Analysis and Design of CMOS Manchester Adders with Variable Carry-Skip

Pak K. Chan and Martine D.F. Schlag*
Computer Engineering
University of California, Santa Cruz
Santa Cruz, California 95064

Abstract

Carry-skip adders compare favorably with other adders for efficient VLSI implementation. Lehman has shown that carry-skip adders with variable-size blocks are faster than adders with constant size blocks. In 1967, Majerski suggested that multi-level implementation of the variable-size carry-skip adders would provide further improvement in speed. Almost two decades later, Oklobdzija and Barnes developed algorithms for determining the optimal block sizes for one-level and two-level implementations, and a generalization of their results was given by Guyot *et al.* In these analyses, all adder cells are implemented with static (restoring) logic. Therefore the ripple-carry propagation delay is linearly proportional to the size of a block. A more popular VLSI adder implementation is the Manchester adder using dynamic (precharge) logic, where the ripple-carry propagation delay of a block is proportional to the *square* of its size. We shall examine two different CMOS implementations of the Manchester adder, analyzing them with the RC timing model, which provides us a unified way of analyzing both CMOS circuits and interconnect. Based on the RC timing model, we develop efficient (polynomial) algorithms to determine near-optimal, as well as optimal block sizes for the one-level Manchester adder with variable carry-skip.

1 Introduction

1.1 Adders and carry-skip adders

The adder is the major component in an ALU. There are many kinds of adders available for conventional number systems, we shall list some implementations of adders according to Blaauw's classification [3]. The adders' synonym and asymptotic time complexity are also given:

1. basic carry-ripple adders: $O(n)$,
2. carry-predict (lookahead) adders: $O(log\ n)$,
3. carry-skip (bypass) adders: $O(n^{1/l})$, where l is the number of skip layers (for a linear ripple, constant skip adder model), and
4. carry-select (conditional sum) adders: $O(log\ n)$.

Speaking in terms of the methods used in the design of an adder to achieve high-speed calculation, the carry-predict adders improve the performance of the basic carry-ripple adder by making the slow signals arrive earlier. The carry-skip adders improve the performance of the basic carry-ripple adder by making the early signals available sooner, trading the available time against resources. In the carry-predict adder, the early signals are duplicated, at the expense of additional resources, to reduce the number of levels in the adder.

Variable skip adders have been investigated by many researchers [2, 4, 6, 7, 9]. In these analyses, they assumed all adder cells were implemented with static (restoring) logic. Therefore the ripple-carry propagation delay was linearly proportional to the size of a block. This is in contrast to a more popular adder implementation in VLSI: the Manchester adder using precharge logic [5, 8, 10, 12, 13], in which the ripple-carry propagation delay is proportional to the *square* of the size of a block.

In this article, we focus on the analysis and design of CMOS Manchester carry-skip adders with variable size blocks. Similar work was reported by Barnes and Oklobdzija in [2]. However, we believe that our timing models for the carry-ripple and carry-skip in dynamic CMOS adders are more accurate. Our analysis shows that the carry-skip delay in a Manchester adder block is linearly proportional to the block size, and is not constant as they reported. Also our approach provides a general paradigm for analysis and design, applicable to different models of ripple-propagation and carry-skip.

1.2 Manchester adder

We are designing an n-bit adder composed of m blocks, as depicted in Figure 1. Buffers are inserted in between the blocks. Each adder block may not necessarily be of equal size. Our design method involves:

1. analyzing the timing of a x-bit adder block,
2. locating the critical path of the n-bit adder, and
3. determining the block sizes to reduce the delay of the critical path.

*Supported in part by NSF Presidential Young Investigator Grant MIP-8896276

Figure 1: Variable-Size-Block Adder

1.3 Adaptation of Manchester adders for dynamic CMOS

Figure 2 shows a 4-bit Manchester adder block implemented in dynamic CMOS logic [13]. This adder circuit operates as follows: the nodes are precharged during phase $\overline{CK}$ through the p-transistors, and the circuit is evaluated during phase CK. Depending on the values of the addends A_i and augends B_i, as well as C_{i-1}, a carry can either be generated or propagated.

$$G_i = A_i \wedge B_i \qquad P_i = A_i \oplus B_i \qquad i = 1, \ldots, 4.$$

The sum S_i and the carry C_i are determined by

$$C_i = G_i \vee P_i \wedge C_{i-1},$$

$$S_i = C_{i-1} \oplus A_i \oplus B_i = C_{i-1} \oplus P_i.$$

The critical path of this circuit is the serially connected pass transistors in the carry chain. The bypass transistor improves the worst-case carry propagation time if all carry propagates P_1, P_2, P_3, P_4 are true.

The additional circuitry that is needed to generate the bypass signal is shown in Figure 4. Note that even though this circuit (to generate B) has similar circuitry to Figure 2, the node capacitance at the intermediate nodes in the AND gate is less than that of the Manchester carry chain.

Figure 3 shows a slightly different implementation of CMOS Manchester adder. Here the NAND gate serves as a buffer and combines the carry-bypass and $\overline{C_4}$ signals. This results in a faster carry-skip but longer buffer delay.

2 Timing analysis of an x-bit adder block

The classical method of determining the speed of adder circuits is by counting the number of gate delays. This method is not appropriate for the CMOS implementations of the Manchester adder shown in Figures 2 and 3 because delays incurred by the pass transistors are not linearly additive. We shall analyze the timing of these circuits using the Resistor/Capacitor (RC) timing model.

2.1 RC timing model

In this model, transistors are modeled as linear resistors, node capacitances as grounded capacitors, and interconnection wires as lumped RC circuits [11]. Thus, circuits and interconnect can be analyzed analytically in a unified manner.

