Fully Asynchronous, Robust, High-throughput Arithmetic Structures

P. Patra and D. S. Fussell

Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA

Abstract

This paper presents some novel circuit designs for bit serial *adders and multipliers built out of some unusual, but well-defined, circuit primitives. The circuits are fully delay-insensitive, provide good reliability and speed, and are easily verified. The structures are flexible and handle inputs of arbitrary lengths while being asymptotically optimal in speed and area. The scalability of these circuits makes them very attractive for applications such as RSA cryptosystems which need very large operands and fast multiplication.*

1 Introduction and previous work

While most digital circuits to date have been designed as synchronous, clocked systems, there has recently been a surge of interest in asynchronous circuits, mainly due to the potential and realized advantages of asynchronous circuits for modern low-power computing and communication applications [6]. The class of "delay-insensitive (DI) circuits" – circuits whose external behaviors make no explicit reference to time and are independent of any (non-negative) delays in their internal components and wires – is a subclass of the class of asynchronous circuits. A delay-insensitive design discipline requires that a second signal is not sent along a wire (or channel) until its predecessor has been acknowledged [11]. This class holds a great potential for low-power applications as well due to factors such as (1) absence of global clock trees to be powered, and (2) the computations or signal activity being driven by 'useful-only' transition events (so, CMOS circuits use little energy when quiescent). The 'moderation' of current peaks is inherent because gates do not switch in a synchronous fashion, thus reducing noise and circuit degradation such as 'ground bounce'. Higher throughputs can result, without a high increase in design complexity, because of the absence of any critical global timing concerns and because circuits compute as fast as the data and the operations allow. Added benefits include resilience to scaling of IC feature size and to large variations in operating conditions such as temperature and voltage.

We give *2-phase*, transition-based designs for a 'flexible' bit-serial adder and a high-throughput, very flexible and scalable bit-serial multiplier. Bit-serial circuits are very important for signal processing as well as for many communication applications ([9, 2]). In an RSA Public Key Cryptosystem, each net user has exactly one private key d and one public key e, irrespective of the number of people on the net. A message M is encrypted by raising M to e to obtain cryptogram C, while M is recovered by raising C to d – all operations done modulo a known, large number N. The numbers involved are usually quite large (e.g. 1024 bits long) to make decryption without the private key computationally infeasible. A high-throughput and scalable multiplier for very wide operands, which make their parallel implementations impracticable, is highly desirable for, e.g., RSA cryptosystems [2, 12].

To reduce the area requirements of pure DI circuits, many researchers and practitioners ([5, 10]) introduce timing assumptions such as *isochronic forks*, bundled-data constraints, etc. or even perform timing simulations to control delays in various circuit paths in line with traditional asynchronous design methods. Of course, this compromise often leads to increased design complexity and erosion of the above-mentioned benefits of truly DI systems. Moreover, the application of the DI technique to data processing circuits has been less thoroughly explored. (The overhead of completion detection is considerable for many parallel-data applications.) Our paper explores the design of fully asynchronous serial-data processing circuits that are guaranteed to be race- and hazard-free and demonstrates many nice and interesting properties of such circuits at significantly more reasonable area overheads than suspected earlier. Some ideas of asynchronous pipeline processing ([8]) can help mitigate the problems of area overhead and throughput when using purely delay-insensitive modules. Furthermore, our designs use 'primitives' from a very small set suitable for implementation as a standard-cell library and amenable to automatic primitive-level optimization techniques.

Other attempts at asynchronous multipliers have been generally suitable for parallel implementation only ([3]).

2 Primitive modules and notation

We need a repertoire of building-block components or **primitives** to build general circuits. Depending on the level of abstraction, it may be just transistors or functional blocks such as memory and ALU. We consider design at the module level and we use the asynchronous primitives shown in Table 1 for realizing larger circuits as a network of these primitives. Each primitive's specification is given next to it in the *Trace Theory* ([11]) formalism.

