Arithmetic on the ELXSI System 6400*

George S. Taylor
ELXSI, San Jose CA 95131
and
Computer Science Division,
Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley CA 94720

ABSTRACT

The ELXSI System 6400 is a new 64-bit general-purpose mainframe computer [1]. This paper decribes its arithmetic instruction set architecture and the organization of the arithmetic processor. The ELXSI instruction set supports a complete implementation of the proposed IEEE floating point standard [2], plus integer and decimal arithmetic. The System 6400 arithmetic processor uses ECL gate arrays to execute these instructions at high speed using a single board of hardware.

1. Introduction

The ELXSI System 6400 is a multiprocessor built around a central system bus (Figure 1).

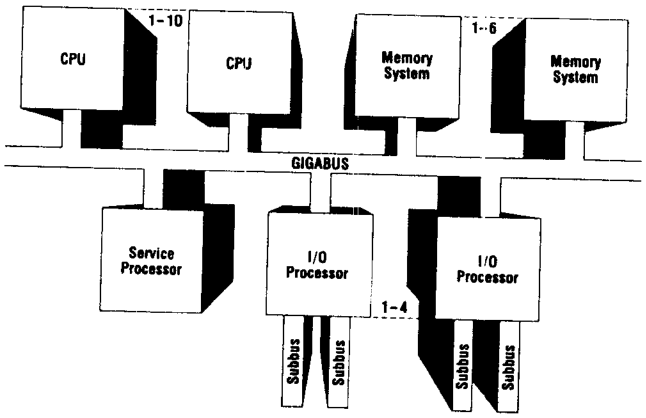


Figure 1. -- ELXSI System 6400 Block Diagram

The bus and most CPU data paths are 64 bits wide. The expansible system of one to ten processors is designed to be transparent to user software. The system behaves as if there were an independent server for each of multiple queues of processes ready to run. Any combination of seven CPU's and I/O processors plus 92 megabytes of main memory fits in a single standard cabinet. Two such cabinets may be connected to form a larger system.

The processors share main memory and I/O resources. The GIGABUS (TM) that interconnects these units can transfer 320 megabytes per second of addresses and data.

The operating system uses messages for interprocess communication. Message system primitives are part of the instruction set, so they are implemented by the microcode and hardware of each processor. The message passing instructions are available to user as well as operating system processes.

2. Arithmetic Instruction Set Architecture

The ELXSI architecture provides 16 64-bit general purpose registers (Figure 2).
These can hold integer, floating point or decimal data. Any register can be used as an index to compute a virtual address. The 32-bit virtual address space is divided into 2K-byte pages. Instructions and data are byte-addressable.

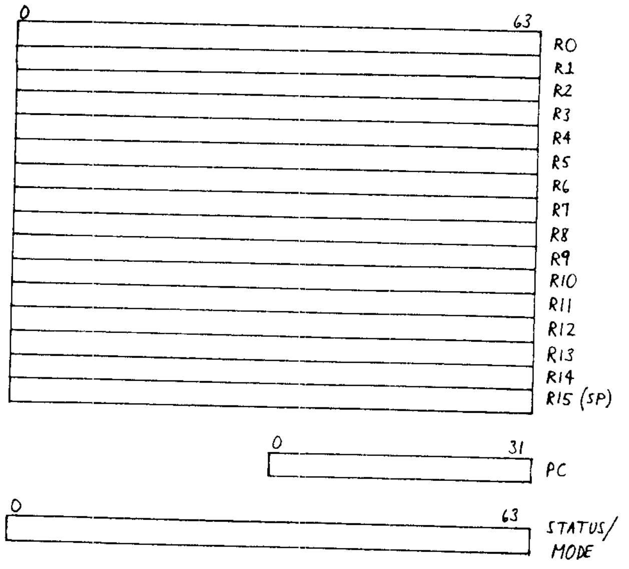


Figure 2. -- Visible Process State

* This report was prepared on equipment supported by the U. S. Department of Energy under contract DE-AT03-76SF00034, project agreement DE-AS03-79ER10358.

