Implementation of Recursive Ling Adders in CMOS VLSI

Neil Burgess
Department of Electrical and Electronic Engineering
University of Bristol
Merchant Venturer Building
Woodlands Road
BRISTOL BS8 1UB U.K.

neil.burgess@bristol.ac.uk

Abstract - **This paper presents a number of practical suggestions for implementing recursive Ling adders in deep-submicron CMOS VLSI. These adders were introduced by Jackson and Talwar in 2002 but no subsequent work has appeared describing their VLSI realisation. In this paper, we discuss how such adders might be implemented and show that recursive Ling adders are up to 34% smaller for the same speed or up to 15% faster for the same area than other recently-announced 16- and 32-b CMOS VLSI adders.**

I. Introduction

In 1981, Ling [1] observed that the $i+1^{th}$ carry output, c(i+1), of an adder operating on two binary words denoted a and b, may be simplified as follows:

$$\begin{aligned} c(i+1) = G_{i:0} &= g(i) + nk(i).g(i-1) + nk(i).nk(i-1).g(i-2) + \ldots \\ &= nk(i).g(i) + nk(i).g(i-1) + nk(i).nk(i-1).g(i-2) + \ldots \\ &= nk(i).\{g(i) + g(i-1) + nk(i-1).g(i-2) + \ldots\} \\ &= nk(i).H_{i:0} \end{aligned} \quad (1)$$

where nk denotes "not kill", defined as nk(i) = a(i) + b(i), g denotes "generate", defined as a(i).b(i), and $G_{i:0}$ denotes "group generate" as a function of all the significances from bit i down to bit 0. $H_{i:0}$ is popularly known as a "Ling carry".

Ling also showed that the corresponding sum bit, s(i+1), is derived as:

$$\begin{aligned} s(i+1) &= p(i+1) \oplus c(i+1) \\ &= p(i+1) \oplus G_{i:0} \\ &= p(i+1) \oplus \{nk(i).H_{i:0}\} \\ &= H_{i:0}.\{p(i+1) \oplus nk(i)\} + !H_{i:0}.p(i+1) \end{aligned} \quad (2)$$

where p denotes "propagate", defined as p(i) = a(i)⊕b(i). That is, the Ling carry, $H_{i:0}$, is connected to the select input of a 2-to-1 multiplexer so that the p⊕nk XOR gate is taken off the critical path of the adder.

Ling's simplification affords an acceleration of approximately 1 FO4 in contemporary CMOS VLSI parallel prefix adders by replacing the first two stages of logic in such an adder by one slightly more complex stage. Specifically, the pairs of 2-input NAND and NOR gates needed at every significance to derive the bit-level generate and not kill signals are replaced by alternate (2,2) AOI and OAI gates that return g(i+1) + g(i) and nk(i).nk(i-1) respectively. Then, the Ling carry, $H_{i:0}$ can be built up as follows:

$$H_{i:0} = \{g(i)+g(i-1)\} + \{nk(i-1).nk(i-2)\}.\{g(i-2)+g(i-3)\} + \ldots \quad (3)$$

In a parallel prefix adder, this has the effect of combining the initial NAND/NOR gates with the first stage of prefix logic, substituting two faster CMOS gates with a delay of around 1.4 FO4 each by one slower gate with a delay of around 1.8 FO4. The 2-to-1 multiplexer used to derive the sum signal in a Ling adder has the same speed as the 2-input XOR gate it replaces. (Indeed, in some cell libraries, it is implemented identically.)

II. Multi-bit Ling addition

Jackson and Talwar [2] showed that more than one not kill bit can be removed from the basic carry expression as:

$$G_{i:0} = D_{i:i-m+1}.R^{(m)}_{i:0} \quad (4)$$

where $R^{(m)}_{i:0}$ denotes a "reduced" carry signal that has had m not kill bits removed, and which Jackson and Talwar call "sub-generate". D is an m-bit correction term that restores the full carry signal, $G_{i:0}$ from the reduced one, and Jackson and Talwar have derived the following expression for $D_{i:i-m+1}$:

$$D_{i:i-m+1} = G_{i:i-m+1} + nK_{i:i-m+1} = G_{i:i-m+2} + nK_{i:i-m+1} \quad (5)$$

where $nK_{i:i-m+1}$ denotes "group not kill" as a function of all the significances from bit i down to bit i-m+1, defined as $nK_{i:i-m+1} = nk_i.nk_{i-1}.nk_{i-2}\ldots nk_{i-m+1}$.

