Specifying and Verifying Imprecise Sequential Datapaths by Arithmetic Transforms

Katarzyna Radecka **Zeljko Zilic**
Concordia University **McGill University**
Montreal, Quebec, Canada

Abstract

We address verification of imprecise datapath circuits with sequential elements. Using Arithmetic Transform (AT) and its extensions, we verify the sequential datapath circuits with finite precision. An efficient formulation of the precision verification is presented as a polynomial maximization search over Boolean inputs. Using a branch-and-bound search for the precision error and the block-level composition of ATs, we verify the approximated, rounded and truncated pipelined datapaths.

1. Introduction

The push for intellectual property (IP) core reuse has created a host of design, verification and test problems. New methods are required for efficient specification, verification, and IP component matching. This is especially true for arithmetic and datapath blocks, which are among the most used cores. Early equivalence checking methods that relied on BDD representations were unable to deal with arithmetic circuits such as multipliers. Word-level graphs like *BMDs [1] remove this obstacle by employing Arithmetic Transform (AT). Extensions to AT in [6] facilitate verification of sequential datapaths and their compositions. Such blocks are either verified individually, or include IP cores given only by their specifications.

Virtually all datapath verification methods assume that the circuit is implemented exactly, implying either infinite precision, or the specification of only one of many implementations within allowed imprecision. In this paper, we address the more complete case of verification within some error tolerance. We use the fact that AT deals efficiently with word-level quantities and allows efficient symbolic manipulation [4] and interpolation [9].

Previous work involving verification of imprecise datapaths considered a bit-level case [2] that does not capture precision properly. In related matching of a (single) IP core, rational-valued polynomials [7] accommodate some imprecision.

1.1 Arithmetic Transforms

Arithmetic Transform is a canonical representation of multi-output Boolean functions $f: B^n \to B^m$ that is often compact for datapaths. AT is a polynomial, obtained by considering outputs at word-level (W), resulting in pseudo-Boolean $f: B^n \to W$.

Definition 1: *Arithmetic Transform is a polynomial with arithmetic "+" operation, binary inputs $x_1,\ldots, x_n$, word-level coefficients $c_{i_1 i_2 \cdots i_n}$ 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 exactly and uniquely interpolates pseudo-Boolean function f.

AT is a multi-output function representation that uses familiar (integer) arithmetic in adding the polynomial terms, each of which has a subset of variables present. For example, AT polynomial $2+x_1+4x_1x_2x_3$ describes a unique 3-bit input, 4-bit output function whose truth table is obtained by polynomial valuation for all inputs, followed by decoding integers as multi-bit vectors. AT achieves compact description because many outputs are grouped in a single word level quantity, often given by a simple arithmetic expression, as in the following example.

Example 1: **AT of Multipliers** Consider an unsigned integer multiplier with inputs x_k and y_k, k= 0 ,..., N-1. Unlike with BDDs, AT is polynomial in size: $AT(x * y) = \sum_{k=0}^{N-1} x_k 2^k * \sum_{k=0}^{N-1} y_k 2^k$.♦

Once AT of a circuit implementation is generated, the comparison to the specification is straightforward. This makes AT particularly suitable for equivalence checking of datapaths [6], verification by test vectors [5] and in their combinations.

1.1.1 Arithmetic Transform Extensions – MAT and MATS

To accommodate increasing use of IP and block-level components, extensions to the basic AT form were presented in [6]. In order to use AT of a block as an input to the second one, we must convert the word-level output of the first block into a binary vector. No simple AT compositions are possible in this way. Instead, we extend AT. The first extension facilitates the compositional approach to representing the complex datapaths. Mixed AT (MAT) treats the inputs as a mix of binary, x_i, and word-level, w_j, quantities, i.e., $f(x_1 \ldots x_n, w_1 \ldots w_k)$, Figure 1.

