

A Novel Low Overhead Fault Tolerant Kogge-Stone Adder Using Adaptive Clocking

Swaroop Ghosh, Patrick Ndai and Kaushik Roy
School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN 47907
<ghosh3, pndai, kaushik>@ecn.purdue.edu

Abstract— As the feature size of transistors gets smaller, fabricating them becomes challenging. Manufacturing process follows various corrective design-for-manufacturing (DFM) steps to avoid shorts/opens/bridges. However, it is not possible to completely eliminate the possibility of such defects. If spare units are not present to replace the defective parts, then such failures cause yield loss. In this paper, we present a fault tolerant technique to leverage the redundancy present in high speed regular circuits such as Kogge-Stone adder (KSA). Due to its regularity and speed, KSA is widely used in ALU design. In KSA, the carries are computed fast by computing them in parallel. Our technique is based on the fact that even and odd carries are mutually exclusive. Therefore, defect in even bit can only corrupt the even Sum outputs whereas the odd Sums are computed correctly (and vice versa). To efficiently utilize the above property of KSA in presence of defects, we perform addition in two- clock cycles. In cycle-1, one of the correct set of bits (even or odd) are computed and stored at output registers. In cycle-2, the operands are shifted by one bit and the remaining sets of bits (odd or even) are computed and stored. This allows us to tolerate the defect at the cost of throughput degradation while maintaining high frequency and yield. The proposed technique can tolerate any number of faults as long as they are confined to either even or odd bits (but not in both). Further, this technique is applicable for any type of fault model (stuck-at, bridging, complete opens/shorts). We performed simulations on 64-bit KSA using 180nm devices. The results indicate that the proposed technique incur less than 1% area overhead. Note that there is very little throughput degradation (<0.3%) for the fault-free adders. The proposed technique utilizes the existing scan flip-flops for storage and shifting operation to minimize the area/performance overhead. Finally, the proposed technique is used in a superscalar processor, whereby the faulty adder is assigned lower priority than fault-free adders to reduce the overall throughput degradation. Experiments performed using Simplescalar for a superscalar pipeline (with four integer adders) show throughput degradation of 0.5% in the presence of a single defective adder.

Keywords: Stuck-at faults, Fault tolerant adder, Adaptive clocking, Kogge-Stone adder, Scheduling.

I. INTRODUCTION

Sub-wavelength lithography, line edge roughness (LER), chemical mechanical polishing (CMP), etching etc cause large variation in transistor and wire geometries. These variations change the strength of transistors (systematically as well as randomly) causing variations in path delays. Manufacturing subtleties such as poor patterning, narrow metal region near via, missing salicide between metal and poly etc introduce resistive bridging in the circuit. Such resistive paths combined with process variation induced delay defects can deter a node from switching. Under a strict timing constraint and at lower operating supply, the delayed switching behaves as stuck-at 0 or stuck-at 1. Similar behavior is observed for stuck open defects (that appears due to electro-migration, poor printing etc) which introduce *inline* resistance in the wire, increasing both rise and fall times. The possibilities of such defects are increasing with aggressive scaling of transistor geometries, supply voltage and, increasing operating frequency requirements. Designing

robust circuits in the presence of a large number of possible timing/open/short defects to meet a yield target is a challenging task. A possible technique to overcome the small delay defects would be to scale up the supply voltage or to reduce the operating frequency. However, this would either increase the power consumption or slow down the performance of the chip, making the chip unworthy to be shipped. Further, such techniques are not fruitful under large timing defects or complete open or short faults.

Several clever techniques have been proposed in past to tolerate delay defects. In [1], the authors isolate the critical paths of random logic circuits and reduce their activation probability by proper synthesis and sizing. If the chip suffers from timing failures in critical paths, the output is evaluated in two-clock cycles. This allows them to maintain high yield and rated clock frequency at the cost of slight throughput degradation. However, it does not address the large delay defect and general fault model scenario that is under consideration in this paper. In [2], the authors proposed a stuck-at tolerant Kogge-Stone Adder. The idea is to add an extra Han-Carlson (HC) stage which computes the even bits from odd bits (or vice versa) for defective adders. Therefore, stuck-at faults are tolerated at the cost of area/delay overhead (due to HC stage and multiplexers). Quadruple time redundancy [3] and triple modular redundancy [4] techniques have also been proposed in order to detect and correct errors at the cost large area overhead.

In this paper, we achieve fault tolerance by adopting a different perspective. We utilize the inherent spatial redundancy present in high-speed circuits such as Kogge-Stone adder in an efficient manner in order to tolerate *any type of defect*. Our technique is based on the fact that the even and odd carries are *mutually exclusive*. Therefore, any defect in the even bits can only corrupt the even Sum outputs while the odd Sums are computed correctly (and vice versa). For example, in a 4-bit KSA, a defect in bit-1 can introduce errors only in Sum_1 and Sum_3 . The other Sum outputs (i.e Sum_0 and Sum_2) are computed in parallel and will be *fault-free*. To efficiently utilize the above property of KSA in presence of defects, we add little overhead in the adder during design time. The adder operates normally (in single clock cycle) if it is found to be fault-free after the manufacturing test. However, if the adder is faulty, the addition is performed in two clock cycles. In cycle-1, one of the correct set of bits (even or odd) are computed and stored at the output registers. In cycle-2, the operands are shifted by one bit and the remaining sets of bits (odd or even) are computed and stored. This allows us to tolerate any kind of defect at the cost of throughput degradation due to increased latency operations while maintaining rated frequency and yield. The fault-free adders operate without any throughput degradation. To alleviate the throughput loss we schedule the faulty adder occasionally by proper micro-architectural techniques. The overall flowchart is shown in Fig. 1(a). Note that we have not considered the test and diagnosis of faults in this work but it is integral part of the overall fault tolerant methodology.

