The Design and Verification of A High-Performance Low-Control-Overhead Asynchronous Differential Equation Solver

Kenneth Y. Yun Peter A. Beerel Vida Vakilotojar Ayoob E. Dooply Julio Arceo

Abstract—

This paper describes the design and verification of a high-performance asynchronous differential equation solver benchmark circuit. The design has low control overhead which allows the average-case delay, tested at $22°\mathrm{C}$ and 3.3V, to be 48% faster than any comparable synchronous design (designed to operate at $100°\mathrm{C}$ and 3V for the slow process corner). The techniques to reduce completion sensing overhead and hide control overhead at the circuit, architectural, and protocol levels are discussed. In addition, symbolic model checking techniques are described that were used to gain higher confidence in the correctness of the timed distributed control.

I. Introduction

It is well known that asynchronous circuits can achieve average-case delay while synchronous circuits achieve a fixed delay determined by their clock frequency that must be preset according to the worst-case conditions. The difference between the average and worst-case conditions stems from both data-dependency and environmental factors. In particular, clock frequencies are set, assuming the chip is running at 100°C, even though the chip usually operates at a much lower temperature. This difference can lead to significant average-performance advantage for asynchronous circuits [1]. Unfortunately, most asynchronous circuits have failed to achieve this goal, primarily because the control overhead associated with asynchronous designs is *typically* very high (with a notable exception of zero-overhead self-timed ring techniques, exemplified by Williams's divider [2]).

This paper describes a number of techniques to hide and minimize asynchronous control overhead that require underlying timing assumptions at the circuit, architectural, and protocol levels. These techniques are described in the context of the design of an asynchronous differential equation solver benchmark circuit. Extensive layout simulations demonstrate that the techniques reduce control overhead to less than 16%. Consequently, when temperature, power supply voltage, and process variations are considered, the design is 48% faster than any comparable synchronous design (designed operate at 100°C and 3V for the slow process corner). Our design style requires some full-custom design of datapath building blocks; it is potentially more difficult to ensure correctness than most traditional asynchronous design methodologies [3], [4], [5]. A brief outline of the design process used is described in Section IV-E.

We consider two sources of control overhead. The first component of control overhead is *completion sensing*, i.e., dynamically determining when a functional unit has finished computing its result. Our design consists of two dynamic 32-bit ALUs and two dynamic 32-bit array multipliers. For both designs, we describe how timing assumptions and optimizing for the average-case can lead to relatively small completion sensing overhead. The second component is *sequencing delay*, i.e., the delay from the termination of one action (e.g., a multiplication) to the start of a subsequent action (e.g., an addition). This sequencing is implemented by control circuits and usually requires at least two gate delays. The architecture of our DIFFEQ design facilitates hiding this delay by performing useful computation (e.g., multiplexing) in between the two actions. To ensure correctness, the worst-case delay of this computation is made smaller than the control delay. In addition, to reduce control delays, timing assumptions are used in the communication protocol, which is implemented with a set of communicating asynchronous FSMs [6].

The application of various timing assumptions made validating the correctness of the design difficult. While most timing assumptions could be verified using SPICE, validating the distributed communication protocol was more difficult because it incorporates timing assumptions that depend on a complex interaction of data-dependent delays. To gain more confidence in the design, we used symbolic model checking [7], [8] based formal verification as a debugging aid. We present our experiences of concurrent verification and design, including the description of the bugs that our verification efforts helped avoid.

The rest of this paper is organized as follows. Section 2 gives an overview of the differential equation solver chip architecture. Section 3 describes the adder and multiplier designs. Section 4 describes the controller design and validation. Section 5 provides the simulation and test results and section 6 draws some conclusions.

II. Chip architecture overview

A high-level description of the differential equation solver algorithm is presented below. This is a widely used high-level synthesis benchmark example from [9].

```
diffeq {
     read ( x, y, u, dx, a );
     while ( x < a ) {
        xl := x + dx;
        ul := u - 3 * dx * (u * x + y);
        yl := y + u * dx;
        x := xl;  u := ul;  y := yl;
     }
     write ( y );
}
```

The algorithm implements the *forward Euler method* to numerically obtain the values of y that satisfy the differential equation $y'' + 3xy' + 3y = 0$ as x ranges from $x(0)$ to a with a step size of dx and with initial values of y and y' equal to $y(0)$ and u, respectively. Using *operator reduction* [9] to convert multiplication by 3 to a shift-and-add operation and *code motion* [9] to pull the shift-and-add operation out of the loop, the data-flow graph depicted in Fig. 1 is obtained.

K. Y. Yun, A. E. Dooply, and J. Arceo are with ECE Dept., University of California, San Diego. P. A. Beerel and V. Vakilotojar are with EE-Systems Dept., University of Southern California.

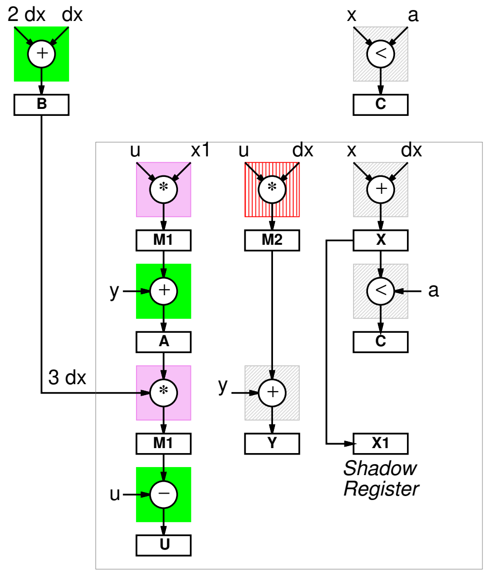

Fig. 1. Dataflow graph.

There are three threads of operations needed per iteration: computing y and y' (denoted as u in Fig. 1) and incrementing x. Computing y' requires a sequence of four operations (multiply → add → multiply → add) whose input operands are the results of immediately preceding operations. However, computing y and updating x only require two operations each. Thus the computation of y' is the critical path of the dataflow graph. The shading of operations in Fig. 1 depicts the *binding* of operations to two ALUs and two multipliers.

We allocated one dynamic ALU and one dynamic multiplier to compute y' (*resource sharing*). After the multiplier finishes computing $u * x1$ ($x1$ is a copy of x), the result is stored in $M1$ register. While the ALU is computing $y + M1$, the multiplier is precharged for the next operation. After $y + M1$ is stored in A register, the multiplier starts evaluating $3dx * A$ and the ALU is precharged. After the multiplication is complete, the result is stored in $M1$. The ALU then begins evaluating $u - M1$ and the multiplier is precharged. Thus each functional unit performs the computation in an alternating fashion. We selected this particular binding of resources to operations in order to remove the precharge delays of the functional units from the critical path.

We allocated a second dynamic ALU and a second dynamic multiplier to compute y and x. Here the precharge delay of the ALU could not be hidden because three consecutive ALU operations are required. Consequently, the execution time of these operations is only slightly smaller than the delay required to compute y', although the evaluation delay of the ALU is substantially less than that of the multiplier.

We used a standard MUX-based architecture as depicted in Fig. 2. In many instances, we reduced the number of controller outputs by carefully scheduling the control signal transitions so that one output can serve multiple functions. For example, LA is used as a MUX select signal and the *opcode* signal for ALU1 as well as the register A load signal. For the registers, we used Svensson-style positive-edge-triggered flip-flops which were made pseudo-static by adding weak feedback transistors. These flip-flops have less overhead than the often-used special asynchronous latches that require full handshake communications for reading and writing. Thus structurally the only difference between our datapath architecture and typical synchronous ones is the presence of the done signals generated from the functional units.

We used an event-driven protocol for control. For example, the external environment initiates the first step of this protocol by raising start (to command both the ALU1 and ALU2 controllers to begin their operations). The ALU1 controller responds by lowering ALU1's precharge signal, which triggers ALU1 to compute $2dx + dx$. Similarly, the ALU2 controller responds by triggering ALU2 to compute the loop test ($X < a$). As soon as each functional unit completes its computation, it raises its done signal. Each ALU controller, upon detecting the done signal from its functional unit, responds by latching the result and sending a communication signal to both multiplier controllers. In addition, each ALU controller prepares for its next operation by setting up the MUX and function select signals and precharging its functional unit. Subsequent multiplication operations begin after the communication signals are asserted by both ALU controllers. The details of control design are described in Section IV.

III. Datapath building blocks: adder and multiplier

This section describes the adder and multiplier designs used in the differential equation solver: in particular, the techniques used to achieve superior average-case delay over synchronous designs and used to minimize the completion sensing overhead.

A. Adder

The 32-bit adder used in our design is self-timed, i.e., the completion of computation is not pre-timed but detected and reported by the circuit itself. Unlike conventional self-timed (ripple carry) adders, however, our design incorporates a carry bypass logic to speed up the worst-case computation. Thus our adder design uses both a worst-case delay reduction mechanism (carry bypass) and an average-case delay reduction feature (completion sensing).

A.1 Motivation

A strong justification for self-timed ripple carry adders is found in one of Von Neumann's early papers which appeared in [10]. It is a well-known fact that the delay of a ripple carry addition corresponds to the length of the longest carry chain (carry propagation). Von Neumann proved that for *random* input statistics the *average length* of the longest carry chain is $\log_2 n$ for n-bit addition. Thus, for random inputs, the average delay of 32-bit addition is equal to the worst-case delay of 5-bit addition.

