Appearing in *IEEE International Conference on Field-Programmable Technology* (FPT 2005), December 11–14, 2005

Pipelining Saturated Accumulation

Karl Papadantonakis, Nachiket Kapre, Stephanie Chan, and André DeHon
Department of Computer Science
California Institute of Technology
Pasadena, CA 91125
kp@caltech.edu

Abstract

Aggressive pipelining allows FPGAs to achieve high throughput on many Digital Signal Processing applications. However, cyclic data dependencies in the computation can limit pipelining and reduce the efficiency and speed of an FPGA 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), the maximum frequency supported by the component's DCM.

1. Introduction

FPGAs have high computational density (*e.g.* they offer a large number of bit operations per unit space-time) when they can be run at high throughput (*e.g.* [1]). To achieve this high density, we must aggressively pipeline designs exploiting the large number of registers in FPGA architectures. In the extreme, we pipeline designs so that only a single LookUp-Table (LUT) delay and local interconnect is in the latency path between registers (*e.g.* [2]). Pipelined at this level, conventional FPGAs should be able to run with clock rates in the hundreds of megahertz.

For acyclic designs (feed forward dataflow), it is always possible to perform this pipelining. 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 which has 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 LUTs in the path) and the total registers in the cycle. This often prevents us from pipelining the design all the way down to the single LUT plus local interconnect level and consequently prevents us from operating at peak throughput to use the device efficiently. We can use the device efficiently by interleaving parallel problems in C-slow fashion (*e.g.* [5, 6]), 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 computational density.

Saturated accumulation (Section 2.1) 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 (Section 2.2). While non-saturated accumulation is amenable to associative transformations (*e.g.* delayed addition [7] or block associative reduce trees (Section 2.4)), 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 3). Once transformed, we use a parallel-prefix computation to avoid the apparent cyclic dependencies in the original operation (Section 2.5). As a concrete demonstration of this technique, we show how to accelerate a 16-bit accumulation on a Xilinx Spartan-3 (X3CS-5000-4) [8] from a cycle time of 11.3ns to a cycle time below 3.57ns (Section 5). The techniques introduced here are general and allow us to pipeline saturated accumulations to any throughput which the device can support.

