AN ASYNCHRONOUS IMPLEMENTATION OF THE MAXLIST ALGORITHM

Chris J. Myers *Hao Zheng* *

Electrical Engineering Department
University of Utah Salt Lake City, UT 84112
{myers,hao}@ee.utah.edu

ABSTRACT

We present an efficient asynchronous VLSI architecture for calculating running maximum or minimum values over a sliding window. Running maximums or minimums are very useful for many signal and image processing tasks. Our architecture performs the calculation using the MAXLIST algorithm. In order to take advantage of the wide delay variations due to data-dependencies and operating conditions, an asynchronous approach is taken to achieve higher performance and lower power. Simulation results demonstrate that our asynchronous architecture is significantly faster than existing and potential synchronous architectures.

1. INTRODUCTION

Many signal and image processing algorithms require the calculation of a running maximum or minimum over a sliding data window. For example, in a normalized least-mean-square (NLMS) adaptation algorithm given in [1], the filter coefficient which is chosen to be modified is the one which is associated with the input sample with the largest absolute value in the window of samples currently in the filter.

In [2], an efficient algorithm is presented for such calculations. This algorithm stores data elements in a pruned list. The data elements which are stored are those which are currently or have the potential of becoming the maximum or minimum within the sliding data window. This pruned list can be substantially smaller than the actual size of the sliding window.

In this paper, we present an asynchronous architecture to implement the MAXLIST algorithm. We have designed and simulated it in VHDL on a large set of correlated random data samples. Our results show a wide variation in delay due to both data-dependencies and operating conditions. We compare our asynchronous design with an existing synchronous design and the best possible synchronous design with an architecture comparable to ours.

2. ALGORITHM

The MAXLIST algorithm generates the pruned list of potential maxima (or minima) as follows. When a new element arrives, it firsts checks to see if an element already on the list has fallen out of the sliding window. If it has, it is removed from the list. Next, it searches the list until it finds the smallest element which is larger than the new element. It adds the new element after this one, and it removes all smaller elements since they will never become the maximum across the window.

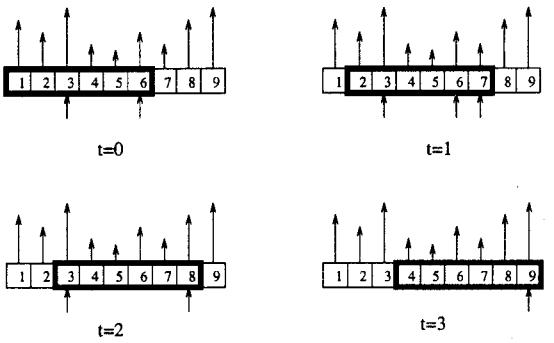

Figure 1. Example of the MAXLIST algorithm.

An example (courtesy of [2]) is shown in Figure 1. In this example, the sliding window is 6 elements long. Initially, element 3 is in the list because it is the maximum, and element 6 is in the list because it is a potential maximum. Elements 1 and 2 are dominated by element 3 since it is larger and appears later in the list. Elements 4 and 5 are dominated by element 6. At time 1, the window shifts, and element 7 is added to the list. At time 2, the window shifts again, element 8 is added, and since it dominates 6 and 7, they are removed from the list. At time 3, element 3 slides out of the window, and element 9 is added, dominating element 8.

By construction, the elements in the list are ordered by size and age. The head of the list is always the maximum and always the oldest element. The remaining elements have the potential to become a maximum as larger, older elements fall out of the sliding window.

In hardware, the pruned list must be of fixed size. If this size is less than the window size, it is possible that the running maximum or minimum may be in error. In [2], it is shown that the average size of the pruned list for random data goes like $ln(n)$ where n is the size of the window. Since small errors can usually be tolerated in signal and image processing algorithms, the list size is usually chosen to be slightly larger than $ln(n)$.

*This research is supported by a grant from Intel Corporation and NSF CAREER award MIP-9625014.