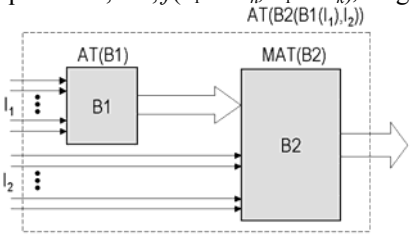

Figure 1: MAT in Composition of ATs

Definition 2: *Mixed AT (MAT) of $f: B^n \times W^k \to W$ is a polynomial with binary exponents $i_1, \ldots, i_n$ and $e_1, \ldots, e_k$:*

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

Example 2: **MAT of Adder and Multiplier.** Consider an unsigned adder and multiplier with word-level a and binary b:

$$MAT(a+b) = a + \sum_{i=1}^{N-1} b_i 2^{-i}, MAT(a*b) = a * \sum_{i=1}^{N-1} b_i 2^{-i}$$

Note that "+" and "*" are the same in AT and MAT. ♦

0-7803-7607-2/02/$17.00 ©2002 IEEE

To describe sequential datapaths we introduce MAT Sequential (MATS), representing the value of f at the n^{th} clock period.

Definition 3: *MAT Sequential* (*MATS*) *is a MAT transform MAT(f)[n], of function f at time instance n*.

To obtain AT from MATS, MAT of a sequential circuit must first be generated from its MATS description by symbolically solving MATS as a recurrence equation [6]. The overall AT is created by substituting ATs for intermediate word-level quantities in all MATs throughout the circuit.

Word	Number Valuation V(x)		
	Unsigned	Sign Extended	2's Complement
Int.	$\sum_{i=0}^{n-1} x_i 2^i$	$(1-2x_{n-1})\sum_{i=0}^{n-2} x_i 2^i$	$\sum_{i=0}^{n-2} x_i 2^i - x_{n-1}2^{n-1}$
Fractional	$\sum_{i=1}^{n-1} x_i 2^{-i}$	$(1-2x_0)\sum_{i=1}^{n-1} x_i 2^{-i}$	$-x_0 + \sum_{i=1}^{n-1} x_i 2^{-i}$
Fixed Point	$\sum_{i=0}^{n-1} x_i 2^{i-m}$	$(1-2x_0)\sum_{i=1}^{n-1} x_i 2^{i-m}$	$\sum_{i=1}^{n-1} x_i 2^{i-m} - x_0 2^{m-n}$

Table 1: Valuations for Common Word Encodings

2. Verifying Imprecise Arithmetic Circuits

Specification and verification of arithmetic circuits consider mainly the cases when results are exact. However, for many practical datapath circuits, results are rounded and otherwise imprecise, due to the finite wordlength of a datapath. The challenge of the verification is to distinguish between cases of an erroneous circuit behavior and outputs that are correct within some error bound due to imprecise arithmetic. Equivalence checking deals only with exact computations and such imprecise cases are declared incorrect. However, the use of the word-level representations, as in the considered Arithmetic Transforms, will be critical for verifying imprecise datapaths efficiently.

AT and related forms deal explicitly with word-level quantities. A word-level encoding is explicitly expressed by the *number valuation* function $V: B^m \rightarrow W$, which defines how a Boolean vector is interpreted in the word-level domain. Table 1 contains several common integer and fractional number valuations. We concentrate on fixed-point representations that are preferred in circuit implementation for their simplicity and rely on tools that compile floating-point code to fixed-point implementations [8].

2.1 Precision Case: Fractional Multiplier

We illustrate the steps undertaken in the precision verification on an example of a multiplier operating over two fractional numbers. In this case, two fractional inputs, a and b, are represented by n bits. While the output should be kept $2n$-bit wide, the implementation assuming the n-bit wide datapath is forcing the result of the multiplication to be represented by n bits. For simplicity, we discuss only the unsigned operations.

Case 1: Rounding and Truncation. The $2n$-bit result is first calculated exactly, and then rounded to first n bits. In the case of rounding, the error is half of the LSB, i.e., $2^{-(n+1)}$. In the case of truncation, the maximum error is bounded by the LSB.

