A Spanning Tree Carry Lookahead Adder

Thomas Lynch and Earl E. Swartzlander, Jr., *Fellow, IEEE*

Abstract— **This paper describes the design of the 56-bit significand adder used in the Advanced Micro Devices Am29050[1] microprocessor. Originally implemented in a 1 μm design rule CMOS process, it evaluates 56-bit sums in well under 4 ns. The adder employs a novel method for combining carries which does not require the back propagation associated with carry lookahead, and is not limited to radix-2 trees as is the binary lookahead carry tree of Brent and Kung. The adder also utilizes a hybrid carry lookahead–carry select structure which reduces the number of carries that need to be derived in the carry lookahead tree. This approach produces a circuit well suited for CMOS implementation because of its balanced load distribution and regular layout.**

Index Terms— **Adders, carry lookahead adders, carry select adders, fast adders, fundamental carry operation, high radix lookahead carry, Manchester carry chain circuit.**

I. Introduction

CLASSIC high speed adders include carry lookahead [1], carry skip [2], carry select [3], and conditional sum [4] adders. A binary variant of the carry lookahead adder, the binary carry lookahead carry adder, was developed by Brent and Kung [5]. Another variant, based on reformulating the carry equations was developed by Ling [6]. Fast implementations of carry lookahead adders have been developed by Hwang and Fisher [7] in CMOS and Bewick *et al.* [8] in ECL. For gate level designs where each gate has unit delay, the carry lookahead adder greatly reduces delay with a modest increase in complexity relative to the ripple carry adder. In dynamic CMOS implementations where the delay depends on the gate size and loading, the use of Manchester carry chains [9] to realize the carry lookahead logic significantly reduces the gate loading, which produces a substantial speed increase.

This paper presents an adder based on a carry tree that produces carries without back propagation by using extra cells to span the gaps between branches in the tree. The idea of unidirectional carry production was originally presented by Brent and Kung [5]; however, their intermediate cell, based on the associativity of the carry combining operation, is limited to use in binary carry lookahead trees. The intermediate cell presented in this paper is based on both the associativity and the idempotency of the carry combination operation and it can be used to modify multiway carry lookahead trees.

Manuscript received October 15, 1991; revised March 12, 1992. This paper is a revised version of a paper which appeared in the Proceedings 10th Symposium on Computer Arithmetic, Grenoble, France, June 1991, pp. 165–170. © 1991 IEEE.

The authors are with the Department of Electrical and Computer Engineering, University of Texas at Austin, Austin, TX 78712.

IEEE Log Number 9201907.

[1] Am29050 is a trademark of Advanced Micro Devices.

In addition to the one pass carry tree, this implementation uses a modification of the known carry lookahead/carry select hybrid adder [3], [10] to reduce the number of carry outputs required from the tree. The traditional carry lookahead tree made from four bit blocks produces carries on integer power of four boundaries (0, 4, 16, 64 bits) without back propagation. If these outputs were used directly to select among ripple carry results, the top of the tree would dominate the evaluation time. It is possible to get carries for 0, 8, 12, 16, 32, 48, and 56 bits from a Manchester Carry Lookahead tree. This produces a better set of select lines, but the carry select section needs to be at least 16 bits wide. In contrast, the modified tree presented in this paper which is made from four bit blocks produces carries on the boundaries of 0, 8, 16, 24, 32, 40, 48, and 56 bits without back propagation, so the uniform carry select section is 8 bits wide. This even distribution of carry outputs makes for a reasonably balanced load and a regular layout.

A four way carry tree was selected because simulations indicated that it represented the optimum tradeoff between the number of levels in the tree and the length of the Manchester carry chains. For this adder the Manchester carry chains would have to be eight bits long to reduce the number of levels in the tree, but the delay increase of the Manchester carry chains would exceed the saving from reducing the number of levels. If the Manchester carry chain length is reduced to three, the number of levels increases to four which also results in an increase in total delay. Our optimization search was limited to the use of regular carry trees with all Manchester carry chains of the same length; mixing different Manchester carry chain lengths [11], [12] could probably improve the speed although it would reduce the regularity of the implementation.

The four way traditional carry tree was modified by adding one spanning cell in order to produce carries on eight bit boundaries. Eight bit boundaries proved to be convenient because the eight bit ripple carry adders operate in slightly less time than the carry tree, thus the sum data are set up at the selection multiplexers just before the select signals arrive. Also the eight bit multiplexers are small enough that their select lines do not present an excessive load to the carry outputs from the Manchester carry chains. A design issue in CMOS is whether to implement circuits in dynamic or static logic. Adders of size greater than a few bits will nearly always be smaller and/or faster when implemented in dynamic logic. However, a requirement of dynamic circuit design is that an idle clock phase must be available between circuit evaluations for the purpose of precharging the gates. This also implies the requirement that the operands be set up at the beginning of the evaluation phase.

The requirements for a precharging phase and operand set up alignment, are often easy to meet, especially in synchronous design. However, in some situations it would be convenient to perform an addition operation asynchronously, so a static design would be more desirable. This brings up the question of how to implement the "fco" operation which is beyond the scope of this paper. The theoretical analysis presented applies equally to all implementations of the "fco" operation, while the implementation discussion is only applicable to dynamic design.

