Design of a low-latency asynchronous adder using speculative completion

S.M.Nowick

Indexing terms: Asynchronous systems, Speculative completion, Adders, Datapath design

Abstract: A new general method for designing asynchronous datapath components, called *speculative completion*, is introduced. The method has many of the advantages of a bundled data approach, such as the use of single-rail synchronous datapaths, but it also allows early completion. As a case study, the method is applied to the high-performance parallel BLC adder design of Brent and Kung. Through careful gate-level analysis, performance improvements of up to 30% over a comparable synchronous implementation are expected.

1 Introduction

Asynchronous design has enjoyed a resurgence in the last five to ten years, with a number of technical and practical advances [1]. In principle, asynchronous systems promise several advantages over synchronous systems: (i) lower power, since an asynchronous component computes only when necessary; (ii) higher performance, since global clock distribution and synchronisation can be avoided; and finally, (iii) greater modularity and ease of design, since there are no global timing constraints.

An important recent trend is the design and, in many cases, fabrication of practical large-scale asynchronous systems, such as microprocessors [2–7] and DSP chips [8]. Critical to these systems is the design of efficient datapath support components, such as adders. This paper proposes a new method for designing asynchronous data-path components, targeted towards high-performance design.

Many approaches have been proposed to designing 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 [1, 9]. This delay may be an inverter chain or a replicated portion of the critical path. This method has been widely used [3–5, 8]. 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 [Note 1].

A *completion detection* method [1, 10] 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 four-phase *RZ* and two-phase *LEDR* (see [1]), and the methods have been applied to a number of designs such as adders [10, 11]. 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 delays between completion and its detection. Furthermore, the increased wiring and switching activity often result in much greater area and power consumption. A promising alternative scheme avoids the detection network [12], but requires special current sensors and still requires a number of gate delays of overhead.

In this paper, we propose a new alternative method for designing asynchronous datapath components, which we call *speculative completion.* Our method has many of the advantages of the bundled data approach, such as the use of a single-rail synchronous datapath. Unlike bundled data, though, we use several different matched delays: a worst-case model delay, and one or more speculative (i.e. early-completion) delays. Therefore, a component can operate at several possible speeds. A speculative delay allow early completion, and is disabled for worst-case data. Unlike existing completion detection methods, however, early completion detection occurs *in parallel* with the datapath computation, not after computation is complete. The completion overhead is therefore minimal.

The method is applicable to a number of datapath designs. As a case study, we illustrate it on a particular example: the design of a carry lookahead adder, based on the high-performance parallel design of Brent and Kung [13]. Through careful gate-level analysis, we estimate performance improvements up to 30% over a comparable synchronous implementation, for random input data. We intend to lay out and simulate the design in the future, so that wiring and fanout capacitance can be considered more accurately.

IEE Proceedings online no. 19960704

Paper first received 4th January 1996 and in revised form 19th June 1996

The author is with the Department of Computer Science, Columbia University, New York, NY 10027, USA

Note 1: Unlike synchronous design though, delay margins may be somewhat tighter, since timing constraints are localised.

2 Overview

2.1 Speculative completion

A standard single-rail bundled datapath is shown in Fig. 1. A single 'model delay' is used, with input *req* and output *ack*. The model delay receives a request, *req*, when data inputs are valid, and will produce an *ack only* after the function outputs are valid. This delay must be slower than the function block under all physical conditions and all data inputs.

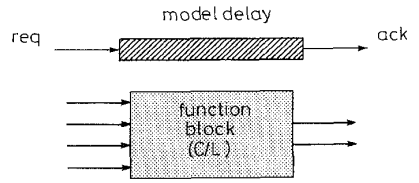


Fig. 1 *Block diagram of standard bundled datapath*

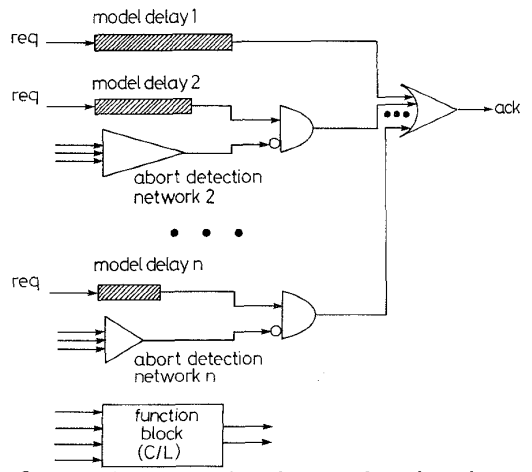


Fig. 2 *General architecture of speculative completion datapath*

Fig. 2 shows the basic architecture of our speculative completion datapath. There are two key features. First, *multiple* (i.e. two or more) model delays are used: one for the worst-case and the remaining ones for speculative completion. Speculative delays are used to provide different speeds of early completion. For example, in a ripple-carry adder, an 'average-case' delay could be used if adder inputs result in short carry chains; a 'best-case' delay could be used if there is no carry (e.g. if an input is 0).

