

Verifying Synchronization Strategies

Chris J. Myers^{*1}, Eric Mercer², and Hans Jacobson³

¹ University of Utah
myers@vlsigroup.ece.utah.edu

² Brigham Young University
egm@cs.byu.edu

³ IBM T.J. Watson Research Center
hansj@us.ibm.com

Abstract. Over the years, there have been numerous methods proposed to solve the synchronization problem. Many of these methods, however, are not sufficiently evaluated before being utilized leading to problems in a system design that are difficult to diagnose and solve. Therefore, it is crucial that strategies for synchronization are critically analyzed and verified before being incorporated in a system design. This paper reviews a number of the known methods for synchronization, discusses issues in their design, and presents techniques for their verification.

1 The Synchronization Problem

While there have been many promising asynchronous design examples [48, 4, 17, 42, 1], asynchronous design is still not seeing widespread use. One important reason for this is that asynchronous designs must communicate with other parts of the system which typically operate synchronously. Unfortunately, this is difficult to do reliably without substantial latency penalties. When this latency penalty is taken into account, most, if not all, of the performance advantage gained by an asynchronous design is lost. Even if no asynchronous modules are used, synchronous modules operating at different clock rates or out of phase can have the same problem. The latter problem is becoming more significant as it becomes increasingly difficult to distribute a single global clock to all parts of the chip. Many designers today are considering the necessity of having multiple clock domains on a single chip, and they will need to face this problem.

A synchronization problem can occur when a synchronous circuit must synchronize an asynchronous input. This can be done using a single D-type flip-flop as shown in Figure 1(a). However, if the clock edge arrives too close in time to data arriving from an asynchronous circuit, the circuit may enter a metastable state in which its output is at neither a logic 0 or logic 1 level, but rather, lies somewhere in between. This behavior is depicted in Figure 1(b). Assume that Q is initially low and that D has recently gone high. If D goes low again at about the same time that CLK rises, the output Q may start to rise and then get stuck between the logic levels as it observes D falling. Should Q rise or fall? Actually, either answer would be okay, but the flip-flop becomes indecisive. At some point, Q may continue to a logic 1 level, or it may drop to the logic 0 level. When this happens, however, is theoretically unbounded. If during this period of indecision, a

* This research is supported by SRC contract 2002-TJ-1024.

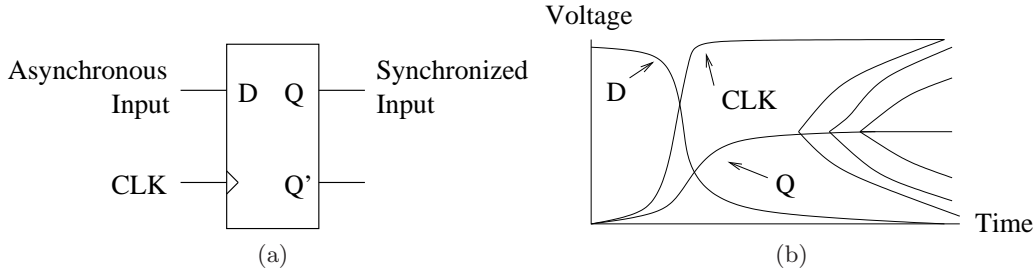


Fig. 1. (a) Simple, dangerous synchronizer. (b) Oscilloscope view of metastable behavior.

circuit downstream from this flip-flop looks at the synchronized input, it sees an indeterminate value. This value may be interpreted by different subsequent logic stages as either a logic 0 or a logic 1. This can lead the system into an illegal or incorrect state, causing the system to fail. Such a failure is traditionally called a *synchronization failure*. If care is not taken, any asynchronous communication between synchronous modules can lead to an unacceptable probability of failure.

The synchronization problem has been known for some time. The earliest paper we have found on asynchronous design addresses it [23]. In Lubkin’s 1952 paper, he makes the following comment about the synchronization problem:

If an entity similar to one of Maxwell’s famous demons were available to convert such “maybe’s” to either “yes” or “no,” even in arbitrary fashion, the problem would be solved.

He goes on to discuss how the designers of the ENIAC used additional flip-flops to allow more time to synchronize asynchronous inputs. This paper states that this technique does not eliminate the chance of error, but rather, it simply reduces its probability. This paper also presents a method of determining the probability of error. The problem appears to be largely ignored until 1966 when Catt rediscovered it and presents a different formulation of this error probability [5]. Again, it appears that the synchronization problem was not widely known or understood. Evidence of this is that several asynchronous arbiters designed in the early 1970s suffered from metastability problems if arrival times of signals are not carefully controlled [9, 10, 31, 33].

Finally, in 1973 experimental evidence of the synchronization problem presented by Chaney and Molnar appears to have awakened the community to the problem [6]. After this paper, a number of papers were published that provided experimental evidence of metastability due to asynchronous inputs, and mathematical models were developed to explain the experimental results [11, 12, 14, 18, 21, 32, 34, 46]. Pechoucek’s paper also shows that the only way to reduce the probability to zero is to generate the clock locally and be able to stop the clock when metastability occurs. Over the years, there have also been several proofs that show

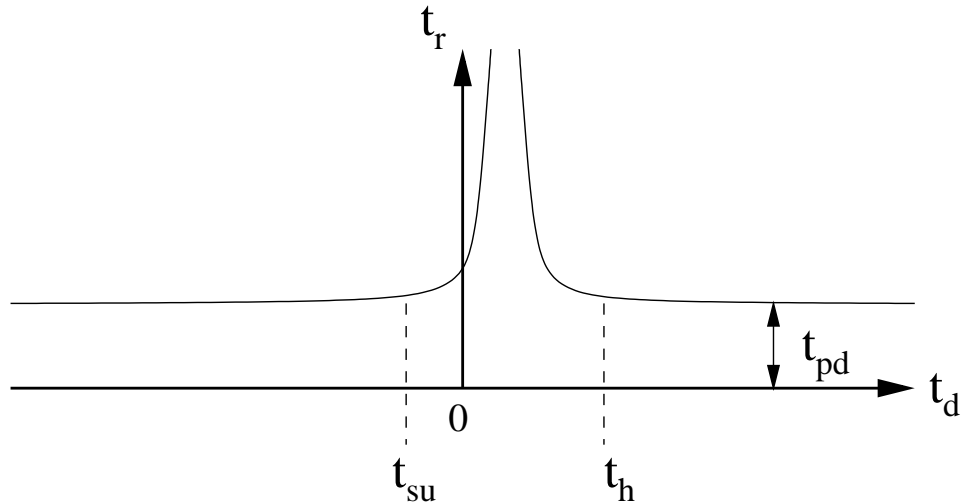


Fig. 2. Flip-flop response time as function of input arrival time in relation to clock arrival time (clock arrives at 0).

that metastability in a bistable is unavoidable [15, 25, 26, 20] and that it is related to other issues such as arbitration and hazards [3, 44].

2 Probability of Synchronization Failure

Most synchronization strategies being employed today have a probability of failure. An acceptance of this fact and a careful analysis of this probability is crucial in designing a reliable system. An excellent discussion of the synchronization problem, ways to reduce the probability of error, and the use of stoppable clocks is given by Stucki and Cox [43]. The mathematical treatment below follows this paper. Figure 2 shows a representative plot based on measured data for the response time of a flip-flop as a function of the arrival time of data, t_d , with respect to the clock. If data only changes before the *setup time*, t_{su} , and after the *hold time*, t_h , of a flip-flop, the *response time*, t_r , is roughly constant and equal to the *propagation delay* through the flip-flop, t_{pd} . If, on the other hand, the data arrives between the setup and hold times, the delay increases. In fact, if the data arrives at just the absolutely wrong time, the response time is unbounded.

If data can arrive asynchronously with respect to the clock, we can consider that it arrives at a time which is uniformly distributed within the clock cycle. Therefore, the probability that the data arrives at a time t_d which falls between t_{su} and t_h is given below.

$$P(t_d \in [t_{su}, t_h]) = \frac{t_h - t_{su}}{T} \quad (1)$$

where T is the length of the clock period. If we assume that the flip-flop is given some bounded amount of time, t_b , to decide whether or not to accept the newly arrived data, the probability of a synchronization

failure is related to the probability that the response time, t_r , exceeds t_b . In the metastable region between t_{su} and t_h , it can be shown that the response time increases in an approximately exponential fashion. Therefore, if t_d falls in this range, the probability that $t_r > t_b$ can be expressed as follows:

$$P(t_r > t_b \mid t_d \in [t_{su}, t_h]) = \frac{1}{k + (1 - k)e^{(t_b - t_{pd})/\tau}} \quad (2)$$

where k and τ are circuit parameters, with k being a positive fraction less than 1 and τ being a time constant with values on the order of a few picoseconds for modern technologies. Combining Equations 1 and 2 using Bayes rule, we get:

$$P(t_r > t_b) = P(t_d \in [t_{su}, t_h]) \cdot P(t_r > t_b \mid t_d \in [t_{su}, t_h]) \quad (3)$$

$$= \frac{t_h - t_{su}}{T} \cdot \frac{1}{k + (1 - k)e^{(t_b - t_{pd})/\tau}} \quad (4)$$

If $t_b - t_{pd} \geq 5\tau$, Equation 3 can be simplified as follows:

$$P(t_r > t_b) \approx \frac{t_h - t_{su}}{T} \cdot \frac{e^{-(t_b - t_{pd})/\tau}}{1 - k} \quad (5)$$

By combining constants, Equation 5 can be changed to

$$P(t_r > t_b) \approx \frac{T_0}{T} \cdot e^{-t_b/\tau} \quad (6)$$

Equation 6 is convenient since there is only two circuit-dependent parameters T_0 and τ that need to be determined experimentally. These parameters appear to scale linearly with feature size. Equation 6 has been verified experimentally and found to be a good estimate as long as t_b is not too close to t_{pd} . It is important to note that there is no finite value of t_b such that $P(t_r > t_b) = 0$. Therefore, the response time in the worst-case is unbounded.

