A Scalable Counterflow-Pipelined Asynchronous Radix-4 Booth Multiplier

Justin Hensley, Anselmo Lastra and Montek Singh
Department of Computer Science
University of North Carolina, Chapel Hill, NC 27514, USA
{hensley,lastra,montek}@cs.unc.edu

Abstract

This paper introduces an asynchronous radix-4 Booth multiplier architecture, which is scalable to arbitrary operand lengths while maintaining a constant cycle time per Booth iteration. Combined with the area efficiency of iterative multiplication, our multiplier architecture therefore is suitable for a range of emerging mobile applications, from handheld 3D graphics computation, which typically requires 32-bit multiplication, to cryptography, which can involve 1024-bit operands.

The multiplier architecture has several novel features, including: (i) a novel counterflow organization, in which the data bits flow in one direction and the Booth commands piggyback on the acknowledgments flowing in the opposite direction, (ii) overlapped execution of multiple iterations of the Booth algorithm, and (iii) design modularity and bit-level pipelining, which enable the multiplier to be scaled to arbitrary operand widths without requiring gate resizing or cycle time overheads.

Spice simulations in a 0.18μm TSMC process at 1.8V, indicate promising performance: the multiplier takes 640–650ps per Booth iteration, regardless of the operand widths, thereby demonstrating the scalability of our approach. For 16-bit operands, this performance corresponds to nearly 200 Mega ops/sec throughput. Furthermore, the multiplier is fully functional at reduced supply voltages (e.g., 1.5V and 1.0V), and thus capable of dynamically trading off performance for energy efficiency.

1. Introduction

This paper introduces an asynchronous radix-4 Booth multiplier architecture that is especially targeted to mobile devices, with the key objectives of high energy efficiency, small chip area, and design reusability. Several emerging consumer electronic applications are likely to increasingly depend on the following capabilities: 3D graphics computation (e.g., cell phones [9], handheld game consoles), digital signal processing (e.g., portable audio players), and cryptographic processing (e.g., smartcards). In each of these application domains, multiplication is a fundamental operation.

Our multiplier is of the iterative radix-4 Booth type, implemented using asynchronous circuits. An iterative implementation was chosen, as opposed to a combinational array type, for higher area efficiency. A Booth implementation can uniformly handle signed as well as unsigned operands. However, a minor modification to the controller can easily transform our design into a simple (i.e., non-Booth) iterative multiplier. Finally, an asynchronous circuit style was chosen because of its high energy efficiency [2, 13, 21]. In particular, asynchronous circuits have the advantage of demand-driven switching activity, effectively providing the benefits of fine-grain clock gating for free. In addition, the greater robustness to timing variations allows an asynchronous circuit to more easily exploit voltage scaling as a technique to further conserve energy.

The multiplier has several interesting features:

- **Counterflow Organization:** A novel multiplier organization is introduced, in which the data bits flow in one direction, and the Booth commands are piggybacked on the acknowledgments flowing in the opposite direction. Our counterflow organization has significant advantages compared with the counterflow pipeline of Sproull et al. [20] because it eliminates the need for complex synchronization and arbitration the latter requires between two distinct data streams. This feature allows shorter critical paths, and therefore higher operating speed.

- **Merged Arithmetic/Shift Unit:** An architectural optimization is introduced that merges the arithmetic operations and the shift operation into the same function unit, thereby obtaining significant improvement in area, energy and speed.

- **Overlapped Execution:** The entire design is pipelined at the bit-level, which allows overlapped execution of

---

*This work was supported by NSF Award CCF-0306478, an ATI Graduate Student Fellowship, and an IBM Faculty Development Award.
multiple iterations of the Booth algorithm, including across successive multiplications. As a result, both the cycle time per Booth iteration, as well as the overall cycle time per multiplication are significantly improved.

- **Modular Design:** The design is quite modular, which allows the implementation to be scaled to arbitrary operand widths without the need for gate resizing, and without incurring any overhead on iteration time. In particular, Booth commands are relayed from one stage to the next instead of being broadcast across the entire width of the operand, thereby allowing for a constant Booth iteration time regardless of operand widths. Further, a sentinel-based approach is used to determine the termination condition (i.e., all multiplier bits have been consumed), instead of using a counter. As a result, the Booth controller implementation becomes independent of operand widths.