As evident from the above discussions, the proposed technique requires addition of multiplexers at the inputs and outputs for shifting the operands and storing the correct output. In this work, we reuse the multiplexers present in scan flops to reduce the area overhead for shift. The storage of even/odd bit Sum outputs is also controlled by output multiplexers (which are reused from scan flops). This will be discussed in detail in Section III.

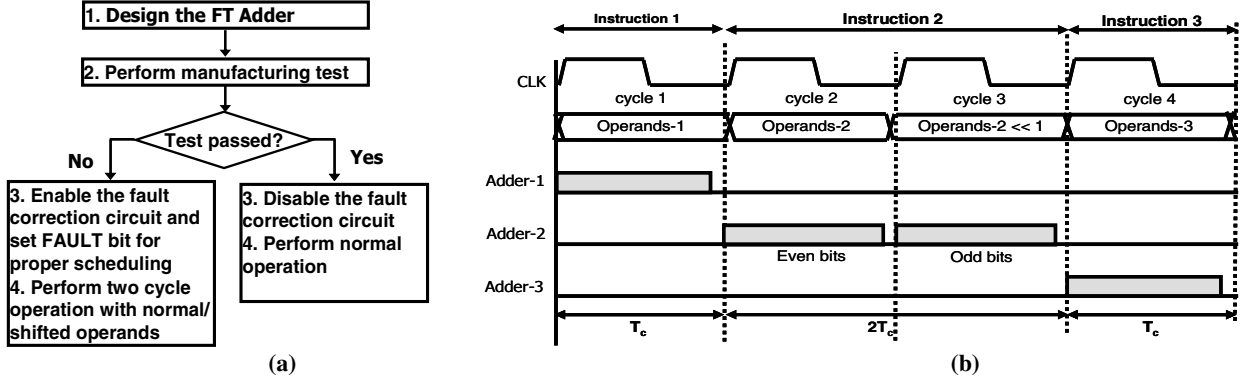


Fig. 1 (a) Proposed fault tolerant methodology, (b) timing diagram for three adders in the execution unit (T_c is the clock period)

The timing diagram of this scheme is elucidated in Fig. 1(b) for three pipelined instructions in a superscalar pipeline (which usually has many adders for parallel instruction issue). For the sake of simplicity, let us assume that these instructions are scheduled to three different adders present in the execution unit. The second adder is *defective* and is always evaluated in two clock cycles whereas the fault-free adders are evaluated in single-clock cycle. The timing diagram shows the scenario when the first instruction is scheduled to adder-1, second instruction is scheduled to adder-2 and so on. The first and third instruction can be evaluated in one clock cycle however, adaptive clocking is performed during the execution of second instruction for correct functionality of the pipeline (Fig. 1(b)). Note that, the second instruction is fired at cycle-2 but completely evaluated only at the end of cycle-3. The even bits are computed in cycle-2 (with operand-2) and registered while the odd bits are discarded. In cycle-3, the odd bits are computed correctly and stored (with operand-2 left shifted by one bit) assuming that the defect is located in odd bit. Note that the even bits are discarded in cycle-3.

In the above toy example, the average cycles-per-instruction (CPI) is 1.33 since three instructions are executed in 4 cycles. Note that here we assumed that adder-2 is scheduled even when other adders are *free* just for the sake of illustration. In reality, the faulty adder should be scheduled only when all other adders are busy (to minimize the throughput penalty). For example, if adder-1 and adder-3 are *busy* in every cycle then ideally two instructions can be scheduled to adder-2 to be finished in 4 cycles (Fig. 1(b)). This will allow to process ten instructions (four instructions each by adder-1/adder-3 and two instruction by adder-2) leading to CPI of 0.4. The ideal CPI would be 0.33 ($=4/12$) when all adders are fault-free. On the other hand, if the defective adder is completely eliminated then the CPI would become 0.66 ($=8/12$). Therefore, better throughput (0.40 compared to 0.66) can be gained by the proposed fault tolerant technique. The CPI computations performed here are simply for illustration. To accurately compute CPI, we would have to take into account the fact that there will *not* be sufficient Instruction Level Parallelism (ILP) to keep all the four adders in the execution unit busy at all times. This simply means that in some cycles, there will be fewer than four instructions ready to execute, and thus having only three adders or having a single defective adder that completes instructions after two cycles will result in less CPI degradation than computed above. Nevertheless, the CPI computations above give a general trend. We provide much more detail in Section IV by using SPEC 2000 benchmarks to simulate realistic workloads.

