23. H. A. Ernst, "TCS, an experimental multiprogramming system for the IBM 7090," IBM Corp., Yorktown Heights, N. Y., Research Rept. RJ248, June 1963, 41 pages.
the IBM 7090," IBM Research Rept. RJ248, June 1963, 41 pages.
24. M. Lehman, R. Eshed, and Z. Netter, "SABRAC, a time sharing low-cost computer," *Commun. ACM*, vol. 6, pp. 427–429, August 1963.
25. R. V. Smith and D. N. Senzig, "Computer organization for array processing," IBM Corp., Yorktown Heights, N. Y., Research Rept. RC 1330, December 1964.
26. A. S. Critchlow, "Generalized multiprocessing and multiprogramming systems," *1963 AFIPS Proc. FJCC*, pp. 107–126.
27. M. E. Conway, "A multiprocessor system design," *ibid.*, pp. 139–146.
28. R. R. Seeber and A. B. Lindquist, "Associative logic for highly parallel systems," *ibid.*, pp. 489–493.
29. R. M. Meade, "604 machine description," IBM internal memo., December 1963, 38 pages.
30. M. Lehman, R. Eshed, and Z. Netter, "SABRAC—a new generation serial computer," *IEEE Trans. on Electronic Computers*, vol. EC-12, pp. 618–628, December 1963.
31. M. W. Allen, T. Pearcey, J. P. Penny, G. A. Rose, and J. G. Sanderson, "CIRRUS, an economical multiprogram computer with microprogram control," *ibid.*, pp. 663–671.
32. W. F. Miller and R. A. Aschenbrenner, "The GUS multicomputer system," *ibid.*, pp. 671–676.
33. G. Estrin, B. Russenl, R. Turn, and J. Bibb, "Parallel processing in a restructurable computer system," *ibid.*, pp. 747–755.
34. G. Gregory and R. McReynolds, "The Solomon computer," *ibid.*, pp. 774–755.
35. H. S. Bright, "A Philco multi-processing system," *1964 Proc. FJCC*, pp. 97–141.
36. R. G. Ewing and P. M. Davies, "An associative processor," *1964 AFIPS Proc. FJCC*, pp. 147–158.
37. H. A. Kinslow, "The time-sharing monitor system," *ibid.*, pp. 443–454.
38. J. Nievergelt, "Parallel methods for integrating ordinary differential equations," *Commun. ACM*, vol. 7, pp. 731–733, December 1964.
39. W. H. Desmonde, *Real Time Data Processing Systems*. Englewood Cliffs, N. J.: Prentice-Hall, 1964.
40. M. Lehman, "Serial mode operation and high-speed parallel processing," *Information Processing, 1965 Proc. IFIP*, pt. 2. New York: Spartan, 1966, pp. 631–633.
41. R. V. Smith and D. N. Senzig, "Computer organization for array processing," *1965 Proc. FJCC*, pp. 117–128.
42. E. W. Dijkstra, "Solution of a problem in concurrent programming control," *Commun. ACM*, vol. 8, p. 569, September 1965.
43. J. B. Dennis, "Segmentation and the design of multiprogrammed computer systems," *J. ACM*, vol 12, pp. 589–602, October 1965.
44. F. J. Corbato and V. A. Vyssotsky, "Introduction and overview of the multics system," *1965 Proc. FJCC*, pp. 185–196.
45. E. L. Glaser, J. Couleur, and G. Oliver, "System design of a computer for time sharing applications," *ibid.*, pp. 197–202.
46. V. A. Vyssotsky, F. J. Corbato, and R. M. Graham, "Structure of the multics supervisor," *ibid.*, pp. 203–212.
47. R. C. Daley and P. G. Neumann, "A general-purpose file system for secondary storage," *ibid.*, pp. 213–229.
48. J. F. Ossanna, L. E. Mikus, and S. D. Dursten, "Communication and input-output switching in a multiple computing system," *ibid.*, pp. 231–241.
49. J. W. Forgie, "A time and memory sharing executive program for quick-response on-line applications," *ibid.*, pp. 599–610.
50. J. D. McCullogh, K. H. Speierman, and F. W. Zurcher, "Design for a multiple user multiprocessing system," *ibid.*, pp. 611–618.
51. W. T. Comfort, "A computing system design for user device," *ibid.*, pp. 619–628.
52. J. P. Anderson, "Program structures for parallel processing," *Commun. ACM*, vol. 8, pp. 786–788, December 1965.
53. B. W. Arden, B. A. Galler, T. C. D. O'Brien, and F. H. Westervelt, "Program and addressing structure in a time-sharing environment," *J. ACM*, vol. 13, pp. 1–16, January 1966.
54. J. H. Katz, "Simulation of a multiprocessor computer system," SR & D Rept. LA-009, February 1966, to be published in *1966 Proc. SJCC*.
55. G. S. Shedler and M. M. Lehman, "Parallel computation and the solution of polynomial equations," IBM, Yorktown Heights, N. Y., Research Rept. RC 1550, February 1966.
56. H. Hellerman, "Parallel processing of algebraic expressions," *IEEE Trans. on Electronic Computers*, vol. EC-15, pp. 82–91, February 1966.
57. J. B. Dennis and E. C. Van Horn, "Programming semantics for multiprogrammed computation," *Commun. ACM*, vol. 9, pp. 143–155, March 1966.
58. N. Wirth, "A note on 'program structures' for parallel programming," *Commun. ACM*, vol. 9, pp. 320–321, May 1966.
59. D. E. Knuth, "Additional comments on problems in concurrent programming control," *ibid.*, pp. 321–322.

Very High-Speed Computing Systems

MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—**Very high-speed computers may be classified as follows:**

1) Single Instruction Stream–Single Data Stream (SISD)
2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream–Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).

"Stream," as used here, refers to the sequence of data or instructions as seen by the machine during the execution of a program.

The constituents of a system: storage, execution, and instruction handling (branching) are discussed with regard to recent developments and/or systems limitations. The constituents are discussed in terms of concurrent SISD systems (CDC 6600 series and, in particular, IBM Model 90 series), since multiple stream organizations usually do not require any more elaborate components.

