Practical Design Assistance at Register Transfer Level using a Data Path Verifier

NAKAMURA, Hiroshi
KUKIMOTO, Yuji**
FUJITA, Masahiro*
TANAKA, Hidehiko**

Institute of Information Sciences and Electronics, University of Tsukuba, Tsukuba, Ibaraki 305, Japan
*FUJITSU LABORATORIES Ltd. Kawasaki, Kanagawa 211, Japan
**Department of Electrical Engineering, University of Tokyo, Bunkyo-ku, Tokyo 113, Japan

Abstract

A practical design assistance system at the register transfer level is proposed in this paper. The unique characteristics of this system is that users are allowed to modify a register-level design manually. Our work attempts to utilize positively designers' experience. The final design is obtained by modifying the initial design manually. The consistency between the data path and its behavioral specification is verified automatically in the proposed system. We have implemented the verifier and applied it to an actual ASIC chip. This verifier has successfully verified it which consists of about 11,000 gates on CMOS gate array.

1 Introduction

Motivation Recently, much attention has been paid to the design assistance at the register transfer level. The objective is to derive efficient and error-free data paths. One of the answers for this problem is high-level synthesis[6]. In high-level synthesis, a data path is synthesized automatically from a given behavior. Though the derived data paths are error-free, they are not as satisfactory as those which are designed manually. The main reason for this is that designers' experience is not reflected in the derived data path.

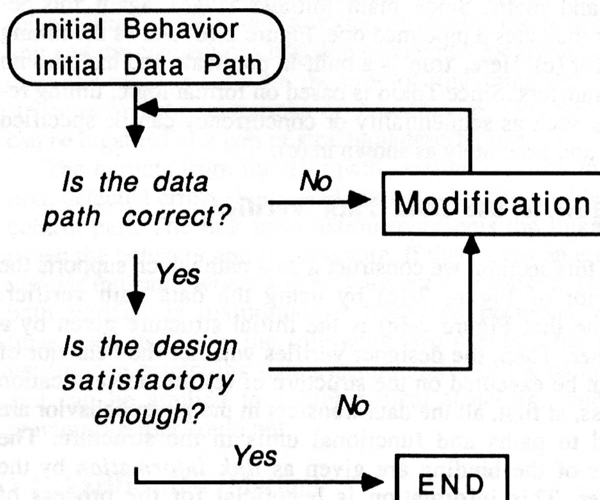


Figure 1. Flowchart of Assistance System

Our work attempts to utilize positively the designers' experience. In an actual design process, designers initially have images of data paths to be designed, because they have designed many similar circuits. It scarcely happens to develop a hardware which is drastically different from ones ever designed. This fact vindicates our approach. It is very important and much required to construct an effective design assistance system based on this approach. However, this approach has not been discussed well so far. Therefore, we propose a practical assistance system of register transfer level design in this paper.

Overview of Assistance System Figure 1 illustrates the design flow of the proposed system. The final design is obtained after the initial design has been modified several times.

At first, the designer gives an initial behavioral description and an initial structure of a data path to be designed. The initial structure is formed through the designers' intuition. Then the data path verifier verifies whether the structure supports the behavior or not. This verifier extracts the information of the linkage between the behavior and the structure. If there is no path (or functional unit) to realize a certain data-transfer, or if a certain path (or functional unit) is occupied by more than one data-transfer at a certain time slot, it results in design error. In such a case, the structure and/or the behavior are modified manually to compensate for that error. This modification is processed with referring to the linkage information. After the modification, they are verified again. If there is not any design error, the performance of the behavior is simulated and the cost of the structure is estimated. If the design is satisfactory, the design at register transfer level finishes and it proceeds to the lower level design such as logic synthesis. If the design does not satisfy the cost/performance constraints, the design flow is followed by the stage of modification and the process of verification. The design flow of Figure 1 is justified because the initial data paths given by designers are fairly good in many cases. If the initial data path is fairly good, the revised data path can be very efficient.