Secondly, an *abort detection network* is associated with each speculative delay. The network determines if the coresponding early completion must be aborted, due to worst-case data. This detection is computed in parallel with datapath computation, and must be completed before the delay produces its output.

2.2 BLC adder design

As a case study, we focus on the parallel carry-lookahead adder of Brent and Kung [13], as adapted by Suzuki *et al.* [14], as shown in Fig. 3. This adder uses a bitwise, or binary, lookahead carry (BLC) method. In a CMOS implementation, the stack depth of each gate is limited to two, and the gate fanout load is mostly limited to two. The design is amenable to regular layout.

The 32-bit adder produces all propagate ($\bar{p}$) and generate ($\bar{g}$) signals in level-0 and produces a sum in level-6. The critical path from input to output is therefore seven 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. In particular, level-1 computes all 2-bit p and g values, level-2 computes all 4-bit values, and so on. (Though the Figure suggests that a 32-bit p is computed in level-5, this is not necessary.) 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} of the preceding stage (taken from level-5).

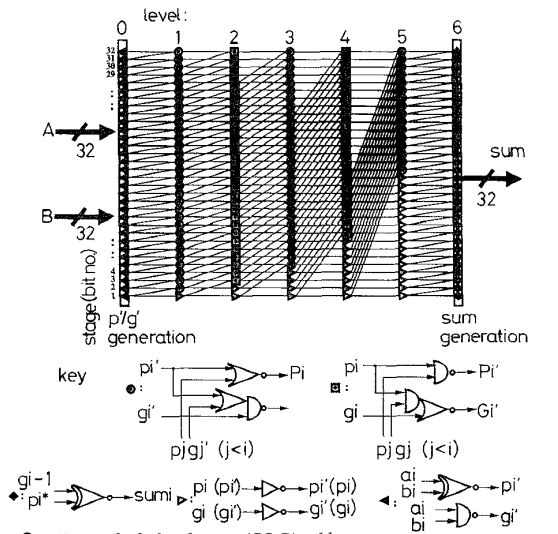


Fig. 3 *Binary lookahead carry (BLC) adder*
pi* from level-0

Our speculative design uses the same basic datapath, with several modifications. There are three key components. First, we use two model delays: one worst-case delay and one speculative average-case delay. The speculative delay is enabled if all final generate bit values are produced by level-3, thus allowing early completion after only five gate delays. Secondly, we design an *abort detection network*, to inhibit early completion. This network computes in parallel with the datapath, and uses safe approximation to determine when to abort. Finally, some modifications of the sum generator in level-6 are needed. As in a standard bundled data approach, there are no hazard-free requirements except on model delays.

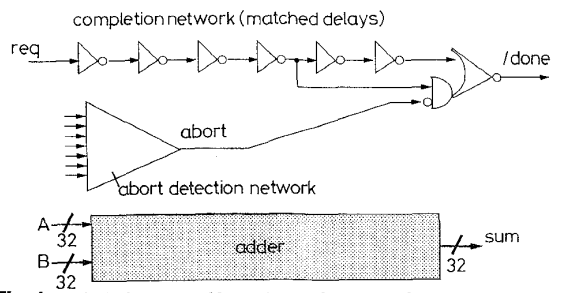


Fig. 4 *Block diagram: adder with speculative completion*

3 Adder design

3.1 Completion network

Fig. 4 shows a block diagram of our speculative completion adder. For simplicity, inverter chains are used for model delays, but extracted portions of the

critical path may 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 seven gate delays. The speculative delay path has five gate delays, and applies to cases where no useful computation occurs in level-4 and level-5, i.e. the final generate values are available in level-3.

In the Figure, the speculative path is disabled by an abort signal, 'abort'. This signal is allowed to glitch. The only timing requirement is that it becomes stable and valid before the speculative path enables the final gate [Note 2]. Therefore, for this completion network, 'abort' must be computed in less than four gate delays.

3.2 Abort detection network

The key component of our design is the *abort detection network*, which generates the abort signal. Unlike existing completion detection schemes, the network does not begin to detect early completion after completion has occurred. Instead, the network detects conditions for early completion in parallel with the datapath computation.

There are three goals in designing the abort detection network:

(i) abort must be asserted whenever late completion occurs

(ii) abort should not be asserted for most cases where early completion occurs; and

(iii) the network should be small and fast.

Condition (i) is a safety requirement; it is satisfied by using a conservative approximation to detect late completion. Conditions (ii) and (iii) are optimality requirements, on hit rate and logic realisation, respectively.

3.2.1 Conditions for late completion: We now derive safe conditions for late completion of the BLC adder of Fig. 3. By early completion, in this paper, 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. This condition is fairly common, and its detection results in an efficient adder implementation.

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 $j = i - 2^{n-1}$ (we ignore the alternating inversions in the given implementation here). 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 eight consecutive level-0 propagate signals. Therefore, the following condition is safe for the detection of late completion:

Condition 3.1: Late completion can only occur if there exists a run of eight consecutive level-0 propagate signals.

We call such a condition an 8-p run. The goal of the abort detection network is to detect any such run. This condition is safe, since the detection of all 8-p runs guarantees the detection of every instance of late completion.

