- x-size of window (5 for most cases)
 *      3rd element -- y-size of window (5 for most cases)
 *      4th element -- total number of window elements (25 for 5 x 5)
 *      5th element -- total size of data (4096 for 64 x 64)
 *      6th element -- 1st data point
 *      7th element -- 2nd data point
 *           .              .
 *           .              .
 *           .              .
 * Input data is stored at label "in_data"
 * After the program run, output is stored at label "in_data"
 *
 *************************************************************************/

.global temp, pause, xfer, xfer1, colchk, offset, xsize, ysize, oper
.global in_data, out_data, win_x, win_y, m_loop, nxtrow, end, cnvert
.global winsize, mask, erode, erode1, cntnu, cntnu1, wzero, end1
.global chk, tmp, tmp1, chksum
.align 4

pause:  r1 = in_data     /* location of input data */
        r2 = out_data    /* location of output data */
        r3 = temp        /* location of temporary sort space */
        r4 = 4           /* constant value for offset */

        r5 = *r1++r4     /* data size (square) */
        r6 = *r1++r4     /* x-size of window */
        r7 = *r1++r4     /* y-size of window */
        r19 = *r1++r4    /* total # of window elements */
        r8 = *r1++r4     /* total size of data */

        r7 = -r7         /* calculate the y-offset for the window routine */
        r4 = r5          /* subtract the data size */
        r4 = r4 + r7
        r4 = r4 + (-1)   /* adjust for countdown */
        r6 = -r6         /* do the same for the x-offset */
        r13 = r5
        r13 = r13 + r6
        r13 = r13 + (-1)
        r7 = -r7         /* reset r7 */
        r5 = r5 + r6     /* offset is equal to datasize - x-size */
```

```
        r5 = r5 * 2     /* offset must be reset for floating-point space */
        r5 = r5 * 2     /* which is 4 bytes per data point */
        r11 = offset    /* set the pointer to the offset */
        *r11 = r5       /* put the data there */
        r6 = -r6        /* restore r6 to original value */
        r7 = r7 + (-2)  /* adjust for countdown */
        r6 = r6 + (-2)  /* adjust for countdown */
        r19 = r19 + (-2)/* adjust for countdown */

/*****************************************************************************
 * put parameters which will be referred to often away in memory ...
 *****************************************************************************/

        r9 = xsize      /* point to x-size */
        r10 = ysize     /* point to y-size */
        *r9 = r6        /* put x-size away in memory */
        *r10 = r7       /* put y-size away in memory */
        r9 = win_x      /* point to win_x */
        r10 = win_y     /* point ot win_y */
        *r9 = r13       /* put x window offset away in memory */
        *r10 = r4       /* put y window offset away in memory */
        r9 = winsize    /* point to the window size */
        *r9 = r19       /* put the window size away in memory */

/*****************************************************************************
 * Move the input data to the output space ... the actual output data will
 * be located at label in_data after the program runs ...
 *****************************************************************************/

        r5 = mask

cnvert: a0 = *r5                /* convert the mask to integer*/
        3*nop
        *r5 = a0 = int(a0)
        nop
        r5 = r5 + 4
        if (r19-- >= 0) goto cnvert
        nop

xfer1:  *r2++ = a3 = *r1++              /* do the transfer */
        if (r8-- >= 0) goto xfer1       /* are we done ? */
        nop

        r1 = in_data
        r12 = out_data          /* set up the pointers */
        r12 = r12 + (-4)

/*****************************************************************************
 * Mainloop routine ... this routine handles all manipulations of the main
 * window pointer ...
 *****************************************************************************/

m_loop: r12 = r12 + 4
        r2 = r12        /* increment window counter */
        r9 = ysize      /* re-initialize the y size for countdown */
        r7 = *r9
        r3 = temp       /* reset pointer to temporary window */
        goto xfer       /* call the window routine to move the window */
        nop             /* to the temporary sort space */
```

```
/*****************************************************************************
 * Transfer routine ... moves data, one window at a time, into temp data
 * space to be operated on ...
 *****************************************************************************/

xfer:   r9 = xsize                      /* reset the counter each time */
        r6 = *r9                        /* after finishing a row ... */

colchk: *r3++ = a3 = *r2++              /* transfer the data */
        if (r6-- >= 0) goto colchk
        nop
        r9 = offset                     /* put the offset into r10 */
        r10 = *r9
        nop
        r2 = r2 + r10           /* add the offset to the window pointer */
        if (r7-- >= 0) goto xfer        /* check for end */
        nop
        goto erode
        nop

