A Two-Speed, Radix-4, Serial-Parallel Multiplier

Duncan J.M. Moss, David Boland, Philip H.W. Leong
School of Electrical and Information Engineering
The University of Sydney, Australia 2006
Email: {duncan.moss,david.boland,philip.leong}@sydney.edu.au

***Abstract*—We present a two-speed radix-4 serial-parallel multiplier for accelerating applications such as digital filters, artificial neural networks, and other machine learning algorithms. Our multiplier is a variant of the serial-parallel modified radix-4 Booth multiplier that adds only the non-zero Booth encodings and skips over the zero operations. Two sub-circuits with different critical paths are utilised so that throughput and latency are improved for a subset of multiplier values. The multiplier is evaluated on an Intel Cyclone V FPGA against standard parallel-parallel and serial-parallel multipliers across 4 different PVT corners. We show that for bit widths of 32 and 64, our optimisations can result in a 1.42-3.36x improvement over the standard parallel Booth multiplier in terms of Area-Time depending on the input set.**

***Index Terms*—Multiplier, Booth, FPGA, Neural Networks, Machine Learning**

I. Introduction

Multiplication is arguably the most important primitive for digital signal processing (DSP) and machine learning (ML) applications, dictating the area, delay and overall performance of parallel implementations. Work on the optimisation of multiplication circuits has been extensive [1], [2], however the modified Booth Algorithm at higher radixes in combination with Wallace or Dadda trees has generally been accepted as the highest performing implementation for general problems [2]–[4]. In digital circuits, multiplication is generally performed in one of three ways: (1) parallel-parallel, (2) serial-parallel and (3) serial-serial. Using the modified Booth Algorithm [5], [6], we explore a serial-parallel Two Speed multiplier (TSM) that conditionally adds the non-zero encoded parts of the multiplication and skips over the zero encoded sections.

In DSP and ML implementations, reduced precision representations are often used to improve the performance of a design, striving for the smallest possible bit width to achieve a desired computational accuracy [7]. Precision is usually fixed at design time and hence any changes in requirements means that further modification involves redesign and changes to the implementation. In cases where a smaller bit width would be sufficient, the design runs at a lower efficiency since unnecessary computation is undertaken. To mitigate this, mixed-precision algorithms attempt to use a lower bit width some portion of time, and a large bit width when necessary [8]–[10]. These are normally implemented with two datapaths operating at different precisions.

This paper introduces a dynamic control structure to remove parts of the computation completely during runtime. This is done using a modified serial Booth multiplier, which skips over encoded all-zero or all-one computations, independent of location. The multiplier takes all bits of both operands in parallel, and is designed to be a primitive block which is easily incorporated into existing DSPs, CPUs and GPUs. For certain input sets, the multiplier achieves considerable improvements in computational performance. A key element of the multiplier is that sparsity within the input set and the internal binary representation both lead to performance improvements. The multiplier was tested using FPGA technology, accounting for 4 different Process-Voltage-Temperature (PVT) corners. The main contributions of this work are:

- The first serial modified Booth multiplier where the datapath is divided into two subcircuits, each operating with a different critical path.
- Demonstrations of how this multiplier takes advantage of particular bit-patterns to perform less work; this results in reduced latency, increased throughput and superior area-time performance than conventional multipliers.
- A model for estimating the performance of the multiplier and evaluation of the utility of the proposed multiplier via an FPGA implementation.

This paper is supplemented by an open source repository supporting reproducible research. The implementation, timing constraints and all scripts to generate the results are made available at: http://github.com/djmmoss/twospeedmult. The rest of the paper is organised as follows. Section III and Section IV focuses on the modified serial Booth multiplier and the two speed optimisation respectively. Section V covers the results and finally, related work and the contributions are summarized in Section II-B and Section VI respectively.

II. Multiplication

Multiplication is arguably the most important primitive for machine learning and digital signal processing applications, with Sze et. al [11] noting that the majority of hardware optimisations for machine learning are focused on reducing the cost of the multiply and accumulate (MAC) operations. Hence, careful construction of the compute unit, with a focus on multiplication, leads to the largest performance impact. This section presents an algorithm for the multiplication of unsigned integers followed by its extension to signed integers [2], [3].

Let x and y be the multiplicand and the multiplier, represented by n digit-vectors X and Y in a radix-r conventional number system. The multiplication operation produces

$p = x \times y$, where p is represented by the $2n$ digit-vector P. Multiplication is described as:

$$p = x \sum_{i=0}^{n-1} Y_i r^i = \sum_{i=0}^{n-1} r^i x Y_i, \quad (1)$$

Equation 1 can be implemented by first computing the n xr^iY_i terms followed by the summation. Computation of the ith term involves a i-position left shift of X and the multiplication of a single radix-r digit Y_i. This single radix-r digit multiplication is a scaling factor of the ith digit in the digit-vector set. In the case of radix-2, this is either 0 or 1. Performing the computation in this manner lends itself to a combinational or parallel multiplication unit.