II. The "fco" Operator

Notation: a_i and b_i denote the ith bits of the words to be added, p_i denotes that a carry will propagate across bit position i (i.e., $p_i = a_i + b_i$), and g_i denotes that a carry is generated at bit position j (i.e., $g_j = a_j b_j$). $p_{i:j}$ signifies that a carry will propagate from bit j (LSB) to bit i (MSB). Similarly $g_{i:j}$ denotes that a carry is generated in at least one of the bit positions from j to i inclusive and propagated to bit position i. The fundamental carry operation, "fco" introduced by Brent and Kung [5] is used:

$$(p_{i:j}, g_{i:j}) = (p_{i:k+1}, g_{i:k+1}) \text{ fco } (p_{k:j}, g_{k:j}) \tag{1}$$

which is defined as

$$p_{i:j} = p_{i:k+1}p_{k:j} \text{ and } g_{i:j} = g_{i:k+1} + p_{i:k+1}g_{k:j}.$$

Brent and Kung have shown the associativity of the "fco" operator. For $i > m > k > j$,

$$[(p_{i:m+1}, g_{i:m+1}) \text{ fco } (p_{m:k+1}, g_{m:k+1})] \text{ fco } (p_{k:j}, g_{k:j}) = (p_{i:m+1}, g_{i:m+1}) \text{ fco } [(p_{m:k+1}, g_{m:k+1}) \text{ fco } (p_{k:j}, g_{k:j})]. \tag{2}$$

The "fco" notation provides an interesting analogy between placing parenthesis in an equation and the different adder configurations. For example, the carry combination equation for a four bit ripple carry adder is

$$(((pg_0 \text{ fco } pg_1) \text{ fco } pg_2) \text{ fco } pg_3). \tag{3}$$

Equation (3) indicates that the propagate and generate signals for the least significant groups pg_0 and pg_1 are combined first, then that result is combined with the next group, etc., in a linear fashion. To combine n groups $n-1$ "fco" operations are performed sequentially.

$$((pg_0 \text{ fco } pg_1) \text{ fco } (pg_2 \text{ fco } pg_3)). \tag{4}$$

Equation (4) indicates that the two lower and upper groups are combined simultaneously, and the then the two results are combined. With this approach $\log_2 n$ sets of "fco" operations are performed. This process can be extended to larger spans as shown in (5) which describes a carry lookahead adder with three bit groups.

$$((pg_0 \text{ fco } pg_1 \text{ fco } pg_2) \text{ fco } (pg_0 \text{ fco } pg_1 \text{ fco } pg_2) \text{ fco } (pg_0 \text{ fco } pg_1 \text{ fco } pg_2)). \tag{5}$$

In the case with a span of k, the number of Manchester carry chain delays is equal to the number of nesting levels (i.e., $\log_k n$) and the order of evaluation in the circuit corresponds to evaluating the lowest level first.

Three candidate adder trees were evaluated: the carry lookahead, carry skip, and binary lookahead carry adder trees. Sixteen bit versions are shown in Fig. 1. As shown the binary lookahead carry adder does not have a carry in. In our implementation the fundamental carry operation, "fco," is performed by a one bit section of the Manchester carry chain. Hence, four adjacent "fco" boxes represent a 4-bit Manchester carry chain. The "Σ" boxes stand for the parity function which combines the carry-in with two operand bits to produce a sum. The connecting lines symbolize connections of the block generate and block propagate signals. The inputs and outputs are not shown.

Carry inputs for the parity blocks are produced by the carry lookahead tree in a two pass process. First, block propagate and generate signals produced on the leaves of the tree are combined into larger groups in succeeding levels as they travel toward the root. Second, block carries are produced from the carry-in and the block generate and propagate signals at the root and travel back toward the leaves, thus causing the Manchester carry chains to re-evaluate and produce all of the intermediate carry outputs.

The multilevel carry skip adder is a propagation time balanced variation of the carry lookahead adder. In this adder the length of the critical delay path is shortened at the expense of noncritical path lengths, for a net gain in adder performance. The version shown in Fig. 1 has a block size distribution of (3553).

The binary lookahead carry tree differs from the carry lookahead tree because the carry-in logic used in back propagation has been removed, and the intermediate carries are produced by a circuit called an inverse tree. The inverse tree spans the carry lookahead tree and more efficiently calculates the intermediate carries in fewer gate levels at the expense of more internal loading.

The 16-bit carry lookahead tree evaluates in ten "fco" delays, the carry skip evaluates in nine, while the binary lookahead carry tree evaluates in six. The 16-bit carry lookahead and the carry skip adders require 20 "fco" operators, while the binary lookahead carry adder requires 26. Sometimes the number of operations is not as important as the number of "fco" levels in the layout [i.e., the number of nesting levels in (5)]. The carry lookahead and carry skip adders require two levels while the binary lookahead carry adder requires three. The largest fanout in the carry lookahead and carry skip adders is one, while in the binary lookahead adder it is four.

The binary lookahead carry adder would be faster and have better internal load distribution if a Manchester carry chain tree with a larger span could be used [13], but no practical method for doing this has been presented in the literature. This paper demonstrates a method for producing an inverse tree for a 56-bit adder. We will show that a version of the inverse tree which calculates carries on linearly separated 8-bit boundaries facilitates the construction of a practical, and fast, high radix lookahead carry/carry select adder.

