Leap in the Formal Verification of Datapath

Theo A. Drane[1] and George A. Constantinides[2]
[1] *Imagination Technologies Ltd., Kings Langley, Hertfordshire, UK*
[2] *Department of Electrical and Electronic Engineering, Imperial College London, UK*

WHITE PAPER

Leap in the Formal Verification of Datapath

Theo A. Drane[1] and George A. Constantinides[2]
[1] *Imagination Technologies Ltd., Kings Langley, Hertfordshire, UK*
[2] *Department of Electrical and Electronic Engineering, Imperial College London, London, UK*

Abstract— The formal verification of datapath continues to prove a challenge to design and verification engineers. The majority of formal verification relies on algorithms operating at the bit level, whereas high level datapath optimizations are performed at a word level. This article shows how this disparity can be overcome by the application of recent research in the area of *polynomial* datapath. We step through a practical procedure, which can be applied directly to RTL code as a preprocessing step before formal verification tools are invoked. The procedure leads to orders of magnitude improvements in the execution time for commonly occurring problems in datapath verification.

Index Terms— front-end verification best practices, datapath, functional verification, RTL.

I. INTRODUCTION

Datapath implementation at an RTL level requires a hardware engineer to work from a high level description, whether specification documentation or a system level model, in order to create an optimized hardware design. Invariably these optimizations will only be confirmed by dynamic simulation between a system model and the RTL design. This is in stark contrast to the optimizations performed by RTL synthesis tools, for which it is an industry standard requirement to use bit level formal equivalence tools to confirm functional equality of the RTL and resultant gate level netlist. It is, however, these high level optimizations that are invariably error prone and generally impossible to exhaustively simulate. As a trivial example consider the identity in equation (1).

$$a^2 - b^2 \equiv (a-b)(a+b) \qquad (1)$$

An industry standard bit level formal verification tool was unable to obtain a proof of this identity, where a and b are unsigned 16 bit integers, in any reasonable time frame. One could argue that reformulations like those in equation (1) should not fall under the remit of RTL formal verification, given that this identity holds over the real numbers, such identities can be proven by algebraic techniques. However, optimizations that rely on the finite bit widths of the arithmetic involved fall squarely under the remit of RTL formal verification. For example consider equations (2) and (3)

$$y_2[2:0] = x^2\left(x^2 - 1\right) \quad (2) \qquad y_3[2:0] = 2x\left(x^2 - 1\right) \quad (3)$$

Where x is an unsigned integer of arbitrary bit width and y_2 and y_3 are both three bits in length. Surprisingly y_2 and y_3 are functionally equivalent, regardless of the size of the input x; this is due to the finite size of the output. To further motivate the discussion that follows, consider the functions found in equations (4) and (5):

$$y_4 = \begin{array}{l} b[1]\,?\,a[8]+a[7]+a[6]: \\ b[0]\,?\,a[5]+a[4]+a[3]: \\ a[2]+a[1]+a[0] \end{array} \qquad (4) \qquad y_5 = a[8:0] << b[1:0] \qquad (5)$$

Now equation (4) is a function of the bits of a and b, whereas equation (5) maintains the word nature of the inputs. As such, intuition would dictate that, to gain confidence in the correctness of the designs, fewer test vectors would be required for y_4 than y_5. Putting this *feeling* on a firmer footing is the subject of this article.

In Section II we will expound the powerful and elegant theory of *polynomial* datapath equivalence as put forward by Shekhar *et al.* in [2]. These stunning results are based upon results in finite field algebra. Section III contains the authors' contribution, which shows how the Shekhar *et al.* results can be used to prove formal equivalence of a range of datapath designs by developing a methodology that requires *super* usage of industry standard bit level equivalence tools. In doing so, we attain proofs that are leaps and bounds beyond current tool capacity.

