Interfacing Synchronous and Asynchronous Modules Within a High-Speed Pipeline *

Allen E. Sjogren and Chris J. Myers
Electrical Engineering Department
University of Utah
Salt Lake City, UT 84112
{sjogren, myers}@ee.utah.edu

Abstract

This paper describes a new technique for integrating asynchronous modules within a high-speed synchronous pipeline. Our design eliminates potential metastability problems by using a clock generated by a stoppable ring oscillator, which is capable of driving the large clock load found in present day microprocessors. Using the `ATACS` *design tool, we designed highly optimized transistor-level circuits to control the ring oscillator and generate the clock and handshake signals with minimal overhead. Our interface architecture requires no redesign of the synchronous circuitry. Incorporating asynchronous modules in a high-speed pipeline improves performance by exploiting data-dependent delay variations. Since the speed of the synchronous circuitry tracks the speed of the ring oscillator under different processes, temperatures, and voltages, the entire chip operates at the speed dictated by the current operating conditions, rather than being governed by the worst-case conditions. These two factors together can lead to a significant improvement in average-case performance. The interface design is tested using the 0.6μm HP CMOS14B process in* `HSPICE`*.*

1: Introduction

Circuit designers are continually pushing the envelope in the race to design faster, more powerful microprocessors. Present day synchronous microprocessors have clock speeds in excess of 300MHz. Distributing a clock signal to all areas of a large chip at this speed with minimal clock skew is a task of growing complexity. The circuit area, power consumption, and design time needed to drive the clock signal to all parts of the chip without significant clock skew are overwhelming [4, 1]. The clock period must also be long enough to accommodate the worst-case delay in every module in the worst process run under the highest temperature and lowest supply voltage. Thus, any speed gained from completing an operation early is lost waiting for the clock, which runs at a rate dictated by the slowest component running in the worst operating conditions.

Asynchronous circuits have attracted new interest as an alternative to synchronous circuits due to their potential to achieve average-case performance while eliminating the global synchronizing clock signal. In asynchronous circuits, an operation begins when all the operations that it depends on have occurred, rather than when the next clock signal arrives. This allows asynchronous circuits to operate as fast as possible, taking advantage of delay variations due to data dependencies and operating conditions. Thus, well-designed asynchronous circuits can achieve better average operating frequencies than synchronous circuits operating at frequencies dictated by the worst-case conditions. Asynchronous circuits also eliminate the global clock, which can reduce circuit area, power consumption, and design time.

* This research is supported by a grant from Intel Corporation and NSF CAREER award MIP-9625014.

The advantages of synchronous circuits though, cannot be overlooked. Some of these advantages include: ease of implementing sequential circuits, simplicity in dealing with hazards, and mature design methods and tools. Also, asynchronous circuits come with their own set of challenges. Since there is no global clock to tell when outputs are stable, asynchronous circuits must prevent any hazards, or glitches, on their outputs. A false transition on an output from one circuit can cause the next circuit to prematurely operate on meaningless results. Additional circuitry is used to prevent hazards. This circuitry can increase the area and delay of the asynchronous circuit. In order to achieve average-case performance, asynchronous circuits require additional circuitry to start each computation and detect the completion of operations. The additional circuitry required for asynchronous design can, in some cases, make the average-case delay of an asynchronous circuit become larger than the worst-case delay for the comparable synchronous circuit.

The ideal system, therefore, uses a combination of both synchronous and asynchronous circuits. The most appropriate timing discipline could then be applied to each module. Combining the two technologies poses a great challenge. The key difficulty is found in trying to avoid *synchronization failure*. If a clock edge from a synchronous circuit changes too close in time to data arriving from an asynchronous circuit, the circuit may enter a *metastable state* [2]. A metastable state is a stable state of the circuit, which is at neither a logic 0 or logic 1 level, but rather lies somewhere in-between. In this case, the data latched from the asynchronous circuit may be at an indeterminate value. The circuit can reside in this state for a non-deterministic amount of time. If this metastable state persists until the next clock cycle, the indeterminate data may be interpreted as either a logic-0 or a logic-1 by different subsequent logic stages. 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 [8]. If care is not taken, the integration of more asynchronous circuitry and communication into a system can lead to an unacceptable probability of synchronization failure.

Many techniques have been devised to address the metastability problem and avoid synchronization failure when interfacing between synchronous and asynchronous modules. The simplest approach is to double-latch asynchronous signals being sampled by a synchronous module. This increases the time allowed for a metastable condition to resolve. The cost though is an extra cycle delay when communicating data from an asynchronous module to a synchronous module, even when there is no metastability. This scheme only minimizes the probability and does not eliminate the possibility of synchronization failure, as there is some chance that a metastable condition could persist longer than two clock cycles. To address this problem, *pipeline synchronization* can be used. Pipeline synchronization extends the double-latching idea by inserting more pipeline latches between the asynchronous and synchronous module [9]. While each added latch reduces the probability of failure, it increases the latency of communication. Also, no matter how many latches are added, some probability of failure always remains. Therefore, this scheme only works when large communication latencies and some failures can be tolerated. This is true of networks, but it is not true of high-speed microprocessor pipelines.

To completely eliminate synchronization failures, 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 it is either not ready to communicate new data or not ready to receive new data. *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 [10, 8]. A stoppable clock is typically constructed from a gated ring oscillator as shown in Figure 1. The basic operation 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 can be stopped synchronously and restarted asynchronously.

Numerous researchers have developed *globally asynchronous locally synchronous* (GALS) architectures based on the idea of a stoppable clock [8, 11, 3, 7, 5, 12]. In each of these approaches, 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

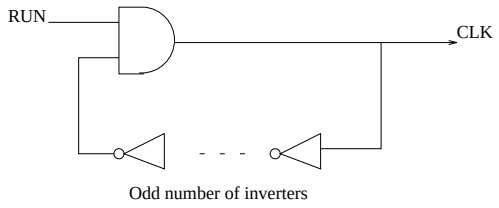

Figure 1. A stoppable ring oscillator clock.

structure of such a module is shown in Figure 2. The module's internal clock is stopped when it must wait for data to arrive from, or to be accepted by, the other modules. The schemes proposed in [11, 7, 5, 12] allow an asynchronous module to request to communicate data to a synchronous module at arbitrary times. A mutual exclusion element is used to guarantee that a synchronous module either receives data from a 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. The schemes proposed in [8, 3] assume that the synchronous unit determines when data is to be transferred to/from the asynchronous modules. This assumption eliminates the need for a mutual exclusion element, since the decision to wait on asynchronous communication is synchronized to the internal clock.

