Concurrency-Oriented Optimization for Low-Power Asynchronous Systems

Luis A. Plana[†] Steven M. Nowick[‡]
Department of Computer Science
Columbia University
New York, NY 10027

Abstract—We introduce new architectural optimizations for low-power asynchronous systems, such as Tangram-based systems of van Berkel *et al.* Our goal is to reduce power consumption by improving system concurrency. We introduce two new sequencer designs, with greater concurrency than existing ones, that provide the opportunity for substantial power savings through voltage scaling. To safely accommodate this added concurrency, new latch designs are presented, for both dual-rail and single-rail implementations.

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.

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 [3, 7]. Chandrakasan *et al.* [3] 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* [3, 7, 13].

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 design is a major focus of recent asynchronous design, including a low-power infrared communications chip [5], an asynchronous implementation of the ARM microprocessor [4], and an asynchronous error corrector for a DCC player [13].

A number of asynchronous design methods have been introduced recently. Several methods approach circuit design as a programming activity. For example, van Berkel *et al.* [13, 14], 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 communicating processes. Every process is mapped to a circuit element in a self-timed library of 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.

The goal of this paper is to present architectural-level optimizations for low-power asynchronous macromodular systems, such as those of van Berkel [13, 14]. In these systems, sequencing control and its interaction with the datapath are critical. Our goal is to *increase the level of concurrency* in the sequencing of data processing actions. This increased concurrency must be achieved *without* increasing the switching activity required for the computation (otherwise power consumption could increase).

In particular, we present the following new contributions. First, we introduce two 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 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, energy is reduced to less than one-half. For single-rail datapaths, two different schemes are referenced. 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. In section 3, existing sequencers are examined, and two new concurrent sequencer designs are introduced. Section 4 introduces new latch and multiplexer designs, for *dual-rail* datapaths, to handle the increased concurrency. Similar modifications for single-rail datapaths are introduced in section 5. Section 6 presents results of analysis and SPICE runs, and Section 7 presents conclusions.

2 Overview

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 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 determined by technology factors. In most CMOS circuits, switching power dominates the other two.

Two factors affect switching power consumption: (*i*) the

[†]Supported by a grant from CONICIT, Venezuela.
[‡]Supported by an NSF CAREER Award MIP-9501880 and by an Alfred P. Sloan Research Fellowship.

ISLPED 1996 Monterey CA USA
0-7803-3571-8/96/$5.00 © 1996

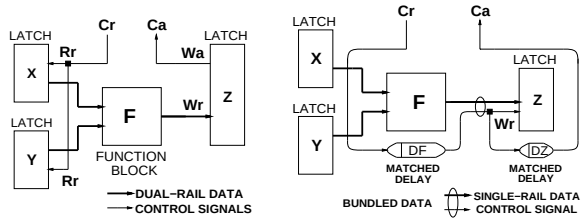
amount of switching activity that takes place, i.e., the number of transitions; and (*ii*) the energy consumed per transition, which is a function of the capacitance that is being (dis)charged and the supply voltage. Power consumption can be reduced by reducing the capacitance, the number of transitions, or the supply voltage. Since power depends quadratically on the supply voltage, *supply voltage scaling* is an especially attractive scheme for power reduction [3]. Unfortunately, voltage scaling has the undesirable effect of reducing the speed of the circuit. Our goal is therefore to *increase the concurrency*, and hence the throughput, of a system. Such throughput improvement compensates for the performance penalty which results from supply voltage reduction. 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 circuits are designed as a network of predefined data and control modules [2]. Instead of a global clock signal, communication channels between modules are used to synchronize their operation and data interchange. These channels can be implemented using different protocols and different codes can be used to represent and transmit data. Two protocols are most common: *dual-rail* and *single-rail*.

• **Dual-rail Data Processing.** In dual-rail datapaths, data is encoded using a *dual-rail code* [14, 6], a *delay-insensitive* code that requires two wires for every data bit. Codes 01 and 10 represent '1' and '0' data values, respectively; code 00 represents the *idle* state; and 11 is an invalid code. 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. Figure 1(a) shows schematically how a typical data processing action, $Z = F(X, Y)$, is implemented using handshake circuits. X, Y, and Z are variables that hold data, and F is usually a block of combinational logic that implements the desired function.

