The Persistence of the Classical Computer Architecture

A Survey from 1950 to the Present*

G. A. Blaauw

INTRODUCTION

Before we can talk about *computer architecture* we must state what we mean with these two terms. We then observe that at an early date a classical computer architecture emerged which was challenged time and again, but still persists until the present. In our historical survey we try to answer the question why this is so.

WHAT IS COMPUTER ARCHITECTURE?

The *architecture* of a computer system we define as *its functional appearance to its immediate user,* that is, its conceptual structure and functional behavior as seen by one who programs in machine language. A computer's architecture is by this definition distinguished from other domains of computer design:

> the logical organization of its data flow and controls, called the *implementation,*
> the physical structure embodying the implementation, called the *realization.*

Architecture concerns the *function* that is provided to the programmer, such as addressing, addition, interruption, and input/output. Implementation concerns the *method* which is used to achieve this function, perhaps a parallel data path and a microprogrammed control. Realization concerns the means used to materialize this method, such as electrical, magnetic, mechanical, and optical devices and the powering and packaging for them. Therefore the realization also includes the visual appearance, the industrial design, of the computer. Figure 1 summarizes the distinctions among architecture, implementation, and realization.

* Symposium on Computational Engines, CWI, Amsterdam, September 14, 1989. The material of this presentation is from *Computer Architecture* by G.A. Blaauw and F.P. Brooks Jr, to be published by Addison Wesley and is used with permission.

Design domain	Question	Description of
Architecture	What?	Function
Functional appearance		
(to the system programmer)		
Implementation	How?	Method
Logical structure		
(performs the architecture)		
Realization	Which?	Means
Physical structure		
(enbodies the implementation)		

FIGURE 1. Domains of computer design.

ARCHITECTURE AS A GENERAL DESIGN CONCEPT

It clarifies concepts to distinguish among the architecture, the implementation, and the realization aspects of all kinds of designs: buildings, bridges, airplanes, appliances, cars, computer programs, operating systems, and computers themselves. To appreciate these distinctions, let us consider clocks. When children are taught to tell time, they are taught the architecture of the analog clock: the dial divided into twelfths and sixtieths, the short hand that goes around twice a day, and the long hand that goes twelve times as fast. They learn first to distinguish the hands from each other and then to relate their positions to the hours and minutes. Equally important, they learn to ignore all aspects of the hands other than their lenghts and angular positions. They learn that the minutes may be marked or not, and the hours may be labeled with arabic or roman numerals, or not numbered at all but merely marked. Once children learn to distinguish the architecture from the accidents of visual appearance, they can tell time as easily from a wrist watch as from the clock on the church tower.

The architecture of the clock, thus specifies the conceptual structure and functional behavior as perceived by the user. The inner structure is not at all specified by the architecture; one does not need to know what makes the clock tick to know what time it is.

Architecture tells us *what* happens; implementations tells us *how* it is made to happen. The implementation of a clock requires two major designs: how the clock is powered and how its time-keeping precision is achieved. The standard clock architecture has seen hundreds of different implementations, e.g.,

a weight, driving a pendulum,
a spring, driving a balance wheel,
a battery, driving a quartz oscillator, and
a remote electrical generator, driving a synchronous motor.

Any one of these implementations involves many design decisions. What period shall the pendulum have? How many gears, of how many teeth, how connected? How shall the escapement deliver power to the pendulum? How shall the weights be wound? The implementation, then, is the logical organization of the inner structure of a designed object.

Below the implementation lies the level of realization. Given the period of the pendulum and the number of teeth of the gears, *where* are they placed in relation to each other and *which* materials are to be used? With what geometry, strength, tolerance, and finish? Clearly any one implementation may have many different realizations.

The clock illustrates that application of the concept of an architecture, as distinct from implementations and realizations, is very old.

ARCHITECTURE AS APPLIED TO COMPUTERS

The architecture of a computer is a minimal behavioral specification - *behavioral* in the sense that software can be written, *minimal* in the sense that the widest possible range of excellence criteria can be chosen for implementations. Since the term *architecture* can be applied to all kinds of designs we can speak of the architecture of an application language, an operating system, a programming language, a machine language, or a microcode language, as illustrated in Figure 2. In those cases where an architecture is implemented in another architecture, as for example a compiler implemented in an assembly language, we can speak of vertical recursion. Thus one or two levels of microcoding are often found below the machine-language level. Each of these levels has an architecture and an implementation. Only the lowest has a realization. We use the term architecture in this text only to refer to the machine-language level of a computer system.

