Optimizing Carry Lookahead Adders for Semicustom CMOS

C. D. Thomborson

Computer Science Department
University of Minnesota, Duluth
Duluth, MN 55812

Y. Sun

Computer Science Department
University of Minnesota, Duluth
Duluth, MN 55812

Abstract

We present a practical method for constructing optimal carry lookahead adders, of width $n \leq 84$. We formulate this design problem as a two-dimensional dynamic program, in which optimization is performed with respect to both adder size and latency. Our adders are fast, modularly built, relatively easy to lay out, and fully exploit the timing characteristics of CMOS standard cells or gate arrays.

1 Introduction

Our work is based on Wei and Thomborson's systematic method of designing optimal VLSI carry-lookahead adders, using dynamic programming [1]. Recently, Chan, Oklobdzija, Schlag and Thomborson (COST) have applied the Wei-Thomborson method to the problem of optimizing carry-skip adders and carry-lookahead adders with variable block widths [2].

In this paper, we show how to find optima in a somewhat narrower class of adders than those considered by COST. Our adders are carry-lookahead structures in which the block width is two. However, since we allow the use of NAND-OR and AND-NOR gates in our circuits, and since our timing models are more exact than that used by COST, our adders may outperform the COST adders by 6% in speed. The evidence for this claim is presented in Section 5; Sections 2, 3, and 4 describe our class of adders, our timing models, and our optimization algorithm, respectively.

2 Carry lookahead adders of block-width 2

As shown in Figure 1, we decompose the problem of binary addition into three stages: the preprocessor, the carry lookahead circuitry, and the postprocessor. The preprocessor is composed of two gates per pair of input bits (a_i, b_i), computing the "propagate" signal $p_i = a_i + b_i$ and the "generate" signal $g_i = a_i b_i$. The lookahead circuitry computes carry-in bits c_{i-1} as a function of the propagate and generate signals from lower-order input pairs (a_j, b_j) for $0 \leq j < i$. The postprocessor computes the sum bits $s_i = a_i \oplus b_i \oplus c_{i-1} = \overline{c_i}(a_i + b_i + c_{i-1}) + a_i b_i c_{i-1}$. In some applications, the postprocessor would also compute the overflow bit $o = c_{n-1} \oplus c_{n-2} = \overline{s_{n-1}} a_{n-1} b_{n-1} + s_{n-1} \overline{a_{n-1} b_{n-1}}$. In our formulation, the carry-in bit is c_{-1}, the least-significant input bits are (a_0, b_0), and the most-significant input bits are (a_{n-1}, b_{n-1}).

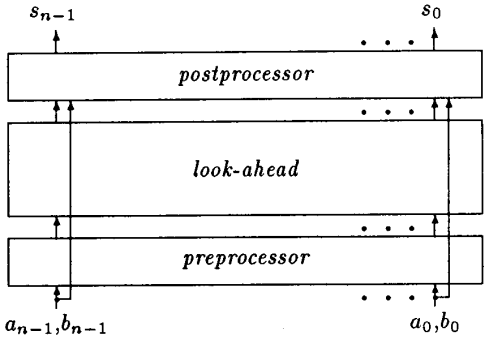

Figure 1: Different component blocks in a lookahead.

Since both preprocessor and postprocessor are very simple 1-level or 2-level circuits, we focus our optimization efforts on the lookahead circuitry. In this paper, we use the standard technique of computing c_{i-1} with a binary tree of operations $\circ$ on propagate and generate bits [4]:

$$(p_i, g_i) \circ (p_j, g_j) = (p_i p_j, g_i p_j + g_j) \qquad (1)$$

and

$$(p_0, g_0) \circ .... \circ (p_i, g_i) = (p_0 ... p_i, \quad g_i + p_i g_{i-1} + p_i p_{i-1} g_{i-2} + ... + p_i p_{i-1} ... p_1 p_0 c_{-1}) \tag{2}$$

or

$$(p_0, g_0) \circ .... \circ (p_i, g_i) = (p_0 ... p_i, c_i) \tag{3}$$

Note that $\circ$ is not commutative but it is associative. Also note that we consider only computations using the binary operator $\circ$. By contrast, COST have constructed optimal adders with k-ary operators $\circ_k$ for $2 \leq k \leq 4$.

To construct an optimal adder using equations (1)–(3), we must make three types of decisions.

- What "low-level" circuit design should we use when computing the $\circ$ operator at each node in the binary tree for c_{i-1}?
- What "shape" should the binary tree take (*i.e.* should it be a perfectly balanced tree or should it have some more complicated form)?
- How should we "share" intermediate results among the similar computations for the various c_i?

We answer the first two of these questions by deferring them, for as long as possible, in our exploration of the design space. That is, we design several "low-level" circuits, and let our optimization routine decide which is the most appropriate for each computation of $\circ$ in an adder of some particular width n. Our optimization routine also finds the most appropriate shape for the binary tree.