2. Background

2.1. 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

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

Table 1. Accumulation Example

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:

```
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:

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

Table 1 gives an example showing the difference between modulo and saturated accumulation.

2.2. Example: ADPCM

The decoder in the Adaptive Differential Pulse-Compression Modulation (ADPCM) application in the `mediabench` benchmark suite [9] 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.* [10]). 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.

2.3. 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.

2.4. Associative Reduce

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

$$y_i = y_{i-1} + x_i \qquad (2)$$

Unrolling the accumulation sum, we can write:

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

Exploiting associativity we can rewrite this as:

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

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

With this reassociation, the delay of the addition tree grows as $\log(N)$ while the number of clock sample cycles grows as N. The unrolled cycle allows us to add registers to the cycle faster (N) than we add delays ($\log(N)$). Consequently, we can select N sufficiently large to allow arbitrary retiming of the accumulation.

2.5. Parallel-Prefix Tree

In Section 2.4, we noted we could compute the final sum of N values in $O(\log(N))$ time using $O(N)$

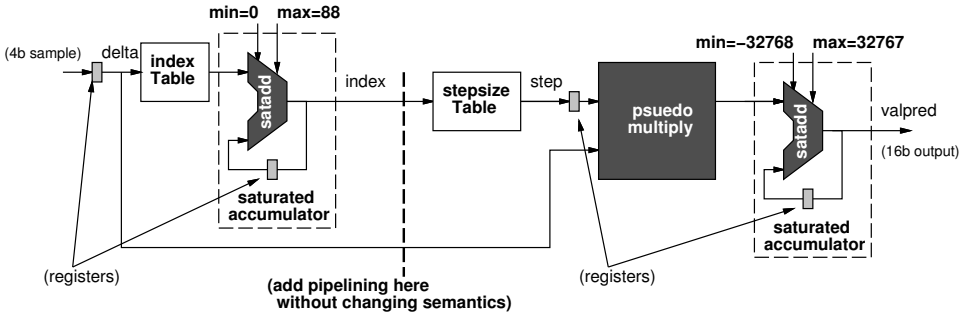


Figure 1. Dataflow for ADPCM Decode

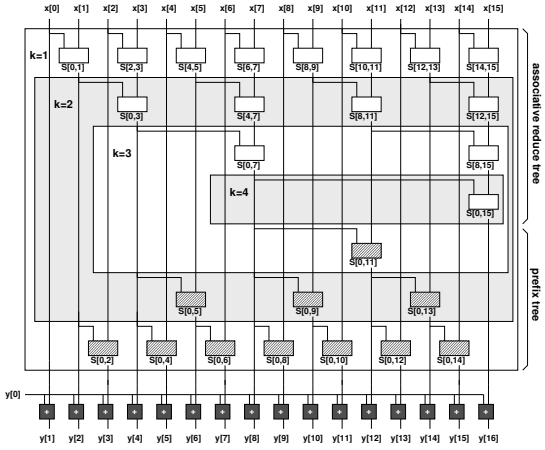


Figure 2. 16-input Parallel-Prefix Tree

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.* [11]).

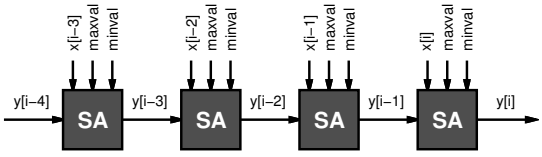
We do this by computing and combining partial sums of the form $\mathtt{S}[s,t]$ which represents the sum: $x_s + x_{s+1} + \ldots x_t$. When we build the associative reduce tree, at each level k, we are combining $\mathtt{S}[(2j)\,2^k, (2j+1)\,2^k{-}1]$ and $\mathtt{S}[(2j+1)\,2^k, 2\,(j+1)\,2^k{-}1]$ to compute $\mathtt{S}[(2j)\,2^k, 2\,(j+1)\,2^k{-}1]$ (See Figure 2). 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 $\mathtt{S}[0, 2^k{-}1]$ prefixes with the $\mathtt{S}[2^{k_0}, 2^{k_0}{+}2^{k_1}{-}1]$ spans ($k_1 < k_0$) to compute the intermediate results. To compute the full prefix sequence ($\mathtt{S}[0,1], \mathtt{S}[0,2], \ldots \mathtt{S}[0,N{-}1]$), 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 2 shows a width 16 parallel-prefix tree for saturated accumulation. For a more tutorial development of parallel-prefix computations see [11, 12].

2.6. Prior Work

Balzola *et al.* attacked the problem of saturating accumulation at the bit level [13]. They observed they could reduce the logic in the critical cycle by computing partial sums for the possible saturation cases and using a fast, bit-level multiplexing network to rapidly select and compose the correct final sums. They were able to reduce the cycle so it only contained a single carry-propagate adder and some bit-level multiplexing. For custom designs, this minimal cycle may be sufficiently small to provide the desired throughput. In contrast, our solution makes the saturating operations associative. Our solution may be more important for FPGA designs where the designer has less freedom to implement a fast adder and must pay for programmable interconnect delays for the bit-level control.