Note that the proposed fault tolerant technique is significantly different from Recomputation with Shifted Operands (RESO) [5] [6], which is a concurrent error detection technique. In RESO, if an arithmetic logic unit (ALU) performs a function f , and x is an input

to the function, then an error in the ALU can be detected by comparing $f(x)$ with the output of $(f(x \ll 1) \gg 1)$. For a given data input, the result of function f is stored in a register and compared against the result obtained using shifted operands. In the proposed technique, the shifting of operands is done occasionally (by proper micro-architectural technique) to *correct* the effect of fault and not to *detect* it. In summary, we make following contribution in this paper:

- We propose a novel and low overhead fault tolerant Kogge-Stone adder. The faults are tolerated by utilizing the inherent redundancy of the adder in time efficient manner. The proposed technique is applicable for *any type of defect* and not limited by a particular fault model.
- We propose a micro-architectural technique to utilize the faulty adder with minimum throughput degradation.

The rest of the paper is organized as follows. In Section II, we analyze the impact of faults on the Kogge-Stone Adder. The fault tolerant KSA is described in detail in Section III. The micro-architectural solution to optimize the throughput degradation in presence of faulty adder and two-cycle operations is presented in Section IV. The related work on fault tolerant adder is discussed in Section V and finally, the conclusions are drawn in Section VI.

II. IMPACT OF FAULTS ON KSA

Kogge-Stone adders are popular choice in high speed ALU design due to its faster operation, regular structure and balanced loading in internal nodes compared to other sparse tree adders. In this section, first we briefly discuss the design, operation and general properties of KSA that are relevant from fault tolerance point of view. Next, we elaborate the impact of faults in the intermediate computations and their effect on the overall *Sum* generation.

A. Basic structure of KSA

KSA belongs to the family of fastest parallel prefix adders with complexity of $\log_2 N$ (where N is the width of the adder) meaning thereby that the addition can be done in $\log_2 N$ stages. The basic structure of 8-bit radix-2 Kogge-Stone adder [7] is illustrated in Fig. 2(a). It operates on the principle of block *propagate*(p) and block *generate*(g) [8]. The block *propagate* determines whether the input carry can propagate through the block of bits or not. The block *generate* determines if the block of bits can generate a carry or not. If a and b are input operands of the adder, the propagate/generate and carry in (C_i)/carry out (C_{i+1}) are related as follows

$$p_i = a_i \oplus b_i; \quad g_i = a_i \cdot b_i; \quad C_{i+1} = g_i + p_i C_i \quad (1)$$

In absence of carry input to the adder core (i.e., $C_{-1} = 0$), the *generate* signal become the carry inputs for the intermediate carry computations. In 8-bit KSA, the carry is computed in $k=3$ stages

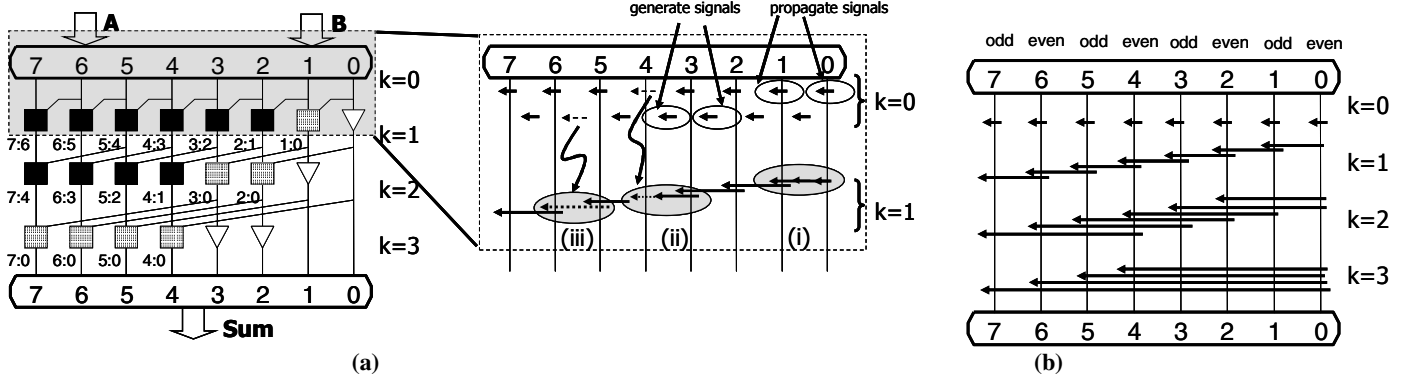


Fig. 2 (a) A basic 8-bit radix-2 Kogge-Stone adder [7] (stages $k=0$ and $k=1$ are expanded to show carry paths for three scenarios) and, **(b)** simplified notation of KSA in terms of carry paths

(where $k = \log_2 N$). In 1st stage ($k=1$), carries and propagates of 2-bit block sizes are computed in parallel. In 2nd stage ($k=2$), carries/propagates of 4-bit block sizes are computed by using 2-bit carries/propagates from stage-1. Therefore, carry till 4th bit gets computed in this stage. In the final stage ($k=3$), the carry of 8th bit is computed by using 4th carry and block *propagate* signal of bit-4 through 7. The carries of other bit positions are also computed in parallel. Once the carry is available, the *Sum* output is computed by evaluating $S_i = p_i \oplus C_i$. In Fig. 2(a), the black boxes denote the computation of *propagate/generate* whereas the grey boxes denote computation of *generate* only.

