A Hybrid Approach for Equivalence Checking Between System Level and RTL Descriptions

Bijan Alizadeh
VLSI Design and Education Center (VDEC),
University of Tokyo
alizadeh@cad.t.u-tokyo.ac.jp

Masahiro Fujita
VLSI Design and Education Center (VDEC),
University of Tokyo
fujita@ee.t.u-tokyo.ac.jp

ABSTRACT
In this paper we present a hybrid method to check the equivalence between an algorithmic specification in C (ASC) as a golden model and RTL implementation in Verilog (RTL). This method is able to look for equivalent nodes automatically without needs for correspondence outputs information. The approach decomposes the designs using cut-planes and expresses the design behavior in a hybrid domain of Boolean and arithmetic operations. A canonical word level representation called Taylor Expansion Diagram (TED) [1] is used to express word-level information whereas BDD or SAT can be used to represent Boolean expressions.

We apply this method to some non-trivial test-cases. In order to do so, equivalence verification between a C-code specification and Verilog RTL implementation of some benchmarks which are components of real system on chip designs in industry is demonstrated to prove the validity of the approach.

1. INTRODUCTION
Rising complexity of modern systems on a chip (SoCs) made design teams create system level models in C, C++, SpecC, SystemC or SystemVerilog, since a higher abstraction level enables faster design changes and more thorough validation due to a higher simulation speed [2]. On the other hand, because of a lack of mature system level synthesis, they are forced to manually recreate their design in RTL to run them through the traditional hardware design flow. This manual process can introduce functional errors. And an equivalence checking method will be desirable to ensure that the design was carried over to RTL from system level description without problem.

Some research has been done on sequential and combinational equivalence checking. In [3] sequential compare point has been introduced to split the equivalence checking problem space to smaller problems that can be handled by the lower level engines. This method is not scalable in the number of cycles since it uses a lower level SAT solver. In addition, observable points into the two descriptions must have same names. In other words, it is necessary to determine which intermediate or output signals of the two descriptions should be checked for equivalence. In [4] a symbolic simulation method is used to check combinational equivalence of two components. This method checks all paths through the two components. Although they proposed early cut point insertion for unrolling loops to overcome logical complexity, it suffers from complexity blow up because it expresses whole design in BDD and word level information is ignored.

In connection with the above stated facts, the basic idea comes from using a hybrid method to prove the equivalence of two arithmetic expressions as well as Boolean expressions. This hybrid proof is very critical in equivalence checker engines because traditional lower level SAT and BDD based solvers suffer from memory explosion problem when dealing with wide arithmetic operations.

A hybrid method of word-level solving for arithmetic parts and lower level SAT solving for Boolean parts is presented in this paper that allows to prove the ASC and RTL are functionally equivalent. Moreover, in contrast to previous approaches [2-4], our technique is able to automatically find equivalent nodes and there is no need to specify state or output mappings. The proposed approach simulates two models symbolically by starting from a cut-plane. A cut-plane is defined as a set of cut-points should be specified in the specification. At each plane, related symbolic expressions of the specification and implementation are constructed and in a similar manner as described in [3] the sequential equivalence checking problem will be converted to a combinational equivalence checking. After that TED, as a word-level representation, is used to avoid bit level analysis as much as possible due to logical complexity blow-up in the arithmetic expressions.

The important point to be noted here is that although TED was created for equivalence checking, it has never used to verify sequential equivalence between ASC and RTL descriptions. On the other hand, in order to evaluate our approach on some industrial benchmarks, we have used TED as a preliminary phase before developing our word-level package. Our results show that the proposed method is applicable to verify real designs.

The principle contributions of this paper are as follows:

- The main idea in this work is to detect equivalent nodes of the two descriptions automatically by introducing a canonical word level representation. Indeed, this method is able to find equivalent nodes in two descriptions without having any information about correspondence output or state variables.
- Partition Boolean and arithmetic expressions and represent them efficiently so that logical blow-up can easily be handled. To manage bit-slicing of
arithmetic variables which are used in Boolean expressions, the related integer variables are decomposed into other integer variables with shorter bit-width.

The remainder of this paper is organized as follows. Related works are reviewed in Section 2. A framework of our hybrid method is described in Section 3. Experimental results are shown in Section 4 and finally, Section 5 contains a brief conclusion and directions for future work.