Case 2: Approximation by *n* most significant bits. In addition to rounding and truncation, there exist approximation modes where significant hardware resources can be spared when precision is not critical. For multipliers over fractional numbers, the n least significant terms are simply not calculated. Savings of half the gates are obtained at the expense of the mismatch:

$$e = \left| AT(a*b) - AT(a*b)_{trunc} \right| = \sum_{i=n+1}^{2n} \left(\sum_{j=1}^{i} a_{i-j} b_j \right) 2^{-i} . \quad (2)$$

As the function is monotonous, the maximum is attained for all inputs equal to one, and the worst case error is $O(n2^{-n})$.

3. Verification Formulation

With AT, the formulation of verification under precision constraints is simple. The precision objective is directly expressed for the output quantities at a word level, unlike the bit-level approaches [2]. Also, inputs are binary values, suitable for binary search. Precision verification compares specification AT (*SpecAT*) to an implementation AT (*IAT*). For a given precision ε, the *maximum absolute value* of the difference between the transforms needs to be smaller than ε for all inputs X:

$$\max | SpecAT(X) - IAT(X) | \leq \varepsilon . \quad (3)$$

When *SpecAT* is itself imprecise, and represents f up to absolute precision error of δ, the same procedure can produce the maximum mismatch between the function value and the circuit implementation. By triangle inequality, the imprecision is:

$$\max | IAT(X) - f(X) | \leq$$

$$\leq \max \left| IAT(X) - SpecAT(X) \right| + \max \left| SpecAT(X) - f(X) \right| \leq \varepsilon + \delta$$

Hence, it suffices to verify the imprecision relative to its AT specification, Eq. 3. In practice, it is often $\delta << \varepsilon$.

Verifying a circuit under the precision constraints amounts to finding a maximum absolute value that the difference between a specification and an implementation takes. Since AT is linear, and the difference of two ATs is still an AT polynomial, we seek either a maximal positive or a minimal negative value of the mismatch AT. Hence, the verification under constraints of arithmetic precision is expressed as a search for a maximum of an *mismatch AT*, i.e., finding binary inputs $x_1, x_2, \ldots x_n$ for:

$$\max \left| \sum c_{i_1 i_2 \cdots i_n} x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n} \right| . \quad (4)$$

3.1.1 Constrained and Unconstrained Cases

If all possible input combinations are allowed, we say that the search is *unconstrained*. In practice, many input combinations are *don't cares*, and the search is *constrained*.

An unconstrained case is easier to handle, as there is no limitation on searching for the solution. Further, for functions and the mismatches that are *unate*, the maximum and minimum values are obtained when all inputs are 1s or 0s. Adders and multipliers are unate for unsigned encoding, and all polynomial coefficients are positive, as seen from Example 1. Also, the mismatch function in Section 2.1 is unate. In that case, the maximum that these functions take is obtained at the input point 11…1. The unconstrained case is useful when considering the easily obtainable bounds on the maximum and minimum value that AT polynomial takes. One such upper bound is obtained as the sum of all polynomial coefficients that are positive and the coefficient $c_{00\ldots0}$, that contributes a constant offset for all input assignments. We call such a bound ub_{coef}, and denote its calculation as:

$$ub_{coef} = c_{00\cdots0} + \sum_{c>0} c_{i_1 i_2 \cdots i_n}$$

AT will clearly attain this upper bound if there is a combination of inputs for which all positive coefficients are multiplied by 1 and all negative coefficients are multiplied by 0. The constant coefficient $c_{00\ldots0}$ is not affected by any input combination. In the same way, we define lb_{coef} as the sum of all negative coefficients and the constant coefficient $c_{00\ldots0}$. Using these two bounds, the following lemma will be needed in the algorithms to follow.