/*****************************************************************************
 * the loading of the temporary data space is completed ... now go and
 * work on the data (i.e. sort, weigh, ...)
 *****************************************************************************/

oper:   if (r4-- >= 0) goto m_loop      /* has the window gone all the way */
        nop                             /* across the data space yet??     */
        if (r13-- >= 0) goto nxtrow     /* has the window reached the bottom */
        nop                             /* of the data space yet??           */
        goto end
        nop

nxtrow: r9 = win_x              /* reset the x window counter for the next */
        r4 = *r9                /* iteration across the data space */
        r8 = xsize
        r9 = *r8
        nop
        r9 = r9 * 2             /* fix the byte alignment */
        r9 = r9 * 2             /* fix the byte alignment */
        r12 = r12 + r9          /* move the window pointer down to the */
        r12 = r12 + 4           /* beginning of the next row */
        goto m_loop             /* do the window operation again */
        nop

/*****************************************************************************
 * The following code performs the erosion on the data ...
 *****************************************************************************/

erode:  r9 = winsize            /* point to the window size       */
        r6 = *r9                /* restore the window size in r6  */
        r3 = temp               /* r3 points to the window        */
        r5 = mask               /* r5 points to the mask          */
        r7 = 0

erode1: r8 = *r3
        r9 = *r5
        nop
        r8 = r8 & r9            /* 'and' the mask and the window  */
        *r3 = r8
        r9 = tmp                /* point to 1                     */
```

```
        *r3 = a2 = float(*r3)   /* convert the result to float    */
        3*nop
        a0 = *r3 - *r9          /* subtract to check equality     */
        3*nop
        if (ane) goto cntnu     /* if the result was not '0' try     */
        nop                     /* again                             */
        r7 = r7 + 1             /* add one to the accumulated 'ands' */
        goto cntnu              /* after adding, keep going ...      */
        nop

cntnu:  if (r6-- >= 0 ) goto cntnu1
        nop
        goto chksum
        nop

cntnu1: r5 = r5 + 4     /* increment the mask pointer   */
        r3 = r3 + 4     /* increment the window pointer */
        goto erode1     /* have checked the window count and found it > 0 */
        nop             /* so return for the next compare                */

wzero:  r9 = 0
        *r1 = r9        /* write a zero to the memory location */
        r1 = r1 + 4
        goto oper       /* return to the main loop             */
        nop

end1:   r9 = 1
        *r1 = r9        /* write a one to the output memory location */
        r1 = r1 + 4
        goto oper       /* return to the main loop                   */
        nop

chksum: r9 = chk        /* point to checksum data location */
        a0 = *r9        /* place it in a0 */
        r9 = tmp1       /* point to the temporary data location */
        *r9 = r7        /* place the accumulated total there    */
        nop
        a1 = float(*r9) /* put the floating equivalent into a1  */
        3*nop
        a2 = a0 - a1    /* subtract for check equal to zero     */
        3*nop
        if (ane) goto wzero     /* if they weren't equal, write */
        nop                     /* a zero and return to oper    */
        goto end1               /* otherwise, write a one and   */
        nop                     /* return to oper               */

end:    nop

/*****************************************************************************
 * memory declaration ... makes space for temporary storage
 *****************************************************************************/

.=0x0800
xsize:          float 0.0
ysize:          float 0.0
offset:         float 0.0
temp:           81*float 0.0
.=0x1000
in_data:
.=0x5010
```

```
win_x:          float 0.0
win_y:          float 0.0
winsize:        float 0.0
chk:            float 11.0
tmp:            float 1.0
tmp1:           float 0.0
mask:           float 0.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0
                float 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0
                float 0.0, 1.0, 1.0, 0.0, 0.0
out_data:
```

```
/*************************************************************************
 * This program performs a DILATION on a 64 x 64 image using a 5 x 5
 * structuring element.
 *
 * Input data has the following form:
 *
 *      1st element -- data size (64 for a 64 x 64 image)
 *      2nd element -- x-size of window (5 for most cases)
 *      3rd element -- y-size of window (5 for most cases)
 *      4th element -- total number of window elements (25 for 5 x 5)
 *      5th element -- total size of data (4096 for 64 x 64)
 *      6th element -- 1st data point
 *      7th element -- 2nd data point
 *           .              .
 *           .              .
 *           .              .
 * Input data is stored at label "in_data"
 * After the program run, output is stored at label "in_data"
 *
 *************************************************************************/