In this assistance system, we have adopted *Tokio*[2,5] as a behavioral description language. Tokio is based on Interval Temporal Logic[7] and can specify concurrency and sequentiality easily and accurately. Using Tokio, designers can directly describe concurrent behaviors of two finite state machines. They need not unfold the parallelism. This is the distinct point of Tokio compared with other behavioral description languages such as ISPS[1]. Owing to this nature, parallel behaviors such

CH2909-0/90/0000/0099$01.00 © 1990 IEEE

as pipelined ones can be specified easily and flexibly in Tokio. On the other hand, it becomes rather difficult for designers to guarantee the consistency between the behavior and the data path. Our strategy is to adopt a flexible behavioral description language and to verify the consistency between the behavior and the data path automatically. Using this approach, the load of the designers is lightened.

We have already implemented the data path verifier. This verifier is the core part in the proposed assistance system. Tools for estimating the performance and the cost remain to be implemented.

Our work is similar to RLEXT[3] in a sense that an interactive register-level design is assisted. In RLEXT, specifications cannot be directly constructed, understood, or modified by designers. RLEXT repairs data paths automatically under the constraint that the behavior is not altered. In our

```
main :- *a <= *a + *c && *b <= *b << 1.
```

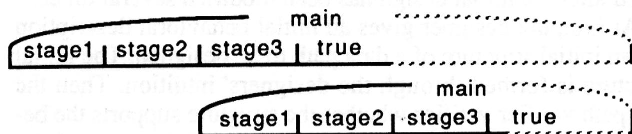
(a) Sequential Behavior in Tokio

```
main :- *a <= *a + *c , *b <= *b << 1.
```

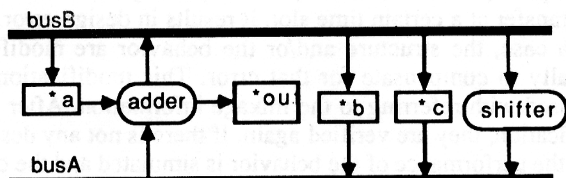
(b) Parallel Behavior in Tokio

```
main :- stage1 && stage2 && ((stage3 && true), main).
stage1 :- *a <= *a + *c.
stage2 :- *b <= *b << 1.
stage3 :- *out <= *a + *b.
```

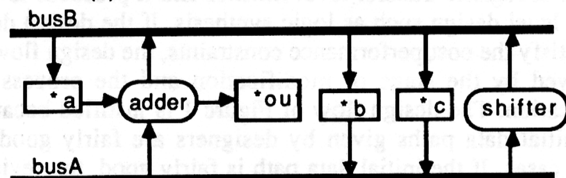
(c) Pipelined Behavior in Tokio



(d) Timing Chart for (c)



(e) Initial Data Path Structure



(f) Modified Data Path Structure

```
main :- stage1 && stage2 && ((stage3 && true), main).
stage1 :- *b <= *b << 1.
stage2 :- *a <= *a + *c.
stage3 :- *out <= *a + *b.
```

(g) Modified Behavior

Figure 2. Example of Register-Level Design

work, behaviors as well as structures can be understood by designers. Our approach is to derive a efficient register-level design by modifying both the structures and the behaviors. This point distinguishes our work from RLEXT.

Content The remainder of this paper is organized as follows. First, simple example of register level design is illustrated in Section 2. In this section, behavioral description language Tokio is explained briefly at first. Then, how to use the data path verifier in register level design is illustrated by using a simple example. In Section 3, the data path verifier is presented. In Section 3.1, the overview of the verifier is described. Then, the verification method is presented in Section 3.2. Experimental results and evaluation of the verifier is given in Section 4. The adopted example is an actual ASIC chip. This paper is concluded in Section 5.

