20

A novel globally asynchronous locally synchronous sliding window DFT implementation

D. S. Bormann
d.bormann@ic.ac.uk

P. Y. K. Cheung
p.cheung@ic.ac.uk

Department of Electrical and Electronics Engineering
Imperial College of Science, Technology and Medicine
Exhibition Road, London SW7 2BT, United Kingdom

Abstract

We demonstrate the design of a Globally Asynchronous Locally Synchronous (GALS) Discrete Fourier Transform circuit. Each locally synchronous stage is surrounded by an "Asynchronous Wrapper" which provides an asynchronous interface to an otherwise synchronous circuit. Every locally synchronous (LS) region operates independently, eliminating problems of clock skew and allowing each region to run at its own clock speed. Metastability can never occur because an asynchronous handshake "stretches" or "pauses" the local clock until data has stabilised. When new data is not available for processing the local clock stretches, automatically preventing the LS block from consuming power. When new data does arrive, the clock starts directly in phase with the handshake without wasted synchronisation time.

The internal DFT stages were designed using typical synchronous techniques. We were therefore able to use VHDL to quickly compose and synthesise the circuit using industry standard tools. Most current asynchronous design methodologies require either manual design or complex specifications which become unwieldy as circuit size grows. Locally synchronous design allows us to take advantage of normal synchronous methods, reducing design time while providing a robust interface that delay-insensitively communicates with the environment.

Keywords

Stretchable or pausible clocks, asynchronous handshaking, GALS, CORDIC, DFT

VLSI: Integrated Systems on Silicon R. Reis & L. Claesen (Eds.)

1 INTRODUCTION

Globally asynchronous locally synchronous circuits combine an asynchronous external interface with an internally clocked circuit. Instead of a fixed period clock the locally synchronous blocks use a variable length stretchable clock. These clocks, as presented elsewhere (Pechoucek, 1976) (Seitz, 1980) (Chapiro, 1984), normally behave just like a normal synchronous clock. However, the clock also has a stretch control which can prevent it from moving out of the current clock period. The stretch signal must be asserted synchronously, but can be released asynchronously. Once the stretch signal is released, the clock continues in its normal manner having simply been displaced outwards in time.

The class of circuits presented here allow a collection of locally synchronous regions to communicate without fear of metastability. The clocks do not stretch over periods of metastability like those using Q-modules (Rosenberger et. al., 1988), but instead stretch to actually prevent metastability from occurring (Pechoucek, 1976). Rather than, as in totally synchronous design, using double or triple buffering or slowing down clock speeds to increase a system's MTBF due to metastability, this allows a designer to run their circuits at maximum speed with an infinite metastability MTBF.

While the theory behind these metastability-free circuits has been known for many years, a simple and coherent methodology for designing them has not been available. By using Extended-Burst-Mode specifications (Yun, Dill, 1993), we have created a small set of asynchronous building blocks. These blocks collectively form an asynchronous "wrapper," surrounding a locally synchronous circuit and making it externally appear as an asynchronous handshake circuit. Armed with these simple circuits, a designer is able to subdivide a globally synchronous circuit into locally synchronous regions, reducing problems of clock skew and improving modularity. Additionally, a totally asynchronous system can be implemented with some or all of the asynchronous modules created using locally synchronous circuits.

Using this set of building blocks we are able to combine asynchronous and synchronous circuits simply and elegantly in a single system. The building blocks are small and compact and offer the possibility of communication between locally synchronous regions with low overheads. This extends the composability of asynchronous circuits into the synchronous domain.

