Self-Timed Multi-Operand Addition

P. Balasubramanian, D. A. Edwards, and W. B. Toms

***Abstract*—**Self-timed addition of multiple data operands is discussed in this paper. Though there are various works in the existing literature targeting dual-operand addition, multiple operand addition has not been exclusively dealt with. In this context, this paper throws light on two important concepts – i) presenting a bit-partitioning scheme that divides input data into groups where additions within the individual groups are carried out in parallel, and ii) proposing novel and efficient (4:2) logic compressor realizations corresponding to weak-indication and robust early output timing regimes. An analysis of the efficiency of addition for a significant case study involving 8 input data, each of size 32-bits, is performed with carry save adders or logic compressors employed for the input field partitions. The simulation results show the proposed early propagative compressor design effectively optimizing the power-delay-area design envelope.

***Keywords*—** Self-timed, Multi-input addition, Carry save adder, Logic compressor, Indication, Early propagation, Standard cells.

I. Introduction

RELIABILITY is labelled as one of the five crosscutting design challenges in the Semiconductor Industry Association's 2008 international technology roadmap on design [1], which drives home the point that 'robustness' is becoming an increasing priority for digital logic design in ultra deep submicron technologies. In this scenario, self-timed design attracts attention on account of its inherent capability to tolerate supply voltage, process parameter and temperature variations [2]. Due to the absence of a global clock reference, self-timed circuits exhibit better noise and electro-magnetic compatibility properties compared to their synchronous counterparts [3]. In addition, they are modular permitting convenient design reuse [4], which is important since design reuse as a percentage of overall logic is expected to be 55% by 2020 [1].

This paper deals with self-timed addition of multiple input operands based on a bit-partitioning scheme that utilizes either carry save adders (CSAs) or logic compressors for the input partitions. Nevertheless, the focus is on novel synthesis of asynchronous logic compressors pertaining to weak-indication and early output timing regimes. To the best of our knowledge, this article is the first work dealing exclusively with self-timed addition of multiple data operands. The remainder of this paper is organized as follows. Section 2 briefly summarizes the various timing models adopted, discusses the attributes of a function block and describes a widely used robust asynchronous signaling convention viz. the 4-phase handshaking. Various logic tree structures available for multi-operand addition are discussed briefly in Section 3. Next, a bit-partitioning strategy that parallelizes the addition of multiple operands of arbitrary size is illustrated in Section 4 that utilizes either CSAs or logic compressors for the input field partitions. In Section 5, an evaluation of self-timed addition involving multiple data operands is performed by considering a significant case study of addition of 8 input data, each of size 32-bits. The efficiency of CSAs and compressors for this multi-operand addition scenario is evaluated on the basis of power, delay and area. Finally, the concluding remarks are made in Section 6.

Manuscript received December 29, 2011: Revised version received. This work was supported in part by the Engineering and Physical Sciences Research Council, UK under Grant EP/D052238/1. The first author was additionally supported by a bursary from the School of Computer Science of the University of Manchester, UK.

P. Balasubramanian was with the University of Manchester, UK. He is now with the Department of Electronics and Communication Engineering, Vel Tech Dr. RR and Dr. SR Technical University, Avadi, Chennai 600 062, Tamil Nadu, India (phone: +91-(0)44-2684 1601; fax: +91-(0)44-2684 0262; e-mail: spbalan04@gmail.com).

D. A. Edwards and W. B. Toms are with the School of Computer Science, University of Manchester, Oxford Road, Manchester M13 9PL, UK (e-mail: doug@cs.man.ac.uk; tomsw@cs.man.ac.uk).

II. Fundamentals of Input/Output Mode Circuits

A. Timing Models

The following circuit models adhere to input/output mode, with no timing assumptions imposed on when the environment should respond to the circuit – a) delay-insensitive (DI), b) quasi-delay-insensitive (QDI), and c) speed-independent (SI).

A DI circuit guarantees correct normal operation irrespective of the delays of its gates and the delays encountered in the communicating signal wires, i.e. unbounded (arbitrary, but positive and finite) gate delay and wire delay models are considered. This is the most robust of all unbounded delay models and such circuits are guaranteed to be *correct by construction*. It was shown in [5] that C-elements and inverters are the only DI elements and so unfortunately, the class of pure DI circuits would be very limited when considering only these two logical operators.

DI circuits with isochronic fork assumptions [5] are referred to as QDI circuits; it is not necessary that every fork should be an isochronic fork in a QDI circuit. The isochronic fork assumption has been defined in [5] as follows: "In an isochronic fork, when a transition on one output is acknowledged, and thus completed, the transitions on all outputs are acknowledged, and thus completed". A recent work by Martin et al. [6] shows that the main building blocks of QDI logic, including realization of the isochronicity assumption, can be successfully implemented even in nano-CMOS technologies where stricter design rules and larger parametric variations could be anticipated. This is an encouraging pointer towards the feasibility of the QDI design paradigm in the nano-CMOS era. Similar to the DI circuit, the QDI circuit conforms to unbounded delay model for gates and wires but with the exclusion of isochronic forks.

A SI circuit operates correctly regardless of gate delays; wires are assumed to have no or negligible delay – hence, unbounded gate delay and bounded wire delay. Every fork is assumed to be an isochronic fork in a SI logic circuit. Technically, wire delays are typically accounted for in the components (gates) according to this timing model and consequently wires are assumed to be ideal (i.e. zero delay).

Referring to the circuit fragment in figure 1(a), d_{g1}, d_{g2} and d_{g3} represent the propagation delay of gates *g1, g2* and *g3* respectively, while d_{w1}, d_{w2} and d_{w3} signify the delay values of corresponding nets. For the DI delay model, d_{g1}, d_{g2}, d_{g3}, d_{w1}, d_{w2} and d_{w3} can be arbitrary, while in case of the QDI delay model; d_{w2} is assumed to be equal to d_{w3} with node *f* being labelled as an isochronic fork junction. According to the SI timing delay model, $d_{w1} = d_{w2} = d_{w3} = 0$, but the wire delays are accounted for in the delay of gate *g1*, whose output acts as an input for gates *g2* and *g3*. Hence the delay of gate *g1* is modeled as $(d_{g1}+d_{w1}+d_{w2})$ or $(d_{g1}+d_{w1}+d_{w3})$ as shown in figure 1(b).