A controller, frequently a sequencer, uses handshake signals C_r and C_a to communicate with this datapath section, using a *4-phase handshake protocol* (see below). Function F is implemented using *hazard-free* combinational logic that operates on dual-rail input data and generates dual-rail outputs. Hazard-free operation is required because any glitch in the data wires can be interpreted as a valid data signal and produce erroneous operation.



(a) Dual-rail operation (b) Single-rail operation

Figure 1: Schematic Datapaths for $Z = F(X, Y)$.

• **Single-rail Data Processing.** *Single-rail encoding* [11, 2, 9] can be used to overcome the large area and power penalty associated with the use of dual-rail data. This code uses one wire for every data bit and one additional wire, called the *data-valid signal*, for control. The collection of all data bits and the data-valid signal is called a *data bundle*.

The correct operation of single-rail circuits relies on a local timing assumption: all data wires must be valid and stable at the inputs of a module before the data-valid signal is asserted. This timing assumption is called a *bundling constraint*. To comply with this constraint, delays are inserted in the data-valid wires. In CMOS implementations, delays depend heavily on the final routing and placement of modules, so safety margins are required for correct operation.

Figure 1(b) shows a schematic handshake circuit for a single-rail implementation of $Z = F(X, Y)$. Unlike dual-rail logic, F need not be hazard-free: a *synchronous* combinational logic block may be used. The delays in the control signals are designed to match worst case delays. DF must equal the worst case delay in F, and DZ must match Z.

3 Control of Computation

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 [13] consists of 48 actions.

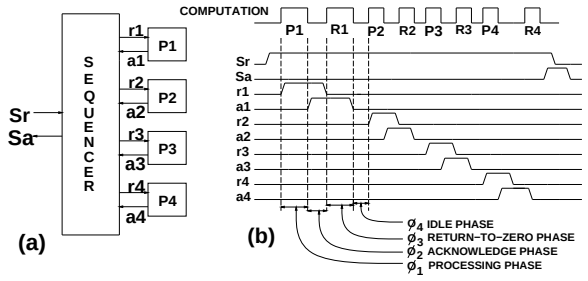


Figure 2: Sequencer Circuit and Timing Diagram.

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 a *request* (r_i) and *acknowledge* signal (a_i). Figure 2(b) shows how the sequencer communicates with each process using a basic *4-phase handshake protocol* [6]. The protocol phases are:

- *Processing phase* (ϕ_1): the sequencer asserts r_i to request processing to take place.
- *Acknowledge phase* (ϕ_2): P_i asserts a_i to signal the completion of the computation.
- *Return-to-zero phase* (ϕ_3): the sequencer de-asserts r_i to request P_i to clear its acknowledge signal.
- *Idle phase* (ϕ_4): P_i de-asserts a_i to signal that it is idle and the sequencer can start the next process.

In this basic *sequential protocol*, the processing and return-to-zero phases of computations are sequenced: $P1$; $R1$; $P2$; $R2$; etc. Useful computation takes place *only* during the *processing phase* (ϕ_1). The rest of the phases, ϕ_2, ϕ_3, and ϕ_4, represent *dead time* from the point of view of computation. ϕ_2 and ϕ_4 correspond to *sequencer latency* and ϕ_3 corresponds to the *return-to-zero phase* of the process.

Figure 2(b) shows schematically how processing (P_i) and return-to-zero (R_i) phases alternate in a *sequential protocol*, resulting in a long dead time between computations. The behavior of this sequencer is described by the expression:

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

Optimization techniques for such sequencers typically focus only on reducing the latency of phases ϕ_2 and ϕ_4. Our goal is to provide significant reductions in dead time by introducing *concurrent operation*. The sequencer can start process P_i as soon as P_{i-1} has finished processing. In this way, every processing phase P_i is *overlapped* with the return-to-zero phase R_{i-1} of the previous process.

