Variable Latency Speculative Addition: A New Paradigm for Arithmetic Circuit Design

Ajay K. Verma AjayKumar.Verma@epfl.ch
Philip Brisk Philip.Brisk@epfl.ch
Paolo Ienne Paolo.Ienne@epfl.ch
Ecole Polytechnique Fédérale de Lausanne (EPFL)
School of Computer and Communication Sciences
CH-1015 Lausanne, Switzerland

Abstract

Adders are one of the key components in arithmetic circuits. Enhancing their performance can significantly improve the quality of arithmetic designs. This is the reason why the theoretical lower bounds on the delay and area of an adder have been analysed, and circuits with performance close to these bounds have been designed. In this paper, we present a novel adder design that is exponentially faster than traditional adders; however, it produces incorrect results, deterministically, for a very small fraction of input combinations. We have also constructed a reliable version of this adder that can detect and correct mistakes when they occur. This creates the possibility of a variable-latency adder that produces a correct result very fast with extremely high probability; however, in some rare cases when an error is detected, the correction term must be applied and the correct result is produced after some time. Since errors occur with extremely low probability, this new type of adder is significantly faster than state-of-the-art adders when the overall latency is averaged over many additions.

1 Introduction and Motivation

Binary addition is one of the most frequently used arithmetic operations. It is a vital component in more complex arithmetic operations such as multiplication and division. Researchers have established lower bounds on the delay and area of the adder [8]. In particular, An n-bit adder must have a delay $\Omega(\log n)$ and area $\Omega(n)$. Certain adders having delay and area of the same complexity as the ones mentioned above have been presented in literature: A *Ripple Carry Adder* requires a linear number of gates, and fast adders such as *Carry Look-Ahead Adders (CLA)*, *Prefix Adders* etc. have logarithmic delays. These bounds indicate that no reliable adder can be implemented with sub-logarithmic delay; however, unreliable adders can be implemented with sub-logarithmic delay. Unreliable adders could, for example, be used in the domain of cryptographic attacks; alternatively, reliable adders could be constructed from unreliable adders by augmenting them with additional circuitry for error detection and correction.

This paper has two major contributions: first, we design an extremely fast unreliable adder that produces correct results for the vast majority of input combinations. For brevity, we will call this adder an *Almost Correct Adder (ACA)*. The second contribution is to design a correct adder, called a *Variable Latency Speculative Adder (VLSA)*, which uses an ACA as a component. Similar in principle to speculative execution, the VLSA will produce the ACA result, and a signal indicating whether the result is correct or incorrect after a delay that is much shorter than the delay of a fast traditional adder. This result will be correct in the vast majority of cases. In the event that an error occurs, the error will be corrected, and the correct result will be produced several cycles later, assuming a pipelined implementation of the VLSA. The VLSA could be implemented in a purely combinatorial fashion as well, if desired.

Contrary to what one may expect, the first contribution is not futile. Applications do exist that can use an incorrect adder without compromising the final result. One such class of applications are those that attempt to deduce a conclusion by repeating some operations on many different inputs. If the conclusion is not sensitive to the result of the operation on any individual input, then the small percentage of incorrect results will not adversely affect the outcome, while speeding up the application significantly.

One such application domain, which occurs in cryptography, is *Ciphertext-Only Attacks*, in which an attacker has access to a large set of encrypted text. The attack is successful if the private key is deduced, or if the encrypted plaintext is retrieved. The most commonly ciphertext-only attacks rely on a frequency analysis, i.e., the fact that in a normal text of any language certain characters and combinations of characters occur more frequently than others. For example, in English, character 'e' occurs with 12.7% frequency, while character 'x' occurs with 0.15% frequency. The attacker deduces a key by first pruning the set of potential keys, and then exhaustively enumerates the decryption procedure using each of the potential keys. Any key for which the deciphered text has a frequency of characters that is similar to what is expected, the key is considered to be valid, and is then analysed using more sophisticated methods.

