Pipelining Saturated Accumulation

Karl Papadantonakis, Nachiket Kapre, *Student Member, IEEE,* Stephanie Chan, and André DeHon, *Member, IEEE*

***Abstract*— Aggressive pipelining and spatial parallelism allow integrated circuits (*e.g.*, custom VLSI, ASICs, FPGAs) to achieve high throughput on many Digital Signal Processing applications. However, cyclic data dependencies in the computation can limit parallelism and reduce the efficiency and speed of an implementation. Saturated accumulation is an important example where such a cycle limits the throughput of signal processing applications. We show how to reformulate saturated addition as an associative operation so that we can use a parallel-prefix calculation to perform saturated accumulation at any data rate supported by the device. This allows us, for example, to design a 16-bit saturated accumulator which can operate at 280MHz on a Xilinx Spartan-3 (XC3S-5000-4) FPGA, the maximum frequency supported by the component's DCM.**

***Index Terms*— Computer Arithmetic, Saturated Arithmetic, Accumulation, Parallel Prefix**

I. Introduction

Over the last few decades, a large fraction of the clock rate increases in microprocessors has come from increased pipelining (*e.g.*, [1]) to the point where modern processors run with about 10 gate delays (*e.g.*, fanout-four inverter (FO4) delays) per clock cycle. ASICs, ASIC-based DSPs, and FPGAs have traditionally not been pipelined as heavily, but their clock rates could also be increased by heavy pipelining (*e.g.*, [2], [3]). For acyclic designs (feed forward dataflow), it is always possible to pipeline designs down to just a few gate delays (or Lookup-Table evaluations for FPGAs). It may be necessary to pipeline the interconnect (*e.g.*, [3], [4]), but the transformation can be performed and automated.

However, when a design has a cycle with a large latency but only a few registers in the path, we cannot immediately pipeline to this limit. No legal retiming [5] will allow us to reduce the ratio between the total cycle logic delay (*e.g.*, number of gates in the path) and the total registers in the cycle. This often prevents us from pipelining the design all the way down to the gate plus local interconnect level and consequently prevents us from operating at peak throughput to use the device efficiently. This phenomena also impacts processors; even though the processor is heavily pipelined, loop-carried data dependencies implied by the cycle prevents the processor from issuing instructions for the single instruction stream at the full clock rate. We can use these devices efficiently by interleaving parallel problems in C-slow (*e.g.*, [5], [6]) or multithreaded (*e.g.*, [7], [8]) fashion, but the throughput delivered to a single data stream is limited. In a spatial pipeline of streaming operators, the throughput of the slowest operator will serve as a bottleneck, forcing all operators to run at the slower throughput, preventing us from achieving high efficiency.

Saturated accumulation (Section II-A) is a common signal processing operation with a cyclic dependence which prevents aggressive pipelining. As such, it can serve as the rate limiter in streaming applications (*e.g.*, Sections II-B and II-C). While non-saturated accumulation is amenable to associative transformations (*e.g.*, delayed addition [9] or block associative reduce trees (Section II-E)), the non-associativity of the basic saturated addition operation prevents these direct transformations.

In this paper we show how to transform saturated accumulation into an associative operation (Section III). Once transformed, we use a parallel-prefix computation to avoid the apparent cyclic dependencies in the original operation (Section II-G). As a concrete demonstration of this technique, we show how to accelerate a 16-bit accumulation on a Xilinx Spartan-3 (X3CS-5000-4) FPGA [10] from a cycle time of 11.3ns to a cycle time below 3.57ns (Section V). The techniques introduced here are general and allow us to pipeline saturated accumulations to any throughput which the device can support. The parallel-prefix techniques further allow our designs to take in multiple inputs per cycle and produce multiple outputs per cycle (Section VI-A). As a result, we can design our saturated accumulation to match any throughput which the device's I/O can support.

Th techniques presented here wer motivated by the high latency of programmable interconnect in FPGAs, and the results were first reported at an FPGA conference [11]. Nonetheless, the techniques are general and apply to any technology, including ASICs which can benefit from microarchitectural transforms which enabled greater pipelining [2] and superscalar and VLIW processors which benefit from transformations which increase instruction-level parallelism. For this journal version, we have included detailed proofs and more tutorial descriptions which could not be included in the shorter conference version, expanded the prior work comparisons, illustrated how to exceed the one result per cycle bound, and included discussion on generalization of these techniques beyond saturated accumulation.

II. Background

A. Saturated Accumulation

Efficient implementations of arithmetic on real computing devices with finite hardware must deal with the fact that integer addition is not closed over any non-trivial finite subset of the integers. Some computer arithmetic systems deal with this by using addition modulo a power of two (*e.g.*, addition modulo 2^{32} is provided by most microprocessors). However, for many applications, modulo addition has bad effects, creating aliasing between large numbers which overflow to small numbers and small numbers. Consequently, one is driven to use a large modulus (a large number of bits) in an attempt to avoid this aliasing problem.

An alternative to using wide datapaths to avoid aliasing is to define saturating arithmetic. Instead of wrapping the arithmetic result in modulo fashion, the arithmetic sets bounds and clips sums which go out of bounds to the bounding values. That is, we define a saturated addition as:

Manuscript received July 22, 2007. Revised December 31, 2007.
K. Papadantonakis is with Myricom, Inc.
N. Kapre is with the California Institute of Technology.
S. Chan is with Numerica Corp.
A. DeHon is with the University of Pennsylvania.
Contact author: A. DeHon <andre@ieee.org>

TABLE I
ACCUMULATION EXAMPLE

input (x_i)	0	50	100	100	11	-2
modulo sum (mod 256)	0	50	150	250	5	3
satsum (y_i) (maxval=256)	0	50	150	250	255	253

```
SA(a,b,minval,maxval) {
  tmp=a+b; // tmp can hold sum
           // without wrapping
   if (tmp>maxval)
           return(maxval);
   elseif (tmp<minval)
           return(minval);
   else
           return(tmp)
}
```

Since large sums cannot wrap to small values when the precision limit is reached, this admits economical implementations which use modest precision for many signal processing applications.

A saturated accumulator takes a stream of input values x_i and produces a stream of output values y_i:

$$y_i = \texttt{SA}(y_{i-1}, x_i, \texttt{minval}, \texttt{maxval}) \tag{1}$$

Table I gives an example showing the difference between modulo and saturated accumulation.

B. Example: ADPCM

The decoder in the Adaptive Differential Pulse-Compression Modulation (ADPCM) application in the `mediabench` benchmark suite [12] provides a concrete example where saturated accumulation is the bottleneck limiting application throughput. Figure 1 shows the dataflow path for the ADPCM decoder. The only cycles which exist in the dataflow path are the two saturated accumulators. Note that we can accommodate pipeline delays at the beginning of the datapath, at the end of the datapath, and even in the middle between the two saturated accumulators (annotated in Figure 1) without changing the semantics of the decoder operation. As with any pipelining operation, such pipelining will change the number of cycles of latency between the input (`delta`) and the output (`valpred`).

Previous attempts to accelerate the mediabench applications for spatial (hardware or FPGA) implementation have achieved only modest acceleration on ADPCM (*e.g.*, [13]). This has led people to characterize ADPCM as a serial application. With the new transformations introduced here, we show how we can parallelize this application.

If we had multiple, independent ADPCM streams to decode, we could C-slow (*e.g.*, [5], [6]) the design and run C interleaved streams through a highly pipelined datapath. The techniques introduced here address the cases where we either want to accelerate a single stream or where it is advantageous to avoid the additional latency, complexity, or state storage required in order to interleave streams.

