THE DESIGN RATIONALÉ OF THE EAI 8400 SCIENTIFIC COMPUTING SYSTEM

NOTICE: REPRINTED WITH PERMISSION OF EAI

BY: D. SINNOTT
OCT. 1964

INTRODUCTION

The design rationale employed in configurating the EAI 8400 digital computing system will be described. The characteristics of this system represent a significant step in the evolution of computer design. In contrast with the approach of establishing scaled down versions of the more prominent large scale systems, the EAI 8400 has been configured to expressly meet the demands of scientific fields of applications. These fields include the general purpose scientific, simulation, industrial control and other real-time and integrated systems.

The active design phase of the EAI 8400 was initiated in 1961, based on a fundamental tenet of EAI management:

> "EAI's primary business is in the simulation and scientific computation field and the Company's future growth depends in major measure upon its ability to serve well the needs of this market."

The background for initiating the EAI 8400 development goes back to 1957 when work was begun on digital approaches to simulation analysis. The early work was performed on the Datatron 205 which was installed and operated at the EAI Princeton Computation Center. Following that effort were a number of studies on the use of Digital Differential Analyzers (DDA) both serial and parallel for simulation. The parallel approach was extended to a large scale parallel-parallel system to be employed in the Pacific Missile Range for computations at better than 20 times real time. Still other efforts were expended in analog/digital linkage systems and EAI's revolutionary parallel logic Digital Operating System. Many of these systems have been installed and operated in advanced simulation facilities, solving problems which up until

that time were impossible to solve on either the analog or digital computer alone.

Significant effort has also been expended in establishing a digital computer for the efficient processing of matrix and vector operations with particular emphasis on linear and non-linear programming and other optimization algorithms.

Coupled with each of these developments has been the groundwork of scientific computation itself--the development and utilization of computational procedures and processes. Without this effort by the various EAI computation centers, and the System Analysis and Advanced Study Groups of the Research and Computation Division, a digital computer development would be reduced to the basic "nuts and bolts" design. The overall background of this group at the starting point <u>and</u> their continuous input through the course of the development cycle permitted the establishment of a digital computer system which would truly meet the demands of the intended fields of application.

Basic Design Premise

The fundamental reason for the existance of a computational tool is to produce results from input stimuli. Therefore, the only applicable measure of performance in a realistic world is the ratio:

Computational Throughput/Total Cost

where: <u>Computational Throughput</u> is measured from the time of problem formulation to the point where results are generated with appropriate forms of documentation.

--and--

<u>Total Cost</u> includes the level of effort expended in translating the problem to an exact and acceptable machine-dependent language, in addition to the initial hardware system investment and attendant maintenance costs.

To maximize this ratio means providing an overall computation system built upon the following design goals:

1. Economical high speed processing
2. Ease of programming

Casting each of these goals in the light of an everyday problem we start with a "user with a particular problem formulation." This problem must be coded into some intermediate language, and the language must be processed. Then the iterative cycle of debugging, running and program modifications must be executed. This phase may require many passes before the ultimate goal of obtaining results is achieved. Therefore, it can be easily seen that the main criteria for establishing a computing tool is not the glamour and elegance of the device itself but rather what a user needs to accomplish his ends. Consequently it was recognized early in the game that the centroid of design was the user--what does he need to accomplish his task and how does he want to use it?--In more terse terms "the dog wagging the tail and not the tail wagging the dog."

In reviewing the workload of the everyday problem described above it is noted that there are two separate portions; ON-LINE operations where the user is processing the intermediate languages, debugging and running the program and the OFF-LINE operations including the initial coding and any program modifications.

Economical High Speed Processing

In the ON-LINE portion of a typical workload, a number of factors are involved. Initially, a given system must have sufficient capacity to handle the program. If not, the only recourse is to partition the problem into a number of segments and run each sequentially with appropriate program linkages. Obviously this criteria can be satisfied by having available a

high capacity system. However with due regard to economic factors this may be prohibitive. Therefore, it is necessary to squeeze as much efficiency as possible out of each system element, in particular the memory.

Secondly, in terms of running a program, the two basic factors involved are the total number of instructions required and the execution time of each instruction. The product of these factors is the running time of a program, either the program for performing the intermediate language translation or the actual object program. Therefore, if internal high speed processing is to be provided both of these factors must be considered. To reduce the total number of program instructions a powerful instruction repertoire is required. This is in contrast to providing a basic instruction set where related operations must be handled with small instruction groups or packets. In addition to increasing the processing speed, a repertoire of singly powerful instructions contributes directly to the overall system efficiency since it will require fewer memory locations to store the program.

The execution time for each instruction is a function of the types of operations required and the logical implementation of the system. The latter aspect includes the types and capabilities of the logic blocks employed, the level of parallel operations available and the basic clock rate. As for desired operations; the most salient point in terms of arithmetic processing is floating point arithmetic. Up to the present time, high speed floating point capabilities have been available on only the most sophisticated systems. Even with this capability, many real-time system applications found it to be advantageous, speed-wise, to use fixed point arithmetic and to program the required scaling and alignment procedures. Quite a paradox! The digital computer via the floating point mode provides a superior facility for automatically handling all scaling operations required in arithmetic operations. This facility markedly reduces the level of programming complexities--An obvious

fact when one notes that floating point arithmetic is the standard arithmetic mode in all algebraic compilers whether or not the machine the compiler was implemented on had built-in floating point hardware. Therefore, it was decided that the system design of the EAI 8400 had to be centered around providing "the natural ability to handle floating point." To be more specific, the floating point facilities provided had to be of such a capability that the programmer need not be tempted to use fixed point arithmetic. This implies for real-time simulation, floating point multiplies in the 5 to 8 usec range with correspondingly faster addition times.

In the past, a separation existed between scientific and data processing type systems, the former with the capability of performing internal high speed arithmetic processing and the latter with the facility for high speed input/output and non-arithmetic data manipulation. It was learned that the separation was fallacious. Scientific problems require a fair amount of data processing, particularly when it comes to manipulating intermediate languages and other forms of list processing. In the other camp the users, who were initially concerned with only simple arithmetic computations, were turning to the more sophisticated matrix and/or statistical models. Therefore, it was recognized that to allow for economical high speed processing, significant data or byte handling facilities must be provided. This manifests itself in both the exchange or input/output system provided and the internal capabilities for manipulating bytes.

