G. M. Amdahl
G. A. Blaauw
F. P. Brooks, Jr.

Architecture of the IBM System/360

Abstract: The architecture* of the newly announced IBM System/360 features four innovations:

1. An approach to storage which permits and exploits very large capacities, hierarchies of speeds, read-only storage for microprogram control, flexible storage protection, and simple program relocation.

2. An input/output system offering new degrees of concurrent operation, compatible channel operation, data rates approaching 5,000,000 characters/second, integrated design of hardware and software, a new low-cost, multiple-channel package sharing main-frame hardware, new provisions for device status information, and a standard channel interface between central processing unit and input/output devices.

3. A truly general-purpose machine organization offering new supervisory facilities, powerful logical processing operations, and a wide variety of data formats.

4. Strict upward and downward machine-language compatibility over a line of six models having a performance range factor of 50.

This paper discusses in detail the objectives of the design and the rationale for the main features of the architecture. Emphasis is given to the problems raised by the need for compatibility among central processing units of various size and by the conflicting demands of commercial, scientific, real-time, and logical information processing. A tabular summary of the architecture is shown in the Appendices.

Introduction

The design philosophies of the new general-purpose machine organization for the IBM System/360 are discussed in this paper.† In addition to showing the architecture* of the new family of data processing systems, we point out the various engineering problems encountered in attempts to make the system design compatible, at the program bit level, for large and small models. The compatibility was to extend not only to models of any size but also to their various applications—scientific, commercial, real-time, and so on.

The section that follows describes the objectives of the new system design, i.e., that it serve as a base for new technologies and applications, that it be general-purpose, efficient, and strictly program compatible in all models. The remainder of the paper is devoted to the design problems faced, the alternatives considered, and the decisions made for data format, data and instruction codes, storage assignments, and input/output controls.

Design objectives

The new architecture builds upon but differs from the designs that have gradually evolved since 1950. The evolution of the computer had included, besides major technological improvements, several important systems concepts and developments:

* The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation.

† Additional details concerning the architecture, engineering design, programming, and application of the IBM System/360 will appear in a series of articles in the *IBM Systems Journal*.

REPRINTED FROM IBM JOURNAL OF RESEARCH AND DEVELOPMENT, VOL. 8, NO. 2, 1964; ©1964, 2000

0018-8646 / 00 / $5.00 © 2000 IBM

1. Adaptation to business data processing.

2. Growing importance of the total system, especially the input/output aspects.

3. Universal use of assembly programs, compilers, and other metaprograms.

4. Development of magnetic recording on tapes, drums, and disks.

5. Hundred-fold expansion of storage capacities.

6. Adaptation for real-time systems.

During this period most new computer models, from the point of view of their logical structure, were improved, enlarged, or technologically recast versions of the machines developed in the early 1950's. IBM products are not atypical; the evolution has gone from IBM 701 to 7094, 650 to 7074, from 702 to 7080, and from 1401 to 7010.

The system characteristics to be described here, however, are a new approach to logical structure and function, designed for the needs of the next decade as a coordinated set of data processing systems.

- *Advanced concepts*

It was recognized from the start that the design had to embody recent conceptual advances, and hence, if necessary, be incompatible with existing products. To this end, the following premises were considered:

1. Since computers develop into families, any proposed design would have to lend itself to growth and to successor machines.

2. Input/output (I/O) devices make systems specifically useful for given applications. A general method was needed for using I/O devices differing in data rate, access, and function.

3. The real value of an information system is properly measured by answers-per-month, not bits-per-microsecond. The former criterion required specific advances to increase throughput for a given internal speed, to shorten turnaround time for a given throughput, and to make the whole complex of machines and programming systems easier to use.

4. The functions of the central processing unit (CPU) proper are specific to its application only a minor fraction of the time. The functions required by the system for its own operation, e.g., compiling, input/output management, and the addressing of and within complex data structures, use a major share of time. These functions had to be made efficient, and need not be different in machines designed for different applications.

5. The input/output channel and the input/output control program had to be designed for each other.

6. Machine systems had to be capable of supervising themselves, without manual intervention, for both real-time and multiprogrammed, or time-shared, applications. To realize this capability requires: a comprehensive interruption system, tamper-proof storage protection, a protected supervisor program, supervisor-controlled program switching, supervisor control of all input/output (including unit assignment), nonstop operation (no HALT), easy program relocation, simple writing of read-only or unmodified programs, a timer, and interpretive consoles.

7. It must be possible and straightforward to assemble systems with redundant I/O, storages, and CPU's so that the system can operate when modules fail.

8. Storage capacities of more than the commonly available 32,000 words would be required.

9. Certain types of problems require floating-point word length of more than 36 bits.

10. As CPU's become increasingly reliable, built-in thorough checking against hardware malfunction is imperative for all systems, regardless of application.

11. Since the largest servicing problem is diagnosis of malfunction, built-in hardware fault-locating aids are essential to reduce down-times. Furthermore, identification of individual malfunctions and of individual invalidities in program syntax would have to be provided.

- *Open-ended design*

The new design had to provide a dependable base for a decade of customer planning and customer programming, and continuing laboratory developments, whether in technology, application and programming techniques, system configuration, or special requirements.

The various circuit, storage, and input/output technologies used in a system change at different times, causing corresponding changes in their *relative* speeds and costs. To take advantage of these changes, it is desirable that the design permit asynchronous operation of these components with respect to each other.

Changing application and programming techniques would require open-endedness in function. Current trends had to be extrapolated and their consequences anticipated. This anticipation could be achieved by direct provision, e.g., by increasing storage capacities and by using multiple-CPU systems, various new I/O devices, and time sharing. Anticipation might also take the form of generalization of function, as in code-independent scan and translation facilities, or it might consist of judiciously reserving spare bits, operation codes, and blocks of operation codes, for new modes, operations, or sets of operations.

Changing requirements for system configuration would demand not only such approaches as a standard interface between I/O devices and control unit, but also capabilities for a machine to directly sense, control, and respond to other equipment modules via paths outside the normal data routes. These capabilities permit the construction of supersystems that can be dynamically reconfigured under program control, to adapt more precisely to specialized functions or to give graceful degradation.

In many particular applications, some special (and often minor) modification enhances the utility of the system. These modifications (RPQ's), which may correct some shortsightedness of the original design, often embody operations not fully anticipated. In any event, a good general design would obviate certain modifications and accommodate others.

- *General-purpose function*

The machine design would have to provide individual system configurations for large and small, separate and mixed applications as found in commercial, scientific, real-time, data-reduction, communications, language, and logical data processing. The CPU design would have to be facile for each of these applications. Special facilities such as decimal or floating-point arithmetic might be required only for one or another application class and would be offered as options, but they would have to be integral, from the viewpoint of logical structure, with the design.

In particular, the general-purpose objective dictated that:

1. Logical power of great generality would have to be provided, so that all combinations of bits in data entities would be allowed and might be manipulated with operations whose power and utility depend upon the general nature of representations rather than upon any specific selection of them.

2. Operations would have to be code-independent except, of course, where code definition is essential to operation, as in arithmetic. In particular, all bit combinations should be acceptable as data; no combination should exert any control function when it appears in a data stream.

3. The individual bit would have to be separately manipulatable.

4. The general addressing system would have to be able to refer to small units of bits, preferably the unit used for characters.

Further, the implications of general-purpose CPU design for communications-oriented systems indicated a radical departure from current systems philosophy. The conventional CPU, for example, is augmented by an independent stored-program unit (such as the IBM 7750 or 7740) to handle all communications functions. Since the new CPU would easily perform such *logical* functions as code translation and message assembly, communications lines would be attached directly to the I/O channel via a control unit that would perform only character assembly and the electrical line-handling functions.

- *Efficient performance*

The basic measure of a good design is high performance in comparison to other designs having the same cost. This measure cannot be ignored in designing a compatible line. Hence each individual model and systems configuration in the line would have to be competitive with systems that are specialized in function, performance level or both. That this goal is feasible in spite of handicaps introduced by the compatibility requirement was due to the especially important cost savings that would be realized due to compatibility.

- *Intermodel compatibility*

The design had to yield a range of models with internal performance varying from approximately that of the IBM 1401 to well beyond that of the IBM 7030 (STRETCH). As already mentioned, all models would have to be strictly program compatible, upward and downward, at the program bit level.