Representative organizations are selected from each class and the arrangement of the constituents is shown.

Manuscript received June 30, 1966; revised August 16, 1966. This work was performed under the auspices of the U. S. Atomic Energy Commission.

The author is with Northwestern University, Evanston, Ill., and Argonne National Laboratory, Argonne, Ill.

INTRODUCTION

MANY SIGNIFICANT scientific problems require the use of prodigious amounts of computing time. In order to handle these problems adequately, the large-scale scientific computer has been developed. This computer addresses itself to a class of problems characterized by having a high ratio of computing requirement to input/output requirements (a partially de facto situation

caused by the unavailability of matching input/output equipment). The complexity of these processors, coupled with the advancement of the state of the computing art they represent, has focused attention on scientific computers. Insight thus gained is frequently a predictor of computer developments on a more universal basis. This paper is an attempt to explore large scientific computing equipment, reviewing possible organizations starting with the "concurrent" organizations which are presently in operation and then examining the other theoretical organizational possibilities.

Organization

The computing process, in its essential form, is the performance of a sequence of instructions on a set of data.

Each instruction performs a combinatorial manipulation (although, for economy, subsequencing is also involved) on one or two elements of the data set. If the element were a single bit and only one such bit could be manipulated at any unit of time, we would have a variation of the Turing machine—the strictly serial sequential machine.

The natural extension of this is to introduce a data set whose elements more closely correspond to a "natural" data quantum (character, integer, floating point number, etc.). Since the size of datum has increased, so too has the number of combinatorial manipulations that can be performed (manipulations on two n bit arguments have 2^{2n} possible outcomes). Of course, attention is restricted to those operations which have arithmetic or logical significance.

A program consists of an ordered set of instructions. The program has considerably fewer written (or stored) instructions than the number of machine instructions to be performed. The difference is in the recursions or "loops" which are inherent in the program. It is highly advantageous if the algorithm being implemented is highly recursive. The basic mechanism for setting up the loops is the conditional branch instructions.

For convenience we adopt two working definitions: *Instruction Stream* is the sequence of instructions as performed by the machine; *Data Stream* is the sequence of data called for by the instruction stream (including input and partial or temporary results). These two concepts are quite useful in categorizing computer organizations in an attempt to avoid the ubiquitous and ambiguous term "parallelism." Organizations will be characterized by the multiplicity of the hardware provided to service the Instruction and Data Streams. The mutiplicity is taken as the maximum possible number of *simultaneous* operations (instructions) or operands (data) being in the same phase of execution *at the most constrained* component of the organization.

Several questions are immediately evident: what is an instruction; what is an operand; how is the "constraining component" found? These problems can be answered better by establishment of a reference. If the IBM 704 were compared to the Turing machine, the 704 would appear highly parallel. On the other hand, if a definition were made in terms of the "natural" data unit called for by a problem, the situation would be equally untenable, since in many problems one would consider a large matrix of data a unit. Thus we arbitrarily select a reference organization: the IBM 704-709-7090. This organization is then regarded as the prototype of the class of machines which we label:

1) Single Instruction Stream–Single Data Stream (SISD).

Three additional organizational classes are evident.

2) Single Instruction Stream–Multiple Data Stream (SIMD)
3) Multiple Instruction Stream-Single Data Stream (MISD)
4) Multiple Instruction Stream–Multiple Data Stream (MIMD).

Before continuing, we define two additional useful notions.

Bandwidth is an expression of time-rate of occurrence. In particular, computational or execution bandwidth is the number of instructions processed per second and storage bandwidth is the retrieval rate of operand and operation memory words (words/second).

Latency or latent period is the total time associated with the processing (from excitation to response) of a particular data unit at a phase in the computing process.

Thus far, categorization has depended on the multiplicity of simultaneous events at the system's component which imposes the most constraints. The ratio of the number of simultaneous instructions being processed to this constrained multiplicity is called the *confluence* (or concurrence) of the system.

Confluence is illustrated in Fig. 1 for an SISD organization. Its effect is to increase the computational bandwidth (instructions processed/second) by maximizing the utility

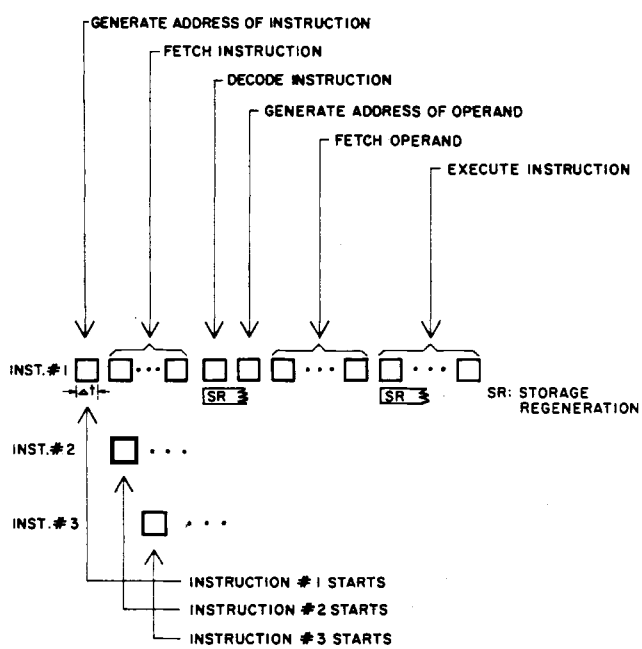


Fig. 1. Concurrency and instruction processing.

of the constraining component (or "bottleneck"). The processing of the first instruction proceeds in the phases shown. In order to increase the computational speed, we begin processing instruction 2 as soon as instruction 1 completes its first phase. Clearly, it is desirable to minimize the time in each phase, but there is no advantage in minimizing below the time required by a particular phase or mechanism. Suppose, for a given organization, that the instruction decoder is an absolute serial mechanism with resolution time, Δt. If the average instruction preparation and execution time is t_c, then the computational bandwidth may be improved by $t_c/\Delta t$. Thus if only one instruction can be handled at a time at the bottleneck, then the maximum achievable performance is $1/\Delta t$. (1 being the multiplicity of the constraint.)