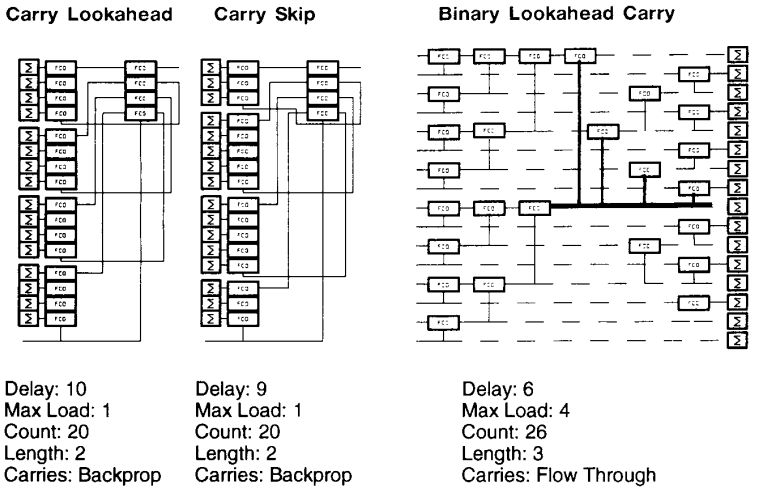


Fig. 1. Candidate carry tree structures for 16-bit operands.

A block diagram of a 64-bit spanning tree carry lookahead adder is shown in Fig. 2. The adder uses a tree of Manchester carry chain carry lookahead modules to calculate the carries for each of the eight bit carry select adders. The first column of Manchester carry chain carry lookahead modules (Mcc) produces group generate and propagate signals for four bit groups, except for the least significant group, where carry c_4 is produced. On the second level of the tree, the first level signals are combined to produce group propagate and generate signals for 16-bit boundaries, and overlapping eight bit boundaries. For example, $p_{47:32}, g_{47:32}$ and the overlapping eight bit boundary signals $p_{39:32}, g_{39:32}$ are produced. Carries c_8 and c_{16} are available from the least significant group on the second level.

On the third level two Manchester carry chain modules are used. The most significant third level module combines the three less significant 16-bit boundary group propagate and generate signals, $p_{15:0}, g_{15:0}$; $p_{31:16}, g_{31:16}$; and $p_{47:32}, g_{47:32}$, and the most significant eight bit boundary group propagate and generate signals $p_{55:48}, g_{55:48}$ to produce carries c_{48} and c_{56} (note that carry c_{32} is generated both here and at the lower module in this column).

The idempotency of the fundamental carry operation must be shown to explain the operation of the least significant third level carry lookahead module. As shown in (2), combining group p and g signals with themselves produces correct results. The result for propagate follows trivially since ANDing any signal, say x, with itself returns x, while the result for generate follows by factoring $g_{i:j} + p_{i:j}g_{i:j} = g_{i:j}(1 + p_{i:j}) = g_{i:j}$. Thus,

$$(p_{i:j}, g_{i:j}) = (p_{i:j}, g_{i:j}) \text{ fco } (p_{i:j}, g_{i:j}). \tag{6}$$

In Fig. 3, a graphic notation based on intervals over an integer number line is used to show that the "fco" can be applied to overlapping group propagate and generate signals. The number line is labeled with bit positions, so a group propagate and group generate signal for bits i through j is represented by an interval extending from i to j. In this notation, the "fco" operation combines two adjacent intervals into one large interval as shown in the first line of Fig. 3. Conversely, a large interval is equivalent to the combination of two smaller intervals. Three intervals can be combined by adjoining the two intervals on the left, and then the one remaining on the right, or vice versa; this follows from the associativity property of the "fco." Two overlapping intervals can be combined with the "fco" into one large interval by first breaking out the overlapping parts of the two intervals, the overlapping parts may be combined freely according to the idempotency property, the remaining three adjacent intervals can be combined with the "fco" into one large interval.

Theorem: Given $i > m \geq k > j$, then $(p_{i:j}, g_{i:j})$ can be derived from $(p_{i:k}, g_{i:k})$ fco $(p_{m:j}, g_{m:j})$.

Proof: The proof begins by applying (1) to expand both terms in $(p_{i:k}, g_{i:k})$ fco $(p_{m:j}, g_{m:j})$:

$$(p_{i:k}, g_{i:k}) = (p_{i:m+1}, g_{i:m+1}) \text{ fco } (p_{m:k}, g_{m:k}) \tag{7}$$

and

$$(p_{m:j}, g_{m:j}) = (p_{m:k}, g_{m:k}) \text{ fco } (p_{k-1:j}, g_{k-1:j}) \tag{8}$$

Substituting these for $(p_{i:k}, g_{i:k})$ and $(p_{m:j}, g_{m:j})$ in $(p_{i:k}, g_{i:k})$ fco $(p_{m:j}, g_{m:j})$ yields

$$(p_{i:k},g_{i:k}) \text{ fco } (p_{m:j}, g_{m:j}) = [(p_{i:m+1}, g_{i:m+1}) \text{ fco } (p_{m:k}, g_{m:k})] \text{ fco } [(p_{m:k}, g_{m:k}) \text{ fco } (p_{k-1:j}, g_{k-1:j})].$$

Associativity from (2) is applied to move the brackets:

$$(p_{i:k},g_{i:k}) \text{ fco } (p_{m:j}, g_{m:j}) = [(p_{i:m+1}, g_{i:m+1}) \text{ fco } [(\mathrm{p_{m:k}}, \mathrm{g_{m:k}}) \text{ fco } (p_{m:k}, g_{m:k})]] \text{ fco } (p_{k-1:j}, g_{k-1:j}).$$

By the idempotency of the "fco" operation from (6) $[(p_{m:k}, g_{m:k}) \text{ fco } (p_{m:k}, g_{m:k})]$ reduces to $(p_{m:k}, g_{m:k})$:

$$(p_{i:k}, g_{i:k}) \text{ fco } (p_{m:j}, g_{m:j}) = [(p_{i:m+1}, g_{i:m+1}) \text{ fco } (p_{m:k}, g_{m:k})] \text{ fco } (p_{k-1:j}, g_{k-1:j}).$$

