Hand-in-hand Verification of High-level Synthesis

C. Karfa, D. Sarkar, C. Mandal
Department of Computer Sc. & Engg.
Indian Institute of Technology, Kharagpur
WB 721302, INDIA
{ckarfa, ds, chitta }@iitkgp.ac.in

C. Reade
Kingston Business School
Kingston University
England KT2 7LB, UK
Chris.Reade-@king.ac.uk

ABSTRACT

This paper describes a formal verification methodology of high-level synthesis (HLS) process. The abstraction level of the input to HLS is so high compared to that of the output that the verification has to proceed hand-in-hand with the synthesis process. The HLS verification is performed in three phases in this work. The verification method is based on equivalence checking of two finite state machines with data-paths (FSMDs). Unlike most reported works that targets the individual phases independently, the proposed method applies to all these three phases. The method is strong enough to accommodate control structure modification of the original behaviour, application of several code motion techniques during scheduling and register optimization during register allocation. It can also verify the correctness of the controller. A hand-in-hand synthesis and verification tool SAST has been developed and tested for effectiveness on several HLS benchmark circuits.

Categories and Subject Descriptors

B.7.2 [**Integrated Circuits**]: VLSI; B.5.2 [**Hardware**]: Register Transfer-level Implementation—*Verification*

General Terms

Verification

Keywords

Formal Verification, Equivalence Checking, FSMD model, High-level Synthesis

1. INTRODUCTION

The high-level synthesis (HLS) process consists in translating a *behavioural specification* into an *RTL structural description* comprising a *data-path* and a *controller* so that the data transfers under the control of the *controller* exhibit the *specified behaviour* [5]. The synthesis process involves several inter-dependent sub-tasks such as, scheduling, allocation, binding and data-path and controller generation. The operations in the behavioural description are assigned time steps through scheduling process. The allocation process computes the minimum number of functional units and registers required to synthesize the design based on the scheduling information and selects proper operators from the component library. The variables are mapped to registers and the operations are mapped to functional units by the binding process. The next task is to set the data-path by providing a proper interconnection path from the source to the destinations for every register transfer (RT) operation. Finally, the controller is generated which produces control signals needed for all the data transfers required among the data-path elements in different control steps and realizes the control flow of the behaviour.

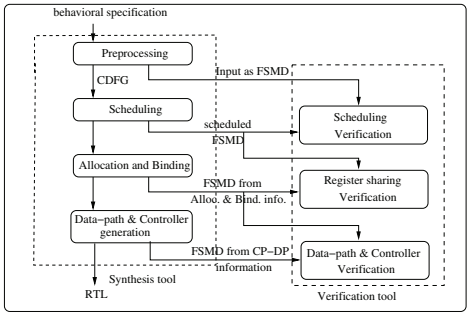

Figure 1: Hand-in-hand synthesis and verification framework

The complexity of the present VLSI systems is very high. The input specification of HLS is given at a very high abstraction level compared to the abstraction level of the output. Also, optimizations of control steps in the scheduling phase, of registers, functional units and interconnections in the allocation and binding phase and of control signals in the controller generation phase are performed. Therefore, a phase-wise verification technique with opportunity to handle the difficulties of each synthesis sub-task separately is necessary for HLS verification. An end-to-end verification method for HLS is very tough and also inadequate in locating the exact sources of errors. We consider a three phase verification which is performed hand-in-hand with the synthesis process. The proposed platform is depicted in figure 1. Phase-I verifies the scheduling process. In phase-II, correctness of sharing of registers among the variables of the input behaviour is verified. This phase is called *register sharing verification*. The correctness of the data-path as well as the controller are verified in phase-III. The input and output of each synthesis phase are encoded as finite state machines with data-paths (FSMD) and verification process is based on equivalence checking of these two FSMDs.

GLSVLSI'07, March 11–13, 2007, Stresa-Lago Maggiore, Italy.

