A Scalable Dual-Clock FIFO for Data Transfers Between Arbitrary and Haltable Clock Domains

Ryan W. Apperson, Zhiyi Yu, Michael J. Meeuwsen, Tinoosh Mohsenin, and Bevan M. Baas, *Member, IEEE*

***Abstract*—A robust, scalable, and power efficient dual-clock first-input first-out (FIFO) architecture which is useful for transferring data between modules operating in different clock domains is presented. The architecture supports correct operation in applications where multiple clock cycles of latency exist between the data producer, FIFO, and the data consumer; and with arbitrary clock frequency changes, halting, and restarting in either or both clock domains. The architecture is demonstrated in both a 0.18-μm CMOS full-custom design and a 0.18-μm CMOS standard cell design used in a globally asynchronous locally synchronous array processor. It achieves 580-MHz operation and 10.3-mW power dissipation while performing simultaneous FIFO READ and WRITE operations at 1.8 V.**

***Index Terms*—Asynchronous, dual-clock first-input first-output (FIFO), scalable, VLSI.**

I. Introduction

EVER shrinking transistor sizes have enabled the integration of a greater number of components onto a single chip—thus making systems-on-a-chip (SoCs) with many complex modules a common design solution. Unfortunately, global interconnect scaling has not been able to maintain the same performance increases [1], causing the timing of high speed global clock signals to become a major concern in system design. This has resulted in clock distribution circuits requiring increasing circuit resources and design time.

Nearly all existing digital systems utilize synchronous design techniques which normally require an accurate and highly synchronized global clock reference to be supplied to all areas of the circuit. One solution for coping with the clock distribution problem is to utilize *self-timed* or *asynchronous* circuits, which do not have a global timing reference signal. However, the lack of mature design tools and the reluctance of industry to incur the cost and risk of moving away from successful synchronous design flows have limited the acceptance of these design styles [2].

An alternative approach is to create systems that mix asynchronous and synchronous design techniques using a globally asynchronous locally synchronous (GALS) [3] design approach. In this paradigm, blocks are built using traditional synchronous design techniques, but these synchronous blocks do not share global timing information and are asynchronous with respect to each other. While it is often convenient to divide a system into multiple subcomponents, it is unlikely that these components will operate autonomously. Accordingly, data transfer is required between local synchronous blocks. Accomplishing this task reliably and efficiently are key challenges in GALS designs.

One structure that is particularly well-suited for this task is the *dual-clock first-input first-output (FIFO)* or *mixed-clock FIFO*. The basic FIFO architecture must be modified to accommodate two independent clock inputs. Data passing through the FIFO module will enter with reference to one clock and exit with reference to the other clock. In this way, data can be passed safely between independent clock domains. Other important applications of dual-clock FIFOs include cases where data are transferred between blocks in clock domains that are not fully asynchronous but yet unsynchronized.

The presented design enables the transfer of data between modules from completely unrelated clock domains with nonzero cycles of latency between the producer and consumer. It is particularly useful in applications where throughput rather than latency is critical—such as in many DSP applications.

TABLE I
Comparison of Various Dual-Clock FIFO Designs

Dual-clock FIFO design	Clock requirement	Storage elements	Clock haltable
Greenstreet [8]	mesochronous	–	–
Chakraborty [9]	requires training time	–	–
Siezovic [10]	arbitrary	linear	–
Chelcea [11]	arbitrary	registers	–
Cummings [12]	arbitrary	memory array	no
This work	arbitrary	memory array	yes

A. Related Work

Dally and Poulton [4] and Balch [5] present high-level views of dual-clock FIFO structures, but details of dual-clock FIFO designs are lacking in the literature.

Fully asynchronous FIFOs often appear in the literature [6], [7], but these designs do not utilize clocks, and therefore, are difficult to apply in cases of synchronizing data between clock domains.

Table I lists several dual-clock FIFO designs. In the work presented by Greenstreet [8], the clocks are derived from the same base frequency, but may have an arbitrary phase difference—which is slightly more general than strict mesochronous.

Manuscript received September 2, 2006; revised March 9, 2007. This work was supported in part by Intel Corporation, by University of California (UC) Micro, by the National Science Foundation under Grant 0430090, by MOSIS, and by a University of California at Davis (UCD) Faculty Research Grant.

R. W. Apperson is with Boston Scientific CRM Division, Redmond, WA 98052 USA.

Z. Yu, T. Mohsenin, and B. M. Baas are with the Electrical and Computer Engineering Department, University of California, Davis, CA 95616 USA (e-mail: zhyyu@ece.ucdavis.edu; bbaas@ucdavis.edu).

M. J. Meeuwsen is with the Digital Enterprise Group, Intel, Hillsboro, OR 97124 USA.

Digital Object Identifier 10.1109/TVLSI.2007.903938

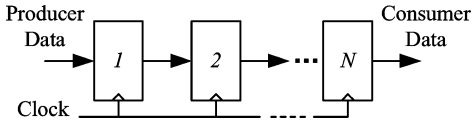


Fig. 1. Linear shift-register FIFO block diagram.

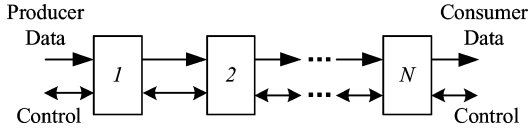


Fig. 2. Linear elastic FIFO block diagram.

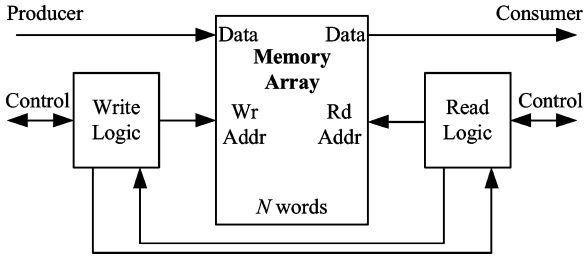


Fig. 3. Circular FIFO block diagram.

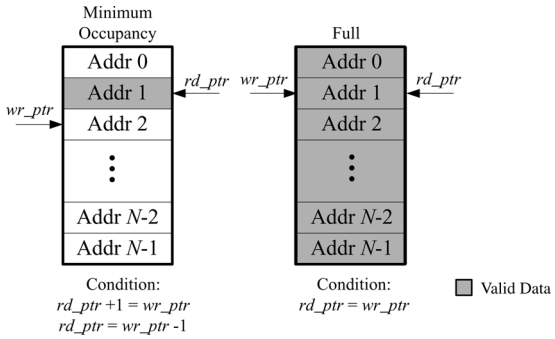


Fig. 4. Typical WRITE and READ address pointer scheme for a circular FIFO and its $full$ definition when $rd\_ptr = wr\_ptr$.