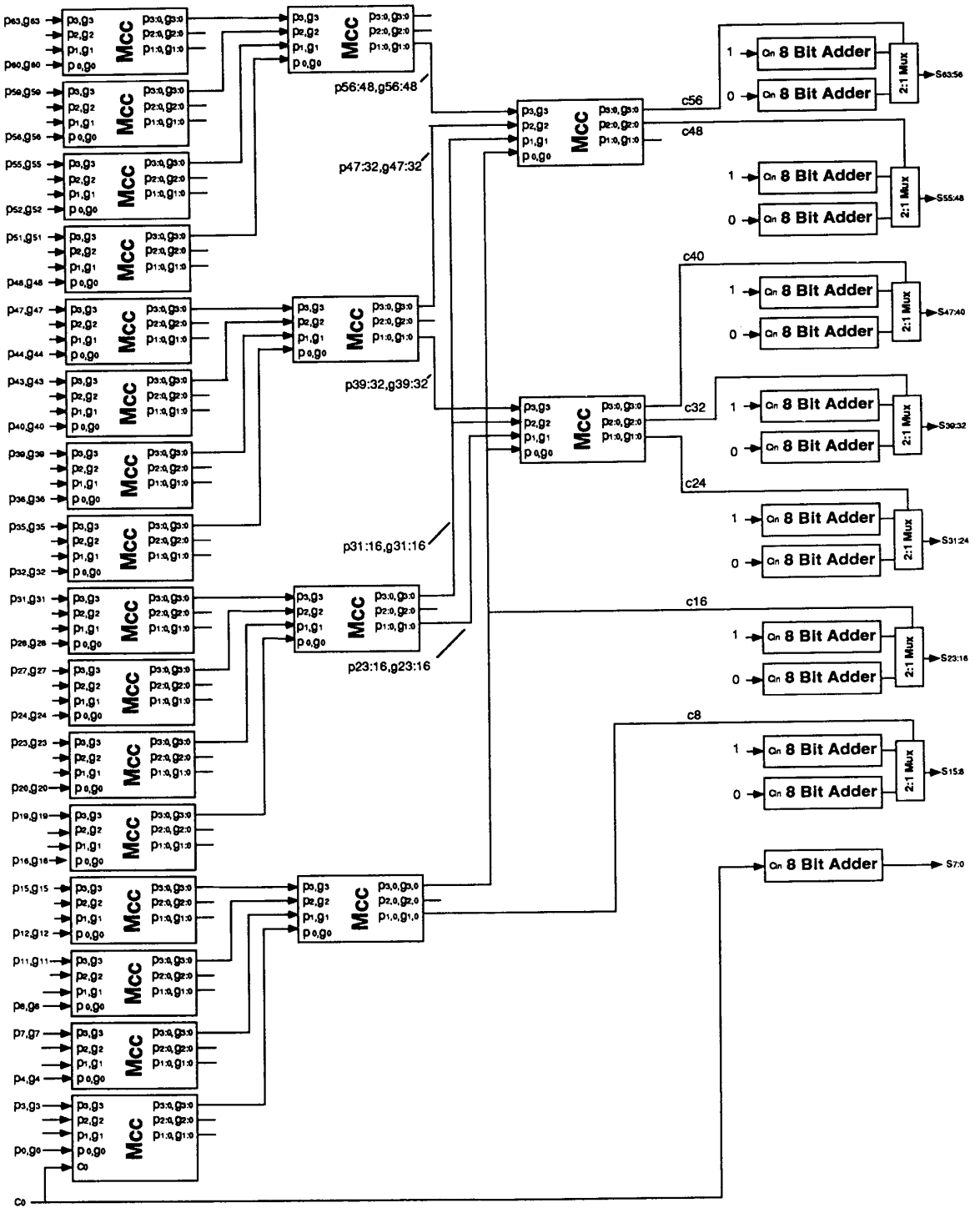


Fig. 2. 64-bit spanning tree carry lookahead adder block diagram.

Applying (1) to the bracketed quantity:

$$(p_{i:k}, g_{i:k}) \text{ fco } (p_{m:j}, g_{m:j}) = (p_{i:k}, g_{i:k}) \text{ fco } (p_{k-1:j}, g_{k-1:j}).$$

Applying (1) again:

$$(p_{i:k}, g_{i:k}) \text{ fco } (p_{m:j}, g_{m:j}) = (p_{i:j}, g_{i:j}). \qquad \text{Q.E.D.}$$

