Arithmetic Transforms for Verifying Compositions of Sequential Datapaths

Katarzyna Radecka and Zeljko Zilic
McGill University, Montreal, Canada
{kasiar,zeljko}@macs.ece.mcgill.ca

ABSTRACT

In this paper, we address the issue of obtaining compact canonical representations of datapath circuits with sequential elements, for the purpose of equivalence checking. First, we demonstrate the mechanisms for efficient compositional construction of Arithmetic Transform (AT), which is the underlying function representation, used in modern word-level decision diagrams. Second, we introduce a way of generating the canonical transforms of the sequential datapath circuits. Using these principles, we verify by AT highly sequential Distributed Arithmetic (DA) architectures.

1. INTRODUCTION

Design verification aims at providing an answer to the question: "Does the implemented circuit conform to the specification?" Verifications are often conducted through equivalence checking. Under this scenario, a canonical representation of the circuit is constructed and compared against the description obtained from the specification.

Original equivalence checking methods that relied on graph-based Binary Decision Diagram representations were unable to deal with arithmetic circuits such as multipliers. However, efficient word-level graphs like Binary Moment Diagrams (BMDs) [4] and their generalizations [5] can overcome this obstacle. Due to their word-level nature, the construction from binary signals is possible by backwards circuit traversal [6], but is generally quite tedious. Further, sequential circuits including arithmetic datapaths present an obstacle to the efficient representation. Related work in [9] partly overcomes these difficulties by using univariate real-number polynomial representations that can deal with sequential components. However, as only real numbers can be used in [9], representations are only approximated.

In this paper we introduce new ways of generating the canonical descriptions of large datapath circuits. We propose techniques for compositional verification of digital signal processing (DSP) datapaths, consisting of adders, multipliers, memories, etc., as in Figure 1. The goal of such verification is to assure correctness of the *composition* of blocks. Blocks are either individually verified, or include Intellectual Property (IP) cores that are only given by their specifications. Our method is based on Arithmetic Transform (AT), which is an underlying representation used in word-level decision diagrams, enabling the compact canonical description of arithmetic circuits. The results are given in this paper in terms of AT, but also apply to graph representations.

In Section 3 we present the extensions to AT that provide the concise and easily obtainable description of datapath arithmetic circuits, including sequential ones. This allows us to develop the scheme for describing and verifying sequential datapaths that have been addressed mainly with complex theorem proving or model checking techniques [1], [7]. In Section 5 we demonstrate the efficiency of our method at work on the example of highly sequential Distributed Arithmetic (DA) circuits, that are well suited for field programmable gate array (FPGA) and deep sub-micron circuit implementations.

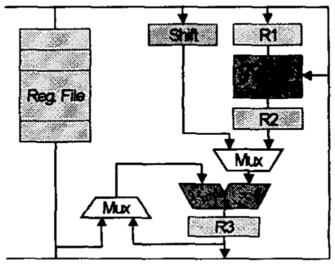

Figure 1: A Datapath Example

2. ARITHMETIC TRANSFORM

Arithmetic Transform is a canonical polynomial representation of multi-output Boolean functions $f : B^n \to B^m$. To describe multi-output functions with a single polynomial, function outputs are "grouped together" into word-level (*W*) quantities, e.g. integers, resulting in a pseudo-Boolean function $f : B^n \to W$.

Definition 1: *Arithmetic Transform (AT) of a pseudo-Boolean function* $f : B^n \to W$ *is a polynomial with arithmetic "+" operation, word-level coefficients* $c_{i_1 i_2 \cdots i_n}$, *binary inputs* $x_1,\ldots, x_n$ *and binary exponents* $i_1,\ldots,i_n$:

$$AT(f) = \sum_{i_1=0}^{1} \sum_{i_2=0}^{1} \cdots \sum_{i_n=0}^{1} c_{i_1 i_2 \cdots i_n} x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \qquad (1)$$

that uniquely and exactly interpolates f.

