Register Sharing Verification During Data-path Synthesis

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

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

Abstract

The variables of the high-level specifications and the automatically generated temporary variables are mapped on to the data-path registers during data-path synthesis phase of high-level synthesis process. The registers in the data-path are usually shared by the variables and the mapping is not bijective as most of the high-level synthesis tools perform register optimization. In this paper, a formal methodology for verifying the correctness of register sharing is described. The input and the output of the data-path synthesis phase are represented as finite state machines with data-paths (FSMD). The method is based on checking equivalence of two FSMDs. Our technique is independent of the mechanism used for register optimization and works for both carrier and value based register optimization. The method also works for both data intensive and control intensive input specification. Our current implementation is integrated with an existing synthesis tool and has been tested for robustness.

1 Introduction

High-level synthesis (HLS) involves translating a behavioral specification into a register transfer level (RTL) structural description containing a data-path and a controller. High-level synthesis process consists of several inter-dependent phases, namely, preprocessing, scheduling, allocation and binding followed by controller design [1]. During allocation, minimum numbers of functional units and registers, required to synthesize the design based on the scheduling information of the operations, are computed and during binding, the variables are bound to the registers and the operators to the functional units (FU). The interconnections among the data-path elements through buses are decided next. Data-path synthesis comprises these three steps. In order to optimize the number of registers, several variables are made to share a register if there respective lifetimes do not overlap. The outputs generated and the final content of the registers may be wrong on two counts - the registers are not shared properly or the data-path is not set properly, that is, the controller generates signals such that wrong register value is put to the FU's input or the output of the FU updates a wrong register. The correctness of the final values of the variables depends on both these issues. In this work, we assume that the controller generates the signals correctly, that is, all the data transfers in the data-path as well as the operations selection for each FUs in each time step is proper and as intended by the behavioural specification. The objective of this work is to ensure that the registers are shared properly among the variables.

Several authors have proposed techniques for verification of synthesized designs. An approach for verification of register level design was proposed in [2], where verification can be integrated with synthesis systems which perform little or no register optimization. This verification technique has a limited use in practice as most of the practical HLS tools perform register optimizations for maximum utilization of the hardware. A compositional model for the functional verification of high-level synthesis is proposed in [3] where the specification and the implementation are encoded as FSMDs. The method in [4] checks the correctness of register transfer level (RTL) description with respect to the scheduled behaviour by model checking. A formal methodology for verification of various register allocation schemes was proposed in [5].

There are two types of register optimization schemes commonly found in high-level synthesis tools. They are *carrier based* [6] and *value based* [7]. Our proposed methodology can handle both the schemes. The input behaviours are either data-intensive or control-intensive in nature. Symbolic model checking [4] is suitable for formal verification of control-dominated applications. For the control intensive behaviours, the control flow is dependent on the arithmetic bit vector operations; an efficient representation of the transition behaviour under such situations is difficult to obtain due to the state space explosion problem [8].

The data intensive descriptions can be verified by means of symbolic simulation [9]. This method, however, allows only reasoning for a finite number of steps. More specifically, the loops in the description cannot be verified for an arbitrary number of iterations [8]. The cut-point based equivalence checking algorithm proposed in this paper has no such limitations.

The input to the data-path synthesis phase, i.e., the scheduler's output, and the output of this phase are represented as FSMDs M_1 and M_2, respectively. We have defined a function f_{rb} that maps the variables (registers) of M_2 to the variables of M_1 in each time step. The present work describes an algorithm for establishing equivalence between the FSMDs M_1 and M_2. For the given FSMDs M_1 and M_2 and the function f_{rb}, the equivalence of M_1 and M_2 indicates that the register sharing is correct. The underlying theorem is also formulated in this work.

This paper is organized as follows. In section 2, the FSMD model, the notion of computations on FSMDs and the formulation of the correctness problem are defined. The verification method is described in section 3. Several important issues that arise during data-path synthesis are analyzed in section 4. Some experimental results have been given in section 5. The paper is concluded in section 6.