The FIFO designed by Chakraborty *et al.* requires time to develop a frequency difference estimate before transferring data, as well as usage of different circuits depending on which clock domain has the higher rate [9]. Siezovic [10] presents a linear FIFO architecture for data synchronization, which has the limitations presented in Section II-A. An alternative FIFO architecture for use in some dual-clock applications is presented by Chelcea and Nowick [11]. The design uses independent registers as storage elements, and each register has its own $empty$ and $full$ signals. This scheme reduces the latency when the FIFO size is small, but is less suitable when the FIFO size is large.

This work uses a dual-port SRAM as the storage element which increases memory density and improves FIFO size scalability [13]. Compared with the most similar previous work [12], this design includes configurable logic to make it suitable for many environments, and also enables complete oscillator halting during idle times to achieve high energy efficiency. The proposed FIFO design has been fabricated in what we believe is the first VLSI implementation of a GALS array processor [14].

B. Paper Outline

Section II introduces key structures and parameters for all styles of FIFO buffers by analyzing the single-clock case. Section III discusses synchronization and metastability issues. Section IV describes the proposed design of an efficient and robust dual-clock FIFO architecture. Finally, Section V describes a specific hardware implementation of the dual-clock FIFO architecture.

II. Single-Clock FIFOs

To best address dual-clock FIFO issues, we first consider the case of a single-clock synchronous FIFO. This section covers these fundamental FIFO principles.

A. Linear FIFOs

The simplest FIFO structure consists of a linear chain of latches or flip-flops connected serially as a shift register. Data is shifted into one end of the chain and propagates through every memory element until it reaches the end as shown in Fig. 1. This FIFO is synchronous since all movement of data requires a common clock.

Alternatively, a linear elastic FIFO uses control signal handshakes to propagate data from location to location. Unlike the synchronous case, a datum can propagate through the FIFO without any new items entering. This results in the FIFO being at various degrees of fullness, hence, the name elastic. FIFOs of this nature work well with asynchronous designs and many examples of these can be found in the literature [15], [16]. A simple example of this type of FIFO is shown in Fig. 2.

Drawbacks of these approaches become evident with large FIFO sizes, and include high latency, low-power efficiency, and low memory density. They do, however, work well for small FIFO sizes. High latency and power dissipation arise from the fact that each datum must flow through every element of the FIFO. Additionally, in the synchronous case every memory element requires an individual clock signal that impedes scalability and increases power consumption. Latches and flip-flops also have large circuit areas per bit.

Many extensions of this basic FIFO structure have been proposed with the key differences being the path by which data travels through the structure, resulting in lower latencies and improved energy efficiency. Examples of these variant structures are square FIFOs, parallel FIFOs, tree FIFOs, and folded FIFOs [16].

B. Circular FIFOs

A sometimes more efficient FIFO construction is to create a circular buffer using an array of arbitrarily addressable memory elements enabling low latencies and high energy efficiencies [17], as shown in Fig. 3. Scalability is dramatically improved due to the fact that clock and data signals are not strongly affected by the FIFO size, and by higher memory densities.

Tracking valid data within an N-word FIFO is typically accomplished by maintaining READ and WRITE address pointers which indicate the beginning and end of the valid data range in the memory—as shown in Fig. 4. Using the READ and

WRITE pointers alone to define the empty and full conditions presents problems in representing all possible states because the case where $rd\_ptr = wr\_ptr$ is ambiguous. A common solution is to increase the size of the address pointers by one bit. Empty detection is accomplished by an equivalence test of the address pointer, and full detection is accomplished by an equivalence test of the lower $\log_2(N)$ bits and an XOR of the address pointers' MSBs. For correct operation, the following inequalities must hold at all times:

$$rd\_ptr \leq wr\_ptr \leq rd\_ptr + N. \tag{1}$$

III. Synchronization

A fundamental problem in systems lacking a single global timing reference is *synchronization*. In general, the timing relationship between a signal and a clock can be cast into one of five categories [4], [18]: 1) synchronous; 2) mesochronous, where the signal is the same frequency as the clock, but has a constant phase difference; 3) plesiochronous where the signal is at a frequency close, but not identical to the clock frequency, which implies a varying phase difference; 4) periodic, where the signal has an unknown relationship to the clock, but is periodic in nature; and 5) asynchronous, where the signal is completely unrelated to the clock and signal transitions are arbitrary.

A. Metastability

Metastability is a fundamental problem present when interfacing asynchronous blocks [4] and is caused by registers not receiving a stable input signal near the active edge of the clock signal. Synchronization methods are used to avoid or reduce the probability of metastability. An approximation for modeling the mean time between failures (MTBF) is shown in (2) [19], where f_c is the clock frequency, f_i is the input data event frequency, t_r is the allowed settling time before sampling, τ is the exponential time constant of the metastability decay rate, and T_0 is the asymptotic width of the time aperture in which the device can enter the metastable state, normalized to a response time of zero [20]

$$\text{MTBF} = \frac{e^{t_r/\tau}}{T_0 f_c f_i}. \tag{2}$$

B. Asynchronous Synchronization Strategies

Fully asynchronous signals are the most difficult to synchronize since no information is available regarding timing of the signals. Solutions to synchronize asynchronous signals fall into one of two categories: 1) increasing time for resolution and 2) clock pausing [19].

One of the most basic synchronizers is the two flip-flop synchronizer [4]. Extensions of this idea are numerous, and variations of the scheme generally tradeoff increased area and/or latency for a lower MTBF. These schemes do not eliminate the probability of a metastable event; they only reduce it.

The second category of solutions uses *pausible clocks* [3], [21], [22]. The main benefit of using pausible clocks is that it can reduce the probability of synchronization failure to zero. However, these solutions also require that each module's clock can be locally controlled. Also, in cases where arbiters are used to resolve asynchronous conflicts, the system must be able to tolerate nondeterministic delays. In general, when the clock is paused the entire system is frozen and must wait for arbitration to complete before any work can be done. This can be costly and complex, especially when interfacing to multiple asynchronous signals.

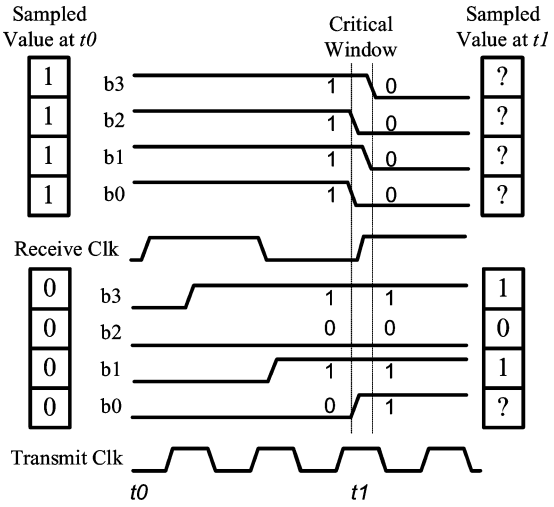


Fig. 5. (top) Sampling of a multibit transition word and (bottom) a single-bit transition word.