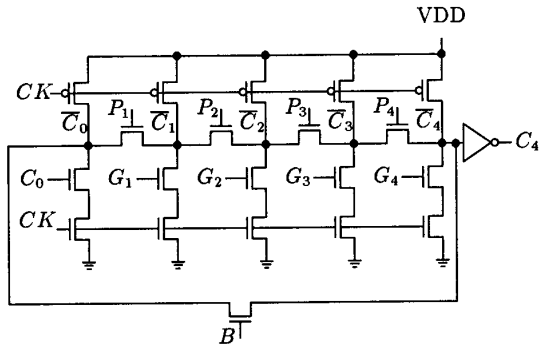

Figure 2: 4-bit Manchester Adder with Carry-Skip: with inverter

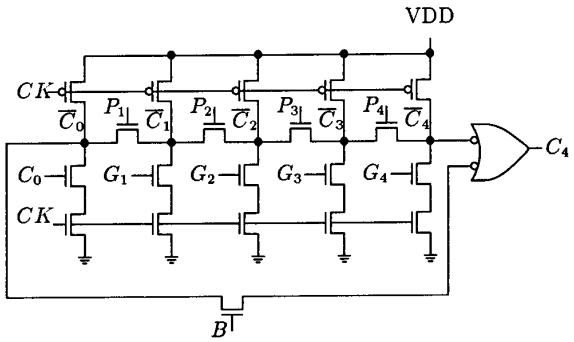

Figure 3: 4-bit Manchester Adder with Carry-Skip: with NAND gate

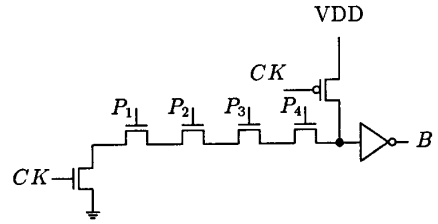

Figure 4: Carry-skip Signal Generation Circuit

2.1.1 Timing analysis of an x-bit adder block: circuit 1

Carry-skip delay: Consider an x-bit adder block. The carry-skip mechanism is activated when all the pass transistors $P_1, \ldots, P_x$ conduct, and hence the bypass transistor B conducts; altogether they form a closed-loop of transistors. When the clock signal CK is high and the input carry C_0 arrives, the circuit will start to discharge. The RC timing model of this situation will be analyzed by a technique suggested in [1], equation (6). The technique involves analyzing the RC models of the x pass transistors $P_1, \ldots, P_x$ and the bypass transistor B separately, and piecing together the results with equation (2). With this in mind, referring to Figure 5, the resistor r_0 models the conducting transistors

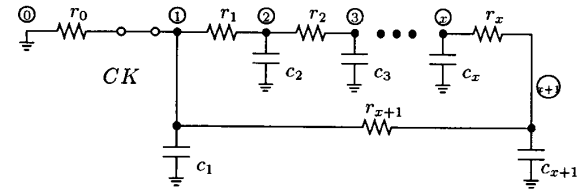

Figure 5: RC Model of CMOS Manchester Adder (inverter implementation)

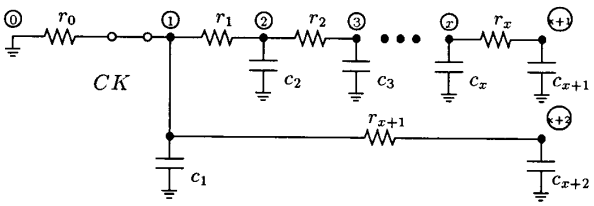

Figure 6: RC Model of CMOS Manchester Adder (NAND gate implementation)

(serially connected) controlled by C_0 and CK, similarly r_i models P_i for $i = 1, \ldots, x$, and finally r_{x+1} models the bypass transistor B. Here we assume that $r_0 = r_i = r, c_i = c$, for $i = 1, \ldots, x+1$.

The signal delay of a series of $x+1$ RC-chain is

$$t' = \sum_{j=1}^{x+1} r_{j,0} c_j = \frac{(x+1)(x+2)}{2} rc. \tag{1}$$

where $r_{j,0}$ is the resistance of the path from node j to node 0. Now we analyze the RC model of the bypass transistor individually, which is simply $r_{x+1}c_{x+1} = rc$; we piece these results together to obtain the carry-skip delay of an x-bit adder block

$$S(x) = t' - \frac{(x+1)rc - t'}{r - (r-(x+1)r)}(r - (x+1)r) \tag{2}$$

which can be simplified to

$$S(x) = \frac{1}{2}(3x+2)rc. \tag{3}$$

Carry-generate delay: Consider a carry that is generated at bit one of an x-bit adder block: $G_1 = 1$, $C_0 = 0$, $P_1 = 0$, $G_i = 0$, and $P_i = 1; i = 2, \ldots, x$. The carry generated ripples through x RC chains; the carry-generate signal delay is therefore equal to

$$G(x) = \frac{x(x+1)}{2} rc \approx \frac{x^2}{2} rc. \tag{4}$$

Carry-ripple delay: Consider a carry input (C_0) that enters an x-bit adder block, propagates through $x-1$ pass transistors, and is finally absorbed at the x^{th} bit, i.e. $P_1 = \ldots = P_{x-1} = 1$, $P_x = 0$, and $G_i = 0; i = 1, \ldots, x$. The carry-ripple signal delay is

$$R(x) = \frac{x(x+1)}{2} rc \approx \frac{x^2}{2} rc. \tag{5}$$

2.1.2 Timing analysis of an x-bit adder block: circuit 2

Carry-skip and Carry-ripple delay: Now we analyze the timing of the Manchester adder implemented with NAND gate as depicted in Figure 3. The RC circuit model that is used to analyze the carry-skip delay for this circuit is shown in Figure 6. The signal delays for carry-skip and carry-ripple are respectively

$$\begin{aligned} S(x) &= \sum_{j=1}^{x+2} r_{j,x+2} c_j \\ &= (r_0 + r_{x+1})c_{x+2} + r_0 \sum_{j=1}^{x+1} c_j = (x+3)rc \end{aligned} \tag{6}$$

where $r_{j,x+2}$ is the resistance of the portion of the path between nodes 0 and $x+2$, which is common with the path between nodes 0 and j; and

$$R(x) = G(x) = \frac{x(x+1)}{2} rc \approx \frac{x^2}{2} rc. \tag{7}$$

Equations (3) to (7) indicate that for both implementations of the Manchester adder, the carry-skip delay is linearly proportional to the block size, whereas the carry-ripple delay is quadratic. This enables us to consolidate the two models into a single one.