The transform coefficient vector $C = \{c_{i_1 i_2 \cdots i_n}\}$ is obtained by multiplying the vector *f* of word-level function outputs with the transform matrix T_n, $C = T_n * f$, where T_n is defined recursively by *arithmetic Davio expansion*:

$$T_n = \begin{bmatrix} T_{n-1} & 0 \\ -T_{n-1} & T_{n-1} \end{bmatrix}, \quad T_0 = 1. \qquad (2)$$

To show that the polynomial with coefficients generated in this way reconstructs exactly (interpolates) a function, note that according to Equation 1 AT is a linear polynomial in n variables. For $n = 1$, the coefficients c_0 and c_1 of a single-variable linear form $f = c_0 + c_1 x$ are obtained from cofactors f_0 and f_1. By multiplying the values with T_1, we get $c_0 = f_0$ and $c_1 = f_1 - f_0$, which is a more familiar form of Davio expansion. Matrix T_n extends this expansion recursively to n-variable functions, one variable at a time.

Once AT of a circuit implementation is generated, the comparison to AT of the specification is straightforward. Its canonicity and compact size for most arithmetic circuits, excluding the dividers, make AT very appropriate to use in equivalence checking of datapaths. Further, the error correcting properties of AT facilitate its application to verification by test vectors [8]. Hence, by using AT, a continuum of verification methods can be applied, including various combinations of formal and vector-based schemes. Additionally, as AT can represent IP core specifications, verification of systems including proprietary IP can be performed as well.

2.1 Word Encoding and Norm Function

AT accepts inputs as binary n-tuples and generates outputs in the word-level form. A word-level encoding is explicitly expressed by the *number norm* function $| \; |: B^m \to W$, which defines how a Boolean vector is interpreted in the word-level domain. Table 1 contains several common integer and fractional number norms.

Word	Number Norm $\lvert x\rvert$		
	Unsigned	Sign Extended	2's Complement
Int.	$\sum_{i=0}^{n-1} x_i 2^i$	$(1-2x_{n-1})\sum_{i=0}^{n-2} x_i 2^i$	$\sum_{i=0}^{n-2} x_i 2^i - x_{n-1}2^{n-1}$
Fract.	$\sum_{i=1}^{n-1} x_i 2^{-i}$	$(1-2x_0)\sum_{i=1}^{n-1} x_i 2^{-i}$	$-x_0 + \sum_{i=1}^{n-1} x_i 2^{-i}$

Table 1: Norm Functions for Common Word Encodings

Definition 2: *Binary encoding* $(x_1, x_2 \ldots, x_n)$ *of a word* w *is the inverse of the norm function,* $|w|^{-1} = (x_1, x_2 \ldots, x_n)$.

Note that there is no closed-form expression for binary encoding; instead, binary conversion algorithm must be applied to obtain $|w|^{-1}$. Hence, there is no simple AT($|w|^{-1}$).

Lemma 1: *Consider a pseudo-Boolean function* $f: B^n \to W$ *with a norm* $| \; |: B^m \to W$. *Then,*

$$AT(f) = |f|.$$

Proof: By applying the transform to quantities in W, an interpolation polynomial is obtained, such that for all inputs $AT(f(x_1, x_2, \ldots x_n)) = |f(x_1, x_2, \ldots x_n)|$. □

Example 1: **AT of Multiplier.** Let the input bits to the unsigned integer multiplier be x_k and y_k, $k = 0, \ldots, N\text{-}1$. Its AT is equal to the number norm of the product:

$$AT(x*y) = \sum_{k=0}^{N-1} x_k 2^k * \sum_{k=0}^{N-1} y_k 2^k = |x*y| = |x|*|y|. \quad \blacklozenge$$

Note that the norm of the sum (product) of two numbers equals the sum (product) of their norms. Further, the norm of an arbitrary algebraic expression is equal to the expression applied to the norms of its components.

3. ARITHMETIC TRANSFORM EXTENSIONS

We consider the construction of AT for the composition of several blocks, Figure 2.a. To compose ATs of two blocks where the outputs of the first block are fed to the inputs of the second one, we must convert the word-level output of the first block into a binary vector, Figure 2.b. The problem with this approach is that the binary encoding has to be explicitly constructed at each interface. No simple AT compositions are possible in this scenario.

Instead of using the binary conversion, we propose the alternative solution that allows the direct composition of transforms, as well as handling of sequential circuits. To achieve these goals, we need to introduce two extensions to basic AT.