Our contributions are:

- Observations on the range of datapath that is actually polynomial in nature, including various numbers formats and fundamental datapath operators.
- Algorithm to discover the polynomial nature of an arbitrary datapath design.
- Waterfall formal verification methodology using the Shekhar *et al.* results and previously mentioned algorithm.

II. POLYNOMIAL DATAPATH

A. Definition

A datapath design is said to be *polynomial* if its inputs are unsigned integers and the function can be written as a polynomial with integer coefficients reduced modulo 2^n, where *n* is the output bit-width. Obviously integer adders and multipliers are *polynomial* but it would appear that the vast majority of industry standard datapath would never fall under this category. However, quite a few very common operators and number formats are in fact polynomial in nature. For example, for *n* bit integer inputs *a* and *b* we have:

Signed Number *a*	$-2^{n-1}a[n-1]+a[n-2:0]$	(6)
Sign Magnitude Number *a*	$(-1)^{a[n-1]}a[n-2:0]=(1-2a[n-1])a[n-2:0]$	(7)
Muxing	$s\,?\,a:b=sa+(1-s)b$	(8)
Inversion	$\bar{a}=2^n-1-a$	(9)
Left Shift	$a<<b=a\left(\left(2^{2^{n-1}}-1\right)b[n-1]+1\right)..(3b[1]+1)(b[0]+1)$	(10)

B. Frequency of Occurrence

So, in fact, *polynomial* datapath is not so uncommon. Is all datapath *polynomial* in some way? Well consider the following function:

$$y_6[0:0] = (a[1:0]>b[0:0])\,?1:0 \quad (11)$$

$$= \left(\frac{1}{6}a(a-1)(7-2a)+\frac{1}{2}a(a-2)(a-3)(1-b)\right)\bmod 2 \quad (12)$$

$$= \left(a[1]+a[0]-a[1]a[0]-a[0]b+a[0]a[1]b\right)\bmod 2 \quad (13)$$

Design y_6 is a comparison between a 2 bit and a 1 bit input. Equation (12) shows how this function can be written as a polynomial expression in a and b however it has rational coefficients, so it is not *polynomial* in its inputs. However equation (13) shows how y_6 is polynomial in the bits of its inputs. In fact all datapath can be written as a polynomial with respect to its inputs; however this will invariably require rational coefficients. Moreover all datapath is polynomial in its input bits. Why? Well all datapath can be expressed as a composition of *NAND* functions (due to universality of *NAND*s), but Equation (14) shows that these functions themselves are *polynomial;* so we conclude that all datapath is *polynomial* in its input bits.

$$NAND(a,b)=1-ab \quad (14)$$

C. *Formally Proving Equivalence of Polynomials*

We now tease out the striking results on the conditions for the formal equivalence of polynomial datapath, which is extracted from [2]. Let's say we have two datapath designs whose formal equivalence we need to ascertain. That is, we need to prove, or find a counter example to (where x_i are the inputs to the two functions):

$$f_1[n-1:0] \equiv f_2[n-1:0] \qquad \forall x_i \quad (15)$$
$$f_1[n-1:0] - f_2[n-1:0] = 0 \qquad \forall x_i \quad (16)$$
$$f_1 - f_2 = 0 \bmod 2^n \qquad \forall x_i \quad (17)$$

Given that the two datapath designs can be expressed as polynomials, we are interested in knowing if the polynomial $f_1 - f_2$ returns zero modulo 2^n for all possible inputs. If so, we call $f_1 - f_2$ a *vanishing* polynomial. So what do these vanishing polynomials look like? How do we know if the difference between two designs can be written as a polynomial that vanishes?