- **Precision-Energy Trade-Off:** Finally, the architecture can be easily modified to allow dynamic specification of operand widths, i.e., successive operations of a given multiplier implementation could operate upon different word lengths. This feature could potentially facilitate a dynamic trade-off between computation precision and energy consumption.

The performance of the multiplier has been quantified through Spice simulations in a 0.18μm TSMC process, at 1.8V nominal supply voltage. The design takes 640–650ps per Booth iteration, regardless of the operand widths, thereby demonstrating the scalability of our approach. The overall computation time varies linearly with the width of the operand: e.g., about 2.6 ns for 8-bit operands, and approximately 10.4ns for 32-bit operands. The energy consumed per multiplication varies as approximately the square of the operand width, as expected: 0.11nJ for 8-bit and 1.42nJ for 32-bit multiplications. Finally, simulations performed at reduced supply voltages of 1.5V and 1.0V demonstrated that the multiplier operates correctly at lower voltages, and is able to trade off some performance for even higher energy efficiency.

Compared with several asynchronous Booth multiplier designs reported recently [4], this iteration time represents a nearly two-fold (1.95x) increase in throughput over the fastest radix-4 Booth multiplier reported in [4]. Although our multiplier consumes more energy per multiplication than the one in [4], it actually has a superior energy-delay² (E×T²) product: 9.65 nJ-ns² compared with 14.8 nJ-ns².

The radix-4 multiplier architecture presented in this paper builds upon our recent radix-2 design introduced in [8]. While preliminary versions of some of the concepts presented in this paper—e.g., the novel counterflow-pipelining approach—were first introduced in [8], they are significantly refined and improved here. In addition, the capability of dynamic precision-energy trade-off is developed in this paper. Further, the emphasis in [8] was on greater robustness: a dual-rail datapath with robust completion detection was used. In this paper, a bundled datapath approach is used instead, as it leads to smaller area, faster operation, and reduced power consumption at the cost of some increase in design effort introduced by the bundling timing constraint. Experimental results indicate that the new multiplier outperforms the design of [8]: 3.3x higher throughput is obtained in the new design while consuming 2.2x lower energy consumption.

The remainder of this paper is organized as follows. Section 2 presents background on a particular asynchronous pipelined circuit style (the high-capacity style, HC [18, 19]) which is used to implement the multiplier, and also briefly reviews certain composite energy-efficiency metrics. Next, related work is summarized in Section 3. Section 4 presents in detail the design and operation of the new multiplier. Section 5 presents the results of simulations. Finally, Section 6 outlines an approach to make our multiplier architecture capable of handling dynamically variable operand widths, to allow precision-energy trade-offs, and Section 7 provides conclusions.

2. Background

This section first reviews the high-capacity asynchronous pipelined circuit style, which forms the basis of our multiplier implementation. Then energy-performance trade-off and metrics are briefly discussed.

2.1. High-Capacity Asynchronous Pipelines (HC)

The implementation style used for the multiplier design is based on the high-capacity (HC) asynchronous pipeline style [18, 19]. The HC style is reviewed here; Section 4.3.1 will present the enhancements to HC that were carried out to meet the design objectives of this paper.

2.1.1. Motivation. The HC style was chosen because of its area— as well as energy efficiency. In particular, the HC datapath uses dynamic logic and is latchless, i.e., no explicit storage elements are used between pipeline stages. Instead, the dynamic function blocks themselves provide implicit storage capability through use of staticizers and by means of careful sequencing of control. The absence of latches translates into significant area and energy savings. Further, unlike some other asynchronous latchless dynamic styles (e.g., PSo [23]), HC does not require intervening “bubble” or “spacer” stages between data items, thereby keeping energy consumption and area low.

2.1.2. Overview. The key idea in the HC approach is one of decoupled control: the pull-up and pull-down of the dynamic gates are made separately controllable, as shown in...
Figure 1(b). Therefore, the precharge and evaluate controls can both be simultaneously de-asserted, allowing the gate to enter a special “isolate phase”—between evaluation and precharge—in which its output is protected from further input changes.

