R 75-286

ABSTRACT

By: James Brakefield

In the context of medium and large computer architecture, hardware design options are presented which allow software algorithms to remain invariant over a range of computer word sizes. Use of descriptors for referring to memory plays an integral role. Advantage is taken of choosing a specific data format for floating point numbers. Advantage is taken of the arithmetic characteristics of the commonly used word sizes. A variety of other hardware and software design possibilities are mentioned.

KEY WORDS:

Computer architecture, Optimal floating point format, Divide by 3 circuit, Write once memory, Barrel shifter, Wallace trees, Descriptors, String, Stack, Dequene, Vector, Symbol table, Precedence parsing.

ASPECTS OF COMPUTER ARCHITECTURE

This paper attempts to survey the subject of computer architecture. The presentation is both explanatory and exploratory. The general slant of the paper is to point out design options which make possible software compatable computer families which span a range of word sizes. Two such family architectures are presented. The subject of programming aids is also considered.

The subject of computer architecture begins with the question of data representation. All present logic families deal with two state signals, usually two voltage levels which are assigned the values 0 and 1. This leads to Boolean switching circuits. In particular, binary arithmetic is readily implemented.

The simplest binary arithmetic is on unsigned numbers, which will be called bytes in the present paper. Since computers are built of actual physical devices, a given arithmetic unit is of fixed length. Thus, if it operates on bytes, the range of unsigned numbers goes from 0 to 2^n-1 and will be said to be n bit wide byte arithmetic. Overflow and underflow can occur during byte arithmetic, and in general it will be assumed that the hardware gives some indication of this. Also it can be noted that an n bit arithmetic unit (AU) can perform arithmetic on m bit ($m < n$) numbers. If overflow and underflow would not occur on an "m" bit byte arithmetic unit, then the n bit AU will give the same answers as the m bit AU.

The next step from unsigned numbers is signed numbers. Thus, if n bits are available with 2^n possible values, then half of these values can be

assigned to negative numbers. There are currently three commonly used such assignments. They are 2's complement, 1's complement, and sign and magnitude. In general, a negative number has a "1" in its sign bit. The magnitude bits are given by the following formulas:

2's complement	$2^{n-1}-a$
1's complement	NOT a
Sign & Magnitude	a

where -a is the negative number.

Generally, the sign bit is a high order bit (next to the 2^{n-1} bit). The one benefit of this location is ease of conversion to and from unsigned numbers. Later we will see reasons for placing the sign next to the 2^0 bit and for choosing the sign and magnitude format over 1's and 2's complement.

Corresponding to binary signed and unsigned numbers, 1's and 2's complement there can be decimal signed and unsigned numbers, 9's and 10's complement. These representations usually allocate four bits for each decimal digit and 1 to 4 bits for the sign. Strangely enough, only sign and magnitude is commonly used for signed decimal numbers and usually the sign is on the low-order end of the number. This is because decimal arithmetic is usually done serially right to left. Examining the signs of the operands first simplifies addition and subtraction. Also truncation of a decimal integer (a common operation) is much simpler if the sign is at the low-order end.

Another representation of decimal numbers is possible (used on B6700 and B7700). It is to represent the number as a binary number with an implicit

decimal scale factor, i.e., $10^{b}\Sigma a_i 2^{i}$ where b is known from context. Typically, b = -2 in business and commerce. As in binary floating point, arithmetic operations will at times require adjustment of the scale factors of the operands or result. For the binary representation of decimal numbers this requires either multiplication by a power of 10 or a division by a power of 10.

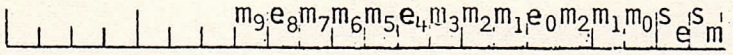
Since the scale factors of the operands and result are implicitly known, the correct power of 10 will also be known. Division by a power of 10 can be done several ways. First is multiplication by the reciprocal, second is binary division, and third, special purpose circuits can be built. Various cost/performance trade offs can be used to select one of the three division techniques. It will be noted that a divide by 10 technique chosen can be also used for binary to decimal conversion such as occurs during output formating.