In Trace-theoretic notation, symbol names are viewed as atomic **trace-structures** representing simple specifications. A symbol name standing for a port (terminal) is sometimes appended with '?' or '!' to denote clearly that the name stands for an input or output, respectively. A more complex specification is built recursively from the primitive specifications by applying following operations on trace structures: '**pref**' is prefix-closure; ';' is sequential composition, but often we will use mere juxtaposition to denote sequential composition. '$\|$' is parallel composition; '$|$' is non-deterministic choice, and '$\star$' is the Kleene-closure on appropriate trace-structures. (For a good introduction to trace theory for specifying circuits, see [1].)

We very often use the *Join* primitives. The $m{\times}n$-*Join* primitive, with the two arguments m and n left unbound, is operationally described as follows: It has m row inputs, n column inputs, and a matrix of $m \times n$ outputs— one for each pair of row and column inputs. The device and its environment repeat the following behavior: The device waits to receive exactly one row-input and exactly one column-input; upon receiving the two inputs it makes a transition on the output corresponding to the input pair. *Joins* are equivalent under swapping of the row and the column inputs. In schematics, we indicate the fanout from an output of a *Join* by drawing lines outward from a filled circle corresponding to the input row and the input column that excite it. ($m \times n$ filled circles are laid out in an array for a $m{\times}n$-*Join* module indicating its output terminals.)

Example:

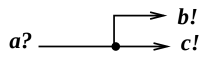
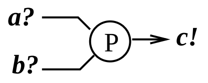
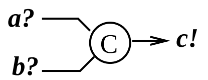
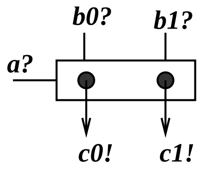
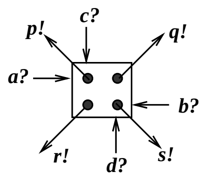
Name	Symbol	Specification
Fork		$\mathbf{pref}(a?(\,b!\|\,c!))^\star$
Merge		$\mathbf{pref}((a?\,|b?)\,c!)^\star$
$1{\times}1$-*Join*		$\mathbf{pref}((a?\|b?)\,c!)^\star$
$1{\times}2$-*Join*		$\mathbf{pref}(((a?\|b0?)\,c0!)$ $\|((a?\|b1?)c1!))^\star$
$2{\times}2$-*Join*		$\mathbf{pref}(((a?\|c?)\,p!)\ \|$ $((a?\|d?)q!)$ $\|((b?\|c?)r!)\ \|$ $((b?\|d?)s!))^\star$

Table 1: A set of DI primitives used here

We illustrate trace-theory notation using, as an example, a $1{\times}2$-*Join*, whose set of traces is given by: $\mathbf{pref}(((a?\|b0?)\,c0!)\ |\ ((a?\|b1?)c1!))^\star$. The symbols with the suffix '?' represent transition events at the three input ports of this module, namely, a, $b0$, and $b1$. Similarly, output symbols $c0$ and $c1$ represent two output actions (and their occurrences). Hence, the input set is $\langle\, a, b0, b1\rangle$ and the output set is $\langle\, c0,\ \ c1\rangle$. Examples of valid partial behaviors are: a, $a\,b0$, $a\,b0\,c0$, $a\,b0\,c0\,b0$, $b0\,a\,c0$, $b1\,a\,c1\,a\,b1\,c1$, $a\,b1\,c1\,b0\,a\,c0$.

Some invalid traces are: (1) $a\,b0\,b1$, (2) $b0\,c0$. (1) is invalid because the environment cannot send both column inputs $b1$ and $b0$ without an intermediate output from the *Join*. (2) is invalid because output $c0$ is produced too early.

A *Fork*, represented by branched lines, accepts one input and then produces two outputs, before repeating self. A *Merge*, also known as a 2-input *Xor*, can accept exactly one input which is followed by an output transition, before it repeats itself.

An "empty circle" at an input terminal of a *Join* device implies an initialization of the *Join* component such that it behaves as if a transition at that terminal has already occurred initially. We often use a circle labeled with 'P' as a pictorial short-hand for a (parity) tree consisting of one or more *Merges*.