.global temp, pause, xfer, xfer1, colchk, offset, xsize, ysize, oper
.global in_data, out_data, win_x, win_y, m_loop, nxtrow, end, two
.global winsize, maski, dilate, dil1, cntnu, cntnu1, wzero, end1, test
.align 4

pause:  r1 = in_data     /* location of input data */
        r2 = out_data    /* location of output data */
        r3 = temp        /* location of temporary sort space */
        r4 = 4           /* constant value for offset */

        r5 = *r1++r4     /* data size (square) */
        r6 = *r1++r4     /* x-size of window */
        r7 = *r1++r4     /* y-size of window */
        r19 = *r1++r4    /* total # of window elements */
        r8 = *r1++r4     /* total size of data */

        r7 = -r7         /* calculate the y-offset for the window routine */
        r4 = r5          /* subtract the data size */
        r4 = r4 + r7
        r4 = r4 + (-1)   /* adjust for countdown */
        r6 = -r6         /* do the same for the x-offset */
        r13 = r5
```

```
        r13 = r13 + r6
        r13 = r13 + (-1)
        r7 = -r7          /* reset r7 */
        r5 = r5 + r6      /* offset is equal to datasize - x-size */
        r5 = r5 * 2       /* offset must be reset for floating-point space */
        r5 = r5 * 2       /* which is 4 bytes per data point */
        r11 = offset      /* set the pointer to the offset */
        *r11 = r5         /* put the data there */
        r6 = -r6          /* restore r6 to original value */
        r7 = r7 + (-2)    /* adjust for countdown */
        r6 = r6 + (-2)    /* adjust for countdown */
        r19 = r19 + (-2)/* adjust for countdown */

/*************************************************************************
 * put parameters which will be referred to often away in memory ...
 *************************************************************************/

        r9 = xsize        /* point to x-size */
        r10 = ysize       /* point to y-size */
        *r9 = r6          /* put x-size away in memory */
        *r10 = r7         /* put y-size away in memory */
        r9 = win_x        /* point to win_x */
        r10 = win_y       /* point ot win_y */
        *r9 = r13         /* put x window offset away in memory */
        *r10 = r4         /* put y window offset away in memory */
        r9 = winsize      /* point to the window size */
        *r9 = r19         /* put the window size away in memory */

/*************************************************************************
 * Move the input data to the output space ... the actual output data will
 * be located at label in_data after the program runs ...
 *************************************************************************/

xfer1:  *r2++ = a3 = *r1++              /* do the transfer */
        if (r8-- >= 0) goto xfer1       /* are we done ? */
        nop

        r1 = in_data
        r12 = out_data          /* set up the pointers */
        r12 = r12 + (-4)

/*************************************************************************
 * Mainloop routine ... this routine handles all manipulations of the main
 * window pointer ...
 *************************************************************************/

m_loop: r12 = r12 + 4
        r2 = r12          /* increment window counter */
        r9 = ysize        /* re-initialize the y size for countdown */
        r7 = *r9
        r3 = temp         /* reset pointer to temporary window */
        goto xfer         /* call the window routine to move the window */
        nop               /* to the temporary sort space */

/*************************************************************************
 * Transfer routine ... moves data, one window at a time, into temp data
 * space to be operated on ...
 *************************************************************************/

xfer:   r9 = xsize                      /* reset the counter each time */
```

```
        r6 = *r9                            /* after finishing a row ... */

colchk: *r3++ = a3 = *r2++                  /* transfer the data */
        if (r6-- >= 0) goto colchk
        nop
        r9 = offset                         /* put the offset into r10 */
        r10 = *r9
        nop
        r2 = r2 + r10           /* add the offset to the window pointer */
        if (r7-- >= 0) goto xfer            /* check for end */
        nop
        goto dilate
        nop

/*****************************************************************************
 * the loading of the temporary data space is completed ... now go and
 * work on the data (i.e. sort, weigh, ...)
 *****************************************************************************/

oper:   if (r4-- >= 0) goto m_loop          /* has the window gone all the way */
        nop                                 /* across the data space yet??     */
test:   if (r13-- >= 0) goto nxtrow         /* has the window reached the bottom */
        nop                                 /* of the data space yet??           */
        goto end
        nop

nxtrow: r9 = win_x                  /* reset the x window counter for the next */
        r4 = *r9                    /* iteration across the data space */
        r8 = xsize
        r9 = *r8
        nop
        r9 = r9 * 2                 /* fix the byte alignment */
        r9 = r9 * 2                 /* fix the byte alignment */
        r12 = r12 + r9              /* move the window pointer down to the */
        r12 = r12 + 4               /* beginning of the next row */
        goto m_loop                 /* do the window operation again */
        nop

