An Asynchronous, Iterative Implementation of the Original Booth Multiplication Algorithm

A. Efthymiou W. Suntiamorntut J. Garside L. E. M. Brackenbury

Department of Computer Science,
University of Manchester,
Oxford Road, Manchester M13 9PL, UK
E-mail: {ae,eve,jdg,lemb}@cs.man.ac.uk

Abstract

One of the main reasons for using asynchronous design is that it offers the opportunity to exploit the data-dependent latency of many operations in order to achieve low-power, high-performance, or low area. This paper describes a novel, asynchronous, iterative multiplier which exhibits data-dependency in both the number of iterations required to produce the result and in the delay of each step of the iteration. The preliminary evaluation of the multiplier, implemented using standard-cells, shows that speed improvements can be achieved in comparison to a standard iterative, radix-4 Booth multiplier.

1. Introduction

One of the main reasons for using asynchronous design is that it offers the opportunity to exploit the data-dependent latency of many operations in order to achieve low-power, high-performance, or low area. A typical example is a ripple-carry adder: statistically, the number of bits that a carry propagates is very small [1], thus its average delay is just a handful of gates. At the same time it is by far the smallest possible adder implementation.

Although data dependent processing is usually linked to delay-insensitive circuits, it can also be achieved with bundled-data circuits, as was shown in [2] and [3]. Bundled-data design produces circuits which are much smaller and, possibly, with a lower power consumption than circuits built with delay-insensitive methods. Thus, bundled-data circuits with data-dependent processing times are a very interesting class of circuits.

Although a number of asynchronous multipliers have been described in the literature [2] [4] [5] [6], none of them implements the *original* Booth algorithm [7]. This algorithm ‘scans’ the multiplier operand and skips chains of consecutive ones or zeros. For example if the multiplier is 00111100, it is converted to $-2^2 + 2^6$ and each of the two terms is multiplied (actually shifted, since they are powers of two) with the multiplicand and then added together. This algorithm can reduce the number of additions required to produce the result compared to the paper and pencil algorithm, where each bit of the multiplier is multiplied with the multiplicand and the partial products are aligned and added together. More interestingly the number of additions is data dependent, which makes this algorithm a good candidate for data-dependent processing.

The commonly used, modified (radix-4) Booth algorithm considers two bits of the multiplier operand at a time. Therefore the number of operations is *fixed* to half the bit-width of the multiplier. Some of these operations are additions with zero, when the considered bits of the multiplier are all ones or zeros.

This paper describes an asynchronous, iterative multiplier exhibiting data-dependency in both the number of iterations required to produce the result and in the delay of each step of the iteration. The design is described progressively, in much the same way as it was conceived, from a simple initial implementation to the final optimised design.

A first, simple implementation of an iterative Booth multiplier that uses a carry-propagate adder is presented in section 3. Then the design is improved in two steps. First the ‘worst-case’ of the original Booth algorithm, where the number of operations is the same as in the paper and pencil algorithm, is identified and a solution is found that reduces the number of operations in this case to half (section 3.2). Second, the carry-propagate adder is replaced with a carry-save adder, speeding-up and reducing the energy consumed in each iteration (section 4). In section 5 the presented designs are evaluated and compared with a standard, radix-4 Booth multiplier and the related work is reviewed in section 6. Finally in section 7 the conclusions are given.

2. Multiplier Architecture Issues

There are two main multiplier architectures: array (or tree) multipliers and iterative multipliers [8]. Array multipliers are implemented by an array structure of adders, while iterative multipliers use a few functional units repeatedly to produce the result.

2.1. Array multipliers

Array multipliers are the common choice for high-speed multiplication. Since they can be easily pipelined, they are used where high-throughput multiplication is required, in DSP applications, for instance.

The most typical implementation of an array multiplier is the radix-4, modified Booth multiplier [8] with carry-save (CS) addition for 'compressing' the partial products to two numbers which are then added with a carry-propagate (CP) adder to produce the final result. Carry-save addition is used as it is much faster than CP addition, since there is no carry propagation, and the circuit area is substantially lower.

In a pipelined, array multiplier there are limited opportunities for exploiting data-dependent delay, due to both the simplicity of the circuits used and the pipeline behaviour.

Although data-dependent delay can be employed in the final CP adder, the CS stages do not exhibit significant data-dependent delay variation. For example using a multiplexer to bypass the CS adder when one of its operands is zero, adds about 2 gate delays to the critical path, while the CS delay itself is about 2 gates. Kearney [2] showed a circuit to combine the multiplexer with the CS adder that exhibits some data-dependent benefit, at the expense of leaving some of the least significant carries unresolved; these then have to be accounted for by the final CP adder, thereby increasing its size and delay.