In a similar manner to Ling addition, the sum bit is obtained using a 2-to-1 multiplexer instead of an XOR gate:

$$\begin{aligned} s(i+1) &= p(i+1) \oplus c(i+1) \\ &= p(i+1) \oplus G_{i:0} \\ &= p(i+1) \oplus \{D_{i:i-m+1}.R^{(m)}_{i:0}\} \\ &= R^{(m)}_{i:0}.\{p(i+1) \oplus D_{i:i-m+1}\} + !R^{(m)}_{i:0}.p(i+1) \end{aligned} \quad (6)$$

Ling addition can now be seen as a special case of this technique with m = 1.

III. Recursive Ling Addition

Jackson and Talwar's technique enables the development of many more prefix adder topologies than hitherto, with arbitrary numbers of "not kill" bits being removed from the carry (i.e. "group generate") signals and being corrected subsequently by the appropriate D terms. The best way to implement this technique appears to be through the use of higher-valency prefix addition, wherein more than two group generate terms are combined to form wider spans of group generate signals in fewer number of logic stages. Conventional (non-Ling) addition would proceed as follows:

Valency-2:

$$G_{i:0} = G_{i:j+1} + nK_{i:j+1}.G_{j:0} \quad (i \geq j)$$

Valency-3:

$$G_{i:0} = G_{i:j+1} + nK_{i:j+1}.G_{j:k+1} + nK_{i:k+1}.G_{k:0} \quad (i \geq j \geq k)$$

Valency-4:

$$G_{i:0} = G_{i:j+1} + nK_{i:j+1}.G_{j:k+1} + nK_{i:k+1}.G_{k:l+1} + nK_{i:l+1}.G_{l:0} \quad (i \geq j \geq k \geq l)$$

Alternatively, Jackson and Talwar's technique affords any of the following factorisations, many of which are simpler than the corresponding non-Ling prefix addition expressions, and all of which are valid for a range of values of m:

Valency-2:

$$\begin{aligned} R^{(m)}_{i:0} &= R^{(m)}_{i:j+1} + nK_{i-m:j+1}.G_{j:0} \\ &= R^{(m)}_{i:j+1} + nK_{i-m:j+1}.D_{j:j-m+1}.R^{(m)}_{j:0} \\ R^{(i-j+m)}_{i:0} &= R^{(m)}_{i:j+1} + R^{(m)}_{j:0} \quad (i \geq j) \end{aligned}$$

Valency-3:

$$\begin{aligned} R^{(m)}_{i:0} &= R^{(m)}_{i:j+1} + nK_{i-m:j+1}.D_{j:j-m+1}.R^{(m)}_{j:k+1} + nK_{i-m:k+1}.D_{k:k-m+1}.R^{(m)}_{k:0} \\ R^{(i-j+m)}_{i:0} &= R^{(m)}_{i:j+1} + R^{(m)}_{j:k+1} + nK_{j-m:k+1}.D_{k:k-m+1}.R^{(m)}_{k:0} \\ R^{(i-k+m)}_{i:0} &= R^{(m)}_{i:j+1} + R^{(m)}_{j:k+1} + R^{(m)}_{k:0} \quad (i \geq j \geq k) \end{aligned}$$