The stage-wise propagation of carries is further illustrated in Fig. 2(a) by expanding two of the stages ($k=0, 1$) of KSA. The solid arrows denote presence of carry whereas the dashed arrows denote absence of carry. The bit-wise *propagates/generates* are computed in setup stage ($k=0$). In 1st stage, they are combined in pairs to produce group carries. Three situation are illustrated in Fig 2(a): (i) $(p, g)=(1,1)$ are asserted and the group carry is generated, (ii) $(p, g)=(0,1)$ and the carry is sunk in, (iii) $(p, g)=(1,0)$ so there is no group carry. The stage-wise group carries for the entire 8-bit adder are shown in Fig. 2(b). It can be observed from this diagram that carry output of 7th bit depends only on carry output of bits 3 and 1. Similarly, carry output of 6th bit depends only on carry output of bit 2. Therefore, even and odd carries are computed independent of each other resulting in area overhead compared to other sparse tree adders. However, the advantage comes from the regularity in layout, balanced fan-out and speed. As we explore in the next paragraphs, this kind of structure also yields fault tolerance due to independent (or parallel) computation of even/odd bits that is done in order to achieve speed.

In the following subsections, we will elaborate the impact of faults in the KSA. For the sake of simplicity we assume that the faults are confined between stages 1 to $\log_2 N$. Moreover, the setup stage (i.e., $k=0$) is assumed to be fault-free.

B. Faults in propagate

A stuck-at 0 fault in propagate may block the carry from propagating to the output. On the other hand, a stuck-at 1 fault may undesirably propagate the carry where it should kill the carry. In both situations, the wrong computation will appear at the *Sum* output. For example, let us consider an 8-bit KSA (Fig. 2(a)) to observe the impact of faults in the propagate signals

$$\begin{aligned} P_{3:0} &= P_{3:2} P_{1:0}, & P_{4:1} &= P_{4:3} P_{2:1}, \\ P_{5:2} &= P_{5:4} P_{3:2} \text{ and } P_{6:3} &= P_{6:5} P_{4:3} \end{aligned} \quad (2)$$

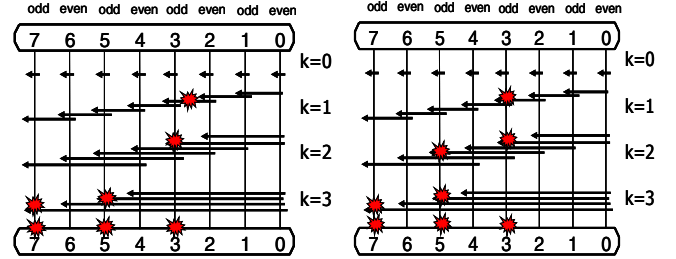


Fig. 3 (a) Effect of fault in *propagate* and, **(b)** effect of fault in *generate*

where $P_{i:j}$ is the block propagate between i^{th} and j^{th} bits.

From equation 2 it is evident that a fault (of any type) in $P_{3:2}$ can only affect $P_{3:0}$ and $P_{5:2}$ whereas $P_{4:1}$ and $P_{6:3}$ are computed correctly. Similarly, a fault in $P_{4:3}$ can only affect $P_{4:1}$ and $P_{6:3}$. Note that, $P_{3:0}$, $P_{4:1}$, $P_{5:2}$ and $P_{6:3}$ are neighboring group propagate signals in bit-3, bit-4, bit-5 and bit-6 (i.e., odd, even, odd, even fashion). Therefore, this example further illustrates the fact that fault in even bit *propagate* can affect only the even bit *propagates* (and consecutively the *Sum* outputs) and vice versa is true for odd bits. Fig. 3(a) elucidates the fault in $P_{3:2}$ further. The fault corrupts the $P_{3:0}$ first (at $k=2$) and $P_{5:2}$, $P_{7:2}$ next (at $k=3$). These faults corrupt the odd *Sum* outputs (i.e., Sum_3 , Sum_5 and Sum_7) in the final stage.

C. Faults in generate

Similar to faults in propagate, a stuck-at 0 fault in generate may kill a carry from generating whereas a stuck-at 1 fault may undesirably produce an intermediate carry. Other types of faults (e.g., complete short/open) will manifest differently. For example, in the 8-bit KSA (Fig. 2(a)), the carry-outs C_5 , C_6 and C_7 are given by

$$C_5 = G_{5:2} + P_{5:2} G_{1:0} = (G_{5:4} + P_{5:4} G_{3:2}) + (P_{5:4} P_{3:2}) G_{1:0} \quad (3)$$

$$\begin{aligned} C_6 &= G_{6:3} + P_{6:3} G_{2:0} = (G_{6:5} + P_{6:5} G_{4:3}) + (P_{6:5} P_{4:3}) G_{2:0} \\ &= (G_{6:5} + P_{6:5} G_{4:3}) + (P_{6:5} P_{4:3}) (G_{2:1} + P_{2:1} G_0) \end{aligned} \quad (4)$$

$$\begin{aligned} C_7 &= G_{7:4} + P_{7:4} G_{3:0} = (G_{7:6} + P_{7:6} G_{5:4}) + (P_{7:6} P_{5:4}) G_{3:0} \\ &= (G_{7:6} + P_{7:6} G_{5:4}) + (P_{7:6} P_{5:4}) (G_{3:2}) + P_{3:2} G_{1:0} \end{aligned} \quad (5)$$