2 CORDIC SLIDING WINDOW DFT ALGORITHM

The Coordinate Rotation Digital Computer (CORDIC) algorithm, first proposed by Volder (1959), rotates a vector through an angle θ using only shift and add operations. This is accomplished by splitting the rotation angle θ into a sequence of m subrotations of angles $\tan^{-1}(2^{-i})$ where i=0...m-1 for m bits of accuracy. One clock cycle is required for each subrotation and thus the entire rotation takes m cycles. After the rotation, the resulting vector also needs to be scaled which requires approximately $m/4$ extra clock cycles (Cavallaro, Luk, 1988). The equation below describes the basic CORDIC rotation operation.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \mathbf{R}(\theta)\begin{bmatrix} x \\ y \end{bmatrix} \text{ where } \mathbf{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \quad (1)$$

Kar and Rao (1996) have proposed a unified CORDIC algorithm for the evaluation of the sliding window DFT, DHT, DCT, and DST. We focus only on the DFT here for simplicity. The DFT of a window of N complex data elements $x(i), x(i+1), \ldots, x(i+N-1)$ is given below for each $k=0...N-1$. Note that the new DFT is based on the previous window where the new values $p(i+N)$ and $q(i+N)$ are added and the oldest values $p(i)$ and $q(i)$ are removed.

$$\begin{bmatrix} P_{i+1}(k) \\ Q_{i+1}(k) \end{bmatrix} = \mathbf{R}\left(\frac{2\pi k}{N}\right)\begin{bmatrix} P_i(k) + p(i+N) - p(i) \\ Q_i(k) + q(i+N) - q(i) \end{bmatrix}. \quad (2)$$

Equation 2 presents the DFT in the form of the CORDIC rotation above. For each stage k=0...N-1, a CORDIC rotation is performed with $\theta = (2\pi k / N)$. The input vector to all stages is the same so this vector can be passed unmodified from one stage to the next as the sliding window progresses.

3 ASYNCHRONOUS WRAPPER

Our goal is to create a circuit where each locally synchronous stage of an N-element sliding window DFT communicates asynchronously with its environment. We choose to use a four-phase bundled data handshake protocol. This means that there are four events per handshake cycle: *Req+*, *Ack+*, *Req-*, and *Ack-*. We adopt a late data-valid scheme (Peeters, 1996) whereby data is guaranteed to be valid when a *Req-* transition occurs and may change at any time after *Ack-*.

To provide the asynchronous interface to the environment, we surround each locally synchronous module with an *asynchronous wrapper* (see Figure 1). The wrapper synchronises the asynchronous data to the local clock by stretching or pausing the clock. When the LS module requires a data exchange, one or more ports are selected with the *Input* and *Output* signals. This causes the port's *Stretch* signal to rise until the handshake occurs. While *Stretch* is high, the *Clock+* transition is prevented until *Stretch* falls. *Stretch* can never shorten the clock period; if *Stretch* falls quickly, *Clock+* will not occur until a normal clock period expires.

The clock is only stretched once to complete a full four-phase handshake on all selected input and output ports. This is possible because each port is actually an asynchronous finite state machine (AFSM) that can execute a complete handshake without directly using the clock. Data can therefore be exchanged on all ports on every clock cycle if desired. This is a considerable advantage of our

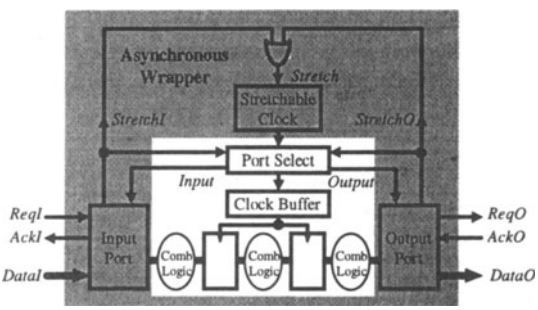

Figure 1 LS module with asynchronous wrapper

method as compared to others such as the Pausible Clock Control circuit (Yun, Donohue, 1996) which requires two clock cycles for each four-phase handshake and can only respond to one port per cycle. For a circuit with N ports, their scheme requires 2×N clock cycles to exchange data on all ports, whereas our scheme can do the same in just one cycle.