Valency-4:

$$\begin{aligned} R^{(m)}_{i:0} &= R^{(m)}_{i:j+1} + nK_{i-m:j+1}.D_{j:j-m+1}.R^{(m)}_{j:k+1} + nK_{i-m:k+1}.D_{k:k-m+1}.R^{(m)}_{k:l} + nK_{i-m:l+1}.D_{l:l-m+1}.R^{(m)}_{l:0} \\ R^{(i-j+m)}_{i:0} &= R^{(m)}_{i:j+1} + R^{(m)}_{j:k+1} + nK_{j-m:k+1}.D_{k:k-m+1}.R^{(m)}_{k:l+1} + nK_{j-m:l+1}.D_{l:l-m+1}.R^{(m)}_{l:0} \\ R^{(i-k+m)}_{i:0} &= R^{(m)}_{i:j+1} + R^{(m)}_{j:k+1} + R^{(m)}_{k:l+1} + nK_{k-m:l+1}.D_{l:l-m+1}.R^{(m)}_{l:0} \\ R^{(i-l+m)}_{i:0} &= R^{(m)}_{i:j+1} + R^{(m)}_{j:k+1} + R^{(m)}_{k:l+1} + R^{(m)}_{l:0} \quad (i \geq j \geq k \geq l) \end{aligned}$$

etc

In every case, the output group generate term G is derived from R by AND-ing it with the appropriate width D term.

It is worthwhile decomposing the D.R combination to expose the "mechanics" of the factorisation. Rewriting $R^{(m)}_{n-1:0}$ as $R^{*(m)}_{n-1:n-m} + G_{n-m-1:0}$ and AND-ing this with $D_{n-1:n-m} = G_{n-1:n-m} + nK_{n-1:n-m}$ yields:

$$G_{n-1:0} = (G_{n-1:n-m} + nK_{n-1:n-m})(R^{*(m)}_{n-1:n-m} + G_{n-m-1:0}) \quad (7)$$

Now, every sub-term in $R^{*(m)}_{n-1:n-m}$ is matched by a corresponding term in $G_{n-1:n-m+1}$ but with the missing nk bit(s) restored:

$$\begin{aligned} G_{n-1:n-m+1} &= g_{n-1} + nk_{n-1}g_{n-2} + nk_{n-1}nk_{n-2}\{g_{n-3} + \ldots \\ R^{*(m)}_{n-1:n-m} &= g_{n-1} + [nk_{n-1}]g_{n-2} + [nk_{n-1}][nk_{n-2}]g_{n-3} + \ldots \end{aligned} \quad (8)$$

where [x] indicates that the literal inside the square brackets may or may not be present depending on the factorisations employed. Thus, for each pair of corresponding sub-terms including some generate bit g_x in both G and R^* we can write the following:

$$\begin{aligned} &(nK_{n-1:x+1}g_x + nK_{n-1:x+1}nK_{x:n-m})([nK_{n-1:x+1}]g_x + G_{n-m-1:0}) \\ &= nK_{n-1:x+1}g_x[nK_{n-1:x+1}]g_x + nK_{n-1:x+1}nK_{x:n-m}[nK_{n-1:x+1}]g_x + nK_{n-1:x+1}g_xG_{n-m-1:0} + nK_{n-1:x+1}nK_{x:n-m}G_{n-m-1:0} \\ &= nK_{n-1:x+1}g_x + nK_{n-1:x+1}nK_{x:n-m}g_x + nK_{n-1:x+1}g_xG_{n-m-1:0} + nK_{n-1:x+1}nK_{x:n-m}G_{n-m-1:0} \\ &= nK_{n-1:x+1}g_x(1 + nK_{x:n-m} + G_{n-m-1:0}) + nK_{n-1:n-m}G_{n-m-1:0} \\ &= nK_{n-1:x+1}g_x + nK_{n-1:n-m}G_{n-m-1:0} \end{aligned} \quad (9)$$

That is, AND-ing the D term with R returns each generate bit, g_x, AND-ed with the required not kill bits up to the m.s.b. of D together with the lower significance G term (from R) also AND-ed with the required not kill bits up to the same significance. Consequently, suppose an R term is constructed by OR-ing two versions of a valency-3 combination:

$$R^{(6)}_{7:0} = R^{(2)}_{7:4} + R^{(2)}_{3:0}$$
$$= g_7 + g_6 + g_5 + nk_5g_4 + g_3 + g_2 + g_1 + nk_1g_0 \quad (10)$$

Then, given that $R^{(6)}_{7:0}$ will be AND-ed with $D_{7:2}$ at some later point in the addition, it does not matter that g_4 has been AND-ed with nk_5 while g_3 and lower terms have not. This brings great flexibility to the possible combinations of R terms in that "spurious" nk bits may be appended almost at random to g bits provided the index of the most significant nk bit never exceeds the significance of the reduced carry being grouped.