A *synchronization error* occurs when t_r is greater than the time available to respond, t_a . A synchronization failure occurs when there is an inconsistency caused by the error. Failures occur less often than errors since a consistent interpretation still may be made even when there is an error. The expected number of errors is

$$E_e(t_a) = P(t_r > t_a) \cdot \lambda \cdot t \quad (7)$$

where λ is the average rate of change of the signal being sampled and t is the time over which the errors are counted. If we assume all errors are failures, set $E_e(t_a)$ to 1, change t to MTBF (*mean time between failure*), substitute Equation 6 for $P(t_r > t_a)$, and rearrange Equation 7, we get

$$\text{MTBF} = \frac{T \cdot e^{t_a/\tau}}{T_0 \cdot \lambda} \quad (8)$$

This equation increases rapidly as t_a is increased. Therefore, even though there is no absolute bound in which no failure can ever occur, there does exist an engineering bound in which there is an acceptably low likelihood of error.

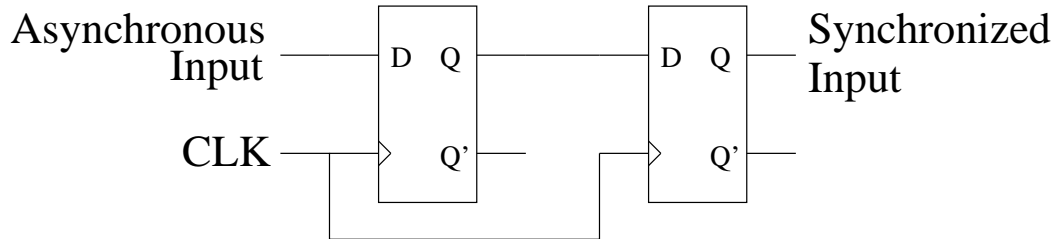


Fig. 3. Double latch solution to reduce synchronization failure.

3 Reducing the Probability of Failure

Many techniques have been devised to address the metastability problem and reduce the probability of synchronization failure to an acceptable level when interfacing between synchronous and asynchronous modules. The goal of each of these techniques is to increase the amount of time to resolve the metastability (i.e., increase t_a). The simplest approach to achieve this is to use two (or more) latches in series as shown in Figure 3 to sample asynchronous signals arriving at a synchronous module. This increases the time allowed for a metastable condition to resolve. In other words, if n extra latches are added in series with an asynchronous input, the new value of t_a is given by

$$t'_a = t_a + n(T - t_{pd}) \quad (9)$$

where T is the clock period and t_{pd} is the propagation delay through the added flip-flops. The cost, though, is an extra n cycles of delay when communicating data from an asynchronous module to a synchronous module, even when there is no metastability. This scheme also only minimizes the probability and does not eliminate the possibility of synchronization failure, as there is still some chance that a metastable condition could persist longer than n clock cycles.

Consider a simple example for perspective. Assume that τ is measured to be about 20 ps and T_0 about 8 ns. If the clock frequency is 2 GHz, T is 500 ps. If asynchronous inputs are coming at an average rate of 1 GHz, λ is 10^9 samples per second. Let us also assume that we can tolerate a metastability for four-fifths of the clock period or $t_a = 400$ ps. Using Equation 8 we find the mean time between failures to be only 30 ms! If the propagation delay through a flip-flop is 120 ps and we add a second latch, then t_a becomes 780 ps, and the mean time between failure becomes about 63 days. If we add a third flip-flop, the mean time between failure increases to over 30 million years!

Many designs have been proposed for synchronizers based upon the concept described above. Veendrick shows that the probability of metastability is independent of circuit noise in the synchronizer and that it could be reduced somewhat through careful design and layout [46]. Kleeman and Cantoni show that using

redundancy and masking does not eliminate the probability of synchronization failure [19]. Manner describes how *quantum synchronizers* can solve the problem in principle, but not in practice [24]. Sakurai shows how careful sizing can reduce the probability of failure in synchronizers and arbiters [36]. Walker and Cantoni recently published a synchronizer design which uses a bank of parallel rather than serial registers operating using a clock period of nT [47]. Another interesting design is published in [8] in which EMI caused by the clock is exploited to build a better synchronizer. One of the most interesting synchronizers was due to Seizovic, in which he proposes to pipeline the synchronization process [39]. *Pipeline synchronization* essentially breaks up the synchronization into a series of asynchronous pipeline stages which each attempt to synchronize their request signal to the clock. When metastability occurs in one stage, its request to the next stage is delayed. When the next stage sees the request, it sees it at a somewhat different time, and it hopefully does not enter the metastability region. As the length of the pipeline is increased, the likelihood that metastability persists until the last stage is greatly reduced. This scheme is used in the Myranet local area network. This network at the time had about 80 million asynchronous events per second with $\tau = 230$ ps. If the synchronous clock rate being synchronized to is at 80 MHz, the MTBF is on the order of 2 hours. Using an eight-stage pipeline for synchronization and a two-phase clock, a latency of 50 ns is incurred in which 28 ns is available for synchronization reducing the MTBF to about 10^{37} years.

4 Eliminating the Probability of Failure

To eliminate synchronization failures completely, it is necessary to be able to force the synchronous system to wait an arbitrary amount of time for a metastable input to stabilize. In order for the synchronous circuit to wait, it is necessary for the asynchronous module to be able to cause the synchronous circuit's clock to stop when the asynchronous module is either not ready to communicate new data or not ready to receive new data. A *stoppable clock* can be constructed from a gated ring oscillator as shown in Figure 4(a). Stoppable clocks date back to the 1960s with work done by Chuck Seitz which was used in early display systems and other products of the Evans and Sutherland company [38, 41].

The basic operation of a stoppable clock is that when the *RUN* signal is activated, the clock operates at a nominal rate set by the number of inverters in the ring. To stop the clock, the *RUN* signal must be deactivated between two rising clock edges. The clock restarts as soon as the *RUN* signal is reactivated. In other words, the clock should be stopped synchronously and is restarted asynchronously. If the synchronous module decides when it needs data from the asynchronous module, the behavior is as follows. When the synchronous module needs data from an asynchronous module, it can request the data on the rising edge of the clock and in parallel set *RUN* low. If you want a guaranteed high pulse width, then *RUN* must be set

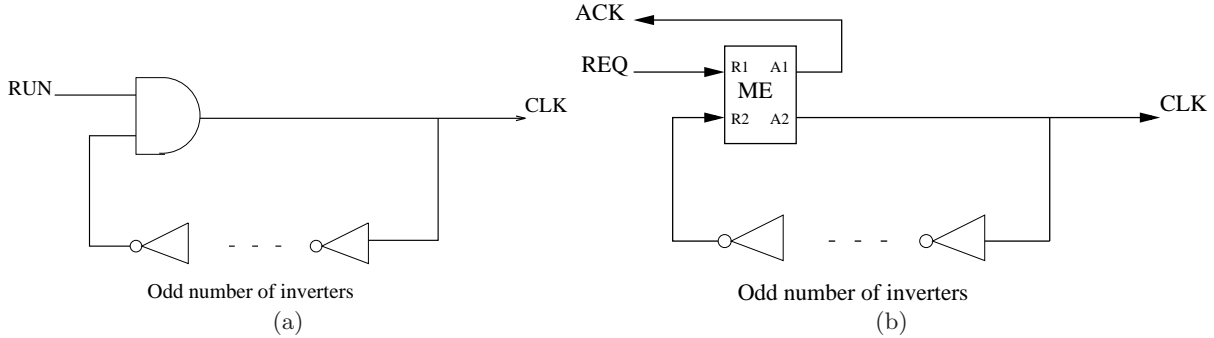


Fig. 4. (a) Stoppable ring oscillator clock. (b) Stoppable ring oscillator clock with ME.

low on the falling clock edge. When the data arrives from the asynchronous module, the acknowledgment from this module can be used to set RUN high. If RUN is set high before the end of the clock cycle, the next clock cycle can begin again without delay. If on the other hand, the asynchronous module is slow in providing data, the low phase of CLK is stretched until the data arrives.

If the asynchronous module can decide when to send data, a *mutual exclusion* (ME) element is needed as shown in Figure 4(b) to guarantee that a synchronous module either receives data from an asynchronous unit or a pulse from the clock generator, but never both at the same time. If the asynchronous data arrives too close to the next clock pulse, both the data and the clock pulse may be delayed waiting for the metastability to resolve before determining which is to be handled first. An ME element has two inputs, $R1$ and $R2$, and two outputs, $A1$ and $A2$. It can receive rising transitions on both inputs concurrently, but it responds with only a single rising transition on one of the corresponding outputs. There are three possible situations. The first is that the asynchronous module does not request to send data during this clock cycle. In this case, the ME simply acts as a buffer and the next rising clock edge is produced. The second case is the asynchronous request comes before the next rising clock edge is needed. In this case, the ME issues an ACK to the asynchronous module, and it prevents the next clock cycle from starting until REQ goes low. The third case is that REQ goes high just as CLK is about to rise again. This causes a metastable state to occur, but the ME is guaranteed by design to either allow the asynchronous module to communicate by setting ACK high and stopping the clock or by refusing to acknowledge the asynchronous module this cycle and allowing CLK to rise. Note that theoretically it may do neither for an unbounded amount of time.

