Implementation of 32-bit Ling and Jackson Adders

Matthew Keeter, David Money Harris, Andrew Macrae, Rebecca Glick, Madeleine Ong, and Justin Schauer*
Harvey Mudd College
301 Platt Blvd. Claremont, CA 91711
{Matthew_Keeter, David_Harris}@hmc.edu
* Oracle Redwood Shores, CA

***Abstract*— Ling adders factor complexity out of the first stage of an adder to shorten the critical path. In 2004, Jackson and Talwar proposed a generalization of the Ling adder that reduces the complexity of the critical generate path at the expense of increased complexity in the propagate logic. This paper compares implementations of 32-bit Ling and Jackson adders to the optimized Sklansky architecture produced by Design Compiler in a 45 nm process. The Ling adder is 3% faster and uses 7% less energy, achieving a delay of 8.3 FO4 inverters. The Jackson adder is only 1% faster and uses 45% more energy. However, this is the first published implementation of a Jackson adder with all details shown.**

I. Introduction

Binary addition is a heavily-studied field because adders are often in the critical paths of computing systems [1]. High-performance adders of 16 bits and larger typically use a prefix tree to compute group generate and propagate signals before computing the sums from the generate prefixes.

The industry-standard Design Compiler logic synthesis tool from Synopsys generates highly optimized prefix adders with a delay of about 9 FO4 inverters for a 32-bit adder.

Ling described a way to factor out some complexity from the initial stage of group generate logic, saving one transistor from the critical path [2]. This comes at the cost of more complex logic on the non-critical path to precompute inputs to a sum-selection multiplexer. Several teams have explored Ling adder implementations using standard cell libraries. Dimitrakopoulos and Nikolos and Lakshmanan and Othman report better results than conventional adders, but they either don't compare directly with the output of Design Compiler or they don't report a normalized FO4 inverter delay [3, 4]. Zhu et al. describe a linear programming technique for exploring the design space, but report delays of 10 FO4 inverters for a 16-bit adder [5].

Jackson and Talwar proposed a generalization to the Ling technique that factors out complexity from subsequent group generate stages as well [6]. The complexity is transferred into the propagate logic as well as into the precomputed sums. The technique is applicable to valency-3 or higher cells. Burgess offers notes on implementation issues and concluded that Jackson adders could be superior to conventional designs [7]. At least two designers have indicated that they have built Jackson adders exceeding the performance of conventional alternatives, but these results have not been published [8, 9].

This paper investigates the implementation of 32-bit Ling and Jackson adders and compares them to conventional adders produced by Design Compiler. We demonstrate small performance gains over Design Compiler at the expense of considerable energy. Section II defines our terminology for conventional prefix addition and shows how the Ling technique or higher-valency cells can be used to reduce the number of logic levels. Section III defines the Jackson recursion. Section IV describes the 45 nm process and standard cell library used in the comparison. Section V presents the Ling and Jackson implementations. The results are summarized in Section VI.

II. Background

Consider adding two N-bit numbers $A = \{a_N, a_{N-1}, \ldots, a_1\}$ and $B = \{b_N, b_{N-1}, \ldots, b_1\}$, along with a carry-in C_{in} to produce an N-bit sum S, discarding any possible overflow. A conventional prefix adder first defines signals for each bit defining whether the bit would generate a carry out (g) or propagate a carry. The propagate term can be computed using either an XOR (x) or OR gate (p); it is often handy to use both because the OR is simpler and results in a faster prefix tree, while the XOR is needed in the final sum logic:

$$g_i = a_i b_i \quad (1)$$

$$p_i = a_i + b_i \quad (2)$$

$$x_i = a_i \oplus b_i \quad (3)$$

The carry-in is handled with the special case that $g_0 = C_{in}$.

Group generate (G) and propagate (P) signals indicating whether a group spanning bits i through j generate or propagate a carry are then obtained with a valency-2 recursion ($i > k > j$):

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j} \quad (4)$$

$$P_{i:j} = P_{i:k} P_{k-1:j} \quad (5)$$

The sums are then determined from the generate prefixes as

$$s_i = x_i \oplus G_{i-1:0} \tag{6}$$

Many alternatives exist for combining the group generate and propagate signals. Design Compiler generates the modified Sklansky architecture [10] (sometimes called Ladner-Fischer [11]) because it has a minimal number of logic levels and no redundant logic, providing high speed and low power. Fig. 1 shows a 16-bit modified Sklansky adder similar to the one produced by Design Compiler. The top row contains the logic to compute *g*, *p*, and *x* for each bit. The middle of the tree contains the group logic. Note that black cells compute both *G* and *P*, while gray cells compute only *G* for cases where *P* is unnecessary. The bottom row computes the sums with a final XOR. The design is called modified Sklansky because some of the non-critical *P* and *G* signals (such as $G_{4:0}$, $G_{5:0}$, and $G_{6:0}$) are buffered to reduce the fanout on the critical path.

