Speculative Completion for the Design of High-Performance Asynchronous Dynamic Adders

Steven M. Nowick* — Dept. of CS, Columbia Univ., NY, NY 10027, nowick@cs.columbia.edu

Kenneth Y. Yun† — Dept. of ECE, UC San Diego, La Jolla, CA 92093, kyy@paradise.ucsd.edu

Peter A. Beerel‡ — EE-Systems Dept., USC, Los Angeles, CA 90089, pabeerel@eiger.usc.edu

Ayoob E. Dooply — Dept. of ECE, UC San Diego, La Jolla, CA 92093, adooply@ucsd.edu

Abstract

This paper presents an in-depth case study in high-performance asynchronous adder design. A recent method, called "speculative completion", is used. This method uses single-rail bundled datapaths but also allows early completion. Five new dynamic designs are presented for Brent-Kung and Carry-Bypass adders. Furthermore, two new architectures are introduced, which target (i) small number addition, and (ii) hybrid operation. Initial SPICE simulation and statistical analysis show performance improvements up to 19% on random inputs and 14% on actual programs for 32-bit adders, and up to 29% on random inputs for 64-bit adders, over comparable synchronous designs.

1 Introduction

Asynchronous design is enjoying a resurgence, with a many recent technical and practical advances [1]. In principle, asynchronous systems promise several advantages over synchronous systems: (i) *low power,* since an asynchronous component computes only when necessary; (ii) *high performance,* since global clock distribution and synchronization can be avoided; and (iii) *scalability and ease of design*, since there are no global timing constraints.

The promise of high-performance datapaths is especially attractive. In principle, a number of components have *data-dependent behavior*: fast operation on certain inputs, and slower operation on other inputs. Therefore, following the *RISC* philosophy of "making the common case fast", asynchronous datapaths have the potential to outperform synchronous designs on average inputs.

In practice, though, this potential is often difficult to realize. Existing methods for asynchronous datapath design can incur significant performance overhead, undercutting the potential benefits. The goal of this paper is to design high-performance asynchronous datapath components, which are faster than synchronous designs and yet have low area overhead.

A number of approaches have been proposed to design asynchronous datapath components. Most fall into one of two categories, depending on how completion is determined: *bundled data* and *completion detection*.

A *bundled data* design uses a *worst-case model delay*, designed to exceed the longest path through the subsystem [6, 1]. This delay may be an inverter chain or a replicated portion of the critical path. This method has been widely used [4, 3, 2, 5]. The main advantage is that a standard synchronous (*i.e.,* non-hazard-free) *single-rail* implementation may be used, so implementations are easy to design, and have low power and limited area. However, the key disadvantage is that completion is fixed to *worst-case computation*, regardless of actual data inputs.[1]

A *completion detection* method [1, 7] detects when computation is actually completed. The datapath is typically implemented in *dual-rail*, where each bit is mapped to a pair of wires, which encode both the value and validity of the data. Different encoding schemes have been used, such as 4-phase *RZ* and 2-phase *LEDR* (see [1]), and the methods have been applied to a number of designs [8, 7]. In principle, this approach has the advantage that the datapath itself indicates when computation is actually completed. The key disadvantage, in many applications, is that a *completion detection network* is usually required, adding several gate

*This research was funded in part by an NSF CAREER Award MIP-9501880 and by an Alfred P. Sloan Research Fellowship.

†This work is funded in part by a NSF CAREER Award MIP-9625034, a gift from the Intel Corporation, and a Hellman Faculty Fellowship.

‡This research was funded in part by an NSF CAREER Award MIP-9502386, a gift from the Intel Corporation, and a research seed grant from the James H. Zumberge Faculty Research and Innovation Fund at USC.

[1] Unlike synchronous design, though, delay margins may be somewhat tighter, since timing constraints are localized.

delays between completion and its detection. Furthermore, the increased wiring and switching activity often result in much greater area and power consumption. An alternative scheme, *current-sensing completion detection*, avoids the detection network [9], but requires special current sensors and still requires a number of gate delays of overhead.

In recent work [12], we introduced a new method for designing asynchronous datapath components, called *speculative completion*. The method has many advantages of the bundled data approach, such as the use of a single-rail synchronous datapath. Unlike bundled data, though, *several* different matched delays are used: a worst-case model delay, and one or more *speculative* delays. Therefore, a component can operate at several possible speeds. A speculative delay allows early completion, and is disabled for worst-case data. However, unlike completion detection methods, early completion detection occurs in parallel with the datapath computation, *not* after computation is complete. Therefore, completion overhead is minimal.

As an initial case study, we presented a design for an asynchronous Brent-Kung adder [12]. This study was quite limited: only one gate-level design was presented. In addition, though careful gate-level analysis was included, we included no SPICE simulations. Our focus was only on static CMOS design, and one particular component of our design ("modified sum generation") was complex.

The contribution of this paper is a detailed case study of the design of high-performance asynchronous adders, using speculative completion. The focus is on dynamic implementations. Detailed SPICE simulations are provided for 5 designs, including both Brent-Kung and Carry-Bypass adders.

We first show that the use of dynamic logic can simplify the design of speculative adders. Both 2-speed and 3-speed designs are presented, for 32- and 64-bit addition.

We then introduce two new variant architectures. The first is designed to handle *addition of small numbers*, and allows very early completion. Small-number addition is an important special case, which arises in two common processor applications: (i) sign-extended addition, and (ii) non-random input distributions. Sign-extension of an operand commonly occurs in processors during branch target address calculation (for conditional branches) and effective address calculation (for memory data transfers). Non-random input distributions occur when running code sequences for actual programs. In this case, operands may be statistically skewed towards small numbers. We introduce the architecture, and apply it to Brent-Kung adders.

The second architecture is a *hybrid design*, which combines speculative completion (for early cases) with completion sensing (for other cases). The goal is to avoid completion sensing overhead for early cases, but obtain the benefit of variable completion sensing for slower cases. We introduce the hybrid architecture, and apply it to a carry-bypass adder.

Finally, SPICE analysis and detailed experimental performance evaluations of the adders are presented, assuming both random and non-random input distributions. Non-random input distributions were obtained from an ARM simulator, evaluated over a small set of programs (*e.g.,* espresso) and benchmarks (*e.g.,* dhrystone). Initial results indicate performance improvements ranging up to 19% on random inputs, and 14% on experimental inputs, for 32-bit addition, and 29% on random inputs for 64-bit addition, over comparable *synchronous* adders.

The case studies in this paper are meant to demonstrate the viability of this approach for high-performance design. It is important to note that speculative completion is not limited to Brent-Kung or Carry-Bypass adders. It can be applied to other adders, as well as multipliers and other components which exhibit data-dependent operation.

2 Background: Speculative Completion

This section reviews the basic speculative completion architecture, as presented in [12].

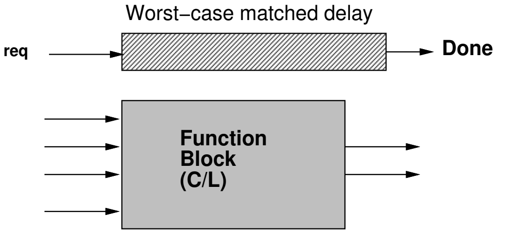
2.1 Basic Architecture

A standard single-rail bundled datapath is shown in Figure 1(a). The function block can be implemented using synchronous (*i.e.,* non-hazard-free) single-rail logic. A single *model delay* is used, with input req and output ack. This delay receives a request, req, when data inputs are valid, and will produce an ack only after the function outputs are valid. This matched delay must be slower than the function block under all physical conditions and all data inputs.