2 Equivalence Problem Formulation

2.1 Finite State Machine with Data Path (FSMD)

An FSMD (*finite state machine with data-path*) is a universal specification model that can represent all hardware designs. The FSMD model was first proposed by Gajski in [1]. The model is used in the present work with the addition of a reset state, for encoding the specification and implementation of the circuit to be verified.

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 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 $I \cup V$ of input and storage variables and $S = \{R(a,b) | a,b \in E$ and R is any arithmetic relation$\}$ represents a set of status signals as arithmetic relations between two expressions from the set E.

The FSMD M_1 before allocation and binding and the FSMD M_2 after this phase for the *DIFFEQ* example, constructed from the result of our HLS tool SAST [10], are given in the table 1 and also depicted in figure 1 (a) and in 1 (b), respectively. The set of storage variables V_2 in M_2 consist of all the registers in the target data-path. The storage variables in V_1 of M_1 and the variables in V_2 of M_2 will be respectively designated as *variables* and *registers* in the subsequent sections.

FSMD M_1	FSMD M_2
M_1 : $\langle Q_1, q_{1,0}, I, V_1, O, f_1, h_1 \rangle$,	M_2 : $\langle Q_2, q_{2,0}, I, V_2, O, f_2, h_2 \rangle$,
where $Q_1 = \{q_{1,i}, 0 \leq i \leq 13\}$	where $Q_2 = \{q_{2,j}, 0 \leq j \leq 13\}$
$I = \{P_1,\ P_2,\ P_3\}$	$I = \{P_1,\ P_2,\ P_3\}$
$V_1 = \{dx,\ x,\ y,\ a,\ u, v0,\ v1,$	$V_2 = \{R_{00},\ R_{01},\ R_{02},\ R_{03},\ R_{04},$
$v2,\ v3,\ v4,\ v5,\ v6\}$	$R_{10},\ R_{11},\ R_{12},\ R_{20},\ R_{21}\}$
$O = \{P_1,\ P_2\}$	$O = \{P_1,\ P_2\}$
f_1 and h_1 are shown in fig 1	f_2 and h_2 are shown in fig 1