C. Synchronizing Multibit Words

Sampling multibit words introduces additional problems beyond metastable sampling circuits. Real systems have varying amounts of delay per wire and therefore nonuniform timing per bit. Every bit within each word can independently take on the old value or the new value—after a sufficient metastability resolution period—possibly producing incorrect results.

A solution to this problem is to allow only single-bit transitions within words, if possible. In that case, only one bit in each word has an unpredictable resolution value. One resolution value results in the previous word and the other resolution value results in the new word—no erroneous words are possible! In the worst case, only one bit is metastable. Examples of these cases are illustrated in Fig. 5.

Binary numbers increasing by one and mapped to *gray codes* exhibit this property. Note that correct operation holds regardless of the relative frequencies of the input signal and the receiving clock. This property is a key link in the design of the proposed dual-clock FIFO.

IV. Proposed Dual-Clock FIFO Architecture

This section describes a dual-clock FIFO architecture which supports data transfer across two clock domains of completely arbitrary phase and frequency. The haltable clock logic, configurable source synchronization, delay cell, and reserve space architecture are also described.

A. Overview of the Proposed Dual-Clock FIFO

Fig. 6 shows a working environment of the proposed dual-clock FIFO, which is used to transfer data through a *producer*

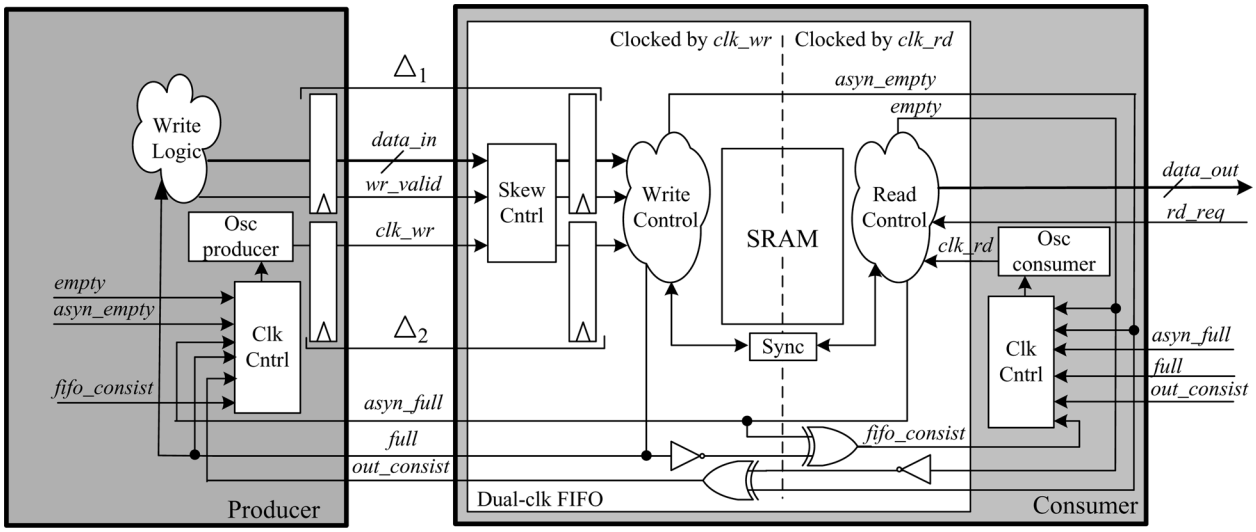


Fig. 6. Example usage of the proposed dual-clock FIFO for transferring data between producer and consumer blocks that each contain haltable oscillators. Note that the architecture supports halting the clock oscillators, and not just stopping the local FIFO clock. The two Δ variables represent delays from the producer to the consumer with Δ_1 equal to the delays for $data\_in$ and wr_valid, and Δ_2 equal to the delay for clk_wr.

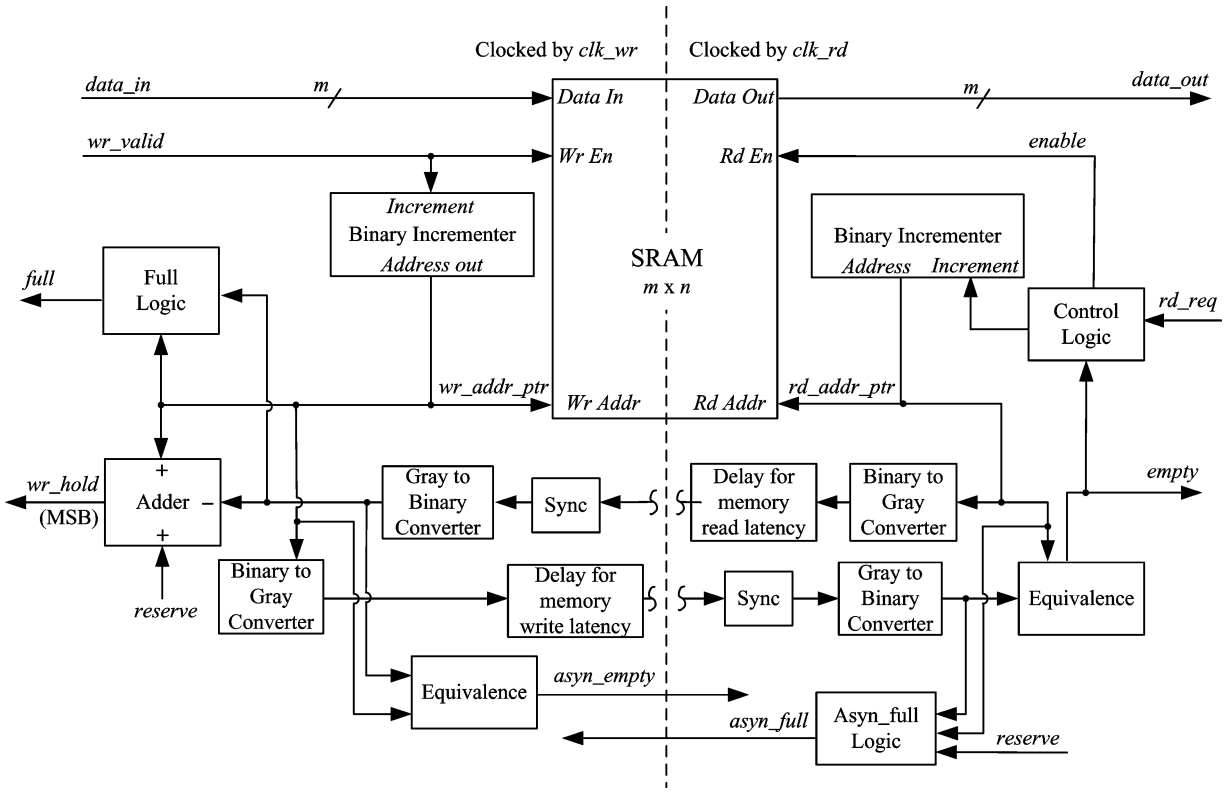


Fig. 7. High-level diagram of the blocks within the proposed dual-clock FIFO.

and a *consumer*. The clocks of the producer and consumer (clk_wr and clk_rd) are completely unrelated.

