# Gradual Synchronization

Sandra Jackson and Rajit Manohar
Computer Systems Laboratory, Cornell University
Ithaca, NY 14853, U.S.A.
{sandra, rajit}@csl.cornell.edu

***Abstract*—System-on-Chip (SoC) designs using multiple clock domains are gaining importance due to clock distribution difficulties and increasing in-die process variations. For the same reasons more emerging SoC designs utilize clock-less domains for parts of the system. Both clock domain crossing and clocked/clock-less domain crossing require a mechanism for inter-domain data transfer that re-synchronizes data to the clock domain of the receiver and avoids metastability. These synchronizers introduce added latency and reduce throughput. This paper proposes merging synchronization with computation in order to reduce latency while keeping throughput high. The method, called Gradual Synchronization (GSync), can reduce synchronization latency at maximum operating frequency by up to 37 percent, with even greater benefit at slower frequencies. We show the benefits of this approach in the scenario of an asynchronous NoC with synchronous end-points.**

## I. Introduction

The presence of multiple clock domains or mixed clocked/clock-less domains in emerging system-on-chip (SoC) designs creates the possibility of errors during communication across domain boundaries. The classic problem of re-synchronizing an asynchronous signal to a clock is known to exhibit the phenomenon of *metastability*. Metastability is an instance of Buridan's principle (named after fourteenth century French philosopher Jean Buridan) which can be stated as follows: a discrete decision based on a continuous range of values cannot be made within a bounded length of time [1]. In this particular case, the decision to be made is whether a signal is at logic zero or logic one, and it must be based on the voltage of the signal—a continuous value. The amount of time required for a circuit (known as a *synchronizer*) to "make up its mind" and determine whether an asynchronous signal is logic zero or logic one cannot be bounded, and the net effect is that this decision *may* require more time than allocated to it by a synchronous circuit—causing the synchronous logic to potentially malfunction. This phenomenon was first experimentally demonstrated by Chaney and Molnar [2].

Fortunately, there are well-known synchronizer implementations that ensure the probability that the circuit has "not made up its mind" decays exponentially with time [3], [4]. The probability that it takes more than time $T$ to make a decision, $\Pr\{t > T\}$, is given by the expression $e^{-T/\tau}$ where $\tau$ is the time constant of the synchronizer. An adequate level of protection from synchronization failure is obtained by simply waiting long enough—i.e., by choosing a large enough $T$.

The most common synchronizer is the "two flip-flop" synchronizer shown in Figure 1. In this design, the transmitted signal to be synchronized is passed through two sequential flip-flops both of which are clocked by the receiver's clock. The presence of two flip-flops ensures that at least a full cycle is permitted for the resolution of any metastability. This interval is deemed to be sufficient for all practical purposes, as it can provide an extremely low failure probability [5], [6]. While this solution is very robust, it incurs a high penalty in terms of synchronization overhead—both in terms of latency as well as throughput. Attempts to "optimize" this synchronizer to reduce its latency can result in erroneous implementations [6].

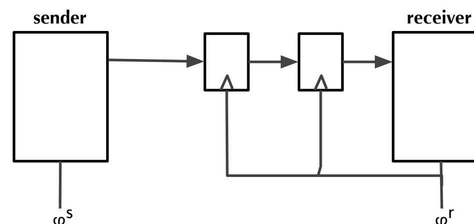


Fig. 1. Classic two flip-flop synchronizer. $\varphi^r$ is the receiver clock and $\varphi^s$ is the sender clock.

In this paper, we present an alternative approach that merges synchronization into pipelined computation stages. This approach reduces the number of cycles lost to synchronization.

## II. Related Work

Correct use of the "two flip-flop" synchronizer requires some flow control between the two communicating clock domains. This is often accomplished in the form of requests and acknowledges. Unfortunately, both must be synchronized to their respective receiving clocks. Carefully optimizing this synchronizer in the context of the flow control structure achieves a lower latency and higher throughput capability [7], the modifications do result in a reduction of the mean time between failures (MTBF). A fast alternative is the even/odd synchronizer [8]. This synchronizer computes a phase estimate based on the relative frequency of the two clocks. The transmitter writes registers on alternate cycles and the receiver uses the phase estimate to determine which register to sample. This synchronizer must be integrated with some form of flow control in order to ensure every data is sampled.

In order to keep the synchronization throughput high both Seizovic [9] and Nowick [10] proposed FIFO based synchronizers. In Seizovic's work an asynchronous FIFO is constructed, each FIFO stage attempts to further synchronize the

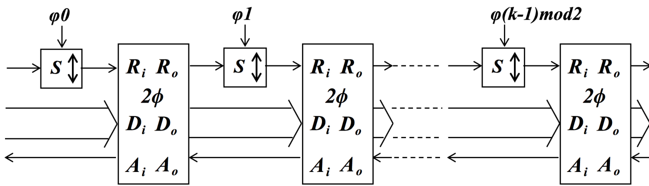


Fig. 2. A detailed view of a pipeline synchronizer

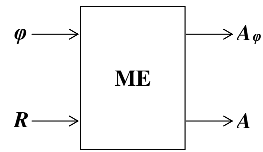


Fig. 3. An ME element with one input connected to a clock.

data so that the final output of the FIFO is synchronized to the receiver clock, effectively creating pipelined synchronization. Section II-A discusses this work in greater detail. Nowick's dual-clock FIFO design only incurs a synchronization overhead when the FIFO is nearly full or nearly empty. While this solution will not have a significant synchronization penalty when the FIFO is busy, is leads to high synchronization latency when the FIFO is either completely full or completely empty. When two domains communicate with differing frequencies, it seems likely that the connection between the two domains will encounter this high latency scenario more often than not.