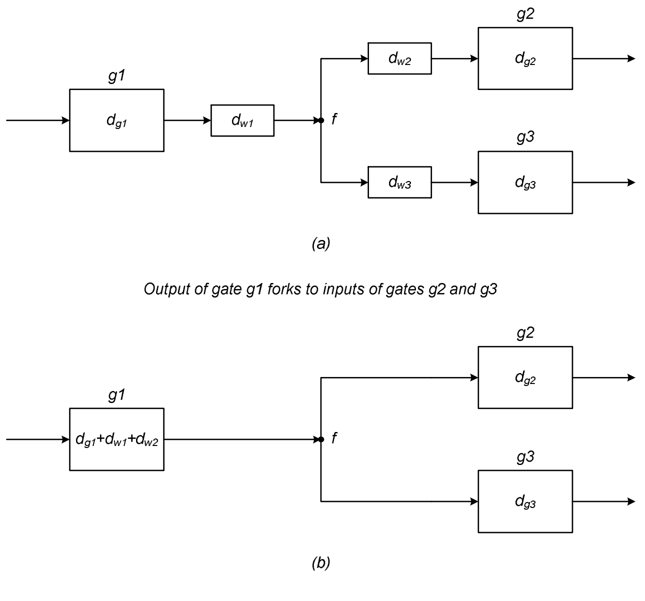

Fig. 1 Illustration of DI, QDI and SI delay models

B. Function Block

Seitz classified the *function block*, which is the asynchronous equivalent of a synchronous combinational logic circuit into two robust categories based on their indicating (acknowledging) mechanism as *strongly indicating* or *weakly indicating* [7]. A strong-indication function block waits for all of its inputs (valid/spacer) to arrive before it starts to compute and produce any output (valid/spacer). On the other hand, a weak-indication function block starts to compute and produce outputs (valid/spacer) even with a subset of the inputs (valid/spacer). However, Seitz's weak timing specifications require that at least one output (valid/spacer) should not have been produced until after all inputs (valid/spacer) have arrived. Given these, when small indicating function blocks are interconnected to compose a larger indicating function block, such as cascading of full adder modules to construct an *n*-bit adder, weakly indicating realizations are preferred compared to strongly indicating ones. This is because the former's performance is data-dependent while the latter's performance is always bound by worst-case latency. The signaling scheme for strong and weak-indication timing regimes in terms of their input and output behavior is shown in figure 2.

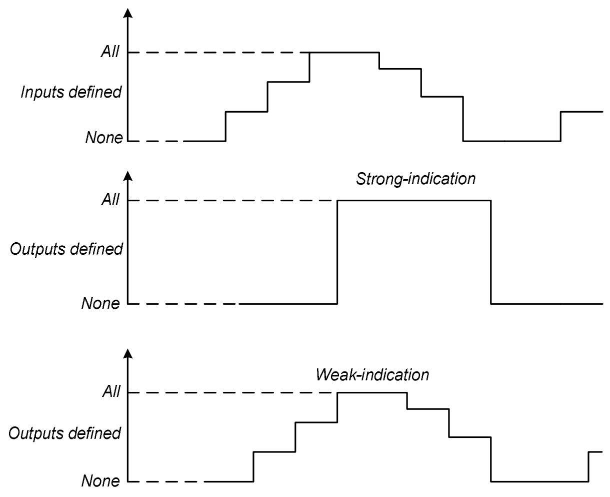

Fig. 2 Portraying strong and weak indication timing constraints

Function blocks can also be non-indicating at the expense of being non-robust. The dual-rail combinational logic style [8] [9] of realizing function blocks belongs to this category. The dual-rail combinational logic (DRCL) style utilizes De-Morgan's theorems of Boolean algebra to implement a combinational logic circuit in an asynchronous style by replacing each gate by its dual-rail equivalent (dual-rail pair). For example, given a logic function $F = ab + cd$, the dual-rail equivalent expressions are specified as: $F1 = a1b1 + c1d1$ and $F0 = (a0 + b0)(c0 + d0)$. The gate level realization of the dual-rail combinational equivalent of *F* is shown below.

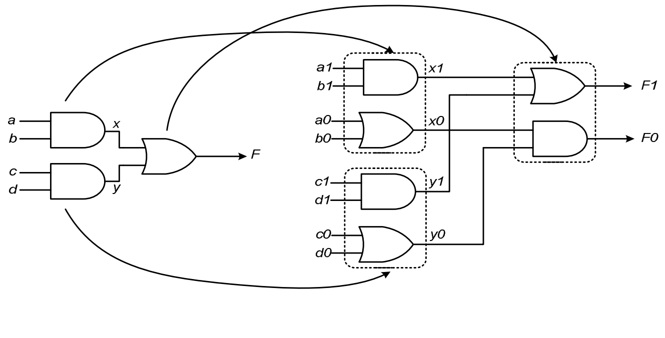

Fig. 3 Dual-rail combinational equivalent realization of $F = ab + cd$

Let us consider two scenarios corresponding to the dual-rail combinational equivalent of a Boolean function, F, to clarify the necessity for ensuring proper indication of signal events at the primary inputs as well as at the intermediate output nodes, and to describe how *wire* and *gate orphans* could possibly arise. Assuming all data inputs to be currently spacers (zeroes), when *a0* and *c0* become defined (logic 1), intermediate signals *x0* and *y0* would become defined and eventually *F0* would become defined. Assuming that *b0* and *d0* also become defined subsequently, these transitions would not be acknowledged by the intermediate signals (*x0* and *y0*) or by the corresponding output in the present evaluation phase resulting in wire orphans. Let us assume that *a1* and *b1* become defined after a return-to-zero phase. This would lead to defining of the intermediate signal *x1*. Assuming that *c1* and *d1* also become defined subsequently during the current evaluation phase, *F1* could have become defined as a result of *x1* alone becoming defined, and hence a late transition on *y1* would not be acknowledged by the primary output giving rise to a gate orphan. From the preceding discussions, it should be clear that the DRCL realization is non-indicating and it conforms to *eager evaluation* owing to the fact that even with a subset of the function block inputs becoming defined/undefined all of the function block outputs could become defined/undefined regardless of the late arriving inputs. Hence the DRCL style is not strongly or weakly indicating but is *early propagative*, i.e. early set and/or reset could occur. Therefore, great care should be taken to circumvent the problem of orphans that could arise in an early output circuit. However, this can be tackled at both the technology-independent and technology-dependent logic optimization stages. Nevertheless, early output function blocks are generally faster than their input-complete counterparts.

Robust function block designs adhere to a 4-phase handshaking convention for simplicity of implementation and can employ any DI data-encoding scheme, with the dual-rail data-encoding scheme being widely preferred. In this scheme, each data wire d is represented using two data wires, d^0 and d^1, with the request signal embedded within the data wires. A low-to-high transition on the d^0 wire indicates that a *zero* has been transmitted, while a low-to-high transition on the d^1 wire indicates that a *one* has been transmitted. Since the request is embedded within the data wires, a transition on either d^0 or d^1 informs the receiver about the validity of the data. The condition of both d^0 and d^1 being a zero at the same time is referred to as the *spacer* (empty state). Both d^0 and d^1 are not allowed to transition simultaneously as it is illegal and invalid, since the coding scheme is *unordered*, i.e. no code word is a subset of another code word.

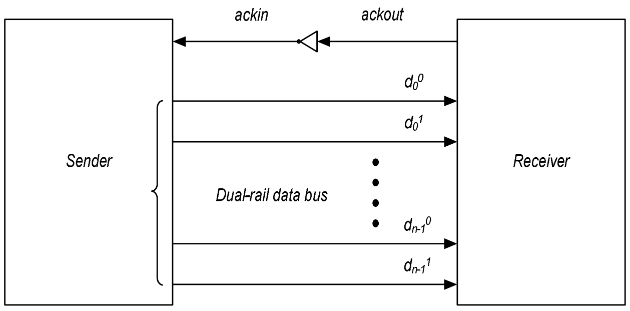

Fig. 4 Dual-rail data encoding and 4-phase handshaking

Referring to the figure 4, the 4-phase handshake protocol is explained as follows[1]:

- The dual-rail data bus is initially in the spacer state. The sender transmits the code word (valid data). This results in 'low' to 'high' transitions on the bus wires, which correspond to non-zero bits of the codeword
- After the receiver receives the codeword, it drives the *ackout* (*ackin*) wire 'high' ('low')
- The sender waits for the *ackin* to go 'low' and then resets the data bus (i.e. spacer state)
- After an unbounded, but finite (positive) amount of time, the receiver drives the *ackout* (*ackin*) wire 'low' ('high'). A single transaction is now said to be complete and the system is ready to resume the next transaction

III. Tree Structures – A Review

Multiple inputs addition is an operation widely prevalent in both multiplication and computation of vector inner products [10] [11]. The carry save adder (CSA) is useful for handling addition of many numbers and is therefore suitable for building multipliers and digital filters where complicated additions are required. Unlike the basic carry-propagate adder (CPA), also known as the ripple carry adder (RCA), in a CSA, the carry output signal of the current bit at a level is not transferred to the next-bit adder of the same level as the carry input signal; instead it is transferred to the next-bit adder in the lower level as the carry input signal. A CSA tree can reduce n binary numbers to two numbers in O(log n) levels [11]. A fast logarithmic time dual-operand adder can then be used to add the two resulting numbers. Hence, CSAs were predominantly used in various tree structures for performing multi-input addition.

