# Concurrency-Oriented Optimization for Low-Power Asynchronous Systems*

Luis A. Plana† Steven M. Nowick‡

Department of Computer Science
Columbia University
New York, NY 10027

CUCS-017-96

October 1996

**Abstract**

We introduce new architectural optimizations for asynchronous systems. These optimizations allow application of *voltage scaling*, to reduce power consumption while maintaining system throughput. In particular, three new asynchronous sequencer designs are introduced, which increase the concurrent activity of the system. We show that existing datapaths will not work correctly at the increased level of concurrency. To insure correct operation, modified latch and multiplexer designs are presented, for both dual-rail and single-rail implementations. The increased concurrency allows the opportunity for substantial system-wide power savings through application of voltage scaling.

**Keywords**

Asynchronous Design, Data Hazards, Handshaking, Hazards, Latches, Low Power, Sequencers, Voltage Scaling.

---

*This is an expanded version of a paper by the same name, appearing in the 1996 International Symposium on Low Power Electronics and Design [19].

†This author was supported by grants from CONICIT and UNEXPO, Venezuela.

‡This author is supported by an NSF CAREER Award MIP-9501880 and by an Alfred P. Sloan Research Fellowship.

# 1 Introduction

Interest in low-power and asynchronous systems has grown considerably in recent years. The constant increase in the use of battery-operated portable devices like cellular phones, notebook computers, and even implantable pacemakers, has made low power consumption a high priority. Power issues are also becoming critical for non-portable systems. Low-power operation can reduce the need for heat dissipation and cooling devices; it can also extend the life of system components, by providing a less stressful operating environment.

A wide range of techniques is used to reduce circuit power consumption. These techniques approach low-power operation at different levels of synthesis, including IC technology optimization, low-power circuit design, architecture or structural optimizations, algorithm level optimization and system-wide low power techniques [4, 3, 13]. Chandrakasan *et al.* [4] show that concurrency is a key to architecture-driven optimizations for low-power operation. The increased throughput obtained through concurrent operation allows the reduction of the power supply voltage, *i.e., voltage scaling* [4, 13, 24].

The focus of this paper is on asynchronous designs for low power. In principle, asynchronous systems have the potential for low power operation for two reasons. First, these systems have no global clock; in contrast, clock distribution is a major source of power consumption in synchronous design. Second, asynchronous circuits have an inherent *automatic power-down operation*: modules are activated only when their operations are needed. Low-power operation is a major focus of recent asynchronous design, including large-scale, fabricated examples like a low-power infrared communications chip [9], an asynchronous implementation of the ARM microprocessor [6], and an asynchronous error corrector for a DCC player [24].

A number of asynchronous design methods have been introduced recently [21, 10, 2, 16, 28, 15, 25]. Several methods build asynchronous circuits as networks of communicating modules. Every module is mapped to a circuit element in a library of self-timed modules. Such systems are *macromodular*, since they are constructed by combining modules into a working system. Macromodular circuits are robust and usually have few timing assumptions. Macromodules are particularly well-suited for methods that approach circuit design as a programming activity. For example, van Berkel *et al.* [24, 25], have developed a method to automatically design low-power asynchronous circuits from high-level *Tangram* programs. The programs are compiled, using syntax-directed translation, into *handshake circuits*, an intermediate-level representation of a circuit as a network of macromodules.

The goal of this paper is to reduce power in asynchronous macromodular systems. Our focus is on *non-pipelined* (*i.e.,* sequential) systems, which are commonly used in low-power DSP applications with modest performance requirements (see [14, 24]). The operation of these systems is controlled by basic elements called *sequencers*.

Our strategy is to use *architectural optimization* to increase the throughput of a sequential system. This increased throughput must be achieved without increasing the switching activity required for a computation (otherwise energy consumption could increase). *Voltage scaling* [4, 13, 24] is then applied, to reduce both power consumption and throughput. The resulting system has no net loss of performance, but a significant reduction in power.

In particular, we present the following new contributions. First, we introduce three new designs for asynchronous sequencers. Each design increases the concurrency of the datapath operations in the entire system. Second, we show that existing asynchronous datapaths will not operate correctly at this increased level of concurrency. We therefore modify the datapath to insure correct operation. Specifically, we introduce new designs for asynchronous latches and multiplexers that handle concurrent operation safely in (a) "dual-rail" datapaths, and (b) "single-rail" datapaths (described below).

For dual-rail datapaths, our new components *allow roughly twice the throughput* of existing sequential designs. In this case, after voltage scaling, the energy dissipation of the entire system is reduced by a factor of 2.5.

For single-rail datapaths, two existing schemes are discussed. Our new components result in twice the throughput of the first scheme, and roughly the same performance as the second one. However, our simpler approach has advantages over the latter in ($i$) ease of design and ($ii$) glitch avoidance in the datapath.

**Organization of the paper.** The paper is organized as follows. Section 2 reviews background on power consumption and asynchronous circuits. Section 3 presents a brief overview of our approach. In section 4, existing sequencers are examined, their limitations are pointed out, and three new low-latency sequencer designs are introduced. Section 5 discusses the operation of dual-rail datapaths and a basic problem related to concurrent operation is described. New latch designs to allow correct operation are introduced in this section. Similar modifications for single-rail datapaths are introduced in section 6. In Section 7, modified multiplexer designs are presented. Section 8 presents results of analysis and SPICE simulations, and Section 9 presents conclusions.

# 2 Background

In this section we review some fundamental aspects of power consumption in CMOS circuits —our target technology—, and basic concepts related to the operation of asynchronous circuits.

## 2.1 Power Consumption in CMOS Circuits

There are three major sources of power consumption in CMOS circuits. *Switching energy* is associated with transitions on gate outputs. *Short-circuit energy* consumption is caused by simultaneous conduction, during a transition, of *pull-up* and *pull-down* stacks, allowing current flow directly from the power supply to ground. Finally, *leakage energy* occurs in standby mode, and is caused by substrate currents and by sub-threshold conduction in off transistors. We only consider transition energy because in most CMOS circuits this energy dominates the other two and contributes up to 90% of the total energy [4].

In asynchronous systems there is no global clock, so the metric of interest is the *total energy of a computation.* A well-known expression for this energy is:

$$\text{Energy of a computation} = \frac{1}{2} \cdot n \cdot C_L \cdot V_{dd}^2$$

Where $n$ is the total number of transitions in the computation, $C_L$ is the load capacitance being charged/discharged, and $V_{dd}$ is the power supply voltage. Energy dissipation can be reduced by reducing the capacitance, the number of transitions, or the supply voltage. Since energy depends quadratically on the supply voltage, *supply voltage scaling* is an especially attractive scheme for power reduction [4]. Unfortunately, voltage scaling has the undesirable effect of reducing the speed of the circuit.

Several techniques are used to compensate for this performance penalty. In particular, *architecture-driven voltage scaling* [4, 3] combines architectural optimizations —to increase the concurrent activity and the throughput of the system— with voltage scaling, to reduce the power consumption without affecting the throughput. Our goal is therefore to *increase the concurrency*, and hence the throughput, of a system. If the increase in performance is achieved without increasing the switching activity required for the computation, a substantial reduction in power is possible after voltage scaling, with no net loss in performance.

## 2.2 Asynchronous Circuit Operation

In this paper, we focus our attention to asynchronous macromodular systems. This type of asynchronous circuit is designed as a network of predefined data and control modules [2]. Instead of a global clock signal, communication channels between modules use handshaking to synchronize their operation and data interchange.

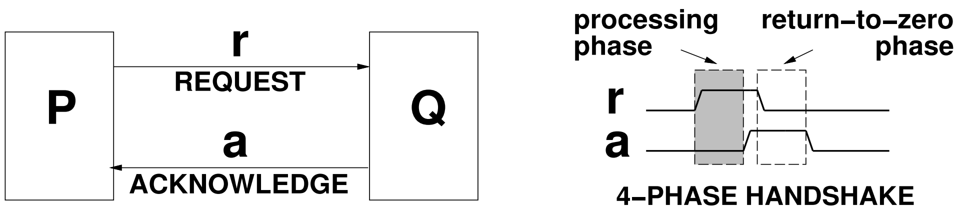



Figure 1: 4-phase handshaking for module communication.

### 2.2.1 Control Signaling

Control signaling usually follows a *4-phase handshake protocol* [20], which is implemented using *request* and *acknowledge* signals. Figure 1 shows two macromodules that communicate with each other. Initially, processes are idle and both signals are de-asserted. *module* $P$ asserts $r$ to request *module* $Q$ to start the *processing phase*. When $Q$ finishes processing, it asserts $a$ to start the *acknowledge phase*. The wires must return to their idle state so $P$

de-asserts $r$, starting the *return-to-zero phase*. Finally, $Q$ de-asserts $a$ to return to the *idle phase*. Usually, computation takes place *only* during the processing phase. The rest of the phases, especially the return-to-zero phase, represent *dead time* from the point of view of computation.

### 2.2.2 Data Communication

When data communication is involved an encoding scheme is used to represent and transmit data. Two encodings are most common [20]: *dual-rail* and *single-rail*.

• **Dual-rail Data.** Data is encoded using two wires for every data bit. Codes 01 and 10 represent '1' and '0' data values, respectively, and code 00 represents the *spacer* or *idle* state. This encoding effectively combines data and control in the same wires: the idle state indicates invalid data, and 01 and 10 indicate valid data and the data value itself. This is a very robust code that guarantees correct operation with arbitrary delays in the circuit. However, implementations typically suffer from severe power and area penalties due to the duplication in the number of wires and transitions.