The same computation can be expressed recursively:

$$\begin{aligned} p[0] &= 0, \\ p[j+1] &= r^{-1}(p[j] + r^n x Y_j) \quad j = 0, 1, \ldots, n-1, \\ p &= p[n], \end{aligned} \quad (2)$$

Expanding this recurrence results in product $p[n] = x \times y$ in n steps. Each time step j consists of a multiplication of x by a radix-r digit, Y_j, similar to Equation 1. This is followed by a digit left shift, and accumulated with the result from the previous time step $p[j]$. The recurrence is finished with a one digit right shift. It is expressed in this manner to ensure that the multiplication can proceed from the least-significant digit of the multiplier y, while retaining the same position with respect to the multiplicand x. An example is given in Figure 1.

Equation 1 can be extended to the signed, two's complement system through the incorporation of a sign bit for the multiplier y:

$$y = -Y_{n-1}2^{n-1} + \sum_{0}^{n-2} Y_i 2^i, \quad (3)$$

and substituting it into Equation 1. The new expression is given by:

$$p = \sum_{i=0}^{n-2} x Y_i r^i - x Y_{n-1} 2^{n-1}, \quad (4)$$

The negation of x $(-x)$ is performed by flipping all of the bits ($bf(1101) = 0010$) then adding a single bit in the least-significant position ($0010 + 1 = 0011$).

A. Multiplier Optimisations

There has been a rich history of ingenious optimisations for the efficient hardware implementation of multiplication, with the multitude of conventional techniques being reviewed in computer arithmetic textbooks [2], [3]. In particular, the signed Booth algorithm was proposed in 1951 [1], and the commonly-used modified Booth algorithm, presented in the previous subsection, in 1961 [5], [6].

Recent work has focused on static reordering of the computation or new layouts for the multiplication hardware on FPGAs. Rashidi et. al. proposed a low-power and low cost shift/add multiplexer-based signed Booth multiplier for a Xilinx Spartan-3 FPGA [12]. The authors used low-power structures, mainly a multiplexer-based Booth encoder with signed shifter blocks and a multiplexer-based Manchester adder. At 50 MHz the design consumes 58 mW with a total latency of 160 nsec. Devi et. al. focused on a fully combinatorial multiplier design which used custom carry select adders to reduce power consumption by 3.82% and 30% compared to standard ripple carry and carry select adders respectively [13]. Two contributions were made: a multi-stage partitioning approach which reduces the overall gate count, and a splitting clock method to reduce the power of the final accumulation. Our work is orthogonal to both works as the same optimisations and structures could be used with our TSM.

$n = 4$	$x = 13$ (X = 1101) $y = 9$ (X = 1001)
p[0]	0000
2^4xY_0	1101
p[1]	11101
2^4xY_1	0000
p[2]	111101
2^4xY_2	0000
p[3]	1111101
-2^4xY_3	0011
p[4]	01110101 = 117

Fig. 1: Unsigned two's complement Multiplication $p = x \times y$, where x is the multiplicand, y is the multiplier and X and Y are their respective $n = 4$ digit-vectors in the radix-2 conventional number system.

Kalivas et. al. described a new bit serial-serial multiplier capable of operating at 100% efficiency [14]. During standard bit serial-serial computation zero bits are added into the input pipeline between successive inputs words to allow time for the the most-significant bits of the product to be produced. Kalivas et. al. removed these bits by adding an additional shift register connected to a secondary output which allows for the most-significant bits of the previous product to be produced while the least-significant bits of the current product are produced. This work differs from our own in two important areas, first our multiplier is a serial-parallel multiplier using the radix-4 Booth algorithm. Secondly, our multiplier can operate at $>100\%$ efficiency since computation is effectively skipped, completing the multiplication in a faster than expected time.

Other work such as Hinkelmann et. al. has focused on specialized multiplication structures for Galois Field multiplication [15]. 10 different multiplier alternatives are explored and compared to a reference architecture. The different strategies for combining integer and Galois field multiplication show area savings up to 20% with only a marginal increase in delay and an increase in power consumption of 25%.

Furthermore, Bahram Rashidi proposed a modified retiming serial multiplier for finite impulse response (FIR) digital filters based on ring topologies [16]. The work involved additional logic which allowed for modification of the scheduling of the FIR filter computation, allowing the number of compute cycles

to be reduced from 32 to 24. To further improve performance of the FIR filter computation, the author proposed a high-speed logarithmic carry look ahead adder to work in combination with a carry save adder.

While the TSM is suited for machine learning and applications with high degrees of sparsity, it differs from the previous research in that the multiplier performs standard signed multiplication and can be used in any application. Our contribution is a new control structure for performing multiplication that dynamically avoids unnecessary computation.

B. Previous Work on Reduced Precision Multiplication for Neural Networks

The most comparable work to this multiplier is the parallel-serial, or shift-add, multiplier. As described in Equation 2, the product p is iteratively calculated by examining individual bits of X each cycle and accumulating a scaled Y [1].

Recent work in bit and digit serial multiplication for FPGAs has focused on on-line arithmetic [17] and efficient mapping of the algorithms to the FPGA architecture. Shi et. al. [18] analysed the effect of overclocking radix-2 on-line arithmetic implementations and quantified the error introduced by timing violations. They found a significant reduction in error for DSP based applications compared with conventional arithmetic approaches. Zhao et. al. [19] presented a method for achieving arbitrary precision operations utilising the on-chip block RAMs to store intermediate values.