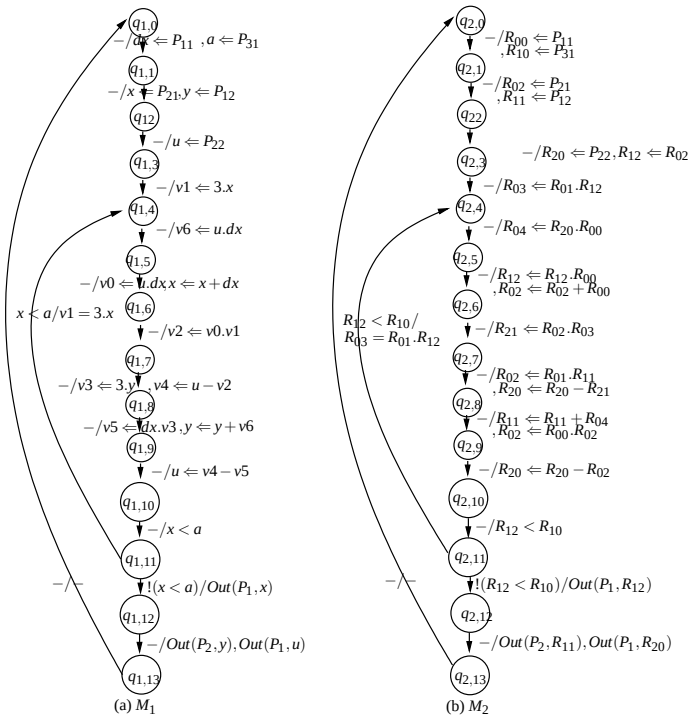
Table 1. The FSMDs M_1 and M_2

2.2 Paths in an FSMD

A *path* α from q_i to q_j, where $q_i, q_j \in Q$, is a finite sequence of states of the form $\langle q_i = q_1 \xrightarrow{s_1} q_2 \xrightarrow{s_2} \cdots \xrightarrow{s_{n-1}} q_n = q_j \rangle$ such that $\forall l, 1 \leq l \leq n-1, \exists s_l \in S$ such that $f(q_l, s_l) = q_{l+1}$, and q_k, $1 \leq k \leq n-1$, are all distinct. The state q_n may be identical to q_1. We aften denote such a path as $\langle q_i \Rightarrow q_j \rangle$, for brevity. The *condition of execution of the path* $\alpha = \langle q_{l_0} \xrightarrow{s_0} q_{l_1} \xrightarrow{s_1} q_{l_2} \cdots \xrightarrow{c_{s-1}} q_{l_k} \rangle$, R_α, is a logical expression over $V \cup I \cup Z$ such that R_α is satisfied by the (initial) data state at q_{l_0} iff the path α is traversed. We assume that inputs and outputs occur through named ports. The i^{th} input from port P_j is a value represented as P_{ji}. Thus if some variable v stores input from port P_j (for the i^{th} time along a path), it is equivalent to the assignment $v = P_{ji}$. The output of an expression e to a port P_j is represented as $OUT(P_j, e)$ and put as a member of a list preserved for each path. The *data transformation of a path* α, r_α, over V 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, the inputs in I and the set of integers Z and the output list $O_\alpha = [OUT(P_{i_1}, e_1), OUT(P_{i_2}, e_2), \ldots]$; the expression e_i in s_α represents the value of the variable v_i after the execution of the path in terms of the initial data state (i.e., the values of the variables at the initial control state); similarly, the expressions in the output list too are over the initial data state of the path.

2.3 Computations in an FSMD

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. Moreover, any computation c of an FSMD M can be looked upon as a computation along some concatenated path $[\alpha_1 \alpha_2 \alpha_3 \ldots \alpha_k]$ 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.



Register	Lifetimes of the variables
R_{00}	$\langle \perp, q_{2,0}, q_{2,0} \rangle, \langle dx, q_{2,1}, q_{2,13} \rangle$
R_{01}	$\langle \perp, q_{2,0}, q_{2,2} \rangle, \langle 3, q_{2,3}, q_{2,13} \rangle$
R_{02}	$\langle \perp, q_{2,0}, q_{2,1} \rangle, \langle x, q_{2,2}, q_{2,4} \rangle$ $\langle u, q_{2,5}, q_{2,5} \rangle, \langle v0, q_{2,6}, q_{2,7} \rangle$ $\langle v3, q_{2,8}, q_{2,8} \rangle, \langle v5, q_{2,9}, q_{2,13} \rangle$
R_{03}	$\langle \perp, q_{2,0}, q_{2,3} \rangle, \langle v1, q_{2,4}, q_{2,13} \rangle$
R_{04}	$\langle \perp, q_{2,0}, q_{2,4} \rangle, \langle v6, q_{2,5}, q_{2,13} \rangle$
R_{10}	$\langle \perp, q_{2,0}, q_{2,0} \rangle, \langle a, q_{2,1}, q_{2,13} \rangle$
R_{11}	$\langle \perp, q_{2,0}, q_{2,1} \rangle, \langle x, q_{2,2}, q_{2,13} \rangle$
R_{20}	$\langle \perp, q_{2,0}, q_{2,2} \rangle, \langle u, q_{2,3}, q_{2,7} \rangle$
R_{21}	$\langle v4, q_{2,8}, q_{2,9} \rangle, \langle u, q_{2,10}, q_{2,13} \rangle$ $\langle \perp, q_{2,0}, q_{2,6} \rangle, \langle v2, q_{2,7}, q_{2,13} \rangle$

(c) mapping from registers to variables

Figure 1. DIFFEQ Example: a. FSMD after scheduling, b. FSMD after allocation & binding and c. Mapping of the registers to the variables