In most applications [11], [12], however, input statistics are not as favorable for self-timed ripple carry adders: i.e., a large fraction of carry chains tends to be long. Arithmetic operations on data (as opposed to address calculations), in particular, tend to generate long carry chains. In order to understand the effects of realistic input statistics, we conducted several experiments us-

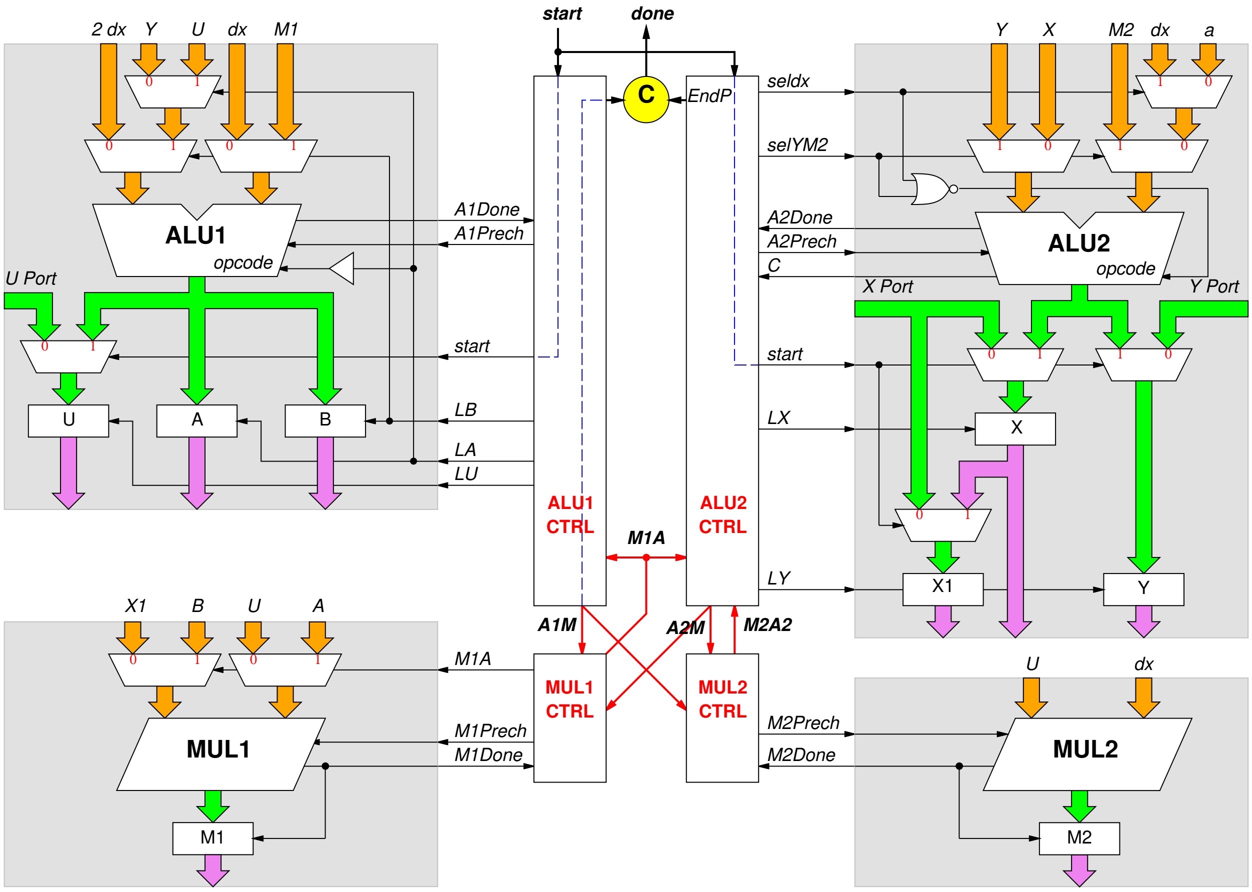

Fig. 2. Datapath and control.

ing an ARM simulator[1] and standard benchmark traces. In fact, according to ARM benchmarks, the average longest carry chain length for 32-bit *data arithmetic* operations is 17, as shown in Fig. 3, which is far from the theoretical average longest carry chain length of 5. However, the worst-case longest carry chain length can be reduced to considerably less than 17 by employing a simple carry bypass or lookahead feature. For example, carry bypass adders with bypassing done at 4-bit boundaries can achieve the *worst-case* longest carry chain length of 12, assuming that the time it takes for one stage of 4-bit carry bypass is the same as that for one bit carry propagate. Under the same assumption, the average longest carry chain reduces to 7 in the benchmarks. The motivation for our self-timed carry bypass adder design is largely based on this experiment.

In order to detect and report the completion of computation whose delay varies depending on data operands, a completion sensing circuit is required. However, to make the average delay of self-timed carry bypass adder circuit as short as possible, the overhead for completion sensing must be minimized. In other words, the difference in time between when the completion is reported and when the last changing sum bit reaches its final value must be minimized. In the benchmark simulation shown in Fig. 3, for the carry bypass case to be faster than the

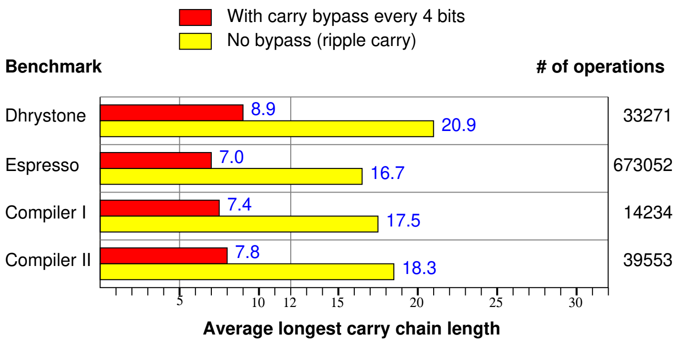

Fig. 3. Average longest carry chain lengths for *data arithmetic* operations from ARM benchmarks.

synchronous worst-case, the average completion sensing overhead must be smaller than the delay equivalent to the carry chain length of 5 (worst-case longest carry chain length minus average-case longest carry chain length).

The design details of our carry bypass adder are described below. We present a basic precharged Manchester carry chain adder, a dual-rail implementation of the Manchester carry chain adder, the carry bypass circuit, and the completion sensing circuit — in particular, we focus on the technique to minimize average completion sensing overhead.

[1]We used a simulator for a commonly used RISC processor, ARM-6, from Advanced RISC Machines, in order to obtain the ALU input operand statistics for data arithmetic operations.

A.2 Design

Our 32-bit adder, as shown in Fig. 4, is divided into eight 4-bit sections and a completion sensing circuit. Each section consists of carry-propagate, carry-kill, and carry-generate bit slices (PKG), sum bit slices, a 4-bit Manchester carry chain (MCC), and a carry bypass circuit.

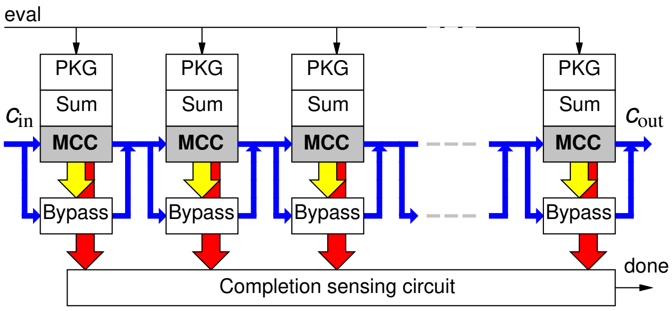

Fig. 4. Self-timed carry bypass adder block diagram.

The heart of our carry bypass adder design is the Manchester carry chain. In the ith bit slice of a single-rail static MCC, carry-propagate (p_i), carry-kill (k_i), and carry-generate (g_i) are used to propagate carry-in (c_i) to carry-out (c_{i+1}), reset carry-out, and set carry-out, respectively; p_i, k_i, and g_i signals are mutually exclusive and defined as:

$$p_i = A_i \oplus B_i \tag{1}$$
$$k_i = \overline{A_i}\ \overline{B_i} \tag{2}$$
$$g_i = A_i\ B_i \tag{3}$$

where A_i and B_i represent the ith input bits. The sum (S_i) then can be computed as below:

$$S_i = p_i \oplus c_i \tag{4}$$

In order to detect whether the carry-out of each bit slice has been computed, our design uses a *dual-rail* carry chain defined as follows:

$$c^{\mathsf{T}}_{i+1} = g_i + p_i\ c^{\mathsf{T}}_i \tag{5}$$
$$c^{\mathsf{F}}_{i+1} = k_i + p_i\ c^{\mathsf{F}}_i \tag{6}$$

where c^{T}_i and c^{F}_i are the *true* and *false* rails of the ith carry bit. To detect that *all* of the carry bits have been computed, the completion sensing circuit implements the following function:

$$\text{done} = \prod_{i=0}^{31} (c^{\mathsf{T}}_i + c^{\mathsf{F}}_i) \tag{7}$$

The circuit implementation of the dual-rail Manchester carry chain is shown in Fig. 5. Before the computation begins, all the internal nodes ($\overline{c^{\mathsf{T}}_i}$'s and $\overline{c^{\mathsf{F}}_i}$'s) are precharged to 1. Thus both true and false rails of carry signals (c^{T}_i and c^{F}_i) are low. After the evaluation begins, exactly one control signal (p_i, k_i, or g_i) per bit slice is asserted. If g_i is asserted, node $\overline{c^{\mathsf{T}}_{i+1}}$ is discharged enabling c^{T}_{i+1} to rise; since k_i and p_i remain low, the voltage level at node $\overline{c^{\mathsf{F}}_{i+1}}$ remains high, keeping c^{F}_{i+1} low. On the other hand, if k_i is asserted, node $\overline{c^{\mathsf{F}}_{i+1}}$ is discharged enabling c^{F}_{i+1} to rise; since g_i and p_i remain low, c^{T}_{i+1} remains low. If p_i is asserted, both g_i and k_i remain low; thus the NMOS transistors controlled by g_i and k_i remain turned off, and the logic values at c^{T}_i and c^{F}_i are propagated to c^{T}_{i+1} and c^{F}_{i+1} respectively. Therefore, in all three cases, for each bit slice exactly one carry rail becomes high when the computation is complete.

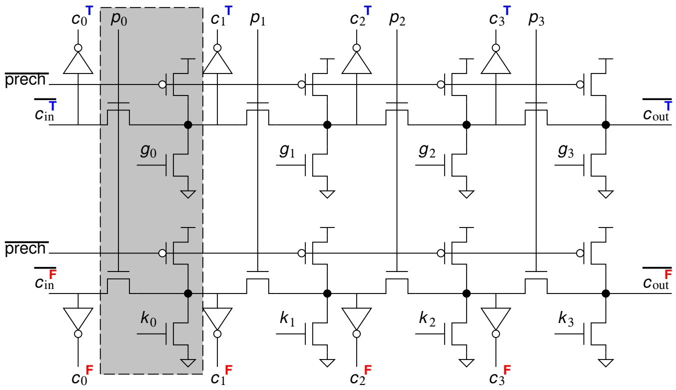

Fig. 5. Dual-rail Manchester carry chain.