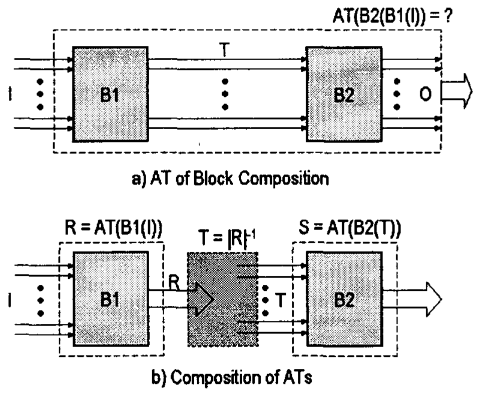

Figure 2: Naive Compositional AT Construction

3.1 First Extension: Mixed Transform

Our first extension facilitates the compositional approach to representing the complex datapaths. We allow the inputs to be a mix of binary, x_i, and word-level, w_j, quantities, i.e., $f(x_1 \ldots x_n, w_1 \ldots w_k)$.

Definition 3: *Mixed AT (MAT) of function f, $f: B^n \times W^k \rightarrow W$ is a polynomial with binary exponents $i_1, \ldots, i_n$ and $e_1, \ldots, e_k$:*

$$MAT(f) = \sum_{i_1=0}^{1} \cdots \sum_{i_n=0}^{1} \sum_{e_1=0}^{1} \cdots \sum_{e_k=0}^{1} c_{i_1 \cdots i_n e_1 \cdots e_k} x_1^{i_1} \cdots x_n^{i_n} w_1^{e_1} \cdots w_k^{e_k}$$

that interpolates f.

We will use the MAT transformation when some sets of inputs are known to be word-level quantities, such as outputs of previous blocks.

Lemma 2: *The coefficients of MAT can be calculated using the transformation given by Equation 2, expanded around binary input variables, with word-level input variables unassigned, i.e., treated as symbols*

$$C(w_1, w_2, \cdots w_k) = T_n * f \ .$$

Proof: Application of the transform to the quantities in B^n results in an interpolation polynomial that, according to Lemma 1, is $MAT(f(x_1, x_2, \ldots x_n, w_1, w_2, \ldots, w_k)) = | f(w_1, w_2, \ldots, w_k)|$. Assigning concrete word-level values to unassigned word-level variables still preserves the norm function, according to the norm properties. Hence, this transform is equal to $| f |$ which implies that the resulting MAT polynomial exactly interpolates the function. □

Example 2: **MAT of MUX.** A multiplexer (MUX) with inputs a, b and the control signal x can perform selection of either bit- or word-level quantities a and b. Arithmetic Davio expansion (Equation 2) around variable x, leads to $MAT(Mux) = a(1-x) + bx$. This example also illustrates the use of MAT for a non-arithmetic function. ♦

MAT facilitates the composition, by which the outputs of one AT can be directly used as inputs to MAT, Figure 3. For example, variables a and b in Example 2 can be given by ATs of previous blocks. To obtain the final AT of the composition, we need to apply the following conversion.

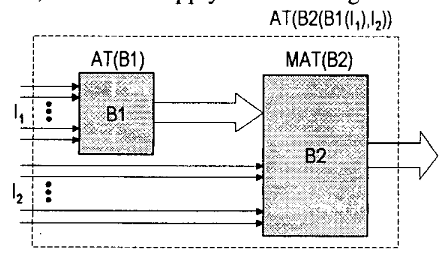

Figure 3: Use of MAT in Composition of ATs

Lemma 3: *AT(f) can be obtained from MAT of function $f: B^n \times W^k \rightarrow W$ through the syntactical replacements, where each word-level input $w_i \in W$ is substituted by its binary encoding $| \, |^{-1} : W \rightarrow B^m$, i.e.,*

$$MAT(f(x_1, \ldots, x_n, | w_1 |^{-1}, \ldots, | w_k |^{-1}) = AT(f) \ .$$

Proof: By using norm of the quantities in W, we get an interpolation polynomial in binary inputs, with $AT(f) = | f |$. This is a correct $AT(f)$ according to Lemma 1. □

Example 3: **MAT of Adder.** Consider an unsigned fractional adder, where input a is represented at word-level, i.e., unsigned fractional, while input b is treated as binary vector. Then, by substituting binary encoding of a in the defining MAT equation, we obtain its AT as:

$$MAT(a+b) = a + \sum_{i=1}^{N-1} b_i 2^{-i} = \sum_{i=1}^{N-1} (a_i + b_i) 2^{-i} = AT(a+b) \ .$$

Note that "+" has the same meaning in AT and MAT. ♦

To calculate AT of the whole combinational circuit, ATs should be generated for each block fed by primary inputs, while MATs should be applied to all other blocks. The overall AT is created by substituting AT transforms for intermediate word-level quantities.

