Efficient Self-Timed Interfaces for Crossing Clock Domains

Ajanta Chakraborty and Mark R. Greenstreet
Department of Computer Science
University of British Columbia
{*chakra,mrg*}*@cs.ubc.ca*

Abstract

With increasing integration densities, large chip designs are commonly partitioned into multiple clock domains. While the computation within each individual domain may be synchronous, the interfaces between these domains often use asynchronous methods. One such approach is the STARI technique[12, 13] where a self-timed FIFO compensates for clock-skew between the sender and receiver. We present implementations of STARI where the FIFO consists of a single, handshaking stage. We start with the simplest case where the sender and receiver operate at exactly the same frequency with an unknown skew. We then generalize this design for links with clocks whose frequencies are rational multiples of each other, clocks whose frequencies are closely matched, and arbitrary clocks. We show that in each of these cases, the STARI interface can exploit the stability of typical clocks to achieve low latencies and negligible probabilities of synchronization failure using very simple hardware.

1 Introduction

Increasing integration densities and clock frequencies drive designers to implement increasing numbers of on-chip clock domains. In this widespread approach to design, circuitry within each clock domain is designed using traditional, synchronous approaches. As tight timing tolerances cannot be guaranteed between timing domains, communication between domains either takes place at a rate slower than the system clock (e.g. one transfer for every two cycles of the clock) or with some kind of mixed synchronous/asynchronous design. This paper presents a family of novel asynchronous circuits for transferring data between clock domains.

Our circuits are based on the STARI [12] approach where a self-timed FIFO is used to compensate for skew between two synchronous systems operating with a common clock. We consider the boundary case where the FIFO consists of a single, handshaking stage, and show that such a design achieves a skew tolerance of nearly two clock periods. We show how this leads to small, low-overhead interfaces. Whereas STARI was originally proposed for designs where the sender and receiver operate at the same clock frequency, we show how our current designs support interfaces where the sender and receiver operate at different frequencies.

Figure 1 shows three typical scenarios where a chip is partitioned into multiple clock domains. In figure 1.a, a single clock drives all timing domains. This is a typical situation in high-performance designs such as microprocessors for general purpose computers. In these designs, clock and data skew [14] arise from a variety of sources [2]. First, scaling trends with decreasing feature sizes decrease gate delays causing a corresponding increase in clock frequencies. For high-performance designs, clock frequencies are further increased by architectural trends favoring deeper pipelines with fewer gates per pipeline stage. Traditionally, designers target clock skews of about 10% of the clock period [21, 27, 20, 16, 18]. Long wire delays and variations in buffer delay make these targets challenging. Accordingly, designers resort to careful layout [1, chapter 7.4] and active skew compensation [33]. Likewise, fabrication engineers reduce wiring delays by deploying copper wires [8] to lower resistance and low-k dielectrics [11] to reduce capacitance. These approaches come at a cost: circuit and layout approaches to lowering clock skew often do so at an increase in circuit complexity and power consumption; improvements in materials are limited by physical constants.

While designers in the asynchronous research community have cited the problems of clock skew for years as a motivation for asynchronous design, synchronous designs are still widespread. In large part, this is accomplished by partitioning designs into small domains within which wire delays and clock skew are small (e.g. [26]) and taking extra measures for communication between clock domains. For example, the Intel Pentium 4 architecture includes pipeline stages whose sole pur-

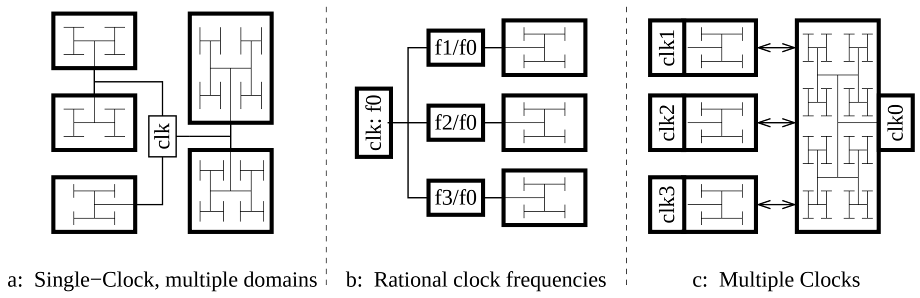

clk
clk: f0
f1/f0
f2/f0
f3/f0
clk1
clk2
clk3
clk0
a: Single−Clock, multiple domains
b: Rational clock frequencies
c: Multiple Clocks

Figure 1. Chips with Multiple Clock Domains

pose is to "drive" data across chip between clock domains [15]. In this paper, we describe asynchronous interfaces between synchronous clock domains that offer high skew tolerance with minimal area and latency overheads.

Figure 1.b depicts a design where different portions of the chip operate at different frequencies, and the various clocks are derived from a common source. This scenario is common in system-on-chip designs where different IP blocks operate at different clock frequencies or in multi-rate digital-signal-processing designs. Although different domains operate at different frequencies, these frequencies are all rational multiples of each other, and these ratios are typically known in advance or are determined by pre-designed operating modes of the design. In this case, we can take advantage of knowing the exact relationship between the various clocks to implement an interface that operates with low latency and without synchronization. Section 6 describes our design for this scenario.

Finally, figure 1.c shows a design where different portions of the chip operate with clocks from independent sources. This occurs, for example, in the design of network routers [19] where each line card receives a bit stream with an embedded clock from a different source. Although each stream comes with its own clock, typically these clocks are very closely matched in frequency. For example, ATM standards specify that bitrates be within one part per million of their nominal values. In section 7, we show that we can adapt our simple interface to exploit this close matching and provide robust data transfer without incurring synchronization overhead in the latency critical path. In section 8, we generalize this solution to designs with arbitrary clocks.

The essential observation behind our designs is that clocks for synchronous systems are designed to be extremely stable. Thus, we can design STARI style interfaces that provide moderate amounts of skew tolerance and dynamically compensate for any long-term drift. As described above, we present designs when the sender and receiver operate at exactly the same frequency; at frequencies that are rational multiples of each other; at closely matched frequencies; and at arbitrary, relatively stable frequencies. In the remainder of the paper, we show that these designs are small and can operate at high clock frequencies with low latencies. We first summarize existing techniques for communication between clock domains.