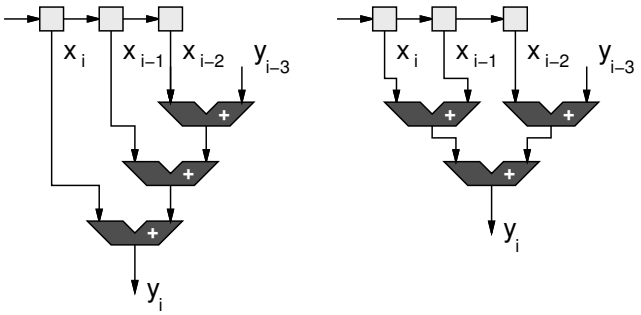


Fig. 2. Using Associativity to Reduce Serial Adder Delay

C. Example: Telecommunication Standards

Many telecommunication standards (*e.g.*, ETSI/3GPP enhanced full rate and adaptive multi rate speech processing, ITU G.723.1, ITU G.729) provide specifications or reference implementations based on limited-precision saturated arithmetic. For new implementations of the standard to be credible, it is advantageous, and often necessary, for the implementations to provide bit-exact results which match the standard. The technique we demonstrate here allows parallelism and pipelining in the saturated accumulations while remaining bit-exact with the serial reference specification.

D. Associativity

Both infinite precision integer addition and modulo addition are associative. That is: $(A + B) + C = A + (B + C)$. However, saturated addition is not associative. For example, consider: 250+100-11

infinite precision arithmetic:
(250+100)-11 = 350-11 = 339
250+(100-11) = 250+89 = 339
modulo 256 arithmetic:
(250+100)-11 = 94-11 = 83
250+(100-11) = 250+89 = 83
saturated addition (max=255):
(250+100)-11 = 255-11 = 244
250+(100-11) = 250+89 = 255

Consequently, we have more freedom in implementing infinite precision or modulo addition than we do when implementing saturating addition.

E. Associative Reduce

When associativity holds, we can exploit the associative property to reshape the computation to allow pipelining. Consider a modulo-addition accumulator:

$$y_i = y_{i-1} + x_i \tag{2}$$

Unrolling the accumulation sum, we can write:

$$y_i = ((y_{i-3} + x_{i-2}) + x_{i-1}) + x_i \tag{3}$$

Exploiting associativity we can rewrite this as:

$$y_i = ((y_{i-3} + x_{i-2}) + (x_{i-1} + x_i)) \tag{4}$$

Whereas the original sum had a series delay of 3 adders, the reassociated sum has a series delay of 2 adders (See Figure 2). In general, we can unroll this accumulation $N-1$ times and reduce the computation depth from $N-1$ to $\log_2(N)$ adders.

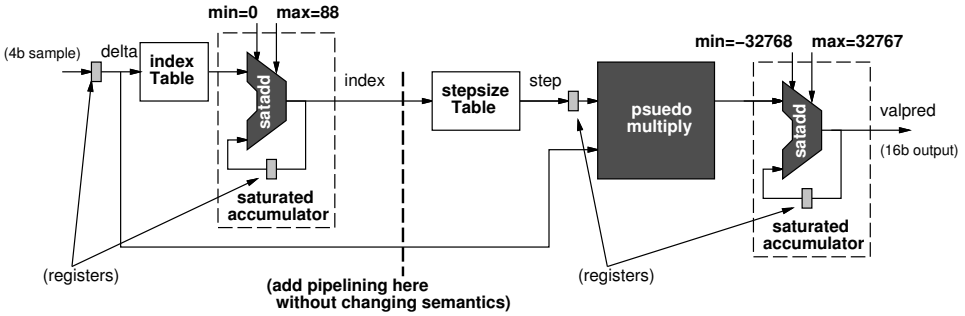
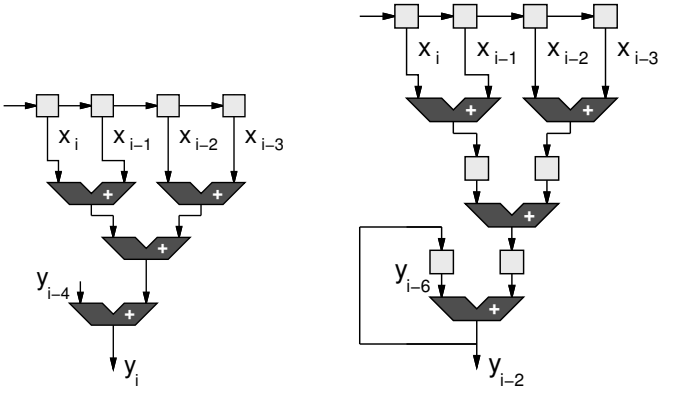


Fig. 1. Dataflow for ADPCM Decode



(a) Fast Path for Feedback

(b) Pipelining Inputs to Match Feedback Cycle

Fig. 3. Asymmetric Associativity to Reduce Delay around Feedback Cycle

F. Asymmetric Associative Reduction and Partial Unrolling

Associativity actually allows us to take things a step further. Instead of building balanced reduce trees, we can build unbalanced trees that allow us to reduce the delay on some inputs more than others (See Figure 3(a)). In particular, this allows us to minimize the delay on the feedback cycle. Consequently, the delay in the cyclic path can be a single operation delay rather than the $O(\log(N))$ delay for a balanced tree. In many cases, it will suffice to unroll the loop only N additions in order to cover the delay of the single operator in the feedback. As shown in Figure 3(b), the associative reduction preceding the feedback path can now be pipelined to match the achievable clock rate of the final feedback cycle.

G. Parallel-Prefix Tree

In Section II-E, we noted we could compute the final sum of N values in $O(\log(N))$ time using $O(N)$ adders. With only a constant factor more hardware, we can actually compute all N intermediate outputs: $y_i, y_{i-1}, \ldots y_{(i-(N-1))}$ (*e.g.*, [14], [15]).

We do this by computing and combining partial sums of the form $\mathrm{S}[s,t]$ which represents the sum: $x_s+x_{s+1}+\ldots x_t$. When we build the associative reduce tree, at each level k, we are combining $\mathrm{S}[(2j)\,2^k, (2j+1)\,2^k-1]$ and $\mathrm{S}[(2j+1)\,2^k, 2\,(j+1)\,2^k-1]$ to compute $\mathrm{S}[(2j)\,2^k, 2\,(j+1)\,2^k-1]$ (See Figure 4). Consequently, we eventually compute prefix spans from 0 to 2^k-1 (the $j=0$ case), but do not eventually compute the other prefixes. The observation to make is that we can combine the $\mathrm{S}[0, 2^k-1]$ prefixes with the $\mathrm{S}[2^{k_0}, 2^{k_0}+2^{k_1}-1]$ spans ($k_1 < k_0$) to compute the intermediate results. To compute the full prefix sequence $(\mathrm{S}[0,1], \mathrm{S}[0,2], \ldots \mathrm{S}[0,N-1])$, we add a second (reverse) tree to compute these intermediate prefixes. At each tree level where we have a compose unit in the forward, associative reduce tree, we add (at most) one more, matching, compose unit in this reverse tree. The reverse, or prefix, tree is no larger than the reduce tree; consequently, the entire parallel-prefix tree is at most twice the size of the associative reduce tree. Figure 4 shows a width 16 parallel-prefix tree for associative accumulation. For a more tutorial development of parallel-prefix computations see [14], [16].

We can also build asymmetric parallel-prefix trees to minimize the delay on the critical feedback cycle as described in Section II-F. In Figure 4, the $y[-1]$ input allows the $y[15]$ term to feedback to the final adder stage with a single adder delay of latency in the case where we have partially unrolled a longer accumulation stream so we can process sixteen inputs in a single adder-delay cycle time.

H. Delayed Addition

For associative operations, we can use a redundant representation for the accumulation sum and exploit delayed addition [9] to achieve full-adder-bit-level pipelining. This will likely result in a more compact implementation than the example in the previous section. However, the associative reduce tree will be more directly applicable to our solution with the transformations introduced in the next section (Section III).

I. Prior Work

de Dinechin *et al.* attacked the problem of saturating accumulation at the DSP instruction level [17]. They show how to get a factor of two speedup by cutting the sequence of saturated additions in half and processing the two halves in parallel. Similar to the technique presented here, their algorithm computes revised maximum and minimum values on the second half of the sequence so they can correctly compose and saturate the second half sum with the saturated sum of the first half. They do not show how to recurse their decomposition or generally describe how to achieve greater parallelism. Further, their algorithm only produces the final result y_{N-1}, where N is their saturated accumulation block size, and not the intermediate results $y_0, y_1, \ldots, y_{N-2}$.