D. *Vanishing Polynomials*

In an attempt to find vanishing polynomials, consider the fact that the product of consecutive numbers are always divisible by 2, hence:

$$x(x-1) = 0 \bmod 2 \qquad \forall x \in \mathrm{Z} \quad (18)$$

Similarly looking at the products of consecutive numbers gives rise to the following observations:

$$x(x-1)(x-2) = 0 \bmod 2 \qquad \forall x \in \mathrm{Z} \quad (19)$$
$$x(x-1)(x-2)(x-3) = 0 \bmod 2^3 \qquad \forall x \in \mathrm{Z} \quad (20)$$
$$x(x-1)(x-2)(x-3)(x-4)(x-5) = 0 \bmod 2^4 \qquad \forall x \in \mathrm{Z} \quad (21)$$

This can be generalized to:

$$x(x-1)(x-2)\ldots(x-SF(2^n)+1) = 0 \bmod 2^n \qquad \forall x \in \mathrm{Z} \quad (22)$$

Where *SF(2^n)* is the least number *k* such that the product of *k* consecutive integers is always divisible by *2^n*, this is the *Smarandache* function. This can be computed succinctly by using the Hamming weight function, denoted here as *Hamm(k)*, which counts the number of ones in the binary representation of *k*.

$$SF\left(2^n\right) = \min\left(k : 2^n \mid k!\right) \quad (23)$$
$$= \min\left(k : n \le k - Hamm(k)\right) \quad (24)$$

Based upon these observations we can attempt to formulate the most general vanishing polynomial in the case when *n*=3:

$$\begin{aligned} & g(x)x(x-1)(x-2)(x-3) \\ & \quad + 4Ax(x-1)(x-2) \\ & \quad\quad + 4Bx(x-1) \\ & \quad\quad\quad + 8Cx \\ & \quad\quad\quad + 8D \end{aligned} \qquad = 0 \bmod 2^3 \quad \forall x \in Z \quad (25)$$

Where *g(x)* is an arbitrary polynomial and A, B, C & D are arbitrary integers. In fact this is the most general vanishing polynomial modulo 2^3, [1]. How can we tell if a given polynomial vanishes? Well that would mean we could rewrite it in the form of Equation (25), *i.e.*:

$$f_1(x) - f_2(x) = \begin{aligned} & g(x)x(x-1)(x-2)(x-3) \\ & \quad + 4Ax(x-1)(x-2) \\ & \quad\quad + 4Bx(x-1) \\ & \quad\quad\quad + 8Cx \\ & \quad\quad\quad + 8D \end{aligned} \qquad (26)$$

All we need do is solve for A, B, C & D. We can simply substitute values for *x*:

$$\begin{aligned} f_1(0) - f_2(0) &= 8D \\ f_1(1) - f_2(1) &= 8C + 8D \\ f_1(2) - f_2(2) &= 8B + 16C + 8D \\ f_1(3) - f_2(3) &= 24A + 24B + 24C + 8D \end{aligned} \qquad (27)$$

We are expecting f_1 and f_2 to be functionally identical so we would expect these functions to match for *x=0,1,2,3*. If they do match for *x=0,1,2,3* then solving the set of equations in Equation (27) would give rise to *A=B=C=D=0* and we would then conclude that $f_1(x)-f_2(x)$ does indeed vanish for all *x*. This is a very surprising result, to reiterate:

$$f_1(x) = f_2(x) \bmod 2^3 \quad \forall x \in Z \quad \Leftrightarrow \quad f_1(x) = f_2(x) \bmod 2^3 \quad x \in [0,1,2,3] \quad (28)$$

This formalizes the notion we had in the introduction, we need only check the first few values for x, after which the polynomials will always be identical. Now we have a way of finding out the least number of inputs we need to check. We can generalize Equation (28) to:

$$f_1(x) = f_2(x) \bmod 2^n \quad \forall x \in Z \quad \Leftrightarrow \quad f_1(x) = f_2(x) \bmod 2^n \quad x \in [0,1,2,..., SF\left(2^n\right)-1] \quad (29)$$

This result can be further generalized to multivariate polynomials, see [2] for details.