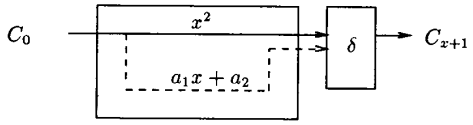

Figure 7: Timing Model of the CMOS Manchester Adders

2.2 Critical path

The succession of switching elements that determines the speed of a circuit is called a *critical path* of the circuit. The critical paths show which parts of a design determine its speed. There may be more than one critical path through a circuit, in which case they all have the same delay.

In determining the critical paths of the adder, we shall use the following list of assumptions.

1. All carry-skip circuitry is properly set at time zero.
2. Delay is additive across blocks.
3. The carry-out signal of a block is initially 0, switches exactly once and only if there is a carry-out from the block.
4. If the carry-in to block j is absorbed (that is, there is a bit position in block j which does not propagate a carry to the next bit) then the delay in determining the carry-out of block j is at most $R(x_j)$.

For an n-bit Manchester carry-skip adder composed of m blocks, the worst-case delay of the adder occurs when a carry is generated at the beginning of some block i, skips p blocks, and assimilates at the end of the block numbered $i+p+1$.

The design problem for an n-bit adder can now be stated as:

> Given the timing models for carry-generate $G(x)$, carry-ripple $R(x)$, and carry-skip $S(x)$, find a configuration $\vec{x} = \{x_1, x_2, \ldots, x_{m-1}, x_m\}$, where x_i is the size of block number i, that minimizes the critical path(s) of the adder, under the constraint that $n = \sum_{i=1}^{m} x_i$.

Since the carry-ripple and carry-generate delays are the same for both CMOS implementations, we shall use $G(x) = R(x)$ in our analysis. Of course, we have to consider the delay E introduced by the restoring buffer in both implementations. After factoring out the common constants, we obtain equations that describe the timing characteristics of the adder blocks for both implementations:

$$\begin{aligned} R(x) &= x^2 \\ S(x) &= a_1 x + a_2 \\ E &= \delta \end{aligned} \tag{8}$$

where a_1, a_2, and δ are *positive* constants.

Similar work was reported by Barnes and Oklobdzija in [2]. However, we believe our timing models for carry-ripple and carry-skip are more accurate than what they assumed. Precisely: our analysis shows that carry-skip time $S(x)$ for both models is *not* a constant but rather is linearly proportional to block size (Equation (8)).

3 Determining block sizes

Due to the discrete nature of the problem, determining the *optimal* configuration of the adder has involved extensive search techniques [2]. We shall introduce efficient techniques for determining *near-optimal*, and *optimal* block sizes.

3.1 Problem with existing approach

A note on related work: Guyot et al. have suggested a method to compute reasonably good block sizes for the case $R(x) = x, S(x) = a_1 x^2 + a_2$, and $E = 0$ [4]. Their method is based on the premise that the configuration is symmetric, i.e., $x_1 = x_m$, $x_2 = x_{m-1}$ etc. A distribution function $f(x_i)$ is sought such that the "life" of a carry is roughly the same between all pairs of blocks. This is a good idea, unfortunately, their technique fails to apply to our model Equation (8), as we shall demonstrate. Following their notation, consider a carry generated at the beginning of block i, which skips over subsequent blocks, and finally assimilates at the end of block j. The "life" of the carry is

$$x_i^2 + \delta + \sum_{q=i+1}^{j-1} (a_1 x_q + a_2 + \delta) + x_j^2. \tag{9}$$

A good approximation of $\sum_{q=i+1}^{j-1} (a_1 x_q + a_2 + \delta)$ is

$$\int_i^j (a_1 t + a_2 + \delta) dt.$$

We seek a distribution function f such that

$$f^2(z) + \delta + \int_z^y (a_1 f(t) + a_2 + \delta) dt + f^2(y) = constant.$$

Setting $z = 0$ and differentiating the above equation with respect to y gives

$$2f(y)f'(y) + a_1 f(y) + a_2 + \delta = 0, \tag{10}$$

and solving this differential equation yields a function

$$f(y) = -(a_2 + \delta)/a_1. \tag{11}$$

Since a_1, a_2, and δ are positive constants, the distribution of block sizes as suggested by $f(y)$ is unrealizable.

3.2 Our first approach: a linear time heuristic

We shall develop an $O(n)$ time heuristic method to obtain approximate solutions to this problem; an $O(n^3 log\ n)$ time algorithm for finding optimal solutions to this problem is presented in Section 4.

The heuristic is also based on the premise of symmetry. The strategy is to select one path, call it $\wp$, between a pair of blocks, pick the rest of block sizes so that the delays of other paths will not exceed $\wp$'s delay. Then, we minimize the delay of $\wp$.

Consider a configuration $\vec{x}$ having an even number of blocks. The life of a carry generated at block 1 and absorbed at the last block $m = 2w$ is

$$\begin{aligned} \tau_m &= 2x_1^2 + 2\sum_{i=2}^{w} (a_1 x_i + a_2) + (m-1)\delta \\ &= 2x_1^2 + 2a_1 \sum_{i=2}^{w} x_i + (m-2)a_2 + (m-1)\delta \\ &= 2x_1^2 - 2a_1 x_1 + a_1 n + (m-2)a_2 + (m-1)\delta \end{aligned} \tag{12}$$

We partial differentiate equation (12) with respect to x_1 to determine the minimum: $x_1 = a_1/2$. Equation (12) suggests that to reduce the delay of this path, we should:

1. pick $x_1 = \lceil a_1/2 \rceil$, and
2. minimize m subject to the constraint that no other path delay will exceed τ_m.