The PKG and sum bit slices are implemented in domino gates as shown in Fig. 6. The sum logic is designed as a domino gate for two reasons: (1) domino gates are faster than static XOR gates and (2) domino outputs are glitch-free (hence generates less noise). However, unclocked domino sum gates require monotonically rising inputs; thus both carry-in and carry-propagate signals must be in dual-rail form:

$$S_i = p^{\mathsf{T}}_i c^{\mathsf{F}}_i + p^{\mathsf{F}}_i c^{\mathsf{T}}_i \tag{8}$$

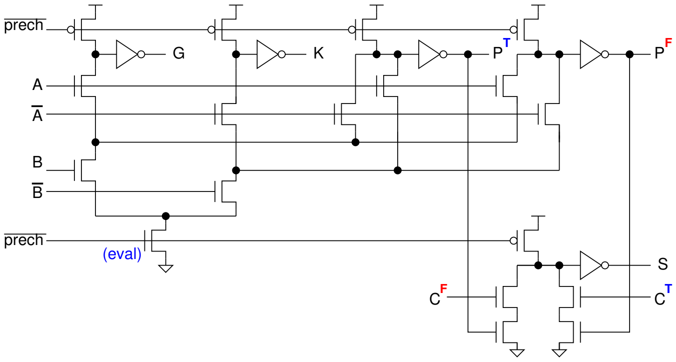

Fig. 6. Precharged carry propagate/kill/generate and sum circuit.

The input-output behavior of the jth 4-bit Manchester carry chain can be described as:

$$c^{\mathsf{T}}_{4j+4} = G_{4j} + P_{4j}\ c^{\mathsf{T}}_{4j} \tag{9}$$
$$c^{\mathsf{F}}_{4j+4} = K_{4j} + P_{4j}\ c^{\mathsf{F}}_{4j} \tag{10}$$

where G_{4j}, K_{4j}, and P_{4j} are group-generate, group-kill, and group-propagate signals for the 4-bit section:

$$G_{4j} = g_{4j+3} + p_{4j+3}\ g_{4j+2} + p_{4j+3}\ p_{4j+2}\ g_{4j+1} +$$

$$
\begin{aligned}
& p_{4j+3}\, p_{4j+2}\, p_{4j+1}\, g_{4j} && (11)\\
K_{4j} \;=\; & k_{4j+3} \;+\; p_{4j+3}\, k_{4j+2} \;+\; p_{4j+3}\, p_{4j+2}\, k_{4j+1} \;+ && \\
& p_{4j+3}\, p_{4j+2}\, p_{4j+1}\, k_{4j} && (12)\\
P_{4j} \;=\; & p_{4j+3}\, p_{4j+2}\, p_{4j+1}\, p_{4j} && (13)
\end{aligned}
$$

In order to speed up the carry propagation for the worst-case addition, in which the carry is propagated from the least significant bit to the most significant, carry bypass circuits are inserted between successive four-bit sections. If every bit slice in the 4-bit section propagates the carry, i.e., group-propagate, then the carry propagation can be sped up by *bypassing* the MCC as shown in Fig. 7.

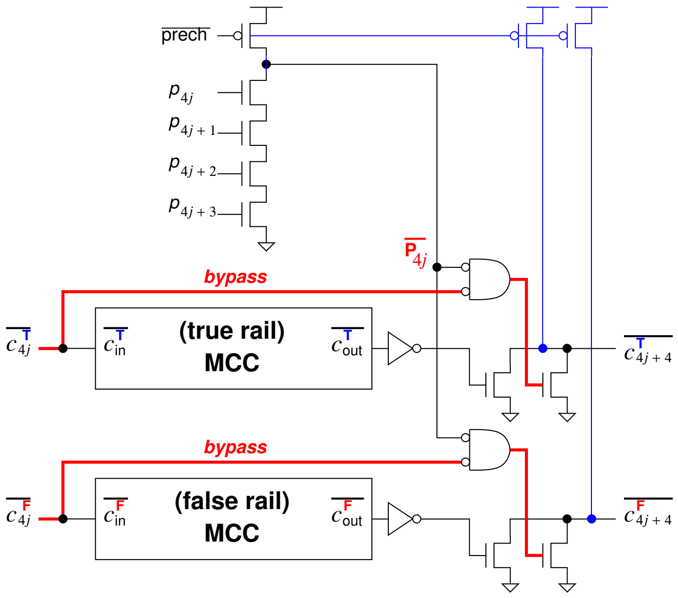

Fig. 7. Carry bypass circuit.

Nodes $\overline{c^{\mathsf{T}}_{4j+4}}$ and $\overline{c^{\mathsf{F}}_{4j+4}}$ are both precharged to 1 before the evaluation begins. If all four propagate signals (p_{4j}, p_{4j+1}, p_{4j+2}, p_{4j+3}) become high after the evaluation begins, the group propagate signal P_{4j} is asserted. This enables $\overline{c^{\mathsf{T}}_{4j}}$ and $\overline{c^{\mathsf{F}}_{4j}}$ to propagate to $\overline{c^{\mathsf{T}}_{4j+4}}$ and $\overline{c^{\mathsf{F}}_{4j+4}}$ respectively. If the group propagate signal is not asserted, the carry signal is either generated or killed in the corresponding 4-bit section of the Manchester carry chain. This enables $\overline{c^{\mathsf{T}}_{\mathrm{out}}}$ or $\overline{c^{\mathsf{F}}_{\mathrm{out}}}$ to be asserted, which in turn enables $\overline{c^{\mathsf{T}}_{4j+4}}$ or $\overline{c^{\mathsf{F}}_{4j+4}}$. This feature allows the carry to propagate to the next MCC at the earliest possible opportunity.

The completion sensing circuit in our design is a domino logic implementation of the OR-AND circuit as shown in Fig. 8. Since the order of sum bit generation cannot be known a priori, the circuit must detect that the logic level of *every* sum bit has reached the final value. In practice, it is sufficient to detect that every *carry* bit has reached the final value, as long as the delay from a carry to the corresponding sum bit is always smaller than the minimum delay through the completion logic, which in fact is the case in our circuit. Thus the delay from carry to sum generation is effectively *hidden* in the completion sensing computation.

In order to further minimize the completion sensing overhead for the worst-case computation, late arriving signals (for the worst-case computation) are placed closer to the output. According to the SPICE simulation (at 3.3V and 22°C) with realistic output loading, although the completion sensing overhead is as much as 50% of the sum delay when there is no carry propagation, it is only 5% when the computation takes the worst-case delay: i.e., when the carry propagates through the entire carry chain. As a result, the actual delay from the beginning of evaluation to the completion sensing is only 12% longer (5% with realistic output loading) than the worst-case delay and usually is much shorter than the worst-case delay.

Our results indicate that the average computation time including completion sensing delay is 2.8ns for ARM data arithmetic benchmark traces. This outperforms most other published self-timed adder designs [13], [11]. It is somewhat slower ($< 15\%$) than recently published self-timed (Brent and Kung style) carry lookahead adders, which requires much greater area ($> 4\times$), using speculative completion technique [14]. More importantly, the completion sensing overhead of our design is only 20% of the average-case delay and 12% of the worst-case delay.

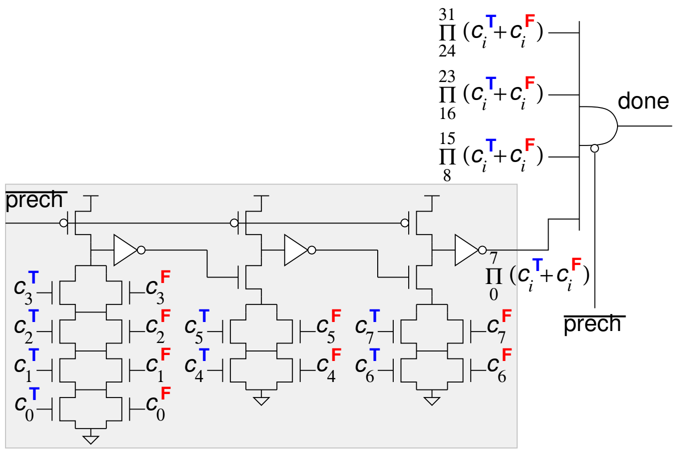

Fig. 8. Completion sensing domino logic.

A.3 Results

We implemented our design in a 0.5μm HP CMOS14TB triple-metal process. Table I shows the results of test and SPICE simulation with 3.3V power supply at 22°C and with each sum gate driving a realistic load (shown in Fig. 9), using the process parasitics from the run (N68MAR) in which our chips were fabricated. We list simulation results as well as test results because not all relevant data are available from tests. The evaluation delays from the test measurements are about 20% smaller than those from the simulation runs. We conjecture that this discrepancy is partly due to the conservative wire capacitance extraction from the layout. The test results for different operating conditions are reported in section V.

Adding -1 and 1, one of the cases that yield the longest carry chain, results in the worst-case eval $\rightarrow$ done delay of 3.96ns. $t_{\mathsf{eval}\rightarrow\mathsf{sum}}$ is omitted from Table I because the sum bits, once precharged, remain 0 for this computation. Adding -2 and 0 is another computation with the longest carry chain. In this case, the *false* rail of the carry is "produced" in the least significant

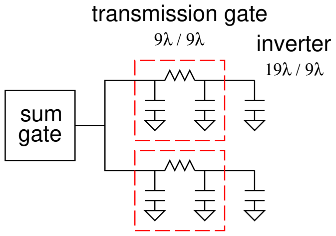

Fig. 9. Sum gate loading.

bit and propagated to the most significant bit. The test result for $t_{\text{eval}\to\text{done}}$ is 3.67ns. In this case, we were able to simulate and observe eval $\to$ sum and eval $\to$ done delays and derive the completion sensing overhead. The overhead of 0.21ns amounts to 5% of the simulated eval $\to$ sum delay.