Other work has focused on periodic clock domains, where a state-machine is used to block communication between two flip-flops that are in different clock domains if the communication may lead to metastability. These solutions rely on a well-defined and predictable clock periodicity, and in some cases they require that the clocks have similar frequencies or frequencies that bear a rational relationship [11], [12], [13]. Most of these approaches also have a significant communication latency, because the core of the design involves communication between two flip-flops in different clock domains. Also, this class of solutions do not operate through gated clocks as they exploit the periodicity of the clock signal, although this may not be a serious drawback if the number of crossings between synchronization domains is small.

The situation is different if the receiver is completely asynchronous. An idle asynchronous circuit is ready to receive its inputs at any time, and therefore will not encounter the metastability problem in the form described above [3]. Even if an asynchronous circuit wishes to sample a signal while it is changing, it can do so safely because the rest of the asynchronous circuit can wait for the synchronizer to "make up its mind" [14]. In certain special cases, one can even optimize the synchronizer design so as to have minimal impact on overall system performance [15].

### A. Pipeline Synchronization

The pipeline synchronizer attempts to change the arrival time of an input signal to a time that is synchronized with its clock input by the end of $k$ stages. Each stage consists of one synchronizer and one FIFO element. The stages are cascaded to form the full pipeline allowing more than one data set to be passing through the synchronizer at a time as shown in figure 2.

The synchronizer is a mutual exclusion (ME) element. The ME element ensures that only one of its input signals can be acknowledged at any one time. Connecting one of the ME element's input signals to a clock has the effect of adding a variable delay to the acknowledge of the other input (see Figure 3). If T is the clock period, the ME will block R from passing to A for half the clock period, and let R pass to A in the other half of the clock period. If R shows up within $T_w$ of the relevant clock edge then a metastability occurs. However, since the clock oscillates, the ME element will either exit the metastable state on its own or be forced out of the metastable state when the clock makes its next transition.

The asynchronous FIFO element follows the specification:

$$*[[R_i]; A_i, R_o; [A_o]] \tag{1}$$

The clocks are two phase non overlapping clocks, $\varphi_0$ and $\varphi_1$. As the total number of stages ($k$) increases, the probability of encountering a metastability in the $k^{th}$ stage decreases. The alternating clock ensures that if no metastability is encountered, a blocking phase will be encountered and the synchronizer will enter its steady state. In the steady state each stage takes $T/2$ and the signals are synchronized with the clock.

The MTBF can be adjusted to meet the need of the system by changing the number of stages in the synchronizer. Each stage introduces a latency of $\frac{T}{2}$. This latency is the major disadvantage of pipeline synchronization. Gradual synchronization uses a similar staged approach and also uses Seitz's implementation of the ME element as a synchronizer, but GSync does not introduce a large latency penalty since it allows synchronization to occur in parallel with existing system computation.

## III. Gradual Synchronization

Gradual Synchronization merges synchronization with computation by dispersing existing pipeline stages into synchronizer stages as shown in Figure 4. In each stage data computation (CL) occurs while the synchronizer (S) block attempts to synchronize the request to the receiving clock. The two tasks which normally occur in isolation now occur in parallel. Designed in this manner the synchronizer now resembles two-phase pipeline operation with the FIFOs as the latches between the stages.

### A. Gradual Synchronizer

Each FIFO in the pipeline requires stable data to be present at the moment it receives a request. Performing computation on the data in each stage means values change once released from the sending FIFO. Without sufficient computation time

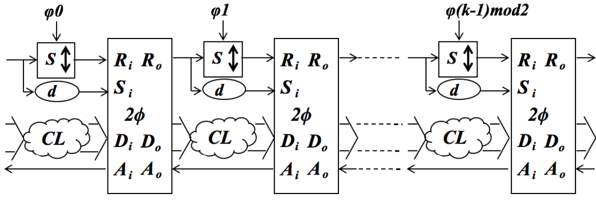


Fig. 4. Asynchronous input, synchronous output two-phase gradual synchronizer.

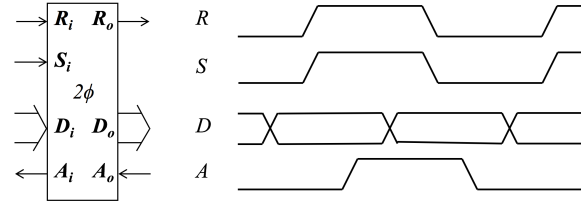


Fig. 5. Two-phase FIFO element with second request input for data safety.

incorrect values will be latched by the receiving FIFO. Data safety is accomplished by adding a computation delay equivalent to the worst case delay in parallel with the synchronization. The send request is split, one copy goes through the synchronizer and the other through the computation delay. The FIFO element has a second request input ($S_i$) as shown in Figure 5. This input allows the data to be latched when stable but will not initiate the send request unless $R_i$ is present as well. The control signals for the modified FIFO follow the specification (two-phase protocol):

$$*[[R_i \wedge S_i]; A_i, R_o; [A_o]]. \tag{2}$$

### B. Correctness Argument

The intuition behind correct operation of the gradual synchronizer is that since the incoming request is split, the synchronization and the computation delay timer begin at the same time. As a result, a few simple implementation requirements are enough to ensure that the $S_i$ signal cannot cause a metastability, nor reduce the throughput of the FIFO structure.