The phrase "strictly program compatible" requires a more technically precise definition. Here it means that a valid program, whose logic will not depend implicitly upon time of execution and which runs upon configuration **A**, will also run on configuration **B** if the latter includes at least the required storage, at least the required I/O devices, and at least the required optional features. Invalid programs, i.e., those which violate the programming manual, are not constrained to yield the same results on all models. The manual identifies not only the results of all dependable operations, but also those results of exceptional and/or invalid operations that are not dependable. Programs dependent on execution-time will operate compatibly if the dependence is explicit, and, for example, if completion of an I/O operation or the timer are tested.

Compatibility would ensure that the user's expanding needs be easily accommodated by any model. Compatibility would also ensure maximum utility of programming support prepared by the manufacturer, maximum sharing of programs generated by the user, ability to use small systems to back up large ones, and exceptional freedom in configuring systems for particular applications.

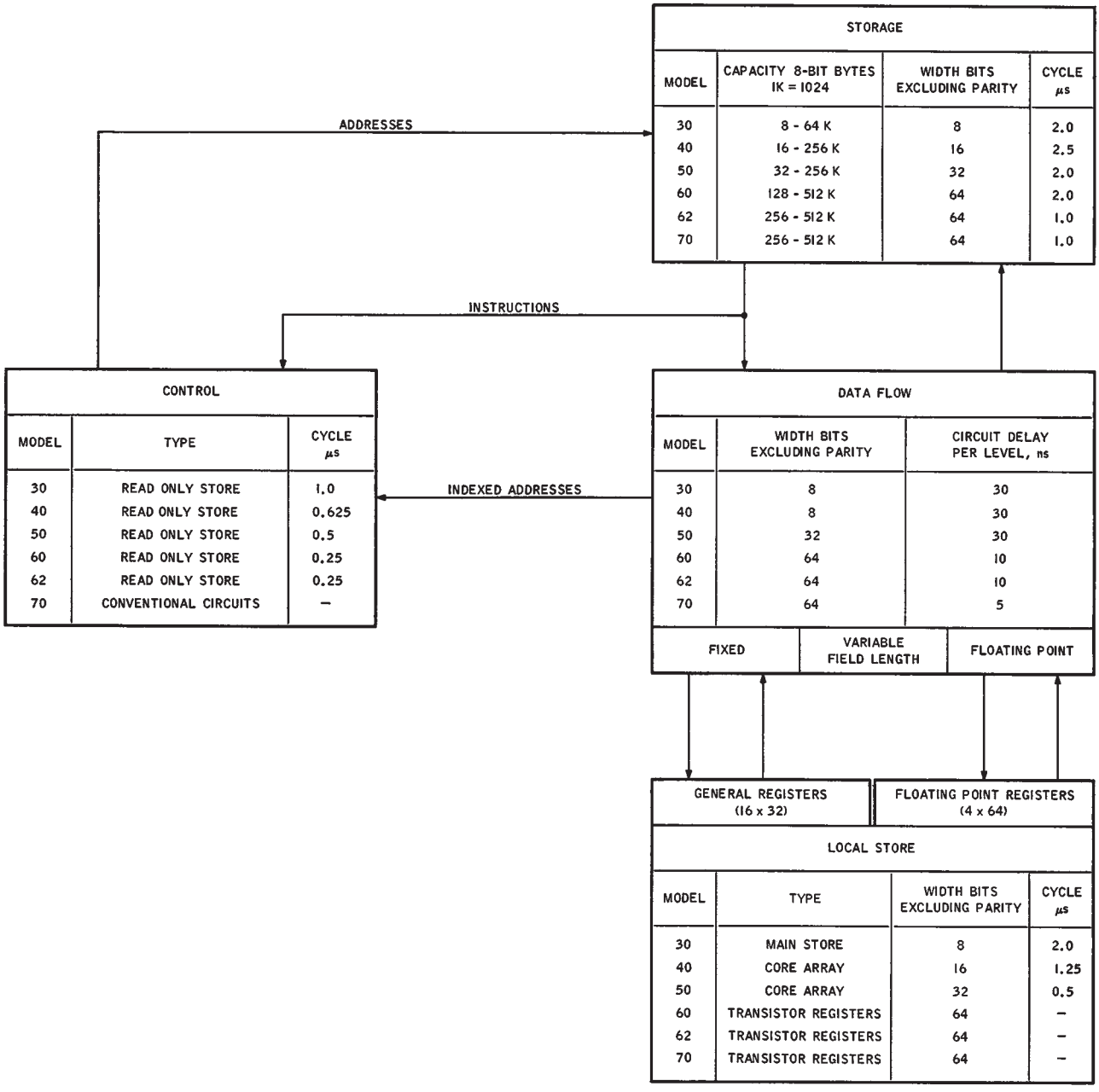
It required a new concept and mode of thought to make the compatibility objective even conceivable. In the last few years, many computer architects had realized, usually implicitly, that logical structure (as seen by the programmer) and physical structure (as seen by the engineer) are quite different. Thus each may see registers, counters, etc.,

that to the other are not at all real entities. This was not so in the computers of the 1950's. The *explicit* recognition of the duality of structure opened the way for the compatibility within System/360. The compatibility requirement dictated that the basic architecture had to embrace different technologies, different storage-circuit speed ratios, different data path widths, and different data-flow complexities. The basic machine structure and implementation at the various performance levels are shown in Fig. 1.

The design decisions

Certain decisions for the architectural design became mileposts, because they (a) established prominent characteristics of the System/360, (b) resolved problems concerning the compatibility objective, thus illuminating the essential differences between small models and large, or (c) resolved problems concerning the general-purpose objective, thus illuminating the essential differences among applications. The sections that follow discuss these de-

Figure 1 **Machine structure and implementation.**



cisions, the problems faced, the alternatives considered, and the reasons for the outcome.

- *Data format*

The decision on basic format (which affected character size, word size, instruction field, number of index registers, input-output implementation, instruction set layout, storage capacity, character code, etc.) was whether data length modules should go as 2^n or 3.2^n. Even though many matters of format were considered in the basic choice, we will for convenience treat the major components of the decision as if they were independent.

Character size, 6 vs 4/8. In character size, the fundamental problem is that decimal digits require 4 bits, the alphanumeric characters require 6 bits. Three obvious alternatives were considered — 6 bits for all, with 2 bits wasted on numeric data; 4 bits for digits, 8 for alphanumeric, with 2 bits wasted on alphanumeric; and 4 bits for digits, 6 for alphanumeric, which would require adoption of a 12-bit module as the minimum addressable element. The 7-bit character, which incorporated a binary recoding of decimal digit pairs, was also briefly examined.

The 4/6 approach was rejected because (a) it was desired to have the versatility and power of manipulating character streams and addressing individual characters, even in models where decimal arithmetic is not used, (b) limiting the alphabetic character to 6 bits seemed short-sighted, and (c) the engineering complexities of this approach might well cost more than the wasted bits in the character.

The straight-6 approach, used in the IBM 702–7080 and 1401–7010 families, as well as in other manufacturers' systems, had the advantages of familiar usage, existing I/O equipment, simple specification of field structure, and commensurability with a 48-bit floating-point word and a 24-bit instruction field.

The 4/8 approach, used in the IBM 650–7074 family and elsewhere, had greater coding efficiency, spare bits in the alphabetic set (allowing the set to grow), and commensurability with a 32/64-bit floating-point word and a 16-bit instruction field. Most important of these factors was coding efficiency, which arises from the fact that the use of numeric data in business records is more than twice as frequent as alphanumeric. This efficiency implies, for a given hardware investment, better use of core storage, faster tapes, and more capacious disks.

Floating-point word length, 48 vs 32/64. For large models addition time goes up slowly with word length, and multiplication time rises almost linearly. For small, serial models, addition time rises linearly and multiplication as the square of word length. Input/output time for data files rises linearly. Large machines more often require high precision; small machines more urgently require short operands. For this aspect of the basic format problem, then, definite conflicts arose because of compatibility.

Good data were unavailable on the distribution of required precision by the number of problems or running time. Indeed, accurate measures could not be acquired on such coarse parameters as frequency of double-precision operation on 36-bit and 48-bit machines. The question became whether to force all problems to the longer 48-bit word, or whether to provide 64 to take care of precision-sensitive problems adequately, and either 32 or 36 to give faster speed and better coding efficiency for the rest. The choice was made for the IBM System/360 to have both 64- and 32-bit length floating point. This choice offers the user the option of making the speed/space vs precision trade-off to best suit his requirements. The user of the large models is expected to employ 64-bit words most of the time. The user of the smaller models will find the 32-bit length advantageous in most of his work. All floating-point models have both lengths and operate identically.