/*****************************************************************************
 * The following code performs the erosion on the data ...
 *****************************************************************************/

dilate: r9 = winsize                /* point to the window size       */
        r6 = *r9                    /* restore the window size in r19 */
        r3 = temp                   /* r3 points to the window        */
        r5 = maski                  /* r5 points to the inverted mask */

dil1:   r9 = two
        a1 = float(*r3)             /* change to float for standard          */
        a2 = *r9                    /* load a "2" into a2 for future compare */
        2*nop
        a0 = a1 + *r5               /* do addition generate the "and"        */
        3*nop
        a1 = a0 - a2                /* compare to 2 ... if true both were 1s */
        3*nop
        if (ane) goto cntnu         /* if the result is not zero, keep going */
        nop
        goto end1                   /* if the result is zero, write a "1"    */
        nop
```

```
cntnu:  if (r6-- >= 0 ) goto cntnu1
        nop
        goto wzero
        nop

cntnu1: r5 = r5 + 4     /* increment the mask pointer   */
        r3 = r3 + 4     /* increment the window pointer */
        goto dil1       /* have checked the window count and found it > 0 */
        nop             /* so return for the next compare                */

wzero:  r9 = 0
        *r1 = r9        /* write a zero to the memory location */
        r1 = r1 + 4
        goto oper       /* return to the main loop             */
        nop

end1:   r9 = 1
        *r1 = r9        /* write a one to the output memory location */
        r1 = r1 + 4
        goto oper       /* return to the main loop                   */
        nop

end:    nop

/*****************************************************************************
 * memory declaration ... makes space for temporary storage
 *****************************************************************************/

.=0x0800
xsize:          float 0.0
ysize:          float 0.0
offset:         float 0.0
temp:           81*float 0.0
.=0x1000
in_data:
.=0x5010
win_x:          float 0.0
win_y:          float 0.0
winsize:        float 0.0
maski:          float 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
                float 0.0, 1.0, 1.0, 1.0, 1.0, 0.0, 1.0, 0.0, 0.0, 1.0
                float 0.0, 0.0, 1.0, 1.0, 0.0
two:            float 2.0
out_data:
```

APPENDIX H: SOURCE CODE: 2-D MEDIAN FILTER

```
/*************************************************************************
 *
 * This program implements a two-dimensional median filter on the
 * input image.  Input data has the following format:
 *
 * Input:        1st element -- data size (for 64 x 64 element = 64)
 *               2nd element -- x-size (width) of window
 *               3rd element -- y-size (depth) of window
 *               4th element -- number of elements in window (3 by 3 = 9)
 *               5th element -- relative median location (5 for a 3 x 3)
 *               6th element -- total number of input data elements
 *               7th element -- 1st data point
 *               8th element -- 2nd data point
 *                   .               .
 *                   .               .
 *                   .               .
 *
 * Input is placed at label "in_data"
 * After the program run, the output is written to label "in_data"
 *
 *************************************************************************/

.global temp, pause, xfer, xfer1, colchk, offset, xsize, ysize, oper
.global in_data, out_data, win_x, win_y, m_loop, nxtrow, end
.global window, compar, swap, noswap, repl, win1, comp1
.align 4

pause:  r1 = in_data     /* location of input data */
        r2 = out_data    /* location of temporary data space */
        r3 = temp        /* location of temporary sort space */
        r4 = 4           /* constant value for offset */

        r5 = *r1++r4     /* data size (square) */
        r6 = *r1++r4     /* x-size of window */
        r7 = *r1++r4     /* y-size of window */
        r18 = *r1++r4    /* total number of elements in window */
        r19 = *r1++r4    /* relative median location */
        r8 = *r1++r4     /* total size of data */

        r18 = r18 + (-2)/* adjust for countdown in bubblesort routine */
        r19 = r19 + (-1)/* adjust r19 to use as an offset for the sort */
        r19 = r19 * 2
        r19 = r19 * 2
        r7 = -r7         /* calculate the y-offset for the window routine */
        r4 = r5          /* subtract the data size */
        r4 = r4 + r7
        r4 = r4 + (-1)   /* adjust for countdown */
        r6 = -r6         /* do the same for the x-offset */
```