2.4 Correctness Problem

Let us now consider how we can prove that the registers are shared among the variables properly. If we consider a computation c_2 in M_2, then the registers are updated through different operations and the updated data is used subsequently along the computation. Finally, the outputs are generated through ports. Let c_1 be a computation in M_1. If the outputs are same for both $c1$ and $c2$ and the value of each register of M_2 at the end of computation $c2$ is the same as the value of its corresponding variable in M_1 at the end of the computation $c1$, then the computations $c1$ and $c2$ are equivalent. If the registers are not shared properly among the variables, then in some stage of the computation some register(s) will be updated with wrong values. Consequently, the outputs as well as the final values of some registers of c_2 mismatch with the corresponding variables in $c1$. Thus, we need to speak about the following mapping functions to capture the correspondence between the register set of M_2 and the variable set of M_1.

2.4.1 The Mapping Functions

Definition 1 The state mapping function $f_{sm} : Q_1 \leftrightarrow Q_2$.

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. So, the FSMD structure in the output does not change in this phase. Hence, the function f_{sm} is a *bijection*.

Definition 2 Register binding function f_{rb}**:** $Q_2 \times V_2 \rightarrow V_1 \cup \{\perp\}$ *maps the registers at each time step in* M_2 *to variables in* M_1*, i.e., it defines the variable contained in a register at each state of* M_2.

If $f_{rb}(q_{2,i}, v_j) = v_k \in V_1$, then v_k is said to be the *corresponding variable* of the register v_j and v_j is said to be the *corresponding register* of the variable v_k at the state $q_{2,i}$. The two basic assumptions about the registers consider here are as follows.

1. The registers initially contain some garbage (undefined) value, denoted as $\perp$.
2. Once a value is stored in a register, it continues to hold it until the register has been updated by some other value.

The 'garbage' value is represented here as $\perp$. The function f_{rb} is total in the sense that any register contains either the value of a variable or the garbage value $\perp$ at each state. However, f_{rb} may not be a bijection as variables may have

non overlapping lifetimes and accordingly share the same register. Consequently, the number of registers in M_2 is less than or equal to the number of variables in M_1. This mapping function can be constructed from the lifetime information of the variables obtained from the allocation and binding information provided by any high-level synthesis tool. The mapping function f_{rb} produced by our SAST tool for the DIFFEQ example is shown in figure 1(c). The tuple $\langle v,\ start,\ end\rangle$ for a register R indicates that the value of the variable v is stored in register R from the state 'start' to the state 'end'. For example, the tuple $< x,\ q_{2,2},\ q_{2,4} >$ for the register R_{02} in figure 1(c) means variable x will be stored in R_{02} from the state $q_{2,2}$ to the state $q_{2,4}$.

So, our main goal is to verify whether M_1 behaves exactly as M_2. This means that for all possible input sequences, M_1 and M_2 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 3 *Two FSMDs M_1 and M_2 are said to be equivalent if for any computation c_1 of M_1, there exists a computation c_2 of M_2 such that c_1 and c_2 are computationally equivalent.*

But, an FSMD may contain an exponential number of computations. So, it is not feasible to enumarate all possible computations in one FSMD and find their equivalent computations in another FSMD. To overcome this problem , we define "finite path cover" as follows.

Definition 4 *A finite set of paths $P = \{p_0,\ p_1,\ p_2,\ ...,\ p_k\}$ is said to cover an FSMD M if any computation c of M can be looked upon as a concatenation of paths from P. P is said to be a "finite path cover" of the FSMD M.*

The above two definitions suggests the following theorem.

Theorem 1 *Two FSMDs M_1 and M_2 are equivalent if there exists a finite path cover $P_1 = \{p_{10}, p_{11}, \cdots, p_{1l}\}$ of M_1 and $P_2 = \{p_{20}, p_{21}, \cdots, p_{2l}\}$ of M_2 such that p_{1i} is equivalent to p_{2i} for all $i = 1$ to l.*

Theorem 1 clearly depicts how the equivalence problem of two FSMDs reduces to finding the equivalence of their paths. The notion of equivalence of paths is included as follows.