• **Single-rail Data.** This code uses one wire for every data bit (as in synchronous designs), and one additional wire, called a *data-valid signal*, for control. The collection of all data bits and the data-valid signal is called a *data bundle*. This code has good power and area costs, comparable to synchronous implementations. The correct operation of single-rail circuits relies on a local timing assumption: all data wires must be valid and stable before the data-valid signal is asserted. This timing assumption is called a *bundling constraint* [21].

## 3 Overview

Two basic computation structures are used in asynchronous systems: *pipelined* and *sequential*. Pipelined computation is usually used in high-performance processors, *e.g.* AMULET2 [6]. The design of asynchronous pipeline control is a very active research area [21, 7, 27]. Sequential computation is used in some DSP systems with moderate timing requirements, usually aimed at low-power operation, *e.g.* FIR filter bank [14], DCC error corrector [24]. Active research is done in sequencer control also [22, 25, 1].

The focus of this paper is on sequential computation. The optimizations introduced in this paper improve the throughput of an asynchronous sequential system by increasing the amount of concurrent activity. A key control element, that has a large impact on concurrency, is the sequencer. In Section 4, we review existing sequencers, point out their limitations, and introduce three new sequencer designs which result in higher system throughput.

Next, we show that existing datapaths will not work correctly at this increased level of concurrency. In particular, *data hazards* may appear. We therefore introduce new latch designs to allow correct operation for dual-rail asynchronous datapaths in Section 5. Two alternative modifications are made for single-rail datapaths in Section 6. Section 7 presents modified multiplexer designs, to be used with both dual-rail and single-rail datapaths.

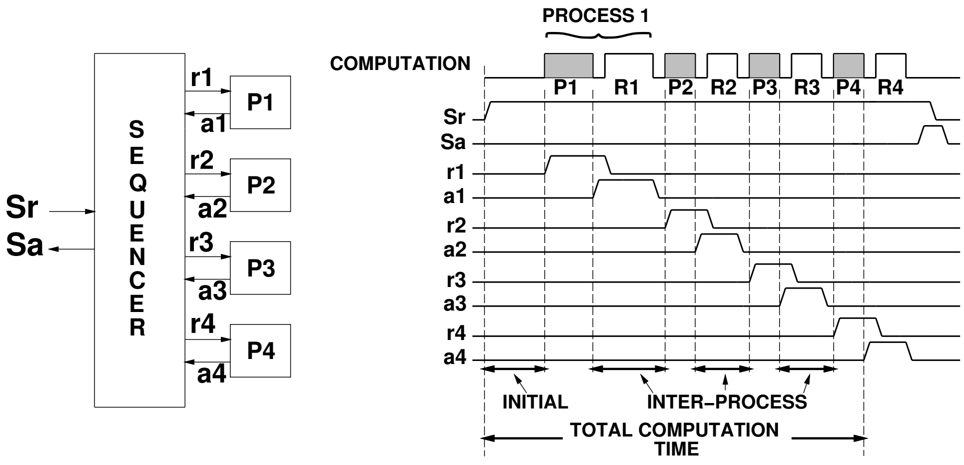
Finally, in Section 8, we combine the architectural optimizations with voltage scaling. The results of SPICE simulations indicate that a significant reduction in power can be achieved over existing designs, without affecting system throughput.

# 4 Asynchronous Control

A basic control operation in macromodular systems is the sequencing of computations or data processing actions. Such sequences can be very long. For example, Bailey [1] reports that the longest sequence in the asynchronous error decoder circuit for a DCC player [24] consists of 48 processes. Two common operation protocols are used in asynchronous sequencers: *sequential* and *concurrent.*

• **Sequential Protocol.** Figure 2(a) shows a sequencer module controlling the operation of four processes ($P_1$, $P_2$, $P_3$, $P_4$). The sequencer communicates with process $P_i$ using request ($r_i$) and acknowledge ($a_i$) signals. Figure 2(b) shows how the sequencer communicates with each process using 4-phase handshaking.

In this basic *sequential protocol*, the sequencer executes a complete 4-phase handshake with process $P_i$ before starting a handshake with $P_{i+1}$. The behavior of this sequencer is




(a) (b)

Figure 2: Sequencer Block Diagram and Sequential Operation.

described by the expression:

$$*(s_r\uparrow; r_1\uparrow; a_1\uparrow; r_1\downarrow; a_1\downarrow; \ldots; r_N\uparrow; a_N\uparrow; r_N\downarrow; a_N\downarrow; s_a\uparrow; s_r\downarrow; s_a\downarrow)$$

Figure 2(b) shows schematically how, in a sequential protocol, processing ($P_i$) and return-to-zero ($R_i$) phases alternate, resulting in a long dead time between computations. In a sequential protocol, process $P_{i+1}$ cannot be started until the previous return-to-zero phase has completed.

This dead or non-computation time between processing phases, shown in Figure 2(b), is called *inter-process latency*. Other important parameters to evaluate the performance of a sequencer, also shown in Figure 2(b), are the *initial latency*, the time from the activation of the sequencer to the start of the first computation, and the *total computation time*, the time from the activation of the sequencer to the end of the last computation. Optimization techniques for such sequencers typically focus on reducing the internal control latency of the acknowledge and idle phases, which contribute to the inter-process latency.

• **Concurrent Protocol.** A more efficient approach is to introduce concurrent operation. In a *concurrent protocol*, the sequencer can start process $P_{i+1}$ without waiting for $P_i$ to complete its return-to-zero phase. In this way, every processing phase $P_{i+1}$ is *overlapped*

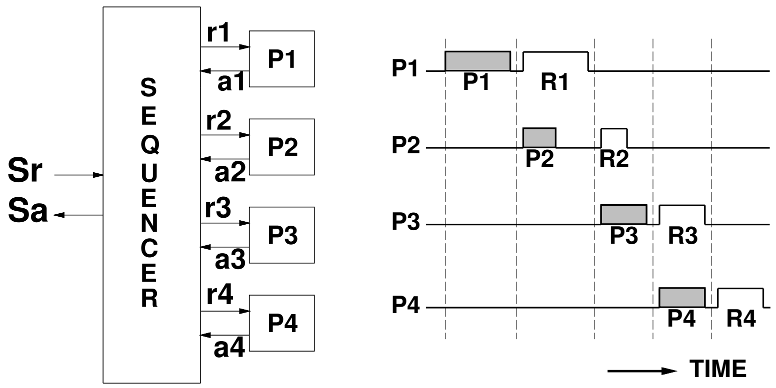



Figure 3: Concurrent Sequencer Operation.

with the return-to-zero phase $R_i$ of the previous process, as shown schematically in Figure 3.

As we will show later, a number of concurrent protocols can be implemented, resulting in different levels of concurrency in the system.

## 4.1 Previous Sequencers

We now review existing sequencers that implement sequential and concurrent protocols and indicate their limitations.

### 4.1.1 Sequential Approaches

• **Tangram Sequencer.** In Tangram, 2-way sequencing is implemented using the *SEQ* operator [25], shown in Figure 4(a). The sequencer is activated on its *passive* port, or channel, $S$ (a passive port is indicated by a small white circle). The sequencer then communicates on *active* ports $P1$ and $P2$ to activate the first and second processes, respectively (an active port is indicated by a small black circle).

Channels are implemented using request and acknowledge wires ($S_r$ and $S_a$ for channel $S$, and $r_i$ and $a_i$ for channel $Pi$). A complete 4-phase handshaking occurs on port $P1$, followed by a complete 4-phase handshaking on port $P2$. The behavior of the SEQ operator can be described by the following expression:

$$*(s_r\uparrow; r_1\uparrow; a_1\uparrow; r_1\downarrow; a_1\downarrow; r_2\uparrow; a_2\uparrow; s_a\uparrow; s_r\downarrow; r_2\downarrow; a_2\downarrow; s_a\downarrow)$$

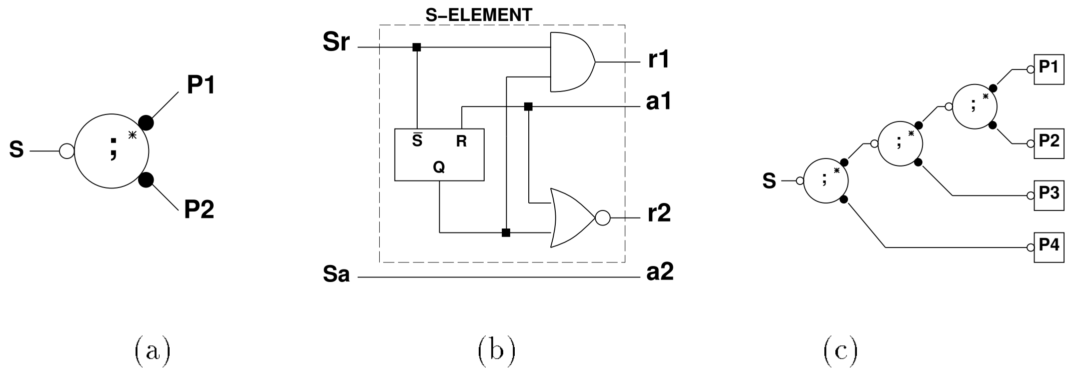



Figure 4: Tangram Sequencer: (a) SEQ Symbol, (b) SEQ Circuit, (c) Tree Sequencer.

An implementation of the SEQ operator is shown in Figure 4(b). This circuit is *speed-independent* [11], *i.e.,* it operates correctly assuming arbitrary, finite, gate delays. An N-way sequencer consists of SEQ operators connected in a tree structure, as shown in Figure 4(c). There are two problems with the Tangram sequencer: (*i*) it has a long initial latency and (*ii*) it has long inter-process latencies (see Section 8, Results).

