An Area- and Energy-Efficient Asynchronous Booth Multiplier for Mobile Devices*

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

The recent explosion in the number of handheld multimedia devices has created a need for energy-efficient computation due to limited battery lifetimes. We focus on multiplication, which is needed in several application domains, e.g., *3D graphics, signal processing, and cryptography. We introduce an asynchronous implementation of a plain Booth multiplier (*i.e., *radix-2) which is both area- and energy-efficient, and therefore suitable for mobile applications.*

This paper makes the following contributions. First, a novel counterflow organization is introduced, in which the data bits flow in one direction, and the Booth commands piggyback on the acknowledgments flowing in the opposite direction. Second, the arithmetic and shifter units are merged together to obtain significant improvement in area, energy as well as speed. Third, our design performs overlapped execution of multiple iterations of the Booth algorithm. Finally, the design is quite modular, which allows scaling to arbitrary operand widths, without gate resizing or cycle time overheads.

Spice simulations in a 0.18μ*m TSMC process at 1.8V, indicate promising performance: the multiplier takes 1.08ns per Booth iteration, regardless of the operand widths, thereby demonstrating the scalability of our approach. In addition, the multiplier is fully functional at reduced supply voltages (*e.g., *1.0V), and thus capable of dynamically trading off performance for energy efficiency.*

1. Introduction

This paper introduces a multiplier design that is especially targeted to mobile devices, with the key objectives of high energy efficiency and small chip area. The motivation for the new design comes from the recent explosive growth in handheld multimedia devices, whose limited battery lifetimes have created a need for energy-efficient computation. For instance, portable consumer electronic devices are likely to increasingly depend on the following capabilities: 3D graphics computation (*e.g.,* cell phones [7], 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.

It is important to note the distinction between energy efficiency and power efficiency. The former represents the energy consumed per operation (*e.g.,* nano-Joules/op), whereas the latter refers to energy consumed per unit time (*e.g.,* milli-Watts). For applications where battery lifetimes are critical, energy efficiency is the more relevant metric. On the other hand, power efficiency is more relevant for those desktop or high-performance applications where heat dissipation or supply current are the limiting factors.

Our multiplier is of the iterative Booth (radix-2) type, implemented using asynchronous circuits. An iterative implementation was chosen, as opposed to a combinational array type, for higher area efficiency. A Booth implementation was chosen so as to 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, 17]. 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. This feature allows shorter critical paths, and therefore higher operating speed.
- *Merged Arithmetic/Shifter Unit:* An architectural optimization is introduced, which merges the arithmetic

*This work was supported by NSF Award CCF-0306478, an ATI Graduate Student Fellowship, and an IBM Faculty Development Award.

operations and the shift operation into the same function unit, thereby obtaining significant improvement in area, energy as well as speed.

- *Overlapped Execution:* The entire design is pipelined at the bit-level, which allows overlapped execution of multiple iterations of the Booth algorithm, including across successive multiplications.
- *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.
- *Precision-Energy Trade-Off:* Finally, the design can be easily modified to allow dynamic specification of operand widths, which enables 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 1.08ns 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 8.5ns for 8-bit operands, and approximately 34ns for 32-bit operands. The energy consumed per multiplication varies as approximately square of the operand width, as expected: 0.22nJ for 8-bit and 2.97nJ 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.

The remainder of this paper is organized as follows. Section 2 presents background on a particular asynchronous pipelined circuit style, and introduces certain 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, and Section 6 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 [15]. 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.

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.,* PS0 [19]), HC does not require intervening "bubble" or "spacer" stages between data items, thereby keeping energy consumption and area low.

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.