The $j^{th}$ event on $R_o^{(i)}$ can occur at time:

$$t^{(j)}_{R_o^{(i)}} = t^{(j)}_{R_i^{(i)}} + \tau_{R_i R_o}, \tag{3}$$

$$t^{(j)}_{R_o^{(i)}} = t^{(j-1)}_{A_o^{(i)}} + \tau_{A_o R_o}, \tag{4}$$

or

$$t^{(j)}_{R_o^{(i)}} = t^{(j)}_{S_i^{(i)}} + \tau_{S_i R_o}. \tag{5}$$

Metastable behavior occurs when $R_o$ arrives at a synchronizer coincident with a rising edge of the clock input. Since there are three times at which $R_o$ can occur, the probability

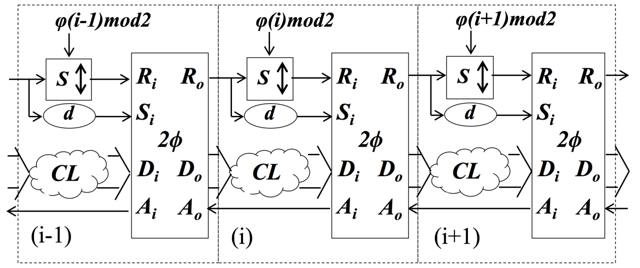


Fig. 6. Segment of a two-phase asynchronous input, synchronous output gradual synchronizer.

that a metastability is encountered at the $(i+1)^{st}$ synchronizer is:

$$P_f^{(i+1)} \leq P_f^{(i+1)}(R_i) + P_f^{(i+1)}(A_o) + P_f^{(i+1)}(S_i). \tag{6}$$

The first term is the probability than an event on $R_{(i)}$ occurs $\tau_{R_i R_o}$ before the clock edge. The last term corresponds to the probability that an event on $S_{(i)}$ occurs $\tau_{S_i R_o}$ before the clock edge. In order to reclaim the most time for computation, computation must occur in parallel with synchronization. Therefore, the computation delay $\tau_d$ for one stage must not exceed the latency introduced per stage:

$$\tau_d + \tau_{S_i R_o} < T/2. \tag{7}$$

$R_o^{(i-1)}$ must show up exactly $t_a$ after the falling clock edge in order for $S_i^{(i)}$ to cause a metastability in the $(i+1)^{st}$ synchronizer. However, since $R_o^{(i-1)}$ will also arrive at the $(i)^{th}$ synchronizer at the same time $\tau_d$ begins, the $i^{th}$ synchronizer will block $R_o^{(i-1)}$ and the $(i)^{th}$ FIFO will be waiting for $R_i^{(i)}$, not $S_i^{(i)}$. Meaning, $P_f^{(i+1)}(S_i) = 0$ and $P_f^{(i+1)}(R_i)$ remains unchanged because the addition of $\tau_d$ and the $S_i$ input to the FIFO cannot create a condition under which $R_i$ causes a metastability without a metastability in the previous stage. This probability is:

$$P_f^{(i+1)}(R_i) \leq P_f^{(i)} e^{-\frac{\frac{T}{2} - \tau_S - \tau_{R_i R_o}}{\tau_0}}. \tag{8}$$

Now, the only thing left to ensure is that the presence of $S_i$ in the stages does not affect $P_f^{(i+1)}(A_o)$. Without the $S_i$ input this probability can be ignored and does not need to be equal to zero because if a metastability on the $j^{th}$ event in the pipeline is caused by a transition on $A_o$ that metastability would be SEM (second-event metastability).

**Defintion 1** *When the input of a synchronizer element S, clocked with $\varphi$, changes state coincident with an arbitrary, jth down-going edge of $\varphi$, and there was at least one prior input event between the $(j-1)^{st}$ and the $j^{th}$ down-going edge of $\varphi$, we shall say that S has entered second-event metastability.*

Since the synchronous circuit at the end of the pipeline can be designed to accept only one event per clock cycle, the SEM that would have caused a metastability will be ignored until the next clock cycle.

The SEM argument would become invalid if $t^{(j-1)}_{R^{(i)}_o}$ could somehow occur at a time:

$$t^{(j-1)}_{R^{(i)}_o} < t^{(j)}_{R^{(i)}_o} - T. \tag{9}$$

However, we know that

$$t^{(j-1)}_{A^{(i+1)}_i} \equiv t^{(j-1)}_{A^{(i)}_o} = t^{(j)}_{R^{(i)}_o} - \tau_{A_oR_o} \tag{10}$$

and the only way $S_i$ could change the behavior of $A^{(i)}_o$ would be if the transition on $S^{(i+1)}_i$ arrived at $FIFO^{(i+1)}$ after $R^{(i+1)}_i$. This would imply that:

$$t^{(j-1)}_{R^{(i)}_o} = t^{(j)}_{R^{(i)}_o} - \tau_{A_oR_o} - \tau_{S_iA_i} - \tau_d \tag{11}$$

Usually, for a two-phase FIFO implementation:

$$\tau_{S_iA_i} \approx \tau_{S_iR_o}. \tag{12}$$

So, adding and meeting the requirement

$$\tau_{S_iA_i} + \tau_d < T/2, \tag{13}$$

is trivial. Therefore, as long as

$$\tau_{A_oR_o} < T/2 \tag{14}$$

is preserved,

$$t^{(j-1)}_{R^i_o} > t^{(j)}_{R^i_o} - T. \tag{15}$$

contradicting equation 9 and the SEM argument holds.

SEM itself can only cause SEM, and the $S_i$ cannot change this since if a metastability occurs in the synchronizer in stage $(i)$, $S_i$ cannot arrive at a time that will cause a metastability in stage $(i+1)$.

The synchronous environment on the receiving end of the synchronizer is designed to accept only one data item per clock cycle. For the best performance, GSync should be able to sustain that throughput at all times. To ensure GSync operates at the desired throughput a few additional requirements are derived below.