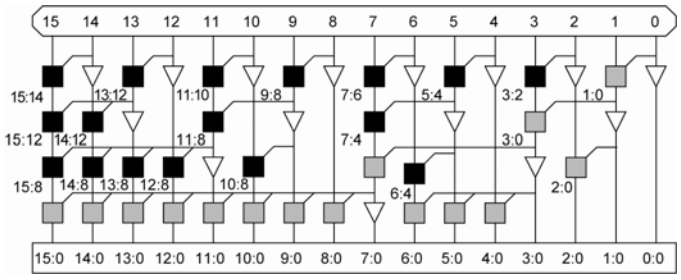

Fig. 1: 16-bit modified Sklansky tree

One logic level could be removed by directly computing pairwise group generates and propagates $G_{i+1:i}$ and $P_{i+1:i}$ from the primary inputs *a* and *b*. However, the generate logic is overly complex, requiring three series transistors:

$$\begin{aligned} G_{i+i:i} &= g_{i+1} + p_{i+1} g_i \\ &= a_{i+1} b_{i+1} + \left(a_{i+1} + b_{i+1}\right) a_i b_i \end{aligned} \tag{7}$$

Ling proposes defining a *pseudogenerate* signal, *H*, such that

$$H_{i:j} = g_i + G_{i-1:j} \tag{8}$$

This is simpler than the conventional generate because it strips out one propagate term:

$$G_{i:j} = g_i + p_i G_{i-1:j} \tag{9}$$

G can be recreated from *H* with an AND gate:

$$G_{i:j} = p_i H_{i:j} \tag{10}$$

Now the pairwise group pseudogenerate logic is simpler and requires only two series transistors

$$\begin{aligned} H_{i+1:i} &= g_{i+1} + g_i \\ &= a_{i+1} b_{i+1} + a_i b_i \end{aligned} \tag{11}$$

Ling also defines a *psuedopropagate* signal, *I*, that is a shifted version of the conventional propagate:

$$I_{i:j} = P_{i-1:j-1} \tag{12}$$

Now, the valency-2 recursion can be expressed using exactly the same logic as with *G* and *P* in EQs (4-5), and can be computed using exactly the same prefix tree:

$$H_{i:j} = H_{i:k} + I_{i:k} H_{k-1:j} \tag{13}$$

$$I_{i:j} = I_{i:k} I_{k-1:j} \tag{14}$$

The sums are computed as

$$\begin{aligned} s_i &= x_i \oplus G_{i-1:0} \\ &= x_i \oplus p_{i-1} H_{i-1:0} \end{aligned} \tag{15}$$

Because the group pseudogenerates are the critical signal, the sum logic can be refactored to use *H* to select a sum based on precomputed options using a multiplexer, thus shifting logic off the critical *H* path:

$$s_i = H_{i-1:0} \,?\left[x_i \oplus p_{i-1}\right] : x_i \tag{16}$$

Sparse trees seek to save energy and area by computing the prefixes for every *m*th bit. Meanwhile, they perform short m-bit ripples to precompute the results for each *m*-bit block assuming the prefix is 0 and 1. Ling adders naturally benefit from a sparseness of 2 by computing prefixes only in the odd-numbered columns (e.g. $H_{1:0}$, $H_{3:0}$, $H_{5:0}$, …, $H_{31:0}$). The sum selection logic now contains a pair of multiplexers as shown in Fig. 2. Note that the *a* and *b* inputs may be buffered before computing *x*, *p*, and *g* to reduce the load on the critical path.