Hexadecimal floating-point radix. With no conflcts in questions of large vs small machines, base 16 was selected for floating point. Studies by Sweeney[1] show that the frequency of pre-shift, overflow, and precision-loss post-shift on floating-point addition are substantially reduced by this choice. He has shown that, compared with base 2, the percentage frequency of occurrence of overflow is 5 versus 20, pre-shift is 43 versus 58, and precision-loss post-shift is 11 versus 18. Thus speed is noticeably enhanced. Also, simpler shifting paths, with fewer logic levels, will accomplish a higher proportion of all required pre-shifting in a single pass. For example, circuits shifting 0, 1, 2, 3, or 4 binary places cover 82% of the base 2 pre-shifts. Substantially simpler circuits shifting 0, 1, or 2 hexadecimal places cover 93% of all base 16 pre-shifts. This simplification yields higher speed for the large models and lower cost for the small ones.

The most substantial disadvantage of adopting base 16 is the shift in bit usage from exponent to fraction. Thus, for a given range and a given *minimum* precision, base 16 requires 2 fewer exponent bits and 3 more fraction bits than does base 2. Alternatively and equivalently, rounding and truncation effects are 8 times as large for a given fraction length. For the 64-bit length, this is no problem. For the 32-bit length, with its 24-bit fraction, the minimum precision is reduced to the equivalent of 21 bits. Because the 64-bit length was available for problems where the minimum precision cramped the user, the greater speed and simplicity of base 16 was chosen.

Significance arithmetic. Many schemes yielding an estimate of the significance of computed results have been proposed. One such scheme, a modified form of unnormalized arithmetic, was for a time incorporated in the design. The scheme was finally discarded when simulation runs showed this mode of operation to cost about one hexadecimal digit of actual significance developed, as compared with normalized operation. Furthermore, the

significance estimate yielded for a given problem varied substantially with the test data used.

Sign representations. For the fixed-point arithmetic system, which is binary, the two's complement representation for negative numbers was selected. The well-known virtues of this system are the unique representation of zero and the absence of recomplementation. These substantial advantages are augmented by several properties especially useful in address arithmetic, particularly in the large models, where address arithmetic has its own hardware. With two's complement notation, this indexing hardware requires no true/complement gates and thus works faster. In the smaller, serial models, the fact that high-order bits of address arithmetic can be elided without changing the low-order bits also permits a gain in speed. The same truncation property simplifies double-precision calculations. Furthermore, for table calculation, rounding or truncation to an integer changes all variables in the same direction, thus giving a more acceptable distribution than does an absolute-value-plus-sign representation.

The established commercial rounding convention made the use of complement notation awkward for decimal data; therefore, absolute-value-plus-sign is used here. In floating point, the engineering virtues of normalizing only high-order zeros, and of having all zeros represent the smallest possible number, decided the choice in favor of absolute-value-plus-sign.

Variable- versus fixed-length decimal fields. Since the fields of business records vary substantially in length, coding efficiency (and hence tape speed, file capacity, CPU speed, etc.) can be gained by operating directly on variable-length fields. This is easy for serial-by-byte machines, and the IBM 1401–7010 and 702–7080 families are among those so designed. A less flexible structure is more appropriate for a more parallel machine, and the IBM 650–7074 family is among those designed with fixed-word-length decimal arithmetic.

As one would expect, the storage efficiency advantage of the variable data format is diminished by the extra instruction information required for length specification. While the fixed format is preferable for the larger machines, the variable format was adopted because (a) the small commercial users are numerous and only recently trained in variable-format concepts, and (b) the large commercial system is usually I/O limited; hence the internal performance disadvantage of the variable format is more than compensated by the gain in effective tape rate.

Decimal accumulators versus storage-storage operation. A closely related question involving large/small models concerned the use of an accumulator as one of the operands on decimal arithmetic, versus the use of storage locations for all operands and results. This issue is pertinent even after a decision has been made for variable-length fields in storage; for example, it distinguishes IBM 702–7080 arithmetic from that of the IBM 1401–7010 family.

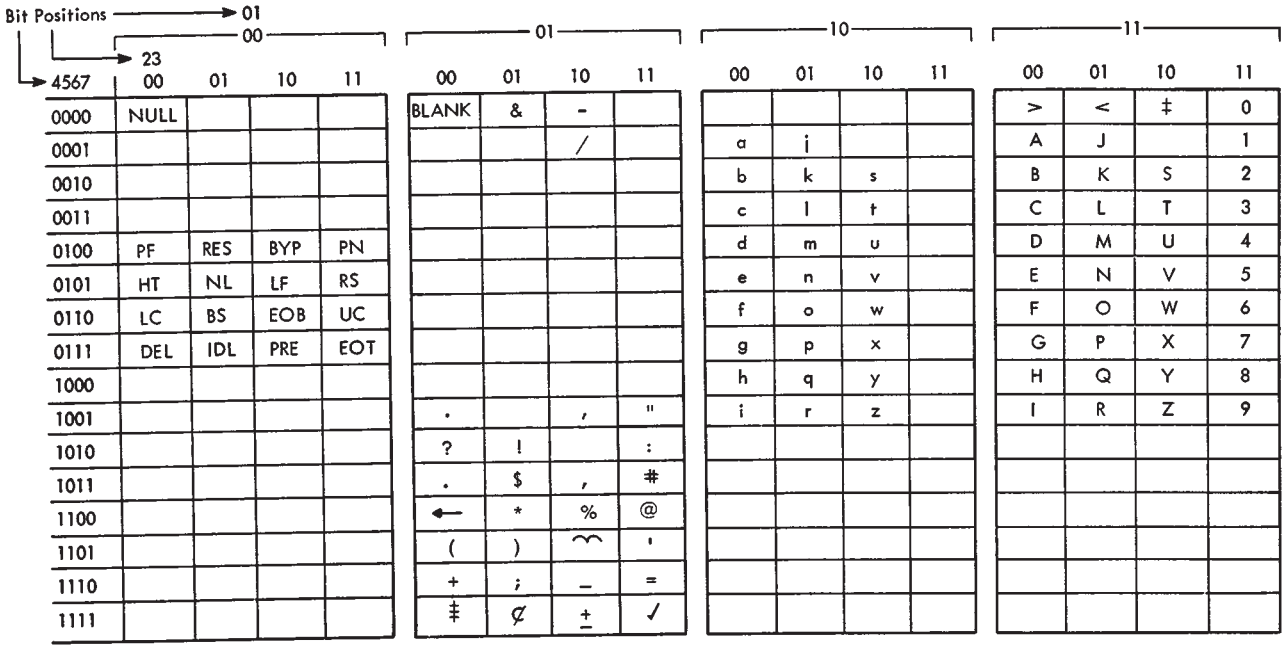
The large models readily afford registers or local stores and get a speed enhancement from using these as accumulators. For the small model, using core storage for logical registers, addition to an accumulator is no faster than addition to a programmer-specified location. Addition of two arbitrary operands and storage of the result becomes LOAD, ADD, STORE, however, and this operation is substantially slower for the small models than the MOVE, ADD sequence appropriate to storage-storage operation. Business arithmetic operations (as hand coded and especially as compiled from COBOL) often take this latter form and rarely occur in strings where intermediate results are profitably held in accumulators. In address arithmetic and floating-point arithmetic, quite the opposite is true.

Field specification: word-marks versus length. Variable-length fields can be specified in the data via delimiter characters or word-marks, or in the instruction via specification of field length or start-finish limits. For business *data*, the word-mark has some slight advantage in storage efficiency: one extra bit per 8-bit character would cost less than 4 extra length bits per 16-bit address. Furthermore, instructions, and hence addresses, usually occupy most core storage space in business computers. However, the word-mark approach implies the use of word-marks on instructions, too, and here the cost is without compensating function. The same is true of all fixed-field data, an important consideration in a general-purpose design. On balance, storage efficiency is about equal; the field specification was put in the instruction to allow all data combinations to be valid and to give easier and more direct programming, particularly since it provides convenient addressing of parts of fields. Length was chosen over limit specification to simplify program relocation and instruction modification.

ASCII vs BCD codes. The selection of the 8-bit character size in 1961 proved wise by 1963, when the American Standards Association adopted a 7-bit standard character code for information interchange (ASCII). This 7-bit code is now under final consideration by the International Standards Organization for adoption as an international standards recommendation. The question became "Why not adopt ASCII as the *only* internal code for System/360?"