A circuit diagram for a CMOS ME element is shown in Figure 5. When this circuit goes metastable, $V1$ and $V2$ differ by less than a threshold voltage, so $T1$ and $T2$ are off. Therefore, both $A1$ and $A2$ remain low. Once $V1$ and $V2$ differ by more than a threshold voltage, either $T1$ or $T2$ turns on, pulling up its corresponding output.

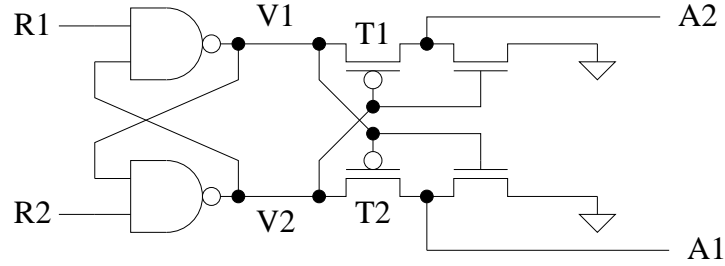


Fig. 5. Circuit for mutual exclusion.

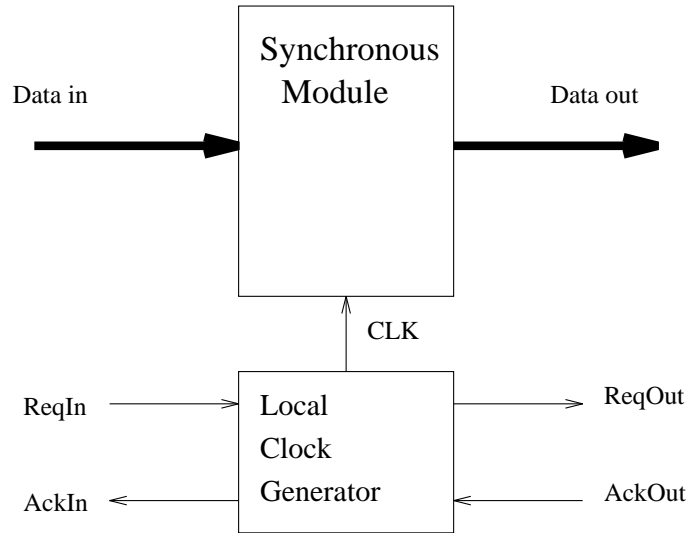


Fig. 6. Basic module of a GALS architecture.

A stoppable clock can be used to design a *globally asynchronous locally synchronous* (GALS) architecture. Communication between modules is done asynchronously using request/acknowledge protocols while computation is done synchronously within the modules using a locally generated clock. The basic structure of such a module is shown in Figure 6. The module's internal clock is stopped when it must wait for data to arrive from, or to be accepted by, other modules. If another module can request to communicate data to a synchronous module at arbitrary times as discussed above, the stoppable clock shown in Figure 4(b) is needed. If the synchronous unit determines when data is to be transferred to/from the other modules, there is no need for a ME element, since the decision to wait on asynchronous communication is synchronized to the internal clock. In this case, the stoppable clock shown in Figure 4(a) can be used.

Numerous researchers have developed GALS architectures based on the idea of a stoppable clock [7, 13, 35, 38, 43, 45, 49]. Some of the schemes such as those proposed in [22, 35, 43, 49] allow an asynchronous module to request to communicate data to a synchronous module at arbitrary times. The approach in [35] is based on an asynchronous synchronizer called a *Q-flop*. This synchronizer receives a potentially unstable input and

a clock, and it produces a latched value and an acknowledgment when it has latched the value successfully. Q-flops are used in *Q-modules* which are essentially synchronous modules clocked using a locally generated stoppable clock. These Q-modules are then interconnected asynchronously. The schemes proposed in [7, 38] assume that the synchronous unit determines when data is to be transferred to/from the asynchronous modules. Recently, a group from IBM introduced a new GALS approach of the second kind called interlocked pipelined CMOS (IPCMOS) [37]. They implemented a test chip in a 0.18 μm 1.5 V CMOS process which consisted of the critical path from a pipelined floating-point multiplier. Their experimental results showed a typical performance of 3.3 GHz with a best-case performance of 4.5 GHz. An analysis comparing synchronous versus GALS is given in [2].

5 Globally Synchronous Locally Asynchronous

The following two sections describe synchronization strategies developed in our research group and the methods that we used to verify their correctness. Sjogren proposed a *globally synchronous locally asynchronous* architecture shown in Figure 7 [40]. One possible approach to increasing a synchronous, pipelined microprocessor's speed is to replace the slowest pipeline stages with asynchronous modules that have a good average-case performance. If the interfacing problem can be addressed, this allows a performance gain without redesigning the entire chip. While the entire system communicates synchronously, one or more local modules may compute asynchronously. In other words, the system is globally synchronous locally asynchronous.

This interface methodology, while similar to the GALS architecture, allows for stages in high-speed pipelines to be either synchronous or asynchronous. The architecture shown in Figure 7 assumes true single-phase clocking configured in such a way that data is latched into the next stage on the rising edge of the clock. The *CLK* signal is generated using a stoppable ring oscillator. Besides being used to sequence data between pipeline stages, the *CLK* signal is also used to generate the handshake protocol that controls the asynchronous modules. The interface controller is composed of the stoppable clock generator, one handshake control circuit for each asynchronous module, and an *AND* gate to collect the *ACK* signals to generate the *RUN* signal. The circuit behavior of the interface controller is as follows. Shortly after the rising edge of the *CLK* signal, the *RUN* signal is set low. The *RUN* signal is set high again only after all the asynchronous modules have completed their computation. Since data moves in and out of each asynchronous module with every cycle in the pipeline, no mutual exclusion elements are necessary.

The interface controller is composed of two sections: the stoppable clock and the handshake controllers. The stoppable clock used is shown in Figure 8, which is somewhat different from the traditional one shown in Figure 4(a). Rather than using an *AND* gate to control the starting and stopping of the clock, a state-holding

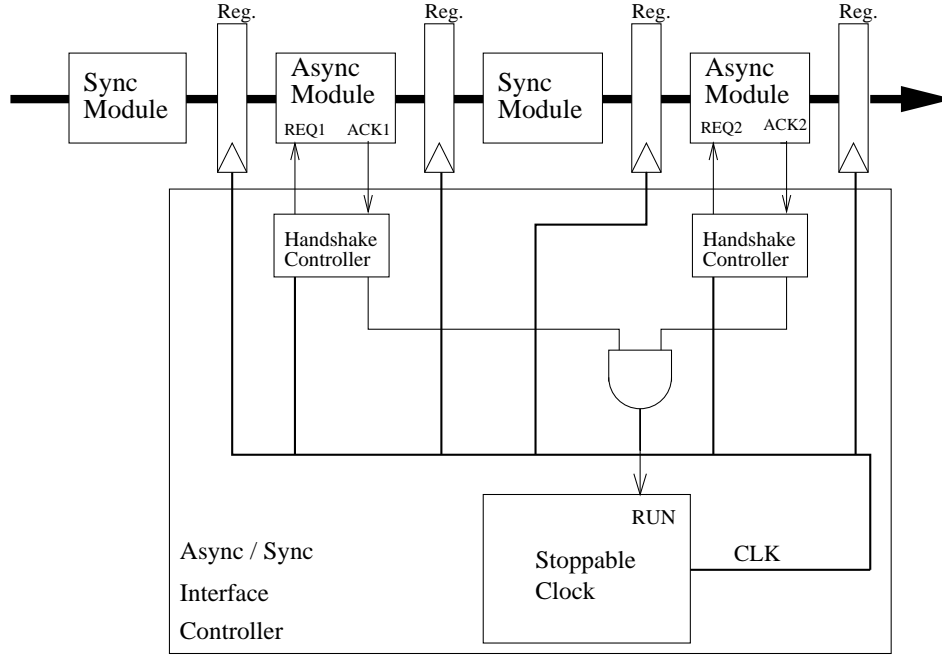


Fig. 7. Architecture for a globally synchronous locally asynchronous system.

gate called a *generalized C-element* is used. This gate sets CLK high when both RUN and $PRECLK$ are high, and it resets CLK when $PRECLK$ is low. When synthesizing this gate, we discovered a timing assumption in the original design that requires the RUN signal to be active until CLK goes low. In order to give more flexibility in setting and resetting the RUN signal, we decided to remove this timing assumption, resulting in the gate shown in Figure 8. In our implementation, the RUN signal can be deactivated at any time after CLK goes high until just before the next rising clock edge. A similar observation is made in [41], and they add a pair of cross-coupled $NAND$ gates to latch the clock in one of their designs. The overhead of the cross-coupled $NAND$ gates is minimized in our approach by implementing the circuit at the transistor-level. The rest of the stoppable clock is a ring oscillator composed of inverters and one $NAND$ gate which is used to set CLK to low during reset. The number of inverters is set such that the delay through the ring oscillator is greater than the worst-case path through the slowest synchronous module.