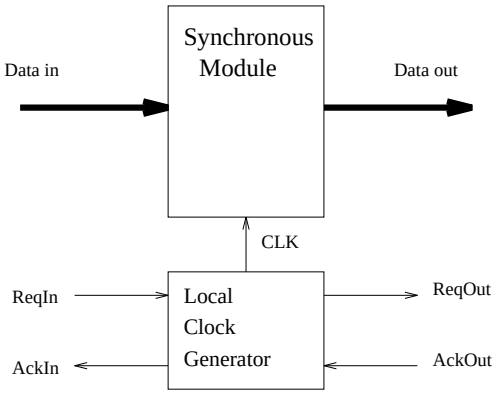

Figure 2. Basic module of a GALS architecture.

This paper describes a new interface methodology for *globally synchronous locally asynchronous* architectures. At present almost every microprocessor is synchronous and pipelined. One viable approach to increasing a microprocessor's speed for a given process is to replace the slowest pipeline stages with asynchronous modules that have a better 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.

Our interface methodology, while similar to the GALS architectures described in [8, 3], allows for stages in high-speed pipelines to be either synchronous or asynchronous as depicted in Figure 3. In this paper, we use true single-phase clocking with Yuan/Svenson latches as in the Alpha [4], 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 behaviour 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.

Our interface controller uses new, highly optimized transistor-level circuits designed using the ATACS design tool [6] to control the ring oscillator and generate the clock and handshake signals. By building the stoppable ring oscillator clock out of a clock buffer network, our clock is capable of driving the large capacitive loads found in present day microprocessors. Our interface technique does not require any redesign of the synchronous circuitry. Utilizing a ring-oscillator to generate the clock signal improves the performance of the circuit by allowing the integration of faster asynchronous modules in the pipeline. Since the speed of the synchronous circuitry tracks the speed of the ring oscillator under different processes, temperature, and voltage, the entire chip operates at the speed dictated by the current operating conditions, rather than being governed by the worst-case conditions. These two factors together can lead to a significant improvement in average-case performance.

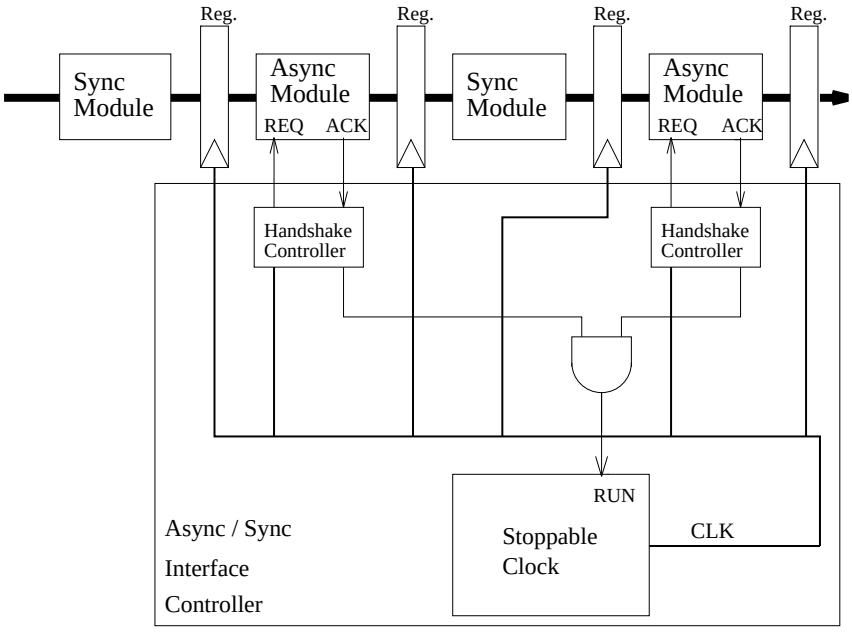

Figure 3. Proposed interface methodology.

This paper is divided into six sections. Section two describes the design of the basic circuits and operation of the interface controller. Section three presents an analysis of a clock buffer network similar to the one used in the 300 MHz DEC Alpha [1]. In section four, we incorporate the clock buffer network into our interface controller so that it can be used in modern high-speed pipelines. In section five, we add a pipeline latch and modify the interface protocol to reduce the control overhead. Section six gives our conclusions.

2: Basic interface controller

This section describes the basic circuits and operation of our asynchronous/synchronous interface controller. We designed the interface circuits described here using ATACS [6], a tool for the synthesis of *timed circuits*. 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 ATACS also allows us to automatically check our timing assumptions. Since the circuits in the interface controller are highly time dependent, they cannot be designed using traditional untimed asynchronous design methods.

The interface controller is composed of two separate sections: the stoppable clock and the handshake controllers. Our stoppable clock, shown in Figure 4, is somewhat different from the traditional one [8]. Rather than using an *AND* gate to control the starting and stopping of the clock, we use a state-holding gate. Our state-holding 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 4. 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 [10], 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 should be set such that the delay through the ring oscillator is greater than the worst-case path through the slowest synchronous module.