Balzola *et al.* show how to achieve bit-exact saturated accumulation by implementing an N-input saturated adder [18], [19]. Their N-input adder structure is similar in spirit to the unrolled associative additions in Sections II-E and II-F in that they unroll by a factor of N to accumulate N values in a delay slightly

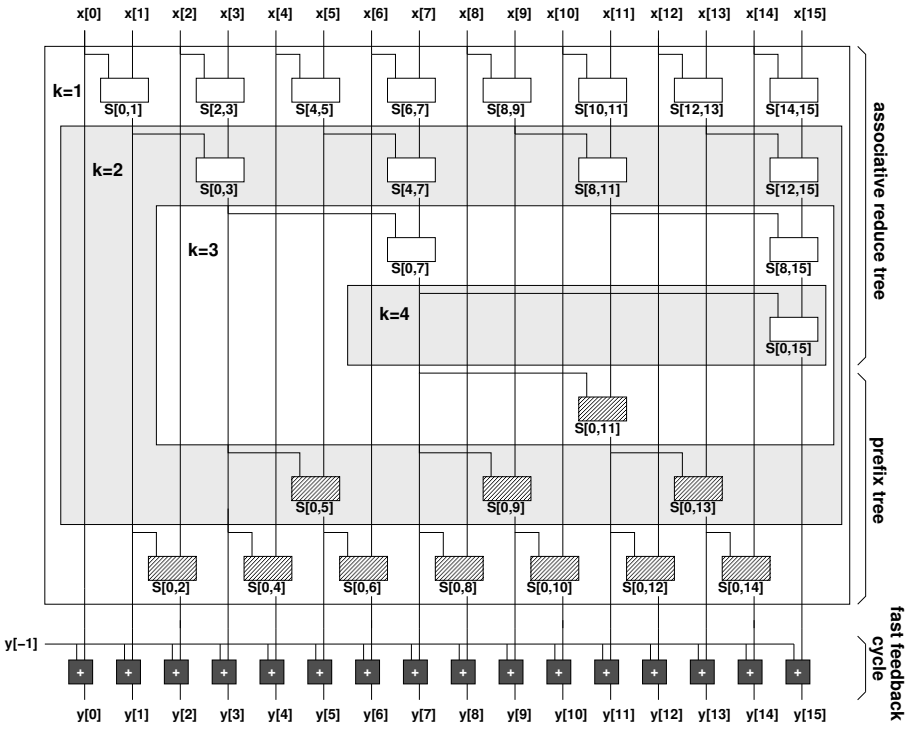


Fig. 4. 16-input Parallel-Prefix Tree

greater than one carry-propagate adder delay. Since the saturated addition is not associative, they independently compute additions for all possible saturations in the prefix; on the critical feedback path, they only need to select the appropriate inputs rather than perform a complete addition for all but the final addition. As a result, their area grows as $O(N^2)$, and the delay of their unrolled is one adder delay plus $O(N)$ mux delays. Asymptotically, the design provides only a constant speedup (*i.e.*, from one adder delay per input to one mux delay per input). Their design also only produces the final result of the N-input accumulation, y_{N-1}, and not the intermediate results $y_0, y_1, \ldots, y_{N-2}$.

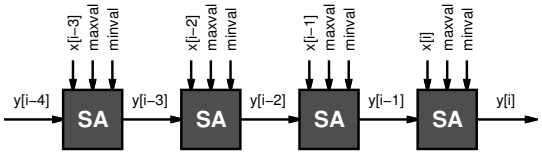
In contrast, we show how to make saturated accumulation associative (Section III), enabling the use of efficient parallel-prefix techniques (Section II-G). Parallel prefix allows us to achieve arbitrary speedups and to produce all the intermediate results $(y_0, y_1, \ldots, y_{N-2})$ in the accumulation. Further, the parallel-prefix technique allows to keep the area linear $(O(N))$ in the unrolling factor, N. Latency from input (x_i) to output (y_i) is $O(\log(N))$. After presenting our sample implementation results in Section V-D, we provide a quantitative comparison to the speedups and area overheads reported for the Balzola implementation.

III. Associative Reformulation of Saturated Accumulation

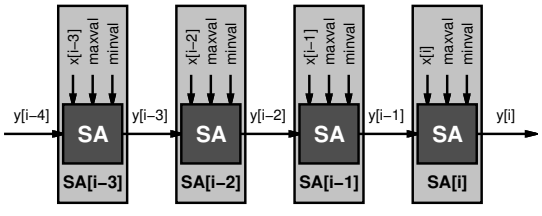
A. Saturated Addition as a Transformation Function

Unrolling the computation we need to perform for saturated additions, we get a chain of saturated additions (SA) as shown in Figure 5(a). We can express SA (Section II-A) as a function using max and min:

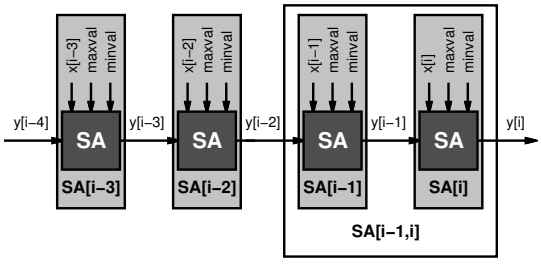
$$\begin{aligned} &\texttt{SA}(y, x, \texttt{minval}, \texttt{maxval}) \\ &\quad = \quad \min(\max((y + x), \texttt{minval}), \texttt{maxval}) \end{aligned} \tag{5}$$



(a) Unrolled Chain of Four Saturated Additions



(b) Viewing Each Saturated Addition as a Transformation Function from One Input to One Output



(c) Composing the Single-Input, Single-Output Functions for a Pair of Connected, Saturated Additions To Define a New Single-Input, Single-Output Transformation Function

Fig. 5. Saturated Addition Sequence Viewed as Function Composition

The saturated accumulation is repeated application of this function. We seek to express this function in such a way that repeated application is function composition. This allows us to exploit the associativity of function composition [20] so we can compute saturated accumulation using a parallel-prefix tree (Section II-G).

Technically, function composition does not apply directly to the formula for SA shown in Equation 5 because that formula is a function of four inputs (having just one output, y). Fortunately, only the dependence on y is critical at each SA-application step; the other inputs are not critical, because it is easy to guarantee that they are available in time, regardless of our algorithm. To understand repeated application of the SA function, therefore, we express SA in an alternate form in which y is a function of a single input and the other "inputs" (x, minval, and maxval) are function parameters:

$$\mathtt{SA}_{[x,m,M]}(y) \stackrel{\text{def}}{=} \mathtt{SA}(y,x,m,M) \tag{6}$$

We define $\mathtt{SA}[i]$ as the ith application of this function, which has $x = x[i]$, $m = \mathtt{minval}$, and $M = \mathtt{maxval}$:

$$\mathtt{SA}[i] \stackrel{\text{def}}{=} \mathtt{SA}_{[x[i],\mathtt{minval},\mathtt{maxval}]} \tag{7}$$

This definition allows us to view the computation as function composition. For example:

$$y[i] = \mathtt{SA}[i] \circ \mathtt{SA}[i{-}1] \circ \mathtt{SA}[i{-}2] \circ \mathtt{SA}[i{-}3](y[i{-}4]) \tag{8}$$

See Figure 5(b).

B. Composing the SA functions

To reduce the critical latency implied by Equation 8, we first combine successive nonoverlapping adjacent pairs of operations (just as we did with ordinary addition in Equation 4). For example:

$$\begin{aligned} y[i] \quad = \quad & ((\mathtt{SA}[i] \circ \mathtt{SA}[i{-}1]) \\ & \circ (\mathtt{SA}[i{-}2] \circ \mathtt{SA}[i{-}3]))\,(y[i{-}4]) \end{aligned}$$

To make this practical, we need an efficient way to compute each adjacent pair of operations in one step:

$$\mathtt{SA}[i{-}1,i] \stackrel{\text{def}}{=} \mathtt{SA}[i] \circ \mathtt{SA}[i{-}1] \tag{9}$$

This composition is shown in Figure 5(c).

Viewed (temporarily) as a function of real numbers, $\mathtt{SA}[i]$ is a continuous, piecewise linear function, because it is a composition of "min", "max", and "+", each of which are continuous and piecewise linear (with respect to each of their inputs). It is a well known fact that any composition of continuous, piecewise linear functions is itself continuous and piecewise linear (we demonstrate this for our particular case below). We can easily visualize the continuity and piecewise linearity of $\mathtt{SA}[i]$ (See Figure 6).