The success and speed of such an attack depends on the amount of ciphertext provided and the number of keys tried. Speeding up the decryption process will significantly reduce the runtime of an attack. Many encryption algorithms first divide the plaintext into fixed-size blocks, which are then encrypted individually with the same key. Similarly, in decryption decrypted blocks are concatenated to get the plaintext. Thus, the incorrect decryption of an individual block in a large corpus of text is unlikely to reduce the overall efficacy of the attack due to the fact that an individual block cannot change the frequencies of characters significantly. If

978-3-9810801-3-1/DATE08 © 2008 EDAA

an ACA is used in place of a full adder in the context of such an attack, decryption, with an extremely high probability of success, could be performed significantly faster. A few blocks may be decrypted wrongly; however, once the correct key has been identified, a correct adder can be used to fix any incorrect blocks of text.

The second contribution, meanwhile, is relevant to general applications where correct results are demanded. The VLSA uses an ACA to produce the sum, but it also checks if the output of the ACA is correct. If the sum is incorrect, an error recovery mechanism is invoked to correct the sum. This occurs with extremely low probability, so the average latency of the VLSA is practically the same as an ACA.

The remainder of the paper focusses on the design of the ACA and VLSA. The next section discusses related work on fast adders. The following section formally introduces the ACA, and analyses its performance and error probability. Section 4 discusses how to build a reliable VLSA using the ACA. Last, we present results in Section 5 followed by conclusions and future work.

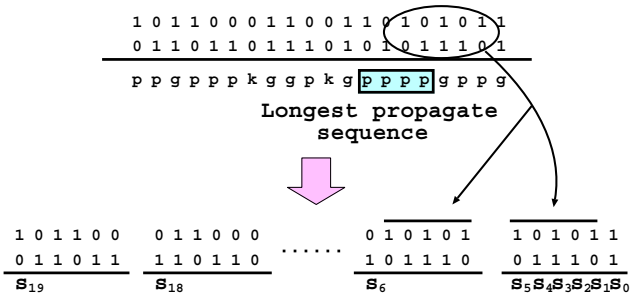

Figure 1. An example showing two 20-bit integers and the corresponding signals at each bit position. Since there is no sequence of more than 4 propagates, the sum bit at any position depends only on the inputs bits of 6 preceding bit positions.

2 State of the Art

Depending on the performance metric, several designs have been proposed for binary addition. The *Ripple Carry Adder* [11] is the smallest adder, however it is significantly slower compared to other fast adders. On the other hand, the *Carry Look-Ahead Adders* based on the parallel prefix computation [13, 1, 7] are the fastest one. These adders focus on minimising logic levels [13], maximum fanout [7], and maximum wire tracks [1]. The adders such as the one presented in [6] consider the trade-offs between conflicting performance metrics. In a recent work, Liu *et al.* [9] formulated the problem of finding Pareto-optimal prefix adders as an integer linear programming problem to solve the problem optimally.

However, all these adders are reliable adders and work correctly on all instances of inputs—hence, they cannot overcome the minimum theoretical delay and area bounds [8]. One possible way to reduce the complexity of addition is to use a redundant number system [11]. However, the conversion from a redundant number system to the binary number system itself is delay-expensive, which means that a redundant number system is typically useful only for multi-input addition.

Some work has been done on using probabilistic arithmetic components made of probabilistic gates to save energy [3]. Ernst *et al.* [2] presented a novel method to save power; it detects and corrects circuit timing error dynamically in order to tune the processor supply voltage. In a similar work by Hegde and Shanbhag [5] error detection and correction methods have been introduced to amend the errors due to the reduction in power supply voltage beyond critical limits. However, these methods are primarily focused on energy saving. In our work, we use the error detection and correction methods to improve the critical path delay without introducing computation errors. A somewhat similar approach has been deployed by Nowick [10] to improve the latency of an asynchronous adder; however, in this paper we target a synchronous design.

