Automatic Procedures for the Behavioral Verification of Digital Designs

by

Filip Van Aelten

Burg. ir., Katholieke Universiteit Leuven (1987)

S.M., Massachusetts Institute of Technology (1989)

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

May, 1992

Signature of Author _____
Department of Electrical Engineering and Computer Science
May 1, 1992

Certified by _____
Srinivas Devadas
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by _____
Campbell L. Searle
Chairman, Department Committee on Graduate Students

Automatic Procedures for the Behavioral Verification of Digital Designs

by

Filip Van Aelten

Submitted to the
Department of Electrical Engineering and Computer Science
on May 1, 1992 in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Abstract

This thesis introduces automatic procedures for the behavioral verification of digital designs. The procedures differ from theorem provers in that they are fully automatic. They differ from most current automatic verification procedures in two respects. First, they leave room for modifications to the input/output behavior of the implementation. Second, one set of procedures is compositional in nature allowing for the verification of larger designs than can be verified with most current automatic procedures.

Both the implementation and the specification are taken to be finite state machines (FSM's), represented at the logic level, with an associated string function describing the input/output behavior. Correctness is taken to mean that one of a collection of relations holds between the string functions associated with the implementation and the specification. Fully general verification procedures for these relations are constructed from FSM equivalence checking algorithms, based on theorems which state necessary and sufficient conditions for correctness in terms of FSM equivalence. These procedures are applicable to small to medium-size circuits only. For large systems, compositional event-based procedures are developed which verify microcoded or array processors against signal flow graphs or dependency graphs. It is verified that the implementation has a satisfactory input/output behavior under the assumption that it executes the same set of operations as the specification. The compositional method is event-based and consists of verifying the combinational correctness of the data path modules in the implementation, and the validity of a collection of Computation Tree Logic (CTL) formulae on the controller. These constitute sufficient conditions for the validity of the β-relation between the implementation and the specification. Experimental results are presented.

Thesis Supervisor: Srinivas Devadas

Title: Assistant Professor of Electrical Engineering and Computer Science

Acknowledgements

In the early stages of this research I benefited greatly from the work of Srinivas Devadas, my supervisor, and Kurt Keutzer, who introduced me to the behavioral equivalence problem and provided fast access to the latest results on logic-level verification. Throughout the work Srinivas provided excellent support by setting up a software platform for verification, helping out with problems at all levels of detail, and assisting in the communication of results to the CAD research community.

In later stages of the research I benefited from the work at IMEC on the synthesis of digital signal processors, which provided a theoretical and physical framework for the verification of these processors. Special thanks are due to Stefan De Troch and Ivo Bolsens for helping with the experiments.

I thank the readers, Jonathan Allen and Charles Leiserson, for suggesting improvements to the thesis document. I also thank Kelly Bai and Stan Liao for their comments and participation in the research.

I thank the members of the 8^{th}-floor group, the members of the European Club, and my room-mates, past and current, for the companionship. Special thanks go to office-mate Khalid Rahmat.

I thank my parents for their constant support and dedicate this thesis to my grandparents.

The work described in this report was done at the Research Laboratory of Electronics of the Massachusetts Institute of Technology. It was supported in part by the Defense Advanced Research Projects Agency under contracts N00014-87-K-0825 and N00014-91-J-1698, in part by a grant from Analog Devices, Inc., and in part by an IBM fellowship.

Contents

Chapter 1

Introduction

1.1 Summary of Contributions

This thesis presents automatic procedures for verifying behavioral equivalence between an implementation and a specification, which are both synchronous logic circuits. An implementation is verified to have a satisfactory input/output behavior without having to realize one particular input/output behavior. Circuits with different degrees of parallelism or different degrees of pipelining have different input/output behaviors, but may correctly implement the same behavioral specification.

Figure 1.1 shows an example of an implementation and a specification which can be declared behaviorally equivalent, although they have different input/output behaviors. The specification, shown at the bottom of the figure, is purely combinational. It takes an input value at every clock cycle, and produces the corresponding output at the same clock cycle. The implementation, for which only the data path is shown, performs the same operations as the specification, but one by one. A new input value is taken every 6^{th} clock cycle, and the corresponding output value is produced 5 cycles later.

Behavioral equivalence is modeled by relations between the string functions corresponding to the specification and the implementation. String functions model the mapping from input streams to output stream realized by a synchronous machine. String function relations were first defined in [9]. The relations defined there are modified and significantly

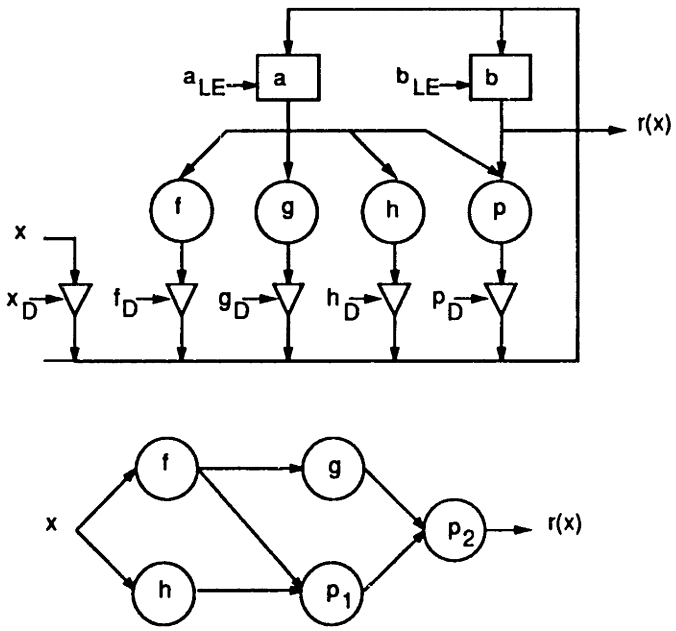


Figure 1.1: Implementation and specification which can be declared behaviorally equivalent

extended in this thesis. Five primitive relations are defined: the "don't care times" relation (β), the parallelism relation (γ) , the encoding relation (δ), the "input don't care" relation (ϵ) and the "output don't care" relation (ζ). These relations have attributes. For instance, the parallelism relation has an attribute corresponding to the degree of parallelism. The implementation and specification of Figure 1.1 are in "don't care times" or β-relation: the implementation is equivalent to the specification if only the input values and output values at relevant time points (i.e. not at "don't care times") are considered.

This thesis contains two results. The first result is a sound and complete proof method for proving arbitrary compositions of the five primitive relations under certain assumptions. The proof method consists of circuit transformations on the specification and the implementation, followed by *one* equivalence check (on the transformed circuits). All of these operations are automatic. It is demonstrated that, if the proof method declares the implementation and specification to be in a certain relation, they are so (soundness), and that, if a certain relations holds, it can be proven with the proof method (completeness).

The sound and complete proof method cannot be directly used for the behavioral verifi-

cation of realistic industrial designs. Only circuits of limited size can be verified, containing two or three arithmetic or logic units and up to about 50 latches.

The second result is an event-based compositional method for verifying large synchronous processors against signal flow graphs (containing state variables) or dependency graphs (containing no state variables). The verification task is reduced to the combinational verification of arithmetic or logic data path modules, and the verification of a collection of conditions on events initiated by the controller. Compositionality relies on the assumption that the implementation performs the same set of operations as the specification. The method constitutes a sound but not a complete proof method for proving the β-relation. Large designs can be verified with it. The worst-case complexity of the program is linear in the number of states of the controller, and linear in the number of register transfers initiated by the controller.

The assumption that the same set of operations is performed is consistent with the following design scenario. A signal flow graph or dependency graph is taken as a specification. Such a specification may be represented in textual form, for instance as a program in the SILAGE language [30]. At some point in the design process data path hardware is allocated for the specified computation. The original signal flow graph can then be expanded so that every operation in the new signal flow graph corresponds to a register transfer in the implementation. The operations in the new signal flow graph have to be scheduled on the allocated data path hardware, and control logic has to be synthesized to implement the schedule. The compositional verification method can be used to verify across the last two steps: scheduling and control synthesis. It allows for the verification of the β-relation between an implementation and an expanded signal flow graph. We have shown that the β-relation has a transitivity property so that, if the expanded SFG is also proven to be in β-relation with the original SFG, the implementation is in β-relation with the original specification.

Two realistic industrial designs were verified with the event-based compositional method. Both of them are microcoded processors. Sufficient correctness conditions on the controller were verified. The data path modules were assumed to be correct -- they can be verified

with combinational verification techniques (e.g. [11], [5], [32], [14], [44]), with the exception of register files and RAM's. The first design is a recursive filter consisting of three cascaded second-order stages followed by a line equalizer. The data path consists of 3 14-bit ALU's and 30 14-bit registers. The controller contains 14 latches and 58 output lines, and initiates 70 register transfers for each new input sample. Verification of this design took roughly 2 minutes on a SUN SPARC-Station 2 and revealed several errors, which could be traced back to bugs in the synthesis system.

The second circuit is an echo cancellor, the specification of which contains if-then-elses and iterative loops. The data path consists of one ALU, one multiplier, two ACU's (all with word lengths of 14 bits), two RAM's (with 128 12-bit words and 256 14-bit words respectively) and 39 registers. The controller contains 20 latches and 95 output lines, and initiates 144 register transfers for each new input sample. Verification took 37 minutes and revealed again several errors. The types of errors found were corruption of values in registers between production and consumption, incorrect jumps, violated precedence constraints, and insufficient delay between decision making operations and register transfers generating status bits for the decision making – the minimal delay corresponds with the number of pipeline stages in loops around the controller and the data path.

The event-based compositional method was also applied to the verification of systolic array designs against dependency graphs.

1.2 Context

Computer-Aided Design is aimed at alleviating two major problems in the design process: the fact that it is long and tedious, on the one hand, and error prone, on the other. The first problem is tackled with synthesis programs that automate certain design steps. The second problem is addressed by verification programs that allow a designer to verify the consistency between an initial specification and a derived implementation.

It has been argued that the emergence of synthesis programs eliminates the need for verification procedures. For a given specification, the implementation would be correct "by

construction". However, for this argument to hold three conditions have to be met. First, the synthesis program should have been proven correct. In principle, the system software on top of which the synthesis program is written, should be proven correct too. Second, the synthesis procedures should never be modified, unless they are verified again. Third, no manual modifications to the design can be allowed. Usually, none of these conditions are met. Program verification for complex synthesis programs in use today is beyond the current state of the art. Furthermore, because synthesis programs address very complex optimization problems, where a vast space of possible designs is to be explored, they tend to be frequently modified. For the same reason, manual enhancements to the design are often applied. All of this points to the need for *independent* verification procedures, and this need is recognized in industry.

While much progress has been made in the area of synthesis, verification is lagging behind. The most common verification method in use today is still simulation, where a limited set of input vectors are applied to the implementation and the specification, and the outputs compared. Symbolic simulation, where symbolic inputs are applied to cover a wider range of the input and state space, is practiced increasingly as well. Still more progress needs to be made in verification. It is not uncommon that a circuit which takes 2 weeks to design, takes another 8 weeks to debug and verify [36].

The ultimate task in verification is to demonstrate that a designed circuit has a correct behavior. Such a circuit can be very large, which makes it imperative that the verification procedure be efficient. The verification task can be split in subtasks. For instance, a first subtask may consist of demonstrating that a layout has a certain Boolean functionality (which can again be divided into extracting the transistor schematics from the layout, and proving that the transistor schematic has a correct Boolean functionality). The remaining task is then to demonstrate that a logic design produces an intended overall behavior.

The intended behavior for a (synchronous) design is not necessarily a specific input/output mapping. Circuits with different degrees of pipelining, or circuits with different degrees of parallelism, produce different input/output functions but may all exhibit satisfactory input/output behavior. By behavioral verification we mean the problem of verifying that a

(usually large) circuit design exhibits a satisfactory input/output behavior. What constitutes "satisfactory input/output behavior" needs to be precisely circumscribed in a given domain of application. The circuit design may be given at any of a number of representational levels, e.g. layout, circuit level, and logic level. If, for instance, a logic level implementation is taken, it remains to be shown that the layout produces a Boolean input/output mapping consistent with the logic-level implementation.

This thesis considers the independent verification of synchronous logic-level designs against a behavioral specification. The "correct by construction" approach is treated in [37] and [16] (the latter paper deals with verification in the context of manual design). The verification of asynchronous designs is treated in [25], [39] and [26]. The verification of layout with respect to circuit schematics is treated in [3], [2] and [48]. The verification of circuit schematics with respect to a logic circuit is treated in [12], [6], [4], [47], [1], [42] and [29]. The thesis is concerned with aspects of functionality, not with speed. The latter is important to determine if a circuit works correctly under a given clock frequency. This problem is called timing verification, and is treated in [38] and [23].

1.3 Previous Work

Henceforth the problem of functionality verification for synchronous logic-level circuits is considered. Extensive work has been done on the verification of specific input/output mappings. Although this does not directly address the *behavioral* verification problem, it does so indirectly by providing procedures that can be used as building blocks in a behavioral verification system.

Two classes of procedures have been developed for verifying that a circuit exhibits a specific input/output mapping: fully automatic procedures, and procedures that require user interventions. The second class is commonly denoted with the term "theorem provers", which is understood to refer to the method of proving theorems, rather than the mere fact that they prove theorems. Theorem provers are built from a general purpose backbone mechanism, which manipulates symbolic expressions by applying inference rules, searching

for a way to derive a desired conclusion from a given premise. In contrast, automatic input/output verification procedures are built from representations and routines that are specifically tailored to the problem at hand. Many of the achievements in theorem proving techniques are being obliterated by recent advances in automatic procedures. However, theorem provers do have strengths that can be used to advantage for verifying large circuits, namely the allowance for functional abstraction and proof by induction. The latter can be used, for instance, to prove that data path modules with arbitrary bit size have correct functionality (e.g. [44]).

Automatic procedures for verifying functional correctness of combinational logic circuits, have been largely based on two representations for Boolean functions: the sum-of-products representation, and the binary decision diagram (BDD) representation. Checking if two logic functions are equivalent is a co NP-complete problem. Procedures based on the sum-of-products representation [7] were introduced first, and are practical to the extent in which a logic function can efficiently be expressed as a sum of products. This is the case for logic functions implemented with two-level logic. Ordered binary decision diagrams (introduced in [11], although variants of this representation were used in industry before) form an efficient representation for a much broader class of circuits. They are at present the most robust Boolean function representation for verification purposes. Although BDDs are an efficient representation in practice, they take, in the worst case, exponential space over the number of inputs.

One way of determining input/output equivalence between two finite state machines (FSM's), for completely specified machines, is based on the canonicity of the reduced state transition graph (STG). A more general method applicable to incompletely specified machines is to construct and inspect the product of the two machines (e.g. [24], [20]). The product machine consists of the two original machines, XNOR-gates taking corresponding outputs of the original machines, and an AND-gate taking all the outputs of the XNOR-gates. This machine produces an output of 1 so long as the two original machines agree on a given input. To verify input/output equivalence, this machine has to be fully traversed. The machines are equivalent if there never appears an output of 0. During traversal, both

inputs and states can be implicitly enumerated, using BDDs. This means that inputs and states are not explored one by one, but in sets, where a set of inputs or states is represented with a BDD. With this technique the STG of the product machine can be traversed in one pass, with no need for backtracking. The worst case complexity of this procedure, however, is still linear in the number of states (which is possibly exponential in the number of flip-flops in the product machine). The procedure will be further elaborated upon in Chapter 3.

Early work on behavioral equivalence has involved the use of theorem provers to prove the correctness of a single microprocessor (e.g. [31], [19], [33]). To the best of my knowledge this work has not delivered any systematic procedure for verifying a collection of processors with respect to a general correctness requirement. Single microprocessors were verified, which, although relatively simple, required elaborate user effort.

Similarly, the work on protocol verification described in [40] and [41] does not seem to involve a general correctness requirement together with a systematic procedure to verify it. Correctness requirements are formulated manually for very specific tasks.

One attempt at formalizing behavioral equivalence and verifying it in a systematic way, is based on p-automata (for input-programmed automata) [22]. The specification is a non-deterministic finite automaton with extra input variables (meta-inputs). An implementation is said to satisfy the specification if it is input/output equivalent to it under some assignment of the meta-inputs. Non-determinism is used to compactly represent a large number of possible input/output sequences: ϵ-moves (which may, but do not have to be taken) allow for input/output sequences with different degrees of parallelism. For verification of an implementation against a specification, the product machine is constructed, and traversal is started with the universal set of values for the meta-inputs. During traversal, parts of the non-deterministic p-automaton are dynamically converted into deterministic counterparts. Also, the set of values for the meta-inputs are split if the output is undefined for the entire set. Sets of values that produce an output of 0 are discarded. The procedure ends when all meta- and regular inputs, and all states, have been exhaustively enumerated.

The p-automata approach is very general, but very computationally intensive also. For

instance, to verify across a behavioral synthesis process consisting of data path hardware allocation, scheduling and controller synthesis, the implementation is verified against a specification consisting of maximally parallel data path hardware and a controller with meta-inputs modeling different possible schedules. Verification is very computationally intensive because all the registers in the large data path contribute to the state space to be explored, and all possible ways of scheduling are captured in the controller of the specification. To alleviate the first problem, work is needed on reducing a global correctness requirement for the overall circuits to requirements on the controller only, so that only the controller has to be traversed. The second problem is more severe.

Another way of formalizing behavioral equivalence is through string function relations, introduced by Bronstein [9] [10]. Both the implementation and the specification are taken to be synchronous machines which have unique string functions associated with them. These string functions map sequences (strings) of input values into sequences of output values. Behavioral equivalence is modeled as a relation between two string functions. Bronstein defined two relations other than strict input/output equivalence: the α- and β-relations, capturing delay and stuttering due to pipelining respectively. Stuttering occurs in a pipelined CPU which, if a branch is taken, does not make progress for a number of clock cycles. Bronstein used the Boyer-Moore theorem prover for the verification of these relations. This allows for the verification of possibly large designs, but requires sophisticated and extensive user intervention. String function theory is further elaborated on in Chapter 2.