Let us now try to understand the mathematical form of the function $\mathtt{SA}[i{-}1,i]$. As the base functions $\mathtt{SA}[i{-}1]$ and $\mathtt{SA}[i]$ are continuous and piecewise linear, their composition (*i.e.* $\mathtt{SA}[i{-}1,i]$) must also be continuous and piecewise linear. The key thing we need to understand is: how many segments does $\mathtt{SA}[i{-}1,i]$ have? Since $\mathtt{SA}[i{-}1]$ and $\mathtt{SA}[i]$ each have just one bounded segment of slope one, we argue that their composition must also have just one bounded segment of slope 1 and have the form of Equation 6.

We can visualize this fact graphically as shown in Figure 7. Any input below minval or above maxval (Figure 7(b)) into the second SA will be clipped to the constant minval or maxval. Input clipping on the first SA coupled with the add offset on the second can prevent the composition from producing outputs all the way to minval or maxval (Figure 7(a)). So, the extremes will certainly remain flat just like the original SA. Between these extremes, both SAs produce linear shifts of the input. Their cascade is, therefore, also a linear shift of the input and results in a slope one region (Figure 7(c)). Consequently, $\mathtt{SA}[i{-}1,i]$ has the same form as $\mathtt{SA}[i]$ (Equation 6). As we observed, the composition, $\mathtt{SA}[i{-}1,i]$, does not necessarily have $m = \mathtt{minval}$ and $M = \mathtt{maxval}$. However, if we allow arbitrary values for the parameters m and M, then the form shown in Equation 6 is closed under composition. This allows us to regroup the computation to reduce the number of levels in the computation.

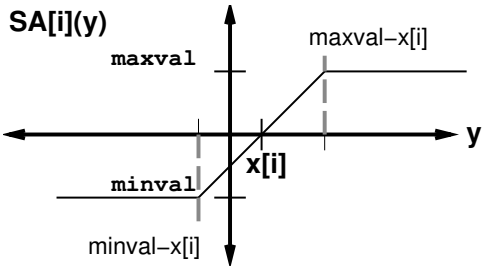


Fig. 6. Transformation Performed by One Saturated Addition

C. Composition Formula

We have just proved that the form $\mathtt{SA}_{[x,m,M]}$ is closed under composition. However, to build hardware that composes these functions, we need an actual formula for the $[x,m,M]$ tuple describing the composition of any two SA functions $\mathtt{SA}_{[x1,m1,M1]}$ and $\mathtt{SA}_{[x2,m2,M2]}$.

Each SA is a sequence of three steps: TRanslation by x, followed by Clipping at the Bottom m, followed by Clipping at the Top M. We write these three primitive steps as $\mathtt{tr}_x$, $\mathtt{cb}_m$, and $\mathtt{ct}_M$, respectively:

$$\begin{aligned} \mathtt{tr}_x(y) &\stackrel{\text{def}}{=} y + x \\ \mathtt{cb}_m(y) &\stackrel{\text{def}}{=} \max(y,m) \\ \mathtt{ct}_M(y) &\stackrel{\text{def}}{=} \min(y,M) \\ \mathtt{SA}_{[x,m,M]} &= \mathtt{ct}_M \circ \mathtt{cb}_m \circ \mathtt{tr}_x \end{aligned} \tag{10}$$

As shown in Figure 8, a composition of two SAs written in the form of Equation 10 leads to a new SA written in the same form. The calculation is the following sequence of commutation and merging of the "tr"s, "cb"s, and "ct"s:

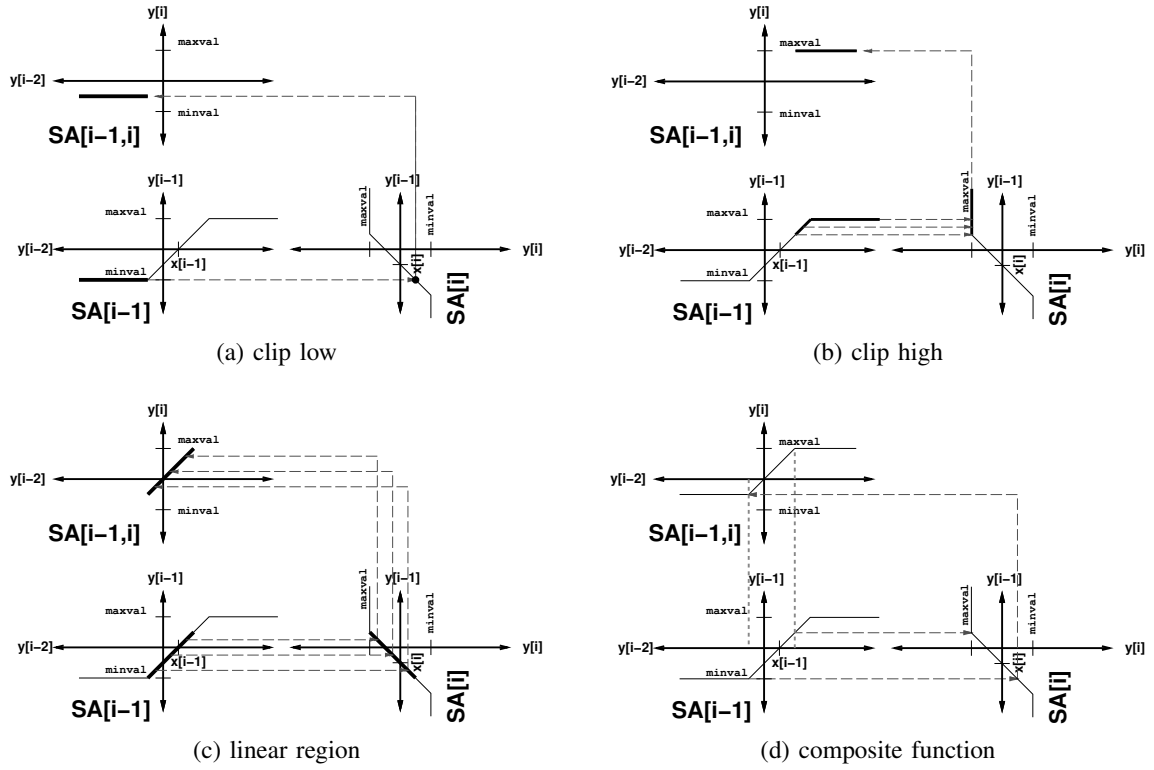
I. **Commutation of translation and clipping.** Clipping at $M1$ (or $m1$) and then translating by $x2$ is the same as first translating by $x2$ and then clipping at $M1{+}x2$ (or $m1{+}x2$).

II. **Commutation of upper and lower clipping.**

$$\mathtt{cb}_{m2} \circ \mathtt{ct}_{M1+x2} = \mathtt{ct}_{\max(M1+x2,m2)} \circ \mathtt{cb}_{m2}$$

This is seen by case analysis: first suppose $m2 \le M1 + x2$. Then both sides of the equation are the piecewise linear function

$$\begin{cases} M1+x2 & , \; y \ge M1+x2 \\ m2 & , \; y \le m2 \\ y & , \; \text{otherwise.} \end{cases} \tag{11}$$



(a) clip low (b) clip high

(c) linear region (d) composite function

N.b. axes are rotated for the $\mathtt{SA}[i]$ transform so that we can align the $y[i-1]$ output from the $\mathtt{SA}[i-1]$ transform with the $y[i-1]$ input to the $\mathtt{SA}[i]$ transform.