In order to process a given number of instructions in a particular unit of time, a certain bandwidth of memory must be achieved to insure an ample supply of operands and operations in the form of instructions. Similarly, the data must be operated on (executed) at a rate consistent with the desired computational rate.

Constituents of the System

We will treat storage, execution, and instruction handling (branching) as the major constituents of a system. Since multiple stream organizations usually represent multiple attachments of one or more of the above constituents, we lose no generality in discussing the constituents of a system as they have evolved in SISD organizations. Indeed, confluent SISD organizations—by their nature—must allow arbitrary interaction between elements of the data stream or instruction stream. Multiple stream organization may limit this interaction, thus simplifying some of the considerations in an area. Therefore, for now, we shall be mainly concerned with techniques to extend computational performance in a context of a confluent SISD system. We will further assume that the serial mechanism which constrains the organization is the decoding of instructions. Thus, on the average, the processing of one instruction per decode cycle will be an upper limit on performance.

The relationship between the constituents is shown in Fig. 2 for the SISD organization.

A. Storage

The instruction and data streams are assumed sequential. Thus, accessing to storage will also be sequential. If there were available storage whose access mechanism could be operated in one decode cycle and this accessing could be repeated every cycle, the system would be relatively simple. The only interference would develop when operands had to be fetched in a pattern conflicting with the instructions. Unfortunately, accessing in most storage media is considerably slower than decoding. This makes the use of interleaving techniques necessary to achieve the required memory bandwidth (see Fig. 3). In the interleaving scheme, n memories are used. Words are distributed in each of the memories sequentially (modulo n). In memory i is stored word i, $n+i$, and $2n+i$, etc. However, the accessing mechanism does not alter the latency situation for the system, and this must be included in the design. Memory bandwidth must be sufficient to handle the accessing of instructions and operands, the storage of results and input–output traffic [9]. The amount of input–output bandwidth is problem-dependent. Large memory requirements will necessitate transfer of blocks of data to and from memory. The resulting traffic will be inversely proportional to the overall size of the memory. However, an increase in the size of the memory to minimize this interference also acts to increase the bulk of the memory, and usually the interconnection distance. The result, then, may be an increase in the latency caused by these communications problems.

Figure 3 was generated after the work of Flores [9] and assumes completely random address requests. The "waiting time" is the average additional amount of time (over the access time) required to retrieve an item due to conflicting

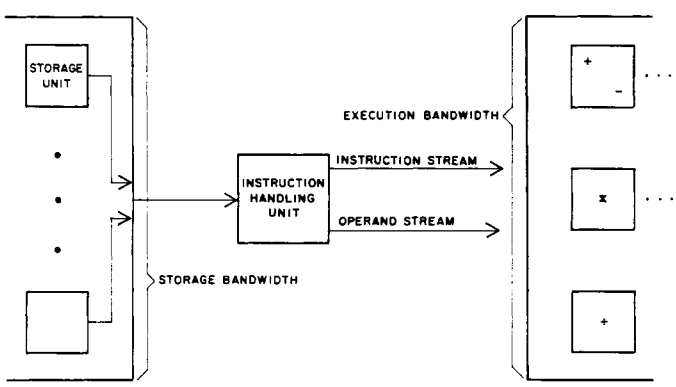
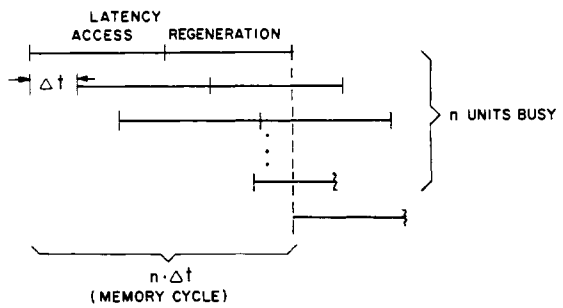


Fig. 2. SISD Organization.



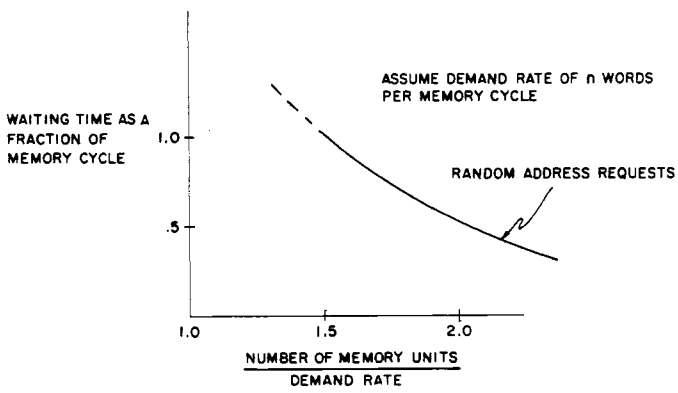


Fig. 3. Interleaving memory units.

requests. Since memory requests are usually sequential for instructions and at least regular for data, the result is overly pessimistic for all but heavily branch-dependent problems. Typical programs will experience about half the waiting time shown.

This latency in the system greatly increases the complexity of the control mechanism for the storage system. The storage unit must now include queuing mechanisms to organize, fetch, and store requests which are in process or could not be honored due to conflicts. These queuing registers, containing outstanding service requests, must be continually compared against new requests for service. Extensive control interlocks must be made available to serve at least the following functions [2], [3]:

1) direct the fetched word to the appropriate requestor;
2) prevent out-of-sequence fetch-store or store-fetch in the same memory location;
3) eliminate duplicate requests of the same memory location (especially where the memory location contains more than one instruction or data unit);
4) analyze "keys" or "boundaries" where fetch or store memory protection are used.

In addition, confluent computing systems frequently employ buffers to minimize traffic requirements on the memory [2]. An instruction buffer might contain the next n instructions in the sequence as well as a history of m instructions together with several levels of alternate paths of branch.