The producer sends signals such as data ($data\_in$), control (wr_valid), and clock (clk_wr) to the consumer through a communication channel with delays Δ_1 and Δ_2. The communication delay for these signals and the additional clock tree delay for the clock signal will change timing and may not meet timing requirements. A *skew control* block, which includes reconfigurable delays, is inserted to balance the timing between signals. The FIFO writes the received data into an SRAM under the control of the producer's clock clk_wr and reads from the SRAM under the control of the consumer's clock clk_rd.

More details of the FIFO's architecture are given in Fig. 7. The WRITE logic shown on the left-hand side of the figure is separated by a dashed line from the READ logic on the right-hand side.

The FIFO WRITE or READ functions are halted when the FIFO is full or empty, respectively. In order to achieve higher energy efficiency, the circuit's oscillator—not just the local clock for the FIFO—is stopped when the FIFO is halted. Signals

$asyn\_full$ and $asyn\_empty$, generated by $asyn\_full$ logic and *equivalence* blocks, respectively, are used for the clock stop and wake up logic that is further described in Section IV-E.

READ and WRITE address pointers are used to indicate the beginning and end of the valid data. To prevent multibit word failures while crossing the clock domain boundaries, the address is transformed to a Gray code representation. *Sync* blocks are used to synchronize the information which is passed across the clock boundary. A configurable multiple register synchronization circuit is used to alleviate metastability issues.

The FIFO calculates whether or not it is empty on the READ side. If the FIFO is not empty, the consumer asserts a *request* signal indicating it would like data. The FIFO indicates whether or not it is full on the WRITE side. The producer should only send data when the FIFO is not full and it indicates valid data by asserting a wr_valid signal. Ideally, the producer should stop writing data immediately when the FIFO is $full$. But in order to stop writing the FIFO, the producer needs to receive the FIFO $full$ signal, through some WRITE logic, then it stops sending data. These steps may cost several clock cycles through the Δ_1 and Δ_2 delays. A configurable *reserve space* (described in Section IV-C) is added to guarantee correct functionality.

B. Address Pointer and Gray Coding

The proposed architecture utilizes READ and WRITE address pointers to track occupancy of the FIFO. The pointers are increased to $\log_2(N)+1$ bits to allow straightforward use of all N memory words. Because many applications do not allow local clock pausing, the technique of increasing the metastability resolution time is used to pass pointers across clock domains.

Since the address pointers are susceptible to multibit word failures, they are transformed to a gray code representation before being passed across the clock boundary. Addresses are then converted back to binary format in the other domain since arithmetic is most naturally performed on binary numbers. As described in Section III-C, this approach guarantees correct pointer transfer regardless of relative clock frequencies or pausing. In the case where the old pointer value is received, the pointer will merely be interpreted as having remained at its old location (i.e., no READs or WRITEs have occurred). While this potentially adds latency to the system, it will never cause incorrect FIFO operation.

Special circuits are required to convert pointers between binary and Gray code formats. Given an n-bit binary vector $(b_{n-1}, b_{n-2}, \ldots, b_1, b_0)$, the equations in (3) can be used to convert to an n-bit gray coded vector $(g_{n-1}, g_{n-2}, \ldots, g_1, g_0)$, where "+" indicates the sum ignoring the carry. This can be accomplished using the XOR function. The worst case gate delay for this calculation is one XOR gate; a total of $n-1$ XOR gates are required

$$\begin{aligned}(\text{MSB}) \quad g_{n-1} &= b_{n-1} \\ g_{n-2} &= b_{n-1} + b_{n-2} \\ g_{n-3} &= b_{n-2} + b_{n-3} \\ &\cdots \\ g_0 &= b_1 + b_0. \end{aligned} \tag{3}$$

To convert back to binary, the equations in (4) can be used where "+" indicates the sum ignoring the carry. These calculations have a worst case gate delay of $n-1$ XOR gates for an n-bit vector and require a total of $n-1$ XOR gates. If necessary, techniques can be used to reduce this worst case delay [13]

$$\begin{aligned}(\text{MSB}) \quad b_{n-1} &= g_{n-1} \\ b_{n-2} &= b_{n-1} + g_{n-2} \\ b_{n-3} &= b_{n-2} + g_{n-3} \\ &\cdots \\ b_0 &= b_1 + g_0. \end{aligned} \tag{4}$$

When exchanging address pointers, it is crucial to take into account memory core READ and WRITE latencies to avoid data corruption and loss. Delays that compensate for these latencies consist of registers placed immediately before the data synchronization circuits as indicated in Fig. 7.

C. Reserve Space and $full$ Detector

Some applications require multicycle delays between a FIFO and the interfacing data producer or consumer. When multiple clock cycles separate the FIFO from the consumer, there is a possibility that data requests will be made to the FIFO that it will not be able to fulfill. This normally does not present a problem since a control signal (e.g., $empty$) accompanying READ data can prevent the misinterpretation of unfulfilled READ requests. However, when multiple clock cycles separate the FIFO from the data producer, the possibility of overflow exists if special precautions are not taken. Fig. 6 illustrates this case showing the critical path from the $full$ detection logic through the data producer's logic, and back to the FIFO's WRITE port. Prevention of overflow can be accomplished in one of two ways: either by the addition of a secondary FIFO, or by the FIFO signaling the data producer to stop writing when its occupancy reaches a certain level before it is full (which we call $reserve$ space). This space unfortunately reduces the effective size of the memory under many conditions. Because of its simpler implementation, the second method will generally result in a more efficient design.

Space left in the FIFO is determined by the difference between the two pointers. If the difference is less than or equal to the $reserve$ value, the FIFO must signal the producer to stop sending data. Any data left in the pipeline between the producer and the FIFO is safely written to the reserve space. Additional logic can be added to intelligently request small bursts of data to utilize some of the reserve space when it is unused.