3. Associative Reformulation of Saturated Accumulation

Unrolling the computation we need to perform for saturated additions, we get a chain of saturated additions (SA), such as:



We can express SA (Section 2.1) as a function using max and min:

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

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 [14] so we can compute saturated accumulation using a parallel-prefix tree (Section 2.5)

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:

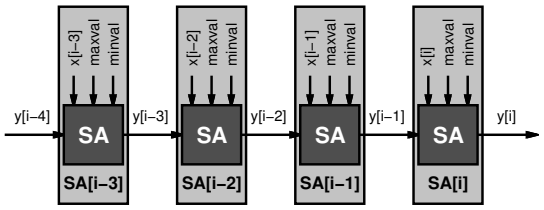
$$\mathtt{SA}_{[x,m,M]}(y) \stackrel{\text{def}}{=} \mathtt{SA}(y,x,m,M) \tag{6}$$

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:

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



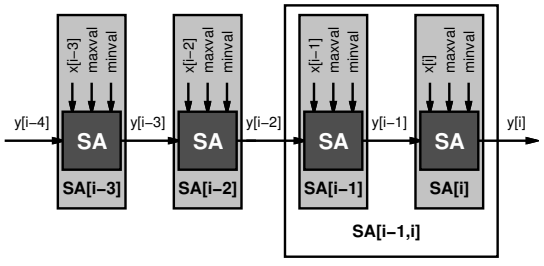
3.1. Composing the SA functions

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:

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

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

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



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",

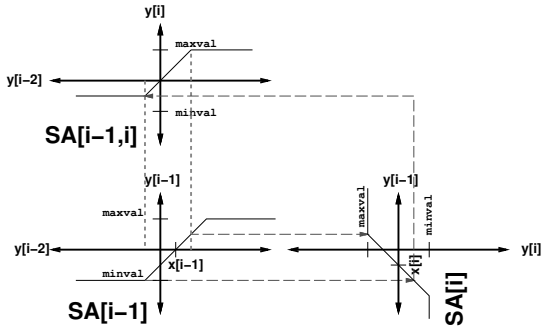
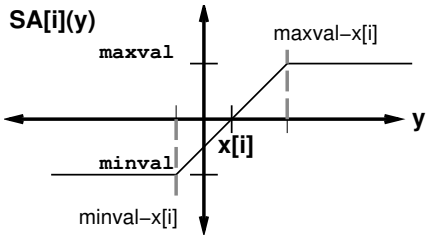


Figure 3. Saturated Add Composition

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 $\mathtt{SA}[i]$:



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 3. Any input below minval or above maxval 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 (See Figure 3). 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 so results in a slope one region. 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.

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

Figure 4. Operator Composition for Chained Saturated Additions

3.2. Composition Formula

We have just proved that the form $\texttt{SA}_{[x,m,M]}$ 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 $\texttt{SA}_{[x1,m1,M1]}$ and $\texttt{SA}_{[x2,m2,M2]}$.

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 $\texttt{tr}_x$, $\texttt{cb}_m$, and $\texttt{ct}_M$, respectively:

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

As shown in Figure 4, 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.**

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

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

$$
\begin{cases}
M1 + x2 & , \; y \geq M1 + x2 \\
m2 & , \; y \leq m2 \\
y & , \; \text{otherwise.}
\end{cases} \qquad (11)
$$

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.

Alternately, this can also be computed directly from the composed function.

3.3. Applying the Composition Formula

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

The SA to be performed on input number i is then:

$$
\begin{array}{rcll}
\texttt{SA}[i](y) & & & (12) \\
= & \min(\max((y + x[i]), \texttt{minval}[i]), \texttt{maxval}[i]) & &
\end{array}
$$

Composing two such functions and inlining, we get:

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

We can transform this into:

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

This is the same thing as Figure 4, 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 $\texttt{SA}[i]$:

$$
\begin{array}{rcll}
x' & = & x[i-1] + x[i] & (15) \\
\texttt{minval}' & = & \max((\texttt{minval}[i-1] + x[i]), & (16) \\
& & \qquad \texttt{minval}[i]) &
\end{array}
$$

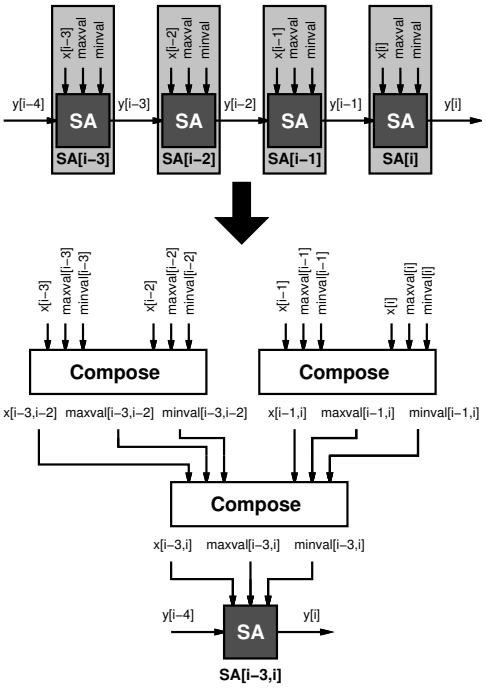


Figure 5. Composition of $\mathtt{SA}[(i-3), i]$