```
        r13 = r5
        r13 = r13 + r6
        r13 = r13 + (-1)
        r7 = -r7        /* reset r7 */
        r5 = r5 + r6    /* offset is equal to datasize - x-size */
        r5 = r5 * 2     /* offset must be reset for floating-point space */
        r5 = r5 * 2     /* which is 4 bytes per data point */
        r11 = offset    /* set the pointer to the offset */
        *r11 = r5       /* put the data there */
        r6 = -r6        /* restore r6 to original value */
        r7 = r7 + (-2)  /* adjust for countdown */
        r6 = r6 + (-2)  /* adjust for countdown */

/*************************************************************************
 * put parameters which will be referred to often away in memory ...
 *************************************************************************/

        r9 = xsize      /* point to x-size */
        r10 = ysize     /* point to y-size */
        *r9 = r6        /* put x-size away in memory */
        *r10 = r7       /* put y-size away in memory */
        r9 = win_x      /* point to win_x */
        r10 = win_y     /* point ot win_y */
        *r9 = r13       /* put x window offset away in memory */
        *r10 = r4       /* put y window offset away in memory */

/*************************************************************************
 * Move the input data to the output space ... the actual output data will
 * be located at label in_data after the program runs ...
 *************************************************************************/

xfer1:  *r2++ = a3 = *r1++              /* do the transfer */
        if (r8-- >= 0) goto xfer1       /* are we done ? */
        nop

        r1 = in_data
        r12 = out_data          /* set up the pointers */
        r12 = r12 + (-4)

/*************************************************************************
 * Mainloop routine ... this routine handles all manipulations of the main
 * window pointer ...
 *************************************************************************/

m_loop: r12 = r12 + 4
        r2 = r12        /* increment window counter */
        r9 = ysize      /* re-initialize the y size for countdown */
        r7 = *r9
        r3 = temp       /* reset pointer to temporary window */
        goto xfer       /* call the window routine to move the window */
        nop             /* to the temporary sort space */

/*************************************************************************
 * Transfer routine ... moves data, one window at a time, into temp data
 * space to be operated on ...
 *************************************************************************/

xfer:   r9 = xsize                      /* reset the counter each time */
        r6 = *r9                        /* after finishing a row ... */
```

```
colchk: *r3++ = a3 = *r2++              /* transfer the data */
        if (r6-- >= 0) goto colchk
        nop
        r9 = offset                     /* put the offset into r10 */
        r10 = *r9
        nop
        r2 = r2 + r10           /* add the offset to the window pointer */
        if (r7-- >= 0) goto xfer        /* check for end */
        nop
        goto window             /* falls through and calls the sort      */
        nop                     /* routine ... also will output a result */

/*****************************************************************************
 * the loading of the temporary data space is completed ... now go and
 * work on the data (i.e. sort, weigh, ...)
 *****************************************************************************/

oper:   if (r4-- >= 0) goto m_loop      /* has the window gone all the way */
        nop                             /* across the data space yet??     */
        if (r13-- >= 0) goto nxtrow     /* has the window reached the bottom */
        nop                             /* of the data space yet??           */
        goto end
        nop

nxtrow: r9 = win_x              /* reset the x window counter for the next */
        r4 = *r9                /* iteration across the data space */
        r8 = xsize
        r9 = *r8
        nop
        r9 = r9 * 2             /* fix the byte alignment */
        r9 = r9 * 2             /* fix the byte alignment */
        r12 = r12 + r9          /* move the window pointer down to the */
        r12 = r12 + 4           /* beginning of the next row */
        goto m_loop             /* do the window operation again */
        nop

/*****************************************************************************
 * Code for the bubblesort routine follows ...
 *****************************************************************************/

window: r6 = r18                        /* restore the count value */
win1:   r17 = r6 + (-1)
        if (r6-- >= 0) goto compar      /* not finished yet, do the compare */
        nop
        goto repl                       /* the sort is completed */
        nop

/*****************************************************************************
 * this section of code compares two adjacent elements to determine if they
 * should be swapped ...
 *****************************************************************************/

compar: r10 = temp              /* r10 set equal to the first element in the */
        r14 = temp + 4          /* window, r14 assigned to r10's neighbor */
comp1:  a0 = *r14 - *r10        /* do the comparison ... if they need to be */
        3*nop
        if (alt) goto swap      /* swapped, swap 'em ... otherwise, don't */
        nop
        goto noswap
        nop
```

