Computer Architecture

Rechner-Architektur

Elektron. Rechenanl. 14 (1972), H. 4, S. 154–159
Manuskripteingang: 12. 5. 1972

von G. A. BLAAUW, Twente University of Technology, Enschede, The Netherlands

In this paper overall design principles of computer architecture are considered. The concept of architecture is defined and contrasted with the implementation and realization of the computer. The realm, documentation and language of the architecture are considered briefly. Subsequently the principle of consistency and the derived principles of orthogonality, propriety and generality are discussed as they pertain to the quality of a design. Since the concept of architecture is not limited to computers, examples from other machines are also used.

In dieser Arbeit werden die Gesamtentwurfsgrundsätze der Rechner-Architektur betrachtet. Der Begriff der Architektur wird definiert und der Implementierung und Realisierung von Rechenanlagen gegenübergestellt. Der Bereich der Architektur, ihre Dokumentation und ihre Sprache werden kurz betrachtet. Anschließend werden der Grundsatz der Konsistenz und die davon abgeleiteten Grundsätze der Orthogonalität, der Angebrachtheit und der Allgemeinheit diskutiert und wie sie zur Qualität eines Entwurfes beitragen. Da der Begriff der Architektur nicht auf Rechenanlagen beschränkt ist, werden auch Beispiele aus anderen Gebieten verwendet.

Three steps of design

In computer design three levels can be distinguished: architecture, implementation and realisation; for the first of them, the following working definition is given: *The architecture of a system can be defined as the functional appearance of the system to the user, its phenomenology.*

Although the term *architecture* was introduced only ten years ago in computer technology (*Buchholz*), the concept of architecture is as old as the use of mechanisms by man. When a child is taught to look at a clock, it is taught the architecture of the clock. It is told to observe the position of the short and the long hand and to relate these to the hours and the minutes. Once it can distinguish the architecture from the visual appearance, it can tell time as easily from a wrist watch as from the clock on the the church tower.

The inner structure of a system is not considered by the architecture: we do not need to know what makes the clock tick, to know what time it is. This inner structure, considered from a logical point of view, will be called the *implementation*, and its physical embodyment the *realisation*.

Implementation and realisation are not the subject of this paper. However, since they complement the architecture, their definition, realm, documentation, aim and procedure will be discussed briefly at the outset. The corresponding characteristics of the architecture will subsequently be discussed, with the major emphasis on design principles.

Implementation

The *implementation* is the logical structure which performs the architecture. Where the architecture tells *what* happens, the implementation describes *how* it is made to happen. For the above mentioned clock, for instance, powering could be achieved through water action, springs, weights, atmospheric pressure, electric current, or body movement, while accuracy could be obtained through a balance, pendulum, crystal, tuning fork, or the period of the electric net.

For the computer, the implementation possibilities are the choices in space and time expressed by the adjectives: serial and parallel; consecutive and anticipatory; fixed and adaptive; polling and interrupting; stored and decoded; and private and shared. The list illustrates the large repertoire of options at the disposal of the implementer. The repertoire has a timeless quality; most alternatives are valid for every computer generation.

Designing to a given architecture is not a real limitation to the implementer. The watchmaker does not feel frustrated by the conventions of the dial. On the contrary, the options mentioned above show that the realm of his design contains many degrees of freedom. Similarly the logical designer need not feel constrained by the architecture. His more natural limitation is complexity, which bounds the extensions in space and time and results in the ultimate expense of thrift and slowness of haste. An example of the first is the minimal micro-coded machine, which requires too much storage space and time. An example of the latter is the over-ambitious look ahead design which guesses too boldly and hence wastes time in correcting its misjudgments.

The result of the implementation, the logical design, is traditionally shown as a series of block diagrams. These blocks represent in effect a series of statements. Actually, a direct presentation of these statements is more suitable and, although, less familiar, more easily understood. The Harvard Mark IV was to large degree designed and described by such statements, as has been the case with several subsequent developments.