Fig. 7. Saturated Add Composition

$$
\begin{array}{lllll}
\mathtt{SA}_{[x2,m2,M2]} \circ \mathtt{SA}_{[x1,m1,M1]} & & & & \\
= & \mathtt{ct}_{M2} \circ \mathtt{cb}_{m2} \circ \mathtt{tr}_{x2} \circ \mathtt{ct}_{M1} & \circ \mathtt{cb}_{m1} & \circ \mathtt{tr}_{x1} \\
\overset{\text{I}}{=} & \mathtt{ct}_{M2} \circ \mathtt{cb}_{m2} & \circ \mathtt{ct}_{M1+x2} & \circ \mathtt{cb}_{m1+x2} & \circ \mathtt{tr}_{x1+x2} \\
\overset{\text{II}}{=} & \mathtt{ct}_{M2} & \circ \mathtt{ct}_{\max(M1+x2,m2)} & \circ \mathtt{cb}_{\max(m1+x2,m2)} & \circ \mathtt{tr}_{x1+x2} \\
\overset{\text{III}}{=} & & \mathtt{ct}_{\min(\max(M1+x2,m2),M2)} & \circ \mathtt{cb}_{\max(m1+x2,m2)} & \circ \mathtt{tr}_{x1+x2} \\
= & \mathtt{SA}_{[x1+x2,\max(m1+x2,m2),\min(\max(M1+x2,m2),M2)]} & & &
\end{array}
$$

Fig. 8. Operator Composition for Chained Saturated Additions

On the other hand, if $m2 > M1 + x2$, then both sides are the constant function $m2$.

III. **Merging of successive upper clipping.** This is associativity of min.

D. Applying the Composition Formula

At the first level of the computation, $m = \mathtt{minval}$ and $M = \mathtt{maxval}$. However, after each adjacent pair of saturating additions ($\mathtt{SA}[i-1]$, $\mathtt{SA}[i]$) has been replaced by a single saturating addition ($\mathtt{SA}[i-1,i]$), the remaining computation no longer has constant m and M. In general, therefore, a saturating accumulation specification includes a different $\mathtt{minval}$ and $\mathtt{maxval}$ for each input. We denote these values by $\mathtt{minval}[i]$ and $\mathtt{maxval}[i]$.

The $\mathtt{SA}$ to be performed on input number i is then:

$$
\begin{aligned}
&\mathtt{SA}[i](y) \qquad (12)\\
&\quad = \min(\max((y + x[i]), \mathtt{minval}[i]), \mathtt{maxval}[i])
\end{aligned}
$$

Composing two such functions and inlining, we get:

$$
\begin{aligned}
&\mathtt{SA}[i-1,i](y) = \mathtt{SA}[i](\mathtt{SA}[i-1](y)) \qquad (13)\\
&\quad = \min(\max((\min(\max((y + x[i-1]),\\
&\qquad\qquad \mathtt{minval}[i-1]),\\
&\qquad\quad \mathtt{maxval}[i-1])\\
&\qquad + x[i]),\\
&\quad \mathtt{minval}[i]),\\
&\mathtt{maxval}[i])
\end{aligned}
$$

We can transform this into:

$$
\begin{aligned}
&\mathtt{SA}[i-1,i](y) = \qquad (14)\\
&\quad = \min(\max((y + x[i-1] + x[i]),\\
&\qquad\quad \max((\mathtt{minval}[i-1] + x[i]),\\
&\qquad\qquad \mathtt{minval}[i])),\\
&\qquad \min(\max((\mathtt{maxval}[i-1] + x[i]),\\
&\qquad\qquad \mathtt{minval}[i]),\\
&\qquad\quad \mathtt{maxval}[i]))
\end{aligned}
$$

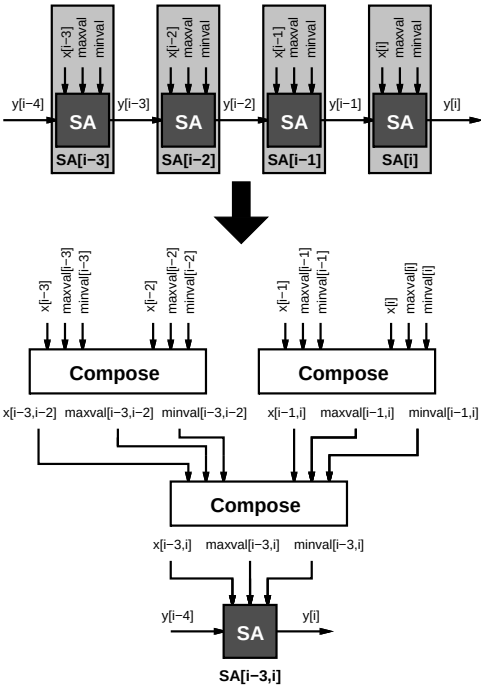


Fig. 9. Composition of $\texttt{SA}[(i-3), i]$

This is the same computation as Figure 8, as long as we let $M2 = \texttt{maxval}[i]$, $m2 = \texttt{minval}[i]$, $M1 = \texttt{maxval}[i-1]$, and $m2 = \texttt{minval}[i-1]$.

Now we define **Compose** as the six-input, three-output function which computes a description of $\texttt{SA}[i{-}1, i]$ given descriptions of $\texttt{SA}[i{-}1]$ and $\texttt{SA}[i]$:

$$x' = x[i{-}1] + x[i] \tag{15}$$

$$\texttt{minval}' = \max((\texttt{minval}[i{-}1] + x[i]), \texttt{minval}[i]) \tag{16}$$

$$\texttt{maxval}' = \min(\max((\texttt{maxval}[i{-}1] + x[i]), \texttt{minval}[i]), \texttt{maxval}[i]) \tag{17}$$

This gives us:

$$\texttt{SA}[i{-}1, i](y) = \min\left(\max\left(\left(y + x'\right), \texttt{minval}'\right), \texttt{maxval}'\right) \tag{18}$$

Note that this is exactly the same form as Equation 5, with the primed variables replacing the original input variables. This allows us to compute $\texttt{SA}[i, j](y)$ as shown in Figure 9. One can note this is a very similar strategy to the combination of "propagates" and "generates" in carry-lookahead addition (*e.g.*, [15], [16], [21]).

E. Wordsize of Intermediate Values

The preceding correctness arguments rely on the assumption that intermediate values (*i.e.*, all values ever computed by the Compose function) are mathematical integers; *i.e.*, they never overflow. For a computation of depth k, at most 2^k numbers are ever added, so intermediate values can be represented in $W{+}k$ bits if the inputs are represented in W bits. While this gives us an asymptotically tight result, we can do better from a practical point of view; we can actually do all computation with $W{+}2$ bits (2's complement representation) regardless of k.

First, notice that $\texttt{maxval}'$ is always between $\texttt{minval}[i]$ and $\texttt{maxval}[i]$. The same is not true about $\texttt{minval}'$, until we make a slight modification to Equation 16; we redefine $\texttt{minval}'$ as follows:

$$\texttt{minval}' = \min(\max((\texttt{minval}[i{-}1] + x[i]), \texttt{minval}[i]), \texttt{maxval}[i]) \tag{19}$$

This change does not affect the result because it only causes a decrease in $\texttt{minval}'$ when it is *greater* than $\texttt{maxval}'$. While it is more work to do the extra operation, it is only a constant increase, and this extra work is done anyway if the hardware for $\texttt{maxval}'$ is reused for $\texttt{minval}'$ (See Section IV). With this change, the interval $[\texttt{minval}', \texttt{maxval}']$ is contained in the interval $[\texttt{minval}[i], \texttt{maxval}[i]]$, so none of these maximum or minimum values ever requires more than W bits to represent.

F. Wordsize of Intermediate x'

In this section we show that we need only use a $(W{+}2)$-bit datapath to compute x' (Equation 15). Whenever x' overflows a $(W{+}2)$-bit datapath, its value is ignored, because a constant function is represented (*i.e.*, $\texttt{minval}' = \texttt{maxval}'$).

To bound all x' that occur for non-constant functions, we make one observation and one assumption:

1. (observation) There is one $(\texttt{minval}, \texttt{maxval})$ for all i such that:
$$\begin{aligned} \texttt{minval}[i] &\geq \texttt{minval} \text{ and} \\ \texttt{maxval}[i] &\leq \texttt{maxval}. \end{aligned} \tag{20}$$
This was demonstrated at the end of the previous section (Section III-E).
2. (assumption) For all original $x[i]$ (*i.e.*, the inputs), we have
$$|x[i]| \leq \Delta \stackrel{\text{def}}{=} \texttt{maxval} - \texttt{minval}$$
This is always true for the inputs when:
$$\texttt{minval} \leq x[i] \leq \texttt{maxval}$$
We use the broader interval 2Δ to deal with intermediate values of x'.