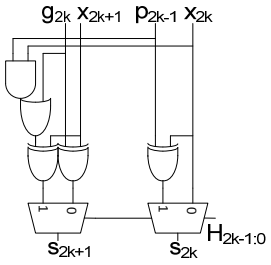

Fig. 2: Ling sum selection for sparseness-2

Yet another option for reducing the number of logic levels in an adder is to combine more than two groups at a time in the prefix tree. For example, the valency-3 recursion ($i > k > l > j$) gives:

$$G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:l} + P_{i:k} P_{k-1:l} G_{l-1:j} \tag{17}$$

$$P_{i:j} = P_{i:k} P_{k-1:l} P_{l-1:j} \tag{18}$$

The complexity of these terms is high enough that higher valency adders tend not to be beneficial in static CMOS circuits. Domino gates are more amenable to complex stacks; Hewlett-Packard built very fast domino adders using a valency-4 Ling design [12, 13], but domino has been phased out due to its high power consumption.

III. The Jackson Recursion

Jackson and Talwar generalized the Ling technique to reduce the complexity of the entire prefix tree, rather than just the first stage. The simplification makes higher-valency cells more attractive in static logic. Observe that the valency-3 generate logic is significantly more complex than the propagate logic in EQs (17-18). Jackson defines reduced generate, *R*, and hyperpropagate, *Q*, signals that balance these complexities and simplify the worst case. First, we must introduce two intermediate signals, *D* and *B*.

$D_{i:j}$ indicates that the group spanning bits i through j either generate or propagate a carry:

$$D_{i:j} = G_{i:j} + P_{i:j} \tag{19}$$

Because $P_{i:j}$ covers the case of generating in bit j and propagating through the rest, the logic can be simplified to

$$D_{i:j} = G_{i:j+1} + P_{i:j} \tag{20}$$

and in the common special case of one-bit blocks,

$$D_{i:i} = p_i \tag{21}$$

$B_{i:j}$ indicates that the group generates a carry in at least one bit:

$$B_{i:j} = \sum_{k=i}^{j} g_k \tag{22}$$

Now, the group generate signal can be rewritten as

$$G_{i:j} = D_{i:k}\left[B_{i:k} + G_{k-1:j}\right] \tag{23}$$

In other words, the group generates a carry if the upper part either generates or propagates and either at least one bit of the upper part generates (indicating that the upper as a whole generates), or the lower part generates.

The bracketed term is called the *reduced generate signal*, R:

$$R^p_{i:j} = B_{i:i-p+1} + G_{i-p:j} \tag{24}$$

It can be viewed as G with the top p propagate signals stripped out. The special cases of $p = 0$ and $p = 1$ correspond to the conventional generate and the Ling pseudogenerate signals, while Jackson adders permit further reductions of $p \geq 2$.

$$R^0_{i:j} = G_{i:j} \tag{25}$$

$$R^1_{i:j} = H_{i:j} \tag{26}$$

The ordinary group generate is recovered from the reduced generate using D; greater reduction of R requires a larger D term. EQ (23) can be rewritten using R as

$$G_{i:j} = D_{i:i-p+1}R^p_{i:j} \tag{27}$$

Jackson also defines a *hyperpropagate* signal, Q, to complete the recursion:

$$Q^p_{i:j} = P_{i:i-p+1}D_{i-p:j} \tag{28}$$

The frequently used special cases of $p = 1$ or 2 for two-bit groups correspond to the ordinary propagate signal, while $p > 2$ produces more complex logic.

$$Q^1_{i+1:i} = Q^2_{i+1:i} = P_{i+1:i} \tag{29}$$

Jackson derives a valency-3 recursion using R and Q:

$$R^{i+1-p}_{i:j} = R^{i+1-k}_{i:k} + R^{k-p}_{k-1:l} + Q^{p-l}_{p-1:m}R^{l-m}_{l-1:j} \tag{30}$$

$$Q^{i+1-p}_{i:j} = Q^{i+1-k}_{i:k}Q^{k-p}_{k-1:l}\left[R^{p-l}_{p-1:m} + Q^{l-m}_{l-1:j}\right] \tag{31}$$

As compared to EQs (17-18), these terms are better balanced, and the larger of the two terms is less complicated.

Jackson omits the valency-2 recursion needed for some of the intermediate terms in a prefix adder, but it is similar and can be proved by expanding both sides in terms of B, G, P, and D. Unfortunately, the recursion is no simpler than the ordinary valency-2 PG logic from EQs (4-5).

$$R^p_{i:j} = R^p_{i:k} + Q^{i-p-k+1}_{i-p:k-q}R^q_{k-1:j} \tag{32}$$

$$Q^p_{i:j} = Q^p_{i:k}\left(R^{i-p-k+1}_{i-p:k-q} + Q^q_{k-1:j}\right) \tag{33}$$

However, Burgess described a simpler valency-2 recursion that is sometimes helpful [7].

$$R^{i-k+q+1}_{i:j} = R^p_{i:k} + R^q_{k-1:j} \tag{34}$$

Jackson also neglects to explicitly define the recursion for D for large groups:

$$D_{i:j} = D_{i:i-p+1}\left[R^p_{i:k} + Q^{i-k-p+1}_{i-p:j}\right] \tag{35}$$

This is proved in the appendix.

