It has recently been suggested that the association of serial-mode functional (arithmetic) units with multi-instruction counter, multiprocessing computing systems may result in highly efficient processing complexes. This follows from the fact of life that in going from parallel to serial realizations of various algorithms the hardware requirements fall much more rapidly than does the speed. Furthermore the speeds of the slower, serial, arithmetic units may be more closely matched to those of memory and to other parts of the system. Thus the need for extra control circuitry, for high-speed registers, queueing circuits and look-ahead control, for example, is reduced and the system's overall cost/performance ratio improved. For appropriate serial configurations then, performance may be improved relative to a parallel system when system costs are to be held constant. Reconfiguration of a fast parallel circuit, for example, can yield a number of slower serial devices which, when kept busy, will increase throughput of the system.

The crux of the problem of optimizing throughput lies, of course, in the extent to which the large fast unit on the one hand and the several associated, slower, smaller units on the other can be kept busy; that is, their relative efficiency within an operating computing environment. As previously demonstrated, this consideration leads quite naturally to the multiprocessing configuration.

Recognizing that serial-mode operation may once again assume significance in the field of high-speed processing, we now wish to outline some appropriate algorithms. Thereby we hope to draw attention to techniques other than those currently considered during architectural, system and logical design activities. We shall not, however, in general, include detailed assessment of performance (speed and/or hardware) nor engage here in comparisons of serial and parallel devices. Such general comparative studies are meaningless when divorced from technologies and actual designs. Extrapolations from them to actual physical design situations are in general not valid and their usefulness is restricted to helping towards a deeper understanding of the various circuits and techniques discussed and compared.

**BASIC CONCEPTS**

**Serial Operation**

Traditionally a serial machine has been one operating on words, numbers, in a bit-by-bit fashion. We now intend to use the term to include bit-parallel, byte-serial, operation. Thus with a $k$-bit byte we may
conveniently regard the machine as operating radix $2^k$. We expect that the cost/performance ratio for the multiprocessing system considered, will be optimized for such serio-parallel operation. For the present discussion we consider $1 \leq k \leq 5$. This range arises from timing and speed considerations which suggest, for example, that the basic radix $2^k$ adder should be able to add two bytes in one bit-time as defined in the next section. Optimum $k$ will then be determined mainly by the fan-in and fan-out of the logical elements being used.

**Time**

In conventional high-speed machines the absolute measure of speed is generally established in terms of the average, effective time delay $t$ (nowadays in nanoseconds) experienced by a signal or signal pattern in passing through a logical element or level. Naturally $t$ will depend on the particular technology being used, that is, on the logical complexity of the element and on its fan-in and fan-out characteristics. Further, for a given product line $t$ will spread over a range of values and the in-place delay of a circuit will depend on its loading. However, we may use $t$ symbolically to indicate speed, giving the duration of a specified operation to a first order of accuracy in terms of the number of logical levels its implementation requires.

In designing synchronous serial circuits it is not necessary or indeed desirable to synchronize signals at each logical level. Thus there emerges a new concept, the bit-time, $t_b$, which represents the time interval between clock signals. $t_b$ then represents the maximum number of logical circuits or levels between points at which clock signals are applied. A lower limit for $t_b$ arises from the maximum operating rate of a flip-flop (latch) assembled from, or at least equivalent to, two or more logical elements. In general, $t_b$ will exceed $2t$ and a practical operating range will be $2t \leq t_b \leq 4t$. Thus typically we may consider elements with a logical structure of complexity comparable to that of an AND/OR complex, with fan-in and fan-out in the range 5 to 10, an average delay of order $1$ns, and operating at a rate of order $3 \times 10^8$ bits per second, that is a bit time of order $3$ns.

When discussing serial techniques we also make frequent use of the term "word-time" which in an $n$-bit binary machine generally refers to $n$ bit-times.

In an $\frac{n}{k}$-byte, radix $2^k$ system, processing the $k$-bit bytes in parallel, the word-time will be $\frac{n}{k}$ bit-times.

**Registers**

The serial shift-register is a component whose ready availability is fundamental to the concept of serial operation. In the past these registers have usually taken a distributed, dynamic, transient form; delay lines (ultrasonic, magnetostrictive, electric) and magnetic drum circulating tracks all having been used. Discrete, static, permanent registers using tubes or transistors have, however, proved economical, serving as both storage registers and shifting devices. With the second type of device it is simple to arrange for fully parallel input or output so that the register may also serve as a serial/parallel converter. This function is of importance in the general type of system where information is processed serially, in parallel, or in some more general mode, performance being optimized at each point in the system according to appropriate parameters of the algorithm being implemented. Typically we might wish to read from memory in parallel but perform arithmetic on the extracted number byte-serially.