2. RELATED WORK

To the best of our knowledge, fewer attempts have been made to apply equivalence checking to the system level and RTL descriptions [3-10]. The authors in [5-6] proposed a solution with CBMC, a C-based bounded model checking engine that takes a C program and a Verilog implementation. The two programs are unwound together, and converted into a Boolean satisfiability checking problem. However, the capacity of CBMC is limited by space and time considerations. This is due to the fact that the reasoning done by this tool is entirely in the Boolean domain.

Another approach to equivalence checking between C descriptions is presented in [7]. This approach extracts the textual differences in the two target programs, and then does a dependence analysis using program slicing, to check for the actual differences in the two programs. It then symbolically simulates this difference and reports the equivalence checking results. This technique, however, is most effective when the two target programs being compared are very similar to each other, in function as well as structure.

In [3] an equivalence checking technique to verify system level design descriptions against their implementations in RTL was proposed. They have presented an automatic technique to compute high level sequential compare points to compare variables of interest in the candidate design descriptions. They start the two design state machines at the same initial state and step the machines through every cycle, until a sequential compare point is reached. At this point the equivalence of the two state machines is proved using a lower (Boolean) level engine which is a zChaff SAT solver [8]. A limitation of this technique is not to be scalable in the number of cycles. As the number of cycles gets larger, the size of the expression grows quadratically, causing capacity problems for the lower level SAT engine. Also high level operations need to be encoded to Boolean expressions so that word level information will be ignored.

In [4] early cut point insertion for checking the equivalence high level software against RTL of combinational components has been proposed. It introduces cut-points early during the analysis of the software model, rather than after generating a low level hardware equivalent. In this way, they overcome the exponential enumeration of software paths as well as the logic blow-up of tracking merged paths. However, it is necessary to synthesize word level information into bit level because they use BDD to represent the symbolic expressions. In addition they do not address sequential equivalence checking.

3. EQUIVALENCE CHECKING APPROACH

In this section, we explain a flow whereby an ASC as a golden model is used to formally verify RTL using a hybrid solving method.

In order to save considerable front-end implementation effort our verification algorithm starts from FSMD since translating from ASC or RTL to an FSMD is a well-known synthesis-like process.

3.1 Equivalence Checking Algorithm

Figure 1 illustrates our proposed equivalence checking flow, where by synthesizing two descriptions based on cut-planes, the related symbolic expressions are extracted. A cut-plane is a set of variables that are interesting for observation in the two models. As a matter of fact, high level synthesis phase pointed out in the figure can be an automatic process for synthesizing RTL description and unrolling loops in ASC description to extract symbolic expressions for a specific cut-plane. But in this work we do this phase manually because our main concentration is on verification algorithm and therefore we avoid front-end implementation as much as possible. As mentioned before, our verification algorithm begins from two FSMDs and we suppose no bugs are inserted during high level synthesis phase.

![Figure 1. Sequential Equivalence Checking Flow](image-url)
describe a module named \textit{idctrow}. Therefore, we are easily able to specify 64 integer outputs of \textit{idctrow} part as a cut-plane. Obviously, it is only necessary to determine some cut-planes in the two descriptions rather than specifying corresponding of variables in the two descriptions as done in [3].

After that, FSMDs of the two systems are traversed in a similar way proposed in [3], but after every cycle appropriate symbolic expressions of the implementation are extracted and during word-level representing equivalent nodes will be found automatically because of canonical form of TED representation. For example suppose arithmetic expressions, extracted from the two designs at a cut-plane, are given. In order to check whether they are equivalent, first the specification is read into TEDify package and the number of TED nodes is reported using \textit{info} command (Before). Then the implementation is fed into the TEDify package and the number of TED nodes is reported (After). If Before = After, the two expressions are considered equivalent. Otherwise, they are inequivalent and by tracing TED nodes it is possible to generate a counter-example.

Whenever some nodes are found equivalent, primary inputs are introduced in their place and during the high level synthesis of the specification and implementation this modification will be applied, as shown in Figure 1, to keep memory usage as low as possible. This process repeats until no cut-plane is available. We assume that intermediate results can be found before primary outputs are reached. This assumption is correct for each design in which two descriptions are developed based on a common algorithm. In spite of this assumption, the proposed method is able to handle this case when dealing with large designs such as FFT64 as demonstrated in Section 4.