When the scale factor of an integer varies such that the programmer can not or does not care to keep track of it, a third numeric representation is used. For hardware minimization and performance both the number and the scale factor are encoded in binary. Both are handled simultaneously by the memory (in medium and large sized CPU's) and usually the AU as well. The representation has a fixed size (in bits) and a fixed format. Either 1's complement, 2's complement, or sign and magnitude can be used for either exponent (e) or mantissa. A fourth format is also used for the exponent which goes by the name "biased exponent". Here the exponent is a byte which

is biased by some fixed negative number. The fixed negative number could be held in a register, but so far it has been wired into the circuits.

In general, a floating point number has the value $M \cdot a^{e \pm k}$. Although M and e are sometimes expressed as decimal numbers (4 bits per digit) inside the computer, most all current and high performance computers express M and e as binary numbers. 'a' is usually chosen to be 2, 8 or 16. The use of 8 and 16 for 'a' waste .415 and 1.0 bits respectively, but allow faster normalization. (To preserve accuracy, 'a' is usually kept as large as the hardware will allow. This normalization is done by left-shifting 'a' and reducing 'e' accordingly).

The choice of the floating point format is dictated by several factors. One is the cost of the hardware and/or its speed. Second is the ease of conversion between integer and floating point. Third is the ease of non-arithmetic operations. Included in the last category is truncation. One of the dominate costs of a computer is its memory. Any reduction in memory requirements has a direct dollar value. Usually the arithmetic unit is chosen large enough (in bit width) to efficiently perform the most frequent operations. If the cost (CPU time or extra hardware) of packing and unpacking operations is low enough, the programmer will use m bit ($m \leq n$) data in memory to save on memory required. When this consideration is dominate, there is an optimum floating point format from a hardware standpoint. The resulting format has a bit reversed mantissa and an interleaved exponent. Both mantissa and exponent are sign and magnitude with sign on the right.



$$N = \pm(\Sigma 2^{-i} m_i) * 2\uparrow\pm k \pm \Sigma 2^{j} e_j$$

Every fourth bit is an exponent bit, but since exponents seldom need to be over 10 bits, this interleaving can stop after e_9. As an example of floating point numbers with a small number of bits, the IBM System 7 has an auto ranging A to D converter. A few wiring changes would result in a 16 bit number of the above format. The format's chief advantages are the exponent limit increases with increasing number of mantissa bits and that truncation and extension are simple and fast. No shifting or sign extension is required. The same sign extension avoidance applies to sign and magnitude representation of integers when the sign is on the low-order end of a number.

Often the arithmetic unit will also perform double precision operations. There is sometimes use for even more precision. To facilitate these forms of arithmetic, there is a category of extra precision considerations. For bytes the add or subtract overflow (carry) operations are useful. Otherwise a test and conditional increment or decrement needs to be performed. For multiple precision multiply, a "multiply and add" (a*b+c→c) instruction saves time and instructions. For integer arithmetic the overflow can be either an increment or decrement of the operand. For signed divide it is convenient if the dividend can consist of two single precision numbers with differing signs. (If A and B are the two parts of the quotient, then the value of quotient = $A \cdot 2^{n-1} + B$ where A and B may differ in sign.) For multiple precision floating point, it is useful to be able to gain access to the residue of a single precision floating point operation. (Where the residue = difference true value "A op B" and "A op B" as performed by AU.)

A byte one bit long can represent a Boolean value, True (1) or False (0) and also being a byte can be used in arithmetic: If A, Then B, Else C can become A * B + (1-A) * C. A byte 6 to 8 bits long is often used to represent a character since most character sets have from 48 to 96 characters. It is common to use organized collections of bytes, integers, and floating point numbers in programs. The most common organizations are as strings, vectors, stacks, and dequenes of similar elements and as records of dissimilar elements. (Elements are similar if all are of the same type (byte, integer, floating point) and same size (in bits). Further, these organizations may be compounded so one can have a string of records. A useful organization which will be slighted is the list as in LISP. For this paper, a list is composed from a string of records, records which in the LISP case consist of two pointers or indexes to other records.