2.1.3. Structure. Figure 1(a) shows a block diagram of a high-capacity pipeline. Each stage consists of three components: function block, a completion generator and a stage controller. The function block is implemented using dynamic logic. It alternately evaluates and precharges, thereby alternately producing data tokens and reset spacers for the next stage. The completion generator indicates completion of the stage’s evaluation or precharge. The third component, the stage controller, generates separate \( pc \) and \( eval \) signals which control the function block and the completion generator. Figure 1(b) shows one gate of a function block in a pipeline stage.

The bundled data scheme [16, 3] is used to implement the asynchronous datapath. A control signal, \( Req \), indicates arrival of new inputs to a stage when it is asserted; precharge of inputs is indicated when \( Req \) is de-asserted. For correct operation, a suitable matched delay must be inserted to ensure that \( Req \) arrives after the data inputs.

The completion generator is implemented using an asymmetric C-element, \( aC \) [6]. The \( aC \)’s output, \( Done \), is set when the stage has entered its evaluate phase (\( eval \) is high), and the previous stage has supplied valid data input (completion signal \( Req \) of previous stage is high). \( Done \) is reset simply when the stage precharges (\( pc \) asserted low).

The stage controller produces the control signals for the function block and the completion generator. It receives three inputs—the request from the previous stage, the delayed \( Done \) of the current stage, and the acknowledge from the next stage—and produces the two decoupled control signals, \( pc \) and \( eval \).

2.1.4. Implementation. Figure 1(a) shows a complete implementation of the stage controller. The two outputs—\( pc \) and \( eval \)—and an internal state variable, \( ok2pc \), are each implemented using a single gate. The 3-input NAND gate asserts \( pc \) when three conditions are met: the current stage has completed evaluation, the next stage has also completed its evaluation (indicated by a high ack), and these two stages contain the same data token (indicated by a high \( ok2pc \)). The state variable \( ok2pc \) is implemented using an asymmetric C-element as follows: \( ok2pc \) is set when \( Req_\text{in} \) is asserted high and \( Done \) is de-asserted low; \( ok2pc \) is reset when \( Req_\text{in} \) is de-asserted low.

An important feature of the HC protocol is that transitions on the \( ok2pc \) signal are designed to be off the critical path. In particular, while in Figure 1(a), \( ok2pc \) appears to add an extra gate delay to the control path to \( pc \), this is not the case: the pipeline protocol allows \( ok2pc \) to be set in “background mode,” so that \( ok2pc \) is typically set before \( Ack_\text{in} \) gets asserted. As a result, the critical path to \( pc \) has typically only one gate delay: from input \( Ack_\text{in} \) through the 3-input NAND gate, NAND3, to the output \( pc \).
2.1.5. Operation. An HC pipeline stage simply cycles through three phases. After it completes its evaluate phase, it enters an isolate phase and subsequently a precharge phase. As soon as precharge is complete, it re-enters the evaluate phase again, completing the cycle.

The introduction of the isolate phase is the key to the new protocol. Once a stage finishes evaluation, it immediately isolates itself from its inputs by a self-resetting operation regardless of whether this stage is allowed to enter its precharge phase. As a result, the previous stage cannot only precharge, but even safely evaluate the next data token, since the current stage will remain isolated. There are two benefits of this protocol: (i) higher throughput, since a stage \( N \) can evaluate the next data item even before stage \( N + 1 \) has begun to precharge; and (ii) higher capacity for the same reason, since adjacent pipeline stages are now capable of simultaneously holding distinct data tokens, without requiring separation by spacers.

The \( ok2pc \) state variable is critical to disambiguating between two pipeline states: one in which pipeline stages \( N \) and \( N + 1 \) both contain valid data corresponding to the same data token, and the other in which \( N \) and \( N + 1 \) contain valid data corresponding to distinct but consecutive data items. In the former case, the protocol ensures that \( ok2pc \) will be asserted, thereby enabling precharge of stage \( N \). In the latter scenario, \( ok2pc \)'s value will be unset, thereby correctly preventing precharge of stage \( N \).