The rudimentary tree structure, also called the iterative CSA array [10], is a straightforward way to accumulate partial products. Indeed, an n-operand array would consist of $(n-2)$ CSAs and a final CPA stage. As a result, the time complexity of the fundamental array topology would be the summation of propagation delay of the CSA tree governed by a height of $(n-2)$ and the propagation delay associated with the CPA

[1] The explanation remains valid for data representation using any DI data-encoding scheme.

stage which is approximately linear. Wallace trees [12] are known for their optimal computation time. In fact, they represent the theoretically fastest adders when reducing multiple operands to two outputs using CSA trees [13]. In Wallace trees, the number of operands is reduced at the earliest opportunity by employing $n/3$ full adders for all the m columns, where 'n' specifies the number of single-rail data operands and 'm' denotes the size of each operand. This procedure tends to minimize the overall delay by making the final CPA stage as compact as possible. Although the Wallace tree guarantees the lowest overall delay, it requires the largest number of wiring tracks (vertical feed-throughs between adjacent bit-slices), thereby compounding their wiring complexity [14]. The iterative CSA array and Wallace trees represent two possible extremes in the spectrum of multi-operand addition [11]. While the former features the simplest and regular structure, it is also the slowest; the latter is the fastest, but is also the most difficult structure to implement. Other tree structures proposed for multi-operand addition lie between these two extremes permitting tradeoffs between regularity and speed [10]. While Wallace used a word-level description of his trees, Dadda gave a refined presentation of the same concept at the bit-level [15]. In Dadda trees, the number of operands is reduced to the next lower number in comparison with the Wallace tree using the fewest number of full adders and half adders possible, i.e. combining of partial product bits takes place as late as possible and this usually leads to a simple CSA tree unlike Wallace's method where partial products are combined at the earliest opportunity. The former strategy minimizes the number of full adders and half adders at the expense of a wider CPA structure, while the latter tends to make the width of the final CPA smaller. Wallace's and Dadda's strategies for constructing CSA trees give rise to Wallace and Dadda tree multipliers. An analysis of Dadda and Wallace multiplier delays was performed for different multiplier sizes [16], and it was found that the former showed improvement in speed compared to the latter by 9%-14%; however, this work assumed the presence of only discrete logic gates (AND2, OR2 and INV cells). It has been clarified in [11] that the above strategies which achieve logarithmic depth reduction based on CSA trees tend to suffer from the drawback of an irregular structure that subsequently complicates the design and layout. Additionally, connections of varying lengths and complex signal paths lead to logic hazards and signal skew in synchronous designs that would have negative implications for power and performance parameters. Overturned-stairs (OS) tree structures [17] can be designed systematically paving the way for a simple and regular interconnection scheme in comparison with the Wallace tree whilst achieving similar speed performance in certain cases. The balanced delay tree [18], on the other hand, requires the smallest number of wiring tracks but suffer from an increased delay compared to the OS trees. Nevertheless, it has been widely understood that iterated or recursive structures that would feature a greater degree of structural regularity, less hardware complexity and promise high-speed such as those incorporating parallel counters or logic compressors are preferable compared to CSA based tree structures [10] [11] [13] [17] [19] [20]. It is to be noted in this context that tree structures are also useful for evaluating the performance potential of arithmetic building blocks [21].

IV. Bit-Partitioning Scheme

In CSAs, row-wise parallel addition is performed where the tree height grows with the increase in the number of input operands by an approximate linear order. Here, a bit-partitioning strategy is considered which involves splitting up the entire group of operands horizontally into sub-groups as desired, and the results of the sub-groups are then added to produce the final sum. The bit-partitioning approach parallelizes the multi-input addition operation and is illustrated through figure 5, where addition of n binary operands with each operand of size m bits is considered, while assuming n to be even. A 'dot' represents a bit position in the figure below.

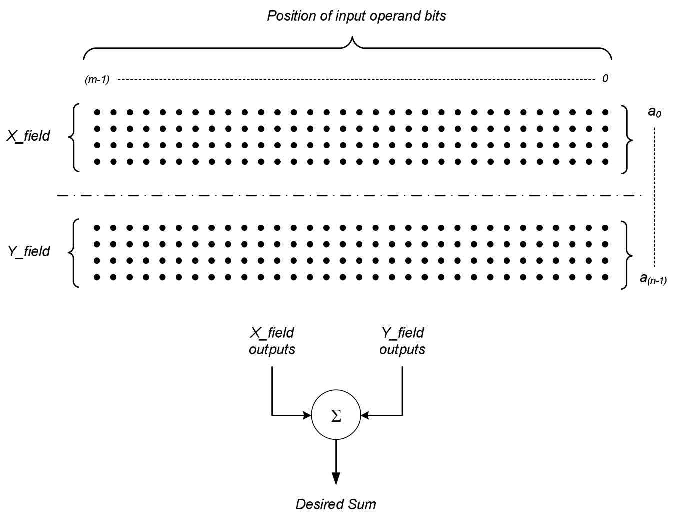

Fig. 5 Illustration of bit-partitioned multiple input addition strategy

The entire set of input operands ($a_0,....,a_{n-1}$) is divided into two equal-sized groups, namely *X_field* (that comprises inputs, $a_0,...,a_{(n-1)/2}$) and *Y_field* (consisting of inputs, $a_{(n+1)/2},...,a_{n-1}$). Addition within the individual fields can be performed using either CSAs or logic compressors. The sum bits generated from these individual fields can be added together using a two-operand adder. Herein, we use a RCA for performing summation of the outputs of *X* and *Y* data fields.

In general, the combinatorial bit-partitioning procedure might effect a slight improvement in delay when many operands have to be added by way of performing parallel column wise addition of row-wise partitions. For example, considering the addition of 32 data operands, each of size 32-bits, the critical path delay of the multi-operand adder equates to 8 full adder delays (assuming the Wallace bound) and the delay of a 36-bit RCA stage. On the other hand, with eight equal-sized input field partitions, the maximum path delay could be reduced by 2 full adder delays. If say 16 operands are to be added, they could be initially partitioned into 4 fields (say, *V, W, X* and *Y*). The outputs of input fields *V* and *W* can be combined into an intermediate output field; likewise with input fields *X* and *Y*. The sum outputs corresponding to the intermediate output fields can then be added to obtain the desired final result. Alternatively, the outputs of the four input fields can be added together using a single multi-operand adder to produce the required result. It should be noted that additions within the partitions would be carried out in parallel, while the final adder stage comprising a simple CPA could perform serial computation. Thus the bit-partitioning procedure is scalable and may benefit in terms of latency reduction as opposed to employing conventional combinatorial tree type structures for problems of higher dimensions. Also, a high regularity would be implicit within the overall architecture as the gate level input partition hardware structures are being duplicated. We shall now discuss about self-timed CSAs and logic compressors in the following sub-sections, as employed for the input field partitions.