$$\begin{aligned}\mathtt{maxval}' &= \min(\max((\mathtt{maxval}[i{-}1] + x[i]),\\ &\qquad \mathtt{minval}[i]),\\ &\qquad \mathtt{maxval}[i])\end{aligned} \tag{17}$$

This gives us:

$$\begin{aligned}&\mathtt{SA}[i{-}1, i](y)\\ &= \min(\max(y + x'), \mathtt{minval}'), \mathtt{maxval}')\end{aligned} \tag{18}$$

This allows us to compute $\mathtt{SA}[i, j](y)$ as shown in Figure 5. One can note this is a very similar strategy to the combination of "propagates" and "generates" in carry-lookahead addition (*e.g.* [12]).

3.4. 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 actually do all computation with $W{+}2$ bits (2's complement representation) regardless of k.

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

$$\begin{aligned}\mathtt{minval}' &= \min(\max((\mathtt{minval}[i{-}1] + x[i]),\\ &\qquad \mathtt{minval}[i]),\\ &\qquad \mathtt{maxval}[i])\end{aligned} \tag{19}$$

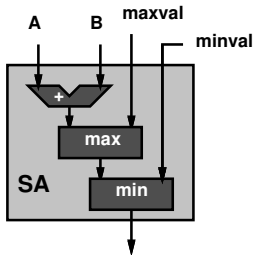


Figure 6. Saturated Adder

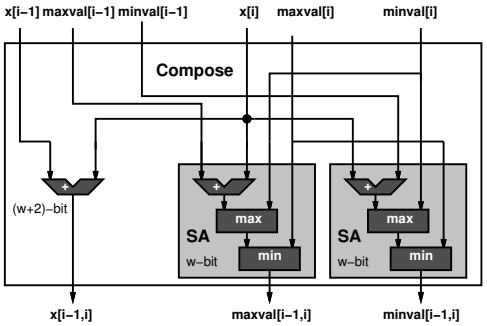


Figure 7. Composition Unit for Two Saturated Additions

This change does not affect the result because it only causes a decrease in $\mathtt{minval}'$ when it is *greater* than $\mathtt{maxval}'$. 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 $\mathtt{maxval}'$ is reused for $\mathtt{minval}'$ (See Section 4). With this change, the interval $[\mathtt{minval}', \mathtt{maxval}']$ is contained in the interval $[\mathtt{minval}[i], \mathtt{maxval}[i]]$, so none of these quantities ever requires more than W bits to represent.

If we use $(W{+}2)$-bit datapaths for computing x', x' can overflow in the tree, as the "x"s are never clipped. We argue that this does not matter. We can show that whenever x' overflows, its value is ignored, because a constant function is represented (*i.e.* $\mathtt{minval}' = \mathtt{maxval}'$). Furthermore, we need not keep track of when an overflow has occured, since if $\mathtt{minval} = \mathtt{maxval}$, then $\mathtt{minval}' = \mathtt{maxval}'$ at all subsequent levels of the computation, as this property is maintained by Equations 17 and 19.

4. Putting it Together

Knowing how to compute $\mathtt{SA}[i, i{-}1]$ from the parameters for $\mathtt{SA}[i]$ and $\mathtt{SA}[i{-}1]$, we can unroll the computation to match the delay through the saturated addition and create a suitable parallel-prefix computation (similar to Sections 2.4 and 2.5). From the previous section, we know the core computation for the composer is, itself, saturated addition (Eqs. 15, 17, and 19). Using the saturated adder shown in Figure 6, we build the composer as shown in Figure 7.

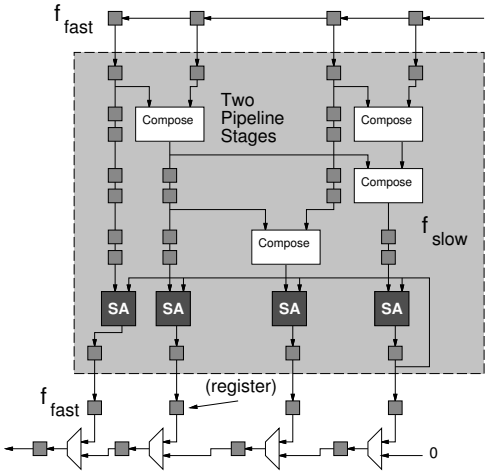


Figure 8. $N = 4$ Parallel-Prefix Saturating Accumulator

5. Implementation

We implemented the parallel-prefix saturated accumulator in VHDL to demonstrate functionality and get 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. We chose a Spartan-3 XC3S-5000-4 as our target device. 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.

Design Details The parallel-prefix saturating accumulator consists of a parallel-prefix computation tree sandwiched between a serializer and deserializer as shown in Figure 8. Consequently, we decompose the design into two clock domains. The higher frequency clock domain pushes data into the slower frequency domain of the parallel-prefix tree. The parallel-prefix tree runs at a proportionally slower rate to accomodate the saturating adders shown in Figures 6 and 7. 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.