```
/*****************************************************************************
 * if the elements have been determined to be out of order, the exchange
 * is performed ...
 *****************************************************************************/

swap:   a3 = *r14               /* swap routine ... performs flip-flop */
        4*nop
        *r14 = a0 = *r10        /* of data passed in upper and lower   */
        4*nop
        *r10 = a1 = a3
        4*nop
        r10 = r10 + 4                   /* increment the pointers */
        r14 = r14 + 4
        if (r17-- >= 0) goto comp1      /* if not end of window */
        nop
        goto win1                       /* reached the end of window */
        nop

/*****************************************************************************
 * even if the elements are in the proper order, pointers still must be
 * incremented and checks made to determine if sort should return ...
 *****************************************************************************/

noswap: r14 = r14 + 4                   /* the elements were in the correct */
        r10 = r10 + 4                   /* order ... increment the pointers */
        if (r17-- >= 0) goto comp1      /* and check the done condition     */
        nop
        goto win1                       /* they were the last two elements  */
        nop                             /* in the window ... go back        */

/*****************************************************************************
 * the following code performs element replacement taking the sorted data from
 * the temporary sorting space and placing it back in the data space pointed
 * to by IN_DATA
 *****************************************************************************/

repl:   r10 = temp      /* reuse r10 as pointer to sorted window space */
        r10 = r10 + r19 /* r10 now points to the correct element in the */
                        /* sorted window */

        *r1++ = a0 = *r10       /* Replace the original data space with the */
        nop
        goto oper
        nop


end:    nop

/*****************************************************************************
 * memory declaration ... makes space for temporary storage
 *****************************************************************************/

.=0x0800
xsize:          float 0.0
ysize:          float 0.0
offset:         float 0.0
temp:           81*float 0.0
.=0x1000
in_data:

.=0x5010
win_x:          float 0.0
win_y:          float 0.0
out_data:
```

APPENDIX I: SOURCE CODE: 2-D LAPLACIAN OPERATOR

```
/*****************************************************************************
 * This program implements a Laplacian filter on the input image data.
 * standard input image size is 64 x 64
 * Input data has the following format:
 *
 *      1st element -- data size (64 for a 64 x 64 image)
 *      2nd element -- x-size (width) of window
 *      3rd element -- y-size (depth) of window
 *      4th element -- total size of window
 *      5th element -- total numer of input data elements
 *      6th element -- 1st data point
 *      7th element -- 2nd data point
 *           .              .
 *           .              .
 *           .              .
 *
 * Input data is placed at label "in_data"
 * After the program has completed, output data is
 * written to label "in_data"
 *****************************************************************************/

.global temp, pause, xfer, xfer1, colchk, offset, xsize, ysize, oper, lapl
.global in_data, out_data, win_x, win_y, m_loop, nxtrow, end, mask, winsize
.global sum, test
.align 4

pause:  r1 = in_data     /* location of input data */
        r2 = out_data    /* location of output data */
        r3 = temp        /* location of temporary sort space */
        r4 = 4           /* constant value for offset */

        r5 = *r1++r4     /* data size (square) */
        r6 = *r1++r4     /* x-size of window */
        r7 = *r1++r4     /* y-size of window */
        r18 = *r1++r4    /* total size of window */
        r8 = *r1++r4     /* total size of data */

        r18 = r18 + (-2)/* adjust window size for countdown */
        r7 = -r7         /* calculate the y-offset for the window routine */
        r4 = r5          /* subtract the data size */
        r4 = r4 + r7
        r4 = r4 + (-1)   /* adjust for countdown */
        r6 = -r6         /* do the same for the x-offset */
        r13 = r5
        r13 = r13 + r6
        r13 = r13 + (-1)
        r7 = -r7         /* reset r7 */
        r5 = r5 + r6     /* offset is equal to datasize - x-size */
```

```
        r5 = r5 * 2     /* offset must be reset for floating-point space */
        r5 = r5 * 2     /* which is 4 bytes per data point */
        r11 = offset    /* set the pointer to the offset */
        *r11 = r5       /* put the data there */
        r6 = -r6        /* restore r6 to original value */
        r7 = r7 + (-2)  /* adjust for countdown */
        r6 = r6 + (-2)  /* adjust for countdown */