In order to detect the FIFO full situation, we examine the case of a nonzero reserve space. Since $wr\_ptr - rd\_ptr$ is the occupancy of the memory, the signal threshold is then

$$wr\_ptr - rd\_ptr \geq N - \text{Reserve} \tag{5}$$

$$wr\_ptr - rd\_ptr + \text{Reserve} \geq N. \tag{6}$$

We prefer the form in (6) because a value can easily be tested to be greater than or equal to N when using $\log_2(N)+1$ bit words by checking the MSB of the sum—the inequality is true if the MSB $== 1$. The left-hand side of (6) is calculated with

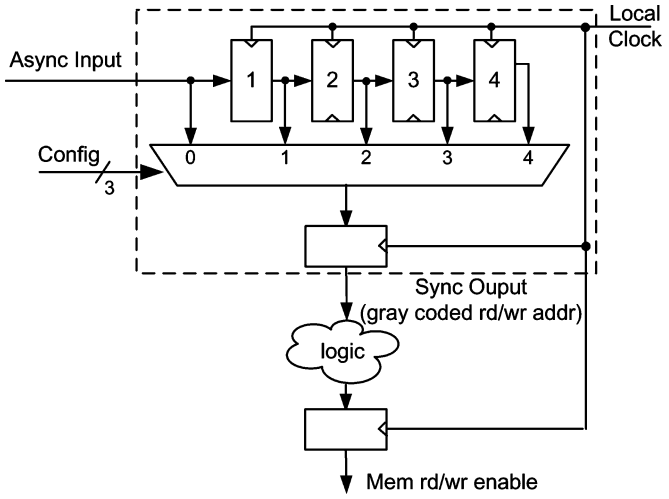


Fig. 8. Synchronizer element with configurable metastability resolution time.

a three-input, $\log_2(N)+1$ bit adder. While converting values to signed 2's complement form will certainly work, it is not required and simple unsigned values will work properly for all cases since modulo $2N$ arithmetic will effectively map all negative values to their value plus $2N$. A fast single carry-save adder stage followed by a carry-propagate stage with simplified full adders results in very fast calculation times [13].

Other required arithmetic logic for the read and write sides is roughly equivalent. Binary incrementers are needed for address generation, and comparators (a bitwise XNOR and a wide AND) are needed for empty detection.

D. Synchronizers

Because of its simplicity and robustness, we chose a pipeline synchronizer to synchronize address pointers. Additionally, since the target system can run at various frequencies, a configurable length synchronizer decouples the resolution time from the clock frequency. In this way at high frequencies, more stages can be used to ensure reliability, but at lower frequencies the number of stages can be reduced for lower latency. The synchronizer circuit is shown within the dotted line box in Fig. 8.

The unsynchronized path ($Config = 0$) was originally included only for metastability characterization purposes and it was thought it would not be used in normal operation. However, no errors were observed with this configuration in actual silicon measurements for more than 10^{14} FIFO transactions. This phenomena can be explained as follows using Fig. 8: although the probability of *Sync Output* becoming metastable is likely to be high with $Config = 0$, it does not directly result in a FIFO failure; instead, the synchronized signals are used to determine the memory rd/wr enable signals in a following pipe stage and the FIFO operation fails only when those later signals are incorrect. Since the delay of the logic after *Sync Output* is much less than one clock period in this design, an additional settling time for asynchronous signals further reduces the likelihood of metastability in critical circuitry.

Using the methods described by Jex and Dike [20] for estimating the metastability time constant τ, and Portmann and Meng's [23] method for estimating T_0, the estimated worst case T_0 and τ measurements for the synchronizer are given in Table II. Table III shows the estimated mean time between failures (MTBF) of the synchronizer architecture. When operating at 500 MHz, which is the approximate clock rate of the fabricated chip, even the case of the unsynchronized path ($N = 2$) has an MTBF of more than 10^{22} years. Details of the measurement procedures and results are discussed elsewhere [13].

TABLE II
METASTABILITY PARAMETERS FROM HSPICE NUMERICAL SIMULATOR
(test conditions = 100 °C AND V_{DD} = 1.62 V)

Test Condition	T_0 (ps)	τ (ps)
Data transition high to low	600	-
Data transition low to high	720	-
Forced to metastable point and released	-	30.5

E. Clock Halting and Restarting

1) Clock Halting and Restarting Method: To make the FIFO function correctly, the producer should stop writing data when the FIFO is full, and the consumer should stop reading data when the FIFO is empty. For high power efficiency, a novel architecture is added to the FIFO which can stop the WRITE and READ clock during these situations. The method is explained by an example shown in Fig. 9 where the WRITE clock clk_wr, halts and restarts in a full FIFO situation. The figure shows the WRITE clock and READ clock are unrelated and can change at any time.

As shown in Figs. 6 and 7, the FIFO full status is checked at the WRITE side. The $full$ signal is generated by comparing the write address and synchronized READ address under the control of clk_wr. The rising edge of $full$ indicates that the clk_wr can be halted when the processor is in a safe state.

The clk_wr should be restarted quickly when the FIFO is READ by the consumer and exits from full status. But, at that time, $full$ stays high since it is controlled by the halted clk_wr, and cannot be used to wake up the clock. The replica signal of $full$, called $asyn\_full$, is used to solve this deadlock problem. $Asyn\_full$ is calculated at the READ side. This signal is generated by comparing the READ address and the synchronized WRITE address, under the control of clk_rd. Since $asyn\_full$ is controlled by clk_rd, it goes low when the FIFO is read. Therefore, it can be used to wake up clk_wr, then $full$ goes low at the rising edge of clk_wr and the system is back in normal operation.

A similar situation exists in the empty situation. The signal $empty$ is generated at the read side and is used to stop clk_rd, and the signal $asyn\_empty$ generated in the WRITE side is used to wake up clk_rd.

It is worth noting that $async\_full$ and $async\_empty$ do not require any synchronization circuits when they cross the clock domain boundary since they activate logic only when one clock domain is off.

2) Implementation of Clock Halting and Restarting Logic: There are several ways to implement the clock halt and restart logic. The most straightforward method is to use rising edge detectors to check signal $full$ and falling edge detectors to check signal $asyn\_full$, then stop or restart the clock as shown in Fig. 9. Two traditional rising edge detector circuits are shown in Fig. 10. One method, shown in Fig. 10(a), is to connect the

TABLE III
ESTIMATED MEAN TIME BETWEEN FAILURES USING SYNCHRONIZER FROM FIG. 8 AND WORST CASE SIMULATED DEVICE PARAMETERS
(T_0 = 720 ps, τ = 30.5 ps, t_{ff} = 300 ps, $t_{\rm mux}$ = 50 ps, AND $t_{\rm logic}$ = 700 ps)

Clock	Number of Synchronizing Flip-Flops				
Frequency	2	3	4	5	6
1000 MHz	5.00 sec	1.4×10^{3} yrs	1.36×10^{13} yrs	1.26×10^{23} yrs	1.17×10^{33} yrs
750 MHz	877 yrs	4.54×10^{17} yrs	2.34×10^{32} yrs	1.21×10^{47} yrs	6.28×10^{61} yrs
500 MHz	1.9×10^{22} yrs	3.07×10^{46} yrs	4.94×10^{70} yrs	7.95×10^{94} yrs	2.87×10^{140} yrs

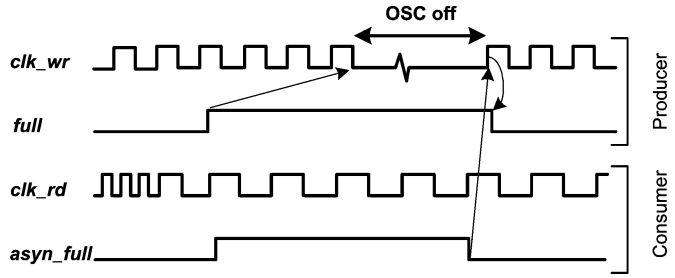


Fig. 9. WRITE clock halting due to the full FIFO and restarting due to non-full FIFO.

