# A New Method for Asynchronous Pipeline Control

Sam S. Appleton, Shannon V. Morton & Michael J. Liebelt
Department of Electrical & Electronic Engineering,
University of Adelaide, Australia.

## Abstract

*We explore the potential for enhanced performance in asynchronous pipelines by the elimination of unnecessary signalling from the critical path, thus making the common case fast. An improvement of 15% over an optimal two-phase signalling approach for both static and dynamic logic control is demonstrated. We describe extensions to the approach that add functionality with no cycle time overhead.*

## 1 Introduction

Asynchronous system design is a well-known alternative approach to the synchronous design paradigm[1]. However, asynchronous systems can suffer severe performance degradation due to the implementation of local signalling, which tends to place gates which implement control functions in the critical path.

We developed an approach that eliminates the performance bottleneck that is inherent in the asynchronous method by taking the control out of the critical path during normal operation. This approach, which we have termed *Flow Controlled Asynchronous* (FOCA), operates at the maximal rate permitted by the logic blocks during normal operation, and can be halted when long-latency or problematic conditions are detected in the pipeline. In this paper, we demonstrate cycle time improvements of approximately 15% over optimal two-phase asynchronous pipelines.

## 2 FOCA Approach

A well-known asynchronous approach is the micropipeline[2]. A micropipeline models the datapath delay in each stage with a delay element, and uses a two-phase NRZ protocol to implement signalling functions between stages. Such an asynchronous pipeline, which uses delay elements to model the computation delay, is shown in Figure 1. The controller uses two-phase NRZ signals to transfer control information, and we use the $\partial$ symbol to separate two-phase lines from data lines[3]. The Control Element (CE), can be *speed-independent* (vis. the micropipeline) or can rely on bounded delays. The use of bounded delays models leads to significant improvements in performance in two-phase asynchronous systems[3]. However, even this approach cannot totally mitigate the performance penalty of asynchronous signalling, because the $i^{th}$ stage must *wait* for the $i+1^{th}$ stage to accept data before it can reset and accept new data. If we are assuming a bounded-delay model, then the delays of all control and datapath components are known to a good approximation.

Therefore, to eliminate the performance bottleneck, we eliminate the signal which causes the control to be on the critical path – the back-propogating acknowledge signal $\partial ack$. This results in the pipeline structure of Figure 2. The $\partial Go$ signals propagate from one stage to the next, without inter-stage interaction to determine whether the next stage has completed operation. To ensure the pipeline operates correctly, the rate at which operands are *issued* into the pipeline is controlled. This method for pipeline control without the asynchronous acknowledge is termed *Flow Controlled Asynchronous* (FOCA). This approach makes the *common case* fast, when the pipeline is operating normally without halts or with extended latency operands, and leaves in functions to handle the rare cases without detracting from the operating speed of the system.

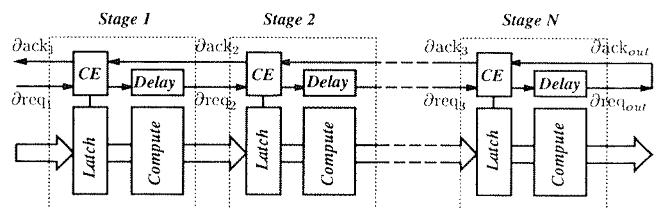



Figure 1: Asynchronous Pipeline

### 2.1 Two-Phase Signalling Gates

The pipelines described in this paper use two-phase signalling, based upon the *Event Controlled Systems*



1066-1395/97 $10.00 © 1997 IEEE

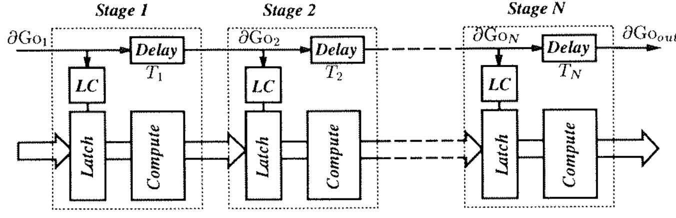
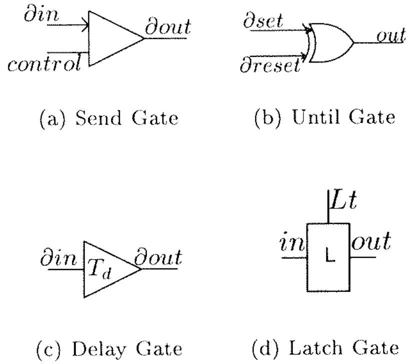



Figure 2: FOCA Pipeline

methodology[3]. The four gates used in this paper are shown in Figure 3.