Many of Jackson and Talwar's factorisations contain terms of the form nK.D. This product is denoted Q by Jackson and Talwar and defined as a "hyper-propagate", as follows:

$$Q^{(m)}_{i:j} = nK_{i:i-m+1}.D_{i-m:j} \quad (11)$$

where the superscript, m, denotes the number of bits in the leading group not kill term, nK. This leads directly to the following expressions for R containing Q:

Valency-2:
$$R^{(m)}_{i:0} = R^{(m)}_{i:j+1} + Q^{(i-m-j)}_{i-m:j+1-m}.R^{(m)}_{j:0} \quad (i \geq j)$$

Valency-3:
$$R^{(m)}_{i:0} = R^{(m)}_{i:j+1} + Q^{(i-m-j)}_{i-m:j+1-m}.R^{(m)}_{j:k+1} + nK_{i-m:j+1-m}.Q^{(j-m-k)}_{j-m:k+1-m}.R^{(m)}_{k:0}$$
$$R^{(i-j+m)}_{i:0} = R^{(m)}_{i:j+1} + R^{(m)}_{j:k+1} + Q^{(j-m-k)}_{j-m:k+1-m}.R^{(m)}_{k:0} \quad (i \geq j \geq k)$$

Valency-4:
$$R^{(m)}_{i:0} = R^{(m)}_{i:j+1} + Q^{(i-m-j)}_{i-m:j+1-m}.R^{(m)}_{j:k+1} + nK_{i-m:j+1-m}.Q^{(j-m-k)}_{j-m:k+1-m}.R^{(m)}_{k:l} + nK_{i-m:j+1-m}.nK_{j-m:k+1-m}.Q^{(k-m-l)}_{k-m:l+1-m}.R^{(m)}_{l:0}$$
$$R^{(i-j+m)}_{i:0} = R^{(m)}_{i:j+1} + R^{(m)}_{j:k+1} + Q^{(j-m-k)}_{j-m:k+1-m}.R^{(m)}_{k:l} + nK_{j-m:k+1-m}.Q^{(k-m-l)}_{k-m:l+1-m}.R^{(m)}_{l:0}$$
$$R^{(i-k+m)}_{i:0} = R^{(m)}_{i:j+1} + R^{(m)}_{j:k+1} + R^{(m)}_{k:l} + Q^{(k-m-l)}_{k-m:l+1-m}.R^{(m)}_{l:0}$$
etc $(i \geq j \geq k \geq l)$

Now, these expressions are inconvenient for implementation because they contain both nK and Q terms. However, Q terms are interchangeable with nK strings with the same indices, as follows:

$$Q^{(i-m-j)}_{i-m:j+1-m}.R^{(m)}_{j:k+1} + Q^{(i-m-j)}_{i-m:j+1-m}.X$$
$$= nK_{i-m:j+1}.D_{j:j+1-m}.R^{(m)}_{j:k+1} + nK_{i-m:j+1}.D_{j:j+1-m}.X$$
$$= nK_{i-m:j+1}.G_{j:k+1} + nK_{i-m:j+1}.(G_{j:j+1-m} + nK_{j:j+1-m}).X$$
$$= nK_{i-m:j+1}.(G_{j:j+1-m} + nK_{j:j+1-m}.G_{j-m:k+1}) + nK_{i-m:j+1}.(G_{j:j+1-m} + nK_{j:j+1-m}).X$$
$$= nK_{i-m:j+1}.(G_{j:j+1-m} + nK_{j:j+1-m}.G_{j-m:k+1}) + nK_{i-m:j+1}.nK_{j:j+1-m}.X$$
$$= nK_{i-m:j+1}.G_{j:k+1} + nK_{i-m:j+1-m}.X$$
$$= Q^{(i-m-j)}_{i-m:j+1-m}.R^{(m)}_{j:k+1} + nK_{i-m:j+1-m}.X \quad (12)$$