Stretchable Clock

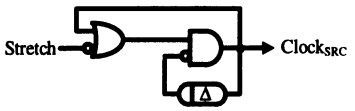

Figure 2 Stretchable Clock

The stretchable clock required for this wrapper specification must produce a clock period that is consistently longer than the worst-case combinational logic delay. *Stretch+* must arrive well before *Clock+* would normally occur to successfully stretch the clock. *Stretch* must not rise at about the same time as *Clock+* because disastrous glitches could occur on *Clock*. *Stretch* can only prevent the clock from rising and will not affect *Clock* outside of a narrow interval around *Clock+*. The circuit shown in Figure 2, an adaptation of a clock from Seitz (1980), is a very simple implementation that will satisfy these requirements.

The stretchable clock must fire all of the internal registers in each clock cycle. To increase the drive strength of the clock to support a large fanout we use an inverter chain where each stage has a fixed output load to input load ratio. The optimal ratio to minimise delay is *e* (Seitz, 1980). This loading will delay the arrival of *Clock+* by some loading value Δ(# loads). Assuming properly sized clock buffers, Δ is approximately proportional to the natural logarithm of the number of loads. For 32 internal registers, the clock buffer delay is about 10 minimum inverter; for 1000 registers, the delay is nearly 20 inverter delays. This delay is a limiting factor in how fast the clock can respond to a change in the *Stretch* signal. One challenge we would like to meet is to effectively use this clock delay time for computation.

We will call the clock signal that reaches internal registers *Clock*, while the unbuffered output at the stretchable clock source will be called *Clock*$_{SRC}$.

Input Port

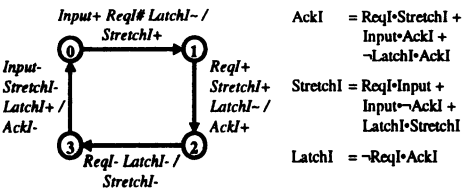

Figure 3 Passive Input Port Specification

Most applications require input ports which wait for the environment to provide an input request. This is called a *passive input port* (van Berkel, 1993) in contrast to an *active* input port which initiates a request and waits for data to be returned. While we have implementations for both passive and active input and output ports (Bormann, Cheung, 1996), we only present the passive input port used in the DFT here.

A *ReqI+* transition may arrive at a passive input port at any time. The request will not be acknowledged, however, until the LS module asserts *Input+*. Since we are using a late data-valid bundling convention, the data is not guaranteed to be valid until after *ReqI-*. Therefore *LatchI+*, the transition which triggers the input latches, must not occur until after *ReqI-*. Furthermore, we must be certain that the time from when the input data is latched to the next *Clock+* is greater than the

combinational logic delay of the first stage of logic. Finally, *AckI-* must not occur before the data is latched since after *AckI-* the data can change at any time.

The extended-burst-mode specification in Figure 3 will satisfy our requirements. Recall that an extended-burst-mode circuit waits for all transitions in the input burst to occur before sending the output burst, but that the input transitions may arrive in any order (Yun, Dill, 1993). A # symbol indicates that the signal is a directed don't-care; it may either remain at 0, monotonically change from 0 to 1, or remain at 1. Similarly, a ~ means that the signal may remain at 1, monotonically change from 1 to 0, or remain at 0.

To ensure correct behaviour of the extended-burst-mode circuit, we must meet three conditions (Yun, Dill, 1993). The *fundamental-mode environmental constraint* requires that a new input burst must not begin until the machine has stabilised. This can be met as long as the environment does not respond to an output burst faster than the internal state machine can recover. The *feedback delay requirement* can be satisfied by the synthesis tool using the conservative unbounded wire delay model. We do not have to worry about the *setup time requirement* because we are not using any conditional signals in our specifications.

Note that the problem of the clock buffer delay mentioned above also affects the input and output ports. The *LatchI* signal will be delayed by Δ(# input bits) which, for 32-bit data paths, is about 10 gate delays. Thus the data will be waiting at the input latch for at least this amount of time; it may wait longer if *ReqI* arrives before *Input*. Since we know that the data will be valid for at least this amount of time, it is possible to insert combinational logic before the input port. If there are several input ports and we would like to compute a function of multiple *DataI*'s we can still safely insert Δ(# input bits) of logic because the clock will not stop stretching until all *StretchI* signals have fallen.