(a) Send Gate (b) Until Gate

(c) Delay Gate (d) Latch Gate

Figure 3: Two-Phase ECS Gates

The *Send* gate, shown in Fig. 3(a), sends an input event $\partial in$ through to the output, $\partial out$, when the control condition, *control*, is high. When *control* is low, any input event waits until *control* becomes high. The *Until* gate, shown in Fig. 3(b), sets a boolean variable, *out*, true when the input event $\partial set$ occurs, 'until' the event $\partial reset$ occurs, which sets *out* low. The *Delay* gate, shown in Fig. 3(c), simply delays the input event $\partial in$ from propagating to the output event $\partial out$ for a time $T_d$. The *Latch* gate, shown in Fig. 3(d), latches *in* to *out* when *Lt* is high, and holds *out* when *Lt* is low. The implementation of this latch is based on a fast CMOS single-phase latch design[9].

## 2.2 Pipeline Control

The elimination of the acknowledge places a constraint on the *issuing* stage which sends operands into the pipeline. If the latencies of the delay elements of Figure 2 are $T_i \ \ \forall i \in [1 \ldots N]$, then the time between request events into the first stage, $T_{issue}$, must be

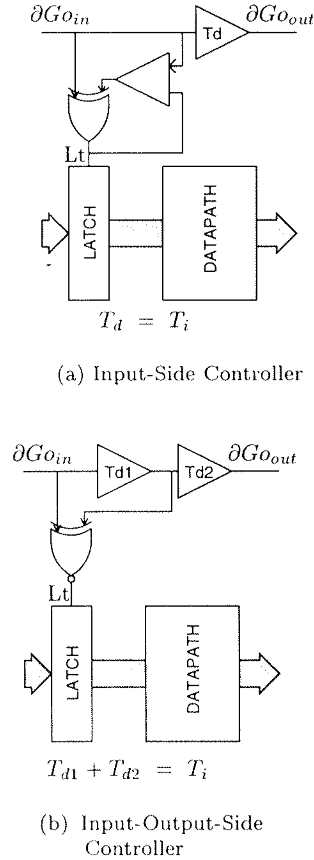
$$T_{issue} \geq max(T_1 + T_{r\,1}, T_2 + T_{r\,2}, \ldots, T_N + T_{r\,N}) + T_{issue\ margin}$$

where $T_{issue\ margin}$ is any delay that might be added as margin for safe operation due to element, temperature or process variation mismatch, which slows down the *issuing* stage of the pipeline. The delays $T_{r\,i}$ are the *recovery* times for each stage – after sending an output $\partial Go$ event, some stages will need recovery time before a new $\partial Go$ can arrive at the input e.g. precharge delays. The delay elements $T_i$ are designed such that

$$T_i = T_{pl} + T_{datapath\ i} + T_{local\ margin}$$

where $T_{pl}$ is the propagation delay of the latches, $T_{datapath\,i}$ is the datapath propagation time, and $T_{local\ margin}$ is any added safety margin required at the local level. Note that this ensures the *forward latency* of the control path is the same as (or greater than) the datapath, which is required for asynchronous pipelines to operate correctly.

The choice of the block which controls the inter-stage latches depends on a number of different factors. Two possible circuits are shown in Figure 4.




(a) Input-Side Controller

(b) Input-Output-Side Controller

Figure 4: FOCA pipeline control circuits

The *Input-Side* (IS) controller only relies on the input

signal, $\partial Go_{in}$, to control the latch line, $Lt$. When a new request event arrives, the controller simply *pulses* the latch line $Lt$ to latch the data. The output event, $\partial Go_{out}$, is issued to the next stage via the delay element. The *Input-Output-Side*(IOS) controller uses both input and output events to control the latch line. An arriving input event, $\partial Go_{in}$, sets the latch line low, and it is set high again $T_{d1}$ after the $\partial Go_{in}$ event. The $T_{d2}$ delay is required to ensure the forward latency of the pipeline matches $T_{pl} + T_i$ i.e.

$$T_{d1} + T_{d2} = T_{pl} + T_{datapath\ i} + T_{local\ margin}$$

The *IOS* controller is more directly suitable for use with dynamic logic and requires fewer control elements, however, the design of the $T_{d1}$ and $T_{d2}$ elements depends on the expected rate of operation of the pipeline. The *IS* controller can be safely *overclocked* without changing any component values, but requires a few extra components to add the control required for dynamic logic.