The discrete, static shift-register appears to be the ideal candidate for realization in integrated or monolithic form when configurations larger than simple gates or adders are considered. Thus extensive use of its properties in systems design should prove rewarding. Its expected ready availability in the technologies of the future will in fact strongly reinforce the trend to increased use of bit-by-bit processing techniques.

**SHIFTING**

Shifting is a basic operation in its own right and is also an essential part of the synthesis of various other functions. In particular it may be required for alignment and fast normalization in floating-point operations and as an integral part of multiplication and division algorithms. Information stored in a shift-register may be immediately shifted any number of places at a rate or order $t_b$'s per bit or byte, a rate comparable to that of the parallel, cyclic shifter. In the most general circuit this shifting can occur in either direction. For more economical devices shift-
ing, say, to the right only, a left $p$-place shift will re­
quire \( \binom{n}{k} \) right shifts with end-around feed. For 
long shifts, the simple circuit is slower than a full 
pyramid-shifting circuit but by using radix $2^k$ shift-
ning registers with multiple path interconnections, as 
in Fig. 1, the maximum serial shift time for any num­
ber of places shifted can be kept below some $2k$-bit 
times. This arrangement permits shifts of any size 
to be obtained. However if shifting can be restricted 
to be by byte, that is, radix $2^k$, it becomes practical 
to consider a very simple register structure as in 
Fig. 2.

![Figure 1. A bidirectional shift register.](image)

![Figure 2. Byte-shifting register.](image)

The requirement for shifts in floating-point opera­
tion is well known. The particular techniques appro­
priate to serial-mode operation are perhaps not so 
familiar. In addition, for example, it is normally 
necessary to obtain a relative shift between two num­
ers and the shift register then has properties like 
those of the serial matrix store.\(^5\) That is, the contents 
of one register are “shifted” relative to those of 
another simply by delaying the application of shift­
control pulses to one or other of the two registers. 
An alternative technique, faster but requiring more 
hardware, would use shifting registers with several 
output points (a tapped “delay-line”). Relative shifts 
are then obtained by choosing the appropriate output 
point.

When dynamic storage devices are being used, 
tapped delay lines with elements of delay $t_b$, $2t_b$, $4t_b$, 
etc., have also been used\(^6\) to permit the simple control 
of from one to $n$ place shifts directly from a shift 
control register.

**ADDITION**

Techniques for serial, fixed point addition in a full 
adder, with dynamic or flip-flop storage of the carry, 
are well known. For radix-$2^k$ operation, the byte sum 
should be available in one bit time and this can be 
done in a $k$-bit parallel adder using a direct imple­
mentation of the radix sum-logic or binary full-adders 
with a carry speed-up network.\(^4\) It is largely the 
practical implementation of this one bit-time parallel
adder that determines an upper bound for $k$. The fixed point addition time will be $\frac{n}{k}$ bit times or $\frac{nt}{k}$, where $t_b$ is the bit time to logic-level-delay ratio. This time must be compared with the time $(1 + \log n) \cdot t$, a lower bound on the speed of any adder which follows from the result of Winograd. That is, the ratio of serial to parallel addition time will be less than

\[ \frac{n}{k (1 + \log n)} \cdot \frac{t_b}{t} \]

For example, if $n, k$ and $t_b$ equal 50, 4 and 3 respectively, the serial by byte adder will be less than 6.6 times as slow as the best possible parallel adder. The serial adder unit will require relatively less carry speed-up hardware (if any) than a single stage of the parallel unit so that the hardware ratio between parallel and serial devices will be in excess of $\frac{n}{k}$. Thus the cost/performance ratio of the serial unit is considerably better than that of the parallel device, an improvement which is maintained or even increased when floating-point adders, multipliers and dividers are considered.

Floating-point addition algorithms require the addition to be preceded by an exponent comparison and, where the exponents are unequal, a shift. Typical ranges for the exponents will require them to be represented by some eight bits or from one to three bytes. It may prove advantageous to process the exponents in a small parallel unit though the time penalty for processing serially will be small, only some two bit times, in the case of three byte exponents. The alignment shift will be obtained using one of the techniques described in the previous section. As shown there, no physical shifting of the operands need be performed and the size of the alignment will not significantly affect operation time.

Normalization of the sum obtained from a floating-point adder appears to be standard programming practice and in many machines normalized addition only is provided. This additional (micro) operation is time-consuming and appears to serve no useful purpose whatsoever. The present discussion will not pursue this point. The authors could not, however, have reviewed techniques for serial normalization in connection with the addition operation without at least questioning its utility.