We handle third implementation decision in the same way as Wei and Thomborson, using buffers to broadcast propagate and generate bits. We must, of course, choose an appropriately-sized buffer for this job; and, once again, we defer the problem of choice to our optimization routine.

To summarize, we design lookahead circuits by binary decomposition, as shown in Figure 2. The black boxes in this figure compute the $\circ$ function on the bits coming from below, producing (p, g) bits on their upper output lines; the white boxes are buffers. Black boxes have "pass-through" wires connecting their left- and right-hand terminals. Finally, a few white boxes have (possibly large) buffers to amplify the signals coming from below, before feeding them to the left. There is one such white box shown in Figure 2, near the middle of the diagram.

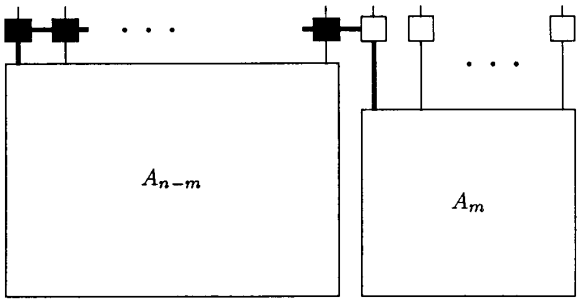

Figure 2: Block A_n with its subadders A_m and A_{n-m}.

We design our "black cells" from the inverting logic available in semicustom design books. The p function is thus a 2-NOR or a 2-NAND; the g function is thus a 3-input OR-NAND or AND-NOR. Such gates are included in most semicustom design books.

Some of our "white cells" are inverting low-power buffers, used to maintain the correct signal polarity. Some white cells are optimized to be just pass-throughs. Finally, a few are high-power buffers, either inverting or non-inverting, broadcasting signals of the correct polarity to a row of black cells.

3 Delay model

We use two timing models in this paper. To compare our results to COST, we use their model:

$$t_{\text{nand}_{\text{out}}} = t_{\text{in}} + 5 \cdot \text{FO} + 20 \cdot \text{FI} \tag{4}$$

$$t_{\text{nor}_{\text{out}}} = t_{\text{in}} + 10 \cdot \text{FO} + 20 \cdot \text{FI} \tag{5}$$

$$t_{\text{and}_{\text{out}}} = t_{\text{in}} + 5 \cdot \text{FO} + 20 \cdot \text{FI} + 17 \tag{6}$$

$$t_{\text{or}_{\text{out}}} = t_{\text{in}} + 5 \cdot \text{FO} + 20 \cdot \text{FI} + 17 \tag{7}$$

$$t_{\text{inv}_{\text{out}}} = t_{\text{in}} + 5 \cdot \text{FO} + 12 \tag{8}$$

$$t_{\text{buffer}_{\text{out}}} = t_{\text{in}} + 5 \cdot \text{FO} + 12 \tag{9}$$

The integer values of the COST delay units can be transformed into nanoseconds, given the delay t of an inverter under unit load in any particular CMOS techology. For example, in a 1.5μm CMOS technology in which an inverter has delay $t = 0.3$ nsec, one COST delay unit is equivalent to $t/12 = 0.025$ nsec.

These equations have been obtained by "fitting data from an ASIC-CMOS standard cell library [5]."

Gate	ID	Unit-load delay (nsec)	Incremental delay (nsec/gate)
2–NAND	X001	1.365	0.540
2–NOR	X004	1.100	0.605
2–AND	X930	2.145	0.350
2–OR	X936	2.330	0.365
INVERTER	X912	0.530	0.115
BUFFER	X914	2.155	0.145
AND-NOR	X945	1.820	0.680
		1.775	0.500
OR-NAND	X939	2.535	0.765
		2.075	0.509

Table 1: Gates with identifications, propagation delays and incremental delays.

In our work, the NAND, NOR, AND and OR gates have fan-in $FI = 2$.

The COST model is a good approximation, but we believe that better adders can be obtained by using the published timing figures for the particular semicustom VLSI design process under consideration. For example, National Semiconductor's 2μm standard cell data book [6] gives the timing data of Table 1, if we take the arithmetic average of the worst-case delays for the HL and the LH transitions. Note that the asymmetric AND-NOR and OR-NAND gates have one "fast" input and two "slower" ones. Also note that, for simplicity, we have listed only the lowest-power gate of each functional type. Our optimization routine is given the timing parameters for the higher-power versions, as well, allowing it to select an optimally "sized" gate in each instance.

4 Algorithm description

We use dynamic programming to search the huge space of possible adder designs. We begin the process by enumerating a few "good" 1- and 2-bit adders. We then construct all "good" 3-bit adders using the composition rule of Figure 2. Note that we have several degrees of freedom here: we can have $m = 1$ or $m = 2$, we have to choose a signal polarity for the outputs of A_m, and we have to choose appropriate implementations of black boxes and the broadcasting white box.