0-8186-7919-0/97 $10.00

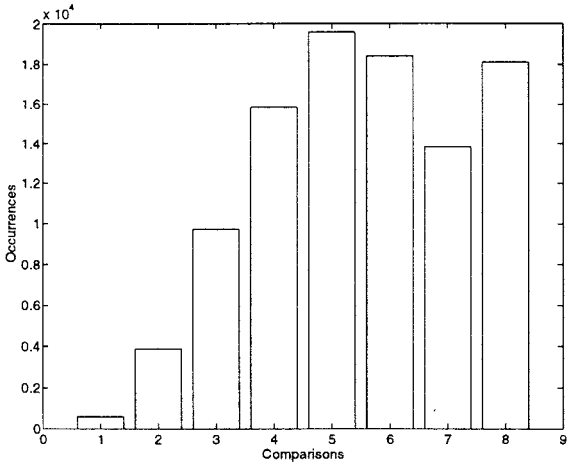

Figure 2. Distribution of forward comparisons.

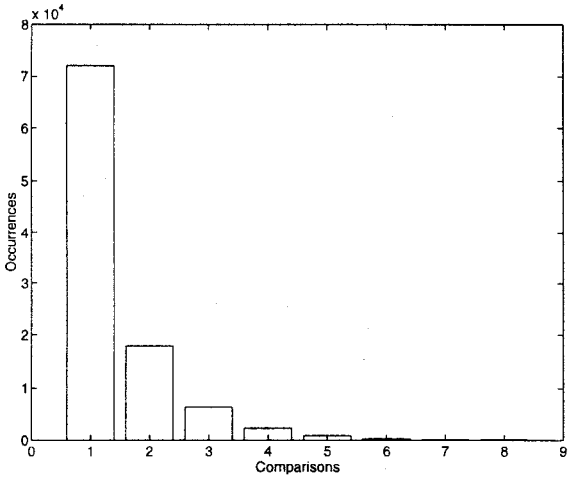

Figure 3. Distribution of backward comparisons.

3. ARCHITECTURE

In our asynchronous architecture, we have chosen to compute the maximum over a sliding window of 256 elements with a list size of 8 elements, where each element is represented as an 8-bit value. It is relatively straightforward to adapt our architecture to minimum calculations and to different size windows and lists.

One important architecture decision is how to search the list to find the location where a new element should be inserted. Our initial architecture began the search at the beginning of the list (i.e., the current maximum element) and worked towards the end. It was brought to our attention that this may result in more comparisons than necessary [3]. As shown in Figures 2 and 3, by starting the search at the end of the list (i.e., the smallest potential maximum or newest element) and searching backwards, the average number of comparisons is reduced from 5.5 to only 1.4.

Our architecture, depicted in Figure 4, is composed of seven main parts: an input latch, a counter, a FIFO, two comparators, an output latch, and a controller. In each data cycle, the following events occur:

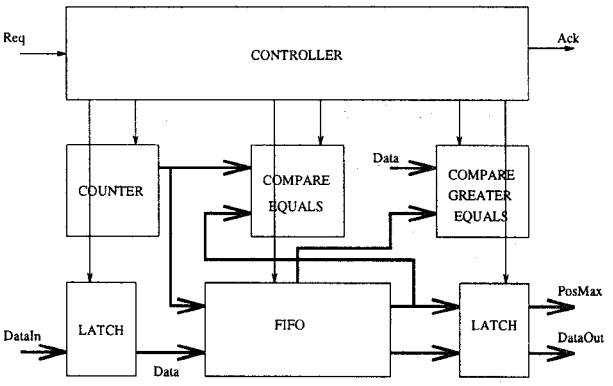

Figure 4. Overall block diagram.