Elastic pipelining in asynchronous circuits causes a slow stage to block its upstream stages. Since it is quite common for at least one stage to be processing its worst possible case, the whole pipeline will be affected [9]. This effect was observed in the array multiplier of Kearney [2] where the data dependent speed advantage of isolated cells was almost completely lost.

As the opportunities of data-dependent delay in array multipliers are limited, we restrict our interest to iterative multipliers.

2.2. Iterative multipliers

Figure 1 shows the shift-add, iterative implementation of a multiplier. Right-shifting is selected as it allows the use of a n-bit adder, where n is the bit width of the operands [8]; shifting left would require a $2n$-bit adder. A common, area-

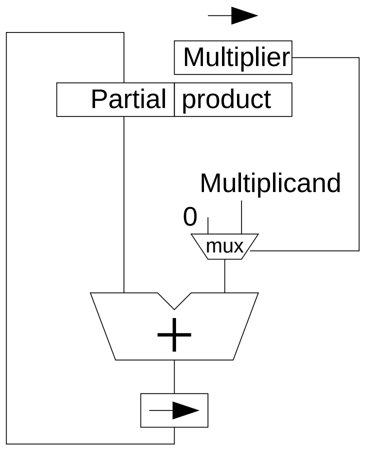

Figure 1. Shift-add, iterative multiplication.

saving optimization is to merge the registers of the multiplier and the low-part of the partial product into one.

In the simplest implementation of this type of multiplier, constant shifting by one is used, thus the product is complete after n iterations. By using the modified Booth algorithm, 2 bits at a time are operated upon, reducing the number of iterations to $n/2$, while constant shifting by 2 is used. The only modifications in the above schematic is that a more complex 'multiplexer' is required and a carry-in input is needed for the adder to form the 2's complement versions of the multiplicand. Higher radix Booth encoding can be used to further reduce the required number of iterations (constant shifting by more than two) at the expense of having to produce not only shifted and complemented versions of the multiplicand (M), but also terms such as $\pm 3 \times M$.

Within each step there are opportunities to exploit variable delay; frequently zero is added to the partial product, which could be detected and result in a move to the next iteration faster than when an addition or subtraction takes place. Moreover the variability in the addition/subtraction can also be exploited by using the appropriate circuits. However, a more sophisticated implementation would use a carry save adder and only use a carry propagation adder at the end to produce the final result. The block diagram of this implementation is shown in figure 2. In this case the variability of each step is severely diminished, as explained for array multipliers.

Generally, in a design where shifting by a constant amount in every step is used, a fixed number of iterations are required to produce the result. However, having a fixed number of iterations means that the control and energy overhead per cycle is paid in full for the whole operation.

An optimization found in a number of modified Booth implementations [10][6] is to detect whether the remaining bits of the multiplier are all 1's or 0's and terminate the operation, since the rest of the partial products to be added are

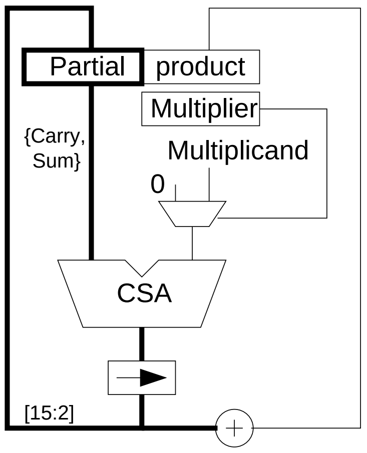

Figure 2. Shift-add, iterative multiplication with CS addition.

all 0. A shifter is required in this case to correctly align the final product.

Although this optimization can be useful in applications where small integers are being multiplied, it cannot be exploited in fractional arithmetic, commonly used in fixed-point applications. In the fractional arithmetic representation the least significant bits are usually zero as they represent the high accuracy digits of the fraction, while the most significant bits contain most of the useful information.

This work uses variable shifting per iteration, by employing the original Booth algorithm, leading to a variable number of iterations. But the requirement for variable shifting means that a 'real' shifter is needed now at the output of the adder. This adds to the time and energy consumption of each iteration, whereas constant shifting comes for free. One of the aims of this work is to establish trade-offs for when either implementation is the best in terms of speed (latency) and energy per multiplication. Obviously a fixed-shifting implementation has always a smaller area, as no shifter is required.

3. A simple, asynchronous, Booth multiplier

The first, simple implementation of the asynchronous Booth multiplier is shown in figure 3. A carry propagate adder is used in this implementation for simplicity. The partial product is kept in two registers: *prod_high*, *prod_low*. In the first iteration *prod_high* is cleared while *prod_low* is loaded with the value of the multiplier operand. A one bit register, *neg*, indicates whether the next operation is a subtraction (neg = 1) or an addition (neg = 0). For subtractions, the multiplicand is inverted and a 1 is added as a carry in to produce its 2's complement.