Thus, by analogy with the R term, a Q term may contain "spurious" g bits which will be covered by R terms elsewhere, and so, in practice, nK strings can be replaced by Q terms with the same indices. This leads immediately to a simple valency-2 recursion on Q:

$$Q^{(i-j+m)}_{i:0} = Q^{(m)}_{i:j+1}.Q^{(m)}_{j:0} \quad (i \geq j)$$

A second valency-2 recursion on Q is also possible in which the superscript is unchanged:

$$Q^{(m)}_{i:0} = Q^{(m)}_{i:j+1}.(R^{(i-m-j)}_{i-m:j+1-m} + Q^{(m)}_{j:0}) \quad (i \geq j)$$

This can be demonstrated as follows:

$$Q^{(m)}_{i:j+1}.(R^{(i-m-j)}_{i-m:j+1-m} + Q^{(m)}_{j:0})$$
$$= nK_{i:i+1-m}.D_{i-m:j+1}.(R^{(i-m-j)}_{i-m:j+1-m} + nK_{j:j-m+1}.D_{j-m:0})$$
$$= nK_{i:i+1-m}.(D_{i-m:j+1}.R^{(i-m-j)}_{i-m:j+1-m} + D_{i-m:j+1}.nK_{j:j-m+1}.D_{j-m:0})$$
$$= nK_{i:i+1-m}.\{G_{i-m:j+1-m} + (G_{i-m:j+2} + nK_{i-m:j+1}).nK_{j:j-m+1}.(G_{j-m:1} + nK_{j-m:0})\}$$
$$= nK_{i:i+1-m}.\{G_{i-m:j+1-m} + nK_{i-m:j+1}.nK_{j:j-m+1}.(G_{j-m:1} + nK_{j-m:0})\}$$
$$= nK_{i:i+1-m}.\{G_{i-m:j+1-m} + nK_{i-m:j-m+1}.G_{j-m:1} + nK_{i-m:0}\}$$
$$= nK_{i:i+1-m}.(G_{i-m:1} + nK_{i-m:0})$$
$$= nK_{i:i+1-m}.(D_{i-m:0}) = Q^{(m)}_{i:0} \quad (13)$$

Other recursions on Q with different valencies then follow:

Valency-3:
$$Q^{(m)}_{i:0} = Q^{(m)}_{i:j+1}.(R^{(i-m-j)}_{i-m:j+1-m} + Q^{(m)}_{j:k+1}.(R^{(j-m-k)}_{j-m:k+1-m} + Q^{(m)}_{k:0}))$$
$$Q^{(i-j+m)}_{i:0} = Q^{(m)}_{i:j+1}.Q^{(m)}_{j:k+1}.(R^{(j-m-k)}_{j-m:k+1-m} + Q^{(m)}_{k:0})$$
$$Q^{(i-k+m)}_{i:0} = Q^{(m)}_{i:j+1}.Q^{(m)}_{j:k+1}.Q^{(m)}_{k:0} \quad (i \geq j \geq k)$$

Valency-4:
$$Q^{(m)}_{i:0} = Q^{(m)}_{i:j+1}.(R^{(i-m-j)}_{i-m:j+1-m} + Q^{(m)}_{j:k+1}.(R^{(j-m-k)}_{j-m:k+1-m} + Q^{(m)}_{k:l}.(R^{(k-m-l)}_{k-m:l+1-m} + Q^{(m)}_{l:0})))$$
$$Q^{(i-j+m)}_{i:0} = Q^{(m)}_{i:j+1}.\ Q^{(m)}_{j:k+1}.(R^{(j-m-k)}_{j-m:k+1-m} + Q^{(m)}_{k:l}.(R^{(k-m-l)}_{k-m:l+1-m} + Q^{(m)}_{l:0}))$$
$$Q^{(i-k+m)}_{i:0} = Q^{(m)}_{i:j+1}.\ Q^{(m)}_{j:k+1}.Q^{(m)}_{k:l}.(R^{(k-m-l)}_{k-m:l+1-m} + Q^{(m)}_{l:0})$$
$$Q^{(i-l+m)}_{i:0} = Q^{(m)}_{i:j+1}.\ Q^{(m)}_{j:k+1}.Q^{(m)}_{k:l}.Q^{(m)}_{l:0} \quad (i \geq j \geq k \geq l)$$
etc

Thus, Jackson and Talwar have discovered two new design dimensions for constructing VLSI adders: factoring out arbitrary numbers of "not kill" bits; practical higher-valency adder construction. Suddenly, the design space for VLSI adders has exploded!