We now show, for any $x[i{-}k, i]$ in the multilevel computation, if $|x[i{-}k, i]| > 2\Delta$, then $\texttt{minval}[i{-}k, i] = \texttt{maxval}[i{-}k, i]$.

For a contradiction, assume that some $S \stackrel{\text{def}}{=} \texttt{SA}[i{-}k, i]$ is not a constant function when $|x_S| > 2\Delta$. Consider points y and y' such that $S(y) \neq S(y')$.

From the form of S, we know that it only takes on values in the interval $[\texttt{minval}_S, \texttt{maxval}_S]$. If $S(y)$ or $S(y')$ are endpoints of this non-empty interval, we can interpolate (extending to real numbers) and find new y, y', so that, without loss of generality, y and y' are both in the region of the domain of S where S has slope 1. Interpolation is a technicality only needed to handle the case where $\texttt{minval}_S + 1 = \texttt{maxval}$, such that there are not two, distinct integer values for y and y' which are in the slope 1 region.

Since S locally has slope 1 around y (and y'), the clipping feature in S must not be active around y. This means that y (and y') are in the interval $[\texttt{minval}_S - x_S, \texttt{maxval}_S - x_S]$,

which is contained in the interval $[\texttt{minval} - x_S, \texttt{maxval} - x_S]$ (observation 1).

Since $|x_S| > 2\Delta$, we deduce that y and y' are outside of the interval $[\texttt{minval} - \Delta, \texttt{maxval} + \Delta]$ since:

$$\texttt{maxval}_S - 2\Delta \leq \texttt{maxval} - 2\Delta = \texttt{minval} - \Delta$$

or

$$\begin{aligned} \texttt{minval}_S - (-2\Delta) &\geq \texttt{minval} - (-2\Delta) \\ &= \texttt{maxval} + \Delta \end{aligned}$$

By interpolation, we can always choose distinct y and y' so that they do not straddle this interval. Now consider what happens when the *first* input in the sequence $x_{i-k} \ldots x_i$ is applied to such a value. Using assumption 2, we see that $y + x[i-k]$ are to one side of the interval $[\texttt{minval}, \texttt{maxval}]$. Therefore $\texttt{SA}[i-k]$ must take y and y' to the same value, and therefore $\texttt{SA}[i-k, i]$ also has this property: *i.e.*, $S(y) = S(y')$, a contradiction. □

How many bits do we need to represent intermediate x'? If we assume the accumulator is a W-bit signed 2's complement value, then:

$$\begin{aligned} \texttt{maxval} &\leq 2^{(W-1)} - 1 \\ \texttt{minval} &\geq -2^{(W-1)} \end{aligned}$$

$$\Delta \leq \left(2^{(W-1)} - 1\right) - \left(-2^{(W-1)}\right) = 2^W - 1$$

We care about an x' only if $|x'| \leq 2\Delta < 2^{W+1} - 1$. Hence we can simply add the 'x's in $(W+2)$-bit 2's complement arithmetic (at all levels of the computation), and if there is an overflow then we do not care about the result.

The 2Δ and $(W+2)$-bit bounds are tight: the computation can really have representations of non-constant functions that use all $W + 2$ bits. For example, suppose $W = 8$, with $\texttt{minval} = -128$ and $\texttt{maxval} = 127$. Suppose $x_0 = x_1 = -254$. The function $\texttt{SA}[0, 1]$ is not constant, as $\texttt{SA}[0, 1](380) = -128$ while $\texttt{SA}[0, 1](381) = -127$, yet $x[0, 1] = -508$ requires 10 bits to represent. One might observe that in this case the function is in fact constant because the accumulator never starts at those values. However, this does not imply that $\texttt{minval} = \texttt{maxval}$, and while we could add extra hardware to make this the case, it would not be worth adding this hardware just in order to save one bit. Finally, restricting the inputs to a smaller bound than Δ is helpful only in small trees, as increments up to Δ can be achieved through a number of small increments.

IV. Putting it Together

Knowing how to compute $\texttt{SA}[i-1, i]$ from the parameters for $\texttt{SA}[i-1]$ and $\texttt{SA}[i]$, we can unroll the computation to match the delay through the saturated addition and create a suitable, asymmetric parallel-prefix computation (similar to Sections II-E through II-G). From the previous section, we know the core computation for the composer is, itself, an unsaturated addition (Equation 15) and two saturated additions (Equations 17 and 19). Using the base saturated adder shown in Figure 10, we build the composer as shown in Figure 11.

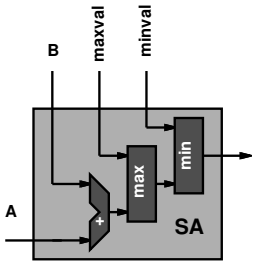


Fig. 10. Saturated Adder

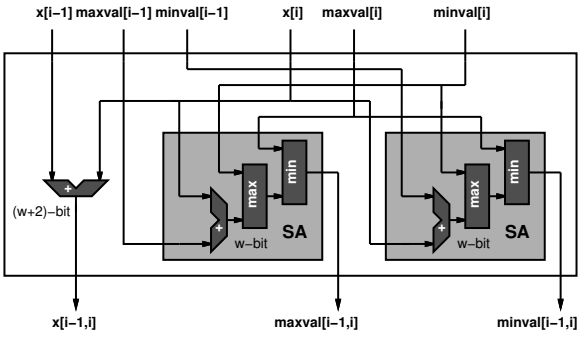


Fig. 11. Composition Unit for Two Saturated Additions

V. Implementation

A. Experiment

We implemented the parallel-prefix saturated accumulator in VHDL and targeted a Xilinx Spartan-3 XC3S-5000-4 FPGA to demonstrate functionality and obtain performance and area estimates. We used Modelsim 5.8 to verify the functionality of the design and Synplify Pro 7.7 and Xilinx ISE 6.1.02i to map our design onto the target device. We did not provide any area constraints and let the tools automatically place and route the design using just the timing constraints. The DCMs on the Spartan-3 (speed grade -4 part) support a maximum frequency of 280 MHz (3.57ns cycle), so we picked this maximum supported frequency as our performance target. We report area in Spartan-3 slices; each Spartan-3 slice contains two 4-input Lookup Tables with fast carry logic such that each slice can serve as two full adder bits.

B. Design Details

The parallel-prefix saturating accumulator consists of a parallel-prefix computation tree with an asymmetric feedback input (*cf.* Section II-F) sandwiched between a serializer and deserializer as shown in Figure 12. Consequently, we decompose the design into two clock domains. The higher frequency clock domain pushes data into the lower frequency domain of the parallel-prefix tree. The parallel-prefix tree runs at a proportionally slower rate to accommodate the saturating adders shown in Figures 10 and 11. Minimizing the delays in the tree requires us to compute each compose in two pipeline stages. Finally, we clock the result of the prefix computation into the higher frequency clock domain in parallel then serially shift out the data at the higher clock frequency.

As introduced in Section II-F, the delay through the composers is actually irrelevant to the correct operation of the saturated

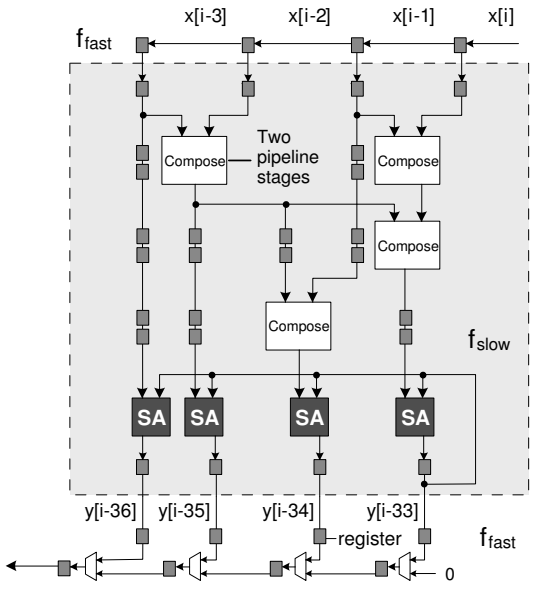


Fig. 12. $N = 4$ Parallel-Prefix Saturating Accumulator

TABLE II
MINIMUM SIZE OF PREFIX TREE REQUIRED TO ACHIEVE 280MHZ

Datapath Width (W)	2	4	8	16	32
Prefix-tree Width (N)	3	3	4	4	4

accumulation. The composition tree adds a uniform number of clock cycle delays between the $x[i]$ shift register and the final saturated accumulator. It does not add to the saturated accumulation feedback latency which the unrolling must cover. This is why we can safely pipeline compose stages in the parallel-prefix tree.