In hybrid equivalence checking approach pointed out in Figure 1, Boolean and arithmetic expressions that are in Boolean and integer domains respectively, are extracted as shown in Figure 2. Then lower level SAT checking can be used to solve Boolean expressions and TED as a word-level canonical representation is used to express arithmetic expressions. By applying this hybrid representation we will be able to use word level information and in contrast to the proposed method in [3], we do not need to encode arithmetic expressions into bit levels. This is one of reasons why our method is applicable to real systems in industry.

If bit-slicing of some word-level variables (see \textit{Var} as a word-level variable and \textit{Var}[j] as a bit-slice of \textit{Var} in Figure 2) are used in Boolean part, we can deal with the interaction between two parts by a decomposition technique. For doing so, whenever a decision is made in Boolean part, the appropriate value which can be a symbolic value \textit{x} as illustrated in Figure 2, is feedback to the arithmetic part and the related variable, i.e., \textit{Var}, is decomposed into other word-level variables according to the following equation:

\[
\text{Var} = 2^{(i+1)} \cdot \text{Var}_i + 2^i \cdot x + \text{Var}_i
\]

where \textit{Var}_i, \textit{x} and \textit{Var}_i are new integer variables.

![Figure 2. Hybrid Equivalence Checking Approach](image)

### 3.2 Example

Figure 3 depicts two main parts of C code of IDCT testcase called \textit{IDCTrow} and \textit{IDCTcol} which are iterated for 8 times. In each iteration they take 8 integer variables as input and after computing some arithmetic and shift operations those variables are updated. The RTL description consists of four main parts \textit{IDCTrow}, \textit{IDCTcol}, \textit{IDCTcnt} and \textit{Memory} as shown in Figure 4. \textit{IDCTrow} and \textit{IDCTcol} read 8 integer variables from \textit{Memory} part at different clock cycles and after applying some arithmetic and shift operations, the results are written back into the \textit{Memory} part at different clock cycles. The \textit{IDCTcnt} provides control signals to other parts and controls the whole process.

We arrive at the following cut planes in ASC, as shown in Figure 3:

- The first to eighth cut-planes are specified as output of the first to eighth iterations in \textit{IDCTrow} respectively. This phase receives block[64] as 64 integer inputs and the results are saved into block[64] as 64 integer outputs.

- The ninth to sixteenth cut-planes are specified as output of the ninth to sixteenth iterations in \textit{IDCTcol} respectively which get 64 integer variables of block[i], updated by the \textit{IDCTrow} part, and after performing some arithmetic and logical operations, the results are saved into the related block[i] again.

Now consider two descriptions in Figure 5 which are some part of C and RTL codes of IDCT test case. These net-lists are extracted by traversing the FSMDs of two descriptions in backward direction. This is a high level synthesis phase which can be done manually or using any high level synthesis tools. After providing the input of TEDify package [1], we have applied them to this package and the related TED was extracted as shown in Figure 6.
In this representation, dashed and solid lines indicate 0-child and 1-child respectively. We have used the topological order in which the signals appear in the C code, so order of variables is \( W_7 > b_1 > b_7 > W_1 > W_3 > W_5 > b_5 > b_3 > W_8 \). In this figure, top node with two incoming edges indicates that \( r_{70} \) is equivalent to \( x_{42} \) and they both express \(-W_7(b_1+b_7)W_8 + (b_1+b_7)W_1 + W_3(b_5-b_3)W_8 - (W_5(b_5+b_3)W_8-128)\).

### 4. EXPERIMENTAL RESULTS

We have applied this method to 64-point Fast Fourier Transform (FFT64), Inverse fast Discrete Cosine Transform (IDCT), Differential Equation solver (DIFFEQ), MPEG2 and Finite Impulse Response (FIR16) as preliminary experimental results. Table 1 summarizes general information about the benchmark circuits. First column (benchmark) gives the benchmark’s name, whereas second (#spec) and third (#impl) columns provide the number of lines of the specification (C code) after unrolling and the number of lines of the implementation (RTL code) after synthesizing respectively.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>#spec</th>
<th>#impl</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT64</td>
<td>1412</td>
<td>1640</td>
<td>3052</td>
</tr>
<tr>
<td>IDCT</td>
<td>690</td>
<td>984</td>
<td>1674</td>
</tr>
<tr>
<td>MPEG2</td>
<td>340</td>
<td>420</td>
<td>760</td>
</tr>
<tr>
<td>DIFFEQ</td>
<td>90</td>
<td>125</td>
<td>215</td>
</tr>
<tr>
<td>FIR16</td>
<td>30</td>
<td>51</td>
<td>81</td>
</tr>
</tbody>
</table>