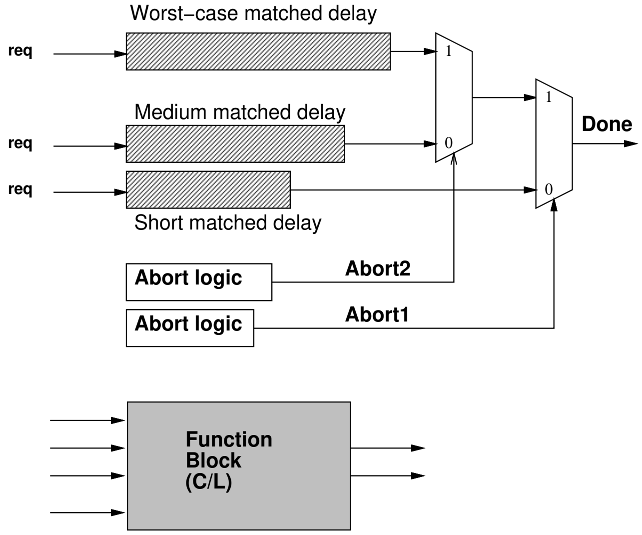
Figure 1(b) shows the basic architecture of our speculative completion datapath. There are three key features. First, we use **multiple model delays**: one for worst-case and the remaining ones for speculative completion. These speculative delays allow different speeds of early completion. For example, in a ripple-carry adder, an "average-case" delay might be used if adder inputs result in short carry chains; a "best-case" delay might be used if there is no carry chain (*e.g.,* if an operand is 0).

Second, an **abort detection network** is associated with each speculative delay. The network determines if the coresponding speculative completion must be aborted, due to worst-case data. Abort detection is computed *in parallel* with datapath computation. The abort signal is allowed to glitch. The only timing requirement is that it become stable and valid faster than the speculative delay.

Interestingly, the abort detection network does not need to detect the exact conditions for abort. Instead, the network can be simplified, to *safely approximate* the abort conditions. In particular, the abort detection network must detect all worst-case data, where abort is required. However, it



(a) Block diagram of standard bundled datapath



(b) Block diagram of speculative completion datapath

Figure 1. Comparison of (a) standard bundled datapath, and (b) speculative completion datapath

may also abort for some “best-case” inputs, where abort is unnecessary. The only impact of an unnecessary abort is to produce a late completion signal. Safe approximation of abort conditions can be used to simplify the abort detection logic.

The third feature, **modified result logic**, is not visible in Figure 1(b). With speculative completion, early completion is allowed when results can be produced early. In practice, though, some datapaths do not allow early generation of results. For example, in certain adder designs (see below), even if all carries are computed early, these carries must pass through several levels of logic before producing the correct sum. Therefore, *bypass logic* is required, to allow the sum to be generated using these early carries. Further details appear in the next subsection.

2.2 A Preliminary Gate-Level Study: Brent-Kung Adder

We now review our previous case study [12]: the design of Brent-Kung 32-bit binary lookahead carry (*BLC*) adder using speculative completion. The design was gate-level only; no simulations were presented. The focus of the study was on static implementations.

2.2.1 BLC Adder Design

A parallel 32-bit carry-lookahead adder of Brent and Kung [10, 11] is shown in Figure 2. This adder uses a bit-wise, or binary, lookahead carry (BLC) method. In a CMOS implementation of this adder, the stack depth of each gate is 2, and the gate fanout load is usually 2. The design is amenable to regular layout.

The 32-bit adder produces all propagate ($\overline{p}$) and generate ($\overline{g}$) signals in Level-0 and produces a sum in Level-6. The critical path from input to output is therefore 7 gate delays. Between Level-0 and Level-6, the adder computes the cumulative P and G values in parallel for each of the 32 bit slices. Specifically, Level-1 computes all 2-bit P and G values (where $P_i^1 = p_i \cdot p_{i-1}$ and $G_i^1 = g_i + p_i \cdot g_{i-1}$), Level-2 computes all 4-bit values (where $P_i^2 = P_i^1 \cdot P_{i-2}^1$ and $G_i^2 = G_i^1 + P_i^1 \cdot G_{i-2}^1$), and so on. In Level-6, the ith sum bit, s_i, is computed as the XOR of propagate bit p_i (taken from Level-0), and the final generate bit (or “carry-out”) G_{i-1}^5 of the preceding stage (taken from Level-5).

2.2.2 Speculative Adder Design

The speculative adder uses the same basic datapath, but with several modifications. We review the three components.

Completion Network

Figure 3 shows a block diagram of our speculative completion network. For simplicity, inverter chains are used for model delays, but replicated portions of the critical path can be used instead. In this figure, each inverter delay roughly corresponds to the delay of one level in the BLC adder. There are two model delay paths. The worst-case delay path has 7 gate delays. The speculative delay path has only 5 gate delays, and applies to cases where all final generate values are *available in Level-3* (*i.e.,* no useful computation occurs in Level-4 and Level-5).

Abort Detection Network

The key component of the design is the abort detection network, which generates the abort signal. By early completion, we mean that all final generate signals are available in Level-3: no further changes occur on generate signals in Level-4 or Level-5. By late completion, we mean that some final generate signal is not available in Level-3: it is computed in Level-4 or Level-5 (*i.e.,* differs from its Level-3 value).

In [12], a necessary condition for late completion was presented:

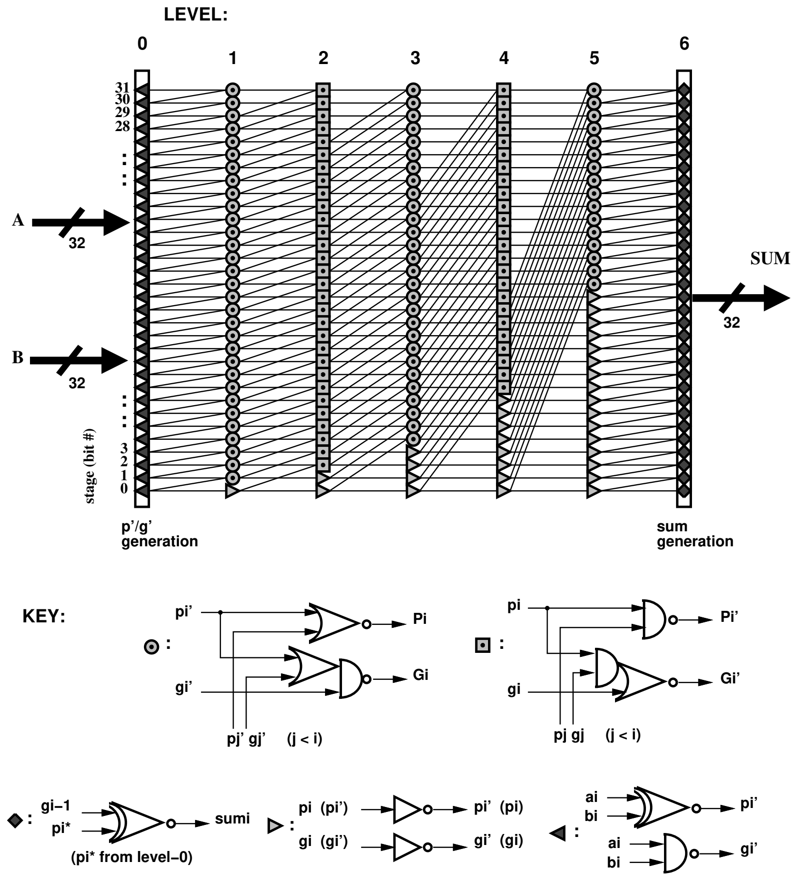


Figure 2. 32-bit Brent-Kung Adder

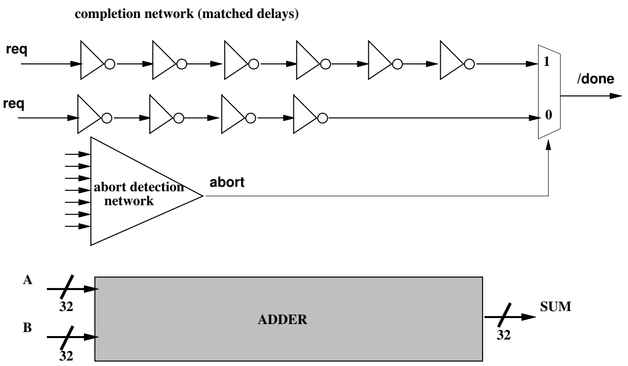


Figure 3. Block Diagram: adder with speculative completion

Condition 2.1. *Late completion can only occur if there exists a run of 8 consecutive Level-0 propagate signals.* □

The result is justified as follows. At the nth level, a generate function of the ith stage is computed as: $G_i^n = G_i^{n-1} + P_i^{n-1} G_j^{n-1}$, where $j = i - 2^{n-1}$ (ignoring the alternating inversions in the actual implementation). Clearly, G_i^n is the *same* as the generate of the preceding level, G_i^{n-1} *if* the propagate term, P_i^{n-1}, is 0. For the given detection, $n = 4$, so *each Level-4 generate signal is the same as the corresponding Level-3 generate signal if each Level-3 propagate signal is 0.* Each Level-3 propagate signal is effectively the product of a run of 8 consecutive Level-0 propagate signals. Such a condition is called an *8-p run*. Therefore, the goal of the abort detection network is to detect any 8-p run, and abort if one occurs.