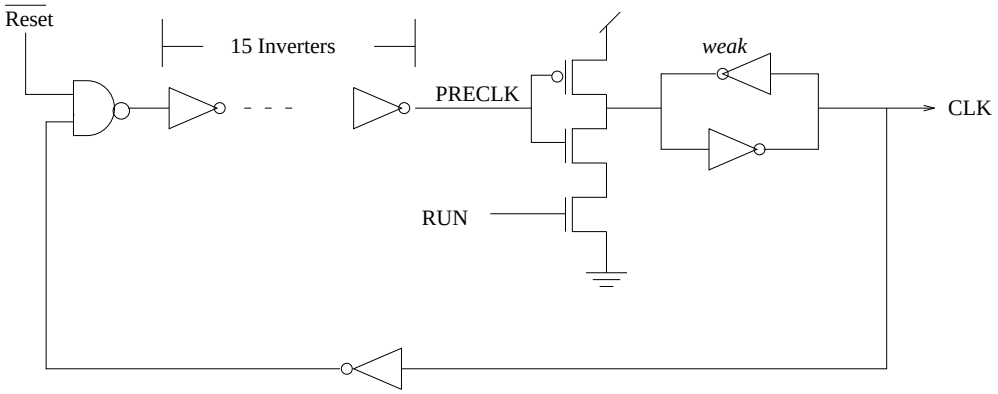

Figure 4. Our basic stoppable ring oscillator clock.

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 very efficient way to implement an asynchronous datapath is to use *domino dual-rail logic*, in which *REQ* low would precharge the logic. When the precharge 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 5.

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.

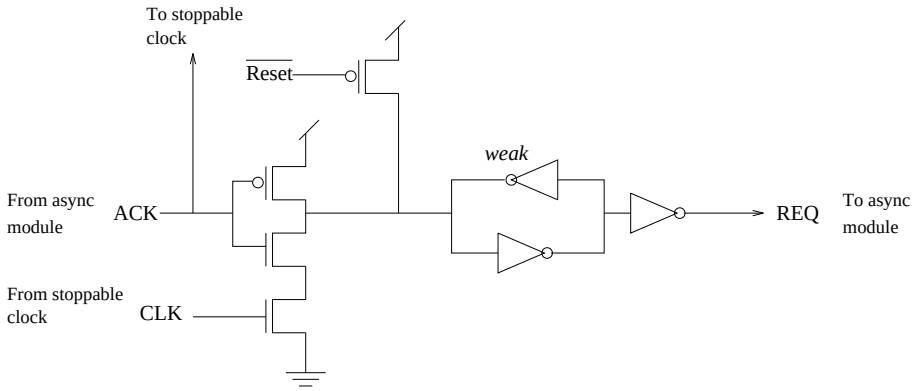

Figure 5. Handshake control circuit.

One may also wonder why there is a second n-transistor gated with the ACK signal in Figure 5. 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 6. 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. In the remainder of this paper, we assume the timing assumption holds, and the handshake control circuit used is the one depicted in Figure 5.

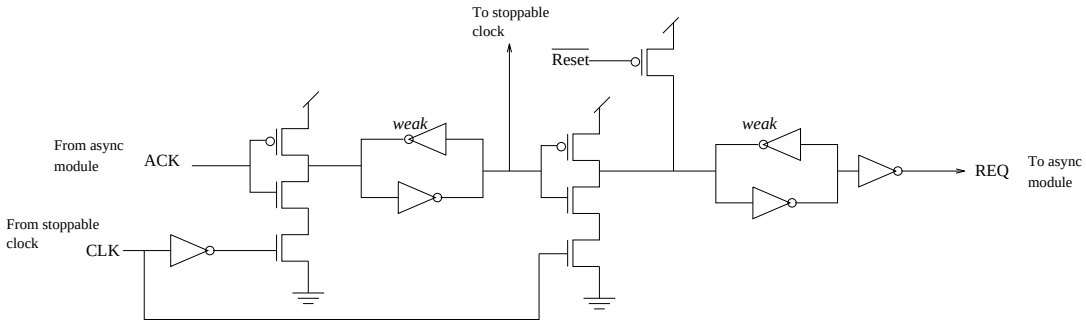

Figure 6. Handshake control circuit with conditioned acknowledgment.

The basic operation of the interface controller is depicted as an idealized waveform shown in Figure 7. 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.

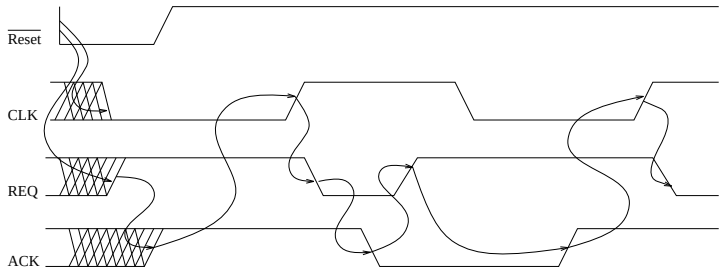

Figure 7. Idealized waveform for the basic interface controller.

We simulated the basic interface controller in `HSPICE` using the $0.6\mu m$ HP CMOS14B process. This process is used for a 120 MHz HP PA-7200 RISC microprocessor. Therefore, we set the ring oscillator clock to run at approximately 120 MHz worst-case by using 19 gates in the ring (1 $NAND$ gate, 16 inverters, the clock stopping gate, and its staticizer). We modeled the datapath using a chain of domino buffers as shown in Figure 8. This circuit has the property that after REQ goes high, ACK goes high after a delay through the entire buffer chain while after REQ goes low, ACK goes low after the delay of one domino buffer. Two waveforms are shown in Figure 9 under worst-case process and operating conditions. The first shows operation when the asynchronous unit finishes early. In this case, the CLK signal runs at a fixed rate. The second shows operation when the asynchronous unit finishes late which stops the clock until after ACK goes high.

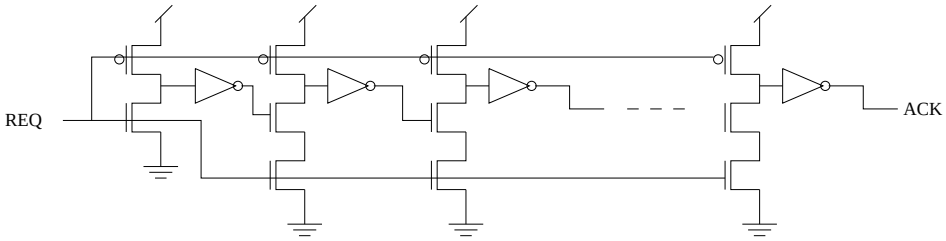

Figure 8. Domino buffer chain used to model the datapath.