The reasons against such exclusive adoption was the widespread use of the BCD code derived from and easily translated to the IBM card code. To facilitate use of both codes, the central processing units are designed with a high degree of code independence, with generalized code translation facilities, and with program-selectable BCD or ASCII modes for code-dependent instructions. Neverthe-

Figure 2a **Extended binary-coded-decimal (BCD) interchange code.**



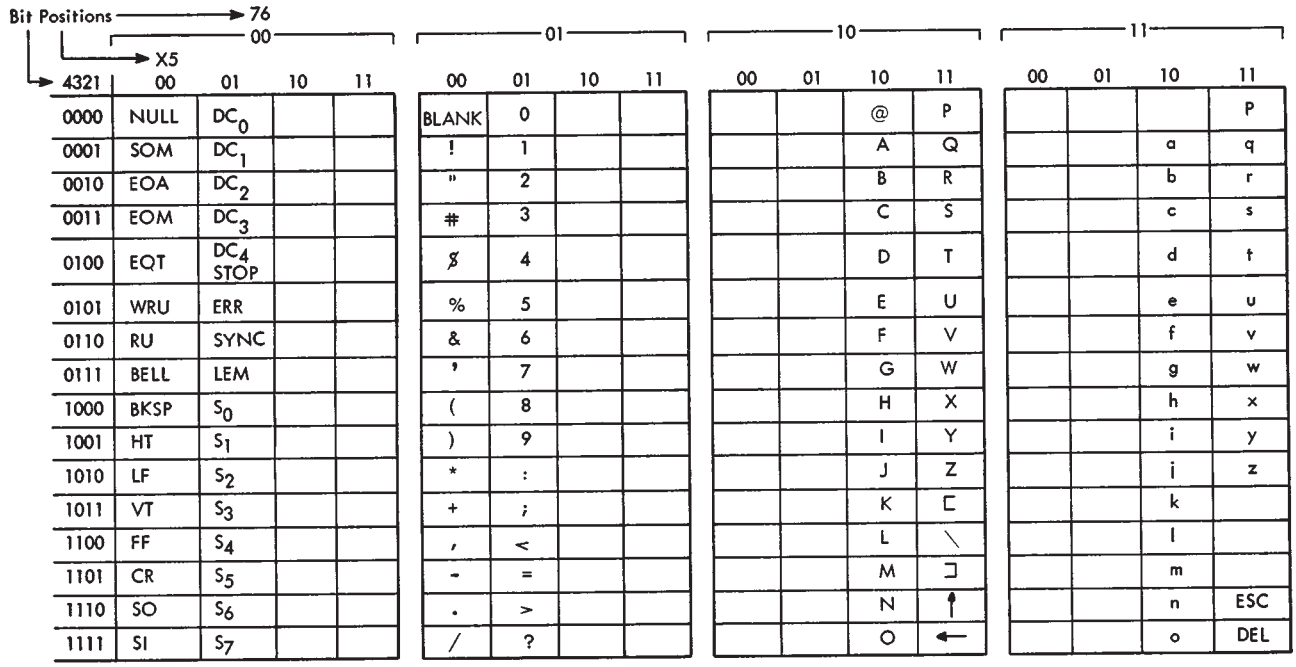
Bit Positions → 01 / 00 / 23 / 4567	00	01	10	11
0000	NULL			
0001				
0010				
0011				
0100	PF	RES	BYP	PN
0101	HT	NL	LF	RS
0110	LC	BS	EOB	UC
0111	DEL	IDL	PRE	EOT
1000				
1001				
1010				
1011				
1100				
1101				
1110				
1111				

01 / 00	01	10	11
BLANK	&	-	
		/	
.		,	"
?	!		:
.	$	,	#
←	*	%	@
(	)	⌒	'
+	;	_	=
‡	¢	±	✓

10 / 00	01	10	11
a	j		
b	k	s	
c	l	t	
d	m	u	
e	n	v	
f	o	w	
g	p	x	
h	q	y	
i	r	z	

11 / 00	01	10	11
>	<	‡	0
A	J		1
B	K	S	2
C	L	T	3
D	M	U	4
E	N	V	5
F	O	W	6
G	P	X	7
H	Q	Y	8
I	R	Z	9

Figure 2b **8-bit representation of the 7-bit American Standard Code for Information Interchange (ASCII).**



Bit Positions → 76 / 00 / X5 / 4321	00	01	10	11
0000	NULL	DC_0		
0001	SOM	DC_1		
0010	EOA	DC_2		
0011	EOM	DC_3		
0100	EQT	DC_4 STOP		
0101	WRU	ERR		
0110	RU	SYNC		
0111	BELL	LEM		
1000	BKSP	S_0		
1001	HT	S_1		
1010	LF	S_2		
1011	VT	S_3		
1100	FF	S_4		
1101	CR	S_5		
1110	SO	S_6		
1111	SI	S_7		

01 / 00	01	10	11
BLANK	0		
!	1		
"	2		
#	3		
$	4		
%	5		
&	6		
'	7		
(	8		
)	9		
*	:		
+	;		
,	<		
-	=		
.	>		
/	?		

10 / 00	01	10	11
		@	P
		A	Q
		B	R
		C	S
		D	T
		E	U
		F	V
		G	W
		H	X
		I	Y
		J	Z
		K	[
		L	\
		M	]
		N	↑
		O	←

11 / 00	01	10	11
			p
		a	q
		b	r
		c	s
		d	t
		e	u
		f	v
		g	w
		h	x
		i	y
		j	z
		k	
		l	
		m	
		n	ESC
		o	DEL

less, a choice had to be made for the code-sensitive I/O devices and for the programming support, and the solution was to offer both codes, fully supported, as a user option. Systems with either option will, of course, easily read or write I/O media with the other code. The extended BCD interchange code and an 8-bit representation of the 7-bit ASCII are shown in Fig. 2.

Boundary alignment. A major compatibility problem concerned alignment of field boundaries. Different models were to have different widths of storage and data flow, and therefore each model had a different set of preferences. For the 8-bit wide model the characters might have been aligned on character boundaries, with no further constraints. In the 64-bit wide model it might have been preferred to have no fields split between different 64-bit double-words. The general rule adopted (Fig. 3) was that each fixed field must begin at a multiple of its field length, and variable-length decimal and character fields are unconstrained and are processed serially in all models. All models must insure that programmers will adhere to these rules. This policing is essential to prevent the use of technically invalid programs that might work beautifully on small models but not on large ones. Such an outcome would undermine compatibility. The general rule, which has very few and very minor exceptions, is that invalidities defined in the manual are detected in the hardware and cause an interruption. This type of interruption is distinct from an interruption caused by machine malfunctions.

- *Instruction decisions*

Pushdown stack vs addressed registers. Serious consideration was given to a design based on a pushdown accumulator or stack.[2] This plan was abandoned in favor of several registers, each explicitly addressed. Since the advantages of the pushdown organization are discussed in the literature,[3] it suffices here to enumerate the disadvantages which prompted the decision to use an addressed-register organization:

1. The performance advantage of a pushdown stack organization is derived principally from the presence of several fast registers, not from the way they are used or specified.

2. The fraction of "surfacings" of data in the stack which are "profitable," i.e., what was needed next, is about one-half in general use, because of the occurrence of repeated operands (both constants and common factors). This suggests the use of operations such as TOP and SWAP, which respectively copy submerged data to the active positions and assist in clearing submerged data when the information is no longer needed.

3. With TOP's and SWAP's counted, the substantial instruction density gained by the widespread use of implicit addresses is about equalled by that of the same instructions with explicit, but truncated, addresses which specify only the fast registers.

4. In any practical implementation, the depth of the stack has a limit. The register housekeeping eliminated by the pushdown organization reappears as management of a finite-depth stack and as specification of locations of submerged data for TOP's and SWAP's. Further, when part of a full stack must be dumped to make room for new data, it is the *bottom* part, not the active part, which should be dumped.

5. Subroutine transparency, i.e., the ability to use a subroutine recursively, is one of the apparent advantages of the stack. However, the *disadvantage* is that the transparency does not materialize unless additional independent stacks are introduced for addressing purposes.

6. Fitting variable-length fields into a fixed-width stack is awkward.

In the final analysis, the stack organization would have been about break-even for a system intended principally for scientific computing. Here the general-purpose objective weighed heavily in favor of the more flexible addressed-register organization.