Figure 7 shows the steady state of a gradual synchronizer with an infinite number of stages. All events on $R_i$ entering even-numbered FIFO blocks arrive $\tau_S$ after the rising edge of $\varphi_0$. All events on $R_i$ entering odd-numbered FIFO blocks arrive $\tau_S$ after the rising edge of $\varphi_1$. All events on $S_i$ entering even-numbered FIFO blocks arrive $\tau_{da}$ after the rising edge of $\varphi_0$ and all events on $S_i$ entering odd-numbered FIFO blocks arrive $\tau_{da}$ after the rising edge of $\varphi_1$. In the steady state, no synchronizer assumes a metastable state, and:

$$t1 = max(\tau_S + \tau_{R_iA_i}, t1 + \tau_{A_oA_i} - \frac{T}{2}, \tau_{da} + \tau_{S_iA_i}) \tag{16}$$

$$t2 = max(\tau_S + \tau_{R_iR_o}, t1 + \tau_{A_oR_o} - \frac{T}{2}, \tau_{da} + \tau_{S_iR_o}) \tag{17}$$

The value $\tau_{da}$ is the portion of the computational delay that takes place after the clock edge, since $\tau_d$ may cross over the clock edge. So, for $\tau_{A_oA_i} < \frac{T}{2}$,

$$t1 = max(\tau_S + \tau_{R_iA_i}, \tau_{da} + \tau_{S_iA_i}) \tag{18}$$

$$t2 = max(\tau_S + \tau_{R_iR_o}, \tau_S + \tau_{R_iA_i} + \tau_{A_oR_o} - \frac{T}{2}, \tau_{da} + \tau_{S_iA_i} + \tau_{A_oR_o} - \frac{T}{2}, \tau_{da} + \tau_{S_iR_o}) \tag{19}$$

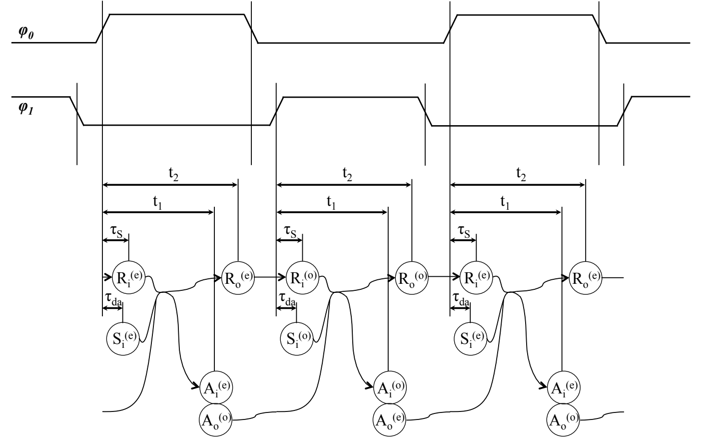


Fig. 7. Steady-state operation of the 2-phase asynchronous-to-synchronous gradual synchronizer.

To maintain the steady state $t_2$ must be less than $\frac{T}{2}$ so,

$$\tau_{A_oA_i} < T/2 \tag{20}$$

$$\tau_S + \tau_{R_iR_o} < T/2 \tag{21}$$

$$\tau_S + \tau_{R_iA_i} + \tau_{A_oR_o} < T \tag{22}$$

$$\tau_{da} + \tau_{S_iR_o} < T/2 \tag{23}$$

$$\tau_{da} + \tau_{S_iA_i} + \tau_{A_oR_o} < T \tag{24}$$

Of the above, $\tau_{da} + \tau_{S_iR_o}$ is already limited to a value less than $\frac{T}{2}$ by a stricter requirement, so this requirement is not in the final list of conditions. It is enough to simply state that:

$$\tau_d = \tau_{db} + \tau_{da}. \tag{25}$$

That leaves only the final inequality in the above equation with the rather ambiguous term $\tau_{da}$. Substituting for $\tau_{da}$ and reducing yields:

$$t2 + \tau_d + \tau_{S_iA_i} + \tau_{A_oR_o} - T. \tag{26}$$

In order to cancel the above term. The inequalitiy:

$$\tau_d + \tau_{S_iA_i} + \tau_{A_oR_o} < T \tag{27}$$

is added to the requirements.

In order to ensure safe operation of the asynchronous-to-synchronous gradual synchronizer the design requirements are:

$$\begin{aligned}
\tau_S + \tau_{R_iR_o} &< T/2 \\
\tau_{A_oR_o} &< T/2 \\
\tau_{AR} &< T/2 \\
\tau_{A_oA_i} &< T/2 \\
\tau_S + \tau_{R_iA_i} + \tau_{A_oR_o} &< T \\
\tau_S + \tau_{R_iA_i} + \tau_{AR} &< T \\
\tau_{RA} + \tau_{A_oR_o} &< T \\
\tau_d + \tau_{S_iR_o} &< T/2 \\
\tau_d + \tau_{S_iA_i} + \tau_{A_oR_o} &< T \\
\tau_d + \tau_{S_iA_i} + \tau_{AR} &< T
\end{aligned} \tag{28}$$

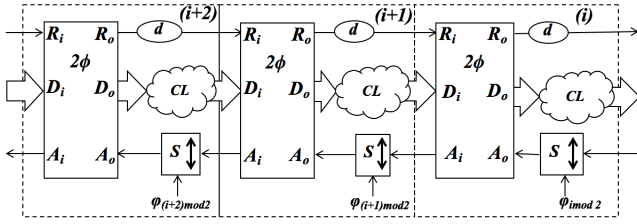


Fig. 8. Segment of the 2-phase synchronous-to-aynchronous gradual synchronizer.