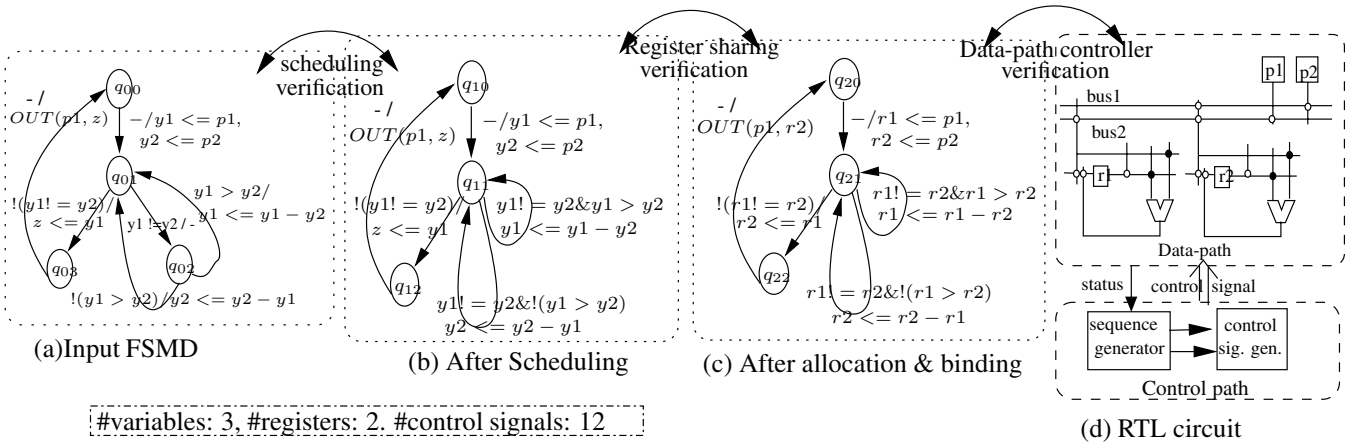

Figure 2: High-level synthesis flow for GCD example

Related Works

The methodologies reported in [3] and [9] are applicable to the scheduling verification phase. Most of these methods are likely to fail for path-based scheduling [2] as well as when code motion techniques [6] are used by the scheduler. Several techniques for verification of the allocation and binding phase are proposed [11], [1], which can be integrated with the synthesis systems which perform little or no register optimization. Most of the verification techniques reported in the literature are applicable to only certain phases of high-level synthesis and they are likely to fail in some circumstances which are very common to the modern synthesis tools. None of the works is generalized enough to handle all the phases of the HLS process.

Aim of the Present Work

In this work, we proposed a *three phases* verification techniques for high-level synthesis. The verification performs *hand-in-hand* with the synthesis process as shown in 1. We have formalized an FSMD based equivalence checking method. The proposed equivalence checking method is applied to all the three phases of HLS verification. The method is also tuned to handle the difficulties of each phase of verification. The construction of the FSMD from the data-path and the controller informations is also discussed.

The paper is organized as follows. An example on high-level synthesis is given in section 2 to illustrate the motivation of this work. The equivalence problem of FSMDs is formulated in section 3. The three phases of the high-level synthesis verification are discussed in the section 4. The experimental results are shown in section 5. The paper concludes in section 6.

2. AN ILLUSTRATIVE EXAMPLE

The example in figure 2 describes the synthesis flow for the GCD example. The FSMDs at the input, after scheduling, after allocation and binding phase and the final generated RTL circuits are shown in the figure. The total number of time steps required to execute and the control structure of the input behaviour (figure 2 (a)) are modified by the scheduler as shown in figure 2 (b). The behaviour after allocation and binding phase is in terms of registers (registers are denoted as r_{ij} in figure 2 (c)); also the number of registers is minimized in the allocation and binding phase. For this example, the number of registers is 3 whereas the number of variables in the input behaviour is 2. Finally, the RTL circuit consisting of a distinct control path and a data path is generated. The RTL circuit may behave erroneously due to some fault in the controller or in the data-path interconnections. It is evident from the figure that the there is no one-to-one correspondence between the input and the output (figure 2 (a) and figure 2 (d)). Hence, an end-to-end verification can hardly identify the exact points of errors. That is why we are opting for phase-wise verification of high-level synthesis with opportunity to handle the difficulties of each phase independently and more accurately.

3. THE EQUIVALENCE PROBLEM FORMULATION

An FSMD (*finite state machine with data-path*) is a universal specification model, proposed by Gajski in [5], that can represent all hardware designs. The model is used in the present work with the addition of a reset state. The FSMD is formally defined as an ordered tuple $\langle Q, q_0, I, V, O, f : Q \times 2^S \rightarrow Q, h : Q \times 2^S \rightarrow U \rangle$, where Q is the finite set of control states, q_0 is the reset state, I is the set of input signals, V is the set of storage variables, O is the set of output signals, f is the state transition function, h is the update function of the output and the storage variables, $U = \{x \Leftarrow e | x \in O \cup V \text{ and } e \in E\}$ represents a set of storage or output assignments, where E represents a set of arithmetic expressions over the set $I \cup V$ and $S = \{R(a,b) | a, b \in E \text{ and } R \text{ is any arithmetic relation}\}$ represents a set of status signals as arithmetic relations between two expressions from the set E.