Jackson describes computing $G_{31:0}$ using a Ling-style first stage to compute pairs followed by two valency-4 stages, but does not provide the logic for a complete adder. Burgess describes implementing a 24-bit adder using Jackson techniques but only shows some of the details. The next two sections of this paper describe standard cell implementations of a 32-bit static adder using the Jackson R/Q equations in a 45 nm process.

IV. Comparison Methodology

One of the challenges in comparing circuits is to ensure that the novel circuit is fairly compared to the best known implementation of the conventional circuit [14]. For this reason, we use Synopsys Design Compiler and compare against a synthesized version of a behavioral description, `assign y = a + b`. We will refer to this as the *behavioral* adder.

The adders are synthesized onto the ARM standard cell library for a 45 nm partially-depleted SOI process. The library uses regular-V_t transistors and a 12 track cell height (1.68 μm). It contains a rich set of complex gates including the AOI211 and OAI211 gates used in a valency-3 Jackson recursion. Timing is characterized in the SS corner at 0.9 V, 125 C. The fanout-of-4 inverter delay (FO4) is 15.0 ps, corresponding to τ = 3.0 ps [1]. A unit-sized (X1) inverter has an input capacitance of 1.6 fF and switching energy of 0.78 fJ. The inputs are driven by an X4 inverter and the outputs are loaded by the capacitance of X4 inverters.

Design Compiler recommends the `compile_ultra` command for standard high-performance synthesis. We also evaluated using the `-inc` option for incremental resynthesis to improve performance at the expense of area. The results displayed 1-2% variation in timing depending on the timing constraints, so we swept timing constraints and selected the best results for each design. To accurately account for routing, the synthesized netlist was passed to Synopsys IC Compiler for placement, routing, and parasitic estimation. The inputs and outputs are constrained to appear in 32 ordered rows with a pitch of one cell per row. Layout utilization is set to 70%, which produces good timing results.

When synthesizing a 32-bit behavioral adder to be as fast as possible, Design Compiler produces the modified Sklansky adder like that of Fig. 1 extended to 32 bits; we will call this a conventional design. The gray cells are built using alternating

AOI21 and OAI21 cells according to DeMorgan's Law. The least significant bits have minor logical optimizations to handle the carry-in, and various gates are cloned to optimize the critical and non-critical paths. Logical Effort analysis [15] of the critical path predicts a delay of 8.9 FO4.

V. IMPLEMENTATIONS

This section describes two static 32-bit adder implementations including a valency-2 Ling adder and a mixed-valency Jackson adder. All designs were verified for logical correctness using directed and random vectors.

A. *Ling Adder*

Figure 3 shows the organization of a 32-bit Ling adder using a modified Sklansky tree. The first row computes the *H* and *I* signals for each pair of bits. The sparseness-2 prefix tree looks just like the modified Sklansky tree from Fig. 1 except that the groups are twice as wide. The sum is selected for each pair of bits using the circuitry from Fig. 2.

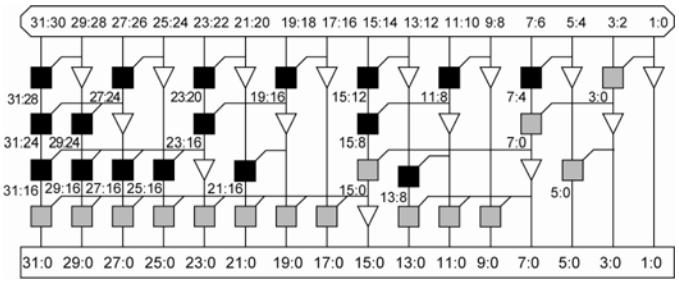

Fig. 3: 32-bit Ling adder using modified Sklansky tree

This design has six logic levels, as compared to seven in the conventional adder. The first level uses compound AOI22 and OAI22 cells to produce the 2-bit *H* and *I* signals. The next levels use alternating OAI21 and AOI21 gates, as in the conventional adder. Each prefix drives a pair of inverting sum-selection multiplexers. If buffered loads present negligible capacitance, the branching effort is 16, just as in a conventional adder. The initial stage is slightly more complex than in a conventional adder but the number of stages is one fewer. Logical Effort calculations predict that the overall delay should be comparable to that of the conventional adder.