3 The Adder

The Adder we wish to design is intended to receive two operand streams of serial data, representing two numbers, on the input *channels* A and B, and emit out the result of their addition as a bit stream (number) on channel D. The input numbers are of arbitrary sizes, the least significant bit (LSB) coming in (going out) first. The last bit of a stream is followed by a signal transition to indicate end of the number involved. The Adder is expected to *handshake* on all the three channels to communicate with its *environment* and to derive appropriate timing information, and to avoid any possible hazards/races irrespective of any delay in the wires or components (primitives) used in the design. To transmit a bit of information asynchronously we need two wires so that the *validity of the data* is encoded within the data itself ([6, 5]). There is also a return signal wire from the receiver to the transmitter to indicate when the current data item was received so that the transmitter can place the next data item. This is needed because there are no global timing assumptions that clocked digital circuits make. The only way for proper synchronization is via some form of 'handshake'. We use *2-phase*, i.e. 'non-return to zero', protocol to partially reduce the handshake time and the ensuing switching energy, and for its simplicity. (Often there is a a penalty in terms of some increased area or load-capacitance.) There is one more wire to signify the end of a number's bit stream.

We use a simple gray-coding of data, different from that suggested in [6]: The value 1 or 0 is associated permanently with one or the other wire of the pair of data wires in a channel. For example, a transition on wire $A_1(A_0)$ indicates a 1(0) value on channel A. A_e is the signal signifying the end of the current number in channel A. A_k is the handshake signal from the receiver of channel A to acknowledge receipt of a 0, 1, or the end signal. A transition itself signifies arrival of a new value or event – no 'spacers' required in this scheme.

The schematic for the Adder is shown in Figure 1(a). The Adder is viewed as a *delay-insensitive composition* ([1]) of two modules EqualizeAB and AddEqual shown in figures 1(b,c). EqualizeAB takes in two numbers A, B and outputs numbers G, H such that $A = G$ and $B = H$ and that the lengths $|G|$ and $|H|$ are equal. EqualizeAB appends a shorter input number with sufficient zeros, and generates the "end

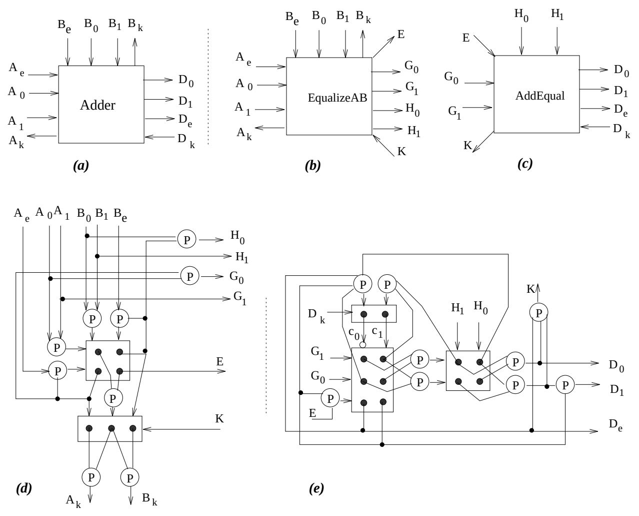


Figure 1: The Serial Adder: handles input numbers of any size

signal" E when it has seen the end of both input numbers. AddEqual is the adder that adds the numbers G, H and outputs the value on channel D and also acknowledges inputs G, H from EqualizeAB via wire K. Any carry/overflow bit from the most significant place during addition is emitted out upon seeing the end signal E, before D_e is generated to complete an addition operation. Note that EqualizeAB acknowledges each input data bit from the environment while delaying the "end" signal of the shorter operand number, if any, until the entire addition is complete.

The final DI decompositions of the two modules EqualizeAB and AddEqual into a set of *Fork*, *Join*, and *Merge*/*Xor* gates are given in figures 1(d) and (e), respectively. (Recall that a parity tree (circle over P) can be decomposed into one or more *Xors*, depending on the number of inputs. The behavior remains delay insensitive as long as at most one input transition occurs at a time.) [4] provides a formal verification of this Adder using *DI Algebra*. For our purposes, you may recall the behavior of a *Join* primitive and manually trace the signals to satisfy yourself that EqualizeAB and AddEqual operate as described above. We indicate the internal signals c_0 and c_1 which represent the carry from an addition of a pair of bits. The 3×2-*Join* and 2×2-*Join* act as two 'half-adders', except that the 3×2-*Join* can process the end signal to emit out any left over carry, from the most significant bit, to complete an addition of two numbers G and H. (This way two n-bit numbers can generate a $(n + 1)$-bit result.) Note that D_k, the acknowledgement from the D channel, serves to synchronize each bit-addition to avoid any possible data over-writes on