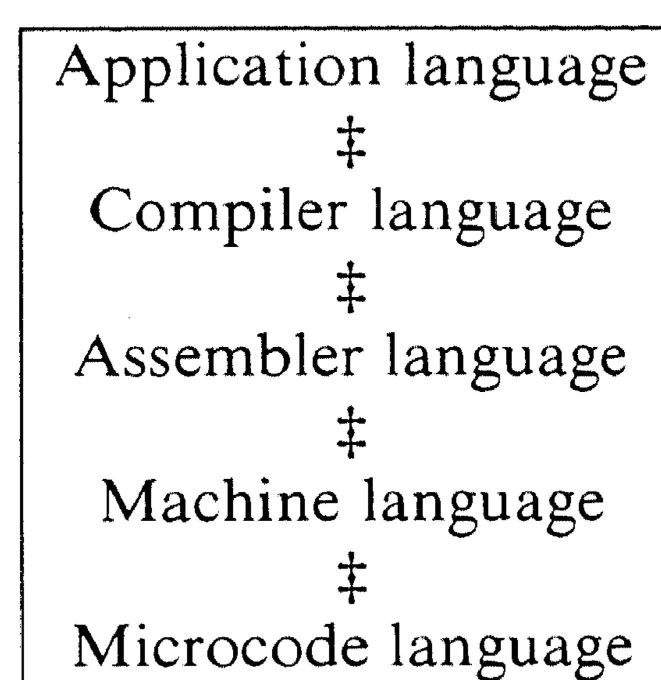

FIGURE 2. Vertical recursion of architecture.

MACHINE LANGUAGE

We define *machine language* to be that representation of programs that resides in memory and is interpreted (executed) directly by the hardware. It usually consists of a string of bits. From this point of view, the design of a computer architecture is the design of its machine language. If, however, we consider the

machine language to be just one of the many programming languages related to a computer, as sketched in the vertical recursion of Figure 2, we must answer the question of how and why machine languages differ from other programming languages.

Although machine languages are often simpler and more constrained than many other programming languages, there is only one essential difference: In machine language the *expressions are costly.* The criterion of costliness is measured in *space* (the number of components of logic and memory that are used) and *time* (the delay caused by the components that are traversed). Each operator and variable in the vocabulary must be implemented and realized by the interpreting mechanism. Each bit in a machine-language program occupies a costly memory cell and must be obtained from that cell at the expense of costly time. So costliness urges *conciseness,* economy of expression. Conciseness is a central consideration in design. It manifests itself in compactness of representation, sparsity and simplicity of constructs, and independence of interpretation.

Whereas the implementer counts logical elements, and the realizer counts components, square microns of silicon, microns or millimeters of connection, and watts, the architect counts bits of representation in his first-instance budget, the *bit budget.* Furthermore, computer technologies from the very beginning have been of such a nature that the critical factor limiting the performance of a computer is the *memory bandwidth,* the number of bits per second the memory subsystem can deliver. When memory bandwidth is limiting, each additional bit in a machine-language program slows its execution. The architect therefore wants to reduce the *bit traffic* between the memory and the processor.

The complexity and number of types of utterances affects cost, because each type requires some interpretation activity unique to it. Specifying a type also costs according to (the logarithm of) the number from which selection is made. So sparsity of operations, addressing modes, and possible addresses all contribute to conciseness. Finally the cost of executing machine-language statements is sharply reduced if each statement can be interpreted independently. Compilation, which requires scanning of the program as a whole, violates this criterion. Interpretation of languages such as APL or Basic also violates it, in that it requires symbol tables to be built and maintained.

Computer generations

To start our historic survey we first look at computer generations as listed in Table 3.

Architecture	Implementation	Realization	Date
Classical computer	Series Parallel	Electro-mechanical Vacuum tube	1945
Interruption Supervision	Pipeline	Transistor Core storage	1955
Memory mapping Peripheral processor	Microcode	Integrated circuit	1965
Vector arithmetic Network	Cache	Very large scale integration	1975

TABLE 3. Computer generations.

This customary division into generations has nothing to do with computer architecture; it is a division based on realization technology alone. Not only is there a dramatic difference in capability and appearance from one realization generation to the next, but once a new realization generation emerges it replaces the preceding generation. Vacuum tubes were completely displaced by discrete transistors and core storage, only to be in turn displaced by integrated logic and memory circuits.