To parameterize these organizations or structures, it is convenient to have hardware assistance. One can compose formats which contain the parameters for the string, vector, stack, dequene, or record.

String: the elements are contiguous in either an ascending or descending direction. It is sufficient to specify the type (1 of 4), size (usually 1-64), starting address, length, and direction (1 of 2). For bytes, integers, and floating points, the size is in bits and for records the size is in words.

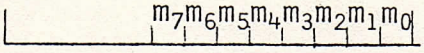
Vector: string but allowing a separation between elements (called Δ).

Stack: Vector but in addition, a limit is specified for the count field (i.e., a partially filled vector).

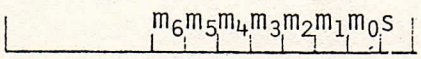
Dequene: type, size, Δ, address, index to first or top element, count, and limit.

Since the dequene includes the others as special cases, it is the most general. The rational for the other forms is to save memory or time: the dequene is often used as a circular buffer, the vector as a one-dimensional array, the stack as a push down list, and the string as a special case of a vector, often for character or bit strings. Records are difficult to parameterize in general. It is usually adequate to specify the size in words and to select sub-fields by offset, size, and type.

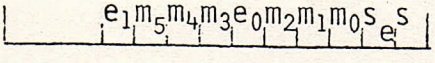
To specify a sub-field or a single byte, integer, float, or record, an element descriptor is used (contains address, size, and type). There is one more data form to be considered. It is the linear sequence. A parameterization is starting value, increment, and count. This is similar in form to the vector descriptor.

Byte 

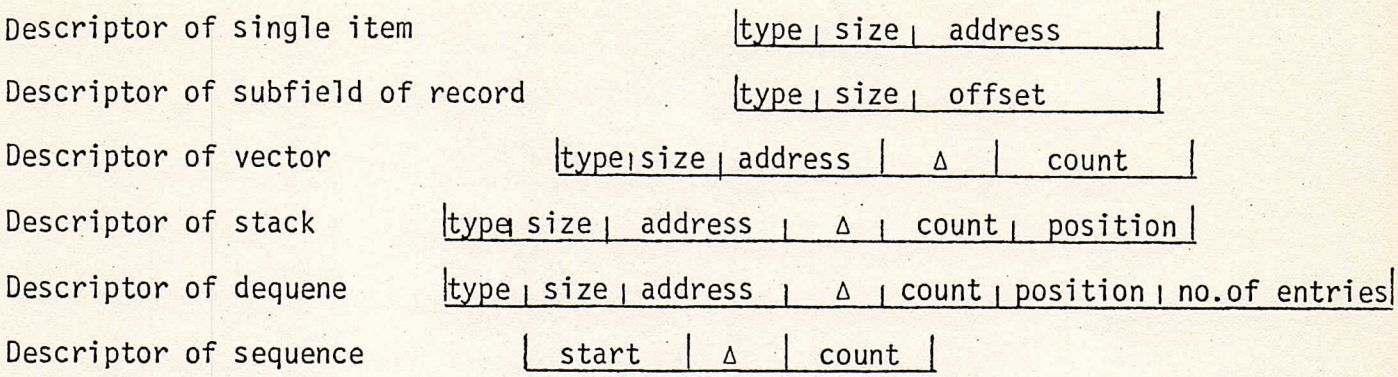
$$\text{value} = \sum_{i=0}^{n-1} 2^i m_i$$

Integer 

$$\text{value} = (2s-1) \sum_{i=o}^{n-2} m_i 2^i$$

Float 

$$\text{value} = (2s-1) \sum_{i=0}^{3(\frac{n-2}{4})} m_i 2^i \cdot 2^{(2s_e-1)\sum_{j=0}^{\frac{n-2}{4}} e_j 2^j}$$



In the future, two dimensional memories may be available (MOS storage tube), in which case it would be convenient to allow strings to have any of eight directions.