For efficiency, this condition is *further safely approximated,* to produce simpler networks. As an example, consider the product $c = p_6\, p_7\, p_8$. This product *covers*, or detects, the 8-p run from p_3 to p_{10}. If the run occurs, then $c = 1$. However, if the run does not occur, then c may or may not be set to 0. The use of c simplifies detection, and detection is "safely approximate": c is never 0 when an 8-p run occurs.

In general, a product covers a *set* of 8-p runs. For example, c covers the 8-p runs from $1 - 8$ through $6 - 13$. To design an abort detection network, products are selected, each of which detects a set of 8-p runs. The abort detection network is constructed out of a sum of such products which, together, cover *all* possible 8-p runs. If any 8-p run occurs, the network will detect it.

A number of different abort detection networks can be used. Each implementation uses a different safe approximation to exact abort detection.

3-Literal Products. Each product contains a run of 3 p-signals (in Level-0). The network contains 5 products; it is given by equation: $p_5p_6p_7 + p_{11}p_{12}p_{13} + p_{17}p_{18}p_{19} + p_{23}p_{24}p_{25} + p_{29}p_{30}p_{31}$. Product $p_5p_6p_7$ covers the 8-p runs from stage: 0 through 7, 1 through 8, . . ., 5 through 12. That is, if any of these runs occurs, this product will be 1. Similarly, the remaining four products cover the other runs.

4-Literal Products. Each product contains a run of 4 p-signals; there are 5 products. The sum-of-products equation is: $p_4p_5p_6p_7 + p_9p_{10}p_{11}p_{12} + p_{14}p_{15}p_{16}p_{17} + p_{19}p_{20}p_{21}p_{22} + p_{24}p_{25}p_{26}p_{27}$. Each product covers fewer 8-p runs than in the preceding 3-literal product implementation, though the same total number of products is used.

5-Literal Products. Each product contains a run of 5 p-signals; there are 7 products. The sum-of-products equation is: $p_3p_4p_5p_6p_7 + p_7p_8p_9p_{10}p_{11} + p_{11}p_{12}p_{13}p_{14}p_{15} + p_{15}p_{16}p_{17}p_{18}p_{19} + p_{19}p_{20}p_{21}p_{22}p_{23} + p_{23}p_{24}p_{25}p_{26}p_{27} + p_{27}p_{28}p_{29}p_{30}p_{31}$. Note that in this case, adjacent products overlap; that is, they have a literal in common.

Alternative *augmented abort networks* can be designed, which use "kill" signals as well (see [12]).

The abort detection network is allowed to have hazards. The only requirement is that it produce a stable and valid result faster than the speculative delay. Because this network is critical, the designer must optimize its performance to meet this timing requirement.

Modified Sum Generation

The final component of the design is sum generation. In a basic Brent-Kung adder, even if all carries are computed early, an early sum cannot be generated (see Figure 4(a)). The problem is that each sum bit, sum_i, uses a generate signal only from *Level-5:* $sum_i = p_i \oplus G^5_{i-1}$. Therefore, *bypass logic* is needed, to allow the sum to use a Level-3 generate signal: G^3_{i-1}.

A gate-level solution is shown in Figure 4(b). The signals are ordered for fast completion using G^3_{i-1} and slower completion (with additional overhead) using G^5_{i-1}. The *late-en* signals are described in more detail in [12]. Basically, a late-enable signal is the output of an abort detection product, which covers this sum bit. Each *late-en* signal is broadcast to the sum bits which it covers.

This solution is fairly complex. One reason is that, in

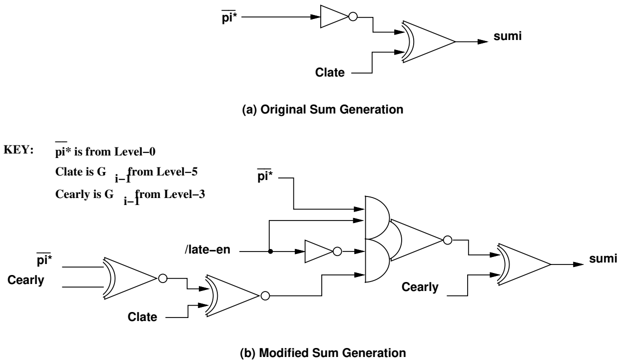


Figure 4. Sum Generation for Brent-Kung Adder

a static CMOS implementation, internal nodes are never reset, so their state is in general unknown. During early completion, once Level-3 G signals are valid and stable, the goal is to use them for early sum generation. Unfortunately, the values of *Level-5* G signals are unknown at this point. Therefore, complex sum generation logic is needed, to insure that a valid early sum is produced, using Level-3 G signals, regardless of the values on Level-5 G signals.

3 Basic Dynamic Brent-Kung Adders

We now introduce the first class of new speculative designs: basic dynamic implementations of Brent-Kung adders. Dynamic logic allows two key improvements: (i) greatly simplified sum generation; (ii) fast abort detection logic. We present designs for (i) 32+32 bit addition, using 1 speculative delay (*i.e.,* 2-speed), and (ii) 64+64 bit addition, using 2 speculative delays (*i.e.,* 3-speed).

3.1 Basic Dynamic P/G Cell.

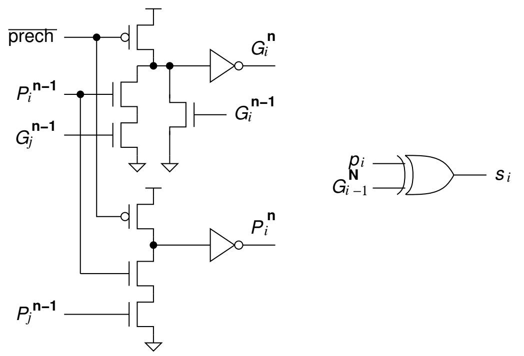


Figure 5. Basic dynamic cell: Brent-Kung adder

A basic dynamic cell is shown in Figure 5. The cell is used for P_i/G_i generation in Level-1 through Level-5. The static implementation alternated between P_i/G_i and $\overline{P_i}/\overline{G_i}$ in adjacent levels. In contrast, the dynamic implementation uses inverters, so it produces P/G at each level. The initial Level-0 p_i/g_i values are produced using dynamic XOR and AND gates, respectively (not shown).

3.2 Dynamic Adder Design: Overview.

Completion Network.

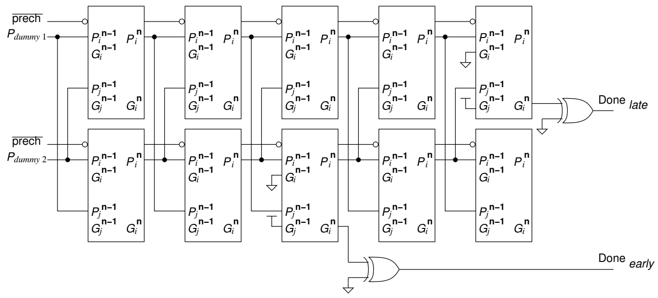


Figure 6. Basic completion network: Brent-Kung adder

A matched completion network for a 32+32 bit dynamic Brent-Kung adder can easily be built. Figure 6 shows two matched delay paths: (i) a speculative path, assuming Level-3 G signals are used for sum, producing $Done_{early}$; and (ii) the default path, assuming Level-5 G signals are used for sum, producing $Done_{late}$.