III. POLYNOMIAL DATAPATH FORMAL VERIFICATION METHODOLOGY

A. *Introduction*

From Section II we saw that polynomial datapath covers a significant amount of datapath designs, moreover we saw that formally proving two polynomial designs as being equivalent only requires checking a reduced set of inputs. How can we exploit these elegant results from Shekhar *et al.*? The contribution of this article is the development of a methodology that shows how the results from Section 2 and existing formal equivalence checking tools can be combined to provide significant orders of magnitude improvement in verification runtimes. The remainder of this section provides the details of this novel *polynomial* datapath formal verification methodology.

The game plan is to first establish in which way two designs are polynomial, then perform a traditional formal verification but with reduced bit widths in line with Equation (29). It is worth noting that we do not need to form the polynomial description of the designs in question, only establish the polynomial nature of the inputs, this is in contrast to the work in [3].

B. *Data-Flow Graph*

We can assess the polynomial nature of a datapath design by first forming an augmented data-flow graph (DFG). Take for example the design described in Equation (30):

$$\begin{aligned} t0[16:0] &= c[15:0] + 1 \\ t1[31:0] &= b[15:0] - t0 \\ t2[31:0] &= d[15:0] << t0 \\ t3[31:0] &= a[15:0] * t1 \\ y[31:0] &= t3 + t2 \end{aligned} \qquad (30)$$

We create a directed graph whose vertices are the inputs, outputs, and operators of the design in question, Figure 1 shows the resultant DFG for the design from Equation (30). The edges of the directed graph correspond to the interconnecting signals of the design, connecting inputs, outputs, and operators as necessary, with the appropriate direction.

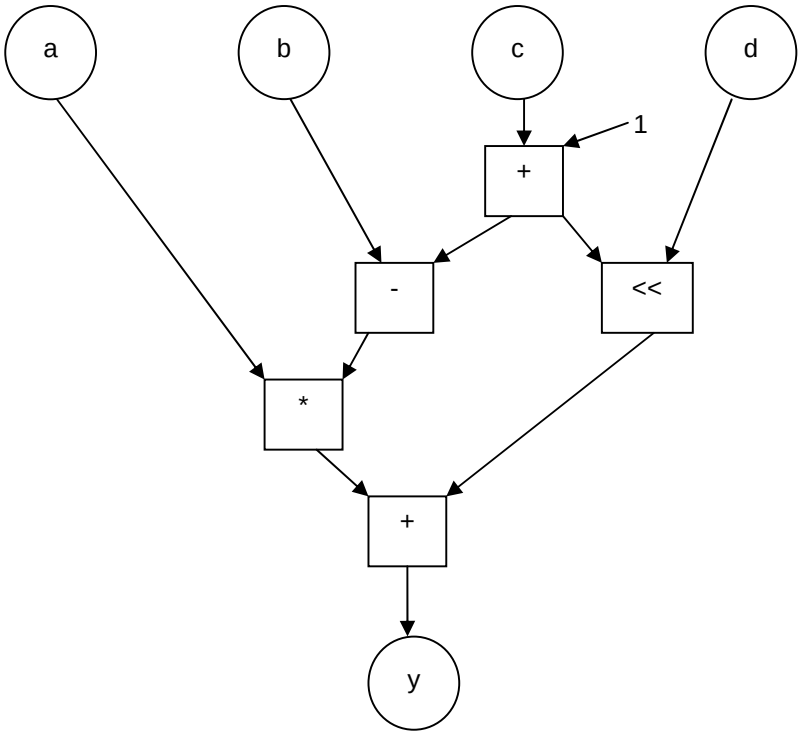

Figure 1: Example Data-Flow Graph.

To work out the polynomial nature of this design we first recall the polynomial behaviour of each operator node, *e.g.*

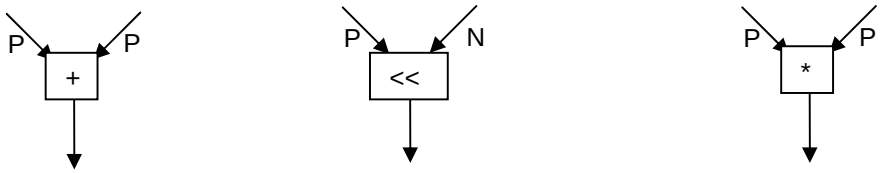

Figure 2: Polynomial Behaviour of Operators.