The picture is quite different for architecture. A classical architecture was established in the first generation, and it has endured until the present. New functions, such as those for supervision, memory management, and communication with peripheral devices, have appeared, but they have been orthogonal to the functions of the classical architecture, and have been added to it. Even details of the architecture often survive several generations of implementation and realization. One reason is that useful programs have very long lifetimes. As a program is used, it is usually extended and modified. Because reprogramming old applications is distasteful and expensive, there is a ready market for new computers that extend an old architecture.

Therefore, an architecture appears to be timeless in contrast to its transient realizations. The fundamental implementation techniques are also timeless. The basic algorithms were all developed in the first generation, with macro techniques—such as pipelining, microcoding, and lookaside—exploited on top of these algorithms. Since the implementation must match a stable architecture to a changing realization, different demands are made in each generation upon the wide arsenal of techniques of implementation.

The stability of computer architecture is not merely caused by the inertia of programming investment. One can identify several eras in computer architecture, each marked by an entirely new field of application or way of using the computer. Table 4 shows some of the more important architectural eras, giving typical examples. Thus the minicomputer era, which placed the computer as an instrument in the laboratory, started afresh with new architectures and new

programs, not tied to previous practice. Yet the minicomputer architecture was at the start not that different from the classical architecture and indeed converged to it in time.

Era	Type	Example	Date
0	Pioneer computer	Z4, Mark I, ENIAC	1940
1	Classical computer	MU1, Univac, IBM 704	1950
2	Supervised computer	IBM Stretch	1955
3	Supercomputer	IBM Stretch, CDC 6600	1960
4	Timeshared computer	GE 645	1965
5	Minicomputer	DEC PDP8, PDP11	1970
6	Microprocessor	Intel 8008	1975
7	Workstation	Motorola MC68000	1980

TABLE 4. Architectural eras.

THE CLASSICAL COMPUTER

What then is this classical computer and what causes its stability? Table 5 shows its emergence in the first generation and the contributions of the pioneers of that period. It may be surprising that Von Neumann does not appear in this table; his contribution is in a different direction. His well-known paper with Burks and Goldstine (1946) constitutes the beginning of computer science.

Concept	Innovator	Date
Arithmetic	Schickard	1624
Sequence control	Jacquard et al.	1804
Decision	Babbage	1867
Binary radix	Atanasof et al.	1940
Floating-point	Zuse	1941
Stored program	Mauchly	1944
Multiple processors	Aiken	1947
Logic	Kilburn	1949
Indexing	Kilburn	1949
Byte addressing	Buchholz	1954
Directly-addressed registers	Buchholz	1956

TABLE 5. Classical computer architecture.

In his thorough study of Babbage's Analytical Engine (1867), Bromley (1982) remarks that this computer is *too much* like a modern computer (his italics). A similar remark could be made about most pioneering computer designs. Indeed it is remarkable how quickly the progression of Babbage's Difference and Analytical Engine, Aiken's Mark I, Zuse's Z4, Kilburn's Mark 1, and Eckert and Mauchly's Univac converges toward the classical computer.

The floating point of Konrad Zuse's Z3 and Z4 was way ahead of its time. A complete set of extrema: *infinity, negative infinity, zero or negligible, and indefinite,* were part of the representation and were consistently used in arithmetic. This design predates the incomplete set of extrema of the IBM Stretch (1961) by more then 15 years and the complete set of extrema of the CDC 6600 (1964) by about 20 years; these extrema are now part of the IEEE Floating-point Standard (1981). Normalized representation with binary radix makes the leftmost coefficient bit redundant. Zuse eliminated this bit, as did Gordon Bell in the PDP11 (1970). This *hidden bit* is also part of the IEEE standard.

The Z4 instruction set includes Negate, Add, Substract, Multiply, Divide, Square, Square Root, Times Two, Times Ten, Times One-half, Times One-tenth; more than a classical machine demands, but quite suitable for a machine without subroutine facilities. Numbers are converted from decimal to floating point on input and back to decimal on output. The Z4 working store is a stack with a maximum depth of two. This function fits well in a design with a relatively fast memory; it predates the stack of the English Electric KDF9 (1963) and of the Burroughs B5000 (1964) by more then 15 years.