2 Simple Example of Register Level Design using Data Path Verifier

2.1 Behavioral Description Language Tokio

We explain the behavioral description language Tokio by illustrating simple examples. Details are mentioned in [2,5].

Figure 2-(a) shows a sequential behavior in Tokio. The statement of '*a <= *a + *c' represents that the data of register '*a' is added to the data of register '*c' and then the result is stored in register '*a'. In the statement of '*b <= *b << 1', '<< 1' represents the operation of one-bit left-shift. In Tokio, all the behaviors are defined on *intervals* (in other words, time period). For example, '*a <= *a + *c' is defined in an interval and '*b <= *b << 1' is defined in another interval. The *chop* operator '&&' represents that the intervals are executed in sequential order. Therefore, '*b <= *b << 1' is executed after '*a <= *a + *c' has been executed. Figure 2-(b) represents parallel execution in Tokio. The operator of *comma* ',' represents that the intervals are the same one. In this example, the operator ',' represents that the interval of '*a <= *a + *c' is the same as the interval of '*b <= *b << 1'. Therefore, the two data-transfers are executed simultaneously.

The next example is Figure 2-(c). The first line in Figure 2-(c) represents that 'stage2' is followed by both '(stage3 && true)' and 'main'. Since 'main' initiates 'stage1' again, this behavior indicates a pipelined one. Figure 2-(d) shows the timing chart for (c). Here, 'true' is a built-in predicate and includes no data-transfers. Since Tokio is based on formal logic, timing relations such as sequentiality or concurrency can be specified easily and accurately as shown in (c).

2.2 How to use Data Path Verifier

In this section, we construct a data path which supports the behavior of Figure 2-(c) by using the data path verifier. Assume that Figure 2-(e) is the initial structure given by a designer. Then, the designer verifies whether the behavior of (c) can be executed on the structure of (e). In the verification process, at first, all the data-transfers in the given behavior are bound to paths and functional units in the structure. The results of the binding are given as *link information* by the verifier. This information is beneficial for the process of

modification shown in Figure 1. In this example, the data-transfers of '*a <= *a + *c' and '*out <= *a + *b' are successfully bound to the structure, but '*b <= *b << 1' is not bound to. (We call this error as 'error1') After the process of the binding, the verifier searches for data-transfers which are executed in parallel. In this example, the verifier detects that '*a <= *a + *c' and '*out <= *a + *b' are executed simultaneously. Since both the two data-transfers require the 'adder' and 'busA', the verifier gives the message that 'adder' and 'busA' are used simultaneously by the two data-transfers of '*a <= *a + *c' and '*out <= *a + *b'. (We call this error as 'error2')

Next, the initial structure and the behavior are modified. 'Error1' is simple. This error is repaired by replacing paths connecting to 'shifter' as shown in Figure 2-(f). As for 'error2', there are some candidates for the way of correction. One solution is to supply the structure with an adder and paths. Another solution is to re-schedule the data-transfers in the behavior. The link information which is given by the verifier shows that the data-transfer of '*b <= *b << 1' does not use 'adder'. Therefore, we can avoid 'error2' by re-schedule the behavior as shown in Figure 2-(g). This modification is recommended because it does not increase the cost of the structure at all. Then the structure of (f) and the behavior of (g) are verified again, and no errors are detected.

What we want to emphasize through this example are the following two points. The first point is that parallel executions can be specified without unfolding the parallelism in Tokio. This nature is beneficial especially in deriving a pipelined behavioral description from a sequential behavioral description. In order to utilize this nature, the verifier should extract non-trivial concurrent data-transfers from the given behaviors. The second point is that modifying the behavior is sometimes better than modifying the structure. This is the reason why designers are allowed to modify the behaviors in our system.

3 Data Path Verifier

3.1 Overview

The inputs to the data path verifier are behavioral descriptions in Tokio, structural descriptions in Prolog, and operation rules in Prolog. Structural descriptions denote the type of each unit and the connections between each unit. Operation rules declare the relation between the type of functional units in the structure and the operations in the behavior. Operation rules can be regarded as a part of structural descriptions.