In the domain of neural networks, Judd et. al. [7] presented a bit-serial approach for reduced precision computation. They showed a 1.3x to 4.5x performance improvement over classical approaches as their arithmetic units only perform the necessary computation for the particular bit width.

III. Radix-4 Booth Multiplication

This section reviews the radix-4 Booth algorithm [1], an extension to the parallel-serial multiplier. This computes $x \times y$

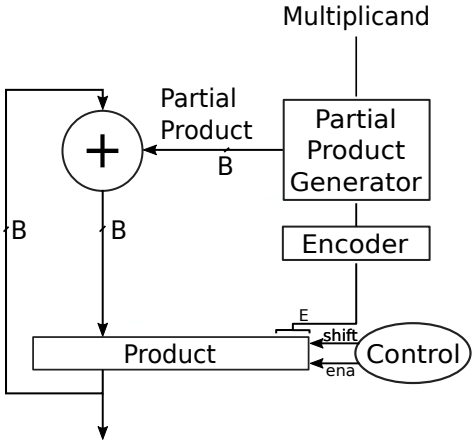


Fig. 2: n bit Serial Multiplier. There are five key components to the standard radix-4 serial Booth multiplier: the shifter, encoder, partial product generator, control and adder. As the partial results are generated in the adder, they are accumulated in the n most-significant bits of the product register.

TABLE I: Booth Encoding

Y_{i+2}	Y_{i+1}	Y_i	e_i
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	2
1	0	0	$\bar{2}$
1	0	1	$\bar{1}$
1	1	0	$\bar{1}$
1	1	1	0

$\bar{2}$ and $\bar{1}$ represent $\quad 2$ and $\quad 1$ respectively.

where x and y are n bit two's complement numbers (the multiplicand and multiplier respectively); producing a $2n$ two's complement value in the product p. The multiplication algorithm considers multiple digits of Y at a time and is computed in N partitions where,

$$N = \lfloor \frac{n+2}{2} \rfloor. \tag{5}$$

An equation describing the computation is given by:

$$p = (Y_1 + Y_0)x + \sum_{i=1}^{N-1} 2^{2i-1}(Y_{2i+1} + Y_{2i} - 2Y_{2i-1})x, \tag{6}$$

Following the notation in Section II, Y denotes the length-N digit-vector of the multiplier y. The radix-4 Booth algorithm considers 3 digits of the multiplier Y at a time to create an encoding e given by:

$$e_i = Y_{2i+1} + Y_{2i} - 2Y_{2i-1}, \tag{7}$$

where i denotes the ith digit. As illustrated in Table I, apart from $Y_{i+2}Y_{i+1}Y_i = 000$ and $Y_{i+2}Y_{i+1}Y_i = 111$ which results in a 0, the multiplicand is scaled by either 1, 2, -2 or -1 depending on the encoding.

This encoding e_i is used to calculate a partial product $PartialProduct_i$ by calculating:

$$PartialProduct_i = e_i x = (Y_{2i+1} + Y_{2i} - 2Y_{2i-1})x, \tag{8}$$

This $PartialProduct$ is aligned using a left shift (2^{2i-1}) and the summation is performed to calculate the final result p. Since the Y_{-1} digit is non-existent, the 0th partial product $PartialProduct_0 = (Y_1 + Y_0)x$. A serial (sequential) version of the multiplication is performed by computing each partial product in N cycles:

$$\begin{aligned} p[0] &= 2^{n-2}(Y_1 + Y_0)x, \\ p[j+1] &= 2^{-2}(p[j] + 2^n(Y_{2j+1} + Y_{2j} - 2Y_{2j-1})x) \\ &\qquad j = 1, \ldots, N-1, \\ p &= p[N], \end{aligned} \tag{9}$$

To better explain the two speed optimisation presented in the next section, Equation 9 is represented as an algorithm in Algorithm 1 and illustrated in Figure 2. Two optimisations are performed to allow for better hardware utilisation. Firstly, the product p is assigned the multiplier y ($p = y$), this removes the need to store y in a separate register and utilises the n least-significant bits of the p register. Consequently, as the product

Algorithm: Booth Radix-4 Multiplication

Data: y: Multiplier, x: Multiplicand
Result: p: Product
$p = y$;
$e = (P[0] - 2P[1])$;
for $count = 1$ **to** N **do**
  $PartialProduct = e * x$;
  p = sra(p,2);
  $P[2 * B - 1 : B] += PartialProduct$;
  $e = (P[1] + P[0] - 2P[2])$;
end

Algorithm 1: x, y are n bit two's complement numbers, p denotes the $2n$ two's complement result, and sra (the shift right arithmetic function). y is assigned to the n least-significant bits of p, hence the encoding, E, can be calculated directly from P.