B. Jackson 2-3-3-2 Adder

Jackson adders have a huge space of possible architectures and circuit implementations [4]. A straightforward approach is to use a modified sparseness-2 Sklansky tree like that of the Ling adder, but to replace three levels of valency-2 cells with two levels of valency-3 cells to reduce the number of logic levels to 5. Such an architecture could handle up to 2 × 3 × 3 × 2 = 36 bits, but the top four bits are truncated to perform 32-bit addition.

Directly applying the Jackson equations yields the prefix tree shown in Fig. 4. The sum precomputation and *D* logic take place in the ovals above the sum selection multiplexers. The signals on the critical path from a_{14} to s_{31} have a high fanout at each stage, resulting in long delay. Both Logical Effort analysis and synthesis results show such an implementation is unattractive. The critical path reported by IC Compiler is highlighted.

Note that $Q^4_{24:21}$ and $Q^6_{26:21}$ used in the third stage do not exist. Instead, we substitute $Q^1_{24:21}$ and $Q^3_{26:21}$, which have extraneous *G* terms that are later covered by *R* terms [7].

Fig. 5 shows a heavily refactored Jackson adder that optimizes the critical path in a variety of ways. It was arrived at after extensive manual exploration of the design space using Logical Effort and synthesis. More complexity is factored out of the upper *R* terms by exploiting EQ (34), pushing it into larger *D* terms. Noncritical signals are buffered so that they do not load the critical path. Italicized signals are buffered from the first stage but the buffers are not shown to avoid cluttering the diagram. The logic computing $Q^9_{26:9}$, $Q^3_{20:9}$, $Q^1_{18:9}$ and $D_{17:9}$ all share a small NOR gate to combine $R^3_{17:12}$ and $Q^3_{14:9}$ without heavily loading the critical path through $R^9_{17:0}$. The dangling $R^5_{25:18}$ signal is used in the *D* logic.

In parallel, relatively simple gates compute the *x*, *p*, *g*, and *D* signals required for the precomputed sums. These would clutter the diagram, so they are described below in equation form and are implemented with up to four levels of 2-input NANDs and NORs.

D Logic

$$D_{i:i} = p_i \qquad \text{for } i = 1, 3, 7, 9, 19$$

$$R^3_{2i+3:2i} = R^1_{2i+3:2i+2} + R^1_{2i+1:2i} \quad \text{for } i = 6, 10, 11, 12, 13$$

$$D_{5:3} = p_5\left[R^1_{5:4} + Q^1_{4:3}\right]$$

$$D_{11:9} = p_{11}\left[R^1_{11:10} + Q^1_{10:9}\right]$$

$$D_{13:9} = D_{13:11}\left[R^3_{13:12} + Q^1_{12:9}\right] = p_{13}\left[R^1_{13:12} + Q^1_{12:11}\right]\left[R^3_{13:12} + Q^1_{12:9}\right]$$

$$D_{15:9} = D_{15:13}\left[R^3_{15:12} + Q^1_{12:9}\right] = p_{15}\left[R^1_{15:14} + Q^1_{14:13}\right]\left[R^3_{15:12} + Q^1_{12:9}\right]$$

$$D_{17:9} = D_{17:15}\left[R^3_{17:12} + Q^3_{14:9}\right] = p_{17}\left[R^1_{17:16} + Q^1_{16:15}\right]\left[R^3_{17:12} + Q^3_{14:9}\right]$$

$$D_{21:19} = p_{21}\left[R^1_{21:20} + Q^1_{20:19}\right]$$

$$D_{23:19} = D_{23:21}\left[R^3_{23:20} + Q^1_{20:19}\right] = p_{23}\left[R^1_{23:22} + Q^1_{22:21}\right]\left[R^3_{23:20} + Q^1_{20:19}\right]$$

$$D_{25:21} = D_{25:23}\left[R^3_{25:22} + Q^1_{22:21}\right] = p_{25}\left[R^1_{25:24} + Q^1_{24:23}\right]\left[R^3_{25:22} + Q^1_{22:21}\right]$$