3.2 Second Extension: Sequential Transform

To describe datapaths with sequential components, we introduce to MAT the notion of a *timed function*. A timed function $f[n]$ represents the value of f at the n^{th} clock period.

Definition 4: *MAT Sequential (MATS) is a MAT transform $MAT(f)[n]$, of timed function f at time instance n.*

Many datapath blocks are *time invariant*. Hence, the timed transform of arithmetic blocks will be exactly the same as the original MAT transform. The simplest role that the timed transform plays is to represent the state information kept in memory elements. A defining equation for a memory is $m_{out}[n] = m_{in}[n-1]$.

Example 4: **MATS of Add-Accumulate Loop.** Consider the addition in n^{th} step, where one of the summands is taken from the primary inputs, while the other is supplied from the register storing the accumulated values of the previous n-1 additions, Figure 4.a. Assume that the register has been initially reset.

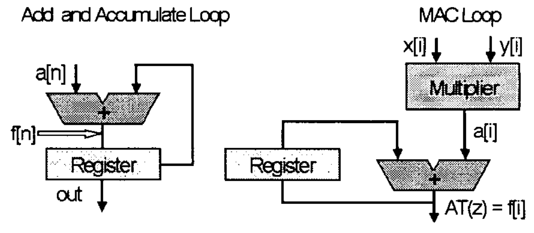

Figure 4: Add- and Multiply-Accumulate Loops

To obtain MATS of the add-accumulate loop, consider the intermediate word-level function $f[n]$, representing the input to the register after n additions. Assuming that the free input $a[n]$ is also a word-level quantity, the value of $f[n]$ is given by the recurrence equation:

$$f[n]=a[n]+f[n-1], \qquad f[0]=0.$$

The corresponding MATS is then:

$MATS(f)[n]=a[n]+MATS(f)[n-1], \quad MATS(f)[0]=0$.

After the n^{th} addition, MATS of the add-and-accumulate loop is obtained as the following solution to the above recurrence equation:

$$MATS\,(f)[n]=\sum_{i=1}^{n} a[i]. \qquad ♦$$

Lemma 4: *MATS of a composition of a combinational element described by function f with sequential blocks can be obtained from MAT(f) by replacing each input that is generated by a sequential block with its defining MATS.*

Proof: Since MAT describes the combinational function of a block, with memory elements disconnected, then, by Definition 4, such a syntactical replacement presents the correct functionality of the sequential function. □

Corollary 1: *MATS of sequential function f can be obtained from MAT of the combinational part of f by replacing each MAT input that is generated by a memory element with its defining MATS.*

Corollary 2: *If at least one input variable of combinational function f is generated by a sequential block, then the transform of the composition, instead of MAT, needs to be presented as MATS:*

$$MATS((f(x_1,\dots w_i\dots,w_k))=MAT(f(x_1,\dots MATS(w_i)\dots,w_k))$$

Lemma 5: *MAT of a circuit at time instance n can be obtained by solving MATS as a recurrence equation. Sequential elements need to be initialized.*

Proof: MATS represents the function of a block at each time instance. By solving the recurrence equation, the circuit behavior at a given time instance is obtained. □

Recurrence equations describing sequential elements in datapaths posses forms that are easily solvable analytically and symbolically by tools such as Maple or Mathematica.

Example 5: **MATS of Multiply-and-Accumulate (MAC) loop** – a common element of DSP datapaths. Consider the transformation of the composition performed over the MAC, Figure 4.b, using previously derived MAT transforms of its individual blocks. Inputs to the MAC at the time instance i are N-bit binary vectors x and y, the output z is a binary vector of size $2N$. MATS of the multiplier block is defined for inputs occurring at time instance i as:

$$MATS(x*y)[i]=\sum_{k=0}^{N-1}x_k[i]2^k*\sum_{k=0}^{N-1}y_k[i]2^k=a[i].$$

The transform of the accumulator-register loop, obtained by solving the same recurrence type as in Example 4 is:

$$MATS(f)[n]=\sum_{i=1}^{n}a[i]=\sum_{i=1}^{n}\left(\sum_{k=0}^{N-1}x_k[i]2^k*\sum_{k=0}^{N-1}y_k[i]2^k\right).$$

Please note that the above representation corresponds to the MAC loop specification. ♦

4. VERIFICATION OF DATAPATHS

A datapath of a typical DSP architecture, Figure 1, performs arithmetic and logic operations. Its main building blocks are: register files, ROMs, shift registers, multiplexers, multipliers and adders connected into a MAC structure, as shown in Figure 1.