For instance consider IDCT test case. Its C code has 690 lines, 64 integer inputs and 64 integer outputs. It contains three loops which are iterated 64 and 8 times and in each iteration, 44 additions, 31 subtractions, 22 multiplications and 30 shift operations are computed on 16-bits operands which are symbolic variables. The RTL code has 984 lines which consist of 197 flip-flops and 2971 gates.

In order to prove that our proposed approach can be applied to large designs even though intermediate results do not exist, FFT64 benchmark is a great test case. It has too many data-dependent computations and it is also the most computationally intensive part of a high-data-rate communication system, e.g. 802.11a Transmitter, so that occupies almost 94.8% of total area of such a system [11].

In the following we present a set of experiments of equivalence checking which are divided into two categories: (1) without Boolean variables and (2) with Boolean variables. In order to have a better estimation of run time and memory requirements, we have presented...
experimentation results of the second category, where Boolean variables are used as well as word-level variables, completely in TED rather than using SAT or BDDs. All experiments have been carried out on an Intel 2.1Ghz Core Duo and 1 Giga-Byte of main memory running Windows XP.

4.1 Equivalence Checking Without Boolean Variables

In the first category, we only apply the proposed method to all benchmarks while no decomposition happens. Experimental results are reported in Table 2. In this table first column (Benchmark) gives benchmark's name, whereas in columns N_TED, N_Var, Mem and CPU_T, the number of TED nodes, the number of variables, memory usage and CPU time needed for equivalence checking of the two descriptions are provided respectively.

Regardless of above-mentioned assumption about looking for intermediate results to reduce size of the problem as much as possible, experimental result of FFT64 test case indicates that TED is able to represent large designs where no intermediate results are found. Although the number of TED nodes is 11220, it can handle this benchmark in terms of memory and run time which are 15.7MB and 510 seconds respectively.

Table 2. Experimental results (without Boolean part)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>N_TED</th>
<th>N_Var</th>
<th>Mem</th>
<th>CPU_T</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT64</td>
<td>11220</td>
<td>190</td>
<td>15700</td>
<td>510</td>
</tr>
<tr>
<td>IDCT</td>
<td>3992</td>
<td>64</td>
<td>4220</td>
<td>91</td>
</tr>
<tr>
<td>DIFFEQ</td>
<td>516</td>
<td>4</td>
<td>3400</td>
<td>11</td>
</tr>
<tr>
<td>MPEG2</td>
<td>472</td>
<td>64</td>
<td>1300</td>
<td>5</td>
</tr>
<tr>
<td>FBL16</td>
<td>10</td>
<td>9</td>
<td>700</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 3. IDCT experimental results (without Boolean part)

<table>
<thead>
<tr>
<th>IDCT part</th>
<th>N_TED</th>
<th>N_Var</th>
<th>Mem</th>
<th>CPU_T</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDCTrow</td>
<td>50</td>
<td>8</td>
<td>630</td>
<td>1</td>
</tr>
<tr>
<td>IDCTrowall</td>
<td>400</td>
<td>64</td>
<td>1440</td>
<td>6</td>
</tr>
<tr>
<td>IDCTcol1</td>
<td>64</td>
<td>8</td>
<td>630</td>
<td>1</td>
</tr>
<tr>
<td>IDCTcolall</td>
<td>512</td>
<td>64</td>
<td>1530</td>
<td>7</td>
</tr>
</tbody>
</table>

4.2 Equivalence Checking With Boolean Variables

In the second category, Boolean part is considered as well as arithmetic part. As mentioned in Section 3, if bit-slice of an integer variable is used in Boolean part, it needs to be decomposed into shorter variables. We have divided experimental results of this category into four configurations: (1) bit-slicing of one variable at different bit positions, (2) bit-slicing of a set of variables at 10th bit position (3) bit-slicing of a set of variables at 5th and 10th bit positions and (4) bit-slicing of a set of variables at 5th, 10th, 15th, and 20th bit positions.