The second part of the interface controller is the handshake control circuit. There is one of these controllers for each asynchronous module. The controller is used to translate the CLK signal into a four-phase handshake with the asynchronous module. In a typical four-phase handshake with an asynchronous datapath element, the signal REQ is asserted high when there is valid data on the inputs and computation is started. The ACK signal goes high to indicate that computation has completed, and there is valid data on the outputs. When REQ is set low, the asynchronous module typically resets. One efficient way to implement an asynchronous datapath is to use *domino dual-rail logic*, in which REQ low precharges the logic. When the precharge

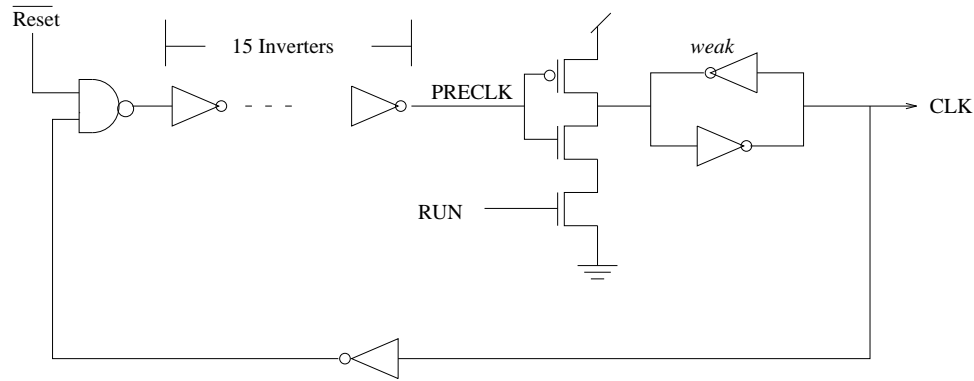


Fig. 8. The GSLA stoppable ring oscillator clock.

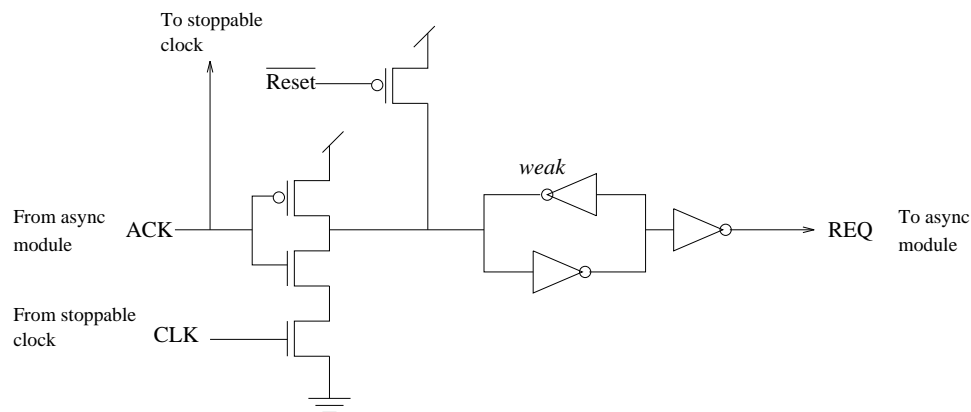


Fig. 9. Basic handshake control circuit.

is completed, the *ACK* signal would go low. This precharge stage eliminates the results of the previous computation, so it should not be done until the data has been latched into the next pipeline stage. Since data is latched into the next stage on the rising edge of the clock, the handshake control circuit should hold *REQ* high until *CLK* goes high to keep the data from the previous calculation stable. After *CLK* goes high, we set *REQ* low to begin the precharge stage. When *ACK* has gone low, the precharge stage has completed, and we can begin the computation by setting *REQ* high. The handshake control circuit is shown in Figure 9.

If we assume that the precharge stage has completed before *CLK* goes low, we could simply use the *CLK* signal as the *REQ* signal. This, however, incurs a performance penalty. Typically, the precharge stage is only a couple of gate delays while the computation stage takes significantly longer. By using *ACK* to generate *REQ* rising, our circuit allows computation to start immediately after precharge completes, which gives the computation more time to complete. This is a significant improvement over traditional synchronous domino-logic design, which wastes half a clock cycle for precharge. Synchronous designers have also noticed this, and they often do what is called “cycle-stealing” to improve performance.

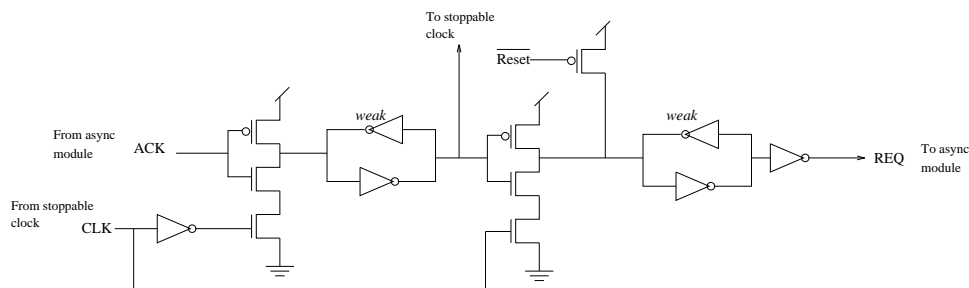


Fig. 10. Handshake control circuit with conditioned acknowledgment.

One may also wonder why there is a second n-transistor gated with the ACK signal in Figure 9. As mentioned above, CLK may not be low when ACK goes low since the precharge stage typically completes very quickly. This transistor cuts off the n-stack when ACK goes low, so there is no fight allowing REQ to go low as early as possible. Note that since CLK cannot go high before ACK goes high, the falling transition of REQ is always triggered by the rising transition of CLK .

There is one other timing assumption which requires the CLK signal to go low before both the precharge and computation stages complete. Otherwise, it is possible that the precharge stage would be reentered, destroying the results of the computation. We believe this to be a reasonable timing assumption. If this timing assumption does not hold, an additional gate can be added between the ACK signal generated by the completion logic and the signal used by the interface control circuits. The handshake control circuit with the conditioned acknowledgment signal is depicted in Figure 10. The additional gate prevents the rising transition of ACK from being seen before CLK goes low. If the timing assumption holds, this gate should be omitted since it adds extra circuitry and delay on the critical path.

The basic operation of the interface controller is depicted as an idealized waveform shown in Figure 11. For simplicity, we assume there is one asynchronous module, so ACK and RUN are the same signal. Initially, \overline{RESET} is asserted low, which sets the CLK signal low and REQ signal high. With the REQ signal high, the asynchronous datapath module eventually sets ACK high during reset. Each cycle after \overline{RESET} is deasserted, the interface controller sets CLK high, which latches the data for each pipeline stage and causes the asynchronous modules to precharge by asserting the REQ signal low. When an asynchronous module completes precharge, it sets its ACK signal low. After ACK has gone low, the computation can be started by asserting REQ high. When an asynchronous module completes computation, it asserts its ACK signal high. Note that the computation can start anywhere in the clock cycle, but it must not complete before CLK goes low. During precharge and computation, the CLK signal goes low and prepares to go high. If any of the asynchronous modules have not asserted their ACK signal, the rising edge of the CLK is delayed until all the asynchronous modules have completed their computation.

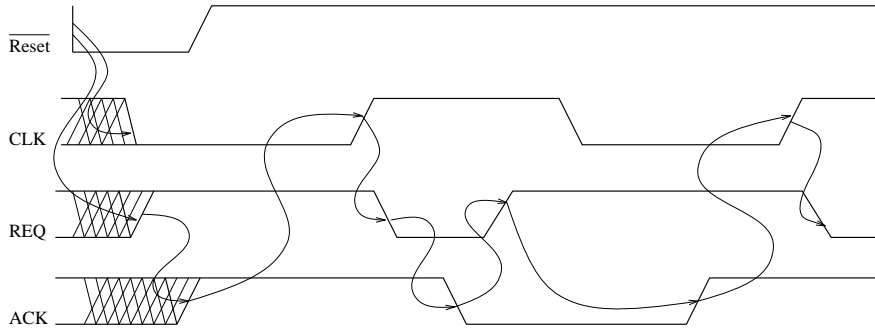


Fig. 11. Idealized waveform for the basic interface controller.

The timing assumptions just discussed are determined using the *timed circuit* synthesis and verification tool ATACS [30]. Timed circuits are a class of circuits in which specified timing information is utilized in the design procedure to optimize the implementation. The resulting circuit is not only more efficient, but also verified correct under the given timing assumptions by ATACS. Since the circuits in the interface controller are highly time dependent, they cannot be designed using traditional untimed asynchronous design methods. The ATACS tool accepts a model written in a hardware description language, such as VHDL. When written in VHDL, a handshaking package described in [28] is utilized. This is then compiled into an intermediate format called a *level-ruled Petri net* (LPN) [27]. The behavior of the interface controller using the basic handshake control circuit is formally modeled using a LPN as shown in Figure 12 (note that models of *req2* and *ack2* are omitted as they are similar to *req1* and *ack1*, respectively). In the following discussion, all signals are initially high except *clk*.