2.4.2 Equivalence of paths of two FSMDs

Given two FSMDs $M_1 = \langle Q_1,\ q_{1,0},\ I,\ V_1,\ O,\ f_1,\ h_1\rangle$ and $M_2 = \langle Q_2,\ q_{20},\ I, V_2,\ O,\ f_2,\ h_2\rangle$ and the mapping function f_{sm} and f_{rb} introduced in subsection 2.4.1, we first define the equivalence of an expression of M_1 with an expression of M_2. An expression e_1 (arithmetic or status) over $V_1 \cup I \cup Z$ at the state $q_{2,i}$ of M_1 is said to be equivalent to an expression e_2 (arithmetic or status) over $V_2 \cup I \cup Z$ 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 $v = f_{rb}(q_{2,j}, r)$. Using an expression e_2 loosely as a function, we denote the above phrase syntactically $e_1 = e_2 \circ f_{rb}$, where '$\circ$' stands for function composition.

Let, $\alpha_1 = \langle q_{1,l} \Rightarrow q_{1,m}\rangle$ and $\alpha_2 = \langle q_{2,r} \Rightarrow q_{2,s}\rangle$ be two paths in M_1 and M_2, respectively. Let there be n variables in the behavioural specification (M_1) and k registers in the data-path (M_2) for the given problem. Let the conditions be $R_{\alpha_1} = c_{11} \wedge c_{12} \wedge \ \ldots \ \wedge c_{1x}$ and $R_{\alpha_2} = c_{21} \wedge c_{22} \wedge \ \ldots \ \wedge c_{2x}$ and the data transformations be $r_{\alpha_1} = \langle s_{\alpha_1}, O_{\alpha_1}\rangle$ and $r_{\alpha_2} = \langle s_{\alpha_2}, O_{\alpha_2}\rangle$. In particular, note that the ordered tuple $s_{\alpha_1} = \langle e_{11},\ e_{12},\ \cdots,\ e_{1n}\rangle$, where each $e_{1i}, 1 \le i \le n$, is an expression over $I \cup V_1 \cup Z$ representing the value of the variable v_i after the execution of the path α_1 in M_1 in terms of the initial data state of the path. Similarly, $s_{\alpha_2} = \langle e_{21},\ e_{22},\ \cdots,\ e_{2k}\rangle$, where each $e_{2i}, 1 \le i \le k$, is an expression over $I \cup V_2 \cup Z$ representing the value of register r_i after the execution of the path α_2 in M_2 in terms of the initial data state of the path. The output list of the respective paths are O_{α_1} and O_{α_2}. The condition R_{α_1} is equivalent to R_{α_2} with respect to the mapping function f_{rb}, i.e. $R_{\alpha_1} = R_{\alpha_2} \circ f_{rb}$, if $c_{1i} = c_{2i} \circ f_{rb},\ \forall i,\ 1 \le i \le x$. Similarly, the data transformations r_{α_1} and r_{α_2} are equal with respect to f_{rb}, i.e., $r_{\alpha_1} = r_{\alpha_2} \circ f_{rb}$, if $\forall i, 1 \le i \le k, \exists j, 1 \le j \le n\ s.t\ v_j = f_{rb}(q_{2,s}\ r_i) \wedge e_{1j} = e_{2i} \circ f_{rb}$ and $O_{\alpha_1} = O_{\alpha_2} \circ f_{rb}$.

Definition 5 *Equivalence of two paths in two different FSMDs:*

Let, the FSMD M_1 be $\langle Q_1,\ q_{1,0},\ I,\ V_1,\ O,\ f_1,\ h_1\rangle$, the FSMD M_2 be $\langle Q_2,\ q_{20},\ I, V_2,\ O,\ f_2,\ h_2\rangle$ and the mapping function $f_{rb} : Q_2 \times V_2 \rightarrow V_1$. A path α_1 of M_1 is equivalent to a path α_2 of M_2 if $R_{\alpha_1} = R_{\alpha_2} \circ f_{rb}$ and $r_{\alpha_1} = r_{\alpha_2} \circ f_{rb}$, whenever 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.