A. CSA Based Multi-Operand Addition

Figure 6 shows the self-timed equivalent of a traditional synchronous CSA structure used for the addition of four dual-rail encoded binary numbers (*a*,*b*,*c*,*d*), each of size *n* bits, and the (*n*+1) sum outputs produced are also in dual-rail format. Inputs and outputs with subscript zero correspond to the least significant bits and those with the maximum subscript notation represent the most significant bits. As shown in figure 6, there are three adders in three levels – two levels of CSAs and one level of RCA to add four input operands. In each CSA, the output carry signal of the current bit at a level is not transferred to the next bit adder of the same level as the input carry. Instead, the output carry is transferred to the next bit adder in the lower level as the carry input signal. In the top-level adder, three numbers (*a*,*b*,*c*) are added simultaneously, i.e. the bits corresponding to any number could act as the input carries for the full adders of the first level CSA. In the next lower level, an extra number (*d*) is added. The adder in the bottom level is a conventional carry-ripple adder that produces the final sum. The propagation delay of the entire multi-operand adder is equal to the sum of the delay of two full adder cells in the first two levels and the delay associated with the RCA at the final level.

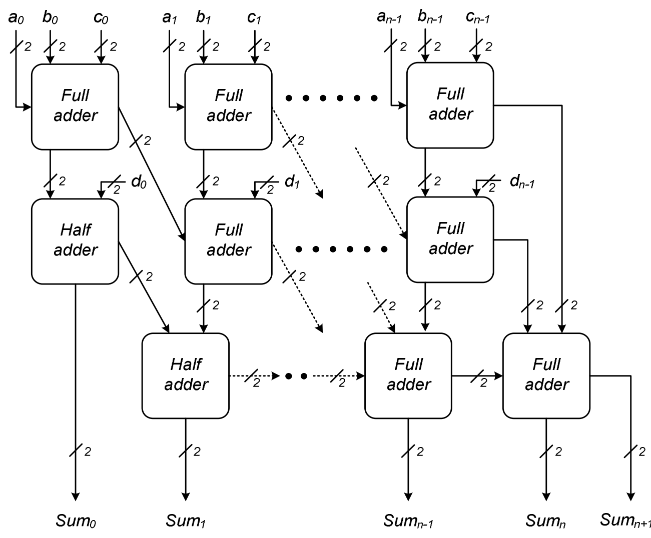

Fig. 6 Self-timed version of *n*-bit CSA to add four data operands

B. Compressor Based Multi-Operand Addition

Rather than using CSAs for the partitions, logic compressors can be employed for adding multiple input operands as shown in figure 7. The (4:2) logic compressor [22] usually takes in five inputs (four inputs in the absence of an input carry) including a carry input from the preceding stage and produces three outputs – two carry outputs, with one carry (*ICarry*) propagating as a carry input to the compressor block of the next column in the same row, while the sum (*Sum*) and carry (*Cout*) outputs are fed as inputs to the final RCA stage. In essence, it is a 5-bit column adder [11].

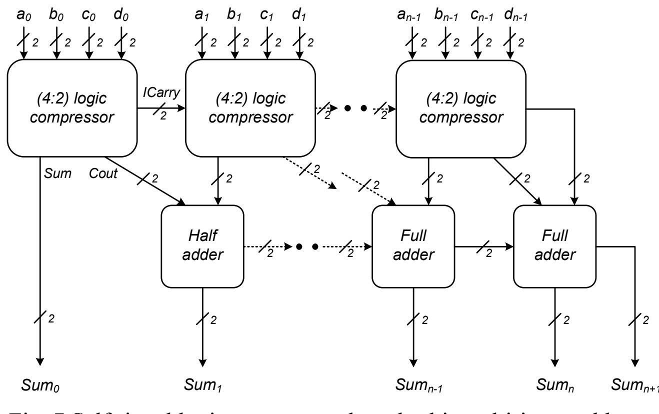

Fig. 7 Self-timed logic compressor based *n*-bit multi-input adder to add 4 data operands

The efficient realization of a (4:2) compressor block is important for multi-operand addition. It is usual practice to realize compressors using full adder blocks [11] [19] that constitutes a scalable approach rather than synthesizing them as a single block – this is owing to the input space demand. For a linear increase in the number of inputs by O(*n*), the input state space expands by an exponential order of O(2^n). A typical (4:2) compressor design [23] using two full adder modules is shown in figure 8. The self-timed version of a (4:2) compressor can be derived by replacement of the synchronous full adder modules with equivalent self-timed blocks, as is the case with Null Convention Logic (NCL) approaches [24] [25]. It may be noticeable that the compressor shown in figure 8 treats a full adder as a CSA and thus the compressor logic is equivalent to that realized by the CSA tree (first two levels, preceding the RCA stage) of figure 6. Alternatively, a (4:2) compressor can be realized using discrete gates as shown in figure 9 [27].

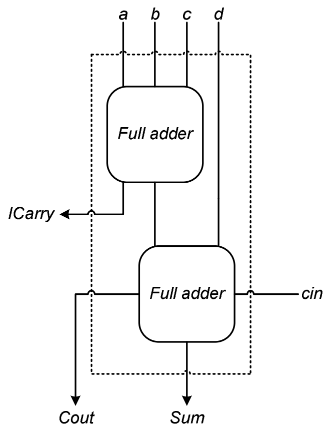

Fig. 8 A synchronous logic compressor realized using full adders

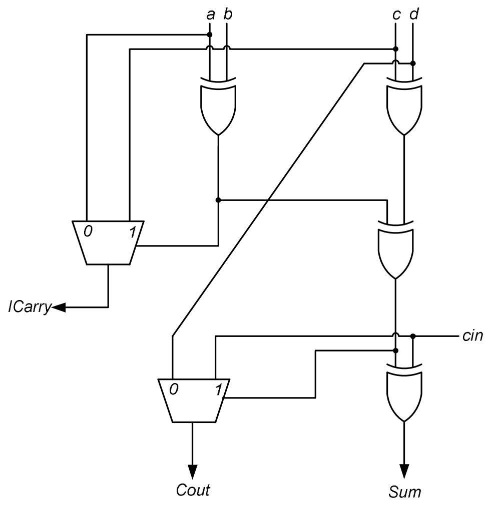

Fig. 9 A synchronous (4:2) compressor based on discrete gates

The weak-indication synthesis of the (4:2) compressor (with input carry), shown in figure 10, may be thought of as a translation of the synchronous version depicted in figure 9. However, this differs from all the NCL methods, which are actually founded upon the DRCL style, where the encoded outputs are duals of each other. In case of the proposed design, however, the encoded outputs make use of disjunctive normal expressions for implementing the outputs.