From the above expressions, it is evident that a fault in $G_{3:2}$ can affect only carry-outs C_5 and C_7 whereas C_6 can be computed correctly (since C_6 is independent of $G_{3:2}$ term). Fig. 3(b) illustrates the fault in $G_{3:2}$ further. The fault corrupts the C_3 first (at $k=2$) and C_5 , C_7 next (at $k=3$). These faults corrupt the odd *Sum* outputs (i.e., Sum_3 , Sum_5 and Sum_7) in the final stage.

Note that the conclusions drawn above for the faulty *propagate* and *generate* signals are *general and not dependent on any particular fault model*.

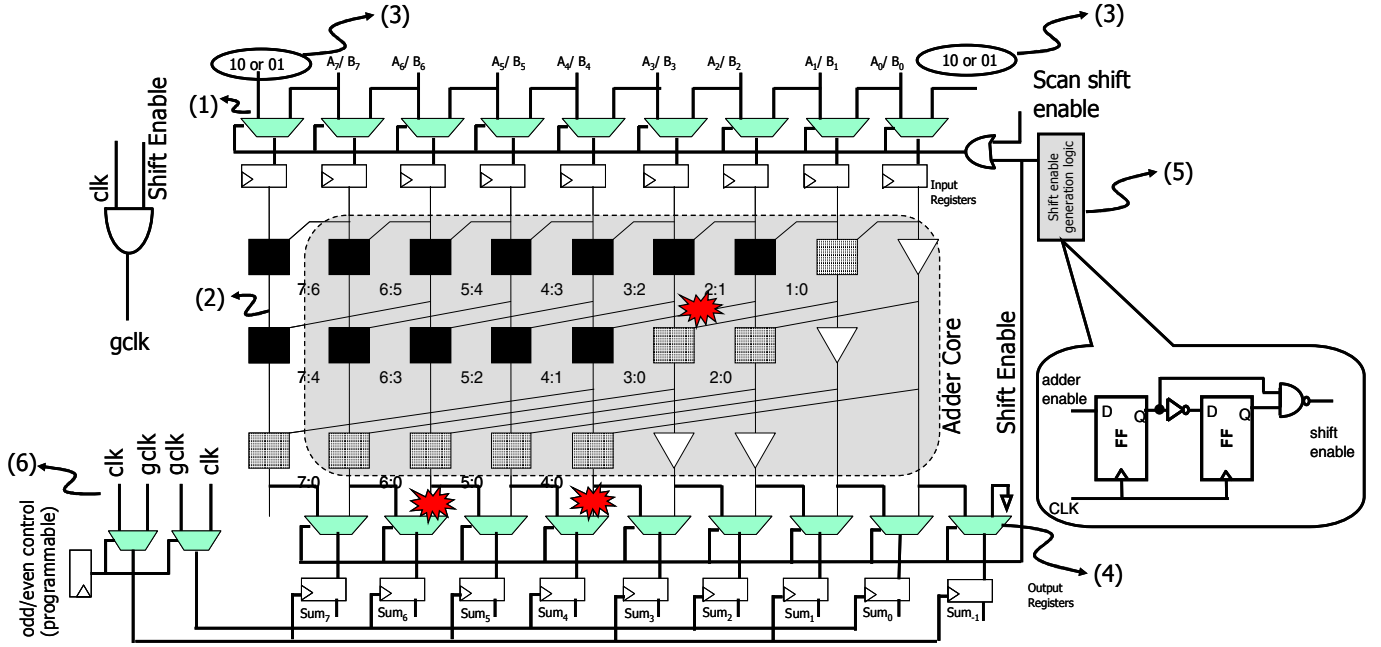


Fig. 4 Block diagram of the proposed 8-bit fault tolerant Kogge-Stone adder (the redundant stage is un-shaded). The components required for fault tolerance are numbered for the sake of clarity. The effect of fault in 3rd bit is also shown

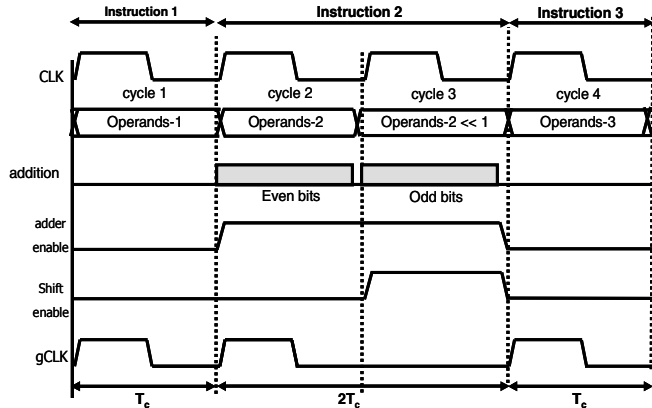


Fig. 5 Detailed timing diagram

III. FAULT TOLERANT KSA

In this section, we describe the proposed fault tolerant Kogge-Stone adder followed by the simulation results.