Instructions specify either two or three operands. The three operand addressing modes are of the form:

R1 ← R2 op R3,
R1 ← R2 op Immediate, or
R1 ← R2 op Memory.

Memory addresses may be formed with up to two index registers plus a 32-bit displacement. For unary operations, the R2 specifier is ignored. The destination is a register for all instructions except stores. The two operand address modes are special cases of the three operand ones, intended to increase code compaction.

2.1. Floating Point

Figure 3 shows the format of ELXSI single, double and extended precision floating point data types. These data types define 32, 64 and 80 bit numbers, respectively. ELXSI extended precision uses 128 bits of storage, although only 80 bits participate in arithmetic operations. Extended precision numbers use register pairs when loaded from memory. The explicit leading bit of an extended precision fraction is a holdover from earlier drafts of the proposed floating point standard. Arithmetic on unnormalized fractions is not supported, except for denormalized numbers with the minimum-valued exponent.

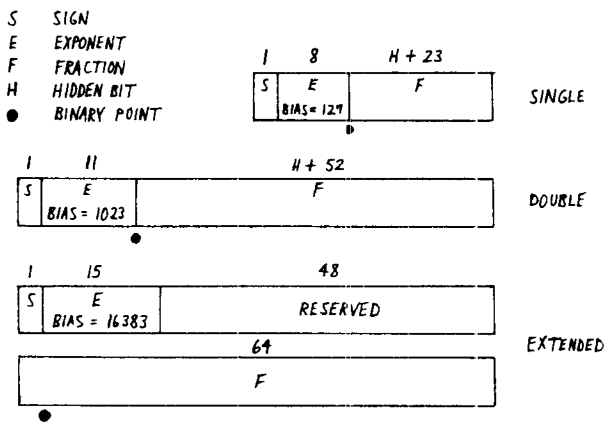


Figure 3. -- Floating Point Data Types

The floating point precisions have three different exponent ranges as well as fraction widths. Moving to a wider format increases both range and precision, unlike most prior architectures with more than one floating point data type.

Three kinds of floating point numbers and two symbols are encoded in each data type. Zeroes, denormalized numbers and normalized numbers are encoded in sign-magnitude form. The denormalized numbers have the minimum-valued exponent and are the default response to underflow. The special symbols in the architecture are plus infinity, minus infinity and Not-a-Number. A NaN is the default response to an invalid or undefined operation.

FADD	32, 64, 80	add
FSUB, FSUBR		subtract (reverse)
FMUL		multiply
FDIV, FDIVR		divide (reverse)
FCMP, FCMPX		compare (exception if unordered)
FSQR		square root
FINP		integer part in FP format
FREM		remainder (argument reduction)
CVTSD, CVTSE		convert single
CVTDS, CVTDE		convert double
CVTES, CVTED		convert extended
CVTSI, CVTDI, CVTEI		convert FP to integer
CVTIS, CVTID, CVTIE		convert integer to FP
READ.STAT		status/mode word to reg
WRITE.STAT		reg to status/mode word

Figure 4. -- Floating Point Instructions

Figure 4 shows the floating point instruction set. There are distinct opcodes for each operation in each of the three precisions. Single and double precision operations can be used with any addressing mode, while extended precision operations are restricted to the three register form. Floating point loads and stores use the normal integer and logical instructions. Two loads or two stores are required to move an extended precision number. Floating point negation and absolute value operations are performed with logical XOR, logical AND, or shift instructions.

The ELXSI floating point architecture differs in structure from the Intel architecture [4] (which also embodies the proposed standard), although the numerical results are identical. One difference is that the ELXSI integer and floating point registers coincide, while Intel's are distinct. A second difference is that ELXSI has separate opcodes for each precision. Consequently, there are instructions for conversion between floating point precisions. Intel's registers hold all numbers in extended precision. Conversions to and from extended precision are implicit in the floating point load and store instructions.