To ensure that no other path in the adder will have delay *greater* than τ_m, we select the sizes of the subsequent blocks x_2, x_3, ..., such that

$$\begin{aligned} R(x_1) + \delta + S(x_2) + \delta &\geq R(x_2) + \delta \\ R(x_1) + S(x_2) + \delta &\geq R(x_2) \end{aligned} \quad (13)$$

or in general

$$R(x_1) + \delta + \sum_{j=2}^{i+1}(S(x_j) + \delta) \geq R(x_{i+1}) + \delta, \quad (14)$$

or even simpler

$$R(x_i) + S(x_{i+1}) + \delta \geq R(x_{i+1}). \quad (15)$$

Substituting the carry-ripple and carry-skip functions into equation (15) gives,

$$x_{i+1} \leq (a_1 + \sqrt{a_1{}^2 + 4(x_i{}^2 + \delta + a_2)})/2 \quad i = 1, \ldots, w \quad (16)$$

which ensures that the critical path always originates from block 1 and the number of blocks is reduced.

A similar analysis can be carried out for a configuration having an odd number of blocks.

Example 1: Design a 24-bit adder, with timing model: $R(x) = x^2, S(x) = (3x + 2)$, and $\delta = 1$. Using $x_1 = 2$, and applying Equation (16), we obtain $x_2 = 4$, and $x_3 = 6$ successively. This is in fact an optimal configuration $\vec{x} = (2, 4, 6, 6, 4, 2)$, with maximum delay of 81.

Example 2: Design a 64-bit adder, with timing model: $R(x) = x^2, S(x) = (3x + 2)$, and $\delta = 1$. Using $x_1 = 2$, and applying Equation (16), we obtain the configuration

$$\vec{x} = (2, 4, 6, 7, 8, 5, 5, 8, 7, 6, 4, 2),$$

with maximum delay of 218, by assuming an even number of blocks. Likewise, by assuming an odd number of blocks, we obtain

$$\vec{x} = (2, 4, 6, 7, 8, 10, 8, 7, 6, 4, 2),$$

with maximum delay of 216.

4 A polynomial time algorithm for finding optimal solutions

This algorithm does not work for arbitrary $R(x)$, $G(x)$, and $S(x)$. But it provides optimal solutions under some reasonable assumptions that we stipulate below, together with the ones that were stated in Section 2.2.

Additional Assumptions:

1. $R(x)$ is a monotonic non-decreasing function in x with a non-negative derivative.
2. $S(x)$ is a linear function in x (i.e. $S(x) = a_1 x + a_2$).
3. $R(x) - S(x)$ is non-decreasing.
4. The inverses of $R(x)$ and $R(x) - S(x)$ both exist.
5. $R(2) - R(0) \geq S(x + 1) - S(x)$.
6. $R'(x) \geq S'(x)$ for $x \geq 2$.

All these assumptions are respected for both CMOS Manchester adders as shown in Figures 2 and 3, as well as for the constant skip model used in [4, 9].

We shall assume for simplicity that $R(x) = G(x)$ as was the case for the derived RC models. For a configuration $\vec{x}$, the delay of a carry generated in the first position of block i and absorbed in the last position of block j is denoted by,

$$D(\vec{x}, i, j) = R(x_i) + R(x_j) + \sum_{l=i+1}^{j-1} S(x_l).$$

The worst-case delay of $\vec{x}$ is given by

$$D(\vec{x}) = \max_{1 \leq i < j \leq m} D(\vec{x}, i, j).$$

Let $\mathcal{C}_n$ denote the set of all configurations of n bits. An optimal configuration of n bits is one with minimum worst-case delay over all possible configurations and the optimal worst-case delay is

$$\mathcal{D} = \min_{\vec{x} \in \mathcal{C}_n} D(\vec{x}).$$

Obtaining an optimal configuration of n bits for a given $R(x)$ and $S(x)$ might appear at first glance a difficult problem, since it involves a mini-max optimization over a possibly exponential search space. It does not satisfy the principle of optimality in the sense that a solution for $n + 1$ bits may be quite different from a solution for n bits. There is, however, a search strategy that allows us to ignore many non-optimal configurations. Given two positive integers a and b such that $a + b \leq n$, we can efficiently construct an optimal configuration whose worst-case delay is between a block of a bits and a block of b bits. The construction for a given pair a, b can be performed in $O(n \log n)$ time, resulting in an $O(n^3 \log n)$ time algorithm for finding the optimal configuration for n bits.

Let $\mathcal{C}_n(a, b)$ denote the set of all configurations of n bits with the worst-case delay between a block of a bits and a block of b bits. That is, if $\vec{x} \in \mathcal{C}_n(a, b)$, then $\exists i, j$ such that $x_i = a$, $x_j = b$ and $D(\vec{x}) = D(\vec{x}, i, j)$. Then

$$\mathcal{D} = \min_{\vec{x} \in \mathcal{C}_n} D(\vec{x}) = \min_{\substack{1 \leq a, b \leq n \\ a+b \leq n}} \{ \min_{\vec{x} \in \mathcal{C}_n(a,b)} D(\vec{x}) \}$$

since the union of all of the $\mathcal{C}_n(a, b)$'s is $\mathcal{C}_n$; an optimal configuration must have a worst-case delay between a pair of blocks whose sizes are between 1 and n. Note that the $\mathcal{C}_n(a, b)$'s are not necessarily disjoint since the worst-case delay might be exhibited by more than one pair of blocks.