Ease of Programming

When considering the OFF-LINE workload--initial program coding and program modification--a number of the factors considered above for Economical High Speed Processing apply. The powerful instruction repertoire, the natural

ability to handle floating point and the capacity for extensive byte manipulations permit a programmer to concentrate on his own problem without spending large amounts of "machine dependent" time. That is, after establishing the requirement for a particular operation, he can then simply select the single instruction to accomplish it. This is in contrast to forcing him to be learned in the internal logical organization and knowing how the machine works before he can set about the primary task of solving a problem.

With regard to program modification, consider the floating versus fixed point question again. If in the initial coding the programmer used "every trick in the book" to squeeze out the maximum amount of computation in the minimum amount of time he is virtually backed in a corner if any program changes are required. Since in more cases than not such changes are required, the experienced programmer builds upon a more straighforward program which in the end will typically use the floating point facilities. The straight forward approach is a doctrine which is widely touted by many, since it permits reconstruction of the programming approach employed after the fact.

The "squeezing" phenomena becomes particularly acute in real time applications. In particular, real time simulations generally require a given computational workload to be executed at an iteration rate of from 20 to 50 cycles per second, thus requiring a 50 to 20 millisecond computing window, respectively. In present day systems this represents a significant chore. However, with the potential for high speed processing--in particular, floating point--this task could be readily handled using straight forward programming. Therefore such programs can be prepared and modified without requiring an extensive detailed programming effort.

Providing an extensive and powerful instruction repertoire can produce the opposite results unless serious consideration is given to the human factors

involved. In addition to providing a "human engineered" mnemonic language, an extensive software system must be made available. This interface must include the programming packages, necessary for operation close to the programmers natural language.

The Moment of Truth

Configuring a digital computing system under the above criteria--economical high speed processing and ease of programming--obviously offers a vast potential for maximizing the initial Computational Throughput/Total Cost ratio. In establishing the overall product line it was recognized that a significant level of software support was required. As such, we at EAI will be one of the largest users of the 8400 system and therefore, the Computational Throughput/ Total Cost ratio is of dual concern.

Programming costs are determined on an instruction per man-month basis. Therefore, unless each of the above design criteria is satisfied any so-called hardware savings will be more than spent in programming. Put somewhat crudely, it has been said that:

> ". . . every internal simplification of digital computer organization which reduces the completeness of its command structure will be taken out of the hide of the programmers . . ."

The following sections of this paper outline in more detail the design rationale employed in establishing the specifications of the EAI 8400 system.

WORD LENGTH CONSIDERATIONS

After establishing the basic system specifications and design goals, the next step involves the detailed analysis of the anticipated computational workload. The results of this analysis set the primary characterization of the system--word length. From this mile post each of the remaining features of the system are refined--instruction repertoire, processing rates and structural organization. In practice, all of these features must be considered concurrently, much like solving a set of simultaneous equations against an optimization critera. However, once this is accomplished, it is easier to sequentially portray each of the steps in the process.

Turning to the word length considerations, the starting points are the basic questions of binary vs. decimal (BCD) representation and fixed vs. variable word lengths. These questions are probably reviewed in the early stages of most systems designs and typically for high speed considerations, the same decision is reached--binary and fixed word length. The coding simplications and efficiency of utilization of the available number of bits in a binary module (2^n), far overshadow the "natural feel" for decimal conventions particularly when the software systems provide a more convenient "decimal interface." In the case of variable word lengths, the problems associated with boundry identifications, (tags and word marks) introduce coding and housekeeping difficulties which again outweigh the potential memory efficiencies available. However, the variable word length concept does provide a key input in analyzing word lengths. This key suggests that more than one fixed word length is desirable with each particular length serving the requirements for which it was intended. Therefore, the word length question is not which size should be selected, but which sizes? Furthermore, if a multitude of sizes are in order, what relationship must each have to one another?

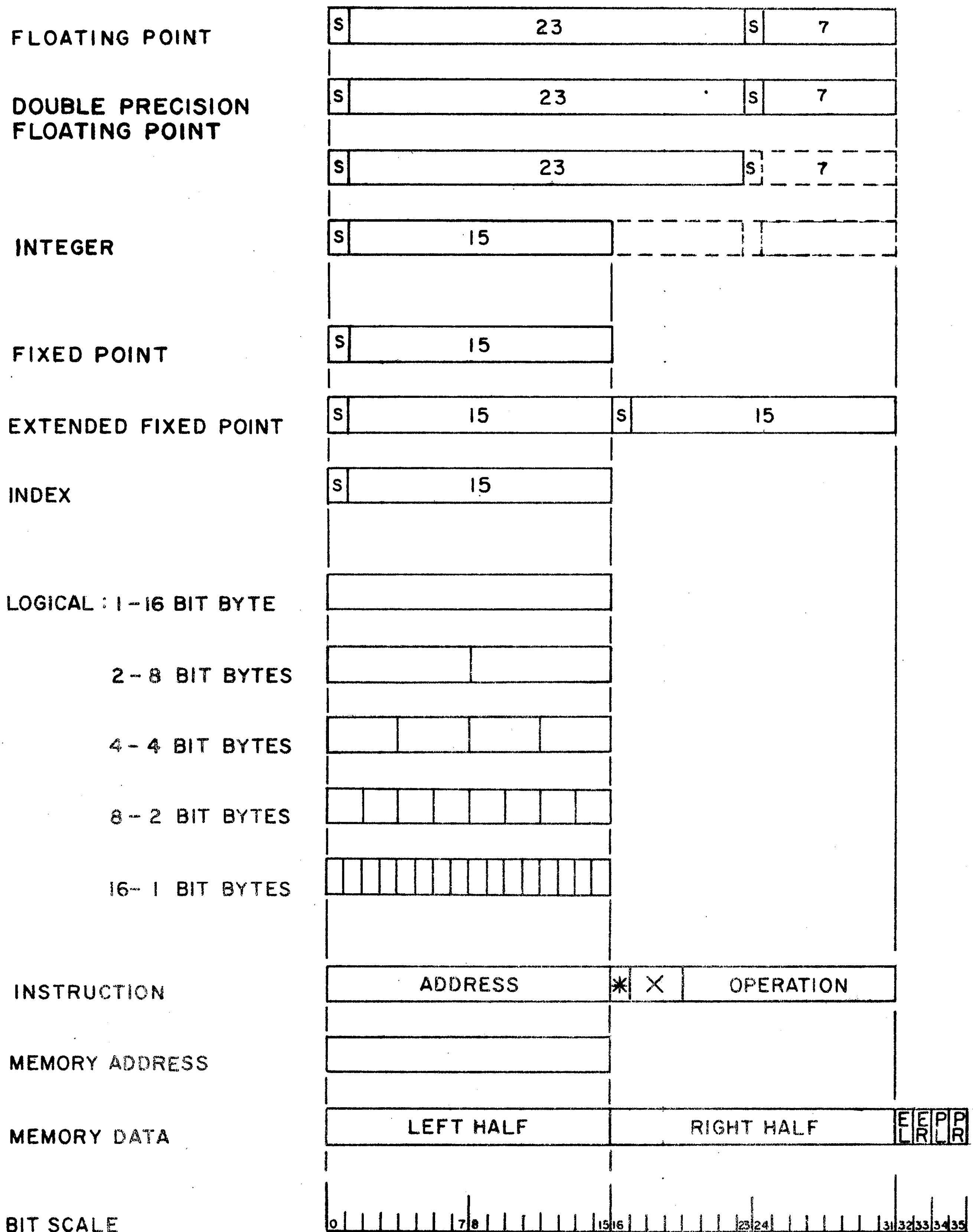
The answers to each of these questions was resolved by considering the following requirements:

1. The accuracy required for arithmetic computations, particularly with regard to "the natural ability to handle floating point"
2. Facilities for handling convenient byte sizes, as related to the non-arithmetic computational workload, including logical operations, data manipulations and list processing.
3. An instruction word length, which can directly address a large memory capacity and provide sufficient latitude for a powerful yet consistently formatted repertoir.
4. With more than one word length, a symetrical organization is required to permit inter-word size manipulations, efficient utilization of memory capacity and a convenient structure for the programmer to work with.
5. Control bit facilities for the flagging of individual memory locations.
6. Finally, error detection features in the form of parity.

The word length complement selected for the EAI 8400 is depicited in Figure 1. In the following material, some of the rationale used in the selection process will be outlined.

Arithmetic Processing

The most important item considered in this category was "the natural ability to handle floating point". This meant that a basic word size had to be selected which could house a respectable floating point word length and, as such, would be available within a single memory cycle. The general approach to attacking this selection process is to define the highest accurracy required in the intended fields of application and set the length accordingly.



EAI 8400 WORD FORMATS

FIGURE 1

However, this is unrealistic. Since hardware costs are directly related to word length a more systematic approach must be employed. Looking deeper into the accuracy requirement, a frequency distribution emerges which indicates that the percentage utilization of the initial highest accuracy estimate is very low exeept for particular applications such as the steps involved in inverting a large matrix. Therefore, the obvious choice is to provide more that one floating point word length. For the EAI 8400, a 32 bit standard size was selected with facilities for 56 bit double precision operations.

Starting with the basic rationale of employing high speed floating point arithmetic for all but the less sophisticated computations, a reasonable look was required at the fixed point facilities to be included. Basically in a system where floating point calculations can be performed faster, and much more conveniently, the fixed point workload reduces to those tasks in which scaling and/or overflow considerations are not encountered. This amounts to low accuracy computations, such as the a) manipulation of function generation tables where the data elements of the table are derived from emperical calculations or transducers, b) computations involving address or indexing words and c) preliminary calculations on data involved in an interface with analog components through analog-to-digital and digital-to-analog converters. With these considerations, the standard fixed point word length was set at 16 bits with the provision for 32 bit extended fixed point operations.

In both the floating and fixed point domains, the word formats for the double precision floating point and the extended fixed point facilities were established to be consistent with their respective standards. That is, in executing a 32 bit floating point multiply, the resultant double precision product is exactly the format selected for the double precision floating point word. A similar situation arises in division. Therefore, the results of 32 bit floating point multiply operations can be accumulated using the double

precision floating point add operations, eg in the case of forming the dot product $\sum Y(I) * Z(I)$. Furthermore, each of the respective most significant and least significant portions of the results can be operated on separately since the sign and exponents are preserved.

In modern computational procedures, it is mandatory to be able to communicate between the floating and fixed point domains. The earliest demand for such a facility was established in requirements for "Mixed Mode" arithmetic statements, or lack of them in algebraic compilers. Typically, computations which involved address indicies in floating point had to converted to integer type operands, - the statement A=I + B being illegal. In other instances, particularly processing converted data with a fixed point analog type source, continuous calling of fixed to floating and floating to fixed routines was required. To eradicate these deficiencies an "Integer" arithmetic mode was included in the EAI 8400. With this facility, data in 16 bit fixed point formats are automatically "floated" as they are fetched from memory for processing and "integerized" as they are stored in memory.

Byte Processing

Turning from the arithmetic processing demands to the data manipulation class of operations, the most salient requirement is established by the byte sizes and codes of peripheral devices. With the recent establishment of a "standard interchange code", designed to bridge the gap between the data processing 6 bit Hollerith or BCD codes and the communication, 5 bit, Baudot codes, a new generation of peripheral devices is being established. While the official code from the X 3.4 committee, the ASC II code, has been adopted as a "standard", IBM in recognition of the translation problems involved in changing from the present BCD code has more recently announced an Extended

BCD Interchange Code. The relationship between the BCD, EBCDIC and the ASCII codes are shown in Figure 2.

By configuring an 8 bit interface to the peripheral device world all of the above codes can be conveniently handled including the present 6 and 5 bit codes. Therefore, this selection was made. At this point, all other word length consideration factors must be treated with regard to the 8 bit selection such that the lengths be multiplies of the 8 bit byte.

Looking at the data handling criteria in more detail, a number of other factors must be included. The standard interchange codes not noly establish 8 bits as the overall byte or character size, but in addition define a 4 bit subset for representing pure numeric or decimal data. This provides more efficient utilization of the 8 bit byte by permitting the packing of 2, 4 bit bytes in an 8 bit byte.

Finally, it is obviously not sufficient to consider these factors only in terms of interfacing with peripheral devices. Of more significant concern is the requirement for internal data manipulation. These facilities take on a high degree of importance in the tasks of manipulating programming language statements, output documentation and the general non-arithmetic portion of a systems workload encountered both in scientific and data processing problems. For these takss, facilities must be available for handling not only the 8/4 bit bytes but their subsets, the 2 and 1 bit bytes and a superset 16 bit byte.