The presence of a historical picture of instructions (in the instruction buffer) allows for the opportunity to store small loops, thus avoiding the penalty of reaccessing instructions. An operand buffer is used by the execution unit as its intrinsic storage medium. The instruction unit would restructure instructions that called for operands from memory into instructions which call for the contents of these registers. The instruction unit would, after decoding the instruction, initiate the requests for the appropriate operands and direct their placement into the operand buffer. Thus the execution unit would act as an independent computer, whose storage would be limited to the contents of this buffer, and whose instructions would be in a shortened format.

B. Execution

We center our discussion on the floating-point instructions since they are the most widely used in large scientific computing, and require the most sophisticated execution. In order to achieve the appropriate bandwidth levels, we can repeat the deployment scheme that was used for memory, only here the independent units will be dedicated to servicing one class of instructions. In addition to directly increasing the bandwidth by providing a number of units, the specialization of the unit aids in execution efficiency in several other ways. In particular, each unit might act as a small insular unit of logic, hence minimizing intra-unit wire communication problems. Secondly, the dedicated unit has fewer logical decisions to make than a universal unit. Thirdly, the unit may be implemented in a more efficient fashion.

One may also consider "pipelining" techniques [6] to improve the bandwidth within a particular insular unit, in addition to optimizing an algorithm to minimize latent time. "Pipelining" is a process wherein natural points in the decision-making hardware are sought to latch up intermediate results and resuscitate the use of the unit. The latching may include extra decision elements for storage, but these may also be used in aiding the control of time skew (differences) in the various parallel paths.

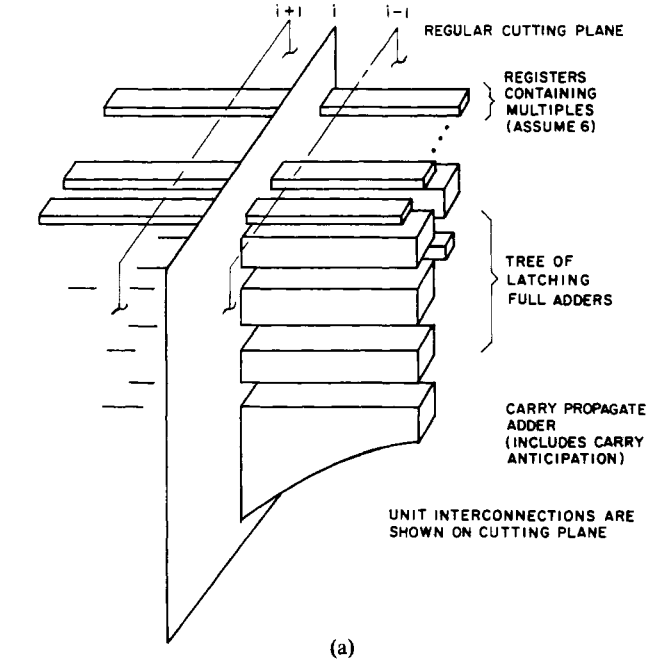
1) Floating Add–Subtract Operations: Floating add–subtract operations consist of three basic parts. First, the justification of the fractional parts of the two operands by the amount of the difference of the exponents. Second, the adding of the fractional parts (or appropriate complement). Third, the postnormalization of the fraction if the result has a leading zero in a significant position or an overflow occurs. Because of the decision-making (shifting) problems associated with the exponent handling, a normal latch point for pipelining the two add operations in one unit (duplexing the unit) is at the interface from the preshift into the adder or after the first level of the adder structure. The floating add class of instruction has been reported [5] to operate in the 120 nanosecond range for a duplexed unit with 56-bit fraction (Systems 360 format).

2) Floating Multiply [5], [10], [12], [13]: The essential decision process of the multiplication algorithm is the addition of the multiplicand to itself by as many times as are indicated by the multiplier. If there are n bits in a multiplier and multiplicand, then the multiplicand must be added to itself n times with a one-place shift before each addition. Standard techniques exist for reducing the number of additions (e.g., multiplier bits: 1111. may be treated as $+10000.$ $-1.$) required by encoding the multiplier into a lesser number of signed bits, say $n/2$. Once this is done, Wallace [12] suggests the direct addition of the $n/2$, n-bit, shifted multiplicands.

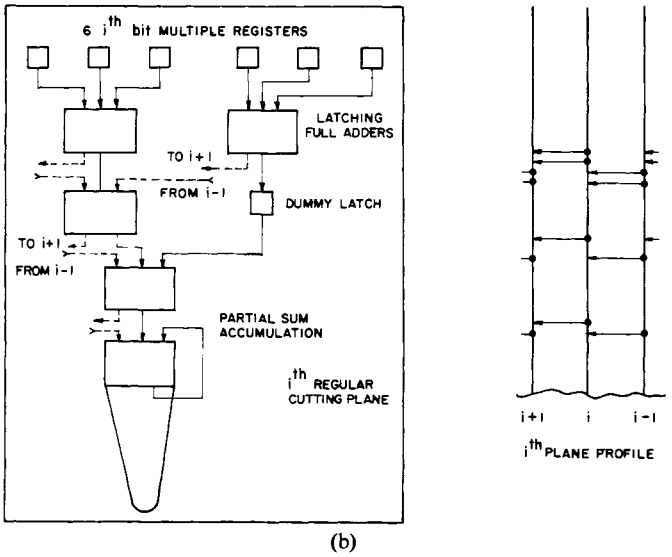
The basic decision element in adding of each bit of the $n/2$ operands is the so-called binary full adder, which takes three inputs of equal significance and produces two outputs, one of the same significance, the sum, and one of higher significance, the carry. By the associative law, the carry is injected into any available lower level add structure of the appropriate significance. (The net effect is sometimes referred to as a flush adder.) The final result, of course, is the reduction of the $n/2$ multiples into two result segments —the sums and the carries—which are then assimilated by a conventional carry-propagate adder with an anticipation mechanism. The basic problem is the implementation of this algorithm. For large n, the plane segments that partition the algorithm are inconsistent with conventional packaging sizes and interconnection limitations (see below).