2 Related Work

Given the challenges of transmitting signals between clock domains, researchers have explored a variety of asynchronous solutions. These range from building completely asynchronous chips [22, 10, 28] to various combinations of synchronous and asynchronous modules in the same design. We focus on the latter approach in this paper. The various methods for combining asynchronous and synchronous modules vary according to the strength of the assumptions that they make about the clock. At one extreme, GALS (i.e. "Globally Asynchronous, Locally Synchronous") designs make very minimal assumptions about clock timing, effectively turning the clock into a bundled completion signal and adding handshaking to clock generation. At the other extreme, "mesochronous" and "plesiochronous" methods rely on exact frequency matching by having a common clock source or on long term frequency stability of clocks [23]. Between these two extremes, synchronizing buffers allow each domain to operate with its own, stable clock, but make minimal assumptions about the relationships between these clocks. Our approaches fall squarely in the mesochronous and plesiochronous camps. To put our designs in context, we summarize these approaches below.

2.1 GALS

As originally proposed by Chapiro [5], GALS (i.e. "Globally Asynchronous, Locally Synchronous") designs use stoppable clocks to allow synchronous modules to communicate using asynchronous protocols. Each synchronous domain has its own clock generator that consists of a ring oscillator with a handshaking stage. When domain X has a value to send to domain Y, X outputs the value, sends a request to Y and stalls its clock until it receives an acknowledgment. Likewise, when domain Y is prepared to receive a value from domain X, it stalls its clock, waits for a request from X, latches the value, sends an acknowledgment to X and restarts its own clock.

Yun and Donohue [36] extended Chapiro's approach by adding a mutual-exclusion element to the ring oscillator. This allows each locally synchronous block to continue operating while polling for input from its neighbors. The mutual exclusion element delays the next clock event, if needed, to allow metastability [4] arising from the polling to resolve. Yun and Donohue's approach allows GALS designs to be very flexible, and their methods have been extended by several research groups, e.g. [25, 31, 24, 29].

GALS makes minimal assumptions about clock stability; in fact, GALS discards the stability and low-jitter of clocks that are the hallmarks of synchronous design. The frequency stability of traditional clocks allows us to determine the relative phase of two independent clocks thousands or more of cycles in advance. As we describe in sections 7 and 8, this predictability allows us to move metastability off of the latency critical paths in our designs. Jitter is the variation in the time between successive clock events. This variation directly degrades the performance of synchronous designs. and clock pausing exacerbates jitter. After pausing a clock, the first edge through the ring oscillator and clock buffer will propagate slower than subsequent events [34]. The loss of long-term timing predictability and the increase of jitter are consequences of the GALS approach of converting synchronous designs into asynchronous ones by adding handshaking control to their clock generators. In this paper we show that more efficient designs are achieved by letting synchronous modules be synchronous and using simple asynchronous interfaces to compensate for clock-skew and other timing uncertainties.

2.2 Synchronizing Buffers

The next step in our taxonomy allows independent, free-running, stable clocks in each domain, and makes minimal assumptions about the timing relationships between them. A common rule-of-thumb for design specifies the use of two or three synchronizing latches whenever a clock domain is crossed [17, chapter 3.11.4]. For high-performance designs with a small number of gate-delays per clock period, even longer chains may be needed to achieve acceptably low probabilities of failure. Seizovic [30] recognized that these synchronizations can be pipelined allowing high throughput even when the time for reliable synchronization is many clock periods. Chelcea and Nowick [7] further optimized this approach by noting that synchronizations are only needed for the receiver when the buffer is close to empty and only needed for the sender when the buffer is close to full. All of these approaches incur the worst-case synchronization latency when the buffers are nearly empty. Iyer and Marculescu [16] evaluated the performance of a superscalar microprocessor design decomposed using Chelcea and Nowick's FIFOs. Superscalars are particularly sensitive to latency, and Iyer and Marculescu found that the performance penalties arising from the added latency outweighed the power savings for the design that they considered.

2.3 STARI

The synchronization latency of the designs described above can be reduced or eliminated by taking advantage of the stability of clocks. This stability allows us to predict the timing relationship of clocks in different domains well into the future. Mesochronous designs (a.k.a. "STARI" [12, 13] or "source-synchronous" [35]) have a common clock source for the sender and receiver, guaranteeing that both operate at the same frequency although the phase difference between the two may be unknown. A FIFO at the receiver is initialized to be roughly half full. During each clock period, the transmitter inserts one item into the FIFO and the receiver removes one value from the FIFO. The FIFO occupancy remains within one of half-full; in particular, overflows and underflows are excluded. This removes the need for testing full and empty conditions and thereby removes the need for synchronization and synchronizers.

The designs presented in the paper are both specializations and generalizations of the original STARI work. In sections 3 and 5 we specialize STARI by focusing on the case where the FIFO consists of a single stage. We show that such implmentations can provide nearly two clock periods of skew tolerance. By optimizing for the single-stage case, we obtain very simple interfaces between the edge-triggered conventions common in synchronous design and the handshaking communication that is characteristic of self timed designs. We then generalize STARI to relax the requirement of exactly matched clocks at the sender and receiver. We present interfaces where the sender and receiver clock frequencies are rational multiples of each other 6, closely matched 7, and arbitrary 8. In all of these designs, we exploit the long-term stability of clocks to obtain simple interfaces with small latencies.

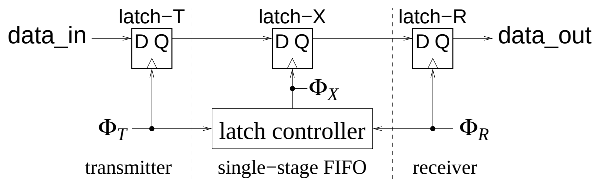


Figure 2. The Single Stage FIFO

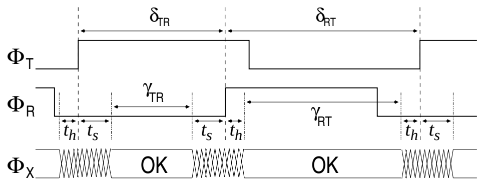
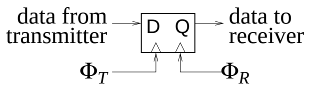


Figure 3. Clock Timing for the FIFO

3 The Single-Stage FIFO

We now describe a simple interface circuit for STARI communication when the FIFO has a single stage. As shown in figure 2, the FIFO consists of a single latch, and a latch controller that generates a clock for this latch based on the clocks from the transmitter and receiver. To the user, this FIFO appears as a latch with two clock inputs:



In this section and the next, we assume that the transmitter and receiver operate at *exactly* the same frequency, only the relative phase difference is unknown. This is easily achieved if both of their clocks are derived from a common source.