3.1 Previous Sequencers

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

3.1.1 Sequential Approaches

• **Tangram Sequencer.** In Tangram, 2-way sequencing is implemented using the *SEQ* operator, as shown in Figure 3(a) [14]. 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).

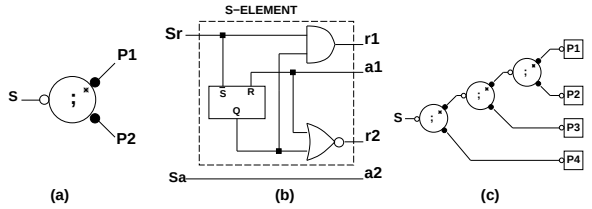


Figure 3: Tangram Sequencing Elements.

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 $P2$:

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

An implementation of the SEQ operator is shown shown in Figure 3(b). This circuit is *speed-independent*, 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 3(c). There are two problems with the Tangram sequencer: (*i*) it has a long *initial latency* (the time it takes the start signal to reach the first process), and (*ii*) it has a long ϕ_2 latency, equivalent to several gate delays.

• **Martin Sequencers.** In [6], Martin presents two n-way sequencers. The Tangram n-way sequencer corresponds exactly to a *Q-element-based* Martin sequencer and it has similar performance problems. A *D-element based* sequencer provides no overall performance improvement.

• **Counter/Decoder Sequencer.** In [1], Bailey 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. The implementation has improved initial and ϕ_2 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.

• **Bailey Chain Sequencer.** Bailey [1] also introduced a distributed sequencer built as a *linear chain* of n modules, each controlling a process. The modules assume *fundamental-mode* operation. In fundamental mode, no new inputs can arrive until the component has stabilized from a previous input change. The long latencies present in the Tangram circuit do not occur in this design, resulting in a more efficient sequencer.

3.1.2 Concurrent Approaches

A concurrent sequencer was introduced by Unger [12]. However, it pays a large penalty in latency, area and energy.

• **Unger Tree Sequencer.** Unger [12] presents 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 can be built as a balanced tree of 2-step modules [12]. There are several problems with this implementation: (*i*) the sequencer has a long *initial latency*, (*ii*) the *inter-process latency* (ϕ_4) 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 6).

3.2 New Concurrent Sequencers

We now introduce 2 new concurrent sequencers. Both designs have good latency, area and power characteristics.

3.2.1 Burst-Mode Concurrent Sequencer

Our first sequencer *tightly controls* the overlap between a processing phase P_i and the previous 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 4.

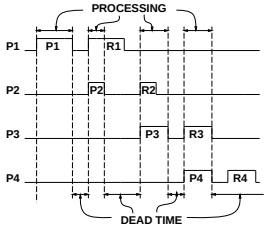


Figure 4: Burst-Mode Sequencer Operation.

We synthesized the circuit using an existing *burst-mode* tool, UCLOCK [8], with extensions to incorporate output feedback. The result is a modular design, well suited for distributed control. Our sequencer has N modules organized into 4 types as shown in Figure 5(a): module M1 controls process P_1, M2 controls P_2, MI modules control P_3 to P_{N-1}, and MN controls P_N.

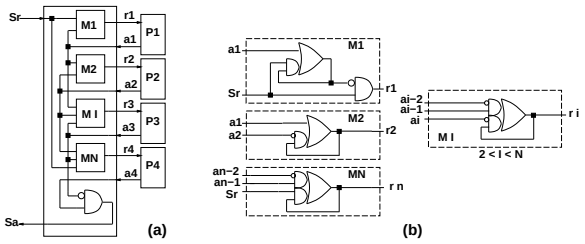


Figure 5: Burst-Mode Sequencer.