A more efficient method, called SFG-tracing, is described in [17]. It is targeted towards digitial signal processing (DSP) applications, where the specification consists of a signal flow graph (SFG). The SFG shows the computation to be performed on each input sample. It can be considered as a maximally parallel machine, with high throughput. The implementation is built from more economical hardware, and realizes a lower throughput. The verification consists of two phases. In a first phase the initialization sequence is verified, while in the second phase the steady state behavior is verified. In each of these phases a symbolic simulation is performed for a number of clock cycles, where for symbolic states and symbolic inputs, symbolic next states and outputs are computed. The BDDs for the next

state and output functions are piecewise constructed and compared with the corresponding subfunctions of the specification. The major disadvantage of this method is that the data path logic, which stays the same for many implementations and can be hard to represent with BDDs, is reconsidered during the verification of every new circuit.

Further relevant research, not directly addressing behavioral equivalence but nevertheless of use in this context, includes the work on several varieties of Temporal Logic (e.g. [18] [27]). These logics contain the usual propositional logic operators (AND, OR, NOT) as well as a set of temporal operators ("always", "sometime", "until", etc.). They express a condition on either a sequence of events, or on a situation (which can be the result of different sequences of events in the past, and can lead to different courses in the future). The best known example of the first variety is Propositional Linear Temporal Logic (PLTL), the one of the second variety Computation Tree Logic (CTL). Formulae in these logics can be verified against a finite state machine by inspecting and manipulating the information inherent in the state transition graph of the machine. This procedure is called model checking, and is fully automatic, in contrast with theorem proving techniques. The model checking problem is in P, for CTL, and is PSPACE-complete, for PLTL. CTL formulae can be checked with the same BDD-based procedures for STG traversal that are used for checking FSM equivalence [15]. Note however that the specification of correctness properties has traditionally been a manual process. I will further elaborate on CTL model checking in Chapter 3.

1.4 The Work Described in this Thesis

To formalize behavioral specifications and behavioral equivalence, I have adopted the approach of Bronstein. I deal with synchronous logic designs, with which a string function can be associated, and as a specification a synchronous machine is taken as well. Behavioral equivalence is captured by relations between (the string functions associated with) an implementation and a specification.

It is sometimes argued that behavioral specifications are needed with no bias towards a specific way of performing the computation. The input/output mapping to be performed

should, in this view, be captured in a pure form, without algorithmic elements indicating how this mapping can be realized. I believe that behavioral specifications without algorithmic bias are, in most cases, not possible. A representation of an intended computation has somehow to link input streams with output streams. This can be done by specifying how the outputs can be computed from the inputs, or how the inputs can be computed from the outputs, or, in general, by a system of equations linking the input stream with the output stream. In most cases an algorithmic description for computing the outputs from the inputs seems the most straightforward way. This is also how computational structures are specified in practice. For example, digital signal processing (DSP) computations are commonly specified with a signal flow graph, which can be interpreted as a maximally parallel circuit. (Usually these signal flow graphs specify the bit-true behavior of the circuit, which means that they can be expanded into logic circuits.) As another example, microprocessors are mostly designed from earlier designs, with the objective of realizing a similar functionality with higher performance.

I define five primitive relations between string functions, other than strict automata equivalence (cf. Section 2). They are the "don't care times" relation (β), the parallelism relation (γ), the encoding relation (δ), the "input don't care" relation (ϵ), and the "output don't care" relation (ζ). The notation (Greek letters) is adopted from Bronstein. Bronstein defined an α-relation (delay) and a β-relation (stuttering). I generalized the β-relation and dropped the α-relation, as it is almost subsumed by my β-relation. All other relations are new. These relations have attributes. For instance, the parallelism relation has an attribute corresponding to the degree of parallelism.

The first part of the thesis is a sound and complete proof method for proving arbitrary compositions of the five primitive relations, under certain assumptions. The proof method consists of circuit transformations on the specification and the implementation, followed by *one* equivalence check (on the transformed circuits). I demonstrate that, if the proof method declares the implementation and specification to be in a certain relation, they are so (soundness), and that, if a certain relations holds, it can be proven with the proof method (completeness).

This result is obtained in several steps. First, I show for each primitive relation how the specification and the implementation can be transformed so that the relation holds if and only if the transformed circuits are equivalent. Then, I show that, given an arbitrary composite relation, attributes for the relations in a canonic composition can be derived (under certain assumptions), such that the original relation holds if and only if the canonic relation holds. In the canonic relation, all the primitive relations occur at most once, and in a predetermined order. For the canonic relation I derive again transformations on the specification and implementation so that the relation holds if and only if the transformed circuits are equivalent.

The sound and complete proof method is only efficient enough to be applicable for verifying small to medium sized circuits, consisting of two or three logic or arithmetic units, and containing up to about 50 flip-flops. This will not suffice for the behavioral verification of realistic digital designs, but could form a building block for a compositional verification method where conditions on fragments of the circuits are verified to derive a conclusion about correctness of the overall circuit.

The second part of the thesis is an event-based compositional method for verifying synchronous processors against signal flow graphs (containing state variables) or dependency graphs (containing no state variables). The specifications can be considered as synchronous processors themselves, with maximally parallel hardware. The compositional method allows for verifying the β-relation between an implementation and a specification. The main application domain for this method is digital signal processing.

The method assumes that the implementation performs the same set of operations as the specification. It represents a sound but not complete proof method for proving the β-relation. The assumption that the same set of operations is performed is consistent with the following design scenario. A signal flow graph or dependency graph is taken as a specification. This specification might be represented in textual form, for instance as a program in the SILAGE language [30], which is an applicative language for describing signal flow graphs. At some point in the design process data path hardware is being allocated for the specified computation. The original signal flow graph can then be expanded so

that every operation in the new signal flow graph corresponds to a register transfer in the implementation. If for instance a multiplication operation is contained in the specification, and only ALU's are allocated in the data path, the multiplication has to be expanded into additions and shifts.

The operations in the new signal flow graphs have to be scheduled on the allocated data path hardware, and control logic has to be synthesized to implement the schedule. My compositional verification method can be used to verify across the last two design steps: scheduling and control synthesis. It allows for the verification of the β-relation between an implementation and an expanded signal flow graph. The β-relation has a transitivity property so that, if the expanded SFG is also proven to be in β-relation with the original SFG, the implementation is in β-relation with the original specification.

The compositional method verifies whether the nodes of the expanded signal flow graph and its edges are properly implemented. The nodes are properly implemented if there are corresponding combinational blocks in the data path of the implementation which compute the same function. The edges are properly implemented if the controller satisfies a number of past-tense Computation Tree Logic (CTL) formulae. It is formally demonstrated that the CTL formulae represent sufficient conditions for the implementation to be in β-relation with the expanded SFG (assuming the data path modules are combinationally correct). This leads to a compositional strategy where the data path modules are verified with combinational equivalence checking techniques, and the controller with CTL model checking procedures, which search the state space of the controller only. In this way much larger designs can be verified than with techniques which search the state space of the entire design.

The compositional method is event-based: the CTL conditions express constraints on *events*, initiated by the controller, such as register transfers. This contrasts with the method described in [17] which reduces the overall correctness condition to constraints on *functions*. The latter method does not achieve a separation between data path and controller.

While I have found it convenient to use CTL as a formalism for expressing the correctness conditions, other propositional temporal logics could have been used instead.

Experimental results are presented both for the first (non-compositional) and the second (compositional) verification procedure. The compositional method has been applied both to microcoded and systolic array processors. The microcoded designs verified were generated with the CATHEDRAL-II synthesis system [28].

In my view, the main contribution of the thesis is the particular way in which manual and mechanical proof work is combined to prove an overall correctness requirement for realistic designs. I prove by hand that the overall correctness condition can be reduced to a set of conditions which in turn can be verified efficiently by mechanical means. The manual portion of the proof needs to be done only once. The result of this proof exercise, together with the verification procedures for verifying the lower-level conditions, is compiled into a fully automatic and efficient procedure for verifying the overall correctness requirement.

The remainder of the thesis is organized as follows. Chapter 2 presents string functions and relations between string functions. In Chapter 3 automata-based verification procedures are reviewed. These are used as building blocks in the rest of the thesis. Chapter 4 presents the sound and complete proof method for verifying arbitrary combinations of the β-, γ-, δ-, ϵ- and ζ-relations, based on automata equivalence checking. Chapter 5 introduces the event-based compositional method for proving the β-relation. The thesis ends with conclusions in Chapter 6.

Chapter 2

String Function Relations

2.1 Introduction

There are two fundamental ways of describing deterministic sequential machines with functions. One way is with a function taking inputs and present states and producing outputs and next states. The other way is with a function taking a sequence (string) of inputs and producing a sequence of outputs, which describes the behavior of a sequential machine for a given initial condition. Both models have their particular merits. The first model is indispensable for implementing a circuit, and is useful for many manipulations in the design process, thanks to its being finite. The second model captures more directly the input/output behavior of a circuit, and is useful for expressing and proving certain properties, but, due to its being infinite (it is a mapping from arbitrarily long input streams to equally long output streams), it seems to be poorly suited for mechanical manipulations. I use both models in the thesis. The string model is used to define behavioral relations between sequential machines, and to prove certain properties of these relations by hand. The operational model is used for the bottom-level, mechanized verification work.

This chapter presents string functions and relations between them. Section 1 introduces strings, string functions, and matters of notation, as they were presented in Bronstein's work [9]. Section 2 presents five new primitive relations between string functions. These relations can be composed in arbitrary ways. The proof obligation for behavioral equivalence

will be that a particular relation holds between the string function corresponding to the implementation, and the one corresponding to the specification. This will be the subject of Chapters 4 and 5.

2.2 Concepts and Notation

Consider an alphabet Σ of values that can appear at the input and output ports of a synchronous circuit. Strings can be defined as finite concatenations of characters in the alphabet. Formally, the set of strings, Σ^*, is defined as $\bigcup(\Sigma^i)_{i\in\omega}$ where ω is the set of natural numbers including 0. The symbols $x, y, z, x_1, y_1, z_1, ...$ will be used for variable strings, $a, u, u_1, ...$ for variable characters, and ε will represent the empty string.

The following operations on strings are useful:

- .: Add: $\Sigma^* \times \Sigma \rightarrow \Sigma^*$, add a character to the right, *i.e.* a most recent character.
- ||: Length: $\Sigma^* \rightarrow \omega$, length of a string.
- $\preceq$: Prefix: $\Sigma^* \times \Sigma^* \rightarrow \{T, F\}$, prefix relation on strings.
  Defined by: $(x \preceq \varepsilon \Leftrightarrow x = \varepsilon)$ and $(x \preceq y.u \Leftrightarrow x = y.u \text{ or } x \preceq y)$.
- .: Concatenate: $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$, concatenate two strings (the second string to the right of the first string). Note that the "." symbol is overloaded. Sometimes the dot will be omitted.
- L: Last: $\Sigma^* \rightarrow \Sigma$, last character of a string.
  Defined by: $L(x.u) = u$ (and $L(\varepsilon) = \varepsilon$ for totality).
- P: Past: $\Sigma^* \rightarrow \Sigma^*$, all characters of a string except the last one. Defined by: $P(x.u) = x$ (and $P(\varepsilon) = \varepsilon$ for totality).
- $\uparrow$: To the power: $\Sigma \times \omega \rightarrow \Sigma^*$, construct a string by repeating a character a number of times.

- $\downarrow$: At position: $\Sigma^* \times \omega \rightarrow \Sigma$, extract a character from a string. I will use $x \downarrow i..j$ as an abbreviation for the string $(x \downarrow i). \ldots .(x \downarrow j)$.

Synchronous systems are constructed from two kinds of building blocks: combinational blocks, implementing a string function f^*, the string extension of a character function f; and registers, which implement a register function R_a that inserts a character a to the left of an input string and cuts off the right most character ($R_a(x.u) = a.x$ and $R_a(\varepsilon) = \varepsilon$).

It is formally demonstrated in [9] that synchronous systems, composed from these primitives, in which every loop contains a register, have a unique associated string function from Σ^* to Σ^*. I will denote such synchronous systems with S_F, S_G, etc., and the corresponding string functions with F and G. It can be seen that such string functions are length preserving (the output string has the same length as the input string) and prefix preserving (if x is a prefix of y, the image of x under the string function is a prefix of the image of y).

As in [9], I use Greek letters for relations between string functions (realizable by synchronous machines). For example, $F \rho G$ means that F is in ρ relation with G. The symbol "$\circ$" (read "after") denotes composition (of functions or relations). $F_2 \circ F_1$ is the composite function that results from applying F_2 after F_1. $F (\rho_2 \circ \rho_1) G$ if and only if there exists a string function K, realizable by some synchronous machine, such that $F \rho_1 K$ and $K \rho_2 G$.

The alphabets of interest to us contain vectors of Booleans of some length. I will use 0 both for the scalar 0 and for vectors containing all 0's. *zero* is the string function taking arbitrary strings x and returning $0 \uparrow | x |$. Likewise for 1 and *one*.

2.3 Primitive Relations

Consider an implementation and a specification, both of them being synchronous systems, S_F and S_G respectively, with corresponding string functions F and G. I want to allow a designer to assert and verify relations between F and G. One such relation is strict equivalence. I have defined five primitive relations between string functions other than strict equivalence:

1. β, the "don't care times" relation: the implementation produces the same output as the specification except at don't care times.

2. γ, the parallelism relation: the implementation is a parallelized version of the specification. [1]

3. δ, the encoding relation: the implementation is identical to the specification modulo combinational input and output encoding.

4. ϵ, the "input don't cares" relation: implementation and specification are identical as long as certain input strings are not applied.

5. ζ, the "output don't cares" relation: implementation and specification are identical if no distinction is made between output strings in certain equivalence classes.

In using Greek letters for the primitive relations, I follow the notation of Bronstein ([10]). Bronstein defined relations α and β. I significantly altered the definition of the β-relation, and dropped the α-relation because it is virtually subsumed by the new β-relation.

This section presents formal definitions of the primitive relations. The relations correspond to changes in the input/output behavior that frequently result from a behavioral or sequential logic synthesis step. Implementations may be pipelined (the don't care times relation describes this), parallel or serial (the parallelism relation describes this), or optimized with respect to sequential don't cares [21] (the input don't cares and output don't cares relations describe this). In general an arbitrarily long chain of behavioral transformations may be applied during synthesis. The resulting implementation is linked with the original circuit by a composition of the five primitive relations, with every primitive relation capturing the modification in input/output behavior caused by one behavioral transformation.

[1] The reverse situation could also occur: the specification is parallel and the implementation serial. One could define a relation γ_a for the first case and γ_b for the second. The properties of γ_b would be slightly different from the ones of γ_a.

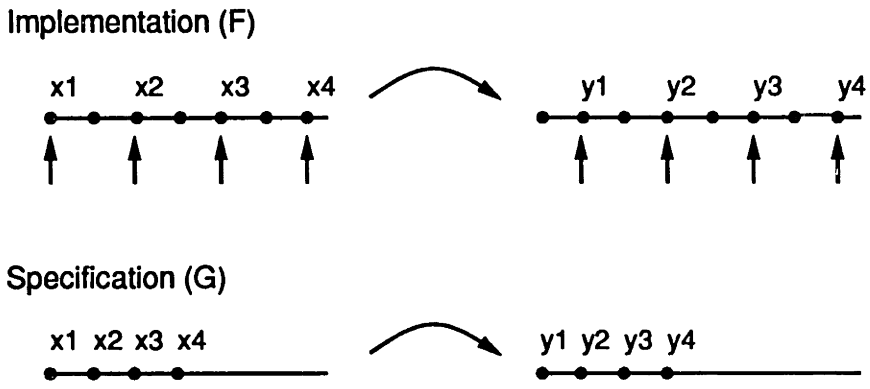


Figure 2.1: Illustration of the β-relation

2.3.1 The "Don't care times" Relation

The "don't care times" relation relates implementation/specification pairs such as the one illustrated in Figure 2.1. The implementation ignores input values at certain time points, and produces irrelevant outputs at certain time points. In addition, the output stream of the implementation is delayed with respect to the one of the specification.

To formally define the β-relation, it is convenient to first define a function that selects the relevant values in a string. *i.e.* omits the ones at "don't care times".

Definition 2.1 $Relevant : \Sigma^* \underline{\times} B^* \rightarrow \Sigma^*$:

$$\begin{aligned} Relevant(\varepsilon, \varepsilon) &= \varepsilon \\ Relevant(x.u, y.v) &= Relevant(x, y) \quad , v = 0 \\ &= Relevant(x, y).u \quad , v = 1 \end{aligned}$$

The symbol $\underline{\times}$ represents the string cartesian product, which combines strings of equal length. The *Relevant*-function takes a string over an arbitrary alphabet and a Boolean-valued string, and returns what remains of the first string after deleting all values for which there is a 0 in the corresponding place in the Boolean-valued string.

If we ignore the fact that the output stream of the implementation can be delayed with respect to the one of the specification, the β-relation could be defined by requiring:

$$\forall x \in \Sigma^*, Relevant(F(x), H(x)) = G(Relevant(x, H(x)))$$

For the example of Figure 2.1, the H-function would be a modulo-2 counter. The requirement above expresses that applying F to x, and then picking out the relevant outputs, yields the same result as applying G to the relevant inputs. The definition of the β-relation is slightly more complicated due to possible non-zero delays between the output streams of the implementation and the specification.

Definition 2.2 *Let H be a length and prefix preserving string function from Σ^* to B^*, realizable by some synchronous machine.*
$F\ \beta_{H,n}\ G \Leftrightarrow \forall x \in \Sigma^*\ s.t.\ |\ x\ | \geq n,$
$Relevant(F(x), R_{0\uparrow n} \circ H(x)) = G(Relevant(x \downarrow 1..(|\ x\ | - n), H(x \downarrow 1..(|\ x\ | - n))))$

Because of the delay of the output stream of the implementation, the filtering function H in the left-hand side of the identity has to be delayed over n cycles. In the right-hand side, the last n characters of the input string have to be eliminated to make the left-hand side and the right-hand side strings of equal length.