A *path* α from q_i to q_j, where $q_i, q_j \in Q$, is a finite transition sequence of states where all the states are distinct. Only, the state q_j may be identical to q_i. The *condition of execution of the path* α, R_α, is a logical expression over the variables in V and inputs I such that R_α is satisfied by the (initial) data state of the path iff the path α is traversed. The *data transformation of a path* α over V, r_α, is the tuple $\langle s_\alpha, O_\alpha \rangle$, where s_α is an ordered tuple $\langle e_i \rangle$ of algebraic expressions over the variables in V and the inputs in I such that the expression e_i represents the value of the variable v_i after the execution of the path in terms of the initial data state of the path and O_α represents the output list along the path α.

A *computation* of an FSMD is a finite walk from the reset state q_0 back to itself without having any intermediary occurrence of q_0. Such a computational semantics of an FSMD is based on the assumption that a revisit of the reset state means the beginning of a new computation and each computation terminates. Any computa-

tion c of an FSMD M can be looked upon as a computation along some concatenated path $[\alpha_1\alpha_2\alpha_3...\alpha_k]$ of M such that the path α_1 emanates from and the path α_k terminates in the reset state q_0 of M and $\alpha_i, 1 \leq i \leq k$, may not all be distinct. Two computations c_1 and c_2 of an FSMD are said to be equivalent if $R_{c_1} = R_{c_2}$, $r_{c_1} = r_{c_2}$, where R_{c_1} and r_{c_1} represents the condition of execution and data transformations of c_1, respectively and R_{c_2} and r_{c_2} represent the same for c_2. Computational equivalence of two paths can be defined in a similar manner. The fact that a path p_1 is computationally equivalent to p_2 is denoted as $p_1 \simeq p_2$.

Definition 1. **Path cover of an FSMD:** *A finite set of paths $P = \{p_0, p_1, p_2, \ldots, p_k\}$ is said to cover an FSMD M if any computation c of M can be looked upon as a concatenation of paths from P.*

3.1 Correctness Problem

Let M_0 and M_1 be two FSMD representations at the input and the output, respectively, of any phase of high-level synthesis. Our main goal is to verify whether M_0 behaves exactly as M_1. This means that for all possible input sequences, M_0 and M_1 produce the same sequences of output values and eventually, when the respective reset states are re-visited, they are visited with the same storage element values. In other words, for every computation from the reset state back to itself of one FSMD, there exists an equivalent computation from the reset state back to itself in the other FSMD and vice-versa. The following definition captures the notion of equivalence of FSMDs.

Definition 2. *Two FSMDs M_0 and M_1 are said to be computationally equivalent if for any computation c_0 of M_0, there exists a computation c_1 of M_1 such that c_0 and c_1 are computationally equivalent and vice-versa.*

From definition 1 and definition 2, the following theorem can be concluded.

Theorem 1. *Two FSMDs M_0 and M_1 are computationally equivalent if there exists a finite cover $P_0 = \{p_{00}, p_{01}, \ldots, p_{0l}\}$ of M_0 for which there exists a set $P_1^0 = \{p_{10}^0, p_{11}^0, \ldots, p_{1l}^0\}$ of paths of M_1 such that $p_{0i} \simeq p_{1i}^0$, $0 \leq i \leq l$, and vice-versa.*

The equivalence problem formulation can be found in more detail in [8].

3.2 The Equivalence Checking Method

The above theorem, therefore, suggests an equivalence checking method which consists of the following steps:

1. Construct the set P_0 of paths of M_0 so that P_0 covers M_0. Let $P_0 = \{p_{00}, p_{01}, \cdots, p_{0k}\}$.
2. Show that $\forall p_{0i} \in P_0, there\ exists\ a\ path\ p_{1j}$ of M_1 such that $p_{0i} \simeq p_{1j}$.
3. Repeat steps 1 and 2 with M_0 and M_1 interchanged.

Because of loops it is difficult to find a path cover of the whole computation comprising only finite paths. So any computation is split into paths by putting *cutpoints* at various places in the FSMD so that each loop is cut in at least one cutpoint. The set of all paths from a cutpoint to another cutpoint without having any intermediary cutpoint is a path cover of the FSMD. The method of decomposing an FSMD by putting cutpoints is identical to the Floyd-Hoare's method of program verification [4, 7]. We choose the cutpoints in any FSMD as follows.

1. The reset state.
2. Any state with more than one outward transition.

Obviously, cutpoints chosen by the above rules cut each loop of the FSMD in at least one cutpoint, because each internal loop has an exit point.