The matched delay paths consist of replicated basic cells for P/G generation. At the left, initial signals, P_{dummy1} and P_{dummy2} are set to 1 in Level-0. As a result, in each subsequent level, the P_i output becomes 1, in turn. In the final stage in each path (Level-3 in speculative, Level-5 in default), the G_j inputs are tied to 1, and G_i inputs are tied to 0, producing a final G_i output of 1 only after input $P_i = 1$ arrives. These signals are each fed into an XOR, to match the sum generation logic. Finally, the resulting $Done_{early}$ and $Done_{late}$ signals are fed into a MUX (not shown in the figure) which is controlled by the abort detection network. The resulting completion network contributes little area overhead to the entire adder.

Abort Detection Network.

A dynamic implementation of an abort detection network is shown in Figure 7. This particular network is the *4-literal/5-product network* described in the previous section. A similar implementation can be used for other networks.

The network has only 2 levels of logic (ignoring inverters), and pull-down stack depth is 2. The network is small, fast and has a low abort rate. The network allows early completion (*i.e.,* no abort) on 72% of random inputs. As shown in Section 6, even a more complex abort network (a

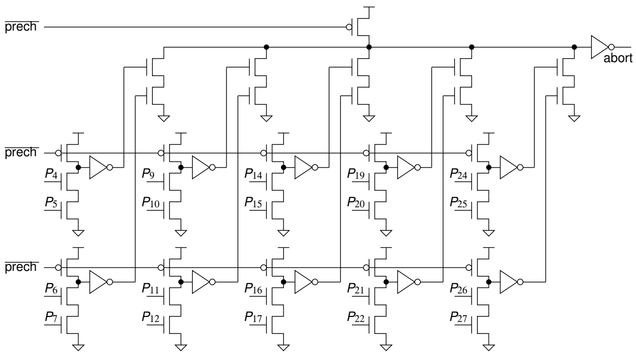
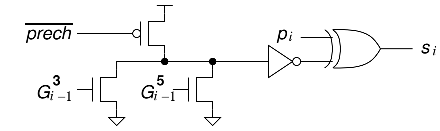


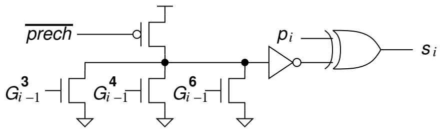
Figure 7. A dynamic abort detection network: Brent-Kung adder

5-literal/7-product network) easily meets the timing requirements (determines abort in less than 1ns, much faster than the speculative delay path).

Modified Sum Generation.



(a) Modified Sum Generation: 2-speed adder



(b) Modified Sum Generation: 3-speed adder

Figure 8. Dynamic modified-sum generation: Brent-Kung adder

Dynamic logic allows a greatly simplified design of the sum logic. In the static design of the previous section, modified sum logic was complicated. The problem was that internal nodes are not reset and, therefore, their state is in general unknown. Therefore, complex modified sum generation logic was needed, to produce a valid early sum result. In addition, *late-enable* signals had to be distributed to the different sum modules.

In contrast, with dynamic logic, all nodes are reset during the precharge phase, so values of internal nodes are known. The new dynamic implementation for modified sum logic is shown in Figure 8(a). The ith sum is given by: $s_i = p_i \oplus (G^3_{i-1} + G^5_{i-1})$. Here, G^3_{i-1} is the early (Level-3) carryout from the $i-1$ stage, G^5_{i-1} is the late (Level-5) carryout from the $i-1$ stage, and p_i is the ith Level-0 propagate bit. No *late-en* signal need to be distributed.

This scheme produces a correct sum in every case. If completion is late (*i.e.,* abort occurs), then either (i) G^5_{i-1} and G^3_{i-1} are both 1, (ii) G^5_{i-1} and G^3_{i-1} are both 0, or (iii) G^5_{i-1} is 1 and G^3_{i-1} is 0. The remaining case, G^5_{i-1} is 0 and G^3_{i-1} is 1, cannot occur, since G^5_{i-1} is a positive unate function of G^3_{i-1}. In all cases, $G^5_{i-1} + G^3_{i-1} = G^5_{i-1}$, so the correct late sum is produced. If completion is early (*i.e.,* no abort occurs), then case (iii) cannot occur, otherwise an abort would have occurred. If $G^3_{i-1} = 1$, then $G^5_{i-1} + G^3_{i-1} = 1$ as desired, regardless of whether G^5_{i-1} has had time to be set to 1. Similarly, if $G^3_{i-1} = 0$, then $G^5_{i-1} + G^3_{i-1} = 0$.

The above scheme extends naturally when there are multiple speculative delays: each appropriate early G signal is fed into the OR gate, as shown in Figure 8(b).

3.3 Adder Examples.

In the Results section, we will consider two basic dynamic Brent-Kung adders.

32+32 BK Adder *(2-speed).* Level-0 is p/g generation, Level-1 through Level-5 is P/G generation, and Level-6 is sum generation. The 5-literal/7-product abort network, described above, is used. The adder operates at 2 speeds. There is one speculative path, which allows early completion after Level-3 G signals have been used for sum generation.

64+64 BK Adder *(3-speed).* Level-0 is p/g generation, Level-1 through Level-6 is P/G generation, and Level-7 is sum generation. In this case, the adder operates at 3 speeds. There are two speculative delay paths, allowing (i) very early completion after Level-3 G signals have been used for sum generation, or (ii) early completion after Level-4 G signals have been used for sum generation.

The very early abort detection network detects all 8-p runs using 12 products, each with 4 literals: $p_4p_5p_6p_7 + p_9p_{10}p_{11}p_{12} + p_{14}p_{15}p_{16}p_{17} + p_{19}p_{20}p_{21}p_{22} + p_{24}p_{25}p_{26}p_{27} + p_{29}p_{30}p_{31}p_{32} + p_{34}p_{35}p_{36}p_{37} + p_{39}p_{40}p_{41}p_{42} + p_{44}p_{45}p_{46}p_{47} + p_{49}p_{50}p_{51}p_{52} + p_{54}p_{55}p_{56}p_{57} + p_{59}p_{60}p_{61}p_{62}$. The early abort detection network detects all 16-p runs using 6 products, each with 8 literals: $p_8p_9p_{10}p_{11}p_{12}p_{13}p_{14}p_{15} + p_{17}p_{18}p_{19}p_{20}p_{21}p_{22}p_{23}p_{24} + p_{26}p_{27}p_{28}p_{29}p_{30}p_{31}p_{32}p_{33} + p_{35}p_{36}p_{37}p_{38}p_{39}p_{40}p_{41}p_{42} + p_{44}p_{45}p_{46}p_{47}p_{48}p_{49}p_{50}p_{51} + p_{53}p_{54}p_{55}p_{56}p_{57}p_{58}p_{59}p_{60}$.

4 Handling Small Numbers

We now introduce two variant architectures, to handle addition of small numbers.

Small-number addition is an important special case, arising in several processor applications. By a small number, we mean a number with small magnitude, either positive or negative. Our focus will be on additions, $A + B$, where one particular operand (say B) is, or may be, small. In contrast, A may be large.

The goal of this work is to allow *very early* completion when handling small numbers. Specifically, for a 32+32 bit Brent-Kung adder, our goal is to produce an early sum *using Level-2 G signals*. In contrast, the basic dynamic 32-bit Brent-Kung adder in the previous section used Level-3 G signals for early sum.

Small-number addition occurs in two common applications: (i) *sign-extension* and (ii) *non-random input distributions*. Sign-extension of an operand, from, say, 16- to 32-bits, often occurs in RISC processors during branch target address calculation (for conditional branches) and effective address calculation (for memory data transfers). Non-random input distributions occur in actual code sequences for real programs. In particular, even in ALU *add* operations, without sign-extension, actual operands may be statistically skewed towards small numbers (see also [13]).

In each case, very early completion is often possible, since carry chains are often short. However, there are two major problems in applying a basic speculative architecture for these cases. First, abort detection logic becomes quite complex, since many short carry chains must be detected. At the same time, abort detection logic must be even faster, since a very early speculative delay is used.

We now describe two variants of the speculative architecture, to handle sign-extension and non-random inputs.

4.1 Sign-Extension

The first design is a 32+32 bit Brent-Kung adder, where the second operand, B, is a 16-bit sign-extended integer. The goal is to allow very early completion, *i.e.,* using Level-2 G signals for the sum.