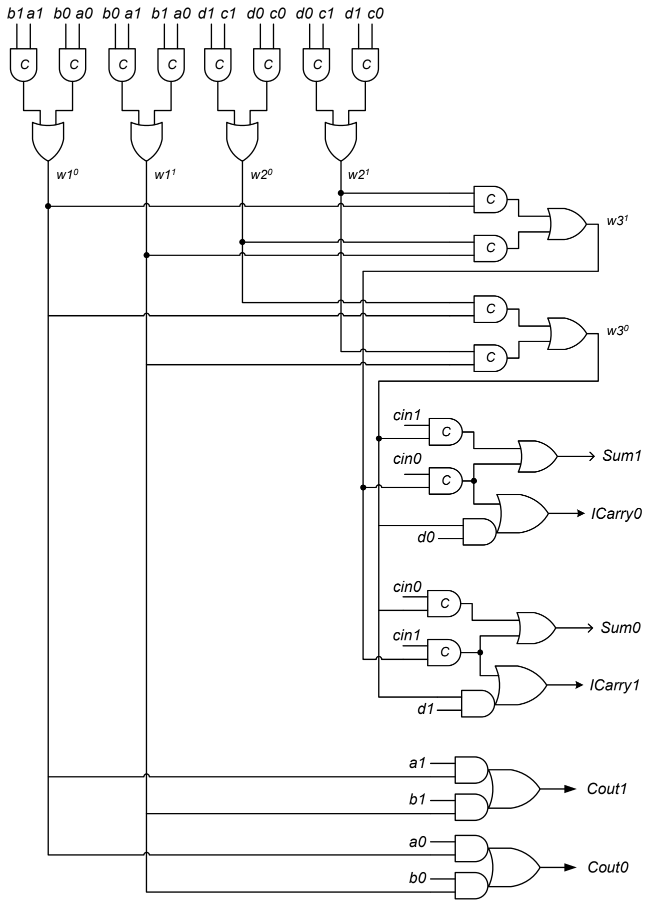

Fig. 10 Weakly indicating (4:2) logic compressor with carry input

Three steps are involved in the proposed compressor synthesis – i) deriving the irredundant disjoint sum-of-products form of the dual-rail logic compressor functionality [28], ii) speed-independent decomposition of logic to facilitate physical realization using standard cells [29], and iii) performing logic optimizations to pave the way for latency reduction. Comparison with NCL designs [24] – [26] is not considered here since the technology mapping procedure would require access to proprietary NCL macros [30] [31]. In the figures, the Muller C-element[2] is represented by the AND gate symbol with the marking *C* on its periphery. The multi-level expressions corresponding to the proposed dual-rail encoded logic compressor design shown in figure 10 are given below. Henceforth, this compressor realization shall be referred to as the ‘Sync_ST_compressor’ in the following discussions. Given these, the synthesis of a compressor module without input carry would be rather straightforward and is shown in figure 11.

[2] The C-element governs the rendezvous of input signals. The C-gate outputs a 1(0) if all its inputs are 1(0) respectively, otherwise it maintains its existing steady state.

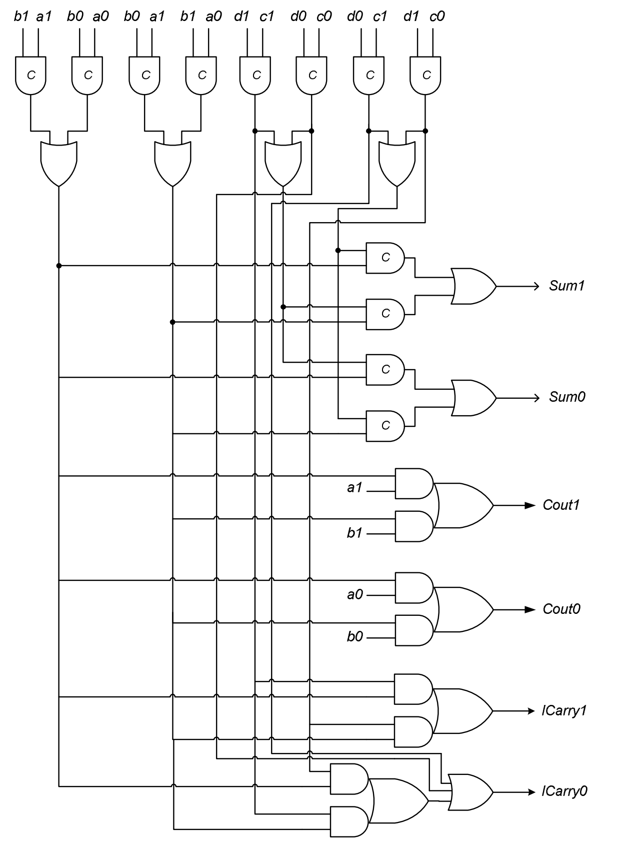

Fig. 11 Weak-indication (4:2) logic compressor without carry input

$$Sum1 = w3^0cin1 + w3^1cin0 \quad (1)$$

$$Sum0 = w3^0cin0 + w3^1cin1 \quad (2)$$

$$ICarry1 = d1w3^0 + cin1w3^1 \quad (3)$$

$$ICarry0 = d0w3^0 + cin0w3^1 \quad (4)$$

$$Cout1 = a1w1^0 + c1w1^1 \quad (5)$$

$$Cout0 = a0w1^0 + c0w1^1 \quad (6)$$

From figures 10 and 11, it may be apparent that the self-timed compressor realizations correspond to the weak-indication timing discipline, with the sum outputs being assigned the responsibility of indicating the arrival of all the primary inputs and the intermediate outputs, while the carry outputs are allowed to be set/reset in an eager or early output fashion.

The logically equivalent early propagative synthesized versions of the self-timed (4:2) logic compressor, shown in figures 10 and 11, are portrayed by figures 12 and 13 respectively. These are derived by resorting to further peephole logic optimizations of the weak-indication equivalent as a post-processing step facilitating the usage of more complex gates. For example, comparing figures 10 and 12, it is evident that the logic corresponding to the intermediate outputs ($w1^0$, $w1^1$) and ($w2^0$, $w2^1$) has been realized using AO22 cells in the latter while C-gates and OR gates are present in the former. Since early output logic modules tend to be set/reset in an eager fashion, the indication of their inputs is taken care of by their associated completion detection circuit – this is discussed in the next section.