This equivalence checking method is applied to all the three phases of HLS verification. However, some additional entities such as, mapping informations from the variables to the registers, etc, may be needed depending upon the functionalities of each phase. The *equivalence of two paths* and the *cutpoints selection rule* are defined in this section. But, they might differ in the different phases depending upon the requirements of each phase. We will discuss the modifications in the following section.

4. HAND-IN-HAND VERIFICATION

The basic steps of equivalence checking, i.e., constructing the path cover by inserting cutpoints in one FSMD and finding the equivalent path of each member of this path cover in the other FSMD, can be applied to all the phases of HLS verification. In this section, the objectives of each HLS verification phase and the required modifications to adapt the basic equivalence method are discussed. The FSMDs corresponding to the behaviours at the input to HLS, after scheduling, after allocation and binding and at the output of HLS are denoted as $M_0 = \langle Q_0, q_{00}, I, V_0, O, f_0, h_0\rangle$, $M_1 = \langle Q_1, q_{10}, I, V_1, O, f_1, h_1\rangle$, $M_2 = \langle Q_2, q_{20}, I, V_2, O, f_2, h_2\rangle$ and $M_3 = \langle Q_3, q_{30}, I, V_3, O, f_3, h_3\rangle$, respectively, in this work. The construction of M_3 from the data-path and the controller informations is not straightforward. This task is also discussed in this work.

4.1 Scheduling Verification

Objectives:

The scheduler tries to schedule all the operations of the input behaviour in minimum number of time steps. To do so, the input behaviour to the scheduler is modified in several ways. For example, the control structure of the input behaviour may be modified by the path-based scheduler [2] as it tries to merge some consecutive path segments of the input behaviour. Also, incorporation of several code motion techniques [6] in the scheduling process leads to transformations in the input specification. For example, some of the operations may be moved beyond the conditional statement (speculation, reverse speculation), extra variables may be used to rename some of the variables in the input behaviour (renaming) and some of the variables and operations of the input behaviour may be eliminated (dead code elimination). Naturally, the results of scheduling do not have a one-to-one correspondence with the input. Hence, it is important to ensure that the scheduling process preserves the behaviour of the original specification, irrespective of the scheduling technique used.

Required modifications:

Due to application of different code motion techniques like renaming and dead-code eliminations along with scheduling, the variable sets V_0 of M_0 and V_1 of M_1 may not be equal. So, the expressions that represent the condition of execution or the data transformation of any path in M_0 or in M_1 need to be restricted to the variable set $V_0 \cap V_1$ and the input set I. It means that they are defined over the variables in $V_0 \cap V_1$ and the inputs I. If the condition of execution and the data transformation of any path of M_0 contain some variable(s) from the set $V_0 - V_1$, then it will become undefined; the same is applicable for any path of M_1 if its condition of execution and the data transformation contain some variable(s) from $V_1 - V_0$.

Also, the final values of only the variables in $V_0 \cap V_1$ are compared while checking the equivalence of two paths. However, the situation where the restrictions on R_α and r_α of a path α make them undefined usually do not occur as argued below. Some of the variables of V_0 may not exist in V_1 when the scheduler eliminates some dead code involving these variables in $(V_0 - V_1)$. Clearly, they have no effect in the condition of execution or in the data transformation of any path in M_0. On the other hand, the scheduler generally uses some extra variables to reduce the data dependencies among the variables to increase the parallelism among the operations in the behaviour. These variables are first assigned some values in terms of $V_0 \cap V_1$ and I and used subsequently. Obviously, these variables (in $V_1 - V_0$) will not occur in the condition of execution or in the data transformation of any path in M_1.

The control structure of the input behaviour may be modified by the scheduler due to path based scheduling algorithm [2] or the application of several code motion techniques [6] like speculation, reverse speculation, branch balancing, etc. Accordingly, the rules to find cutpoints defined in the subsection 3.2 do not work always. In the following, we propose one method which combines the first two steps of the method described in subsection 3.2 into one. More specifically, the method constructs a path cover of M_0 and also finds its equivalent path set in M_1 hand-in-hand. The following definition is used in the algorithm.

Definition 3. *Corresponding states: Let* $M_0 = \langle Q_0,\ q_{00}, I, V_0, O, f_0, h_0\rangle$ *and* $M_1 = \langle Q_1, q_{10}, I, V_1, O, f_1, h_1\rangle$ *be the two FSMDs having identical input and output sets,* I *and* O*, respectively, and* $q_{0i}, q_{0k} \in Q_0$ *and* $q_{1j}, q_{1l} \in Q_1$.