In Figure 12, each signal in the interface controller is modeled as a separate process. For example, the model of the *clk* signal is shown in Figure 12(a). This model shows that *clk* can go high 1 to 3 time units after *run* and *preclk* are high which is true in the initial state. After *clk* goes high, Figure 12(b) shows that *req1* (and also *req2*) can go low. Note that *preclk* is also enabled to go low, but its minimum delay of 19 time units is much more than the maximum fall time of *req1* which is 3 time units. After *req1* goes low, as shown in Figure 12(c), *ack1* can go low in 6 to u time units. The value of u is important as described later. After either *ack1* or *ack2* goes low, *run* can go low as shown in Figure 12(d). Here is where it gets interesting. Sometime after *clk* went high, *preclk* goes low which in turn sets *clk* low as shown in Figures 12(a) and (e). Also, in parallel, *ack1* and *ack2* may go high at some point indicating that the asynchronous modules have completed. As described earlier, the exact timing of these events is important. If it is possible, for *ack1* or *ack2* to go high before *clk* goes low, then the handshake control circuit with conditioned acknowledgment shown in Figure 10 is required. This occurs when l in Figure 12(c) is less than 17 time units. Assuming that l is 17 time units or greater, the value of u must be less than $l + 4$ time units. If it is greater than this, it

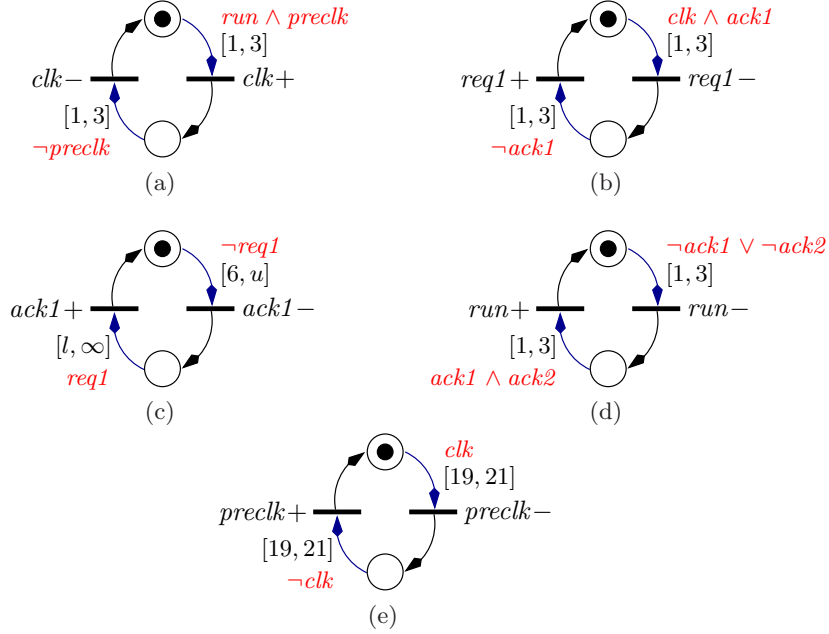


Fig. 12. Level-ruled Petri net models for: (a) the stoppable clock, (b) the handshake control circuit, (c) the asynchronous module, (d) the *run* signal, and (e) the ring oscillator.

is possible for one of the two acknowledgment signals to go low, causing *run* to go low, and then go high again enabling *run* to go high erroneously since both asynchronous modules have not completed. Another interesting fact about u is if it is 16 time units or less, then the handshake control circuit that generates *req1* is reduced to a NAND gate with inputs of *ack1* and *clk*. Now, after *clk* goes low, *preclk* goes high again. At this point, *clk* does not go high unless *run* has gone high. The *run* signal only goes high after *ack1* and *ack2* have both gone high. Therefore, if either of the asynchronous modules have not completed, the clock stops.

The type of analysis just described is readily accomplished using the ATACS tool. The circuit is verified under a number of different timing scenarios. Whenever there is the potential for a timing violation such as a hazard on one of the gates, an error trace is reported to the user. This error trace can then be used to determine the cause of the problem.

6 Interlocked Synchronous Pipelines

While asynchronous and GALS techniques hold promise in alleviating global synchronization in future system-on-chips, the challenge of folding such techniques into existing design methodologies remains. To make benefits of interlocking more immediately available, *interlocked synchronous pipelines* [16] have been developed at IBM. Such pipelines provide a stepping stone towards the integration of asynchronous techniques into synchronous circuits by providing interlocking based on synchronous, rather than asynchronous,

handshakes. IBMs' interlocking pipeline takes advantage of the redundant data storage capacity of synchronous master/slave based pipelines to provide properties such as progressive stalling and optimal clock gating normally only found in asynchronous pipelines.

The circuits in the previous section though tricky are still quite small. While it is conceivable that manual analysis could have been employed in this design, for a more concurrent and complex synchronization strategy, automated verification methods are essential. This section illustrates our timing verification methodology on a substantially larger and more complex synchronization strategy in the form of IBM's synchronous interlocked pipelines.

A typical synchronous pipeline stage is active on every clock cycle even if data is not present. A significant amount of power is thus dissipated for no reason at the leaf nodes of the clock tree where the data latches reside. Asynchronous pipelines overcome this by interlocking control signals in the forward and backward directions of the pipeline to activate stages on demand. A stage cannot forward data to the next stage unless the stage has data to forward and the next stage is ready to accept that data. This forward and backward interlock in the pipeline prevents wasteful power dissipation at the data latches.

The synchronous interlocked pipeline from IBM interlocks at the stage level in a manner similar to asynchronous pipelines, only the interlock is achieved using the global clock signal in conjunction with stage level control signals. The forward interlock in a synchronous pipeline is readily implemented by a valid bit that moves forward in the pipeline with valid data. Stages activate only if the valid bit is set. The backward interlock is less obvious because it must implement a progressive stall. A progressive stall works stage by stage, moving from one stage to its predecessor on each clock edge until it reaches the front of the pipeline. It implies that if there are N data items in a full pipeline, then it takes $2N$ storage locations to progressively stall the pipeline assuming that data are inserted on each clock edge until the first stage is stalled. Without a progressive stall, a synchronous pipeline cannot be interlocked in the backward direction.

An *elastic synchronous pipeline*, shown in Figure 13, implements the progressive stall and creates the backward interlock. The pipeline relies on two-phase clocking for correct operation to store data in every other stage of the pipeline; thus, an N -stage pipeline holds $\frac{N}{2}$ data items when it is full. The pipeline illustrated in Figure 13 is equivalent to a synchronous split-latch master/slave pipeline. In a split-latch pipeline, combinational logic resides both between master and slave, as well as, slave and master latches. Subsequently, odd stages in the illustrated pipeline correspond to the master latch stages, and the even stages to the slave latch stages, of a positive edge triggered master/slave pipeline. The *stall* signal is propagated backward in the pipeline by one-bit latches and moves backward one stage on each edge of the two-phase clock. The clock inputs to the data latches are gated by the respective stall signals at each stage. Note that

it takes N two-phase clock edges to propagate the *stall* signal from the last stage of the pipeline to the first stage of the pipeline. This is equivalent to $\frac{N}{2}$ clock cycles. If a new data item is inserted into the pipeline on every rising edge of the clock, then the pipeline holds N data items by the time it is stalled: $\frac{N}{2}$ data items are inserted into a pipeline that already holds $\frac{N}{2}$ data items before the first stage sees the stall signal. The extra $\frac{N}{2}$ data items inserted into the pipeline before it is stalled are inserted into the empty stages that naturally exist as a result of the two-phase clocking; thus, the empty stages become full stages when the pipeline stalls to absorb the latency of propagating the stall progressively backward in the pipeline.

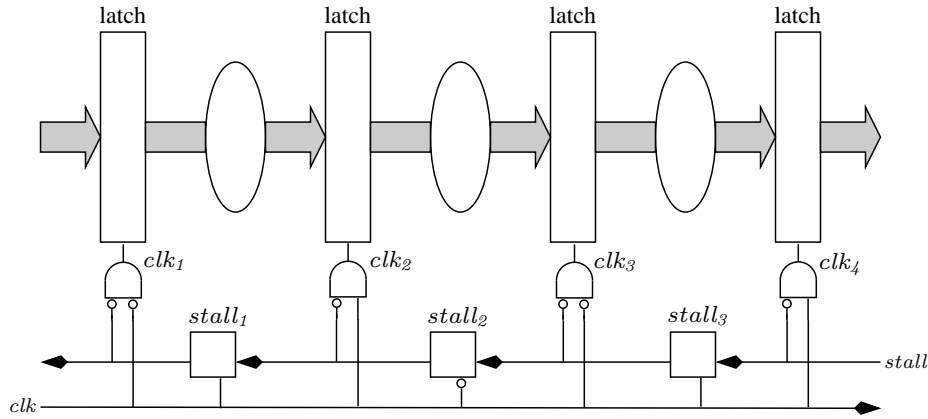


Fig. 13. Elastic synchronous pipeline.

The key synchronization points in the elastic synchronous pipeline are at the clock gates for each of the data latches. These gates must be hazard-free, or in other words, must not produce glitches on the local clock signals. Two properties must be observed for correct hazard-free operation. First, the clock polarity at the gating function must be such that it masks any glitches on the stall signal while the stall latch is transparent. Second, the timing between the global clock edge and the stall signal must be such that the stall signal stabilizes at the gating point before the next clock edge arrives. If either of these properties do not hold, the circuit verification will find that the circuit is hazardous. While functional and timing correctness may seem trivial for the logic in Figure 13, hazard-free operation in real two-phase clock splitters operating at high frequencies is difficult to verify manually due to the additional logic depth and complexity added by required support for scan and test, and phase shifting of the local clocks. Formal verification tools can provide valuable help for such verification.