To verify the complete datapaths, we will rely on the verified implementations of its basic blocks, done either through equivalence checking [5], or vector-based methods [8]. For overall datapath verification, we can use the library of datapath blocks, together with their transforms. The transform of each element only depends on its functionality, and not on the actual implementation. For example, AT of an adder is the same, for a ripple, look-ahead or skip adder. The transforms (AT, MAT or MATS) of the most common arithmetic circuits were presented in Examples 1-5. Next, we introduce the algorithm to verify the datapath composition from blocks.

4.1 Transform of Complete Datapaths

Having defined transforms of individual blocks in a datapath, the automated construction of the overall AT can proceed by forward traversal from primary inputs, as shown in Algorithm 1. For each block encountered, the transform is constructed from its immediate inputs. Each combinational block depending entirely on primary inputs requires only AT description (line 4, Algorithm 1). Blocks with inputs generated by previous blocks will be represented in MAT form (line 6, Algorithm 1) if none of its inputs perform a sequential function, Corollary 2.

```
Generate MATS of the network of blocks
1.    for each block B_i in topological order
2.    {
3.     Assign: inputs (B_i) to output(predecessor(B_i))
4.      if (combinational(B_i) && all_inputs_primary)
5.          f_i = AT(B_i);
6.      if (combinational(B_i) && no_seq_input)
7.          f_i = MAT(B_i, assigned_input_list);
8.      else    /*sequential(B_i)*/
9.          f_i = MATS(B_i, assigned_input_list);
10.         f_i = reccurence_solve(f_i );
11.    }
12.    Overall_AT = f_i
```

Algorithm 1: Composition of Block Transforms

Every time a sequential block is encountered, its recurrence equation is solved symbolically. The outputs are then expressed in terms of the block inputs, *MATS(w)[n]*, at the time n of their occurrence at the block inputs w. The inputs are then substituted by transforms of the previous blocks in the overall expression. The process is repeated until all the blocks are traversed, and all the outputs are expressed in terms of primary outputs of the circuit.

5. VERIFICATION OF DISTRIBUTED ARITHMETIC CIRCUITS

Distributed Arithmetic (DA) refers to datapath architectures where the inner product of a vector by a constant is performed as a bit-serial operation [10]. DA leads to efficient FPGA implementations of specialized DSP circuits like filters, but also of basic arithmetic functions, such as multipliers [3]. In filter applications, the word-level quantities are usually fractions. Hence, the signed numbers are represented as: $x = -x_0 + \sum_{i=1}^{M-1} x_i 2^{-i}$, and for unsigned numbers, the sign digit x_0 is not used.

The calculation of inner products of a vector of constants A_k with input vectors x_k, $k = 1, 2,\dots,N$:

$$y = \sum_{k=1}^{N} A_k x_k, \quad where \quad x_j = -x_{j0} + \sum_{i=1}^{M-1} x_{ji} 2^{-i}$$

is performed bit-wise. Instead of inputting the whole M-bit x_k vector in parallel, and then executing the multiplication by a M-bit constant A_k, all vectors are serially shifted in. In each cycle every shifted bit is "multiplied" in parallel by its constant A_k. Partial sums maintained in this way do not resemble the standard partial results of inner product calculations. The product:

$$y = \sum_{j=1}^{N} A_j x_j = \sum_{j=1}^{N} A_j \left(-x_{j0} + \sum_{i=1}^{M-1} x_{ji} 2^{-i} \right)$$

is transformed by changing the order of summation into

$$y = -\sum_{j=1}^{N} A_j x_{j0} + \sum_{i=1}^{M-1} \left(\sum_{j=1}^{N} A_j x_{ji} \right) * 2^{-i}. \qquad (3)$$

Equation 3 represents the form in which the inner product is calculated in the DA arithmetic.