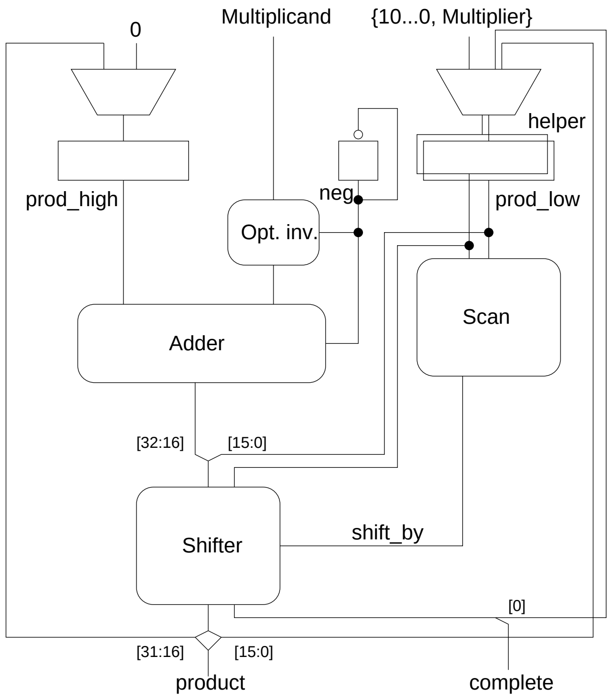

Figure 3. Iterative Booth multiplier with carry-propagation in each step.

Since the number of iterations is variable, an extra register, *helper*, is used to indicate when to stop the process. It is initialized to *1000...0*, the 1 being at the bit position above the MSB of the multiplier operand, and is shifted with the partial product in each iteration. When the single 1 in the helper is found at the rightmost position, the whole multiplier has been consumed and the multiplication is complete.

The scan block, shown in figure 4, scans the multiplier from right to left detecting the next position where an operation should be performed, i.e. where there is a break in consecutive zeros or ones. It is implemented as a combinational circuit. First the multiplier bits are bitwise XORed to produce a 1 where there is a 'break'. The result is passed into a priority enforcer which produces an output with a single 1, at the leftmost position where a 1 was found in its input. The output of the priority enforcer can be either used directly in a barrel shifter implementation, or encoded for a logarithmic shifter.

3.1. Description of operation

In each iteration the adder is adding a partial product while, in parallel, the scan block is searching for the next 'discontinuity' in the multiplier. When both results are available, the output of the adder is shifted by as many bits as scan indicates, so that the product will be correctly