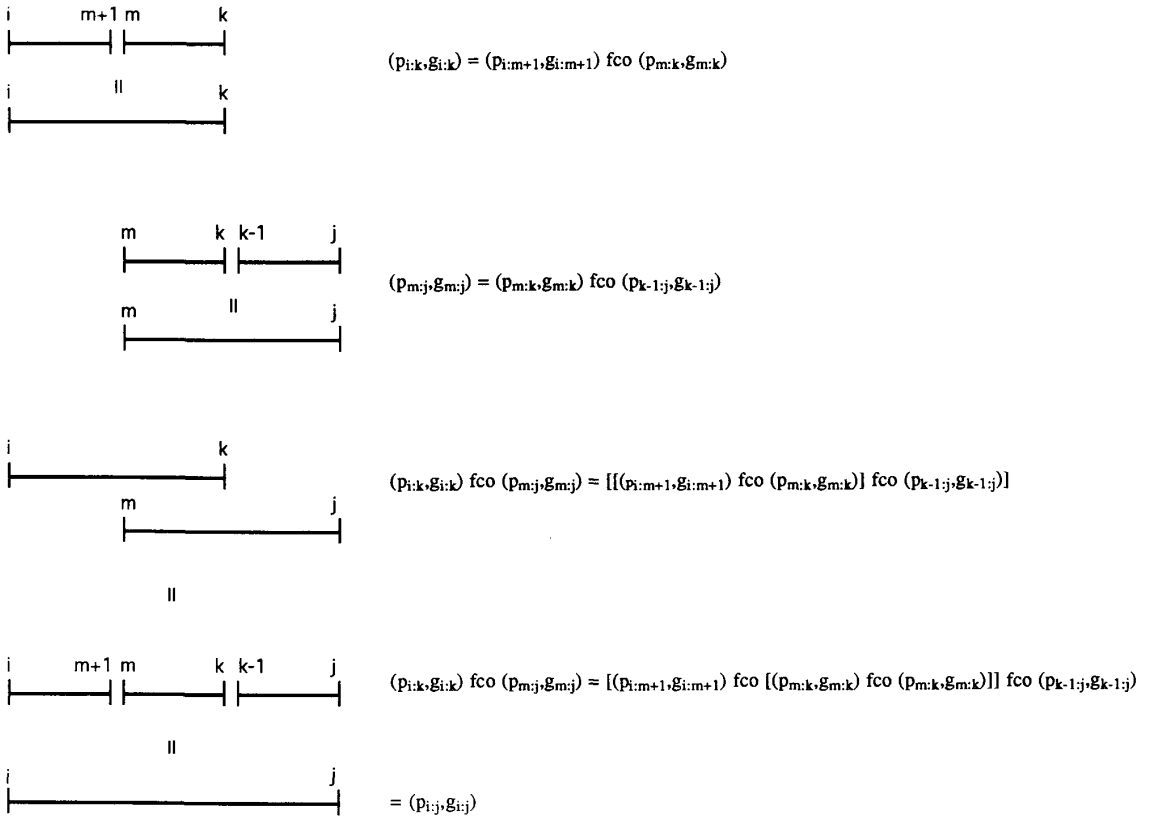


Fig. 3. Illustration of the overlapped carry proof.

Since overlapping regions can be combined with the fundamental carry operation, the overlapping group propagate and generate signals $p_{31:16}, g_{31:16}$; $p_{23:16}, g_{23:16}$; and c_{16} can be applied to the least significant carry lookahead module on the third level to produce carries for c_{24}, c_{32}, and c_{40}. Since carries on all eight bit boundaries are now known, these are used in standard carry select fashion to select the correct result from the eight bit adders.

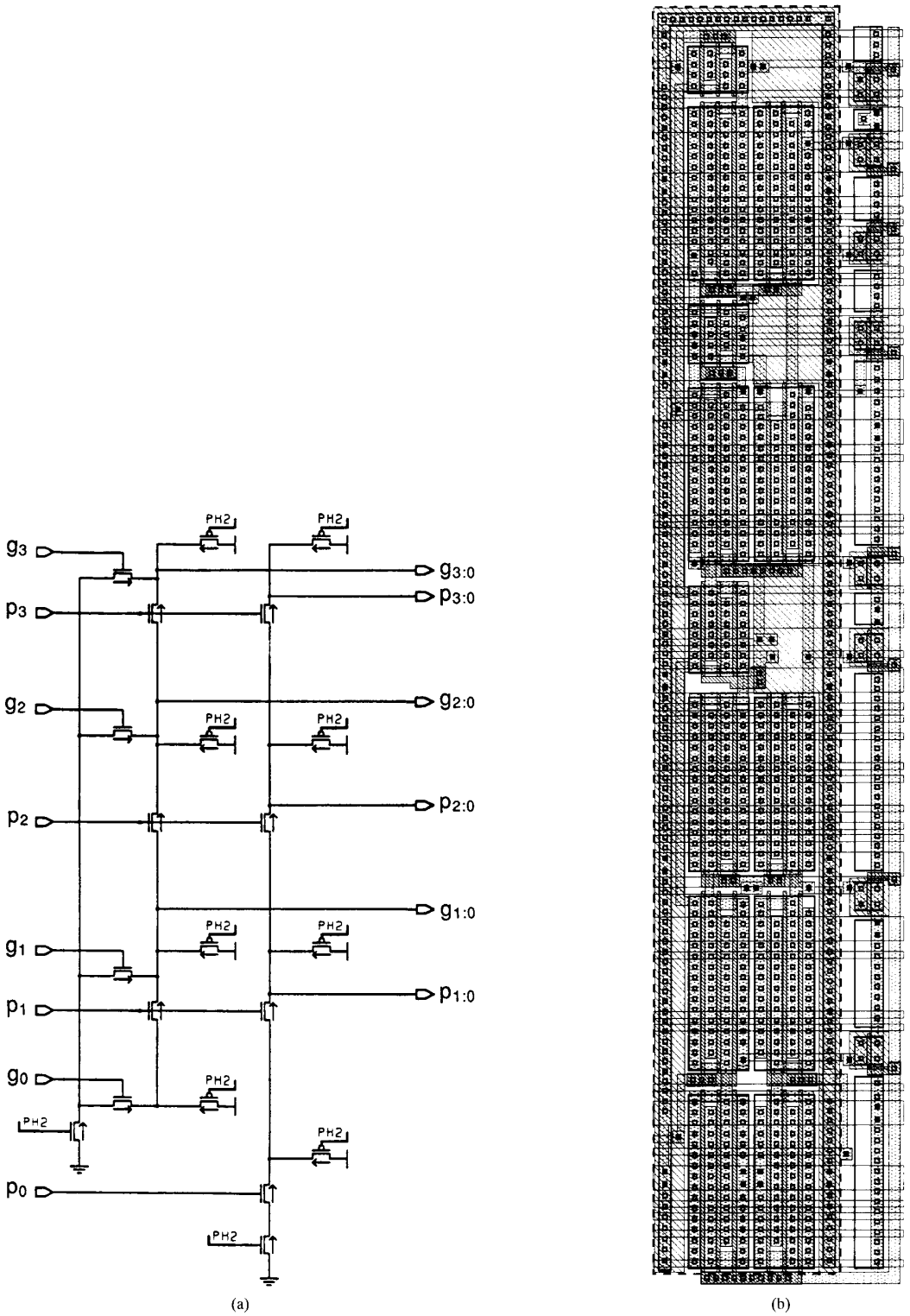
III. Implementation of the Spanning Tree Carry Lookahead Adder