3 Main Idea and Analysis

This section introduces the unreliable ACA. Throughout the paper, binary integers are denoted by uppercase letters, e.g., A, B, X etc.; the i^{th} least significant bit of an integer X is denoted by x_{i-1}. In order to add two n-bit integers A and B, one can define generate, propagate and kill signals at each bit position as follows:

$$g_i = a_i b_i,$$
$$p_i = a_i \oplus b_i, \text{ and}$$
$$k_i = \overline{a_i + b_i}.$$

Using these signals the carry output c_i at each bit position i is generated and is used to compute the sum bits. The recurrence for c_i is shown below.

$$c_i = \begin{cases} 0 & \text{if } k_i = 1, \\ 1 & \text{if } g_i = 1, \\ c_{i-1} & \text{otherwise } (p_i = 1). \end{cases}$$
$$s_i = a_i \oplus b_i \oplus c_{i-1}.$$

Note that the carry bit c_i depends on the carry bit c_{i-1} only if the propagate signal p_i is true, otherwise c_i can be determined locally based on the values of g_i and k_i. Similarly, c_{i-1} depends on c_{i-2} only if p_{i-1} is true. This means c_i depends on c_{i-2} only if both p_i and p_{i-1} are true. In general c_i will depend on c_{i-k} only if every propagate signal between bit position i and $i-k+1$ (inclusively) is true.

If an oracle provides us with the longest sequence of propagate signals in advance, then an extremely fast adder could be constructed. For example, in Fig. 1 we want to add two 20-bit integers. Since the longest sequence of propagate signals is 4, the carry output c_i at bit position i will be independent of c_{i-5}. Hence, s_i can be computed only using the input bits of 6 preceding bit positions starting from i^{th} bit position. In other words, we can form several 6 bit adders, each computing the carry-in and sum bit for a particular bit position as shown in Fig. 1. The delay of this particular 20-bit adder will be virtually the same as that of a 6-bit adder.

Ideally, one would like to know the longest sequence of propagate signals in the input addenda. There are, in fact, no bounds on the length of the longest propagate sequence. In extreme cases such as for integers $A = 11\ldots1$, $B = 00\ldots0$ the length of the longest propagate sequence is the same as the bitwidth of A and B. However, in the next section, we show that on average, the length of the longest propagate sequence is approximately $\log n$, where n is the bitwidth of the integers.

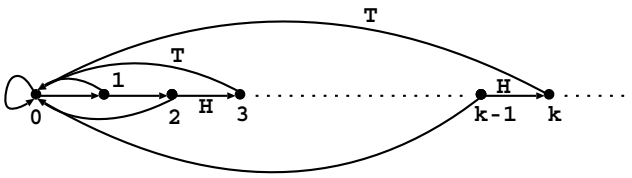

Figure 2. The infinite line graph where at each step a coin is tossed, the head leads to next node, while tail leads to node 0.

Bitwidth	Longest run of 1's with 99% probability	Longest run of 1's with 99.99% probability
64	11	17
128	12	18
256	13	20
512	14	21
1024	15	22
2048	16	23

Table 1. Bounds on the longest run of 1's with high probability.

3.1 Longest sequence of propagates

Here we derive the probabilistic bounds on the longest sequence of propagates, that occur in the addition of two binary integers A and B. Since $p_i = a_i \oplus b_i$, the length of the longest sequence of propagates is the same as the longest run of 1's in $A \oplus B$. It can also be shown that if two n-bit integers A and B are chosen randomly, then their XOR has a uniform distribution over $\{0,1\}^n$. Combining the two statements we can deduce that proving the bounds on the longest sequence of propagates in integer addition is equivalent to prove the bounds on the longest run of 1's in an n-bit integer, or equivalently, the longest run of heads in n independent random unbiased coin flips.

Next we need to find bounds on the longest run of heads in n independent random coin tosses. The next theorem indicates that on average the longest run of heads in random coin tosses is approximately $\log n - 1$.

Theorem 1 *In order to achieve a run of k heads one must flip a fair coin at least $2^{k+1} - 2$ times on average.*

Proof Consider the following experiment involving a walk on an infinite line graph. Consider the infinite line graph shown in Fig. 2, where each node i has exactly two outgoing edges: one to node $(i+1)$, and other to node 0. Suppose a person starts at node 0. At each step the person flips a coin and depending on the outcome, he or she chooses one of his two outgoing edges to take. In case of head, the person advances to the next node; otherwise, the person returns to node 0.