Note 2: As an alternative, if the abort signal is guaranteed hazard-free, and is used to enable the early output, timing requirements can be relaxed, and the signal can arrive later.

A small optimisation is that, for addition, the carry-in to the first stage (stage 1) is 0, so a run of eight level-0 propagate signals from stage 1 to stage 8 can be ignored. Therefore, for purposes of this paper, we will ignore the contribution of level-0 p_1. Our goal is to detect any run of eight consecutive level-0 propagate signals, from stage 2 to stage 32.

3.2.2 Detecting late completion: Abort detection networks can now be designed which safely detect the above condition. For efficiency, we further approximate this condition, safely, to produce simpler networks.

Definition 3.1: A product c of level-0 propagate signals covers a given 8-p run, $p_i\, p_{i+1} \ldots p_{j-1}\, p_j$, from i to j if each propagate signal p_x, that is an input to product c, is also contained in the run; that is, $i \leq x \leq j$.

By this definition, a product covers an 8-p run if the product consists only of a non-empty subset of propagate signals in the run. If the 8-p run holds, all the propagate signals from p_i to p_j are 1, and therefore product c will be 1. As an example, product $c = p_5\, p_6\, p_7$ covers the 8-p run from p_3 to p_{10}. In this case, c detects the run using a safe approximation. If the run occurs, then $c = 1$. 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.

Our approach is to select products, 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. In this case, if any 8-p run occurs, the network will detect it.

A number of alternative designs are presented below. In each implementation, every 8-p run between stages 2 and 32 is detected. Each design uses a different safe approximation to exact abort detection. In Section 4, we will evaluate the trade-offs between 'hit rate' and latency/area of these designs.

3.2.3 Simple detection networks: A simple sum-of-products detection network can be used, where each product contains a short run of level-0 propagate signals ('p-signals').

3-Literal products: Each product contains a run of three p-signals (in level-0). The network contains four products; it is given by equation: $p_7p_8p_9 + p_{13}p_{14}p_{15} + p_{19}p_{20}p_{12} + p_{25}p_{26}p_{27}$. Product $p_7p_8p_9$ covers the 8-p runs for stages: 2 – 9, 3 – 10, ... , 7 – 14. If any of these runs occurs, this product will be 1. The remaining three products cover the remaining runs, similarly.

4-Literal products: Each product contains a run of four p-signals; there are five products. The sum-of-products equation is: $p_6p_7p_8p_9 + p_{11}p_{12}p_{13}p_{14} + p_{16}p_{17}p_{18}p_{19} + p_{21}p_{22}p_{23}p_{24} + p_{26}p_{27}p_{28}p_{29}$. Each product covers fewer 8-p runs than in the preceding 3-literal product design, so more products are required.

5-Literal products: Each product contains a run of five p-signals; there are six products. The sum-of-products equation is: $p_5p_6p_7p_8p_9 + p_9p_{10}p_{11}p_{12}p_{13} + p_{13}p_{14}p_{15}p_{16}p_{17} + p_{17}p_{18}p_{19}p_{20}p_{21} + p_{21}p_{22}p_{23}p_{24}p_{25} + p_{25}p_{26}p_{27}p_{28}p_{29}$. Note that in this case, adjacent products overlap; that is, they have a literal in common.

3.2.4 Augmented detection networks: Alternative networks can be used, where each product in a simple detection network is augmented with a new

literal: $\overline{k_i}$. For example, the 3-literal simple network can be augmented to: $\overline{k_6}p_7p_8p_9 + \overline{k_{12}}p_{13}p_{14}p_{15} + \overline{k_{18}}p_{19}p_{20}p_{21} + \overline{k_{24}}p_{25}p_{26}p_{27}$. Consider the augmented product, $\overline{k_6}p_7p_8p_9$. Here, k_6 is the (level-0) 'kill' condition for stage 6, $A_6.\overline{B_6}$, where both A and B input bits are 0. The augmented product uses the complement of this condition, $\overline{k_6}$; that is, either p_6 or g_6 is 1.

Our motivation is that this augmented product covers the same 8-p runs covered by the original product, but is a tighter approximation (i.e. indicates fewer unnecessary aborts). To see this, consider the 8-p runs covered by the original product: 2 – 9, 3 – 10, ..., 7 – 14. If 8-p run 2 – 9 occurs, for example, then p_6, p_7, p_8 and p_9 are 1. Therefore $\overline{k_6}$ is also 1, and the augmented product is 1. A similar result holds for all 8-p runs except 7 – 14. For this last case, if 8-p run 7 – 14 occurs, then p_7, p_8 and p_9 are 1. If p_6 or g_6 is 1, the augmented product is 1 (indicating abort). However, if k_6 is 1, then even though the 8-p run occurs, it has no effect, since it cannot propagate from a preceding kill stage, k_6. Therefore, the abort produced by the original product, p_7, p_8, p_9, is unnecessary. In contrast, the augmented product is 0, and no abort occurs.