The basic Manchester carry chain cell is shown in Fig. 4(a). Outputs are tapped from the appropriate points in the chain. In order to make the Manchester carry chain as fast as possible, each series transistor is sized to approximately fill the bit cell where it is placed. The tree is laid out by placing the Mcc modules in roughly the same position as they are shown in the block diagram of Fig. 2. The third level (and higher level if necessary) Mcc modules are placed in holes left in the column of second level Mcc modules to reduce the width of the layout.

Some variations of the Manchester carry chain cell are possible since not all of the outputs were used in all of the cells. In the basic Manchester carry chain shown on Fig. 4(a) [layout shown on Fig. 4(b)] the string of series transistors used to calculate the block propagate differs from the string of transistors used for propagating the generate signals only in the connection of P_0. The block propagate chain is removed in the reduced Manchester carry chain shown on Fig. 4(c) and replaced by an extra transistor extending off the bottom. The group generate is now discharged through the propagate chain. This saves four transistors and reduces the internal loading in the carry tree, but eliminates the intermediate outputs. The carry in variation shown on Fig. 4(d) contains two extra transistors for supporting a carry in. This carry in variation is used once at the bottom of the tree.

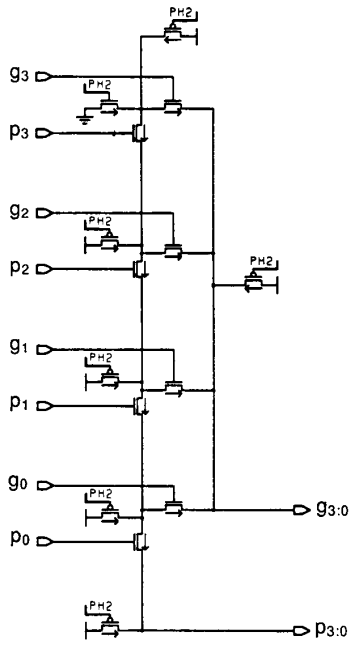
The adder floor plan is shown in Fig. 5. Metal two is used for long horizontal runs carrying the input operands, the bit propagate and generate signals, the calculated carries, and the results. Metal one runs vertically, and is used for local interconnect. The LSB of the adder is at the bottom. The inputs come from the left, while the outputs leave at the right. The first block in the floor plan (going from left to right) is the propagate and generate logic. The second block is a stack of four bit Mcc modules connected in pairs to form eight bit ripple carry sections. A ZERO carry is input to each section in this column.

The third column consists of ripple carry sections with a carry in of ONE. Next are the exclusive-OR gates for calculating the sums from the ripple carries and the bit propagates. The fifth column is the carry tree. The last column

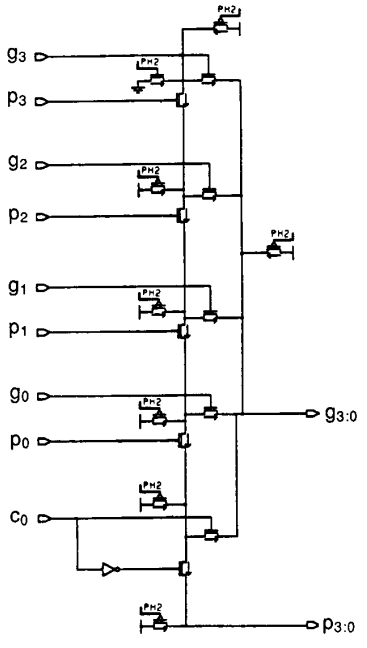


(a) (b)

Fig. 4. (a) Basic Manchester carry chain circuit. (b) Layout of basic Manchester carry chain circuit.



(c)



(d)

Fig. 4. *(Continued)* (c) Manchester carry chain circuit without intermediate outputs. (d) Manchester carry chain circuit with carry input.

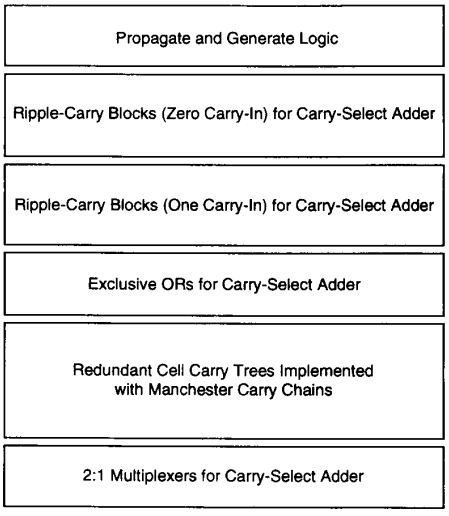


Fig. 5. Spanning tree carry lookahead adder floor plan.

consists of multiplexers which select between the eight bit add results.

For the IEEE 754 floating point standard, double precision significand calculation, only 56 bits are required, so carry c_{56} is the carry out, and the top carry select adder (bits 56 through 63) and its multiplexer are omitted. Also, the associated carry tree logic may be omitted, which includes the top four Manchester carry chain blocks on the first level and the top Manchester carry chain block on the second level.