The person will reach node k only after having a run of k heads. We must prove that the average number of steps taken to reach node k for the first time is $2^{k+1} - 2$. Let us denote the average number of steps taken to reach node k by T_k.

Since the only incoming edge to node k is from node $k-1$; hence, T_k is equal to T_{k-1} plus the average number of steps to advance to node k from node $k-1$. After reaching node $k-1$, the person moves to node k or node 0 with equal probability. In the first case the number of steps taken to advance to node k from $k-1$ is 1, however in the second case the number of steps required is $1 + T_k$ (one step to reach node 0 and T_k steps to reach node k from node 0). This means the average number of steps taken from node $k-1$ to node k is the average of 1 and $1 + T_k$. In other words,

$$T_k = T_{k-1} + \frac{1 + (1 + T_k)}{2}.$$

The solution to the recurrence is $T_k = 2^{k+1} - 2$, completing the proof. □

Theorem 1 indicates that the minimum number of coin tosses required to achieve a run of k heads is exponential in k on average. Hence, one should expect that the longest run of heads in n coin tosses will be logarithmic in n on average. This does not, however, allow us to conclude anything about the distribution of the longest run.

Schilling [12] proved that the expected length of the longest run of heads in n random, independent, unbiased coin flips is $\log n - 2/3$ with variance 1.873. In n random coin tosses if the number of instances where longest run of head does not exceed x is denoted by $A_n(x)$, then

$$A_n(x) = \begin{cases} 2^n & \text{if } n \leq x, \\ \sum_{0 \leq j \leq x} A_{n-1-j}(x) & \text{otherwise.} \end{cases}$$

This recurrence relation does not have an elegant closed form; however, we can use a computer program to compute $A_n(x)$ for given values of n and x. We used this recurrence to find the fraction of integers where the longest run of 1's is bounded by some constant. Table 1 shows the upper bound on the longest run of heads for most of the instances (e.g., 99%, 99.99%) in n random coin tosses.

Table 1 shows that the longest run of 1's in an n-bit integer is less than under $\log n + 12$ with extremely high probability. Gordon *et al.* [4] also showed that the probability approaches to 1 exponentially fast when the bound on the longest run of 1's is increased (e.g., in Table 1 the error probability reduces from 1% to 0.01% just by increasing the bound by 7).

The consequence is that when adding two integers, the carry propagates only a small way in the vast majority of cases. In case of a 1024-bit adder the largest carry propagation is under 22 bits in 99.99% cases. In other words, if we implement a 1024 bit adder using several 24-bit adders similar to Fig. 1, then the result produced will be correct in 99.99% of cases.

3.2 Area Overhead

One concern about a circuit implementation of the ACA is the area overhead due to a multitude of small adders. As shown in Fig. 1, to implement a 20-bit adder, fifteen 6-bit adders are required. This results in large fanout of primary inputs as well as a fairly large area. This section, shows how to effectively share the computation blocks among the small adders in order to conserve area and reduce the fanout on primary inputs.

We do not need all the bits of the small adders but only the most significant bit. Thus we must determine the exact functionality of each small adder block. Based on the recurrence $c_i = g_i + p_i c_{i-1}$,

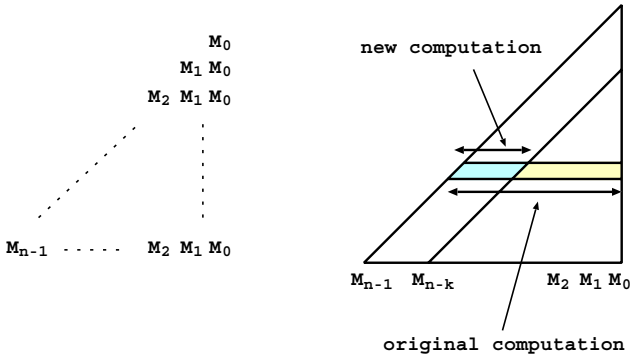

Figure 3. A comparison of the computations of the traditional adder and of our fast adder.