• **Martin Sequencers.** In [10], Martin presents two N-way sequencers. The Tangram N-way sequencer corresponds exactly to a *Q-element-based* Martin sequencer and it has comparable performance. A *D-element based* sequencer provides no overall performance improvement.

• **Josephs/Bailey Counter-Decoder Sequencer.** In [1], Bailey and Josephs introduced a centralized sequencer, based in a *counter-decoder* architecture. The counter centralizes the state of the sequencer, and the decoder distributes the signals to the processes. The circuit is speed-independent and it is currently used in several designs. This sequencer was designed using a formal procedure based on SI-Algebra (see [8] for details). The implementation has improved initial and inter-process latencies compared to the Tangram tree sequencer. Minor problems are that the circuit is not modular and is designed to work with an even number of processes.

• **Josephs/Bailey Chain Sequencer.** Bailey and Josephs [1] also introduced a distributed sequencer built as a *linear chain* of n modules, each controlling a process. The modules assume *fundamental-mode* operation [23]. In fundamental mode, no new inputs can arrive until the component has stabilized from a previous input change. This design also has better initial and inter-process latencies than Tangram's.

To summarize, all of these sequencers implement a sequential protocol and, as a result, have long inter-process latencies due to the return-to zero phase. In addition, some designs also have a long initial latency.

### 4.1.2 Concurrent Approaches

There have been a few attempts to implement concurrent sequencers, but each has limitations.

• **Unger Tree Sequencer.** Unger [22] presented a *2-step module* that implements a concurrent 2-way sequencer. The 2-step assumes fundamental-mode operation and relies on reasonable timing assumptions. An N-way sequencer is built as a balanced tree of 2-step modules [22]. There are several problems with this implementation: (*i*) the sequencer has a long initial latency, (*ii*) the inter-process latency is different for every pair of processes and can be several gate delays, depending on how far up and down the tree the signals have to propagate, and (*iii*) the area and power consumption of this structure are significantly worse than the previous designs (see Section 8, Results).

• **Farnsworth 2-way Sequencer.** Farnsworth *et. al* [5] introduced a concurrent 2-way sequencer as part of a FIFO control unit. No N-way extensions were presented. We explored three different tree structures to obtain an N-way sequencer: a left-branching tree, a right-branching tree, and a balanced tree. Our attempts resulted either in (*i*) a long initial latency, or (*ii*) long inter-process latencies.

Our goal is to implement a concurrent sequencer efficiently, overcoming the problems and deficiencies pointed out in the previous designs.

## 4.2 New Concurrent Sequencers

We now introduce 3 new concurrent sequencer designs that have good latency, area and power characteristics.

### 4.2.1 Burst-Mode Concurrent Sequencer

Our first sequencer implements a concurrent protocol with *tightly-coupled* overlap: processing phase $P_i$ overlaps *exactly* return-to-zero phase $R_{i-1}$. The key point is that this sequencer waits until *both* concurrently operating phases, $R_{i-1}$ and $P_i$, complete before starting the next two overlapped phases, $R_i$ and $P_{i+1}$, as shown in Figure 5.

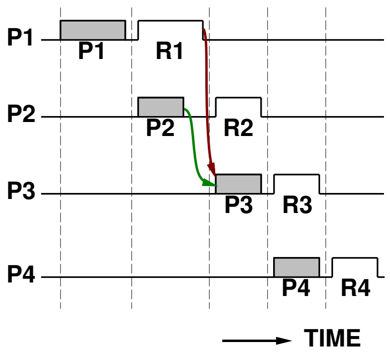


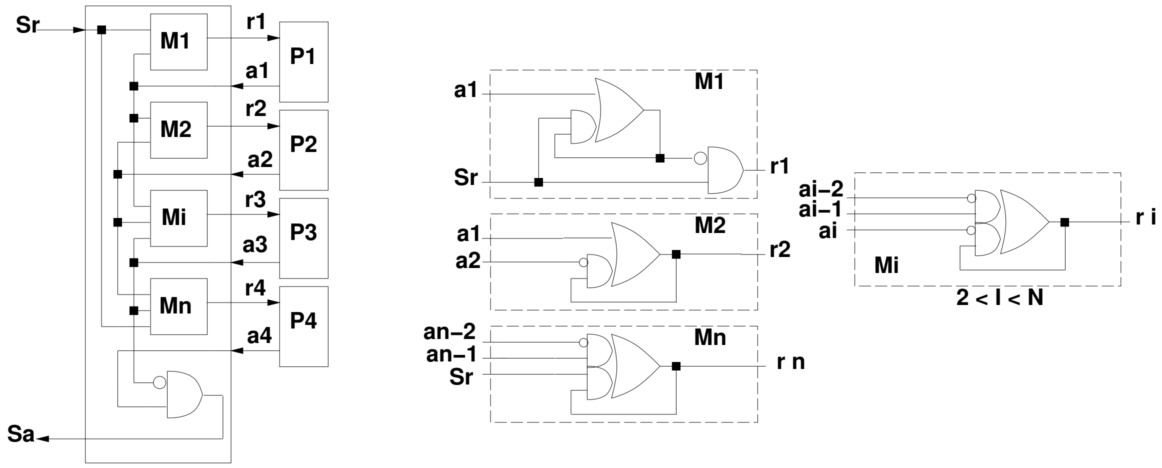

Figure 5: Burst-Mode Sequencer Operation.

We synthesized the circuit using an existing *burst-mode* asynchronous tool, UCLOCK [15], with extensions to incorporate output feedback. The result is a modular design, well suited for distributed control. Our N-way sequencer has $N$ modules organized into 4 types as shown in Figure 6(a): module $M_1$ controls process $P_1$, $M_2$ controls $P_2$, $M_i$ modules control $P_3$ to $P_{N-1}$, and $M_n$ controls $P_N$.

Figure 7 shows two $M_i$ modules, each controlling a process. The modules have good latency, area and power. In typical computation, the inter-process latency, —the time from completion of $P_i$'s processing phase ($a_{i-1}\uparrow$) to the start of $P_{i+1}$'s processing phase ($r_i\uparrow$)—, is *only 2 CMOS gate delays* (an AOI-gate followed by an inverter).[1] Also, each module contributes only 8 gate output transitions and 14 transistors to the energy consumption and area of the system.

The correct operation of the sequencer relies on modest timing assumptions related to the fact that each process' acknowledge signal, $a_i$, is forked to two different modules. In

[1]If a return-to-zero phase is unusually long, the inter-process latency may increase due to synchronization dependencies. In particular, $P_i$ can be delayed by $R_{i-2}$.




(a) (b)

Figure 6: Burst-Mode Sequencer: (a) Block Diagram, (b) Module Implementations.

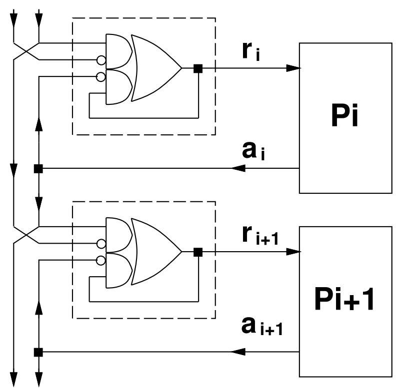



Figure 7: Burst-Mode Modules controlling Processes.

particular, $a_i\uparrow$ generates, concurrently, $r_i\downarrow;a_i\downarrow$ in module $i$, and $r_{i+1}\uparrow$ in module $i+1$. The change in $r_{i+1}$ must propagate back to the input of the complex gate before $a_i\downarrow$ arrives at this gate, to avoid an unspecified change in $r_{i+1}$. Since $r_{i+1}$ has to propagate through a short wire while $r_i$ has to propagate through process $P_i$, this restriction is quite reasonable in practice.

One restriction of our burst-mode sequencer is that a long return-to-zero phase, such as $R_1$ in Figure 5, may unnecessarily delay the start of the next processing phase ($P_3$). This observation leads to our second design.

#### 4.2.2 Optimized Concurrent Sequencer

Our second sequencer allows greater concurrency, by using a more relaxed synchronization requirement. By starting $P_{i+1}$ *as soon as* $P_i$ is finished, *independently* of the status of $R_i$, a faster sequence of processing phases is allowed. The operation of this *optimized sequencer* is shown in Figure 8. Processing phase $P_{i+1}$ depends *only* on the completion of $P_i$. The return-to-zero phases have similar dependencies to guarantee complete synchronization of the system.

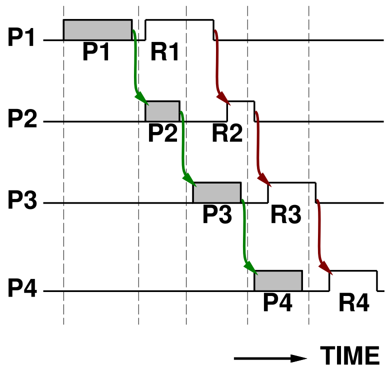



Figure 8: Optimized Sequencer Operation.

Our optimized sequencer design is shown in Figure 9(a). Although similar to the burst-mode sequencer, three improvements are clear: $(i)$ a wire replaces the AND gate that generates $S_a$; $(ii)$ each module has one fewer input ($a_{i-2}$), resulting in a reduced fan-out of the processes' acknowledge signals; and $(iii)$ the module implementations, shown in

Figure 9(b), are more efficient in terms of area and power.