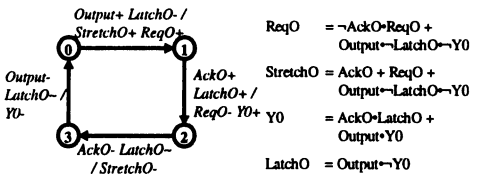

Figure 4 Active Output Port Specification

Output Port

Since the active output port initiates the handshake, *ReqO+* can occur immediately in response to *Output+*. *Output+* will also fire the output latches after Δ(# output bits). Data must be valid before *LatchO+* but since we know the minimum delay from *Output+* to *LatchO+*, we can use this time for computation as with the input port.

We would ideally like to have the internal logic stages balanced to have the same maximum combinational delay. Since the delay from *Output+* to *LatchO+* is fixed, we can delay *Output+* so that *LatchO+* lines up with the buffered *Clock+* signal. Then the output stage can have the same amount of combinational logic as internal stages and the full clock cycle can be used.

To synchronise *Clock+* and *LatchO+* we can wait to begin the output cycle until $Clock_{SRC}$+. *Clock+* always follows $Clock_{SRC}$+ by Δ(# loads), so we cannot prevent *Clock+* from sending new data to the output port. The output port will not allow the current clock cycle to complete until a full handshake completes. Therefore there is no danger of losing the current output data value because there is nothing

that can prevent *LatchO+* since the previous handshake has definitely completed when $Clock_{SRC}$+.

Note that the port may produce valid data well in advance of *ReqO-* if the environment is slow to *AckO+*. This is acceptable using the late data-valid bundling convention; we only need to ensure that data is valid before *ReqO-* and does not change until after *AckO-*.

Port Control

For each clock cycle the port control is responsible for generating the correct set of *Input* and *Output* port commands. If an input handshake is to be performed in a cycle, *Input+* occurs in response to *Clock-*. Output commands will usually be generated after *Clock+* as discussed above. The corresponding port's *Stretch* signal will rise directly in response to the command and then fall when the handshake has been completed. The falling *Stretch* signal from each port clears the *Input* or *Output* command from the port control to prepare for the next cycle.

Some LS modules may perform a full set of input and output handshakes on every cycle. The sliding window DFT algorithm, however, only needs to execute each input and output handshake every m cycles where m is the desired number of bits of accuracy in the result. Coming out of reset the DFT will require input data to begin computation. Thus each stage will be stretched until a request arrives on that stage's input port. After m cycles, a new input vector is loaded and the old vector is output to the next DFT stage. After another k cycles, the scaled vector is sent out through the lower output port. Each of these handshakes will only occur every m cycles.

Eliminating Metastability

An essential requirement for any locally-synchronous circuit is to eliminate the possibility of failure due to metastability. In this section, we demonstrate how this circuit avoids metastability and the requirements to exclude this possibility in any locally-synchronous system.

Whenever an asynchronous input is sampled by a synchronous system, there is a chance that the latched value may remain metastable for an unbounded period of time (Chaney, Molnar, 1973). This is because the input may be changing just as it is sampled and thus can be neither high nor low.

Totally synchronous systems can be designed to reduce the risk of failure to an acceptable level, but can never eliminate the possibility entirely. One method employed is to use multiple stages of buffers on asynchronous inputs, seeking to prevent a metastable signal from propagating through all the stages and into the internal logic. Another solution is simply to slow down the clock, since the probability of failure reduces as the clock period increases. These methods increase circuit area and latency in the first instance and decrease throughput in the second. These costs go towards decreasing the risk, but never remove the threat entirely.

Locally synchronous systems are a fundamental solution to the problem in that they remove the risk of metastability altogether (Pechoucek, 1976). The MTBF due to metastability increases to infinity, and we do not have to reduce our performance to reach the goal.