$$D_{27:21} = D_{27:25}\left[R^3_{27:24} + Q^1_{24:21}\right] = p_{27}\left[R^1_{27:26} + Q^1_{26:25}\right]\left[R^3_{27:24} + Q^1_{24:21}\right]$$

$$D_{29:25} = D_{29:27}\left[R^3_{29:26} + Q^1_{26:25}\right] = p_{29}\left[R^1_{29:28} + Q^1_{28:27}\right]\left[R^3_{29:26} + Q^1_{26:25}\right]$$

$$D_{31:27} = p_{31}\left[R^1_{31:30} + Q^1_{30:27}\right] = p_{31}\left[R^1_{31:30} + Q^1_{30:29}\left(Q^1_{28:27} + R^1_{29:28}\right)\right]$$

Misc. 1-bit signals for Sum: ($i = 0 \ldots 15$)

$$x_{2i+1} = A_{2i+1} \oplus B_{2i+1} \qquad p_{2i+1} = A_{2i+1} + B_{2i+1}$$

$$x_{2i} = A_{2i} \oplus B_{2i} \qquad g_{2i} = A_{2i} B_{2i}$$

Sum Selection

$$s_1 = x_1 \oplus a_0$$

$$s_{32} = R^5_{31:0} \,?\left(x_{32} \oplus D_{31:27}\right) : x_{32}$$

Middle cells: $i = 1 \ldots 15$

$$s_{2i} = R^p_{2i-1:0} \,?\left(x_{2i} \oplus D_{2i-1:2i-p}\right) : x_{2i}$$

$$s_{2i+1} = R^p_{2i-1:0} \,?\left(x_{2i+1} \oplus \left(D_{2i-1:2i-p} x_{2i} + g_{2i}\right)\right) : \left(x_{2i+1} \oplus g_{2i}\right)$$

Fig. 6 shows the circuit-level implementations of each of the cells. In a valency-2 stage, *l* and *u* refer to the lower and upper groups. In valency-3, *x* refers to the third (most significant) group. The second and third stages are noninverting, each using two levels of logic and thus allowing sharing of common terms. The extra logic levels also help drive the larger fanout on the branching paths. The first stage must be noninverting to produce NANDs rather than NORs on the critical path in the later stages. The fourth stage and final multiplexer are inverting to reduce parasitic delay. Logical Effort predicts a delay of 8.4 FO4 along the critical path identified by IC Compiler.

VI. Results

Table 2 summarizes the post-layout delay, energy, and area results for the architectures considered.

Table 2: Comparison of 32-bit adder results

	Behavioral	**Behavioral**	**Ling**	**Jackson 2332**
Compiler Option	ultra	–inc	–inc	–inc
Delay (ps)	130.2	128.4	124.4	127.6
Energy (fJ)	796	1016	938	1469
Area (μm^2)	606	797	682	1053

The behavioral adder results are quite good and took months of design refinements to beat. The behavioral adder synthesized with Design Compiler Ultra achieved 8.7 FO4 delays including layout parasitic. Applying incremental synthesis achieved a 1.4% speedup at a ~30% increase in energy and area. The Ling adder offers a further 3% speedup and 7% energy reduction relative to the fastest (incremental) behavioral adder. The Jackson adder was also faster, but we were unable to produce results better than Ling. The Ling and Jackson adders with better delay results had to be specified as a structural netlist; Design Compiler was unable to achieve such good results with a more abstract description such as Boolean equations. Moreover, the Jackson adder required initial sizing in the structural netlist to achieve good results.

VII. Conclusions

Design Compiler is the industry standard logic synthesis tool. The quality of the arithmetic circuits it generates is now very high. This paper has investigated applying the Ling and Jackson techniques to produce even faster adders.

This paper has described a 32-bit Ling adder using a sparseness-2 modified Sklansky tree to achieve a post-layout delay of 8.3 FO4 in a 45 nm process. This is 3% faster at 7% less energy than the best that Design Compiler produces from a behavioral description of `y = a + b`.

This paper has presented the first published implementation of a Jackson adder with all details shown. The adder also uses a sparseness-2 modified Sklansky tree with a mix of valency-2 and valency-3 stages to save one level of logic relative to Ling. The synthesis results are 1% faster than the behavioral adder but consume significantly more energy.