The aim of the implementer is an optimal, that is minimal, cost-performance ratio for a given application and performance range. An indication of the quality of the design is the scope of the cost and performance considerations. Rather than local optimization against a given instruction mix all ramifications of cost and performance should be taken into account early during the design. They include the cost of items like education, maintenance and operating environment and their effect upon design decisions, such as error neglection, detection, or correction.

Realisation

The physical structure, which embodies the logical design, will be called the realisation. It is often considered part of the implementation. Here, however, the '*which*' and '*where*' of component selection, allocation, placement and connection will be considered separate from the '*how*' of the logical structure.

An example of machines with very similar implementations, but quite different realisations, are the IBM 709 (1959) and 7090 (1960). The latter is basically a transistorised version of the former.

The logical design can make significant use of computer aided design, often in the form of a conversational system (*Blaauw*, 1971). The most prominent tool of the realisation is design automation, usually in batch processing form. Here the attention of the designer has shifted from the design itself, to the tools which he uses in designing. Design automation can successfully complete the task of component placement and wire routing, and promises to be increasingly effective as better algorithms are developed.

A good realisation should be manufacturable and maintainable. Its emphasis is therefore upon achieving a proper package and using it to its best advantage.

Design boundaries

The architectural definition and the logical design each act as an interface. They subdivide the overall design problem and permit a simultaneous effort on all fronts. This procedure of divide and conquer is extended within each design area.

Setting up fences between architecture, implementation and realisation does not prevent the designers from looking from one area to the other. In fact: "good fences make good neighbours". Each area can influence the others significantly. However, recognising clearly how one area affects another should lead to greater freedom of design for all concerned.

The policy of dividing through interfaces has been a major key to conquering the hardware design process and making it predictable in time and result. For software design, the ability to define proper interfaces is equally important.

An every day example of a division by standard interface is the electric plug and outlet connection. It should be noted that neither plug nor outlet constitutes the interface, but the electrical and mechanical definition, to which both adhere. The example also points to the gain in modularity and its attendant freedom of configuration and future development. This interface has a parallel in the input/output interface of computer systems. Less public, and hence less standard, are the interfaces in the central unit, and the connections to storage and channels.

Realm of architecture

Each time an interface is established its definition constitutes an architecture. It therefore is necessary to ascertain with what reference the term *architecture* is used. For computers, it may, for instance, apply to an operating system, a programming language, the machine language, the microcode language, or the specification of a storage unit. To avoid confusion, the term is used here only to refer to the machine language.

In the definition of architecture its realm was restricted to the functional appearance, rather than the visual appearance. The covers and colors of the industrial design are not part of the machine architecture, nor are the cost/performance of the implementation, nor the manufacturability and maintainability of the realisation, even though the user is certainly aware of these.

There are of course many borderline cases. A typical example is the behaviour of the system in the presence of a component fault. The source of the malfunction is the realisation. The user, however, must be made aware of this via the architecture, for instance, through a program interruption. Further, remedial action may be necessary via diagnostic operations specified in the machine language. It is academic to determine the formal boundary of the architecture in a case like this. Instead, the architectural documentation should state clearly in each case where the boundaries of the architecture are drawn. As an example, the operation 'Diagnose' of System /360 was defined alike for all implementations in its format, while its diagnostic action was declared to be implementation-dependent.

It is essential that the boundaries drawn around the architecture are realistic. Eventually, as will be shown, it is the user, not the designer, who determines the limits of the architecture.

Documentation of architecture

There always is an architecture, whether it is defined in advance – as with modern computers – or found out after the fact – as with many early machines. For architecture is determined by behaviour, not by words. Therefore, the term *architecture*, which rightly implies the notion of the *arch*, or prime, structure, should not be understood as the vague overall idea. Rather, the product of the computer architect, the principle of operations manual, should contain all detail which the user can know, and sooner or later is bound to know. For, even if a programmer may be forgiving, his programs are merciless in exposing every detail.