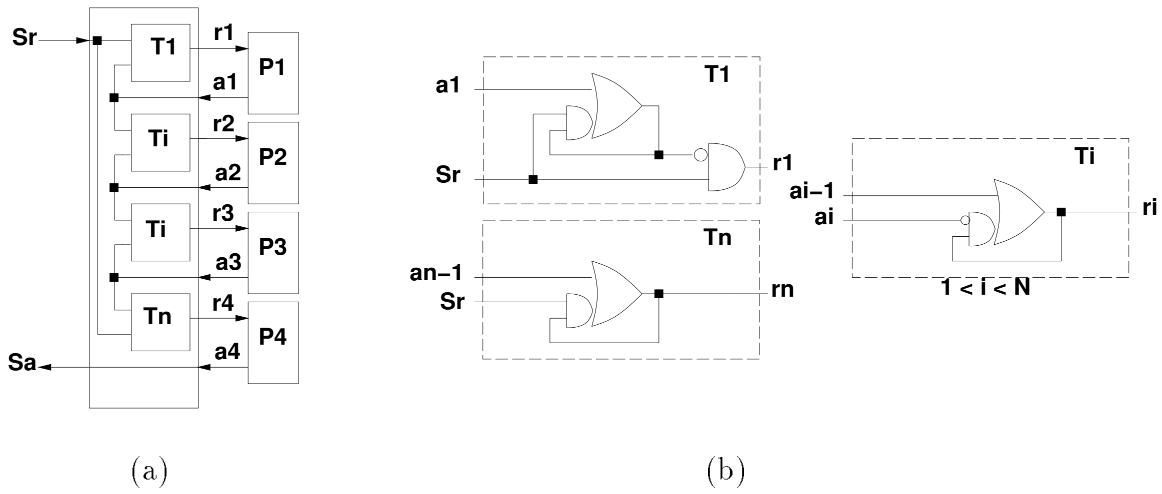



Figure 9: Optimized Sequencer: (a) Block Diagram, (b) Module Implementations.

Figure 10 shows two optimized modules, each controlling a process. These modules are more efficient in terms of speed, power and area: the inter-process latency is guaranteed to be 2 CMOS gate delays, and the contributions of a module to the energy dissipation and area are only 6 gate output transitions and 10 transistors, respectively.

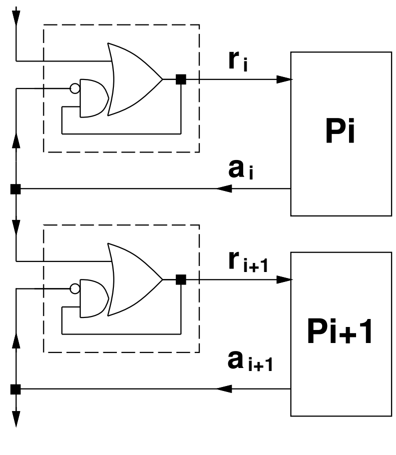
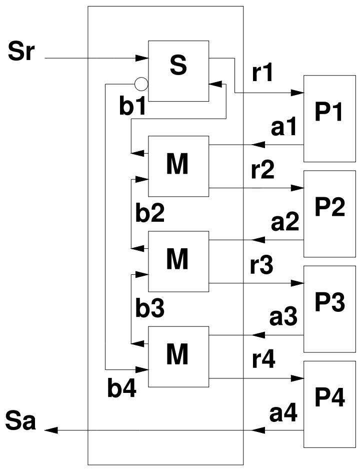



Figure 10: Optimized Modules.

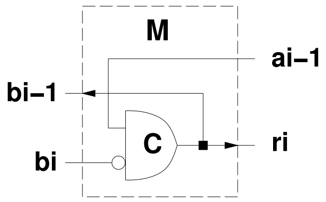
The correct operation of the sequencer relies on similar timing assumptions as the burst-mode sequencer. In realistic settings, these assumptions are very reasonable and should not be a problem. If these timing requirements cannot be met, a more robust sequencer must be used, as presented in the following subsection.

### 4.2.3 Speed-independent Concurrent Sequencer

In a speed-independent circuit, an input is not allowed to change until its previous change has been acknowledged through an output change. As pointed out in the previous sections, the burst-mode and the optimized sequencers do not meet this requirement, due to the fork in signal $a_i$. Figure 11(a) shows our *speed-independent sequencer*. The fork in $a_i$ is eliminated, sending the signal only to module $i+1$, which generates $r_{i+1}$. A signal from this module to the previous one ($b_i$) is added in order to report that the change in $a_i$ has been absorbed, allowing further changes in $r_i$ to take place.




(a)




(b)

Figure 11: Speed-Independent Sequencer: (a) Block Diagram, (b) Module Implementation.

The first process, $P_1$, is controlled by a handshake S-element [26], shown in Figure 4 above. Each of the remaining processes is controlled by an $M$ module shown in Figure 11(b). The module has a very efficient implementation: a single *C-element*.[2] Also, each module contributes only 6 gate output transitions and 12 transistors to the energy

[2] A 2-input Muller C-element produces a '1' output when both inputs are '1' and a '0' output when both inputs are '0'. When the inputs are different, the output remains at its previous value.

consumption and area of the system.[3]

# 5 Modified Dual-rail Datapaths

We now briefly review the operation of dual-rail datapaths and examine their interaction with concurrent sequencers. We point out that existing datapaths will not operate correctly at the increased concurrency provided by the new sequencers. In particular, data hazards can arise. We then present modified dual-rail latch and multiplexer designs that allow correct operation of the datapath. Single-rail datapaths will be covered in the following section.

## 5.1 Basic Dual-Rail Datapath Operation

Figure 12 shows schematically a 2-stage dual-rail datapath. In this example, a sequencer controls two processing actions, *Process 1* implements $Z = F(Y)$ and *Process 2* implements $X = G(W)$. $W$, $X$, $Y$, and $Z$ are variables that hold data, and $F$ and $G$ are blocks of combinational logic that implement the desired functions.

$$Z = F(Y);$$
$$X = G(W);$$

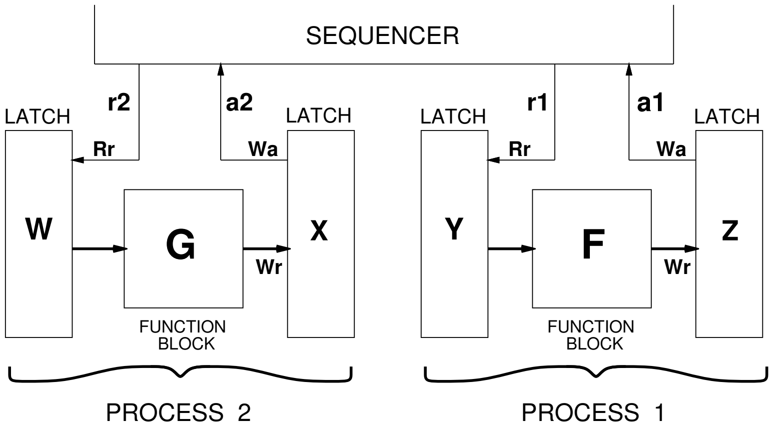



Figure 12: Sequencer controlling 2 Dual-Rail Datapath Processes.

Functions are implemented using *hazard-free* combinational logic that operates on dual-rail input data and generates dual-rail outputs. Hazard-free operation is required because

[3]This sequencer works for $N > 2$ (the circuit will deadlock for $N = 2$). A 2-way sequencer can be built using an S-element to control the first process, an $M$-module to control the second process and an additional $M$-module to generate $S_a$.

any glitch in the data wires can be interpreted as a valid data signal and produce erroneous operation. Dual-rail variables are usually implemented using a latch with separate read and write ports. The latches are opaque when inactive.

The operation of *Process* 1 is as follows. The initial request $r_1\uparrow$ propagates to source variable $Y$ as a *read request*. $Y$ sends data to block $F$ which computes $F(Y)$. Dual-rail data is used and there is no separate handshake signal to control this communication. When $F(Y)$ is computed, the result is sent to the destination variable ($Z$). The dual-rail data itself serves as the *write request* signal for $Z$. When the data is stored, the processing phase is complete and $Z$ sends its acknowledge signal to the controller as $a_1\uparrow$. The return-to-zero phase is initiated with $r_1\downarrow$ which propagates to $Y$ to finish the read operation. Data signals return to the idle state to indicate the end of the write request to $Z$. $Z$ responds by de-asserting $a_1$ which completes the operation.

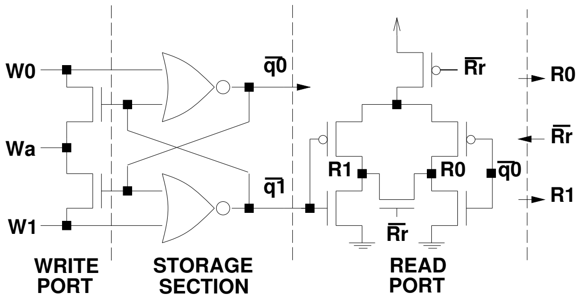



Figure 13: Tangram Dual-Rail Latch.