The multiplications in brackets, Equation 3, do not have to be calculated with multipliers. Instead, bit products can be pre-computed for all the possible combinations of bits x_{ji} of vectors x_j, and stored in ROM. Then, based on the actual combination of x_{ji} among all x_j's, the corresponding ROM location is addressed [10]. One possible hardware implementations of the DA inner product of four 4-bit vectors is presented in Figure 5.

5.1 Transforms of Building Blocks of DA

A typical DA circuit consists of the following blocks. *Shift registers* are used to serially input the vector bits. *ROMs* store the pre-computed partial sums, and *add-accumulate loops* maintain the partial results of the multiplication, Figure 5.

Transforms of a few major datapath blocks were derived in Sections 3.1 and 3.2. Now, we present the transforms of the remaining DA blocks. For presentation purposes, we initially assume that the sign bits are zero, or, equivalently, that the unsigned encoding is employed.

AT of ROM. The content of ROM represents the bit multiplication of a constant A_k by bit vector x_k, $k = 1,\dots,N$, Figure 5. It is easily verifiable from Figure 5 that, for the case $N = 4$, AT of ROM is given by

$$AT(ROM) = A_1 x_1 + A_2 x_2 + \dots + A_N x_N = \sum_{j=1}^{N} A_j x_j \ ,$$

where each variable x_j is one-bit wide and the coefficients A_j, $j = 1, \dots, N$ are fractional numbers.

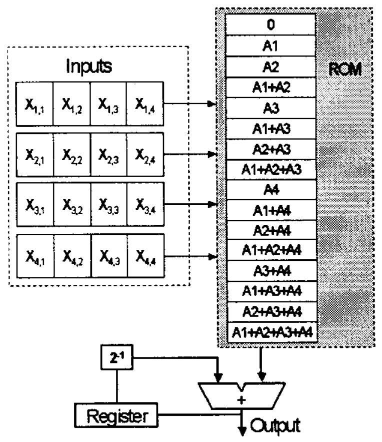

Figure 5: Distributed Arithmetic – ROM/Accumulator Structure

MATS of Shift Registers. A shift register is a purely sequential circuit, consisting of a chain of storage cells. For each register, its discrete time defining equation is given as $f[n] = a[n-1]$, where the inputs $a[n]$ are treated as bits. Building the shift register out of individual storage cells amounts to composing the word-valued transfer functions. For j^{th} shift register with M-1 stages, MATS at time k is $MATS(SR(x_j))[k] = x_{j(M-1-k)}$.

MATS of Add-Accumulate Sum with Delay Element. MATS of this block is calculated by constructing and solving the recurrence: $f[n] = a[n] + \frac{1}{2} f[n-1], f[0] = 0$, where variable $a[n]$ is a word-level ROM output quantity and the initial condition in the register is 0. By solving the recurrence equation at time n, we get:

$$f[n] = \sum_{i=0}^{n} a[i] * 2^{i-n} = \sum_{i=0}^{n} a[i] * 2^{-(n-i)} \ .$$

5.2 AT Description of Complete Unsigned DA

The transform of the complete circuit in Figure 5 is obtained by forward traversal, following Algorithm 1. The intermediate variables will be substituted in all the expressions during the traversal. By substituting the outputs of the shift registers to the ROM inputs, we obtain the following timed expression:

$$MATS(ROM)[k] = \sum_{j=1}^{N} A_j x_{j(M-1-k)} \,.$$

Next, substituting these variables, as the inputs to the add-accumulate sum results in:

$$f[n] = \sum_{k=0}^{n} \sum_{j=1}^{N} A_j x_{j(M-1-k)} 2^{-(M-1-k)} \,. \qquad (4)$$

The overall function $f[M\text{-}2]$ is completed at time instance M-2 as Equation 4 at time M-2:

$$AT(f) = MATS(f)[M-2] = \sum_{k=0}^{M-2} \sum_{j=1}^{N} A_j x_{j(M-1-k)} 2^{-(M-1-k)}.$$

The result is the same as in Equation 3 (circuit specification) with the assumption that the sign bits are zero. To verify that, we notice that by applying the change of indices of the first summation, the final AT is given by the expression that is equivalent to Equation 3:

$$\sum_{k=0}^{M-2} \sum_{j=1}^{N} A_j x_{j(M-1-k)} 2^{-(M-1-k)} \Big|_{i=M-1-k} = \sum_{i=1}^{M-1} \sum_{j=1}^{N} A_j x_{ji} 2^{-i}.$$

5.3 AT Description of Signed and Encoded DA

The signed DA algorithm differs from the previous case in $(M\text{-}1)^{th}$ step added at the end of the execution, when the sign bits are incorporated. Then, $f = f[M\text{-}1]$ is obtained as

$$AT(f) = MATS(f)[M-1] = -\sum_{j=1}^{N} A_j x_{j0} + MATS(f)[M-2].$$

This expression is equal to Equation 3; hence, the equivalence is proven.