Figure 3 depicts the timing for the single-stage FIFO. For simplicity, we assume that latch-T, latch-X and latch-R are positive edge triggered. Our design easily generalizes to other latching styles. For proper operation, the latch controller must generate Φ_X so as to satisfy the set-up and hold requirements of latch-X and latch-R. To satisfy the requirements of latch-X, the rising edge of Φ_X must occur at least $t_{set-up} + t_{prop}$ (abbreviated t_s in the figure) after the previous Φ_T event, and at least $t_{hold} - t_{prop}$ (abbreviated t_h in the figure) before the next Φ_T event, where t_{set-up}, t_{hold}, and t_{prop} denote the set-up and hold times and propagation delay of the latches respectively. To satisfy the requirements of latch-R, the rising edge of Φ_X must occur at least $t_{hold} - t_{prop}$ after the previous Φ_R event, and at least $t_{set-up} + t_{prop}$ before the next Φ_R event. The "exclusion" regions corresponding to these requirements are indicated by cross-hatched regions in figure 3.

There are two windows of opportunity for generating Φ_X: a rising edge of Φ_X may occur between a rising edge of Φ_T and the subsequent rising edge of Φ_R, or between the rising edge of Φ_R and the subsequent rising edge of Φ_T. We refer to these scenarios according to the *last* event (Φ_T or Φ_R) that occurs prior to each Φ_X event. Thus, if Φ_X occurs after a Φ_T event but before the next Φ_R event, we refer to this situation as "transmitter-last," and we write "receiver-last" to refer to the other case. In the figure, δ_{TR} denotes the time from the rising edge of Φ_T to the next rising edge of Φ_R. Likewise, δ_{RT} denotes the time from the rising edge of Φ_R to the next rising edge of Φ_T. Let P denote the clock period. Now, let γ_{TR} denote the width of the window of opportunity for the transmitter-last scenario, and γ_{RT} denote the width of the window of opportunity for the receiver-last case. We have:

$$\begin{array}{rrcl} & \gamma_{TR} & = & \delta_{TR} - 2(t_{set-up} + t_{prop}) \\ & \gamma_{RT} & = & \delta_{RT} - 2(t_{hold} - t_{prop}) \\ \Rightarrow & \gamma_{TR} + \gamma_{RT} & = & \delta_{TR} + \delta_{RT} - 2(t_{set-up} + t_{hold}) \\ & & = & P - 2(t_{set-up} + t_{hold}) \\ \Rightarrow & \max(\gamma_{TR}, \gamma_{RT}) & \geq & P/2 - (t_{set-up} + t_{hold}) \end{array} \tag{1}$$

In other words, if the clock period is greater than $2(t_{set-up} + t_{hold})$, then the window of opportunity for at least one of the transmitter-last or the receiver-last case is non-empty, and the latch-controller can generate a clock that ensures proper operation of the interface. In particular, if $\gamma_{TR} > 0$, then the latch controller can safely generate a rising edge $t_{set-up} + t_{prop}$ after the rising edge of Φ_T; otherwise; γ_{RT} must be positive, and the latch controller can safely generate a rising edge $t_{hold} - t_{prop}$ after the rising edge of Φ_R. In section 5 we show how the latch controller can be initialized to operate in one of these two scenarios. For the remainder of this section we will consider steady state operation.

The latch controller can be implemented by a C-element. To see this, first consider operation in a transmitter-last scenario with $\gamma_{TR} > 0$. Following each rising edge of Φ_T event, the latch controller outputs a corresponding rising edge for Φ_X. Then, there will be the next rising event for Φ_R followed by a rising event for Φ_T before the controller outputs the next rising edge of Φ_X. Conversely, for the receiver-last case, following each rising edge of Φ_X event, the controller sees a rising event for Φ_T followed by a rising event for Φ_R event before generating the next rising edge for Φ_X. In either case, between producing consecutive rising edges of Φ_X, the latch controller receives rising edges from *both* Φ_T and Φ_R.

In our design, timing is determined by the rising edges of the clocks. Accordingly, we use a self-resetting [6, 32] implementation as shown in figure 4.

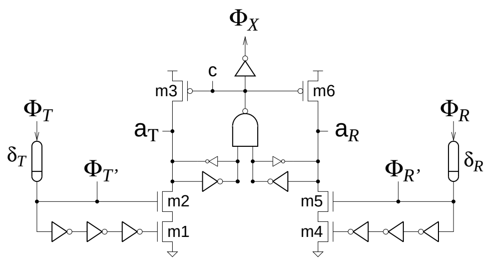


Figure 4. The Latch Controller

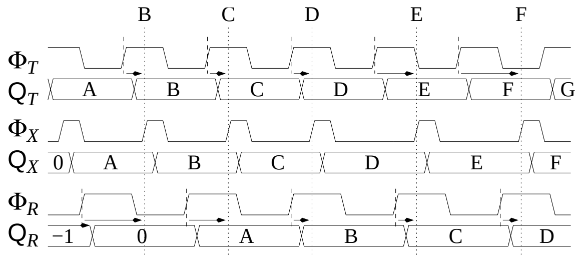


Figure 5. Drifting Skew

On a rising edge of Φ_T, transistors m1 and m2 pull node a_T low. The three-inverter chain to the gate of m1 disables the pull-down path shortly after Φ_T has gone high to make the circuit edge-sensitive rather than level-sensitive. Likewise, node a_R drops on a rising edge of Φ_R. When both have dropped, node c goes low, resetting the edge-triggered C-element and generating a pulse on Φ_X. Delay δ_T ensures that the delay from a rising edge of Φ_T to a rising edge of Φ_X is greater than $t_{set-up}+t_{prop}$. Likewise, delay δ_R ensures that the delay from Φ_R to Φ_X is greater than $t_{hold}-t_{prop}$. As promised, our design is extremely simple and requires very little layout area.