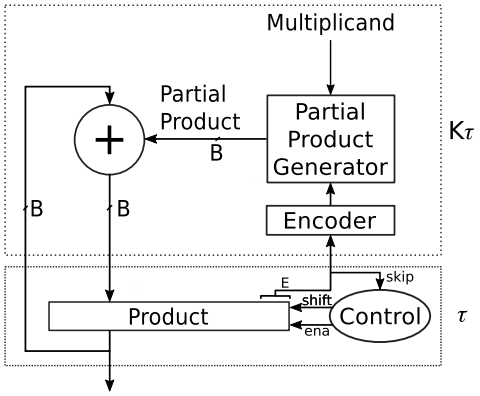


Fig. 3: n bit Two Speed Multiplier. This contains an added control circuit for skipping and operating with two different delay paths.

p is shifted right (p=$sra(p, 2)$), the next encoding e_i can be calculated from the three least-significant bits (LSBs) of p. The second optimisation removes the realignment left shift of the partial product (2^n) by accumulating the $PartialProduct$ to the n most-significant bits of the product p ($P[2*B-1 : B]$ += $PartialProduct$).

IV. Two Speed Multiplier

This section presents the TSM which is an extension to the serial Booth multiplication algorithm and implementation. The key change is to partition the circuit into two paths; each having critical paths, τ and $K\tau$ respectively (see Figure 3). The multiplier is clocked at a frequency of $\frac{1}{}$, where the $K\tau$ region is a fully combinatorial circuit with a delay of $K\tau$. K is the ratio of the delays between the two subcircuits. $K = \lceil K \rceil$ is the number of cycles needed for the addition to be completed before storing the result in the product register; used in the hardware implementation of the multiplier.

As illustrated in Algorithm 2, before performing the addition, the encoding, e, (the three least-significant bits of the product) is examined and a decision is made between two cases: (1) The encoding and $PartialProduct$ are zero and $0x$, respectively, or (2) the encoding is non-zero. These two cases can be distinguished by generating:

$$skip = \begin{cases} 1, & \text{if } P[2:0] \in \{000, 111\}, \\ 0, & \text{otherwise,} \end{cases} \tag{10}$$

When $skip = 1$ only the right shift and cycle counter accumulate need to be performed, with a critical path of τ. In the case of a non-zero encoding ($skip = 0$), the circuit is clocked K times at τ. This ensures sufficient propagation time within the adder and partial product generator, allowing the product register to honour its timing constraints. Hence the total time T taken by the multiplier can be expressed as Equation 11, where N is defined by Equation 5, and O is the number of non-zero encodings in the multiplier's Y digit-vector.

$$T(O) = (N - O)\tau + OK\tau, \tag{11}$$

The time taken to perform the multiplication is dependent on the encoding of the bits within the multiplier y. The upper and lower bound for the total execution time occurs when $O = N$ and $O = 0$ respectively. From Equation 11, the max and min are:

$$N\tau \leq T \leq NK\tau, \tag{12}$$

The input that results in the minimum execution time is when $y = 0$. In this case all bits within the multiplier are 0, and every three LSB encoding results in a $0x$ scaling and $O = 0$. There are a few input combinations that result in the worst case, $O = N$. One case would be a number of alternating 0 and 1, ie. 1010101..10101..10101. In this case, each encoding results in a non-zero $PartialProduct$.

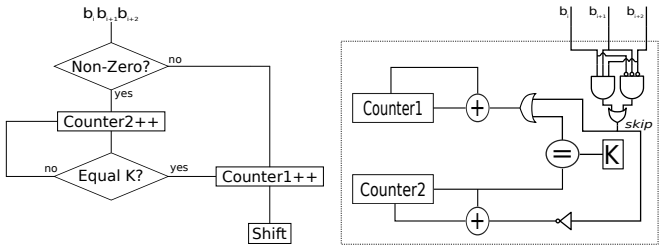
Algorithm: Two Speed Booth Radix-4 Multiplication

Data: y: Multiplier, x: Multiplicand
Result: p: Product
$p = y$;
$e = (P[0] - 2P[1])$;
for $count = 1$ **to** N **do**
  p = sra(p,2);
  // If non-zero encoding, take the $K\tau$ path, otherwise the τ path
  if $e \neq 0$ **then**
    // this path is clocked K times
    $PartialProduct = e * x$;
    $P[2 * B - 1 : B] += PartialProduct$;
  end
  $e = (P[1] + P[0] - 2P[2])$;
end

Algorithm 2: When $E = 0$, zero encodings are skipped and only the right shift arithmetic function is performed.

A. Control

As shown in Figure 4a and Figure 4b, the control circuit consists mainly of: one $\log_2(N)$ accumulator, one $\log_2(K)$ accumulator, three gates to identify the non-zero encodings



(a) Controller flowchart (b) Control Circuit

Fig. 4: Two counters are used to determine (a) when the multiplication is finished, and (b) when the result of the $K\tau$ circuit has been propagated.