- *The respective reset states* q_{00}, q_{10} *are corresponding states.*
- *If* $q_{0i} \in Q_0$ *and* $q_{1j} \in Q_1$ *are corresponding states and there exist* $q_{0k} \in Q_0$ *and* $q_{1l} \in Q_1$ *such that, for some path* α *from* q_{0i} *to* q_{0k} *in* M_0*, there exists a path* β *from* q_{1j} *to* q_{1l} *in* M_1 *such that* $\alpha \simeq \beta$*, then* q_{0k} *and* q_{1l} *are corresponding states.*

4.1.1 The Algorithm

Input: The FSMDs M_0 and M_1.
Output: P_0: a path cover of M_0,
E: ordered pairs $\langle \beta, \alpha\rangle$ of paths of M_0 and M_1, respectively, such that $\beta \in P_0$ and $\beta \simeq \alpha$.

Steps:

1: Let η be the set of corresponding state pairs. Let $\eta \leftarrow \langle q_{00}, q_{10}\rangle$. Insert cutpoints in M_0 using the rule stated in the previous section. Let P_0' be the set of all the paths of M_0 from a cutpoint to a cutpoint having no intermediary cutpoint. Let P_0 and E be empty.

2: If $P_0' = empty$, then return P_0 as a path cover of M_0 and E as the set of ordered pairs of equivalent paths of M_0 (from P_0) and M_1 and *exit (success)*; else go to step 3.

3: Find a path of the form $\langle q_{0i} \Rightarrow q_{0f}\rangle$ from P_0' s.t. q_{0i} has a corresponding state q_{1j}. If no path is obtained, then go to step 4; else go to step 5.

4: Delete all the paths in P_0' which are covered by some path in P_0. If $P_0' \neq empty$, then report "M_0 *may not be equivalent to* M_1" and *exit (failure)*; else return P_0 as a path cover of M_0 and E as a set of ordered pairs of equivalent paths of M_0 (from P_0) and M_1 and *exit (success)*.

5: Let the path obtained in step 2 be $\beta = \langle q_{0i} \Rightarrow q_{0f}\rangle$. Let $\langle q_{0i}, q_{1j}\rangle$ be the corresponding states pair in η. If R_β or r_β is undefined then *exit(failure)* else find a path of M_1 emanating from q_{1j} which is equivalent to the path β. If such a path is found, then go to step 6; else go to step 7.

6: Let this path of M_1 be α. $\eta \leftarrow \eta \bigcup \{\langle endState(\beta), endState(\alpha)\rangle\}$, $E \leftarrow E \bigcup \{\langle \beta, \alpha\rangle\}$, $P_0 \leftarrow P_0 \bigcup \{\beta\}$, $P_0' \leftarrow P_0' - \{\beta\}$. go to step 2.

7: $P_0' \leftarrow P_0' - \{\beta\}$. Extend β $(= \langle q_{0i} \Rightarrow q_{0f}\rangle)$ in M_0 by moving through the cutpoint q_{0f} till the next cutpoints but without moving through the reset state or any cutpoint more than once. Let B_m be the set of all such extensions of the path β. $P_0' \leftarrow P_0' \bigcup B_m$. go to step 8.

8: If $B_m = empty$, then report "β *does not have any equivalent in* M_1 *and cannot be extended*" and *exit (failure)*; else go to step 2.

4.2 Register Sharing Verification

Objectives:

The variables of the scheduled behaviour share registers in the data-path. The number of registers in the data-path is less than equal to the number of variables in the behaviour as most of the high-level synthesis tools perform register optimization during register allocation. The goal of this phase of verification is to ensure the correctness of register sharing among the variables.

Additional informations:

In the allocation and binding phase, the scheduler output is mapped to the hardware with specific intention of using minimum number of registers, FU, muxes, demuxes, etc. Optimization, like reduction of total time to execute, is not considered in this phase. As a result, the control structure of the scheduled FSMD does not get modified in this phase. So, there exists a bijective function, namely *state mapping function*, $f_{sm} : Q_1 \leftrightarrow Q_2$ from the states of M_1 to the states of M_2. The variable set V_1 of M_1 contains all the behavioural variables and the temporary variables introduced by the scheduler and the variable set V_2 of M_2 consists of all the data-path registers. A *register binding function* f_{rb}: $Q_2 \times V_2 \rightarrow V_1$ can be obtained from the life time informations of the variables. Clearly, this function may not be bijective.