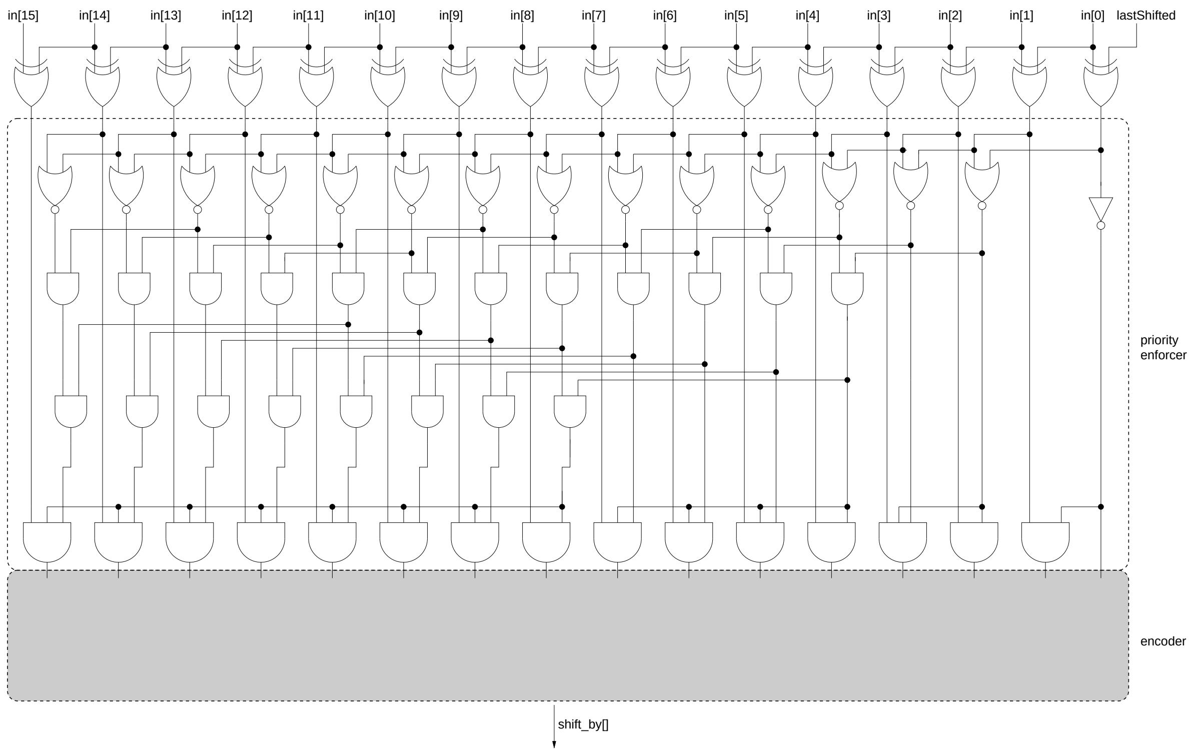

Figure 4. The scan logic for finding the next discontinuity in the multiplier operand.

aligned for the next operation. In the very first iteration no addition or subtraction is performed, while the first scan operation takes place. Thus the total number of iterations equals the number of multiplier discontinuities plus one. In this iteration multiplication by zero is also detected and the operation is completed in a single step.

The order of operations is fixed: the first discontinuity is always going to be 10, thus the first operation is always a subtraction, while the next is always an addition (01 in the multiplier). This observation is very useful as it simplifies the calculation of the next value of *neg* to a single inverter.

3.2. Reducing the number of operations

Since the original Booth multiplier performs one operation per ‘break’ in the multiplier, it does more work than necessary when it encounters the patterns ‘010’ and ‘101’. In the case of ‘010’ instead of performing a single addition, two operations are performed: a subtraction, for the 10 part and then an addition for the 01 part. The inverse happens for ‘101’ patterns.

In the worst case, when the multiplier operand is a series of alternating ones and zeroes, the total number of operations to compute the result is equal to the multiplier’s bit width, n. On the contrary, the modified, radix-4 Booth algorithm considers two bits at a time, thus it is guaranteed to perform $n/2$ operations.

As this property of the original Booth algorithm is clearly unwanted, a simple improvement was done on the previous implementation to avoid this situation. The scan block is modified so that it can detect when one of these patterns occur by looking at one bit further than the current discontinuity. When such a pattern is detected, the current iteration continues as normal, but an extra bit of information is stored to show that *(i)* the next discontinuity should be ignored and *(ii)* the next operation is the inverse of what it was supposed to be (or, in other words, the same as the one in the current step). For example, if the current operation is an addition the next operation, skipping the next discontinuity, will be an addition too.

When the first break in such a pattern is detected, the next discontinuity is known to be at the next bit of the multiplier. A simple modification in the priority enforcer can be used to skip it, without requiring a full iteration through the circuit, while keeping the adder idle. As the least significant bits of the priority enforcer have a lower logic depth, the added gates do not increase the critical path of the scan block. Moreover, the pattern detection happens in parallel with the shift operation, i.e. away from the critical path. The detection signal is used as a load-enable for the register keeping the *neg* bit, preventing it from getting inverted for the next operation, as it would normally do.

4. Using carry-save addition

In a multiplication a number of partial products have to be added together either using a tree of adders, or using a single adder many times. It is well known that using a carry-propagate adder for adding any more than two numbers is not the most efficient way of doing this operation [8]. Carry save addition is much better, deferring the slow carry propagation process for when it is actually needed. Thus a natural improvement of the iterative, original-Booth multiplier presented here is to implement it using carry-save addition. Figure 5 shows the block diagram of the circuit. Note that the thick lines represent two busses: one for the 'sum' and one for the 'carry' part of the partial product.

For reasons that will become apparent later, it was decided that the low part of the product should not be kept in a redundant, carry-sum representation, so as to avoid a $2n$ carry propagation adder at the end of the iterative process. In each iteration the part of the product that falls into the least significant part is resolved after the shift, in *CPAlow*, and stored in the low-part of the product. Thus only the most significant part of the product is kept in the redundant, carry-sum, representation. In the last step, the most-significant bits of the product are resolved, in *CPA high*, and concatenated with the least significant part to form the final result.