Another problem than can impact upon the choice of latch controller and the design of the delay elements is that of *delay skew*. The series of delay elements shown in Figure 5 is the control signal path in the FOCA pipeline. Each element has a rising edge delay $T_{r\,i}$ and a falling edge delay $T_{f\,i}$ (the delay elements are assumed to be non-inverting).

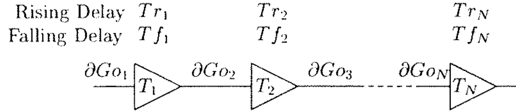



Figure 5: Control Path in FOCA pipelines

Each element will have a small difference between rising and falling delay times, which we term *'delay skew'*(even though it may be extremely small in each element with careful circuit design), and is defined as

$$T_{ds\,i} = T_{r\,i} - T_{f\,i}$$

i.e. the delay skew, $T_{ds\,i}$, is positive if a rising edge is *slower* than a falling edge. As the edge propogates through the delay elements, these delay skews add up, and thus a control circuit for the $i^{th}$ stage *sees* a total delay skew of

$$T_{tds\,i} = \sum_{j=1}^{j<i} T_{ds\,j}$$

i.e. the $i^{th}$ stage sees a difference between rising and falling edges, $T_{tds\,i}$, at the input that is the sum of the delay skews of all previous stages. The response of the control circuit to skew buildups can vary. Even though each stage's delay element ensures that data is ready a set time after the input event, the time between input events varies, which may cause problems with the datapath or the controller. When using the *IOS* controller, an event arriving *early* can cause failure because the latch line $Lt$ has not had time to reset. However, the problem can be minimised when using *identical* delay elements in each stage by inverting the output of each delay, thus inverting the sign of the delay skew in each stage, making the total input delay skew to each stage acceptably small. Even if the delay elements are not equal, the delay skew *sign* can be controlled in each stage (by using inversions), and thus we can minimise delay skew in any one stage to be below an acceptable value.

## 3 Static Logic Pipelines

The maximum date rate of three asynchronous approaches was determined using HSPICE simulations of extracted layouts in a $0.8\mu m$ CMOS process, and compared against a similarly determined performance figure for a FOCA pipeline. We used a Level 13 model with nominal parameters, at $75^oC$ with a 5V supply. A logic delay of 15ns was assumed for each stage, and a five-stage pipeline was constructed. Single-phase latches were used as inter-stage registers[9]. The results are shown in Table 1.

Table 1: Static Logic Pipeline Performance

| Parameter | $\mu$Pipe[2] $2\phi$ SI | AMU[5] $4\phi$ BD | ECS[3] $2\phi$ BD | FOCA $2\phi$ BD |
|---|---|---|---|---|
| Cycle Time | 30.8ns | 26.0ns | 19.4ns | 16.5ns |
| Latency | 102.6ns | 85.4ns | 83.6ns | 79.9ns |
| $P_{avg} \cdot T_{cycle}$ | 403 | 504 | 235 | 226 |

The $\mu$Pipe is the micropipeline, a two-phase speed-independent controller modified to use single-phase latches[4]. The AMU pipeline[5] is a four-phase controller which uses delay elements to model the computation delays. The ECS pipeline[3] is an optimal two-phase controller that uses a bounded-delay model.

The FOCA pipeline improves cycle time by 15%, 37%, and 46% compared to two-phase (ECS), four-phase (AMU), and micropipeline approaches, respectively. The relative performance increase will improve as logic depth is decreased, as FOCA thoughput scales with processing latency without added control overhead.