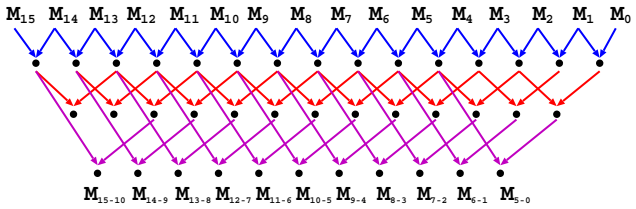

Figure 4. An example illustrating how to implement the new adder with shared logic.

the expression for the carry output c_i can be written as a matrix multiplication:

$$\begin{pmatrix} c_i \\ 1 \end{pmatrix} = \begin{pmatrix} p_i & g_i \\ 0 & 1 \end{pmatrix} \begin{pmatrix} c_{i-1} \\ 1 \end{pmatrix} = M_i \begin{pmatrix} c_{i-1} \\ 1 \end{pmatrix}, \text{ and hence,}$$

$$\begin{pmatrix} c_i \\ 1 \end{pmatrix} = M_i M_{i-1} \cdots M_{i-k+1} \begin{pmatrix} c_{i-k} \\ 1 \end{pmatrix}.$$

Note that the matrix multiplication described above is based on simple OR and AND operations. In a traditional adder, one is supposed to compute the product $M_i M_{i-1} \cdots M_0$ in order to compute the carry bit c_i. However, in our adder we assume that no carry propagates more than k-bit position for some k. Hence, in our adder, one needs to compute only the product $M_i M_{i-1} \cdots M_{i-k+1}$ to compute the carry bit c_i. This is denoted as a triangle in Fig. 3, where the matrices are written horizontally and each row in the triangle correspond to the product of matrices beneath the row. However, the new adder computes only product of k matrices as shown by a slanted strip of width k in Fig. 3. In each of the small adder blocks of Fig. 1 one of these matrix product is computed.

Hence, the goal is to compute the products of k matrices with minimal area, without increasing the critical path delay. First, we compute the product of two consecutive matrices, i.e., $M_1 M_0$, $M_2 M_1$, …, $M_{n-1} M_{n-2}$. Second, these products are re-used to get the products of 4 consecutive matrices, e.g., $M_3 M_2 M_1 M_0 = (M_3 M_2)(M_1 M_0)$. The same process is repeated for $\lfloor \log k \rfloor$ steps to compute the products of all $2, 4, 8, \ldots, 2^{\lfloor \log k \rfloor}$ consecutive matrices. In the last step some of these products are multiplied to compute the final products of all k consecutive matrices. This process is shown in Fig. 4 for $n = 16$ and $k = 6$.

Each step in the above process computes the product of two matrices at most n times, and there are $\log k + 1$ steps. Thus the total number of matrix multiplications is $O(n \log k)$. Since k is of complexity $O(\log n)$, the space complexity of this adder is $O(n \log \log n)$, which is near-linear, even for sufficiently large values of n. Since each intermediate matrix is used at most 3 times, all gates will have bounded fanout.

The resulting ACA is slightly larger than a ripple carry adder, and is exponentially faster than fast traditional adders. However, it may produce incorrect result on a very small fraction of inputs. In the next section we augment the ACA with error detection and correction mechanisms.

4 Average Fast Exact Adder

Although the ACA is ideal for applications such as cryptographic attacks, the user may also want to know whether or not the computed sum is correct. The incorrect sums can then be ignored and recomputed with a slower traditional adder. Alternatively, the ACA can be augmented with error detection and correction mechanisms, ensuring that the final result is correct.

4.1 Error Detection

This section presents a circuit that flags an error if the sum computed by the ACA is incorrect. This only occurs when there is a chain of more than k propagates in the addenda. To check for the presence of an error, we must consider all chains of length $k + 1$, and check if any of them contain solely propagates. The expression for error signal is stated as follows:

$$ER = \sum_{i=0}^{n-k-1} p_i p_{i+1} \cdots p_{i+k}.$$

The critical path delay to compute error signal has complexity $O(\log k + \log(n-k))$. Since $k = O(\log n)$, the error signal complexity can be reduced to $O(\log n)$. The critical path delay of error detection has the same complexity as that of the critical path delay of a traditional adder; however, the error detector only requires simple gates, such as AND, OR, etc., which are faster than the complex gates such as OR-AND gates used to compute expressions such as $g + pc$ used in traditional adders. Experimentally, we report that the delay of the error detection signal is approximately two-thirds of the delay of a traditional adder.