A. Structure of fault tolerant Kogge-Stone adder

The overall structure for an 8-bit fault tolerant adder example is illustrated in Fig. 4. It can be observed that following components are required for fault tolerance:

1. *Multiplexers at the inputs:* Multiplexers are required at the inputs for shifting the operands to the *left side* by 1-bit during second cycle (for the *faulty* adder). We observe that scan flip-flops [9] are generally used for testability purpose where the test patterns are shifted sequentially in test mode, applied to the circuit-under-test and responses are scanned out to attain required fault coverage. The scan flops utilize multiplexer at the D-input to clock-in either test data or normal data (based on the *Scan Shift Enable* signal). In this work, we reuse these multiplexers for shifting the operands. This is achieved by simply ORing the *scan shift enable* signal with the operand *shift*

TABLE I: PROCESSOR CONFIGURATION

Processor	8-way issue, 128 RUU, 64 LSQ, 4 integer ALUs, 1 integer mul/div units, 4 FP ALUs, 4 FP mul/div units, 2 Wr/Rd ports
Branch Prediction	Combined, 16-entry RAS, 512-set 4-way BTB, 8 cycle mis-prediction penalty
Caches	64KB 2-way 2 cycle ID L1, 2MB 2-way 18 cycle L2
Main Memory	300 cycle latency, 32-byte wide bus

enable to control the scan flop multiplexers (Fig. 4).

2. *Extra bit computation:* An extra computation column is added in the Kogge-Stone tree to make sure that even if the fault affects Sum_7 , the correct Sum can be computed by this column and restored in cycle-2. This extra column (un-shaded in Fig. 4) contributes towards the area overhead.

3. *Application of non-controlling values in LSB and MSB:* During cycle-2, non-controlling values must be applied to the LSB so that $(p_0, g_0) = (1, 0)$. This is required since LSB is not used for computation during cycle-2 in faulty adders. Forcing $p_0=1$ makes the group propagate signal (i.e., the product of individual propagates) independent of p_0 . Similarly, $g_0=0$ is required to suppress false carry input from the LSB. Both of these conditions are achieved by providing 10 or 01 inputs through the multiplexer. The MSB from extra column (i.e., 8th bit) output is ignored during cycle-1. However, a 10 or 01 input can be forced to ensure non-controlling p/g values during cycle-1 of *fault-free* as well as *faulty* adders.

4. *Multiplexers at output:* Multiplexers are required at the outputs for shifting the partially correct $Sums$ to the *right side* by 1-bit during second cycle in *faulty* adder. We again leverage the scan flop's for the shifting of outputs. Note that the $Sums$ are shifted *right* to preserve the bit ordering. For example, in Fig. 4, Sum_4 and Sum_6 are faulty in cycle-1 whereas other $Sums$ are computed correctly. After *right* shifting the $Sums$ in cycle-2 and re-computation, the correct computations ($Sum_1, Sum_3, Sum_5, Sum_7$) from cycle-1 is stored in $Sum_0, Sum_2, Sum_4, Sum_6$ registers whereas the new computations are

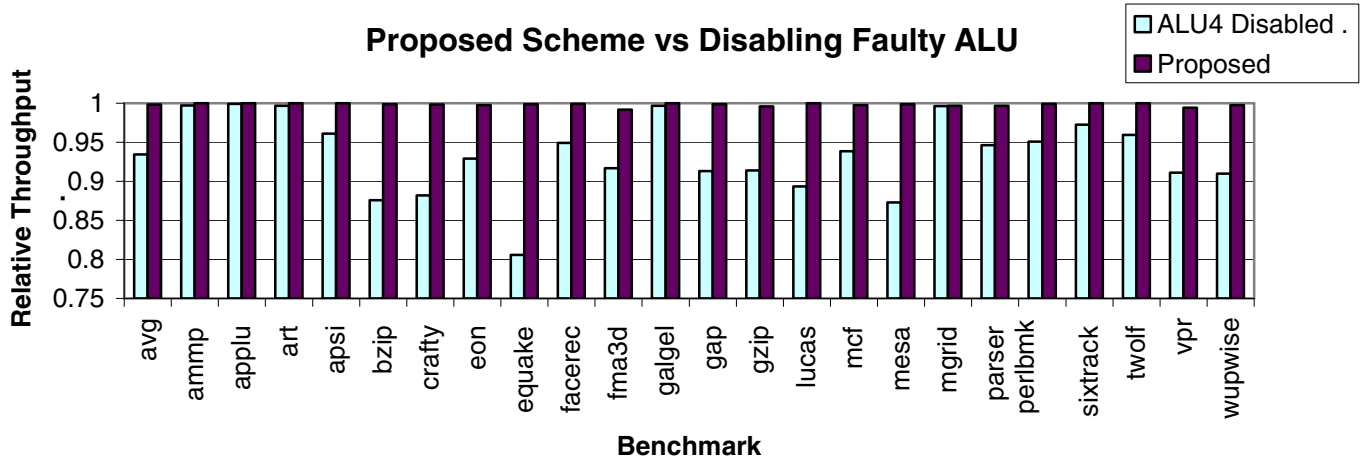


Fig. 6 SimpleScalar results. The throughput improvement (compared to scenario when faulty adder is completely discarded) for SPEC2000 benchmarks are shown

clocked-in to the Sum_1 , Sum_2 , Sum_3 , Sum_5 registers. The final Sum of the addition can be obtained from Sum_1 through Sum_7 registers.