Why did the classical computer emerge so rapidly and persist so tenaciously? We believe the answer is found in the costliness constraint mentioned above. Costliness, and its various ramifications, explain why the classical direct addressing was not abandoned for associative addressing, why the classical machine language level was not abandoned upward towards a high-level language nor downward by adopting microcoding as the machine language. It also explains why elaborate and unbalanced instruction sets quickly converge to the almost standard instruction set of the classical computer. Even for parallelism, currently much in discussion, costliness supplies the most promising answer, that of networks of classical computers.

The classical computer survived certainly not for lack of ideas. Even if we consider only general-purpose mass-produced commercial products, we find enough contenders. Table 6 lists a few of them. We briefly look at each in turn.

Abundant function	1955
Associative main memory	1960
Microcode as machine language	1970
High-level machine language	1975
Concurrency	1980

TABLE 6. Attacks on the classical computer.

ABUNDANT FUNCTION

A repeatedly recurring idea is to give the user function directly in the machine language through many operations, options, and formats.

Strangely enough some of the oldest computers were more richly endowed with arithmetic operations than the modern computer. As stated, the Z4 had

square root as a machine-language operation; so had the ENIAC (1946). The Harvard Mark I (1944) even had operations for sine, exponentiation, logarithm, and interpolation. By 1950 these complex operations had been removed from the Mark I. The reasons were in part peculiar to Mark I circumstances. Some operations were infrequently used and hence, when used, proved unreliable due to dust on the electromechanical contacts. More seriously, the hardware operations were provided in full 24-decimal-digit precision, which was rarely needed, so programmed subroutines could usually outrun them. To a surprising extent, however, the reasons for removing these advanced operations are universally valid. Their specialized hardware was rebuilt into generally useful registers, thus increasing the size of memory. This is an example of the principle of costliness working on too rich an operation set.

The Mark I example reminds us that all such choices involve a quid pro quo. Although a bodily conversion of components as in the Mark I is not likely to occur again, the designer in fact trades one function for another. The hardware which an operation requires and the software it entails can with equal total cost also be applied to improvements of the remaining functions, perhaps to better over-all advantage. An operation is never free.

Once it was understood that a compiler can provide as rich a set of functions as desired by means of subroutines, it became clear that making a limited instruction set more effective is more profitable than extending it. Nevertheless, at times the rich instruction (and option) set re-emerges, most noticably in the IBM Stretch (Buchholz, 1962) and to a minor degree in the DEC VAX11 (1977).

Associative main memory

In programming languages, names frequently refer to groups of data and instructions, such as files, tables, arrays, and procedures. In the machine language, however, the address is normally linear; addresses are integers and can be computed by regular arithmetic.

The basic and major impropriety of addressing is to map group names upon the linear address space. All the programmer wants to specify is a set of named objects. The allocation of space for these objects in a memory is not his concern and in some languages, such as APL, he is not even confronted with this issue. Moreover, since the size of named objects may change in time, the allocation problem is very complex. The driving problem of addressing thus turns out to be the mapping of names upon a linear address space - the *binding* of names to addresses.

The most extreme solution to the binding problem is a store in which each location is independently bound to its name. In such a store each physical location is built to contain not only a datum, but a name as well. When a particular name is specified for access, the name fields of all storage locations are searched, and the datum whose stored name matches the search name is acccessed. Because the data are associated with more or less arbitrary names, this is commonly called *associative addressing*.

Associative addressing attacks the memory allocation problem by concealing

all physical adjacency and contiguity. The store itself consists of named objects bearing no relationship to one another. Since the associative store removes the need for programmed allocation it appears as a perfect solution, except that each memory location contains an address next to the stored information. Hence in comparison to the regular directly addressed memory the bit budget is seriously impaired. As a consequence its implementation is either not simple or not fast.

The fastest associative addressing implementations provide a 1-bit comparator for each of the M bits of the name field of each word. In comparison to a directly-addressed memory, each word has M extra bits of memory. If a comparator takes twice the circuitry of a memory bit, the additional cost per word is 3 x M bits. If, for example, the datum length is also M, such a memory takes four times as much circuitry as a directly-addressed memory of comparable speed and capacity. Put more vividly, one could have a directly-addressed memory of four times the capacity for the same cost.

Through the years proponents of associative main memory have done cost estimates in current technology and been appalled. Yet they have asserted that an associative memory would become attractive when the next technology appeared and made components cheap. This fallacy confuses the properties of the implementations with those of their realizations.