2.1.6. Performance. If the evaluation and precharge times for a stage are denoted by \( t_{\text{Eval}} \) and \( t_{\text{Prech}} \), and the delay through the NAND and \( \& C \) elements by \( t_{\text{NAND}} \) and \( t_{\text{AC}} \), respectively, then the analytical cycle time of the pipeline is given by:

\[
T_{\text{HC}} = t_{\text{Eval}} + t_{\text{Prech}} + t_{\text{AC}} + t_{\text{NAND}} + t_{\text{INV}}
\]

Also, a stage's latency is simply its evaluation delay:

\[
L_{\text{HC}} = t_{\text{Eval}}
\]

2.2. Energy–Performance Trade-Off, and the \( E_T \) and \( E^2 \) Metrics

The freedom from stringent, hard-to-satisfy timing assumptions in asynchronous implementations greatly facilitates a trade-off between performance and energy consumption, through voltage scaling. While the system throughput drops approximately linearly with voltage—within certain voltage limits—the drop in power consumption and radiated noise is more dramatic: they decrease as the square of the voltage.

Two composite energy-performance metrics have been proposed for a fair comparison between different implementations of the same system: the energy-delay product, \( E_T \), and the energy-delay\(^2 \) product, \( E^2 \). Here, \( E \) refers to the energy consumed per operation, and \( \tau \) is the execution time per operation (or the cycle time, which is inverse of the throughput). The energy-delay product, normalized with respect to \( \lambda^2 \) (\( \lambda \) = feature size), is fairly invariant across fabrication processes [7]. The second metric, \( E^2 \), on the other hand, is invariant to moderate voltage scaling [14, 15]. Thus, depending upon the particular comparison being made, one or the other metric can be used.

3. Related Work: Asynchronous Multipliers

Several asynchronous multipliers have been reported in the literature, both array as well as iterative. It was shown in [22] that array multipliers not only have lower latencies, but may also have better energy efficiency than iterative multipliers. [1] presents a novel design of a bundled-data array concurrent multiply-accumulator unit, which reduces power consumption by eliminating unnecessary evaluation of certain partial products (i.e., those corresponding to a zero value for the multiplier bit). By taking advantage of data-dependent evaluation times, their design was able to improve average throughput by 14% when compared to an equivalent synchronous design.

A number of iterative multipliers have been introduced recently. In [11], an area-efficient low-power multiplier is described for use in a hearing aid. [10] targets both array and iterative multiplication, and is able to show a 20% improvement for a bundled-data self-timed multiplier compared to an equivalent synchronous one.

Several iterative implementations increase the operating speed by processing more than one multiplier bit per iteration. For example, [12] reported a 32x32-bit iterative modified-Booth multiplier, using a new 4-phase asynchronous handshaking scheme. Their design uses two CSA adders and two 2-bit Booth encoders to reduce the number of iterations by half. A high-throughput iterative multiplier is presented in [17], which produces the product in \( n/4 \) iterations; however, it takes more than twice the area of a shift-and-add iterative multiplier.

Recently, [5] has proposed several multiplier implementations, including both the original radix-2 Booth algorithm, as well as the radix-4 Booth algorithm. The key novelty of their radix-2 implementation is that it is able to exploit data dependency to speed up its operation: it skips over arbitrarily long runs of ones and zeros in the multiplier operand, instead of performing sequential single shifts. However, due to the added complexity, its iteration time is actually longer, and therefore it exhibits performance advantages only for certain corner cases. Their radix-4 implementation does not exploit data dependency, but still obtains fairly good operating speed: 1.2 ns cycle time per Booth iteration, in 0.18\( \mu \)m technology.

Finally, we have recently introduced a radix-2 Booth multiplier [8], which forms the starting point for the ar-
chitecture introduced in this paper. Preliminary versions of some of the concepts presented in this paper—e.g., the novel counterflow-pipelining approach—were first introduced in [8], but they are significantly refined and improved here. Further, the emphasis in [8] was on greater robustness: a dual-rail datapath with robust completion detection was used. In this paper, a bundled datapath approach is used instead, as it leads to smaller area, faster operation, and reduced power consumption at the cost of some increase in design effort introduced by the bundling timing constraint.