Replacing the CP adder with a much faster CS one, puts pressure on the scan block to produce the *shift_by* output at least at the same time as the adder. The output of the CS adder is ready within a few gate delays from the time the registers are loaded. The scan shift-control output, as shown in figure 4, takes at least five, which makes it the slower of the two blocks. In order to improve the speed of the scan block, the bitwise XOR stage can be removed. Instead of storing the multiplier value in the low-part of the product register, its pairwise XOR can be stored instead. As this information does not change while the circuit operates, there is no need to compute them again in each iteration, thus the XORs can be removed from the scan block.

4.1. Variable-width carry-propagate adder

A crucial component of this multiplier is the variable-length, carry-propagate adder, *CPAlow*. Since an arbitrary number of bits of the intermediate product can be shifted into the lower half, this adder must be able to handle variable bit-width operands. Moreover, some of the least significant bits of the intermediate product have already been resolved, thus no addition should be performed at these bit positions.

A bit mask, *enf_out*, from scan is used to determine which part of the shifted intermediate product is going to be resolved. This mask is derived from the output of the

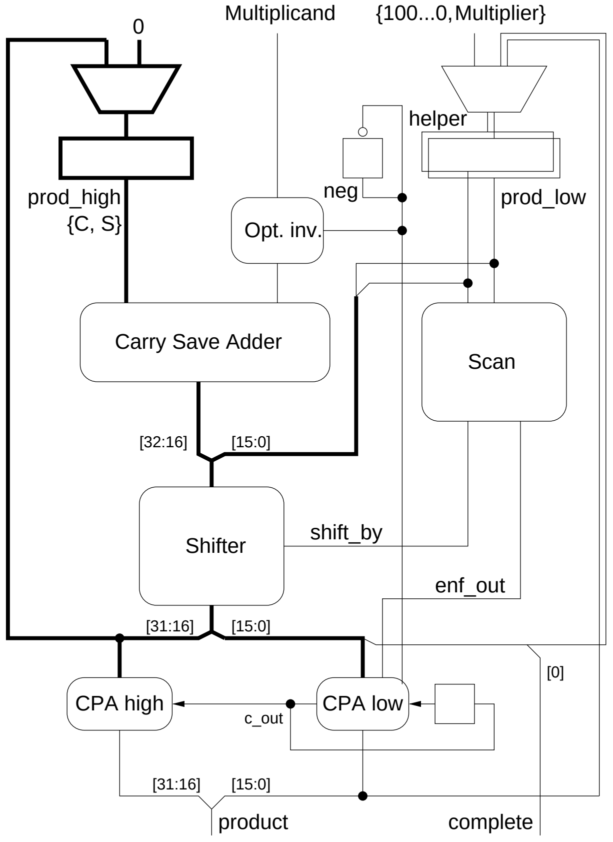

Figure 5. Iterative Booth multiplier with carry-save addition.

priority enforcer, so as all the bits, which correspond to the skipped bits of the multiplier, produce a 1, while the rest are 0. The resulting mask has as many 1's as the number of bits that are shifted and, consequently, as the number of bits that are going to be added.

Figure 6 shows the implementation of this block. It is essentially a carry-ripple adder, where a carry-in can be inserted at any bit position and one of the two operands can be either the carry part of the intermediate product, *neg*, or zero.

A carry out of *CPAlow* becomes the next carry into *CPAlow* at the next iteration. This is automatically aligned to the correct bit position, as the bit position of the carry out will become the least significant bit to be added in the next iteration. Obviously, in the last iteration, the carry out of *CPAlow* becomes the carry-in of *CPAhigh*.

Now that a CS adder is used to add the intermediate product with the current partial product, when the 2's complement of the multiplicand has to be added, there is no available input in the CS adder for the carry-in needed to convert the inverted multiplicand to its 2's complement. Thus this extra 1 bit is added in the *CPAlow*, by selecting *neg* as its