The key to solving this problem is in the use of the stretchable clock. Recall that

during most of the clock cycle, Clock is unaffected by any changes to *Stretch*. It is only when the clock delay has expired and Clock+ is about to occur that the clock can be prevented from rising until *Stretch* falls. Any glitches or noise on the *Stretch* signal are filtered out by the stretchable clock during the remainder of the cycle. Even if we constructed a circuit where both the *Stretch+* and *Stretch-* conditions could become true simultaneously, briefly causing *Stretch* to remain at an undefined value, the clock would be unaffected as long as this does not occur within a narrow time around *Clock+*. Therefore as long as we can restrict when *Stretch+* can occur, the rest of the circuit will be unaffected. The asynchronous wrapper only asserts *Stretch+* in response to the port select signals, and since these only occur early in a clock cycle, anomalous behaviour will not occur on the clock signal.

Any time a metastability-free locally synchronous circuit needs new input from the environment, the clock must stretch until the data is available. The LS circuit must commit to sample an input by raising the *Stretch* signal. We cannot construct a circuit that 'polls' for an asynchronous input using this technique. While we prefer metastability-free asynchronous wrapper implementations, there is no reason that we cannot use a Q-module type polling input port if it is required (Rosenberger et. al., 1988). We call this element a Q- Port (Bormann, Cheung, 1997).

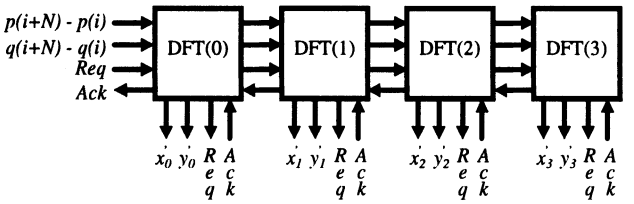

Figure 5 Systolic locally synchronous sliding window DFT implementation (N=4)

4 LOCALLY SYNCHRONOUS DFT IMPLEMENTATION

The *N*-element sliding window DFT algorithm we have implemented is composed of *N* Coordinate Rotation Digital Computer (CORDIC) stages. Each stage has its own stretchable clock and requires *m* clock cycles, one cycle for each bit of the input operands, to complete computation. The CORDIC algorithm requires that the resultant vector must be scaled after rotation, consuming an extra *k* cycles. However, since subsequent stages do not depend on the scaled vector, only the latency but not the throughput is affected by scaling. The input

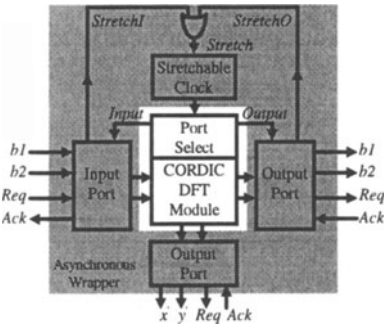

Figure 6 Asynchronous wrapper surrounding locally synchronous DFT stage

operands are unchanged from one stage to the next, and are passed after m cycles. Therefore the total time for computation of the first complete DFT is $N \times m+k$ cycles, but subsequent updated windows will be available after only m cycles.

Initially, all stages of the DFT are inactive and consume no power. When data enters the first stage from the left, the input port will signal the clock control to cease stretching and computation begins. For each of the m cycles of the CORDIC rotation, the clock is never stretched since no handshaking is required. After m clock cycles, the *b1* and *b2* inputs are passed, unchanged, through the output port to the next stage. At this point the CORDIC rotation has completed, so new data is needed to continue. The clock may stretch whilst waiting for a new set of inputs from the left. As soon as new input data has arrived, the clock will resume and k cycles later, the scaled DFT vector will be passed through the lower output port.

5 EVALUATION AND LIMITATIONS

In this section we analyse the overhead an asynchronous wrapper imposes on a locally synchronous circuit. We evaluate the area penalties of the wrapper and the maximum clock speed of a locally synchronous region, as well as comparing the power consumption to a fully synchronous implementation.