ELXSI's compare and remainder (argument reduction) instructions have extensions beyond the requirements of the proposed standard. Floating point comparison is based on one of 32 predicates. The comparison may result in a branch, an exception or no action. The predicates are all combinations of five bits. Four of these match the four possible relations between two floating point numbers: less than, equal, greater than, or unordered. If the fifth bit is set, then an Invalid Operation Exception occurs if the relation between the operands is unordered. This bit can be used to prevent the proposed standard's Not-A-Number symbols from confusing a program's control flow. By allowing all possible predicates, the floating point compare instructions give the programmer and the compiler writer complete flexibility across languages and when porting old programs.

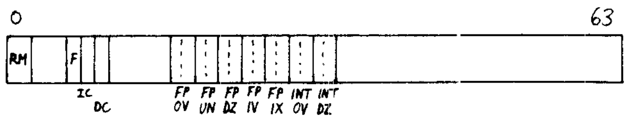
The floating point remainder instruction takes two floating point arguments and delivers two results. One result is the last 64 bits of the rounded two's complement integer quotient. This is not required by the floating point standard, but is useful for both periodic functions and others such as exponential. The second result is the remainder expressed in floating point format and ranging in value from -½ abs(divisor) to +½ abs(divisor). The usual relation

dividend = (full-length quotient * divisor) + remainder

holds, although not all of the quotient is delivered if it is longer than 64 bits.

2.2. Status/Mode Word

The process status word contains all of the mode bits and status flags that the user can modify and test (Figure 5).



RM	rounding mode
F	flush to zero on untrapped underflows
IC	integer carry bit
DC	decimal carry bit
FP OV	floating point overflow (trap enable, flag) pair
FP UN	floating point underflow (trap enable, flag) pair
FP DZ	floating point divide by zero (trap enable, flag) pair
FP IV	floating point invalid operation (trap enable, flag) pair
FP IX	floating point inexact result (trap enable, flag) pair
INT OV	integer overflow (trap enable, flag) pair
INT DZ	integer divide by zero (trap enable, flag) pair

Figure 5. -- Process Status/Mode Word

Seven mode bits are required by the proposed floating point standard for implementations that allow exceptions to be trapped. Two bits select one of four rounding modes: to nearest even, chopped, toward plus infinity and toward minus infinity. Five trap enable bits allow the user to select trapping or a default result for the following floating point exceptions: underflow, overflow, divide by zero, invalid operation and inexact result. Trapping means that control passes to a procedure supplied by the user for the particular exception, if there is one. Otherwise, control passes to the system trap handler. Each trap enable bit is paired with a flag bit that is set if an exception occurs with its trap masked off. In this case, the computation continues after the default result is delivered to the instruction's destination.

There are no condition code bits in the status word because ELXSI compare and branch instructions combine both steps into one operation.

The flush to zero mode is an extension to the standard that allows the user to prohibit the formation of denormalized numbers on underflow. This may make it simpler to port programs written for machines that always flush to zero on underflow. Flush to zero is relevant only if the underflow trap is masked off.

2.3. Integer

ELXSI provides 16-bit, 32-bit and 64-bit two's complement integers and the arithmetic operations shown in Figure 6. Two pairs of bits in the status/mode word correspond to the integer overflow and integer divide by zero exceptions. The trap enable bit and the occurrence flag in each pair allow the user to control integer exceptions in the same manner as floating point exceptions.

The integer instructions are defined to support multiple precision arithmetic well. In multiple precision addition and subtraction, all segments except the leftmost use instructions that generate a carry bit, but do not check for overflow. The leftmost segment uses a regular add or subtract operation to check for overflow and clear the carry bit. Thus single length arithmetic is a special case of multiple precision arithmetic. Only the leftmost segment differentiates between magnitude and signed numbers. The integer carry bit is available in the status word for the user to manipulate, although the usual sequences of instructions would make this unnecessary. Multiple precision multiplication is supported by instructions which deliver the 128-bit product of signed or unsigned numbers.