Where P/N labels polynomial/non polynomial respectively. Multiplication and addition are polynomial in both inputs. Left shift is polynomial in the shifted value but non polynomial in the shift value. Using the polynomial nature of each operator we can build up the polynomial nature of the entire design. The steps required in establishing the polynomial nature of the inputs is found in Algorithm 1:

Inputs: Data Flow Graph and pre calculated operator polynomial behaviour
Output: Labelled Data Flow Graph and Inputs

Step 1: Label the outputs as 'P'

Step 2: **For** every node for which all outputs are labelled **then**
If all outputs are labelled 'P'
Then label the inputs as per the known operator's polynomial behaviour
Else label all the inputs as 'N'

Step 3: Repeat Step 2 until all edges are labelled

Step 4: If all edges from an input are labelled 'P'
Then label the input 'P'
Else label the input 'N'

Algorithm 1

By applying this to Figure 1 we get Figure 3.

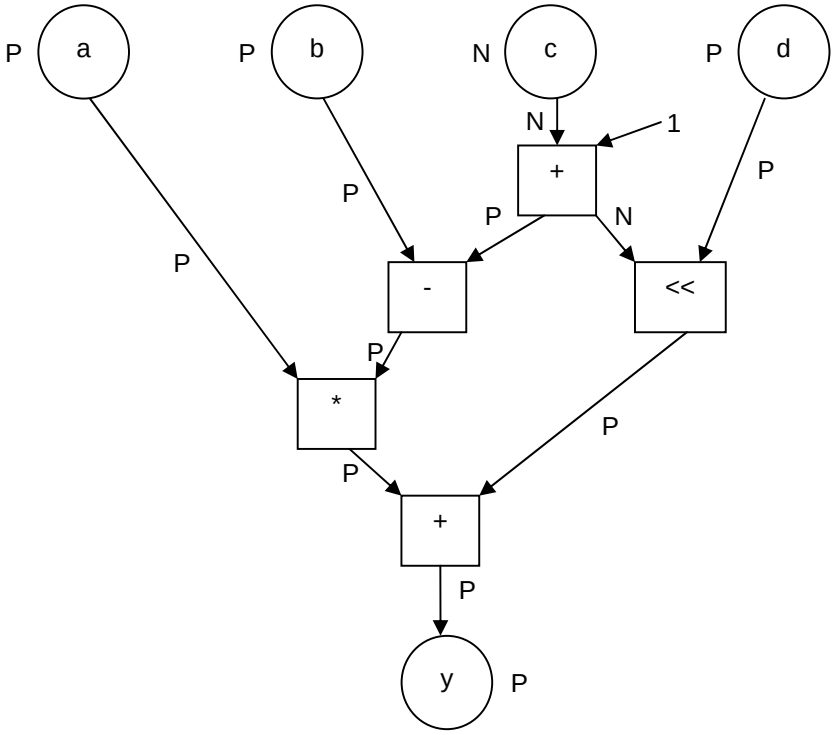

Figure 3: Augmented Data-Flow Graph.

In this case we would say that this particular design is polynomial only with respect to inputs a, b, and d. Hence we have a method of determining the polynomial nature of each input.