Consider two operands, $A = a_{31}a_{30}\ldots a_1a_0$ and $B = b_{31}b_{30}\ldots b_1b_0$, where B is sign-extended 16-bit number; that is, $b_{31} = b_{30} = \ldots = b_{16} = b_{15}$. We shall refer to bits 15 to 31 as the *upper bits*, and to bits 0 to 14 as the *lower bits*. Since the upper bits of B are identical (all 0 or all 1), abort detection can be greatly simplified.

Our basic strategy is to use *partial abort detection*: we detect abort conditions only in the lower bits (roughly). However, the problem with this approach is that long carries in upper bits are still possible! This problem is addressed in the sequel.

Complete Abort Detection.

A *complete abort detection network* could be used, to detect all long carries. To allow very early completion, using Level-2 G signals, it is sufficient to check for any 4-p run, *i.e.,* run of 4 consecutive Level-0 propagate signals, $p_i \ldots p_{i+3}$. If no 4-p run occurs, then no late completion is necessary, as a corollary of Condition 2.1.

A complete abort detection network must detect every 4-p run, from $0-3$ to $28-31$. For example, using 3-literal products: product $p_1p_2p_3$ detects two 4-p runs, $0-3$ and $1-4$; $p_3p_4p_5$ detects two 4-p runs, $2-5$ and $3-6$; etc. The resulting network has 15 products, and detects all 4-p runs.

Partial Abort Detection.
A better alternative is to use a *partial abort detection network.* The network is much simpler; it only detects the 4-p runs from $0-3$ through $15-18$. The implementation is shown in Figure 9.

To prove that this partial network is sufficent, two cases must be considered, depending on whether the sign-extended operand, B, is positive or negative.

Case I: B Is Positive.

In this case, the key observation is that each upper bit, i ($i = 15\ldots31$), is either a *propagate (p)* bit (*i.e.,* $p_i = a_i \oplus b_i = 1$) or a *kill (k)* bit (*i.e.,* $k_i = \overline{a_i} \cdot \overline{b_i} = 1$). No upper *generate (g)* bit (*i.e.,* $g_i = a_i \cdot b_i$) can occur, since bits $b_{15} - b_{31}$ are all 0.

We now show that *only* 4-p runs up to $15-18$ need to be detected. Consider the run, $15-18$. This is a *crossover run*: it is the lowest 4-p run that contains only sign-extension bits of B. In this case, bits $15-18$ can have only p or k values.

Suppose $15-18$ is a 4-p run (*i.e.*, contains all p values). In this case, an abort is required, since a long carry chain (length ≥ 4) through bit 19 may occur, resulting in a late sum.

Alternatively, suppose $15-18$ is not a 4-p run. In this case, no carry chain is possible in the higher bits. In particular, some bit $j \in 15\ldots18$ is not a propagate (p) bit, so it must be a kill (k) bit. This bit, j, effectively *kills* any carry into the next bit, 19. As a result, no carry out will occur in *any* higher bit, since these bits are either p or k, and not g. Therefore, no carry chain occurs in the upper bits, so no higher 4-p runs ($16-19\ldots28-31$) need to be detected. The partial abort detection network can safely be used.

Case II: B Is Negative.

In this case, the key observation is that each upper bit, i ($i = 15\ldots31$), is now either a *propagate* bit or a *generate* bit. No upper *kill* bit can occur, since bits $b_{15} - b_{31}$ are all 1.

This case is dual to the positive case, but there is a subtle difference: long carry chains can now be *generated* in the upper bits! For example, suppose a_{20} is 1, and bits a_{21} through a_{28} are all 0. Bit 20 is a generate bit (since b_{20} is 1), while bits $21-28$ are propagate bits forming an 8-p run (since $b_{21} - b_{28}$ are all 1). Therefore, a carry chain of length 8 occurs, generated in bit 20. These carry chains can cause late changes in the upper sum bits. Our goal is to *avoid detecting* these long upper-bit carry chains, yet still produce a correct early sum.

We now prove that only 4-p runs up to $15-18$ need to be detected. Again, consider the crossover run, $15-18$. Here, bits $15-18$ can have only p or g values.

Suppose $15-18$ is a 4-p run (*i.e.,* all p values). In this case, an abort is required, since a long carry chain into bit 19 may, or may not, occur. The abort is required, since the value of sum_{19} cannot safely be resolved during early completion: it depends on whether the carryin to 15 actually occurs.

Alternatively, suppose $15-18$ is not a 4-p run. We show that every higher bit now has a carryout. In this case, some bit $j \in 15\ldots18$ is not a propagate (p) bit, so it must be a generate (g) bit. This bit, j, effectively *generates* a carry into the next bit, 19. Since each upper bit is either p or g, this carry initiates a carry chain, insuring that every higher bit, $i > j$ (whether p or g) produces a carryout.

As an example, suppose $j = 17$, the upper bits 23 and 28 are g, and the remaining bits $18-22$, $24-27$ and $29-31$ are all p. Here, 17 is g, initiating a carry chain, of length 5, through the p-run from $17-22$, and insuring a carryout of each of these bits. The remaining g bits, 23 and 28, already initiate carry chains into p-runs $24-27$ and $29-31$, respectively. Therefore, *every upper bit produces a carryout.*

In this case, long carry chains in upper bits can occur. And yet, this condition of "all-carryouts" in upper bits can be used to insure a correct early sum, in spite of the long carry chains.

Modified Upper Sum Generation.
The solution is to modify the upper sum bits. We first show that, for each upper bit i, a_i is the correct sum_i for early completion, and $p_i \oplus G^5_{i-1}$ is the correct sum_i for late completion. This result is justified below.

In Case I (B is positive), if there is no abort, we showed that there is no carryout from any upper bit i, $i \geq 18$. Therefore, an upper bit sum, sum_i, $i > 18$, is:

$$p_i \oplus carryout_{i-1} = p_i \oplus 0 = p_i = a_i \oplus b_i = a_i \oplus 0 = a_i.$$

In Case II (B is negative), if there is no abort, we showed that there is always a carryout from every upper bit i, $i \geq 18$. In this case, an upper bit sum, sum_i, $i > 18$, is:

$$p_i \oplus carryout_{i-1} = p_i \oplus 1 = \overline{p_i} = a_i \oplus \overline{b_i} = a_i \oplus \overline{1} = a_i \oplus 0 = a_i.$$

In each case, if there is no abort, the ith upper sum bit is $sum_i = a_i$. This result holds, regardless of long carry chains in the upper bits (in Case II).

Based on this result, *modified sum logic* for the upper sum bits sum_i, $i \geq 19$, must be designed. Figure 10 shows our new implementation. Each upper bit is implemented as: $sum_i = p_i \oplus (G^5_{i-1} + b_{15}\overline{Z})$. Here, Z is the $abort$ signal, and b_{15} is the sign bit of the 16-bit operand.[2] The AND of signals $\overline{Z}$ and b_{15} is broadcast to each of these upper sum bits. If there is no abort, these upper sum bits generate the correct result, a_i, quickly, even in Case II (B is negative) where there may be long carry chains in the upper bits.

To see that this upper sum logic is correct, consider the two cases. In Case I, B is positive, so b_{15} is 0. Therefore, $sum_i = p_i \oplus (G^5_{i-1} + 0) = p_i \oplus G^5_{i-1}$. For early completion,

[2]A faster alternative is to use product, $p_{15}p_{16}p_{17}p_{18}$, as the Z signal. This product detects the cross-over run, $15-18$, and can safely replace $abort$, for the sign-extension case. However, timing constraints were easily met using $abort$ as Z.