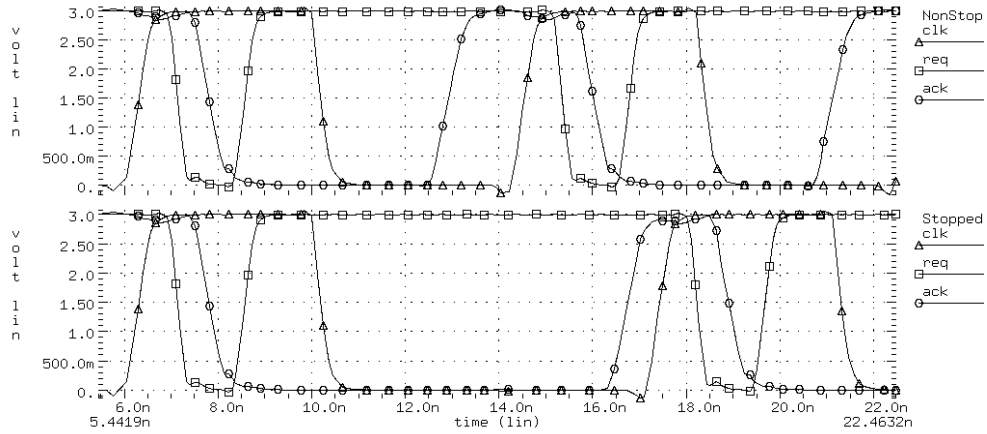

Figure 9. `HSPICE` **results for the basic interface controller.**

Table 1 shows the tabulated delay results under four different process and operating conditions. The first column shows the worst-case conditions (high temperature, low voltage, worst-case n- and p-type transistors). The middle two columns show more typical process and operating conditions running hot and cool, respectively. The last column shows the best-case conditions. The first row shows the delay of the ring oscillator with RUN (i.e., ACK) set high early. The frequency of the ring oscillator clock varies from 122 MHz to 285 MHz. Since the ring oscillator is built on the same chip as the rest of the circuits, the variation in delay of the other circuits track the variation in delay of the ring oscillator. This translates directly into an improvement in performance for not only the asynchronous circuitry, but also the synchronous circuitry. In other words, if the chip becomes hot, both the ring oscillator and logic circuits slow down. If the chip becomes cool, both the ring oscillator and logic circuits speed up. The same effect would take place for voltage and process variations. The next three rows show the delays in the interface control gates. The last row shows the amount of time which is available from the clock cycle for precharge and computation without stopping the clock. This is calculated by subtracting the control overhead from the minimum ring oscillator delay under the given conditions. This shows that the control overhead of using an asynchronous module is between 25 and 35 percent of the cycle. This means that an asynchronous module needs to have an average-case performance that is at least 25 percent less than the worst-case performance of the comparable synchronous module in order to see a performance gain.

Table 1. Clock period break down for the basic interface controller.

Temperature Voltage Process	90°C 3.0V WCNWCP	70°C 3.3V TYPICAL	25°C 3.3V TYPICAL	0°C 3.6V BCNBCP
$CLK\uparrow \rightarrow CLK\uparrow$	8.22 ns	5.42 ns	4.02 ns	3.51 ns
$CLK\uparrow \rightarrow REQ\downarrow$	0.99 ns	0.72 ns	0.58 ns	0.52 ns
$ACK\downarrow \rightarrow REQ\uparrow$	0.38 ns	0.28 ns	0.22 ns	0.19 ns
$ACK\uparrow \rightarrow CLK\uparrow$	0.75 ns	0.64 ns	0.57 ns	0.48 ns
Precharge & computation	6.10 ns	3.77 ns	2.65 ns	2.32 ns

3: Clock buffer tree analysis

In order to stop the clock signal in a modern high-speed microprocessor, one must first understand how the clock signal is buffered. Ideally, a clock signal is distributed to every point in the chip at nearly the same time, in phase, with fast rise and fall times. However, this is not an easy task. The capacitive load seen by the clock can be astronomical. For example, the 300 MHz DEC Alpha microprocessor has a clock load of 3.75nF [1]. In order to drive such a substantial load, the clock is buffered through 10 stages fanning out to 96 branches. The clock buffer tree is depicted in Figure 10 [1]. The crystal clock from the I/O pin is fed into the trunk of the network, and it is distributed to all points through buffered branches. There are shorting bars between the outputs of each buffer stage, which smooth out any asymmetry in the incoming wavefront. The result is that each level of inverters in the clock tree is equivalent to a single parallel n-transistor and a single parallel p-transistor. The final CLK driver inverter has an equivalent transistor width of 58 cm.

A concern with the buffer tree is the amount of delay through it for one clock pulse. If the delay is long enough, multiple clock pulses would be propagating through the tree. In this case, if the clock needs to be stopped, the decision may need to be made one or more clock cycles in advance. To measure the severity of this problem, we measured the delay through a clock buffer network similar to the 300 MHz DEC Alpha's. Since all branches are identical, we recreated one complete branch of the 96 branches and simulated it using `HSPICE`. The part of the clock tree simulated is shown in Figure 11. The non-terminated branches fan out to make up the other 95 individual branches.

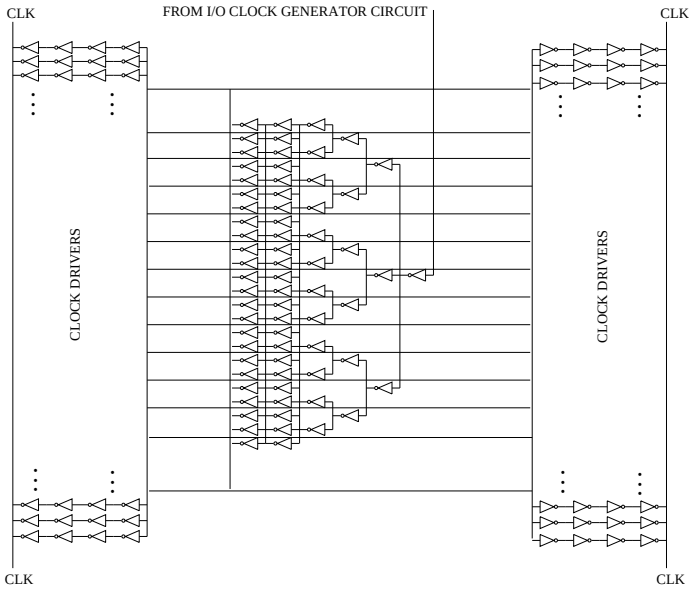

Figure 10. Clock buffer network for the 300 MHz DEC Alpha RISC microprocessor.