### C. Synchronous to Asynchronous

With an asynchronous to synchronous synchronizer established only half of the job is done. For a full synchronous-to-synchronous synchronizer a synchronous-to-asynchronous synchronizer is needed as well. In this case, data, and hence computation, move in the opposite direction from the synchronization. The FIFO blocks are basic asynchronous FIFO blocks with no extra inputs necessary. As shown in figure 8 computation is performed on the data sent out by the synchronous domain. The request must be delayed in order to ensure data safety, but the synchronization must happen on the acknowledge. The proof and requirements derivation for this direction can be found in [16]. The requirements for this direction are

$$\begin{aligned} \tau_S + \tau_{A_oA_i} &< T/2 \\ \tau_{R_iA_i} + \tau_d &< T/2 \\ \tau_{RA} &< T/2 \\ \tau_{R_iR_o} + \tau_d &< T/2 \\ \tau_S + \tau_{A_oR_o} + \tau_{R_iA_i} + \tau_d &< T \\ \tau_S + \tau_{A_oR_o} + \tau_{RA} &< T \\ \tau_{AR} + \tau_{R_iA_i} + \tau_d &< T. \end{aligned} \tag{29}$$

### D. Four-Phase-Protocol Gradual Synchronizers

In the four-phase version of the gradual synchronizer, the FIFO elements are four-phase protocol FIFOs and the synchronizers are asymmetric detecting only certain edges. The delay element is also asymmetric. There are many ways to reshuffle the handshake of the four-phase protocol, one such possibility is:

$$*[[R_i \wedge S_i]; A_i\downarrow, [\overline{R_i} \wedge \overline{S_i}], A_i\uparrow, R_o\uparrow; [\overline{A_o}], R_o\downarrow; [A_o]]. \tag{30}$$

The requirements for this direction must include the delays for the full set of four-phase handshake transitions. The proof and requirements for the four-phase gradual synchronizer can be found in [16]. For the above handshaking expansion the requirements for the four-phase gradual synchronizers can be obtained by making the following substitutions in the two-phase synchronizer requirements stated above:

$$\begin{aligned} \tau_{R_iA_i} &= \tau_{R_i\uparrow A_i\downarrow} + \tau_{A_o\downarrow R_o\downarrow} + \tau_{R_i\downarrow A_i\uparrow} \\ \tau_{R_iR_o} &= \tau_{R_i\uparrow A_i\downarrow} + \tau_{A_o\downarrow R_o\downarrow} + \tau_{R_i\downarrow R_o\uparrow} \\ \tau_{A_oA_i} &= \tau_{A_o\uparrow A_i\downarrow} + \tau_{A_o\downarrow R_o\downarrow} + \tau_{R_i\downarrow A_i\uparrow} \\ \tau_{A_oR_o} &= \tau_{A_o\uparrow R_o\uparrow} \\ \tau_{S_iA_i} &= \tau_{S_i\uparrow A_i\downarrow} + \tau_{A_o\downarrow R_o\downarrow} + \tau_{R_i\downarrow A_i\uparrow} \\ \tau_{S_iR_o} &= \tau_{S_i\uparrow A_i\downarrow} + \tau_{A_o\downarrow R_o\downarrow} + \tau_{R_i\downarrow R_o\uparrow}. \end{aligned} \tag{31}$$

### E. MTBF

When using a synchronizer it is important to ensure it can meet the mean time between failures (MTBF) requirement of the system. If $T_{oh}$ is the overhead introduced by the asynchronous FIFOs in a gradual synchronizer then the probability of metastability failure at the synchronous boundary after $k$ stages is

$$P_f^{(k)} = P_f^{(0)} e^{\frac{-k(\frac{T}{2} - T_{oh})}{\tau_o}}. \tag{32}$$

$P_f^{(0)}$ is the rate that metastability occurs at the inputs to the synchronizer at the asynchronous end. This rate is calculated as

$$R(metastability) = T_w F_C F_D, \tag{33}$$

where $T_w$ is the window around the sampling edge of the clock during which a change in the input data could cause a metastability, $F_C$ is the clock frequency, and $F_D$ is the injection rate of the input data. The synchronizer then reduces the chance that a metastability at its input will cause a failure in the circuit at its output,

$$R(failure) = T_w F_C F_D e^{\frac{-k(\frac{T}{2} - T_{oh})}{\tau_o}}. \tag{34}$$

The MTBF is the inverse of the failure rate,

$$MTBF_{GS} = \frac{e^{\frac{k(\frac{T}{2} - T_{oh})}{\tau_o}}}{T_w F_C F_D}. \tag{35}$$

## IV. Performance

FIFO synchronizers like the pipeline synchronizer and the Dual-clock FIFO exhibit best case throughput, allowing data through at the highest rate allowed by the relative frequency of the circuits the synchronizer connects. However, the ”two flip-flop” synchronizers such as the simple four-phase, fast four-phase and fast two-phase have better forward latency. The goal of GSync is to maintain maximum possible throughput at a lower latency than current methods. In this section we validate the gradual synchronizer method through simulation and compare it to the aforementioned synchronizers. In addition, we present an implementation of a network interface (NI) using GSync and compare its capabilities with a pipeline synchronizer NI and a fast-four-phase synchronizer NI.

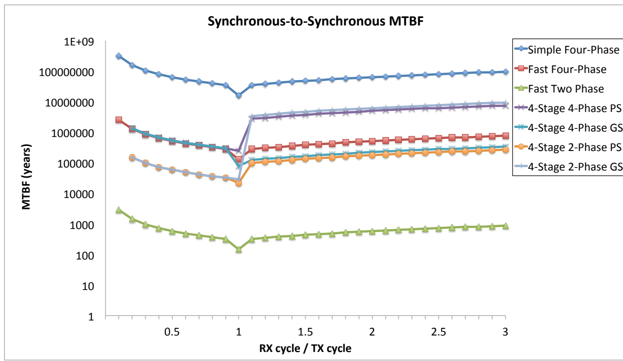


Fig. 9. MTBFs of synchronizer methods over a range of clock cycle ratios for 900MHz.