The sequencer operates as follows. A request on S_r activates module M1 which starts a 4-phase handshake with process P_1 by $r_1\uparrow$. P_1 then responds with $a_1\uparrow$; modules M1 and M2 both receive this signal. BM1 will respond with $r_1\downarrow$ while, *concurrently*, M2 will start a 4-phase handshaking with with P_2 by $r_2\uparrow$. As a result, the reset phase of the first process ($R1$) overlaps the next computation ($P2$). The sequencer then waits for the completion of *both* phases to proceed: once $a_1\downarrow$ and $a_2\uparrow$ have both arrived, M2 continues the handshaking with P_2 concurrently with starting a 4-phase handshake with P_3. As a result, R_2 overlaps P_3. The same behavior continues until the end of the sequence.

Note that in module MI, shown in Figure 5(b), the critical path from completion of P_{i-1}'s active phase ($a_{i-1}\uparrow$) to the start of P_i's active phase ($r_i\uparrow$) is *only 1 gate delay*, allowing

fast activation of the next process[1]. The other modules, M2-M4, have similar latency.

3.2.2 Optimized Concurrent Sequencer

Our second sequencer allows greater concurrency, by using a more relaxed synchronization requirement.

In our burst-mode sequencer, a long return-to-zero phase, like R_1 in Figure 4(a), may unnecessarily delay the start of the next processing phase (P_3). In this case, P_2 completed early and it is the only requirement to start P_3. By starting P_3 *as soon as* P_2 is finished, *independently* of the status of R_1, a faster sequence of processing phases is allowed.

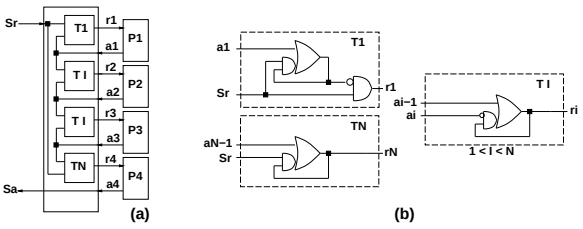


Figure 6: Optimized Sequencer.

Our new sequencer design is shown in Figure 6(a). Although similiar 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 implementation, shown in Figure 6(b), is more efficient in terms of area and power. [2]

4 Dual-rail Datapaths

We now examine the interaction of concurrent sequencers with the actual datapath, and point out problems that can arise. We then present modified latch and multiplexer designs that allow safe overlapped operation in the datapath.

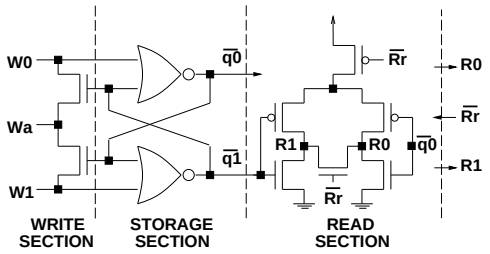


Figure 7: Dual-rail Latch.

A dual-rail variable is usually implemented by a latch with separate read and write ports [14] (see Figure 7). The latch is opaque when inactive. 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. *Concurrent read and write operations to this latch are not allowed!* If this occurs, the latch may malfunction, as indicated below.

4.1 Data Hazards during Overlapped Operation

When concurrent sequencers are used, a new process is started as soon as the computation phase of the previous one is complete. Unsafe overlapped operation of the datapath is caused by concurrent processing and return-to-zero phases accessing the same latch. 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 new computation reads data that has already been written to the latch and is stable. (iii) *Write after Write* (WAW): the read port of the latch is opaque so the new data can be written without causing problems.

The only hazardous interaction is: (iv) *Write after Read* (WAR) hazard: The new computation may write new data while the read port is still transparent. In this case, a read is first initiated (e.g. $R_r\uparrow$; $R_0\uparrow$ or $R_1\uparrow$). Before it is completed ($R_r\downarrow$; $R_0\downarrow$ or $R_1\downarrow$), a write is initiated (e.g. $W_0\uparrow$ or $W_1\uparrow$).

A classical example where this hazard arises is in a dual-rail shift-register (SR). In the framework of Figure 1(a), the shift register is a special case: function block F is deleted, as is latch Y. Latch X is the single source, feeding destination latch Z. This structure is replicated: another latch Q, in turn, is a source (to the left), feeding destination latch X, and so on. 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 [14] for details). A sequencer controls the 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.