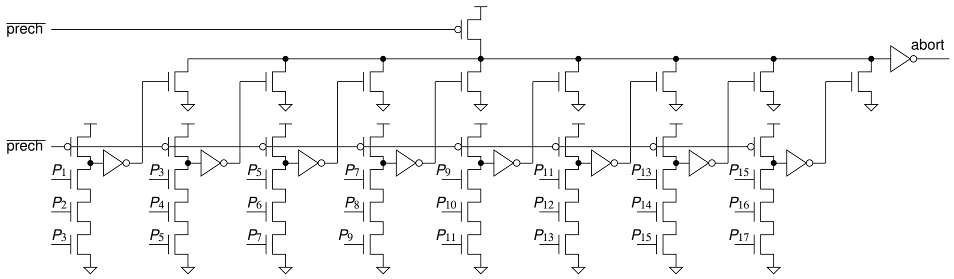


Figure 9. Partial Abort Detection for Sign-Extension: Brent-Kung Adder

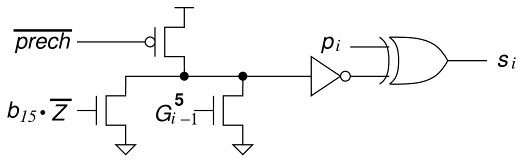


Figure 10. Modified Upper Sum Bit for Sign-Extension (bit i=19-31): Brent-Kung Adder

there are no upper carries, so G^5_{i-1} remains at 0, and $sum_i = p_i \oplus 0 = p_i = a_i \oplus b_i = a_i \oplus 0 = a_i$. For late completion, $sum_i = p_i \oplus G^5_{i-1}$, as desired. In Case II, B is negative, so b_{15} is 1. For early completion, $Z = 0$, so $sum_i = p_i \oplus (G^5_{i-1} + b_{15}\overline{Z}) = p_i \oplus (G^5_{i-1} + 1) = p_i \oplus 1 = \overline{p_i} = a_i \oplus \overline{b_i} = a_i \oplus \overline{1} = a_i$. For late completion, $Z = 1$, so $sum_i = p_i \oplus (G^5_{i-1} + b_{15}\overline{Z}) = p_i \oplus (G^5_{i-1} + 0) = p_i \oplus G^5_{i-1}$. In both cases, the logic produces the correct sum: a_i if no abort, otherwise $p_i \oplus G^5_{i-1}$.

As shown in Section 6, the partial abort detection network and modified upper sum logic easily meet all timing constraints. They allow very fast completion, using Level-2 G signals.

4.2 Case 2: Non-Random Input Distributions.

Our second design is a 32+32 bit Brent-Kung adder where the second operand, B, is *frequently small*. This case arises in practice, where a 32+32 bit adder receives non-random input distributions, *e.g.,* when running programs where the inputs are skewed to small numbers.

This case is more general than Case 1, since B may not always be a sign-extended 16-bit number. Again, the goal is to allow very early completion, after Level-2 (*not* Level-3) G signals are produced.

Our solution is a simple modification of our approach for sign-extension. We simply check if operand B is a sign-extended number. If it is, very early completion is used (if there is no abort), as before. If not, the default late completion is used (using Level-5 G signals to produce the sum).

Two hardware modifications are needed, over the Case 1 design. First, a $signextend_B$ detection network is added, to check if B is a sign-extended number. This network consists of two subnetworks, $upper_0$ and $upper_1$. Output $upper_0$ is the NOR of bits $b_{15} - b_{31}$, and determines if these bits are all 0; if so, B is a sign-extended positive number. Output $upper_1$ is the AND of bits $b_{15} - b_{31}$, and determines if these bits are all 1; if so, B is a sign-extended negative number. Operand B is a sign-extended number if $signextend_B = upper_0 + upper_1 = 1$.

Second, $signextend_B$ is used to augment the abort detection network. In particular, the term $\overline{signextend_B} = \overline{upper_0} \cdot \overline{upper_1}$ is ORed with the original abort network. The new abort detection network is shown in Figure 11. An abort occurs *if* a 4-p run is detected in the lower bits (as before), *or* if B is not a sign-extended number ($signextend_B = 0$). The idea is that, if B is not a sign-extended number, we conservatively abort, since a 4-p run may occur in the upper bits, and will not be detected.

There are no changes to upper sum bits; the same implementations are used as in sign extension (see Figure 10).

5 Hybrid Carry-Bypass Adder

This section illustrates how the abort detection scheme can be efficiently combined with existing completion sensing strategies. Our goal is to avoid completion sensing overhead for fast cases, but obtain the benefit of variable completion sensing for slower cases.

We target a dynamic 32-bit asynchronous carry-bypass adder (CBA), illustrated in Figure 12(a) [15]. The adder contains eight 4-bit dual-rail Manchester carry groups that generate dual-rail carry signals, as illustrated in Figure 12(b). For bit i, either c_i^T or c_i^F rising signifies that the carry generation is completed. Because the carry bits can complete in any order, the completion sensing logic must detect when *all* 32 carry bits are completed. Thus, the completion sensing logic consists of a 32-bit OR-AND network.

Figure 12(c) illustrates our implementation consisting of

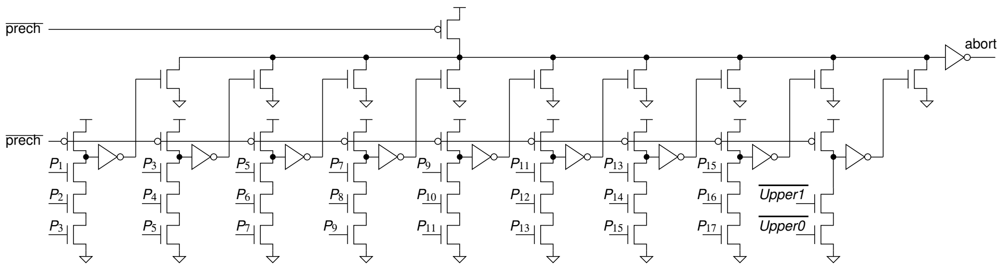


Figure 11. Partial Abort Detection for Small Numbers: Brent-Kung Adder

a tree of domino logic gates that we optimized to minimize the worst-case delay. Specifically, the delay from c_{31} to the done signal goes through two fast 2-input domino AND gates, while the delay from other carry signals such as c_1 go through up to three additional domino gates.

The sum logic is faster than the fastest completion sensing delay and thus is guaranteed to complete before *done*+ is generated. In fact, *done*+ occurs up to 1 ns after the last sum bit changes, representing significant delay overhead.

In order to reduce completion sensing overhead, we combine the completion sensing logic with a speculative completion scheme, as illustrated in Figure 13. Here, a matched average-case delay line, qualified with the output of an abort network, is ORed with the existing variable-delay completion sensing network. When an early case occurs, both inputs to the OR gate will rise, but the *first* to rise causes *done*+, signifying completion. Since the matched delay line is fast, we can often save a *significant* fraction of the completion sensing overhead. For the non-early cases (*abort = 1*), only the completion sensing network rising causes *done*+.

A statistical analysis by Garside et al. [14] guided our choice of abort detection networks. He observed that real data often exhibits a two-humped carry-chain length distribution, one hump near a carry-chain length of 5 and one much closer to the worst-case. Since the original adder was already designed to minimize worst-case delay, we chose to target the abort network towards additions having very short carry-chains.

As illustrated in Figure 14, the abort detection network consists of a group of eight 4-p product terms. The upper 7 terms form the main portion of the detection network, where each 4-p term bridges consecutive 4-bit groups. (The role of the bottom 8th term will be discussed shortly.) To avoid charge-sharing problems, these 4-bit products are implemented in two levels of domino gates. Essentially, these products detect when the maximum effective carry-chain delay consists of 5 consecutive carry propagates or more, assuming that all carry delays are equal and that the carry-bypass delay equals a carry delay. In reality, however, the carry bypass delay and the carry propagate between 4-bit groups, referred to as *inter-group propagate*, are significantly larger than the others carry propagate delays. Consequently, using this 4-p network the average-case matched delay must be larger than:

- PG delay + 1 carry bypass + 1 carry propagate + sum delay and
- PG delay + 3 carry propagates + 1 inter-group propagate + sum delay.

Notice that in the both equations, the generation of the group propagate signal does not appear. This is because it is usually not in the critical path, *i.e.,* it is stable by the time the carry must be bypassed. This, however, is not the case for the group propagate of the first 4-bit group. To address this problem, we could make the matched delay longer to account for this delay. However, this makes the hybrid scheme ineffective in reducing average-case delay. Thus, instead, we abort if this case is detected using an 8th product term, consisting of $p1 \cdot p2 \cdot p3 \cdot p4$.

6 Results

We completed the transistor-level design of the four adders in 0.5 micron HP CMOS14TB three-metal process. This section describes our SPICE analysis to determine various critical delays, as well as statistical analysis to obtain a measure of average-case performance.