It is worthwhile to note that the delay through the composers is actually irrelevant to the correct operation of the saturated 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.

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

Table 2. Minimum Size of Prefix Tree Required to Achieve 280MHz

Area We express the area required by this design as a function of N (loop unroll factor) and W (bitwidth). Intuitively, we can quickly see that 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, and each adder requires roughly $W/2$ 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 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 a single SRL16, so this should add no more than $3N\log_2(N)W$ slices.

$$A(N,W) \approx 3N\log_2(N)W + \frac{17}{2}NW \qquad (20)$$

This approximation does not count the overhead of the control logic in the serializer and deserializer since it is small compared to the registers. For ripple carry adders, $N = O(W)$ and this says area will scale as $O\left(W^2 \log(W)\right)$. If we use efficient, log-depth adders, $N = O(\log(W))$ and area scales as $O(W \log(W) \log(\log(W)))$.

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 speed of the device. In Table 2 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.

Results Table 3 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. In Table 3 we show the actual area in 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

Datapath Width (W)	2	4	8	16	32
Simple Saturated Accumulator					
Delay (ns)	6.2	8.1	9.1	11.3	13.4
SLICEs	10	14	24	44	84
Parallel-Prefix Saturated Accumulator ($N = 4$)					
Delay (ns)	2.8	2.7	3.1	2.9	3.3
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

Table 3. Accumulator Comparison

speedup over an unpipelined simple saturating accumulator; for $N = 4$, $5\frac{2}{3}N \approx 23$, so this is consistent with our intuitive prediction aboves.

6. Summary

Saturated accumulation has a loop dependency that, naively, limits single-stream throughput and our ability to fully exploit the computational capacity of modern FPGAs. 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 an FPGA.

Acknowledgments

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.

7. References

[1] A. DeHon, "The Density Advantage of Configurable Computing," *IEEE Computer*, vol. 33, no. 4, pp. 41–49, April 2000.

[2] B. V. Herzen, "Signal Processing at 250 MHz using High-Performance FPGA's," in *FPGA*, February 1997, pp. 62–68.

[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 *FPGA*, 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 *FPGA*, 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 *FPGA*, 2003, pp. 185–194.

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

[8] *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>.

[9] 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.

[10] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal, "Maps: A Compiler-Managed Memory System for Raw Machines," in *ISCA*, 1999.

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

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

[13] 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.

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

A Transforming Composed Functions

Here we show the detailed steps involved in transforming from Equation 13 to Equation 14.

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

I. Commutation of translation and clipping