In this paper, the principal comparison of our multiplier will be to the radix-4 implementation of [5], and the radix-2 implementation of [8].

4. Multiplier Design

This section presents the new multiplier design. Section 4.1 presents the overall architecture, highlighting a novel counterflow organization. Next, Section 4.2 discusses the operation of the multiplier, which performs overlapped execution of multiple Booth iterations. Finally, Section 4.3 presents some of the key details of the implementation, including a novel asynchronous pipeline handshake style that builds upon the HC style of Section 2.1.

4.1. Architecture

4.1.1. Overview. Figure 2 shows the overall architecture of our Booth multiplier. The new multiplier has a novel counterflow organization: the Booth commands (i.e., add, add 2x, subtract, subtract 2x, or shift) are bit-level pipelined, i.e., relayed from one bit to another. In contrast, existing iterative organizations involve a broadcast of the command to all bits, which makes those designs less scalable than ours. Another feature of our design is the folding together of the ALU and shifter units, resulting in a simple linear pipeline with area and energy advantages. Finally, our multiplier allows overlapped execution of multiple iterations of the Booth algorithm, including across successive multiplications.

The design of the multiplier is now presented in detail.

4.1.2. Novel Counterflow Organization. The multiplier has a counterflow organization: data and commands flow in opposite directions. In particular, data bits flow from left to right in the pipeline, whereas commands generated by the Booth controller (i.e., add, add 2x, subtract, subtract 2x, or shift) flow from right to left. Wherever carry bits are generated, as a result of an add or subtract command, they are embedded in, and considered part of, the command itself and relayed to the stage on the left.

Our counterflow approach allows data and command to transform each other when they interact. In particular, when data is evaluated by a pipeline stage, the actual operation performed on it depends not only on the functional implementation of the stage, but also on the command received by the stage. Similarly, our approach permits a command to be arbitrarily transformed by the data it interacts with, before it is relayed to the left neighbor. In our particular implementation of the Booth multiplier, this command transform-
mation applies to the carry bits which are embedded within the command: the carry bits are replaced by the new carry bits that are generated when the Booth command interacts with the incoming data.

A key novelty of this architecture is that it performs overlapped execution of multiple iterations of the Booth algorithm. In particular, each command that is inserted by the Booth controller into the right end of the counterflow pipeline effectively performs one iteration of the Booth algorithm as it flows from right to left through the pipeline. However, the bit-level pipelined architecture enables multiple commands to be simultaneously “in flight,” thereby effectively allowing multiple iterations of the algorithm to be overlapped. For instance, the lower significant bits of the accumulated result, near the right end of the pipeline, can commence a subsequent Booth iteration by interacting with a later command, while the higher significant bits are still waiting to complete earlier commands.

Comparison with Counterflow Pipeline of Sproull et al. It must be emphasized that our counterflow pipeline organization is quite different from another counterflow organization proposed by Sproull et al. [20]. In particular, the architecture in [20] uses two distinct pipelines to carry two different data streams in opposite directions, and introduces interlocks to allow the two streams to interact. A drawback of their approach is that arbiters are required between the two pipelines to ensure that corresponding data packets in the two streams do not “skip past” each other, leading to significant implementation complexity and also non-determinism in the system’s operation. In contrast, our approach simply “piggybacks” commands on top of the acknowledge signals already required for asynchronous handshaking, and therefore does not suffer from these drawbacks.

4.1.3. Architectural Optimization: Folding Arithmetic Unit into Shifter. An architectural optimization was used to obtain significant improvement in area, speed, as well as power consumption: the arithmetic operations (i.e., add and subtract) and the shift operation were merged into the same function unit.

Figure 3 shows the block diagram of a folded ALU/shift stage. Each such stage has three input sources and two output destinations. The first input stream, representing the current accumulated result at that bit position (labeled Z), enters the block from the left. The second input is applied to the top of the stage, and represents the corresponding constant multiplicand bit (labeled B). The third stream represents the Booth commands, along with embedded input carry bits (labeled Cik), and is accepted from the right. As a result of the command, the stage generates the new accumulated bit, and communicates it to the right, effectively causing a shift operation as well. The stage also produces a second output, which consists of the Booth command that was just executed, along with the new value of the carry bit (labeled Cout); this second result is communicated to the left.