The best-case delay is 1.56ns, which results from $0 + 0$ computation. Again we were unable to deduce the completion sensing overhead for this addition, because the sum bits do not change once precharged. Instead, we use another case $(-1 + (-1))$ to deduce the completion sensing overhead for the computation with the carry chain length of 0. The test result for $t_{\text{eval}\to\text{done}}$ is 1.67ns. The overhead of 0.70ns is 50% of the simulated eval $\to$ sum delay of 1.37ns.

We observe that the computation with the longest carry chain has the *minimum* completion sensing overhead and the one with the carry chain length of zero has the *maximum* overhead, which is exactly the desired characteristic to achieve superior average-case delay and yet maintain very competitive worst-case delay. We confirmed that the completion sensing overhead for random cases fall between 0.2ns and 0.7ns using both SPICE and software simulations.

Case	Test	Simulation		
	$t_{\text{eval}\to\text{done}}$	$t_{\text{eval}\to\text{done}}$	$t_{\text{eval}\to\text{sum}}$	Overhead
-1 + 1	3.96ns	4.76ns	–	N/A
-2 + 0	3.67ns	4.68ns	4.47ns	0.21ns
-1 + 0	3.71ns	4.42ns	4.16ns	0.26ns
-1 + (-1)	1.67ns	2.07ns	1.37ns	0.70ns
0 + 0	1.56ns	2.03ns	–	N/A

TABLE I
ADDER TEST AND SIMULATION RESULTS (3.3V, 22°C, MOSIS N68MAR RUN); EACH SUM GATE DRIVES A LOAD DEPICTED IN FIG. 9.

Using the extracted delay components from the SPICE simulations, we computed the average-case delay for the ARM data arithmetic operation benchmarks. The simulated average-case delay was 2.8ns (with no loading on outputs). We also compared our design to a comparable synchronous design: the same design as our self-timed design but excludes the completion sensing circuit. Conventional synchronous designs typically use a single-rail carry chain and static XOR gates for sum generation. However, "our synchronous design" contains a dual-rail carry chain and domino sum logic. The cost of adding an additional carry chain is *one* additional transistor load on p_i's, but it is more than offset by faster precharged sum gates. We simulated "our synchronous design" in the same operating environment. The *worst-case* delay (when -1 and 1 are added) was 3.8ns. Thus we concluded that the average-case delay (including completion sensing overhead) of our self-timed design is 36% faster than comparable synchronous designs, with only 12% overhead (with no loading on outputs) for the worst-case computation. This speedup is based solely on data dependency, *not* considering variations in environmental factors such as temperature and power supply voltage. The effects of temperature and voltage variations are discussed in Section V.

B. Multiplier

Our 32-bit multiplier is an array multiplier which consists of 16 rows of carry save adders (CSA) and a 32-bit carry propagate adder (CPA) operating in parallel with the CSA array as depicted in Fig. 10. Because the upper 32 bits of the 64 bit product are *not needed in this design*, only the right half of the CSA array is used. We show below that requiring only lower 32 bits of the product yields interesting optimization opportunities.

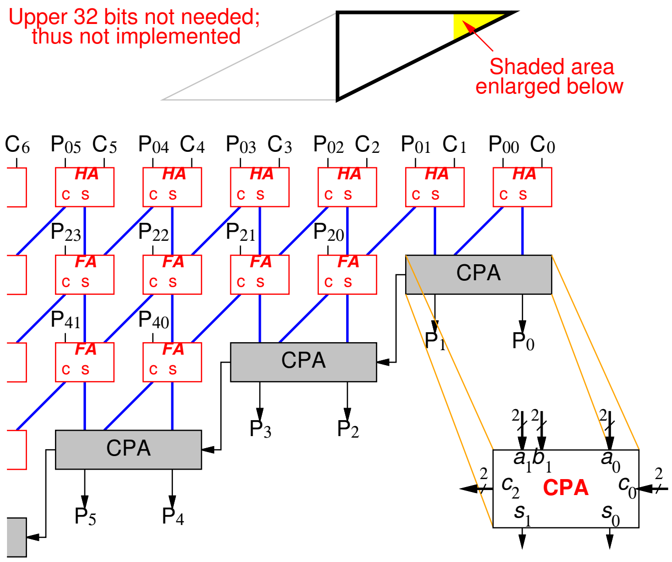

Fig. 10. A fragment of domino dual-rail array multiplier. $P_{2i,j}$ represents the jth bit of partial product P_{2i}; $C_{2i} = 1$ $(C_{2i+1} = 1)$ if partial product P_{2i} is two's complement of A $(2A)$ (see Table II).

B.1 Architecture

Our decision to select a non-pipelined array multiplier is based on the nature of this application. The differential equation solver is to solve one equation at a time; thus the most important criterion is latency, not throughput. For a 32-bit by 32-bit multiplication, the total number of partial products is normally 32. In our design, we use the radix-4 Booth encoding to reduce the number of partial products to 16. Given a multiplicand, A, and a multiplier, B, the product, $A \times B$, is:

$$\begin{aligned} A \times B &= A \times \{-b_{31}2^{31} + \sum_{i=0}^{30} b_i 2^i\} \\ &= A \times \sum_{i=0}^{15} \{-b_{2i+1}2^{2i+1} + b_{2j}2^{2j} + 2b_{2i-1}2^{2i-1}\} \end{aligned}$$

$$= \sum_{i=0}^{15} A \times \hat{b}_{2i} 2^{2i}, \tag{14}$$

where $b_{-1} = 0$ and $\hat{b}_{2i} = -2b_{2i+1} + b_{2i} + b_{2i-1}$. The partial product generation is summarized in Table II.

$\hat{b}_{2i}$	Partial Product P_{2i}	C_{2i+1}	C_{2i}
-2	Two's comp of $2A$	1	0
-1	Two's comp of A	0	1
0	0	0	0
1	A	0	0
2	$2A$	0	0

TABLE II
PARTIAL PRODUCT GENERATION BASED ON RADIX-4 BOOTH ENCODING.

Each row of the CSA array computes the running sum of the partial products in carry-save form, i.e., carry bits are not propagated in ripple carry fashion but are forwarded to the next row with twice the weight. Thus each row requires a set of full adders (FA) to add a partial product to the running sum in carry-save form. Because the radix-4 Booth encoding requires that each successive partial product be shifted two bit positions to the left as derived in Eqn. (14), each row of the CSA array generates "spillover" bits, which form two 32-bit numbers that need be added, as depicted in Fig. 10. The *key observation* here is that a self-timed CPA can be employed to perform this addition in parallel with the CSA computation, as soon as the "spillover" bits are generated from the CSA. Our design uses a 2-bit CPA per each row of the CSA array to add the "spillover" bits from the CSA array, as illustrated in Fig. 10. As long as the 2-bit CPA is as fast as the FA in the CSA array (though it needs not be faster for the multiplier to function correctly), our multiplier can compute lower 32 bits of the 64-bit product in the time required for adding 16 partial products in carry-save form. Thus, for applications requiring *only the least significant 32 bits*, our technique, which is generally applicable to both synchronous and asynchronous designs, leads to a significant reduction in latency.

B.2 Circuit implementation

Our design uses a self-timed dual-rail domino logic to speed up the latency of the carry save addition. The self-timed domino allows the signals to propagate (or "dominos to fall") at its natural speed unencumbered by the evaluation clock adjusted for the worst-case. Since unclocked domino logic requires that all inputs be monotonically rising once the evaluation begins, our design uses the dual-rail signaling. Dual-rail domino gates for sum and carry logic are shown in Fig. 11. There are three dual-rail inputs to each domino gate: sum and carry from the preceding row and a partial product bit. The position of partial product bits in the series stack plays a significant role in avoiding charge sharing problems. In our design, the partial product input (pp) is connected to a^{T} and its complement ($\overline{pp}$) to a^{F}. When the evaluation begins, internal nodes 1 and 2 are precharged to $V_{\rm dd}$, because the partial product bits are set up before the evaluation begins. This makes the effective stack size to be two during evaluation, which makes the charge sharing problem (caused by one of the transistors controlled by b^{T} or b^{F} turning on while the transistors controlled by c_i^{T} and c_i^{F} are off) negligible.

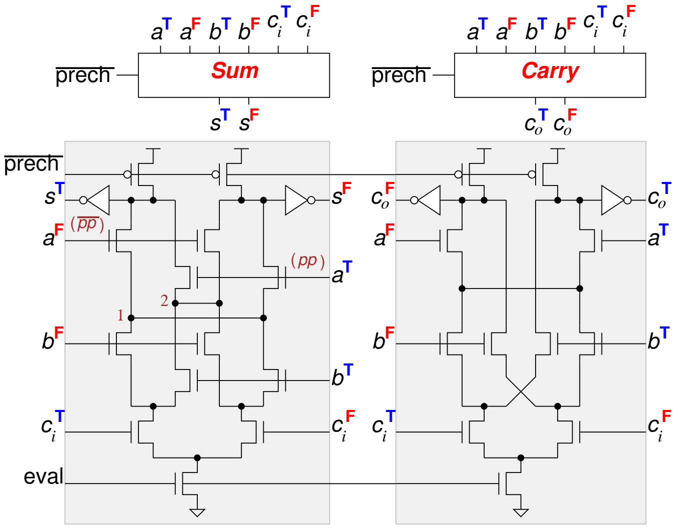

Fig. 11. Dual-rail domino full adder.