Area Overheads

Each locally synchronous region must be surrounded by its own asynchronous wrapper. The wrapper consists of a local clock, a clock buffer, and a collection of input and output ports.

Each port has a set of latches and one AFSM. The input port AFSM requires just nine 2 and 3 input gates. The output port AFSM is also made up of only nine gates. The latches would be required even if the circuit was implemented with a normal fully synchronous methodology, so we do not include them as a wrapper overhead. Therefore the number of gates added by the ports is just nine times the total number of input and output ports.

This does not include the buffer circuit to increase the drive strength of the *Latch* signal to drive the port latches. This area would normally be included in the global clock buffer circuit which is unnecessary in a LS circuit. For a 32-bit port, a fast buffer will have 3 inverter stages. Each inverter is e times larger than the last, so the inverters will have sizes e, e^2, and e^3 ($\approx$20) times the minimum size inverter.

Since the number of buffer stages only increases with the log of the number of loads, the clock buffer will only require about one extra stage than the *Latch* signal buffer for each CORDIC DFT block.

Buffer Delay

For 32-bit data, the input buffer delay will be approximately 10 inverter delays from *ReqI-* to *LatchI+*. Since we have made no effort to put combinational logic outside of the input ports for this design, this time is actually wasted on each input to the LS DFT. However, we only have to pay this penalty when data is input into the LS circuit. Since new data is only read into the LS circuit every m cycles (m = # of bits) for a CORDIC rotation, the overhead is about 0.3 inverter delays per

clock cycle. The total latency of the DFT will increase the input buffer delay times the sliding window size *N*.

Power

A locally synchronous circuit will not consume power when there is no data available for processing. This is also possible in fully synchronous designs with the use of clock gating. However, unlike a clock gated synchronous circuit, the LS implementation will not even consume power in the clock or clock buffers when inactive. This may be a significant savings in high speed designs.

The Pausible Clock Control circuit proposed by Yun and Donohue (1996) samples the handshake signals instead of stopping and waiting for a request to arrive. Thus even when there is no data available for processing, their circuit will consume the same amount of power.

Maximum Clock Speed

When determining the maximum clock speed for a locally synchronous circuit it is necessary to take into account not just the worst-case combinational logic delays but also any delays imposed by the asynchronous wrapper. Note that it is possible to have a nominal clock period that is shorter than the period for cycles in which handshaking occurs. However, we expect that most circuits will exchange data with the environment in a high proportion of clock cycles. If that is the case, most clock cycles will be longer than the worst-case combinational logic delay, thus wasting computation time unnecessarily.

For every clock cycle that handshakes with the environment, the maximum clock period will be determined by the time it takes to complete the handshake plus the delay imposed by the clock buffer. For the moment we assume that the output port of one LS module connects directly to the input port of the next. For an input cycle, the delay from *Input+* to *Stretch-* can be as little as 7 gate delays. For an output cycle, the current implementation of the output port requires that *StretchO* is high for the entire handshake. This includes the delay introduced by the *LatchO* buffer as well as the time required by the receiving input port to latch the data. Therefore from *Output+* to *Stretch-* delay is summarised in Table 1.

Table 1 Time from the start of an I/O cycle to Stretch-

Passive Input Port Delay		*Active Output Port Delay*	
		Output+ → *ReqO+*	= 2
		ReqO+ → *AckO+*	= 2
Input+ → *AckI+*	= 2	*AckO+* → *ReqO-*	= Δ(out) - 4
AckI+ → *ReqI-*	= 2	*ReqO-* → *AckO-*	= Δ(in)
+ *ReqI-* → *Stretch-*	= 3	+ *AckO-* → *Stretch-*.	= 3
	= 7 gate delays		= 23 gate delays

This delay is 23 gate delays assuming the loading delay for both the input and output ports is 10 gate delays. We are making the conservative assumption that the inverter delays in the buffers are as long as the gate delays of the AFSM. The clock period must include the clock buffer delay as well, so the period in this case would be approximately 33 gate delays.