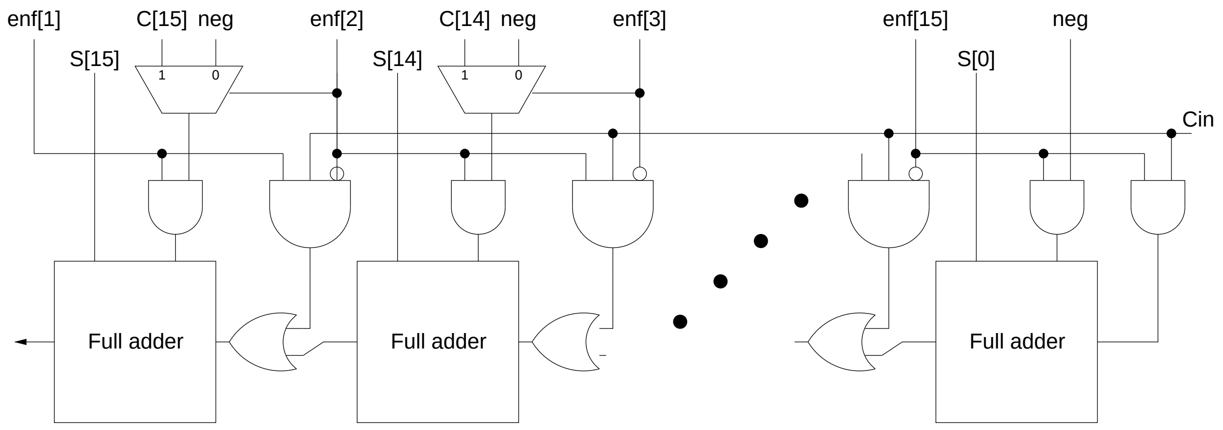

Figure 6. Variable width, carry-propagate adder.

input.

For the part of *prd_low* where the previously resolved bits of the product (and or the multiplier operand) are input, the bit mask ensures that the other operand of the adder (the shifted *helper*) is cleared, thus preserving these bits.

4.2. Control unit

Figure 7 shows the control unit for the multiplier. The SR latch is initialised to 0. When a request for multiplication is received, *load* rises and 'clocks' the registers in the datapath, all of which are edge-triggered. This also makes the SR latch output rise which, in turn, causes *load* to return to zero.

The first delay element matches the delay through the CS adder (or the scan) and the shifter. This delay is not data dependent. The following delay element is designed to match the delay through *CPAlow*. This delay is data dependent and its implementation is discussed in the following section. When the variable-width, CP adder has finished, if this is not the last iteration, the upper AND gate (*A1*) is activated which resets the SR latch and creates another pulse on load which restarts the cycle. For the last iteration the low AND gate (*A2*) is activated which, after a delay that matches that of *CPAhigh*, gives an acknowledgement signifying that the result is ready. When, the request returns to zero, in response to the acknowledge signal, the SR latch is reset again and it is ready for the next multiplication.

4.3. Discussion

Using a CS adder in this implementation unfortunately reduces the opportunity for data-dependent delay within an iteration. The circuit of a CS adder is too shallow in logic depth to justify bypassing it with a multiplexer. Moreover efficient shifter implementations do not exhibit data dependent delay either.

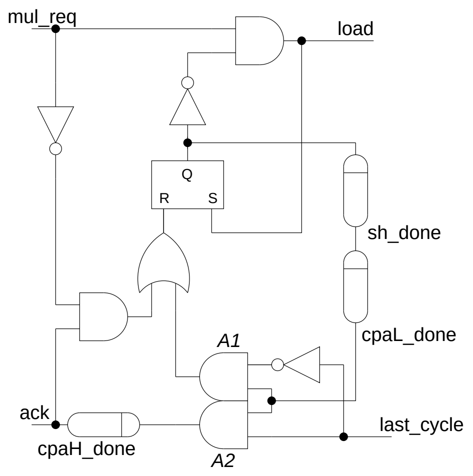

Figure 7. Control logic for the multiplier.

As a result the only circuit where data-dependent delay can be exploited is the variable-width, carry-propagate adder (*CPAlow*). In the implementation, shown in figure 6, instead of using completion detection, a variation of the simpler technique of speculative completion [3] was used. As explained above, the number of bits that are going to be added is known by the *shift_by* signal. This number determines the maximum number of bits a carry can propagate in the *CPAlow* adder and can be used to select the appropriate delay-matched element, as shown in figure 8.

At first the selection of a carry-ripple adder to implement *CPAlow* might appear the wrong choice as it is in the critical path. In practice, the maximum number of bits a carry can propagate in the whole multiplication can only be 16 for the specific adder (and another 16 for *CPAhigh*). This is because the total number of bits that can be shifted out in the whole multiplication process can be 16. Thus the larger the number of carry propagations that take place in this iter-

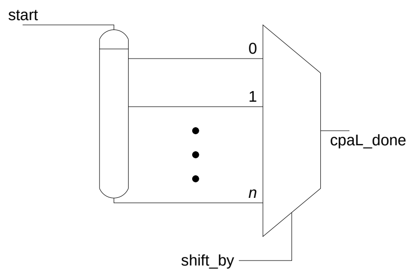

Figure 8. Selection of the matched CP adder delay.

ation, the less propagation will happen in the next step, and so on. As a result a carry ripple adder is a very area efficient way to implement *CPAlow* and at the same time the performance is not affected.

5. Implementation and evaluation

The multiplier datapaths have been modelled in RTL-level Verilog and synthesised for a 0.18μm technology, using a standard-cell library. The results presented here include the parasitic effects, as the designs were placed and routed.