In order to make the 2-bit CPA faster than the FA used in the CSA array, our design uses a 2-bit dual-rail domino carry lookahead adder (CLA) as described below:

$$c_1^{\mathsf{T}} = c_0^{\mathsf{T}} a_0^{\mathsf{T}} \tag{15}$$
$$c_1^{\mathsf{F}} = c_0^{\mathsf{F}} + a_0^{\mathsf{F}} \tag{16}$$
$$c_2^{\mathsf{T}} = c_0^{\mathsf{T}} a_0^{\mathsf{T}} (a_1^{\mathsf{T}} + b_1^{\mathsf{T}}) + a_1^{\mathsf{T}} b_1^{\mathsf{T}} \tag{17}$$
$$c_2^{\mathsf{F}} = (c_0^{\mathsf{F}} + a_0^{\mathsf{F}})(a_1^{\mathsf{F}} + b_1^{\mathsf{F}}) + a_1^{\mathsf{F}} b_1^{\mathsf{F}} \tag{18}$$
$$p_1^{\mathsf{T}} = a_1^{\mathsf{T}} b_1^{\mathsf{F}} + a_1^{\mathsf{F}} b_1^{\mathsf{T}} \tag{19}$$
$$p_1^{\mathsf{F}} = a_1^{\mathsf{T}} b_1^{\mathsf{T}} + a_1^{\mathsf{F}} b_1^{\mathsf{F}} \tag{20}$$
$$s_0 = a_0^{\mathsf{T}} c_0^{\mathsf{F}} + a_0^{\mathsf{F}} c_0^{\mathsf{T}} \tag{21}$$
$$s_1 = p_1^{\mathsf{T}} c_1^{\mathsf{F}} + p_1^{\mathsf{F}} c_1^{\mathsf{T}} \tag{22}$$

Because the longest series stack in the c_2 logic of our 2-bit CPA is 3 whereas the stack size of the sum gate in the FA is 4, the c_2 bit of a 2-bit CPA is computed before the outputs of the FAs in the corresponding row of the CSA array. In other words, the carry propagation never impedes the evaluation of a CPA stage; the CPA stage begins the evaluation as soon as the sum and carry signals from the CSA arrive.

Normally, the 32-bit CPA requires a completion sensing circuit implemented in 32-bit OR-AND as shown in Eqn. (7). Since the generation of carry signals is staggered, a domino *chain* is the most suitable circuit topology to use. However, in this design, we can further simplify the circuit as shown in Fig. 12 by exploiting the fact that $c_0 \ldots c_{30}$ are always computed before c_{31}. Thus our implementation of done logic is a domino OR gate: $c_{31}^{\mathsf{T}} + c_{31}^{\mathsf{F}}$. Although the sum generation takes an additional domino gate delay after the carry bits are computed, done signal is guaranteed not to rise until all of the sum bits of the CPAs, i.e., the final product bits, have been generated, because the done signal is buffered significantly to drive the controller or the registers.

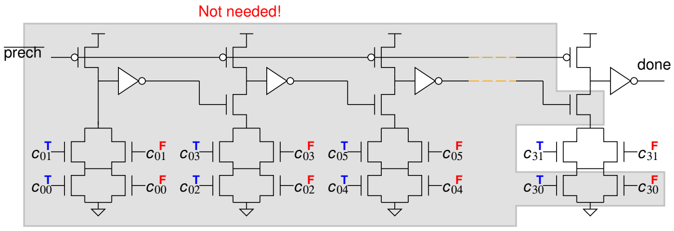

Fig. 12. Completion sensing circuit for domino multiplier.

	Test	Simulation		
Temperature Power supply Process	22°C 3.3V N68MAR	22°C 3.3V N68MAR	100°C 3.0V Slow	0°C 3.6V Fast
Evaluate delay	6.7ns	7.7ns	12.8ns	5.9ns
Precharge delay	3.8ns	4.4ns	7.2ns	3.5ns

TABLE III
MULTIPLIER SIMULATION AND TEST RESULTS.

B.3 Results

We laid out our multiplier so that the input to output data flow is in the same direction as the rest of the datapath. The floorplan of the multiplier is shown in Fig. 13. Both the multiplicand and the multiplier arrive from the right horizontally (bit 31 at the top and bit 0 at the bottom) and the 32-bit product is generated on the left.

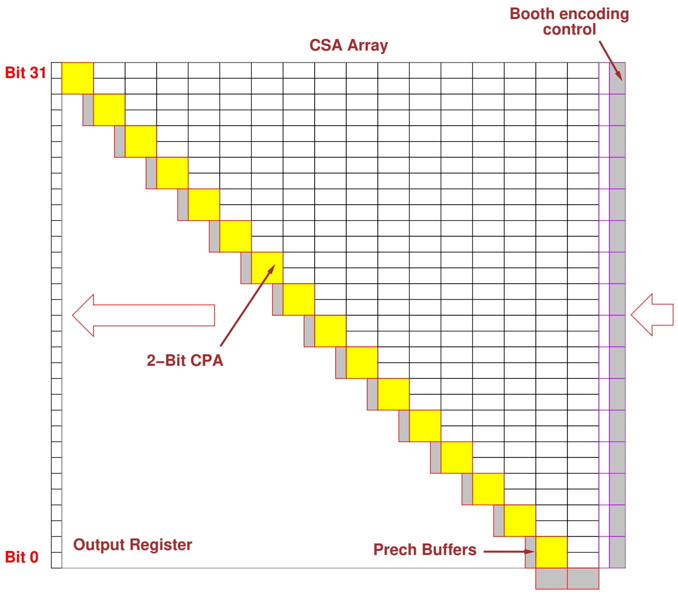

Fig. 13. Multiplier floorplan.

We implemented our design in a $0.5\mu\mathrm{m}$ HP CMOS14TB triple-metal process and simulated the backannotated design for three test corners as shown in Table III. The average delay from the beginning of evaluation to the assertion of done is 7.7ns, when simulated at 22°C and 3.3V using the process parasitics from the run (N68MAR), in which our chips were fabricated. The CSA delay (delay from evaluation to CSA done) is 7.1ns; thus the computation time beyond the CSA delay is only 0.6ns. The completion sensing overhead is actually *zero*, if the buffering delay for the done signal is excluded. Finally, the precharge buffer delay is 1.5ns.

We measured the multiplier evaluation and precharge delays of the fabricated chips. The test results are also shown in Table III. The actual measured delays are somewhat faster than the simulation results — the measured evaluation and precharge delays are 6.7ns and 3.8ns whereas simulated evaluation and precharge delays are 7.7ns and 4.4ns respectively. Our best guess for this discrepancy is that the wire capacitance extraction from the layout is too conservative.

To the best of our knowledge, our design outperforms, in terms of *latency*, almost all published 16-bit or 32-bit array multiplier designs (fabricated in a similar technology) which generate 32-bit products. The closest competitor in the same application domain is the 500MHz 16-bit pipelined array multiplier used in a DSP core [15], which has 8ns latency. This chip was fabricated in $0.35\mu\mathrm{m}$ process and uses 2.5V power supply.

IV. CONTROLLER DESIGN AND VALIDATION

A. *Overview*

The basic difference between our asynchronous design and traditional synchronous designs is in the scheduling of control. In synchronous designs, the assertion of the control signals is tied to clock edges, which can often lead to non-trivial *dead time* between the completion of a computation and the start of the next clock cycle. However, the event-driven sequencing based on completion sensing in our design reduces the dead time between the completion of a computation and the start of the next computation significantly.

A.1 Distributed control

We implemented the control using distributed *burst-mode* controllers, a class of asynchronous FSMs described in Section IV-B, to reduce the latency of control. As in synchronous systems, operations must be assigned to state transitions a priori; however, state transitions are event-driven and occur as soon as a specified set of input transitions arrive. To obtain the flexible operation sequencing, we used four control units, one for each datapath unit, as depicted in Fig. 2. A high-level synthesis tool, such as ACK [16], would have been beneficial partitioning the design and insuring the correctness of partitioning, although we did not make use of it.

This distributed control paradigm was also motivated by layout considerations. The controller for each datapath unit is located next to the unit. This minimizes the length of the datapath control signals, thereby reducing the control wire delays in the critical path; however, it makes a few inter-controller communication wires run half the length of the chip. However, the longer delays on these communication wires do not affect the performance adversely because, by and large, their delays are *hidden* in the datapath computation, as described in Section IV-C.

A.2 Inter-controller protocol optimization

Communication wires are typically designed in pairs connecting a designated sender to a designated receiver. The sender traditionally initiates the communication by toggling one wire and the receiver acknowledges the communication by toggling the other wire. The process would either repeat using the opposite phase as in *two-phase handshaking* or the signals would reset in preparation for the next communication as in *four-phase handshaking*. Typically, designs consist of either all two-phase or all four-phase handshaking. In our design, however, we use a mixture of both. The communication signals M1A and A1M follow a two-phase protocol while the rest of the signals follow a four-phase protocol. Our design is also different in that some *acknowledgment* signals indicate the existence of new data. In particular, M2A2+ acts to acknowledge A2M+ as well as indicate to ALU2 that register $M2$ has been updated with $u * dx$.

To reduce the number of communication wires, however, we used *broadcast communication signals* which connect a sender to multiple receivers. For example, A1M connects ALU1 controller to both multiplier controllers. This is efficient because both multipliers must wait for U to be updated by ALU1 before starting the subsequent loop iteration. Similarly, A2M is used to indicate to both multipliers that the loop test ($X < a$) passes and a loop iteration should start.

Furthermore, in our design, not all broadcast request signals are explicitly acknowledged by their receivers. For example, a broadcast request signal, A1M, is sent to both MUL1 and MUL2 but acknowledged explicitly only by MUL1 controller. Typical *speed-independent* asynchronous protocols do not allow this because it cannot be guaranteed that MUL2 controller would have a chance to properly react to A1M+ before A1M− occurs (see Fig. 14). In our design, however, we analyzed the relative timings of all signals and verified that A1M+ always occurs when MUL2 is waiting for it and that MUL2 always has a chance to properly react before A1M− occurs. Thus, an explicit acknowledgment would be redundant. This analysis allowed us to keep the controllers simple, which improved their latencies up to 15%.

The interpretation of all the inter-controller communication signals is summarized in Table IV. The implementation and validation of the communication protocol are discussed in detail in Section IV-D.

Signal Name	Trans	Interpretation
A2M	+	Start new iteration
	−	Reset
A1M	+	U/B updated
	−	A updated with $y + M1$
M1A	+	$M1$ updated with $u * x1$
	−	$M1$ updated with $B * A$
M2A2	+	$M2$ updated with $u * dx$
	−	Reset
start	+	X, $X1$, Y, U, dx, and a initialized
	−	Reset
EndP	+	Terminate loop
	−	Reset
done	+	Solver done
	−	Reset

TABLE IV
INTER-CONTROLLER COMMUNICATION DESIGN.