To verify hazard freedom in the clock gates, the pipeline must first be formally modeled. A level-ruled Petri net model for the second stage of the pipeline in Figure 13 is shown in Figure 14. There are four processes in this figure. Figure 14(a) models the latch to generate $stall_2$ based on its input $stall_3$. Figure 14(b) models

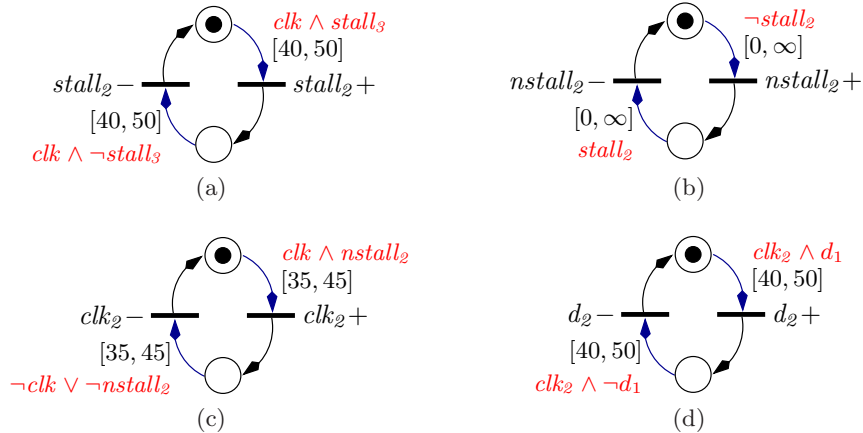


Fig. 14. Level-ruled Petri net models for: (a) the latch to generate the $stall_2$ signal, (b) the inverter to generate the $nstall_2$ signal, (c) the AND gate to generate the clk_2 input to the data latch, and (d) a data latch to model a single bit of the data path.

the inverse of the $stall_2$ input, $nstall_2$. Figure 14(c) creates the actual clock input to the data latches. It is an AND gate. Its output is clk_2 . Finally, Figure 14(d) is a data latch to model a single bit of the data path. The remaining stages in Figure 13 are modeled similarly.

To complete the model of the elastic synchronous pipeline, the behavior of the clk and $stall$ environments must be defined. The environment in Figure 15(a) is the global clock input. This model simply toggles the clk signal with a regular period. The $stall$ environment model is shown in Figure 15(b). The structure of this net forces a random choice based on the state of the $stall$ signal in the marking shown. If $stall$ is high, the net can either fire the $stall -$ transition or it can do nothing by firing t_1 . Similarly, if $stall$ is low, then the net can either fire it high, or do nothing also. The random $stall$ input enables an analysis algorithm to check correctness under all possible input conditions. The effect of this is that a correct implementation is correct under all input scenarios and timing conditions allowed by the environment and circuit model. This result is more important than one obtained through the simulation of inputs on corner cases. Another important property of the $stall$ environment in Figure 15(b) is that it does not generate multiple $stall$ transitions in a given clock cycle. It either transitions $stall$ high or low; or it does nothing. This type of environment is optimistic and assumes an input that is free of hazards, although it is possible to model an environment with hazards if needed.

The gate level model of a 3-stage elastic synchronous pipeline is analyzed by ATACS. As previously stated, the clock gating logic must not produce any glitches, and setup and hold times must be satisfied at the latches for the data and stall bits. This design is simple enough to quickly verify correctness in a few seconds. The analysis did, however, find an important timing assumption for correctness to hold. Data at a latch input

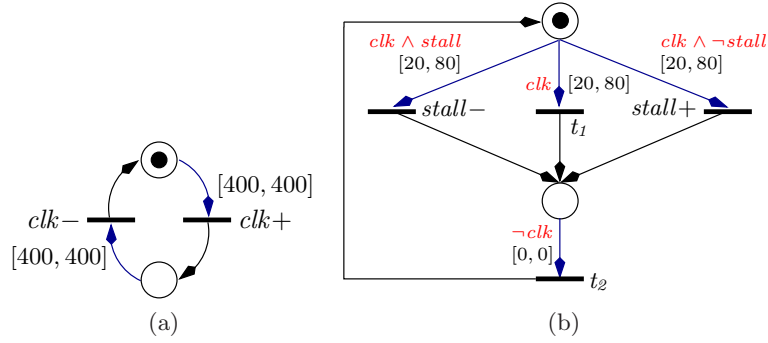


Fig. 15. The environment models for the elastic pipeline. (b) The level-ruled Petri net model for the clock input. (c) The level-ruled Petri net model of a random environment that captures all behaviors of the *stall* signal.

cannot change faster than the delay through an inverter and a single *AND* gate. If this timing constraint is not met, then there is a race condition between the positive phase and negative phase adjacent pipeline stages. If there is no logic between the stages, then the noninverted inputs of the clock gating logic needs to be buffered to minimize clock skew in the design. Although the constraint is typical in most latch based pipeline designs, the analysis formalizes the constraint and checks it through each iteration of the design. This type of analysis fosters confidence in the correctness of the final implementation.

Using the basic structure of the elastic pipeline, it is now possible to implement a two-phase fully interlocked pipeline as shown in Figure 16. The pipeline implements a forward and backward interlock using the *valid* and *stall* signals with the *gclk* signal. A high *valid* signal indicates that there is valid data at the stage. A high *stall* signal indicates that the stage must hold its current data if it is valid and propagate the stall signal backward. If data at the current stage is invalid, however, then the *stall* signal is ignored at that stage and is not propagated backward. Note that the stall signal is always ignored until the stage has valid data. The schematic in Figure 16 is verified at a gate level for 3 stages by *ATACS*. The analysis reveals that the logic used to compute the stall input at each stage may generate glitches. This signal is never critical, however, as any possible glitch is masked either by the stall latch being opaque, or the polarity of the global clock at the gating point. latch opens. *ATACS* verifies that the stall signal meets setup and hold times of the stall latch under normal synchronous operation and that any glitches on the stall signal can never produce a glitch on the local clock.

An advantage of modeling the interlock logic formally in *ATACS* is that its specification for verification can not only be used to ensure the correctness of the specified system, but may also be able to synthesize more efficient implementations of the modeled system. An alternative behavioral model of the pipeline is constructed to see if the glitch can be removed from the design. The logic to compute the stall signal at each stage is moved into the stall latch. The analysis in *ATACS* shows this design to not produce any glitches on any

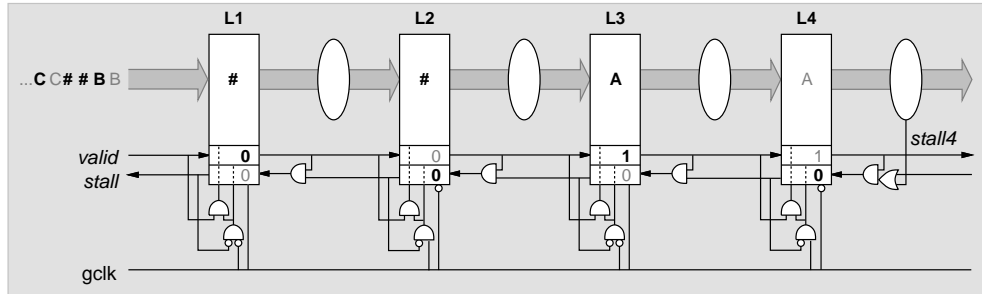


Fig. 16. Two-phase interlocked pipeline.

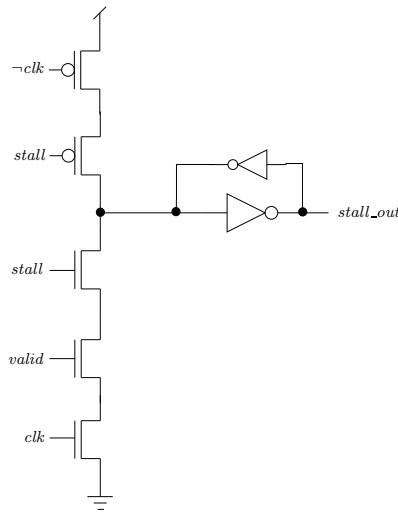


Fig. 17. Hazard free circuit for stall signal.

of its internal wires, thus implementing a more efficient circuit with potentially lower power dissipation. The synthesis of this model shows that the latch for the stall signal at each stage is replaced by the generalized C-element shown in Figure 17. This generalized C-element is the latch with the stall logic folded into it. If standard cell design is not an issue, then the internal glitches can be removed using the new gate. The synthesized results also show that the other gates in the design remain unchanged.

The pipeline previously illustrated in Figure 16 assumes an early arriving valid, i.e., the valid signal has to stabilize in the first half of the clock cycle. This provides symmetry in the gating logic, as the same logic can be used for both master and slave stages. In high frequency pipelines however, there is not always time to compute the valid signal and perform the gating in only half a clock cycle. Figure 18 illustrates one stage of a non-split master-slave pipeline where the gating logic allows the valid to arrive late in the clock cycle. The master-slave latch is used to store two data items to absorb the latency of the stall propagating backward in the pipeline. The resulting gating logic in a late valid implementation is no longer symmetric and requires

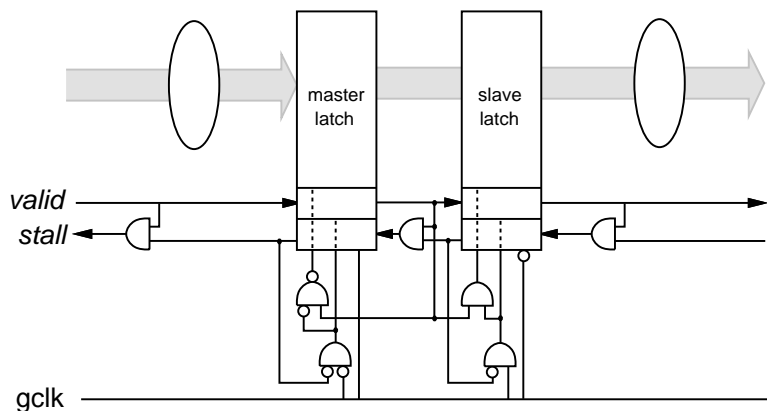


Fig. 18. Master-slave interlocked pipeline with late arriving valid.

two full master-slave stages to fully cover the interlocking between different combinations of master and slave stages.