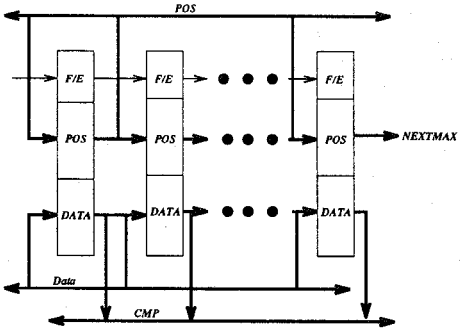

Figure 5. Block diagram of the FIFO.

1. When the request signal goes high, the data is latched, and the counter is incremented.
2. The current count and the position of the maximum are compared. If they are equal, the maximum has fallen out of the window, and it is shifted out of the FIFO.
3. The new data element is compared with each element in the list beginning with the most recently added element until the insertion position has been found.
4. The new data element is placed in the location of the oldest element that it is greater than or equal to. If it is smaller than all elements in the list, it is placed in the first empty location. If the list is full, the element is discarded.
5. The maximum data element and its position are output, and the acknowledge signal is asserted.

4. IMPLEMENTATION

The major blocks which must be implemented in our asynchronous MAXLIST architecture are the FIFO, two comparators, and the controller. The structure of the FIFO is shown in Figure 5. The FIFO must be able to shift data when the element at the head of the list has left the data window, put data on the *CMP* bus for the search through the list, and accept inserted data at arbitrary locations while clearing all subsequent locations. The information stored in the FIFO is composed of three parts: a Full/Empty bit, the position (i.e., the count when the data arrived), and the data itself.

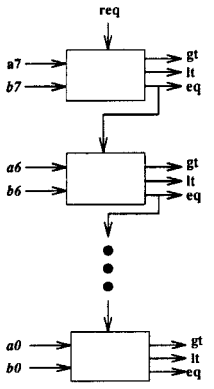

Figure 6. Block diagram of a comparator.

The comparator is composed of eight 1-bit comparators as shown in Figure 6. It is started with a request to the highest order bit. Each bit of the comparator returns whether *ai* is greater than (*gt*), less than (*lt*), or equal (*eq*) to *bi*. For the *compare equals* block, not equal is returned when any bit returns *gt* or *lt*. For the *compare greater equals* block, greater than or equal is returned for *gt* and less than is returned for *lt*. If the two bits are equal (*eq*), the next bit is compared. Finally, if the last bit returns *eq*, then equal is returned for the *compare equals* block, and greater than or equal is returned for the *compare greater equals* block. This block is highly data-dependent as the comparison may complete at varying times. The asynchronous design methodology takes advantage of this data-dependency to produce a more efficient architecture.

The last important block is the controller. This block is split into ten separate control blocks as shown in Figure 7. The *main* block accepts the request when a new datum is ready and sends the acknowledge when the current maximum has been determined, controls the input latches, output latches, and the counter. It also coordinates the *shift* and *insert* control blocks. The *shift* block is called much like a subroutine in software. When called, it handles the control signals related to the counter and maximum position comparison, and it executes the FIFO shift when the comparison determines that they are equal. The *ins8* block is called to check if the new datum can be inserted in the last location. If it can, the *ins8* block asks the *ins7* block to check, etc. until one block cannot accept the data. At that point, a signal is sent back to tell the previous block the data should be inserted in the list position that it controls. That block inserts the data in the list position that it controls, and it forwards an acknowledgement through the *ins* blocks to its left to the *main* block. Each of these control blocks has been described in behavioral VHDL which can be synthesized by our asynchronous synthesis system ATACS [4, 5].