We can push $+x[i]$ into the min, using:

$$\min(a, b) + c = \min((a + c), (b + c)) \tag{21}$$

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

We can push $+x[i]$ into max, using:

$$\max(a, b) + c = \max((a + c), (b + c)) \tag{22}$$

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

II. Commutation of upper and lower clipping

Let:

$$a = \max((y + x[i{-}1] + x[i]), \texttt{minval}[i{-}1] + x[i]) \tag{23}$$

$$b = \texttt{maxval}[i{-}1] + x[i] \tag{24}$$

$$c = \texttt{minval}[i] \tag{25}$$

So, we have:

$$\begin{aligned}
&\texttt{SA}[i{-}1,i](y) \\
&\quad = \min(\max(\min(a, b), c), \texttt{maxval}[i])
\end{aligned} \tag{26}$$

Using the identity (See Max-inside-Min Lemma in Appendix A.1):

$$\max(\min(a, b), c) = \min(\max(a, c), \max(b, c)) \tag{27}$$

This can become:

$$\begin{aligned}
&\texttt{SA}[i{-}1,i](y) \\
&\quad = \min(\min(\max(a, c), \max(b, c), \texttt{maxval}[i])
\end{aligned} \tag{28}$$

Substituting back in the expressions a, b, c, we get:

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

Using the associativity of max:

$$\max(\max(a, b), c) = \max(a, \max(b, c)) \tag{29}$$

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

III. Merging of successive upper clipping

Using the associativity of min:

$$\min(\min(a, b), c) = \min(a, \min(b, c)) \tag{30}$$

This finally gives us:

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

A.1 Max-inside-Min Lemma

Lemma: Identity 27 is true for all a, b, c.
That is, the following is an identity relation:

$$\max(\min(a, b), c) = \min(\max(a, c), \max(b, c))$$

Proof:
Assume $a > b$:
We can immediately reduce the left-hand-side of the identity to:

$$\max(\min(a, b), c) = \max(b, c) \tag{31}$$

We now turn to the right-hand-side of the identity: If $c < a$:

$$\max(a, c) < \max(b, c) \tag{32}$$

If $c \geq a$:

$$\max(a, c) = \max(b, c) \tag{33}$$

So, for all c:

$$\max(a, c) \geq \max(b, c) \tag{34}$$

From this we simplify:

$$\min(\max(a, c), \max(b, c)) = \max(b, c) \tag{35}$$

We see from Eq. 31 and 35 that both sides of the claimed Identify 27 are equivalent under the assumption $a > b$. Note that the roles of a and b are symmetric. So if $b < a$, we have an analogous case, so the identify will also hold.

If $a = b$, Eq. 31 is unchanged, and Eq. 34 still holds, so Eq. 35 must still hold. So, the Identify also holds when $a = b$.

Therefore, we see the Identity must hold for all a, b, c. $\diamond$

B Wordsize of Intermediate x'

In this appendix we show that we need only use a $(W+2)$-bit datapath to compute x' (Equation 15). As suggested in Section 3.4, 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

$$\begin{array}{rcl} \texttt{minval}[i] & \geq & \texttt{minval} \text{ and} \\ \texttt{maxval}[i] & \leq & \texttt{maxval}. \end{array} \tag{36}$$

   This was demonstrated at the end of Section 3.4.

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 generallity, y and y' are both in the region of the domain of S where S has slope 1. Interpolation is a technicallity 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

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

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. $\square$

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:

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

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

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 don't 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.