Consider a simpler variation [Fig. 4(b)] [5]: here only n/m multiples of the multiplicand are retired per iteration (m iterations are required). The n/m multiples are decoded into $n/2m$ additions of the multiplier. These multiples are

then inserted into the first level of the adder tree [Fig. 4(b)]. Now the add assimilation of both carries and sums proceeds as before with the introduction of intermediate staging points where the results are temporarily latched. The storage points act as skew (relative timing) control and improve the overall execution bandwidth efficiency of the algorithm. After the first set of multiples has been inserted into the first level of the tree and assimilated, it is latched at the first latch point. The storage point serves to decouple the first level from subsequent levels of the hardware; therefore the first set of multiples may now proceed to be absorbed in the second level and, simultaneously, a second set of multiples may be inserted into the first level of the tree. The process continues and, at the bottom of the tree, the carries and sums are assimilated in an adder with carry propagation. Notice that the latency in this variation may be twice that of the original algorithm. Also notice that potential bandwidth in this algorithm has not suffered since a second multiplication may proceed independently as soon as the last set of multiples is inserted and has passed the first level of the tree. Implementations of this second scheme have yielded a performance of 180 nanoseconds for a floating-point multiply including post shifting and exponent updating (56-bit fraction, Systems 360 format).



(a)



(b)

Fig. 4. Outline of a multiplier tree.

Figure 4, parts (a) and (b) are three-dimensional representations of the second variation of the multiply algorithm. Figure 4(a) is an isometric projection and Fig. 4(b) is a cross-section (or "regular cutting plane") and profile view. This representation was chosen to illustrate some of the difficulties of implementing high-speed systems in general. It is well known [1] that propagation delay and nonuniform transmission line loading are major factors in the switching speed of a logic stage. One would, therefore, desire a physical package consistent with the "natural" communication pattern of the algorithm so that the implementation could be optimized. The difficulties of a planar package are obvious. Communications between "regular cutting planes" [profile view, Fig. 4(b)] must be made axially in the physical implementation. Indeed, there is no assurance that the capacity of the physical package-plane will match the requirements of the regular cutting plane—several package-planes may be required.

Notice that the first variation of the algorithm had much more extensive requirements per plane, thus its implementation would very likely be less efficient than the second variation.

3) Floating Point Divide [5], [10], [11]: Historically, divide has been limited by the fact that the iterative process was dependent on previous partial results. Recently, attention has been given to techniques which do not have this limitation. These techniques include use of Newton-Raphson iterations for series approximations to quotients.

Assume

$$\text{Quotient} = \frac{a}{b}$$

Let

$$\frac{1}{b} = \frac{1}{1-(-x)} = 1 - x + x^2 - x^3 + \cdots \pm x^n \mp \cdots$$

which can be rewritten as

$$\frac{1}{b} = (1 - x)(1 + x^2)(1 + x^4) \cdots (1 + x^{2m})$$

Thus:

$$\text{Quotient} = a \cdot (1 - x) \cdot (1 + x^2) \cdot (1 + x^4) \cdots (1 + x^{2m}).$$

Recall that in binary the factor $(1+x^m)$ is related to $(1-x^m)$ by a complementation operation. Thus, the denominator is rewritten as $(1+x)$, complemented to form $(1-x)$, the product is $(1-x^2)$ which is complemented to form $(1+x^2)$, which continues the development.

Notice that the speed is substantially enhanced by the presence of a high-speed multiplier. In fact, as many as two multiplications might proceed simultaneously, one for the quotient development and one for the next denominator term, if the hardware permits. In forcing quick convergence of the series, usually a table lookup arrangement is used to provide the first approximation for the quotient. Thereafter, subsequent iterations develop double the precision of the first approximation. Floating-point divides under 700 nanoseconds have been reported using this scheme [5].

4) Algorithms for Achieving Maximum Efficiency in the Concurrent Execution of Independent Units: One of the purposes in having independent units dedicated to individual instruction types is to improve the execution of each of the instruction classes. The other advantage is that these independent units may operate concurrently and serve to increase the overall bandwidth of the execution unit. Tomasulo [4] describes an elegant algorithm to allow the achievement of maximum efficiency in the concurrent execution of units (Fig. 5, illustration from Amdahl [1]). The operation of this algorithm takes advantage of the latent time following the issuing of the request to access the operands from storage into the memory in the execution area (either virtual or addressable buffers). When an instruction is forwarded to the execution area, a word in the execution unit memory is reserved for its operand, but the operand has not yet arrived. For example, if the instruction to be performed consisted of a multiply, then it could be immediately forwarded to the multiplication unit. The multiplier would require a tag representing the operand. The execution area is provided with a common data bus and the tag representing the operand is broadcast on the bus one cycle early, then the execution functional units examine their queues of required operands and gate in immediately an appropriate one without waiting for it to go to the buffer storage. The tag forwarding frees the buffer storage of having any responsibility for this particular operand. Of course, at the same time, a load could then be executed into that same word. Thus, the second load instruction and the multiply instruction could proceed concurrently in a fashion impossible before. Results, as they become available, might also be forwarded to units (the adder in Fig. 5) requesting action by a similar mechanism, rather than proceeding directly to storage, whether virtual or addressed. This avoids intermediate stops and hence improves overall efficiency of execution by improving the overlap in the concurrency system.

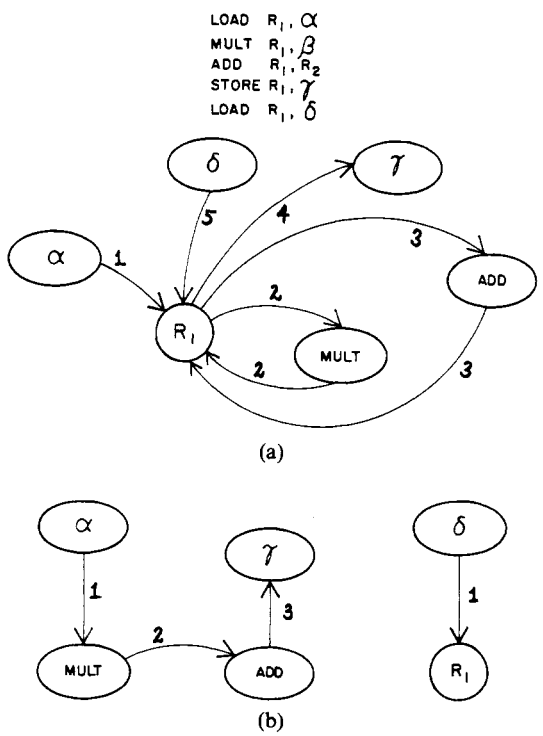


Fig. 5. Effect of Tomasulo concurrency algorithm. (a) Sequence with conventional dependency. (b) Sequence as performed after tag was forwarded.