Instruction Word Length

While a latter section of this paper outlines in more detail the considerations involved in configuring the instruction word format, it is worthwhile to note the basic delinations. The address field was established to be exactly equal to the standard fixed point word length which in turn

BIT POSITIONS	2 3			
4 5 6 7	0 0	0 1	1 0	1 1
0 0 0 0	ƀ	BLANK	–	+
0 0 0 1	1	/	J	A
0 0 1 0	2	S	K	B
0 0 1 1	3	T	L	C
0 1 0 0	4	U	M	D
0 1 0 1	5	V	N	E
0 1 1 0	6	W	O	F
0 1 1 1	7	X	P	G
1 0 0 0	8	Y	Q	H
1 0 0 1	9	Z	R	I
1 0 1 0	Ø	‡	!	?
1 0 1 1	=	,	$	.
1 1 0 0	'	(	*	)
1 1 0 1	:	~	]	[
1 1 1 0	>	\	;	<
1 1 1 1	√	⧻	▲	⧧

a.) STANDARD 6 BIT BCD CODE

BIT POSITIONS	2 3 (01: 00)				01				10				11			
4 5 6 7	0 0	0 1	1 0	1 1	0 0	0 1	1 0	1 1	0 0	0 1	1 0	1 1	0 0	0 1	1 0	1 1
0 0 0 0	NULL				BLANK	&	–						>	<	‡	0
0 0 0 1							/		a	j			A	J		1
0 0 1 0									b	k	s		B	K	S	2
0 0 1 1									c	l	t		C	L	T	3
0 1 0 0	PF	RES	BYP	PN					d	m	u		D	M	U	4
0 1 0 1	HT	NL	LF	RS					e	n	v		E	N	V	5
0 1 1 0	LC	BS	EOB	UC					f	o	w		F	O	W	6
0 1 1 1	DEL	IDL	PRE	EOT					g	p	x		G	P	X	7
1 0 0 0									h	q	y		H	Q	Y	8
1 0 0 1					.		,	"	i	r	z		I	R	Z	9
1 0 1 0					?	!		:								
1 0 1 1					.	$	,	#								
1 1 0 0					←	*	%	@								
1 1 0 1					(	)	~	'								
1 1 1 0					+	;	—	=								
1 1 1 1					‡	¢	±	√								

c.) EXTENDED BCD INTERCHANGE CODE (EBCDIC)

BIT POSITIONS	2 3			
4 5 6 7	0 0	0 1	1 0	1 1
0 0 0 0	0	+	–	BLANK
0 0 0 1	1	A	J	/
0 0 1 0	2	B	K	S
0 0 1 1	3	C	L	T
0 1 0 0	4	D	M	U
0 1 0 1	5	E	N	V
0 1 1 0	6	F	O	W
0 1 1 1	7	G	P	X
1 0 0 0	8	H	Q	Y
1 0 0 1	9	I	R	Z
1 0 1 0	ƀ	?	!	‡
1 0 1 1	=	.	$	,
1 1 0 0	'	)	*	(
1 1 0 1	:	[	]	~
1 1 1 0	>	<	;	\
1 1 1 1	√	⧧	▲	⧻

b.) 6 BIT COLLATING CODE

BIT POSITIONS	X 5 (76: 00)				01				10				11			
4 3 2 1	0 0	0 1	1 0	1 1	0 0	0 1	1 0	1 1	0 0	0 1	1 0	1 1	0 0	0 1	1 0	1 1
0 0 0 0	NULL	DC$_0$			BLANK	0					@	P				p
0 0 0 1	SOM	DC$_1$			!	1					A	Q			a	q
0 0 1 0	EOA	DC$_2$			"	2					B	R			b	r
0 0 1 1	EOM	DC$_3$			#	3					C	S			c	s
0 1 0 0	EOT	DC$_4$ STOP			$	4					D	T			d	t
0 1 0 1	WRU	ERR			%	5					E	U			e	u
0 1 1 0	RU	SYNC			&	6					F	V			f	v
0 1 1 1	BELL	LEM			'	7					G	W			g	w
1 0 0 0	BKSP	S$_0$			(	8					H	X			h	x
1 0 0 1	HT	S$_1$			)	9					I	Y			i	y
1 0 1 0	LF	S$_2$			*	:					J	Z			j	z
1 0 1 1	VT	S$_3$			+	;					K	[			k	
1 1 0 0	FF	S$_4$			,	<					L	\			l	
1 1 0 1	CR	S$_5$			–	=					M	]			m	
1 1 1 0	SO	S$_6$			.	>					N	↑			n	ESC
1 1 1 1	SI	S$_7$			/	?					O	←			o	DEL

d.) AMERICAN STANDARD CODE FOR INFORMATION INTERCHANGE (ASCII)