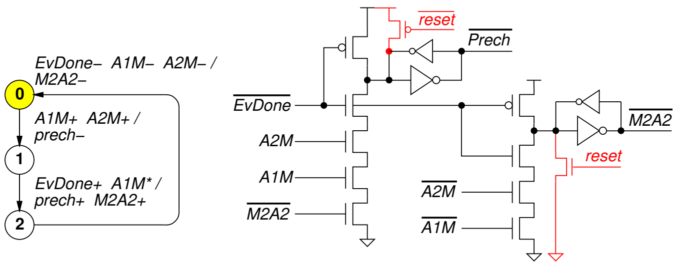

Fig. 14. MUL2 controller specification and implementation.

B. Controller specification and implementation

This subsection describes the specification and implementation of one of the controllers and the constraints on its environment.

B.1 Extended burst-mode specification

The controllers were specified in extended burst-mode (XBM), an asynchronous FSM specification format. We only describe the details of the XBM specification relevant to the discussions in this paper. The complete details are described in [6].

Fig. 14 describes the controller for MUL2, one of the four extended burst-mode controllers used in our design. Signals ending with + or − are *terminating signals*; the ones ending with $*$ are *directed don't cares*. If a state transition is labeled with a directed don't care a^*, then the following state transition must be labeled with a^* or a^+ or a^-. A terminating signal a^+ denotes a $0 \rightarrow 1$ transition of a if a is initially 0, and no transition at all if a is initially 1. A sequence of state transitions labeled with a^* and terminated with a^+ represents a *single* $0 \rightarrow 1$ transition of a at any point in the sequence. A terminating signal not immediately preceded by a directed don't care represents a *compulsory* transition.

An input burst is a non-empty set of input transitions (*terminating* or *directed don't care*), at least one of which must be compulsory. An output burst consists of a possibly empty set of output transitions. In a given state, when all the terminating transitions in the input burst have appeared, the machine generates the corresponding output burst and moves to a new state. Specified transitions in the input burst may appear in arbitrary temporal order; outputs may be generated in any order.

B.2 Controller implementation style

The controller specifications were implemented in an efficient but robust form of multiple-input-change circuits using the synthesis tool *3D-gC* [17]. Each output is typically implemented with a single generalized C-element. As an example, the implementation of the MUL2 controller is shown in Fig. 14. This design style uses an efficient strategy for the insertion of state variables. In particular, state variables change concurrently with an output burst. Consequently, state variables are never in the critical path of the circuit and the latency of the circuit is very low. The circuits, however, have a fundamental-mode constraint. That is, after the output burst, the circuit requires time to stabilize, before any compulsory transition of a next input burst appears. A detailed timing analysis could be performed to identify the specific required settling time for each state transition [18]. Then, SPICE simulations of the controller and its environment could be performed to ensure that the environment is sufficiently slow in all cases. This involves a large number of time-consuming and tedious SPICE simulations. Fortunately, the environment is typically so slow (relative to the controller settling time) that the timing assumptions are satisfied with large safety margins. Therefore, we can approximate the settling time with a constant that is larger than the settling time of each of the individual state transitions without introducing many false negatives. We then can use our formal verification techniques to automatically validate these timing assumptions, as explained below.

C. Controller-datapath interface design and validation

We reduced control circuit overhead by hiding controller delay within datapath computation. This is achieved by simultaneously latching the result of one operation and asserting a communication signal to initiate the next operation. Thus much of the controller delay, if not all, can be overlapped by latch and multiplexor delays, as illustrated in Fig. 15. In this example, when ALU1 completes its evaluation, the controller asserts LA+ to latch the result in register A and A1M− to signal MUL1 to start the next operation using the value of A. Using this assumption, however, means that we must be careful to ensure that the inputs to MUL1 have stabilized before evaluation starts. Otherwise, the domino logic may malfunction due to accidental discharge. Simple static timing analysis and SPICE simulations were used to ensure that sufficient timing margins exist.

The example in Fig. 15 also illustrates a second timing assumption we needed. ALU1 controller asserts prech and LA concurrently after ALU1 has finished evaluating. The assumption here is that the data is loaded into register A before precharging of ALU1 changes the data (i.e., not violating data hold time). This assumption is safe because prech signal is heavily loaded. This type of timing assumptions was also verified using static timing analysis and SPICE simulation.

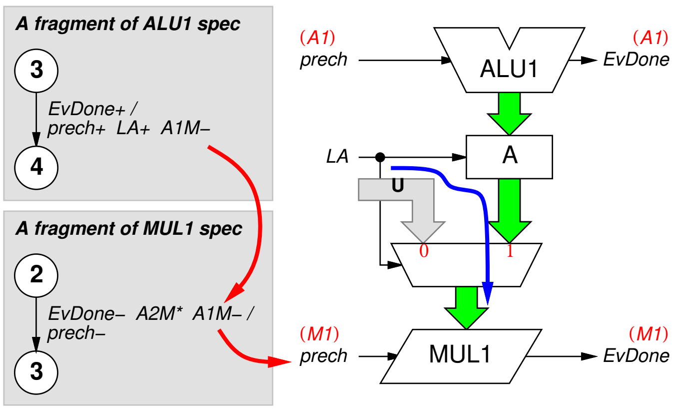

Fig. 15. Timing assumptions.

D. Inter-controller protocol design and validation

Our formal verification efforts were directed to insure that the controller designs are correct: i.e., that the interaction of controllers and datapaths generated no illegal transitions which may cause the circuit to fail. Specifically, we attempted to identify the existence of any transition that violates the specification of a controller or violates the fundamental mode constraint (i.e., transitions that occur before the controller is guaranteed to have stabilized). It is important to note that we used formal verification only as a design aid to check whether certain properties hold. We did not use it to guarantee correctness with respect to some model.

Our verification efforts validated that the final controller designs are correct. More importantly, it helped us quickly identify many bugs in preliminary versions of the design. We discuss below some interesting bugs. A more detailed description of our verification technique can be found in [8].

D.1 Verification models: discrete vs. continuous

We adopted a discrete time model and used symbolic model checking to verify our system. Discrete time models were selected over continuous time models for efficiency. Continuous time models, albeit more expressive, require the management of *region graphs*, which limits the size of the system that can be analyzed [19], [20]. Discrete time models do have the disadvantage that discretization of timer regions leads to large state spaces, referred to as the *state explosion problem*. However, *finite* state spaces (even very large ones) resulting from discretization can be represented efficiently using BDDs. In particular, symbolic model checking techniques using BDDs have been successfully applied to many practical designs, some of which have huge state spaces [7]. This motivated the application of symbolic model checking to our discrete time models.

In a discrete time model, one state transition of the system represents the passage of a *time step* amount of time. Smaller step sizes lead to more accurate representations of the design, reducing the chance of a false negative verification results. This higher resolution, however, leads to a larger state space that must be analyzed. To keep verification run-times manageable, we abstract the implementation details of each component by modeling its specification rather than its implementation. Moreover, we model the behavior of only those signals that are involved in inter-controller communication.

The discretization of delays, if not done carefully, can introduce false positive verification results [21], [22]. Other researchers have developed guidelines which ensure the absence of false positives in particular classes of systems [22], [23]. We are currently investigating whether this property is satisfied in our systems. Our primary goal, in any case, was to find bugs that are difficult to detect by simulation and not necessarily to guarantee 100% correctness.

D.2 Modeling burst-mode controllers

Fig. 16 illustrates how we model controllers, using the model of the MUL2 controller as an example. The model consists of four communicating FSMs: a deterministic FSM for tracking the XBM state (state FSM), a deterministic FSM for tracking

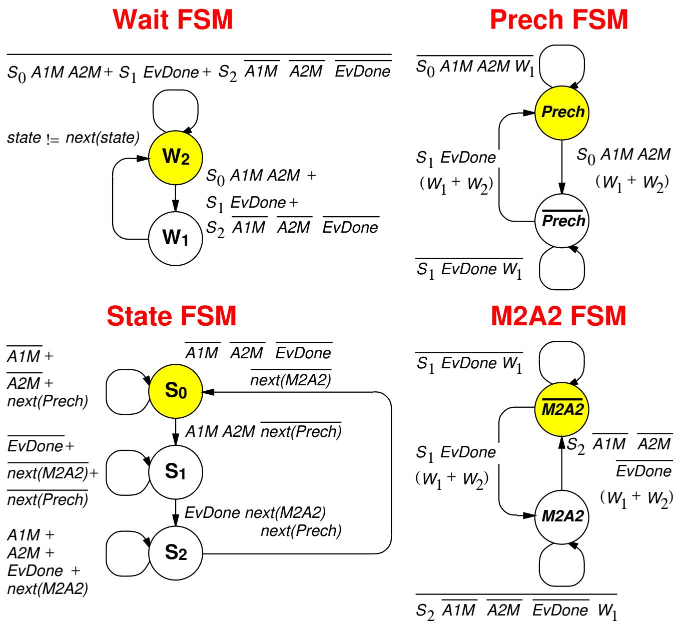

Fig. 16. Distributed non-deterministic FSM models for MUL2 controller using a time step of 0.5ns.

the time in each XBM state (wait FSM), and a non-deterministic FSM for each of the two output signals used in inter-controller communication. Note that we do not model the behavior of the latch control and function/MUX select output signals because they are not involved in inter-controller communication.

The state FSM changes state simultaneously with the latest occurring output transition after all specified input transitions occur (the state FSM examines the current and next state values of both output FSMs to determine when to change state). The number of states in the wait FSM equals the maximum quantized delay of the slowest output signal transition. Whenever a state transition occurs in the state FSM, the wait FSM resets to the highest numbered state simultaneously with the state transition in the state FSM (by examining both the current and next state values of the state FSM). When all terminating input transitions have occurred, the wait FSM starts to count down deterministically. The output FSMs model when the output signals can change, using variables from wait and state FSMs as inputs. For example, once A1M+ and A2M+ occur in state s0, Prech non-deterministically changes after one time unit *or* two. This non-determinism captures both the innate delay variations due to different logic implementations and the roundoff errors caused by quantization. For example, if the delay of an output is actually 1.5 time units, we model that the output shall change after 1 time unit or 2 time units.

Our models of the functional units are similar and not shown due to the space limitation.