In the first configuration, we assume a variable should be decomposed at different bit positions. We have applied the proposed method to 8 iterations of IDCTrow module, i.e., $IDCTrowall$, whereas $b_{5+8i}$ for $i=0$ to 7 (i.e., $b_5, b_{13}, b_{21}, b_{29}, b_{37}, b_{45}, b_{53}, b_{61}$), have been decomposed into other variables at different bit positions. Table 4 gives the results, whereas memory usage and run times are measured in Kilo-bytes and seconds respectively. Column CASE describes different cases as follows:

case1: bit position 3, i.e., $b_{5+8i}[3]=b_{5+8i03}$
case2: bit positions 3 and 7
case3: bit positions 3, 7 and 10
case4: bit positions 3, 7, 10 and 12
case5: bit positions 3, 7, 10, 12 and 16
case6: bit positions 3, 7, 10, 12, 16 and 20
case7: bit positions 3, 7, 10, 12, 16, 20 and 25
case8: bit positions 3, 7, 10, 12, 16, 20, 25 and 30

Table 4. Experimental results (with Boolean part and Bit-slicing of a variable at different bit positions)

<table>
<thead>
<tr>
<th>CASE</th>
<th>N_TED</th>
<th>N_Var</th>
<th>Mem</th>
<th>CPU_T</th>
</tr>
</thead>
<tbody>
<tr>
<td>case1</td>
<td>528</td>
<td>80</td>
<td>1490</td>
<td>&lt; 9</td>
</tr>
<tr>
<td>case2</td>
<td>656</td>
<td>96</td>
<td>1560</td>
<td>&lt; 12</td>
</tr>
<tr>
<td>case3</td>
<td>784</td>
<td>112</td>
<td>1610</td>
<td>&lt; 16</td>
</tr>
<tr>
<td>case4</td>
<td>912</td>
<td>128</td>
<td>1660</td>
<td>&lt; 21</td>
</tr>
<tr>
<td>case5</td>
<td>1040</td>
<td>144</td>
<td>1720</td>
<td>&lt; 26</td>
</tr>
<tr>
<td>case6</td>
<td>1168</td>
<td>160</td>
<td>1770</td>
<td>&lt; 30</td>
</tr>
<tr>
<td>case7</td>
<td>1296</td>
<td>176</td>
<td>1830</td>
<td>&lt; 38</td>
</tr>
<tr>
<td>case8</td>
<td>1408</td>
<td>192</td>
<td>1880</td>
<td>&lt; 44</td>
</tr>
</tbody>
</table>

In the next set of experiments eight cases of a set of variables are taken into account to apply various bit-
slicing to all variables in IDCT test case. These cases are described as follows:

- **case0**: \( b_{8j} \); for \( j = 0 \) to 7
- **case\( i \) and case\( i-1 \)**; for \( j = 0 \) to 7 and \( i = 1 \) to 7

For instance, case\( 0 \) only includes \( b_0, b_8, b_{16}, b_{24}, b_{32}, b_{40}, b_{48}, \) and \( b_{56} \) variables while case\( 7 \) consists of all 64 variables. Each case was executed with three configurations of bit-slicing as follows:

**1-pos**: bit-slicing at bit position 10 (one bit position)

**2-pos**: bit-slicing at bit positions 5 and 10 (two bit positions)

**4-pos**: bit-slicing at bit positions 5, 10, 15, and 20 (four bit positions)

Table 5 summarizes the results of each configuration. The number of TED nodes, the number of variables, memory usage (Kilo-Bytes) and CPU time (Seconds) required checking the equivalence of the two descriptions for different configurations (1-pos, 2-pos and 4-pos) are given.

For example, row **case1** provides experimental results whereas \( b_{8j} \) (i.e., \( b_8, b_{16}, b_{24}, b_{32}, b_{40}, b_{48}, \) and \( b_{56} \)) and \( b_{1+8j} \) (i.e., \( b_1, b_9, b_{17}, b_{25}, b_{33}, b_{41}, b_{49}, \) and \( b_{57} \)) must be decomposed into other variables according to the following equations:

**1-pos**:

\[
\begin{align*}
b_{8j} &= 2048*b_{(8j)H} + 1024*b_{(8j)10} + b_{(8j)L} \\
b_{1+8j} &= 2048*b_{(1+8j)H} + 1024*b_{(1+8j)10} + b_{(1+8j)L}
\end{align*}
\]