Full vs truncated addresses. From the beginning, the major challenge of compatibility lay in storage addressing. It was clear that large models would require storage capacities in the millions of characters. Small (serial) models would require short addresses to conserve precious core space and instruction fetch time. Some help was given by the decision to use register addressing, which reduces address appearances in the instruction stream by a factor approaching 2.

An early decision had dictated that all addresses had to be indexable, and that a mechanism had to be provided for making all programs easily relocatable. The indexing technique had fully proved its worth in current systems.[4] This technique suggested that abundant address size could be attained through a full-sized index register, used as a base. This approach, coupled with a truncated address in the instruction, gives consequent gains in instruction density. The *base-register* approach was adopted, and then augmented, for some instructions, with a second level of indexing.

Now the question was: How much capacity was to be made directly addressable, and how much addressable only via base registers? Some early uses of base register techniques had been fairly unsuccessful, principally because of awkward transitions between direct and base addressing. It was decided to commit the system completely to a base-register technique; the direct part of the address, the *displacement*, was made so small (12 bits, or 4096 characters) that direct addressing is a practical programming technique only on very small models. This

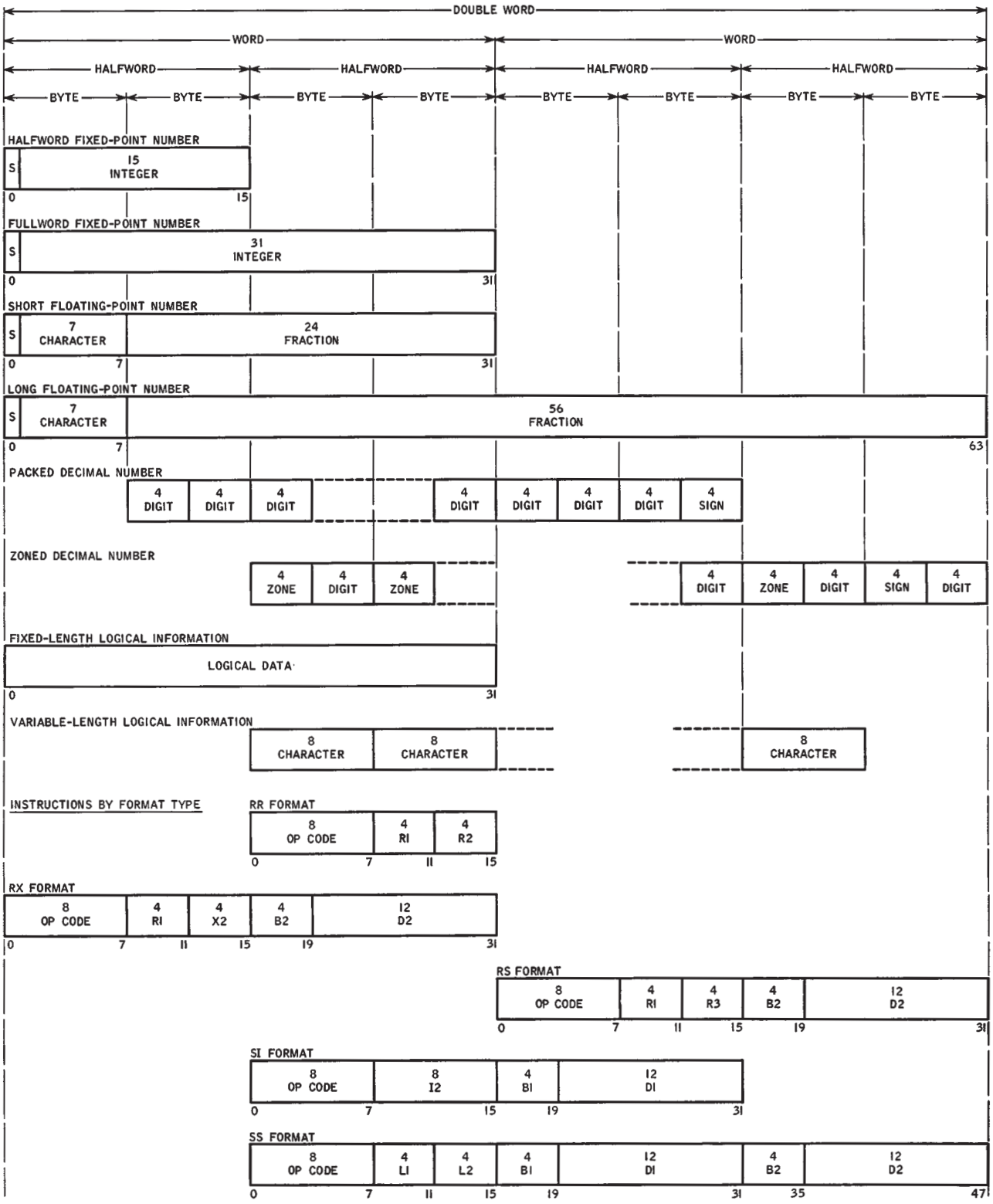


Figure 3 **Boundary alignment of formats.**

commitment implies that all programs are location-independent, except for constants used to load the base registers. Thus, all programs can easily be relocated. This commitment also implies that the programming support effectively and efficiently handles the mechanics of base-register use. The assembler automatically constructs and assigns base-plus-displacement addresses as it constructs the symbol table. The compilers not only do this, but also allocate base registers to give efficient programs.

Decimal vs binary addressing. It was decided to use binary rather than decimal addressing, because (a) assembly programs remove the user one level from the address, thus reducing the importance of familiar usage, (b) binary addressing is more efficient in the ratio 3.32/4.00, and (c) table exploitation is easier and more general because any datum can be made into or added to a binary address, yielding a valid address. This decision, however, represented some conflict with past approaches. Machines for purely business applications had often used decimal addressing (in the ancestral machine of the family). Most business computers now have binary addressing or have evolved to mixed-radix addressing.

Multiple accumulators. An extrapolation of technological trends indicated the probable availability of small, high-speed storage. Consequently, the design uses a substantial number of logically identifiable registers, which are physically realized in core storage, local high-speed storage, or transistors, according to the model. There are sixteen 32-bit general-purpose registers and four 64-bit floating-point registers in the logical design, with room for expansion to eight floating-point registers. Surprisingly enough, the multiple-register decision was not a large-small conflict. Each model has an appropriate (and different) mechanization of the same logical design.

Storage hierarchies. Technology promises to yield a continuing spectrum of storage systems whose speed varies inversely with capacity for equal cost-per-bit. Of equal significance, problem requirements naturally follow a matching pattern — small quantities of data are used with great frequency, medium quantities with medium frequency, and very large quantities with low frequency. These facts promise substantial performance/cost advantages if storage hierarchies can be effectively used.

It was decided to accept the engineering, architectural, and programming disciplines required for storage-hierarchy use. The engineer must accommodate in one system several storage technologies, with separate speeds, circuits, power requirements, busing needs, etc., all requiring asynchronous operation of all storage with respect to the CPU. The system programmer must contend with awkward boundaries within total storage capacity and must allocate usage. He must devise addressing for very large capacities, block transfers, and means of handling, indexing across and providing protection across gaps in the addressing sequence.

Separate vs universal accumulators. There are several advantages of having fixed- and floating-point arithmetic use the same logical (as opposed to physical) registers. There are some less obvious disadvantages which weighed in favor of separate accumulator sets. First, in a given register specification (4 bits, in our case) the use of separate sets permits more registers to be specified because of the information implications of the operation code. Second, in the large models instruction execution and the preparation of later instructions are done concurrently in separate units. To use a single register set would couple these closely, and reduce the asynchronous concurrency that can be attained. Historically, index registers have been separated from fixed-point registers, limiting analysis of register allocation to index quantities only. Integration of these facilities brings the full power of the fixed-point arithmetic operation set to bear upon indexing computations. The advantages of the integration appear throughout program execution (even compiler and assembly execution), whereas the register allocation burdens only compilation and assembly.

- *Input/output system*

The method of input/output control would have been a major compatibility problem were it not for the recognition of the distinction between logical and physical structures. Small machines use CPU hardware for I/O functions; large machines demand several independent channels, capable of operating concurrently with the CPU and with each other. Such large-machine channels often each contain more components than an entire small system.