The operation of the new ALU/shift stage is quite simple. Whenever it receives a Booth command from its right neighbor, and data from its left neighbor, it performs the command on the data, and transfers the result to its right neighbor. Thus, every command effectively causes a shift operation as well. If the command processed was a shift command, then the data is simply passed along unmodified; otherwise, for add and subtract commands, the multiplicand (B) and input carry (Cik) bits are combined with the data (Z) bit to generate the results.

Our folding in of the ALU into the shifter has several advantages: (i) lower area, because no explicit shifter unit is required; (ii) faster operation, because the results of an arithmetic operation are immediately available for a subsequent arithmetic operation (in the next stage), thereby allowing shorter iteration times; and (iii) better energy efficiency because overall there is less movement of data, and hence fewer transistors are switched.

4.1.4. Command Representation. The commands that are generated by the Booth controller are represented using a 1-of-6 (i.e., one-hot) encoding that is delay-insensitive. The six representable commands are initialize, add, add 2x, subtract, subtract 2x, and shift. Besides being delay-insensitive, the encoding has two advantages: (i) no decoding circuitry required, and (ii) good energy efficiency because each command causes switching activity on only one wire.

When a command reaches the left half of the multiplier pipeline (i.e., the rightmost ALU/shift stage), the 1-of-6 encoding for the command is augmented by a 1-of-2 code to allow a carry bit to be also carried within the command (see Figure 3), making each command an 8-bit value (labeled “Meta-command” in the figure).

4.1.5. Data Representation. Each data bit is also represented using a 1-of-n encoding. In the left half of the pipeline, a 1-of-2 (or “dual-rail”) encoding is used: one wire represents the logic “1” value, another wire represents the logic “0” value. In the right half of the pipeline,
however, a 1-of-3 encoding is necessary because an additional value must be represented: a sentinel value, which encodes a terminating condition. When a stage contains the sentinel, stages to the right hold the remaining bits of the multiplier. Once the sentinel reaches the controller, the controller senses the terminating condition, and stops issuing further Booth commands. Subsequently, once the results of the multiplication have been consumed, the multiplier is ready to process its next set of operands.

There are two significant benefits of using a sentinel-based encoding: (i) the design of the Booth controller is greatly simplified, i.e., no counter is required, and (ii) the length of the multiplier can be dynamically specified by providing the sentinel in the appropriate bit position. While our current implementation uses a fixed position for the sentinel, Section 6 outlines how our implementation can be easily modified to provide the ability to dynamically specify the length of both the multiplicand and the multiplier, independent of each other. This approach can potentially facilitate a dynamic trade-off between energy consumption and the precision of computation.

4.2. Operation

Computation proceeds in three phases: initialization, execution, and termination.

4.2.1. Initialization. The controller starts computation by issuing the initialize command. This command effectively copies the operand into the right half of the multiplier pipeline (see Figure 2). In particular, when a stage in the right half of the pipeline (i.e., a load/shift stage) receives the initialize command, it loads the multiplier value at that bit position from the A input on its top, and passes the same command to its left neighbor.

When the initialize command reaches the sentinel stage, it causes that stage’s output to get initialized to the sentinel value. This value marks the position immediately to the left of the most significant bit of the multiplier, and represents a termination flag that the Booth controller can sense.

Finally, as the initialize command reaches each ALU/shift stage, it causes both the sum and the carry output of the stage to be initialized to the logic “0” value, effectively clearing the contents of the accumulator so that computation may begin.

Actually, the initialize command serves another purpose that was omitted from the discussion here: to indicate completion of the previous computation. This function is explained in Section 4.2.4, when initialization for the next round of computation is discussed.

4.2.2. Execution. After the initialization, the controller generates successive Booth commands: add, add 2x, subtract, subtract 2x, or shift. Each command corresponds to one iteration of the Booth algorithm. Since the operations are pipelined, multiple commands could be flowing through the pipeline, effectively causing multiple iterations of the Booth algorithm to be executed concurrently.

The load/shift stages in the right half of the multiplier pipeline interpret each of these commands as a shift command, and cause their contents to shift one position to the right.