4.2 Hazard-free Overlapped Operation

Two different approaches can be used to eliminate WAR data hazards: (*i*) *hardware solution – stall* the write operation until the read is completed, and (*ii*) *complier 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 8(a), makes $\overline{R_r}$ an enable signal for the write data. $\overline{R_r}$ remains low while the read port is transparent, disabling write operations. Note that, in practice, the interlock circuitry should have minimal impact on performance: using a concurrent sequencer, $R_r\downarrow$ will typically arrive before the write request ($W_0\uparrow$ or $W_1\uparrow$) from the next stage, and no stall will occur.

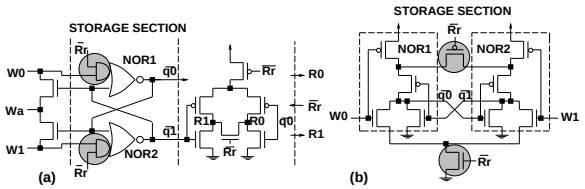


Figure 8: Modified Latches for Overlapped Operation.

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 8. Figures 8(a) and 8(b) highlight the gate-level and transistor-level changes, respectively. 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

[1] In a CMOS VLSI implementation, an AND/OR gate is implemented using an AOI-gate and inverter, resulting in 2 gate delays.

[2] Our sequencers have modest fundamental-mode requirements; we have also developed a more robust "speed-independent" version of this sequencer [10], for use if these timing requirements cannot be met.

tightly-coupled burst-mode sequencer, which allows WAR interactions between two consecutive computations only.

4.3 Overlapped Multiplexers

The operation of the datapath often requires *multiple* modules to write to the *same* latch. Since latches only have one write port, the different write requests must be multiplexed to this port. An existing handshake multiplexer [14], shown in Figure 9(a), requires *mutually-exclusive requests* on its two channels. A new multiplexer design, shown in Figure 9(b), allows overlapped requests. In this design, an overlapped request is stalled at the AND gate until the first operation is completed.

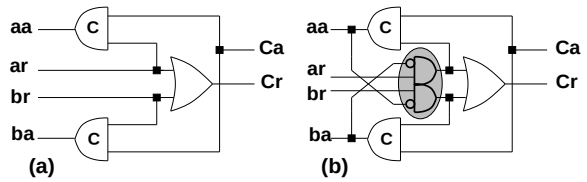


Figure 9: Multiplexers: Original and Modified.

5 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.

Figure 10(a) shows the latches used to implement the variables which appear in Figure 1(b) (see [9]). Complementary signals *en* and *ne* are generated by the latch control circuit, shown in Figure 10(b). 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.

5.1 Previous Approaches

Two recent schemes have been proposed for single-rail datapath operation by Peeters and van Berkel [9].

Conservative Scheme. The conservative scheme uses a *sequential* controller, such as Bailey chain sequencer, with the single-rail latch (Figure 10(a)). Performance is poor, since the sequencer does not allow overlapped operation.

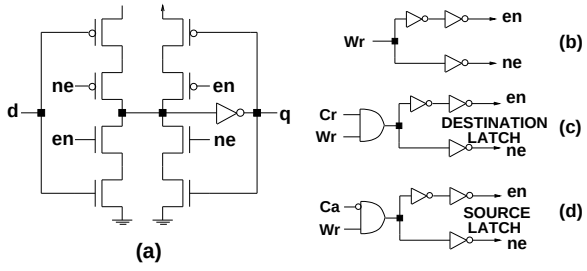


Figure 10: Single-rail Latch and Control Circuits.

In this scheme, the result of the computation is valid at the end of the *processing phase* (ϕ_1). Once processing is complete, the latches becomes transparent (see Figure 1(b)). The key point, in this scheme, is that the result *remains stable* throughout the return-to-zero phase (ϕ_3), allowing the destination latch to remain open with valid and stable data.