Required modifications:

We need to modify the definition of equivalence of paths as the FSMDs M_1 and M_2 involve different variable sets with different cardinalities. An expression e_1 (arithmetic or status) over $V_1 \cup I$ at the state $q_{2,i}$ of M_1 is said to be equivalent to an expression e_2 over $V_2 \cup I$ at state $q_{2,j}$ of M_2 if $f_{sm}(q_{1,i}) = q_{2,j}$ and $e_1 = e_2$ when all the registers $r \in V_2$ occurring in e_2 are replaced by $v \in V_1$, where $v = f_{rb}(q_{2,j}, r)$. Treating the expression e_2 loosely as a function, we denote the above phrase syntactically as $e_1 = e_2\ _o\ f_{rb}$, where 'o' stands for function composition. A path α_1 of M_1 is equivalent to a path α_2 of M_2 if $R_{\alpha_1} = R_{\alpha_2}\ _o\ f_{rb}$ and $r_{\alpha_1} = r_{\alpha_2}\ _o\ f_{rb}$, provided both the paths start with the same initial data state, i.e., all v_k of V_2 and $f_{rb}(q_{2,0}, v_k)$ of V_1 have the same data state initially. The equivalence checking method stated in the subsection 3.2 is directly applicable to this phase with this modified path equivalence definition.

4.3 Data-path and Controller Verification

Objectives:

The final output of the high-level synthesis is a control-path and a data-path (CP-DP) so that (i) all the register transfer operations and the status conditions in M_2 are indeed provided for the DP components and theirs interconnections and (ii) the set of all control signals generated and status signals sensed in the states of the controller FSMD M_3 do realize the corresponding register transfers of FSMD M_2. In other words, the verification task in this phase

is defined as follows: *given the control signal assertion pattern in each state of the controller FSM, it is required to identify the corresponding register transfer operation in the DP and construct the FSMD M_3. Next, check the whether FSMD M_3 is equivalent to the FSMD M_2 or not.*

4.3.1 Construction of FSMD M_3

The following two informations have to be extracted from the CP-DP description in order to accomplish the objectives.

1. *The set of all possible micro-operations in the data-path.* Let this set be denoted as $\mathscr{M}$. A data movement from a data-path component y to another data-path component x is encoded by the micro-operation $x \Leftarrow y$.

2. *The control signal assertion pattern for every micro-operation in $\mathscr{M}$.* A control signal assertion pattern is represented as an ordered n-tuple of the form $\langle\ u_1,\ u_2, \ldots,\ u_n\rangle$, where u_i represents the value of the control signal c_i, n is the number of control signals; $u_i \in \{0,\ 1,\ X\},\ 1 \leq i \leq n$, is the asserted value of c_i. $u_i = X$ implies that the control signal c_i is not required for a particular micro-operation corresponding to the assertion pattern. Let $\mathscr{A}$ be the set of all possible control assertion patterns. So, a function f_{mc} can be constructed from the set $\mathscr{M}$ of all micro-operations possible in the given data-path to the set $\mathscr{A}$ of control signal assertion patterns. Thus, the data-path structure, in its entirety, is captured by the function f_{mc}.

In each state of the FSM, the controller generates a control signal assertion pattern to execute a set of micro-operations in the data-path. So, the next task is to obtain the set of micro-operations $\mathscr{M}_A$ ($\subseteq \mathscr{M}$) for a given control assertion pattern A. It is, however, not possible to obtain the set $\mathscr{M}_A$ of micro-operations directly from the control signal assertion pattern A by examining its individual control signals because a micro-operation may be accomplished by a set of control signals rather than an individual control signal. There is no information available in an assertion pattern to group the control signals so that each group defines a micro-operation around a data-path component. The following definition is in order.

Definition 4. **Superposition of Assertion patterns:**
Let A_1 and A_2 be two arbitrary control signal assertion patterns. Let $\pi_i(A)$ denote the i-th projection of an assertion pattern A which is the asserted value u_i of the control signal c_i. The assertion pattern, $A_1\ \theta\ A_2$ obtained by superposition θ of A_1 and A_2, satisfies the following conditions. For all i,

$$\begin{aligned}\pi_i(A_1\ \theta\ A_2) &= X,\ if\ \pi_i(A_1) = X \\ &= \pi_i(A_1),\ if\ \pi_i(A_1) \neq X\ and\ \pi_i(A_2) \neq X \\ &\quad and\ \pi_i(A_1) = \pi_i(A_2) \\ &= undefined,\ if\ \pi_i(A_1) \neq X\ and \\ &\quad \pi_i(A_2) \neq X\ and\ \pi_i(A_1) \neq \pi_i(A_2).\end{aligned}$$

Using the above definition and the function f_{mc}, it is possible to construct $\mathscr{M}_A$ from the assertion pattern A by the following definition of $\mathscr{M}_A$. $\mathscr{M}_A = \{\mu_i \mid f_{mc}(\mu_i)\ \theta\ A = f_{mc}(\mu_i)\}$.