Design examples where don't care times occur are:

- A pipelined CPU which "stutters" when taking a branch. Because of the pipelining, not enough information is available for timely execution of a conditional branch. By default the branch is not taken, and when it later appears that the branch should have been taken, the pipeline has to be purged during a number of clock cycles. The outputs produced at those clock cycles are irrelevant.

- Implementation/specification pairs such as the one shown in Figure 2.2. The controller of the implementation is not shown. It repetitively sequences through 6 states, performing all of the operations of the specification serially. (Taking an input is also counted as an operation.)

The "don't care times" relation is inspired from the stutter-relation in [9], but is slightly different. The α-relation was defined in [9] in the following way:

$$F\ \alpha_z\ G \Leftrightarrow \forall z' \mid\ |\ z\ | = |\ z'\ |, \forall x \in \Sigma^*,\ F(x.z') = z.G(x)$$

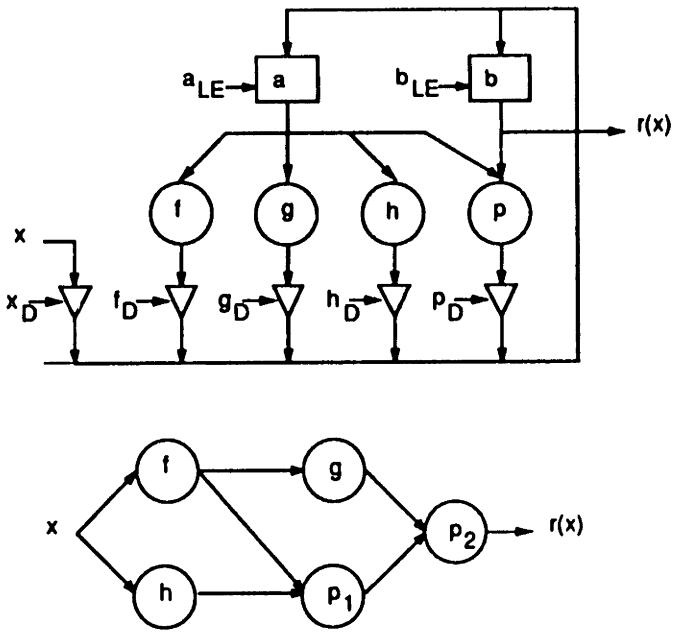


Figure 2.2: Example of an implementation and a specification which are in β relation

It is almost subsumed by the β-relation. If only the delay of F with respect to G is important, and not the characters in z, the relationship between F and G is better described by $\beta_{one,|z|}$.

2.3.2 The Parallelism Relation

The definition of the parallelism relation relies on a function C_m which compresses a string in such a way that m consecutive values are merged into one value with m components. In other words, C_m parallelizes a string with a factor m. By compression, we mean compression in time, not in space.

Definition 2.3

$$C_m : (\Sigma^m)^* \to (\Sigma^*)^m : C_m(\varepsilon) = (\underbrace{\varepsilon, \varepsilon, \ldots, \varepsilon}_{m}) \ \wedge$$

$$C_m(x.\underbrace{u.v.\ \ldots\ .w}_{m}) = C_m(x).(u, v, \ \ldots\ , w)$$

$(\Sigma^m)^*$ contains concatenations of strings of length m. $(\Sigma^*)^{\underline{m}} = \Sigma^* \underline{\times} ... \underline{\times} \Sigma^*$, m times, and $\underline{\times}$, the string cartesian product, combines strings of equal length. I will denote the inverse function of C_m by E_m (where E stands for Expansion).

Definition 2.4 $F\ \gamma_m\ G \Leftrightarrow \forall x \in (\Sigma^m)^*, F \circ C_m(x) = C_m \circ G(x)$

In words, F is in γ relation with G if compressing (parallelizing) the input stream before applying F produces the same result as applying G followed by compression. [2]

Example 2.1 $F((u,v,w),...) = (a,b,c)...$ and $G(uvw...) = abc...$ implies that F and G are in γ_3-relation.

2.3.3 The Encoding Relation

Definition 2.5 *Let k and l be invertible combinational functions.*
$F\ \delta_{f,g}\ G \Leftrightarrow F \circ k^* = l^* \circ G$

The encoding relation is motivated by the fact that a circuit can sometimes be better optimized after input and output encoding.

2.3.4 The "Input don't cares" Relation

Definition 2.6 *Let IDC (Input Don't Care set) be a set of strings.*
$F\ \epsilon_{IDC}\ G \Leftrightarrow \forall x \in \Sigma^*, \neg(\exists y \in IDC \mid y \preceq x) \Rightarrow F(x) = G(x)$

In words, input strings that do not have a prefix in the input don't care set, should result in identical outputs, for F and G to be in ϵ-relation.

The "input don't cares" relation is motivated by the fact that in certain cases a number of input string cannot occur, as the circuit providing these inputs cannot produce them. The freedom to associate arbitrary outputs with these inputs can be exploited to optimize the design.

[2] As mentioned before, one could also imagine the implementation (F) being serial and the specification (G) parallel. To make the theory more complete, one would have to define a γ_a-relation for the first case and a γ_b-relation for the second. The interactions of γ_b with the other primitive relations are different from the ones of γ_a and would result in a different canonic composite relation, to be discussed in Chapter 4.

2.3.5 The "Output don't cares" Relation

Definition 2.7 *An output don't care set ODC is a pair*

($\{ODC_i \mid ODC_i \subset \Sigma^{n_i} \wedge \bigcap ODC_i = \Phi \wedge \{ODC_i\}$ is finite $\wedge$ no element of $\bigcup ODC_i$ is a prefix of another element $\wedge$ every ODC_i has at least one element $\}$,

Special),

where Special takes sets ODC_i, and returns one (special) element of that set.

$Subst_{ODC} : \Sigma^* \rightarrow \Sigma^* : Subst_{ODC}(x) = (if (\exists i, y \mid x = y.z \wedge y \in ODC_i)$ *then* $Special(ODC_i).z$ *else* $x)$

$Subst_{ODC}$ replaces prefixes that are in some ODC_i with the special element of that set. In order for this function to be well-defined, all the sets ODC_i should be disjoint, and no element in it should have a prefix in one of the sets ODC_i. ODC should be finite for a theorem to be presented in Chapter 4 to hold.

Definition 2.8 *Let ODC be an output don't care set.*

$F \zeta_{ODC} G \Leftrightarrow Subst_{ODC} \circ F = Subst_{ODC} \circ G$

"Output don't cares" occur when the circuit to which the outputs are presented does not distinguish between certain strings. This can again be exploited during optimization of the design.

Chapter 3

Automata-theoretic Verification Procedures

3.1 Introduction

This chapter presents elementary verification procedures which are based on the finite automaton-model. Two classes of procedures are presented: procedures for verifying the input/output equivalence of two deterministic finite state machines, and model checking procedures for verifying Computation Tree Logic (CTL) formulae against a state transition graph.

Both classes of procedures perform traversals of a state transition graph. For equivalence checking, a product machine is constructed from the two machines, which produces an output of 1 if and only if the two machines agree on a given input. The transition graph of the product machine has to be traversed to see if for all states and under all input combinations an output of 1 is produced. CTL formulae too can be verified by traversing a transition graph from a certain set of states.

There are a number of different strategies for traversing a state transition graph. Starting from a certain state, the input combinations leading to a transition out of the state can be enumerated explicitly (i.e. one by one) or implicitly (i.e. in sets). Likewise, the states in the transition graph can be enumerated explicitly or implicitly. For implicit enumeration, different representations can be used for sets of input combinations or sets of states (e.g. cubes, sum-of-products, binary decision diagrams). Finally, the traversal can be done depth-

first or breadth-first. In this work no commitment is made to any of these strategies. However, the experiments are done with implicit input and state enumeration, based on BDDs, with breadth-first traversal. This is at present the most robust (i.e. most widely applicable) traversal technique.

Section 3.2 of this chapter presents the binary decision diagram (BDD)-representation for Boolean functions. Section 3.3 presents image computation routines, for computing the set of states reachable in one step from a given set of states under certain input conditions. These procedures perform implicit input and state enumeration, using BDDs, and can be used for breadth-first traversal. They form the core of equivalence and model checking procedures presented in Section 3.4 and 3.5 respectively.

3.2 Binary Decision Diagrams

A *binary decision diagram* for a function $f(x_1, x_2, \ldots, x_n)$ (for an example see Figure 3.1) is a graph with non-terminal vertices annotated with input variables, and terminal vertices annotated with the constant 0 or 1. All non-terminal vertices have one or more parent vertices and two children vertices, except for the unique root vertex, which has no parents. The terminal vertices have one or more parents and no children. The two edges going down from a non-terminal vertex to their children are annotated with a 0 and a 1. For a given assignment to the inputs variables x_i, one can go down the decision diagram choosing 1-edges where a variable has the value 1, and 0-edges where a variable has the value 0. The value of the terminal vertex is the value of f under that input combination.

An *ordered BDD* is one where the ordering of the variables is the same along any path from the top vertex to a terminal vertex. A *reduced BDD* has no vertex which has 1- and 0-edges pointing to the same vertex, or pointing to isomorphic subgraphs. The example BDD in Figure 3.1 is both ordered and reduced. Reduced ordered BDDs (ROBDDs) were introduced in [11]. They have a number of properties that make them very well suited for verification. Foremost is the fact that they are canonical, so that equivalence checking between two functions is done by checking for isomorphism between the corresponding

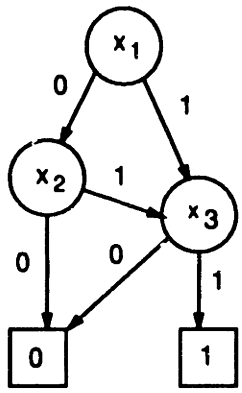


Figure 3.1: Reduced ordered binary decision diagram representing $f = x_1x_3 + \overline{x_1}x_2x_3$

BDDs, which requires linear time.

It is not necessary for a Boolean function representation to be canonical to be suitable for equivalence checking. An alternative way of checking equivalence is to apply the XOR-function to the outputs of both functions, and to check for satisfiability of the resulting function. Both operations can be efficiently performed with BDDs. Satisfiability checking requires constant time, and applying a Boolean function to two BDDs requires time proportional to the product of the sizes of the BDDs to which the function is applied.

The apply-operation is also used to construct the BDD for a given logic function. Starting from the input variable (for which the BDD is trivial), Boolean functions AND, OR, XOR, XNOR, etc. are applied to BDDs of subfunctions, until a BDD is obtained for the overall function.

The apply-operation can be performed recursively as follows. To compute f_1 $\langle op \rangle$ f_2, given ROBDDs f_1 and f_2 with top vertices v_1 and v_2 respectively, do the following:

1. If v_1 and v_2 are terminal vertices, generate a terminal vertex annotated with *value* (v_1) $\langle op \rangle$ *value* (v_2).

2. If v_1 and v_2 are annotated with the same variable, construct a vertex annotated with that variable, having as 0-child the result of applying $\langle op \rangle$ to the 0-children of v_1 and v_2, and as 1-child the result of applying $\langle op \rangle$ to the 1-children of v_1 and v_2.

3. If the variables are different, or if one vertex is a terminal vertex, construct a vertex annotated with the variable coming earliest in the variable ordering (i.e. the one that has to appear first along any path from top to terminal vertex). Say that this variable corresponds with v_1. Give to the constructed vertex as 0-child the result of applying $\langle op \rangle$ to v_2 and the 0-child of v_1, and as 1-child the result of applying $\langle op \rangle$ to v_2 and the 1-child of v_1.

The size of an ROBDD is critically dependent on the ordering of input variables. To obtain small BDDs, the variables have to be ordered in such a way that assigning values to the first m input variables ($0 \leq m \leq n$) can, for the purpose of computing f, be "remembered" with less information than the detailed assignment (which can take 2^m different values). For example, for computing the sum of two integers represented as bit vectors, it is advantageous to interleave the two input vectors, and order the bits from lowest order to highest order. In that case, the only information to be "remembered" from an assignment to the lower order input bits, for the computation of the higher order bits, is one carry-bit.

Such variable orderings are not always feasible. For instance, it has been shown that an ROBDD for a multiplier takes $\Omega(1.09^n)$ space regardless of the variable ordering [13]. Most other practical Boolean functions, however, can be represented efficiently as an ROBDD.

3.3 Image Computations Using BDDs

Image computations constitute the core of the verification procedures to be discussed in the next two sections. The problem is as follows. Let f be a function from B^n to B^m, where $B = \{0, 1\}$. Let A be a subset of B^n. Compute $f(A)$, the image of A under f, defined as $\{y \in B^m \mid \exists x \in B^n \text{ s.t. } y = f(x)\}$.

A simple and elegant method for solving this problem is the transition relation method, first introduced in [20]. It works with a BDD representation of the transition relation R corresponding to the function f. The transition relation maps inputs from B^{n+m} into B. It produces an output of 1 if and only if the vector y composed of the last m inputs and the vector x composed of the first n inputs are such that $y = f(x)$.

Subsets A of B^n are in one-to-one correspondence with functions from B^n to B. The characteristic function of a set A is a function that returns 1 if and only if the input is an element of A. I overload the symbol A and use it for characteristic functions as well as for sets. Likewise, $f(A)$ is used for characteristic functions and sets.

The image $f(A)$ consists of y-vectors such that there is an x-vector for which both R and A are 1. The existential quantification can be computed with the smoothing operator, which is defined as follows.

Definition 3.1 *Let $f : B^n \rightarrow B$ be a Boolean function, and $x = (x_{i_1}, \ldots, x_{i_k})$ a set of input variables of f. The smoothing of f by x is defined as:*

$$\begin{aligned} S_x f &= S_{x_{i_1}} \ldots S_{x_{i_k}} f \\ S_{x_{i_j}} f &= f_{x_{i_j}} + f_{\overline{x_{i_j}}} \end{aligned}$$

In this definition f_a designates the cofactor of f by the literal a. For instance $f_{x_{i_j}}$ is f with x_{i_j} restricted to be 1, and $f_{\overline{x_{i_j}}}$ is f with x_{i_j} restricted to be 0. It can be seen that

$$(\exists x \mid R(x,y) \wedge A(x)) \Leftrightarrow S_x(R(x,y) \cdot A(x))$$

and therefore

$$f(A)(x) = S_x(R(x,y) \cdot A(x)) \tag{3.1}$$

Cofactoring with respect to a literal is a trivial operation on BDDs. For instance, cofactoring by x_i is done by deleting the x_i vertices and attaching their 1-children to their parents, reducing the resulting BDD if necessary. Applying OR- and AND-operations can be done as explained in Section 3.2.

The smoothing and AND-operation can be performed simultaneously [15]. The resulting procedure has the same recursive and case-structure as the one for the apply operation. The only difference occurs when the variable corresponding to a newly constructed top vertex has to be smoothed away. In that case the if-then-else of the left child, the constant 1, and the right child is computed. This operation is recursive in itself.

The transition relation method can also be used for inverse image computations, where outputs y are given, and inputs x have to be found such that $f(x) = y$.

An alternative method for image computations, making use of the Boolean function f only, and not of the transition relation R, was introduced in [20]. This method can only be used for forward image computations, not for inverse ones. A good exposition of the method, as well as an extensive set of experimental results, can be found in [43], which reports that the transition relation method was superior for all the test cases considered.

3.4 FSM Equivalence Checking Procedures

Once it is known how to perform image computations, it becomes easy to check the input/output equivalence of two FSM's. First, construct the product machine, consisting of the original machines with corresponding outputs feeding XNOR gates, and the XNOR gates feeding one AND gate. The product machine is such that it produces an output of 1 if and only if the two component machines agree on a given input.

Second, compute the set of reachable states of the product machine by iteratively computing sets C_i until $C_{i+1} = C_i$. Let s_0 be the initial state of the machine, I the set of all possible inputs, and f the next-state-function.

$$\begin{aligned} C_0 &= \{s_0\} \\ C_{i+1} &= C_i \bigcup f(C_i \times I) \end{aligned}$$

C_i is the set of states that can be reached in i or fewer steps. The final C_n (which is equal to C_{n-1}) constitutes the set of all reachable states.

With g being the output function, the two machines are equivalent if and only if $g(C_n \times I)$ is tautologous with 1.

3.5 CTL Model Checking Procedures

CTL is one particular instance of a Temporal Logic. In general, Temporal Logics contain the usual propositional operators (AND, OR, NOT) as well as a set of temporal operators

("always", "sometime", "until", etc.) [18] [27]. They express a condition on either a sequence of events, or on a situation (which can be the result of different sequences of events in the past, and can lead to different courses in the future).

The particular logic that I use for verification, past-tense Computation Tree Logic (CTL), can be best understood if future-tense CTL* is defined first. CTL* consists of state formulae, which are true in a specific state, and path formulae, which are true along a specific path [18]. A state formula is either:

- an atomic proposition,
- $\neg f$, $f \wedge g$ or $f \vee g$, where f and g are state formulae, or,
- $A(f)$ or $E(f)$, where f is a path formula.

$A(f)$ is true when f is true along all paths leaving from the state under consideration. $E(f)$ is true when there exists a path from the state along which f is true.

A path formula is either:

- a state formula, or
- $\neg f$, $f \wedge g$, $f \vee g$, X^+f, fU^+g, F^+f, or G^+f, where f and g are path formulae.

A path formula consisting of a state formula is true if the state formula is true in the first state. X^+f, which is read as "next time f", is true if f is true in the second state on the path. fU^+g, read "f until g", is true if there is a state in the path in which g is true, and in all states prior to it f is true. F^+f, read as "sometimes f" or "eventually f" or "henceforth f", is true if there is a state in the path in which f is true. G^+f, read as "always f", is true if f is true in all states on the path.

Past-tense CTL* is similar to future-tense CTL*. It has as temporal operators X^-, U^-, F^- and G^-, which have the same meaning as the corresponding future-tense operators, but with time reversed.

CTL is a subset of CTL* in which each path quantifier is immediately followed by exactly one of the temporal operators G, F, X or U. This restriction reduces the complexity of

verifying a formula against a state transition graph. This problem is called model checking. The state transition graph is called a model for the temporal logic formula if the formula is true for it. CTL Model checking is in P.