A final motivation for this approach is that each $\overline{k_i}$ condition has a simpler CMOS implementation (2-input NOR of the data inputs, $\overline{A_i + B_i}$) than p_i (2-input XOR of the data inputs, $A_i \oplus B_i$).

Each of the above three simple networks can be transformed into an augmented network using this method. The simple abort networks with 3-, 4- and 5-literal products can be notated as ($3p$, $0k$), ($4p$, $0\overline{k}$) and ($5p$, $0\overline{k}$), respectively. The corresponding augmented networks are notated as ($3p$, $1\overline{k}$), ($4p$, $1\overline{k}$) and ($5p$, $1\overline{k}$).

Each product can also be augmented with a 2-bit $\overline{k}$ input as well. For the original product $p_7p_8p_9$, a 2-bit kill is given by: $k_{5,6} = k_6 + k_5p_6$. The complement is used to produce an augmented product, $\overline{k_{5,6}}p_7p_8p_9$. The corresponding augmented networks are notated as ($3p$, $2\overline{k}$), ($4p$, $2\overline{k}$) and ($5p$, $2\overline{k}$).

3.2.5 Timing requirements:

As indicated in Section 3.1, the abort detection network is allowed to have hazards. The only requirement is that it produces a stable and valid result within four gate delays (i.e. when level-3 computation is completed in the adder datapath).

Network ($3p$, $0\overline{k}$) can easily be implemented in four gate delays using simple CMOS gates with stack size 2 (three gate delays for the network, one gate delay for the initial $p/g/k$ generation). The remaining networks can be implemented in five equivalent gate delays (in a couple of cases, using one three-input NOR each).

Because this network is critical, we assume that the designer can optimise its performance to meet this timing requirement. This assumption is reasonable considering that transistor sizing and/or dynamic logic can be aggressively used to speed up this network, regardless of any general optimisations used on the larger adder.

3.3 Sum generation

We have described the design of the completion network and abort detection network, and their interaction. The abort detection network will allow early completion in many cases, and will always abort if late completion is required. However, the sum-generation in Figs. 3 and 5 does not yet produce correct results. In particular, 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} of the preceding stage, taken from level-5. For early completion, we want the carry-out G_{i-1} of the preceding stage taken from level-3, since all level-3 'generate' signals are final. Therefore, sum generation needs a bypass, so it can receive both level-3 and level-5 G_{i-1} signals.

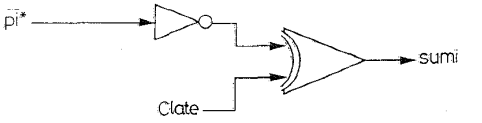


Fig. 5 *Original sum generation*
*pi** is from level-0
Clate is G_{i-1} from level-5

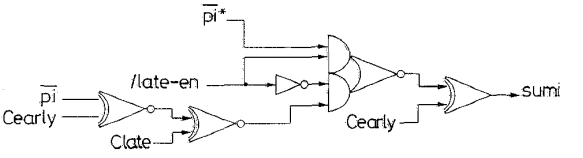


Fig. 6 Modified sum generation
*pi** is from level-0
Clate is G_{i-1} from level-5
Cearly is G_{i-1} from level-3

A corrected gate-level sum generator is shown in Fig. 6. The level-3 G_{i-1} signal is labelled *Cearly*, and the level-5 G_{i-1} signal is labelled *Clate*. For now, assume that the late-enable signal, */late-en*, is the inverted abort signal produced by the abort detection network. The abort signal is broadcast to every sum module. This approach requires excessive delay, but will be corrected below.

In each case, a correct sum is produced. Suppose completion is early and */late-en* is unasserted (*/late-en* = 1). The network correctly computes the XOR of p_i and *Cearly*. Since *Cearly* passes through only one gate, an early sum is generated in a total of 4 + 1 = 5 gate delays.

Alternatively, suppose that */late-en* is asserted (*/late-en* = 0). In this case, early or late completion may occur. Once the final *Clate* is produced, three cases are possible: (i) *Cearly* and *Clate* are 0, (ii) *Cearly* is 0 and *Clate* is 1, and (iii) *Cearly* and *Clate* are both 1. The remaining case, *Cearly* is 1 and *Clate* is 0, is impossible, since *Clate* is a positive unate function of *Cearly*. In each case, the correct result is computed. In case (i), AOI-gate produces p_i, and $sum_i = p_i \oplus Cearly = p_i \oplus Clate$. In case (ii), the AOI-gate produces $\overline{p_i}$, and $sum_i = \overline{p_i} \oplus Cearly = p_i \oplus Clate$. In case (iii), the AOI-gate produces p_i and, again, $sum_i = p_i \oplus Cearly = p_i \oplus Clate$. Therefore, the result is alway functionally equivalent to the original sum in Fig. 5. Since *Clate* passes through three gates, a late sum is generated in a total of 6 + 3 = 9 gate delays.

The problem with the above approach is that it requires the critical */late-en* signal to reach all 32 sum modules within three gate delays, while the abort detection network itself may require four gate delays to generate it. Instead of broadcasting the inverted 'abort' signal as a late-enable for each sum bit, we note that the abort detection network is a sum-of-products, where each product covers, or detects, a set of 8-p runs. We now show that these products individually can be used as */late-en* signals.