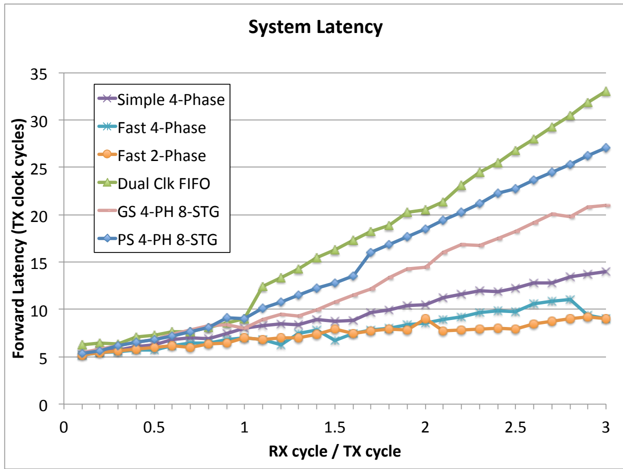


Fig. 10. Total forward latency for five transmitter pipeline stages plus synchronization at 900MHz.

### *A. Throughput and Latency*

The synchronizer circuits were custom designed and simulated with HSIM using 90nm technology. Synchronizers were placed between synchronous domains with the sender always transmitting and the receiver always ready. The operating frequency of the faster side is 900MHz with the slower side scaled to match the desired frequency ratio. Faster frequencies are possible however figure 9 shows that 900MHz provides a comparison point where the MTBFs of the FIFO methods fall in the range between the best and worst two flip-flop MTBFs.

For the latency metric five sender pipeline stages are included in the measurement. GSync is merged with these stages. We assume fully utilized pipeline stages, which is the worst case scenario for the gradual synchronizer because the overhead of the method creates a small amount of additional computation which when merged, may not fit safely within a cycle along with existing computation.

Figure 10 shows the latency from the point one request is put on the input of the first of the five pipeline stages until it appears on the output of the synchronizer. When the dual-clock FIFO buffer fills up because the sender is faster than the receiver the in-place nature of the buffer causes pointer

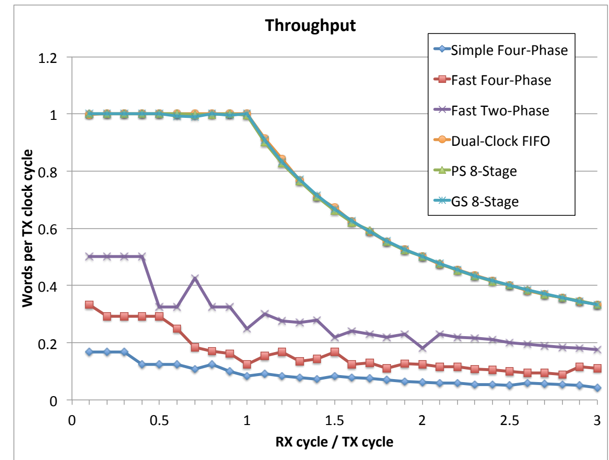


Fig. 11. Throughput comparison of synchronizers between two clocked endpoints. All the FIFO synchronizers share the same curve.

collision. Requests wait in the FIFO and also experience the synchronization latency required to resolve the pointer collision. The pipeline synchronizer latency is lower because moving the storage location of requests as they move forward in the FIFO decouples the sender and receiver even in high occupancy cases. GSync keeps latency at least an additional 22 percent lower then PS.

When the sender is slower (ratios less than one) the benefit of merging synchronization into pipeline stages isn't as significant since the forward latency of synchronizing to a fast receiver is smaller. Still the GSync's forward latency is slightly lower or the same as the other FIFO synchronizers.

At lower frequencies the benefit of gradual synchronization will be even greater since the method overhead uses a smaller percentage of the total cycle time leaving more computation time available in each stage. The forward latency of GS is still higher than that of the two flip-flop synchronizers, this is a result of these synchronizers only servicing one request at a time. This is why throughput must be taken into account.

Figure 11 shows the throughput of the various synchronizers over a range of clock ratios. A ratio less than one means that the receiver has a faster frequency. In these cases the sender can inject requests into the FIFO synchronizers every cycle. When the receiver is slower (ratios greater than one) the FIFO synchronizer's data rate falls predictably based on how fast the receiver can empty the FIFO. However, even the best two-flip flop synchronizer (fast two-phase) must complete a full handshake exchange between the endpoints before accepting a new request, reducing the data rate to at most half the FIFO synchronizers'.

When the sender needs to send more requests the throughput capability has a larger effect on the total latency then the forward latency does. A very busy system does better with some buffering capability. However, even a system that only sometimes has more than one request to inject benefits from a method with some buffering in order to avoid intermittent bottlenecks.

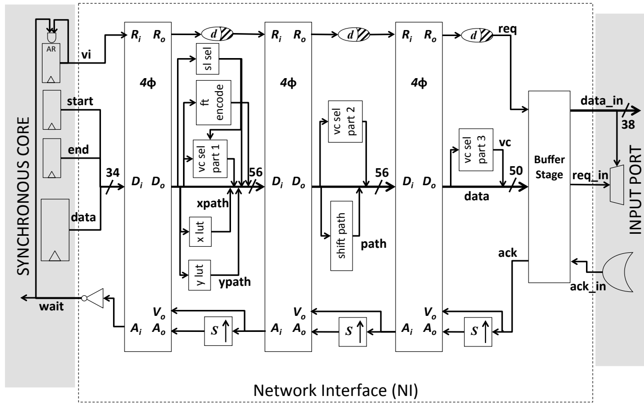


Fig. 12. Core to network NI design.