Bit Representation	Action	Time	PartialProduct
1 1 1 1 0 1 0 0 0 1 0 0 0	skip	τ	$0x \times 2^{0}$
1 1 1 1 0 1 0 0 0 1 0	add	$\tau + \bar{K}\tau$	$1x \times 2^{2}$
1 1 1 1 0 1 0 0 0	skip	$2\tau + \bar{K}\tau$	$0x \times 2^{4}$
1 1 1 1 0 1 0	add	$2\tau + 2\bar{K}\tau$	$1x \times 2^{6}$
1 1 1 1 1 0	add	$2\tau + 3\bar{K}\tau$	$-1x \times 2^{8}$
1 1 1	skip	$3\tau + 3\bar{K}\tau$	$0x \times 2^{10}$

Fig. 5: Control example: Non-zero encodings result in an "add" action taking $K\tau$ time, whereas zero encodings allow the "skip" action, taking τ time. For the first encoding, only the two least-significant bits are considered with a prepended 0 as described in Section III.

and a comparator. $Counter2$ is responsible for counting the number of cycles needed for the addition without violating any timing constraints, i.e, K. When the encoding is non-zero, $Counter2$ is incremented. $Counter1$ accumulates the number of encodings that have been processed. As shown in Section III, the number of cycles needed to complete a single multiplication is N, therefore the accumulator and $Counter1$ needs to be $\log_2(N)$ bits wide. $Counter1$ is incremented when the comparator condition has been met, $Counter2 = K$, or a zero encoding is encountered. When $Counter1$ increments, the signal is given to perform the right shift.

The control needs to distinguish between the zero and non-zero encodings. It contains a three gate circuit, performing Equation 10; taking in the three LSBs of the multiplier y. Two cases of zero encoding exist. The three gates are designed to identify these non-zero encodings; an inverter is connected to the accumulator of $Counter2$, incrementing, in these cases.

B. Example

Figure 5 provides an example of the control operating in the multiplier and the time taken to perform the multiplication. Each cycle, the three least-significant bits of the multiplier y are examined and an action is generated based on their encoding. Since 000 results in a $0x$ partial product, the first action is a "skip" and only the right shift is performed in τ time. The next three bit encoding, 010, is examined and results in a $1x$ partial product. This generates the "add" action in which $Counter2$ is accumulated to K and the product register is held constant. After $K\tau$ time, the value stored in the register has had enough time to propagate through the adder and the result is latched in the product register without causing timing violations. The multiplier continues operating in this fashion until all bits of y have been processed and the final result produced. In Figure 5, the total time is $3\tau + 3K\tau$ since there are three "skips" and three "adds".

C. Set Analysis and Average Delay

Given an input set D of length l and a function $f(y)$ (given by Equation 13) that calculates the number of non-zero encodings for a given multiplier y, the probability distribution p of encountering a particular encoding can be calculated by Algorithm 3.

$$f(y) = \neg(Y_1 \oplus Y_0) + \bigwedge_{i=1} (\neg(Y_{2i+1} \oplus Y_{2i}) \wedge \neg(Y_{2i} \oplus Y_{2i-1})), \quad (13)$$

where $\neg$, $\oplus$ and $\wedge$ are the logical 'not', 'xor' and 'and' symbols respectively.

Figure 6 shows the Gaussian and Uniform encoding probability distribution for 32-bits. There are significantly less numbers in the lower, non-zero encoding region compared with the higher, non-zero encoding region, resulting in increased computation time. However, as discussed in Section V, for other workloads, the distributions can shift and change depending on the problem and optimisation techniques used.

Using the probability p, the average delay of the multiplier can be calculated using Equation 14.

$$\mathbb{T} = \frac{1}{N} \bigwedge_{i=0} p(i)T(i), \quad (14)$$

where T is calculated using Equation 11 and $p(i)$ denotes the probability of encountering an encoded number with i non-zeros.

D. Timing

During standard timing analysis, the $K\tau$ path would cause a timing violation for the circuit operating at frequency $\frac{1}{}$. There are two ways to address this issue. The first involves a standard 'place and route' of each individual multiplier as it is instantiated in the design. An additional timing constraint is included to address the otherwise violated $K\tau$ path, allowing

Algorithm: Probability of a given encoding
Data: D: Input Set
Result: p: Product
$Count\{0, 1, \ldots, N\} = \{0\}$;
for $i = 0$ **to** l **do**
  $Count[f(X_i)] += 1$
end
$p = Count \oslash l$

Algorithm 3: Probability of encountering a particular encoding given an input data set, $\oslash$ denotes element-wise division.

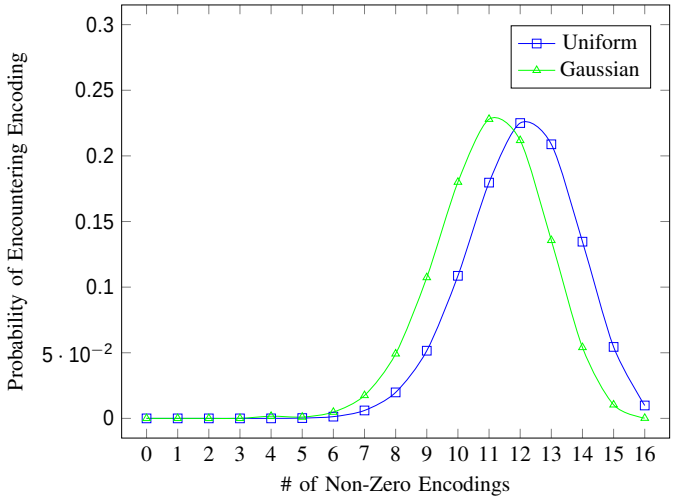


Fig. 6: $p(i)$ 32 bit distribution: the distribution of the frequency of particular non-zero encoded numbers for the Gaussian and Uniform distributions.