As an example, in network $(3p, 0\bar{k})$, product $c = p_{191}p_{20}p_{21}$ detects a set of 8-p runs, one of which is 14–21. In this case, the run might cause a late carry-out of stage 21. Therefore, c itself can be used (actually, the corresponding NAND $\bar{c}$) as */late-en* to stage 22. If the 8-p run, 14–21, does not occur, no late carry-out from stage 21 can occur, by Condition 3.1. That is, *Cearly* = *Clate* in the sum module of stage 22. In this case, */late-en* is unasserted (i.e., 1), and the sum is correctly generated: $sum_{22} = p_{22} \oplus Cearly$. Alternatively, if the 8-p run, 14–21, occurs, a late carry-out is possible. In this case, $\bar{c}$ is asserted (i.e., 0), as is $\overline{abort}$, and the sum is correctly generated, as before, $sum_{22} = p_{22} \oplus Clate$. In each case, a correct result is produced.

Intuitively, for each sum module which changes value late, due to a change from *Cearly* to *Clate*, the corresponding product, $\bar{c}$, which covers this case will be asserted. A change from *Cearly* to *Clate* cannot occur without the corresponding 8-p run adjacent to it (in the above example, from 14 to 21 enabling a change in 22).

Product c may be assumed to have two gate delays in an optimised abort implementation, hence it will produce a valid and stable result after three gate delays (one delay is for level-0). Since each individual */late-en* is produced within three gate delays, the given timing requirements are satisfied. As a result, an early sum will be valid and stable after five gate delays, and a late sum will be valid and stable after nine gate delays.

Note that the fanout load of $\bar{c}$, in producing */late-en* signals, depends on the abort detection network. For network $(3p, 0\bar{k})$, each product covers four 8-p runs, so the fanout load is four */late-en* signals. Either $\bar{c}$ from the abort detection network can directly drive as */late-en* in Fig. 6, or the logic for $\bar{c}$ can be replicated as needed to reduce the load. We are currently exploring better techniques to implement */late-en* and the modified sum generation circuit. Preliminary work suggests that alternative designs may reduce worst-case sum generation to only seven gate delays [Note 3].

4 Results

We now analyse the performance and operation of an asychronous BLC adder designed for speculative completion. In the analysis below, we assume that all gate delays are equal. We also assume that the abort detection network can be implemented to meet timing constraints, so early completion takes five gate delays and late completion takes nine gate delays. In addition, a random input distribution is assumed.

Table 1 lists our initial results, on the nine abort detection networks described in Section 3.2; other networks can also be considered. We compare against a synchronous implementation, which is assumed to have a latency of seven gate delays. The performance improvement ranges from 5 to 30% over a comparable synchronous implementation. Interestingly, for the best abort detection network, $(5p, 2\bar{k})$, early completion occurs for over 90% of inputs.

Several of the table entries have asterisks (*). In these cases, products in the abort detection network overlap (see Section 3.2). As a result, these products are not independent, and probabilistic analysis of 'early completion (%)' is more subtle. Details of our analysis are presented in the Appendix.

Note 3: Personal communication with Prof. Charles Zukowski (Columbia University)

Table 1: Performance and area results

Abort detection network	Abort detection area (no. lits.)	Early completion (%)	Average adder latency (no. gates)	Performance improvement (%)
$(3p, 0\bar{k})$	12	59	6.64	5
$(4p, 0\bar{k})$	20	72	6.12	14
$(5p, 0\bar{k})$*	30	83	5.68	23
$(3p, 1\bar{k})$	16	67	6.32	11
$(4p, 1\bar{k})$	25	79	5.84	20
$(5p, 1\bar{k})$*	36	87	5.52	27
$(3p, 2\bar{k})$	24	72	6.12	14
$(4p, 2\bar{k})$*	35	84	5.65	24
$(5p, 2\bar{k})$*	48	90	5.39	30

5 Conclusions

This paper has introduced a general method for the design of asynchronous datapath components. The method has many advantages of a bundled data approach, but also allows early completion. The method was applied to a high-performance parallel BLC adder design. Through careful gate-level analysis, we estimate performance improvements up to 30% over a comparable synchronous implementation.

An important contribution of this work is to demonstrate a bundled-data design method which can take advantage of typical-case delays, and thereby outperform one of the most efficient synchronous adder designs.

Critical to the success of our method is the ability to implement a fast abort detection network, which meets the given timing requirements. We intend to simulate and layout our design, so that the wiring and fanout capacitance can be considered more accurately. We will also explore different designs for modified sum generation. In addition, we intend to apply our method to other asynchronous adders, such as 4-bit carry-lookahead adders, as well as other datapath components such as multipliers and comparators.

6 Acknowledgments

The authors wish to thank Prof. Al Davis of the University of Utah, and Prof. Charles Zukowski and F.-C. Cheng of Columbia University, for helpful discussions. This work was supported by an NSF CAREER award MIP-9501880 and by a grant from IBM Corporation.