An existing handshake latch that stores 1 bit of data [25] is shown in Figure 13. $W_0$ and $W_1$ correspond to the dual-rail write data, and $W_a$ is the write acknowledge signal. $R_r$ is the read request, and $R_0$ and $R_1$ are the dual-rail data outputs. When the latch is inactive all handshake wires are low. A read operation is started by the asserting the read request ($R_r\uparrow$). The latch responds with $R_0\uparrow$ —to indicate that a "0" is stored—, or $R_1\uparrow$ —to indicate a "1"—. To complete the read, $R_r$ is de-asserted, followed by $R_0\downarrow$ or $R_1\downarrow$. In a similar fashion, a write operation is started by asserting $W_0$ or $W_1$ as a write request and, also, as an indication of the value to be stored. The new value is stored in the *crossed-coupled NOR latch*. $W_a\uparrow$ acknowledges that the data is stored. Note that, if the new data is the same as the data stored in the latch, no signals need to propagate inside the latch and the write operation will be acknowledged immediately. The return-to-zero phase starts with $W_0$ or $W_1$ going low and is completed when $W_a$ is de-asserted. *Concurrent read and write operations to this latch are not allowed!* If this occurs, the latch may malfunction, as indicated below.

## 5.2 Data Hazards in Overlapped Operation

Unsafe overlapped operation of the datapath is caused by concurrent processing and return-to-zero phases accessing the *same* latch. Figure 14 shows a datapath structure where 2 processes access the same latch ($X$). Of four possible forms of interaction, three are free of data hazards. (i) *Read after Read* (RAR): data does not change and remains stable. (ii) *Read after Write* (RAW): The second computation reads data that has already been written to the latch and is stable. (iii) *Write after Write* (WAW): no read operation is performed so data can be written without causing problems (actually, the use of multiplexers, covered in section 7, prevents the occurrence of this type of interaction).

$$Z = F(X);$$
$$X = G(W);$$

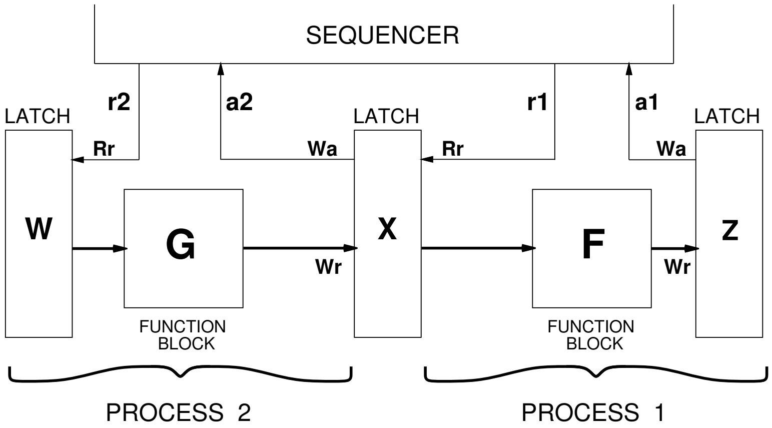



Figure 14: Scenario for a WAR Data Hazard.

The only hazardous interaction, shown in Figure 14, is a *Write after Read* (WAR) hazard: The second computation may write new data to the latch while its read operation has not completed. The new data can propagate through the latch and cause incorrect operation.

A WAR hazard occurs when a read is first initiated ($R_r\uparrow$), making the read port transparent ($R_0\uparrow$ or $R_1\uparrow$). Before the read operation is completed ($R_r\downarrow$; $R_0\downarrow$ or $R_1\downarrow$), a write is initiated ($W_0\uparrow$ or $W_1\uparrow$). The new data in the write port can propagate through the latch to the read port and cause undesired changes in the output data.

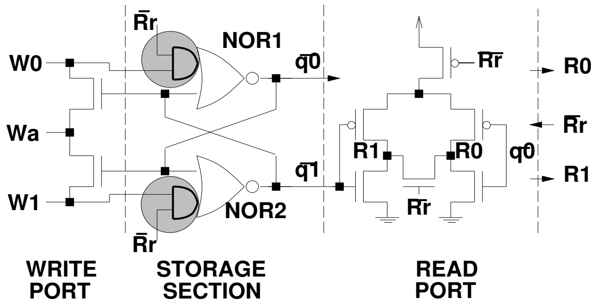
A classical example where this hazard arises is in a shift-register. The shift register is a special case of the datapath shown in Figure 14, in which function blocks $F$ and $G$ correspond to simple wires (the identity function). In an n-stage shift register, the $R_0$ ($R_1$) output of each stage is connected to the $W_0$ ($W_1$) input of the next stage. A read request to one stage therefore produces a write to the adjacent stage (see [25] for details). A sequencer controls each 1-bit shift operation. The sequencer generates a read request ($R_r$) to each stage in turn, and receives the adjacent write acknowledge ($W_a$). If a concurrent sequencer is used, a WAR hazard will occur in every latch.

### 5.3 Solutions for Data-Hazard-free Overlapped Operation

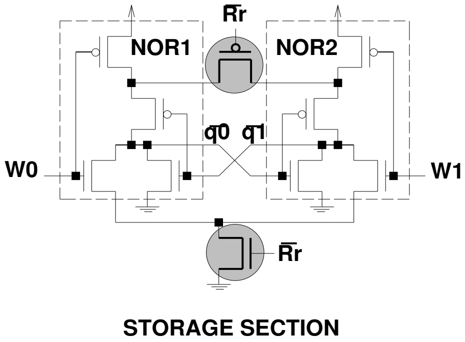
We propose two different approaches to eliminate WAR data hazards: ($i$) a *hardware solution* – *stall* the write operation until the read is completed, and ($ii$) a *compiler solution* – *avoid overlapped accesses* to the same register.

• **Hardware Solution.** Interlock circuitry is used to stall the write operation until the read port of the latch is opaque. The addition of the AND gates, shown in Figure 15(a), makes $\overline{R_r}$ an enable signal for the write data. Signal $\overline{R_r}$ remains low while the read port is transparent, disabling write operations. Note that, in practice, the interlock circuitry should have minimal impact on performance for two reasons. ($i$) The interlock mechanism will rarely be activated because $r_1\downarrow$ propagates through a wire to produce $R_r\downarrow$, while $r_2\uparrow$ must propagate through $W$ and $G$ to generate a write request ($W_0\uparrow$ or $W_1\uparrow$). ($ii$) In the event that the interlocked is engaged, it will be active only for the duration of the race and not for the entire phase.

If the data being written to the latch is equal to the data already stored in it, the write operation is *not* stalled and is acknowledged *immediately*, regardless of the state of the read port. This is a safe optimization: No changes are caused in the latch and no glitches are generated or propagated. Two versions of our modified latches are shown in Figure 15. Figures 15(a) and 15(b) highlight the gate-level and transistor-level changes, respectively.




(a)




(b)

Figure 15: Modified Dual-Rail Latch Implementations: (a) Gate-Level, (b) Transistor-level.

The latter solution requires only 2 added transistors.

• **Compiler Solution.** At the algorithm level, a compiler can easily identify a WAR hazard between two *consecutive* computations. Therefore, the compiler can insert an unrelated operation between them, to eliminate the hazard. In a case in which such reordering is not possible, the compiler either inserts a special, *null* operation or falls back on the use of the modified latch. This technique requires the use of our tightly-coupled burst-mode sequencer, which allows WAR interactions only between two consecutive computations.

# 6 Modified Single-rail Datapaths

Dual-rail datapaths are very robust but pay a large penalty in terms of area and power dissipation. We now examine single-rail datapaths as an alternative implementation.

## 6.1 Basic Single-Rail Operation

Figure 16 shows schematically a sequencer controlling a 2-stage single-rail datapath that implements $Z = F(Y); X = G(W)$. Unlike dual-rail logic, $F$ and $G$ need not be hazard-free: combinational logic blocks that operate in *synchronous* systems may be used.

To comply with the bundling constraint, *i.e.,* guarantee that all data wires are valid and stable before the data-valid signal is asserted, delays are inserted in the data-valid wires. Typically, these delays are designed to match the worst case delay of the corresponding

$$Z = F(Y);$$
$$X = G(W);$$

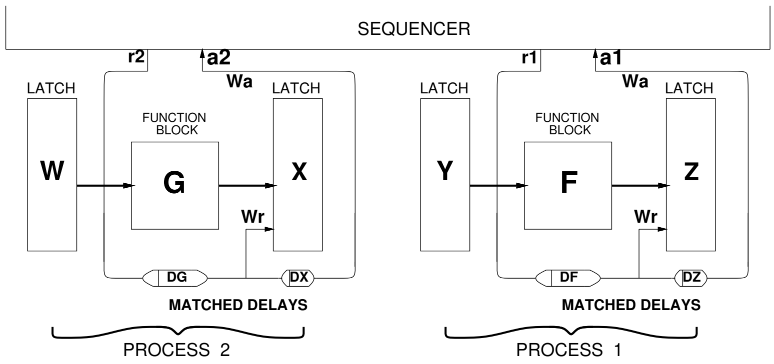



Figure 16: Sequencer controlling a Single-Rail Datapath.

datapath block; that is, $DF$ must equal the worst case delay in the combinational logic block that implements function $F$, and $DZ$ must be equal to the worst case delay in latch $Z$. In CMOS implementations, delays depend heavily on the sizes of transistors and their loading, and also on the final routing and placement of modules, so safety margins are required for correct operation.

Figure 17 shows a latch used to implement the variables which appear in Figure 16 (see [18]). Each latch is normally opaque, and stored data is always readable at its output. A write request ($W_r\uparrow$) makes the latch transparent for writing; the subsequent $W_r\downarrow$ makes the latch opaque, latching the result. Complementary signals $en$ and $ne$ are generated by the latch enable circuit.