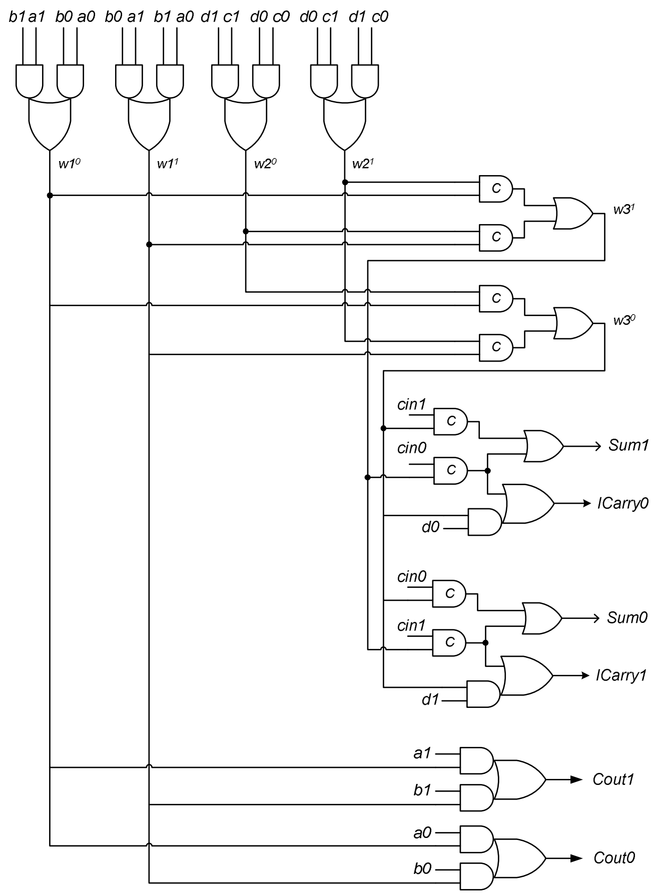

Fig. 12 Early output (4:2) logic compressor with carry input

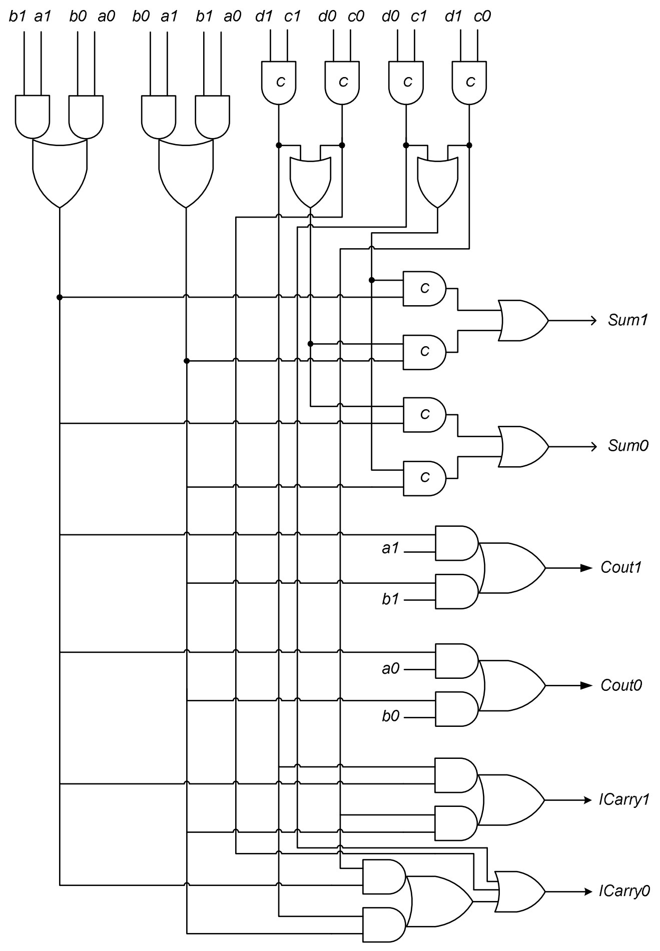

Fig. 13 Early output (4:2) logic compressor without carry input

V. Simulation Results

In order to analyze the efficacy of CSAs and compressors forming part of the partitions in case of a multi-operand adder, an example scenario of self-timed addition of 8 input data operands, each of size 32 bits was considered. The inputs were divided into two equal input fields containing 4 operands each and the individual summation result of these two partitions gives rise to 34 intermediate sum outputs. These were subsequently added using a 34-bit RCA to generate the final result that consists of 35 sum outputs. The delay, area and power parameters of this bit-partitioned addition process, assuming CSAs for the input field partitions are given in Table 1. The delay parameter refers to the maximum propagation delay encountered in the data path, which approximately equals the latency of the function block. The delay metric was estimated using PrimeTime. To avoid the notion of a clock source, the option of a virtual clock was used that only acts as a remote reference to constrain the input and output ports of the design. The area and power metrics correspond to the input registers, completion detection logic and the function block. The delay and power metrics consider estimated parasitics in addition to the parameters associated with the actual components. The area metric gives a combined account of the area of all the logic cells and was estimated as part of the PrimeTime tool suite. The total/average power dissipation is the summation of dynamic and static power components, where dynamic power is in turn the gross of switching and internal power consumption figures. NC-Sim has been used for functional simulation and also to obtain the switching activity files corresponding to the gate level simulations of Verilog descriptions. Input data were supplied to the function blocks at specific intervals through test benches, which modeled the environment. The switching activity files obtained were subsequently used for power estimation using PrimeTime PX. The simulations targeted a PVT corner of the 130nm bulk CMOS standard cell library with a supply voltage of 1.32V and a junction temperature of -40°C. All the circuit inputs were configured to possess the driving strength of the minimum sized inverter of the cell library, while the outputs were associated with fanout-of-4 drive strength. Suitable buffering for the acknowledge input was provided where necessary to eliminate timing violations. Since identical registers and a similar completion detection circuit were used for all the adder realizations, the area and power metrics can be correlated with that of the function block, thus paving the way for a legitimate comparison between different self-timed logic realization methods. Random input data sequences were used for the adder simulations and they were supplied at time intervals of 25ns to the function blocks. Weak-indication adders corresponding to various self-timed design methods were constructed and were also subsequently optimized for minimum latency taking into account the library constraints[3].

The optimal values of design metrics achieved by a specific self-timed design method are highlighted in bold-face in the Tables. From Table 1, it is clear that with respect to delay and area SSSC_CSA is optimal. However, in terms of total power Toms_CSA betters the SSSC_CSA by reducing power to the tune of 12.5%. Nevertheless, the latter minimizes critical path delay and area occupancy by 36.2% and 7.3% respectively. Moreover, the SSSC_CSA being a weak-indication adder reduces the cycle time for passage of data-spacer wave fronts while Toms_CSA being a strongly indicating adder encounters maximum latency for both valid data and spacers.

Table 1. Delay, area and power parameters corresponding to bit-partitioned CSA based self-timed addition of 8 inputs (size 32 bits)

Multi-input adder realization style	**Delay (ns)**	**Area (μm^2)**	**Power (μW)**
Seitz_CSA [7]	9.2	45805	3068.3
DIMS_CSA [32]	16.6	66303	3245.0
Petrify_CSA [33]	9.7	42701	2943.5
Toms_CSA [34] [35]	14.1	44866	**2397.3**
SSSC_CSA [36]	**9.0**	**41586**	2740.6

Compressor designs based on a number of self-timed logic realization methods were found to exacerbate the area requirement and this eventually has an adverse impact on delay

[3] A 130nm bulk CMOS standard cell library was used. The fan-in of AND gates and OR gates in the library is 4 and 3 respectively. The C-element has a granularity of up to 4 inputs.

and power metrics due to an increase in the number of logic levels and requirement of more library cells. This is because the (4:2) compressor logic would quadruple the input space consideration in comparison with a full adder block. Figure 14 gives a graphical sketch of the area expenditure of various self-timed compressor realizations. The values on the y-axis signify the area occupancy in micrometer square, while the different compressor realization styles are mentioned in the x-axis. The values specified above the vertical bars of the bar chart signify the area figures for a cell-based implementation. Indeed, the area figures correspond to optimized designs.