Note also that we neither fix the number of blocks nor the location of the two blocks of known size. We only require that the worst case is between a block with a bits and a block with b bits.

Let i denote the index of the block with a bits and let j denote the index of the block with b bits. These indices i and j are used for reference; we do not know the exact value of $j - i$ apriori. The values of i and j will only be known when the construction is complete. Figure 8 shows a configuration in $\mathcal{C}_n(a, b)$. The blocks are indexed along the horizontal axis and the height of each column is the number of bits in the block. At the start only the number of bits in blocks i and j are known. The shaded regions must be filled with the remaining bits under the constraint

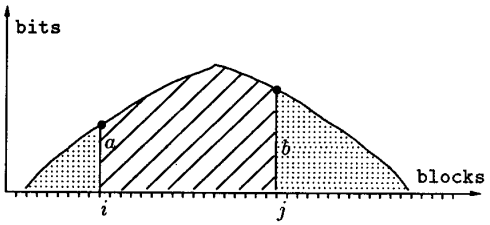

Figure 8: An element of $\mathcal{C}_n(a, b)$.

that the worst-case delay remain between i and j. These constraints are represented by the curves anchored at the points (i, a) and (j, b). This representation is similar to the one used in [4]. In our case, the curves must be calculated explicitly for each a and b due to the discrete nature of the problem.

Constructing $\mathcal{C}_n(a, b)$ amounts to minimizing the delay between blocks i and j subject to the constraint that the delay between all other pairs of blocks must not exceed the delay between blocks i and j. Clearly the delay between blocks i and j is minimized by placing as few bits as possible in the blocks between i and j; we want to minimize the number of bits which are placed in blocks between i and j (the slashed region in Figure 8). This amounts to maximizing the number of bits which can be placed to the left (respectively, to the right) of blocks i and j subject to the delay constraint (the dotted regions in Figure 8). The delay constraints can be computed solely with respect to a and b in most cases; it will be shown that the pathological cases in which this is not true and the algorithm fails to produce an element of $\mathcal{C}_n(a, b)$ can be ignored.

To compute the size of the blocks to the left of i (remember $x_i = a$) we proceed as follows. R^{-1} is the inverse of $R(x)$.

Procedure *Fill_End*$(a, i, R(x), S(x))$
$x_i \leftarrow a$
$k \leftarrow i$
while $(x_k > 0)$
$x_{k-1} \leftarrow \lfloor R^{-1}(R(x_i) - \sum_{s=k}^{i} S(x_s)) \rfloor$
$k \leftarrow k - 1$
end
End *Fill_End*

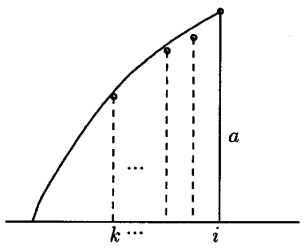

Figure 9: Maximizing the number of bits to the left of a.

Starting with block $i - 1$ and proceeding to the left as in Figure 9, the maximum number of bits is placed in each block subject to the constraint

$$R(x_k) + \sum_{s=k+1}^{i} S(x_s) \leq R(a). \tag{17}$$

Constraint (17) assures that $D(\vec{x}, k, h) \leq D(\vec{x}, i, h)$ for any block $h > i$. Note that the integer nature of the problem requires us to make x_k the largest integer satisfying the constraint and recompute the constraint for x_{k-1} with respect to x_i. With minor modifications, *Fill_End* is also used to determine the maximum number of bits and their configuration which can be placed to the right of x_j. This greedy algorithm maximizes the number of bits to the left of i under assumption 5, i.e. $R(2) - R(0) \geq S(x+1) - S(x)$ for all $x > 0$. In other words, adding one less bit to a block would not relax the constraint of a subsequent block enough so that 2 or more bits could be added. Constraint (17) is based on a, the number of bits and the number of blocks in between i and k; it is independent of how the bits are configured within these blocks. Suppose a non-greedy strategy could place more bits to the left of i while still satisfying constraint (17) for each block to the left of i. In order to increase the total number of bits, this strategy would have to sacrifice skipping d bits to absorb the cost of rippling through an extra $d + 1$ bits. If this were possible, it would happen for $d = 1$ since $S(x)$ is linear by assumption 2 and $R(x)$ grows at least as fast as $S(x)$ by assumption 6. But by assumption 5 it does not happen for $d = 1$.

Once we have computed the number of bits n_l and n_r which can be placed to the left and right of i and j respectively, the remaining bits $n_m = n - n_l - n_r - a - b$ must be configured in blocks in between i and j. Since by assumption 2, $S(x)$ is linear in x, minimizing the number of blocks in between i and j achieves the minimization of the $D(\vec{x}, i, j)$. Hence this minimization can be achieved by maximizing the number of bits in each block subject to the delay constraints with respect to x_i and x_j imposed by having $D(\vec{x}, i, j)$ as the worst case.

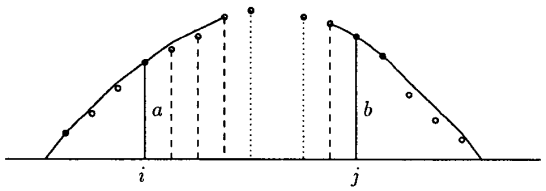

Figure 10: Filling in the blocks between i and j.

Specifically, each of the blocks k in between i and j must satisfy

$$R(x_i) + \sum_{s=i+1}^{k} S(x_s) \geq R(x_k)$$

and

$$R(x_k) \leq R(x_j) + \sum_{s=k}^{j-1} S(x_s).$$

As illustrated in Figure 10, the blocks between i and j are filled by calculating the maximum size of the next blocks with respect to both x_i and x_j and filling the smaller of the two blocks (ties are broken arbitrarily). If instead the larger of the two blocks were filled, the last block might fail to meet both constraints. The greedy strategy is optimal by assumptions 1 and 2 since placing fewer bits in any block cannot allow a greater number of bits to be placed in later blocks.