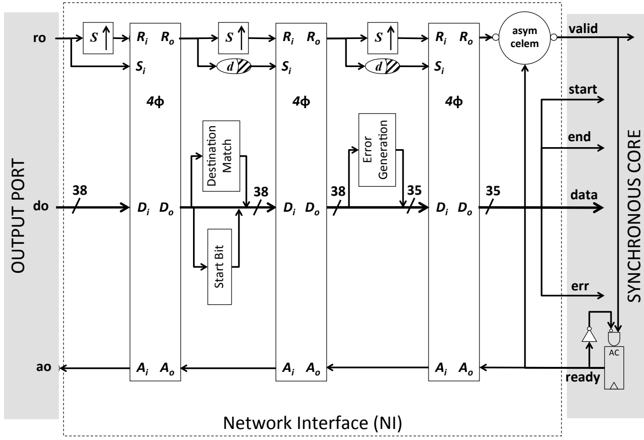


Fig. 13. Network to core NI design.

## B. Network Interface

The benefit of gradual synchronization depends heavily on the computation available with which to merge synchronization. What computation exists is only know in the presence of a practical application. In this section we implement a network interface (NI) for connecting synchronous cores to an asynchronous network. We assume a basic 2-D mesh architecture connected by QNoC asynchronous routers [17] with four virtual channels and four service levels. The QNoC asynchronous router uses a 4-phase asynchronous handshake, making a 4-phase gradual synchronizer a natural choice. We compare the GSync NI to an NI implemented with a fast-four phase synchronizer and an NI using the pipeline synchronizer.

Figure 12 shows the core to network NI. The synchronous core issues a send request by asserting vi when it places message data on its outputs. The NI sends an acknowledge back to the core. The acknowledge is used to simulate a handshake signal on vi from the core. Then the message is processed and prepared for the network. The buffer in the final stage only stores the second flit of a header message, the first flit is forwarded directly to the network input port.

Figure 13 shows the network to core direction of the NI. The output port pushes a flit into the NI by asserting $ro$ and placing the data on $do$. The NI and the network exchange a handshake that is complete when the flit is latched in the first FIFO. The remaining stages unpack the flit into the message destined for the core. We assume the synchronous core is able to accept one message every cycle.

The implementation of the core to network NI assumes the network is always ready to accept new flits, so the NI does not experience network stalls. An acknowledge can only occur after a request has been issued so the output port will be busiest when the core is constantly trying to send. We use this rate as the basis for MTBF calculation so that the design will meet requirements in the worst case. The fast 4-phase MTBF is used as the minimum allowed MTBF and the MTBFs of the other synchronizers are required to be the same or better.

Table I shows the results of simulating the core to network NI designs at several different core frequencies. The network frequency is the frequency of change of the acknowledge signal, this helps determine the MTBF, and changes according to the synchronization method chosen. The latency reported is the time from when a message packet is injected into the NI to when the flit(s) appears on the NI output.

Increasing the core frequency to 800 MHz causes relocation of some flit preparation computation into an additional synchronous pipeline stage in the F4PH NI. This leads to increased latency because the throughput bottleneck of this method can cause a packet to get stalled in the first stage of the NI if another packet is already in the synchronizer stage. The 800 MHz PS NI experiences the same computation bloat, plus additional latency degradation from the additional synchronizer stage needed to meet the MTBF requirement. The 800 MHz GSync NI places the extra computation into the extra synchronizer stage required to meet MTBF requirements which results in lower latencies.

The F4PH NI can transmit one message every three cycles at all frequencies. The GSync NI and PS NI can transmit one message per clock cycle at a core frequency of 400MHz. At 600MHz they maintain that rate if the core is transmitting mostly body/tail messages. The network side cannot keep up if a lot of headers are transmitted, since headers result in the transmission of two flits. At 800MHz the GSync NI and PS NI experience a data rate degradation for both types of messages.

For the network to core portion of the interface, we design the GSync NI and the PS NI MTBF to be the same as or higher than the F4PH NI MTBF. This results in a 3-stage PS and GSync NI for 400MHz and 600MHz and a 4-stage PS and GSync NI for 800MHz. The computation required to unpack flits fits into one clock cycle even at 800MHz, meaning the additional stage added passes data directly to the next stage without performing computation.

Performance for the network to core interface is shown in table II. The F4PH NI can handle one flit every three cycles and the PS and GSync NI one every cycle. The minimum case latencies are seen when the NI is empty and waiting for a new flit. The more common case is closer to the maximum latency. The GSync NI has lower latencies and the same data rate as

TABLE I
OUTGOING MESSAGE NI SIMULATION RESULTS

| Sync Type | TX Clock (MHz) | Network (MHz) | MTBF (years) | Latency (ns) Header min | Header max | Body or Tail min | Body or Tail max | Data Rate (Mflits/s) Head | BT |
|---|---|---|---|---|---|---|---|---|---|
| Fast 4-Phase | 400 | 272 | $1.84x10^{40}$ | 5.897 | 8.454 | 2.473 | 5.027 | 132 | |
| | 600 | 397 | $4.12x10^{19}$ | 4.221 | 5.906 | 1.632 | 3.313 | 198 | |
| | 800 | 531 | $2.05x10^{9}$ | 4.732 | 8.475 | 2.525 | 6.277 | 264 | |
| Pipeline | 400 | 400 | $5.63x10^{52}$ | 4.776 | 4.805 | 3.75 | 3.78 | 800 | 400 |
| | 600 | 600 | $6.48x10^{21}$ | 3.954 | 7.663 | 2.928 | 2.955 | 954 | 600 |
| | 800 | 800 | $5.55x10^{13}$ | 5.207 | 10.02 | 4.172 | 6.576 | 956 | 640 |
| Gradual | 400 | 400 | $2.04x10^{51}$ | 2.625 | 2.656 | 1.549 | 1.586 | 800 | 400 |
| | 600 | 600 | $6.20x10^{20}$ | 2.839 | 6.654 | 1.759 | 2.254 | 952 | 600 |
| | 800 | 800 | $2.47x10^{12}$ | 2.997 | 7.971 | 1.973 | 4.836 | 948 | 639 |