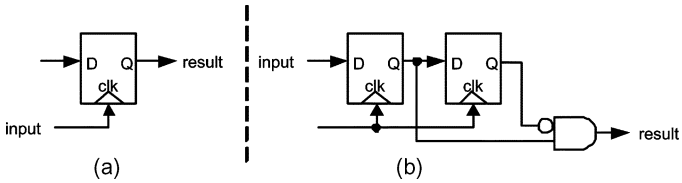


Fig. 10. Traditional methods to detect signal rising edge.

desired signal directly into the *clock* port of a flip-flop. The circuit is simple, but it is sensitive to noise and is not safe for many physical design flows. Another method, shown in Fig. 10(b), is to use two registers to check changing data. This is a safe method, but requires the availability of the clock signal, which is not guaranteed in our situation.

In the proposed design, simple and safe combinational logic is used to control the clock halt and restart functions. This structure slightly changes the scheme shown in Fig. 9: clk_wr is stopped when both $full$ and $asyn\_full$ are high, and clk_wr is restarted when either of them goes low. This logic can be simply implemented using an AND gate. The same logic exists in the FIFO empty situation: clk_rd is stopped when both $empty$ and $asyn\_empty$ are high, and it is restarted when either of them goes low.

The simplified clock halt and restart circuit block diagram is shown in Fig. 11. The $fifo\_consist$ and $out\_consist$ signals are discussed in Section IV-E3.

3) The Consistent Signals: Simply using the combination of $full$ and $async\_full$ to stop the clock results in wasted power dissipation in some cases. One example is shown in Fig. 12. As shown in Fig. 12, the producer is stopped due to its *own* FIFO being empty and its clock clk_wr is, therefore, off. When the consumer FIFO becomes $empty$, it is supposed to stop its clock too. However, $asyn\_empty$ in the consumer is controlled by the producer's clock, clk_wr, which is halted. As a result, $asyn\_empty$ stays low and the consumer's clock clk_rd keeps running and wastes power.

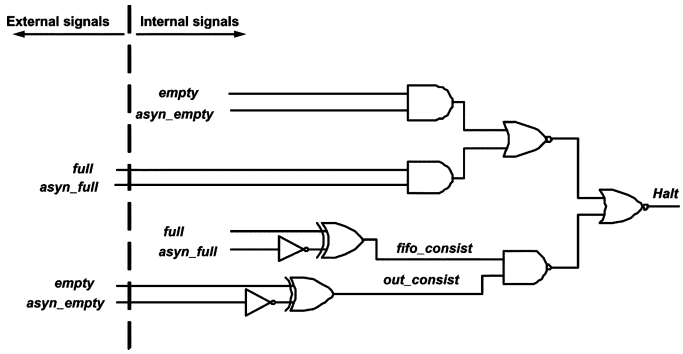


Fig. 11. Simplified clock halting and restarting circuit.

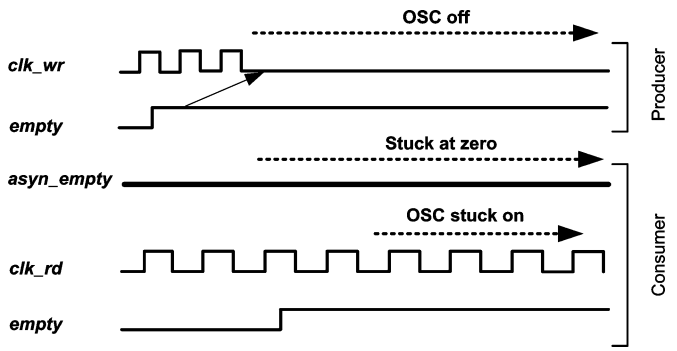


Fig. 12. Example showing clock stopping with a stuck-on consumer's clock before using the *consistent* signal.

The *consistent* signal is added into the clock control circuit to solve this problem, as shown in Fig. 11. For the processor to halt, it must either read an empty FIFO ($empty$ and $asyn\_empty$ from *internal signals*), or write a full FIFO ($full$ and $asyn\_full$ from *external signals*). Stalling also requires the $fifo\_consist$ and $out\_consist$ to be high.

The key point is thus: oscillators must be turned on briefly when critical information is inconsistent in the two clock domains of the FIFO until the inconsistency is resolved.

The signal $fifo\_consist$ is high when both $asyn\_full$ and $full$ from *internal signals* are the same. $out\_consist$ is high when both $asyn\_empty$ and $empty$ from *external signals* are the same. Fig. 13 shows the modified waveform of Fig. 12 after adding the *consistent* signal. When the consumer FIFO becomes empty, $empty$ and $asyn\_empty$ signals do not match therefore $out\_consist$ goes low. As a result, it wakes up the producer's clock, clk_wr which makes $asyn\_empty$ go high. Thus, after a few cycles both producer processor and consumer processor will stop their clocks correctly.

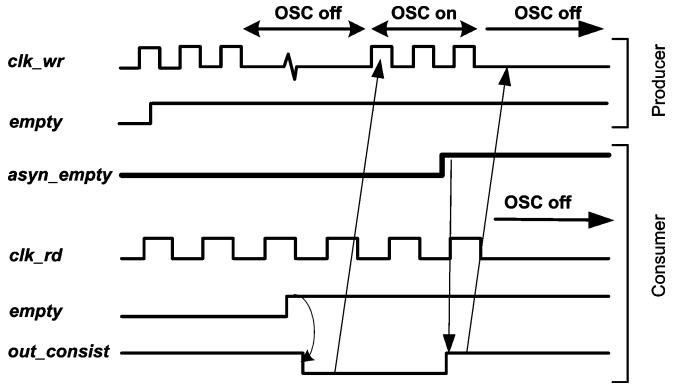


Fig. 13. Example showing clock stopping with a properly stalled consumer's clock oscillator using the *consistent* signal.

V. Hardware Implementations of the Dual-Clock FIFO

Two implementations of the proposed dual-clock FIFO architecture have been built: a full-custom design and a standard cell based design. The standard cell design is used in a multiprocessor GALS chip.

A. Full-Custom Implementation

A 32-word implementation of the proposed dual-clock architecture is designed in a 0.18-μm CMOS technology utilizing a full-custom design approach and scalable design rules.