As an illustration, consider the many details which the public gets to know about an elevator system. Very quickly one becomes aware whether multiple elevator banks are coupled in their control, to what extent calls are remembered, when the direction of motion is changed, what governs the closing of doors, and what happens upon overload.

A computer example is provided by the IBM 1401 (1960). Its manual only mentioned the permissable operations. However, the unmentioned functions, as for instance the actions which resulted from the spare operation codes, were soon found out, and some proved to be very convenient. As a consequence, in emulating the 1401, an addendum to the manual of about a hundred items had to be included in the design, since this in effect was part of the architecture. In the IBM System /360 (1964) this problem was recognised in advance (*Amdahl*). A mere statement against using spare codes was felt to be insufficient. Hence the architecture specified a program interruption upon the use of these spares. This so-called *policing* is an effective means of safeguarding the architectural intention.

The necessity for a complete documentation is not only important to the user. The builder, the designer of the implementation, also must know how every detail is to be treated. Many seemingly small architectural requirements may greatly affect the implementation. Ignoring these details early in the design may involve disproportionate large costs as the design nears completion.

In System /360 it was originally overlooked that, in conflict with the specification, the last adapter on an input/output channel needed to be switched on whenever the channel was

used. This restriction arose out of the definition of the logical zero for the interconnecting circuits. On paper the statement may appear small, in a functioning reality the nuisance is major. Remedying this mistake was very costly, as it had to be done when the system was already in the field.

Specification language

A proper definition should be extensively verified by simulation. This emphasizes the importance of the language used to describe the architecture. The use of written English has been conventional and is indispensable. Its main advantage is the large audience which it reaches, including, besides the immediately involved system programmer and logical designer, the less immediately, but not less vitally, involved people in education, maintenance machine operation, sales and management.

However, written English, with its ease of expression lacks the rigor to control. It must be redundant to be understood, and in fact, to be believed. If, for instance, the architecture specifies that the sign of a product is determined by the signs of multiplier and multiplicand, the experienced manual reader immediately raises the question of the sign of the product of zero and a negative number. To forestall such questions, a written manual soon becomes abundant in words and stylized in expression, using terms in carefully defined senses. Thus it comes to resemble the carefully crafted wording of guarantee statements, contracts and laws.

Because of these difficulties a more rigorous irredundant algorithmic language, such as APL (*Iverson*), is desirable. An algorithmic description has its own disadvantages, such as the unfamiliarity of the symbols and the catastrophic effect of each error. It therefore is desirable to use both methods side by side. Possible misconceptions in the text can be eliminated by the expressions, while the expressions in turn are explained by the text. Each description, however, should be complete in itself. This aids in answering questions about the architecture. In turn, the process of making the algorithmic description poses many useful questions to the architect.

To the implementer an algorithmic description of the architecture is extremely helpful in assuring the correctness of his design. The process of logical design can be considered as a translation from the language of the architecture into the language of the implementation statements. When this process is sufficiently controlled, the equivalence of the implementation and the architecture can to a large degree be ensured. Both the design process and the ultimate implementation may well reflect this desire for an early assurance of correctness.

A classical example of an algorithmic machine description is the APL description of System/360 (*Falkoff*). The description was completed concurrent with the system development and not actually used by the designers. Nevertheless, it showed, that a complex system can be described in all its details by a programming language.

Quality of architecture

Good architecture is consistent. That is, with a partial knowledge of the system the remainder of the system can be predicted. For example, the mere decision to incorporate a square root operation in an instruction list almost fully defines the operation. The data and instruction formats should be the same as for other arithmetic operations. Rounding, precision and significance should be handled as with other results. Even taking the square root of a negative number should yield a result similar to other exception cases, such as division by zero. An example of lack of consistency can be found in the Hollerith punched card code. Here knowledge of the codes for the letters A through R leads one to expect an S where the slash (/) is found.