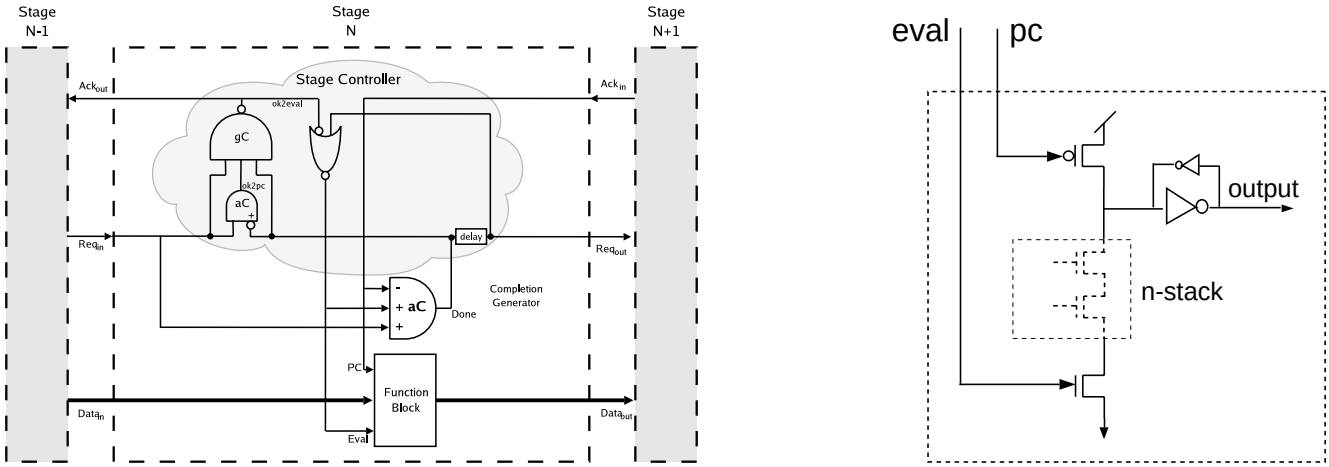
Structure. Figure 1(a) shows a block diagram of a high-capacity pipeline. Each stage consists of three components: *function block,* a *completion generator* and a *stage controller.* The function block is implemented using dynamic logic. It alternately evaluates and precharges, thereby alternately producing 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 [13, 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.[1]

The *completion generator* is implemented using an asymmetric C-element, *aC* [5]. 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*.

[1] Section 4.3.1 extends high-capacity pipelines to dual-rail with completion detection, in order to make its operation more robust to process variations.



(a) Pipeline fragment showing three stages

(b) Details of a gate within a function block

Figure 1. The high-capacity (HC) pipeline style

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 gate labeled gC is a state-holding *generalized C-element* [5], which behaves as follows: (a) when all three inputs are asserted high, its output is driven low; (b) when either of Req_{in} or *ok2pc* inputs is de-asserted low, the gC's output is driven high; and (c) for all other input combinations, the gC holds its output. The state variable *ok2pc* is implemented using another asymmetric C-element as follows: *ok2pc* is set when Req_{in} is asserted high and *Done* is de-asserted low; *ok2pc* is reset when Req_{in} is de-asserted low.

Operation. An HC pipeline stage simply cycles through three phases. After it completes its evaluate phase, it enters its isolate phase and subsequently its 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 can not 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.

Performance. If the evaluation and precharge times for a stage are denoted by t_{Eval} and t_{Prech}, and the delay through the aC and gC elements by t_{aC} and t_{gC}, respectively, then the analytical cycle time of the pipeline is given by:

$$T_{\mathrm{HC}} \;=\; t_{\mathrm{Eval}} + t_{\mathrm{Prech}} + t_{\mathrm{aC}} + t_{\mathrm{gC}} + t_{\mathrm{NOR}}$$

2.2. Energy–Performance Trade-Off, and the $E\tau$ and $E\tau^2$ 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, $E\tau$, and the energy-delay^2 product, $E\tau^2$. Here, E refers to the energy consumed per operation, and τ is the execution time per operation (or the *cycle time*, which is inverse of the *throughput*). The energy-delay product, normalized with respect to λ^2 (λ = feature size), is fairly invariant across fabrication processes [6]. The second metric, $E\tau^2$, on the other hand, is invariant to moderate voltage scaling [11, 12]. Thus, depending upon the particular comparison being made, one or the other metric can be used.