5. RESULTS

We implemented our architecture in VHDL and simulated it for 100,000 correlated random data elements. The data was generated by filtering pseudo-random Gaussian white noise by a single-pole filter, and the output is then scaled and quantized to an 8-bit value. Due to the asynchronous na-

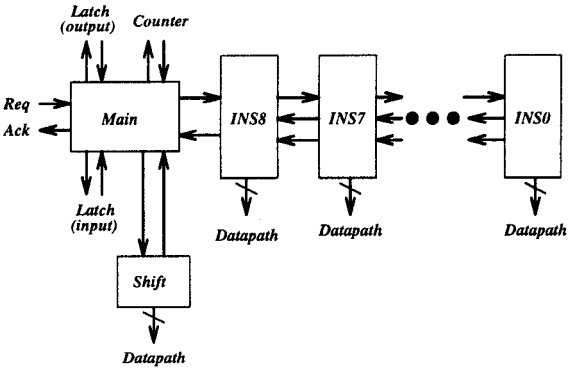

Figure 7. Block diagram of the controller.

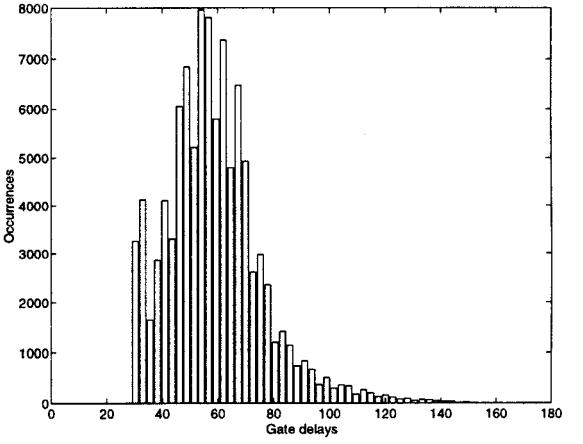

Figure 8. Data cycle delay distribution (fixed).

ture of our architecture, it is able to take advantage of data-dependent delay variations. The sources for data-dependent delay variations are in the counter, each comparator, and the number of elements in the FIFO. These variations result in an extremely variable data delay cycle as shown in Figure 8 which depicts a histogram of the delay to accept a new datum and output the current maximum. Over the course of the 100,000 elements, our minimum delay was as small as 29 gate delays and our maximum was as large as 161 gate delays. The average delay is 58.6 gate delays with a standard deviation of 17.3. As mentioned earlier, since the list size is much smaller than the window size, elements may need to be discarded. This event happened 8925 times, but *never* did the dropped element become a maximum in the sliding window.

One advantage of asynchronous design is the ability of an asynchronous design to adapt to operating conditions. The delay of a transistor in a VLSI design can vary significantly depending on the quality of the process run, the operating temperature, and the supply voltage. In a synchronous design, this variation is taken into account by adding a substantial margin to the clock cycle to guarantee that the chip operates correctly even in the most adverse circumstances. In reality, a chip typically comes from an average processing run and runs much cooler and at a higher supply voltage than in the worst-case. The speed of an asynchronous design adapts to the current operating conditions. We took

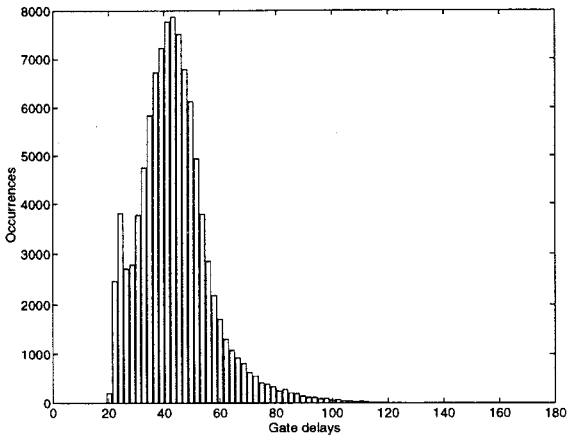

Figure 9. Data cycle delay distribution (bounded).

this fact into account in the simulation by replacing all fixed delay parameters by delay parameters which are randomly generated each cycle within a delay bound from the worst-case down to 50 percent of the worst-case. Our simulation results using these bounded delays are shown in Figure 9. The average delay improves to 43.9 gate delays with a standard deviation of 12.9. The minimum and maximum delays also improve to 19.1 and 124.3 gate delays, respectively.