Channel instructions. The logical design considers the channel as an independently operating entity. The CPU program starts the channel operation by specifying the beginning of a channel program and the unit to be used. The channel instructions, specialized for the I/O function, specify storage blocks to be read or written, unit operations, conditional and unconditional branches within the channel program, etc. When the channel program ends, the CPU program is interrupted, and complete channel and device status information are available.

An especially valuable feature is *command chaining*, the ability of successive channel instructions to give a sequence of different operations to the unit, such as SEARCH, READ, WRITE, READ FOR CHECK. This feature permits devices to be reinstructed in very short times, thus substantially enhancing effective speed.

Standard interface. The generalization of the communication between the central processing unit and an input/output device has yielded a channel which presents a standard interface to the device control unit. This interface was achieved by making the channel design transparent, passing not only data, but also control and status

information between storage and device. All functions peculiar to the device are placed in the control unit. The interface requires a total of 29 lines and is made independent of time through the use of interlocking signals.

Implementation. In small models, the flow of data and control information is time-shared between the CPU and the channel function. When a byte of data appears from an I/O device, the CPU is seized, dumped, used and restored. Although the maximum data rate handled is lower (and the interference with CPU computation higher) than with separate hardware, the function is identical.

Once the channel becomes a conceptual entity, using time-shared hardware, one may have a large number of channels at virtually no cost save the core storage space for the governing control words. This kind of *multiplex channel* embodies up to 256 conceptual channels, all of which may be concurrently operating, when the total data rate is within acceptable limits. The multiplexing constitutes a major advance for communications-based systems.

Conclusion

This paper has shown how the design features were chosen for the logical structure of the six models that comprise the IBM System/360. The rationale has been given for the adoption of the data formats, the instruction set, and the input/output controls. The main features of the new machine organization are its general-purpose utility for many types of data processing, the new approaches to large-capacity storage, and the machine-language compatibility among the six models.

The contributions discussed in this paper may be summarized as follows:

1. The relative independence of logical structure and physical realization permits efficient implementation at various levels of performance.

2. Tasks that are common to operating a system for most applications require a complement of instructions and system functions that may serve as a base for the addition of application-oriented functions.

3. The formats, instructions, register assignment, and over-all functions such as protection and interruption of a computer can be so defined that they apply to many levels of performance and that they permit diverse specialization for particular applications.

It is hoped that the discussions of these design features will shed some light on the present and future needs of data processing system organization.

Appendices

The design resulting from the decision process sketched above is tabulated in five appendices showing formats, data and instruction codes, storage assignments and interruption action. (*Appendices 1 through 5 appear on the following four pages.*)

Acknowledgments

The implementation of System/360 depends upon diverse developments by many colleagues. The most important of these developments were glass-encapsulated semi-integrated semiconductor components, printed circuit backpanels and interconnections, new memories, read-only storages and microprogram techniques, new I/O devices, and a new level and approach to software support.

The scope of the compatibility objective and of the whole System/360 undertaking was largely due to B. O. Evans, Data Systems Division Vice-President—Development.

References

1. D. W. Sweeney, "An Analysis of Floating-Point Addition and Shifting," to be published in the *IBM Systems Journal.*
2. See, for example, R. S. Barton, "A New Approach to the Functional Design of a Digital Computer," *Proc. WJCC* 19, 393–396 (1961).
3. F. P. Brooks, Jr., "Recent Developments in Computer Organization," *Advances in Electronics* 18, 45–64 (1963).
4. G. A. Blaauw, "Indexing," in *Planning a Computer System,* W. Buchholz, ed., McGraw-Hill Book Company Inc., 1962, pp. 150–178.

Received January 21, 1964

Appendix 1 **All operation codes are shown in the following table. The 8-bit codes are grouped by the main classes, such as fixed-point arithmetic, floating-point arithmetic and logical operations. The codes are furthermore grouped according to the five main instruction formats RR (register-register), RX (register-indexed storage location), RS (register-storage), SI (storage-immediate information) and SS (storage-storage).**

OPERATION CODES

FORMAT	RR	RR	RR	RR
CLASS	BRANCHING AND STATUS SWITCHING	FIXED-POINT FULLWORD, AND LOGICAL	FLOATING-POINT LONG	FLOATING-POINT SHORT
xxxx	0000xxxx	0001xxxx	0010xxxx	0011xxxx
0000		LOAD POSITIVE	LOAD POSITIVE	LOAD POSITIVE
0001		LOAD NEGATIVE	LOAD NEGATIVE	LOAD NEGATIVE
0010		LOAD AND TEST	LOAD AND TEST	LOAD AND TEST
0011		LOAD COMPLEMENT	LOAD COMPLEMENT	LOAD COMPLEMENT
0100	SET PROGRAM MASK	AND	HALVE	HALVE
0101	BRANCH AND LINK	COMPARE LOGICAL		
0110	BRANCH ON COUNT	OR		
0111	BRANCH/CONDITION	EXCLUSIVE OR		
1000	SET KEY	LOAD	LOAD	LOAD
1001	INSERT KEY	COMPARE	COMPARE	COMPARE
1010	SUPERVISOR CALL	ADD	ADD N	ADD N
1011		SUBTRACT	SUBTRACT N	SUBTRACT N
1100		MULTIPLY	MULTIPLY	MULTIPLY
1101		DIVIDE	DIVIDE	DIVIDE
1110		ADD LOGICAL	ADD U	ADD U
1111		SUBTRACT LOGICAL	SUBTRACT U	SUBTRACT U

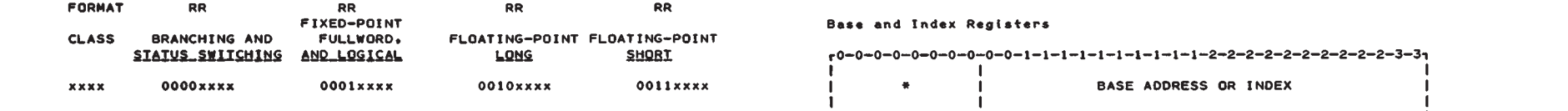
FORMAT	RX	RX	RX	RX
CLASS	FIXED-POINT HALFWORD AND BRANCHING	FIXED-POINT FULLWORD AND LOGICAL	FLOATING-POINT LONG	FLOATING-POINT SHORT
xxxx	0100xxxx	0101xxxx	0110xxxx	0111xxxx
0000	STORE	STORE	STORE	STORE
0001	LOAD ADDRESS			
0010	STORE CHARACTER			
0011	INSERT CHARACTER			
0100	EXECUTE	AND		
0101	BRANCH AND LINK	COMPARE LOGICAL		
0110	BRANCH ON COUNT	OR		
0111	BRANCH/CONDITION	EXCLUSIVE OR		
1000	LOAD	LOAD	LOAD	LOAD
1001	COMPARE	COMPARE	COMPARE	COMPARE
1010	ADD	ADD	ADD N	ADD N
1011	SUBTRACT	SUBTRACT	SUBTRACT N	SUBTRACT N
1100	MULTIPLY	MULTIPLY	MULTIPLY	MULTIPLY
1101		DIVIDE	DIVIDE	DIVIDE
1110	CONVERT-DECIMAL	ADD LOGICAL	ADD U	ADD U
1111	CONVERT-BINARY	SUBTRACT LOGICAL	SUBTRACT U	SUBTRACT U

Appendix 2 continued

from addressable registers and storage. The PSW is stored upon interruption. The Channel Command Word controls input/output operation and sequencing. The commands which may be given to the channel are listed as part of the table. The Channel Address Word is used to initiate input/output sequencing. The Channel Status Word indicates the channel status at the completion of an input/output operation or, when specified, during an I/O operation.