The most common technique for normalization shifts the sum argument one bit or byte at a time until an indication is obtained that standard form has been achieved. This cyclic procedure, widely used in both serial and parallel arithmetic units, is economical but slow and makes the duration of additions operand dependent. Its use, however, difficult to avoid in parallel units except through the provision of a large amount of extra hardware which may double the cost of the adder complex and also slow it down significantly. In serial units, however, a simple algorithm, used already in the Manchester Mark I computer, permits determination of the normalizing shift and hence the exponent increment during the actual addition process. A counter is required which is reset initially and every time two neighboring sum bits are found to be unequal. Otherwise it is incremented by one during each bit time of the addition process, yielding immediately on its completion the shift number/exponent increment. Use of the fast shifter then permits rapid completion of the floating-point addition.

An alternative, faster technique requires the provision of an additional register. As the sum output appears from the adder it is received in the conventional fashion by the sum register. A second register is connected by a parallel transfer mechanism to the sum register and receives its contents whenever the exponent increment counter is reset to zero, that is, whenever two succeeding sum bits are unequal. Thus on completion of the addition the normalized sum will appear in the second register and no further shifting is required.

MULTIPLICATION

Exponent arithmetic has been adequately covered in the preceding section and we may, in our discussion of multiplication, restrict ourselves to the fixed point variety, considering first the purely bit-serial device. The basic cyclic multiplier, requiring of order $n^2$ bit times, that is $n^2 t_b$, is well known. By using a number, $a$, of adders the multiplication time may be reduced to be of order $\frac{n^2}{a} \cdot t_b$. Thus in the Manchester Mark I machine 20 adders were used (with a 40-bit word) to yield a two basic-cycle multiplier. In the limit by using $n$ adders (or fewer than $n$ adders with a recoding scheme), as in Fig. 3, a two word time multiplier is obtained. The least significant half of the double length product is here available after one word time as is the most significant half, but the latter is in redundant (pseudo-sum and carry) form. Thus serial multiplication may be further speeded

From the collection of the Computer History Museum (www.computerhistory.org)
up by attaching a parallel carry-assimilation circuit as in Fig. 4 to yield a multiplication time of order \((n + \log n) t_b\). It is interesting to note that the scheme that now results is almost identical to the parallel stored-carry multiplier scheme first advocated by Burkes, Goldstine and von Neuman\(^8\) and implemented in the Illiac II.\(^9\)

The circuit may be further improved by recoding the multiplier bits in \(k_2\)-bit groups, using the output of the recoder to control the selection of appropriate multiples of the multiplicand in the selector circuits. Also we may revert to \(k_1\)-bit byte operations to obtain, as in Fig. 5, a multiplier array still containing only \(n\) full-adders. The development of the circuit may also be taken one stage further as in Fig. 6 by providing \(q\) such arrays each processing, say, \(\frac{n}{q}\)-bit portions of the multiplicand.

In all these arrangements we clearly recognize the inadequacies of the classical terms *serial* and *parallel* for classification purposes, nor is it terribly important to attach such a name to the circuit. What is
Figure 5. Partially parallel shift multiplier.

Figure 6. Multiarray shift multiplier.
important is that by starting with a serial approach to multiplier design and by making use of the properties of synchronous, sequential operation in and with a shifting register, there has emerged a circuit whose performance may, at much lower cost, approach that of the fully parallel inverted-pyramid multipliers. In the Appendix we quote some preliminary results for a multiplier design corresponding to the configuration of Fig. 5. These are compared with an iterative parallel multiplier designed to be of the same speed. These and other results substantiate a more general assessment that the cost/performance of the new configurations exceeds that of the parallel structures by a factor of between two and three.

DIVISION

The basic restoring and nonrestoring division techniques are inherently slow since they require a minimum of order \( n \) subtraction and/or addition cycles. They may be supplemented by one or more of a number of speed-up techniques, but Freiman's analysis indicates that in practice these alone cannot yield a speed-up factor greater than about four.

We restrict ourselves therefore to a brief discussion of a technique previously outlined for decimal dividers. The technique is illustrative of a class of division methods which integrate readily with fast multiplier circuits to form efficient arithmetic units.