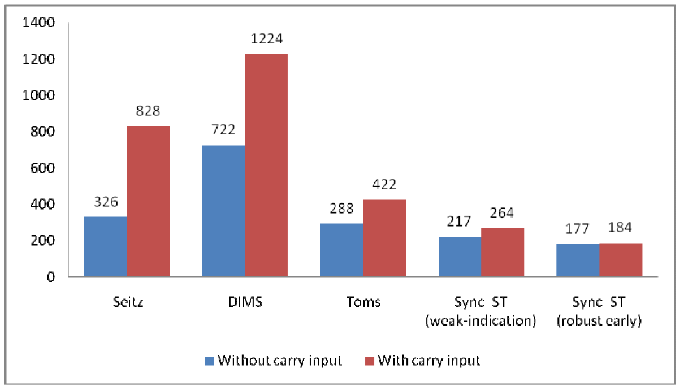

Fig. 14 Area comparison of (4:2) asynchronous compressors

The design metrics corresponding to bit-partitioned multi-operand addition that considers logic compressors for the input partitions are given in Table 2. (4:2) logic compressors based on Seitz, DIMS and Toms approaches were constructed in a semi-custom design style with delay-oriented logic optimizations resorted to where feasible. The DIMS weak-indication compressor design involved only speed-independent logic decomposition, while Seitz's weak-indication compressor entailed speed-independent logic decomposition of higher fan-in AND gates and replacement of second-level AND gates by C-gates to ensure gate orphan freedom. Moreover, Seitz's and Petrify design methods incorporate timing assumptions in inputs completion detection.

Table 2. Delay, area and power metrics corresponding to bit-partitioned compressor based self-timed addition of 8 data inputs, each of size 32 bits

Multi-operand adder (compressor based)	**Delay (ns)**	**Area (μm^2)**	**Power (μW)**
Seitz_compressor [7]	9.7 (12.8%)	77611 (2.18×)	3605.3 (53.9%)
DIMS_compressor [32]	17.3 (101.2%)	111757 (3.14×)	3974.4 (69.7%)
Toms_compressor [34] [35]	22.5 (161.6%)	51950 (1.46×)	2418.5 (3.3%)
Sync_ST_compressor (weak-indication)	8.8 (2.3%)	40608 (1.14×)	2588.6 (10.5%)
Sync_ST_compressor (robust early)	**8.6**	**35568**	**2341.6**

The common point between the Sync_ST_compressor (weak-indication) and Sync_ST_compressor (robust early) realizations is that both these guarantee gate-orphan freedom. The problem of wire orphans is nullified by the isochronic fork assumptions. However, the primary point of distinction between the Sync_ST_compressor (weakly indicating) and the Sync_ST_compressor (robust early) design is that the former leads to a *locally indicating* self-timed system architecture, where the function block individually acknowledges the arrival of the primary inputs, while the latter facilitates a self-timed system configuration which is *globally indicating* with respect to acknowledging the arrival of the primary inputs [37]. This is because, in case of the latter, the function block computes in an eager fashion and therefore the completion detection logic preceding it becomes responsible for indicating the arrival of primary inputs into the function block, as portrayed by the system architecture in figure 15. Given this, isochronicity is assumed with regard to the primary inputs that are fed into the early output function block and the completion detection circuit associated with a stage.

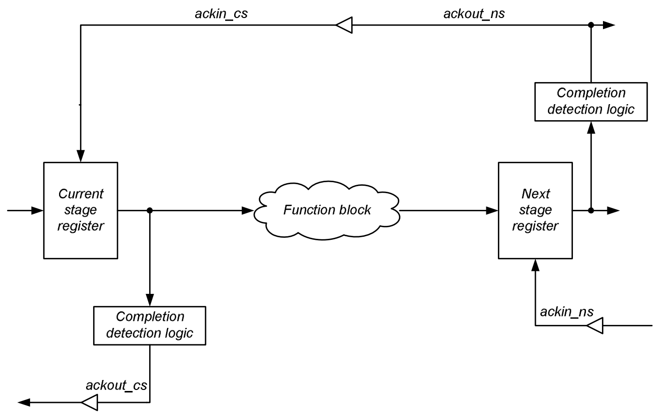

Fig. 15 A typical self-timed system architecture

The operation of the above self-timed system configuration is explained as follows. Let us consider that all the registers are initially in the spacer (empty) state and therefore the acknowledge signals *ackout_ns* and *ackin_cs* would assume logic 'low' and logic 'high' states respectively. Thus the current stage register would be active and ready to accept a new set of valid data. When the inputs become defined, valid data would be passed through the current stage register onto the function block for processing and the function block outputs would reach the next stage register in the pipeline. The completion detection logic [38] performs the validity/neutrality tests of the input codeword during the set and reset phases respectively [39]. With respect to dual-rail encoding, the completion detector would have an OR gate assigned for each dual-rail input and the outputs of all such OR gates would be synchronized by means of a C-element tree. The completion detector associated with the current stage register would check the validity of the data at its inputs and subsequently asserts the *ackout_cs* signal to logic 'high' if the check is true. This signal disables the previous register and prepares it for storing the spacer data wave front. Thus it paves the way for flushing the function block by permitting spacer data through the current stage register. The collision between two data wave fronts is avoided by means of alternating set and reset phases (i.e. through the valid data-spacer-valid data sequence) according to the 4-phase handshaking convention.

The increase in data path delay, total power dissipation and relative increase in area occupancy of all the multi-input adders in comparison with the Sync_ST_compressor (robust early) based multi-input adder is highlighted in Table 2 within brackets for a quick comparison. Comparing Tables 1 and 2, it can be inferred that the bit-partitioned multi-input adder employing CSAs rather than logic compressors for the partitions are preferable with respect to power, delay and area in the case of Seitz, DIMS and Toms approaches. This is owing to the greater input space consideration for a direct compressor realization as opposed to a full adder based realization. The Sync_ST_compressor (robust early) based multi-input adder features less latency and area occupancy in comparison with the SSSC_CSA based multi-input adder by 4.4% and 14.5% respectively. Even in terms of total power dissipation, the Sync_ST_compressor (robust early) based multi-input adder is preferable compared to Toms_CSA based multi-input adder owing to a reduced power consumption of 2.3%. Therefore, the Sync_ST_compressor (robust early) based multi-operand adder is found to be an efficient design with respect to simultaneous optimization of the power-delay-area envelope.

It has also been observed from the simulations that usage of a hybrid input encoding scheme for self-timed multi-input adders, by way of employing a mixture of DI data encoding (say, dual-rail and 1-of-4 codes), results in increase of delay, area and power over pure dual-rail encoded counterparts. This is most likely due to the reason that only the primary inputs of the multi-operand adder can be grouped together and encoded using the hybrid input encoding mechanism, while all the intermediate and primary outputs necessitate maintaining of the dual-rail convention. The reductions in power dissipation and area metrics gained by the hybrid input encoded compressor logic tends to be nullified by the extra power dissipation and area occupancy of its associated encoding circuitry. Hence, encoding of the primary inputs in a heterogeneous fashion does not appear to have a beneficial impact on the resultant multi-input adder implementations. This effect is likely even in case of bit-partitioned multi-input adder that employs CSAs for the input field partitions. Hence it is opined that dual-rail encoding might be an optimum DI data encoding mechanism for effectively implementing self-timed multi-operand addition as opposed to any other heterogeneous DI data encoding scheme.

VI. Conclusion

Self-timed addition of multiple data operands based on a bit-partitioning strategy was discussed in this paper. The impact of CSAs and compressors on the parallel input field partitions was analyzed for the case study of a sizeable addition operation involving 8 data operands, each of width 32 bits. It is inferred that the robust Sync_ST_compressor realization corresponding to early output logic exhibits a superior performance with respect to simultaneous optimization of delay, area and power metrics as regards this case study, and the Sync_ST_compressor (robust early) could serve as an efficient building block from the design viewpoint. Hence, it can be potentially used to build optimal higher order self-timed multi-operand adders.