The algorithm for filling the blocks between i and j is given below. Let RS^{-1} denote the inverse of the function $R(x)-S(x)$. For the sake of simplicity, the algorithm below uses i and j to index the blocks even though, as mentioned, we do not know the values of i and j apriori. An implementation would need to use a linked list or separate arrays.

```
Procedure Fill_Middle(a, i, b, j, n_l, n_r, R(x), S(x))
  n_m ← n − n_l − n_r − a − b
  l ← i
  r ← j
  while (n_m > 0)
        nextL ← ⌊RS^{-1}(R(x_i) + Σ_{s=i+1}^{l} S(x_s))⌋
        nextR ← ⌊RS^{-1}(R(x_j) + Σ_{s=j-1}^{r} S(x_s))⌋
        if (n_m ≤ nextL) and (n_m ≤ nextR) then { l ← l + 1
                                                  x_l ← n_m
                                                  n_m ← 0 }
        else if (nextL < nextR) then { l ← l + 1
                                       x_l ← nextL
                                       n_m ← n_m − nextL }
        else { r ← r − 1
               x_r ← nextR
               n_m ← n_m − nextR }
  end
End Fill_Middle
```

To establish the correctness of this method we need to prove that the algorithm does in fact produce a configuration in which the worst-case delay is between blocks $x_i = a$ and $x_j = b$. Unfortunately, if $|a - b|$ is large enough, the algorithm may fail to produce a configuration in which the worst-case delay is between blocks i and j. Essentially, if b is much larger than a, in computing the maximum number of bits to the right of j the worst case of the constructed configuration may be between blocks j and k for some $k > j$. This situation is depicted in Figure 11. In this case the maximum number of bits to the right of x_j should not have been determined solely with respect to $x_j = b$. Fortunately, we are able to dismiss this pathological case by showing that it cannot lead to the optimal solution. We shall show that in this case there is always some other pair a', b' with $\max\{a', b'\} < \max\{a, b\}$ for which the optimal configuration has no greater delay. Hence the pair a and b can be ignored.

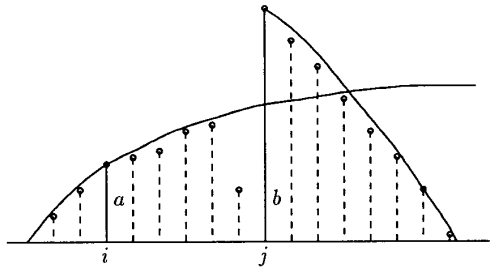

Figure 11: Algorithm may fail when $|a - b|$ is large.

The time complexity of both *Fill_End* and *Fill_Middle* is $O(n \log n)$. The $O(\log n)$ factor in the procedures stems from computing the inverses of $R(x)$ and $R(x) - S(x)$ by binary search. $R(x)$ and $S(x)$ are implemented as subroutines of the main program. Since there are $O(n^2)$ pairs of a, b to try, this gives an $O(n^3 \log n)$ time algorithm for finding the optimal worst-case delay configuration for n bits.

To establish that our algorithm indeed constructs the optimal element of $\mathcal{C}_n(a, b)$, we shall assume without loss of generality that $a \leq b$ since the delay computation is symmetric. By construction, *Fill_End* and *Fill_Middle* generate an optimal configuration satisfying the following inequalities; these inequalities must hold for any element of $\mathcal{C}_n(a, b)$.

$$x_i = a \tag{18}$$

$$x_j = b \tag{19}$$

$$\forall 1 \leq k < i, \quad R(x_k) + \sum_{s=k+1}^{i} S(x_s) \leq R(x_i) \tag{20}$$

$$\forall i < k < j, \quad R(x_k) \leq R(x_i) + \sum_{s=i+1}^{k} S(x_s) \tag{21}$$

$$\forall i < k < j, \quad R(x_k) \leq R(x_j) + \sum_{s=k}^{j-1} S(x_s) \tag{22}$$

$$\forall j < k \leq m, \quad R(x_k) + \sum_{s=j}^{k-1} S(x_s) \leq R(x_j) \tag{23}$$

Unfortunately, these constraints are not sufficient to guarantee that $\vec{x} \in \mathcal{C}_n(a,b)$. Theorem 1 provides the additional requirement. Lemma 2 guarantees that this condition is satisfied if at least one block of b or more bits is placed in between i and j. However, the condition must be checked explicitly when there is no such block added as is the case in Figure 11. The additional condition introduced by Theorem 1 ensures that the worst case is between blocks i and j, as opposed to between j and some block k to its right ($j < k$).

Theorem 1 If $\vec{x}$ is a configuration satisfying inequalities (18) through (23) for two block indices i and j, and

$$\forall h \geq 1,\ R(x_i) + \sum_{k=i+1}^{j-1} S(x_k) \geq \sum_{k=j+1}^{j+h-1} S(x_k) + R(x_{j+h}), \tag{24}$$

then the worst-case delay of $\vec{x}$ is between blocks i and j.

Proof: Suppose $\vec{x}$ is a configuration satisfying inequalities (18) through (24) for two block indices i and j. We establish

$$D(\vec{x},i,j) = D(\vec{x}) = \max_{f<g} D(\vec{x},f,g)$$

by showing that for any pair of blocks f and g,

$$D(\vec{x},f,g) \leq D(\vec{x},i,j).$$

There are several cases to consider according to the relative values of f,g with respect to i,j. If $f < j$ and $g > i$ then by inequalities (20) and (21) a carry generated at block f would arrive no later at block g than a carry generated at block i. Similarly by inequalities (22) and (23), if $g > i$ and $f < j$ then a carry generated in block f will be absorbed at the end of block g no earlier than it would be absorbed at the end of block j. In both of these cases $D(\vec{x},f,g)$ is bounded from above by $D(\vec{x},i,j)$. The remaining cases to consider are when $g \leq i$ (and hence $f < i$) or when $f \geq j$ (and hence $j < g$).