A positive aspect of this scheme 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 of this scheme is that, 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 *fast scheme* that achieves a desirable high density of computation by a novel distribution of the computation throughout the phases of the handshake protocol. While the fast scheme can be twice as fast, it has potential problems in terms of datapath power dissipation and ease of realization.

In this scheme, delays are designed to match only *half* the value of the worst-case delay in the functional blocks. As in the previous scheme, C_r 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, 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. $C_r\downarrow$ starts the return-to-zero phase and propagates through the matched delay. At this point the result of the computation is available and stable in the data wires that feed the latch. When the control signal reaches Z, the latch is closed.

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 density of computation 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 when data 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. This results in unpredictable power consumption that can be large, especially if the latch is connected to deep combinational circuits.

5.2 Overlapped Single-rail Operation

Our solution is to use one of our *concurrent sequencers* with the conservative datapath protocol, where the matched delay matches the full computation block. This results in essentially the same performance as the fast scheme but without the drawbacks: the latch is transparent when data is stable, eliminating glitch propagation, and the delays are matched to the worst-case value of the associated functional block.

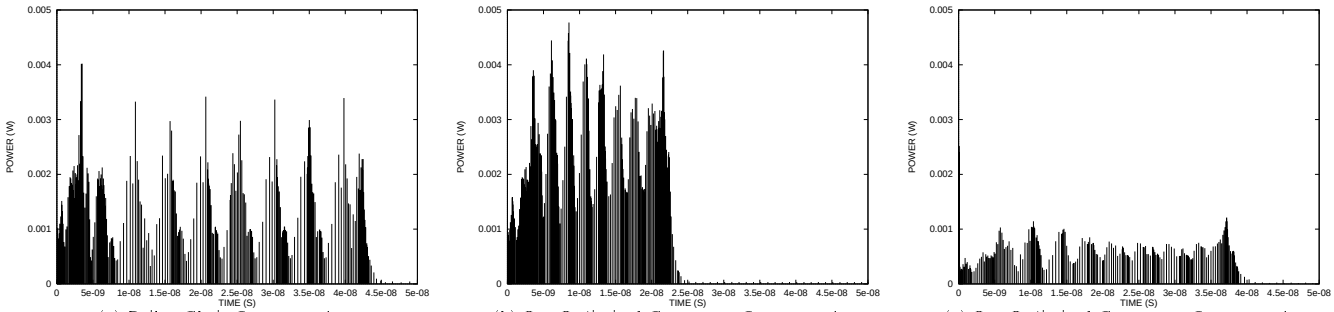
This approach is a valid solution, except for one problem: in the interaction with the latches. As before, overlapped operation introduces the possibility of hazards if operations interact with the same latch. An analysis of the operation of the single-rail datapath (equivalent to the analysis of the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

5.3 Modifications for Safe 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 a source latch (X or Y). Latch Z already stored the information and is only waiting for $C_r\downarrow$ to propagate through the matched delays as a close signal. Figure 10(b) shows the existing latch enable circuit. Two different solutions can be used:

Early close scheme. We can *fast-forward* $C_r\downarrow$ to the *destination latch* so it closes early in the return-to-zero phase instead of at the end. Figure 10(c) shows this simple modification to the latch enable circuit. The latch will not open early so no glitch propagation will occur. This scheme uses some 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 destination latch acknowledge signal is used as an



(a) Bailey Chain Sequencer + Tangram Latches

(b) Our Optimized Concurrent Sequencer + Modified Latches (BEFORE VOLTAGE SCALING)

(c) Our Optimized Concurrent Sequencer + Modified Latches (AFTER VOLTAGE SCALING)

Figure 11: Simulated Power Consumption for 8-bit Dual-Rail Shift-Registers.

enable to the source latch write request. Figure 10(d) shows this modification. Stalling the write operation reduces the performance improvement but guarantees correct operation, independently of the delays in the circuit.