6.1 SPICE Analysis

We simulated all designs using Mentor Graphics Accusim (SPICE) simulator at 50°C with a 3.3V power supply. For each of the four Brent-Kung Adders, we simulated a few input cases and report the results in Table 1.

Column *Abort* indicates the delay required for the abort network to complete. For the 64-bit design, the delays for both abort networks are given. For each design, the columns *G2* through *G6* show the delay of the *Done* signal, for the various matched delay paths. For example, *G2* indicates the done signal for very fast completion, where signal Level-2 G signals are used to generate the sum. In each example, these columns have a checkmark if the associated delay represents the delay of the addition. The last column, *Last bit*, gives the delay of the last-changing sum bit. All delays are in nano-seconds.

We also performed SPICE analysis on the Hybrid Carry-Bypass Adder and present a breakdown of delays for various

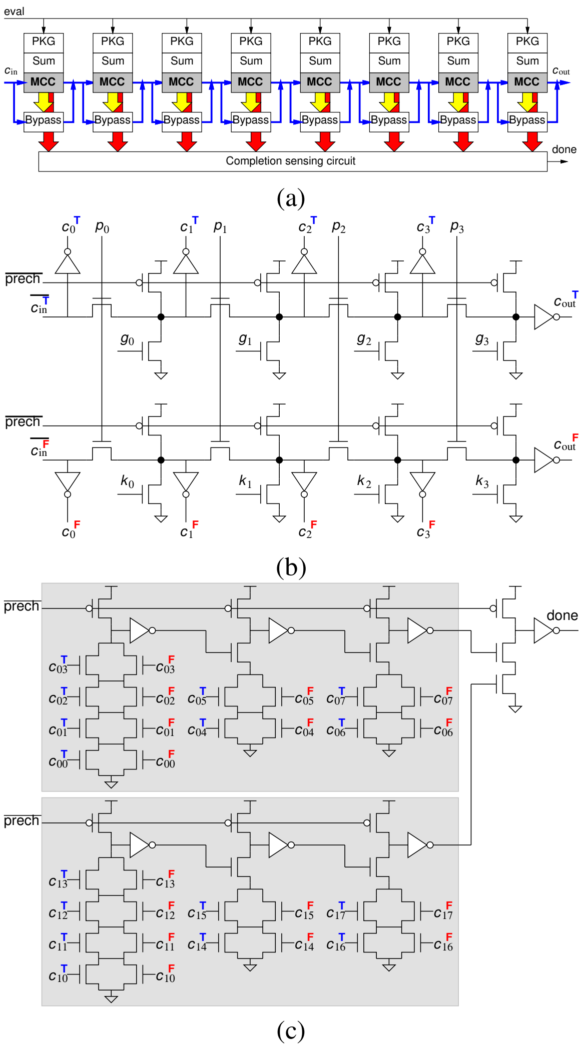


Figure 12. (a) Top-level view of 32-bit carry-bypass adder; (b) dual-rail Manchester carry-chain; and (c) two of the four 8-bit groups constituting the domino logic implementation of the completion sensing tree.

examples in Table 2. The column *PGK Gen. Delay* provides the delay for generating the p_i, g_i, and k_i signals; it is a constant for all examples. The column *Comp. Detect.*, provides the delay through the completion sensing tree. This gives an indication of the delay of the adder if no abort network existed. The column *Matched Delay* contains the delay of the matched delay line; it is essentially constant for all examples. The column *Last Bit* identifies the delay of the last sum bit changing along with its bit #. The column *Done* provides the actual delay of the adder. The last column, *Saved* is the difference between the actual adder delay (given by the *Done* column) and the adder assuming no abort network were used (given by the *Completion Detect.* column). This gives an indication of how much time the abort network saved.

6.2 Statistical Performance Analysis

We statistically analyzed the average-case performance of four of the speculative-completion adders we described: the 32-bit BK, the 32+16-bit BK, the 32-bit+small BK, and the 32-bit Hybrid CBA. For each of these adders we considered

- random data, where each operand bit has 0.5 probability of being 1, and
- real data, obtained by running benchmark programs on an ARM simulator in which we incorporated software performance models of our adders derived from our SPICE analysis.

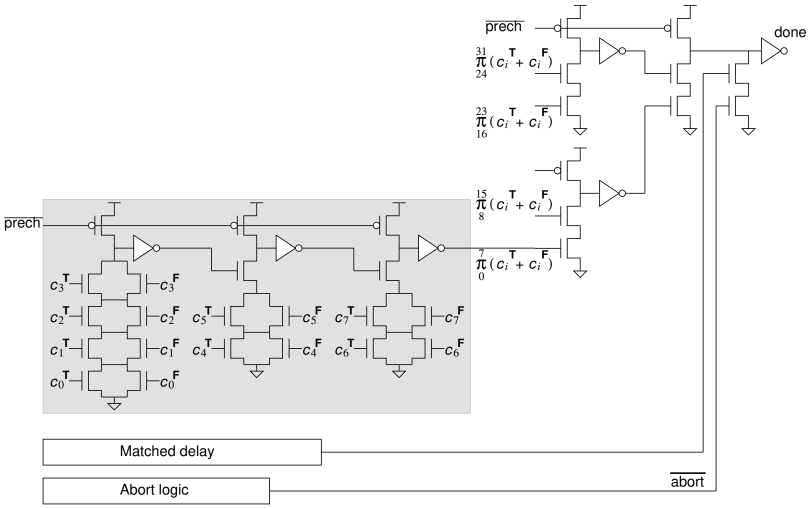


Figure 13. Illustration of the hybrid approach in which completion-sensing is combined with a matched delay line using an abort network.

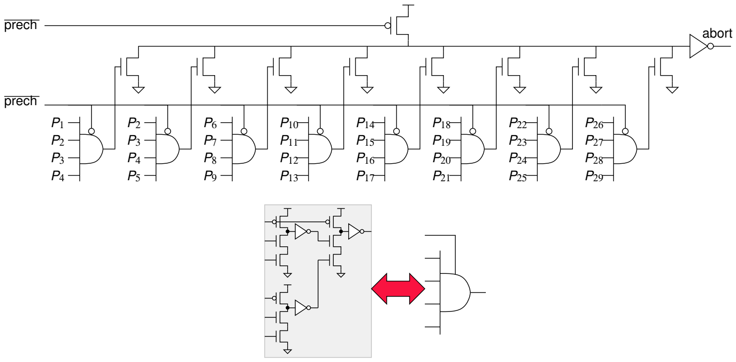


Figure 14. Dynamic transistor-level implementation of the proposed abort network.

For the random case only, we also considered 64-bit BK adders (the real data was for 32-bit additions only). We compared our speculative Brent-Kung adders to synchronous Brent-Kung adders, to demonstrate the advantage of speculative completion. For our hybrid CBA, however, we compared to an asynchronous completion-sensing CBA without speculative completion, to demonstrate the advantage of the hybrid approach.

Random Data. The analysis on random data indicates that speculative completion yields significant performance improvements. On average, the 64-bit BK speculative adder is 29% faster than a 64-bit synchronous BK adder. The 32+32-bit BK adder is 19% faster, and the 32+16-bit BK adder is 8% faster, than a 32-bit synchronous BK adder. The 32-bit results are summarized in Table 3. (The Table only lists the 32-bit adders, since the ARM simulations were only for 32-bit addition.)

Real Data. We obtained real data by running an ARM simulator on four benchmark programs and analyzing all the additions and subtractions performed by the ALU. These operations are partitioned into three sets. The first partition consists of branch-target additions in which a 24-bit sign-extended offset is added to a 32-bit PC address. The second partition consists of address calculations in which 24-bit sign-extended offset is added to a 32-bit base-address. The third partition consists of arithmetic (ALU) 32-bit additions.

Since in our benchmark programs the branch-target offset could always be represented with less than 16-bits, we used the branch partition to analyze our 32+16-bit adder. Furthermore, since a significant fraction of address calculations involved numbers with less than 16-bits, we used the address partition to analyze our 32+small BK adder. Table 3 reports the average improvements obtained for each data partition.