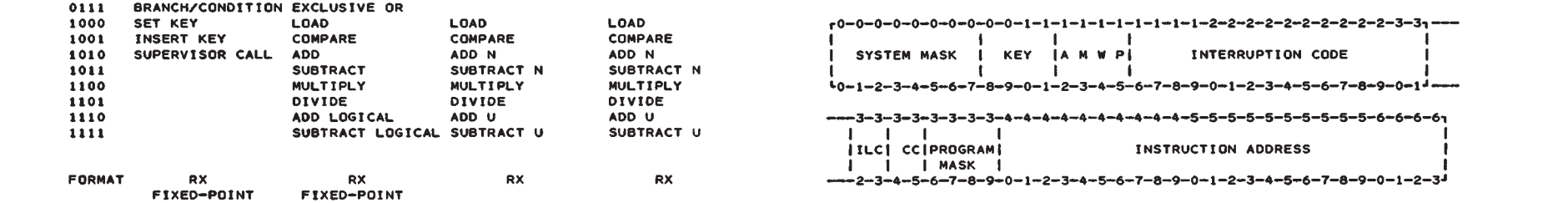
CONTROL WORD FORMATS

Base and Index Registers



0 - 7 Ignored
8 - 31 Base address or index

Program Status Word



0 - 7 System mask
- 0 Multiplexor channel mask
- 1 Selector channel 1 mask
- 2 Selector channel 2 mask
- 3 Selector channel 3 mask
- 4 Selector channel 4 mask
- 5 Selector channel 5 mask
- 6 Selector channel 6 mask
- 7 External mask

8 - 11 Protection key
12 ASCII mode (A)
13 Machine check mask (M)
14 Wait state (W)
15 Problem state (P)
16 - 31 Interruption code
32 - 33 Instruction length code (ILC)
34 - 35 Condition code (CC)
36 - 39 Program mask
- 36 Fixed-point overflow mask
- 37 Decimal overflow mask
- 38 Exponent underflow mask
- 39 Significance mask

40 - 63 Instruction address

VOL. 8, NO. 2, 1964, REPRINT

Appendix 1 continued

FORMAT CLASS	RS, SI BRANCHING, STATUS SWITCHING AND SHIFTING	RS, SI FIXED-POINT, LOGICAL, AND INPUT/OUTPUT		
xxxx	1000xxxx	1001xxxx	1010xxxx	1011xxxx
0000	SET SYSTEM MASK	STORE MULTIPLE		
0001		TEST UNDER MASK		
0010	LOAD PSW	MOVE		
0011	DIAGNOSE			
0100	WRITE DIRECT	AND		
0101	READ DIRECT	COMPARE LOGICAL		
0110	BRANCH/HIGH	OR		
0111	BRANCH/LOW-EQUAL	EXCLUSIVE OR		
1000	SHIFT RIGHT SL	LOAD MULTIPLE		
1001	SHIFT LEFT SL			
1010	SHIFT RIGHT S			
1011	SHIFT LEFT S			
1100	SHIFT RIGHT DL	START I/O		
1101	SHIFT LEFT DL	TEST I/O		
1110	SHIFT RIGHT D	HALT I/O		
1111	SHIFT LEFT D	TEST CHANNEL		

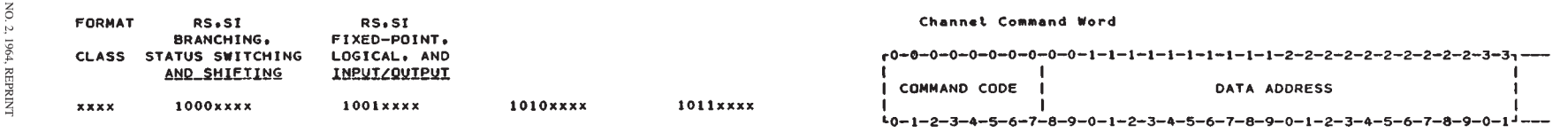
FORMAT CLASS		SS LOGICAL		SS DECIMAL
xxxx	1100xxxx	1101xxxx	1110xxxx	1111xxxx
0000				
0001		MOVE NUMERIC		MOVE W OFFSET
0010		MOVE		PACK
0011		MOVE ZONE		UNPACK
0100		AND		
0101		COMPARE LOGICAL		
0110		OR		
0111		EXCLUSIVE OR		
1000				ZERO AND ADD
1001				COMPARE
1010				ADD
1011				SUBTRACT
1100		TRANSLATE		MULTIPLY
1101		TRANSLATE AND TEST		DIVIDE
1110		EDIT		
1111		EDIT AND MARK		

Legend N = Normalized U = Unnormalized
SL = Single logical S = Single
DL = Double logical D = Double

Appendix 2 **The formats of all control words required for CPU and channel operation are shown in the following table. The base and index registers provide 24 bits of address and are specified by the B and X fields of instructions. The Program Status Word controls instruction sequencing and indicates the complete CPU status apart**

Appendix 2 continued

Channel Command Word

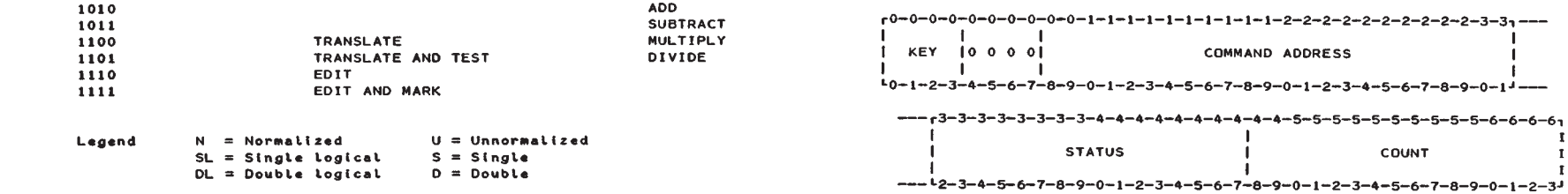


0 - 7 Command code
8 - 31 Data address
32 - 36 Command flags
32 Chain data flag
33 Chain command flag
34 Suppress length indication flag
35 Skip flag
36 Program-controlled interruption flag
37 - 39 Zero
40 - 47 Ignored
48 - 63 Count

Channel Address Word

0 - 3 Protection key
4 - 7 Zero
8 - 31 Command address

Channel Status Word



0 - 3 Protection key
4 - 7 Zero
8 - 31 Command address
32 - 47 Status
32 Attention
33 Status modifier
34 Control unit end
35 Busy
36 Channel end
37 Device end
38 Unit check
39 Unit exception
40 Program-controlled interruption

(*continued overleaf*)

Appendix 2 continued

```
   41    Incorrect length
   42    Program check
   43    Protection check
   44    Channel data check
   45    Channel control check
   46    Interface control check
   47    Chaining check
48 - 63 Count
```

Appendix 3 **All permanently assigned storage locations are shown in this table. These locations are addressed by the CPU and I/O channels during initial program loading, during interruptions and in order to update the timer. During initial program loading 24 bytes are read from a specified input device into locations 0 to 23. This information is subsequently used as CCW's to specify the locations of further input information and as a PSW to control CPU operation after the loading operation is completed. During an interruption the current PSW is stored in the "old" location and the PSW from the "new" location is obtained as the next PSW. The timer is counted down and provides an interrupt when zero is passed. All permanently assigned locations may also be addressed by the program.**

PERMANENT STORAGE ASSIGNMENT

	ADDRESS	LENGTH	PURPOSE
0	0000 0000	double word	Initial program loading PSW
8	0000 1000	double word	Initial program loading CCW1
16	0001 0000	double word	Initial program loading CCW2
24	0001 1000	double word	External old PSW
32	0010 0000	double word	Supervisor call old PSW
40	0010 1000	double word	Program old PSW
48	0011 0000	double word	Machine old PSW
56	0011 1000	double word	Input/output old PSW
64	0100 0000	double word	Channel status word
72	0100 1000	word	Channel address word
76	0100 1100	word	Unused
80	0101 0000	word	Timer
84	0101 0100	word	Unused
88	0101 1000	double word	External new PSW
96	0110 0000	double word	Supervisor call new PSW
104	0110 1000	double word	Program new PSW
112	0111 0000	double word	Machine new PSW
120	0111 1000	double word	Input/output new PSW
128	1000 0000		Diagnostic scan-out area*

---

* The size of the diagnostic scan-out area depends upon the particular model and I/O channels.

Appendix 4 continued

Legend

available	Unit and channel available
busy	Unit or channel busy
carry	A carry out of the sign position occurs
complete	Last result byte nonzero
CSW ready	Channel status word ready for test or interruption
CSW stored	Channel status word stored
equal	Operands compare equal
F	Fullword
g zero	Result is greater than zero
H	Halfword
halted	Data transmission stopped. Unit in halt-reset mode
high	First operand compares high
incomplete	Nonzero result byte; not last
L	Long precision
l zero	Result is less than zero
low	First operand compares low
mixed	Selected bits are both zero and one
not oper	Unit or channel not operational
not working	Unit or channel not working
not zero	Result is not all zero
one	Selected bits are one
overflow	Result overflows
S	Short precision
stopped	Data transmission stopped
working	Unit or channel working
zero	Result or selected bits are zero

Note

The condition code also may be changed by LOAD PSW, SET SYSTEM MASK, DIAGNOSE, and by an interruption.