IV. Performance

The width of the adder is roughly proportional to $\log n$. The second level of the adder has holes large enough to contain three Manchester carry chains, so the third (and fourth if required) levels of the tree can be packed into the second level for adders of up to 256 bits. The total width is the sum of the widths of the blocks in the floor plan:

$$W = W_{pg} + (1 + \log_{16} N)W_{\mathrm{Mcc}} + W_{\mathrm{csa8}}. \tag{9}$$

Where: W is the total width of the N bit adder, W_{pg} is the width of the propagate/generate logic, W_{Mcc} is the width of the Manchester carry chain logic, and W_{csa8} is the width of the eight bit carry select adder. W is 450 μm wide (for $24 \leq N < 256$), as implemented in the Advanced Micro Device 1 μm CMOS technology. The total area is the product of the width times the height:

$$A = WNH_c. \tag{10}$$

Where: H_c is the height of a bit slice adder cell. Since each bit cell is 71.6 μm high, the 56-bit adder shown on Fig. 6 is 4010 μm high for a total area of 1.8×10^6 square μm. The height and area of adders for word sizes from 24 to 256 bits scales in direct proportion to the word size.

As with any carry select adder, the delay is determined by the slower of the sum computation and the carry computation. In this implementation, the sum computation is faster. For the carry computation, propagate and generate signals are already set up when the clock asserts. First, the least significant Manchester carry chain in the first level fires. The critical delay path signal travels through three levels of Manchester carry

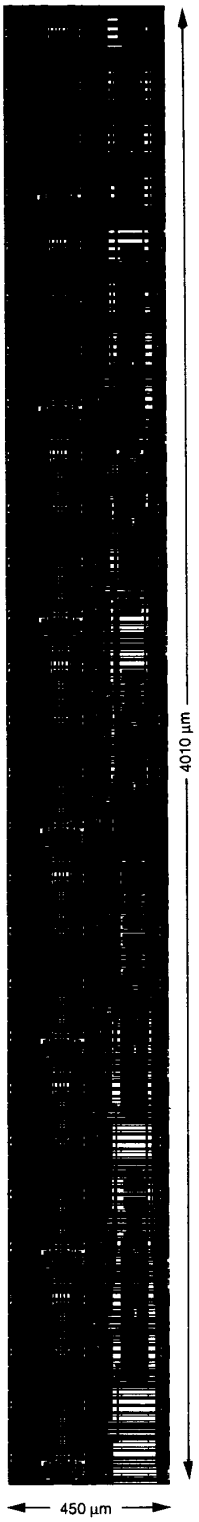


Fig. 6. Spanning tree carry lookahead adder layout.

chains. The worst case load is seen by c_{16} since it drives two Manchester carry chain inputs; the multiplexers are well buffered, so they present a small load. Finally the critical carry signal arrives at the multiplexer for the most significant eight bit adder.

$$D = (\log_4 N)D_{\mathrm{Mcc}} + D_{\mathrm{mux}}. \tag{11}$$

Where: D is the delay of the N bit adder, D_{Mcc} is the delay of the Manchester carry chain, and D_{mux} is the delay of a 2:1 multiplexer. In the Am29050 microprocessor, the time from the rising edge of the clock to the sum is approximately 3.2 ns as shown in the electron beam timing waveforms on Fig. 7. With comparable CMOS technologies the speed should be nearly constant for $24 \leq N < 64$.

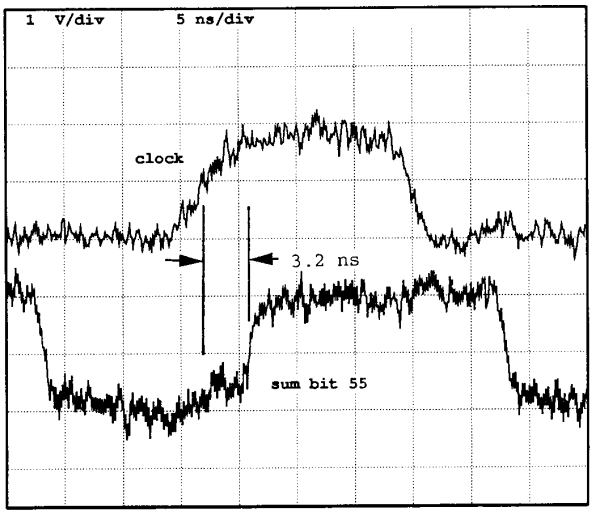


Fig. 7. Measured performance of the spanning tree carry lookahead adder. The top trace is the clock and the lower trace is the most significant sum bit.

V. Conclusion

Proof of the idempotency of the fundamental carry operation allows its use in developing higher radix versions of the binary lookahead carry adder. The higher radix lookahead tree exhibits small and relatively uniform internal loading which is important in CMOS implementations. This paper has described a combination high radix lookahead carry/carry select adder. The size and delay both grow in proportion to the logarithm of the wordsize. As implemented in 1 μm CMOS, this adder achieves a 3.2 ns measured add time for 56-bit operands and is of reasonable size, as shown by the Am29050 microprocessor implementation.

Acknowledgment

This paper is dedicated to the memory of Stephen McIntyre who provided assistance in the development of the adder. His untimely death is a loss to the technical community and his friends. Thanks are due to D. English and T. Burghart for their help in implementing this adder and to the various people who have commented on the preliminary version of this paper [14].

REFERENCES

[1] A. Weinberger and J. L. Smith, "A logic for high-speed addition," *National Bureau of Standards Circular* 591, pp. 3–12, 1958.