4 Skew Tolerance

To analyze the skew tolerance of our design, we start with a transmitter-last scenario; proper operation requires $\gamma_{TR} > 0$ which is equivalent to $\delta_{TR} > 2(t_{set-up} + t_{prop})$. If the initial time difference, δ_{TR0}, is greater than this value, then the transmitter may be further delayed by up to $\delta_{TR0} - 2(t_{set-up} + t_{prop})$ without malfunction of the interface.

Figure 5 shows what happens starting from a transmitter-last scenario where transmitter events occur progressively earlier due to drift in the skew. In this figure, the transmitter outputs the sequence of values: $[-1, 0, \mathrm{A}, \mathrm{B}, \mathrm{C}, \ldots]$ on node Q_T. The values shown for Q_X, and Q_R show how transmitter data is propagated to the other two latches. For each Φ_X event, the figure shows a vertical dotted line labeled with the value loaded into latch-X by that event, and with arrows from Φ_T and Φ_R events showing the two events that triggered the latch controller. The rising edges of Φ_X for values B and C are transmitter-last events; for value D, the Φ_T and Φ_R events are coincident; and for values E and F, Φ_X is generated by receiver-last events. In all cases, the latch controller waits until it has received events on *both* inputs. The relative order of arrival of the rising edges of Φ_T and Φ_R doesn't matter; thus, no synchronization is necessary.

When the Φ_T event precedes the Φ_R event, then the interface operates with the receiver last, starting with $\delta_{RT} = P$. The interface continues to operate without dropping a value as long as $\delta_{RT} > 2(t_{hold} - t_{prop})$. Starting with an initial time difference of δ_{TR0}, transmitter events can occur up to $P - \delta_{TR0}$ time units earlier before the receiver-last scenario occurs. At this point $\delta_{RT} = P$, and the transmitter can occur up to $P - 2(t_{hold} - t_{prop})$ time units earlier without malfunction.

To summarize, if the interface starts in a transmitter-last scenario with $\delta_{TR} = \delta_{TR0}$, then the Φ_T can be delayed with respect to Φ_R by up to $\delta_{TR0} - 2(t_{set-up} + t_{prop})$ time units, and it can be advanced by up to $2P - \delta_{TR0} - 2(t_{hold} - t_{prop})$ time units without malfunction of the interface. The total width of the interval of relative delays for which the interface operates correctly is $2(P - t_{set-up} - t_{hold})$. Equivalent arguments hold starting from a receiver-last scenario. If the latch set-up and hold window is small relative to the clock period, then our design offers nearly two clock periods of skew tolerance.

In this section, we have described necessary and sufficient conditions to ensure that the set-up and hold requirements for latch-X and latch-R are satisfied. In addition to these requirements, node c in the latch controller must return high when an Φ_X pulse is generated before the arrival of the next rising edge on Φ_T or Φ_R. Let η be the time between triggering the latch controller and the subsequent return of node c to a high value. Let $\delta_{T'R'}$ and $\delta_{R'T'}$ denote the time from a rising edge of $\Phi_{T'}$ to the next rising edge of $\Phi_{R'}$ and vice-versa. As explained in [3], proper operation with the transmitter last requires $\delta_{T'R'} > \eta$, and proper operation with receiver last requires $\delta_{R'T'} > \eta$. At least one of the two modes is feasible if

$$P > 2\eta \tag{2}$$

For our proof-of-concept test-chip [3], our latch set-up and hold windows were significantly smaller than the latch controller's cycle time. Thus, these cycle time constraints are the critical ones for our design.

5 Initialization

Under many circumstances, γ_{TR} and γ_{RT} are both positive. When this occurs, the interface can operate in either transmitter-last or receiver-last mode. This section describes two ways to decide which mode is "better" and initialization procedures to achieve the desired mode. Throughout this section, we assume that it is acceptable for the interface to drop values, duplicate values, and/or exhibit metastable behavior during initialization. Of course, it must deliver data without error after completion of initialization.

5.1 Maximum Robustness

Clock jitter, temperature drift, and other fluctuations cause the skew on physical chips to vary while the chip is operating. Typically, this variation is just as likely to make the transmitter earlier as it is to make it later. Thus, to maximize robustness to skew variation, we want to choose the mode tolerates the largest skew change in either direction. This corresponds to starting in the transmitter-last mode if $\delta_{TR} > \delta_{RT}$ and in the receiver-last mode if $\delta_{TR} < \delta_{RT}$.

An easy way to achieve this is to insert an adjustable delay into the self-reset cycle of the latch-controller. If this delay is initially very large, then neither mode is feasible and the latch-controller will generate ill-timed clock signals. By gradually decreasing this delay, the circuit will reach a point where exactly one of the two modes is feasible and after one or two cycles the latch controller will operate stably in that mode. As the delay is further decreased, the latch controller will remain in the first mode that became feasible. This is the mode with the larger skew margin. Thus, the analog dynamics of our circuit provide a very simple mechanism for initialization.

A metastable situation can occur if $\delta_{T'R'} \approx \delta_{R'T'} \approx P/2$: there is a metastable periodic behavior separating the periodic attractors corresponding to the "transmitter-last" and "receiver-last" modes. The probability of remaining in this metastable situation drops exponentially in time with the keeper inverters for nodes a_T and a_R providing the feedback. To ensure robust operation, an implementation should either use a "strong keeper" circuit for these inverters or allow extra time for initialization.

On our proof-of-concept chip, we implemented this form of initialization. The variable delay is achieved by modulating the ground potential for circuits in the self-resetting C-element. See [3] for details.

5.2 Minimum Latency

When both transmitter-last and receiver-last modes are feasible, the transmitter-last mode has a latency that is one clock period less than that of the receiver-last mode. For designs where latency is critical for performance (e.g. [9, 16]), it may be desirable to select transmitter-last whenever possible. The following initialization procedure achieves this behavior:

1. Start the interface running at full-speed (no need for the speed adjustment used in section 5.1). The latch controller will settle into one of its two modes.