Appendix 5 **All interruptions which may occur are shown in the following table. Indicated here are the code in the old PSW which identifies the source of the interruption, the mask bits which may be used to prevent an interruption, and the manner in which instruction execution is affected. The instruction to be performed next if the interruption had not occurred is indicated in the instruction address field of the old PSW. The length of the preceding instructions, if available, is shown in the instruction length code, ILC, as is further detailed in the table.**

INTERRUPTION ACTION

INTERRUPTION SOURCE IDENTIFICATION	INTERRUPTION CODE MASK PSW BITS 16-31	ILC BITS SET		INSTRUCTION EXECUTION
Input/Output (old PSW 56, new PSW 120, priority 4)				
Multiplexor channel	00000000 aaaaaaaa	0	x	complete
Selector channel 1	00000001 aaaaaaaa	1	x	complete
Selector channel 2	00000010 aaaaaaaa	2	x	complete

VOL. 8, NO. 2, 1964, REPRINT

Appendix 4 **All instructions which set the condition code (bits 32 and 33 of the PSW) are listed in the following table. All other instructions leave the condition code unchanged. The condition code determines the outcome of a** BRANCH ON CONDITION **instruction. The four-bit mask contained in this instruction specifies which code settings will cause the branch to be taken.**

CONDITION CODE SETTING

	0	1	2	3
Fixed-Point Arithmetic				
ADD H/F	zero	l zero	g zero	overflow
ADD LOGICAL	zero	not zero	zero,carry	carry
COMPARE H/F	equal	low	high	--
LOAD AND TEST	zero	l zero	g zero	--
LOAD COMPLEMENT	zero	l zero	g zero	overflow
LOAD NEGATIVE	zero	l zero	--	--
LOAD POSITIVE	zero	--	g zero	overflow
SHIFT LEFT DOUBLE	zero	l zero	g zero	overflow
SHIFT LEFT SINGLE	zero	l zero	g zero	overflow
SHIFT RIGHT DOUBLE	zero	l zero	g zero	--
SHIFT RIGHT SINGLE	zero	l zero	g zero	--
SUBTRACT H/F	zero	l zero	g zero	overflow
SUBTRACT LOGICAL	--	not zero	zero,carry	carry
Decimal Arithmetic				
ADD DECIMAL	zero	l zero	g zero	overflow
COMPARE DECIMAL	equal	low	high	--
SUBTRACT DECIMAL	zero	l zero	g zero	overflow
ZERO AND ADD	zero	l zero	g zero	overflow
Floating-Point Arithmetic				
ADD NORMALIZED S/L	zero	l zero	g zero	overflow
ADD UNNORMALIZED S/L	zero	l zero	g zero	overflow
COMPARE S/L	equal	low	high	--
LOAD AND TEST S/L	zero	l zero	g zero	--
LOAD COMPLEMENT S/L	zero	l zero	g zero	--
LOAD NEGATIVE S/L	zero	l zero	--	--
LOAD POSITIVE S/L	zero	--	g zero	--
SUBTRACT NORMALIZED S/L	zero	l zero	g zero	overflow
SUBTRACT UNNORMALIZED S/L	zero	l zero	g zero	overflow
Logical Operations				
AND	zero	not zero	--	--
COMPARE LOGICAL	equal	low	high	--
EDIT	zero	l zero	g zero	--
EDIT AND MARK	zero	l zero	g zero	--
EXCLUSIVE OR	zero	not zero	--	--
OR	zero	not zero	--	--
TEST UNDER MASK	zero	mixed	--	one
TRANSLATE AND TEST	zero	incomplete	complete	--
Input-Output Operations				
HALT I/O	not working	halted	stopped	not oper
START I/O	available	CSW stored	busy	not oper
TEST CHANNEL	not working	CSW ready	working	not oper
TEST I/O	available	CSW stored	working	not oper

Appendix 5 continued

Selector channel 3	00000011 aaaaaaaa	3	x		complete
Selector channel 4	00000100 aaaaaaaa	4	x		complete
Selector channel 5	00000101 aaaaaaaa	5	x		complete
Selector channel 6	00000110 aaaaaaaa	6	x		complete
Program (old PSW 40, new PSW 104, priority 2)					
Operation	00000000 00000001		1,2,3		suppress
Privileged operation	00000000 00000010		1,2		suppress
Execute	00000000 00000011		2		suppress
Protection	00000000 00000100		0,2,3		suppress/terminate
Addressing	00000000 00000101		0,1,2,3		suppress/terminate
Specification	00000000 00000110		1,2,3		suppress
Data	00000000 00000111		2,3		terminate
Fixed-point overflow	00000000 00001000	36	1,2		complete
Fixed-point divide	00000000 00001001		1,2		suppress/complete
Decimal overflow	00000000 00001010	37	3		complete
Decimal divide	00000000 00001011		3		suppress
Exponent overflow	00000000 00001100		1,2		terminate
Exponent underflow	00000000 00001101	38	1,2		complete
Significance	00000000 00001110	39	1,2		complete
Floating-point divide	00000000 00001111		1,2		suppress
Supervisor Call (old PSW 32, new PSW 96, priority 2)					
Instruction bits	00000000 rrrrrrrr		1		complete
External (old PSW 24, new PSW 88, priority 3)					
External signal 1	00000000 xxxxxxx1	7	x		complete
External signal 2	00000000 xxxxxx1x	7	x		complete
External signal 3	00000000 xxxxx1xx	7	x		complete
External signal 4	00000000 xxxx1xxx	7	x		complete
External signal 5	00000000 xxx1xxxx	7	x		complete
External signal 6	00000000 xx1xxxxx	7	x		complete
Interrupt key	00000000 x1xxxxxx	7	x		complete
Timer	00000000 1xxxxxxx	7	x		complete
Machine Check (old PSW 48, new PSW 112, priority 1)					
Machine malfunction	00000000 00000000	13	x		terminate

Legend

a Device address bits
r Bits of R1 and R2 field of SUPERVISOR CALL
x Unpredictable

INSTRUCTION LENGTH RECORDING

INSTRUCTION LENGTH CODE	PSW BITS 32-33	INSTRUCTION BITS 0-1	INSTRUCTION LENGTH	INSTRUCTION FORMAT
0	00		Not available	
1	01	00	One halfword	RR
2	10	01	Two halfwords	RX
2	10	10	Two halfwords	RS or SI
3	11	11	Three halfwords	SS

Gene M. Amdahl

B.S. in Physics 1948, South Dakota State College; M.S. in Physics, 1949, and Ph.D., 1952, University of Wisconsin. Between 1952 and 1956 he was associated with the IBM Development Laboratory at Poughkeepsie. During the period he was project engineer and systems designer of the IBM 704 and STRETCH computers. From 1956 to 1960 Dr. Amdahl worked at the Ramo Wooldridge Corporation and at Aeronutronic Systems, Inc., where he was manager of Data Processing Engineering. He rejoined IBM in Research in 1960 as Director of Experimental Machines. He has been Advanced Systems Design Manager in the Data Systems Division, Poughkeepsie, and is currently Technical Manager of Large Scientific Systems.

Gerrit A. Blaauw

B.S. in Electrical Engineering, Lafayette College, 1948; Ph.D. in Applied Science, Harvard University, 1952. While at Harvard, was a member of the staff of the Computation Laboratory and participated in the design of the Mark III and Mark IV calculators. From 1952–1955 was a member of the staff of the Mathematics Center in Amsterdam, Netherlands, where he cooperated in the design of the ARRA and FERTA computers. Joined IBM in 1955 at the Poughkeepsie Product Development Laboratory. He has been engaged in the systems planning of various machine projects in Data Systems Division. Is a member of the ACM, IEEE, and Sigma Xi.

Frederick P. Brooks, Jr.

A.B. in Physics, Duke University, 1953. Received S.M., 1955 and Ph.D., 1956, Harvard University, for graduate work in design and application of computers. From 1956 to 1959, Dr. Brooks participated in the planning of the STRETCH and HARVEST computers at the IBM Product Development Laboratory in Poughkeepsie. From 1959 to 1960 he studied computer organization theory at IBM Yorktown Research Center. At present he is IBM Processor Manager, in Poughkeepsie, and is responsible for the specification and development of new computers. Member of IEEE, ACM, Phi Beta Kappa, and Sigma Xi.