FIGURE 2

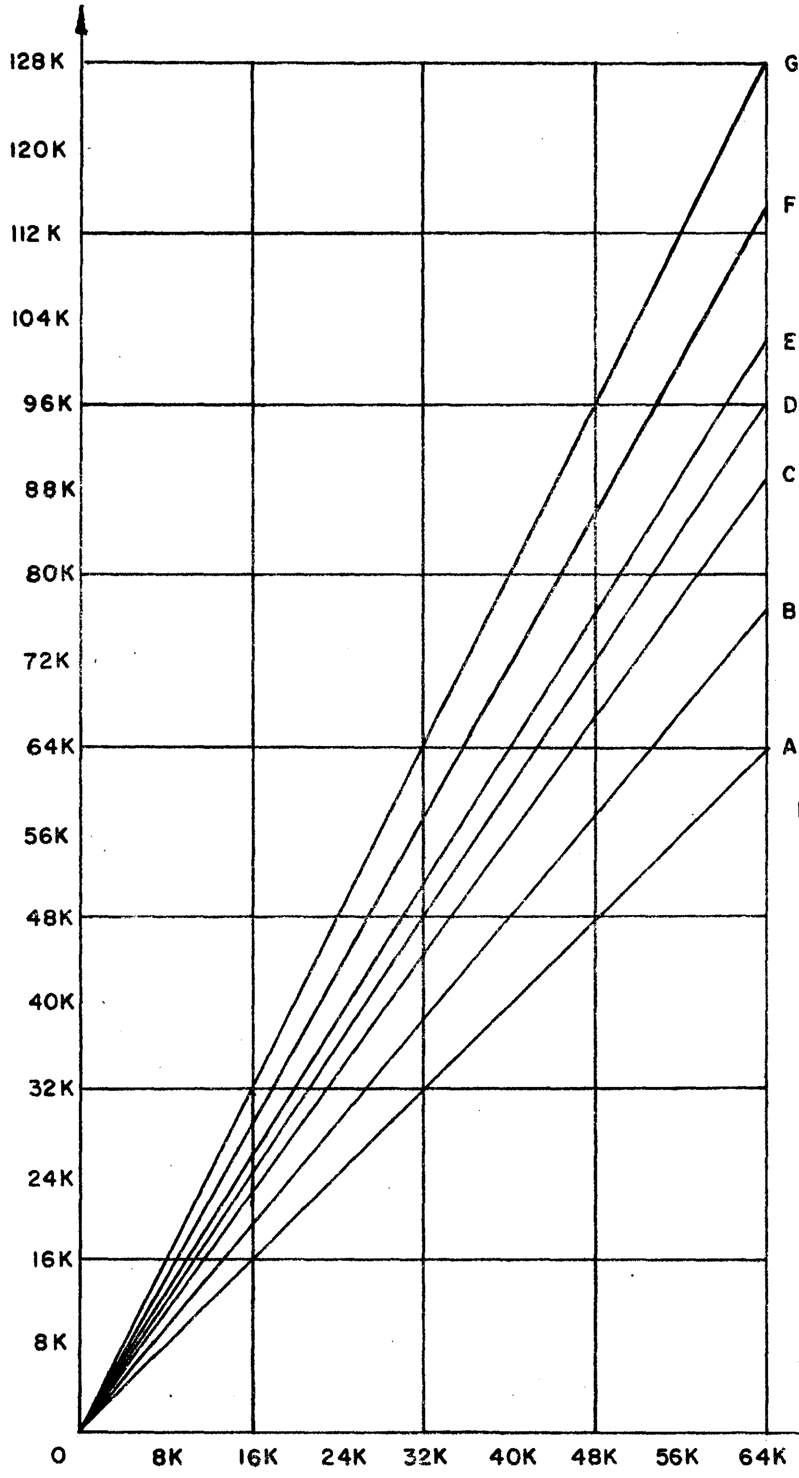
is equal to the remaining fields of the other half of the instruction word. In both fields sufficient laditude is provided for direct addressing of a large memory capacity and establishing a powerful instruction repertoire.

Symetrical Organization & Memory Efficiency

Completing the complement of word lengths included in the EAI 8400 are the memory address and memory data words. The former, a 16 bit word size, is consistently formatted with the address field in the instruction word and the index word.

The memory data word provides 32 data bits and 4 control bits. The function of the control bits will be described in a following section. Centering on the 32 data bits, it can now be appreciated what is meant by symetrical organization. Looking over the entire complement of word formats, three main sizes dominated-the 32 bit full word and 16 bit half word both multiplies of the 8 bit byte. Consequently, the user need only remember the key numbers 32/16/8 to remember the primary characterization of the system.

Furthermore, since the 32 bit memory data words can be manipulated as one 32 bit full word or two 16 bit half words, maximum memory packing efficiency can be obtained. This is better illustrated in Figure 3 which outlines the "effective memory capacity" as a function of the percentage of full to half words. In the limit a 64K full word memory capacity represents an effective memory capacity of 128K half words each of which can be directly addressed. Actually since the byte manipulating facilities in the EAI 8400 permit addressing of 8, 4, 2 and 1 bit bytes the effective byte memory capacity in terms of 64K full words is extended to 256K, 8 bit bytes; 512K, 4 bit bytes; 1,024K, 2 bit bytes and 2,048K, 1 bit bytes.



PROGRAM MIX	% FULL WORD	% HALF WORD	% EWC/FWC
A	100	0	100
B	80	20	120
C	60	40	140
D	50	50	150
E	40	60	160
F	20	80	180
G	0	100	200

FIGURE 3.

Control Bits

In the overall computational workload of a digital computer system it becomes necessary to be able to mark memory data words. More specifically in the case of memory protection, dynamic relocation of object programs and stack or table pointing the particular data words which require special attention must be separated from the remaining words. For example, when relocating a program from one section of memory to another a base address must be added to some but not all of the instruction word address fields.

This facility is provided for via a pair of EXEC bits included in every memory data word. The Left and Right EXEC bits relate to the Left and Right 16 bit half words respectively. Therefore each half word can be individually "marked."

The instruction repertoire provides the facility to set, reset and test individual EXEC bits. This unique facility permits a wide latitude control system heretofore not available in high speed digital computing systems.

Error Control

As the demands for computational throughput increase the requirement for error control becomes more stringent. This manifests itself in the processing of data with regard to memory and peripheral device operations. While the intrinsic reliability of memory systems has increased significantly in recent years, if a single error occurs it would be virtually impossible to discover it when considering the vast error sources available--the combinatorial electronics which manipulate over 2×10^6 bits in a 64K system. The two memory bits included in every memory data work are partitioned one for each half word, thereby increasing the level of error sensitivity and providing individual half word pin-pointing of an error source.

As for peripheral devices, a 9th bit is appended to the 8 bit byte format in accordance with the EBCDIC conventions. With existing BCD devices, the 7th bit is coupled into the bit 9 position.

ESTABLISHING THE INSTRUCTION REPERTOIRE

The basic system specifications relating to the instruction repertoire are:

1. High speed processing as a function of both the number of instructions required and the time to execute each instruction. Therefore, to reduce the total number of instructions, a powerful instruction repertoire must be provided such that a number of related operations can be specified in a single instruction as opposed to using a group or packet from a more basic instruction set.
2. Ease of programming as related to the facilities available to the user in the form of a convenient instruction set properly "human engineered" to reduce the OFF-LINE programming workload.

In establishing the instruction repertoire for the EAI 8400 a number of studies were performed and analyzed to decide on the following features.

Addressing Configurations

For the normally encountered address manipulations involving locations in core memory, a full range of indirect and indexing facilities were required. The indirect and indexing facilities must be independent to permit multi-level indirect addressing with indexing at each level. This feature finds wide use in handling subscripted arrays of data, subroutine transfer vectors and the table pointing operations involved in list processing. With regard to index register capacity, it was found that the number of index registers which may be gainfully employed ranges from 2 to 7 depending on the depth of inner loops and any attendant housekeeping (tallying) operations required. Therefore, it was decided to include independent selection of up to 7 index registers, each of which is 16 bits in length commensurate with the directly addressable capacity range.