6. COMPARISON

We compared our results with several synchronous implementations of the MAXLIST algorithm that were designed as class projects at the University of Utah. The best implementation designed by Julsgaard and Xu [6] had a clock frequency of 75 MHz for a $1.2\mu m$ CMOS process, and it required $6 + 2X$ cycles to accept a new datum and output the current maximum where X is the number of comparisons required. On average, they need 1.4 comparisons, or $117ns$. Assuming a $0.5ns$ gate delay for this process, this synchronous design requires on average 234 gate delays per data cycle.

In order to draw a fairer comparison, we examine the performance of a hypothetical synchronous design which uses the same architecture as our asynchronous design. For each data element in a synchronous design, one cycle would be required to latch the data and increment the counter. Another cycle is needed to perform the position comparison to see if a shift is necessary. If a shift is necessary, a clock cycle would be needed to perform it. Next, a minimum of two cycles are needed for each comparison that is going to be performed to find the location in which to insert the data into the FIFO. One is needed to determine and obtain the next element to be compared against, and the second is to perform the comparison. After the position is determined, one cycle is needed to insert the element. Finally, one cycle is required to output the current maximum. Putting it all together, we get the following:

$$\text{data cycle delay} \quad = \quad 4 + p(\text{shift}) + 2 \cdot \text{avg}(\text{cmp})$$

In the 100,000 data samples, the list needs to be shifted only 227 times, so $p(\text{shift})$ is negligible. Using 1.4 as the average number of compares, the approximate average data cycle delay in a synchronous design would be about 6.8 cycles. The counter and comparator would require at least one gate delay per bit and at least two more for control and latching data in and out. Thus, the fastest possible clock cycle time would be at least 10 gate delays. Using a 10 gate delay cycle time, the synchronous design would require on average 68 gate delays per data cycle. Therefore, our asynchronous design is at least 14 percent faster considering only data-dependent delay variations and fixed delays, and at least 35 percent faster when operating conditions are also considered using bounded delays.

If we are given a fixed throughput requirement, this speed improvement can be turned into improved power performance by lowering the supply voltage. For example, to get the same performance as the best synchronous design at 5 volts, our asynchronous design can be run at 3.2 volts. This leads to a 59 percent savings in power, since power scales as the square of the voltage.

7. CONCLUSION

As clock speeds increase, difficulties in distributing a global clock is forcing many designers to consider asynchronous architectures as a viable design alternative to synchronous ones. Asynchronous designs also can take advantage of delay variations due to data-dependencies and operating conditions at a very fine grain. The complexity of the computations in the MAXLIST algorithm are largely data-dependent, so we designed an asynchronous architecture to implement it. Due to the fact that the clock cycle in a synchronous design must be set for worst-case delays, we are able to show over a five times improvement in speed when compared with an existing synchronous design. We are also able to show that our asynchronous design can outperform an extremely aggressive, comparable synchronous design by more than 35 percent in speed or 59 percent in power.

ACKNOWLEDGMENTS

We would like to thank Professor Scott Douglas of the University of Utah for introducing us to the MAXLIST algorithm. We would like to thank Kashif Ikram and Syed Rab for their contributions in the initial architecture design.

REFERENCES

[1] S. C. Douglas. A family of normalized LMS algorithms. *IEEE Signal Processing Letters*, 1(3):49–51, March 1994.

[2] S. C. Douglas. Running max/min calculation using a pruned ordered list. *IEEE Transactions on Signal Processing*, 44(11):2872–2877, November 1996.

[3] S. C. Douglas. Private communications, 1996.

[4] C. J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Stanford University, 1995.

[5] H. Zheng and C. J. Myers. Specification and compilation of mixed-timed systems using VHDL. forthcoming paper.

[6] K. Julsgaard and Z. Xu. A VLSI implementation of the MAXLIST algorithm. Project report for CS/EE 542, University of Utah, 1995.