ADD	16	add
SUB, SUBR	32	subtract (reverse)
MUL	64	multiply
DIV, DIVR		divide (reverse)
REM, REMR		remainder (reverse)
CMP, CMPU		compare (unsigned)
ADDUC		add and generate carry
SUBUC, SUBUCR		subtract and generate carry
NEG		negate
LD	8	load sign extended
LDZ	16	load zero extended
ST	32	store
STV	64	store with overflow check
STI, STIN		store immediate (negative)
ASCII.ADD		add
ASCII.ADDC		add and generate carry
ASCII.SUB		subtract
ASCII.SUBC		subtract and generate carry
CVT.AI		convert ascii to binary integer
CVT.IA		convert binary integer to ascii

Figure 6 -- Integer and Decimal Instructions

2.4. Decimal

Decimal arithmetic is defined for digits encoded i ASCII. There is no packed decimal format. A registe. holds eight decimal digits representing a positive integer or zero. The six decimal instructions are "add", "add and generate carry", "subtract", "subtract and generate carry", "convert decimal to binary" and "convert binary to decimal". The carry instructions are for multiple precision arithmetic. The decimal carry bit in the status register is separate from the binary integer carry bit.

Decimal multiplication and division are carried out through conversion to binary. This technique is efficient because a small amount of hardware makes conversion between binary and decimal extremely fast.

3. 6400 Arithmetic Processor (AP)

The arithmetic processor performs all floating point, integer and decimal arithmetic. It fits on a single board, yet adds double precision floating point numbers in 150 nanoseconds and multiplies them in 300 nanoseconds. Other register to register instruction times are given in Figure 7.

	Single	Double	Extended
FADD	150	150	300
FMUL	200	300	500
FDIV	1000	1750	2100
FSQR	1600	3050	3750

ADD	100
MUL.64	250-350
MUL.128	450
DIV	600 + (25 * #quotient bits)
ASCII.ADD	200
ASCII -> INT	500
INT -> ASCII	2500

Figure 7. -- Register to Register Execution Times (ns)

Two principles governed the AP's design. The first was to avoid slowing arithmetic on normalized floating point numbers due to checking for special cases. All such cases are detected by special hardware and then handled by microcode at whatever speed is appropriate. The detection hardware never interferes with the critical path for regular normalized numbers. In most cases, the complexity of the floating point architecture is pushed into microcode rather than hardware.

The second principle was to reuse hardware for multiple functions. One example is that the exponent box computes the number of qoutient bits to generate for integer division, floating point square root and floating point remainder operations. Another example is the two-directional barrel shifter used for prealignment and postnormalization in floating point addition/subtraction. The shifter also aligns operands in multiplication and division, and forms part of the priority encoder.

3.1. Interface to the Cache and Main CPU

The 6400 central processing unit consists of three boards: cache, main CPU, and AP (Figure 8).

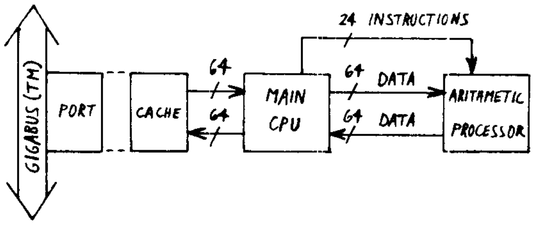


Figure 8. -- System 6400 Central Processor

The cache is 16 Kbytes, two-way set associative with a 100-ns access time. The microcode cycle time of the main CPU and the AP is 50 nanoseconds. All control store is held in RAM.