For each multiplier, two variations were designed depending on the shifter implementation: one using tri-state drivers and one using multiplexers. The designs with the tri-state based shifters were consuming about twice the power of the others, mainly due to the brief short-circuits caused when a driver is not fully turned off before another is turned on. With a standard-cell based design, the timing of the tri-state enable signals could not be made precise enough to stop this effect. For this reason these design variations were deemed not suitable for standard-cell based implementation and are not described any further here.

The four multipliers that were evaluated are the following: 'CPA' is the first implementation presented in section 3. 'CPA-impr' is similar to 'CPA' but includes the 'look-ahead' improvement in the scan logic, as described in section 3.2. 'CSA' is the carry save implementation presented in section 4, which includes the 'look-ahead' optimization. Finally, 'Booth-r4' is a standard, iterative, radix-4, Booth multiplier with carry-save addition.

Table 1 shows the delay per iteration for each multiplier. The last column shows the time needed after the last iteration to produce the final result. For CPA designs the product is ready at the end of the final iteration, so no extra time is needed. On the contrary, an addition is required for CSA designs to convert the most significant bits of the product from the sum-carry representation into standard binary; thus the extra delay is non-zero here.

Table 1. Delay per iteration.

Design	cycle (ns)	last cycle (ns)
CPA	2.0	0
CPA-impr	2.1	0
CSA	2.5–4.4	+0.6
Booth-r4	1.2	+0.5

Of these designs only CSA has a data-dependent delay within each iteration, with quite a wide range between the two extremes. The two variations that use CPA addition could potentially have data-dependent delays too, but since the intention was to eventually use a carry-save adder, the CPA implementations just use standard look-ahead adders for simplicity. These implementations are presented mainly to demonstrate the effect of the modification in the scan block, which guarantees a maximum number of $n/2$ iterations.

From the delay results, it is clear that Booth-r4 has the fastest cycle time. This was expected since Booth-r4 does not have to perform a variable shift in each iteration. The improvement in the scan block for detecting the worst case patterns had a small penalty on the delay, but compared to the number of iterations that it can save, this penalty is very low.

Although a carry-save adder is faster than a carry-propagate adder, 'CSA' has a higher cycle time than CPA on which it was intended to improve. This is mainly due to two reasons: Changing the CP adder with a CS adder, puts the scan block in the critical path. The delay through that block is comparable to that of a CP adder. Moreover, the variable-width adder is inherently quite slow, as it needs to be able to insert a carry-in at any bit position. The combination of these two effects have caused the CSA design to be slower. We believe that this result is greatly affected by the standard-cell based implementation style; a full-custom implementation should greatly improve the speed of the 'CSA' multiplier.

The delay figures do show a potential speed advantage of the original Booth algorithm: To complete a multiplication Booth-r4 needs 8 cycles, 10.1ns in total. Dividing this time by the cycle time of CPA gives 5. Thus if fewer than 5 iterations are needed on average, CPA would be performing faster than Booth-r4, proving that variable shifting can be beneficial. Further work is needed to determine how many iterations are needed on average for 'typical' data. Further work is needed to gather such data from instruction-level simulations of benchmark programs.

Table 2 shows the average power consumption of the multipliers using our test-bench. The results are produced from Synopsys PowerCompiler and include the effect of parasitic capacitance. Since the test-bench was designed to

Table 2. Average power consumption.

Design	Power (mW)	Norm
CPA	25.5	1.8
CPA-impr	23.6	1.6
CSA	22.0	1.5
Booth r4	14.4	1.0

Table 3. Area comparison.

Design	Cells	Area ($10^3 \mu m^2$)	Norm
CPA	999	26.2	1.7
CPA-impr	1078	28.7	1.8
CSA	1949	52.1	3.3
Booth r4	928	15.7	1.0

use values to test the validity of the designs, they are not the average values that are likely to be used in multiplications in real applications. Thus the results presented here can be used as rough estimations only.

Unfortunately all the variable shift multipliers appear to be consuming significantly more power than Booth-r4. The ‘look-ahead’ improvement in the scan logic appears to be paying off in the power consumption as a small decrease can be observed in the table.

In the original Booth implementations, the majority of the power is consumed in the shifters which accounts for approximately 40% of the total. As explained earlier, the shifters were implemented using multiplexers because the implementation is based on standard-cells. A full-custom, barrel-shifter implementation is expected to produce much better results.

It is already stated that the areas of the original Booth multipliers is expected to be higher in comparison to the Booth-r4. Table 3 confirms this and shows that the size of CSA is particularly high in comparison to the CPA designs. The major reason is that using carry-sum representation has almost doubled the size of the shifters and registers that keep the upper-half of the partial product. Moreover, the variable-width adder is considerably larger than a standard adder and a third adder, *CPAhigh*, is included to produce the upper-half part of the product on the last cycle.