3. Related Work: Asynchronous Multipliers

Several asynchronous multipliers has been reported in literature, both array as well as iterative. It was shown in [18]

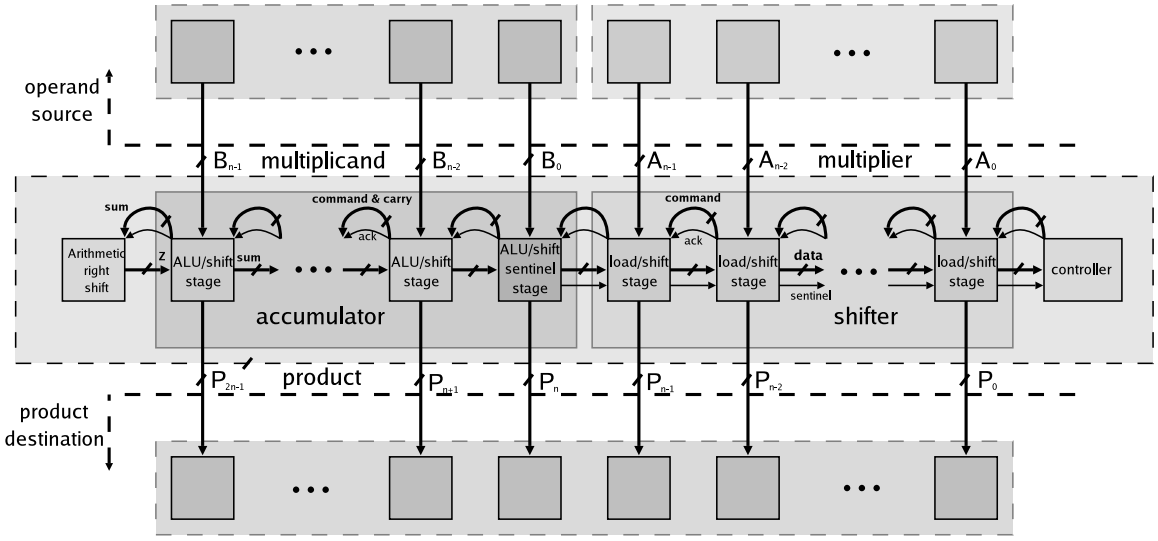


Figure 2. The counterflow Booth multiplier

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.

Several iterative multipliers have been introduced recently. In [9], an area-efficient low-power multiplier is described for use in a hearing aid. [8] 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, [10] 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 [14], which produces the product in $n/4$ iterations; however, it takes more than twice the area of a shift-and-add iterative multiplier. Recently, [4] has proposed an implementation of the original Booth algorithm, which is able to skip over arbitrarily long runs of ones and zeros. However, due to the added complexity, its iteration time is actually longer, and therefore it exhibits performance advantages only for certain cases.

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 which 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, subtract and 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 flow (*i.e.*, add, subtract or shift) from right to left. Wherever carry or borrow 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.

The flow of data and commands is interlocked to achieve correct operation. In particular, a data bit flows through a processing stage (*i.e.,* moves right from its input side to its output side) only after that stage has received the associated Booth command. Similarly, a command is relayed from a stage to its left neighbor only after it has interacted with valid data.

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 transformation applies to the carry/borrow bits which are embedded within the command: the carry/borrow bits are replaced by the new carry/borrow 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. 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.

It must be emphasized that our counterflow pipeline organization is quite different from another counterflow organization proposed by Sproull et al. [16]. In particular, the architecture in [16] 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 thereby 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.

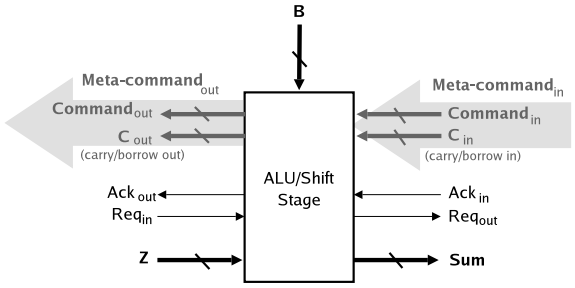


Figure 3. The merged arithmetic/shift unit

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/borrow bits (labeled C_{in}), 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/borrow bit (labeled C_{out}); 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/borrow (C_{in}) 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-4 (*i.e., one-hot*) encoding that is delay-insensitive. The four representable commands are *initialize*, *add*, *subtract* and *shift*. Besides being delay-insensitive, the encoding has three advantages: (i) no decoding circuitry required, (ii) ease of completion detection, and (iii) good energy ef-

ficiency, 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-4 encoding for the command is augmented by a 1-of-2 code to allow a carry/borrow bit to be also carried within the command (see Figure 3), making each command a 6-bit value.

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, our design can be easily modified to provide the ability to dynamically specify the multiplier length, which in turn can be exploited to enable 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 multiplier 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 terminating condition that the Booth controller can sense.