Lemma 1: *The sum of the bounds lb_{coef} and ub_{coef} is equal to the sum of the function values $f_{00\ldots0}$ and $f_{11\ldots1}$:*

$$ub_{coef} + lb_{coef} = f_{00\cdots0} + f_{11\cdots1}$$

Proof: From definitions of lb_{coef} and ub_{coef}, their sum is:

$$ub_{coef} + lb_{coef} = c_{00\cdots0} + \sum_{c>0} c_{i_1 i_2 \cdots i_n} + c_{00\cdots0} + \sum_{c<0} c_{i_1 i_2 \cdots i_n}$$

which is equal to $c_{00\ldots0}$ and the sum of all coefficients. The former is equal to the function value at the point (minterm) 00…0, and the latter is equal to the value at 11…1. □

In the unconstrained case, this lemma is used to obtain one bound (say upper) from another (lower) and the function values at points 00…0 and 11…1. The bounds can be used in the unconstrained case as well, but will be less tight. The tighter bounds obtained for function restrictions, i.e., when a subset of variables is assigned a value, will be used as well.

Example 3: Let $AT(f) = 2 - x_1 + x_3 - 3x_1x_2 + x_1x_2x_3$. The maximum and the minimum values are $f_{max} = 3$ and $f_{min} = -2$. The bounds are quickly obtained from $AT(f)$ as $ub_{coef} = c_{00\cdots0} + \sum_{c>0} c_{i_1 i_2 \cdots i_n} = 2+1+1 = 4$ and $lb_{coef} = -2$. The lower bound is equal to the minimum value. By setting inputs to all 0s and 1s, we obtain $f_{000}=2$ and $f_{111}=0$. Hence, we verify that:

$$ub_{coef} + lb_{coef} = f_{00\cdots0} + f_{11\cdots1}.$$ ♦

3.1.2 Branch-and-Bound Search for Imprecision Error

We developed an algorithm for finding the maximum of AT (Eq. 4), with binary inputs $X = \{x_i, i=1, ..., n\}$ and polynomial terms multiplied by word-level coefficients. We seek the mismatch maximum by a *branch-and-bound* search, where bounds on polynomial and Lemma 1 guide early terminations.

We deal separately with cases when the assignment of variables is known without a search at each call of the main search routine. For simplicity, we refer to it as *preprocessing*.

Preprocessing: The search is preceded by an iterative application of two rules for assigning a value to binary input x_i, and, hence, reducing the search space:

1) If all coefficients in x_i are positive, assign x_i=1.
2) If all coefficients in x_i are negative, assign x_i=0.

Clearly, these assignments lead to the maximum overall value of the polynomial. In the example of the truncated multiplier in Section 2.1, we notice that rule 1 is sufficient to find the maximum error in Eq. 2, as all the coefficients (2^{-i}) are positive.

Main Search Loop: The search investigates first, in a heuristic manner, the variables that likely lead to a function maximum. For each variable, we calculate the sum of all coefficients multiplying terms in which the variable is present. The *most positive variable* is the one for which this sum is the largest. In Example 3, x_3 is the most positive variable that contributes most to the maximum value. In the case of a draw, the easy lower bounds (by Lemma 1), are compared as well. Only algorithm speed, rather than its correctness depends on that heuristic choice.

The search is terminated when a current upper bound is smaller than the currently largest value. The upper bound of the current function restriction $ub_{coef}(AT_{x=v})$ is the sum of all positive AT coefficients when a variable x is set to v. Algorithm 1 summarizes the search for the absolute value of the mismatch AT polynomial.

```
for each variable xi                    /*preprocessing */
   if (all c*xi* >0)  xi=1 elsif (all c*xi* <0) xi=0;
ub_coef = c_00...0 + Σ_{c>0} c_{i1 i2 ... in} ; lb_coef = f_00...0 + f_11...1 − ub_coef
CurrentBest=lbcoef; Current=ubcoef;
...
Max_Abs (AT, Current)
 if NonasignedVars  {
   x=MostPositiveVariable;
   CAT=AT(f)x=1     /*try x=1*/
   if (ubcoef(CAT)<CurrentBest) backtrack;   /*cut search branch */
     else   Current=Max_Abs(CAT,Current);  /*recur further*/
   CAT=AT(f)x=0      /*try x=0*/
    if (ubcoef(CAT)<CurrentBest) backtrack;
      else    Current=Max_Abs(CAT,Current);}
else{
  Current = AT;  /* all variables assigned, leaf case */
   if (Current>CurrentBest) {CurrentBest=Current; return(Current)}}
```