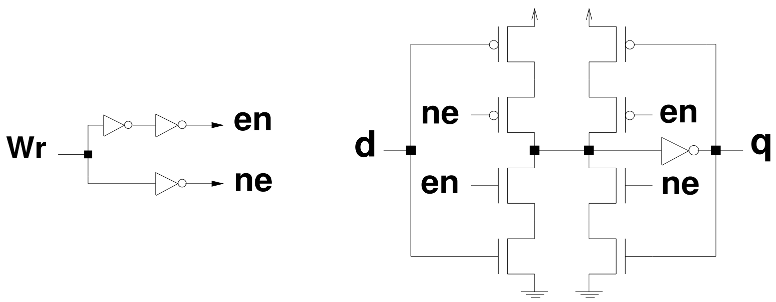



Figure 17: Tangram Single-Rail Latch and Enable Circuit.

The operation of the datapath depends on the type of sequencer used and on how the delay-matching is done. We now examine two schemes for single-rail datapath operation recently presented by Peeters *et al.* [18, 17], then introduce our new concurrent approach.

### 6.1.1 Previous Approaches

• **Conservative Scheme.** The conservative scheme uses a *sequential* controller, such as the Josephs/Bailey counter-decoder sequencer, with the single-rail latch (Figure 17). The delays in the control signals are designed to match worst case delays in the associated blocks; that is, $DF$ must equal the worst case delay in $F$, and $DZ$ must match the response time of latch $Z$.

Datapath sections operate as follows. The sequencer generates an initial request $r_1$. Data from $Y$ is already present as inputs to $F$. The request signal propagates through the matched delay$DF$ as $F(Y)$ is being computed. When computation is complete the data is stable and valid. The output of the delay acts as the data-valid signal for the result. This data bundle is sent to $Z$. The arrival of the data-valid signal makes $Z$ transparent and, after propagating through $DZ$, it is sent back to the controller as acknowledge signal $r_1$. At this point, the processing phase is complete. The sequencer starts the return-to-zero phase by de-asserting $r_1$, which propagates through $DF$ and arrives at $Z$, making it opaque again. After propagation through $DZ$, $a_1$ is de-asserted.

In this scheme, the result of the computation is valid at the end of the processing phase. Once processing is complete, the destination latch ($Z$) becomes transparent (see Figure 16). The key point in this scheme, shown in Figure 18(a), is that the latch is transparent *only* when data is valid and stable, so no undesired glitches are propagated to the rest of the circuit. The drawback is that performance is poor, since the sequencer does not allow overlapped operation. In particular, even though the result of the computation is ready at the end of the processing phase, the stage still must go through the return-to-zero phase before the next computation can begin.

• **Fast Scheme.** A scheme that, using a sequential controller, can achieve higher throughput by a novel distribution of the computation throughout the phases of the handshake protocol. In this scheme, called *standard true four-phase protocol* by Peeters [17], delays are designed to match only *half* the value of the worst-case delay in the functional blocks. As in the previous scheme, $r_1$ propagates through $DF$ and becomes the data-valid signal for the output data from $F$. The difference is that, at the time this signal is asserted,

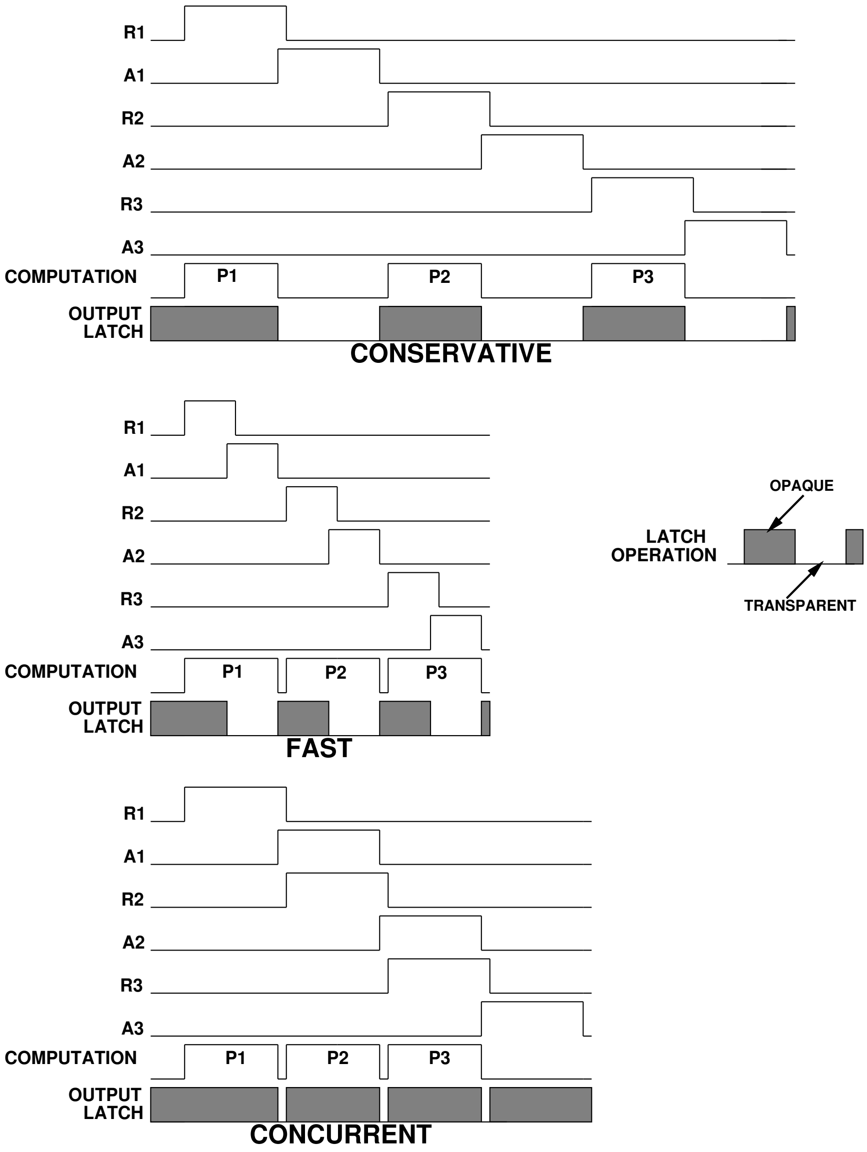



Figure 18: Single-rail Operation Schemes: (a) Conservative, (b)Fast, (c) Concurrent.

only half of the computation time has elapsed, and data is not ready! The signal arrives as a write request to $Z$, making it transparent. The latch acknowledge signal goes to the controller as an indication of a completed processing phase even though computation is still going on. $r_1\downarrow$ starts the return-to-zero phase and propagates through the matched delay. At this point the result of the computation is stable and available in the data wires that feed the latch. When the control signal reaches $Z$, the latch is closed.

Figure 18(b) shows the operation of the fast scheme. In this case, $a_i\downarrow$ indicates that computation is complete, as opposed to $a_i\uparrow$ in the previous scheme. The advantage of this scheme is that it reduces to *a half* the length of the processing and return-to-zero phases of the handshake, obtaining roughly twice the throughput of the conservative scheme. However, the scheme has two key drawbacks: ($i$) the matching of delays to half the value of the delay in the functional blocks is not straightforward, and, more significantly, ($ii$) the destination latch is made transparent while data is unstable. In fact, the outputs of the combinational circuit $F$ can glitch many times during this period and these glitches will be propagated to every processing stage connected to the latch (see discussion in [17]). This results in unpredictable power consumption that can be large, especially if the latch is connected to deep combinational circuits.[4]

## 6.2 A New Approach: Overlapped Operation

Our solution is to use one of our *concurrent sequencers* with the conservative datapath scheme, where the matched delays match the full computation block worst-case delays.

The operation of this scheme is shown in Figure 18(c). The concurrent sequencer initiates active phases of adjacent processes in immediate succession, allowing high throughput (see Figures 3 and 8). At the same time, each process uses the simple delay-matching approach of the conservative scheme. This results in essentially the same performance advantage as the fast scheme but without the drawbacks: a latch is transparent *only* when

[4] Peeters [17] suggests a *low-power true four-phase protocol* to reduce the glitch-propagation problem. Essentially, read ports are added to the latches to block the glitches. However, Peeters also points out that, although the low-power variant minimizes the number of transitions, it is likely that the added power consumption in the latches will eliminate any power savings due to the reduced transitions.

data is stable, eliminating glitch propagation. In addition, the delays are matched to the worst-case value of the associated functional block.

This scheme is a valid solution, except for one problem. As in dual-rail datapaths, overlapped operation introduces the possibility of data hazards if operations interact with the same latch. An analysis of the operation of the single-rail datapath (equivalent to the analysis of the dual-rail datapath in the previous section) reveals that three type of interaction are safe (RAR, RAW, and WAW); only WAR interactions are unsafe and require modifications.

Unsafe overlapped operation of the datapath is caused by concurrent processing and return-to-zero phases that access the same latch. For example, in Figure 16, assume that latch $Y$ is the same as latch $X$. The sequencer will start the return-to-zero phase of $Process$ 1, concurrently with the processing phase of $Process$ 2. There is a race: $Process$ 1 is making $Z$ opaque ($r_1\downarrow$) while $Process$ 2 is making $X$ transparent ($r_2\uparrow$). If $Process$ 2 wins the race, the new data in $X$ may reach $Z$ before it becomes opaque, causing it to store the wrong data.