C. The Waterfall Verification

For the verification method to be viable the datapath must be extractable into a DFG with unsigned inputs, this means that there must be no bit slicing of the inputs, *i.e. unbroken* inputs. Moreover the DFG does not contain internal bit widths; hence these must not affect functionality. So the first step in a waterfall verification would be between an original design A, and a modified design A' which has unsigned and unbroken inputs and internal bit widths which are the same as the output width. So, for example in the case of Equation (30) we would derive a design based upon Equation (31):

$$\begin{aligned} t0[31:0] &= c[15:0] + 1 \\ t1[31:0] &= b[15:0] - t0 \\ t2[31:0] &= d[15:0] << t0 \\ t3[31:0] &= a[15:0] * t1 \\ y[31:0] &= t3 + t2 \end{aligned} \qquad (31)$$

Now this design has a 32 bit output and is polynomial in the 16 bit inputs *a, b,* and *d* and in all the inputs bits of input *c*. So according to Equation (29) we need only check the first $SF(2^{32})=34$ values on each of the inputs in which the design is polynomial in. So we may in fact restrict the sizes of *a*, *b,* and *d to being* 6 bits in length. So we make the design A" as described in Equation (32):

$$\begin{aligned} t0[31:0] &= c[15:0] + 1 \\ t1[31:0] &= b[5:0] - t0 \\ t2[31:0] &= d[5:0] << t0 \\ t3[31:0] &= a[5:0] * t1 \\ y[31:0] &= t3 + t2 \end{aligned} \qquad (32)$$

Figure 4 illustrates the complete verification process when formally verifying designs A against B.

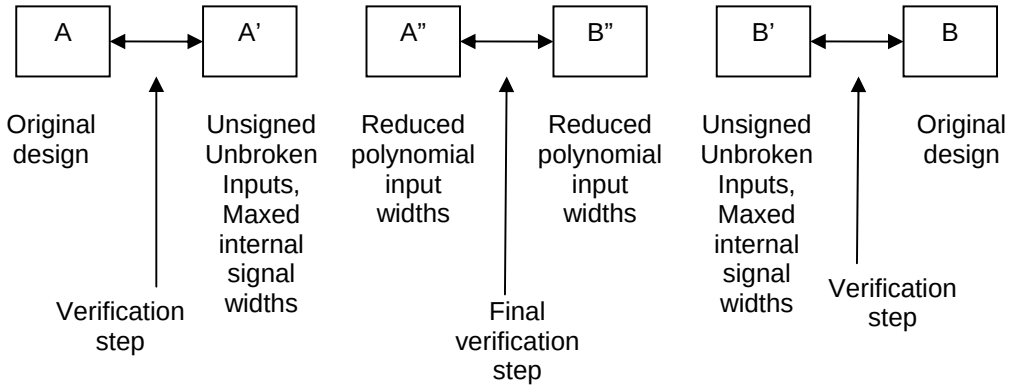

Figure 4: Formal Verifications Required.

The precise steps of the verification methodology are:

1. The first step is to produce designs *A*' and *B*' from which a DFG can be created as described in Section III. C. The resultant DFG has unsigned inputs, all internal signals are of bit width n and no inputs are bit sliced during the DFG. If a signal is sliced in two during the design, then two inputs should be created. If an input is in floating point format, then it will need to be split into sign, exponent and mantissa. Sign magnitude inputs will need their most significant bit separated into a new input. Twos complement signed numbers will similarly need their most significant bit separated into a new input. The result should be designs *A*' and *B*' whose inputs are unsigned, internal bit widths all equal the largest output bit width and inputs are used in their entirety without being bit sliced. If the inputs to *A*' and *B*' now differ, the fewest number of new inputs are created such that no bit splicing occurs in the two designs. A formal verification using the standard tools will then be performed between *A* and *A*' as well as between *B* and *B*'. If either of these fail then the method is not applicable to the given verification.

2. Secondly create a DFG from *A*' and *B*', apply Algorithm 1 to each DFG. The inputs which have been labeled as 'P' on both DFGs are then defined as *strictly polynomial*.

3. For each strictly polynomial input with width w_j compute where *n* is the maximum output width:

$$\lambda_j = \min\left(w_j, ceil\left(\log_2\left(SF\left(2^n\right)\right)\right)\right)$$

4. Create *A"* and *B"* which are identical to *A'* and *B'* except for the fact that the strictly polynomial inputs are reduced in size to their corresponding width λ_j. If the verification between *A"* and *B"* succeeds then the designs *A* and *B* are formally equivalent, otherwise they are not.