D.3 Verification specifications

An XBM specification represents, as any asynchronous specification does, a contract between the environment and the circuit. As detailed in Section IV-B.1, it dictates what the circuit should do and what the environment is allowed to do. While the circuit's conformance to the specification is guaranteed by the synthesis methodology, the environment's compliance to the contract needs to be verified.

For every state of the XBM controller, these constraints together define a set of input transitions that are *illegal*. For example, when MUL2 controller is in state s0, it is illegal for A2M to fall (because A2M is expected to rise in this state). We write this specification in computation tree logic (CTL) as follows:

$$\texttt{SPEC AG}\ ((\texttt{state} = \texttt{s0})\ \&\ \texttt{A2M} \rightarrow \texttt{AX A2M}),$$

which states that an invariant for all states (AG) is that if the controller is in state s0 and A2M equals 1, every possible next state (AX) should have A2M equal to 1.

The complete set of illegal transitions, which are derived automatically from burst-mode specifications, is described in detail in [8].

D.4 Design bugs found using symbolic verification

Since ALU2 is responsible for implementing the loop test, we designed its controller to be the master controller. If the loop test passes, the controller asserts a communication signal that triggers the multiplier controllers to initiate their respective operations. The multiplier controllers, when the multipliers are done, assert communication signals that trigger the next operations, including ALU1 loop operations. Consequently, no direct communication between ALU1 and ALU2 controllers is needed; it is done through the MUL1 controller instead. Since ALU2 is also responsible for calculating y, its controller indicates to the environment when the entire computation is done.

This absence of direct communication between ALU1 and ALU2, however, created a timing problem in the initial design that our verification process uncovered. Initially, the extra C-element shown in Fig. 2 was not present and the ALU2 controller communicated directly with the external environment. In the initial design, ALU2 may assert done+ after the loop test fails and the final y calculation completes but while ALU1 is still computing U. If the environment is fast, start signal may be lowered before ALU1 expects it. Moreover, if ALU2 negates done quickly, then start+ may occur before ALU1 re-enters the reset state. This scenario violates the constraints of ALU1 controller. To fix this error, the C-element was introduced into the design with A1M and EndP (the original done signal) as its inputs and done as its output. The C-element ensures that done+ does not fire until the calculation of U is complete, signaled by A1M+.

Another design problem discovered by verification was caused by the broadcasting of A1M by ALU1 controller. The problem occurs in the special case in which the main computation loop is never entered because the loop test fails before entering the loop. In this case, ALU1 controller sends A1M+ to both multiplier controllers (as if entering the loop) but later sends A1M− when the external environment resets start. The multipliers during this time remain in their reset states in which A1M− is an illegal transition. Fortunately, upon examining the implementation of the multiplier controllers, we determined that this illegal transition would not cause the circuit to malfunction. Thus, we left the protocol unaltered.

D.5 Verification run-times

Initially, a simplified model of the chip was used for verification. Controllers were modeled as having a fixed delay of one unit of time, with output signal transitions all occurring simultaneously with the state transition. Bounds for ALU and multiplier precharge and evaluate delays were estimated. Even with this simple model, the verification identified several design problems, including the ones described above.

After laying out the design, more precise delay information was extracted from SPICE simulations; subsequently, a more accurate verification model was created. First, buffers that drive control signals over long wires, which were not in the earlier model, were added to the back-annotated model. For each component/signal transition, we conservatively modeled its delay by non-deterministically setting it to a value that lies within the tightest range of discrete time points that encompass the delay bounds obtained by SPICE.

The more detailed design was verified using a number of different time steps. For the resolution of 0.5 nanoseconds, the finest one used, no specifications were violated. Even when coarse resolution models were used, all but three of the specifications were verified correct. Some of these false negatives can be removed by locally increasing the accuracy of our models by, for example, modeling a known order of output events.

The verification was performed on a SPARC-5 with 32 Megabytes of memory using the symbolic model checker SMV [7]. The run-times of all the verification runs are shown in table V. The columns indicate the time step used, number of CTL specifications checked, the total number of BDD nodes allocated during verification, the number of reachable states, and the verification run time. The run-times for the earlier models were quite small, which facilitated a short design cycle. The run-times of the back-annotated designs were somewhat longer but still acceptable.

Run-time results for verifying DIFFEQ				
State transition time step	Total # specs	Total BDD nodes	Reached states	Run time (sec)
Initial models				
1.00 ns	58	12,494	17,212	21.5
1.00 ns	58	43,151	22,072	99
Backannotated models				
0.50 ns	51	457,643	3.82e+7	23,452
0.75 ns	51	255,504	10.0e+7	7,295
1.00 ns	51	210,293	2.78e+6	3,042
1.25 ns	51	162,089	9.5e+5	1,792

TABLE V
VERIFICATION RUN-TIMES USING DIFFERENT TIME STEPS.

E. Summary of the control design process and future work

Although our design process is tool-assisted, the tools do not constitute a fully integrated design methodology. In fact, our design process required significant manual effort. For example, the datapath and distributed controllers were designed manually without the assistance of a high-level synthesis tool. Hence, insuring correctness was more difficult. This suggests that achieving the same results for larger designs may be more challenging without the application of more powerful tools.

We did, however, synthesized the controllers and derived the verification models automatically from the 3D burst-mode controller specifications. In the future, it may be possible to derive correctly partitioned structural VHDL models of the system using a high-level synthesis tool, both for synthesis and for verification.

For the inter-controller communication design, we started out with a conservative speed-independent protocol and looked for optimization opportunities based on what we knew about the datapath/environment. After modifying the protocol based on timing assumptions, we verified the new design using SMV. The optimization and verification were repeated until we were satisfied with the estimated performance of the design.

For this design, our verification efforts were limited to the inter-controller protocol design. Thus a simple verification technique using a "monolithic transition relation" was sufficient. To verify larger designs, we can employ more sophisticated verification approaches using "partitioned transition relations" [7]. Moreover, a more sophisticated timing analysis tool, such as [18], can be used to verify the timing assumptions at the controller-datapath interface.

V. RESULTS

We completed the design, verification, simulation, and testing of the differential equation solver. The chip (die photo shown in Fig. 17) was fabricated in $0.5\mu\mathrm{m}$ HP CMOS14TB triple-metal process through MOSIS. The chip area (including pads) is $13058\lambda \times 13058\lambda$ ($3917.4\mu\mathrm{m} \times 3917.4\mu\mathrm{m}$); the core area is $10550\lambda \times 10145\lambda$ ($3165\mu\mathrm{m} \times 3043.5\mu\mathrm{m} = 9.63\mathrm{mm}^2$). We conducted a comprehensive set of tests, with some runs exceeding 1,500,000 iterations. Table VI lists both the simulation and test results of an example run.

	Testing	Simulation		
Iter	22°C 3.3V N68MAR	22°C 3.3V N68MAR	50°C 3.3V Typical	100°C 3.0V Slow
1	36.1ns	41.4ns	46.2ns	66.2ns
2	37.6ns	43.8ns	48.9ns	70.7ns
3	37.3ns	43.3ns	48.3ns	69.4ns
4	36.6ns	42.8ns	47.5ns	68.3ns
5	36.0ns	41.5ns	46.4ns	67.7ns
6	35.4ns	41.4ns	46.2ns	66.3ns
7	34.8ns	40.3ns	44.9ns	64.4ns
8	34.9ns	40.5ns	45.0ns	64.9ns
9	34.9ns	40.2ns	44.7ns	64.4ns
10	34.7ns	39.9ns	44.1ns	64.1ns
Average	35.6ns	41.2ns	46.2ns	66.6ns

TABLE VI
EXPERIMENTAL RESULTS OF AN EXAMPLE RUN USING THE INITIAL VALUES OF $x = 1$, $dx = 1$, $a = 11$, $u = 3$, AND $y = 3$.

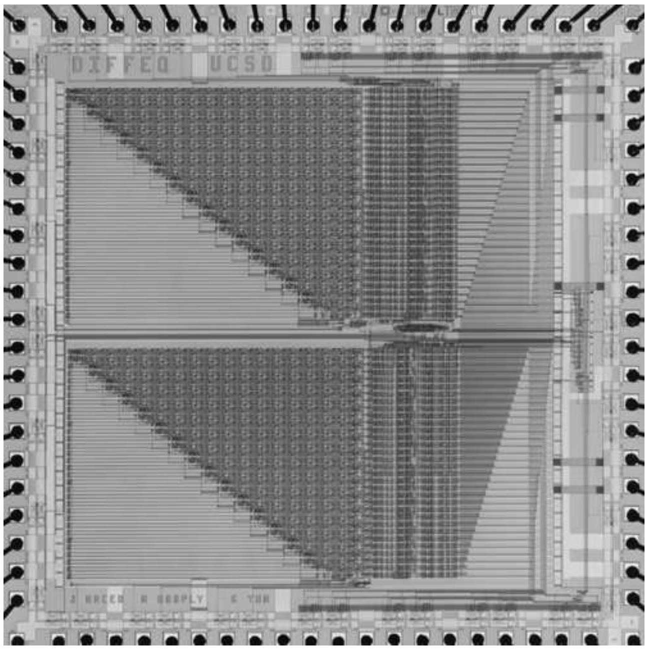

Fig. 17. DIFFEQ die photo.

The average delay per iteration of the algorithm is 35.6ns, when tested at 22°C with 3.3V power supply. As discussed earlier, the simulated delay of 41.2ns is somewhat longer than the test result. The simulated average delay per iteration under the worst-case condition is 66.6ns; the worst-case over all the data-dependent delays we simulated is 70.7ns. The critical path delay calculated by accumulating only the datapath component delays as shown in Fig. 18 is 61ns. This delay would correspond to the delay of a *comparable* synchronous implementation, which has no control overhead, no clock skew, no register setup time, and no cycle-to-cycle jitter, but uses the same class of adders, multipliers, multiplexors, and registers. Since our design's simulated average iteration time under typical operating condition is 41.2ns, it is at least 48% faster, when operating under typical operating condition, than comparable synchronous designs (designed for the worst case, i.e., 100°C, 3V, and slow process corner).