4.2 Error Recovery

Once an error has been detected, one could simply employ a traditional correct adder to produce the sum. Instead, we have developed a novel error recovery technique that uses a computation inside the ACA to reduce both the critical path delay and hardware area.

The matrix product $M_i M_{i-1} \cdots M_{i-k+1}$ in Section 3.2 computes the propagate and generate signals for the block between bit position i and $i - k + 1$. Thus, the ACA computes the propagate and generate signals for each k-bit block. If we divide the input integers into n/k blocks of k-bits, the values of propagate and generate for each block can be taken from the ACA. An n/k-bit carry look-ahead adder then takes these values and computes the carry for each of the blocks, as shown in Fig. 5. Meanwhile, we compute the propagate and generate signals for each bit in a block

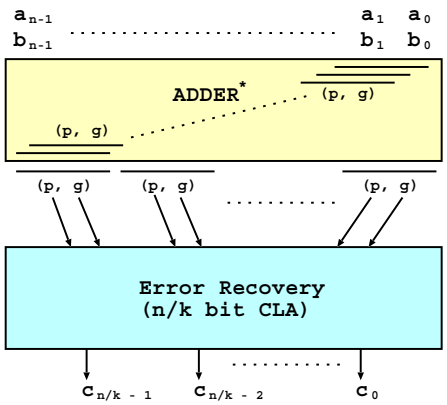

Figure 5. Significant amount of computation of the fast adder can be reused for error recovery.

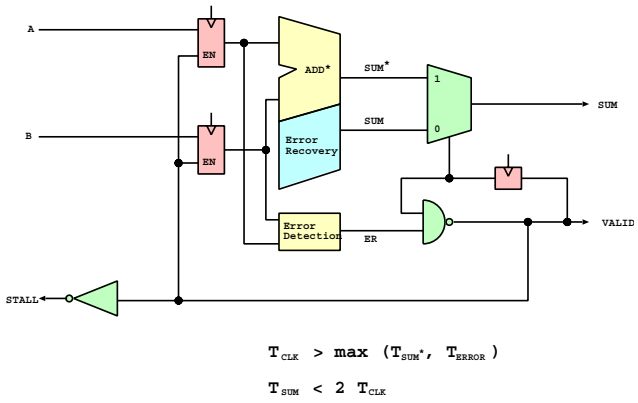

Figure 6. An implementation of the variable latency adder. Almost in all cases the adder will have a latency of one cycle. However, seldom it will require two cycles to compute the result.

other than the last bit, which was already computed by the ACA. The carry bits computed by carry look-ahead adder can be used to compute the correct sum.

The overhead of error recovery is similar to carry look-ahead addition, except for the fact that propagate and generate signals are computed by the ACA. Since there is no guarantee that k is the optimal group size, the process of error recovery may be slower than a traditional adder; however our experiments shown that the delay of ACA addition plus error recovery is approximately the same as a traditional adder.

Since the critical path of error recovery is the same as the delay of a fast carry look-ahead adder, the proposed speculative adder has virtually no advantage when implemented in a combinatorial circuit. On the other hand, this adder could be used inside a processor: ACA additions and error/no-error signals are quickly produced in a single cycle. In the vast majority of cases, there is no error and the correct result is produced quickly. In the rare event of an error, the processor must wait an additional cycle or two to receive the corrected sum. Since errors are extremely rare, the