## 4 Dynamic Logic Pipelines

Dynamic logic is frequently employed in CMOS VLSI to improve power, area and time performance. However, dynamic gates typically require an *activation* signal to separate the precharge and evaluation phases of the logic. The maximum data rates of two asynchronous controllers designed for dynamic logic control, and of a FOCA pipeline using an *IOS* controller, were determined using HSPICE simulation on extracted $0.8\mu m$ CMOS layouts (with the same parameters as for the static pipelines case). The results are shown in Table 2 for evaluation and precharge delays of 15ns and 3ns, respectively.

Table 2: Dynamic Logic Simulation Results

| Measured Parameter | FOCA $2\phi$ BD | MPP $2\phi$ BD[3] | Amulet-2 $4\phi$ BD[5] |
|---|---|---|---|
| Cycle Time | 19.2ns | 22.8ns | 32.1ns |
| Latency | 83.0ns | 92.9ns | 93.0ns |

The MPP is a fast two-phase dynamic logic pipeline structure using ECS[3]. The Amulet-2 is a four-phase controller modified to use dynamic logic[5].

The cycle time improvement is 15% and 40% over two and four phase approaches, respectively. The design of the FOCA controller was deliberately conservative, and improvements could be gained by being more aggressive with the design of interaction between the control and datapath.

## 5 Implementing Haltability

The ability to halt FOCA pipelines is required because at certain times errors or long-latency operands will enter the pipeline. We assume that the stage which will cause the problem can *detect* it and assert a signal, called *HaltDetect*. All previous stages must then *stop* and wait for the error condition to be resolved. One method of doing this would be to drive the *HaltDetect* signal back up the pipeline, and have this signal stop any new events propagating between stages. However, the pipeline is still essentially asynchronous, and this global *halt* signal may cause metastability failures in earlier components. Another method, more preferable but more complex, is to propagate the halt back to the previous stage when the current stage becomes *busy*, and the halt condition is active. A dynamic logic circuit for this function is shown in Figure 6.

An arriving halt signal, *HaltIn*, only causes this stage to halt when the stage is *busy* i.e. is processing an

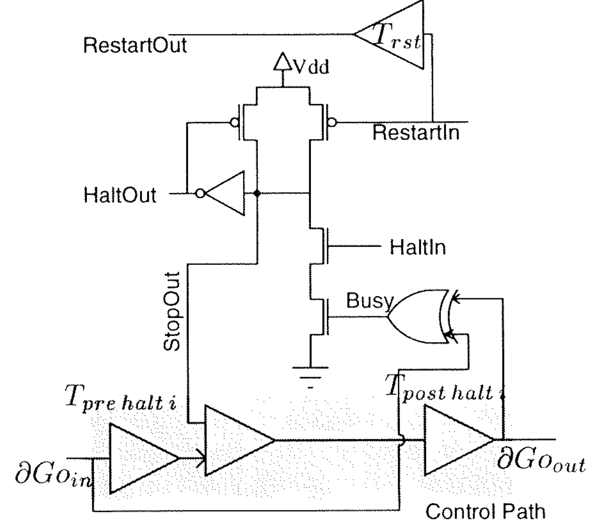



Figure 6: FOCA dynamic halting circuit

operand. This signal then propagates to the previous stage. The delay $T_{rst}$ is an asymmetric delay element that propagates the *Restart* condition back up the pipeline when the halt is removed, and is required in the general case to ensure the pipeline operates correctly when exiting the halted state. The *Restart* signal is generated by the stage that generated the halt to indicate that the condition has been resolved and the pipeline can start moving again. If the stage delay $T_i$ is such that,

$$T_i \gtrsim T_{issue} - T_{gate\ delay}$$

then this delay element, $T_{rst}$, can be omitted.

To determine whether a pipeline can be safely halted, we must ensure that the latest the *StopOut* signal can be asserted is earlier than the arrival of the input event to the *Send* gate which halts the stage. For a $N$-stage pipeline, the time difference $T_{hm\,i}$ in each stage must be positive in order to halt, where

$$\begin{aligned} T_{hm\,i} &= T_{pre\,halt\,i} - (T_{back\,i} + T_{control} - T_{sm}) \\ \text{where } T_{back\,i} &= T_{hp} + max(T_{busy}, T_{back\,i+1} - \\ &\qquad T_{issue} + T_i, T_{hd\,i}) \\ \text{and } T_{back\,N} &= max(T_{busy}, T_{hd\,N}) + T_{hp} \end{aligned}$$

where $T_{issue}$ is the issuing rate of the pipeline, $T_{control}$ is the margin required between an input event to a *Send* gate and a change in control input, $T_{sm}$ is the delay between *HaltOut* and *StopOut* in the halt circuit, $T_{hp}$ is the delay of the halt generation circuit, $T_{busy}$ is the

time taken to generate the busy signal from the arrival of the event $\partial Go_{in}$, and $T_{hd\,i}$ is the time taken to detect a halt condition in the $i^{th}$ stage (set to zero if the $i^{th}$ stage does not generate a halt signal).