When the command reaches an ALU/shift stage, that stage performs an arithmetic operation if specified, and a shift operation. The stage also generates a carry out that is bundled with the Booth command and relayed to its left neighbor.

4.2.3. Termination. When the sentinel reaches the Booth controller, the computation terminates, and the controller stops issuing further commands. The controller’s internal state and history bit are cleared, and it prepares for the next set of operands. Strictly though, at this point there can be some commands that are still flowing through the pipeline. The initialization phase of the next iteration is used to handle this situation, as described below.

4.2.4. Initialization (next round of computation).

Upon termination, the Booth controller re-initializes the multiplier by generating a new initialize command. In addition to what was described above in Section 4.2.1, the initialize command also serves the purpose of ensuring that all prior Booth commands that are still flowing through the pipeline are correctly completed before the multiplication result is read.

In particular, when the initialize command reaches any pipeline stage it causes that stage to copy the output of its left neighbor onto its P output, which represents the final product value for that radix-4 digit position, in dual-rail form (see Figure 2). Taken together, \( P_{2n-2} \cdots P_0 \) represents the result of the multiplication, where \( n \) is the number of radix-4 digits in the operands (i.e., \( 2n \)-bitoperands). Even though the lower significant product bits are produced earlier than the higher significant ones, the dual-rail encoding of \( P_t \) ensures that the completion of the computation and validity of the result are correctly and robustly indicated.

4.2.5. Overlapped Execution of Consecutive Computations. Just as successive iterations of the same computation are executed in an overlapped fashion, our implementation allows an overlap between the last (or last few) iterations of one computation, with the first (or first few) iterations of the next computation. In particular, the Booth controller immediately commences issuing new Booth commands for the next round of computation, even though one or more commands for the previous computation may still be flowing leftward through the pipeline.
This ability to overlap successive computations is quite advantageous, resulting in significantly reduced latency for the multiplier. The results of our experiments indicate that the benefit is around a 60% reduction in latency.

4.3. Implementation

4.3.1. Pipeline Handshake Circuits. The pipeline handshake circuits used in the multiplier implementation are based closely on the HC style of [18] (see Section 2.1), but include a significant enhancement to enable bi-directional communication, which in turn is critical to enabling the counterflow organization.

Figure 4(a) shows the top-level view of the new pipeline stage. Compared with Figure 1(a), the new stage has an extra input and an extra output: it receives Command_in from its right neighbor, along with Ack_in, and similarly produces Command_out for its left neighbor, along with Ack_out.

Figure 4(b) shows the internal organization of a generic stage of the new pipeline. There is a key difference with respect to the HC pipeline of Section 2.1: a buffer must be added to store the incoming command, because the command arrives at the start of a precharge phase, but it will be needed in a subsequent evaluation. During that evaluation, the command is combined with the input data to not only produce output data, but also to generate a new command for the left neighbor.

As an optimization, some preprocessing of the command is performed when it is stored in the command buffer: e.g., since operand /B is always available even if /A has not yet been received from the left neighbor, the command could actually be combined with /B to produce an intermediate result. As a result, the main function block, labeled “Data Function,” only needs to combine the intermediate result with operand A, resulting is reduced complexity for the function block. This optimization has the benefit of speeding up the critical forward path through the pipeline; the command buffering and preprocessing is off of the critical path.

Finally, Figure 4(c) shows the actual modification made to the HC handshake circuit. The new handshake circuit produces two additional outputs—Command_precharge and Command_evaluate—which serve as the control signals for the command buffer. The figure highlights the additional gates needed to generate the new signals. The function block’s precharge signal triggers the evaluation of the command buffer, causing the incoming command to be stored. The command buffer is subsequently precharged only after the command is “consumed” by the function block upon its next evaluation. The new inverter and NAND gate directly implement this functionality.

4.3.2. Booth Controller. Figure 5 shows the block diagram of the Booth controller. The history buffer stores a copy of the most recently examined multiplier bit. This history bit is communicated to the Booth encoder stage as a command. The Booth encoder processes this history bit along with the two new least significant multiplier bits—LSB1 and LSB0—and generates the appropriate Booth command, as summarized in the truth table of Figure 6.
5. Experimental Results

This section presents the results of electrical simulations of our new Booth multiplier. The multiplier was designed using the Cadence tool suite, and simulated using Spectre in a 0.18μm TSMC CMOS process, using the NCSU Cadence Design Kit. Tables 1 and 2 summarize the results of our simulations.