2. Wait long enough to ensure that the probability of metastability failures is insignificant.

3. Suppress one transmitter clock event. If the latch controller had been in the transmitter-last mode, it will now see two receiver events before the next transmitter event and continue in transmitter-last mode. On the other hand, had it been in the receiver-last mode, the latch controller will see one receiver event before the next transmitter event and switch to transmitter-last. If $\delta_{T'R'} > \eta$, then the controller will remain in transmitter-last, otherwise it will miss a receiver event when the controller's internal reset completes after the arrival of a rising edge of $\Phi_{R'}$ and then resume operation in receiver-last.

4. Allow adequate time for the resolution of metastability that can occur if the controller falls back into receiver-last mode.

As described, this procedure make no guarantees of robustness when forcing the transmitter-last mode. To provide some robustness against skew drift clock jitter, the latch controller can be operated with a slight slow-down during this initialization and brought to full speed under normal operation. Alternatively, section 7 describes a (near-) miss detector circuit that can detect when the controller is close to its limits; in which case, the controller can be returned to the receiver-last mode by suppressing a Φ_R event.

6 Rational Clock Frequency Multiples

We now consider the situation depicted in figure 1.b: the frequencies of the sender's and receiver's clocks are pre-determined rational multiples of each other. Let P_T be the period of the transmitter's clock and P_R be the period of the receiver's clock. Let N_R and N_T be positive and mutually prime with $N_R/N_T = P_T/P_R$ (N_T and N_R correspond to the frequencies of the respective clocks).

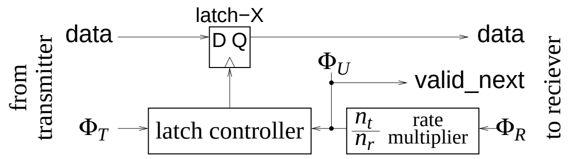


Figure 6. An Interface with Rational Clocks

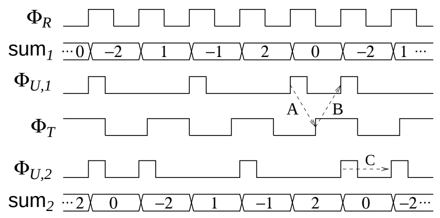


Figure 7. Exploiting periodic jitter

We develop our designs assuming $N_R > N_T$ and describe the $N_T > N_R$ case at the end of this section.

Figure 6 shows our design for $N_R > N_T$. By the assumption that the receiver operates at a higher rate than the transmitter, there will be receiver cycles for which no new transmitted data is available. The rate multiplier outputs N_T pulses on node Φ_U to the latch controller for every N_R cycles of the receiver's clock as shown below:

```
sum := 0;
for each cycle of $\Phi_R$ do
   if sum $\geq$ 0 then
      output a pulse on $\Phi_U$;
      sum := sum + $N_T - N_R$;
   else sum := sum + $N_T$;
   endif
od
```

By analogy with Φ_R and Φ'_R, let $\Phi_{U'}$ be the internally delayed version of Φ_U in the latch controller, and let $\delta_{T'U'}$ and $\delta_{U'T'}$ be defined as $\delta_{T'R'}$ and $\delta_{R'T'}$ respectively. The rate multiplier introduces periodic jitter into Φ_U with a period of $N_R P_R = N_T P_T$. Let $\delta_{U'T'0}$ denote the time from the rising edge of $\Phi_{U'}$ produced by the $(kN_R)^{th}$ rising edge of Φ_R to the next rising edge of $\Phi_{T'}$ for any integer k. It is straightforward to show:

$$\begin{array}{rcl} \min(\delta_{U'T'}) & = & \delta_{U'T'0} \\ \max(\delta_{U'T'}) & = & \delta_{U'T'0} + P_R - \frac{P_R}{N_T} \\ \min(\delta_{T'U'}) & = & P_T - \delta_{U'T'0} - P_R + \frac{P_R}{N_T} \\ \max(\delta_{T'U'}) & = & P_T - \delta_{U'T'0} \end{array} \quad (3)$$

The cycle time constraints of the latch controller can be satisfied if:

$$\max(\delta_{U'T'0}, P_T - \delta_{U'T'0} - P_R + \frac{P_R}{N_T}) \quad > \quad \eta \quad (4)$$

which holds for any value of $\delta_{U'T'0}$ if:

$$P_T - (1 - \frac{1}{N_T})P_R \quad > \quad 2\eta \quad (5)$$

For designs where the latch set-up and hold requirements are the dominant constraints, similar bounds for P_T and P_R can be derived. Thus, our one-stage FIFO can be used to interface to synchronous domains operating at different, rationally related frequencies. Furthermore, the initialization methods described in section 5.1 apply directly to the rational clocks case. Comparing with equation 2 shows that the minimum clock period has been increased by the jitter of Φ_U created by the rate multiplier.

As noted above, the jitter introduced by the rate multiplier is periodic. We can exploit this predictability to increase the robustness of the interface. For every N_R consecutive cycles of the receiver's clock, the variable sum takes on each value in $\{N_T - N_R, \ldots, N_T - 1\}$ exactly once. The initial value of sum is arbitrary, and we can use this freedom to increase the skew tolerance of our design.

Figure 7 shows the operation of our interface where the transmitter clock frequency is $3/5$ that of the receiver. The traces for sum$_1$ and $\Phi_{U,1}$ show the worst-case sequence for sum: with this choice $\eta < P/2$ must hold for proper operation. In particular, if the Φ_U event generated when sum$_1$ transitions from 2 to 0 triggers the latch controller to produce a Φ_X event, then the self-reset cycle must complete in time for the rising edge of Φ_T that occurs $P_R/2$ later (indicated by the arrow labeled A in the diagram). On the other hand, if this Φ_U event does not trigger the latch controller, then the subsequent Φ_T event must, and the resulting self-reset cycle must complete in time for the next Φ_U event, again $P_R/2$ later (indicated by the arrow labeled B in the diagram).

The traces for sum$_2$ and $\Phi_{U,2}$ show the optimal sequence for sum for the same transmitter and receiver clocks. For this scenario, the critical timing occurs when the rising edge of Φ_U that is produced when sum$_2$ goes from 2 to 0 triggers a Φ_X pulse. The self-reset of the latch controller must complete prior to the next Φ_U pulse P_R time units later. Thus, with this choice of the sum sequence, the latch controller can operate at half the rate as required by the worst-case sequence. We first derive the constraints on P_T and P_R that ensure proper operation for any phase difference between the two clocks assuming the optimal sum sequence. We then describe how our initialization technique from section 5.1 can be adapted to find this sequence.