Currently, there is a variety of memory word sizes available, and if one chooses to match memory word size to AU word size, there is a continuing rational for these various sizes. Consider word sizes 32, 36, 42, 48, 56, 60, and 64. Each allows the word to be divided up in various ways. Those divisible by 6, allow 6 bit characters to be packed nicely, those divisible by 7, and 8 similarly. However, keeping track of which character to reference is not so nice. If a simple scan of a character string is desired, an accumulator is loaded with a word from memory, the characters are shifted out one at a time, when the accumulator is empty, the next word is gotten. If one moves about in a character string, one would like to index the characters from the front or back of the string. If there are other than 2^x characters per word, this implies a divide for a remainder (used to select proper character within word) and quotient (word index). Division is usually slow and

expensive to speed up. However, in this case the division is by a small constant, with respect to a given word size. This can be used to advantage to provide a low cost high speed division circuit. The circuits I am aware of are: multiplication by reciprocal using simplified Wallace trees, serially connected ROMs, serially connected n-rail permutation networds (made from multiplexers). These three techniques will perform the division, producing a remainder in 1-3 clock times. This is comparable to the time for indexing (fetch index register and add to address or Δ).

Besides choosing the word size, one can choose whether to have arbitrary bit sizes (up to some limit, say 64) or whether to have only submultiples of the basic word size. The submultiples are:

32 -- 1, 2, 4, 8, 16, 32

36 -- 1, 2, 3, 4, 6, 9, 12, 18, 36

42 -- 1, 2, 3, 6, 7, 14, 21, 42

48 -- 1, 2, 3, 4, 6, 8, 12, 16, 24, 48

56 -- 1, 2, 4, 7, 8, 14, 28, 56

60 -- 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30, 60

64 -- 1, 2, 4, 8, 16, 32, 64

If one is content with submultiples, the size and address fields of a descriptor can be combined in such a way to save a few bits. The address is examined low order to high order looking for the first "1" bit. The rest of the address gives an address, the location of the first "1" gives the size.

32	(see * below)		56	xxxxxl1	bit
36	xxxxx11	bit adr		xxxx101	2 bit
	xxxx101	2 bit		xxx1001	4 bit
	xxx1001	4 bit		xx10001	8 bit
	xxxx110	3 bit		xxxxx10	7 bit
	xxx1010	6 bit		xxxx100	14 bit
	xx10010	12 bit		xxx1000	28 bit
	xxxx100	9 bit		xx10000	56 bit
	xxx1000	18 bit	60	xxxxx11	bit
	xx10000	36 bit		xxxx101	2 bit
42	xxxxx11	bit		xxx1001	4 bit
	xxxx101	2 bit		xxxx110	3 bit
	xxxx110	3 bit		xxx1010	6 bit
	xxx1010	6 bit		xx10010	12 bit
	xxx1100	7 bit		xxx1100	5 bit
	xx10100	14 bit		xx10100	10 bit
	xxx1000	21 bit		x100100	20 bit
	xx10000	42 bit		xxx1000	15 bit
48	xxxxx11	bit		xx10000	30 bit
	xxxx101	2 bit		x100000	60 bit
	xxx1001	4 bit	*32 & 64	xxxxxxx1	bit
	xx10001	8 bit		xxxxxx10	2 bit
	x100001	16 bit		xxxxx100	digit
	xxxxx10	3 bit		xxxx1000	"byte"
	xxxx100	6 bit		xxx10000	16 bit
	xxx1000	12 bit		xx100000	32 bit
	xx10000	21 bit		x1000000	64 bit
	x100000	48 bit			

For the 32 bit word case, this reduction in available sizes increases possible memory space by factor of 32 or allows the five extra bits to be used

for something else. This technique also allows records to have any power of two size up to and including the entire memory space.