Ideas for future work include exploring truncating the bottom rather than top four bits to simplify the critical *R* signal in a Jackson 2-3-3-2 architecture, investigating two-stage valency-4 gates, and determining whether Jackson adders hold greater benefit at 64 bits. The Ling and Jackson adders were carefully designed for delay but not tuned for energy; potential remains to reduce the energy by optimizing noncritical paths.

Acknowledgments

This work was supported by a grant from Oracle and by the Clay-Wolkin Fellowship at Harvey Mudd College.

References

[1] N. Weste and D. Harris, *CMOS VLSI Design*, 4th ed., Boston, MA: Addison Wesley, 2011.

[2] H. Ling, "High-speed binary adder," *IBM J. Research and Dev.*, vol. 25, no. 3, May 1981, pp. 156-166.

[3] G. Dimitrakopoulos and D. Nikolos, "High-speed parallel-prefix VLSI Ling adders," *IEEE Trans. Computers*, vol. 54, no. 2, Feb. 2005, pp. 225-231.

[4] A Lakshmanan and M. Othman, "High-speed hybrid parallel-prefix carry-select adder using Ling's algorithm," *Intl. Conf. Semiconductor Electronics*, 2006, pp. 598-602.

[5] Y. Zhu, J. Liu, H. Zhu, and C.K. Cheng, "Timing-power optimization for mixed-radix Ling adders," *Asia/South Pacific Design Automation Conf.*, 2008, pp. 131-137.

[6] R. Jackson and S. Talwar, "High speed binary addition," *Proc. Asilomar Conf. Signals, Systems, and Computers*, Nov. 2004, pp. 1350-1353.

[7] N. Burgess, "Implementation of recursive Ling adders in CMOS VLSI," *Proc. Asilomar Conf. Signals, Systems, and Computers*, 2009.

[8] R. Jackson, personal communication, 5 July 2010.

[9] E. Mahurin, personal communication, 25 October 2010.

[10] J. Sklansky, "Conditional-sum addition logic," *IRE Trans. Electronic Computers*, vol. EC-9, Jun. 1960, pp. 226-231.

[11] R. Ladner and M. Fischer, "Parallel prefix computation," *J. ACM*, vol. 27, no. 4, Oct. 1980, pp. 831-838.

[12] S. Naffziger, "A subnanosecond 0.5 µm 64b adder design," *Proc. IEEE Intl. Solid-State Circuits Conf.*, 1996, pp. 362-363.

[13] S. Naffziger, "High speed addition using Ling's equations and dynamic CMOS logic," US Patent 5,719,803, 1998.

[14] R. Zimmermann and W. Fichtner, "Low-power logic styles: CMOS versus pass-transistor logic," *IEEE J. Solid-State Circuits*, vol. 32, no. 7, July 1997, pp. 1079-1090.

[15] I. Sutherland, B. Sproull, and D. Harris, *Logical Effort: Designing Fast CMOS Circuits*, San Francisco, CA: Morgan Kaufmann, 1999.

APPENDIX: PROOF OF *D* RECURSION

To prove EQ (35), expand both sides in terms of *G* and *P* and then combine terms and simplify:

$$
\begin{aligned}
D_{i:j} &= D_{i:i-p+1}\left[R^{p}_{i:k} + Q^{i-k-p+1}_{i-p:j}\right] \\
&= G_{i:k} + D_{i:i-p+1}Q^{i-k-p+1}_{i-p:j} \\
&= G_{i:k} + \left(G_{i:i-p+1} + P_{i:i-p+1}\right)\left(P_{i-p:k}\right)\left(G_{k-1:j} + P_{k-1:j}\right) \\
&= G_{i:j} + P_{i:j}
\end{aligned}
$$