Regardless of how the sum sequence is chosen, the Φ_U clock has a jitter of $(1 - 1/N_T)P_R$ with respect to an evenly spaced clock with period P. The maximally robust sequence for sum centers the Φ_U jitter interval as closely as possible on the Φ_T clock. The interval may be off center by as much as $\gcd(P_R, P_T) = P_R/N_R$ due to the discrete set of choices for the sum sequence. From these observations, the smaller of the time from a rising edge of $\Phi_{T'}$ that triggers the latch controller to the next rising edge of $\Phi_{U'}$ or vice-versa is $P_T - P_R/2$. It is also possible that a rising edge of $\Phi_{U'}$ triggers the latch controller and have the next input event for the controller be the next rising edge of $\Phi_{U'}$. The minimum time between two such rising edges is $\lfloor N_R/N_T \rfloor P$. Combining these two constraints yields that there is a feasible sequence for sum such that the cycle time constraints of the latch controller are satisfied as long as:

$$P_T - \max\left(\frac{1}{2}, \frac{N_R \bmod N_T}{N_T}\right) P_R \quad > \quad \eta \qquad (6)$$

Comparing with equation 5 we see that choosing the optimal sum sequence can greatly relax the cycle time requirement for the latch controller, or, equivalently, greatly increase the robustness of the interface. For example with $P_R = 1ns$ and $P_T = 1.2ns$, equation 5 (fixed choice for the sum sequence) requires $\eta < 0.1$ns. With the optimal choice for the sum sequence, equation 6 requires $\eta < 0.7$ns, a reduction in the speed required by a factor of 7 for this example.

The optimal sum sequence can be selected as part of the initialization of the interface. We base our approach on two observations. First, the optimal sequence works with a larger value of η than any other sequence. Second, we can shift from one sequence to another by adding $N_T + 1$ (resp. $N_T - N_R + 1$) to sum instead of N_T (resp. $N_T - N_R$). Generalizing the initialization technique described in 5.1, we start with a large value for η and gradually decrease it. Each time the latch-controller fails to reset in time for the next $\Phi_{T'}$ or $\Phi_{U'}$ event, we shift to the next sum sequence. When η is small enough that the latch controller can operate with the optimal sum sequence, but not the others, then the rate multiplier will switch from one sequence to the next until it reaches the optimal one. At this point, the latch controller will successfully reset after each cycle in time for the next $\Phi_{T'}$ and $\Phi_{U'}$ events, and the rate multiplier will remain with the optimal sequence.

Figure 8 shows our circuit that reports when a rising edge of Φ_T or Φ_U arrives at the latch controller prior to the completion of the controller's internal reset. We call such an event a "miss" and call our circuit a "miss detector." A miss occurs if a rising edge is received while the c signal of the latch controller (see figure 4) is still low. Noting that Φ_X is an inverted version of c, we use the Φ_X signal in the series stacks of transistors that detect such events. The delay of the inverter that produces Φ_X gives our circuit a little extra margin: it also reports

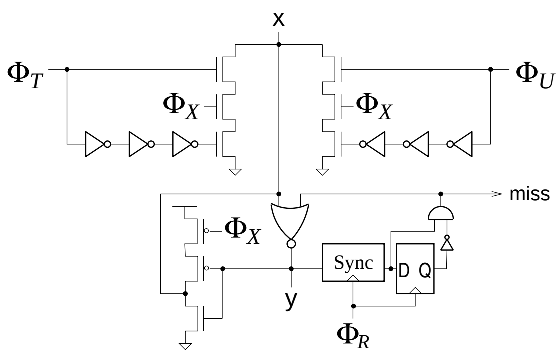


Figure 8. A Miss Detector

"near misses." We use this feature in the next section. When a (near) miss occurs, node x goes low and node y goes high. These transitions occur asynchronously with respect to Φ_R. The synchronizer provides a delayed version of y in the receiver's clock domain. The synchronizer is only active during initialization and does not contribute to the latency of data transfers under steady-state operation, accordingly, the synchronizer can have a large latency and correspondingly minuscule probability of failure.

In the designs shown in this section, the rate multiplier tells the receiver when it can take data from the interface. In other words, the interface takes the active role in flow control. It is straightforward to convert these designs to ones where the interface is passive and the receiver is active by adding a FIFO and a little control logic. This FIFO and control logic function entirely in the receiver's clock domain and add only the delay of a multiplexer to the latency of the interface.

This section has shown how our simple interface based on a one-stage, self-timed FIFO can be used in designs with multiple, rationally related clock frequencies. We have focused on the case where the receiver clock frequency is greater than that of the transmitter. If the transmitter has the higher clock frequency, equivalent designs can be used with the rate multiplier in the transmitter's clock domain.

7 Plesiochronous Interfaces

We now consider the designs with multiple clock domains with independent clocks that are closely matched in frequency; these are called "plesiochronous" interfaces (see [23]). Such designs occur, for example, when the sender and receiver are physically separated (e.g. networks), or when separate clock generators are used to avoid introducing a single point of failure into the design. Typically, the clock frequencies will be matched to within a few parts per million, a tolerance that is easily achieved with crystal oscillators.

With close frequency matching, the relative timing of clock edges at the latch interface changes very slowly. We can modify the miss detector circuit from figure 8 to provide an output indicating when a rising edge from Φ_T shortly after the latch controller completes its reset, and another output that indicates when the timing of Φ_R is close to the margin. If these signals indicate when only $0.1P$ of margin remains, thousands of cycles remain before an error could actually occur. Thus, we can synchronize these signals to the transmitter and receiver clocks with extremely high reliability, and use the synchronized versions to take appropriate corrective action. For example, if a rising edge of Φ_R occurs less than $0.1P$ after the latch controller completes its reset, then the receiver can skip clocking the latch controller on a subsequent cycle. This will switch the interface from operating with the rising edge of the transmitter's clock arriving much after the corresponding edge of the receiver's clock to operation where the receiver edge arrives slightly after the transmitter edge. Likewise, if a rising edge of Φ_T occurs less than $0.1P$ after the latch controller completes its reset, then the transmitter can skip sending data and clock on a subsequent cycle. Such protocols are commonly implemented using "stuff bytes" [9], and are easily implemented in the framework of our latch controller and miss detector.