For operations involving half-word operands, an immediate addressing facility was included. This permits handling the actual operand in the address

field of the instruction word. In addition to conserving memory capacity, this feature reduces the number of memory accesses required for this class of instruction and, therefore, increasing the processing speed. Furthermore, since the immediate operand is located in the address field of the instruction word, it can be modified by contents of a specified index register. This modification is affected prior to using the immediate operand. Therefore, one can think, in terms of "effective immediate operands."

An extension of the immediate operand system is employed in the shift class of instructions. With this class, the number of shifts desired is specified as an arithmetic operand with the sign of the shift count specifying the direction. As above, shift counts can be modified by the contents of a specified index register.

Finally, the accumulator itself can be specified in the address field to permit inter-register operations and direct communication with external devices.

Two's Complement Notation

For all the arithmetic modes of operations in the EAI 8400, the two's complement notation for negative numbers was selected. This notation provides a unique representation of zero and eliminates the requirement for recomplementation, thereby increasing the processing speed. The programmer need not concern himself with this particular selection in as much as software system provides a number system interface.

Universal Accumulator

The internal register complement for arithmetic and byte processing, provides a Universal Accumulator interface to the programmer. In earlier digital computing systems it was a common practice to configure the arithmetic registers such that the appropriate operands were available in different

registers as a function of the type of operation required and the internal logic required to perform the different types of operations. More specifically, while the operations of addition and subtraction were performed in the accumulator, multiplication operations were performed with the multiplier in an MQ or Q register, and the resultant product formed in the accumulator. This generated a housekeeping task for the programmer and necessitated the inclusion of inter-register transfer type instruction further complicating the programming workload.

Since the underlying design rationale for a single address digital computing systems specifies one operand at the memory address location and implicity assumes the location of the second operand in the accumulator, the latter need only exist with regard to the programmer at a single location. Therefore, in the EAI 8400, this location was universally established as the accumulator. All operations involving the implicit operand simply refer to the accumulator and the internal control logic automatically sequences the appropriate inter-register transfers in the execution of the operations. In addition to eliminating the requirement for special instructions and their attendant programming difficulties much processing time is saved by eliminating the execution time required for these special instructions.

Push-Pop Stacks and Multiple Accumulators

In recent years, much attention has been directed towards the use of register stack configurations which can be operated in a push-pop mode. While this mode is in line with the Polish notation employed in algebraic compilers and permits the use of "short", "relative addressing" type instructions, it has some serious limitations. Among the most significant are the special instructions and housekeeping operations required to manipulate the stack and the fact that the stack cannot be bottomless in a realistic system. The

latter limitation tends to magnify housekeeping problems by another order of magnitude.

Multiple accumulators on the other hand, is in the digital computing time span, a relatively old idea. One of the early 1950 vintage system has some 20 accumulators. Then as now, the primary reason for including more than one accumulator is to permit retention of operands in high speed flip-flop storage in contrast to core memory locations. The real question is how many accumulators are required? As each addition accumulator is added, it requires more space in the instruction word to specify which one(s) is to operated on and in which one the final result should be placed. Furthermore, as the available number of accumulators increases, the programmer is faced with another housekeeping chore. In analyzing typical workloads, it was found that the addition of one additional register satisfied the high speed demands of from 70% to 80% of the programs. This can be recognized when one considers the frequency distribution of the instructions.

STORE	in	TEMP 1
ADD	from	TEMP 1

and the number of times a TEMP 1 and TEMP 2 are used.

In light of this fact, it was decided to include in the EAI 8400 one such register -- the SAVE register. Operation with only one register permits a number of powerful functions without placing serious demands on the capacity of the instruction word. These functions include a) SAVEing the contents of the universal accumulator prior to the execution of the instruction in which its specified and b) addressing the contents of the SAVE register in the address field of an instruction word. Use of each of these functions is outlined in the following examples:

1. Form the dot product: $\sum Y(I)Z(I)$

```
               LOAD A       with   Y(1)
               MULTIPLY A   by     Z(1)    Form Y(1)Z(1)
SAVE A and     LOAD A       with   Y(2)    SAVE Y(1)Z(1)
               MULTIPLY A   by     Z(2)    Form Y(2)Z(2)
               ADD A        with   SAVE    Form Y(1)Z(1) + Y(2)Z(2)
SAVE A and     LOAD A       with   Y(3)    SAVE Y(1)Z(1) + Y(2)Z(2)
               -            -      -       -                 -
               -            -      -       -                 -
               ADD A        with   SAVE    Form Y(I)Z(I)
```

In this program, the SAVE register is used to hold the accumulated element products. Establishing an indexing loop would shorten the program.

2. Evaluate the polynomial: $F(x) = B(n)X^n + B(n-1)X^{n-1} + \ldots + B(1)X + B(0)$

In the nested format: $F(x) = ((\ldots(B(n)X + B(n-1))\ X + \ldots + B(1))X + B(0))$

```
               LOAD A       with   X
SAVE A and     LOAD A       with   B(n)    SAVE X and load B(n)
               MULTIPLY A   by     SAVE    Form B(n)X
               ADD A        with   B(n-1)  Form B(n) X + B(n-1)
               MULTIPLY A   by     SAVE    Form (B(n) X + B(n-1))X
               -            -      -       -
               -            -      -       -
               ADD A        with   B(0)    Form F(x)
```

This program serves to illustrate the use of the SAVE register as a high speed location for the variable, X, and as in the first example can be shortened with indexing.

Obviously, the power of the SAVE register adds a new dimension to high speed processing.

In addition to this facility and in recognition of the fact that a limited number of high speed storage locations can increase computational throughput a 16 word rapid access file is available. The individual locations in this file are specified as any other memory location and as such can be used to house either instructions or operands or both. In the former case, short high speed loops can be preloaded into the file and then operated upon from their relocated position. This provides for an increase in throughput on operations such as

table searching with a wide variety of test criterias, and each of the above sample programs in their indexing format.

Byte Size Operations

In recognition of the logical and data manipulation or list processing facilities required in a scientific computing system an extremely extensive repertoire of byte size operations has been included. In addition to the ability to select either 16, 8, 4, 2, or 1 bit bytes, individual byte positions can be specified as outlined in Figure 1. This eliminates the necessity for establishing complicated masks and permits more rapid processing. Furthermore, to increase processing rates even further all 16 possible logical connectives are provided. This means that any desired byte processing step can be specified in a single-instruction -- byte size, byte position and logical connective.