Finally, as the *initialize* command reaches each ALU/shift stage, it causes both the sum and the carry/borrow 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 which was omitted from the discussion here: to indicate completion of the *previous* computation. This function is explained later 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*, *subtract*, 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 three commands as a *shift* command, and cause their contents to shift one position to the right.

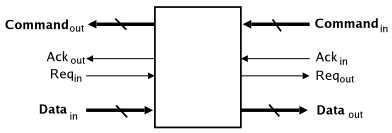
When the command reaches an ALU/shift stage, that stage performs an arithmetic operation if specified, and a shift operation. The stage also generates a carry/borrow out that is bundled with the Booth command and relayed to its left neighbor.

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 throwing the pipeline. The *initialization* phase of the next iteration is used to handle this situation, as described below.

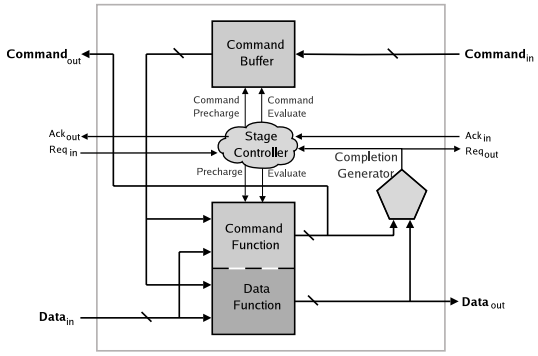
4.2.4. Initialization (next round of computation). Upon termination, the Booth controller re-initializes the multiplier by generating a new *initialize* command. In addition to what was described above in Section 4.2.1, the *initialize* command also serves the purpose of ensuring that all prior Booth 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 bit position in dual-rail form (see Figure 2). Taken together, $P_{2n-1} \cdots P_0$ represents the result of the multiplication. Even though the lower significant product bits are produced earlier than the higher significant ones, the dual-rail encoding of P_i ensures that the completion of the computation and validity of the result are correctly and robustly indicated.

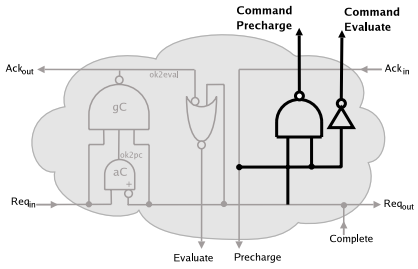
4.2.5. Overlapped Execution of Consecutive Computations. Just as successive iterations of the *same* computation are executed in an overlapped fashion, our implemen-



(a) Block diagram



(b) Detailed view



(c) Stage controller

Figure 4. Pipeline Handshake Controller

tation 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 [15] (see Section 2.1), but include a significant enhancement to enable bi-directional communication, which in turn is critical to enabling the counterflow organization.

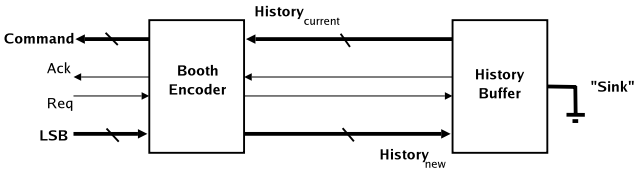


Figure 5. Booth Controller

LSB	History bit 0	1
0	*shift*	*add*
1	*subtract*	*shift*
sentinel	*init*	*init*

Figure 6. Booth commands

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 are two key differences with respect to the HC pipeline of Section 2.1. First, completion information is no longer generated using a matched delay. Instead, a robust delay-insensitive completion generator is used. Since the datapath is encoded using a 1-of-n code, completion is easily detected: when all rails of a bit a are reset, precharge is indicated; when exactly one rail is set, completion of evaluation is indicated. The second modification is the addition of a buffer 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.

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 new least significant multiplier bit, and generates the appropriate Booth command (*i.e.*, add, subtract or shift), as summarized in Figure 6.