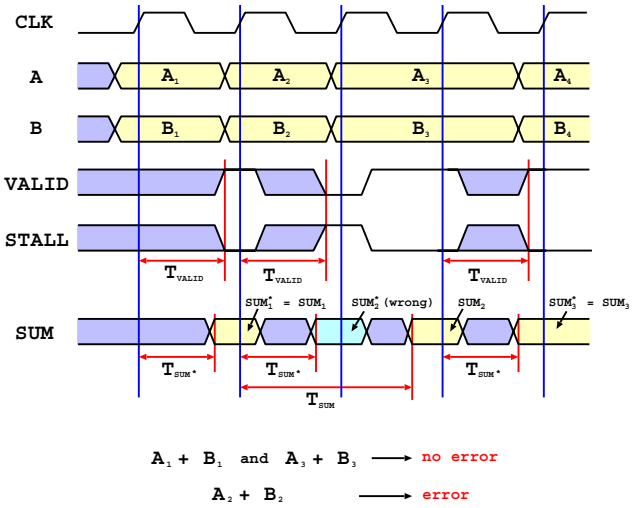

Figure 7. The timing diagram showing the execution of variable latency adder. In this example, on the first and third operands it produces the correct output, while on the second operand it produces the incorrect sum and then corrects it.

average time to produce a correct sum will be comparable to the ACA. In the next section we explain how a variable latency adder can be designed.

4.3 Variable Latency Speculative Adder

In Section 5, we observe that for 64 bits, the delay of the ACA and error detection mechanisms are approximately equal and, individually, both are significantly less than the delay of a traditional adder. Based on this observation, we have designed the circuit shown in Fig. 6 whose clock period is slightly greater than the critical path delay of the error detection circuit.

After one cycle, the circuit produces the result of the ACA and a bit indicating whether or not an error has been detected. If there is no error, then the circuit provides SUM^* as its output and will also set the $VALID$ bit to 1. Since $STALL$ is the complement of valid, the circuit will be ready to accept a new set of input addenda. If an error occurs, the valid bit will be set to 0 and the stall bit will be set to 1. After two cycles, the corrected sum value will be available and the valid bit will be set to 1. At this point, the circuit is ready to accept new inputs. We call this type of adder a *Variable Latency Speculative Adder (VLSA)*.

The timing diagram of the VLSA is shown in Fig. 7. In this case, the ACA produces the correct sum for the first and third pairs of integers that are summed. It produces the wrong sum and then corrects it on the second pair of inputs. Since the ACA produces a correct sum in more than 99.99% of all cases, the average latency will be 1.0001 cycles. The effective latency of the circuit is almost half of the latency of the fastest traditional adder.

5 Experimental Results

We have written a C++ program which takes the value n as input and generates VHDL files corresponding to the circuit of

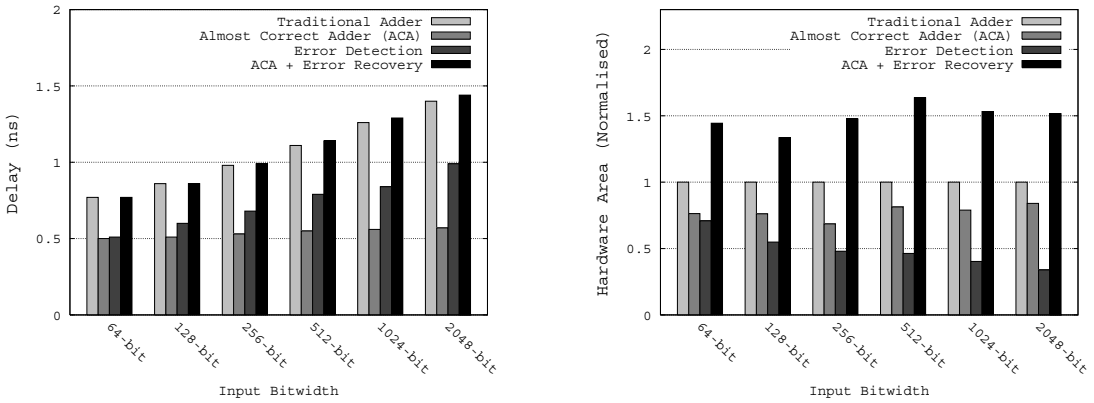

Figure 8. Comparison of delay and area of the new adders with respect to a traditional adder.