5. *Generation of shift enable signal:* The shift enable signal is generated by a mono-pulse generation circuitry as shown in the inset of Fig 4. A mono-shot pulse is generated whenever the adder is scheduled (i.e., $adder\ enable = 1$). The operands are shifted left and the outputs are shifted right whenever the *shift enable* is asserted.

6. *Clocking of output registers:* If manufacturing test determines that the fault is located in *even* bits then the corresponding *even* registers should be programmed (Fig. 4) to receive gated clock ($gclk$) whereas the *odd* registers are programmed to receive the normal clock (clk). This is done to prevent destruction of correct data in *even* bit registers (that were computed in cycle-1) during cycle-2. Similarly, programming bit should be configured if the fault is located in *odd* bits. The $gclk$ is generated by ANDing the normal clock with *shift enable* signal. The timing diagram of *shift enable* and $gclk$ is shown in Fig. 5 for the example discussed in Section-I.

The area overhead in the proposed fault tolerant adder is minimal and essentially comes from the extra computation stage, shift enable/gated clock generation logic and 1-bit programmable register for clocking the output flops. It should also be noted that the proposed technique is generic and can be extended to design fault tolerant sparse-tree adders (e.g., Han-Carlson, Brent-Kung [8] etc). Since these adders have less redundancy present, the fault tolerance can be achieved by performing more than two shifts and re-computations.

B. Simulation results

We implemented the 64-bit fault-tolerant KSA in Verilog. We synthesized the circuit in Synopsys Design Compiler [10] using 180nm IBM technology. To estimate the overhead due to the fault recovery features, we provided the same constraints to both the nominal circuit and the proposed circuit. The total area overhead was 0.9%, and this was due to the additional computation column and the logic required for modifying the clocking. The performance overhead of the fault-free adder (during normal operation) was 0.3%, which was primarily due to the additional load on the clock network.

IV. APPLICATION IN SUPERSCALAR PIPELINE

In last section, we proposed the fault tolerant Kogge-Stone Adder which uses two-clock cycles to tolerate the faults. In this section, we present a micro-architectural solution to minimize the

throughput penalty if faulty adders are present in a superscalar pipeline.

In superscalar processors, there are typically several functional units (integer ALUs, integer multipliers, floating point ALUs, etc) of the same type. Consider a processor that has a faulty integer ALU. In this case, the designers would either have to discard the faulty chip, or disable the faulty functional unit. The first option results in a significant yield impact, particularly as the number of faults increase as expected in scaled technologies. The second solution, that is, to disable the faulty ALU, is significantly more attractive since it allows the faulty chip to be salvaged, albeit at a significant throughput penalty due to the availability of fewer ALUs.

For this work, we assume that the faulty functional unit is the adder core of the integer ALU. Instead of completely disabling the faulty adder, we use it for computation, but employ adaptive clocking to allow it to perform computations in two-clock cycles. There are two major challenges to employing this scheme. The first one is to ensure that most of the instructions are not executed by this *faulty* functional unit. Here, we assign the lowest priority to this *faulty* ALU, such that we only use it for computation whenever all other *non-faulty* ALUs are in use. In addition to this, we must ensure that dependent instructions are not woken up before the *faulty* adder has completed the computation.

Both these tasks require some slight modification to the schedule and issue logic of the superscalar processor. The schedule and issue logic is responsible for issuing of instructions to the functional units. Whenever an instruction is ready to be issued (all its other dependent instructions have completed), the scheduler locates an available functional unit. If a functional unit is available, the scheduler issues the instruction to the functional unit, and informs the wake-up logic to wakeup instructions that are dependent on the one that has been issued after a given number of cycles. In addition, each functional unit has a REQUEST signal to indicate that it is available for execution.

In order to implement the modification to the scheduling policy, each functional unit requires an additional status bit to indicate when it is faulty. The scheduler checks this FAULT bit in addition to the REQUEST signal. If the FAULT bit is set, the scheduler wakes up dependent instructions after two-clock cycles. Note that the FAULT bit is set during test and does not change during execution, thus the performance and power overhead is small.

We modified *SimpleScalar* [11] to accommodate the changes in the scheduling policy. We used ref inputs, fast forwarded 500

million instructions and simulated 1 billion instructions for 23 of 25 benchmark in the SPEC 2000 toolset (we had difficulty simulating *vortex* and *swim*). The processor configuration is shown in Table 1. As shown in the configuration, the integer execution unit consists of four integer ALUs. Assuming that one of the ALUs had a faulty adder core, we simulated two scenarios: (a) disabling the faulty ALU, and (b) applying the proposed scheduling policy as described above.