Data is transferred into the slower domain by serializing it in the faster domain and allowing the slower frequency domain to "capture" the signal synchronously on its clock edge. This, encapsulated dual frequency clocking scheme allows the rest of the system to have a consistent interface with this design.

C. Area

We express the area required by this design as a function of N (loop unroll factor) and W (bitwidth). The area required for the prefix tree is roughly $5\frac{2}{3}N$ times the area of a single saturated adder. The initial reduce tree has roughly N compose units, as does the final prefix tree. Each compose unit has two W-bit saturated adders and one $(W+2)$-bit regular adder. As noted, a Spartan-3 slice can support two full-adder bits, so each adder requires roughly $W/2$ slices. Similarly, the W-bit maximum and minimum multiplexers also each require $W/2$ slices. Together, this gives us $\approx 2 \times (2 \times 3 + 1) NW/2$ slices. Finally, we add a row of saturated adders to compute the final output to get a total of $\frac{17}{2}NW$ slices. Compared to the base saturated adder (*i.e.* Figure 10) which takes $\frac{3}{2}W$ slices, this is a factor of $\frac{17N}{3} = 5\frac{2}{3}N$.

Pipelining levels in the parallel-prefix tree roughly costs us $2 \times 3 \times N$ registers per level times the $2\log_2(N)$ levels for a total of $12N\log_2(N)W$ registers. The pair of registers for a pipe stage can fit in half a slice (*i.e.*, SRL16 configuration), so this should add no more than $6N\log_2(N)W$ slices.

$$A(N, W) \approx 6N\log_2(N)W + \frac{17}{2}NW \quad (21)$$

This approximation does not count the overhead of the control logic in the serializer and deserializer since it is small compared to the registers.

To pipeline down to the gate or Lookup Table level, we must unroll to cover the delay through the base saturated adder (Figure 10). This delay is one W-bit adder delay plus a small constant number of gate delays output multiplexing. If we use ripple carry adders, then we need an unroll factor, N, which is $O(W)$. Substituting $N = O(W)$ into Equation 21 and we get $O\left(W^2 \log(W)\right)$. If, instead, we use an efficient, log-depth adder, we substitute $N = \log(W)$ into Equation 21, and we see that area scales as:

$$A_{fullpipe-satadd}(W) = O\left(W \log(W) \log(\log(W))\right) \quad (22)$$

If the size of the tree is N and the frequency of the basic unpipelined saturating accumulator is f, then the system can run at a frequency $f \times N$. By increasing the size of the parallel-prefix tree, we can make the design run arbitrarily fast, up to the maximum attainable clock rate of the device. As Section VI-A notes, we can continue to exploit parallelism to run even faster if the application context provides and consumes more than one input and output per cycle. In Table II we show the value of N (*i.e.*, the size of the prefix tree) required to achieve a 3ns cycle target. We target this tighter cycle time (compared to the 3.57ns DCM limit) to reserve some headroom going into place and route for the larger designs. We observe that a value of $N = 4$ is adequate to make the design run as fast as the device can support.

TABLE III
ACCUMULATOR COMPARISON

Datapath Width (W)	2	4	8	16	32
Simple Saturated Accumulator					
Delay (ns)	6.2	8.1	9.1	11.3	13.4
Area (slices)	10	14	24	44	84
Parallel-Prefix Saturated Accumulator ($N = 4$)					
Delay (ns)	2.8	2.7	3.1	2.9	3.3
Area (slices)	215	333	571	1065	2085
Ratios: Parallel-Prefix/Simple					
Freq.	2.2	3.0	2.9	3.6	4.1
Area	22	24	24	24	25

D. Results

Table III shows the clock period achieved by all the designs for $N = 4$ after place and route. We beat the required 3.57ns performance limit for all the cases we considered. Since we only constrained the synthesis tools to optimize for a 3ns cycle, variations in cycle time around 3ns arise from imperfectly estimated physical routing delays. For $N = 4$, the latency from $x[i]$ to $y[i]$ is 38 fast clock cycles (See Figure 12) or, roughly 136ns at the 3.57ns clock period.

In Table III we show the actual area in terms of the Spartan-3 slices required to perform the mapping for different bitwidths

W. A 16-bit saturating accumulator requires 1065 slices which constitutes around 2% of the XC3S-5000. We also show that an area overhead of less than $25\times$ is required to achieve this speedup over the unpipelined base saturating accumulator (Figure 10); for $N = 4$, $5\frac{2}{3}N \approx 23$, so this is consistent with our intuitive prediction above.

Balzola's 5-input saturated adder [19] is equivalent to our $N = 4$ unrolling in that both can take in 5 inputs in a cycle. They compare their accelerated designs to a "serial" case that actually contains a simple combinational cascade of 4 saturated adders. Their fastest design uses 5.7 times the area of the 4 saturated adder cascade or $4\times5.7\approx23$ times the area of the base saturated adder. With this design, they achieve a speedup of 3.5 times the "serial" case. As a result their area overhead and throughput enhancement are quite similar to ours at $W = 16$ and $W = 32$ (See Table III). The Balzola design has a smaller increase in the latency from $x[i]$ to $y[i]$ than our design. Note that:

1) Our design spends roughly half of its area producing intermediate $y[i]$ outputs that the Balzola design does not produce; if we were to omit these intermediate outputs to functionally match the Balzola design, our area overhead would be half the reported size.
2) Our design has better asymptotic scaling, both in area ($O(N\log(N))$ vs. $O(N^2)$) and achievable delay (arbitrary versus constant factor speedup). Since these designs are already reaching parity in area overhead at this small N, this suggests our design will be smaller for $N > 4$.

A factor of 25 in area is a large cost to pay. However, the base saturated adder is tiny and usually only a small fraction of the area in a spatial design (*e.g.* Figure 1) or in a DSP (*i.e.*, memories often take up much more space than all the arithmetic processing logic combined, and the dedicated multipliers are much larger than the adders). Since the saturated addition is only a small fraction of the design area, the $25\times$ area expansion of this one unit may only increase the area of the overall design by a modest amount. When the saturated addition is the single bottleneck that prevents the entire system from running at high throughput, it will often makes sense to pay this cost.

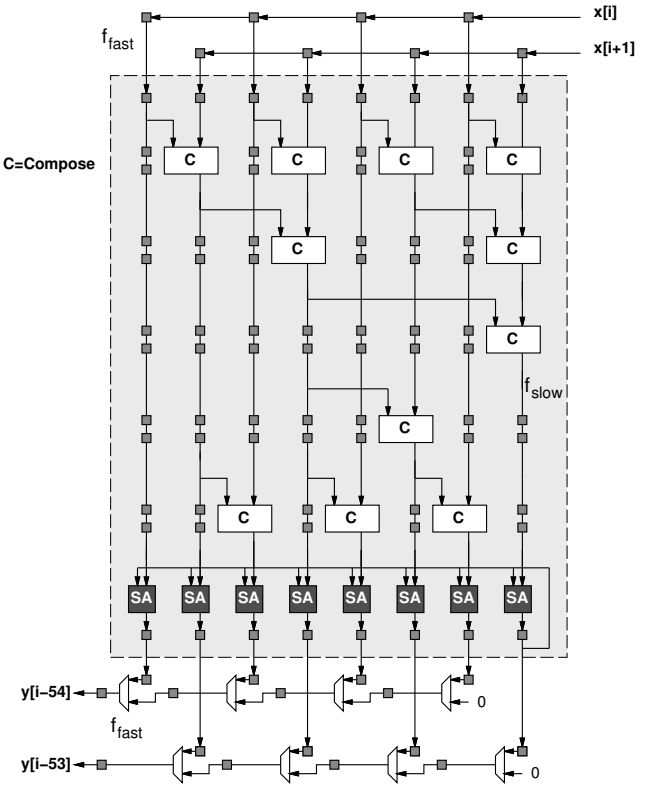


Fig. 13. $N = 8$ Parallel Prefix Saturating Accumulation with Two Inputs and Two Outputs per Cycle (f_{fast}=4$\times f_{slow}$)

VI. Generality and Open Questions

A. Beyond One Result per Clock

The clock period on the device is limited by the minimum overhead time on registers (setup, hold, clock jitter) and a minimum amount of logic between registers. For example, the design in Figure 12 has one mux between registers on the fast clock domain. In CMOS, 6–8 Fanout-4 inverter delays is considered a common lower bound on the clock period (*e.g.*, [22]).