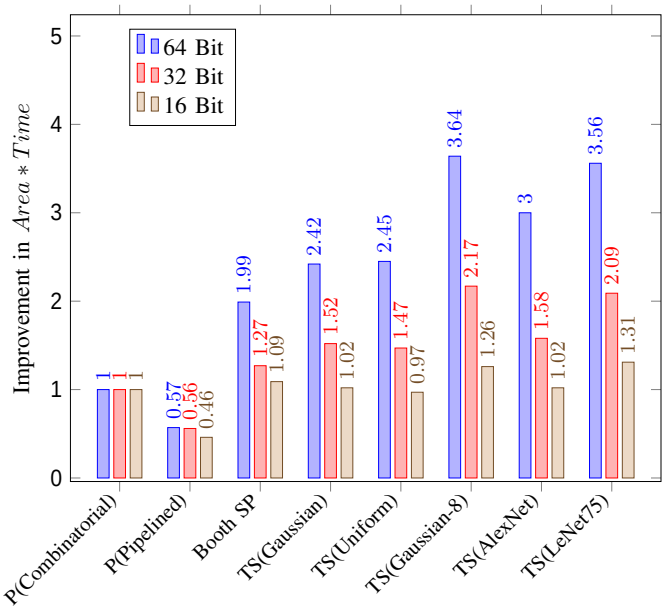


Fig. 7: The improvement in $Area * Time$ for 4 different multiplier configurations respectively. Five different sets are presented for the TSM.

timing driven synthesis and placement to achieve the best possible layout. The second option is to create a reference post-'place and route' block that is used whenever the multiplier is instantiated. This ensures each multiplier has the same performance and is placed in exactly the same configuration.

There are downsides to each option. The first option gives the tools freedom to place the blocks anywhere, however the performance of individual instantiations may differ if the $K\tau$ and τ sections cannot be placed at the same clock rate. For the second option, placing a reference block requires availability of free resources in the layout specified. While this ensures high performance, placing the reference block may become increasingly difficult as the design becomes congested.

V. Results

This section presents implementation results of the TSM. The multiplier is compared against the standard 64, 32 and 16 bit versions of parallel-parallel and serial-parallel multipliers. For all configurations tested up to 64 bits, the K scaling factor in the $K\tau$ subcircuit of Figure 3 was always less than two. This allows the comparison of K with a counter in Figure 4b to be simplified to a bit-flip operation.

A. Implementation Results

The area and delay of different TSM instantiations are given in Table II for an Intel Cyclone V 5CSEMA5U23C6 FPGA, with the results obtained using the Intel Quartus 17.0 Software Suite. During place and route the software performs static timing analysis across four different PVT corners, keeping voltage static. Specifically: (1) Fast 1100mv 0C, (2) Fast 1100mv 85C, (3) Slow 1100mv 0C and (4) Slow 1100mv 85C. The TSM was 'placed and routed' using the timing constraint based methodology and all frequencies reported for each multiplier represent the upper limit for each one considered as a standalone module. Unless otherwise specified, $Time$ is considered to be the result latency, and $Area$, the number of logic elements. The TSMs were evaluated using the Gaussian and Uniform sets, as they are important sets in machine learning applications, as well as two neural network weight sets.

All sets were generated in single precision floating-point and converted to fixed-point numbers. The integer length was determined by taking the maximum value of the set and allocating sufficient bits to represent it fully, hence saturation did not need to be performed. The number of fractional bits are the remaining bits after the integer portion has been accounted for. The Gaussian set was generated with a mean of zero and standard deviation of 0.1. For the Gaussian-8 set, the numbers were scaled such that they are represented in 8 bits. The uniform set was generated by selecting numbers between -1 and 1.

The neural network weight sets are from two convolutional neural networks, AlexNet [20] and a 75% sparse variant of LeNet [21], LetNet75, trained using the methodology presented by Han. et. al [22]. The Parallel(Combinatorial) and Parallel(Pipelined) multipliers are radix-4 Booth multipliers taken from an optimised FPGA library provided by the vendor and are designed for high performance [23]. Since the performance of a Parallel (Pipelined) multiplier is a function of its pipeline depth, the reported values are the best results from numerous configurations to ensure a fair comparison. The Booth Serial-Parallel (SP) multiplier also uses the radix-4 Booth algorithm, illustrated in Algorithm 1 whereas the TSM implements Algorithm 2.