TABLE II
INCOMING MESSAGE NI SIMULATION RESULTS

| Sync Type | RX Clock (MHz) | Network (MHz) | MTBF (years) | Latency (ns) min | Latency (ns) max | Data Rate (Mflits/s) |
|---|---|---|---|---|---|---|
| Fast 4-Phase | 400 | 271 | $1.14x10^{40}$ | 5.12 | 7.01 | 132 |
| | 600 | 395 | $3.93x10^{19}$ | 4.51 | 5.096 | 198 |
| | 800 | 527 | $1.95x10^{9}$ | 3.26 | 3.64 | 264 |
| Pipeline | 400 | 400 | $3.99x10^{51}$ | 5.00 | 9.13 | 400 |
| | 600 | 600 | $1.37x10^{21}$ | 4.38 | 5.83 | 600 |
| | 800 | 800 | $6.02x10^{12}$ | 3.82 | 4.31 | 800 |
| Gradual | 400 | 400 | $3.98x10^{51}$ | 2.56 | 6.53 | 400 |
| | 600 | 600 | $1.22x10^{21}$ | 2.54 | 3.31 | 600 |
| | 800 | 800 | $6.02x10^{12}$ | 2.65 | 3.06 | 800 |

the PS NI. Latencies of the GSync NI are close to or less than those of the F4PH NI. Since there are at least two flits for every incoming message, the throughput capability makes the GSync NI the clear choice for this scenario.

## V. CONCLUSION

This paper presents a methodology for performing synchronization in parallel with pipelined computation called gradual synchronization. Components designed using the method can interface synchronous or asynchronous domains of an on-chip system. High operating frequency simulations verify high throughput capability, with up to 37 percent reduction of forward latency as compared to previous high throughput methods. The method has the potential for even greater latency reduction at lower operating frequencies. Our implementation of a NI using gradual synchronization demonstrates that the method can be applied to a real-world problem and that it results in a significant benefit in many cases.

In the future we would like to create a mixed synchronous asynchronous pipeline architecture with gradual synchronization. In this case the nature of the gradual synchronizer may allow fine grained application of asynchronous circuits to benefit specific processor components.

## REFERENCES


[1] L. Lamport. *Buridan's Principle*, Digital Systems Research Center Report, 1984.

[2] T. J. Chaney and C. E. Molnar. "Anomalous Behavior of Synchronizer and Arbiter Circuits", *IEEE Trans. on Comput.*, pp.421-422, 1973.

[3] C. L. Seitz. "System Timing," Chapter 7 in *Introduction to VLSI Systems* by Carver Mead and Lynn Conway, Addison-Wesley, 1980.

[4] F. U. Rosenberger, C. E. Molnar, T. J. Chaney, and T. Fang. Q-Modules: Internally Clocked Delay-Insensitive Modules. *IEEE Trans. on Comput.*, **37**(9):1005–1018, 1988.

[5] W. J. Dally and J. Poulton. *Digital Systems Engineering*, Cambridge University Press, 1998.

[6] R. Ginosar. "Fourteen Ways to Fool your Synchronizer", *Proc. IEEE Int. Symp. Asynchronous Circuits and Systems* , 2003.

[7] R. Dobkin and R. Ginosar. "Fast Universal Synchronizers", *Workshop on Power and Timing Modeling, Optimization, and Simulation (PATMOS)*, 2008.

[8] W. J. Dally and S. G. Tell. "The Even/Odd Synchronizer: A Fast, All-Digital, Periodic Synchronizer," *Proc. IEEE Int. Symp. Asynchronous Circuits and Systems*, 2010.

[9] J. Seizovic. "Pipeline Synchronization", *Proc. Int. Symp. Asynchronous Circuits and Systems*, 1994.

[10] T. Chelcea and S. Nowick. "Robust Interfaces for Mixed-Timing Systems with Application to Latency-Insensitive Protocols", *Proc. Design Automation Conf.*, 2001.

[11] L. R. Dennison, W. J. Dally, and D. Xanthopolous. "Low-latency plesiochronous data re-timing", *Proc. 16th Conf. Advanced Research in VLSI*, 1995.

[12] L. F. G. Sarmenta, G. A. Pratt, and S. A. Ward. "Rational Clocking", *Proc. of ICCD*, 1995.

[13] U. Frank and R. Ginosar. "A Predictive Synchronizer for Periodic Clock Domains", *Workshop on Power and Timing Modeling, Optimization, and Simulation (PATMOS)*, 2004.

[14] M. Nyström, R. Manohar, and A. J. Martin. Method and Apparatus for a failure-free synchronizer. US Patent No. 6,609,203, February 2004.

[15] R. Manohar, M. Nyström, and A. J. Martin. "Precise Exceptions in Asynchronous Processors", *Proc. 19th Conf. Advanced Research in VLSI*, March 2001.

[16] S. Jackson, "Gradual Synchronization", Ph.D. dissertation, Dept. Elect. Comp. Eng., Cornell Univ., Ithaca, NY, 2014.

[17] R. Dobkin, R. Ginosar, and A. Kolodny. "QNoC Asynchronous Router", *INTEGRATION, the VLSI Journal*, vol.42, pp.103-115, 2009.