Algorithm 1: Finding Maximum Mismatch

Example 4: Consider searching for maximum of the following AT specification/implementation mismatch:

$$2x_1 - 3x_2 + 2x_3 + 3x_4 - 3x_1x_3 + 3x_2x_3 - 2x_1x_4 + 3x_1x_2x_4$$

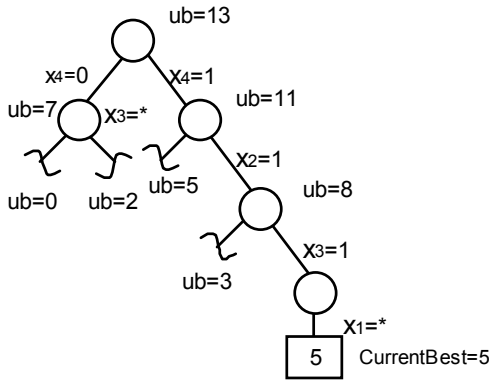

Figure 2: Execution of Max_Abs for Example 4

The upper bound is equal to the sum of all positive coefficients, i.e., 13; the lower bound is -8. The preprocessing step cannot assign any variable, and the main search loop is invoked. By adding coefficients in each variable, we obtain that the coefficient sums are 0, 3, 2 and 4 in variables x_1, x_2, x_3 and x_4, respectively. The most positive variable x_4 is first set to 1, followed by variables x_2 and x_3. The final assignment to x_1 leads to the terminal node in the search tree, when the current best value is 5. The algorithm then backtracks to traverse the search tree, as shown in Figure 2. Current upper bounds (ub) are inscribed next to each node. First, the variable x_3 is re-assigned to 0. The upper bound (3) at this stage is found to be lower than the current best value, and the search is terminated. Similarly, the other backtracks are terminated when the upper bound is smaller or equal to the currently obtained value. Upon reaching the root, the

order of variables changes, as variable x_3 is examined first. After exhausting input space, the maximum value is 5. ♦

While Algorithm 1 seeks the solution for the unconstrained case, the constrained case is dealt by adding one more condition for terminating the search. The search will also stop when a subtree of the search space to be visited is included in *don't care* set. To represent *don't care* sets, one can, in general, use BDDs. The traversal of the BDD mimics nicely the traversal of the input search space tree, as in Figure 2. In our case, restrictions on the input space described by AT and an inequality, e.g., $AT(X)>CaresetBound$ are often most compact. For that reason, we use an inequality as a default representation of the input space constraints. Constraints given by a BDD can be used as an option.

3.1.3 Representing Pipelined Datapaths

Pipelined circuits contain registers separating combinational elements into pipeline stages. Pipelined datapaths have been hard to verify. While formal methods suffer from the state space explosion, the vector-based verification cannot deal with the inherent deep sequentiality of pipelined circuits. In contrast, to verify *k*-stage pipelined circuits by MATS, it suffices to specify that the output produces the result delayed by *k* clock cycles.

Example 5: **MATS of Add-Multiply Pipeline Stage.** A commonly used pipeline stage consists of a multiplier-adder pair, fed by two sets of primary inputs *x* and *z*, and the outputs of a circuit *y*. The output *a* of the adder is registered, and the overall MATS is: $MATS(f)[n] = z[n-1] + x[n-1] * MATS(y)[n-1]$. ♦

3.2 Verifying Complete Datapaths

The imprecise datapath composition verification is outlined in Algorithm 2. For each block encountered in a forward traversal, the transform is constructed from its immediate inputs. Each combinational block depending entirely on primary inputs requires only AT (line 3). Blocks with inputs from previous blocks require MAT form (line 5) if none of its inputs perform a sequential function. Every time a sequential block is encountered, its recurrence equation is solved. The process is repeated until all the blocks are traversed, and the outputs are expressed in terms of primary inputs. Finally, we invoke Algorithm 1 to verify required precision (line 10).