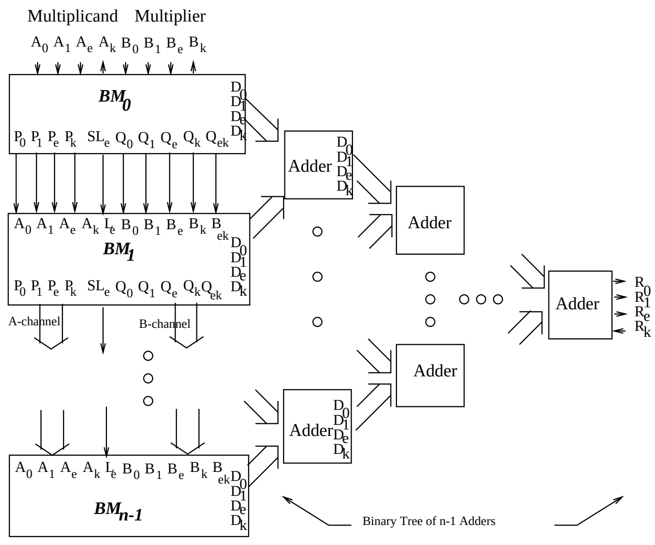


Figure 2: Top-level decomposition of the multiplier

channel D.

4 The Multiplier

We consider a bit-serial multiplier that bit-serially accepts, two unsigned binary operands A and B and produces a result product R. The multiplicand A is of arbitrary length. The design considers at most n least significant bits of the multiplier operand B. It ignores the rest. n is the design constant of our circuit. The result R is $|A| + max(n, |B|)$ bits long, where $|A|$ is the length of operand A, etc. We use A, B and R to stand for the corresponding channels also. The channels have four associated wires just as we have for the Adder circuit. We use 2-phase transition signalling as before.

Consider Figure 2. The two input bit streams of numbers are received bit-serially, LSB first, in their respective handshake channels from the top. The product is output through the R handshake channel, also LSB first. The multiplier circuit consists of a column of BM modules which is a linear pipeline shifting the appropriate bits down. The number of BM modules is n, the design constant. Each BM module generates a partial product that is fed to the tree of $n-1$ Adders on the right. (The Adder is the circuit we designed in the previous section.)

4.1 Decomposition of BM components

Figure 3 shows the DI decompositions of the BM modules. Note that the 'boundary' modules BM_0 and BM_{n-1} are some what simpler than the intermediate modules BM_i, $0 < i < n-1$. Each BM_i, $0 < i < n-1$, has an interface at the bottom which is identical to the one on the top. Each BM strips off a bit from the B stream to process and passes the rest down. On the other hand, each bit of A is processed as well as copied down. The P interface of an upper BM is one end of the channel whose other end is the A interface of the BM just below it. Q and B are related similarly. SL_e and L_e similarly form a pair. The 2×3-*Join* on the left handles the stripping and passing of the B bit stream. The two *Joins* on the right of a BM multiply the A bits with the particular bit stripped and saved by the 2×3-*Join*. This is how the partial product i is generated by BM_i, which then feeds it to an Adder on the right.

The L_e signal is an artifact to generate appropriate padding zeros so that each bit-column of the partial products has exactly n bits to be fed to the tree of Adders on the right. *The circuit is initialized such a way that a transition is generated on the SL_e output wire of BM_0. This wire can be merged with a system initialization signal to generate the appropriate transition (we have not shown this in the figure).* This initialization kicks in the initial generation of the zeros mentioned above. This is a crucial trick to obtaining this fast multiplier. These zeros are again generated after the A_e is received during a multiplication, making them ready for the next multiplication operation.

An additional signal wire Q_{ek} serves to acknowledge input B_e so a B operand of arbitrary length can be handled.