IV Practical recursive Ling adder designs

The key to designing high-performance recursive Ling adders lies in balancing the relative complexities of the R and D terms. That is, the number of nk bits to be factored out of an R term in deriving a group generate term must be weighed against the increased logic depth of the required D term. Now, given that D is XOR'd with p before the output multiplexer that returns the sum signal, D should be available at least 2 FO4 (i.e. the delay of an XOR gate) before R. This condition in turn implies the first recommendation:

- no nk bits should be factored out in the final stage of the R tree (i.e. the final stage should be non-Ling, probably valency-2, since this is the only logic expression available as a single CMOS cell in most cell libraries)

Secondly, the advantage secured by using (2,2) AOI and OAI gates in the first stage of the R tree, as discussed earlier, is too great to be discarded. Thus, the second recommendation is:

- factor out 1 nk bit in the first stage of the R tree

Thirdly, in order to avoid excessive fan-out loading along the critical path through the R tree, the D terms should be derived from *buffered* versions of the $R^{(1)}$ and $Q^{(1)}$ terms in the first stage of the tree. Thus, recalling that the D term critical path must also accommodate an XOR gate (with a delay of 2 approximately FO4), the third recommendation is:

- D terms should be constructed from logic at least 3 FO4 shallower than their corresponding R terms

A final observation is that the valencies selected will depend on the population of the cell library being used to implement the design. Current practice typically limits the number of inputs of a CMOS logic cell to four, with a further restriction on the number of series p-FETs to three or even two. Consequently, only valency-2 and (some) valency-3 equations might be realisable as single CMOS cells; however, the new adders also provide many options for implementing higher-valency equations, some of which might prove a better fit to a particular cell library than others.

Figure 1 shows an example 24-bit recursive Ling adder (R outputs only); black squares are R functions; grey squares are Q functions; white squares are identity (buffer-only) functions. In addition, valencies have been included in the diagram, as have the widths of the D terms. Not all buffers have been shown in order to clarify the connectivity of the R and Q functions; in practice, there is scope for inserting buffers so as to reduce fan-out load on critical nodes e.g. the $R^{(1)}_{11:10}$ cell in the first row. Note the use of valency-3 expressions, exploiting the availability of cells realising the logic functions a+b+c.d or a.b.(c+d) in this particular cell library. Also note that the D terms -- up to 4 bits in this example -- can also all be implemented as single cells from this cell library: for example $D_{16:13} = G_{16:14} + nK_{16:13} = g_{16} + Q^{(1)}_{16:15}(R^{(1)}_{15:14} + Q^{(1)}_{14:13})$.

Table I gives the results obtained from a number of trial adder designs undertaken using a structured place and route methodology in a 65nm CMOS process technology. All the results have been normalised to a 32-b Kogge-Stone adder, which was also presented in all the comparison papers. This was done to take account of the different CMOS technologies used in the 3 papers (90nm [3]; 180nm [4]; 65nm in this work). After normalisation, the 16-bit Kogge-Stone and Ladner-Fisher adders reported in [4] also closely match our results, giving further confidence that the comparisons of other adders are valid. The 24-b adder reported is that shown in Figure 1.

The Table shows that compared with [3], the new adders are always faster for comparable area by up to 15%, and that compared with [4], the new adders are always smaller for comparable speed by up to 35%. (The new adders are also superior to the non-Ling adders.) This is highlighted by the Area × Delay (A×T) comparison, which shows that the new adders consistently have a superior area-delay characteristic. Moreover, the 20-b adder has a better AT value than either of the previously-published 16-bit adders, and the 24-b adder has the same AT value as a Kogge-Stone 16-b adder!