CTL model checking can be done by computing the set of states in which the CTL formula is true, and checking if it is equivalent with the set of all reachable states of the state transition graph. Computing the set of states in which a formula is true can be done by first applying a number of reductions. Universally quantified formulae can be reduced to existentially quantified ones:

- $AXf = \neg EX\neg f$
- $AFf = \neg EG\neg f$
- $AGf = \neg EF\neg f$
- $A(fUg) = \neg E(\neg gU(\neg f \wedge \neg g)) \wedge \neg EG\neg g$

These reductions are valid for both future-tense and past-tense CTL. It is assumed that time goes on indefinite in the future and in the past. The last reduction above has to be modified if this assumption is not valid. For instance, for the case of past-tense CTL where there is an initial time point beyond which there is no past, the following reduction has to be made:

$$A(fU^{-}g) = \neg E(\neg gU^{-}((init \vee \neg f) \wedge \neg g))$$

Formulae with F and G operators can be reduced to ones with the U operator only:

- $Ff = 1Uf$
- $Gf = \neg F\neg f$

We can now evaluate every CTL formula if we can evaluate $E(fUg)$ and EXf. We explain how EXf and $E(fUg)$ can be evaluated for past-tense CTL. The corresponding

procedure for future-tense CTL is identical, except that inverse image computations have to be performed instead of forward ones.

$EX^{-}f$ is evaluated by computing the image of the set in which f is true.

$E(fU^{-}g)$ is evaluated as follows:

1. Start with the set of states in which g is true.
2. Add to the current set the set of states which can be reached in one step from the current step and in which f is true.
3. If new states were added in the last step, go back to 2. Otherwise, return the current set of states.

This procedure could be formally derived as follows: $E(fUg)$ can be defined as the least fixed point of the following recursive equation:

$$E(fUg) = g \vee (f \wedge EX(E(fUg))) \tag{3.2}$$

By "least fixed point", a minimal cardinality set satisfying the recursive equation is meant. The recursive equation can be written as

$$E(fUg) = T(E(fUg))$$

where T is a set transformer. There is a theorem which says that, for a recursive equation of form (3.2), the least fixed point equals

$$\lim_{n \to \infty} T^{n}(\Phi)$$

The procedure above corresponds exactly with the computation of this limit. Repeated image computations are computed until convergence is reached.

Chapter 4

Proving String Function Relations through Automata Equivalence Checking

4.1 Introduction

The previous two chapters presented a formal definition of behavioral equivalence between an implementation and a specification, on the one hand, and a set of basic procedures for verifying certain correctness properties, on the other hand. The rest of the thesis is concerned with linking these two. I will show how the overall behavioral equivalence requirement can be reduced to correctness conditions which can be verified with the basic procedures.

This chapter presents a verification strategy which allows, under certain assumptions, for the verification of arbitrary compositions of the primitive relations defined in Chapter 2. It works for arbitrary implementations and specifications, and it is sound and complete. The assumptions are that multiple occurrences of β have modulo counters as H-functions, that parallel implementations have a prefix preserving serial equivalent, and that encodings on parallel vectors can be decomposed into encodings on the components of the vector. Soundness means that if the verification program declares an implementation and a specification behaviorally equivalent, they are equivalent. Completeness means that if an implementation and a specification are behaviorally equivalent, the verification program will declare them so. Soundness and completeness are achieved by reducing the overall correctness re-

quirement to more basic requirements, verifiable with the basic procedures of Chapter 3, which are necessary and sufficient for correctness. The verification method presented in this chapter is only efficient enough for small to medium-sized circuits, consisting of two or three logic or arithmetic units, and containing up to about 50 latches. This will not suffice for the behavioral verification of realistic digital designs. However, the verification method could form a building block for a compositional method where conditions on fragments of the circuits are verified to derive a conclusion concerning the correctness of the overall circuit.

This chapter is organized as follows. In Section 1, I show how for each primitive relation the specification and implementation can be transformed so that the relation holds if and only if the transformed circuits are equivalent. In Section 2, I show that, given an arbitrary composite relation, attributes for the relations in a canonic composition can be derived (under certain assumptions), such that the original relation holds if and only if the canonic relation holds. In the canonic relation, all the primitive relations occur at most once, and in a predetermined order. For the canonic relation I derive again transformations on the specification and the implementation so that the relation holds if and only if the transformed circuits are equivalent.

4.2 Verification of Primitive Relations

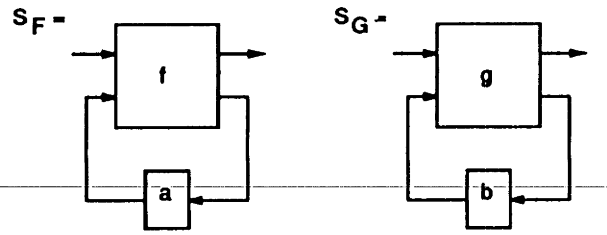
4.2.1 Verification of the "Don't care times" Relation

Recall the definition of the β-relation:

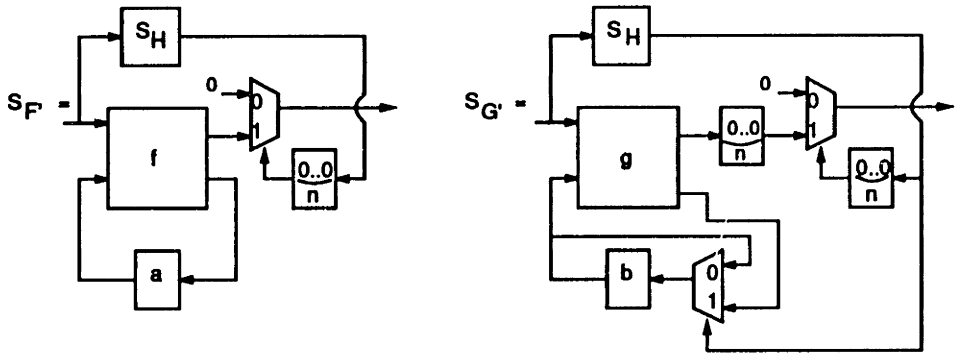
Definition 2.2 *Let H be a length and prefix preserving string function from Σ^* to B^*, realizable by some synchronous machine.*
$F\ \beta_{H,n}\ G \Leftrightarrow \forall x \in \Sigma^*\ s.t.\ |\ x\ | \geq n,$
$Relevant(F(x), R_{0\uparrow n} \circ H(x)) = G(Relevant(x \downarrow 1..(|\ x\ | - n), H(x \downarrow 1..(|\ x\ | - n))))$

Coarsely speaking, this definition says that string functions F and G are in β-relation if applying F to an input stream, and then picking out the relevant output values, gives the same result as applying G to the relevant input values only.

Theorem 4.1 *Given:*



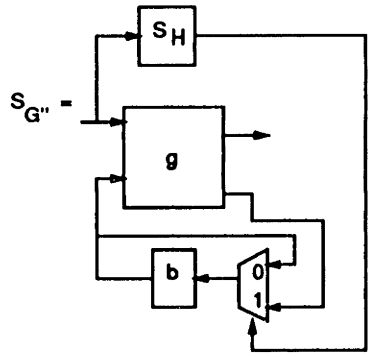
$F\ \beta_{H,n}\ G \Leftrightarrow F' = G'$ *where*



In the figures above, the boxes marked with f and g are combinational blocks computing the functions f and g, and the ones marked with a and b are registers with initial values a and b. The trapezoid shaped blocks are multiplexers. At any time, they select one of two data inputs in accordance with a Boolean valued control input. The boxes marked with "0..0" represent a chain of n registers with initial values of 0.

Proof:

Let g_1 be the part of g that computes the primary outputs, and g_2 the part that computes the next states. Let $Y_P^K(x)$ be the string at place P in system S_K, if x is seen at the input. Let $S_{G''}$ be the following system:



First, we prove inductively that for all prefixes y of x:

$$L(Y^{G''}_{R_{in}}(y)) = L(Y^{G}_{R_{in}}(Relevant(y, H(y))) \ (*)$$

1. true for $y = \epsilon$

2. assume true for y

   $L(H(y.u)) = 0 \Rightarrow \quad L(Y^{G''}_{R_{in}}(y.u)) = L(Y^{G''}_{R_{in}}(y))$ and
   $Relevant(y.u, H(y.u)) = Relevant(y, H(y))$

   By the inductive hypothesis, (*) follows.

   $L(H(y.u)) = 1 \Rightarrow \quad L(Y^{G''}_{R_{in}}(y.u)) = g_2(u, L(Y^{G''}_{R_{in}}(y)))$ and
   $L(Y^{G}_{R_{in}}(Relevant(y.u, H(y.u)))) = g_2(u, L(Y^{G}_{R_{in}}(Relevant(y, H(y)))))$

   By the inductive hypothesis, (*) follows.

With this result, we prove that $Relevant(G''(x), H(x)) = G(Relevant(x, H(x)))$:

For any prefix y of x:

$$\begin{aligned}
L(Y^{G''}_{out}(y)) &= g_1(L(y), L(Y^{G''}_{R_{out}}(y))) \\
&= g_1(L(y), \text{if } y = \epsilon, \epsilon, \text{ else if } \mid y \mid = 1, b, \text{ else } L(Y^{G''}_{R_{in}}(P(y)))) \\
&= g_1(L(y), \text{if } y = \epsilon, \epsilon, \text{ else if } \mid y \mid = 1, b, \text{ else } L(Y^{G}_{R_{in}}(Relevant(P(y), H(P(y)))))
\end{aligned}$$

$$\begin{aligned}
L(Relevant(Y^{G''}_{out}(y), H(y))) &= g_1(L(Relevant(y, H(y))), \text{if } y = \epsilon, \epsilon, \\
&\qquad \text{else if } \mid y \mid = 1, b, \text{ else } L(Y^{G}_{R_{in}}(Relevant(P(y), H(P(y)))))) \\
&= L(Y^{G}_{out}(Relevant(y, H(y))))
\end{aligned}$$

Therefore, $Relevant(Y^{G''}_{out}(x), H(x)) = Y^{G}_{out}(Relevant(x, H(x)))$

and thus $Relevant(G''(x), H(x)) = G(Relevant(x, H(x)))$

Since, furthermore,

$Relevant(R_{0\uparrow n} \circ G''(x), R_{0\uparrow n} \circ H(x)) = Relevant(G''(x \downarrow 1..(|\, x \,| - n)), H(x \downarrow 1..(|\, x \,| - n)))$

We get our final result:

$$\begin{aligned} F'(x) = G'(x) &\Leftrightarrow Relevant(F(x), R_{0\uparrow n} \circ H(x)) = Relevant(R_{0\uparrow n} \circ G''(x), R_{0\uparrow n} \circ H(x)) \\ &\Leftrightarrow Relevant(F(x), R_{0\uparrow n} \circ H(x)) = \\ &\quad G(Relevant(x \downarrow 1..(|\, x \,| - n), H(x \downarrow 0..(|\, x \,| - n)))) \end{aligned}$$

□

The theorem above effectively allows for the verification of the β-relation. Given a logic-level specification S_G and implementation S_F, transformations can be applied to S_F and S_G to obtain $S_{F'}$ and $S_{G'}$, and finite state machine (FSM) equivalence algorithms can then be used to check the β-relation between the specification and the implementation. Note that no commitment is made to any particular FSM equivalence algorithm.

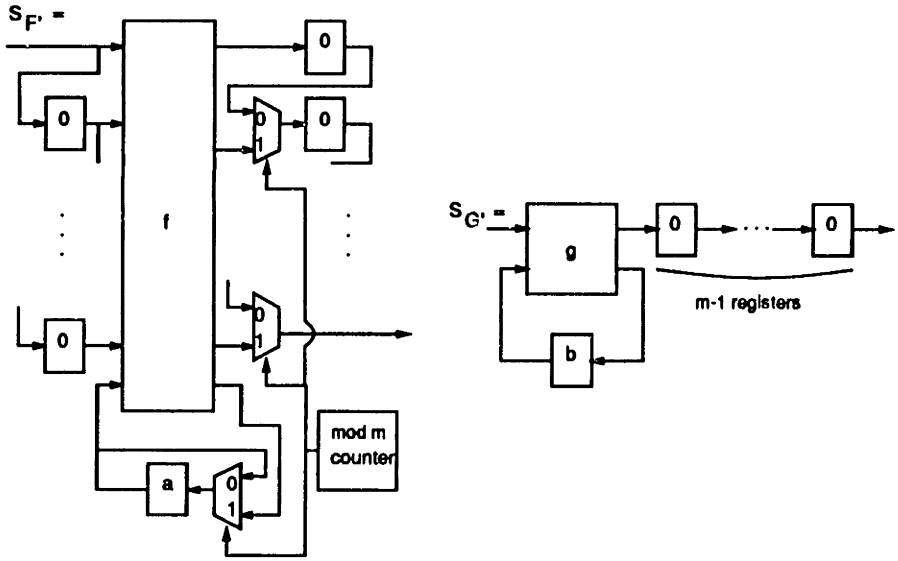
4.2.2 Verification of the Parallelism Relation

Recall the definition of the γ-relation:

Definition 2.4 $F\ \gamma_m\ G \Leftrightarrow \forall x \in (\Sigma^m)^*, F \circ C_m(x) = C_m \circ G(x)$

In words, the γ-relation applies if applying F after compression (i.e. after parallelizing) gives the same result as compressing after applying G.

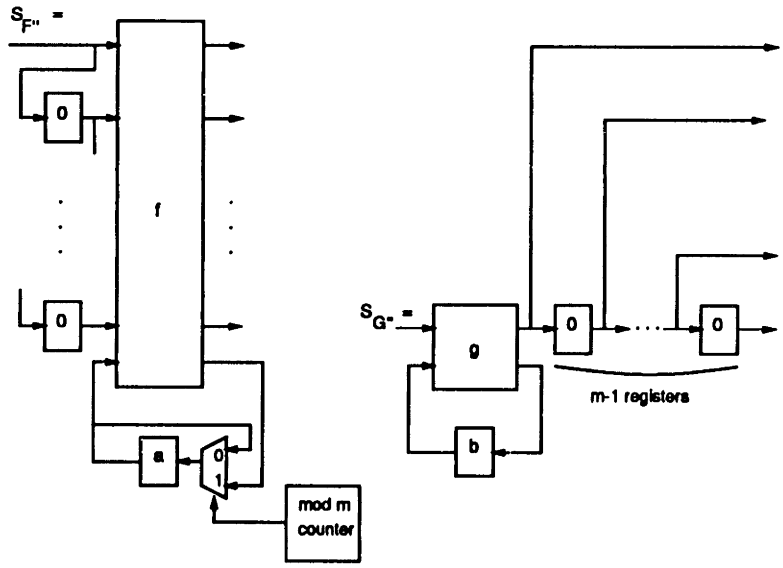
Theorem 4.2 $F\ \gamma_m\ G \Leftrightarrow \forall x \in \Sigma^*, F' = G'$ *where:*



with the "mod m counter" outputting a 1 every m^{th} cycle, and a 0 otherwise.

Proof:

Let $S_{F''}$ and $S_{G''}$ be as follows:



$$F \ \gamma_m \ G \Leftrightarrow \forall x \in (\Sigma^m)^*, F \circ C_m(x) = C_m \circ G(x)$$
$$\Leftrightarrow \forall x \in (\Sigma^m)^*, F''(x) = G''(x)$$
$$\Leftrightarrow \forall x \in \Sigma^*, F'(x) = G'(x)$$

□

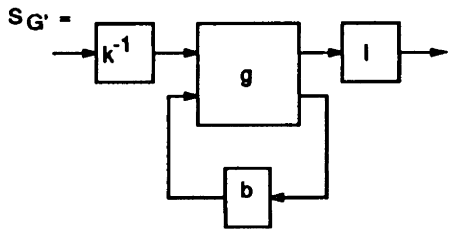
4.2.3 Verification of the Encoding Relation

Recall the definition of the δ-relation:

Definition 2.5 *Let k and l be invertible combinational functions.*
$F\ \delta_{f,g}\ G \Leftrightarrow F \circ k^* = l^* \circ G$

Theorem 4.3 $F\ \delta_{k,l}\ G \Leftrightarrow F = G'$

where S_F is as before and



Proof:
$F\ \delta_{k,l}\ G \Leftrightarrow F \circ k^* = l^* \circ G \Leftrightarrow F = l^* \circ G \circ k^{-1^*}$ □

4.2.4 Verification of the "Input don't cares" Relation

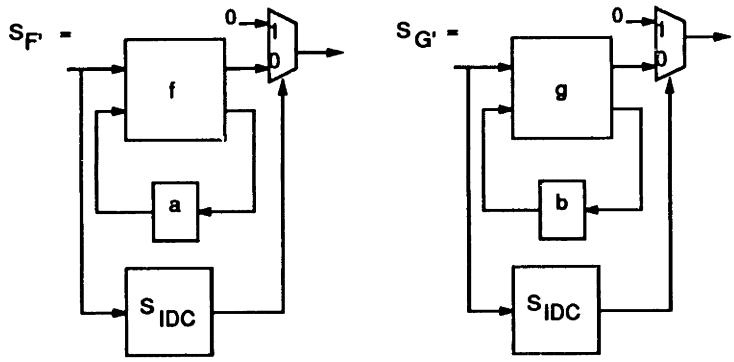
Recall the definition of the ϵ-relation:

Definition 2.6 *Let IDC (Input Don't Care set) be a set of strings.*
$F\ \epsilon_{IDC}\ G \Leftrightarrow \forall x \in \Sigma^*, \neg(\exists y \in IDC \mid y \preceq x) \Rightarrow F(x) = G(x)$

In words, input strings that do not have a prefix in the input don't care set, should result in identical outputs, for F and G to be in ϵ-relation.

Theorem 4.4 $F\ \epsilon_{IDC}\ G \Leftrightarrow F' = G'$

with



where S_{IDC} recognizes strings in IDC, and keeps outputting a 1 after detecting such a string.

This theorem is trivial.