7 References

1 BIRTWISTLE, G., and DAVIS, A., (Eds.): 'Asynchronous digital circuit design' (Springer–Verlag, London, 1995)
2 MARTIN, A.J., BURNS, S.M., LEE, T.K., BORKOVIC, D., and HAZEWINDUS, P.J.: 'The design of an asynchronous microprocessor'. Proceedings of Conference on *Advanced research VLSI*, 1991
3 FURBER, S.B., DAY, P., GARSIDE, J.D., PAVER, N.C., and WOODS, J.V.: 'A micropipelined ARM'. Proceedings of *VLSI 93*, 1993, pp. 5.4.1–5.4.10
4 BRUNVAND, E.: 'The NSR processor'. Proceedings of *26th HICSS*, 1993, Vol. 1, pp. 428–435
5 SPROULL, R.F., SUTHERLAND, I.E., and MOLNAR, C.E.: 'The counterflow pipeline processor architecture', *Des. Test Comput.*, 1994, **11**, (3), pp. 48–59
6 NANYA, T., UENO, Y., KAGOTANI, H., KUWAKO, M., and TAKAMURA, A.: 'TITAC: design of a quasi-delay-insensitive microprocessor', *Des. Test Comput.*, 1994, **11**, (2), pp. 50–63

7 DEAN, M.E.: 'STRIP: a self-timed RISC processor architecture'. PhD thesis, Stanford University, 1992
8 VAN BERKEL, K., BURGESS, R., KESSELS, J., PEETERS, A., RONCKEN, M., and SCHALIJ, F.: 'Asynchronous circuits for low power: a DCC error corrector', *Des. Test Comput.*, 1994, **11**, (2), pp. 2–32
9 SUTHERLAND, I.E.: 'Micropipelines', *Commun. ACM*, 1989, **32**, pp. 720–738
10 MARTIN, A.J.: 'Asynchronous datapaths and the design of an asynchronous adder', *Form. Methods Syst. Des.*, 1992, **1**, (1), pp. 119–137
11 HWANG, K.: 'Computer arithmetic: principles, architecture and design' (Wiley, New York, 1979)
12 DEAN, M.E., DILL, D.L., and HOROWITZ, M.: 'Self-timed logic using current-sensing completion detection (CSCD)'. Proceedings of *ICCD*, 1991
13 BRENT, R.P., and KUNG, H.T.: 'A regular layout for parallel adders', *IEEE Trans. Comput.*, 1982, **C–31**, pp. 260–264
14 SUZUKI, K., YAMASHINA, M., and NAKAYAMA, T.: 'A 500 MHz, 32 bit, 0.4 μm CMOS RISC processor', *IEEE JSSC*, 1994, Vol. 29, pp. 1464–1473

8 Appendix: Probability analysis

This section gives details of our probabilistic analysis of early completion, which was used to generate results in Table 1.

Inputs are assumed to be randomly distributed. An abort detection network contains products P_i, $1 \le i \le n$ where the P_is are identical functions of different bits. There are two cases: (i) products P_i in the abort detection network are all disjoint (i.e. have no shared bits), and (ii) some products P_i overlap (i.e. share bits).

8.1 Case I: Nonoverlapped P_is

In this case, the probability that $P_i = 0$ is independent of the probability that $P_j = 0$ for all $i \neq j$. Therefore, the probability that every product in the abort network is 0 is

$$\begin{aligned} Prob_{off} &= Prob(\overline{P_1} \cdot \overline{P_2} \cdots \overline{P_n}) \\ &= Prob(\overline{P_1}) \cdot Prob(\overline{P_2}) \cdots Prob(\overline{P_n}) \\ &= Prob(\overline{P_i})^n = (1 - Prob(P_i))^n \end{aligned} \quad (1)$$

where $Prob(P_i)$ is the probability that an arbitrary product P_i is 1 (all P_is have identical probability). $Prob_{off}$ is the probability of no abort detection, hence of early completion.

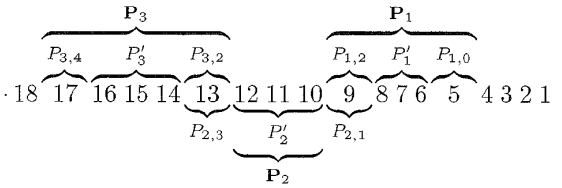
As an example, the abort detection network $(4p, 0\bar{k})$ has $n = 5$ nonoverlapped products, each covering a 4-p run. Each p_l propagate bit in product P_i has a 0.5 probability of being 1 (since $p_l = a_l \oplus b_l$ for corresponding data inputs a_l and b_l), so $Prob(P_i) = (0.5)^4 = 0.125$. Therefore, the probability of early completion is: $Prob_{off} = (1 - 0.125)^5 = 0.7242$.

8.2 Case II: Overlapped P_is

In this case, products in the abort detection network may overlap, i.e. share bits. For example, in network $(5p, 0\bar{k})$, product $P_1 = p_5p_6p_7p_8p_9$ and product $P_2 = p_9p_{10}p_{11}p_{12}p_{13}$ overlap in bit 9. For simplicity, we assume that only adjacent products overline may overlap (i.e., P_i and P_{i+1}); this assumption holds for all overlapped cases in Table 1.