6 Results

We have simulated results using SPICE, targeted to dual-rail implementations. In particular, we simulated several versions of an an 8-stage dual-rail ripple shift-register. Figure 11(a) shows the simulated power consumption of a shift register using a 5 volt power supply; the datapath uses Bailey's chain sequencer and van Berkel's Tangram latches. Figure 11(b) shows power consumption of the design using our optimized concurrent sequencer and modified latches at 5 volts. Our design obtains an 85% improvement in throughput with roughly equivalent total energy consumption. Figure 11(c) shows that the power supply of our new design can be dropped to 3.3 volts, still retaining over 8% of throughput improvement, with energy consumption reduced to *less than a half* (42%) of the original design of Figure 11(c).

SEQUENCER	AREA # transistor	POWER # transition	TIMING MODEL	INIT. ISSUES
TANGRAM	18N-18	10N-10	SI	SELF
CNT/DCODE	15N-6	7N-2	SI	EXT.
BAILEY CHAIN	12N+4	8N-2	FM	EXT.
UNGER TREE	36N-36	16N-16	FM	SELF
BURST-MODE	14N-6	8N-4	FM	EXT.
OPTIMIZED	10N+2	6N	FM	EXT.

TABLE 1: STATIC CHARACTERISTICS OF N-WAY SEQUENCERS.

Table 1 lists relevant analytical results 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. Table 2 shows expected performance of each sequencer controlling N identical processes. G is roughly the delay associated with a 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.

SEQUENCER	INITIAL LATENCY $S_r\uparrow; r_1\uparrow$	$P_i \rightarrow P_{i+1}$ DEAD TIME $a_i\uparrow; r_{i+1}\uparrow$	COMPUTATION TIME $S_r\uparrow; S_a\uparrow$
TANGRAM	(2N-2)G	5G + R	(6N-8)G+NP+(N-1)R
CNT/DCODE	2G	4G + R	(4N-4)G+NP+(N-1)R
BAILEY CHAIN	2G	3G + R	(3N-2)G+NP+(N-1)R
UNGER TREE	2(logN)G	[4logN-2]G MAX	$[\sum_{}^{logN} 2^{logN-i}(4i-2)]$G + 4(logN)G + NP
BURST-MODE	2G	2G	(2N+1)G+NP
OPTIMIZED	2G	2G	2NG+NP

TABLE 2: DYNAMIC BEHAVIOR OF N-WAY SEQUENCERS.

7 Conclusions

This paper has focused on concurrency optimizations, targeted to low-power asynchronous systems. We introduced two new sequencer designs, with greater concurrency than existing designs. New latch and multipliexer designs, that safely accommodate the added concurrency, were presented for both dual-rail and single-rail implementations. In the dual-rail case, results showed improved throughput, providing the opportunity for substantial power savings through voltage scaling. We also indicated attractive features of our single-rail approach over existing approaches.

References

[1] A. Bailey and M. Josephs. Sequencer circuits for VLSI programming. In Asynchronous Design Methodologies, pages 82–90, 1995.

[2] E. Brunvand. Translating Concurrent Communicating Programs into Asynchronous Circuits. PhD thesis, CMU, 1991.

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

[4] 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.

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

[6] 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, 1990.

[7] L.S. Nielsen and J. Sparsø. Low-power operation using self-timed and adaptive scaling of the supply voltage. In International Workshop on Low Power, Napa, California, 1994.

[8] S.M. Nowick and B. Coates. UCLOCK: Automated design of high-performance asychronous state machines. In ICCD, pages 434–441, October 1994.

[9] A. Peeters and K. van Berkel. Single-rail handshake circuits. In Asynchronous Design Methodologies, pages 53–62, May 1995.

[10] L.A. Plana and S.M. Nowick. Concurrency-oriented optimization for low-power asynchronous systems. Technical Report CUCS-017-96, Columbia University, April 1996.

[11] I.E. Sutherland. Micropipelines. CACM, 32(6):720–738, June 1989.

[12] S.H. Unger. A building block approach to unclocked systems. In HICSS, pages 339–348, January 1993.

[13] 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, 1994.

[14] 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.