For larger models, it becomes increasingly difficult to analyze them efficiently. This explosion in complexity is a result of the analysis ignoring the inherent modular structure of the design and instead trying to verify the entire system in one pass. We have implemented *abstraction* [50] and *partial order reduction* [27] within ATACS for modular synthesis and verification to enable the analysis of larger models. Abstraction takes the entire model of the design with a module as input. The goal is to verify the correctness of the module in the context of the larger design. This is done by removing any transitions not on the interface of the module from the model of the design while conservatively preserving the behavior of the module. Removing transitions from the model reduces its analysis cost in terms of time and memory to enable the analysis of larger systems. Partial order reduction also reduces the number of transitions the analysis must consider in verifying a module in a larger design. It differs from abstraction, though, because it does not alter the original design model before analysis. Partial order reduction looks at transitions that can fire concurrently and, when possible, only explores a single firing order of those transitions to reduce the number of transition orderings it must consider in verifying the module. For example, if two transitions a and b can transition in the model, then a normal analysis must consider the case where a fires and then b , as well as the case where b fires and then a . A partial order reduction, however, only considers the case of a firing and then b if both orders lead to the same state of the model. The total analysis cost can be greatly reduced by removing these redundant orderings.

The cost of analyzing the entire design at once is 700 seconds. With both of the reduction techniques, the verification time is reduced to 540 seconds if each of the 36 individual gates is verified separately in the larger design. Both abstraction and partial order reduction can be run together since abstraction happens

before analysis and partial order reduction during analysis. It is possible to further reduce the analysis cost by grouping gates into larger modules. If a module is the complete master-slave pair with the accompanying logic, then the average cost is 80 seconds for the analysis of each module. The complete analysis of the two-stage master-slave gate level specification is 160 seconds. This is well under the time to complete the flat analysis. Using the modular approach, it is possible to analyze designs that were previously too large to consider.

The analysis of the non-split master-slave pipeline shows the design to be correct. Although the *AND* gate between the master and slave latches once again may glitch, this does not affect the correctness of the master-slave stage because the glitches are masked by the latches and clock polarity.

Figures 19 and 20 illustrate how non-linear structures, such as fork, branch, join, and select, can be built in synchronous interlocked pipelines. Note that the clock gating logic has been omitted in the figures for purpose of clarity. The resulting pipeline interlocking and clock gating forms a complex system of interacting state machines. Such a complex system is not possible to verify flat. However, verification could be successfully performed through use of outlined modular verification techniques.

The first non-linear pipeline structure to be verified was the fork stage. A pipeline fork stage copies data to N parallel downstream stages. A fork stage must stall if any of its downstream stages stall. Nonstalled downstream stages must be prevented from receiving duplicate copies of the data when the fork stage is stalled; thus, the valid signals to all downstream stages are lowered until all downstream stall conditions are removed. The data are now received by all downstream stages simultaneously. The logic for a one to two fork structure in a synchronous interlocked pipeline is shown in Figure 19(a). The gate level specification of this circuit cannot be analyzed flat by either timing method on the test machine. The fork stage and the two receiving stages are shown to be correct in under 100 seconds of running time using the modular approach. The stall logic is again hazardous when the stall latch is not active. The logic must be moved into the latch to suppress the glitch.

A pipeline branch stage copies data from an upstream stage to one of N parallel downstream stages. The decision to which of the downstream stages data is to be copied is determined by the datapath logic that generates a set of N one-hot encoded enabling signals. These enable signals mask the branch stage valid signal through a set of *AND* functions such that a valid is propagated only to the selected downstream stage. The branch stage must be stalled only if an already stalled downstream stage is selected as the destination of the data. The logic for a one to one-of-two branch structure is shown in Figure 19(b). The gate level specification is too large to analyze flat. To give perspective on the size of these models, the number of states explored in the reduced state space (abstracted and partial order reduced) of the left stage in Figure 19(b)

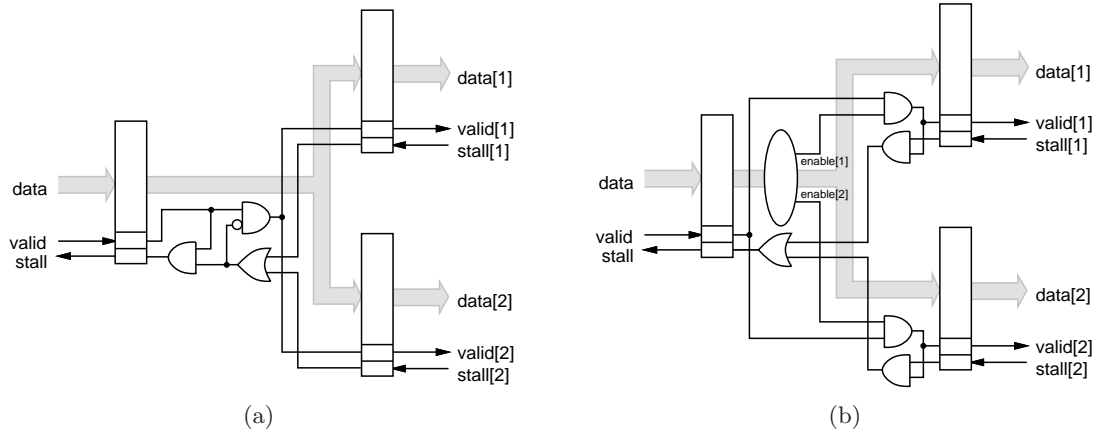


Fig. 19. (a) A one to two fork implementation. (b) A one to one-of-two branch implementation.

is 114405 zones in 113304 states. The zones are a measure of the cost incurred by the representation of time in the analysis algorithm, and the states are the number of explored markings in the model. The analysis completes in 148Mb of memory with 522 seconds of running time. The total analysis takes around 1600 seconds, but shows the design to be correct.

A pipeline join stage is a merge that concatenates data from N upstream stages to one downstream stage. The join stage waits until data is valid in all upstream stages before concatenating and propagating the data to the downstream stage. A join stage is used to synchronize and align the data streams of multiple pipelines. Any stage that becomes valid must be stalled until all stages have become valid and the data can be propagated to the downstream stage since data in different upstream stages can become valid at different times. If the join stage stalls, then all upstream stages must stall. The logic for a two to one join structure is shown in Figure 20(a). The gate level specification is too large to analyze flat. The number of states explored in the reduced state space of the left top stage in Figure 20(a) is 11224 zones in 11199 states. This completes in 23Mb with 31 seconds of running time. The total analysis completes in under 120 seconds.

A pipeline select stage is a selector that propagates data from one of N upstream stages to one downstream stage. A select stage implements a basic if-then-else multiplexer function. A select stage waits until data is valid in at least one of the upstream stages. One stage is then chosen through priority based selection and its data is propagated to the downstream stage. An upstream stage that contains valid data must stall until it is selected. The logic for a one-of-two to one priority select structure is shown in Figure 20(b). The gate level specification is too large to analyze flat. The number of states explored in the reduced state space of the top left stage in Figure 20(b) is 12247 zones in 12241 states. It completes in 23Mb of memory with a running time of 31.17 seconds. The total cost of analysis of all stages is under 120 seconds of running time.

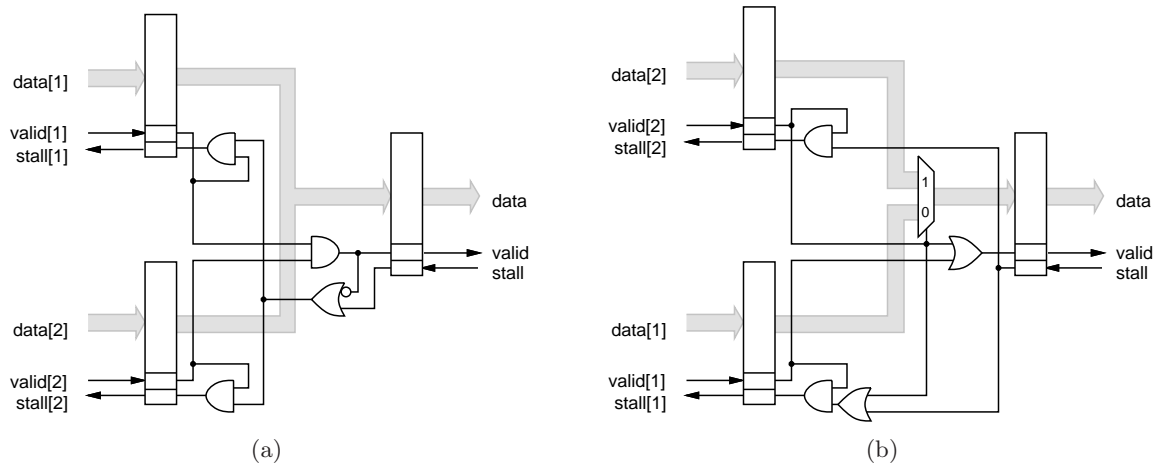


Fig. 20. (a) A two to one join implementation. (b) A one-of-two to one priority select implementation.