Nonetheless, since the core saturated addition operations can now be performed in parallel, we can achieve throughputs that exceed the clock-cycle bound if it is possible to bring in and produce multiple values in parallel. Figure 13 shows the generalization on Figure 12 where $N = 8$ and the design consumes/produces two values per cycle on the fast clock. As with the Figure 12 design, the slow clock has a period 4 times the fast clock.

B. Beyond Accumulation

The techniques used here are actually quite general. Functional composition is associative, so we can always unroll the loop, associate each loop stage with its inputs, and perform a parallel-prefix reduction on the loop instances. This composition is applicable even if there are a series of different operations in each loop body or even different operations between loop instances. For this to be useful, however, the composed function of multiple loop instances must have shallower depth than the original, serial path through the set of loop instances. Also, it must be inexpensive to compute the composed function; in this case, computing the arguments to the composed function was, asymptotically, the same complexity as computing the function (*cf.* Equations 15, 17, and 19 to Equations 5 and 6).

In formulating the associativity of saturated accumulation, we worked with the composition of the functions $\max$, $\min$, and addition where one input to each function was early bound—*i.e.*, bound outside of the loop. As a result of the early-bound inputs, the computation was a single chain of dependent computations, and we showed how to use parallel prefix to compute the outputs of the chain with low latency. Multiplication with one early-bound input can be added to this group. More generally, we can compute efficient functional compositions on any, potentially heterogeneous, chain composed from this extended function group.

We can also perform this prefix optimization on this function group even if intermediate results are forwarded to multiple functions. With multiple use of intermediates, we do not strictly have a single chain but rather a tree. In these cases, we can still extract the chain producing each output and perform a parallel prefix on each chain. This may require that we duplicate the chain prefixes which feed into multiple tree branches.

An important open question for future research is to generally characterize the class of functions that have this kind of light-

weight composition. That is, more generally, for a given composition of functions:

- How expensive is it to compute the composed function?
- How much shorter is the path through the composed function than the sum of the paths through the original functions?

These associative transformation can be powerful options for exploiting area-time tradeoffs. High-level design automation tools can exploit them for optimizing performance. With a sufficiently broad set, it may be possible to integrate these into a superscalar processor design, allowing the processor to issue a set of dependent instructions and reduce them associatively.

VII. Summary

Saturated accumulation has a loop dependency that, naively, limits single-stream throughput and our ability to fully exploit the computational capacity of modern integrated circuits, particularly as clock rate scaling slows and future performance improvements depend more on exploiting the increased area capacity to improve throughput. We show that this loop dependence is actually avoidable by reformulating the saturated addition as the composition of a series of functions. We further show that this particular function composition is, asymptotically, no more complex than the original saturated addition operation. Function composition is associative, so this reformulation allows us to build a parallel-prefix tree in order to compute the saturated accumulation over several loop iterations in parallel. Consequently, we can unroll the saturated accumulation loop to cover the delay through the saturated adder. As a result, we show how to compute saturated accumulation at any data rate supported by the device's clocking and I/O.

Acknowledgment

This research was funded in part by the NSF under grant CCR-0205471. Stephanie Chan was supported by the Marcella Bonsall SURF Fellowship. Karl Papadantonakis was supported by a Moore Fellowship. Scott Weber and Eylon Caspi developed early FPGA implementations of ADPCM which helped identify this challenge. Michael Wrighton provided VHDL coding and CAD tool usage tips.

References

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures," in *Proceedings of the International Symposium on Computer Architecture*, 2000, pp. 248–259.

[2] D. Chinnery and K. Keutzer, *Closing the Gap between ASIC & Custom: Tools and Techniques for High-Performance ASIC Design*. Kluwer Academic Publishers, 2002.

[3] W. Tsu, K. Macy, A. Joshi, R. Huang, N. Walker, T. Tung, O. Rowhani, V. George, J. Wawrzynek, and A. DeHon, "HSRA: High-Speed, Hierarchical Synchronous Reconfigurable Array," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 1999, pp. 125–134.

[4] D. P. Singh and S. D. Brown, "The Case for Registered Routing Switches in Field Programmable Gate Arrays," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, February 2001, pp. 161–169.

[5] C. Leiserson, F. Rose, and J. Saxe, "Optimizing Synchronous Circuitry by Retiming," in *Third Caltech Conference On VLSI*, March 1983.

[6] N. Weaver, Y. Markovskiy, Y. Patel, and J. Wawrzynek, "Post-Placement C-slow Retiming for the Xilinx Virtex FPGA," in *Proceedings of the International Symposium on Field-Programmable Gate Arrays*, 2003, pp. 185–194.

[7] B. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System," in *Proceedings fo the Symposium on Real-Time Signal Processing*, 1981, pp. 241–248.

[8] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm, "Exploiting choice: Instruction fetch and issue on an implementable simultatneous multithreading processor," in *Proceedings of the International Symposium on Computer Architecture*, 1996, pp. 191–202.

[9] Z. Luo and M. Martonosi, "Accelerating Pipelined Integer and Floating-Point Accumulations in Configurable Hardware with Delayed Addition Techniques," *IEEE Transactions on Computers*, vol. 49, no. 3, pp. 208–218, March 2000.

[10] *Xilinx Spartan-3 FPGA Family Data Sheet*, Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124, December 2004, dS099 <http://direct.xilinx.com/bvdocs/publications/ds099.pdf>.

[11] K. Papadantonakis, N. Kapre, S. Chan, and A. DeHon, "Pipelining Saturated Accumulation," in *Proceedings of the International Conference on Field-Programmable Technology*. IEEE, December 2005, pp. 19–26.

[12] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems," in *International Symposium on Microarchitecture*, 1997, pp. 330–335.

[13] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Maps: A Compiler-Managed Memory System for Raw Machines," in *Proceedings of the International Symposium on Computer Architecture*, 1999.

[14] W. D. Hillis and G. L. Steele, "Data Parallel Algorithms," *Communications of the ACM*, vol. 29, no. 12, pp. 1170–1183, December 1986.

[15] R. P. Brent and H. T. Kung, "A Regular Layout for Parallel Adders," *IEEE Transactions on Computers*, vol. 31, no. 3, pp. 260–264, March 1982.

[16] F. T. Leighton, *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*. Morgan Kaufmann Publishers, Inc., 1992.

[17] B. D. de Dinechin, C. Monat, and F. Rastello, "Parallel Execution of the Saturated Reductions," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, 2001, pp. 373–384.

[18] M. Schulte, P. Balzola, J. Ruan, and J. Glossner, "Parallel Saturating Multioperand Adders," in *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2000, pp. 172–179.

[19] P. I. Balzola, M. J. Schulte, J. Ruan, J. Glossner, and E. Hokenek, "Design Alternatives for Parallel Saturating Multioperand Adders," in *Proceedings of the International Conference on Computer Design*, September 2001, pp. 172–177.

[20] J. H. Hubbard and B. B. H. Hubbard, *Vector Calculus, Linear Algebra, and Differential Forms: A Unified Approach*. Prentice Hall, 1999.

[21] S. Winograd, "On the Time Required to Perform Addition," *Journal of the ACM*, vol. 12, no. 2, pp. 277–285, April 1965.

[22] M. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar, "The Optimal Logic Depth Per Pipeline Stage is 6 to 8 FO4 Inverter Delays," in *Proceedings of the International Symposium on Computer Architecture*, 2002, pp. 14–24.