These non-terminated branches are used to simulate the capacitive loading on each stage of the single branch. The buffer tree is terminated with a 0.039 nF capacitor to model the portion of the load this branch is responsible for driving. We ran the *CLKin* frequency at 120 MHz, since as mentioned before this appears to be about the maximum clock rate for a synchronous microprocessor in the $0.6\mu m$ HP CMOS14B process. Our simulation of this clock buffer tree under worst-case conditions is shown in Figure 12. This figure shows that a transition on the *CLK* signal at the I/O pin is actually seen on the internal *CLK* signal up to nearly half a cycle later. Delays through the clock buffer network are shown for various conditions in Table 2. Our results show that the delay through the tree ranges from 1.41 ns to 3.16 ns. Thus with a clock period of 8.33 ns (i.e., a clock frequency of 120 MHz), there is only one clock pulse in the tree at a time.

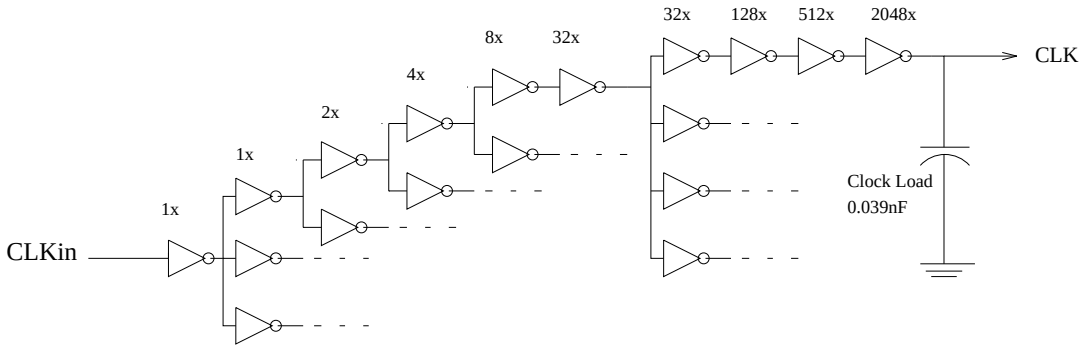

Figure 11. Clock buffer tree model simulated in `HSPICE`.

While our simulated results are encouraging, in that the decision to stop the clock does not need to be made one or more cycles ahead, the buffer tree makes it difficult to find the right time and place in which to stop the clock. In the GALS approach, if the microprocessor is to be considered as the locally synchronous component, these techniques would try to stop the clock at the I/O pin. The result would be that the clock is stopped nearly half a cycle too late. One of the major results of this paper is that the interface methodologies described in the subsequent sections are designed to work in an environment with such large clock buffer networks.

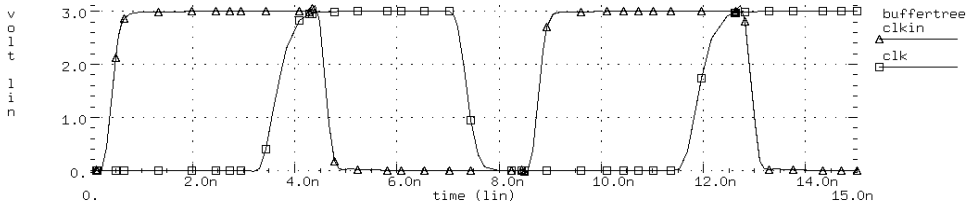

Figure 12. Delay from $CLKin$ to CLK modeled in HSPICE.

Table 2. Delay between $CLKin$ and CLK through the clock buffer tree.

Temperature	90°C	70°C	25°C	0°C
Voltage	3.0V	3.3V	3.3V	3.6V
Process	WCNWCP	TYPICAL	TYPICAL	BCNBCP
$CLKin\uparrow \rightarrow CLK\uparrow$	3.16 ns	2.35 ns	2.17 ns	1.72 ns
$CLKin\downarrow \rightarrow CLK\downarrow$	2.83 ns	2.05 ns	1.88 ns	1.41 ns

4: Interface controller with clock buffering

This section describes a modification of the basic interface controller to allow it to work in systems with large clock buffer networks. The basic idea is that the clock buffer network is used as part of the ring oscillator to generate the clock as depicted in Figure 13. The gate used to stop the clock is located at each of the leaves of the clock tree and makes up the last two inverter stages of the clock buffer network. For simulation, we use transistor sizes similar to those used in the 300 MHz DEC Alpha as in the previous section. This means that the n-transistors in the clock stopping gate are over $300\mu m$ wide (the p-transistor is over $600\mu m$), and over $1mm$ wide in the output inverter. As in the clock buffer network discussed in the previous section, there are shorting bars between the outputs of each buffer stage. This means there is a single global clock wire distributed around the chip. Since the RUN signal is generated from near-minimum size gates in the asynchronous datapath logic, it must also be buffered in order to control such a large gate. The buffer tree for the RUN signal is very similar to the one for the CLK signal. This results in a substantial delay from when ACK is asserted by the asynchronous modules to when the RUN signal is actually asserted.

Figure 14 shows an HSPICE simulation of the interface control circuit in Figure 13 under the worst conditions. The top plot shows the CLK and REQ signals, and the bottom plot shows the ACK and RUN signals. The simulation assumes there is only one asynchronous module. If multiple asynchronous modules are used, all of their ACK signals would have to be ANDed together to produce the RUN signal. This additional AND operation would not substantially change the overall delay from ACK to RUN as it is small in comparison to the buffer network, and it could be made to serve as part of the buffer network. The simulation shows that RUN falls over 2 ns after ACK falls. To prevent a synchronization failure, RUN must fall before CLK is ready to rise again. Assuming the precharge delay is short, RUN falls well before CLK can rise. The falling delay of RUN is not on the critical path as computation is done in parallel with this delay. The rising delay of RUN is on the critical path as the next rising CLK edge cannot come until after RUN goes high.