Note that with our asynchronous wrapper, the number of ports per LS module only affects the clock speed by increasing the fan-in of the OR-gate driving *Stretch*. This is in sharp contrast with the Pausible Clock Control method (Yun, Donohue, 1996) which requires a tree of arbiters to merge all incoming handshake signals. As well as adding the area of one arbiter per port, this will increase the response time of the circuit. Using the arbiter delay given in their paper, the delay from an incoming request to an acknowledge will increase by approximately 1 ns per port for both the *Req*+→*Ack*+ and the return-to-zero phase of each handshake.

Timing Assumptions

Locally synchronous systems strongly rely on bounded-delay assumptions about circuit elements. First we have assumed that a bundled data protocol can be met. More critically, we are relying on a circuit delay element to produce a clock period which is consistently long enough to allow combinational worst-case timing requirements to be met.

Since these are both one-sided requirements, they should not prove difficult to satisfy. Furthermore, if worst-case combinational delays are unacceptable for some part of a design, this methodology allows asynchronous completion detection logic to stretch the clock if necessary.

We cannot allow *Stretch*+ to occur within about one gate delay of $Clock_{SRC}$+. If *Stretch*+ is allowed to happen during this period it is possible for the clock to glitch causing the circuit to fail. This is also not difficult to satisfy by firing *Input*+ and *Output*+ in response to $Clock_{SRC}$ transitions. *Stretch*+ takes three gate delays from either *Input*+ or *Output*+ and thus will never cause glitches on the clock.

6 CONCLUSIONS

Some excellent work has been published on locally synchronous circuits in the past, most notably in Chapiro's 1984 thesis. Using Chapiro's "Escapement" circuits or the more recent Pausible Clock Control circuit (Yun, Donohue, 1996), however, it is necessary to stretch for each incoming handshake transition. This means that a four-phase handshake protocol needs to stretch the clock twice to complete, requiring a minimum of two full clock cycles. Furthermore, even when using two-phase handshaking, Escapement systems need to wait for all requests to arrive before acknowledging any of them. Pausible Clock Control circuits have the additional penalty of requiring arbitration between all incoming handshake signals and can only respond to one port request per clock cycle.

We have overcome these concurrency problems through the use of asynchronous state machines. Using extended-burst-mode specifications, we were able to design modules to complete asynchronous handshaking outside of the clock. This ensures that the clock only needs to run when actual computation is taking place, and that the clock does not stretch unnecessarily.

Q-modules are another alternative for globally asynchronous locally synchronous design (Rosenberger et. al., 1988). This method is similar to that used by the Pausible Clock Control circuits. We prefer our method for several reasons. First, we do not require specially designed circuits for detecting metastability after

latching signals. Second, since Q-modules rely exclusively on polling external signals, they cannot respond directly to handshake requests and may require up to one full clock cycle to realise that a new request has arrived. Finally, this type of busy-wait behaviour consumes power unnecessarily, the internal clock running at full speed even when no new data is available.

Globally asynchronous locally synchronous design offers many of the advantages typically cited by totally asynchronous circuits. The clock signal does not need to be distributed globally, and thus clock skew becomes much less of a problem. Furthermore, by using request-acknowledge signalling instead of relying on global timing constraints, these circuits are more modular and composable than totally synchronous designs.

LS blocks only consume power when it is needed, stretching the clock at all other times. However, rather than the block itself waiting for an input as in asynchronous logic, it is the clock that is waiting. This means that unlike synchronous techniques such as clock gating, even the clock will not be drawing power when the circuit is inactive.

Asynchronous circuits are often claimed to be faster than their synchronous counterparts. This advantage is gained in part because synchronous clocks need to be slowed down to accommodate worst case timing constraints. Worst case timing incorporates factors such as clock skew, environmental sensitivity, process variations, and completion time variance. These locally synchronous systems suffer less from clock skew since the clock is less widely distributed. Furthermore, since the local clocks are fabricated within the LS region, the clock speed will track environmental and process variations (Dean, 1992). Finally, the use of request/acknowledge signalling allows completion detection and reduced clock periods for quickly completing computations.