In *any* technology suitable for realizing the simultaneous comparison implementation, and for any number of circuits one can assemble and afford, one could alternatively use the circuits for associative addressing or for about four times the capacity in directly-addressed memory. Regardless of technology the architect must choose between function or capacity (Brooks, 1965).

The answer to the binding problem has been the paging and segmentation functions that can be added to a classical architecture. Kilburn had the first glimpse of this function in his Mark 1 (1949); it was more fully present in his Atlas design (1962) and has since been used extensively in many designs.

MICROCODE AS MACHINE LANGUAGE

The two basic parts in processor implementation are data path and control. The datapath is made up of the components holding and transforming data. The control specifies the use of these components in time. This specification is done by *gate signals* that originate in the control and activate the gates of the datapath; conversely the outcome of the datapath action, such as a positive or negative sign, may be used as a *test signal* to affect the sequence generated by the control. Early designs aimed to minimize the number of components. Hence a great variety of encodings in which little structure could be recognized was used for the control.

In 1951 Wilkes pointed out that the task of the control is similar to the task of a computer: Specific actions are to be performed in a proper sequence. The actions are simply the opening and closing of gates; the sequence includes frequent, but simple, decision-making based upon the test signals. The gate signals and decision specifications are combined as a *microinstruction;* the microinstructions necessary for the proper computer action constitute a

microprogram; the *microprogram* is placed in a storage, the *microstore*. This method of control is called *microcoding* (Wilkes, 1951). Why not use this microcode as machine language? It might give an adaptable computer geared to a given application or compiler language. Thus the Burroughs B1700 (Wilner, 1972) was considered a microcoded machine that interpreted a variety of S-languages, each atuned to a compiler language.

The machine language is the interface offered to the user and maintained with stability by the manufacturer. Since microcode is by definition implementation-derived, language stability at this level implies much more serious constraints on implementations than apply today. This means original costs will be higher and that cost-savings due to implementation changes must be foregone. If on the other hand the architecture is not kept stable and the machine language changes with the implementation needs, the user must reprogram his applications as time goes on.

By far the most serious drawback is the effect on the size of the user communities. A major conceptual advance made by the IBM System/360 - indeed, the major advance - was the establishing of a single machine language as an architecture against which a variety of implementations of different performances were built. This meant that all of these implementations constituted a single base for software support. Software costs are a very substantial part of both manufacturer and user costs for computers. Consider how much worse this would be if the base for sharing were fragmented.

A microcode allows a user to design his own machine. But, this puts him a position of splendid isolation, a position which moreover is very short-lived. Worse, the freedom to build your own machine is sharply limited. A microcode is highly specialized towards its target: the machine language for which it was intended in the first place. Many functions we may want to perform by microcoding may prove to be awkward or inefficient.

The architecture of the B1700 was wisely kept stable through six implementation changes; so programming investment was protected. The S-languages turned out to be a compiler designer's option, only marginally influenced by the underlying machine language. Most important, on closer inspection the B1700 machine language proved to be very much that of a classical machine, with its bit addressing reminiscent of the IBM Stretch.

High-level machine language

Why have assembly-level machine language? Is it not the outworn relic of outmoded thinking? Since most applications are programmed in Fortran, Cobol, Pascal, C, etc., why not implement these directly in microcode?

Even if all applications were written in high-level languages, there would still be strong reasons for defining a computer architecture at a lower level. The most compelling reason arises from the properties of the high-level languages themselves. For most such, translation to execution ideally involves two steps, one at compile time and one at execute time. An alphanumeric mnemonic name, for example, can obtain a compactly represented name at compile time. This in turn can be related to a memory address (itself a compact name) at

load time. Such early, one-time compilation vastly reduces the work the interpreter must do iteratively. Ideally, a machine language should stand precisely between compilation and interpretation such that the architectural cost criterium of independent interpretation of the machine language is maintained.

The semantics of the high-level language and the machine language are basically different. The first assumes global knowledge of the program, the second assume only local knowledge. This difference has been called the *semantic gap* (Gogliardi, 1973).

Many have proposed reducing, if not eliminating the "semantic gap" (Myers, 1978). The foregoing considerations, however, show that it is inherent —such a gap *must* occur in the language hierarchy. We believe it not only proper, but indeed natural and perhaps optimal for the semantic gap to serve as the upper bound of the machine language level (Blaauw, 1980). The experience with machines that drastically raised the machine-language level and at the same time tried to preserve generality with respect to user languages seems to confirm this (e.g., the Burroughs 5500 (1964) and the Intel 432 (1980)).