If products P_i and P_{i+1} overlap, probabilities $Prob(\overline{P_i})$ and $Prob(\overline{P_{i+1}})$ are not independent. In this case, $Prob(\overline{P_i} \cdot \overline{P_{i+1}}) \neq Prob(\overline{P_i}) \cdot Prob(\overline{P_{i+1}})$, so the results of case I do not hold.

We first introduce notation to partition each P_i into overlapped and nonoverlapped segments: $P_{i,i+1}$, P'_i, and $P_{i,i-1}$. This notation is illustrated on a fragment of network $(5p, 0\bar{k})$, showing bits covered by products P_1, P_2 and P_3:



$Prob(P_i)$ is given by the product of the independent probabilities of its three segments:

$$Prob(P_i) = Prob(P_{i,i+1}) \cdot Prob(P'_i) \cdot Prob(P_{i,i-1}) \quad (2)$$

We distinguish segments $P_{i-1,i}$ and $P_{i,i-1}$, although they cover the same bits. Segment $P_{i-1,i}$ corresponds to product P_{i-1}, and segment $P_{i,i-1}$ corresponds to product P_i [Note 4].

An outline of our approach is: first, compute the probability, $Prob_{on}$, that at least 1 product, P_i in the network is 1 (i.e., abort); secondly, use this result to compute $Prob_{off} = 1 - Prob_{on}$, the probability that no product is one in the network (i.e., early completion).

(*i*) Computing $Prob_{on}$: To compute $Prob_{on}$, we build up disjoint cases. First, we introduce the notation: $P_{i/-} = 1$ if product $P_i = 1$ and no other $P_j = 1$ for $j < i$. That is, $P_{i/-} = 1$ if P_i is the rightmost product (i.e., lowest index) that is 1. At most one condition, $P_{i/-}$, holds in the abort network at any time, since at most one P_i is the rightmost product which is 1. Therefore, the probability $Prob_{on}$ that some $P_i = 1$ can be broken into a sum of disjoint cases

$$\begin{aligned} Prob_{on} &= Prob(P_1 + P_2 + \cdots + P_n) \\ &= Prob(P_{1/-} + P_{2/-} + \cdots + P_{n/-}) \\ &= Prob(P_{1/-}) + Prob(P_{2/-}) + \cdots + Prob(P_{n/-}) \end{aligned} \quad (3)$$

These probabilities, $Prob(P_{i/-})$, can be computed recursively. For the base case

$$Prob(P_{1/-}) = Prob(P_1) \quad (4)$$

since, whenever P_1 is 1, it is also the rightmost product which is 1.

Assume we have $Prob(P_{i/-})$ for all $i \le n$, for some n. We now derive $Prob(P_{n+1/-})$, which is the probability that P_{n+1} is the rightmost product which is 1. We first define a near-solution, X, and then adjust it

$$X = Prob(P_{n+1}) \cdot Prob(\overline{P_{n-1}} \cdot \overline{P_{n-2}} \cdots \overline{P_1}) \quad (5)$$

X is the probability that product P_{n+1} is 1 and all lower P_i are 0, $1 \le i \le n-1$ (ignoring $i = n$). These P_is are not adjacent to P_{n+1} (only P_n and P_{n+2} are adjacent); hence, by assumption, they do not overlap P_{n+1}. The above equation is correct, since $Prob(P_{n+1})$ is independent of the probability in the second clause. Therefore

$$\begin{aligned} X &= Prob(P_{n+1}) \cdot (1 - Prob(P_1 + P_2 + \cdots + P_{n-1})) \\ &= Prob(P_{n+1}) \cdot (1 - Prob(P_{1/-} + P_{2/-} + \cdots + P_{n-1/-})) \\ &= Prob(P_{n+1}) \\ &\quad \cdot (1 - Prob(P_{1/-}) - Prob(P_{2/-}) - \cdots - Prob(P_{n-1/-})) \\ &= Prob(P_{n+1}) \cdot \left(1 - \sum_{i=1}^{n-1} Prob(P_{i/-})\right) \end{aligned} \quad (6)$$

Note 4: This distinction is important only for augmented detection networks. For example, in network $(5p, 1\bar{k})$, product $P_1 = \overline{k_4}\, p_5p_6p_7p_8p_9$ and product $P_2 = \overline{k_8}\, p_9p_{10}p_{11}p_{12}p_{13}$ overlap in bits 8 and 9. Similarly, P_2 and P_3 overlap in bits 12 and 13. P_1 is broken into three segments: $P_{1,0} = \overline{k_4}p_5$, $P'_1 = p_6p_7$, and $P_{1,2} = p_8p_9$; and P_2 is broken into three segments: $P_{2,1} = \overline{k_8}p_9$, $P'_2 = p_{10}p_{11}$, and $P_{2,3} = p_{12}p_{13}$. In this case, the probabilities of $P_{1,2}$ and $P_{2,1}$ differ, since $P_{1,2}$ contains only p bits (p_8p_9), while $P_{2,1}$ contains both p and $\bar{k}$ bits ($\overline{k_8}\,p_9$: $Prob(P_{1,2}) = 0.75 \cdot 0.50 = 0.375$, and $Prob(P_{2,1}) = 0.5 \cdot 0.50 = 0.25$.