Table 1 presents the latency and energy for several instances of our multiplier with different operand widths, using a pair of random inputs. The results indicate that, as expected, the latency varies linearly with operand width, from 1.28ns for a 4x4-bit multiplier, to 10.4ns for a 32x32-bit multiplier. However, as indicated in the rightmost column, the iteration time remains constant across different widths, thereby demonstrating the scalability of our approach. The middle three columns indicate the energy consumed, as well as the composite energy-performance metrics, $E\tau$ and $E\tau^2$ (see Section 2.2).

Table 2 presents the results of our experiments with voltage scaling for the 16-bit version of our multiplier. Latency and energy measurements were taken at three different voltages: 1.8V, 1.5V and 1.0V. As expected, as the voltage was lowered, the latency degraded, along with a reduction in energy consumption. Interestingly, $E\tau^2$ is fairly invariant from 1.8V to 1.5V, as predicted by [14, 15], but showed variation when the voltage was further decreased to 1.0V, which is no longer in the linear operating region.

Finally, Table 3 tabulates the results of the new multiplier (“radix-4”) together with the radix-4 design of [5] (“efth-r4”), and the radix-2 design of [8] (“radix-2”).

Compared with efth-r4 [4], our multiplier represents a nearly two-fold (1.95x) increase in throughput. Although our multiplier consumes more energy per multiplication than the one in [4], it actually has a superior energy-delay product: 9.65 nJ · (ns)$^2$ compared with 14.8 nJ · (ns)$^2$.

However, it is important to note that while our architecture is scalable to arbitrary operand widths (e.g., 128-bit, 1024-bit, etc.), with practically no overhead to iteration time, the efth-r4 multiplier is not easily scalable since it requires broadcast of the Booth command across the entire width of the operands. Therefore, the advantages of our design will be amplified for wider operands, e.g., those used in cryptosystems.

Finally, the table indicates that our new multiplier outperforms the radix-2 design of [8]: 3.3x higher throughput is obtained in the new design while consuming 2.2x lower energy consumption.

6. Dynamic Precision-Energy Trade-off

This section outlines an approach to achieving a dynamic trade-off between computation precision and energy consumption in our multiplier architecture. Only a brief sketch of this idea is provided here; experimental validation is part of ongoing work.

Figure 7 shows the overall architecture proposed for handling variable width operands. There are several key differences between this picture and Figure 2.

First, the variable-precision architecture employs full ALU/shift units in each multiplier stage. In contrast, in Figure 2, the right half of the multiplier only had simpler shift units, with no ALU capability. This modification is required because the right half of the multiplier pipeline could now be involved in ALU computations for short operands.
Second, two sentinels are always needed: one to demarcate the MSB-end of the multiplier, and another to indicate the MSB-end of the multiplicand. The two operands are now supplied packed together as a single word.

Third, the function blocks in the ALU stages, which generate/relay the command for their left neighbor need a slight modification. In particular, when an add or subtract Booth command enters the multiplier from the right end, it is initially regarded by all the ALU units as a shift command. However, as the command makes its way through the circuit, its interpretation is changed to that of the actual add or subtract operation. As a result, the subsequent ALU stages interpret the command as a shift command, and the command is not relayed to any of the stages to the left.

The final difference is that the product also contains a sentinel to indicate its MSB-end.

With these modifications, the multiplier architecture is capable of handling dynamically-varying operand widths, thereby giving it a powerful architectural tool to allow the system to dynamically trade computation precision for energy efficiency.

7. Conclusion and Future Work

This paper presents a novel radix-4 Booth multiplier implementation using a novel asynchronous pipelining style and a counterflow architecture. The design allows for overlapped execution of multiple Booth iterations through bit-level pipelining. Simulation results are quite promising with a nearly two-fold speedup over the fastest reported radix-4 asynchronous Booth multiplier, and a 3.3x speedup over a comparable radix-2 implementation.

References