Case 1. ($g \leq i$) In this case, since the blocks only increase in size from g towards i, the delay would only increase if the carry were absorbed at block i rather than block g. Hence we need only consider the case where $g = i$. By inequality (20), we have

$$\begin{aligned} D(\vec{x},f,i) &= R(x_f) + \sum_{s=f+1}^{i-1} S(x_s) + R(x_i) \\ &= R(x_f) + \sum_{s=f+1}^{i} S(x_s) - S(x_i) + R(x_i) \\ &\leq R(x_i) - S(x_i) + R(x_i) \\ &\leq R(x_i) + R(x_i) \\ &\leq R(x_i) + R(x_j) \\ &\leq D(\vec{x},i,j). \end{aligned}$$

Case 2. ($f \geq j$) In this case, since the blocks only increase in size in moving left from f towards j, we need only consider the case where $f = j$. The inequality below follows from condition (24) with $h = g - j$.

$$\begin{aligned} D(\vec{x},j,g) &= R(x_j) + \sum_{s=j+1}^{g-1} S(x_s) + R(x_g) \\ &\leq R(x_j) + R(x_i) + \sum_{k=i+1}^{j-1} S(x_k) \\ &\leq D(\vec{x},i,j). \end{aligned}$$

□

Lemma 2 If $\vec{x}$ is a configuration satisfying inequalities (18) through (23) for two block indices i and j and

$$\exists k,\ i < k < j \ \text{ and } \ x_k \geq b,$$

then the worst-case delay of $\vec{x}$ is between blocks i and j.

Proof: Suppose the largest block added in between i and j is k and $x_k \geq b$. Then x_k satisfies inequality (21) which is the bound computed for it with respect to i. Subtracting $S(x_k)$ from both sides of inequality (21) gives us,

$$R(x_i) + \sum_{s=i+1}^{k-1} S(x_s) \geq R(x_k) - S(x_k).$$

Adding the skip of blocks k through $j-1$ to the left-hand-side does not alter this inequality,

$$R(x_i) + \sum_{s=i+1}^{j-1} S(x_s) \geq R(x_k) - S(x_k). \tag{25}$$

Since $x_k \geq b$ and $R(x) - S(x)$ is non-decreasing by assumption 3, we have

$$R(x_k) - S(x_k) \geq R(x_j) - S(x_j). \tag{26}$$

By rearranging inequality (23) we obtain

$$\forall h \geq 1,\ R(x_j) - S(x_j) \geq R(x_{j+h}) + \sum_{s=j+1}^{j+h-1} S(x_s). \tag{27}$$

Putting (25) through (27) together we obtain,

$$\forall h \geq 1,\ R(x_i) + \sum_{s=i+1}^{j-1} S(x_s) \geq R(x_{j+h}) + \sum_{s=j+1}^{j+h-1} S(x_s),$$

which by Theorem 1 ensures that the worst-case delay is between blocks i and j. □

Finally we must show that when condition (24) is not satisfied by the configuration constructed by *Fill_End* and *Fill_Middle*, we can ignore $\mathcal{C}_n(a,b)$ since there is always an equivalent or better solution in some $\mathcal{C}_n(a',b')$ where $\max\{a',b'\} < b$. By Lemma 2, if condition (24) is not satisfied by the configuration constructed by the algorithm, then no block was added in between i and j with b or more bits.

The next theorem shows that if *Fill_End* and *Fill_Middle* fail to produce a configuration satisfying condition (24), then we can ignore the pair a, b in our search for the optimal configuration.

Theorem 3 If $\vec{x}$ is the configuration constructed by procedures *Fill_End* and *Fill_Middle* and it does not satisfy condition (24), then there exist $a' < b$ and $b' < b$ such that

$$\min_{\vec{x} \in \mathcal{C}_n(a',b')} D(\vec{x}) \leq \min_{\vec{x} \in \mathcal{C}_n(a,b)} D(\vec{x}).$$

Sketch of Proof: If the configuration constructed by procedures *Fill_End* and *Fill_Middle* does not satisfy condition (24), there must be some $h \geq 1$ such that, the delay between blocks i and j is exceeded by the delay between blocks j and $j + h$. We can show that in this case $a < b$, since otherwise inequality (24) would have been satisfied.

Let $\vec{z}$ be an element of $\mathcal{C}_n(a, b)$ with minimum worst-case delay and the fewest number of blocks with exactly b bits. Then $\vec{z}$ must satisfy inequalities (18) through (23) as well as condition (24) for some pair of indices i' and j'. We can argue that there is no block with more than b bits in $\vec{z}$. We can also show that there must be some block k to the right of j' ($j' < k$), which could be increased by one bit without causing the delay of a carry generated at i' to be absorbed later than if it were absorbed at j'. This is true because block sizes which satisfied inequality (23) had to be reduced in $\vec{z}$ in order to satisfy inequality (24). Using this fact we modify $\vec{z}$ by moving one bit from block j' to block k to obtain a new configuration $\vec{y}$. Specifically, $\vec{y}$ is the configuration defined by,

$$y_l = \begin{cases} z_j - 1 & l = j' \\ z_k + 1 & l = k \\ z_l & l \neq j' \text{ and } l \neq k. \end{cases}$$

Using the assumptions about $R(x)$ and $S(x)$ stated at the beginning of this section, we can show that $D(\vec{y}) \leq D(\vec{z})$. Basically, since only block k increased in size in moving from $\vec{z}$ to $\vec{y}$, we need only to show that any carry involving block k in $\vec{y}$ is no greater than the carry between block i' and j' in $\vec{z}$; all other carries in $\vec{y}$ are bounded by their counterparts in $\vec{z}$.