4.2 Properties of the multiplier

This DI multiplier has a latency of $\mathcal{O}(\log n)$ – the binary tree of Adders introduces that. When the inputs are ready after the present pair of operands are accepted, the effective latency can reduce to zero. This is possible because the BM column can start processing a new pair of operands before the complete $|A| + max(|B|, n)$ bits of the product are output. In other words, a constant through-put can be maintained if no "bubbles" are introduced into the pipeline by the environment. The maximum through-put in this case is limited by the propagation delay through an Adder, which is optimal and a constant independent of n. As noted above, the design is very robust and flexible in allowing variable length input operands.

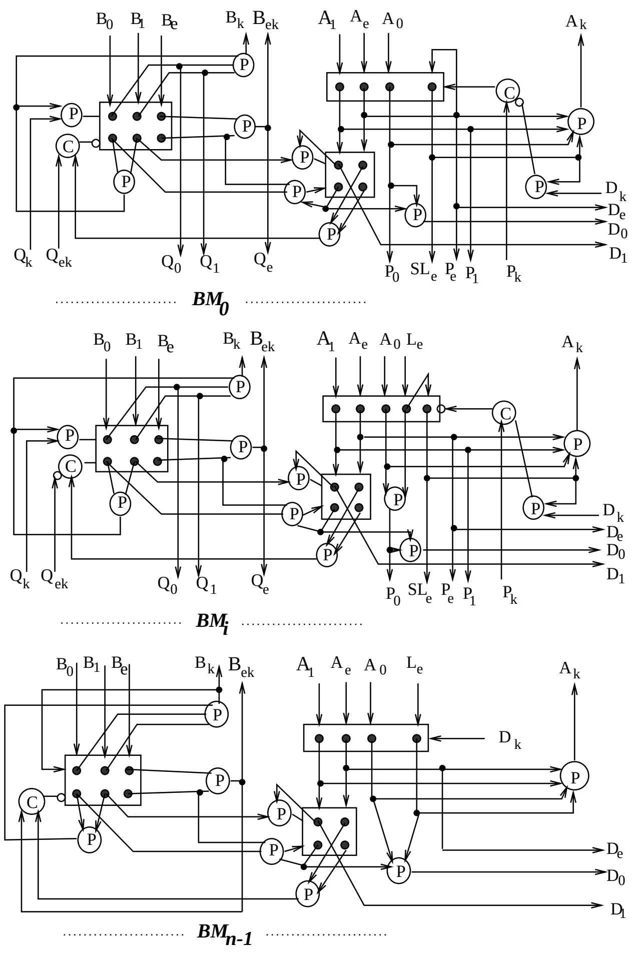
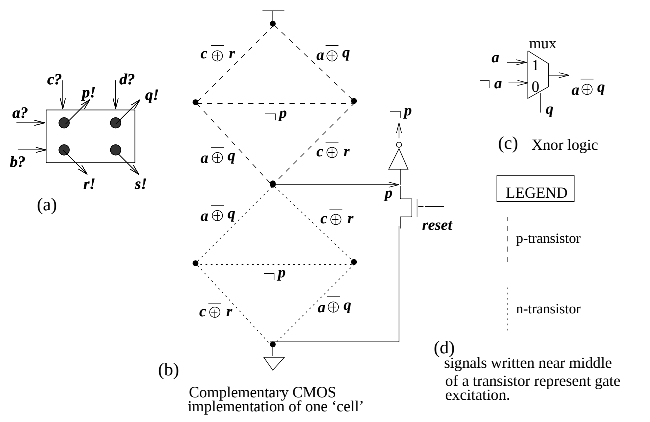


Figure 3: Decompositions of the 'shifter' components

5 Implementation issues and performance estimation

In the following we show area- and time- efficient and, yet, robust transistor designs for our primitives. All input and output wires are assumed to be initialized to a logical LOW, unless indicated otherwise by the "empty circles".

Figure 4(b,c) shows a representative implementation (and the legend) for 'one of four' quarters of a 2×2-*Join* corresponding to the four symmetric outputs. The multiplexor in Figure 4(d) was implemented as a 5-transistor Xnor gate with inputs suitable buffered. The "bridge" structure in fig.4(b) provides two parallel signal paths, aiding speed, while switching. The other 3 quarters of the 2×2-*Join* can be implemented by suitable renaming of external inputs and outputs. For initialization, the simplest approach of adding a strong n-mosfet at an output is adopted as shown in fig.4(b). Note that if all the



(a)
(b)
Complementary CMOS implementation of one 'cell'
reset
mux
(c) Xnor logic
LEGEND
p-transistor
n-transistor
(d) signals written near middle of a transistor represent gate excitation.