Now, it is required to find the set of register transfer (RT) operations that are performed by the micro-operations in $\mathscr{M}_A$. Each RT operation that appears in the RTL behaviour is accomplished by a set of concurrent micro-operations. For example, an RT-operation $r_3 \Leftarrow r_1 + r_2$ may be accomplished by the concurrent micro-operations $bus1 \Leftarrow r_1$, $bus2 \Leftarrow r_2$, $fuLeftIn \Leftarrow bus1$, $fuRightIn \Leftarrow bus2$, $fuOut \Leftarrow fuLeftIn + fuRightIn$, $bus3 \Leftarrow fuOut$, $r_3 \Leftarrow bus3$. Finding an RT-operation from a given set of micro-operations is also not trivial because of two reasons. First, there may be more than one RT-operation in that particular state of the FSM. Secondly, there is a spatial sequence of concurrent micro-operations needed to accomplish an RT-operation but these are available in an unordered manner in $\mathscr{M}_A$.

The RT-operations accomplished by the set $\mathscr{M}_A$ of micro-operations are identified using a *rewriting method*. The method also reveals the spatial sequence of data flow needed for an RT-operation in a reverse order. The basic rewriting method consists in rewriting terms one after another in an expression. The micro-operations in which a register occurs in the left hand side (lhs) are found first. Such a micro-operation has the form $r \Leftarrow r\_in$, where r is a register and r_in is its input terminal. Next, the right hand side (rhs) expression "r_in" is rewritten by looking for a replacement (micro-operation) in $\mathscr{M}_A$ of the form "$r\_in \Leftarrow s$" or "$r\_in \Leftarrow s_1\ <op>\ s_2$". So, after rewriting "$r\_in$" we have rhs expression, either of the form "s" or of the form "$s_1\ <op>\ s_2$". In the next step, s (or s_1 *and* s_2 for the latter case) are rewritten *provided they are not registers*. The process terminates successfully when all s_i's in the expression in hand are registers.

The control structure of the FSMD M_3 can be obtained from the controller FSM and the RT-operations of each control state as constructed by the mechanism described above.

Verification of data-path and the controller:

Improper data-path interconnections and incorrect control signal assertion patterns can be found out during construction of FSMD M_3. They can be detected as follows. In course of the rewriting process two cases might arise. First, *more than one replacement are found for a non-register term s_i of an rhs expression*. It suggests that the control assertion pattern in this state is wrong. Secondly, *no replacement is found and some of the terms in the rhs expression are non-register data-path elements*. It can happen if either the data-path interconnections are not proper or control assertion pattern in this state is wrong.

Finally, the correctness of the RT-operations constructed from CP-DP informations and the controller FSM are ensured by the equivalence checking of FSMDs M_2 and M_3. The methodology described in subsection 3.2 is applicable to this phase without any modification.

5. EXPERIMENTAL RESULTS

A high-level synthesis tool, SAST [10] is developed. The SAST takes the behavioural description in VHDL and produces a synthesizable RTL code in Verilog. The proposed verification methodology is integrated with this HLS tool. This tool generates the input and the output FSMDs of each phase of verification, the state mapping function f_{sm} and the register binding function f_{rb} as byproducts with the synthesis results. The condition of execution and the data transformation of a path involves the whole integer arithmetic for which canonical form does not exist. Instead, we adapted a normal form [12] to get a uniform representation of the arithmetic expressions in SAST. Incorporation of the normalization technique for arithmetic expression makes our equivalence checker more powerful. The tool has been run on an Intel Pentium 4, 1.70 GHz, 256MB RAM machine on the outputs generated by SAST for several HLS benchmarks as shown in table 1. The number of sates in M_0 and in M_1, the CPU time and memory usage are tabulated for each HLS benchmarks. The number of states in M_2 and M_3 are the same as that of M_1. It may be noted that the CPU time and the memory usage for overall verification process are much lower than those of the overall synthesis time. The bars in figure 3 represent the number of variables in the input behaviour (first bar) and the number of registers in the data-path generated by SAST (sec-

Name	#state		CPU time in ms		memory used in kb	
	M_0	M_1	verifi-tion	synth-esis	verifi-tion	synt-hesis
DIFFEQ	9	12	6.372	54×10^3	32.3	4910
EWF	17	35	4.740	127×10^3	56.5	4436
GCD	7	4	9.340	21×10^3	23.5	6318
DCT	10	29	3.546	163×10^3	52.1	4631
TLC	7	8	10.660	101×10^3	31.5	7375
MODN	6	7	12.128	90×10^3	26.5	9654
PERFECT	9	6	11.128	53×10^3	23.5	9987

Table 1: Results for different high-level synthesis benchmarks

ond bar) for each HLS benchmarks. It is evident from this figure that SAST optimizes the number of registers and our verification works well in this case. The bars in figure 4 represent the number of micro-operations possible in the data-path (first bar) and the number of control signals involved in this micro-operations.