In the design of System/360 the floating point operation 'halve' was added at a late moment. Because of implementation problems it was felt necessary to omit the post-normalisation. This lack of consistency with the other floating point operations made the function virtually useless. Soon after the machine was in use the design had to be corrected at this point.

Consistency is more frequently used than mentioned in human thinking. It tells us not to link what is independent, not to introduce what is immaterial, and not to restrict what is inherent, thus leading to orthogonality, propriety and generality. It is stimulating and self-teaching, because it confirms and encourages our expectation. Thus it provides a solution to the conflict between ease of use and ease of learning. Ease of learning requires a simple architecture, as with fixed point arithmetic; ease of use a more complex one, as with floating point arithmetic. By making one a subset of the other and both part of a consistent design, the user's comprehension of the architecture can grow naturally. Nevertheless, what is really consistent is not always evident. Proper human engineering, including observation and experiment, can be applied here with much profit.

Design principles

From the broad principle of consistency a number of other principles can be derived. The three main design principles to be discussed here, from which others in turn can be derived, are orthogonality, propriety and generality.

Orthogonality

The principle of keeping independent functions separate in their specification is called *orthogonality*. This term is borrowed from mathematics, where it denotes algebraic functions, which do not affect each other. Thus for a clock a set of functions might be:

a. visibility of the time in the dark, the lighted dial,
b. signaling at a preset time, the alarm, and
c. repeated signaling at intervals of several minutes, the slumber alarm.

Clearly functions a. and b. are independent of one another, while there is no point to function c. if b. is not present. Functions a. and c. are also independent and the principle of orthogonality would be violated, if the slumber alarm could only operate with lighted dial. Seeing the dial at night has nothing to do with wanting to rise at a slow pace.

In a program, the criterion of a decision is independent of the arrangement of the program in main- and sub-routines. Orthogonality is therefore violated, if 'branch on plus' is present, while 'branch on not-plus' is not available, as was the case in the Whirlwind (1950), and in similar ways in later machines, like the IBM 1401.

Orthogonality does not concern the absence or presence of a function. It may be entirely proper for a manufacturer to

offer a package deal, which consists of a clock with lighted dial and slumber alarm. In fact, this may be consistent, since a certain level of refinement at one point makes one anticipate refinement at other points too. What is required by orthogonality, is that independent functions, if provided, be invoked independently in use.

The IBM 7030, the Stretch computer (1960) (*Buchholz*), included three flag bits in the floating point format. These bits were logical variables tied to numeric quantities. This 'unequal yoke' proved unsatisfactory. Arithmetic was burdened with concern about flags, requiring 'load with flag' next to 'load', while the flag logic could be performed much better in separate arrays by the regular logical operations.

Symmetry

The orthogonality of branch criterion and branch direction is secured in most modern computers by a set of symmetric branch conditions. Thus, orthogonality leads to symmetry, whenever direction is an independent option. If the direction of travel of a locomotive is recognised as being independent of its ability to provide traction, a symmetric design results. The implementation of these functions may however favor an asymmetric design, as is clear from the steam locomotive.

In computers symmetry as a rule is not difficult to implement. However, a complete set of options proliferates operation codes and may require too much space in the machine formats. The symmetric branch operations require hardly any added logical circuits. The added operation codes, on the other hand, may not be readily available. Thus, in computers a lack of orthogonality results rather from the desire to avoid proliferation, than from the limitations of the implementation.

As is true for all principles, symmetry should not be made a goal in itself. The Stretch computer, which as the name indicates deliberately tried to stretch computer technology, provided all 16 connective functions of two binary variables. The symmetry of this set is apparent to every logician. The average user, however, is satisfied with the 'and', 'or' and 'exclusive-or' of propositional statements. Although this set is not much cheaper, it requires far fewer operation codes, which explains why only these connectives are found in System/360.