4.2.5 Verification of the "Output don't cares" Relation

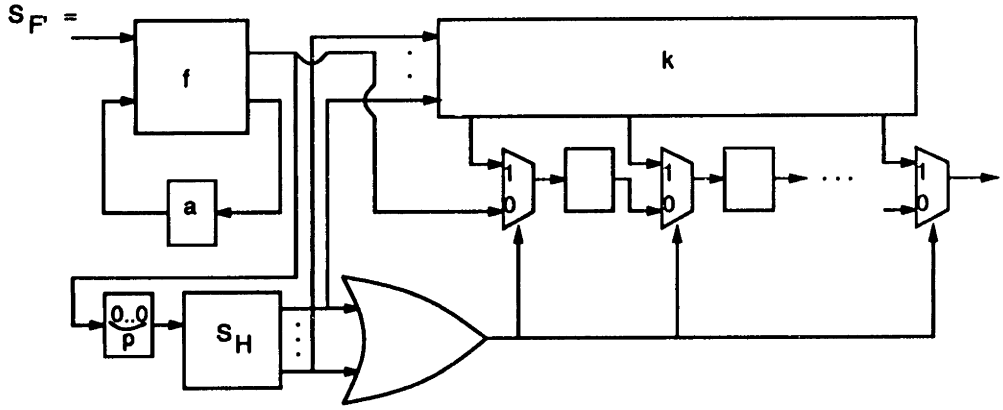
Recall the definition of the ζ-relation:

Definition 2.8 *Let ODC be an output don't care set.*
$F\ \zeta_{ODC}\ G \Leftrightarrow Subst_{ODC} \circ F = Subst_{ODC} \circ G$

"Output don't cares" occur when the circuit to which the outputs are produced does not distinguish between certain strings.

Theorem 4.5 $F\ \zeta_{ODC}\ G \Leftrightarrow F' = G'$

where



and $S_{G'}$ is similar. p is the length of the longest element in all the ODC_i. S_H produces a 1 on output i if it detects an element of ODC_i and a 0 otherwise. k outputs $Special(ODC_i)$ if its ith input is 1.

This theorem is trivial.

4.3 Verification of Composite Relations

Consider now arbitrary compositions of the five primitive relations. This corresponds to the application of an arbitrarily long sequence of behavioral transformations during synthesis. One way of verifying a composite relation is by verifying all its components, which would require as many equivalence checks as there are primitive relations, besides requiring from the user that he provide all the intermediate circuits.

In this section I show how this process can be simplified so that no intermediate circuits have to be provided, and so only one equivalence check is needed. The result established in this section consists of two parts. The first part is that, given a composite relation, where the primitives occur in *arbitrary order* and occur each *arbitrarily often*, under certain assumptions a *fixed order* composite relation can be constructed, where each primitive relation occurs at most *once*, such that the fixed order holds between a specification and an implementation if and only if the original relation holds. Therefore, in order to prove the original relation, it suffices to prove the canonic relation. The second part of the result is that the canonic relation can be verified by applying composite circuit transformations to the specification and the implementation and performing *one* equivalence check.

The first part of the result is established in two steps. First, I show that the primitive relations can be ordered in such a way that, for any pair of primitive relations ρ and σ, where ρ precedes σ in the ordering, if two string functions F and G are related by $\sigma_b \circ \rho_a$ where a and b are *arbitrary* attributes, they are also related by $\rho_{a'} \circ \sigma_{b'}$ where a' and b' are derived attributes, and if F and G are related by $\rho_{a'} \circ \sigma_{b'}$ where a' and b' are not arbitrary, but have a form that can be derived from attributes a and b, they are also related by $\sigma_b \circ \rho_a$. The following lemma is representative.

Lemma 4.1 *Given F, G where F is m-serializable:*

$(\exists K \mid F\ \beta_{H,n}\ K\ \wedge\ K\ \gamma_m\ G) \Leftrightarrow (\exists K' \mid F\ \gamma_m\ K'\ \wedge\ K'\ \beta_{H^s,mn}\ G)$

F, G and K are string functions realizable by synchronous machines, as are all string functions mentioned from here onwards. F being m-serializable means that the m-parallel output vectors can also be produced serially (implying that F needs to be prefix preserving *within* parallel vectors, in addition to being prefix preserving across such vectors). H^s is the serial equivalent of H, i.e. whenever H returns a 1 for a new input vector, H^s returns a sequence of 1's for the sequential equivalent of the input vector. (With C'_m being a function that takes m-wide vectors and returns the first element of it, $C'_m \circ H^s = H \circ C_m$. Recall that C_m compresses serial streams into parallel ones.)

The theorem states that F and G are related by $\gamma_m \circ \beta_{H,n}$, with m, H, and n arbitrary, if and only if they are related by $\beta_{H^s,mn} \circ \gamma_m$. The lemma suggests to take β after γ in the canonic composite relation, since, if two string functions are related by $\gamma \circ \beta$ with certain attributes, they are also related by $\beta \circ \gamma$ with derivable attributes. The inverse order would not do. If $F\ \gamma_m\ K$ and $K\ \beta_{H,n}\ G$, it is not in general possible to construct attributes H', n' and m' such that $F\ \beta_{H',n'}\ K'$ and $K'\ \gamma_{m'}\ G$ for some K'. The reason is that H', taking sequences of parallel vectors, is necessarily coarser than H.

Proof:

$F\ \beta_{H,n}\ K\ \ \wedge\ \ K\ \gamma_m\ G$

$\Leftrightarrow \forall x \in (\Sigma^*)^{\underline{m}}, Relevant(F(x), R_{0\uparrow n} \circ H(x)) =$

$\quad K(Relevant(x \downarrow 1..(|\,x\,| - n), H(x \downarrow 1..(|\,x\,| - n)))) \ \wedge$

$\quad \forall x' \in (\Sigma^m)^*, K \circ C_m(x') = C_m \circ G(x')$

$\Leftrightarrow \forall x' \in (\Sigma^m)^*, Relevant(\underbrace{F \circ C_m}_{C_m \circ K'}(x'), R_{0\uparrow n} \circ \underbrace{H \circ C_m}_{C'_m \circ H^s}(x')) =$

$\quad \underbrace{K \circ C_m}_{C_m \circ G}(Relevant(x' \downarrow 0..(|\,x'\,| - mn), H^s(x' \downarrow 0..(|\,x'\,| - mn)))),$

$\quad$ where $K' = F^s$

$\Leftrightarrow \forall x' \in (\Sigma^m)^*, F \circ C_m(x') = C_m \circ K'(x')\ \ \wedge$

$$\forall x' \in (\Sigma^m)^*, Relevant(C_m \circ K'(x'), C'_m \circ R_{0\uparrow mn} \circ H^s(x')) =$$
$$C_m \circ G(Relevant(x' \downarrow 0..(|\, x' \,| - mn), H^s(x' \downarrow 0..(|\, x' \,| - mn)))),$$
$$\Leftrightarrow \forall x' \in (\Sigma^m)^*, F \circ C_m(x') = C_m \circ K'(x') \ \wedge$$
$$\forall x' \in (\Sigma^m)^*, Relevant(K'(x'), R_{0\uparrow mn} \circ H^s(x')) =$$
$$G(Relevant(x' \downarrow 0..(|\, x' \,| - mn), H^s(x' \downarrow 0..(|\, x' \,| - mn)))),$$
$$\Leftrightarrow F\ \gamma_m\ K' \ \wedge$$
$$\forall x' \in \Sigma^*, Relevant(K'(x'), R_{0\uparrow mn} \circ H^s(x')) =$$
$$G(Relevant(x' \downarrow 0..(|\, x' \,| - mn), H^s(x' \downarrow 0..(|\, x' \,| - mn)))),$$
since H^s, K' and G are prefix preserving.
$$\Leftrightarrow F\ \gamma_m\ K' \ \wedge\ K'\ \beta_{H^s,mn}\ G$$ □

For every pair of primitive relations, a lemma like the one above is needed. These lemmas can be found in Appendix A (Lemmas 4.2 through 4.10).

We also have to demonstrate that if two string functions are related by $\rho \circ \rho$ with certain attributes, they are also related by ρ with a derivable attribute. In other words, repetitive occurrences of the same relation can be merged into one such relation. Lemmas 4.12 through 4.16 in Appendix A have this effect.

This leads to the following result:

Theorem 4.6 *Consider two string functions F and G, realizable by synchronous machines. Consider an arbitrary order composition of β, γ, δ, ϵ and ζ, with each primitive relation occurring arbitrarily many times. If there are multiple occurrences of β, it is assumed that the H-functions are delayed modulo-counters. If γ occurs, it is assumed that F has a prefix preserving serial equivalent, and that encodings k and l, associated with δ relations coming after γ, can be decomposed into identical encodings on individual elements of the input and output vector.*

A canonic composite relation $\delta \circ \beta \circ \epsilon \circ \zeta \circ \gamma$ can be constructed with derived attributes for each primitive relation, such that the original relation holds between F and G if and only if the canonic relation holds.

Example: Consider the following composite relation:

$$\beta_{H_1,n_1} \circ \delta_{k,l} \circ \gamma_m \circ \beta_{H_2,n_2}$$

Every pair of primitive relations can be reordered to comply with the fixed order. This relies on lemmas such as Lemma 4.1. Applied to the example, Lemma 4.1 allows the interchange of the last two relations (γ and β). Similarly the first and second relation can be interchanged so as to comply with the fixed order. This yields:

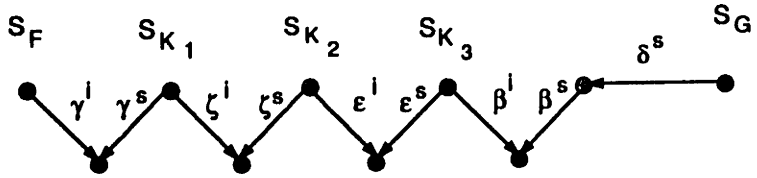
$$\delta_{k,l} \circ \beta_{H_1 \circ k^{-1^*},n_1} \circ \beta_{H_2^s,mn_2} \circ \gamma_m$$

Once all relations occur in the proper order, it suffices to merge repetitions of the same primitive relation. This can be done for all primitive relations. In our example, two β-relations have to be merged. This leads to the desired composite relation:

$$\delta_{k,l} \circ \beta_{H_3,n_3} \circ \gamma_m$$

where H_3 and n_3 can be computed following Lemma 4.12 (see Appendix A). □

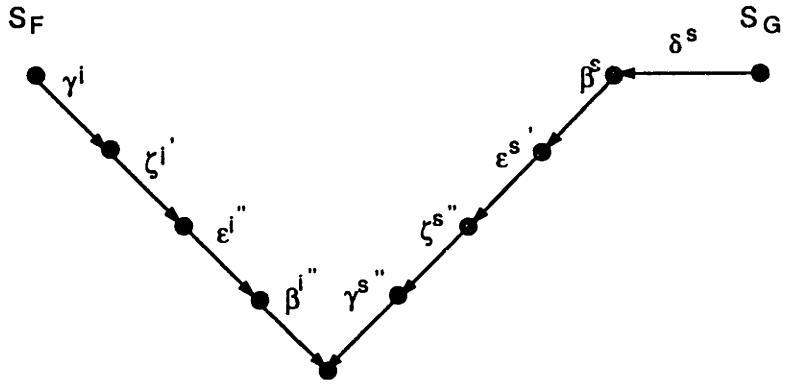
Consider finally the verification of the fixed order composite relation. Assume an implementation F and a specification G are related by it. By the theorems in the previous section, each primitive relation can be verified by transforming one or both circuits, and checking strict automata equivalence of the results. This leads to the following picture:



In this picture, γ^i, γ^s, etc., denote circuit transformations, on implementations and specifications respectively. The symbols γ etc. are overloaded, being used for relations as well as for circuit transformations.

To verify the composite relation between F and G, one could verify each primitive relation separately, given all the intermediate circuits. It can be seen immediately that the last intermediate circuit, adjacent to the arrows labeled β^s and δ^s, is not needed,

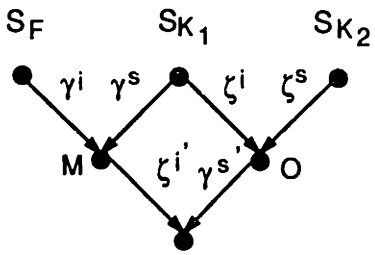
and that one equivalence check suffices for $\delta \circ \beta$ (checking $\beta^i(S_{K_3}) = \beta^s \circ \delta^s(S_G)$). It is possible to further reduce the number of equivalence checks by pushing transformations on implementations to the left, and ones on specifications to the right:



The first step to arrive at the picture above is to eliminate S_{K_1} in the previous picture. This is shown in the picture below. It is possible to construct transformations $\zeta^{i'}$, and $\gamma^{s'}$ from ζ^i, and γ^s so that

1. if $\zeta^{i'}(M) = \gamma^{s'}(O)$, there is a finite state machine S_{K_1} such that $\gamma^s(S_{K_1}) = M$ and $\zeta^i(S_{K_1}) = O$, implying that if $\zeta^{i'} \circ \gamma^i(S_F) = \gamma^{s'} \circ \zeta^s(S_{K_2})$, S_F is in $\zeta \circ \gamma$ relation with S_{K_2}, and

2. if S_F is in $\zeta \circ \gamma$ relation with S_{K_2}, then $\zeta^{i'} \circ \gamma^i(S_F) = \gamma^{s'} \circ \zeta^s(S_{K_2})$.



Appendix A shows how the transformations $\zeta^{i'}$ and $\gamma^{s'}$ can be derived. It also shows how S_{K_2} and S_{K_3} can be eliminated. This leads to the following result.

Theorem 4.7 *The fixed order composite relation can be verified with one equivalence check.*

Circuit	bits	latches	time
parallel adder	3	13	1 s.
versus serial	8	28	53 s.
pipelined minmax	3	30	20 s.
versus unpipelined	4	39	410 s.
pipelined minmax	3	38	8 s.
versus 2 cycle block	8	74	3503 s.(*)
stuttering CPU	3	27	4 s.
versus unpipelined	8	42	80 s.
pipelined filter	3	15	2 s.
versus unpipelined	8	40	133 s.

(*) obtained by verifying outputs separately

Table 4.1: Performance of the verification algorithm

4.4 Implementation and Results

I implemented the verification of string function relations within `misII` [8], extended with the FSM equivalence checking procedures described in [43]. These procedures take two FSM's, construct the product machine (one that produces an output of 1 as long as both FSM's agree on a given input), and perform a breadth-first search of the product machine, applying implicit enumeration based on BDDs.

I wrote a new command in `misII` that takes a logic-level specification, a logic-level implementation, and a description of the relation between the implementation and the specification. The program performs the necessary circuit transformations and applies an FSM equivalence algorithm.

Table 1 shows some experimental results. Five implementation/specification pairs were verified. For each of them, a 3 bit version was verified, as well as an 8 bit version (or the largest bit size that can be verified, if an 8 bit version is too large). For the larger bit sizes, variable orderings for the BDDs were done manually. Run times were measured on an IBM RS-6000 model 540. They were obtained with the `misII` *time* command, and include system time. The number of latches shown in the table is for the product machine.

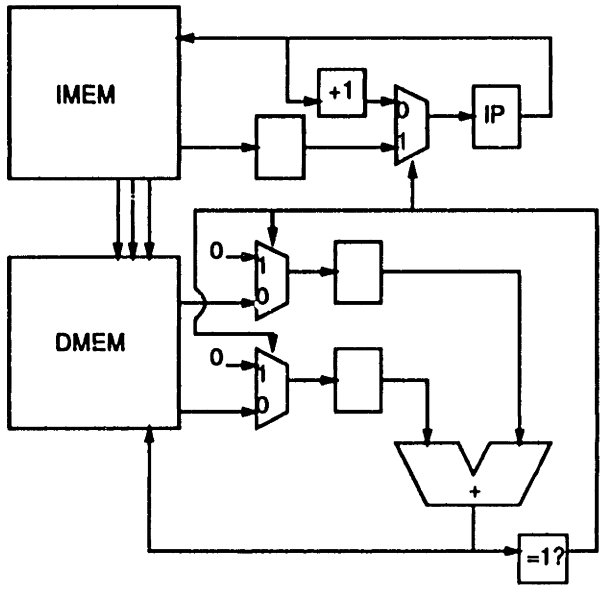


Figure 4.1: CPU implementation

The first implementation/specification pair consists of a parallel adder and the corresponding bit-serial adder, related by the γ-relation. The second pair consists of a pipelined minmax block and an unpipelined one, related by $\beta_{one,1}$ (the implementation is delayed by one clock cycle with respect to the specification). The minmax circuit has been used as a benchmark in a recent workshop on formal methods for correct VLSI design [45]. The output function of this circuit contains a comparator feeding an adder, which leads to large BDDs. This problem is bypassed in the third implementation/specification pair by putting latches between the comparator and adder.

The β-relation was also verified between a simplified, stuttering CPU, and the corresponding unpipelined CPU. The implementation and specification are shown in Figure 4.1 and Figure 4.2 respectively. Both circuits interact with an instruction memory (IMEM) and a data memory (DMEM). The instruction memory outputs instructions which consist of a branch target and addresses of argument data and result data. The instruction and data memories and their direct connections are not included in the verification.

In the implementation, the path from the instruction pointer (IP) back to itself over the

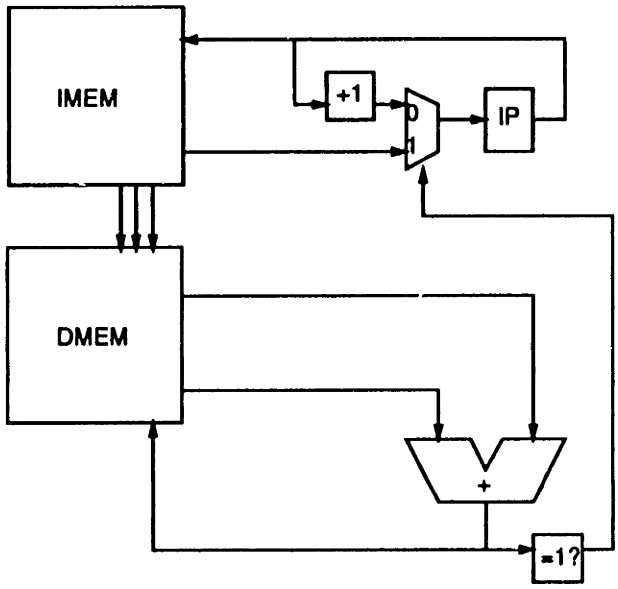


Figure 4.2: CPU specification

instruction memory has one delay more than the path over the incrementer. This means that the next instruction pointer is predicted to be the current one incremented by one, and that this prediction is confirmed or rejected at the next cycle. If rejected, there is an invalid output for one cycle. This is captured by an H-function which is derived from the node which signals whether to branch or not.