Although synchronizations are required during operation, the latency of these synchronizations is not critical for the latency of the data path. The data latency for our interface is always less than $2P$. By adding an arbiter to detect when the latch controller is in receiver-last mode with enough margin to be able to safely switch to transmitter last, the worst-case data latency can be reduced to slightly greater than P with an average latency slightly greater than $P/2$.

8 Arbitrary Clock Frequencies

We now consider the case where the transmitter and receiver operate with independent clocks at arbitrary frequencies. Initially, it might seem that such a design requires the overhead of synchronizing buffers as described in section 2.2. However, we can take advantage of the fact that in nearly all synchronous designs the clock frequencies are extremely stable. We can exploit this stability even if the frequencies aren't known in advance. We combine our designs from the previous two sections to support communication with arbitrary clock frequencies.

Firstly, the transmitter and receiver forward their clocks to each other. Each uses a counter to produce an initial estimate of the clock frequency of the other. This assumes that each can implement a counter that operates at least as fast as the other clock. Since we are focusing on single-chip designs, this seems reasonable, and we note that with frequency prescaling, this assumption could be removed. These estimates provide a rational approximation of the ratio of two clock frequencies. If the nominal clock frequencies are known in advance, this step can be skipped.

Secondly, if the receiver's clock frequency is higher than that of the transmitter, it uses a rate-multiplier to create an approximation of the transmitter's clock. Likewise, the transmitter uses a rate multiplier if the transmitter has the higher clock frequency. We operate the latch controller with the (possibly rate-multiplied) clock provided by the transmitter and receiver.

Because the frequency values that we have for the two clocks are only approximations, albeit very accurate ones, the FIFO will be prone to occasional underflow or overflow. We use separate "near-miss" signals for Φ_T and Φ_R and forward near miss events to the client with the faster clock, i.e. the one using the rate-multiplier. This client updates its estimate of the other client's clock frequency thus changing the rate of events output by its rate multiplier. This is a second-order control system, and a little bit of care is needed to ensure stability. A simple approach is that the client with the faster clock uses a counter to measure the time between near miss events and uses this information to update its estimate of the other client's clock frequency. This process is quadratically convergent and stable. At the same time as updating the frequency estimate, a first-order correction can be applied by adding an offset to sum to bring the latch controller back to a point near the center of its safe operating region. If the near miss was for the clock of the client with the faster clock, then this offset should be positive, otherwise it should be negative.

As for the plesiochronous interface described in the previous section, synchronizations are required during operation. Again, the latency of these synchronizations is not critical for the latency of the data path. These synchronizations are infrequent; their rate is determined by the resolution of the rate-multiplier and the drift rate of the clock frequencies.

9 Conclusion

We have presented a very simple design for source-synchronous communication. It is based on a self-timed, ripple FIFO with a single stage. Whereas a single stage pointer FIFO provides no skew compensation (such a pointer FIFO is simply a latch clocked by the transmitter), the single-stage ripple FIFO provides nearly two clock periods of skew tolerance and can operate correctly for any initial phase offset between the transmitter and the receiver of the channel. The simplicity of the single-stage FIFO enables simplifications and optimization, thus taking good advantage of self-timed design. We presented a design consisting of a self-resetting, edge-triggered C-element that generates a clock intermediate to the clocks of the transmitter and receiver. This intermediate clock strobes a latch that conveys data from the transmitter to the receiver. The timing of this clock signal ensures that the set-up and hold requirements of the receiver and the intermediate latch are all satisfied. Our design can be initialized to provide maximal robustness against clock jitter and skew drift by adjusting the speed of the self-resetting C-element during its initial operation. Alternatively, the interface can be initialized for minimum latency by deliberately suppressing a transmitter clock event to the latch during initialization.

Sections 6 through 8 showed how this design can be adapted for more generalized clocking scenarios including clocks with rationally related frequencies, closely matched clocks, and arbitrary clocks. In all of these designs, any synchronization is carried out on a path whose latency does not impact the data path. Thus, our designs can achieve latencies that are at most slightly more than one clock period and typically about half that. To achieve this performance, we exploit the medium to long-term frequency stability of clocks in synchronous designs.

In general, the overheads of synchronization and handshaking are only needed to address timing issues that cannot be resolved statically. When the clocks of the transmitter and receiver are identical in frequency, then only the relative phase needs to be resolved, and this can be done by a simple handshaking circuit such as our latch controller shown in figure 4. When the clocks are rationally related, the client with the faster clock can use a rate multiplier to construct an approximation of the other client's clock. There are numerous isomorphic sequences of events that can be generated by the rate multiplier, and we use synchronization during initialization to determine the optimal sequence. When the clocks are closely matched in frequency, only the long-term drift needs to be identified. The synchronizer that detects this drift can have high latency without impacting the data path latency. Finally, when arbitrary clock frequencies are used, we can still exploit the frequency stability of these clocks. Again, synchronization is only needed to detect long-term drift, and this does not impact the data path latency.

We have designed a proof-of-concept chip in the TSMC 0.18μ CMOS process for our design for clients operating at identical clock frequencies and it is currently in fabrication. We are currently designing a chip to demonstrate our interfaces based on miss-detectors and will fabricate it in 2003.

Acknowledgments

Jo Ebergen of SUN Microsystems pointed out the similarities between our controller and GasP handshaking circuits. Roozbeh Mehrabadi, Roberto Rosales, and Brian Winters have provided valuable suggestions during the design of our chip. This work has received support from NSERC and CMC.

References

[1] K. Bernstein, K. M. Carrig, et al. *High Speed CMOS Design Styles*. Kluwer, 1999.

[2] K. A. Bowman, S. G. Duvall, and J. D. Meindl. Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, 37(2):183–190, Feb. 2002.

[3] A. Chakraborty and M. R. Greenstreet. A minimalist source-synchronous interface. In *Proceedings of the 15th IEEE ASIC/SOC Conference*, pages 443–447, Sept. 2002.

[4] T. Chaney and C. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, Apr. 1973.