SPICE Simulation of Brent-Kung Adders							
Example	Abort	G2	G3	G4	G5	G6	Last bit
64+64 bit BK Adder			1.71	1.88		2.42	
7FFFFFFFFFFFFFFF+ 0000000000000000	0.86/ 1.09					√	2.33
00000000000001FF+ 0000000000000001	1.01/-			√			1.87
000000000000001F+ 0000000000000001	-/-		√				1.64
3FB000000000001F+ 0010000000000001	-/-		√				1.68
32+32 bit BK Adder			1.63		2.13		
00000001+ 7FFFFFFF	0.81				√		2.11
63A9CB2B+ BA26A3D9	-		√				1.55
32+16 bit BK Adder		1.41			2.15		
0FFB0400+ 00000F0D	-	√					1.29
0FFF0000+ 00000F0D	0.95				√		0.83
0FFFC000+ 00004F0D	0.89				√		1.88
70F84000+ FFFF880D	0.98				√		2.08
70FFC000+ FFFF880D	-	√					1.09
70FFC000+ FFFF9B77	-	√					1.08
32+small-number BK Adder		1.41			2.14		
0FFB0400h + 80000F0Dh	0.87				√		1.29
0FFB0400h+ 00000F0Dh	-	√					1.29

Table 1. SPICE simulation of 0.5 micron Brent-Kung Adders at $50°$C and 3.3V on various inputs.

SPICE Simulation of 32-bit Hybrid Carry-Bypass Adder							
Example	PGK Gen.	Comp. Detect.	Matched Delay	Abort Gen.	Last Bit	Done	Saved
AC6EC2A7+ EEC45692	0.54	1.83	1.59	-	1.18/ 24	1.65	0.18
FFFFFFFF+ 00000001	0.54	4.39	1.55	0.93	-	4.52	-0.13
0000001F+ 00000001	0.54	2.17	1.52	1.07	1.74/ 5	2.33	-0.16
F8000000+ 00000000	0.54	1.88	1.60	-	1.64/ 31	1.66	0.22
01FDFDFC+ 00040404	0.54	2.19	1.54	-	1.69/ 9	1.70	0.49

Table 2. SPICE simulation of 0.5 micron Hybrid Carry-Bypass Adder at $50°$C and 3.3V on various inputs.

As mentioned earlier, it has been observed that real data is often skewed towards the worst case, exhibiting longer average carry-chain lengths than would be predicted using random data [14]. For this reason, asynchronous adders often perform poorer in practice than a theoretical analysis using random data might expect. However, it is also important to note that results from real data often exhibit significant variances and can be a manifestation of the unique properties of an individual benchmark. Thus, when making performance judgments, we believe that both real and random data should be critically analyzed.

Results are presented in Table 3. The 32+16-bit BK adder had the lowest average delay (8.52% improvement) and lowest individual delay (on dhrystone, 13.5% improvement) for branch calculations. The 32+small BK adder, however, performs relatively poorly on address calculations, primarily because the percentage for very fast completion is particularly low in the Dhrystone benchmark. This suggests that, for this application, a three-tiered abort network able to complete after either G2 or G3 may be preferred. Due to the lack of time, such a circuit could not be simulated using SPICE and thus a more detailed analysis could not be presented.

We also observed a high variance in the performance of the CBAs. The hybrid CBA does surprisingly well for address and branch calculations, over a base asynchronous CBA, but is often slow when doing arithmetic adds. Simulations show that the percent improvement delivered by the abort detection network ranged from 1.4% (Dhrystone arithmetic) to 19.84% (Dhrystone branches).

Acknowledgements

We are greatly indebted to the AMULET group at the University of Manchester for insightful discussions, and wish to thank Prof. Jim Garside for providing us with an ARM simulator. We also thank Prof. Charles Zukowski of Columbia University and Prof. Al Davis of University of Utah for helpful discussions.

References

[1] G. Birtwistle and A. Davis (eds.), *Asynchronous Digital Circuit Design*, Springer-Verlag (Workshops in Computing series), London 1995.

[2] S. B. Furber, P. Day, J.D. Garside, N.C. Paver and J.V. Woods, “A Micropipelined ARM”, in *Proceedings of VLSI 93*, September 1993, pp. 5.4.1–5.4.10.

[3] E. Brunvand, “The NSR Processor”, in *Proceedings of 26th HICSS*, vol. I, January 1993, pp. 428–435.

[4] R.F. Sproull, I.E. Sutherland and C.E. Molnar, “The Counterflow Pipeline Processor Architecture”, in *Design and Test of Computers*, vol 11, Fall 1994, pp. 48–59.

[5] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken and F. Schalij, “Asynchronous Circuits for Low Power: a DCC Error Corrector”, in *Design and Test of Computers*, vol 11, Summer 1994, pp. 2–32.

[6] I.E. Sutherland, "Micropipelines", in *Communications of the ACM*, vol. 32, June 1989, pp. 720–738.

[7] A.J. Martin, "Asynchronous Datapaths and the Design of an Asynchronous Adder", in *Formal Methods in System Design*, volume 1:1, July 1992, pp. 119–137.

[8] K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, John Wiley and Sons, Inc., 1979.

[9] M.E. Dean, D.L. Dill and M. Horowitz, "Self-Timed Logic Using Current-Sensing Completion Detection (CSCD)", in in *Proceedings of ICCD*, October 1991.

[10] R.P. Brent and H.T.Kung, "A Regular Layout for Parallel Adders", in *IEEE Trans. on Cptrs.*, vol. C-31, March 1982, pp. 260–264.

[11] K. Suzuki, M. Yamashina and T. Nakayama, "A 500 MHz, 32 bit, 0.4 μm CMOS RISC Processor", in *IEEE JSSC*, vol. 29, December 1994, pp. 1464–1473.

[12] S.M. Nowick, "Design of a Low-Latency Asynchronous Adder Using Speculative Completion", in *IEE Proceedings - Computers and Digital Techniques*, vol. 143, no. 5, pp. 301-307 (September 1996).

[13] L.S. Nielsen and J. Sparsoe, "A Low-Power Asynchronous Data Path for a FIR Filter Bank", in *Proceedings of Async96*, March 1996, pp. 197–207.

[14] J.D. Garside, "A CMOS VLSI implementation of an asynchronous ALU. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 181-207. Elsevier Science Publications, 1993.

[15] K. Y. Yun, P. A. Beerel, V. Vakilotojar, A.E. Dooply and J. Arceo, "The design and verification of a high-performance low-control-overhead asynchronous differential equation solver", in *Proceedings of Async97*.

Statistical Performance Analysis of Various Adders

Data Set Source	32-bit BK	32+16-bit BK	32+small BK	32-bit CBA
Random data				
Avg. % Early	80.0	34.4	N/A	N/A
Avg. Delay	1.73	1.90	N/A	N/A
% Improvement	19	8	N/A	N/A
Branch calculations partition				
	% Early			
Dhrystone	56.90	55.70	N/A	63.60
Espresso	52.80	41.30	N/A	45.50
Compiler 1	40.30	30.00	N/A	47.10
Compiler 2	8.50	21.50	N/A	7.70
Avg. % Early	39.62	37.12	N/A	40.97
Avg. Delay	1.94	1.88	N/A	2.43
% Improvement	5.17	8.52	N/A	11.88
Address calculations partition				
	% Early			
Dhrystone	73.60	N/A	8.40	68.20
Espresso	63.40	N/A	27.50	43.30
Compiler 1	45.80	N/A	14.10	41.40
Compiler 2	67.00	N/A	26.70	65.20
Avg. % Early	62.45	N/A	19.18	54.53
Avg. Delay	1.83	N/A	2.01	2.18
% Improvement	10.96	N/A	2.04	15.50
Arithmetic calculations partition				
	% Early			
Dhrystone	11.30	N/A	N/A	10.00
Espresso	33.30	N/A	N/A	31.30
Compiler 1	24.10	N/A	N/A	22.30
Compiler 2	22.40	N/A	N/A	20.90
Avg. %	22.77	N/A	N/A	21.12
Avg. Delay	2.03	N/A	N/A	3.27
% Improvement	0.90	N/A	N/A	3.25

Table 3. Statistical performance analysis on random and ARM-simulation data.