The main CPU and the AP are co-processors, each with its own copies of the register file. The two micro-engines are synchronized by branch lines and interrupts. Consistency is maintained because the main CPU controls writes to the register file copies on both boards. The main CPU contains the instruction fetch unit and handles all interactions with the cache, but the AP decodes instructions for itself. When instructions are handled by the main CPU, the AP is idle. On instructions handled by the AP, the main CPU fetches non-register operands for the AP and then waits for it to signal completion. This protocol allows execution times to be data dependent. Communication between the main CPU and the AP is fast because unusual slow cases are caught by interrupts. The microcode proceeds on the usual fast path until interrupted.

3.2. AP Block Structure

The arithmetic processor's data paths and functional units are shown in Figure 9.

Data is received from and sent to the main CPU over two unidirectional 64-bit busses. Instructions arrive over a separate 24-bit bus. The AP's major functional units are designed to accomodate the three floating point data types and integers with minimal replication of hardware and microcode. For example, the microcode for single and double precision floating point addition is identical. The same is true for floating point comparison.

The AP hardware consists of 49 Motorola MCA-1200 gate arrays, 50 1K and 4K ECL RAMs, and 600 ECL 10K series MSI packages. The gate arrays taken as a whole contain as much logic as 1000 MSI packages. There are eight gate array designs, providing the following functions: multiply, divide(2), unpack, binary ALU, exponent difference, barrel shift, and BCD add/conversion to binary.

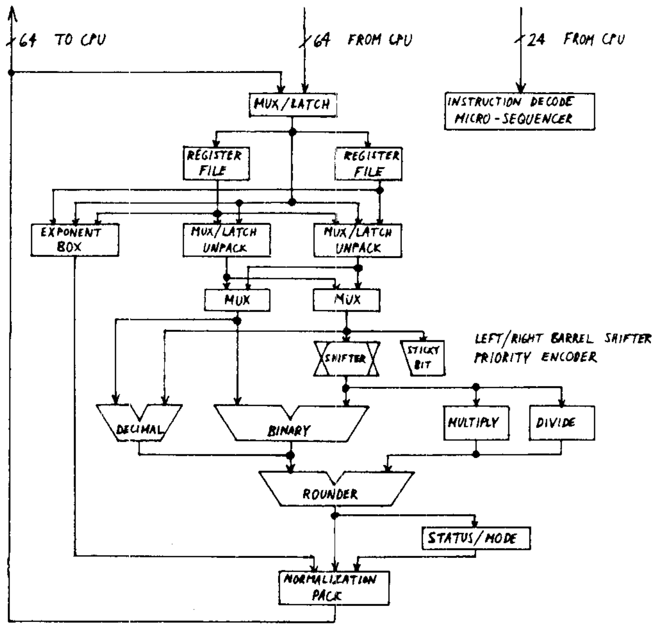


Figure 9. - Arithmetic Processor Block Diagram

3.3. Microcode

The AP microcode consists of approximately 1000 72-bit words. Figure 10 shows their distribution among instructions. Of the 800 words used for floating point arithmetic, 170 cover the cases of zero or denormalized number operands and 135 cover the cases of infinity or Not-A-Number operands. Another 75 words are used to detect exception conditions or to respond after the hardware has detected them. For the basic arithmetic operations, more than half of the microcode is devoted to the proposed floating point standard's denormalized numbers and special symbols.

		NORMAL	ZERO	DENORM	INFINITY	NAN	TOTAL
FADD, FSUB, FMUL, FDIV, FCMP, FCMPX	SGL/DBL	54	13	33	25	26	151
	EXT	48	12	46	10	7	123
FSQR, FINP, FREM	SGL/DBL	129	8	16	10	7	170
	EXT	70	4	5	6	3	88
CONVERT	FP → FP	32	9	15	6	30	92
	FP → INT	51	3	3	3	3	63
		384	49	118	60	76	687
CONVERT	INT → FP						32
DECIMAL							72
INTEGER							80
EXCEPTION HANDLING (USER VISIBLE EXCEPTIONS)	FP						74
	INT						8
OTHER							52
							1005

SINGLE/DOUBLE Floating Point	466
EXTENDED Floating Point	327
Integer	88
Decimal	72
Other	52
Total	1005

Figure 10. -- Arithmetic Processor Microcode Allocation

The micro-sequencer contains a large number of branch conditions and infrequently-used control lines encoded in "miscellaneous" microword fields. The microcode uses 30 two-way branches, 14 four-way branches and 3 eight-way branches based on 45 independent branch conditions. About 25 of these conditions are directly attributable to the proposed floating point standard.