References

1) Semiconductor Industry Association's International Technology Roadmap for Semiconductors (ITRS) 2008 design report, Available: http://www.itrs.net
2) A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic and P.J. Hazewindus, "The first asynchronous microprocessor: the test results," *ACM SIGARCH Computer Architecture News*, vol. 17, no. 4, pp. 95-98, June 1989.
3) G.F. Bouesse, G. Sicard, A. Baixas and M. Renaudin, "Quasi delay insensitive asynchronous circuits for low EMI," *Proc. 4th International Workshop on Electro-Magnetic Compatibility of Integrated Circuits*, pp. 27-31, 2004.
4) C.H. van Berkel, M.B. Josephs and S.M. Nowick, "Scanning the technology: applications of asynchronous circuits," *Proc. of the IEEE*, vol. 87, no. 2, pp. 223-233, February 1999.
5) A.J. Martin, "The limitation to delay-insensitivity in asynchronous circuits," *Proc. 6th Conference on Advanced Research on VLSI*, MIT Press, pp. 263-278, 1990.
6) A.J. Martin and P. Prakash, "Asynchronous nanoelectronics: preliminary investigation," *Proc. 14th IEEE International Symposium on Asynchronous Circuits*, pp. 58-68, 2008.
7) C.L. Seitz, "System Timing" in *Introduction to VLSI Systems*, C. Mead and L. Conway (Eds.), Addison-Wesley, MA, USA, pp. 218-262, 1980.
8) V.I. Varshavsky (Ed.), *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete systems*, Chapter 4: Aperiodic Circuits, pp. 77-85, (Translated from the Russian by Alexandre V. Yakovlev), Kluwer Academic Publishers, 1990.
9) J. Cortadella, A. Kondratyev, L. Lavagno and C. Sotiriou, "Coping with the variability of combinational logic delays," *Proc. IEEE International Conference on Computer Design*, pp. 505-508, 2004.
10) K. Hwang, *Computer Arithmetic: Principles, Architecture and Design*, John Wiley and Sons Inc, New York, 1979.
11) B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*, Oxford University Press, New York, 2000.
12) C.S. Wallace, "A suggestion for a fast multiplier," *IEEE Trans. on Electronic Computers*, vol. EC-13, no. 1, pp. 14-17, February 1964.
13) W. Waser and M.J. Flynn, *Introduction to Arithmetic for Digital Systems Designers*, Oxford University Press, New York, 1985.
14) P. Reusens, W.H. Ku and Y.H. Mao, "Fixed-point high-speed parallel multipliers in VLSI," in *VLSI Systems and Computations*, H.T. Kung et al. (Eds.), pp. 301-310, Springer-Verlag, New York, 1981.
15) L. Dadda, "Some schemes for parallel multipliers," *Alta Frequenza*, vol. 34, no. 5, pp. 349-356, March 1965.
16) W.J. Townsend, E.E. Swartzlander Jr. and J.A. Abraham, "A comparison of Dadda and Wallace multiplier delays," *Proc. SPIE Advanced Signal Processing Algorithms, Architectures and Implementations XIII*, Franklin T. Luk (Ed.), vol. 5205, pp. 552-560, 2003.
17) Z.-J. Mou and F. Jutand, "Overturned-stairs adder trees and multiplier design," *IEEE Trans. on Computers*, vol. C-41, no. 8, pp. 940-948, August 1992.
18) D. Zuras and W.H. McAllister, "Balanced delay trees and combinational division in VLSI," *IEEE Journal of Solid-State Circuits*, vol. SC-21, no. 5, pp. 814-819, October 1986.
19) Mi Lu, *Arithmetic and Logic in Computer Systems*, John Wiley and Sons Inc, NJ, 2004.
20) A.R. Omondi, *Computer Arithmetic Systems: Algorithms, Architecture and Implementations*, Prentice Hall International (UK) Ltd, 1994.

21) C.-H. Chang, J. Gu and M. Zhang, "A review of 0.18μm full adder performances for tree structured arithmetic circuits," *IEEE Trans. on VLSI Systems*, vol. 13, no. 6, pp. 686-695, June 2005.
22) A. Weinberger, "4:2 carry-save adder module," *IBM Technical Disclosure Bulletin*, vol. 23, January 1981.
23) I. Koren, *Computer Arithmetic Algorithms*, Prentice-Hall International (UK), 1993.
24) M. Ligthart, K. Fant, R. Smith, A. Taubin and A. Kondratyev, "Asynchronous design using commercial HDL synthesis tools," *Proc. 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pp. 114-125, 2000.
25) A. Kondratyev and K. Lwin, "Design of asynchronous circuits by synchronous CAD tools," *IEEE Design and Test of Computers*, vol. 19, no. 4, pp. 107-117, July-August 2002.
26) S.C. Smith, R.F. DeMara, J.S. Yuan, D. Ferguson and D. Lamb, "Optimization of null convention self-timed circuits," *Integration, the VLSI Journal*, vol. 37, no. 3, pp. 135-165, August 2004.
27) P. Prasad and K.K. Parhi, "Low-power 4-2 and 5-2 compressors," *Proc. 35th Asilomar Conference on Signals, Systems and Computers*, vol. 1, 2001, pp. 129-133.
28) P. Balasubramanian and D.A. Edwards, "Self-timed realization of combinational logic," *Proc. 19th International Workshop on Logic and Synthesis*, pp. 55-62, 2010.
29) P. Balasubramanian and D.A. Edwards, "A new design technique for weakly indicating function blocks," *Proc. 11th IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems*, pp. 116-121, 2008.
30) K.M. Fant and G.E. Sobelman, "Null convention threshold gate," US Patent 5664211, February 1997.
31) K.M. Fant and S.A. Brandt, "Null convention logic system," US Patent 5828228, October 1998.
32) J. Sparso and J. Staunstrup, "Delay-insensitive multi-ring structures," *Integration, the VLSI Journal*, vol. 15, pp. 313-340, 1993.
33) J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers," *IEICE Trans. on Information and Systems*, vol. E80-D, no. 3, pp. 315-325, 1997.
34) W.B. Toms, D.A. Edwards, "Efficient synthesis of speed-independent combinational logic circuits," *Proc. 10th Asia and South Pacific Design Automation Conference*, pp. 1022-1026, 2005.
35) W.B. Toms, "Synthesis of quasi-delay-insensitive datapath circuits," *PhD Thesis*, University of Manchester, 2006.
36) P. Balasubramanian and D.A. Edwards, "A delay efficient robust self-timed full adder," *Proc. 3rd IEEE International Design and Test Workshop*, pp. 129-134, 2008.
37) P. Balasubramanian, N.E. Mastorakis, "Analyzing the impact of local and global indication on a self-timed system," *Proc. 5th European Computing Conference*, pp. 85-91, 2011.
38) J. Sparso and S.B. Furber (Eds.), *Principles of Asynchronous Circuit Design: A Systems Perspective*, Kluwer Academic Publishers, 2001.
39) A.J. Martin and M. Nystrom, "Asynchronous techniques for system-on-chip design," *Proc. of the IEEE*, vol. 94, no. 6, pp. 1089-1120, June 2006.