Propriety

Orthogonality is desirable, since the linking of independent functions introduces a constraint which is not proper to the functions as such. *Propriety*, that is the need for a function or feature to be proper to the essential requirements of the system, is in itself a major principle, which follows from consistency. Its opposite is *extraneousness*, the introduction of something strange to the purpose to be served. A typical example is the gear shift of a car, which also illustrates the main source of extraneousness: the implementation. Shifting gears is not proper to driving. It is required because of the limited power range of the piston engine.

Similarly, it may not be allowed to turn the hands of a clock backwards. Again this rule is due to the implementation of the clock and of no benefit to the user. In fact, many modern clocks permit time to be adjusted in both directions.

An example of extraneousness in computers is the sign of zero in 1-complement notation and in absolute-value-and-sign notation. In mathematics zero has no sign. The introduction of the sign results in a set of rules which do not bring the user any closer to solving his problems. Statements, such as: 'all zero results are positive' and 'plus zero equals minus zero', are required to prevent a behavior contrary to mathematical convention. One of the advantages of 2-complement notation is that only one zero is represented.

A comparison of the IBM 2938 (*Ruggiero*) and the Illiac IV (*Barnes*) shows how the implementation can enter the architecture. Both machines allow array operations.

The 2938 takes the architecture as a starting point. The Illiac, being a more experimental machine, starts with the implementation.

The 2938 operations are practically independent of operand size. In the Illiac, however, the user must fit his problem to the machine, a procedure which resembles the mythological Procrustean Bed.

Parsimony

Where orthogonality may lead to proliferation, propriety leads to parsimony. Parsimony expresses the thought that a function which is not germane to the system should not be present.

There appears to be a phase in each technological development cycle where there is a tendency to add bells and whistles. This is often the result of passing design decisions to the user, by giving him all the options. This is typically the case when the art is developed to the point where the options can be provided, but maturity is still lacking in evaluating their use.

The Stretch computer, for example, introduced the concept of variable byte size, which permitted characters of 1 through 8 bits to participate in arithmetic and logical operations. In System/360 this concept was dropped again. It generalised a problem, rather than providing a solution. The introduction of such a concept carries with it a series of problems which are strange to the user. For instance: how should bytes of unequal size be matched? If extended, what bits are supplied? If truncated, is the deletion of significant bits signalled?

Parsimony also says that a function should not be provided in two competing ways. Otherwise redundant knowledge and an unnecessary choice are forced upon the user. Since basically all computer operations but one are redundant (*van der Poel*), the introduction of more operations is only justified by such efficiency of expression that no competition exists. A typical example are the floating point operations, which are justified because they eliminate cumbersome scaling procedures.

Transparency

Another concept which results from propriety is transparency. Just as the glass in a window, which preserves the comfort of a home, should not alter the view throught the window, so the system functions introduced by an implementation should not impose themselves upon the user.

In dialing long distance, the user should not have to choose the channels and exchanges through which his call is forwarded. All he needs to specify is the number of the person he wants to call. The fact, that only a limited number of calls can be made simultaneously, should equally be hidden from him.

A computer example is the desire for visual fidelity on a terminal. What is keyed in by the user should be placed as such in storage and reproduced again unaltered.

Virtual system

When the actual limitations of the implementation are hidden in a transparent fashion, such that only the functions which are proper to the purposes of the user are present, a *virtual system* results. The automatic gear shift is such a virtual system. It presents to the driver a car without gear shift, even though the engine still requires this function. Because the virtual system pretends what is not actually true, it usually must admit its real limitation under stress. Thus dialing an area code can immediately result in a busy signal, even though the subscriber to be reached is capable of accepting the call. This reveals that the dialing system makes use of intermediate facilities, which may become overloaded.

In computers also a virtual system is desirable, but vulnerable. For instance, the architecture may specify a character-oriented storage, while actually storage is organised in multiples of characters. If this fact is not properly hidden by the logical designer, the programmer will use it in optimising his programs and thus reintroduce it into the architecture. A virtual system therefore carries with it the danger of spoiling the architecture by introducing extraneous implementation details.