The AP is designed so that the usual cases of the most frequent instructions run entirely under hardware control. These instructions include single add and multiply, double add and multiply, and integer multiply. The microcode follows along checking for unusual cases when the appropriate branch conditions have become valid.

3.4. Extended Precision

Although the microcode for single and double precisions often blends into just one code segment, extended precision operations are always controlled by independent microcode. The primary reason is that extended precision results occupy two registers. The AP must pack two words in succession to pass to the main CPU. It is difficult to fold extended precision microcode into that for single and double without slowing them. In most cases all microwords in the critical path already contain multi-way branches, making it expensive to add another branch bit to detect extended precision.

3.5. Functional Units

In the following sections, we describe the hardware for multiplication, division, rounding, normalization and decimal arithmetic.

3.5.1. Rounding and Normalization

To conserve hardware, the normalization unit adjusts fractions by a one bit shift to the right, no shift, or a one bit shift to the left. This allows magnitude additions, multiplications and divisions to be normalized and rounded in one pass. Most magnitude subtractions will finish in one pass, also: all of them with exponent difference greater than or equal to two, and some of them with exponent difference equal to one. Subtractions requiring postnormalizations greater than one send the fraction back around to the top for one pass through the barrel shifter.

The intermediate fraction result after magnitude subtraction is usually a negative two's complement number. The logical sequence of operations on this fraction is to complement it, normalize it and round it. Negation and rounding both require a full-length carry propagate adder. The AP uses only one adder by performing negation and rounding at the same time. Rounding occurs at one of three possible bit locations, depending on the normalization which will follow. This method is somewhat faster than the straightforward scheme because the uphill part of the adder's carry lookahead logic runs in parallel with the normalization and rounding logic. The data lines can be kept as the critical path rather than the select lines. If the normalization multiplexer were first in the data path, the select lines would be the critical path.

3.5.2. Multiplication

The multiplier accepts two's complement integers and unsigned floating point fractions as operands. This capability is built into the 8-bit by 8-bit gate array multipliers. Eight of them are connected to form a 64-bit by 8-bit slice that is clocked every 25 ns (Figure 11). The redundant partial product registers keep the time for a 64-bit carry lookahead addition out of the inner loop.

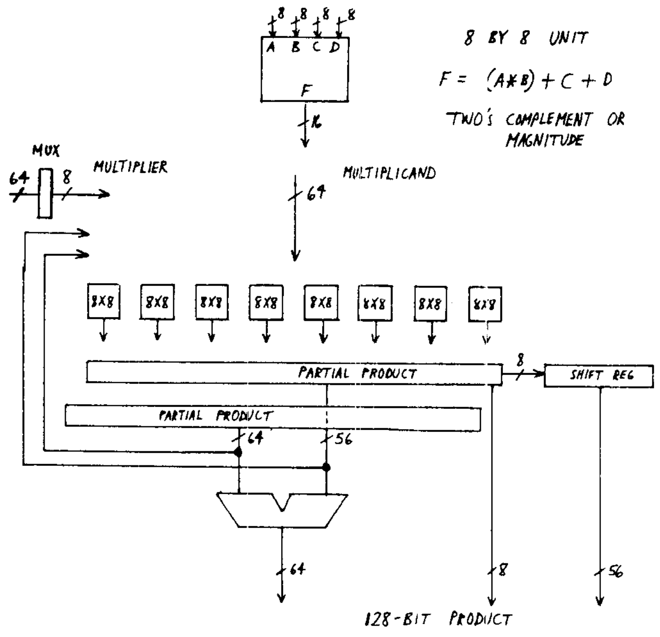


Figure 11. -- Multiplier Organization

The multiplier runs 3 steps for single precision, 7 steps for double precision and 8 steps for extended precision floating point. It runs 4 steps for integer multiplication if both operands have less than or equal to 32 significant bits. Otherwise it runs 8 steps to form the complete 128-bit product and tests for overflow. The multiplier runs under hardware rather than microcode control to allow an odd number of steps (two steps per microcode cycle). Using the exact number of steps necessary minimizes alignment shifting and saves 50 ns in single and double precision floating point