[2] M. Lehman and N. Burla, "Skip techniques for high-speed carry propagation in binary arithmetic units," *IRE Trans. Electron. Comput.*, vol. EC-10, pp. 691–698, 1961.

[3] O. J. Bedrij, "Carry-select adder," *IRE Trans. Electron. Comput.*, vol. EC-11, pp. 340–346, 1962.

[4] J. Sklansky, "Conditional-sum addition logic," *IRE Trans. Electron. Comput.*, vol. EC-9, pp. 226–231, 1960.

[5] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *IEEE Trans. Comput.*, vol. C-31, pp. 260–264, 1982.

[6] H. Ling, "High-speed binary adder," *IBM J. Res. Develop.*, vol. 25, pp. 156–166, May 1981.

[7] I. S. Hwang and A. L. Fisher, "A 3.2 ns 32-bit CMOS adder in multiple output domino logic," in *1988 IEEE Int. Solid-State Circuits Conf. Dig. Tech. Papers*, pp. 140, 141, 332, and 333.

[8] G. Bewick *et al.*, "Approaching a nanosecond; A 32-bit adder," in *Proc. 1988 IEEE Int. Conf. Comput. Design: VLSI in Computers and Processors*, pp. 221–226.

[9] T. Kilburn, D. B. G. Edwards, and D. Aspinall, "Parallel addition in digital computers: A new fast 'carry' circuit," *IEE Proc.*, vol. 106, pt. B, pp. 464–466, 1959.

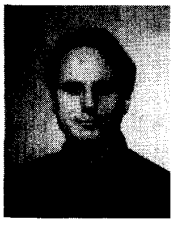
[10] J. J. Cavanaugh, *Digital Computer Arithmetic: Design and Implementation*. New York: McGraw-Hill, 1984, pp. 112–122.

[11] P. K. Chan and M. D. F. Schlag, "Analysis and design of CMOS Manchester adders with variable carry skip," *IEEE Trans Comput.*, vol. 39, pp. 983–992, 1990.

[12] P. K. Chan *et al.*, "Delay optimization of carry-skip adders and block carry-lookahead adders," in *Proc. 10th Symp. Comput. Arithmetic*, 1991, pp. 154–164.

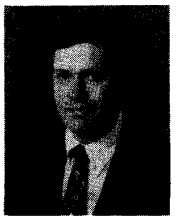
[13] T. Han and D. A. Carlson, "Fast area-efficient VLSI adders," in *Proc. 8th Symp. Comput. Arithmetic*, 1987, pp. 49–56.

[14] T. Lynch and E. E. Swartzlander, Jr., "The redundant cell adder," in *Proc. 10th Symp. Comput. Arithmetic*, 1991, pp. 165–170.



Thomas Lynch received the Bachelor of Science degree in electrical engineering from the University of Texas at Austin in 1986 and is currently studying for the Master of Science degree.

He designed robotics equipment and then VLSI circuits at Advanced Micro Devices, Austin, TX. He has four patents in computer arithmetic. He currently consults in computer arithmetic. His research interests include the practical and theoretical aspects of computation.



Earl E. Swartzlander, Jr. (S'64–M'72–SM'79–F'88) received the B.S. degree in electrical engineering from Purdue University in 1967, the M.S. degree in electrical engineering from the University of Colorado in 1969, and the Ph.D. degree from the University of Southern California in 1972.

In 1990 he became a Professor of Electrical and Computer Engineering at the University of Texas at Austin, where he holds the Schlumberger Centennial Chair in Engineering. Previously he was with TRW for 15 years, where (among other assignments) he managed the Independent Research and Development program for TRW Defense Systems Group from 1987 to 1990 and managed the Digital Processing Laboratory in TRW Electronic Systems Group from 1985 to 1987. His research interests are in application specific computing and the interaction between computer architecture and technology. He received the doctorate in computer design with the support of a Howard Hughes Doctoral Fellowship. He has written the book *VLSI Signal Processing Systems* (Norwell, MA: Kluwer, 1986) and edited five books including two collections of reprints on Computer Arithmetic (Los Alamitos, CA: IEEE Computer Society Press, 1990). He has written or co-written approximately 100 papers in the fields of computer arithmetic, signal processing, and VLSI implementation.

Dr. Swartzlander is a member of the IEEE Signal Processing Society ADCOM, a former member of the IEEE Computer Society Board of Governors (1987–1991), Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTERS, the hardware area editor for *ACM Computing Reviews*, and the founding Editor-in-Chief of the *Journal of VLSI Signal Processing*. He was an Editor of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS (1989–1990), an Editor of IEEE TRANSACTIONS ON COMPUTERS (1982–1986), and an Associate Editor of the IEEE JOURNAL OF SOLID-STATE CIRCUITS (1984–1988). He was the General Chair of the first three IEEE Real Time Systems Symposia (1980–1982), General Chair of the 5th IEEE International Conference on Distributed Computing Systems (1985), General Chair of the first IEEE International Conference on Wafer Scale Integration (1989), and Co-General Chair of the Applications Specific Array Processors Conference (1990). He was Co-Program Chair of the 9th Symposium on Computer Arithmetic and is the General Chair of the 11th Symposium on Computer Arithmetic which will be held in Windsor, Canada, in July 1993. He belongs to Eta Kappa Nu, Sigma Tau, and Omicron Delta Kappa honorary fraternities, and is a Registered Professional Engineer in Alabama, California, Colorado, and Texas. In 1989 he was named a Distinguished Engineering Alumnus of Purdue University and won an award for innovation from the TRW Defense Systems Group.