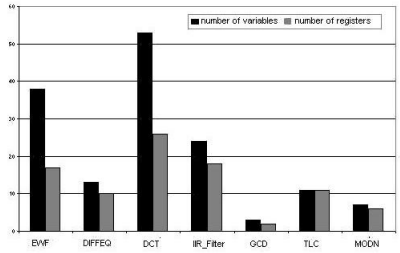

Figure 3: The number of variables and the number of registers required for different HLS benchmarks

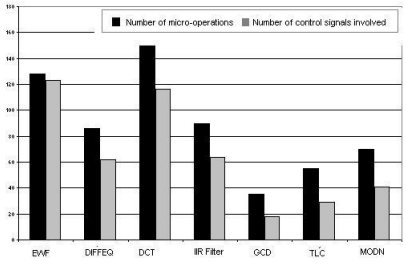

Figure 4: The number of micro-operations in the data-path and the number of control signals involved for different HLS benchmarks

6. CONCLUSION

Advances in VLSI technology have enabled its deployment into complex circuits. Synthesis flow of such circuits comprises various phases where each phase performs the task algorithmically providing for ingenious interventions of experts. The gap between the original behaviour and the finally synthesized circuits is too wide to be analyzed by any reasoning mechanism. The validation tasks, therefore, must be planned to go hand-in-hand with each phase of synthesis. The present work concerns itself with the hand-in-hand verification of high-level synthesis process. The validation task has been treated as an equivalence problem of FSMDs. In this work, we proposed an equivalence checking method which is applicable to the all three phases of HLS verification. This method is strong enough to handle several difficulties of each synthesis phase. The method has been implemented and also incorporated with an existing HLS tool. The experimental results suggests that the method is quite useful.

7. REFERENCES

[1] P. Ashar, S. Bhattacharya, A. Raghunathan, and A. Mukaiyama. Verification of rtl generated from scheduled behavior in a high-level synthesis flow. In *Proceedings of the IEEE/ACM ICCAD*, pages 517–524, 1998.

[2] R. Camposano. Path-based scheduling for synthesis. *IEEE transactions on computer-Aided Design of Integrated Circuits and Systems*, Vol 10 No 1:85–93, Jan. 1991.

[3] H. Eveking, H. Hinrichsen, and G. Ritter. Automatic verification of scheduling results in high-level synthesis. In *Proc. Conf. DATE*, pages 59–64, March 1999.

[4] R. W. Floyd. Assigning meaning to programs. In *Proceedings the 19^{th} Symposium on Applied Mathematics*, pages 19–32. American Mathematical Society, 1967. Mathematical Aspects of Computer Science.

[5] D. Gajski and L. Ramachandran. Introduction to high-level synthesis. *IEEE transactions on Design and Test of Computers*, pages 44–54, 1994.

[6] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Using global code motions to improve the quality of results for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 23(2):302–312, Feb 2004.

[7] C. A. R. Hoare. An axiomatic basis of computer programming. *Commun. ACM*, pages 576–580, 1969.

[8] C. Karfa, C. Mandal, D. Sarkar, S. Pentakota, and C. Reade. A formal verification method of scheduling in high-level synthesis. In *In Proc. ISQED '06*, pages 71–78, March 2006.

[9] Y. Kim, S. Kopuri, and N. Mansouri. Automated formal verification of scheduling process using finite state machine with datapath (FSMD). In *Proc. Conf.ISQED 2004)*, pages 110–115, March 2004.

[10] C. Mandal and R. M. Zimmer. A genetic algorithm for the synthesis of structured data paths. In *13th International Conference on VLSI Design*, pages 206–211, 2000.

[11] N. Mansouri and R. Vemuri. A methodology for automated verification of synthesized rtl designs and its integration with a high-level synthesis tool. In *In Proceedings of FMCAD*, pages 204–221, 1998.

[12] D. Sarkar and S. De Sarkar. Some inference rules for integer arithmetic for verification of flowchart programs on integers. *IEEE Trans Software. Engg.*, 15(1):1–9, 1989.