C. Branching

Assume for the moment that memory bandwidth and execution bandwidth have now been arranged so that they more than satisfy the requirement of one instruction processed per decode cycle. What then would limit the performance of a SISD system? If it is assumed that there are a fixed number of data-dependent branch points in a given program, then, by operating in a confluent or other high-speed mode, we essentially bring the branch points closer together. However, the resolution time of the data-dependent branch point is basically fixed for a system with a given execution and/or accessing latency.

The basic retrogressive factor is the presence of branch dependencies in the instruction stream. Among the many types of branch instructions, we have [2], [7]:

1) *Execute:* The operand which is fetched is to be treated as an instruction (or an instruction counter). This presents some problems since the access latency is inserted into the instruction stream. It is essentially the same problem as indirect addressing or a double operand fetch. One could anticipate some of this difficulty by keeping ahead in the instruction stream. However, this particular instruction does not pose a series degradation problem because its use is normally restricted to linkages.
2) *Branch Unconditional:* The effective address so generated becomes the contents of the instruction counter. This subclass of instruction presents the same problem as execute.
3) *Branch on Index:* The contents of an index register is decremented by one on each iteration until the register is zero, upon which the alternate path is taken. Only the access latency is a factor since zero can be anticipated.
4) *Data-Dependent Branch:* The paths of the branch are determined by the condition (sign, bit, status, etc.) of some data cell. Invariably, this condition is dependent on the execution of a previously issued instruction.

Here the degradation is serious and unavoidable. First the execution of the condition-generating data must be completed. Then, the test is made and the path is selected. Now, the operands can be fetched and confluency can be restored. It is presumed that both instruction paths were previously fetched—but note that this is done only at the expense of greater memory bandwidth requirements. Another slight improvement can be made if the operands are fetched for one alternative path so long as the unresolved branch path is not executed (to avoid serious recovery and reconstruction problems if the guess proves wrong).

Of the two "loop closing" or conditional branch instructions, Branch on Index has less degradation due to resolution of latency than Data-Dependent Branch. When an option exists (as in the performance of a known number of iterations) the programmer should select the former.

Assuming that the Data-Dependent Branch-resolving latency is a constant for any particular machine, we may show its relationship (Fig. 6) to performance by assuming various percentages (of occurrence in executed code) of this type instruction. The latency represents the sum of average execution time plus operand access time from memory.

Figure 6 assumes that the organization under consideration has enough confluency to perform one instruction/cycle on the average without branch-resolution interruptions. To resolve the branch, it is generally necessary to fully execute and test the result of the preceding instruction. During this time, fetching of instructions and data may proceed for one branch path and possibly a few instructions may be fetched for the alternate. Fetching of both paths doubles the bandwidth required and increases the waiting time (Fig. 3). Thus the latency includes an average execution time, test time, and a percentage (wrong path guesses) of the operand-access time.

Notice that while we have studied degradation due to latency for the SISD organization, multiple stream organizations may exhibit branch induced degradation due to either this same phenomenon or an analog spatial inefficiency in which only one path activates or determines the outcome of a dependency.

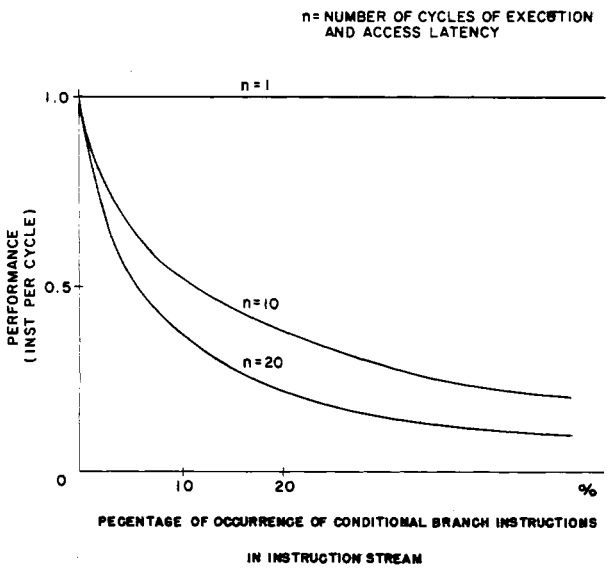


Fig. 6. Degradation due to data-dependent branch instructions.

Classes of Organization

In this section we shall consider each of the organizational classes listed in the first section of the paper. We will remark on or illustrate representative systems that fall into each class. The various configurations are by no means exhaustive.

A. Confluent SISD

The confluent SISD processor (IBM STRETCH [7], CDC 6600 series [8], IBM 360/90 series [2]–[5]) achieves its power by overlapping the various sequential decision processes which make up the execution of the instruction (Figs. 1 and 2). In spite of the various schemes for achieving arbitrarily high memory bandwidth and execution bandwidth, there remains an essential constraint in this type of organization. As we implied before, this bottleneck is the decoding of one instruction in a unit time, thus, no more than one instruction can be retired in the same time quantum, on the average. If one were to try to extend this organization by taking two, three, or n different instructions in the same decode cycle, and no limitations were placed on instruction interdependence, the number of instruction types to be classified would be increased by the combinatorial amount (M different instructions taken n at a time represents M^n different outcomes) and the decoding mechanism would be correspondingly increased in complexity. On the other hand, one could place restrictions on the occurrence of either specified types of instructions or instruction dependencies. This, in turn, narrows the class of problems for which the machine is suitable and/or demands restrictive programming practices. Indeed, this is a characteristic of multiple stream organizations since the multiplicity (or "parallelism") implies independent simultaneous action.

B. SIMD [14]–[18]

SIMD-type structures have been proposed by Unger [14], Slotnik [15] (SOLOMON, ILLIAC IV), Crane and Githens [16], and, more recently, by Hellerman [17].