HSPICE simulation estimates from extracted layout predict the critical path on the write side consists of the Gray-to-binary converter, adder, and a flip-flop. The delays under typical conditions are 535, 515, and 250 ps, respectively. The total minimum cycle time delay is approximately 1.3 ns. The read side's critical path delay is also 1.3 ns. The resulting maximum clock frequency for the entire FIFO is, therefore, approximately 770 MHz.

The final design supports a throughput of one datum per clock cycle up to its maximum clock frequency. This occurs when the consumption and production rates are similar and is not limited by the clock frequency. The minimum latency is an imprecise number since it depends on the phases and frequencies of the read and write clocks and the synchronizer's configurations. With two synchronization registers, latency is bounded to no more than three WRITE clock cycles plus three READ clock cycles—which corresponds to 7.8 ns with both clocks at their maximum frequency in this design.

The layout of the dual-clock FIFO module without global wiring is shown in Fig. 14. It occupies approximately 44 761 μm^2 of active area with a minimum rectangle area of 66 500 μm^2. This area is larger than it otherwise would be for three major reasons: 1) transistor sizes are large and circuits are optimized for high speed, 2) layout is done using relaxed scalable design rules that allow easy portability across many vendors and technology generations, but also significantly increase area, and 3) the layout is a first-generation design and has not been significantly optimized for reduced area. Table IV shows the active areas for the individual components of the FIFO.

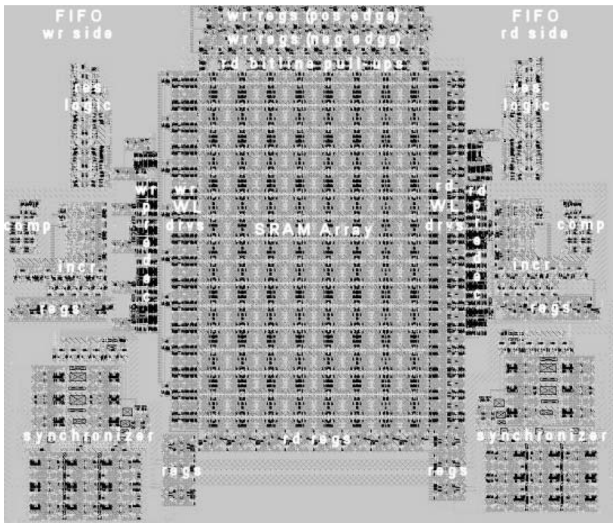


Fig. 14. Final layout for the dual-clock FIFO module.

TABLE IV
Area Breakdown for the FIFO Hardware Module

	Units utilized per FIFO	Active area (μm^2)
SRAM	1	30,507
Synchronizer	2	3,867
Reserve logic	2	908
Comparator	2	300
Gray to binary conv.	2	336
Binary to Gray conv.	2	186
Incrementer	2	1,530
Total		**44,761**

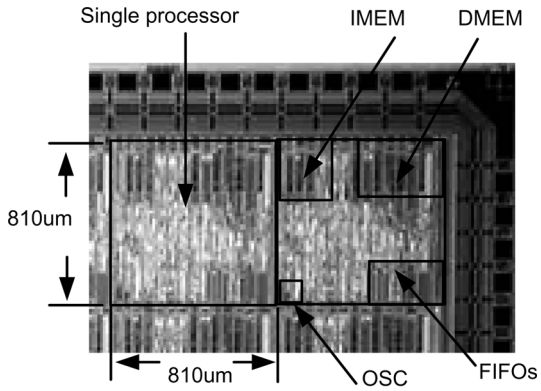
B. Standard Cell Based Implementation

The proposed FIFO is also implemented in 0.18-μm CMOS using standard cells in a GALS multiple processor array [14].

The chip contains multiple simple processors. Each processor is clocked by its local oscillator that is totally unrelated to other processors. Each processor contains two dual-clock FIFOs described in this paper for inter-processor communication. We believe this is the first chip implementing a GALS array processor. Fig. 15 shows a detail of the chip micrograph including two neighboring processors.

The chip is fully functional on first-pass silicon running above 580 MHz at 1.8 V. To test the robustness of the FIFO, several test programs with different configurations of reserve space, delay, and synchronization were created. The test programs utilize groups of five processors, all running at different clock frequencies and arbitrarily halting and restarting. Clock oscillators come cleanly out of their halted state to full rate in less than one clock cycle and present no special challenges to READ and WRITE logic. Measurements verify correct operation over more than 10^{14} FIFO transactions. Combined FIFO READ and WRITE operations consume 10.3 mW at 580 MHz and 1.8 V.

Several applications are mapped onto the GALS chip. A configurable test bit allows disabling of the clock halting method described in this paper. Measurements show the clock halting method reduces power dissipation by 53% and 65% for two complex multiprocessor applications.



Technology	TSMC 0.18 μm
FIFO Area	25,000 μm^2
Max FIFO Speed	580 MHz @ 1.8 V
FIFO Power	10.3 mW @ 580 MHz @ 1.8 V
Energy/Op	17.7 pJ/Op =17.8 μW/MHz

Fig. 15. Chip micrograph of two processors in a GALS chip, where each processor contains two of the proposed dual-clock FIFOs.

VI. Conclusion

The proposed dual-clock FIFO architecture is well suited for many dual-clock applications and achieves high energy efficiency, good scalability and area utilization, high clock rates, and arbitrarily high robustness. This architecture can be utilized as a drop-in module to many applications.

The FIFO is implemented using 0.18-μm standard cell technology and embedded in a GALS array processor. The FIFO occupies 25,000 μm^2 and operates over 580 MHz at 1.8 V, with simultaneous FIFO READs and FIFO WRITEs consuming 10.3 mW under those conditions.

Acknowledgment

The authors would like to thank R. Krishnamurthy, M. Anders, S. Mathew, E. Work, other members of the VCL Laboratory, and Artisan for their support and assistance.

References

[1] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," *Proc. IEEE*, vol. 89, no. 4, pp. 490–504, Apr. 2001.

[2] G. Semeraro and G. Magklis *et al.*, "Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling," in *Proc. Int. Symp. High-Perform. Comput. Arch.*, 2002, pp. 29–40.

[3] D. M. Chapiro, "Globally-asynchronous locally-synchronous systems," Ph.D. dissertation, Dept. Comput. Sci., Stanford Univ., Stanford, CA, 1984.

[4] W. J. Dally and J. W. Poulton, *Digital Systems Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 1998.

[5] M. Balch, *Complete Digital Design*, 1st ed. New York: McGraw-Hill, 2003.

[6] J. Ebergen, "Squaring the FIFO in GasP," in *Proc. Int. Symp. Asynch. Circuits Syst.*, 2001, pp. 194–205.

[7] C. E. Molnar, I. W. Jones, W. S. Coates, and J. K. Lexau, "A FIFO ring performance experiment," in *Proc. Int. Symp. Asynch. Circuits Syst.*, 1997, pp. 279–289.