V Summary

This paper has reviewed Jackson and Talwar's proposal for new high-speed binary adders, and proposed a number of guidelines for their successful realisation in DSM CMOS VLSI. Further work is needed to formalise adder construction using this theory and to investigate if this approach may be combined to good effect with other adder topologies such as carry-select, carry-increment.

Acknowledgement

The author is grateful to P. Patel and H. Raajaraajan, two MSc students at the University of Bristol, for performing trial builds of some of the adders reported in this paper.

References

[1] H. Ling, "High Speed Binary Adder", *IBM Journal of Research and Development*, **25** (1981), pp.156-166

[2] R. Jackson and S. Talwar, "High Speed Binary Addition", Proc. 38th IEEE Asilomar Conf. Signals, Systems and Computers, pp. 1350-1353, Asilomar, CA, Nov. 2004

[3] S. Das and S.P. Khatri, "A Hybrid Parallel-Prefix Adder Architecture with Efficient Timing-Area Characteristic", *IEEE Trans. VLSI Systems*, **16** (2008), pp. 326-331

[4] G. Dimitrakopoulos and D. Nikolos, "High-Speed Parallel-Prefix VLSI Ling Adders", *IEEE Trans. Computers*, **54** (2005), pp. 225-231

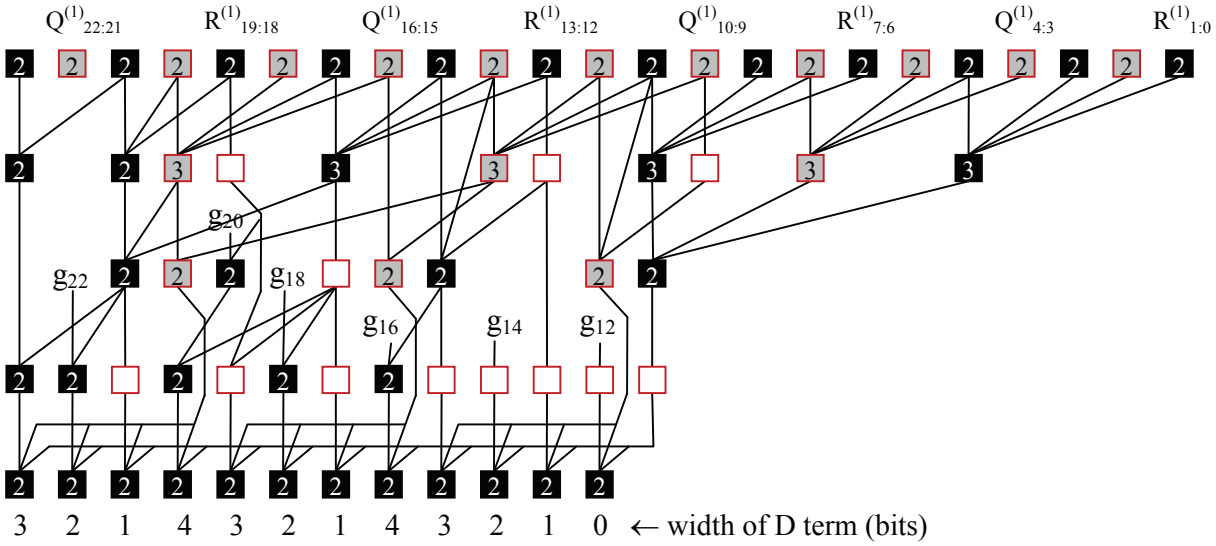

Fig. 1. R tree for 24-bit recursive Ling adder

TABLE I
COMPARISON OF ADDERS' DELAY, AREA, AND AT PRODUCT

	Normalised Delay			Normalised Area			A×T		
	[3]	[4]	This work	[3]	[4]	This work	[3]	[4]	This work
32-b Ks	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
16-b Ks	0.82		0.82	0.43		0.41	0.35		0.34
16-b Lf	0.86		0.86	0.32		0.34	0.28		0.29
32-b Ling			0.92			1.0			0.92
32-b	1.01	0.87	0.88	0.72	1.01	0.75	0.73	0.88	0.66
16-b	0.88	0.70	0.75	0.35	0.43	0.28	0.31	0.30	0.21
20-b			0.76			0.37			0.28
24-b			0.78			0.46			0.34