SOLOMON is the classic SIMD. There are n universal execution units each with its own access to operand storage. The single instruction stream acts simultaneously on the n operands without using confluence techniques. Increased performance is gained strictly by using more units. Communication between units is restricted to a predetermined neighborhood pattern and must also proceed in a universal, uniform fashion [Fig. 7(a)]. (Note: SOLOMON has been superseded by ILLIAC IV as a system being actively developed, which is no longer completely SIMD.)

The difficulties with SIMD are:

1) Latency in the instruction stream for SIMD branches is now replaced by latency in the data stream caused by

operand communication (forwarding) problems.

2) Presently, the number of classes of problems whose operand streams have the required communication regularity is not well established. SIMD organizations are inconsistent with standard algorithmic techniques (including, and especially, compiler techniques).
3) The universality of the execution units deprive them of maximum efficiency.

C. MISD

These structures have received much less attention [18], [19]. An example of such a structure is shown in Fig. 7(b). It basically employs the high bandwidth dedicated execution unit as described in the confluent SISD section. This unit is then shared by n virtual machines operating on program sequences independent of one another. Each virtual machine has access to the execution hardware once per cycle. Each virtual machine, of course, has its own private instruction memory and interaction between instruction streams occurs only via the common data memory. Presumably, if there are N instruction units then the bandwidth of the common data storage must be N times greater than the individual instruction storage. This requirement could be substantially reduced by use of a modified version of Tomasulo's tag-forwarding algorithm and a separate common forwarding bus.

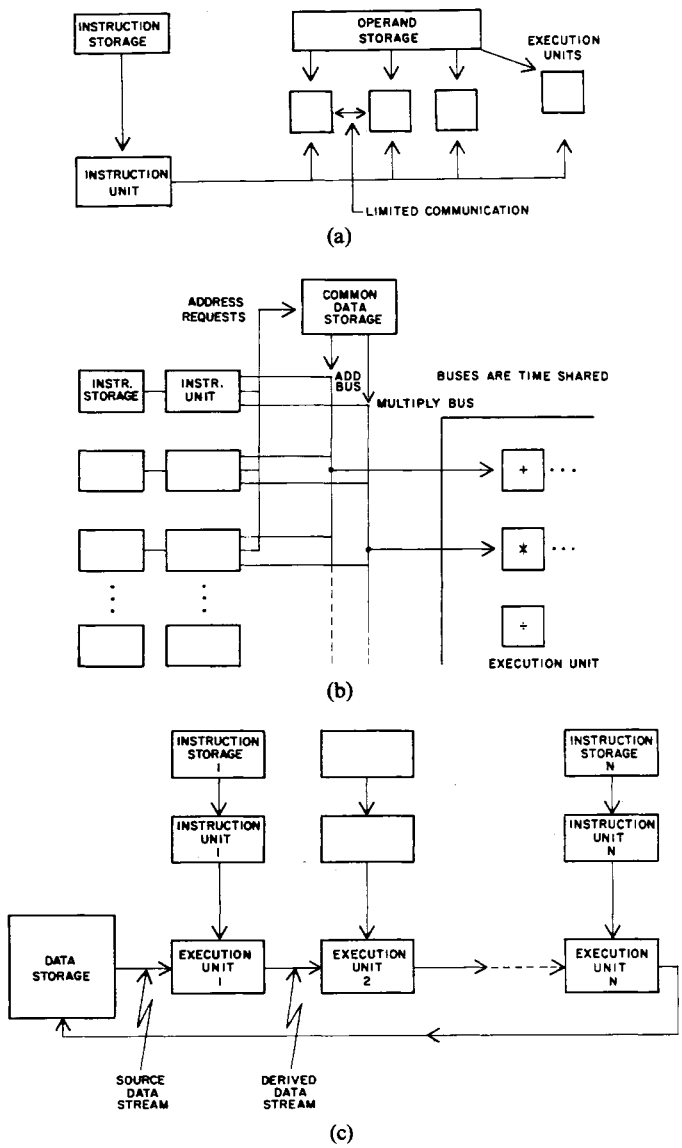


Fig. 7. (a) SIMD. (b) MISD. Converts to MIMD if instructions and data are privately maintained (together). (c) MISD.

Another version of this [Fig. 7(c)] would force forwarding of operands. Thus the data stream presented to Execution Unit 2 is the resultant of Execution Unit 1 operating its instruction on the source data stream. The instruction that any unit performs may be fixed (specialized such that the interconnection (or setup) of units must be flexible), semi-fixed (such that the function of any unit is fixed for one pass of a data file) or variable (so that the stream of instructions operates at any point on the single data stream). Under such an arrangement, only the first execution unit sees the source data stream and while it is processing the ith operand the ith execution unit is processing the ith derivation of the first operand of the source stream.

D. MIMD

If we reconstruct the organization of Fig. 7(b) so that the data and instruction streams are maintained together in private memories, we have an example of MIMD. There is no interaction or minimum interaction allowed between these virtual machines. So long as the latent time for execution of any operation is less than the memory cycle, no problems arise due to branching. Thus, such an arrangement allows maximum advantage of the allowable bandwidths of execution unit and memory unit. Of course, such an approach might well be criticized on the basis that the requirement of independence of the instruction sequences does not address itself to the requirements of large scientific problems. It would be better suited to the needs of the time-sharing and/or utility environment.

This restricted MIMD points up the shortcoming of our organizational definitions. The specifications fail to include a classification of how the streams may interact, thus a restricted MIMD may be organizationally much simpler than a confluent SISD.

General MIMD structures have been more widely described [20]–[23] with large-scale multiplicity being envisioned by Holland [21] and more restricted implementations being undertaken by Burroughs and Univac [23].

In his original proposal, Holland considers an array of processors each with a one-word storage. The modules are independent and are capable of concurrent execution. The processors have arithmetic ability but communication is limited by their need to "build a path" to the appropriate operand. In path building, the displacement and direction of the operand is specified and the intervening module processors form a vinculum.

Such an arrangement might solve some of the essential blockage problems of SISD and SIMD, since independent program segments proceed simultaneously. Also, memory bandwidth is always adequate.