Further optimization of the presented designs is possible; we are investigating circuit and organization modifications. A possible improvement would be to build a transistor-level implementation based on pass transistor logic, which is very well suited for arithmetic circuits and multiplexers.

The above results, although preliminary, show that improvements in performance seem likely to be achieved by the presented class of multipliers, although their power consumption and area are quite high.

6. Related work

As explained in the introduction, we believe that this work is the first implementation of the original Booth algorithm using asynchronous circuits. Several implementations of iterative multipliers have been published, mostly using the modified, radix-4 Booth algorithm.

The design by Killpack *et al.* [5] is very similar to ‘Booth-r4’ used in our evaluation. As can be expected from the organization of the specific multiplier, data-dependent latency was not included in their design goals. Moreover early termination was not implemented.

Kearney *et al.* [2] have presented a bundled-data, iterative multiplier using carry-save addition and data dependent delays. The data dependent delay focused on bypassing the carry save adder when one of the operands is zero and on the implementation of the final carry-propagate adder. The number of iterations in their implementation is fixed, equal to the number of bits of the multiplier operand, as no variation of the Booth algorithm was employed. In comparison, the implementation presented here offers a variable number of iterations and variable delay per iteration.

Bartlett and Grass [4] have combined the data dependent carry-save adders of the above paper with conditional evaluation and used dynamic logic for the implementation to achieve ultra low power in an array multiplier.

Kim and Jeong [6] have presented an asynchronous, iterative, multiplier with early termination (called early completion in that paper) using dual-rail dynamic logic (DCVSL). A data-dependent delay variation of 9 times difference between the best and worst cases was achieved due to early termination.

Finally, the multiplier of AMULET3 [10] uses two levels of carry-save addition per iteration and is capable of early termination. It employs the radix-4 Booth algorithm, thus the intermediate product is shifted by a fixed amount in each iteration. Unfortunately it is designed in full custom in a different technology to the one used for the multipliers presented here, so no comparisons were made.

7. Conclusions

A novel, asynchronous, iterative multiplier has been presented. It implements the original Booth algorithm which allows it to exploit data-dependent delays not only within each iteration, but also in the number of iterations required to produce the result. This is achieved by including a shifter in the circuit, allowing a variable number of equal, consecutive bits of the multiplier operand to be skipped, unlike the common, radix-4 Booth algorithm which only considers two bits at a time.

First a simple implementation of the algorithm has been presented and has then been improved in a number of steps by making key modifications. The preliminary evaluation of these multipliers, implemented using standard-cells, show that speed improvement can be achieved although the power consumed is significantly increased in comparison to a standard iterative, radix-4 Booth multiplier.

Further work is now needed to develop a full-custom implementation of the proposed multiplier and evaluate it using ‘real-world’ values.

References

[1] Jim D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In S. Furber and M. Edwards, editors, *Asynchronous Design Methodologies*, volume A-28 of *IFIP Transactions*, pages 181–207. Elsevier Science Publishers, 1993.

[2] David Kearney and Neil W. Bergmann. Bundled data asynchronous multipliers with data dependant computation times. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 186–197. IEEE Computer Society Press, April 1997.

[3] Steven M. Nowick, Kenneth Y. Yun, and Peter 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*, pages 210–223. IEEE Computer Society Press, April 1997.

[4] V. A. Bartlett and E. Grass. A self-timed multiplier using conditional evaluation. In Anne-Marie Trullemans-Anckaert and Jens Sparsø, editors, *Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 429–438, October 1998.

[5] K. Killpack, E. Mercer, and C. J. Myers. A standard-cell self-timed multiplier for power and area critical synchronous systems. In *Advanced Research in VLSI Conference (ARVLSI)*, pages 188–201, March 2001.

[6] Do-Wan Kim and Deong-Kyoon Jeong. A 32x32 self-timed multiplier with early completion. In *Proc. AP-ASIC*. IEEE Computer Society Press, 1999.

[7] A. D. Booth. A signed binary multiplication technique. *Quarterly J. Mechanics and Applied Mathematics*, 4(2):236–240, June 1951.

[8] Behrooz Parhami. *Computer Arithmetic, Algorithms and Hardware Designs*. Oxford University Press, 2000.

[9] Mark R. Greenstreet and Brian de Alwis. How to achieve worst-case performance. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 206–216. IEEE Computer Society Press, March 2001.

[10] J. Liu. *Arithmetic and control components for an asynchronous microprocessor*. PhD thesis, Department of Computer Science, University of Manchester, 1997.