The final step is to show that $\vec{y} \in \mathcal{C}_n(a', b')$ for $\max\{a', b'\} < b$. We know that $\vec{y}$ doesn't have any block of more than b bits since $\vec{z}$ didn't have any and block k in $\vec{z}$ had to have fewer than b bits. (In fact, we can show $z_k < b - 1$.) Suppose $\vec{y} \in \mathcal{C}_n(a', b')$. If $b' = b$ we can show that $\vec{y} \in \mathcal{C}_n(a, b)$, which is not possible since it has one fewer block of b bits than $\vec{z}$. The remaining case is $a' = b$. This block of b bits, say h, would have to be in between blocks i' and j', since $a < b$, $i' < h$ and $D(\vec{y}, h, g) \leq D(\vec{y}, i', g)$ for all blocks g. Again we establish $\vec{y} \in \mathcal{C}_n(a, b)$, which is impossible. So both a' and b' must be less than b. □

The buffer delay δ is not treated explicitly by this algorithm, but it is easily handled by adding δ to both $R(x)$ and $S(x)$. The resulting delays between any two blocks will be off by exactly one δ. This has no effect on the optimization since each pair is equally overcharged.

Example 3: We redo examples 1 and 2 from the previous section for model $R(x) = x^2$, $S(x) = 3x + 2$, and $\delta = 1$. Our algorithm reports $\vec{x} = (2, 4, 6, 6, 4, 2)$ with delay 81 as the optimal configuration for 24 bits, while for 64 bits it finds a better configuration: $(2, 4, 6, 8, 10, 10, 9, 7, 5, 3)$ with delay 215.

To illustrate the nonlinear growth in the size of the blocks, consider the optimal configuration for 200 bits of delay 651,

$$(1, 4, 6, 8, 9, 11, 13, 14, 16, 18, 15, 16, 15, 13, 11, 10, 8, 6, 4, 2)$$

which took a Sun 3/60 326 seconds of cpu time to compute.

5 Conclusions

We have analyzed two implementations of the Manchester carry-skip adder in CMOS with the *RC* timing model. Based on the *RC* timing model, we have developed efficient (polynomial) algorithms to determine near-optimal, as well as optimal block sizes for the one-level Manchester adder with variable carry-skip.

For one-level skip, the time complexity for both of the adders is $O(n)$.

6 References

[1] Pak K. Chan and Kevin Karplus. *Computing Signal Delay in General RC Networks by Tree-Link Partitioning.* In *Proceedings 26th ACM/IEEE Design Automation Conference*, Las Vegas, Nevada, June 1989.

[2] E.R. Barnes and V.G. Oklobdzija. New Scheme for VLSI Implementation of Fast ALU. *IBM Technical Disclosure Bulletin*, Vol. 28, No. 3, pp.1277-1282, August 1985.

[3] Gerrit A. Blaauw. Digital System Implementation. Prentice-Hall, pp.47-50, 1976.

[4] Alain Guyot, Bertrand Hochet, and Jean-Michel Muller. A Way to Build Efficient Carry-Skip Adders. *IEEE Transactions on Computers*, Vol.C-36(4):1144–1151, October 1987.

[5] T. Kilburn, D.B.G. Edwards, and D. Aspinall. A Parallel Arithmetic Unit Using a Saturated Transistor Fast-Carry Circuit. *Proc. IEE*, Pt.B, Vol.107:573–584, November 1960.

[6] M. Lehman and N. Burla. Skip Techniques for High-Speed Carry-Propagation in Binary Arithmetic Units. *IRE Transactions on Electronic Computers*, Vol.EC-10(4):691–698, December 1961.

[7] Stanislaw Majerski. On Determination of Optimal Distributions of Carry Skips in Adders. *IEEE Transactions on Electronic Computers*, Vol.EC-16(1):45–58, February 1967.

[8] Carver Mead and Lynn Conway. Introduction to VLSI Systems. Addison-Wesley, pp.150-152, 1980.

[9] Vojin G. Oklobdzija and Earl R. Barnes. Some Optimal Schemes for ALU Implementation in VLSI Technology. In *Proceedings 7th Symposium on Computer Arithmetic*, pp.2-8, June 1985.

[10] Michael Pomper, Wolfgang Beifuss, Karlheinrich Horninger, and Walter Kaschte. A 32-Bit Execution Unit in an Advanced NMOS Technology. *IEEE actions on Electronic Computers*, Vol.EC-16(1):45–58, February 1967.

[8] Carver Mead and Lynn Conway. Introduction to VLSI Systems. Addison-Wesley, pp.150-152, 1980.

[9] Vojin G. Oklobdzija and Earl R. Barnes. Some Optimal Schemes for ALU Implementation in VLSI Technology. In *Proceedings 7th Symposium on Computer Arithmetic*, pp.2-8, June 1985.

[10] Michael Pomper, Wolfgang Beifuss, Karlheinrich Horninger, and Walter Kaschte. A 32-Bit Execution Unit in an Advanced NMOS Technology. *IEEE Journal of Solid-State Circuits*, Vol.SC-17(3):533–538, June 1982.

[11] J. Rubinstein, Paul Penfield Jr., and Mark Horowitz. Signal Delay in *RC* Tree Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-2(3):202–211, July 1983.

[12] E. Soutschek, M. Pomper, and K. Horninger. PLA Versus Bit Slice: Comparison for a 32-bit ALU. *IEEE Journal of Solid-State Circuits*, Vol.SC-17(3):584–586, June 1982.

[13] Neil Weste and Kamran Eshraghian. Principles of CMOS Design: A Systems Perspective. Addison-Wesley, pp.322-325, 1985.