3.2.1 Experiments with Pipelined Cosine Circuits

We implemented Algorithm 2 in Mathematica, for its ease of integration with its recurrence solver `rsolve`. The algorithm has been tested on pipelined cosine circuits. The function $cos(x)$ is approximated in interval $[0, \pi/4)$ using the Taylor series. As the required series are infinite, the actual calculation is bound to introduce error. To approach the verification, specification is given by AT of sufficient precision, *SpecAT(cos)*, augmented with the bounds on the error *e*. The verification goal is to assert that the implementation $AT(cos)$, is within error bounds.

We compared the pipelined cosine circuits with respect to their precision, AT size and the time required to verify the circuit. The verified implementations are: 64-bit, 8-bit rounded and truncated (Section 2.1, case 1), and approximated by 8 bits (case 2), respectively, as shown in Table 2. Obtained AT is compact, as shown by the 3rd column with sizes of all intermediate (and final) AT, MAT and MATS polynomials. The last column presents worst case times spent on a 440 MHz Ultra 10 workstation with 128MB of main memory. Note that the results could be improved by compiled code instead of interpreted Mathematica code.

Circuit	Terms	AT Size	Error	Time [s]
cos – 64bit	4	6111	9.9E-16	121
cos – 8bit round	4	447	3.7E-6	73
cos – 8bit trunc	4	367	7.1E-6	51
cos – 8bit only	4	54	5.1E-3	39

Table 2: Results for Verifying Pipelined Cosine Circuits

```
Verify_Imprecise (network, SpecAT, ε)
1.   for each block Bi in topological order{
2.    Assign: inputs (Bi) to output(predecessor(Bi))
3.      if (combinational(Bi) && all_inputs_primary)
4.            fi = AT(Bi);
5.      if (combinational(Bi) && no_seq_input)
6.            fi = MAT(Bi, assigned_input_list);
7.      else     /*sequential(Bi)*/
8.            fi = MATS(Bi, assigned_input_list);
9.            fi = reccurence_solve(fi );   }
10.  return(Max_Abs(fi – SpecAT ) < ε); //max | fi – SpecAT|)
```

Algorithm 2: Verification of Imprecise Datapaths

4. Conclusions and Future Work

We presented an algorithm for verifying the correctness of imprecise arithmetic operations. Compact descriptions of large circuits can be quickly generated by symbolic composition of transforms of individual blocks. The advantage of AT is that it directly expresses the sought worst case error in its numerical value, while the input search space is kept in the binary form. Also, compared to the traditional analysis methods, where the worst case is first obtained for each block, the obtained mismatch bound on the overall datapath is tighter and more accurate.

While we used the maximum absolute mismatch as a precision measure, other measures are possible within the proposed framework. We believe that these techniques can be used in future for floating-point verification.

5. References

[1] R.E. Bryant and Y-A. Chen, "Verification of Arithmetic Functions with Binary Moment Diagrams", *Proc. DAC*, pp. 535-541, 1995.

[2] M. Huhn, K. Schneider, Th. Kropf and G. Logothetis, "Verifying Imprecisely Working Arithmetic Circuits", *Proc. DATE*, pp. 65-69,1999.

[3] S. Kimura, "Residue BDDs and its Application to the Verification of Arithmetic Circuits", *Proc. DAC*, pp. 123-126, 1995.

[4] A. Peymandoust and G. De Micheli, "Symbolic Algebra and Timing-Driven Data-Flow Synthesis", *Proc. International Conference on Computer-Aided Design*, pp. 300-305, 2001.

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

[6] K. Radecka and Z. Zilic, "Arithmetic Transforms for Verifying Compositions of Sequential Datapaths", *Proc. ICCD*, pp. 348-353, 2001.

[7] J. Smith and G. De Micheli, "Polynomial Circuit Models for Component Matching in High-level Synthesis", *IEEE Trans. VLSI*, pp. 783-800, Dec. 2001.

[8] Synopsys Inc, "*Co-centric Fixed Point Designer Datasheet*", 2002.

[9] Z. Zilic and K. Radecka, "On Feasible Multivariate Interpolations over Arbitrary Fields", *Proc. of ACM International Symposium on Symbolic and Algebraic Computing"* pp. 67-74, Vancouver, Aug. 1999.