[8] M. R. Greenstreet, "Implementing a STARI chip," in *Proc. IEEE Int. Conf. Comput. Des.*, 1995, pp. 38–43.

[9] A. Chakraborty and M. R. Greenstreet, "Efficient self-timed interfaces for crossing clock domains," in *Proc. Int. Symp. Asynch. Circuits Syst.*, 2003, pp. 78–88.

[10] J. N. Siezovic, "Pipeline synchronization," in *Proc. Int. Symp. Asynch. Circuits Syst.*, 1994, pp. 87–96.

[11] T. Chelcea and S. M. Nowick, "A low-latency FIFO for mixed-clock systems," in *Proc. IEEE Comput. Soc. Workshop VLSI*, 2000, pp. 119–126.

[12] C. Cummings, "Simulation and synthesis techniques for asynchronous FIFO design," Synopsys Users Group, San Jose, CA, 2002.

[13] R. W. Apperson, "A dual-clock FIFO for the reliable transfer of high-throughput data between unrelated clock domains," M.S. thesis, Dept. Electr. Comput. Eng., Univ. California, Davis, CA, 2004.

[14] Z. Yu, M. Meeuwsen, R. Apperson, O. Sattari, M. Lai, J. Webb, E. Work, T. Mohsenin, M. Singh, and B. Baas, "An asynchronous array of simple processors for DSP applications," in *Proc. IEEE Int. Solid-State Circuits Conf.*, 2006, pp. 428–429, 663.

[15] I. Sutherland, "Micropipelines," *Commun. ACM*, vol. 32, no. 6, pp. 720–738, Jun. 1989.

[16] E. Brunvand, "Low latency self-timed flow-through fifos," in *Proc. Adv. Res. VLSI*, 1995, pp. 76–90.

[17] J. T. Yantchev, C. G. Huang, M. B. Josephs, and I. M. Nedelchev, "Low latency asynchronous FIFO buffers," in *Proc. Asynch. Des. Method.*, 1995, pp. 24–31.

[18] J. M. Rabaey, A. Chandrakasan, and B. Nikolic, *Digital Integrated Circuits, A Design Perspective*. Upper Saddle River, NJ: Prentice-Hall, 2003.

[19] C. J. Myers, *Asynchronous Circuit Design*. New York: Wiley, 2001.

[20] J. Jex and C. Dike, "A fast resolving BiNMOS synchronizer for parallel processor interconnect," *IEEE J. Solid-State Circuits*, vol. 30, no. 2, pp. 133–139, Feb. 1995.

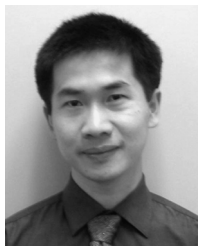
[21] M. Pechoucek, "Anamolous response times of input synchronizers," *IEEE J. Solid-State Circuits*, vol. 25, no. 2, pp. 133–139, Feb. 1976.

[22] C. L. Seitz, "System timing," in *Introduction to VLSI Systems*, C. A. Mead and L. A. Conway, Eds. Reading, MA: Addison-Wesley, 1980, ch. 7.

[23] C. L. Portmann and T. H. Y. Meng, "Metastability in CMOS library elements in reduced supply and technology scaled applications," *IEEE J. Solid-State Circuits*, vol. 30, no. 1, pp. 39–46, Jan. 1995.

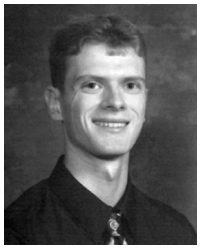
Ryan W. Apperson received the B.S. degree in electrical engineering (*magna cum laude*) from the University of Washington, Seattle, and the M.S. degree in electrical and computer engineering from the University of California, Davis.

He is currently an IC Design Engineer with Boston Scientific CRM Division, Redmond, WA. His research interests include multiclock domain systems and SRAM design.



Zhiyi Yu received the B.S. and M.S. degrees in electrical engineering (with honors) from Fudan University, Shanghai, China. He is currently pursuing the Ph.D. degree in electrical and computer engineering from the University of California, Davis.

He was a key contributor and designer of the 36-processor programmable GALS Asynchronous Array of simple Processors (AsAP) chip. His research interests include high-performance and energy-efficient digital VLSI design with an emphasis on many-core GALS clocking and efficient processor interconnects.



Michael J. Meeuwsen received the B.S. degrees with honors in electrical engineering and computer engineering (both *summa cum laude*) from Oregon State University, Corvallis, and the M.S. degree in electrical and computer engineering from the University of California, Davis.

He is currently a Hardware Engineer with Intel Digital Enterprise Group, Hillsboro, OR, where he works on CPU hardware design. His research interests include digital circuit design and IEEE 802.11 a/g algorithm mapping.

Tinoosh Mohsenin received the B.S. degree in electrical engineering from Sharif University, Tehran, Iran, and the M.S. degree in electrical and computer engineering from Rice University, Houston, TX. She is currently pursuing the Ph.D. degree in electrical and computer engineering from the University of California, Davis.

She is the designer of the Split-Row and Multi-Split-Row Low Density Parity Check (LDPC) decoding algorithms. Her research interests include energy efficient and high performance signal processing and error correction architectures including multi-gigabit full-parallel LDPC decoders and many-core processor architectural design.

Bevan M. Baas (M'99) received the B.S. degree in electronic engineering from California Polytechnic State University, San Luis Obispo, in 1987, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1990 and 1999, respectively.

In 2003, he became an Assistant Professor with the Department of Electrical and Computer Engineering, University of California, Davis. He leads projects in architecture, hardware, software tools, and applications for VLSI computation with an emphasis on DSP workloads. Recent projects include the Asynchronous Array of simple Processors (AsAP) chip, applications, and tools; low density parity check (LDPC) decoders; FFT processors; viterbi decoders; and H.264 video codecs. From 1987 to 1989, he was with Hewlett-Packard, Cupertino, CA, where he participated in the development of the processor for a high-end minicomputer. In 1999, he joined Atheros Communications, Santa Clara, CA, as an early employee and served as a core member of the team which developed the first IEEE 802.11a (54 Mb/s, 5 GHz) Wi-Fi wireless LAN solution. During the summer of 2006 he was a Visiting Professor in Intel's Circuit Research Lab.

Dr. Baas was a National Science Foundation Fellow from 1990 to 1993 and a NASA Graduate Student Researcher Fellow from 1993 to 1996. He was a recipient of the National Science Foundation CAREER Award in 2006 and the Most Promising Engineer/Scientist Award by AISES in 2006. He is an Associate Editor for the IEEE JOURNAL OF SOLID-STATE CIRCUITS and has served as a member of the Technical Program Committee of the IEEE International Conference on Computer Design (ICCD) in 2004, 2005, and 2007. He also serves as a member of the Technical Advisory Board of an early stage technology company.