The more common operations on descriptors can be classed as using the descriptor to reference memory and as modifying the descriptor. Strings and vectors can be indexed and referenced element accessed. Push and Pull can be done on stacks and dequenes in addition to indexing. A useful operation on strings and vectors is reading the first element and shortening the string or vector. If part of a loop, a suitable branch condition can be used to terminate the loop when the string or vector is exhausted. Reversing a vector can be done by modifying adr to point to end of vector and changing the sign of Δ. A subfield specification and descriptor of a vector of records can be combined to yield a descriptor of a vector of a subfield. A vector descriptor with two sets of Δ and count can be considered as describing a matrix.

Sets may be represented in two ways, one as a bit string, the other as an increasing list of indices to the "one's" in a corresponding bit string. One representation may be preferable to the other in a given situation. If the AU provides the set operations on bit strings ($\wedge$, V, not, $\oplus$), then it is convenient if the AU also provides them on index lists, as well as a way of going between the two representations.

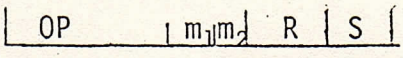
Programming a computer with all of the above features, is fairly straightforward, especially if a ALGOL recursion stack (hereafter called "the stack")

is provided by the hardware. References for simple values refer to values in the stack, references to vectors, etc. refer to a descriptor in the stack which holds the indirect address. If the stack values are self-identifying, then one codes operation codes and stack references. This results in very compact code. There are about 16 common operations, and usually 16-1000 different stack locations. In case the stack values are not self-identifying then this identification must be included in the stack reference or op code fields. Since everything outside of the stack is reached through descriptors with type fields, the non-stack memory need not be self-identifying. This is just as well. The stack is usually used on a word basis, whereas the rest of memory often has several items packed per word and self-identification would be difficult.

If a recursion stack is not provided, one can have 1, 2, or 3 address instructions. With the ease of including general purpose registers that exists with current IC technology, a 2 or 3 address format with addresses referring to registers is convenient. The stack type architecture is preferable, but smaller word sizes call for simpler CPU's which tends to rule out a recursion stack, and descriptors, and floating point. For these situations, the two register format with registers being used as operands or degenerate descriptors is suitable.

The following describes a register organized CPU for 16-24 bit word sizes and a stack organized CPU for 32-64 bit word sizes:

A register oriented architecture along the lines of the PDP11. 64 16 bit registers (possibly more in the 18 and 24 bit versions) with four groups, an address group, an operand group, and two mode groups. A 4 bit operation code and two operands with 2 bits to select the group and 4 bits to select a register within the group. Specifying an address group or data group uses a register within the group as is. Register 0, address group, implies direct addressing using a word following the instruction as a direct address. Register 0, data group, implies an immediate operand following the instruction. Register 0, mode group 1, applies the mode to a following immediate operand. Register 0, mode group 2, applies the mode to the data at a following direct address. The 2 operand instructions are: Move, Move and Clear, Swap, Compare, Max-Swap, Add, Subtract, Multiply, Divide, And, Or, Exclusive-or, Branch and Save, Move Mode, and Shift. The single operand instructions are: Clear, Set, Complement, Negative, Absolute Value, Test (compare with zero), Increment, Decrement, Add Carry, Subtract Carry, Pop Count, Leading Zeros Count, Times Ten, Divide by Ten, Divide by Hundred, and Execute. Conditional branches are relative using a 8 bit 2s' complement immediate operand. The conditionals are: Underflow, Less than Zero, Equat to Zero, Greater than Zero, Overflow, Even, and the Not's of the preceeding conditions.