Using VHDL we have been able to quickly design a circuit which can be optimised using the best industry standard tools available. We are able to leverage the synchronous design techniques that have been developed over the past 50 years, and yet still gain many asynchronous benefits. Industry may be more receptive to the asynchronous design world when they can preserve so much of their past investment. Finally, locally synchronous design offers the possibility of combining asynchronous and synchronous circuits simply and elegantly in a single system. This extends the composability of asynchronous circuits into the synchronous domain.

7 ACKNOWLEGEMENTS

This work was funded in part by LSI Logic, Europe.

We are deeply indebted to Pedro Molina for his comments and suggestions. Pedro's ability to spot good ideas and to persevere has been critical in bringing this work forward.

Finally, David would like to thank the asynchronous group at Sun Microsystems Labs and Ivan Sutherland for the opportunity to work with them under their internship program. The insight gained by being immersed in their years of asynchronous experience has been invaluable.

8 REFERENCES

van Berkel, K. Handshake Circuits: an Asynchronous Architecture for VLSI Programming. Volume 5 of International Series on Parallel Computation, Cambridge University Press, 1993.

Bormann, D. S. and Cheung, P. Y. K. (1996) Mixed Asynchronous/Synchronous Circuits Using Stretchable Clocks. *Proceedings of the ACiD-WG Workshop*, Groningen, The Netherlands.

Bormann, D. S. and Cheung, P. Y. K. (1997) Asynchronous Wrapper for Heterogeneous Systems. Proceedings of the International Conference on Computer Design (ICCD), IEEE Computer Society Press.

Cavallaro, J. R. and Luk, F. T. (1988) CORDIC arithmetic for an SVD processor. Journal of Parallel and Distributed Computing, 5(3), 271-290.

Chapiro, D. M. (1984) Globally-Asynchronous Locally-Synchronous Systems. PhD thesis, Stanford University.

Dean, M. E. STRiP: A Self-Timed RISC Processor Architecture. Ph.D. thesis, Stanford University, 1992.

Kar, D. C. and Bapeswara Rao, V. V. (1996) A CORDIC-Based Unified Systolic Architecture for Sliding Window Applications of Discrete Transforms. *IEEE Transactions on Signal Processing*, 44(2).

Pechoucek, M. (1976) Anomalous response times of input synchronizers. *IEEE Transactions on Computers*, 25(2):133-139.

Peeters, A. M. G. (1996) Single-Rail Handshake Circuits. Ph.D. thesis, Eindhoven University.

Rosenberger, F. U., Molnar, C. E., Chaney, T. J. and Fang, T. P. (1988) Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005-1018.

Seitz, C. L. (1980) System timing, in *Introduction to VLSI Systems* (ed. C. A. Mead and L. A. Conway), Addison-Wesley.

T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. IEEE Transactions on Computers, C-22(4):421-422, April 1973.

Volder, J.E. (1959) The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, vol. EC-8, 330-334.

Yun, K. Y. and Dill, D. L. (1993) Unifying synchronous/asynchronous state machine synthesis. *Proceedings of the ICCD*, IEEE Computer Society Press, 255-260.

Yun, K. Y. and Donohue, R. P. (1996) Pausible Clocking: A First Step Toward Heterogeneous Systems. Proceedings of the ICCD, IEEE Computer Society Press.

9 BIOGRAPHY

David Bormann received BS degrees in electrical engineering and in computer science from the University of Wisconsin, Madison in 1992. He is now studying for the PhD in electrical engineering at Imperial College, University of London. Other research interests include computer simulation and visualisation, computer architecture and VLSI design.

Peter Cheung is currently Reader in Digital Systems at Imperial College, University of London. His research interests include asynchronous systems, VLSI architecture, reconfigurable computing and hardware/software codesign.