The last example in the table is a second-order bandpass filter. The implementation is pipelined with respect to the specification, and is in β-relation with it. In this case, the H-function is a modulo-2 counter: every second input and output values are relevant.

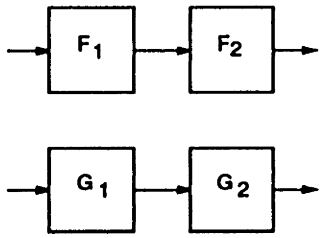
4.5 Conclusion

This chapter presented a sound and complete proof method for verifying arbitrary compositions of five primitive relations between arbitrary synchronous logic circuits. The proof method is practical for small to medium-sized circuits only, consisting typically of a few registers and a few logic or arithmetic units. It is not by itself applicable to the verification of realistic industrial designs. It is clear that, for large designs, a compositional verification

method is needed, where state space traversals are applied to restricted portions of the circuit, and not to the entire circuit.

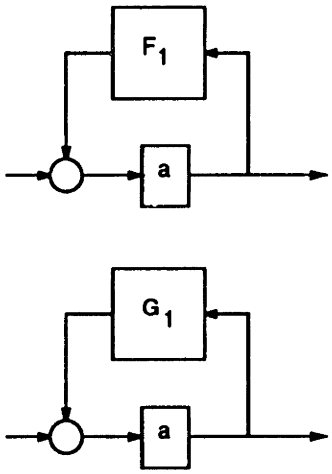
One possible compositional method consists of verifying relations between components of the implementation and the specification, and deriving a conclusion as to the relation of the overall implementation and specification, according to theorems such as the following:

Theorem 4.8 *Given:*



$F_1\ \alpha_{z_1}\ G_1$ *and* $F_2\ \alpha_{z_2}\ G_2 \Rightarrow F\ \alpha_{z_2.z_1}\ G$

Theorem 4.9 *Given:*



G_1 *combinational,* $F_1\ \alpha_z\ G_1 \Rightarrow F\ \beta_{H_d,0}\ G$ *where* H_d *picks out the values* $1 + nd$, *where* $n = 0, 1, \ldots$, *and* $d = \mid z \mid +1$.

In both cases a conclusion about the relation between the entire circuits can be reached from the validity of certain relations between components. If a verification method can be developed along these lines, it seems to be suited to implementation/specification pairs with

roughly the same amount of hardware. A possible application domain is the verification of microprocessor designs against earlier designs, or against a design which has a desired functionality but inadequate performance.

This direction is not further pursued in this thesis. Instead, an event-based compositional method is developed which allows for the verification of digital signal processors against signal flow or dependency graphs. The graphs, interpreted as synchronous logic machines, typically contain much more hardware than the implementation.

Chapter 5

Event-based Compositional Verification of the β-relation

5.1 Introduction

The previous chapter provides interesting results from a theoretical perspective. The proof method presented there is sound and complete, and can be applied to arbitrary synchronous implementation/specification pairs. The input/output behaviors of the implementation and the specification are compared with each other, with no assumptions about the internal structure of the computation.

From a practical standpoint, this proof method is not yet efficient enough for the verification of industrial designs. This is because the state space of the entire implementation and specification is traversed. This state space has a size exponential in the number of latches of the implementation and the specification.

The size of the circuit is less critical when theorem proving techniques are applied for verification, as opposed to exhaustive state traversal methods. Several large circuits have been successfully verified with these techniques [31] [19] [33]. However, these techniques require extensive user interaction.

In this chapter I describe a verification method which is both automatic and applicable to large circuits. These two features are obtained at the expense of restricting the method to a certain class of implementation/specification pairs. The specification needs to be supplied in

the form of a signal flow graph (containing state variables) or dependency graph (containing no state variables), consisting of operations and arcs between operations. Such a graph can be considered as a maximally parallel logic circuit. The implementation has to execute the same set of operations as the specification (possibly in a different order, with a different degree of parallelism, and with a different latency) to realize a comparable input/output behavior. This allows for an event-based compositional strategy in which the data path modules in the implementation are verified combinationally against the operations in the specification, and the control circuitry of the implementation is checked to see if it initiates events in the data path which are consistent with the flow of information realized by the specification.

The designs I consider in this thesis are systems with globally timed control. In a system with globally timed control the timing of each operation (for example when it starts and finishes) is fixed in a static, system-wide scheme ([46], p. 172). The time taken by each operation is viewed as a constant and designed into the system timing. Typically the control circuitry of a globally timed system is localized and constitutes an identifiable subsystem. In contrast, locally timed (or self-timed) systems allow each module to control the time it spends on individual operations by means of additional control signals. I believe that the definition of the overall correctness requirement and the event-based compositional verification method outlined in this chapter can be adapted to systems with locally timed control as well, but there are no results in this thesis to support this claim.

Two types of architectures with globally timed control are considered in this thesis. The first one consists of microcoded architectures, composed of a data path and a single controller containing a program counter and a microcode ROM. The controller dictates the sequence of operations in the data path. The second type of architecture consists of array processors of the Single Instruction Multiple Data (SIMD) and systolic varieties. These systems have globally timed control because there is a fixed system-wide scheme for the timing of operations, but this scheme is not necessarily captured by control logic, as is the case with microcoded processors. Control may reside in the data stream, as will be explained in a later section. However, it is always possible to construct additional control logic, solely

for verification purposes, which captures the timing of operations. The additional control logic can be constructed from information generated during scheduling in the design process. This point will be further elaborated upon in a later section.

The control logic, present in the implementation or constructed from scheduling information, dictates a sequence of registers transfers (RT's) in the data path of the implementation. The information regarding which vector of control signals in the implementation realizes a register transfer corresponding to a given operation in the specification, can be derived from information generated during the design process. Thus, once it is established that the data path modules have a correct functionality (which can be done with combinational verification techniques), attention may be restricted to the controller.

The controller can be verified by checking a collection of past-tense Computation Tree Logic (CTL) formulae on it, which are provably sufficient for correctness. Each formula expresses constraints on events in the implementation, such as register transfers. We have found CTL to be a convenient logic for describing the correctness properties, but other propositional temporal logics could also have been used.

Proving that the CTL formulae are sufficient is done manually, at "compile time", i.e. prior to the verification of any concrete design. Once this manual exercise is completed, a program can be written which automatically generates CTL formulae expressing sufficient correctness conditions, and automatically verifies these formulae according to algorithms described in Chapter 3.

The verification strategy outlined in this chapter has been used for the independent verification of circuits produced by the CATHEDRAL-II synthesis system, described in [28]. This system produces, under user interaction, microcoded digital signal processors, starting from a specification in the SILAGE language [30], which essentially describes a signal flow graph (SFG). The implementation can be verified against an intermediate SFG, which is an expansion of the original specification such that each operation in the new SFG corresponds to a register transfer in the implementation. CATHEDRAL-II produces such an expanded SFG. In such an SFG, complex arithmetic operations such as multiplications are decomposed into simpler ones, such as shifts and additions, and new operations are

introduced for maintaining iteration indices and computing addresses of memory locations.

A similar strategy has been applied to a number of systolic array processors, designed from dependency graphs. It is again required that the granularity of the dependency graph be fine enough such that operations in the specification correspond to register transfers in the implementation.

The remainder of the chapter is organized as follows. Section 5.2 presents a modified definition of the β-relation. Section 5.3 outlines the overall verification strategy, and the assumptions made, for the verification of microcoded processors. In Section 5.4 sufficient correctness conditions for microcoded processors, expressed as past-tense CTL formulae, are derived. Section 5.5 presents implementation details and results for the verification of microcoded processors. Section 5.6 contains extensions to the verification strategy for the verification of array processors. Section 5.7 concludes the chapter.

5.2 A Modified Definition of the β-relation

A microcoded processor performs an initialization sequence to set certain state variables to correct initial values, and then iteratively performs a set of operations on vectors of input samples, producing vectors of output samples. While the specification takes and produces these vectors in parallel, the implementation may process them serially. At every iteration, it typically takes the components of the input vector one by one, and produces the elements of the output vector one by one. The width of the input vector may differ from that of the output vector. This calls for a modification of the β-relation. A composition of the β-relation as defined in Chapter 2 with the γ-relation defined there, is inadequate since it does not cover different degrees of parallelism for the input and output vectors.

Recall the definition of the *Relevant*-function:

Definition 2.1 $Relevant : \Sigma^* \underline{\times} B^* \rightarrow \Sigma^*$:

$$
\begin{aligned}
Relevant(\varepsilon, \varepsilon) &= \varepsilon \\
Relevant(x.u, y.v) &= Relevant(x, y) \quad , v = 0 \\
&= Relevant(x, y).u \quad , v = 1
\end{aligned}
$$

The symbol $\underline{\times}$ represents the string cartesian product, which combines strings of equal length. Recall also that $R_{0\uparrow n}$ is the string function associated with a concatenation of n registers with initial value 0. Let mod_n be a Boolean valued string function which returns a value of 1 at cycles $1 + nN$, $N = 0, 1, 2, \ldots$, irrespective of the input string. The new β-relation can be defined as follows:

Definition 5.1 *Let $i_1 \ldots i_I$ and $o_1 \ldots o_O$ be positive integers. Let H be a delayed modulo counter: $R_{0\uparrow n'} \circ mod_n$, where $n \geq M$ and $M = MAX(i_1, \ldots, i_I, o_1, \ldots, o_O)$.*

$F\ \beta_{H;i_1 \ldots i_I;o_1 \ldots o_O}\ G \Leftrightarrow \forall x \in \Sigma^*$ *s.t.* $L(R_{0\uparrow M} \circ H(x)) = 1$,

$Relevant(F(x), R_{0\uparrow o_1} \circ H(x)) \underline{\times} \ldots \underline{\times} Relevant(F(x), R_{0\uparrow o_O} \circ H(x))$

$= G(Relevant(x, R_{0\uparrow i_1} \circ H) \underline{\times} \ldots \underline{\times} Relevant(x, R_{0\uparrow i_I} \circ H))$

The periodicity n of the modulo counter should match with the length of every iteration in the implementation. The reason the modulo-n counter may be delayed is that the implementation may perform an initialization sequence before processing the first input data. The integers $i_1 \ldots i_I$ should mark the time points (relative to the beginning of every iteration, marked by H), at which the inputs are presented to the circuit. Likewise for $o_1 \ldots o_O$ marking output events.

The identity in the definition is required when the last character of $R_{0\uparrow M} \circ H(x)$ has a value of 1. At these time points all strings involved in the string cartesian products have the same length. After an initialization sequence of length n', I inputs are taken and O outputs produced. This is repeated every n clock cycles. At time points where $R_{0\uparrow M} \circ H(x) = 1$, all inputs in one sample period have been taken in, and all outputs of that same sample period produced, so that all the *Relevant*-operations involved produce strings of equal length.

The new β-relation defined here is a generalization of the one defined in Chapter 2 (Definition 2.2). The example implementation/specification pair shown in Figure 2.1 is also related by the new β-relation. This pair can be verified with the non-compositional techniques discussed in Chapter 4, because its state space is not to large. However, for bigger circuits a compositional technique is imperative.

The β-relation is geared to systems with globally timed control, where the timing of the

computations is fixed in a static, system-wide scheme, so that it is known ahead of time when inputs are taken and when outputs are produced. The timing of input events and output events is not known ahead of time for systems with locally timed control (self-timed systems). These systems take whatever time they need to finish a computation, and take a new input when they are ready for it. The definition of the β-relation would have to be slightly modified for these systems.

The following transitivity property of the β-relation will be needed for my verification strategy.

Theorem 5.1 $F\ \beta_{R_{0\uparrow n'}omod_n;i_1\ldots i_I;o_1\ldots o_O}\ L\ \wedge\ L\ \beta_{R_{0\uparrow m'}omod_m;j_1\ldots j_J;k_1\ldots k_K}\ G$

$\Leftrightarrow\quad F\ \beta_{R_{n'+nm'}omod_{nm};S_1;S_2}\ G$

where $S_1 = i_1 + nj_1, \ldots, i_I + nj_1,\ \ldots\ , i_1 + nj_J, \ldots, i_I + nj_J$

and $S_2 = o_1 + nk_1, \ldots, o_O + nk_1,\ \ldots\ , o_1 + nk_K, \ldots, o_O + nk_K$

The proof of this theorem can be found in Appendix A.

5.3 Microcoded Processors: Strategy and Assumptions

This section and the following two sections deal with the verification of microcoded processors. The last section in this chapter contains extensions for the verification of array processors.

The specification is taken to be a signal flow graph, i.e. a directed graph with combinational functions as nodes, and possibly with delays on edges. This graph can be viewed as a synchronous machine with maximally parallel hardware. A verification procedure is desired that establishes the β-relation between the implementation and the specification. It is useful to require that the implementation not only exhibits the same input/output behavior as the specification, but also performs the same set of operations (possibly in a different order, with a different degree of parallelism, and with a different latency). This additional requirement makes the verification problem computationally easier. It allows for an event-based compositional verification strategy, which consists of verifying, first, that the

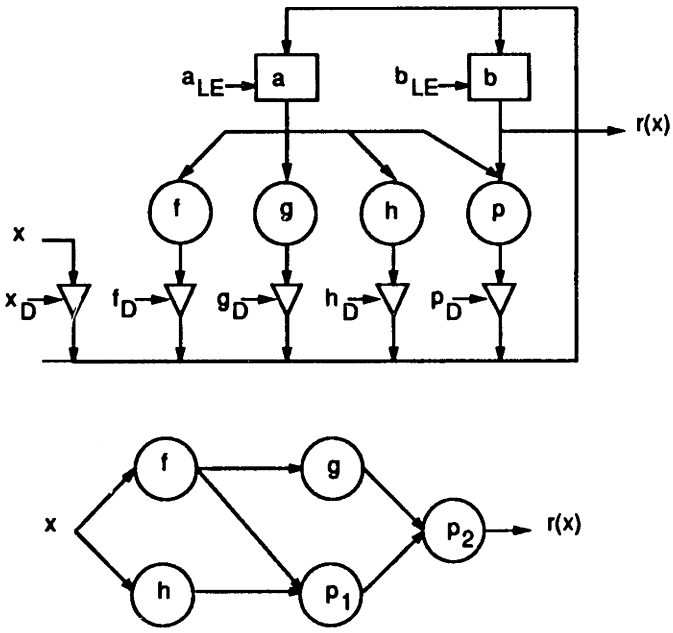


Figure 5.1: Example of an implementation/specification pair which can be verified with the compositional method

combinational data path modules have the right functionality, and second, that the flow of information between the combinational blocks in the implementation is in accordance with the connections in the specification.

A typical application for this compositional method is digital signal processing (DSP). A simple example where the method applies is the implementation/specification pair shown in Figure 5.1. The specification, shown below, takes the form of a signal flow graph and can be viewed as a synchronous machine that takes a new input at every cycle and computes the corresponding output at the same cycle. The implementation, shown above, needs 6 clock cycles for the same computation. It performs the same operations as the specification, but one by one.

DSP algorithms are often described in applicative languages such as SILAGE ([30]). Appendix B shows an example of a SILAGE specification. SILAGE programs are similar to signal flow graphs. Although a SILAGE program can contain if-then-else constructs or iterative loops, these constructs do not really represent control. The iterative loops are to

be interpreted as shorthands for the corresponding expanded expressions. For example, a description of a 256-point DFT algorithm which iterates over 256 points in the time-domain, really represents a parallel network taking the 256 time-domain inputs all at once. A serial implementation should be in β-relation with the specification.

In case the specification contains an iterative loop and the implementation is serial, the implementation will contain data path hardware for storing and incrementing the iteration index, which is not contained in the specification. Iteration indices appear in the specification as suffixes of signal names, but do not materialize as signals themselves. Therefore, there will be operations in the implementation that do not correspond to any operation in the specification. Thus, the compositional method consisting of the combinational verification of data path modules and the verification of conditions on possible input/output sequences of the controller, will not do to directly verify an implementation against a specification. What is needed is an intermediate circuit which, like the specification, contains maximal data path hardware and no control, but which is serial, like the implementation. In the CATHEDRAL-II synthesis environment, this intermediate circuit is generated in the form of a list of untimed register transfers (RT's), constructed from the specification after allocation of data path hardware. Appendix B contains an example of such an expanded SFG.

Another set of operations which can be found in the implementation and the intermediate SFG, but not in the original SFG, consists of address computations for signals that are stored in RAM's, as opposed to register files in the data path. In CATHEDRAL-II circuits, these computations are performed by an address computation unit (ACU), which also maintains iteration indices. For the expanded SFG to be in β-relation with the original specification, the address computations should be such that, if the precedence constraints inherent in the expanded SFG are satisfied, signals are correctly communicated (i.e. are not corrupted between production and consumption).

One more difference between the intermediate circuit and the original specification is that arithmetic operations might be expanded into simpler operations, e.g. a multiplication into shifts and additions.

This chapter will be concerned with proving the β-relation between the implementation and the circuit corresponding to the expanded SFG. This constitutes a verification step across scheduling and optimal controller synthesis. A remaining proof obligation, not addressed in this thesis, is that the expanded SFG is in β-relation with the original specification. From the transitivity theorem for the β-relation in Section 5.2 it follows that the validity of β between the implementation and the expanded SFG, and between the expanded SFG and the original specification, implies the validity of β between the implementation and the original specification.