**2-pos**:

\[
\begin{align*}
b_{8j} &= 2097152*b_{(8j)H2} + 1048576*b_{(8j)10} + 65536*b_{(8j)H1} + 32768*b_{(8j)05} + 2048*b_{(8j)H2} + 1024*b_{(8j)10} + 64*b_{(8j)H1} + 32*b_{(8j)05} + b_{(8j)L} \\
b_{1+8j} &= 2097152*b_{(1+8j)H2} + 1048576*b_{(1+8j)10} + 65536*b_{(1+8j)H1} + 32768*b_{(1+8j)05} + 2048*b_{(1+8j)H2} + 1024*b_{(1+8j)10} + 64*b_{(1+8j)H1} + 32*b_{(1+8j)05} + b_{(1+8j)L}
\end{align*}
\]

**4-pos**:

\[
\begin{align*}
b_{8j} &= 2097152*b_{(8j)H4} + 1048576*b_{(8j)20} + 65536*b_{(8j)10} + 32768*b_{(8j)15} + 2048*b_{(8j)H2} + 1024*b_{(8j)10} + 64*b_{(8j)H1} + 32*b_{(8j)05} + b_{(8j)L} \\
b_{1+8j} &= 2097152*b_{(1+8j)H4} + 1048576*b_{(1+8j)20} + 65536*b_{(1+8j)10} + 32768*b_{(1+8j)15} + 2048*b_{(1+8j)H2} + 1024*b_{(1+8j)10} + 64*b_{(1+8j)H1} + 32*b_{(1+8j)05} + b_{(1+8j)L}
\end{align*}
\]

Table 5. Experimental results (with Boolean part and Bit-slicing of a set of variables at several bit positions)

<table>
<thead>
<tr>
<th>CASE</th>
<th>Number of TED Nodes</th>
<th>Number of Variables</th>
<th>Memory Usage (KiloBytes)</th>
<th>CPU Time (Seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1-pos</td>
<td>2-pos</td>
<td>4-pos</td>
<td>1-pos</td>
</tr>
<tr>
<td>case0</td>
<td>448</td>
<td>496</td>
<td>592</td>
<td>80</td>
</tr>
<tr>
<td>case1</td>
<td>576</td>
<td>752</td>
<td>1080</td>
<td>96</td>
</tr>
<tr>
<td>case2</td>
<td>672</td>
<td>944</td>
<td>1464</td>
<td>112</td>
</tr>
<tr>
<td>case3</td>
<td>800</td>
<td>1200</td>
<td>1976</td>
<td>128</td>
</tr>
<tr>
<td>case4</td>
<td>848</td>
<td>1296</td>
<td>2144</td>
<td>144</td>
</tr>
<tr>
<td>case5</td>
<td>976</td>
<td>1552</td>
<td>2656</td>
<td>160</td>
</tr>
<tr>
<td>case6</td>
<td>1072</td>
<td>1744</td>
<td>3040</td>
<td>176</td>
</tr>
<tr>
<td>case7</td>
<td>1200</td>
<td>2000</td>
<td>3552</td>
<td>192</td>
</tr>
</tbody>
</table>

Figure 7. Comparison of 1-position, 2-positions and 4-positions decomposing in terms of memory usage and CPU time.
5. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a hybrid solving method to check the equivalence between the ASC as a golden model and RTL. This approach can be seen as an integrated technique which uses bit-level SAT checking and word-level canonical representation to lead to a more robust equivalence checking solution. It incorporates benefits of both so that represents Boolean and Arithmetic expressions in a canonical form which is a key point to equivalence checking problems. Since both the ASC and RTL use arithmetic computations, it is necessary to propose a technique that proves equivalence of such operations without synthesizing them down to bit-level Boolean representations like SAT or BDDs. Another advantage, in contrast to other methods described in [3-6], is the ability to automatically find equivalent nodes without requiring any information about correspondence variables.

One possible avenue for future work would be to run this approach on other system-based designs in which more complicated optimizations such as retiming have been done. We are also going to develop an environment in which high level synthesis phase will be integrated to our hybrid equivalence checking engine for achieving better performance.

ACKNOWLEDGEMENTS

We would like to thank the Semiconductor Technology Academic Research Center (STARC) for supporting this research work.

REFERENCES