Consider a required remainderless division:

\[
Q = \frac{a}{b}
\]

Suppose a factor \( m \) could be determined such that \( mb = 1 \), then \( Q = ma \). The algorithm to be described defines an iterative procedure for determining a sequence \( m_i \) such that

\[
b_i \pi m_i \rightarrow 1
\]

hence

\[
a_i \pi m_i \rightarrow Q
\]

both quadratically. Let

\[
b_0 = b, \quad b > 0
\]

\[
b_0 = 2 + b, \quad b < 0
\]

We may write

\[
b_0 = 1 + \delta \cdot 2^{-q} + \epsilon 2^{-r}
\]

\[
1 > |\delta| \geq 2^{-q} \text{ or } 0.1 > \epsilon \geq 0.
\]

Define

\[
m_0 = 1 - \delta \cdot 2^{-q} = 2 - \delta \cdot [b_0 \cdot 2^{-q}]
\]

Consider

\[
b_1 = m_0 b_0 = 1 - (\delta - \epsilon) 2^{-q} \geq 2 - \delta 2^{-r}
\]

\( b_1 \) deviates from one at least quadratically less than \( b_0 \). Generalizing we may thus define an iterative, convergent procedure

\[
b_{i+1} = m_i b_i
\]

\[
t_{i+1} = 2t_i
\]

\[
m_{i+1} = 2 - 2^{-i+1} [b_i \cdot 2^{i+1}]
\]

\[
a_{i+1} = m_i a_i
\]

\[
t_0 = 1
\]

\[
m_0 = 3/2 \quad b > 0
\]

\[
= 1/2 \quad b < 0
\]

This yields \( b_i \rightarrow 1, a_i \rightarrow Q \) to the accuracy of \( a, b \) in some \( 2 + \lceil \log_2 n \rceil \) iterations.

Convergence can be speeded up by determining an initial value for \( m_0 \) (that is, some \( m_i \)) from a wired-in table look-up procedure — as a function of the leading bits of \( b \). A 16-entry table based on four bits of \( b \), as in Table 1, is practical and would save two iteration cycles, producing a 32-bit quotient in 4 cycles or 33 to 64 bits in 5 cycles.

Table 1. A Typical \( m_0 \) Coding Table.

<table>
<thead>
<tr>
<th>Five Most Significant Bits of Normalized Positive Divisor</th>
<th>( m_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10000</td>
<td>1.1110</td>
</tr>
<tr>
<td>0.10001</td>
<td>1.1110</td>
</tr>
<tr>
<td>0.10010</td>
<td>1.1100</td>
</tr>
<tr>
<td>0.10011</td>
<td>1.1010</td>
</tr>
<tr>
<td>0.10100</td>
<td>1.1000</td>
</tr>
<tr>
<td>0.10101</td>
<td>1.1000</td>
</tr>
<tr>
<td>0.10110</td>
<td>1.0110</td>
</tr>
<tr>
<td>0.10111</td>
<td>1.0110</td>
</tr>
<tr>
<td>0.11000</td>
<td>1.0100</td>
</tr>
<tr>
<td>0.11001</td>
<td>1.0100</td>
</tr>
<tr>
<td>0.11010</td>
<td>1.0010</td>
</tr>
<tr>
<td>0.11011</td>
<td>1.0010</td>
</tr>
<tr>
<td>0.11100</td>
<td>1.0010</td>
</tr>
<tr>
<td>0.11101</td>
<td>1.0010</td>
</tr>
<tr>
<td>0.11110</td>
<td>1.0001</td>
</tr>
<tr>
<td>0.11111</td>
<td>1.0001</td>
</tr>
</tbody>
</table>
The algorithm leads to a divider faster than the shortcut technique only if a fast multiplier is available. Figure 7, drawn for simplicity for bit-by-bit operation, indicates that a circuit realization integrates well with the fast multiplier discussed in the previous section. For $k$-bit byte operation a total division time of order $\frac{n}{k} \lfloor \log_2 n \rfloor$ bit times should be attainable. This speed can be obtained using a single, split, multiplier circuit; since the length of the multiplier $m_i$ can always be held less than $\frac{n}{2}$ bits except in the last iteration cycle. In that cycle, however, only one product $a \cdot m_i$ has to be determined.

A first estimate of hardware requirements for the integrated multiplier-divider and its control has been made. The results are presented in the Appendix.

CONCLUSION

The paper has presented a review of some techniques for the realization of the common arithmetic function. It is considered that generalized performance figures for the various circuits are not of direct interest if based on a particular technology or system design. Care has therefore been taken to avoid repeated comparisons or claims. The results presented in the Appendix enable initial evaluation of the multiplier techniques discussed and their comparison, in terms of speed, cost, and cost/performance ratio, with more conventional units. They are included so as to indicate that some of the intuitive notions presented can be substantiated in fact.