In general, our strategy is to construct all possible k-bit adders by enumerating over all m, over all good $(k-m)$-bit and m-bit adders, and over all available implementations of black boxes and broadcasting white boxes. We keep two lists of twenty of the "best" k-bit adders, one for each output polarity, for use when constructing larger adders. In the context of this paper, "best" means that an adder has a faster g_{k-1} output than any k-bit adder that is either smaller or has a slower p_{k-1} output.

If a twenty-first "best" adder is discovered, we randomly select a candidate for deletion, in such a way that each adder has equal likelihood of appearing in the final list. This "random pruning" heuristic is novel. We believe it to be a valuable addition to the list of pruning techniques described in the COST paper [2].

adder width	COST adder	our adder	ratio
16	489	525	1.07
32	627	702	1.12
48	716	814	1.14
64	797	924	1.16
84	856	1029	1.20

Table 2: Delays of optimal-time adders for various adder widths, using the timing model of COST.

5 Results and analysis

We developed a C-language computer program to perform the optimization described in the previous section. Our first set of results, shown in Table 2, shows that our adders are 7% to 20% slower than those of COST [2], if we are not allowed to use NAND-OR or NOR-AND gates. This is not a surprising finding, given that we are optimizing over a smaller design space. Clearly, the use of variable-block widths in the COST adder is a good idea, especially for large adder widths.

Our second set of results, in Table 3, shows that significantly faster adders can be built from 2μm National Semiconductor standard cells, if one takes advantage of the 3-input AND-NOR and NOR-AND cells. Such cells are not included in the COST timing model, perhaps because it is not at all clear how one might extend their functional definition to large fanin.

Figure 3 shows a time-optimal 48-bit adder produced by our program, for the National Semiconductor standard cell timing data, using NAND-OR and NOR-AND gates in the black cells. The size of the

adder width	Restricted design (nsec)	Unrestricted design (nsec)	ratio
16	19.67	13.74	0.70
32	24.70	18.54	0.75
48	29.08	21.72	0.75
64	32.51	24.44	0.75
84	35.86	27.94	0.78

Table 3: Delays of our adders constructed from 2μm National Semiconductor standard cells, for various adder widths. Restricted designs do not use AND-NOR or OR-NAND cells.

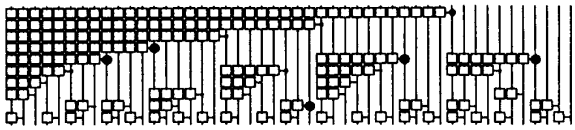

Figure 3: Layout of the optimal-time 48-bit adder with black cells using AND-NOR and OR-NAND gates.

black dots is an indication of the strength of the buffer at that position.

Judging from the timing ratios of Table 3, we see that an optimal width-84 carry-lookahead adder with NAND-OR and OR-NAND gates is $1/0.78 - 1 = 28\%$ faster than one built only from 2-input AND, OR, NAND, and NOR gates, buffers, and inverters. By contrast, Table 2 shows only a 20% speedup in a width-84 adder built of 2-, 3- and 4-input AND, OR, NAND, and NOR gates, in comparison to an optimal adder built of 2-input logic gates.

We tentatively conclude that the use of NOR-AND and OR-NAND gates is of greater benefit to the design of fast adders than is the use of large-fanin gates. Quantitatively, our best estimate is that our 84-input adder would have worst-case delay $1.20 \times 0.78 = 0.94$ times that of the 84-bit COST adder, which is to say that we expect to see approximately a 6% speedup over that structure, in a National Semiconductor standard cell implementation. In the near future, we intend to use a Mentor Graphics design system to perform a timing analysis on our adder, and the COST adder, for various technologies, to see what speedups are predicted on this "level playing field."

Acknowledgements

Renato Milanesi wrote an early version of our adder optimization routines [3]. The optimal adders found by Renato's implementation are exactly as fast as the adders found by our current implementation, giving us confidence in the accuracy of both programs.

References

[1] B. W. Y. Wei and C. D. Thomborson, *Area-time Optimal Design,* IEEE Transactions on Computers, Vol.39, No.5, May 1990.

[2] P. K. Chan, V. G. Oklobdzija, M. D. F. Schlag and C. D. Thomborson, *Delay Optimization of Carry Skip Adders and Block Carry Lookahead Adders Using Multidimensional Dynamic Programming,* IEEE Transactions on Computers, Vol.41, No.8, August 1992.

[3] R. C. Milanesi, *Optimal Look-ahead Adders,* M.S. Thesis, University of Minnesota, Duluth, 1991.

[4] R. E. Ladner and M. J. Fischer, *Parallel Prefix Computation,* Journal of ACM, October 1980.

[5] V. G. Oklobdzija and E. R. Barnes, *On Implementing Addition in VLSI Technology,* Journal of Parallel and Distributed Computing, Vol.5, 1988.

[6] National Semiconductor Corporation, *ASIC Design Manual, CMOS Gate Arrays, CMOS Standard Cells,* 1987 Edition.