A prominent example of a virtual function is virtual storage. Here the implementation pretends to offer a large amount of information on 'one level' (*Kilburn*), which actually resides on devices with different accessability – if space is allocated at all.

There are many minor examples of virtual operation. In fact, the inner structure of the computer has come to resemble the architecture less and less. With faster and cheaper components, the implementer can more and more effectively hide the extraneous implementation details, making the computer into an almost totally virtual machine. This also explains why most innovations in circuits and storage have only a very indirect effect upon the architecture (*Blaauw*, 1970).

Compatibility

Transparency concerns the removal from the user's concern of salient implementation characteristics. As a rule a transparent architecture is suitable to a wide range of implementations. Implementations which have the same architecture are called *compatible*. Thus an upright and a grand piano are compatible, since they have the same keyboard and pedals.

Computer compatibility is restricted to the architectural definition. Speed is therefore deliberately excluded from consideration. Programs which should run on any implementation may not be implicitly time-dependent. They also should be independent of variation allowed within the architecture, such as storage size, or system configuration.

Transparency and compatibility both should be exact. Half compatibility or transparency is really no compatibility or transparency at all. In this respect computers are required to be perfect – a very unusual requirement. It can only be met by stating clearly within which boundaries these attributes apply. The restrictions upon compatibility stated above give an indication of such boundaries.

Compatibility has been a major factor in establishing the need for an explicit computer architecture. With early computers the manner in which they worked – if they worked at all – was the architecture. Hence, there was no need to distinguish architecture and implementation. It subsequently proved that an architecture, however ad hoc, had a remarkably long life through successor machines. Therefore, as new machines were planned, their design had to fit a wide range of performance and size. Since programming investment also became a major factor, the need for a common architecture was clear.

Multiple compatible implementations also made it possible to enforce the architecture. With a single implementation, the paper words of the architecture are hard to uphold against deviations in a 'hard'-ware implementation. Time and money are all on the side of the latter. With multiple implementations, on the other hand, it is clear to the implementer that he can only deviate at his own risk. This explains why in System/360 the CPU architecture, which was implemented by five teams, was more tightly controlled than the peripheral device architecture, which often had only a single implementation.

Generality

Charles Babbages' first computer was the special purpose Difference Engine (ca. 1832). During the development of this machine he realised, that by generalizing his design, its application would be much wider. This consideration resulted in his second machine: the Analytical Engine (ca. 1867). Thus, generality from the start has proven to be a powerful architectural principle. In fact, the success of the modern computer has been mostly due to its general purpose character.

Generality is the ability to use a function for many purposes. It expresses the professional humility of the designer, that users will be inventive beyond his imagination and needs may change other than in his expectation. Two major ways in which generality can be achieved are open-endedness and completeness.

Open-endedness

As a safeguard towards future developments the designer is wise to leave spares in the spaces his design has created. Thus spare format bits and codes should be provided. One of the reasons for a departure from the successful IBM 701-7094II line (1953 – 1964) was the lack of address space provided in its instruction format. In System/360 this space was expanded a hundredfold, with easy extension by another factor of 256. Within a few years, serious demands for this added space were made in the model 67 (1966).

Open-endedness also requires that the user have full access to the information with which he must work. Hence, there should be no configurations which have an irrepressible delimiting or control function. For instance, the IBM 705 (1955) used a group mark (code 11 111) to stop data transfer in a write operation. As a result arbitrary codes, or binary information, could not be handled by the machine.

This requirement leads to separation of data and control information, which in turn affects an entire computer design. The formats of the Burroughs 5000 (1962) family are a prominent example of mixing data and control information. Such a design implies a joint hardware-software development and requires a high level of assurance in both areas. In this case this challenge was met deliberately, and with success. However, too often the opportunity to safeguard a design by generality is passed by through sloppiness.

In short, the designer should not limit the use of a general function by his own notions about its use, until they are widely accepted as the superior way of doing things. In the absence of knowledge freedom should be provided.