On Line Flag Register

In the course of running typical programs, a number of comparison operations and control steps are required. These include the ability to algebraically compare operands with respect to other operands and in the more specific case with regard to zero. The IF statement in algebraic compilers asks if an operand is "equal to", "greater than", or "less than" zero. Coupled with comparison operations are the actions desired as a function of the result. To date it has been common practice to provide "SKIP" or "JUMP" type facilities. The former (SKIP) being more popular from a logical designers standpoint because of the problems associated with including jump addresses in the shorter instruction word formats. However, this approach has many shortcomings when considering what the programmer needs. First of all, use of "SKIP" type operations typically forces the programmer to carefully manipulate the instruction steps following the "SKIP" instruction. Generally, one or more of these

following steps must be a JUMP instruction. So why not use JUMP instead of SKIP in the first place? Secondly, jumping is not the only action desired as a result of some comparison. In many instances, operations such as HALT, LINK to a subroutine, EXEcute are desired. With the JUMP only facility, the instruction at the JUMP address must be filled in with the desired operation. Again, why not provide the desired operation in the first place instead of forcing the programmer to wind his way around the mulberry bush of many other instructions? Thirdly, in addition to the results of algebraic comparisons there are a number of other internal status conditions that the programmer must concern himself with such as CARRY, OVERFLOW and console settings of flags. Finally, the testing of status conditions is only the first step in control type sequences. It is necessary to be able to SET, RESET and/or TRIGGER their conditions either in the form of initialization sequences or the establishment of branching steps.

Therefore, in the EAI 8400 it was decided to provide a) the ability to execute a full range of control type operations, JUMP, HALT, LINK, EXECUTE b) the entire ganut of internal status conditions and 8 console flags and c) the capacity of control (SET, RESET, TRIGGER) the status conditions. In line with the philosophy of singly powerful instructions all of these facilities can be specified in one instruction. This instruction class operates in conjunction with a 16 bit Flag register. The status of the individual bits of this register are established following the execution of other instructions. For example, after the execution of an ADD operation, the results in the accumulator are automatically checked and the ZERO, GREATER THAN or LESS THAN bits set. In this way, the FLAG register maintains the on-line status of the program.

Indexing Control

Under the earlier discussion on addressing configurations, the need for seven index registers was established. For complete program modification, control facilities must be available to increment or decrement and test their contents with regard to an established base. For indexing through the multi-dimensional arrays in matrix, function generation and list processing operations, it must be possible to increment or decrement by a quantity equal to or greater than 1. Furthermore, it most be possible to mainpulate arrays in both directions -- up or down a table. Again with the singly powerful instruction philosophy all of the facilities for a) modifying index values, b) algebraically testing the results of modifications and c) conditional jumping to desired locations are combined in one instruction.

Completing the Repertoire

Rounding out the instruction repertoire are the facilities for shifting, register loading and storing, status and function line control, EXEC bit control and EXCHANGE communication and control with Access devices. Some details on the latter class will be presented in a later portion of this paper.

Human Engineering

One of the most difficult tasks encountered in establishing a powerful instruction repertoire is that of finding the proper mnemonic interface to permit a programmer to make use of its power. If the list of available instructions is too extensive then its is difficult if not impossible to remember all of its elements such that the proper one can be singled out for use. Consequently, the programmer will tend to ignore most of the set and concentrate on using a subset. In this mode he will be back to writing small groups or packets of instructions to perform operations he could otherwise specify with a single instruction.

In the repertoire for the EAI 8400 this problem was solved by properly subdividing the instruction repertoire into convenient classes (C) each with a consistent set of class modifiers (M). In this way, the programmer need only remember C + M mnemonics to have at his fingertips C x M instructions. For example, the arithmetic operations include:

CLASSES	MODIFIERS
F - Floating Point	CA - Clear Add
D - Double Precision Floating Point	AD - Add
I - Integer	CS - Clear Subtract
- Fixed Point	SB - Subtract
E - Extended Fixed Point	CP - Compare
X - Index	MP - Multiply
	DV - Divide
	CD - Clear and Divide
	ST - Store
	SR - Store Rounded

Combining each of these FCA, FAD, etc. provide 6 x 10 = 60 instructions by remembering 6 + 10 = 16 mnemonics. Other instruction classes are arranged in a similar fashion.

Coupling the human engineered mnemonics, the symetrical structuring of 32/16/8 word formats, and a clear and consistent scheme of address modification provides a programmer with a powerful computational tool whose facilities can be easily and readily drawn upon.

THE EXCHANGE

Through the evolution of digital computing system, the facilities provided for the exchange of data between processing and memory elements and access devices have ranged from simple input/output gating systems to data assembly and editing complexes. The basic premise was one of attempting to balance the exchange type sequences required against the internal processing workload. This is a particularly difficult task because a) the ratio of exchange to processing workloads covers a very wide range - a fact which led to the establishment of separate data processing and scientific computing systems, and b) the range of access device transfer rates covers several orders of magnitude. This last fact is illustrated in Figure 4.

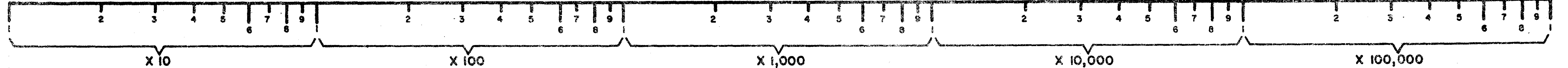
Another facet of the problem centers around the myriad of devices depicted in Figure 4. Each of these devices present different interface requirement which on earlier systems meant special control and/or translation units.

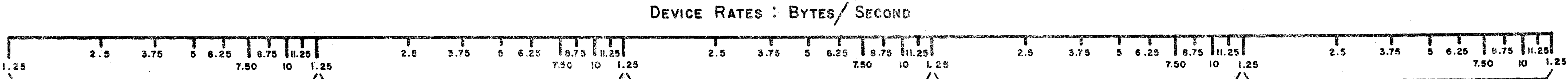
As in any system synthesis procedure, having formulated the problem the next task is to sort out each of the applicable elements to establish a solution. Analyzing the transfer rate data in Figure 4 and recognizing the fact that

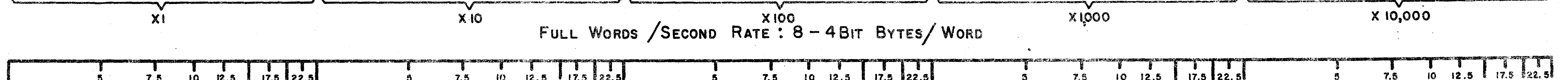
a) Scientific workloads demand a wide range and level of exchange operations - particularly when one considers the assembly, compilation and other language processing tasks.

b) Over the wide range of problem formulations it is typical to operate at least one input and one output device simultaneously.

c) Since exchange sequences can be reasonably "batched", to increase the overall computational throughput some sorm of concurrent exchange/ processing is advantageous.