To prove the β-relation between the implementation and the expanded SFG, it has to be demonstrated that:

1. all combinational blocks in the specification (except selectors corresponding to if-then-else constructs) have an equivalent block in the data path of the implementation, and,
2. the flow of information between the data path modules in the implementation is in accordance with the connections in the expanded SFG.

From here onwards it is assumed that the first condition is satisfied. It can be verified with combinational equivalence checking procedures. It is possible that certain control inputs to a module in the implementation have to be set for it to be equivalent to an operation in the specification. This is the case for an ALU module. The module then has to be compared to the corresponding operation in the specification under the application of the appropriate control signals. In a design methodology where data path modules are taken from a library, the verification of these modules can be done once, when they are added to the library.

It is furthermore assumed that every pair of combinational operators (f, g) that is directly connected in the specification, is connected in the implementation via a multiplexer, a demultiplexer, a register file, and again a multiplexer and demultiplexer, as shown in Figure 5.2, except for connections in the specification between memory writings and readings. Buses can be viewed as multiplexers, and the addressing portion of a register file as a demultiplexer. The verification system is provided with information as to how each pair of blocks is connected in the implementation. In Figure 5.2 the subscript EX stands for

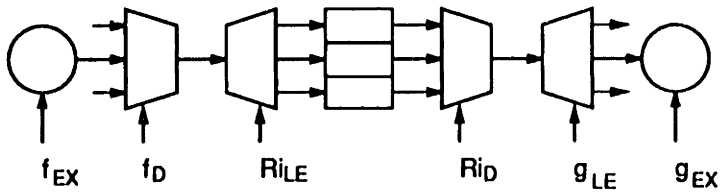


Figure 5.2: Connection between combinational blocks in the implementation

"execute", D for "drive", and LE for "load enable". R_i is the particular register through which the two operators communicate. The control signals f_{EX} and g_{EX} exist only if the corresponding modules can perform different functions (e.g. ALU's). One or both of the multiplexer/demultiplexer pairs may be absent. There may even be a direct connection between f and g – in that case no work has to be done to verify correct flow of information. In case g takes more than one input from f, the symbols f_D, R_{iLE}, R_{iD} and g_{LE} will denote conjunctions of several control signals. For an interconnection in the specification between a combinational block and a register, only the portion between the f-block and the register file is relevant; for an interconnection between a register and a combinational block, only the portion between the register file and the g-block is relevant.

For memory writings and readings, the address computations should be such that, if the precedence constraints in the expanded SFG are satisfied, signals are correctly communicated (and not corrupted between production and consumption). This has to be true for the expanded SFG to be in β-relation with the original SFG. When verifying the implementation against the expanded SFG, it only has to be ensured that the precedence constraint is satisfied. This will happen when we ignore R_{iD} and let R_{iLE} have a value of 0 all the time.

5.4 Microcoded Processors: Sufficient Conditions

This section presents sufficient conditions for the implementation and the expanded specification to be in $\beta_{H;i_1 \ldots i_I;o_1 \ldots o_O}$-relation. These conditions are specified on the controller – the data path modules are assumed to be correct. H is of the form $R_{0\restriction n'} \circ mod_n$, and n',

n, $i_1 \ldots i_I$ and $o_1 \ldots o_O$ are specified by the user.

5.4.1 No If-then-elses, No Iterative Loops

Consider first the case without if-then-elses or iterative loops in the specification. For every operation f in the specification, we define an event f (overloading the symbol) as the conjunction of control signals which leads to the register transfer corresponding to operation. In other words, if f takes its input from register R_j and puts its result in R_i, the event f is defined as $R_{j_D} \wedge f_{LE} \wedge f_{EX} \wedge f_D \wedge R_{i_{LE}}$. If f takes more than one input, or produces more than one output, R_{j_D}, f_{LE}, etc. are conjunctions of multiple control signals. For input events (output events), control signals for reading signals from input pads (writing signals to output pads) are needed, rather than control signals for accessing registers. Every operation in the specification should have a distinct conjunction of control signals associated with it. If necessary, conditions on the program counter should be added.

Let $in_1 \ldots in_I$ be the input events, and $out_1 \ldots out_O$ the output events. First we require, for all j between 1 and I, and for all k between 1 and O:

$$R_{0\uparrow i_j} \circ H \Leftrightarrow in_j \tag{5.1}$$

$$R_{0\uparrow o_k} \circ H \Leftrightarrow out_k \tag{5.2}$$

Another set of requirements is as follows. Let f and g be adjacent combinational functions in the specification graph, with f feeding g. Let R_i be the register (registers) through which f and g communicate. For every pair (f, g), we require:

$$g \Rightarrow AX^- A(\neg R_{i_{LE}} U^- f) \tag{5.3}$$

This formula expresses that when g is fired, it should be true that R_i, from which g gets its inputs, has not been corrupted since f was fired and wrote a value in it.

These conditions are almost sufficient for correctness. The expanded specification graph contains logic for computing the initial values of state registers (from constants), and logic

for computing, from inputs and present states, outputs and next states. Let $X = (x_j)_{1 \leq j \leq I}$ be the vector of inputs to the specification graph, and $Y = (y_k)_{1 \leq k \leq M}$ the vector of intermediate points and outputs in the specification. Over time, these vectors form sequences $(X^l)_{l=1,2,...}$ and $(Y^l)_{l=1,2,...}$. A subset Y' of Y contains state variables, which have an initial value Y'^0. The specification graph shows how Y'^0 is computed from constants, and how Y'^l is computed from X^l and Y'^{l-1}.

All the elements in these vectors have corresponding nodes in the implementation. Given a way of discriminating relevant time points, the values of these nodes at the relevant time points form sequences $(X^l)_{l=1,2,...}$, $(Y^l)_{l=1,2,...}$ and $(Y'^l)_{l=0,1,2,...}$. The relevant time points for the outputs in Y are discriminated by $R_{0\uparrow o_k} \circ H$. Let the relevant time points for the other nodes be determined by formula (5.3), as follows. Given a relevant time point for the result of g, a relevant time point for the input to g from f is the last firing of f prior to the firing of g, where it is understood that the contents of the register where the value is put does not change between the firing of f and the one of g. This results in the sequences $(X^l)_{l=1,2,...}$, $(Y^l)_{l=1,2,...}$ and $(Y'^l)_{l=0,1,2,...}$. It follows from formula (5.3) that for any $l \geq 1$, Y^l is a correct function of X^l and Y'^{l-1}, and that Y'^0 is correctly computed. It remains to be shown that the relevant time points for the inputs, induced by formula (5.3), are identical to the ones marked by $R_{0\uparrow i_j} \circ H$. It is still possible, for instance, that the first few operations in the specification graph fire once only, and that the subsequent operations work from the never-changing results of these first operations.

To preclude this situation, we require (for all pairs (f, g) of operations that are adjacent in the specification):

$$g \Rightarrow AX^- A(\neg g U^- f) \tag{5.4}$$

This says that if g fires, going back in time, g should not have fired another time since f fired. This makes it possible to prove that, for arbitrary j, k, and l, the input event for input j induced by formula (5.3) from the l^{th} output event for output k, coincides with the l^{th} firing of $R_{0\uparrow i_j} \circ H$. Assume this is not the case. We know from (5.1) that input events for input j occur at firings of $R_{0\uparrow i_j} \circ H$. We also know that an input event induced via (5.3)

from an output event occurs prior to the output event. Therefore, the only way to have a mismatch between the induced and the correct input event, is that different output events lead, via (5.3), to the same input event. But then condition (5.4) is violated somewhere along the path from the output event to the input event. Thus the induced input event has to coincide with the correct input event.

The verification of conditions (5.1) through (5.4) can be accelerated in a number of ways. Direct verification of conditions (5.1) and (5.2) requires constructing a delayed modulo-n counter, and composing it with the controller of the implementation. This can be avoided as follows. Assume that all of the i_j's are smaller than all of the o_k's. It will be clear how to modify the argument if this is not the case. The following requirements are equivalent with (5.1) and (5.2):

$$in_1 \Leftrightarrow (EX^-)^{n'+i_1} init$$
$$in_2 \Leftrightarrow (EX^-)^{i_2-i_1} in_1$$
$$\ldots$$
$$in_I \Leftrightarrow (EX^-)^{i_I-i_{I-1}} in_{I-1}$$
$$out_1 \Leftrightarrow (EX^-)^{out_1-in_I} in_I$$
$$\ldots$$
$$out_O \Leftrightarrow (EX^-)^{out_O-out_{O-1}} out_{O-1}$$
$$in_1 \Leftrightarrow (EX^-)^{n-out_O+i_1} out_O$$

Proposition *init* is true in the initial state of the controller. The integer n' is the delay of the modulo counter represented by H (cf. the definition of the β-relation). $(EX^-)^m$ is a shorthand for a concatenation of EX^-, m times. $(EX^-)^m f$ is computed by performing m image-computations, starting from the set of states in which f is true. Thus, the first formula, for instance, says that $n' + i_1$ cycles after the initial state, the controller should be in a state in which it fires in_1, and in_1 is fired in no other state than one that can be reached in $n' + i_1$ cycles from the initial state. Taken together, the formulae above are equivalent with the combination of (5.1) and (5.2).

Formulae (5.3) and (5.4) can be combined into:

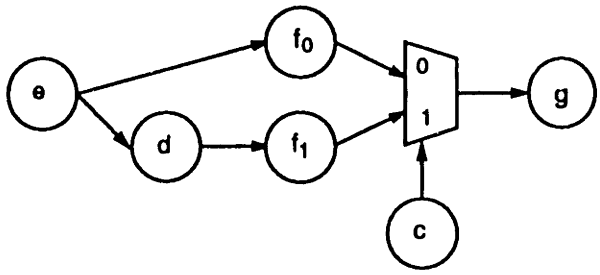


Figure 5.3: Decision making in the specification

$$g \Rightarrow AX^{-}A(\neg(R_{iLE} \vee g)U^{-}f) \tag{5.5}$$

This condition is definitely satisfied if $g \Rightarrow AX^{-}f$, which is easier to verify, because it requires only one image computation. Therefore, the formula above is only verified if the simpler one evaluates to false. The simpler formula evaluates to true quite frequently, as adjacent operations in the specification graph often map to register transfers at consecutive control steps.

5.4.2 If-then-elses

Consider now the case with if-then-elses in the specification. In systems with globally timed control, the two branches take an equal number of clock cycles in the implementation. Otherwise, the time it takes to process an input sample would be data-dependent, implying locally timed control.

While selection is performed by the data path in the specification (Figure 5.3), we assume it is performed by the controller in the implementation. We assume that the controller not only performs the selection, but also computes the Boolean function c from the values of status registers in the data path at various time points. Let c_{0ij} and c_{1ij} be the operations that produce these values in the status registers. The meaning of the subscripts is as follows. For the selection of operation f_0, it has to be true that, for all values of i, there is a value of j such that c_{0ij} fired with a result $cond_{0ij}$. Likewise, for the selection of f_1, it has to be true that, for all values of i, there is a value of j such that c_{1ij} fired with a result $cond_{1ij}$.

This corresponds to product of sums expressions. For example, when f_1 has to fire if a and b fired with a result 1, the following assignment would take place: $c_{111} = a$, $cond_{111} = 1$, $c_{121} = b$, $cond_{121} = 1$, $c_{011} = a$, $cond_{011} = 0$, $c_{012} = b$ and $cond_{012} = 0$.

In order to make conditions $cond_{0ij}$ and $cond_{1ij}$ available in CTL formulae, which are built from atomic propositions expressing conditions on *states*, the inputs to the controller (connected to status registers in the data path) are latched into flip-flops which are added to the controller for verification purposes. $cond_{0ij}$ and $cond_{1ij}$ are conditions on the values of these flip-flops (i.e. on the values of the inputs to the controller at the previous clock cycle).

The computations c_{0ij} and c_{1ij} have an effect on the values of the input latches after a number of clock cycles. Assume there are q pipeline stages in loops around the controller and the data path. To keep the formulae simple, assume that exactly one of these cuts through the controller. For example, in CATHEDRAL-II processors every loop around the controller and the data path is cut by the instruction register, a status register, and the program counter (therefore $q = 3$). Only the program counter cuts the controller. The instruction register is considered part of the data path. Including it in the controller would lead a larger state space while making no difference as to correctness – all events would be translated uniformly in time.

This means there is a delay of q clock cycles between the firing of an event c_{0ij} at the controller output and the time when the corresponding input latch contains the result of the computation. In the CATHEDRAL-II case, a firing of c_{0ij} at the controller output, leads to a corresponding register transfer at the next clock cycle, and a loading of the controller input latch two cycles later, so that the input latch contains the result after three clock cycles.

For edges (e, f_0), (e, d) and (d, f_1) in Figure 5.3 the requirements are just as before. Edges (f_0, g) and (f_1, g) can be combined by defining a global event f as $f_0 \vee f_1$. Conditions (5.3) through (5.4) can then be applied to this new edge. The only requirements left are that the controller selects correctly one of f_0 and f_1, and that the register transfers producing the status bits fire frequently enough. This can be expressed as follows:

$$f_0 \Rightarrow \bigvee_j (((AX^-)^q c_{0ij}) \wedge cond_{0ij}) \vee$$
$$AX^- A(\neg f U^- \bigvee_j (((AX^-)^q c_{0ij}) \wedge cond_{0ij})) \quad (5.6)$$

One such condition is needed for every i. Similar conditions are needed for f_1. $(AX^-)^q$ is short for a concatenation of (AX^-), q times. The second term in the implication is similar to the implication in (5.4), with the difference that pipelining has to be taken into account, and that it has to be checked that one of the events c_{0ij} fired with the right outcome. The first term in the implication is necessary for the following reason. Assume that f_0 fires, which is justified by a previous firing of c_{0ij} with outcome $cond_{0ij}$ (for one particular j). The latest time point at which c_{0ij} could fire is q cycles before the firing of f_0. This case is not covered by the second term in the implication (which requires c_{0ij} to have happened at least $q+1$ clock cycles ago). It is covered by the first term.

The proof from the previous subsection can easily be adapted to demonstrate that the conditions above are sufficient for the implementation to be in $\beta_{H;i_1 \ldots i_I;o_1 \ldots o_O}$-relation with the expanded specification. Condition (5.3) combined with (5.6) implies again that the outputs at times $R_{0\uparrow o_k} \circ H$ are correct functions of certain inputs. It remains again to demonstrate that these inputs are exactly the ones arriving at $R_{0\uparrow i_j} \circ H$. As before, the only way this could not be true is that outputs at different times are computed from the same inputs. But that is impossible due to (5.4) and (5.6).

While the addition of formula (5.6) leads to *sufficient* correctness conditions in the case of if-then-elses, the formula may be too strong in certain situations. This is the case if a processor correctly chooses to execute an operation f_0, but is oblivious as to why that decision is taken. It jumps correctly to the right state, but once in that state there is no way to tell how it came to it. Say an operation c_{0ij} was executed q cycles ago with an outcome justifying selection of f_0. If, when f_0 is executed, the system has no memory of where it came from, $((AX^-)^q c_{0ij}) \wedge cond_{0ij}$ will not be true, and condition (5.6) will be violated. This problem occurred once in the experiments, and was remedied by adding latches to the controller for remembering whether or not c_{0ij} occurred q clock cycles ago. This leads to a

splitting of the states in which f_0 fires, where in each state it is known whether or not $c_{0;j}$ happened q clock cycles earlier.

5.4.3 Iterative Loops

In systems with globally timed control, loop counter increments and loop exits are data-independent. If they were data-dependent, the time it takes to process an input sample would be data-dependent, implying locally timed control.

If all operations in an iterative loop are distinguished between (e.g. an operation f at iteration 1 is distinguished from the corresponding operation at iteration 2), the correctness conditions derived so far are still sufficient for correctness. There is an if-then-else inherent in an iterative loop (if ..., perform an iteration, else, exit the loop), which is handled as indicated in the previous subsection, and all other operations are dealt with as before.

However, distinguishing between recurring operations in an iterative loop is problematic for two reasons. First, if the iteration is long it leads to a large number of operations, and therefore to a large number of conditions to be checked. Second, in certain circuits (such as the ones generated by CATHEDRAL-II), the iteration index is maintained by a data path module (e.g. an ACU), not by the controller. This data path module has to be added to the controller in order to get a handle on operations occurring at specific iterations, and a composition of the two subcircuits has to be traversed for the verification of CTL formulae referring to operations at specific iterations.

Fortunately, even when no distinction is made between corresponding operations at different iterations, the resulting CTL conditions are still sufficient for correctness. This can be seen as follows. The only thing that can go wrong when corresponding operations at different iterations are collapsed into one operation, is that a function g which takes inputs from a function f, takes inputs from an instance of f at the wrong iteration. Since this instance of f happens necessarily prior to g, there would be fewer occurrences of f than of g. But then condition (5.4) would be violated.

This argument is similar to the one made in Section 5.4. In fact, all the operations in a specification graph can be seen as forming one iteration in an implicit loop over different

input samples, which goes on indefinitely. We do not distinguish between such operations at different iterations. If all operations ever to be performed could be distinguished, requirement (5.3) (together with (5.1) and (5.2)) would be sufficient for correctness. Because we cannot (and do not want to) make this distinction, we add requirement (5.4).

There is a problem with collapsing operations at different iterations into one operation, in that the CTL conditions may become too strong. Assume operations $g[i]$ at different iterations all depend on the same operation f outside the iterative loop. After collapsing the $g[i]$ into one operation g, condition (5.4) (or (5.5)) is not satisfied any more. To solve this problem, we replace (5.5) with the following:

$$g \Rightarrow AX^{-}A(\neg(R_{iLE} \vee g)U^{-}m2) \tag{5.7}$$

$$m2 \Rightarrow A(\neg R_{iLE}Um1) \tag{5.8}$$

$$m1 \Rightarrow A(\neg(R_{iLE} \vee g)U^{-}f) \tag{5.9}$$

$m2$ is an event on the first cycle of the iteration. In CATHEDRAL-II controllers, this is usually an operation for updating the iteration index, which is not itself an operation that depends on an operation outside the iterative loop. $m1$ is an event on the last cycle before the iteration. In the formulae above the trajectory from g back to f is split into three parts. On the first part, from g back to $m2$, we require that R_i is not corrupted and that g does not occur. On the second part, from $m2$ back to $m1$, we require only that R_i is not corrupted. On the third part, from $m1$ to f, we require again that R_i is not corrupted and that g does not occur. The only difference with the previous requirement (5.5) is that g is allowed to occur between $m2$ and $m1$ (i.e. at previous iterations).