Completeness

Generality can at times be met by completeness. Not just a selection of functions, but all functions of a given class are provided. However, this may introduce the problem of proliferation. The full range of choices may result in an abundance of control and excess of format space. Thus, a variable field length instruction in the Stretch computer required 64 bits, even though the byte address was only 21 bits song. 33 Bits were required by bit-addressing, field length, sign and radix control, indexing, index mode control, byte size selection and a relative shift, all of which applied to all operations of this class.

Proliferation may be prevented by localising the options through decomposition. Instead of multiplying all cases by the given set of options, only one basic function may be so expanded. The screwdriver set, which includes several handles and a variety of bits which can fit in each of these handles, is such a solution. It requires far fewer parts than a full set of complete screwdrivers.

In contrast to Stretch, in System/360 control of operand sign was delegated to the 'load' operations. Thus only one operation was expanded fourfold, instead of so expanding all arithmetic operations. Similary, indirect addressing was provided by loading an index, instead of including this option with each address. These decisions also improved the efficiency of expression.

Conclusion

The architecture of a system creates a world in which other people in turn can be creative. Thus *Steinway* and his predecessors introduced a world which made the compositions of *Chopin* possible, which in turn allowed pianists like *Rubinstein* to be creatively active. In the world created by the computer architect, programmers and implementers must be able to work creatively. If the architecture is well designed, there will be the freedom to do so.

The principles which lead to a good design are not peculiar to computers, as has been illustrated by the examples drawn from other fields of engineering. In fact, an even wider scope could have been taken, since the principle of consistency also applies to other areas of creative endeavor, as illustrated by the laws of drama concerning unity of time, place and person, and by *Ockham's* famous razor.

Although the principles discussed are not mathematically defined as such, they nevertheless have sufficient precision to be used and to identify their consequences. Violating them may result in a minor nuisance, or in a major expenditure in time and cost, even involving redesign. On the other hand, the adherence to good design principles results in equipment which is attractive to learn and to use, and which can be build and operated effectively.

Acknowledgements

The thoughts expressed in this paper have resulted from many discussions with colleagues at the Technische Hogeschool Twente and at the IBM Development Laboratories. Of these I am most indebted to Dr. *F. P. Brooks* for his contributions in substance and wording.

Literature

Amdahl, G. M., G. A. Blaauw, and *F. P. Brooks,* Jr., Architecture of the IBM System/360. IBM Journal of Research and Development, vol. 8, no. 2, pp. 87–101, April 1964.

Barnes, G. H., R. M. Brown, M. Kats, D. J. Kuck, D. L. Slotnick, and *R. A. Stokes,* The ILLIAC IV Computer. IEEE Transactions on Computers, vol. 17, no. 8, pp. 746–757 (1968).

Blaauw, G. A., Hardware Requirements for the Fourth Generation, in: Fourth Generation Computers: User Requirements and Transition, ed. *F. Gruenberger.* Prentice-Hall, Inc., Englewood Cliffs, N. J., 1970.

Blaauw, G. A., The use of APL in computer design, in: MC-25 Informatica Symposium, Mathematical Centre Tracts 37. Mathematisch Centrum, Amsterdam, 1971.

Buchholz, W., ed., Planning a Computer System. McGraw-Hill, New York, N. Y., 1962.

Falkhoff, A. D., K. E. Iverson, and *E. H. Sussenguth,* A formal description of System/360. IBM Systems Journal, vol. 3, no. 3, pp. 198–261, 1964.

Iverson, K. E., A Programming Language. Wiley, New York, N. Y., 1962.

Kilburn, T., R. B. Payne, M. J. Lanigan, and *F. H. Summer,* One level storage system. IRE Transactions on Electronic Computers, vol. 11, no. 2, pp. 223–235, (1962).

Ruggiero, J. F. and *D. A. Corryell,* An auxiliary processing system for array calculations. IBM Systems Journal, vol. 8, no. 2, pp. 118–135 (1969).

Van der Poel, W. L., The logical principles of some simple computers. Amsterdam (1956).