Figure 7 presents the improvements in $Area * Time$ for the four different multipliers, with the Parallel(Combinatorial) illustrating baseline performance for each configuration. $Area * Time$ is an important metric for understanding architecture design attributes and the magnitude of possible tradeoffs be-

TABLE II: Multiplier Implementation Results

B	Type	*Area* (LEs)	Max Delay (ns)	Latency (Cycles)	*Power* (mW)
64	Parallel(Combinatorial)	5104	14.7	1	2.23
	Parallel(Pipelined)	4695	6.99	4**	9.62
	Booth Serial-Parallel	292	3.9	33	2.23
	Two Speed	304	1.83 ( )	45.2*	5.2
32	Parallel(Combinatorial)	1255	10.2	1	1.33
	Parallel(Pipelined)	1232	4.6	4**	5.07
	Booth Serial-Parallel	156	3.8	17	1.78
	Two Speed	159	1.76 ( )	25.6*	3.18
16	Parallel(Combinatorial)	319	6.8	1	0.94
	Parallel(Pipelined)	368	3.2	4**	3.49
	Booth Serial-Parallel	81	2.72	9	1.67
	Two Speed	87	1.52 ( )	14*	4.35

For Two Speed, the Max Delay represents the τ subcircuit and $K = 2$, hence 2τ is the delay of the adder subcircuit.
* This is the average latency over all of the tested sets.
** While the latency of the pipelined multiplier is four, the throughput is one.

tween area and speed [24]. The fixed cycle times of the Booth Serial-Parallel, Parallel(Combinatorial) and Parallel(Pipelined) multipliers result in the same performance regardless of the input set. However, the TSM is designed to take advantage of the input set and outperforms all other multipliers in the 32 bit and 64 bit configuration. In the 16 bit configuration, the TSM exhibited similar performance to the baseline.

The highest performing set is the 64 bit Gaussian-8; showing a speed up of 3.64x. For the Gaussian and Uniform sets, the 64 bit configuration provides a 2.42x and 2.45x improvement respectively. At 32 and 16 bits, the TSM's improvements range from 1.47-1.52x and 0.97-1.02x respectively. The Gaussian-8 set illustrates that inefficiencies introduced by using a lower bit representation are alleviated by the TSM; the majority of the most-significant bits are either all 0's in the positive case, or all 1's in the negative case, allowing multiple consecutive "skips".

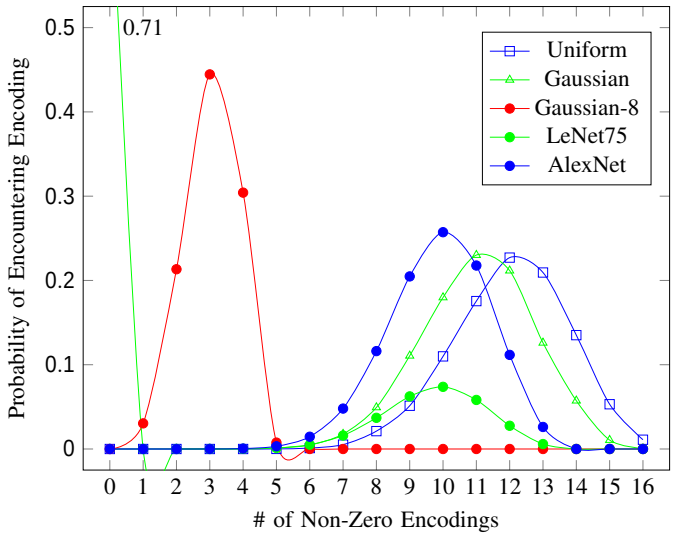


Fig. 8: $p(i)$ 32 bit set: The probability that y will be a particular encoding.

Figure 8 shows the probability distributions of the five problems tested at 32-bits. It illustrates the differences between the Gaussian, Uniform, AlexNet, Gaussian-8 and LeNet75 sets and why particular sets perform better than others. For Gaussian-8, the majority of the encoding is in the $2-4$ range, resulting in a significant number of "skips" for each input. While the non-zero numbers in the LeNet75 set contain high encoding numbers, the set also contains 71% zeros, therefore the majority of the computations are "skips".

B. Multiplier Comparison

Table III compares different multiplier designs in terms of six important factors: $Area$, $Time$, $Power$, $Area * Time$, $Time * Power$ and $Area * Time * Power$, with the specific application often dictating which is most appropriate. Typically, tradeoffs are analysed and the variant with the highest performance is chosen. For area, either the Booth Serial-Parallel or TSM are the best choices as they have the smallest footprint. Alternately, when both area and speed are factors, the TSM outperforms the Booth Serial-Parallel multiplier as illustrated in Table III and Figure 7. If area is not a concern, the Parallel(Combinatorial) multiplier may be preferred. When taking power into account, the Parallel(Combinatorial) multiplier outperforms the Parallel(Pipelined) multiplier.

As highlighted in Table III, in terms of $Area * Time * Power$, the Booth Serial-Parallel multiplier offers the highest performance and is 1.9x better on this metric than the Parallel(Combinatorial) multiplier for a bit width of 64. However the TSM still provides a sizeable improvement, achieving a 1.29x improvement on average, peaking at 1.5x for LeNet75 and Gaussian-8.