We simulated the interface controller with buffering in HSPICE under several different conditions and tabulated the results in Table 3. The results show that when buffering is taken into account, the control overhead is now from 54 to 57 percent of the clock cycle time. This means that in a high-speed microprocessor, an asynchronous module needs to have an average-case performance that is at least 54 percent faster than the worst-case performance of the comparable synchronous module in order to see a performance gain. While the good news is that an asynchronous module can be inserted into a high-speed microprocessor with minimal design changes, the bad news is that the applications where it results in a substantial performance improvement may be severely limited.

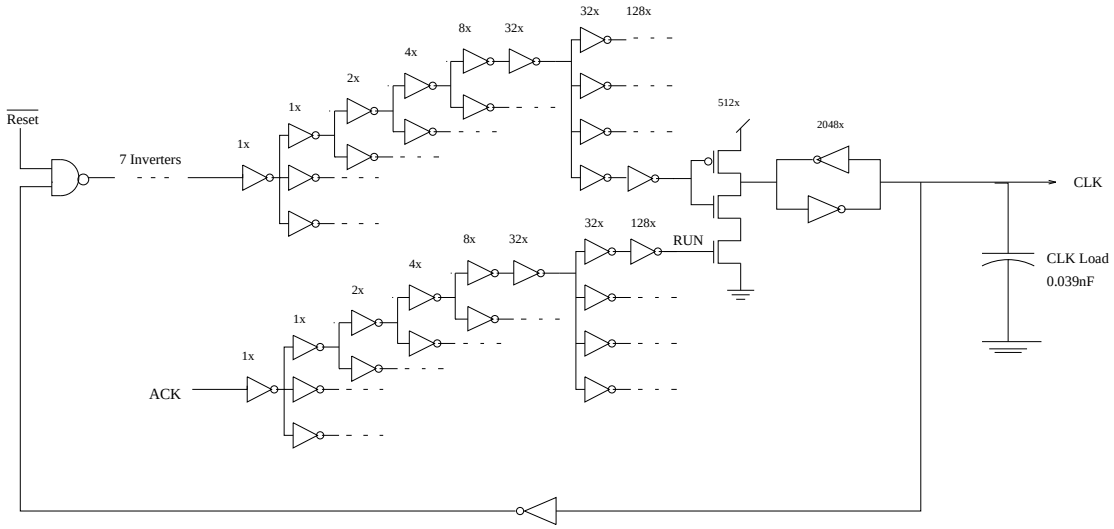

Figure 13. Stoppable ring oscillator clock with buffering.

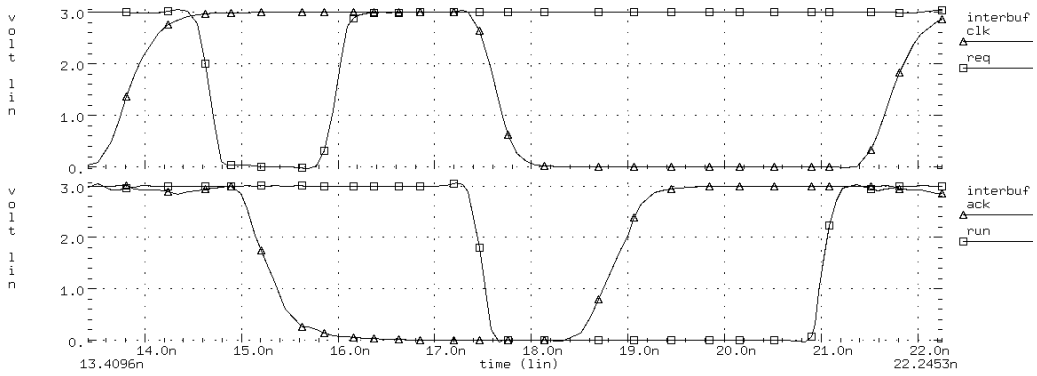

Figure 14. `HSPICE` **results for the interface controller with buffering.**

Table 3. Clock period break down for the interface controller with buffering.

Temperature Voltage Process	90°C 3.0V WCNWCP	70°C 3.3V TYPICAL	25°C 3.3V TYPICAL	0°C 3.6V BCNBCP
$CLK\uparrow \rightarrow CLK\uparrow$	7.94 ns	5.58 ns	4.98 ns	3.66 ns
$CLK\uparrow \rightarrow REQ\downarrow$	1.10 ns	0.82 ns	0.74 ns	0.57 ns
$ACK\downarrow \rightarrow REQ\uparrow$	0.38 ns	0.28 ns	0.26 ns	0.19 ns
$ACK\uparrow \rightarrow RUN\uparrow$	2.32 ns	1.62 ns	1.47 ns	1.10 ns
$RUN\uparrow \rightarrow CLK\uparrow$	0.49 ns	0.34 ns	0.30 ns	0.23 ns
Precharge & computation	3.66 ns	2.41 ns	2.19 ns	1.57 ns

5: Pipelined interface controller

As shown in Table 3, the largest overhead in the interface controller with buffering is the delay from ACK rising through the RUN signal's buffer network until RUN rises. This constitutes about 30 percent of the clock cycle time. During this time, no useful work is being done. Recall that the delay from ACK falling to RUN falling is hidden behind the computation delay, so it is not on the critical path. We would like to hide the rising delay of the RUN signal behind the precharge delay, so useful work can be done in parallel with this delay. The problem is, however, that we cannot begin the precharge stage until CLK rises latching the results from the previous computation. To solve this problem, we have added an additional pipeline register after each asynchronous module. As depicted in Figure 15, this new register latches data on the rising edge of the ACK signal. Therefore, as soon as the computation is completed, the new data is latched, so that the precharge stage can start immediately and run in parallel with the rising delay for RUN through the buffer tree.