[5] D. M. Chapiro. *Globally-Asynchronous, Locally-Synchronous Systems*. PhD thesis, Department of Computer Science, Stanford University, Oct. 1984. Tech. Report STAN-CS-84–1026.

[6] T. I. Chappell, B. A. Chappell, et al. A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture. *IEEE Journal of Solid-State Circuits*, 26(11):1577–1585, Nov. 1991.

[7] T. Chelcea and S. M. Nowick. Robust interfaces for mixed-timing systems with application to latency-insensitive protocols. In *Proceedings of the 38th ACM/IEEE Design Automation Conference*, pages 21–26, June 2001.

[8] B. Davari. CMOS technology: Present and future. In *Proceedings of 1999 Symposium on VLSI Circuits*, pages 5–10. IEEE, June 1999.

[9] L. R. Dennison, W. J. Dally, and D. Xanthopoulos. Low-latency plesiochronous data retiming. In *Proceedings of the Sixteenth Anniversary Conference on Advanced Research in VLSI*, pages 304–315, 1995.

[10] S. Furber, D. A. Edwards, and J. D. Garside. AMULET3: a 100 MIPS asynchronous embedded processor. In *Proceedings of the 2000 International Conference on Computer Design*, pages 329–334, Sept. 2000.

[11] S. Geissler, D. Appenzeller, et al. A low-power RISC microprocessor using dual PLLs in a 0.13μ SOI technology with copper interconnect and low-k BEOL dielectric. In *Proceedings of the 2002 International Solid-State Circuits Conference*, pages 148–149, Feb. 2002.

[12] M. R. Greenstreet. *STARI: A Technique for High-Bandwidth Communication*. PhD thesis, Department of Computer Science, Princeton University, Jan. 1993.

[13] M. R. Greenstreet. Implementing a STARI chip. In *Proceedings of the 1995 International Conference on Computer Design*, pages 38–43, Austin, Texas, Oct. 1995.

[14] D. Harris and S. Naffziger. Statistical clock skew modeling with data delay variations. *IEEE Transactions on VLSI Systems*, 9(1):888–898, Dec. 2001.

[15] G. Hinton, M. Upton, et al. A 0.18μ CMOS IA-32 processor with a 4-GHz integer execution unit. *IEEE Journal of Solid-State Circuits*, 36(11):1617–1627, Nov. 2001.

[16] A. Iyer and D. Marculescu. Power-performance evaluation of globally asynchronous, locally synchronous processors. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 158–168, June 2002.

[17] H. Johnson and M. Graham. *High-Speed Digital Design: A Handbook of Black Magic*. Prentice Hall, 1993.

[18] G. K. Konstadinidis, K. Normoyle, et al. Implementation of a third-generation 1.1-GHz 64-bit microprocessor. *IEEE Journal of Solid-State Circuits*, 37(11):1461–1469, Nov. 2002.

[19] G. Kornaros, D. Pnevmatikatos, et al. ATLAS 1: Implmenting a single-chip ATM switch with backpressure. *IEEE Micro*, 19(1):30–41, Jan/Feb 1999.

[20] A. Kowalczyk, V. Adler, et al. The first MAJC microprocessor: A dual CPU system-on-a-chip. *IEEE Journal of Solid-State Circuits*, 36(11):1609–1916, Nov. 2001.

[21] N. A. Kurd, J. S. Barkatullah, et al. Multi-GHz clocking scheme for Intel® Pentium® 4 microprocessor. In *Proceedings of the 2001 International Solid-State Circuits Conference*, pages 404–405, Feb. 2001.

[22] A. J. Martin, A. Lines, et al. The design of an asynchronous MIPS R3000 microprocessor. In *Proceedings of the 17th Conference on Advanced Research in VLSI*, pages 164–181, Sept. 1997.

[23] D. G. Messerschmitt. Synchronization in digital system design. *IEEE Journal on Selected Areas in Communications*, 8(8):1404–1419, Oct. 1990.

[24] S. Moore, G. Taylor, et al. Point to point GALS interconnect. In *Proceedings of the Eigth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 62–68, Apr. 2002.

[25] J. Mutterbach, T. Villiger, and W. Fichtner. Practical design of globally-asynchronous, locally-synchronous sytems. In *Proceedings of the Sixth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 52–59, Apr. 2000.

[26] K. Olukotun, B. A. Nayfeh, et al. The case for a single-chip multiprocessor. In *Proceedings of the Seventh International Symposium on Architectural Support for Parallel Languages and Operating Systems*, pages 121–132, Oct. 1996.

[27] P. J. Restle, T. G. McNamara, et al. A clock distribution network for microprocessors. *IEEE Journal of Solid-State Circuits*, 36(5):792–799, May 2001.

[28] P. Riocreux, L. Brackenbury, et al. A low-power self-timed viterbi decoder. In *Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 15–24, 2001.

[29] T. Seceleanu, J. Piosila, and P. Liljeberg. On-chip segmented bus: A self-timed approach. In *Proceedings of the 15th IEEE ASIC/SOC Conference*, pages 216–220, Sept. 2002.

[30] J. N. Seizovic. Pipeline synchronization. In *Proceedings of the First International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87–96. IEEE Computer Society Press, 1994.

[31] A. E. Sjogren and C. J. Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. *IEEE Transactions on VLSI Systems*, 8(5):573–583, Oct. 2000.

[32] I. Sutherland and S. Fairbanks. GasP: A minimal FIFO control. In *Proceedings of the Seventh International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 46–53, Apr. 2001.

[33] S. Tam, S. Rusu, et al. Clock generation and distribution for the first IA-64 microprocessor. *IEEE Journal of Solid-State Circuits*, 35(11):1545–1552, Nov. 2000.

[34] A. J. Winstanley, A. Garivier, and M. R. Greenstreet. An event spacing experiment. In *Proceedings of the Eigth International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 42–51, Manchester, UK, Apr. 2002.

[35] E. Yeung and M. A. Horowitz. A 2.4 Gb/s/pin simultaneous bidirectional parallel link with per-pin skew compensation. *IEEE Journal of Solid-State Circuits*, 35(11):1619–1628, Nov. 2000.

[36] K. Y. Yun and R. P. Donohue. Pausible clocking: A first step toward heterogeneous systems. In *Proceedings of the 1996 International Conference on Computer Design*, pages 118–123, Oct. 1996.