Interpretation of a high-level language is split into two parts by the machine language. The compiler takes care of one part, for instance from Fortran to machine language; the implementation takes care of the second part, for instance via microcode. The implementation can optimize better when it gets a larger part of the interpretation process because it can know run-time information. The compiler, on the other hand, can know global information about the entire program. It can exploit its knowledge of the algorithm and the operands in the optimization (for example, substituting a shift for multiplication by a constant power of two). Thus a low-level architecture is more attractive as the target of compilation than a high-level architecture.

An architecture in which most, if not all, operations can be implemented in a single datapath action and which has few constructs is called a *reduced instruction set computer* (RISC). Early examples are Van der Poel's Zebra (1959), and first-generation microprocessors such as the Intel 8080 (1974) and the Motorola 6800 (1975).

The absence of operations requiring subcycles, such as multiply, divide, and multiple-bit shifts creates a programming inconvenience; a RISC architecture explicitly assumes a compiler or macroassembler that can eliminate it. The instruction set is furthermore designed to be a suitable target for an optimizing compiler. At the same time the operation set allows a fast implementation with the majority of the circuits serving the data flow rather than the control thereoff (Radin, 1982).

Concurrency

Concurrency can occur in the design domains of architecture and implementation.

Architectural concurrency is visible to the user and should match the inherent independencies of parts of his computing task. The most familiar example is the concurrency of a central processing unit (CPU) for general processing and a peripheral processing unit (PPU) for the input/output actions.

Implementation concurrency is invisible to the user; he cannot see whether the implementation is concurrent or not. Implementation concurrency arises from independencies that follow from the definition of the computer architecture. A 32-bit fixed-point Load instruction, for example, allows the implementation complete freedom to fetch 1, 2, 4, ..., 128, or some other number of bits at a time; it also permits the implementation to process this instruction concurrent with preceding and following instructions in a pipelined fashion.

There need not be a one-to-one correspondence between architectural concurrency and implementation concurrency. A multiprocessor architecture may quite feasibly be implemented by a single processor. Thus the ten PPUs of the CDC 6600 (1964) are really virtual machines whose arithmetic functions are implemented by one common processor. Conversely, a single processor architecture may be implemented by several concurrent processors. The CDC 6600 CPU, for instance, incorporates one logical unit, one integer add unit, one floating-point add unit, two multiply units, one divide unit, two increment units, one shift unit, and a branch unit. All of these can operate concurrently, even though the machine language invokes them sequentially (Thornton, 1970).

Inherent concurrency in an application always allows but never demands full architectural concurrency in the system designed to perform that application. Processors are digital and operate with discrete actions; they sample and change the processes they control at discrete intervals. Because of this discretization, a single time-sliced system has the same effect as a set of concurrent processors so long as it maintains the necessary rates observed external to the system. Because concurrent subprocesses are easily simulated on a single central processor, the decision to provide multiple processors in the architecture is dominated by the implementation questions: How fast? How efficient? In our historic survey we just mention a key problem, a common fallacy, and a preliminary answer.

Many attempts at concurrency have failed by paying insufficient attention to the bit traffic. Main memory is the most critical resource to be shared. The speed with which the total bit traffic of an application can be handled is to a major degree determined by the time required for memory access. Adding more processors does not help once memory access is saturated. This design error is sometimes erroneously called the *Von Neumann bottleneck.*

It would be nice if memory access were so fast that it could naturally satisfy a multitude of processors. The realization technology of memory, however, is similar to that of processors. For such a technology the implementation can nicely match the access offered by a memory system with the memory access required by one or two processors and their peripheral processors. But sharing that access with more processors only lowers the effective performance of the processors. A faster technology does no good: The processor-memory balance is determined by the implementation, not the realization. Bit traffic also affects the choice between *fine granularity* (where each processor is capable only of one or a few actions) and *coarse granularity* (where all processors have a classical architecture).