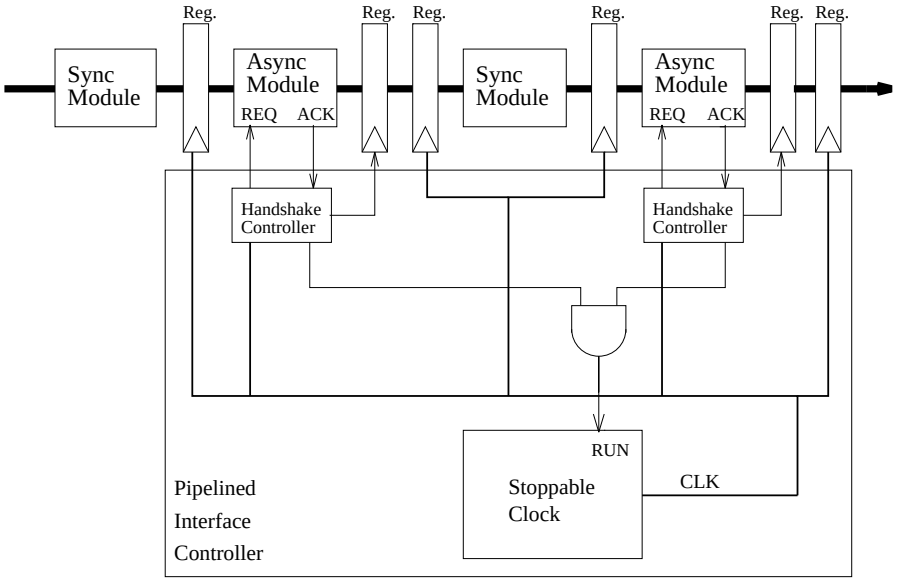

Figure 15. Interface controller with pipeline latch.

This new design requires a new interface protocol and several changes to the interface control circuitry. The stoppable clock circuit is basically the same as before with one small exception. We need to move the $NAND$ gate used for resetting the stoppable clock up into the buffer network to remove a race between starting the clock and lowering the RUN signal after initialization. The handshake control circuit has to be redesigned, and the new circuit is shown in Figure 16. In the new interface protocol, the precharge stage is completed in parallel with the RUN signal and subsequently the CLK signal being set high. Therefore, when CLK goes high, we are ready to start a new computation immediately by setting REQ high. When ACK goes high, the computation is completed, and the results are latched into the new pipeline latch. At this point, the precharge stage can be started by resetting REQ. Note there is an extra p-transistor gated on CLK being low before REQ goes high. This transistor is necessary to guarantee that the REQ signal is not being pulled high and low at the same time. If the computation delay is guaranteed to be longer than half a clock cycle, then CLK is always low before ACK goes high. This timing assumption would allow this p-transistor to be removed. In the following analysis, we left the transistor in to allow more variance in the computation completion timing. If we can guarantee the timing assumption, it would improve our results somewhat by speeding up the falling delay of REQ.

Figure 17 shows an `HSPICE` simulation of the pipelined interface control circuit under the worst-case conditions. This figure illustrates some of the timing assumptions needed to make this design

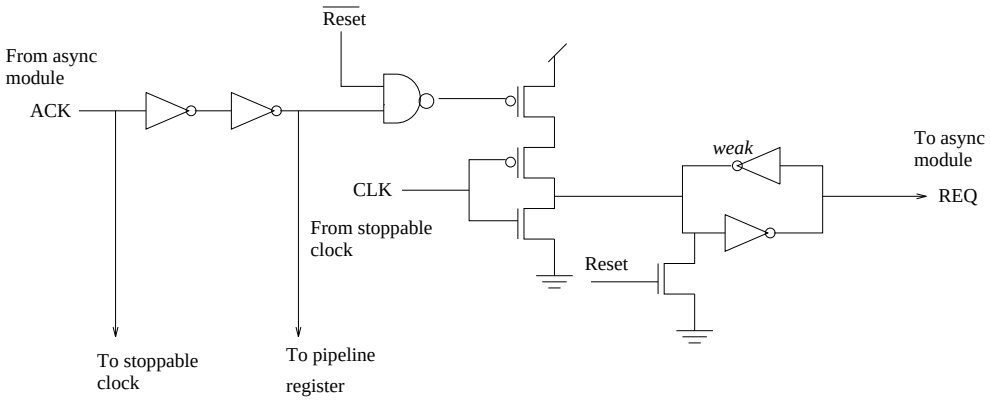

Figure 16. New handshake control circuit.

work. The first thing to notice is that RUN goes low in response to ACK going low in the previous cycle. If RUN falls too soon, CLK cannot rise. Therefore, ACK cannot fall earlier than the size of the ACK to RUN delay before CLK rises. Also, since CLK sets REQ high to start the computation, ACK must fall, signaling the end of the precharge stage, before CLK rises. When the CLK is being stopped by an asynchronous module, this timing assumption is easily met. At other times, this puts a 1 to 2 ns window (depending on conditions) on when ACK is allowed to fall. If the computation can be fast enough to violate this timing assumption, it can be easily fixed by adding a minimum bundled delay path to the violating asynchronous module. This minimum delay path would cause the rising transition of the ACK signal to be delayed in the fast cases. Note that this does not impact performance because if an asynchronous module is computing fast enough to avoid stopping the clock, its actual speed does not affect the speed of the chip.