Figure 4: Implementation of 2×2-*Join*

inputs are guaranteed low during initialization, then only two outputs of 2×2-*Join* such as P and S, or Q and R need have the resetting n-mosfets.

Any larger *join* module can be DI-decomposed into the set $\{Fork,\ Merge,\ 2\times2\text{-}Join,\ 1\times1\text{-}Join,\ 2\times1\text{-}Join\}$ ([7]). Alternatively, they may be implemented directly if appropriate.

The HSPICE simulation of the 2×2-*Join* extracted from its unoptimized layout using MAGIC shows a propagation delay of 3.5ns in a 2-micron MOSIS SCMOS process, assuming a load of 8 minimum sized inverters at each output. Minimum-size, allowable N transistors with matching P transistors were used.

Throughput of the multiplier is bounded by the speed of the Adder, which has an approximate average delay of about 8ns. The depth of the Adder tree determines the latency (i.e., when the multiplier pipeline is empty). This depth is logarithmic in the length of the B operand. Hence, for a multiplier 1024 bits long, we roughly calculate the maximum latency to be approximately 105ns.

6 Summary and conclusions

We have presented novel circuit designs for bit serial adders and multipliers. The circuit structures are shown to be very flexible, and robust while being optimal in speed. These structures may also be adapted to circuits using other asynchronous techniques while trading off absolute delay-insensitivity for area and some speed gains. The scalability of these circuits make them very attractive for applications such as RSA cryptosystems using very large operands and fast multiplication. 'H-tree' type of layout may be used for the tree-like topology of the multiplier.

We have only emphasized the high-level "architectural design" of the serial arithmetic circuits presented here. There are certain practical efficiency problems to be overcome before the design styles become appealing for commercialization. The circuit primitives are relatively new to the VLSI world and therefore there exist no commercially available implementations for them. Moreover, there are no CAD tools available yet to assist in low-level optimization of the asynchronous circuits presented here. This leaves a lot open for future research.

Acknowledgements

The first author thanks Mark Josephs for encouragement and for providing the Adder circuit's algebraic verification in the unpublished joint work [4].

References

[1] Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. *Distributed Computing*, 5(3):107–119, 1991.

[2] Rodney M. Goodman and Anthony J. McAuley. An efficient asynchronous multiplier. In K. Bromley, S.-Y. Kung, and E. Swartzlander, editors, *Proceedings of the Second International Conference on Systolic Arrays*, pages 593–599. IEEE Computer Society Press, May 1988.

[3] Jaco Haans, Kees van Berkel, Ad Peeters, and Frits Schalij. Asynchronous multipliers as combinational handshake circuits. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 149–163. Elsevier Science Publishers, 1993.

[4] M.B. Josephs and P. Patra. An asynchronous bit-serial adder and its delay-insensitive decomposition. Technical report, Oxford University Computing Lab., Oxford, 1992. Unpublished Manuscript.

[5] Alain J. Martin, Steven M. Burns, T. K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proceedings of the Decennial Caltech Conference on VLSI*, pages 351–373. MIT Press, 1989.

[6] Anthony J. McAuley. Four state asynchronous architectures. *IEEE Transactions on Computers*, 41(2):129–142, February 1992.

[7] P. Patra and D.S. Fussell. Efficient building blocks for delay insensitive circuits. In *Proc. of International Symposium on Advanced Research in Asynchronous Circuits and Systems*, November 1994. (To appear).

[8] P. Patra and D.S. Fussell. Optimization of delay-insensitive circuits – a case study. Technical report, Dept. of Computer Sciences, The Univ of Texas at Austin, 1994.

[9] S.G. Smith and P. B. Denyer. *Serial Data Computation*. Kluwer Academic Publishers, 1988.

[10] Ivan E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[11] Jan Tijmen Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[12] I-Chen Wu. A fast 1-d serial parallel systolic multiplier. *IEEE Transactions on Computers*, C-36:1243–1247, Oct 1987.