7 Conclusions

This paper has presented a review of the synchronization problem and strategies that have been developed to address it. It is crucial that designers know whether their selected strategy eliminates the probability of failure or simply minimizes it. In the latter case, a mean time between failure must be calculated to determine the reliability of the system. Finally, this paper presented two case studies of synchronization strategies and the techniques that are used to verify their correctness. As clock speeds increase, synchronization issues are becoming increasingly important, so such a formal verification approach to the analysis of synchronization strategies is essential to produce reliable systems in the future.

References

1. André Abrial, Jacky Bouvier, Marc Renaudin, Patrice Senn, and Pascal Vivet. A new contactless smart card IC using an on-chip antenna and an asynchronous microcontroller. *IEEE Journal of Solid-State Circuits*, 36(7):1101–1107, 2001.
2. Morteza Afghahi and Christer Svensson. Performance of synchronous and asynchronous schemes for VLSI systems. *IEEE Transactions on Computers*, 41(7):858–872, July 1992.
3. José C. Barros and Brian W. Johnson. Equivalence of the arbiter, the synchronizer, the latch, and the inertial delay. *IEEE Transactions on Computers*, 32(7):603–614, July 1983.
4. Kees van Berkel, Ronan Burgess, Joep Kessels, Ad Peeters, Marly Roncken, and Frits Schalijs. A fully-asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid-State Circuits*, 29(12):1429–1439, December 1994.
5. I. Catt. Time loss through gating of asynchronous logic signal pulses. *IEEE Transactions on Electronic Computers*, EC-15:108–111, February 1966.
6. T. J. Chaney and C. E. Molnar. Anomalous behavior of synchronizer and arbiter circuits. *IEEE Transactions on Computers*, C-22(4):421–422, April 1973.
7. Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984.
8. J. F. Chappel and S. G. Zaky. EMI effects and timing design for increased reliability in digital systems. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 44(2):130–142, February 1997.
9. P. Corsini. Self-synchronizing asynchronous arbiter. *Digital Processes*, 1:67–73, 1975.

10. P. Corsini. Speed-independent asynchronous arbiter. *IEE journal on Computers and Digital Techniques*, 2(5):221–222, October 1979.
11. G. R. Couranz and D. F. Wann. Theoretical and experimental behavior of synchronizers operating in the metastable region. *IEEE Transactions on Computers*, 24(6):604–616, June 1975.
12. W. Fleischhammer and O. Dortok. The anomalous behavior of flip-flops in synchronizer circuits. *IEEE Transactions on Computers*, 28(3):273–276, March 1979. Comments: see Lacroix 1982.
13. A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Öberg, P. Ellervee, and D. Lundqvist. Lowering power consumption in clock by using globally asynchronous, locally synchronous design style. In *Proc. ACM/IEEE Design Automation Conference*, 1999.
14. Jens U. Horstmann, Hans W. Eichel, and Robert L. Coates. Metastability behavior of CMOS ASIC flip-flops in theory and test. *IEEE Journal of Solid-State Circuits*, 24(1):146–157, February 1989.
15. Marco Hurtado. *Structure and Performance of Asymptotically Bistable Dynamical Systems*. PhD thesis, Sever Institute of Technology, Washington Univ., St. Louis, MO, 1975.
16. Hans M. Jacobson, Prabhakar N. Kudva, Pradip Bose, Peter W. Cook, Stanley E. Schuster, Eric G. Mercer, and Chris J. Myers. Synchronous interlocked pipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–12, April 2002.
17. Joep Kessels and Paul Marston. Designing asynchronous standby circuits for a low-power pager. *Proceedings of the IEEE*, 87(2):257–267, February 1999.
18. D. J. Kinniment and J. V. Woods. Synchronization and arbitration circuits in digital systems. *Proceedings of the IEE*, 123(10):961–966, October 1976.
19. Lindsay Kleeman and Antonio Cantoni. Can redundancy and masking improve the performance of synchronizers. *IEEE Transactions on Computers*, 35:643–646, July 1986.
20. Lindsay Kleeman and Antonio Cantoni. On the unavoidability of metastable behavior in digital systems. *IEEE Transactions on Computers*, C-36(1):109–112, January 1987.
21. G. Lacroix, P. Marchegay, and G. Piel. Comments on ‘the anomalous behavior of flip-flops in synchronizer circuits’. *IEEE Transactions on Computers*, 31(1):77–78, January 1982. See: Fleischhammer 1979.
22. W. Lim. Design methodology for stoppable clock systems. *IEE Proceedings, Computers and Digital Techniques*, 133(1):65–69, January 1986.
23. S. Lubkin. Asynchronous circuits in digital computers. *Mathematical Tables and other Aids to Computation*, pages 238–241, October 1952.
24. R. Männer. Metastable states in asynchronous digital systems - avoidable or unavoidable. *Microelectronics and Reliability*, 28(2):295–307, 1988.
25. Leonard R. Marino. The effect of asynchronous inputs on sequential network reliability. *IEEE Transactions on Computers*, 26:1082–1090, 1977.
26. Leonard R. Marino. General theory of metastable operation. *IEEE Transactions on Computers*, C-30(2):107–115, February 1981.
27. E. Mercer, C. Myers, T. Yoneda, and H. Zheng. Modular synthesis of timed circuits using partial orders on lpsns. In *Theory and Practice of Timed Systems (TPTS 2002)*, April 2002.
28. Chris Myers. *Asynchronous Circuit Design*. John Wiley & Sons, 2001.
29. Chris J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Elec. Eng., Stanford University, October 1995.
30. Chris J. Myers, Wendy Belluomini, Kip Killpack, Eric Mercer, Eric Peskin, and Hao Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference*, pages 335–340, February 2001.
31. R. C. Pearce, J. A. Field, and W. D. Little. Asynchronous arbiter module. *IEEE Transactions on Computers*, 24:931–932, September 1975.
32. Miroslav Pečhouček. Anomalous response times of input synchronizers. *IEEE Transactions on Computers*, 25(2):133–139, February 1976.
33. W. W. Plummer. Asynchronous arbiters. *IEEE Transactions on Computers*, 21(1):37–42, January 1972.
34. Fred U. Rosenberger and Charles E. Molnar. Comments on ‘metastability of CMOS latch/flip-flop’. *IEEE Journal of Solid-State Circuits*, 27(1):128–130, January 1992. Reply by Robert W. Dutton pages 131–132 of same issue.
35. Fred U. Rosenberger, Charles E. Molnar, Thomas J. Chaney, and Ting-Pien Fang. Q-modules: Internally clocked delay-insensitive modules. *IEEE Transactions on Computers*, C-37(9):1005–1018, September 1988.
36. T. Sakurai. Optimization of CMOS arbiter and synchronizer circuits with submicron MOSFETs. *IEEE Journal of Solid-State Circuits*, 23(4):901–906, August 1988.
37. S. Schuster, W. Reohr, P. Cook, D. Heidel, M. Immediato, and K. Jenkins. Asynchronous interlocked pipelined CMOS circuits operating at 3.3-4.5 GHz. In *Proc. International Solid State Circuits Conference*, February 2000.
38. Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

39. Jakov N. Seizovic. Pipeline synchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87–96, November 1994.
40. Allen E. Sjogren and Chris J. Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. *IEEE Transactions on VLSI Systems*, 8(5):573–583, October 2000.
41. R. F. Sproull and I. E. Sutherland. Stoppable clock. *Technical Memo 3438*, Sutherland, Sproull, and Associates, January, 1985.
42. K. S. Stevens, S. Rotem, R. Ginosar, P. Beerel, C. J. Myers, K. Y. Yun, R. Koi, C. Dike, and M. Roncken. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits*, 36(2):217–228, February 2001.
43. M. J. Stucki and Jr. J. R. Cox. Synchronization strategies. In Charles L. Seitz, editor, *Proceedings of the First Caltech Conference on Very Large Scale Integration*, pages 375–393, 1979.
44. Stephen H. Unger. Hazards, critical races, and metastability. *IEEE Transactions on Computers*, 44(6):754–768, June 1995.
45. W. S. VanScheik and R. F. Tinder. High speed externally asynchronous/internally clocked systems. *IEEE Transactions on Computers*, 46(7):824–829, July 1997.
46. Harry J.M. Veendrick. The behavior of flip-flops used as synchronizers and prediction of their failure rate. *IEEE Journal of Solid-State Circuits*, 15(2):169–176, 1980.
47. Jacqueline Walker and Antonio Cantoni. A new synchronizer design. *IEEE Transactions on Computers*, 45(11):1308–1311, November 1996.
48. Ted E. Williams and Mark A. Horowitz. A zero-overhead self-timed 160ns 54b CMOS divider. *IEEE Journal of Solid-State Circuits*, 26(11):1651–1661, November 1991.
49. Kenneth Y. Yun and A. E. Dooply. Pausible clocking-based heterogeneous systems. *IEEE Transactions on VLSI Systems*, 7(4):482–488, December 1999.
50. H. Zheng, E. Mercer, and C. J. Myers. Modular verification of timed circuits using automatic abstraction. *IEEE Trans. on Computer-Aided Design*, 22(9), September 2003.