Equivalence of path α_1 and α_2 is denoted as $\alpha_1 \simeq \alpha_2$.

3 Verification Method

Theorem 1 suggests that instead of finding all possible computations in one FSMD, it is better to find the finite path cover of that FSMD and try to find the equivalent path of each path of that set. Owing to the presence of loops it is difficult to find a path cover of an FSMD 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 [11, 12]. In any FSMD, we define cutpoints as follows

1. The reset state is a cutpoint.
2. A state q_i is a cutpoint if there is a divergence of flow from q_i.

Obviously, the 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. We can easily construct the finite path cover of $M_1(M_2)$ by the above rule.

We have already discussed in section 2 that the structure of both M_1 and M_2 would be the same and have equal number of states. As a result, the number of cutpoints would be equal in M_1 and M_2; also the path covers P_1 and P_2 of M_1 and M_2 respectively have the same number of paths. It is also possible to find the correspondence between the paths among the sets P_1 and P_2 using the state mapping function f_{sm}. The path $\langle q_{1,1} \xrightarrow[s_1]{} q_{1,2} \xrightarrow[s_2]{}, \cdots, \xrightarrow[s_{n-1}]{} q_{1,n}\rangle$ of M_1 and the path $\langle q_{2,1} \xrightarrow[s'_1]{} q_{2,2} \xrightarrow[s'_2]{}, \cdots, \xrightarrow[s'_{n-1}]{} q_{2,n}\rangle$ of M_2 are said to the corresponding to each other if $\forall i,\ 1 \leq i \leq n,\ q_{2,i} = f_{sm}(q_{1,i})$. So, it would suffice to show the equivalence between the corresponding paths of the sets P_1 and P_2. The verification algorithm is given next.

3.1 Verification Algorithm

Input: The FSMDs M_1, M_2 and the mapping functions f_{sm}, f_{rb}.
Output: The 'yes/no' answer for "M_1 is equivalent to M_2".
Method:
Insert cutpoints in M_1 and in M_2.
Find the path covers P_1 and P_2, where P_1 is the set of paths of M_1 and P_2 is the set of paths of M_2 and each path spans from a cutpoint to a cutpoint with no intermediary cutpoint;

$\forall \alpha \in P_1$
do
  match=0;
  $\beta = getPath(\alpha,\ P_2)$; /* This function returns a path β from P_2 which is the corresponding path of α */
  match = $checkEquivalent(\alpha,\ \beta,\ f_{rb})$;
  /* This function returns 1 if $\alpha \simeq \beta$; 0 otherwise */
  if(!match) Report:"M_1 and M_2 are not equivalent"; exit;
end do;
Report: "M_1 and M_2 are equivalent";

□

A stronger equivalence checker can be found in our paper [13]. We have restricted our equivalence checker in this work as the register sharing verification does not required such strong equivalence checking.

4 Analysis

In this section, we analyze some important issues like different register optimization schemes and the nature of the input specification that arises during data-path synthesis and show how our algorithm works for these cases.

4.1 Register Optimization Schemes

In the carrier based approach, two or more variables share a register if their respective lifetimes do not overlap. One variable always maps to only one register. Therefore, the mapping from the specification variables to the registers is a many-to-one relation. On the other hand, in the value based approach, two or more variables are assigned the same register if they use the same data value or the life span of at least one data value used by each variable is non-overlapping. It is obvious that a variable during its lifetime may assume different values. Also, it is possible that the same value is assigned to different variables. So, the association of specification variables and the registers is a many-to-many relation in this case. In both these cases, at any state, each register must contain the value of only one variable and this variable is called the corresponding variable of this register in that state. Also, one variable is mapped to only one register at each state. During equivalence checking of two corresponding paths, our algorithm compares the value of each register at the end state of that path with the value of corresponding variable. Hence, the algorithm is independent of the schemes used for register optimization.