## 6.3 Modifications for Correct Overlapped Operation

The WAR hazard arises because the destination latch $Z$ remains transparent throughout the return-to-zero phase while the overlapped processing phase can write to the source latch ($X$). Latch $Z$ already stored the information and is only waiting for $r_1\downarrow$ to propagate through the matched delays as a close signal. We propose two different solutions:

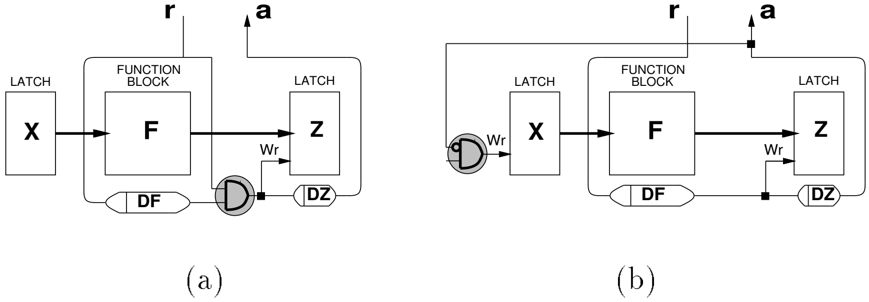



Figure 19: Latch Enable Circuit: (a) Early-Close Modification, (b) Interlock Modification.

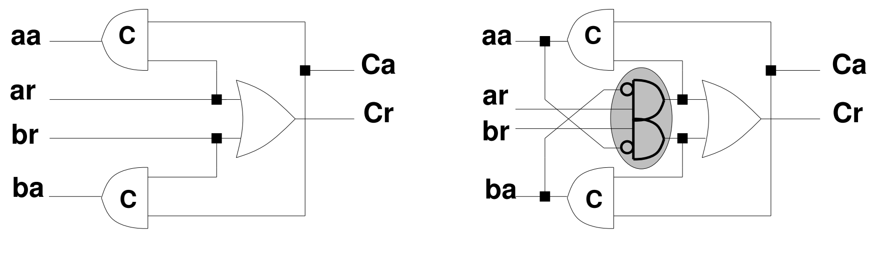
• **Early close scheme.** We can *fast-forward* $r_1\downarrow$ to the *destination latch* so it closes early in the return-to-zero phase instead of at the end. Figure 19(a) shows this simple modification to the latch enable circuit. The latch will not open early so no glitch propagation will occur. This scheme relies on reasonable timing assumptions for correct operation.

• **Interlock scheme.** A more robust approach is to stall the writing of the *source latch* until $Z$ is opaque again. In this case, the acknowledge signal from the destination latch ($a$) is used as an enable to the source latch write request. Figure 19(b) shows this modification. Stalling the write operation guarantees correct operation, independently of the delays in the circuit, but provides less performance improvement than the previous scheme.

# 7 Overlapped Multiplexers

The operation of a datapath, either dual-rail or single-rail, often requires *multiple* accesses to the *same* process. In this case, the different requests must be multiplexed together. An existing handshake multiplexer for control signals [25], shown in Figure 20(a), requires *mutually-exclusive requests* on its two channels. A new control multiplexer design, shown in Figure 20(b), allows overlapped requests. In this design, a second request is stalled at the AND gate until the first operation is completed.

Control multiplexers have been extended to data multiplexing [17]. Similar extensions can be made to our design of Figure 20(b).




(a) (b)

Figure 20: Multiplexers: (a) Existing, (b) Modified.

# 8 Results

We now show how the increased throughput obtained by the new sequencers combines effectively with the application of voltage scaling to produce significant energy savings of an entire asynchronous system. We compare important features of all the sequencers discussed above and then show results of several SPICE simulations.

| SEQUENCER | AREA # transistors | ENERGY # gate output transitions | TIMING MODEL |
|---|---|---|---|
| *Previous Designs* | | | |
| Tangram | 18N–18 | 10N–10 | SI |
| Martin | 18N–18 | 10N–10 | SI |
| Josephs/Bailey Counter-Decoder | 15N–6 | 7N–2 | SI |
| Josephs/Bailey Chain | 12N+4 | 8N–2 | FM |
| Unger Tree | 36N–36 | 16N–16 | FM |
| Farnsworth (Extended N-way)† | 14N–14 | 8N–8 | FM |
| *New Designs* | | | |
| Burst-mode | 14N–6 | 8N–4 | FM |
| Optimized | 10N+2 | 6N | FM |
| Speed-independent | 12N+2 | 6N+6 | SI |

†In [5], Farnsworth et. al used a concurrent 2-way sequencer. No N-way extensions were presented. We explored three different structures to obtain an N-way sequencer: a left-branching tree, a right-branching tree, and a balanced tree. All extensions lead to the same results.

Table 1: Static Characteristics of N-way Sequencers.

Sequencer characteristics are summarized in two tables. Table 1 compares static characteristics for the different sequencers. The information is given as a function of $N$, the number of processing stages being sequenced. The total number of transistors and gate-output transitions are used as first order approximations to area and power consumption. The results show that the new designs are very competitive in both dimensions. In fact, the optimized and speed-independent sequencers have better features than all the others.

Table 2 compares dynamic behavior of the different sequencers, where each sequencer is assumed to control $N$ identical processes. $g$ is roughly the delay associated with a

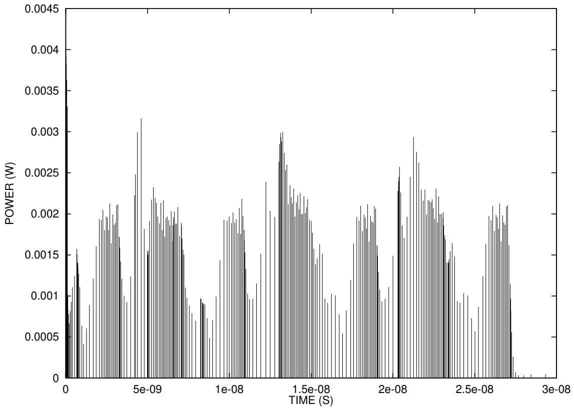
| SEQUENCER | INITIAL LATENCY | INTER-PROCESS LATENCY | TOTAL COMPUTATION TIME |
|---|---|---|---|
| *Previous Designs* | | | |
| Tangram | (2N-2)g | 5g + R | (6N-8)g+NP+(N-1)R |
| Martin | (2N-2)g | 5g + R | (6N-8)g+NP+(N-1)R |
| Josephs/Bailey Counter-Decoder | 2g | 4g + R | (4N-4)g+NP+(N-1)R |
| Josephs/Bailey Chain | 2g | 3g + R | (3N-2)g+NP+(N-1)R |
| Unger Tree | 2(logN)g | MIN=2g<br>MAX=[4(logN)-2]g | (6N-6)g+NP |
| Farnsworth (Extended N-way)† | (2N-2)g | 2g | (4N-4)g+NP |
| *New Designs* | | | |
| Burst-mode | 2g | 2g‡ | (2N+2)g+NP |
| Optimized | 2g | 2g | 2Ng+NP |
| Speed-Independent | 2g | 2g | 2Ng+NP |

†In [5], Farnsworth et. al used a concurrent 2-way sequencer. No N-way extensions were presented. We explored three different structures to obtain an N-way sequencer: a left-branching tree, a right-branching tree, and a balanced tree. The results shown here correspond to our left-branching-tree extension. All extensions lead to the same total computation time.

‡If a return-to-zero phase is unusually long, the inter-process latency may increase due to synchronization dependencies.

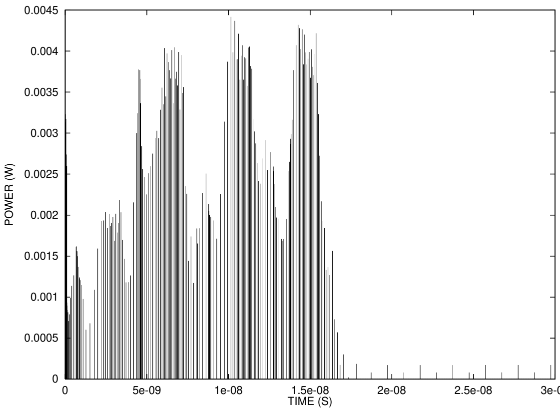
Table 2: Dynamic Behavior of N-way Sequencers.

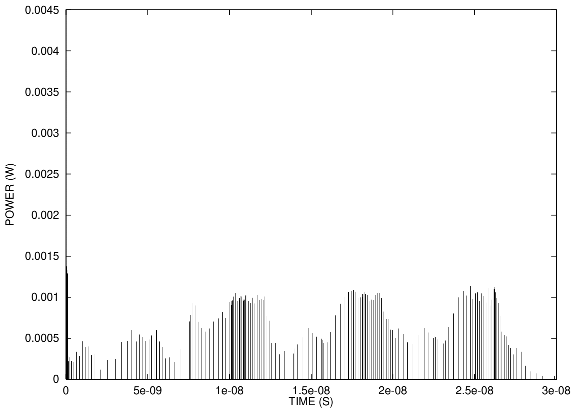
CMOS complex gate or an inverter, $P$ represents the length of a processing phase, and $R$ is the length of the return-to-zero phase. Again, the table shows that the new designs are very competitive. The substantial improvement in the computation time is due to the concurrent operation of the new sequencers, which eliminates the $(N-1)R$ term, and to the efficient implementation that reduces the number of gate delays, both for initial and inter-process latencies, resulting in the best total computation time.