ACA (the one with 99.99% accuracy), error detection, and error recovery. Note that the error recovery uses ACA as discussed in Section 4.2. The three circuits are synthesised using a common standard-cell library for UMC $0.18\mu m$ CMOS technology. We compare the delays and hardware areas of these circuits with the library implementation of fast adders provided by DesignWare. We have also implemented a carry look-ahead adder and compared its delay with the adder provided by DesignWare, and observed that the adder provided by DesignWare is slightly faster.

Fig. 8 compares the delay and hardware area of the DesignWare adder with the delays and hardware areas of the circuits generated by our algorithm. As we can see, the circuit for ACA has a speedup of 1.5–$2.5\times$ over the DesignWare adder and is 25% smaller. Similarly, the circuit for error detection signal has a critical path delay almost $2/3$ of the critical path delay of the DesignWare adder. This is due to the fact that both circuits have $O(\log n)$ levels of logic, although each level in the latter is more complex.

Finally, we compare the error recovery circuit with the DesignWare adder. Note that the error recovery circuit contains an ACA too, this is why it has a larger area. Although the delay of the error recovery and the DesignWare adder is almost same, but the error recovery circuit is used only on a minuscule fraction of inputs. This means that, on average, the effective delay of the VLSA, which is composed of ACA, error detection, and error recovery, is almost the same as the delay of error detection. In other words, on average VLSA has a $1.5\times$ speedup over the DesignWare adder.

6 Conclusions and Future Work

We have presented an adder design which works correctly in almost all the instances of input integers. The adder presented here is exponentially faster compared to reliable adder. Our experiments indicate a speedup of 1.5–$2.5\times$. This design is particularly useful in computation-intensive applications which are robust to small errors in computation. We also present a methodology to detect the errors and correct them. This results in a fast variable latency adder, which has a speedup of $1.5\times$ on average on the traditional fast adder. As a future work, we plan to design fast *almost correct design* for other arithmetic components such as multipliers, multi-input adders, etc.

References

[1] R. P. Brent and H. T. Kung. A regular layout for parallel adders. *IEEE Transaction on Computers*, C-31(3):260–64, 1982.

[2] D. Ernst, N. S. Kim, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, and K. Flautner. Razor: Circuit-level correction of timing errors for low-power operation. *IEEE MICRO special issue*, Mar. 2005.

[3] J. George, B. Marr, B. E. S. Akgul, and K. V. Palem. Probabilistic arithmetic and energy efficient embedded signal processing. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 158–68, 2006.

[4] L. Gordon, M. F. Schilling, and M. S. Waterman. An extreme value theory for long head runs. In *Probability Theory and Related Fields*, pages 279–87, 1986.

[5] R. Hegde and N. R. Shanbhag. Soft digital signal processing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, VLSI-9(6):813–23, Dec. 2001.

[6] S. Knowles. A family of adders. In *IEEE Symposium on Computer Arithmetic*, pages 277–81, 2001.

[7] P. M. Kogge and H. S. Stone. A parallel algorithm for the efficient solution of a general class of recurrence. *IEEE Transaction on Computers*, C-22(8):783–91, 1973.

[8] I. Koren. *Computer Arithmetic Algorithms*. Prentice-Hall Inc., New Jersey, 1993.

[9] J. Liu, Y. Zhu, H. Zhu, C. K. Cheng, and J. Lillis. Optimum prefix adders in a comprehensive area, timimg and power design space. In *Proceedings of the Asia and South Pacific Design Automation Conference*, pages 609–15, 2007.

[10] S. M. Nowick. Design of a low-latency asynchronous adder using speculative completion. In *IEE Proceedings on Computers and Digital Techniques*, Sept. 1996.

[11] B. Parhami. *Computer Arithmetic: Algorithms and Hardware Design*. Oxford University Press, New York, 2000.

[12] M. F. Schilling. The longest run of heads. *The college Mathematics Journal*, pages 196–207, 1990.

[13] J. Sklansky. Conditional sum addition logic. *IRE Transaction on Electronic Computers*, EC-9:226–31, 1960.