Fig. 6 shows our results for SPEC2000 benchmarks. In virtually all benchmarks, disabling a faulty ALU results in 5-10% IPC loss. However, by applying our scheme, the IPC loss is reduced to below 0.5%. In most benchmarks, there was less than a 0.1% decrease in IPC. The average improvement in throughput by using our technique (instead of disabling the faulty adder) is found to be $\sim 7.14\%$. Note that although we have considered only one faulty adder out of four available adders in the pipeline, the proposed technique can be easily extended when more than one faulty adder is present.

V. RELATED WORK

Several techniques have been proposed in past to tolerate various kinds of defects in arithmetic and logic circuits. In [1], the authors isolate the critical paths of random logic circuits by proper synthesis and sizing. If the chip suffers from timing failures in critical paths, the output is evaluated in two-clock cycles. This allows them to maintain high yield and rated clock frequency at the cost of slight throughput degradation due to occasional two-cycle operations. However, it does not address the large delay defects or open/short scenario that is under consideration.

In [2], the authors proposed a stuck-at tolerant Kogge-Stone Adder. The idea is to add an extra Han-Carlson (HC) stage which computes the even bits from odd bits (or vice versa) for defective adders. Therefore, stuck-at faults are tolerated at the cost of area/delay overhead (due to HC stage and multiplexers). The authors quote a 16% increase in delay during fault-correcting mode. If applied in a superscalar data path, this would require a 16% in reduction in frequency, and consequently a significant reduction in throughput. In fact, the throughput degradation introduced by performing the fault correcting in [2] is worse than the throughput loss that would be seen if the adder was entirely disabled. Therefore, it would be difficult to apply [2] directly to a high speed data path.

The popular fault tolerant technique, Triple Modular Redundancy (TMR) [4] assumes that only 1-out-of-3 adders can be faulty at a time. Therefore, it instantiates the adder thrice and uses a voter to produce the majority output. This leads to large area/delay overhead. To avoid the area overhead, Time Shared Triple Modular Redundancy (TSTMR) [12] divides the operands (of width N) into three parts and uses three $\lfloor N/3 \rfloor$ size adders to compute the addition and a voter to choose 1-out-of-3 output. Therefore the entire addition requires three clock cycles. Since operand widths are usually divisible by four, Quaternary Time Redundancy (QTR) [3] adder is proposed to utilize this fact and improve the area/delay overhead compared to TSTMR. In this technique the operands are divided-by-four and one quarter is instantiated three times with a majority voter. The entire computation is performed in four clock cycles. Note that all of the above techniques (i.e., TMR, TSTMR and QTR) are *concurrent error detection/correction* techniques.

VI. CONCLUSIONS

Defects can significantly impact the yield of high performance design. With aggressive scaling and lithographic limitations, a large number of defects can be observed that can manifest themselves as stuck-0/stuck-1, opens or shorts. Hence, to improve yield we presented a technique to utilize the inherent spatial redundancy

present in high-speed circuits in order to tolerate *any kind of fault*. We apply this technique to a Kogge-Stone adder to achieve fault tolerance even in presence of faults. The faulty adder is operated in two-clock cycles (instead of one-clock cycle) for complete computation of correct *Sum*. The results show that the proposed technique has very low overhead in terms of area and delay. It can be used to tolerate any number of faults as long as they are confined to either even or odd bits with small area and performance overhead. We proposed a micro-architectural solution to utilize the *faulty* adder efficiently in a superscalar pipeline and minimize the throughput degradation (due to two-cycle operations). The technique can also be extended to other sparse tree adders where the defect can be tolerated by adaptive 1-cycle (for *good* adders) and N -cycle (for *faulty* adders) operations (where $N \geq 2$).

VI. ACKNOWLEDGEMENTS

The authors acknowledge the support of the Gigascale Systems Research Focus Center, one of five research centers funded under the Focus Center Research Program, a Semiconductor Research Corporation program.

REFERENCES

- [1] S. Ghosh et al., "Tolerance to small delay defects by adaptive clock stretching," *IOLTS*, 2007.
- [2] P. Ndai et al., "Fine-grained redundancy in adders," *ISQED*, 2007.
- [3] W. J. Townsend et al., "Quadruple time redundancy adders," *DFT*, 2003.
- [4] B. W. Johnson, "Design and analysis of fault tolerant digital systems," Addison Wesley Publishing Company, 1989.
- [5] J. H. Patel et al., "Concurrent error detection in ALUs by recomputing with shifted operands," *Trans. Comput.*, 1982.
- [6] K. Wu et al., "Algorithm level RE-computing with shifted operands - a register transfer level concurrent error detection technique," *ITC*, 2000.
- [7] P. M. Kogge et al., "A parallel algorithm for the efficient solution of a general class of recurrence equations," *TComp*, 1973.
- [8] J. Rabaey, "Digital Integrated Circuits: A Design Perspective", Prentice Hill, Second Edition, 2003.
- [9] M. Breuer, "Digital system testing and testable design," IEEE Press, 1995.
- [10] Synopsys Design Compiler, www.synopsys.com.
- [11] T. Austin et al., "SimpleScalar: An infrastructure for computer system modeling," *Computer*, 2002.
- [12] Y. M. Hsu, "Concurrent error correcting arithmetic processors," PhD dissertation, University of Texas at Austin, 1995.