For a pipeline where all the delays are equal (i.e. $T_i = T_{issue}\ \forall i$), and the maximum time to detect a halting condition in any stage is $\approx 5ns$, then $N_{halt}$, the number of pipeline stages which can be halted safely, is

$$\begin{aligned} N_{halt} &= (T_{issue} - T_{hd})/T_{hp} \\ &\approx 10 \end{aligned}$$

i.e. for a pipeline with a $15ns$ computation delay and a gate delay of $\approx 1ns$, a 10-stage pipeline can be halted safely. Note that if the issuing rate is modified such that

$$T_{issue} \geq T_{hp} + max(T_1, T_2, \ldots, T_N)$$

then an arbitrary length pipeline can be halted safely. In a pipeline which uses dynamic logic, this condition is automatically satisfied because the precharge time is typically longer than the latch delay. This precharge time must be added to the issuing rate as a *recovery* time, satisfying the above constraint and making the halting of an arbitrary length dynamic pipeline possible.

## 6 Conclusion

The FOCA method eliminates the asynchronous acknowledge signal from the critical communication path in an asynchronous pipeline. This eliminates the performance bottleneck, resulting in cycle time improvements of 15% over an optimal two-phase bounded-delay asynchronous method when using both static and dynamic logic. The design of the controllers was deliberately conservative, and more aggressive design will improve performance.

We are currently extending the FOCA approach to variable delay stages and multi-rate pipelines, and applying the method to the design of digital filters, interchip signalling components for asynchronous systems, and a RISC microprocessor. The work proceeds in parallel with our work on the two-phase bounded-delay asynchronous approach *Event Controlled Systems*, which by itself significantly improves the performance level of asynchronous components, as can be seen from the performance figures quoted in this paper.

We would like to thank Dr. D.A. Pucknell for his pioneering contributions and Andrew Beaumont-Smith for comments, and the financial support of the Australian Research Council and the Frank Perry Scholarship.

## References

[1] Hauck, S., "Asynchronous Design Methodologies : An Overview", *Proceedings of the IEEE*, Vol. 83, No. 1, January 1995.

[2] Sutherland, I.E., "Micropipelines", *Communications of the ACM*, pp.720-738, June 1989.

[3] Appleton, S.S., Morton, S.V., & Liebelt, M.J., "High Performance Two-Phase Asynchronous Pipelines", *to appear in IEICE ED Journal*, March 1997.

[4] Day, Paul, & Woods, J. Viv, "Investigation into Micropipeline Latch Design Styles", *IEEE Transactions on VLSI Systems*, Vol. 3, No. 2, June 1995.

[5] Furber, S.B., & Day, P., "Four-Phase Micropipeline Latch Control Circuits", *IEEE Transactions on VLSI Systems*, Vol. 4, No. 2, June 1996.

[6] Furber, S.B., & Liu, J., "Dynamic Logic in Four-Phase Micropipelines", *Second Intn'l Symposium on Adv. Research in Async. Circuits & Systems*, March 1996, Aizu-Wakamatsu, Japan.

[7] Weste, N., & Eshragian, K., "Prinnciples of CMOS VLSI Design", *Second Edition*, Addison-Wesley.

[8] Morton, S.V., Appleton, S.S., & Liebelt, M.J., "An Event Controlled Reconfigurable Multi-Chip FFT", *Intn'l Symposium on Adv. Research in Async. Circuits and Systems*, Salt Lake City, Utah, November 1994.

[9] Svensson, Christer, & Yuan, Jiren, "High-Speed CMOS circuit technique", *IEEE Journal of Solid-State Circuits*, 27(3):382-388, March 1992.