The same approach was used to prove the equivalence of additional DA forms, including encoded DA circuits from [10] that achieve the reduction in ROM sizes by alternative internal number encoding. Each new circuit can have a different, non-trivial DA implementation. The presented method can check the correctness of DA realization already on the algorithmic level.

We implemented Algorithm 1 in Mathematica, for its ease of integration with the recurrence solver `rsolve` (by M. Petkovsek), already incorporated in Mathematica. The obtained representation is canonical and compact. By comparing the sizes (measured in words) of the resulting AT with the sizes of ROMs employed in DA circuits, as in Table 2, we observe that the overall AT representation is logarithmic in the size of a ROM. The last column presents worst case times spent on a 440MHz Ultra 10 workstation with 128MB of main memory in generating overall AT from ATs of DA components. As Mathematica is an interpreter with lots of overhead, these times could be greatly reduced by compiled code.

#	Unsigned		Signed		Encoded		Time
Terms	ROM	AT	ROM	AT	ROM	AT	[s]
4	16	60	32	64	8	64	0.5
8	256	120	512	128	128	128	2.0
16	16k	240	32k	256	8k	256	9.2
32	4G	480	8G	512	2G	512	32.0

Table 2: AT vs. ROM Sizes in DA Implementations

6. Conclusions

We presented two extensions to Arithmetic Transform that facilitate the compositional verification of sequential datapaths. The first extension makes the easy composition of transforms of individual blocks possible, while the second one allows sequential circuit representations. The compact canonical descriptions of large circuits can be quickly obtained by symbolic composition of transforms of individual blocks. Verification of highly sequential Distributed Arithmetic architectures was presented using these transforms. We intend to apply the method to the larger classes of sequential datapaths.

7. References

[1] M.D. Aagaard and C-J. H. Seger, "The Formal Verification of Pipelined Double-precision IEEE Floating-point Multiplier", In *Proc. IEEE/ACM Int'l Conf. CAD*, ICCAD, pp. 7-10, 1995.

[2] R. Amirtharajah, T. Xanthopoulos and A. Chandrakasan, "Power Scalable Processing Using Distributed Arithmetic", In *Proc. ISLPED*, pp. 170-175, 1999.

[3] A. Berkeman, V. Owall, and M. Torkelson, "A Complex Multiplier with Low Logic Depth", *IEEE Int'l. Conf. On Electronics, Circuits and Systems*, pp. 47-50, 1998.

[4] R.E. Bryant and Y-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams", In *Proc. of 32nd Design Automation Conference*, pp. 535-541, 1995.

[5] R. Drechsler, B. Becker and S. Ruppertz, "The K*BMD: A Verification Data Structure", *IEEE Design and Test of Computers*, Vol. 14, No. 2, pp. 51-59, April-June 1997.

[6] K. Hamaguchi, A. Morita and S. Yajima, "Efficient Construction of Binary Moment Diagrams for Verification of Arithmetic Circuits", In *Proc. ICCAD*, pp.78-82, 1995.

[7] J. D. Kim and S-K. Chin, "Assured VLSI design with formal verification", In *Proc. 12th Annual Conf. Computer Assurance*, COMPASS, pp. 13 -22, 1997.

[8] K. Radecka and Z. Zilic, "Using Arithmetic Transform for Verification of Datapaths via Error Modeling", *Proc. of VLSI Test Symposium*, pp. 271-277, 2000.

[9] J. Smith and G. De Micheli, "Polynomial methods for allocating complex components", In *Proc. Design, Automation and Test in Europe*, pp. 217 -222, 1999.

[10] S.A. White, "Applications of Distributed Arithmetic to Digital Signal Processing: A Tutorial Review", *IEEE ASSP Magazine*, pp. 4-19, July 1989.