In order to validate the robustness of our design, we conducted a set of tests by varying the operating temperature and the power supply voltage. Fig. 19 illustrates the average iteration time for 24 different operating conditions. We conducted our tests at three different operating temperature: −30°C, 22°C, and 55°C. We could not perform the worst-case temperature testing at 100°C, because we were unable to raise the operating temperature beyond 55°C with our simple test setup. We varied the power supply voltage from 2.2V to 5.0V in 0.2V increments for each operating temperature. As expected, we observed the increase in iteration time as the operating temperature was raised. Similarly, the increase in the iteration time was observed as the power supply voltage was reduced. Our chips operated correctly down to 1.85V at 55°C, 1.89V at 22°C, and 2.07V at −30°C. Although the plot does not show, we observed that our chips operate correctly up to a power supply voltage of 7.0V. Since HP CMOS14TB process is optimized for operating at 3.3V, our design can tolerate a 40% drop or more than a 100% increase in the power supply voltage. This result proves that our design is extremely robust and, more importantly, designs with safe timing assumptions can lead to higher performance without sacrificing robustness.

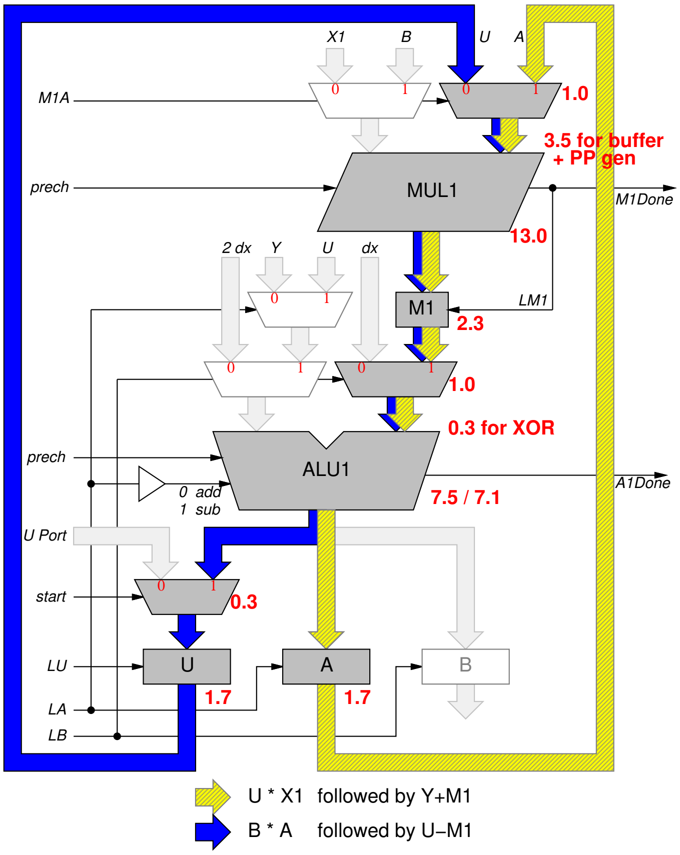

Fig. 18. Critical Path.

We use *control overhead per iteration* to estimate the effectiveness of our overhead reduction techniques:

$$\mathsf{OH}_{\mathrm{iter}} = \frac{t_{\mathrm{iter}} - \hat{t}_{\mathrm{iter}}}{\hat{t}_{\mathrm{iter}}},$$

where $\mathsf{OH}_{\mathrm{iter}}$ is the control overhead per iteration, t_{iter} the actual iteration time, and $\hat{t}_{\mathrm{iter}}$ the iteration time of an "ideal" circuit, i.e., the sum of all datapath element delays incurred in the critical path of an iteration. For our chip, the average control overhead is *estimated* to be 16%.

VI. Conclusion

We designed a high-performance (48% faster than comparable synchronous designs), low-control-overhead (16% control overhead) asynchronous differential equation solver. In the course of the design, we developed novel techniques for reducing completion sensing overhead for self-timed datapath building blocks. When safe, we used timing assumptions to hide control overhead. Some of the trickier problems at the protocol level were debugged using a symbolic model checking based formal verification tool.

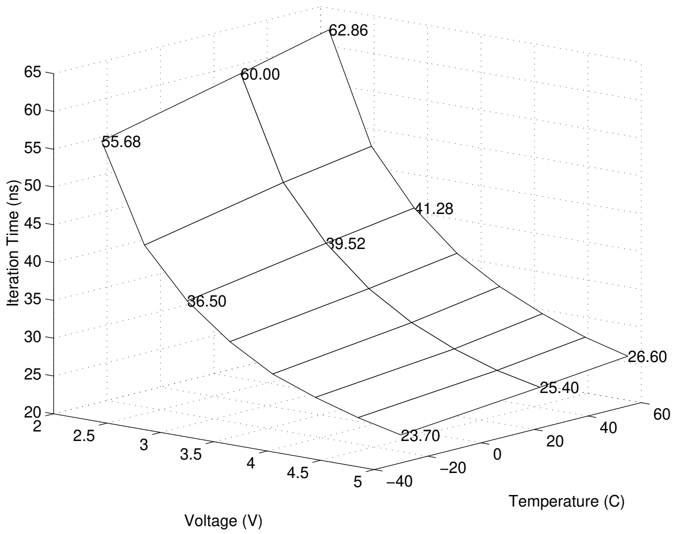

Fig. 19. Iteration time test results with varying operating temperature and power supply voltage.

The design process consisted of an iterative design and verification loop. At the earlier design stages, simpler models of XBM controllers were used, which enabled detection of many bugs quickly. In the final stages of the design, more accurate models which incorporate backannotated delays were verified in a reasonable amount of time, providing higher confidence in the design.

The fabricated chips perform as simulated, albeit somewhat faster. The test results show that our design is very robust with respect to variations in operating temperature and power supply voltage. We conclude that exploiting design margins using safe timing assumptions can lead to robust, high-performance circuits. Moreover, we argue that our techniques to achieve high performance by reducing control overhead while maintaining robustness are generally applicable. In synchronous designs, clock periods are set to fit the critical path delay plus some safety margin; however, a considerable amount of extra safety margin exists in non-critical paths. Our chip, on the other hand, demonstrates that event-driven control can lead to higher performance by reducing excess dead time, i.e., by reducing the safety margin of significantly more paths to the level used in the critical paths. In addition, techniques to hide control overhead with the delay through MUXes are generally applicable to MUX-based architectures and can be extended to bus-based architectures.

Acknowledgment

We would like to thank Jim Garside at University of Manchester for providing us with an ARM architectural simulator and for many insightful discussions.

References

[1] M. E. Dean, *STRiP: A Self-Timed RISC Processor Architecture*, Ph.D. thesis, Stanford University, 1992.

[2] T. E. Williams and M. A. Horowitz, "A zero-overhead self-timed 160ns 54b CMOS divider," *IEEE Journal of Solid-State Circuits*, vol. 26, no. 11, pp. 1651–1661, Nov. 1991.

[3] K. van Berkel, *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, vol. 5 of *International Series on Parallel Computation*, Cambridge University Press, 1993.

[4] S. B. Furber, "Computing without clocks: Micropipelining the ARM processor," in *Asynchronous Digital Circuit Design*, Graham Birtwistle and Al Davis, Eds. 1995, Workshops in Computing, pp. 211–262, Springer-Verlag.

[5] A. J. Martin, "Programming in VLSI: From communicating processes to delay-insensitive circuits," in *Developments in Concurrency and Communication*, C. A. R. Hoare, Ed. 1990, UT Year of Programming Series, pp. 1–64, Addison-Wesley.

[6] K. Y. Yun, *Synthesis of Asynchronous Controllers for Heterogeneous Systems*, Ph.D. thesis, Stanford University, Aug. 1994.

[7] K. L. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[8] V. Vakilotojar and P. A. Beerel, "RTL verification of timed asynchronous and heterogeneous systems using symbolic model checking," in *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 1997.

[9] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.

[10] B. E. Briley, "Some new results on average worst carry," *IEEE Transactions on Computers*, vol. C-22, no. 5, pp. 459–463, 1973.

[11] J. D. Garside, "A CMOS VLSI implementation of an asynchronous ALU," in *Asynchronous Design Methodologies*, S. Furber and M. Edwards, Eds. 1993, vol. A-28 of *IFIP Transactions*, pp. 181–207, Elsevier Science Publishers.

[12] D. J. Kinnement, "An evaluation of asynchronous addition," *IEEE Transactions on VLSI Systems*, vol. 4, no. 1, pp. 137–140, Mar. 1996.

[13] H. Dhanesha and A. Albicki, "Self-timed adder with pipelined output," in *Proceedings of the Midwest Symposium on Circuits and Systems*, 1993, pp. 855–858.

[14] S. M. Nowick, K. Y. Yun, and P. A. Beerel, "Speculative completion for the design of high-performance asynchronous dynamic adders," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*. Apr. 1997, pp. 210–223, IEEE Computer Society Press.

[15] C. Lemonds, "A high throughput 16 by 16 bit multiplier for DSP cores," in *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 1996, pp. 477–480.

[16] P. Kudva, G. Gopalakrishnan, and H. Jacobson, "A technique for synthesizing distributed burst-mode circuits," in *Proc. ACM/IEEE Design Automation Conference*, 1996, pp. 77–82.

[17] K. Y. Yun, "Automatic synthesis of extended burst-mode circuits using generalized C-elements," in *Proc. European Design Automation Conference (EURO-DAC)*, Sept. 1996, pp. 290–295.

[18] S. Chakraborty, D. L. Dill, K. Y. Yun, and K. Chang, "Timing analysis for extended burst-mode circuits," in *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Apr. 1997, pp. 101–111.

[19] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[20] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine, "Symbolic model checking for real-time systems," *Information and Computation*, vol. 111, pp. 193–344, 1994.

[21] J. A. Brzozowski and C.-J. H. Seger, "Advances in asynchronous circuit theory; part II: Bounded inertial delay models, MOS circuit design techniques," *Bull. EATCS*, , no. 43, pp. 199–263, Feb. 1991.

[22] J. R. Burch, *Trace Algebra for Automatic Verification of Real-Time Concurrent Systems*, Ph.D. thesis, Carnegie Mellon University, 1992.

[23] T. G. Rokicki, *Representing and Modeling Circuits*, Ph.D. thesis, Stanford University, 1993.