2 operand

| 15 | 3 | m_2 | OP | S | 1 operand

| 15 | 0-11 | ±Δ | relative branches

Shift m_1 = ox left shift
1x right shift
xo 4 bit immediate
x1 data register holds shift count

Move Mode m_1 = 0x $m_{x,r} \leftarrow S$ (load mode)
= 1x m_2 = 0y $m_{x,r} \leftarrow m_{y,s}$ (Mode to mode move)
m_2 = 1y $R \leftarrow m_{y,s}$ (store mode - either in address or data register)

| type | size | stk stg | right left | chrg no cl | before after | index | offset | mode | register # |

The mode registers contain bit fields for the following: size of operand, type, stack or string, right or left stack or string, change or no change address, index or no index, offset or no offset, register number for index or offset, and mode of register number so the mode process can be cascaded.

The Max-Swap instruction swaps the operands if the first operand is not the largest. Useful in sorting and merging operations as well as finding the maximum element of a vector.

55XX

A stack oriented architecture along the lines of B5500 or MUV.A 1k stack operating from both ends. Left end holds operands, descriptors, subroutine paramenters and working space. Right end holds return addresses and level

pointers. Top of left stack is the accumulator. Top of right stack can be used as a loop counter. Nominally, 32 to 64 bit word sizes, ½ million or more word address space. Short instructions specify operation to take place between top of left stack and a stack location. Long instructions specify operation, mode, and two stack locations. One stack location specifies a descriptor and the second either an index or offset. An intermediate length instruction is possible which does without an index or offset. Examples of intermediates are push or pull from a stack or dequene, or taking first element from a vector or string and shortening the string or vector. Operation classes:

Byte arithmetic
Integer arithmetic
Floating point arithmetic
Boolean arithmetic
Relative branches
Stack manipulation
Descriptor manipulation
Subroutining
Vector arithmetic

Stack references are as in B6700-B7700 computers except that right stack can also be referenced. Intermediate length instruction modes are: Stack, Dequene front, Dequene back, Access and Shorten String, Access and Shorten Vector, Indirect and Increment Address by Size, Indirect and Decrement address by Size, Indirect, and Immediate. Long length instruction modes are: Indexed, Indexed Immediate, Subfield Selection, and Subfield Selection from Stack Area. These four modes to apply to the several kinds of descriptors.

A memory organization used to simulate associative memories is the orthogonal memory. It may be accessed on either a word slice or bit slice or several other variations in between. Such a memory requires a permutation network which is similar to a barrel shifter. ~~SARTRAN~~ STARAN has such a memory. A rough component count gives the following for K orthogonal memory arrays, for a barrel shifter, a permutation network and for an ALU:

mem: K x N	2^n bit RAMs $N = 2^n$	K x N IC's
shift: N x n	2 to 1 multiplexors	n/4 x N IC's
permute: N x n	2 to 1 multiplexors	n/4 x N IC's
ALU: N	ALU bit slice	1/2 x N IC's

The considerable expense for the shift and permute network can be balanced by increasing K. Also ~~SARTRAN~~ STARAN type computers can make fairly good use of the large ALU. If one counts bits added per memory cycle, the ~~SARTRAN~~ STARAN machine will come out ahead of Von Neuman type CPU's when N is sufficiently large and if the problem to be programmed has sufficient parallelism. As an example, N can be 4096 and cycle time ∿500 ns. This gives ∿8 bits/ns, which is comparable to the fastest pipeline CPU's.

In general, computers are not as useful without software aids. These aids take the form of operating systems, compilers, assemblers, data base systems, and I/O utilities. The hardware designer can examine these "aids" to find features which should be included in the hardware. One can also examine the range of aids to find ways of achieving more commonality.