4.2 Nature of the Input Specification

Our algorithm is also independent of the nature of input specifications. The control intensive behaviours are broken into path segments by putting cutpoints in all the branch states. These sets of path segments constitute the path cover of the FSMD. Each path segment is then checked for equivalence with the corresponding path segments in M_2. Equivalence of all the corresponding paths among two FSMDs implies that, for any computation of one FSMD, there is an equivalent computation in the other FSMD. It means that for all possible executions, the registers are shared properly. On the other hand, a data-intensive specification in general have only one path in each FSMD. Equivalence among these two paths proves the correctness of register sharing.

5 Experimental Results

The proposed algorithm has been implemented in 'C' and integrated with an existing high-level synthesis tool, SAST [10]. This tool generates the input and the output FSMDs of the data-path synthesis phase, the state mapping

function f_{sm} and the register binding function f_{rb} as byproducts with the synthesis results.

The algorithm 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 2. The number of registers in the data-path, the number of variables in the input behaviour, the number of states in each of the FSMDs M_1 and M_2, the number of paths in each FSMD and the CPU time are tabulated for each benchmark. It is evident from the table that the sharing of registers among the variables are quite high. For example, only 26 registers are required for 53 variables in the DCT example. It may be noted that EWF, IIR FILTER, DCT examples are data-intensive behaviours as there is only one path in each behaviour. On the other hand, DIFFEQ, GCD, MODN, TLC examples are control-intensive behaviour as number of paths are quite high compared to the numbers of state in the respective FSMDs. Our algorithm can successfully verify the register sharing in both the cases. The execution times for different benchmarks suggest that our algorithm is also quite efficient.

Name	*#var in* M_1	*#reg in* M_2	*#FSMD state*	*#FSMD paths*	*CPU time*
EWF	38	17	20	1	2.174 ms
IIRF	24	18	16	1	1.922 ms
DCT	53	26	24	1	2.048 ms
DIFFEQ	13	10	14	3	1.874 ms
GCD	3	2	5	4	1.955 ms
MODN	7	6	7	8	1.851 ms
TLC	11	11	7	12	2.512 ms

Table 2. Results for different high-level synthesis benchmarks

6 Conclusions

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 validation of the register sharing among the specification variables during datapath synthesis phase. Both the behaviours prior to and after the datapath synthesis have been modeled as FSMDs. The validation task has been treated as an equivalence problem of FSMDs. Our technique is independent of the mechanism used for register optimization and works for both carrier and value based register mapping. The technique is also independent of the nature of the input specification. The current implementation has been integrated with a synthesis tool and tested for several high-level synthesis benchmarks successfully.

References

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

[2] 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*, pp. 204–221, 1998.

[3] D. Borrione, J. Dushina, and L. Pierre, "A compositional model for the functional verification of high-level synthesis results," *IEEE Transactions on VLSI Systems*, vol. 8, pp. 526–530, October 2000.

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

[5] N. Mansouri and R. Vemuri, "Accounting for various register allocation schemes during post-synthesis verification of rtl designs," in *Proceedings of the DATE'99*, pp. 223–230, March 1999.

[6] N.-S. Woo, "A global, dynamic register allocation and binding for data path synthesis system," in *Procs. of 27th DAC*, pp. 505–510, 1990.

[7] F. Kurdhai and A. Parker, "Real: A program for register alocation," in *Procs. of 24th DAC*, pp. 210–215, 1987.

[8] C. Blank, "Formal verification of register binding," in *Procs. of Workshop on Advances in Verification (WAVE) 2000*, 2000.

[9] Y. Morihiro and T. Toneda, "Formal verification of data-path circuits based on symbolic simulation," in *Procs. of 9th ATS*, pp. 329–336, Dec 2000.

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

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

[12] C. A. R. Hoare, "An axiomatic basis of computer programming," *Communications ACM*, pp. 576–580, 1969.

[13] 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*, pp. 71–78, March 2006.