**TANGRAM DESIGN**

Josephs/Bailey chain sequencer + Tangram dual-rail latches




**OUR DESIGN**

**No Voltage Scaling** **With Voltage Scaling**

Optimized concurrent sequencer + modified dual-rail latches

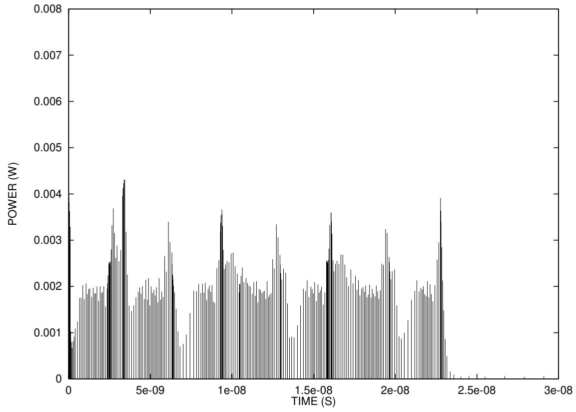
Figure 21: Simulated Power Consumption for 4-stage Dual-Rail System.

Finally, to analyze the combined impact on system power consumption of the new sequencers and the application of voltage scaling, we have simulated results using SPICE [12]. We present results on both dual-rail and single-rail sequential systems. In particular, we simulated several versions of a sequencer controlling a 4-stage datapath with identical processes.

Simulation results for a dual-rail system are shown in Figure 21. Figure 21(a) shows the power consumption of a sequential implementation of the system, using a Josephs/Bailey chain sequencer for control and Tangram dual-rail latches, with a 5 volt power supply. In comparison, Figure 21(b) shows the power consumption of a new concurrent implementation of the system, using our optimized sequencer and modified dual-rail latches, operating at 5 volts. Our design obtains an 73% improvement in total computation time, compared to the previous design. Finally, Figure 21(c) shows the result of the application of voltage scaling to our system: the power supply can be dropped to 3.3 volts with the total energy consumption of the entire system reduced *by a factor of 2.5* compared to the sequential design of Figure 21(a).

Figure 22 shows the simulation results for the single-rail system. Figure 22(a) shows a sequential implementation of the conservative scheme, using a Josephs/Bailey chain sequencer for control and Tangram single-rail latches, with a 5 volt power supply. Figure 22(b) shows our optimized design using the early close scheme, also operating at 5 volts. In this case, our design obtains a 66% improvement in total computation time. Finally, Figure 22(c) presents the simulation of our optimized design after voltage scaling is applied (power supply reduced to 3.3 volts). The total energy consumption of the entire system is reduced *by a factor of 2.4*.

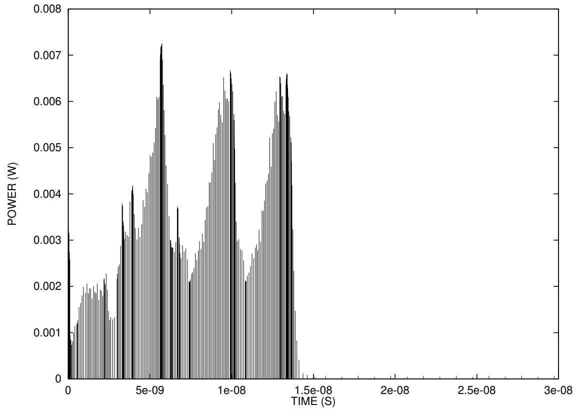
The fast scheme was also simulated. As expected, we obtained roughly the same total computation time as with our optimized design. However, as mentioned earlier, our simpler approach has two key advantages: (*i*) ease of design (delay matching to the entire computation block, rather than to half of the block) and (*ii*) glitch avoidance in the datapath.

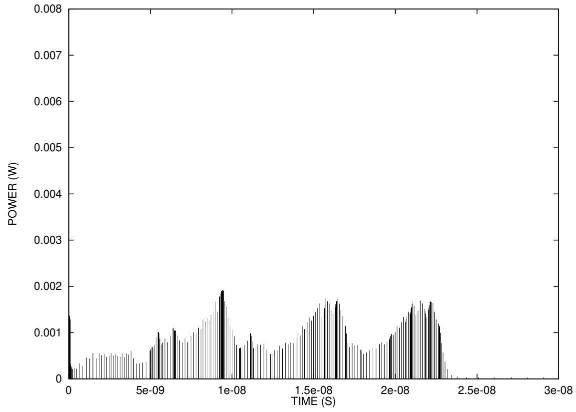



**TANGRAM DESIGN**

Conservative Scheme

Josephs/Bailey chain sequencer + Tangram single-rail latches




**OUR DESIGN**

**No Voltage Scaling** **With Voltage Scaling**

Optimized concurrent sequencer + Tangram single-rail latches with early-close scheme

Figure 22: Simulated Power Consumption for 4-stage Single-Rail System.

# 9 Conclusions

This paper has focused on architectural optimizations for low-power asynchronous systems. These optimizations were targeted to sequential, *i.e.,* non-pipelined, computation. We presented three new sequencer designs that effectively increase the concurrent activity of the system. We also showed that existing datapaths will not work correctly at the increased level of concurrency. New latch and multiplexer designs, that safely accommodate the added concurrency, were presented for both dual-rail and single-rail implementations. In the dual-rail case, our optimizations resulted in improved throughput, providing the opportunity for significant power savings through voltage scaling. Similar improvements were demonstrated over one existing single-rail scheme. We also indicated benefits of our single-rail approach over a second single-rail scheme.


# Acknowledgments

The authors would like to thank Prof. Stephen Unger (Columbia University) for critically reviewing a draft version of this manuscript and making useful suggestions. They would also like to thank Dr. Craig Farnsworth (Cogency Ltd.) for introducing them to his work, and Prof. Stephen Furber (University of Manchester) and Dr. Ad Peeters (Philips Research Laboratories) for their useful comments.


# References


[1] A. Bailey and M. Josephs. Sequencer circuits for VLSI programming. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 82–90. IEEE Computer Society Press, May 1995.

[2] E. Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[3] A. P. Chandrakasan, M. Potkonjak, R. Mehra, J. Rabaey, and R. W. Brodersen. Optimizing power using transformations. *IEEE Trans. on Computer-Aided Design*, 14(1):12–31, January 1995.

[4] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low-power CMOS digital design. *IEEE J. of Solid-State Circuits*, 27(4):473–484, April 1992.

[5] C. Farnsworth, D. A. Edwards, J. Liu, and S. S. Sikand. A hybrid asynchronous system design environment. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 91–98. IEEE Computer Society Press, May 1995.

[6] S. Furber. Computing without clocks: Micropipelining the ARM processor. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, pages 211–262. Springer-Verlag, 1995.

[7] S. B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems*, 4(2):247–253, June 1996.

[8] M. B. Josephs and A. M. Bailey. Design of sequencer circuits: a case study in SI-algebra. Technical Report SBU-CISM-94-12, South Bank University, September 1995.

[9] A. Marshall, B. Coates, and P. Siegel. Designing an asynchronous communications chip. *IEEE Design & Test of Computers*, 11(2):8–21, Summer 1994.

[10] A. J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C.A.R. Hoare, editor, *Developments in Concurrency and Communication*, UT Year of Programming Series, pages 1–64, Addison-Wesley, Reading MA, 1990.

[11] R. E. Miller. *Sequential Circuits and Machines*, volume 2 of *Switching Theory.* John Wiley & Sons, New York, 1965.

[12] L. W. Nagel and D. O. Pederson. Simulation program with integrated circuit emphasis (SPICE). Technical Report ERL-M383, Electronics Research Lab., University of California, Berkeley, April 1983.

[13] L. S. Nielsen, C. Niessen, J. Sparsø, and K. van Berkel. Low-power operation using self-timed circuits and adaptive scaling of the supply voltage. *IEEE Transactions on VLSI Systems*, 2(4):391–397, December 1994.

[14] L. S. Nielsen and J. Sparsø. A low-power asynchronous data-path for a FIR filter bank. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 197–207. IEEE Computer Society Press, March 1996.

[15] S. M. Nowick and B. Coates. UCLOCK: Automated design of high-performance asychronous state machines. In *Proc. International Conf. Computer Design (ICCD)*, pages 434–441. IEEE Computer Society Press, October 1994.

[16] S. M. Nowick and D. L. Dill. Automatic synthesis of locally-clocked asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 318–321. IEEE Computer Society Press, November 1991.

[17] A. Peeters. *Single-Rail Handshake Circuits.* PhD thesis, Eindhoven University of Technology, June 1996.

[18] A. Peeters and K. van Berkel. Single-rail handshake circuits. In *Proc. Working Conf. on Asynchronous Design Methodologies*, pages 53–62, May 1995.

[19] L. A. Plana and S. M. Nowick. Concurrency-oriented optimization for low-power asynchronous systems. In *Proc. International Symposium on Low Power Electronics and Design*, pages 151–156, August 1996.

[20] C. L. Seitz. System timing. In C. A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, chapter 7. Addison-Wesley, Reading, MA, 1980.

[21] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[22] S. H. Unger. A building block approach to unclocked systems. In *Proc. Hawaii International Conf. System Sciences*, volume I, pages 339–348. IEEE Computer Society Press, January 1993.

[23] Stephen H. Unger. *Asynchronous Sequential Switching Circuits*. Wiley-Interscience, New York, 1969.

[24] K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. Asynchronous circuits for low power: A DCC error corrector. *IEEE Design & Test of Computers*, 11(2):22–32, Summer 1994.

[25] K. van Berkel and M. Rem. VLSI programming of asynchronous circuits for low power. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, pages 152–210. Springer-Verlag, 1995.

[26] Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[27] K. Y. Yun, P. A. Beerel, and J. Arceo. High-performance asynchronous pipeline circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 17–28. IEEE Computer Society Press, March 1996.

[28] K. Y. Yun and D. L. Dill. Automatic synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)*, pages 576–580. IEEE Computer Society Press, November 1992.