The difficulties with such an arrangement of MIMD generally include [20]:

1) interconnections between units, which pose serious interference problems
2) the universal nature of the individual module, which limits its efficiency
3) the class of problems which could utilize MIMD organization, which is presently small.

Systems Requirements

The effectiveness of the overall computing processes must be measured on a basis much larger than performance (nanoseconds per instruction) alone. Even on this primitive basis, it is obvious that different instruction repertoire may imply substantially different effectiveness with the same average nanosecond per instruction ratio.

Despite our original assumption that the very large problem does not employ significant input/output, it is soon realized that this requirement is unduly restrictive. Presently, high-speed systems remain so limited. It may be some time before broader effective utilization may be realized through new developments in input/output equipment.

Of course, the overall measure of efficiency of the processor is the number of correctly completed computations and programs done over an extended period of time. Included in this measure is the reliability and the maintainability of the system. Complex systems with very large numbers of components are naturally very difficult to maintain, unless features which provide fault location are included. A major component of such features would be checking (or fault detection) of all operations and data transfers. On detection of error, hardware-aided diagnostics should be provided so that servicing and maintenance might be readily and easily accomplished. If checking is not included in the hardware, then it is, of course, incumbent on the user to program a thorough check of his results. This, of course, represents an overhead which penalizes the effective performance and utilization of the equipment.

Notice that some of the suggested organizations cater to restrictive problem sets—particularly the SIMD, where multiple operands are executed by the same instruction stream. Such organizations are clearly limited for general-purpose processing where there may be many interactions between operand elements. Similarly, with the MISD organizations, in the organization shown on Fig. 7(b), a number of independent instruction streams are involved and their very independence precludes MISD use on one large problem composed of essential dependencies; this, of course, is typical of large scale scientific programming.

Conclusions

Conventional Single Instruction Stream-Single Data Stream (SISD) processors may be enhanced by concurrency (confluence) of instruction handling, operand acquisition, and execution. There is, however, a limitation of the order of the execution of one instruction per instruction decoding time, and this may be further degraded by the occurrence of "conditional branch" instructions.

By multiplexing the instruction stream or the data stream or both, new classes of processors which do not necessarily share this limit are developed. Their effectiveness depends on the nature of the problem and it is an open question as to whether new algorithms which will serve to extend their usefulness can be developed.

Acknowledgment

The author is indebted to D. Jacobsohn and R. Aschenbrenner of the Argonne National Laboratory for several valuable discussions on some of the material.

References

[1] G. M. Amdahl and M. J. Flynn, "Engineering aspects of large high speed computer design," *Proc. Symp. on Microelectronics and Large Systems*. Washington, D. C.: Spartan, 1965, pp. 77–95.
[2] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, "The Model 91: machine philosophy and instruction handling," *IBM J. Res. and Dev.*, November 1966.
[3] L. J. Boland, G. D. Granito, A. U. Marcotte, B. M. Messina, and J. W. Smith, "The Model 91 storage system," *ibid.*
[4] R. M. Tomasulo, "An efficient algorithm for automatic exploitation of multiple execution units," *ibid.*
[5] S. F. Anderson, J. Earle, R. E. Goldschmidt, and D. M. Powers, "The Model 91 execution unit," *ibid.*
[6] L. W. Cotton, "Circuit implementation of high-speed pipeline systems," *1965 Proc. AFIPS FJCC*, p. 489.
[7] W. Buchholz, Ed., *Planning a Computer System*. New York: McGraw-Hill, 1962.
[8] J. E. Thornton, "Parallel operation in the Control Data 6600," *Proc. AFIPS 1964 FJCC*, pt. II, pp. 33–40.
[9] I. Flores, "Derivation of a waiting-time factor for a multiple bank memory," *J. ACM*, vol. 11, pp. 265–282, July 1964.
[10] M. Lehman, D. Senzig, and J. Lee, "Serial arithmetic techniques," *Proc. AFIPS 1965 FJCC*, pp. 715–725.
[11] R. E. Goldschmidt, "An algorithm for high-speed division," M.S. thesis, Mass. Inst. Tech., Cambridge, June 1965.
[12] C. S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. on Electronic Computers*, vol. EC-13, pp. 14–17, February 1964.
[13] D. Jacobsohn, "A suggestion for a fast multiplier," *IEEE Trans. on Electronic Computers (Correspondence)*, vol. EC-13, p. 754, December 1964.
[14] S. H. Unger, "A computer oriented toward spatial problems," *Proc. IRE*, pp. 17–44, October 1958.
[15] D. L. Slotnick, W. C. Borch, and R. C. McReynolds, "The Solomon Computer—a preliminary report," *Proc. 1962 Workshop on Computer Organization*. Washington, D. C.: Spartan, 1963, pp. 66–92.
[16] B. A. Crane and J. A. Githens, "Bulk processing in distributed logic Memory," *IEEE Trans. on Electronic Computers*, vol. EC-14, pp. 186–196, April 1965.
[17] H. Hellerman, "Parallel processing of algebraic expressions," *IEEE Trans. on Electronic Computers*, vol. EC-15, pp. 82–91, February 1966.
[18] D. N. Senzig and R. V. Smith, "Computer organization for array processing," *Proc. AFIPS 1965 FJCC*, pp. 117–129.
[19] R. Aschenbrenner and G. Robinson, "Intrinsic multi-processing," Argonne National Laboratory, Argonne, Ill., ANL Tech. Memo. 121, June 1964.
[20] W. Comfort, "Highly parallel machines," *Proc. 1962 Workshop on Computer Organization*. Washington, D. C.: Spartan, 1963, pp. 126–155.
[21] J. H. Holland, "A universal computer capable of executing an arbitrary number of sub-programs simultaneously," *1959 Proc. EJCC*, pp. 108–113.
[22] R. Reiter, "A study of a model for parallel computation," University of Michigan, Ann Arbor, Tech. Rept., ISL-65-4, July 1965.
[23] D. R. Lewis and G. E. Mellen, "Stretching LARC's capability by 100—a new multiprocessor system," presented at the 1964 Symp. on Microelectronics and Large Systems, Washington, D. C.