/*****************************************************************************
 * put parameters which will be referred to often away in memory ...
 *****************************************************************************/

        r9 = xsize      /* point to x-size */
        r10 = ysize     /* point to y-size */
        *r9 = r6        /* put x-size away in memory */
        *r10 = r7       /* put y-size away in memory */
        r9 = win_x      /* point to win_x */
        r10 = win_y     /* point ot win_y */
        *r9 = r13       /* put x window offset away in memory */
        *r10 = r4       /* put y window offset away in memory */
        r9 = winsize    /* point to the window size */
        *r9 = r18       /* put window size away     */

/*****************************************************************************
 * Move the input data to the output space ... the actual output data will
 * be located at label in_data after the program runs ...
 *****************************************************************************/

xfer1:  *r2++ = a3 = *r1++              /* do the transfer */
        if (r8-- >= 0) goto xfer1       /* are we done ? */
        nop

        r1 = in_data
        r12 = out_data          /* set up the pointers */
        r12 = r12 + (-4)

/*****************************************************************************
 * Mainloop routine ... this routine handles all manipulations of the main
 * window pointer ...
 *****************************************************************************/

m_loop: r12 = r12 + 4
        r2 = r12        /* increment window counter */
        r9 = ysize      /* re-initialize the y size for countdown */
        r7 = *r9
        r3 = temp       /* reset pointer to temporary window */
        goto xfer       /* call the window routine to move the window */
        nop             /* to the temporary sort space */

/*****************************************************************************
 * Transfer routine ... moves data, one window at a time, into temp data
 * space to be operated on ...
 *****************************************************************************/

xfer:   r9 = xsize                      /* reset the counter each time */
        r6 = *r9                        /* after finishing a row ... */

colchk: *r3++ = a3 = *r2++              /* transfer the data */
        if (r6-- >= 0) goto colchk
        nop
```

```
        r9 = offset                         /* put the offset into r10 */
        r10 = *r9
        nop
        r2 = r2 + r10           /* add the offset to the window pointer */
        if (r7-- >= 0) goto xfer            /* check for end */
        nop
        goto lapl                           /* do the Laplacian operation */
        nop

/*****************************************************************************
 * the loading of the temporary data space is completed ... now go and
 * work on the data (i.e. sort, weigh, ...)
 *****************************************************************************/

oper:   if (r4-- >= 0) goto m_loop      /* has the window gone all the way */
        nop                             /* across the data space yet??     */
        if (r13-- >= 0) goto nxtrow     /* has the window reached the bottom */
        nop                             /* of the data space yet??          */
        goto end
        nop

nxtrow: r9 = win_x              /* reset the x window counter for the next */
        r4 = *r9                /* iteration across the data space         */
        r8 = xsize
        r9 = *r8
        nop
        r9 = r9 * 2             /* fix the byte alignment                  */
        r9 = r9 * 2             /* fix the byte alignment                  */
        r12 = r12 + r9          /* move the window pointer down to the     */
        r12 = r12 + 4           /* beginning of the next row               */
        goto m_loop             /* do the window operation again           */
        nop

lapl:   r5 = mask               /* r15 points to the first element of mask */
        r10 = winsize           /* put the window size back in r18         */
        r3 = temp               /* r3 points to the extracted window       */
        r6 = 0
        r18 = *r10
        *r1 = r6

sum:    a2 = *r5                /* place the mask values in a2             */
        a0 = float(*r3)         /* place data values in a0                 */
        3*nop
        a3 = a2 * a0            /* multiply the mask and the window        */
        r5 = r5 + 4             /* increment pointers                      */
        r3 = r3 + 4
        nop
        *r1 = a3 = a3 + *r1
        3*nop
test:   if (r18-- >= 0) goto sum        /* check to see if done            */
        nop
        r1 = r1 + 4                     /* increment the output pointer    */
        goto oper
        nop

end:    nop

/*****************************************************************************
 * memory declaration ... makes space for temporary storage
 *****************************************************************************/
```

```
.=0x0800
xsize:          float 0.0
ysize:          float 0.0
offset:         float 0.0
temp:           81*float 0.0
.=0x1000
in_data:
.=0x5010
win_x:          float 0.0
win_y:          float 0.0
mask:           float 0.0, 1.0, 0.0, 1.0, -4.0, 1.0, 0.0, 1.0, 0.0
winsize:        float 0.0
out_data:
```