The concurrency of processing and input/output operation went through

four stages: a. *direct input/output,* with the general (arithmetic and logical) processor waiting while an input/output operation proceeds, as in Kilburns MU1 (1949); b. *overlapped input/output,* where single input/output operations are concurrent with each other and with general processing, as in the IBM 701 (1953); c. *channel,* or direct memory access (DMA), where streams of input/output instructions are concurrent with each other and with central processing, as in the IBM 709 (1959); d. *peripheral processors,* which have full input/output and general power and can operate concurrently with each other and with the central general processors, as in the CDC 6600 (1964). This development from a single action, via a specialized processor, to a classical computer illustrates the move from fine granularity to coarse granularity to reduce bit traffic.

A second example is the Bull Gamma 60 (1959). This computer comprised an arithmetic processor, a logic processor, a compare processor, and several input/output processors, all capable of functioning concurrently, each with its own instruction stream. The processors had fine granularity: The arithmetic processor could not perform logic, nor could it compare (Dreyfuss, 1958). Hence, the instruction sequences that each processor handled did not match the instruction sequences that are natural in a computation. Comparisons normally are intermixed with arithmetic and logic; they are not separate activities. As a consequence the programs constantly switched from processor to processor, at the expense of extra bit traffic.

More recent examples of fine-grained processors are the data-flow computer proposed by Dennis (1974) and Hillis' Connection Machine (1985).

Human communication can result in *synergism:* Jointly people can achieve more than the sum of their separate efforts. A gifted song writer and a gifted composer may jointly produce musicals of a level that they could not reach when working at it sequentially. To expect synergism from communicating processors, however, is an anthropomorphism: The joint performance of communicating processors is never more than the sum of their individual performances—rather, it is always less because of the hardware and software costs of communication.

Multiple processors concurrently executing a single task may, however, significantly reduce the elapsed time required for that task. This time reduction may well be worth the loss of efficiency caused by extra programming and hardware costs.

Communicating processors with coarse granularity constitute a network of classical machines. Although achieving a rapid response with such a network is still a major effort, fortunately no special processor hardware need be designed (for instance, a supercomputer may be used).

Conclusion

Our brief historic survey shows that a classical computer architecture was established during the first computer generation. It was enhanced with orthogonal functions, as for supervision and memory management. It was used as main processor, peripheral processor, super-, mini- and microprocessor. It can

provide fast concurrent operation in a network. We claim that the classical computer architecture has survived many attacks upon its design because in computer architecture expressions are costly.

REFERENCES

1. BLAAUW, G.A., 1980: Hogere Programmeertalen Gezien vanuit de Machinetaal. *Colloquium Hogere Programmeertalen en Computerarchitektuur,* MC Syllabus 45, 149-162.
2. BROOKS, F.P., JR., 1965: The Future of Computer Architecture. *Proceedings of the IFIP Congress '65,* 87-91.
3. BUCHHOLZ, W., ed., 1962: *Planning a Computer System,* McGraw Hill, New York, New York.
4. BURKS, A.W., H.H. GOLDSTINE, and J. VON NEUMANN, 1946: Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. *Report to U.S. Army Ordnance Department,* reprinted in Bell and Newell, 1971, 92-119.
5. BROMLEY, A.G., 1982: Charles Babbage's Analytical Engine. *Annals of the History of Computing,* vol. 4, no. 3, 196-217.
6. DENNIS, J.B. and R.P. MISUNAS, 1974: A Preliminary Architecture for a Basic Data Flow Processor. *Second International Symposium on Computer Architecture,* Computer Architecture News, vol. 3 no. 4. 126-132.
7. DREYFUS, P. 1958: System Design of the Gamma 60. *Proceedings of the Western Joint Computer Conference,* 130-133.
8. GOGLIARDI, U. O., 1973: Report on Software Related Advances in Computer Hardware. *Proceedings of a Symposium on the High Cost of Software,* Stanford Research Institute, Menlo Park, California.
9. HILLIS, W.D. 1985: *The Connection Machine,* The MIT Press, Cambridge, Massachusetts.
10. MYERS, G.J., 1978: *Advances in Computer Architecture,* Wiley, New York, New York.
11. RADIN, G., 1983: The 801 Minicomputer. *IBM Journal of Research and Developmment,* vol. 27, no. 3, 837-847.
12. THORNTON, J.E., 1970: *The Control Data 6600,* Scott, Foresman and Company, Glenview, Illinois.
13. WILKES, M.V., 1951: The Best Way to Design an Automatic Calculating Machine. *Manchester University Computer Inaugural Conference,* Manchester, 16-21.
14. WILNER, W.T., 1972: Design of the B1700. *AFIPS Conference Proceedings,* vol. 41, part 1, 489-497.