The multiplicand passes through the shifter on its way to the multiplier chips so that it can be moved left by 3 bits in double precision floating point and 32 bits in integer multiply with short operands. With this arrangement, products are correctly aligned when they leave the multiplier on the way to the rounder.

64 by 64-bit unsigned integer multiply is equivalent to the fraction multiply for extended precision, so the sign-magnitude representation for floating point imposes no additional hardware requirements beyond those of multiple precision integer arithmetic.

3.5.3. Division

The division unit is used for floating point divide, remainder and square root; integer divide and remainder; and binary to decimal conversion operations. It produces two quotient bits or one square root bit per 50 ns cycle. The design is based on the divide and square root hardware developed for a floating point accelerator at UC Berkeley [5].

3.5.4. Decimal

Decimal adds and subtracts are carried out with an eight byte BCD adder. Decimal to binary conversion uses random logic in the BCD gate array to convert 4 digits to 14 bits. The high and low order halves of an eight BCD digit number are converted in parallel. Then the top half's 14 binary bits are multiplied by 10,000 and added to the bottom half. Binary to decimal conversion depends on the divider. The 27 bit binary input is divided successively by 1000 to form sections of 7, 10 and 10 bits. Each of these is converted to three BCD digits by a RAM lookup table on the output of the division box.

4. Conclusions

The System 6400 arithmetic processor supports the proposed IEEE floating point standard with fast hardware of moderate size. The costs of the standard were confined to microcode size and a longer design cycle. The extra microcode could be a severe problem in the future, either in numbers of IC's or a slower cycle time. We do not expect to reduce extended precision's microcode cost, but we want the reduce the number of microwords devoted to handling special cases in all three floating point precisions. One approach would be to extend the architecture so that we can trap to assembly language subroutines from microcode when special cases are detected.

Much more time was spent designing hardware and writing microcode for this architecture than would have been required for an implementation of DEC VAX or IBM 370 floating point. But we expect future implementations to build on the knowledge gained this time.

The simple set of decimal arithmetic instructions has worked well in our Cobol compiler and language-independent format conversion utilities. Decimal arithmetic is extremely fast, but requires little hardware. We have provided well for multiple precision integer addition and multiplication. To improve multiple precision division, we need to add instructions to divide 128-bit dividends by 64-bit divisors in both magnitude and two's complement forms.

5. Acknowledgement

The ELXSI arithmetic processor was designed and brought to life by Steve Kosic, John Lien and the author.

6. References

[1] Announcement in Journal of the Society of Exploration Geophysicists, October, 1982. Deliveries began in March, 1983.

[2] IEEE Computer Society Microprocessor Standards Committee Task P754, "A Proposed Standard for Binary Floating Point Arithmetic, Draft 10.0," January, 1983. Draft 10.0 incorporates substantial revisions from Draft 8.0 published in Computer, 14, No. 3, March, 1981, pp. 52-63.

[3] Robert Olson, B. Kumar and Leonard Shar, "Messages and Multiprocessing in the ELXSI System 6400," Compcon Proceedings, San Francisco, March, 1983.

[4] The 8086 Family User's Manual, Numerics Supplement, Intel Corp., July, 1980.

[5] George Taylor, "Compatible Hardware for Division and Square Root," Proceedings Fifth IEEE Symposium on Computer Arithmetic, May, 1981, pp. 127-134.