The outputs from the data path verifier are: link information, detected errors if any, and the state transition table of the control part. The link information represents the linkage between the behavior and the structure. If there exist an error, the verifier indicates where and why the error is caused. This data path verifier is distinguished from the previous version (discussed in [8]) in the following two points. That is, this verifier can extract the state transition table of the control part and can be applied to a hierarchical structure, where the previous verifier could not.

3.2 Verification Method

A. Derivation of Link Information

At first, the data-transfers in the behavior are bound to the structure and link information between the behavior and the structure are derived. The derivation is processed in accordance with the following steps.

a. Find the functional unit or a set of functional units which realizes the operation of each data-transfer.
b. Search for data paths from the source register to the input of the functional unit and those from the output of the functional unit to the destination register.

Operations in the behavior are linked to the functional units by using the operation rules. This verifier can be applied to a structure which contains modules of specific functions if their functions are declared in the operation rules. Consistency considering the bit-width is also verified provided that the bit-width of each operation is declared in the operation rules. Names of the registers and memories in the behavior are assumed to be the same as those in the structure.

If a certain data-transfer cannot be bound to the structure, it results in design error. If a certain path or functional unit is used in different data-transfers and if the data-transfers occur simultaneously, it also results in design error. In order to detect the latter kind of errors, concurrency in Tokio must be unfolded. This unfolding are processed in the next stage.

B Derivation of State Transition Table:

In this stage, concurrency is unfolded and all the intervals occurring simultaneously are listed at first. The state transition table is also derived in this process. We call this process as *traversing transitions*. After that, all the listed intervals are checked whether they contain data-transfers occupying the same path or the same functional unit. This process is called as *conflict detection.*

(1) Traversing Transitions
Here, the intervals which occur simultaneously are searched for by traversing all the transitions in depth-first. In this process, the state transition table of the control part is also extracted. A state in Tokio description is defined by {*intervalName, clockNumber*}. The computations are as follows.

(step1) First, the initial interval I_{init} and initial clock I_{clock} (usually 0) is selected. $S_0 = \{[I_{init}, I_{clock}]\}$

(step2) If S_i has predicate calls, all the called intervals are added to S_i and S'_i is obtained. S'_i is a list of intervals which occur simultaneously at the *i*-th clock. The obtained S'_i are recorded with the transitive conditions. The ancestor of the predicate call is also recorded. This is because intervals called by different ancestors have different successors. However, if the predicate call occurs in the last interval of the predicate, the ancestors are not required to be recorded, because the ancestors have no successors.

(step3) To proceed one clock, and S_{i+1} next to S'_i are obtained. The clock number I_{clock} is increased if the obtained

clock number is not larger than the length of that interval. Otherwise, the traversing transfers to the next interval through *chop* operator &&. *(step3)* is followed by *(step2)*.

Halting Condition:

Let S'_n be the newly obtained state in *(step2)*. If S'_n is included by a certain S'_i where $0 \leq i < n$, or if S_{n+1} next to S'_n cannot be obtained, then the execution will halt. The execution always halts.

Proof.(abstract)

The number of states is finite unless ancestors of predicate calls are recorded infinitely in *(step2)*. Ancestors are not recorded if the predicate calls exist in the last interval. Since recursive calls are allowed to exist only in the last interval in Tokio, infinite ancestors are not created in *(step2)*. Therefore, the number of states is finite and the execution of traversing transitions always halts.

(2) Conflict Detection

This stage is divided into the following two steps.

(step1) Listing all the simultaneous intervals: All the elements in each S'_i are executed simultaneously unless they have exclusive transitive conditions.

(step2) Detecting data path conflict: Using the *link information*, all the listed intervals are checked whether they use the same paths or functional units. If they use the same one, it results in data path conflict.