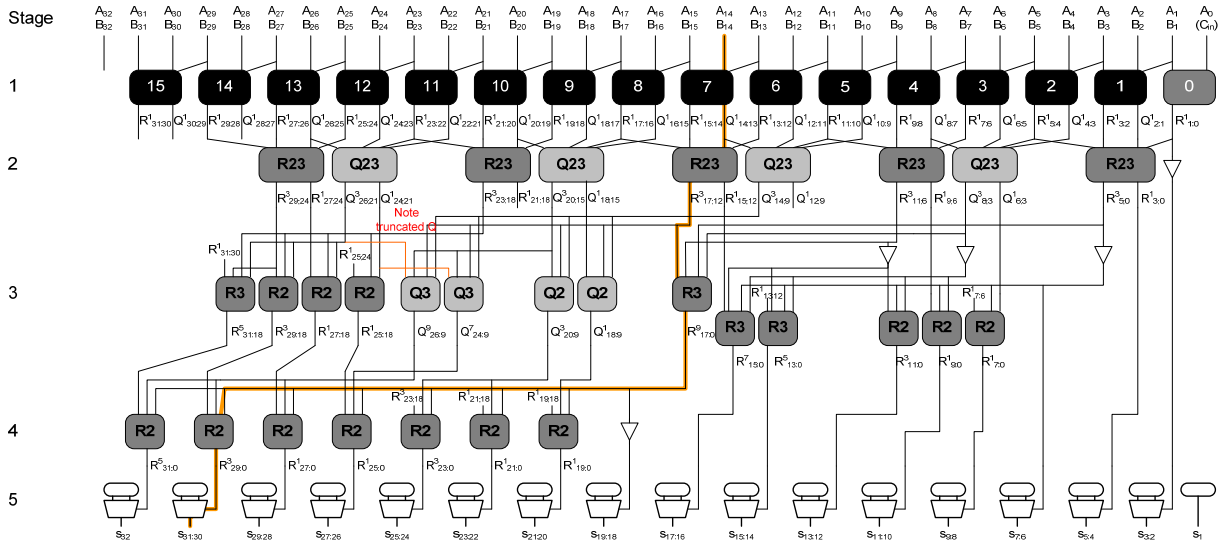

Fig. 4: 32-bit Jackson 2-3-3-2 adder using modified Sklansky tree

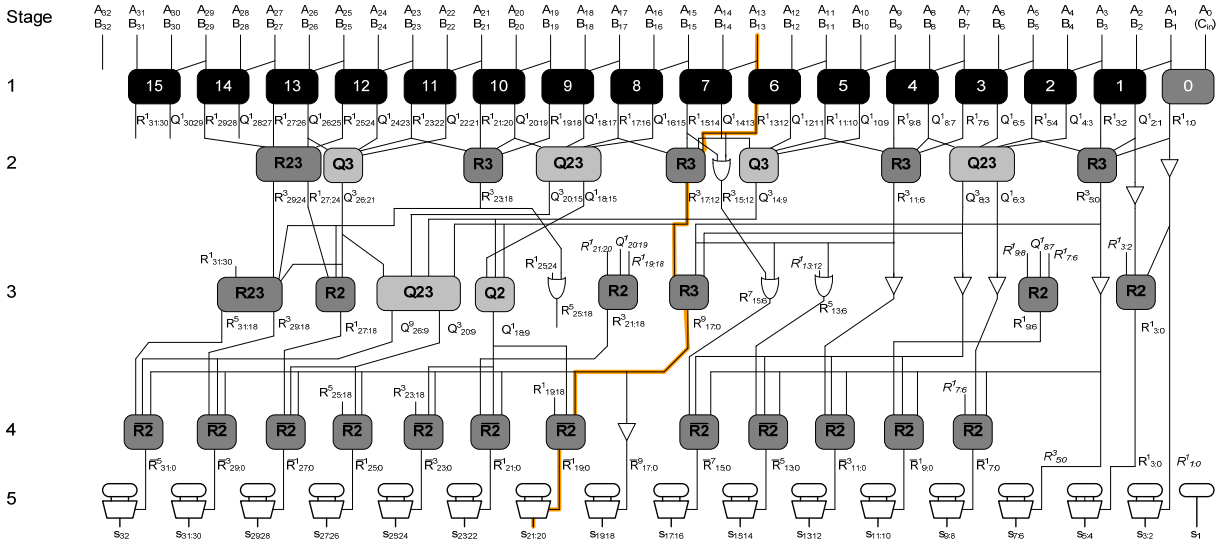

Fig. 5: Optimized 32-bit Jackson 2-3-3-2 adder

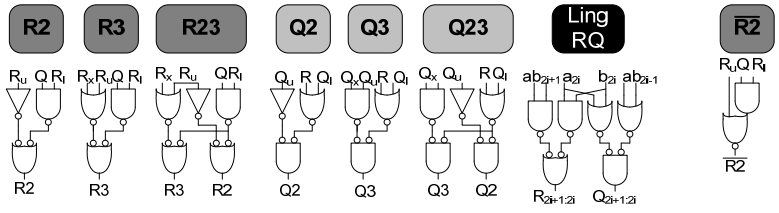

Fig. 6: Cell designs for optimized Jackson adder