Figure 9 illustrates the $Area * Time$ trade-off as the bit-width is increased. For latency, the TSM has the lowest $Area * Time$ compared to the other multipliers. Calculating the $Area * Time$ with respect to throughput, shows that the Parallel(Pipelined) multiplier achieves a 1.84-2.29x performance improvement over the Parallel(Combinatorial) multiplier for bit widths 16, 32 and 64. These results are shown

TABLE III: Multiplier Performance Metrics - Latency and Throughput

B	Type	*Area* LEs	*Time* ns	*Power* mW	*Area* *Time*	*Time* *Power*	*Area* *Time* *Power*
64	Parallel(Combinatorial)	5104	**14.7**	2.23	75028	**32.78**	167314
	Parallel(Pipelined)	4695	27.96	9.62	131274 (32818)	268.98	315716
	Booth Serial-Parallel	**292**	128,7	2.23	37696	287.00	**84062**
	Two Speed*	**304**	82.71	5.2	**25187**	430.12	130972
32	Parallel(Combinatorial)	1255	**10.2**	1.33	12808	**13.57**	**17034**
	Parallel(Pipelined)	1232	18.4	5.07	22678 (**5667**)	93.29	114977
	Booth Serial-Parallel	**156**	64.6	1.78	10116	114.99	18007
	Two Speed*	**159**	45.05	3.18	**7186**	143.27	22852
16	Parallel(Combinatorial)	319	**6.8**	0.94	2169	**6.39**	**2039**
	Parallel(Pipelined)	368	12.8	3.49	4714 (**1177**)	44.67	16452
	Booth Serial-Parallel	**81**	24.48	1.67	1987	40.88	3319
	Two Speed*	**87**	21.28	4.35	**1851**	92.56	8053
-	(8*8 Signed) (90nm) [12]	160	160	58	25600	9280	1484800
	(16*16 Signed) (90nm) [16]	631	62.4	-	39374	-	-
	(32*32 Signed) (90nm) [13]	20319	65.10	106.87	1322949	6890	139997910

* Average over all tested sets, the individual results will change for specific applications.

in parentheses in the $Area * Time$ column as well as the Pipeline(Throughput) plot in Figure 9. The TSM still shows favourable results for both the Uniform and Gaussian sets, while outperforming on the Gaussian-8 and neural network sets.

To the best of our knowledge there are only three recent publications in the domain of FPGA micro-architecture multiplier optimisations, targeted at serial-parallel computation of the Booth algorithm [12], [13], [16]. All of these works were implemented on 90nm FPGAs, making a direct comparison difficult since they were not only slower and higher power consumption due to technology, their architecture was also different, e.g. they used four input lookup tables, and performance of the larger multipliers, such as 32-bit and 64-bit, were not reported. A fair comparison is thus impossible but the reported results are listed at the bottom of Table III, and we note that for the 16-bit and 32-bit cases the TSM improved $Area * Time$ and $Area * Time * Power$ by an order of magnitude. Both the Parallel(Combinatorial) and Parallel(Pipelined) multipliers are taken from libraries that implement the latest multiplier optimisations and serve as a good comparison between our work and the industry standard.

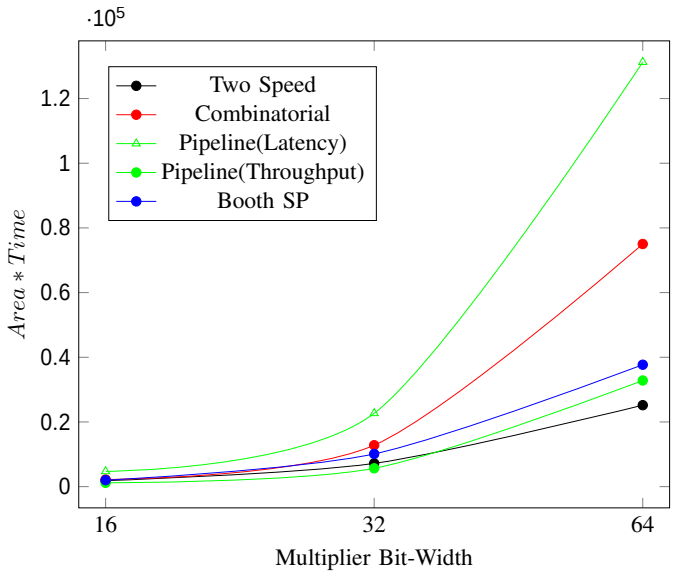


Fig. 9: (a) presents the classic encoder and partial product generator. (b) is the optimised version since the $0M$ calculations don't need to be performed

VI. Conclusion

In this work we presented a TSM, which is divided into two subcircuits, each operating with a different critical path. In real-time, the performance of this multiplier can be improved solely on the distribution of the bit representation. We illustrated for bit widths of 32 and 64, typical compute sets, such as Uniform and Gaussian and neural networks, can expect substantial improvements of 3x and 3.56x using standard learning and sparse techniques respectively. The cost associated with handling lower bit width representations, such as Gaussian-8 on a 64 bit multiplier are alleviated and show up to a 3.64x improvement compared to the typical Parallel multiplier. Future work will focus on techniques for constructing M to take full advantage of the Two Speed optimisation.

References

[1] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[2] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. New York, NY, USA: Oxford University Press, Inc., 2000.

[3] M. Ercegovac and T. Lang, *Digital Arithmetic*, ser. Morgan Kaufmann Series in Comp. Morgan Kaufmann, 2004.

[4] B. Dinesh, V. Venkateshwaran, P. Kavinmalar, and M. Kathirvelu, "Comparison of regular and tree based multiplier architectures with modified booth encoding for 4 bits on layout level using 45nm technology," in *2014 Intl. Conf. on Green Computing Communication and Electrical Engineering*, 2014, pp. 1–6.