ACCESS DEVICE TRANSFER RATES
FIGURE 4

d) With the adoption of "standard interchange codes" as described in the section on Word Length Considerations, a common denominator for byte handling could be reasonably established.

e) In considering multi-processor and/or time sharing systems, facilities must be provided for both "private" and "common" exchange configurations.

f) For real-time and other integrated systems involving special considerations, a flexible system interface is required including interrupt, single and multi-line control and data communication facilities.

Data Channels

The Exchange system established for the EAI 8400 as a function of each of the above elements includes both a data channel complex and system interface. The data channel complex provides for up to 8 bi-directional channels each with the capacity for handling up to 15 access devices. Each channel performs device control, data assembly/disassembly, collating code conversion and parity generation or checking. To permit the connection of many types of access devices without providing an all encompassing channel, the "standard interchange codes" was established as a common denominator. That is, all channels are designed to handle 8 bit bytes plus parity. All devices whose internal code is a subset of the standard code can be readily accommodated. In addition, each channel is capable of executing exchange sequences on 16 bit half words and the 4 bit byte "standard code subset." Therefore, the complete thread of consistency is continued through the byte processing powers provided in the instruction repertoire and the data exchange facilities.

With the common denominator channel, the specialized facilities required for the various types of devices are handled with individual device controllers. With this approach all of the existing devices and those being planned for the near future can be connected to a data channel.

The exchange sequences can be controlled from either single instructions in the main line of instructions or in a block transfer mode concurrently with internal processing. In this way, the programmer can elect either form of control depending on the particular operation desired.

System Interface

The demands of particular real-time and other integrated systems require a wide variation of control facilities. Since a standard set cannot be established and/or defined each of them must be tailored to the respective systems. Custom designs however tend to be very uneconomical. Therefore, in the design of the EAI 8400 it was decided to establish a standard, general purpose system interface. This interface includes interrupts, single and multi-line controls and data communication facilities. From the interface outward a wide variety of configurations may be established either internally to the EAI 8400 with standard logic elements or in external devices of systems. The decision as to which to employ is based on the particular control, device or system in question and the economies offered with each approach.

The interrupt system provides a full range of up to 256 levels of masked priority control. The masking feature permits, via internal instruction sequences, the capability of re-allocating priority levels. As such, it could be accurately defined as a dynamic priority interrupt system.

For the setting, resetting or testing of external control lines a comprehensive single or multi-line control facility is provided. This provides for establishing the condition of external functions and monitoring status line. The multi-line facility permits parallel function line control or status line testing.

The data communication feature in the system interface premits direct communication with up to 16 input and 16 output busses. Each of these busses are independent of the data channel system and can be directly addressed. A single instruction provides for the transfer of 16 bit half words between memory and the addressed bus. In this way, **external** data and/or control registers can be treated as intimately as the internal processing registers. This approach provides an extremely powerful facility for integrating a system configured with a wide variety of components.

Request/Response Organization

In both the data channel complex and the system interface all communication exchanges are synchronized under a request/response organization. That is, for each transfer, a request line is raised and is maintained in that status until an appropriate response is received. Where applicable, the next sequence is held in abeyance until the request/response action is fully satisfied. This organization eliminates the necessity for programmers to calculate timing sequences and device delays and further eliminates the dead time spans commonly introduced by the programmer to circumvent timing overlap problems.

Access Devices

As mentioned a number of times in the above discussions, the exchange design has been based on providing a general purpose facility for connecting to a wide variety of present and planned access devices. At the present time this includes all of the common document handling devices from paper tape to magnetic tape in addition to linkage systems, CRT display monitors and other real-time or integrated system components. In the near future the capacity and capabilities of many of these devices will be improved or augmented. However,

a more pronounced effort will be expended in establishing a more intimate form of man-machine communications. Extensions of the CRT, simple and complex remote or inquiry stations and other facilities are required and will be provided as the state-of-the-art advances.

STRUCTURAL ORGANIZATION

Having established all of the EAI 8400 system parameters, the next step is to perform the detailed logical design and establish the physical structural elements. Since the potential range of system configurations was wide, a modular organization was required. Actually, the modular approach proved to offer many important advantages including:

a) The potential for configuring systems from their most basic form up to and including high capacity single and multi-processor systems.

b) The capacity for providing "in field expansions" of individual elements as the user's computational demands increased.

c) An autonomous relationship between elements which provides adequate provisions for up-dating the system as a function of conceptual advances.

This last item deserves special attention particularly in light of the rapid studies being made in both system and component design in the digital computing field. With the autonomous organization each of the system elements are essentially separate entities--Processor, Memory, Exchange and Access Devices. The built in capacity to append forthcoming Access Devices was discussed in an earlier section. In other areas such as memory, with an autonomous structure as new memory devices or improved versions of present devices are made available, they can be included into the system without modification of existing elements.

In the case of processing elements, the range of expansibility has been further provided for by not completely saturating the instruction repertoire. Spare capacity is available for adding more complex arithmetic operations or algorithms such as square root.

Finally, with the standard system interface described in the Exchange section a wide laditude of external configurations can be established as a function of advances in real-time or other integrated systems.

Design Technology

Turning to the details of the present system structuring, the high processing speeds has been achieved by a hybrid logic element implementation. The standard EAI, DB^3's (Digital Basic Building Blocks) are combined with the latest monolythic integrated circuit elements. The 20 Mc DB^3's designed and manufactured by EAI have many hours of in-field, reliable operation and the integrated circuit elements have been designed and validated against MIL specifications. Each of these elements are woven into a high speed, high quality signal transmission system over a sophisticated ground plane structure. The overall system is combined into a modular packaging scheme. The individual racks are integrated in a "domino" type organization to provide compact system configurations, in-field expansion and the shortest possible transmission paths.

To further enhance in-field up time, a number of dynamic maintenance facilities have been included to permit more rapid execution of preventative maintenance routines. In particular, a full range of marginal controls are available each of which can be set and tested by control instructions. This permits the setting of marginal operating status followed by the high speed execution of the appropriate diagnostic routine.