5. Experimental Results

This section presents the results of electrical simulations of our new Booth multiplier. The multiplier was designed at the transistor level 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 4.21ns for a 4x4-bit multiplier, to 34.17ns 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 1. Spice simulation results: latency and energy vs. operand size

Word Size	delay, τ (ns)	E (nJ)	$E\tau$ $(10^{-18}Js)$	$E\tau^2$ $(10^{-24}Js^2)$	iter time (ns)
4-bit	4.21	0.07	0.27	0.001	1.08
8-bit	8.49	0.22	1.85	0.02	1.08
16-bit	17.05	0.79	13.42	0.23	1.08
32-bit	34.17	2.97	101.52	3.47	1.08

Table 2. Effect of voltage scaling on 16-bit multiplier

Supply Voltage	delay, τ (ns)	E (nJ)	$E\tau$ $(10^{-18}Js)$	$E\tau^2$ $(10^{-24}Js^2)$	iter time (ns)
1.8V	17.05	0.79	13.42	0.23	1.08
1.5V	21.53	0.53	11.43	0.25	1.34
1.0V	48.07	0.23	10.84	0.52	2.91

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 [11, 12], but showed variation when the voltage was further decreased to 1.0V, which is no longer in the linear operating region.

6. Conclusion and Future Work

This paper presented a radix-2 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.

In ongoing work, we are extending the design to implement the modified Booth (radix-4) approach. In addition, we are exploring handling variable-length operands, and also handling variable-length shifts to further optimize our design.

References

[1] V. Bartlett and E. Grass. A low-power concurrent multiplier-accumulator using conditional evaluation. *ICECS 1999*, pages 629 – 633, 1999.

[2] C. H. K. v. Berkel, M. B. Josephs, and S. M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE*, 87(2):223–233, Feb. 1999.

[3] A. Davis and S. M. Nowick. Asynchronous circuit design: Motivation, background, and methods. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 1–49. Springer-Verlag, 1995.

[4] A. Efthymiou, W. Suntiamorntut, J. Garside, and L. Brackenbury. An asynchronous, iterative implementation of the original booth multiplication algorithm. *ASYNC 2004*, pages 207 – 215, 2004.

[5] S. B. Furber and J. Liu. Dynamic logic in four-phase micropipelines. In *Proc. Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Mar. 1996.

[6] R. Gonzalez and M. Horowitz. Energy dissipation in general purpose microprocessors. *IEEE Journal of Solid-State Circuits*, 31(9):1277–1284, Sept. 1996.

[7] M. Kameyama, Y. Kato, H. Fujimoto, H. Negishi, Y. Kodama, Y. Inoue, and H. Kawai. 3d graphics lsi core for mobile phone 'z3d'. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics Hardware*, pages 60–67, 2003.

[8] D. Kearney and N. Bergmann. Bundled data asynchronous multipliers with data dependent computation times. *ASYNC'97*, pages 186 – 197, 1997.

[9] K. Killpack, E. Mercer, and C. Meyers. A standard-cell self-timed multiplier for energy and area critical synchronous systems. *ARVLSI 2001*, 2001.

[10] D.-W. Kim and D.-K. Jeong. A 3232 self-timed multiplier with early completion. *AP-ASIC'99*, pages 319 – 322, 1999.

[11] A. Martin, M. Nystroem, and P. Penzes. *Power-Aware Computing*, chapter ET^2: A Metric For Time and Energy Efficiency of Computation. Kluwer Academic Publishers, 2001. `http://caltechcstr.library.caltech.edu/archive/00000308`.

[12] A. J. Martin. Towards an energy complexity of computation. *Information Processing Letters*, 77:181–187, 2001.

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

[14] M.-C. Shin, S.-H. Kang, and I.-C. Park. An area-efficient iterative modified-booth multiplier based on self-timed clocking. *ICCD 2001*, pages 511 – 512, 2001.

[15] M. Singh and S. M. Nowick. Fine-grain pipelined asynchronous adders for high-speed DSP applications. In *Proceedings of the IEEE Computer Society Workshop on VLSI*, pages 111–118. IEEE Computer Society Press, Apr. 2000.

[16] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. The counterflow pipeline processor architecture. *IEEE Design & Test of Computers*, 11(3):48–59, Fall 1994.

[17] C. Tristram. It's time for clockless chips. *Technology Review Magazine*, 104(8):36–41, Oct. 2001. `http://www.technologyreview.com/magazine/oct01/tristram2.asp`.

[18] Y. Wang, Y. Jiang, and E. Sha. On area-efficient low power array multipliers. *ICECS 2001*, pages 1429 – 1432, 2001.

[19] T. E. Williams. *Self-Timed Rings and their Application to Division*. PhD thesis, Stanford University, June 1991.