Virtual memory allows the programmer to avoid I/O operations between

primary and secondary storage. The read only, execute only, write only memory access modes allow some memory protection. A variant of write only memory is write once memory. Memory can be written only if it is initially clear. Once "written" it becomes non-zero and cannot be changed. On computers with partial word operations the zero/non-zero condition would be that of the partial word and not the whole word. This would make it handy to fill in packed records piecemeal or to provide audit trails. The mode of operation becomes to allocate a write once file or memory segment. One writes the descriptors first and then accesses the remainder of the file through these descriptors. The only caution is not to overlap subfields of a packed record. Presumably hardware could be provided which would have a suitable descipline which would prohibit overlapping fields. One such discipline is to allocate space in the file and in records on an incremental basis (low memory locations get allocated first). Record descriptors and subfield descriptors would be labeled so a given subfield descriptor is only used with a single record descriptor.

The read and clear instruction is useful for implementing semaphores as well as other initialization purposes. The advantage of a read and clear over a test and set instruction is that it is easier to check for zero rather than a specific non-zero code. Also, the read and clear can be applied to any size field (word size down to bit size). The convention with the read and clear semaphore is that if it is zero some other process "has" the semaphore. The read and clear will get the semaphore

if it is available. To return the semaphore, some non-zero code is posted (stored) in the field.

Two commonalities of operating systems, compilers, assemblers, and data bases are use of a symbol table and use of a grammar. It would be beneficial if a given computer system provided a general purpose symbol table facility. In particular, it would be beneficial to have only one symbol table per computer system. This requires a graph structured table.

Loops in the graph are undesirable, thus the graph can be restricted to a tree. A useful extension is the Octapus system at Livermore, which allows nodes of the tree to be parts of other trees. This allows one to link the existing symbol table entries in any order. In general, exact specification of a given symbol table entry involves naming the nodes and reaching an entry by a node name sequence-unrequired or implicit node names may be omitted.

Although semantics are a long way from standardization, syntax standardization is possible. The prefered syntax is the precedence variety or a multilevel precedence variety. Extension of the syntax involves defining or redefining the left and right handed priorities of a synbol. Since the priorities are the only parsing information required and since they can be put in the symbol table, parsing becomes very straightforward. A generalization of left and right precedence is above and below precedence as well as right and left which allows two-dimensional parses.

An even further generalization is graph parsing where node has a given precedence to each of the other nodes it is connected to.

Although a precedence parse always results in a parse tree, errors in the input or other considerations result in modes of the parse tree not being in standard form, either operands or operators may be missing. The missing operator situation can be corrected by a fill-in operator. The missing operand possibility can be treated by converting the operator to one requiring fewer operands. Many operators have two operand and one operand meanings. The zero operand meaning, presumably, is to treat the operator as a value and not as an operator. The reduced operand meanings can be essentially error reports. The fill-in operator can be defined or redefined to also be an error report. In this way, syntax errors can be handled within the precedence parse technique.

Several of the existing compilers have pioneering features. BALM is a LISP type language with a precedence parse syntax. As such, it has LISP power and precedence parse extensibility. PASCAL attempts to provide a human-engineered programming language. It is interesting for the features which it has omitted as well as those included. APL is an older language originally designed as a mathematical notation. Its forte is a set of vector and array operations. Although the notation or set of vector operations provided may not be optimum, it results in shorter programs.

In summary, from the hardware standpoint, the process of extending and contracting memory operands as they flow to and from the AU requires that the barrel shifter be somewhat separated from the AU and placed in the path between AU and memory. At the same time, the need for a barrel shifter in the AU is reduced to floating point normalization. Data packing and unpacking is handled implicitly by descriptor specification and by the ability to bit or character address memory. The collection of shift instructions found in most medium and large computers can be relegated to bad programming practice. The specification of operands by descriptors and the manipulation of operands by instructions results in hardware more suited to "structured programming". The ability to provide similar discriptor and instruction mechanisms across a range of word sizes provides software compatability across the same range of word sizes. The importance of an optimal hardware floating point format is that it provides some form of argument for choosing a standard format across this range of word sizes.

This, in a nutshell, is the way I see the future of computer architecture.