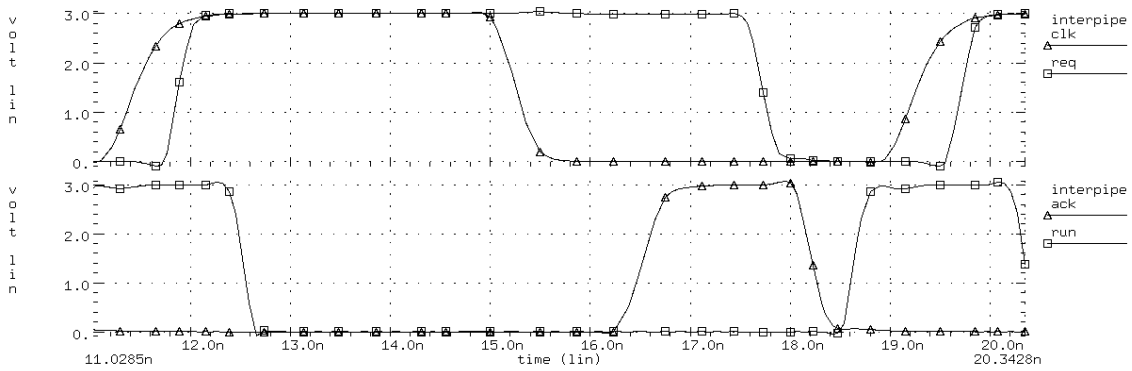

Figure 17. `HSPICE` **results for the pipelined interface controller design.**

Simulation results for the pipelined interface controller are given in Table 4 for several different process and operating conditions. In this design, the only control overhead is when REQ is changing state. From the time that ACK goes high until the time when CLK goes high, this circuit can reset REQ and precharge. As mentioned above, the precharge stage must complete before CLK goes high, but there is plenty of time available under all conditions. Other than when REQ is being set high or the CLK is being set high after the previous computation (i.e., after ACK goes high), the rest of the time is available for computation. This means that nearly 60 percent of the clock cycle is available for computation and useful work can be done in nearly 75 percent of the clock cycle. Therefore, for the pipelined interface controller, the asynchronous module only needs to be around 25 percent faster on average than its comparable synchronous module. This significantly improves the applicability of using mixed synchronous/asynchronous design.

Table 4. Clock period break down for the pipelined interface controller.

Temperature Voltage Process	90°C 3.0V WCNWCP	70°C 3.3V TYPICAL	25°C 3.3V TYPICAL	0°C 3.6V BCNBCP
$CLK\uparrow \rightarrow CLK\uparrow$	7.79 ns	5.44 ns	4.93 ns	3.64 ns
$CLK\uparrow \rightarrow REQ\uparrow$	0.54 ns	0.48 ns	0.45 ns	0.29 ns
$ACK\uparrow \rightarrow REQ\downarrow$	1.47 ns	1.04 ns	0.94 ns	0.71 ns
$ACK\uparrow \rightarrow RUN\uparrow$	2.18 ns	1.52 ns	1.39 ns	1.03 ns
$RUN\uparrow \rightarrow CLK\uparrow$	0.45 ns	0.34 ns	0.30 ns	0.23 ns
Precharge	1.16 ns	0.82 ns	0.75 ns	0.55 ns
Computation	4.61 ns	3.09 ns	2.80 ns	2.09 ns

6: Conclusions

Mixing synchronous and asynchronous modules within a high-speed pipeline shows a lot of promise to improve processor performance. The challenge is to do so without synchronization failure while minimizing communication latency and control overhead. This is further complicated by the large clock buffer tree networks in current high-speed microprocessors. Our analysis of a typical clock tree shows that while the delay in these networks is substantial, only one pulse is in the network at a time. This makes it possible to construct a stoppable clock based interface control. Our initial stoppable clock based interface controller integrates the clock buffer network into the ring oscillator, but it loses over half of the clock cycle to control overhead. By adding additional registers, our pipelined interface controller can hide all of the control overhead due to the clock buffer network, reducing the control overhead to about 25 percent. While this limits the applicability of these methods to cases where there is a substantial performance gain due to data-dependence, this shows that it is feasible, even in a very adverse environment such as a modern high-speed microprocessor. The performance gains are achieved with only minimal design changes. In addition to performance gains due to data-dependence in asynchronous modules, a by-product of using a ring oscillator clock is that even the synchronous circuitry adapts to operating conditions. This can improve performance by up to 100 percent, and the design will typically run about 50 percent faster.

In the future, we plan to investigate more complicated pipeline structures. We also plan to study the implementation of these or similar methods to mixed synchronous and asynchronous design at the system-level. Finally, we would like to design a test chip using this interface technology to control a pipelined processor.

Acknowledgments

We are grateful to Professor Alain Martin and his research group at Caltech for their comments on an earlier version of this work. In particular, we would like to thank Dr. Peter Hofstee of IBM for directing us to consider the effects of large clock buffer trees. We would also like to thank Professor George Gray, Robert A. Thacker, Luli Josephson, Wendy Belluomini, Eric Mercer, and Brandon Bachman of the University of Utah for their helpful comments on this paper. Finally, we thank Professor Steve Nowick of Columbia University for his insightful comments.

References

[1] B. Benschneider, A. Black, W. Bowhill, S. Britton, D. Dever, D. Donchin, R. Dupcak, R. Fromm, M. Gowan, P. Gronowski, M. Kantrowitz, M. Lamere, S. Mehta, J. Meyer, R. Mueller, A. Olesin, R. Preston, D. Priore, S. Santhanam, M. Smith, and G. Wolrich. A 300-mhz 64-b quad-issue cmos RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 30(11):1203–1211, November 1995.

[2] 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.

[3] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Stanford University, October 1984.

[4] D. W. Dobberpuhl, R. T. Witek, R. Allmon, R. Anglin, R. Bertucci, S. Britton, L. Chao, R. A. Conrad, D. E. Dever, B. Gieseke, S. M. N. Hassoun, and G. Hoeppner. A 200 mhz 64 bit dual-issue cmos microprocessor. *IEEE Journal of Solid-State Circuits*, 27(11):1155–1167, November 1992.

[5] G. Gopalakrisnan and L. Josephson. Towards amalgamating the synchronous and asynchronous styles, 1993. In collection of papers of the *ACM International Workshop on Timing Issues in the Specification of and Synthesis of Digital Systems.*

[6] Chris J. Myers. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits.* PhD thesis, Dept. of Elec. Eng., Stanford University, October 1995.

[7] 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.

[8] Charles L. Seitz. System timing. In Carver A. Mead and Lynn A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, 1980.

[9] Jakov N. Seizovic. Pipeline synchronization. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 87–96, November 1994.

[10] R.F. Sproull and I.E. Sutherland. Stoppable clock, January 1985. Technical Memo 3438, Sutherland, Sproull, and Associates.

[11] 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.

[12] Kenneth Y. Yun and Ryan P. Donohue. Pausible clocking: A first step toward heterogeneous systems. In *Proc. International Conf. Computer Design (ICCD)*, October 1996.