4 Experimental Results and Evaluation

Experimental Results We have applied the data path verifier to the design of an actual ASIC chip. The adopted example is a network interface processor (NIP) in PIE64[4]. PIE64 is a parallel Prolog-based machine which is under development in our laboratory. This machine consists of 64 inference units and two high-speed interconnection networks. NIP has been already designed and is going to be implemented on CMOS gate arrays. The total number of the gates is about 18,000 including both the data path and the control part. The structure of the NIP is divided into four parts. Each part has its own controller and acts in cooperation. The data path verifier has been applied to the main two parts. In these parts, the data transfers and the process synchronization between inference units are processed respectively. These parts consist of about 11,000 gates in total, and are operated in 6-stage pipelining. The verification results of the two parts are shown in Table 1. The data path verifier is implemented using SICStus-Prolog on SUN4/260 (about 80KLIPS).

Evaluation In data path verification, most of the CPU-time is spent in the derivation of state transition table. Since all the state transitions are traversed here, its cost is proportional to the number of the transitions. The overhead for checking the halting condition increases proportionally to the number of the states presently. This is because traced states are recorded with enumeration. Thus the computational complexity for deriving state transition table is $O(n * m)$, where n is the number of state transitions and m is the number of states. If traced states are recorded in a binary tree, for example, the complexity will become $O(n * log(m))$.

	CPU Time (sec)		Number of Derived States	Number of State Transition
	Derivation of Link Information	Derivation of State Transition Table		
Data Transfer Part	7.89	183.3	81	379
Process Synchronization Part	13.82	1488	236	1168

Table 1. Results of Verifying NIP

5 Conclusion

This paper has described a practical design assistance system at the register transfer level. The unique characteristics of this system is that users are allowed to modify a register-level design manually. The consistency between the data path and its behavioral specification is verified automatically in the proposed system. We have illustrated a simple example of register level design in this approach.

In the design flow of Figure 1, data path verification is repeated several times. The experimental results of the data path verifier shows that the cost of the verification is small enough and that the practical hardware design shown in Figure 1 can be assisted using this data path verifier.

Acknowledgement

The first author would like to appreciate Prof. Nakazawa of University of Tsukuba for encouragement.

This research was supported by the Ministry of Education, Science, and Cultures of the Japanese Government under the Grant-in-Aid for Encouragement of Young Scientists (No. 01790381).

References

[1] M.R.Barbacci. Instruction Set Processor Specification (ISPS): The Notation and its Application. *IEEE Transactions on Computers*, C-30, No.1:24-40,1981

[2] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Aid to Hierarchical and Structured Logic Design using Temporal Logic and Prolog. In *Proceedings.Pt.E*, IEE, 1986

[3] D. W. Knapp. An Interactive Tool for Register-Level Structure Optimization. In *26th Design Automation Conference*, ACM/IEEE 1989

[4] H. Koike and H. Tanaka. Multi-Context Processing and Data Balancing Mechanism of the Parallel Inference Machine PIE64. In *Fifth Generation Computer Systems*, ICOT, 1988

[5] S. Kono, T. Aoyagi, M. Fujita, and H. Tanaka. Implementation of Temporal Logic Programming Language Tokio. In *Logic Programming Conference '85*, LNCS-221, Springer-Verlag, 1985

[6] M.C. McFarland, A.C.Parker, and R. Camposano. Tutorial on High-Level Synthesis. In *25th Design Automation Conference*, ACM/IEEE,1988

[7] B. Moszkowski. A Temporal Logic for Multi-Level reasoning about Hardware. In *CHDL'83*, IFIP, 1983

[8] H. Nakamura, M. Fujita, S. Kono, M. Nakai, and H. Tanaka. A Data Path Verification System Using Temporal Logic Based Language Tokio. In *Working Conference on CAD Systems Using AI Techniques*, IFIP, 1989