IV.WORKED EXAMPLE AND RESULTS

To complete the example introduced in Equation (30) consider designs A and B:

Design A

$$t0[16:0] = c[15:0] + 1$$
$$t1[31:0] = b[15:0] - t0$$
$$t2[31:0] = d[15:0] << t0$$
$$t3[31:0] = a[15:0] * t1$$
$$y[31:0] = t3 + t2$$

Design B

$$t0[16:0] = c[15:0] + 1$$
$$t1[31:0] = d[15:0] << c[15:0]$$
$$t2[31:0] = a[15:0] * b[15:0]$$
$$t3[31:0] = a[15:0] * t0$$
$$y[31:0] = t2 - t3 + 2 * t1$$

Following Section III. C. we derive A' and B':

Design A'

$$t0[31:0] = c + 1$$
$$t1[31:0] = b - t0$$
$$t2[31:0] = d << t0$$
$$t3[31:0] = a * t1$$
$$y[31:0] = t3 + t2$$

Design B'

$$t0[31:0] = c + 1$$
$$t1[31:0] = d << c$$
$$t2[31:0] = a * b$$
$$t3[31:0] = a * t0$$
$$y[31:0] = t2 - t3 + 2 * t1$$

The DFG of A' can be found in Figure 1. By applying Algorithm 1 to the DFG we found the strictly polynomial inputs to be a, b and d. The output bit width in this case is 32 so we need to compute *SF(2^{32})*:

$$SF\left(2^{32}\right) = \min\left(k : 32 \le k - Hamm(k)\right) = 34$$
$$\lambda_j = \min\left(16, ceil\left(\log_2\left(SF\left(2^{32}\right)\right)\right)\right) = 6$$

Now we can derive the designs with reduced bit width, A" and B":

Design A"

$$t0[31:0] = c[15:0] + 1$$
$$t1[31:0] = b[5:0] - t0$$
$$t2[31:0] = d[5:0] << t0$$
$$t3[31:0] = a[5:0] * t1$$
$$y[31:0] = t3 + t2$$

Design B"

$$t0[31:0] = c[15:0] + 1$$
$$t1[31:0] = d[5:0] << c[15:0]$$
$$t2[31:0] = a[5:0] * b[5:0]$$
$$t3[31:0] = a[5:0] * t0$$
$$y[31:0] = t2 - t3 + 2 * t1$$

Formally verifying A" against B" is a much simpler verification than A against B. SF(2^n) is of order n, thus reducing exponential complexity with linear. An industry standard formal equivalence tool was used to attempt the waterfall proof as well as the original A versus B formal verification, the results can be found in Table 1:

Table 1: Waterfall Formal Verification Runtimes.

Reference	Implementation	Formal Verification Runtime (s)
A	B	Unfinished after 24 hrs
A	A'	0.80
A"	B"	123.24
B'	B	37.72

V. CONCLUSION

This article has shown how recent research into polynomial datapath can be used to formally verify word level datapath optimizations by performing a waterfall verification, which is currently infeasible using industry standard formal equivalence tools. Future work would include automating this methodology on extracted polynomial datapath elements from the two designs.

REFERENCES

[1] D. Singmaster. On polynomial functions (mod m). *Journal Number Theory*, vol. 6, pages 345–352, 1974.

[2] N. Shekhar, P. Kalla, M. B. Meredith, and F. Enescu. Simulation bounds for equivalence verification of polynomial datapaths using finite ring algebra. *IEEE Transactions on VLSI Systems*, vol. 16, no. 4, pages 376-387, 2008.

[3] D. Li, Z. Fan and X. Yang. Abstraction of polynomial functions from arithmetic transform for fixed-size arithmetic datapath. 2nd International Conference on Information Engineering and Computer Science (ICIECS), pages 1-4, 2010.