To compute $Prob(P_{n+1/-})$, X must be modified to account for the overlapping product P_n. In particular, X includes the case where the overlapped product, P_n, is 1. In this case, product P_n is 1 and the nonoverlapped segments, $P_{n+1,n+2}$ and P'_{n+1} of P_{n+1} are also 1. After removing this case, the result is the probability that P_{n+1} is the rightmost product which is 1

$$\begin{aligned} &Prob(P_{n+1/-}) \\ &= X - Prob(P_{n+11,n+2}) \cdot Prob(P'_{n+1}) \cdot Prob(P_{n/-}) \\ &= Prob(P_{n+1}) \cdot \left(1 - \sum_{i=1}^{n-1} Prob(P_{i/-}) \right. \\ &\qquad \left. - \frac{Prob(P_{n+1,n+2}) \cdot Prob(P'_{n+1})}{Prob(P_{n+1})} \cdot Prob(P_{n/-}) \right) \\ &= Prob(P_{n+1}) \cdot \left(1 - \sum_{i=1}^{n-1} Prob(P_{i/-}) - \frac{Prob(P_{n/-})}{Prob(P_{n+1,n})} \right) \end{aligned} \tag{7}$$

This result uses eqn. 2 to simplify the second line above. $Prob(P_{n+1})$ is the probability that product P_{n+1} is 1, and $Prob(P_{n+1,n})$ is the probability that the 'lower overlap' segment $P_{n+1,n}$ of product P_{n+1} is 1. Since each product is an identical function, we can simplify, arbitrarily replacing $Prob(P_{n+1})$ by the constant $k_1 \equiv Prob(P_j)$ and $Prob(P_{n,n+1})$ by the constant $k_2 \equiv Prob(P_{j,j-1})$, for some arbitrary j (e.g., $j = 1$). The final equation is

$$Prob(P_{n+1/-}) = k_1 \cdot \left(1 - \sum_{i=1}^{n-1} Prob(P_{i/-}) - \frac{Prob(P_{n/-})}{k_2} \right) \tag{8}$$

Eqn. 8 gives $Prob(P_{n+1/-})$ as a recurrence relation, defined in terms of n previously computed probabilities, $Prob(P_{i/-})$ ($1 \le i \le n$). An alternative, but equivalent, recurrence relation defines $Prob(P_{n+1/-})$ in terms of only two previously-computed probabilities, $Prob(P_{n/-})$ and $Prob(P_{n-1/-})$ (for $n \ge 3$)

$$\begin{aligned} Prob(P_{n+1/-}) &= \left(1 - \frac{k_1}{k_2}\right) \cdot Prob(P_{n/-}) \\ &\quad - \left(k_1 - \frac{k_1}{k_2}\right) \cdot Prob(P_{n-1/-}) \end{aligned} \tag{9}$$

This equation is derived from eqn. 8 by computing $Prob(P_{n+1/-})$ and $Prob(P_{n/-})$, subtracting $Prob(P_{n+1/-}) - Prob(P_{n/-})$, and simplifying.

As an example, network ($5p$, $0\bar{k}$) has $n = 6$ products, each with five propagate inputs: $P_1 = p_5p_6p_7p_8p_9$, $P_2 = p_9p_{10}p_{11}p_{12}p_{13}$, etc. Products overlap in 1-bit segments (e.g. p_9). The probability $Prob(P_i)$ that product $P_i = 1$ is $(0.5)^5 = 0.03125$. Since the overlap segment has 1 bit, the probability $Prob(P_{i,i+1})$ that the ith segment is 1 is $(0.5)^1 = 0.5$. By eqn. 5, $Prob(P_1/-) = Prob(P_1) = 0.03125$ is the probability that P_i is the rightmost product which is 1. Using eqn. 8

$$\begin{aligned} Prob(P_2/-) &= Prob(P_1) \cdot \left(1 - \frac{Prob(P_{1/-})}{Prob(P_{0,1})}\right) \\ &= 0.03125 \cdot \left(1 - \frac{0.03125}{0.5}\right) = 0.029297 \\ Prob(P_3/-) &= Prob(P_1) \cdot \left(1 - Prob(P_1) - \frac{Prob(P_{2/-})}{Prob(P_{0,1})}\right) \\ &= 0.03125 \cdot \left(1 - 0.3125 - \frac{0.029297}{0.5}\right) \\ &= 0.028442 \end{aligned}$$

Once the remaining probabilities are computed, they are added to compute $Prob_{on}$ (eqn. 3), which is the probability of an abort.

(ii) Computing $Prob_{off}$ Once $Prob_{on}$ is computed, the computation of $Prob_{off}$ is trivial: it is $Prob_{off} = 1 - Prob_{on}$. $Prob_{off}$ is the probability of no abort, and hence the probability of early completion.