Despite the title of this paper, the discussion has not been limited to techniques of a purely bit-by-bit or byte-by-byte nature. The aim has rather been to indicate that with serial processing as a starting point, algorithms and circuits can be developed that will bear comparison with the classical high-speed configurations. In general they will yield speeds slower than the latter. However there exists an ultimate barrier to ever higher processing speeds using parallel structures due to the saturation effects of circuit speeds, cost, and system complexity. Thus
the use of techniques such as those described in a multiprocessing, multi-instruction counter configuration is believed capable of yielding performance and throughput superior to that of presently envisaged conventional systems.

In the past designers have, on the whole, pursued an evolutionary path. With the very rapid development that has taken place in computer applications, in systems concepts and in technology it is, however, necessary from time to time to reexamine some of the more fundamental notions of machine structure. How, for example, should the fact that the cost of a monolithic, serial shift-register is likely to approach that of an individual flip-flop, with its input and output stages, affect systems structure? Decisions which were valid in 1947 are not necessarily valid in 1965 and concepts should not be rejected merely because they are unconventional or because they have been rejected or abandoned in the past. In essence therefore this paper suggests and illustrates, with one example, what may be gained by fundamental rethinking in approaching the problems of system and logical design.

APPENDIX

A circuit design for the serio-parallel multiplier has been completed. The implementation is straightforward and represents only the first round of the normal iterative, optimizing design procedure. Hence it is expected that the hardware requirements summarized below could be somewhat reduced. A matching design for a more conventional, iterative, pyramid multiplier as in Fig. 8 was undertaken so as to enable an objective comparison of the two approaches. Since the designer who undertook both designs (J. Lee) had had previous experience on the design of conventional multipliers, it is expected that any bias in the degree of optimization of the two designs would operate in favor of the latter.

All designs were undertaken for 24, 48 and 56 bit operands with a 3-bit byte. They included all logic, registers and powering elements required to produce a double-length product. Circuits used were of two basic types. The first: a current-switching

![Figure 8. Conventional iterative pyramid multiplier.](image-url)
NOR/OR circuit with a fan-in of up to 4, a fan-out of 10 on each of the 2 outputs and a direct connection (dotting) capability of 4 for either an additional AND or OR function. The second circuit, termed a spread-gate, yields functions, each of 3 out of 6 input variables, on 4 outputs. Fan-out and dotting capabilities remain at 10 and 4 respectively.

The comparative results of the designs are given in Table 2.

It should be noted that (in this age of integrated and monolithic circuits) numbers of circuits rather than transistor counts are the really significant cost characteristic. The latter has, however, been recorded to provide a standard of comparison with earlier designs and also reliability may ultimately be related to that figure.

This investigation was based on the serio-parallel design. That is, having obtained the results for that circuit, iterative multipliers were designed to require an approximately equal number of logic levels, that is, they were designed to be of approximately the same speed. An alternative would have required determination of the speed (logic-levels) of parallel multipliers using a like number of circuits. This approach proved impractical since the number of circuits permitted would have been too small to make a true iterative pyramid multiplier, with carry assimilator, possible.

The conclusions from the results of the table are clear.

The serio-parallel circuit is more than twice as "efficient" as the parallel scheme. This result is substantiated by a comparison with still another design for which estimates were available. This design was for a 48 bit multiplier yielding a single length product in some 31 levels and required about 10,000 circuits or 33,000 transistors.

A circuit estimate for a 24-bit integrated, bit-by-bit multiplier divider as in Fig. 7 has also been undertaken. Results are quoted in Table 3.

<table>
<thead>
<tr>
<th>Word-Length</th>
<th>Serio-Parallel</th>
<th>Parallel, Iterative Pyramid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bits</td>
<td>Circuits</td>
<td>Transistors</td>
</tr>
<tr>
<td>24</td>
<td>1,085</td>
<td>5,412</td>
</tr>
<tr>
<td>48</td>
<td>2,165</td>
<td>8,355</td>
</tr>
<tr>
<td>56</td>
<td>2,419</td>
<td>9,869</td>
</tr>
</tbody>
</table>

The table indicates that the cost of a multiplier-divider is some 50 percent greater than that of the multiplier alone. Moreover, division time is some three or four times as great as multiplication time. Enlarging the \( m_0 \) table from 3 to 6 bits costs some 250 circuits and yields a 25 percent increase in speed and would normally be well worthwhile. Finally, comparing Tables 2 and 3, we see the effect of changing from bit serial to byte serial operation. In the examples given the change to the 3-bit byte has resulted in more than doubling the speed for an additional hardware investment of only some 14 percent.

**REFERENCES**

1. M. Lehman, "Serial Mode Operation and High Speed Parallel Processing," *Information Proc-
serial arithmetic techniques