Condition (5.6) has to be replaced too in case f_0 happens at multiple iterations and the c_{0ij} before the iterative loop. The replacements are similar to the previous case:

$$f_0 \Rightarrow AX^{-}A(\neg fU^{-}m2) \tag{5.10}$$

$$m2 \Rightarrow A(1U^{-}m1) \tag{5.11}$$

$$m1 \Rightarrow A(\neg fU^{-} \bigvee_{j}(((AX^{-})^{q}c_{0ij}) \wedge cond_{0ij})) \tag{5.12}$$

Events $m1$ and $m2$ have to be qualified in this case so that they lead back to a firing of one of the c_{0ij} with the right outcome. If they are too coarse (covering paths not leading to a firing of one of the c_{0ij} with the right outcome), the conditions are too strong: condition (5.12) will be violated. Qualifying $m1$ and $m2$ can be done by ANDing with conditions on the controller state (in the CATHEDRAL-II case: conditions on the condition code register, which contains an encoding of relevant information from the history of inputs from the status registers in the data path). [1]

5.5 Microcoded Processors: Implementation and Results

The verification method described above was implemented and applied to a number of circuits generated with the CATHEDRAL-II synthesis system ([28]). CATHEDRAL-II takes specifications in the SILAGE language ([30]), which are essentially textual descriptions of signal flow graphs. These specifications are first transformed into a list of untimed register transfers (RT's), after allocation of data path hardware. The RT's are scheduled, and control logic is constructed consisting of two PLA's, a microcode ROM, intermediate registers, and a counter. I perform a verification step across the last two synthesis steps: scheduling and logic synthesis for the controller.

The following is an example of an RT occurring in the list of untimed RT's (local IMEC format).

```
j5:reg_file_2_alu_0 <-
  j8:reg_file_1_alu_0,
  j9:reg_file_2_alu_0
```

[1] The same condition will also be violated if one of the events c_{0ij} lies maximally close (i.e. q clock cycles) to f_0. (This did not occur in the experiments conducted for this thesis.) The formulae have to be modified to accommodate this case, and will be similar to formula (5.6).

```
| core_alu_0 = sub,
  sh_alu_0 = upsh[0,5],
  shmux_alu_0 = up,
  bus_2 = alu_0 <0> %18;
```

This RT states that the signal $j5$ in the signal flow graph is to be computed from $j8$ and $j9$ by applying certain control signals to the data path. These control signals can be derived from the information following the vertical bar (|), and from the identity of the source and destination registers. The RT only indicates register files in certain modules, but the exact registers can be found in other data structures generated by the synthesis system. From this information, an event $j5$ can be defined by adding the following to a copy of the control logic.

```
.names reg_file_2_alu_0_cw[2] reg_file_2_alu_0_cw[1] reg_file_2_alu_0_cw[0] \
reg_file_1_alu_0_cr[2] reg_file_1_alu_0_cr[1] reg_file_1_alu_0_cr[0] \
reg_file_2_alu_0_cr[2] reg_file_2_alu_0_cr[1] reg_file_2_alu_0_cr[0] \
core_alu_0_cinstr[1] core_alu_0_cinstr[0] sh_alu_0_sh[2] sh_alu_0_sh[1] \
sh_alu_0_sh[0] shmux_alu_0_cshupdo j5
100001100010000 1
```

This says that event $j5$ occurs whenever a certain condition on control signals is satisfied. In some cases conditions on the program counter have to be added in order to make all events distinct.

Similarly, events *alu*0_*rega*1 and *alu*0_*regb*4 can be defined, signifying writings in respectively register 1 of register file 1 in module *alu*0, and register 4 of register file 2 in *alu*0. The first of these is defined as follows.

```
.names reg_file_1_alu_0_cw[2] reg_file_1_alu_0_cw[1] \
reg_file_1_alu_0_cw[0] alu0_rega1
001 1
```

Besides constructing extended control logic from the untimed RT's, it is also possible to derive a flowgraph description suited for verification purposes. The RT above leads to the following entries in such a flowgraph description.

```
.arc j8 j5 alu0_rega1
.arc j9 j5 alu0_regb4
```

In general these entries have the form

$$\texttt{.arc } f\ g\ R_{iLE}$$

and express the fact that event f is a direct predecessor of g, communicating over register R_i. In the above arc-description f, g and R_{iLE} have the same meaning as in formulae (5.3) and (5.4), which can now be verified for each arc. One additional condition that has to be checked is

$$f \Rightarrow R_{iLE} \tag{5.13}$$

because we know, from the way we defined g and R_{iLE}, that g will read from R_i, but we are not assured of the fact that f writes its result in that register.

Besides "arcs", the specification graph can also contain "cond_arcs", "iter_arcs" and "iter_cond_arcs". The first of these have the form

```
.cond_arc f0 f c1 cond1 c2 cond2 ...
```

and lead to the verification of condition (5.6). "iter_arcs" are specified with

```
.iter_arc f m1 m2 g r
```

and lead to the verification of formulae (5.7) through (5.9), together with (5.13). "iter_cond_arcs" are specified with

```
.iter_cond_arc f0 f m1 m2 c1 cond1 c2 cond2 ...
```

and lead to the verification of formulae (5.10) through (5.12). Finally, there are statements of the form

```
.delay h1 h2 n
```

which lead to verification of $h2 \Leftrightarrow (EX^-)^n h1$, as is needed for checking (5.1) and (5.2).

Two CATHEDRAL-II generated circuits were verified.

1. "rec3"

   This is a recursive filter consisting of three cascaded second-order stages followed by a line equalizer. The specification contains no if-then-elses or iterations. The data path of the implementation contains 3 14-bit ALU's and 30 14-bit registers. The circuit performs 70 RT's for each input sample. Verification was completed in *2 minutes and 10 seconds* on a SUN SPARC-Station 2. A number of errors were found, which could be traced back to bugs in the scheduling program.

2. "echo"

   This is an echo cancellor, the specification of which contains if-then-elses as well as iterations, leading to correctness conditions that require more time to verify than correctness conditions for specification graphs that do not contain these constructs. The data path of the implementation contains one ALU, one multiplier, two ACU's (all with a word-length of 14 bits), two RAM's (with 128 12-bit words and 256 14-bit words respectively), and 39 registers. The circuit performs 144 RT's for each input sample. Verification took *37 minutes* on a SUN SPARC-Station 2. Again, errors were found. Appendix B contains the SILAGE specification and the list of untimed RT's (i.e. the expanded SFG) for this circuit.

The errors found are of four types. The first type is corruption of register values between production and consumption. The second type is violation of a precedence constraint. The third type is incorrect jumping so that certain operations are skipped. The fourth type is insufficient delay between decision making operations and register transfers generating status bits for the decision making – the minimal delay corresponds with the number of pipeline stages in loops around the controller and the data path. These errors could easily be identified from the invalid CTL conditions and the functional dependence of the Boolean components of them (f, g, r, etc.) upon state variables in the controller (e.g. the value of the program counter). In "rec3" 3 arcs of the signal flow graph were found to be incorrectly implemented (i.e. they did not satisfy the CTL conditions). In "echo" 13 arcs are improperly implemented.

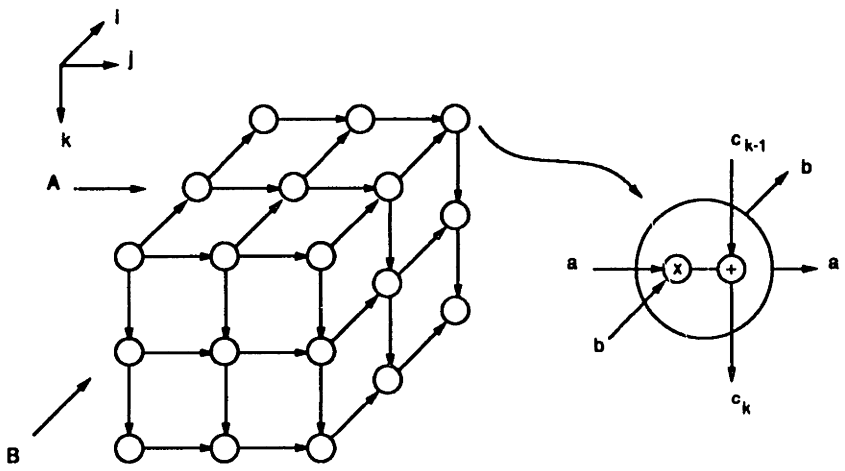


Figure 5.4: Dependency graph for matrix multiplication (internal vertices obscured)

The computation time needed for verification scales smoothly with the size of the circuit. The computation time is linear in the number of CTL conditions, which is linear in the number of register transfers for each input sample (roughly four CTL conditions per register transfer). The verification of each CTL condition takes, in the worst case, linear time in the number of states of the controller (but on average much less, since only a portion of the state transition graph has to be traversed, and since states are enumerated implicitly using BDDs).

5.6 Extensions for Array Processors

5.6.1 Introduction

The strategy of the previous sections has been extended to allow for the verification of array processors designed from dependency graphs (as in [34]) or from signal flow graphs. The difference between dependency graphs (DG's) and signal flow graphs is that the first are purely functional while SFG's can have state variables. Since, in the verification methodology above, state variables in SFG's are not treated differently from other variables communicated between two operators, both DG's and SFG's can be taken as a specification.

In [34] a methodology is presented for designing array processors from dependency

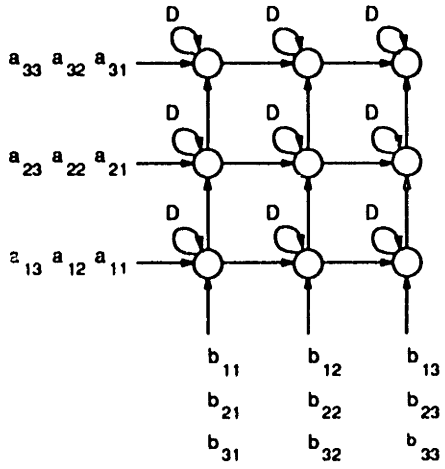


Figure 5.5: Signal flow graph for matrix multiplication

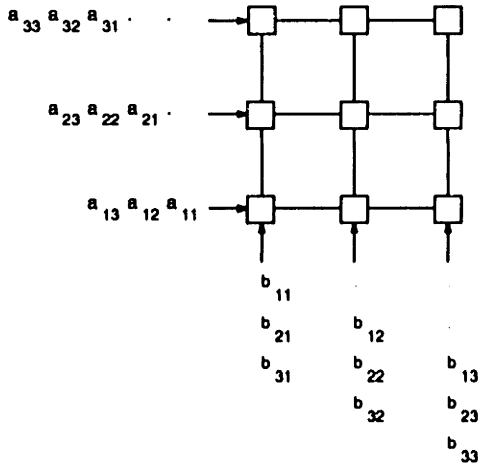


Figure 5.6: Systolic array processor for matrix multiplication

graphs. An example of a DG for matrix multiplication is shown in Figure 5.4. The graph is three-dimensional: internal nodes are obscured. The A matrix is presented at the left side plane and is passed on to the right. The B matrix is presented at the front plane and is passed on straight ahead. The vertical rows each compute one element of the product matrix:

$$\sum_{k=1}^{n} a_{ik} b_{kj}$$

DG's can have an arbitrary number of dimensions. Since only two dimensions are available in the implementation, the DG is collapsed onto a signal flow graph with maximally two dimensions (Figure 5.5). This step involves processor assignment and scheduling. In our example, one processor element (PE) is assigned to every vertical row of operations in the DG. These operations are scheduled linearly on the corresponding PE. The last step consists of mapping the SFG into a single instruction multiple data stream (SIMD) array, a systolic array, a wavefront array, or a multiple instruction multiple data stream (MIMD) array. Figure 5.6 shows a systolic array for the matrix multiplication algorithm.

This section focuses on the verification of systolic arrays against dependency graphs. The same principles apply to MIMD processors as implementations and signal flow graphs as specifications. The verification of SIMD's is not different from that of microcoded processors.

5.6.2 Verification of Systolic Array Processors

The basic idea for verifying systolic array processors is the same as for microcoded processors. The specification graph is assumed to be fine-grain, so that the operations correspond to register transfers in the implementation. Correctness is guaranteed if a set of CTL conditions on the RT's and on individual register writings are satisfied.

This result is easily obtainable if all the operations in the dependency graph are realized with distinct control signals in the implementation. In systolic array, as in microcoded processors which perform iterative loops, this is not generally the case. However, whereas, for microcoded processors, individual operations can be distinguished by adding the loop

counter to the controller, there may be no hardware which allows the same for a systolic array. For example, the systolic array for matrix multiplication in Figure 5.6 may be implemented to take matrices separated by marker symbols, where the partial sums of each PE are initialized to 0 every time a marker appears at the input. At subsequent cycles, when real matrix elements appear at the input, the current value of the partial sum, c_{k-1}, is incremented by $a_{ik}b_{kj}$. For this implementation, there is no way of deducing the value of k from observing a certain PE. This is because, in some sense, control resides in the data stream and not in control logic.

The collapsing of recurring operations into a single operation for verification purposes posed no problem for the microcoded designs we verified. Even with the collapsing the correctness conditions were sufficient, and for the designs we verified so far they are not overly stringent. This was not the case for the systolic array processors we verified. For example, when a systolic array is designed to take relevant input values separated by markers, so that control resides in the data stream, correctness of the implementation depends on the length of relevant input segments between markers. The implementation will work improperly if it gets inappropriate input streams.

The problems above can be remedied by creating control logic, purely for verification purposes, which captures the assumptions that were made about the input stream. For the example with markers and relevant input signals, this controller has an output indicating whether a marker or a relevant input symbol appears at the input. The number of times a relevant input symbol is signaled in between signaling of marker inputs, has to be exactly three for our example of the multiplication of 3×3 matrices. This can be verified by checking CTL conditions on the controller. In general, additional control logic for verification purposes can be generated from information as to how the operations in the dependency graph are scheduled on the array processor. This information resolves the ambiguity as to which operation in the implementation corresponds to a particular operation in the specification.

Apart from the fact that additional control logic may have to be created for systolic arrays, the differences between verifying array processors and microcoded processors are minor. With fictitious registers at the input and output ports for skewing the input stream

and rectifying the output stream, conditions (5.1) and (5.2) from Section 5.4 are still valid. Conditions (5.3) and (5.4) for regular arcs in the specification remain valid as well. Condition (5.6) for selection of the appropriate operation in if-then-elses, changes because of the absence of pipelining, and because the condition under which a certain operation is to be selected can be expressed in terms of signals which are computed at the time of selection (as opposed to being produced in a previous register transfer). This allows for dropping AX^- in the formulae. The new conditions (one for every possible value of i) become:

$$f_0 \Rightarrow \bigvee_j (c_{0ij} \wedge cond_{0ij}) \tag{5.14}$$

Finally, the way condition (5.4) is to be relaxed in case recurring operations (i.e. operations occuring in an iterative loop) depend on an operation before the loop for their input, is different from the way this is done for microcoded processors. For microcoded processors, the condition is relaxed so that the iterative loop is traversed no more than once. In systolic array processors there is no such loop. In this case we replace event g in the right-hand side of (5.4) with an event $g2$ which has a value of 0 at repeated occurrences of g.

5.6.3 Results

Two systolic array designs were verified. The first is "mul8", a multiplier for 8×8 matrices, analogous to circuit shown in Figure 5.6. The input matrices are not separated by markers, requiring modulo-8 counters in each processor element. Verification took 171 s on a SUN SPARC-Station 2.

"gauss3" is a design for solving the system of equations $Ax = b$ with Gaussian elimination ([35], p. 82), where A is a 3×3 matrix. The processor elements contain minimal control, relying on markers in the input stream. Additional control logic is needed for verification purposes. Verification took 6 s.

5.7 Conclusion

This chapter presented an event-based verification strategy for verifying microcoded processors and systolic array processors against signal flow graphs or dependency graphs. The strategy assumes that the implementation performs the same set of operations as the specification, possibly with a different degree of parallelism, a different schedule, and a different latency. The strategy consists of the combinational verification of the data path modules in the implementation against the corresponding operations in the specification, and the verification of a collection of past-tense CTL formulae with respect to the controller of the implementation. These formulae are generated automatically, and it has been demonstrated that they are sufficient for correctness. The method has been applied to realistic DSP processors.

There are several opportunities for further work. While it has been demonstrated that the CTL conditions are *sufficient* for correctness, there is no guarantee that they are *necessary*. Thus, there could be designs which are correct but violate the CTL formulae. More experiments have to be conducted to obtain concrete examples of such designs. If necessary, the CTL formulae have to modified to accommodate correct designs that violate the current set of formulae. Perhaps a set of formulae can be derived which are both necessary and sufficient for correctness under certain assumptions.

The verification of the CTL formulae can be accelerated by evaluating universally quantified formulae directly, instead of first reducing them to existentially quantified ones. This relies on the particular form that the CTL conditions derived in this chapter have. There may be other ways of accelerating the verification process.

A final topic for future work is the verification of the β-relation between original and expanded signal flow graphs.

Chapter 6

Conclusion

This thesis presented automatic procedures for the behavioral verification of digital designs. The specification is taken to be a synchronous machine, just like the implementation. This is consistent with current practice, where, for instance, digital signal processors are designed from signal flow graphs or dependency graphs which can be interpreted as maximally parallel circuits, or where microprocessors are designed from microprocessors of a previous generation. In general, a design can be verified against another design with a desired functionality but inadequate area, power or performance characteristics.

Behavioral equivalence is formalized as a relation between the string functions associated with the implementation and the specification. String functions capture the input/output behavior of a synchronous machine in a pure form, without reference to the internal state of the circuit. The string functions of the implementation and the specification do not need to be identical. The implementation may have a different latency, degree of parallelism, etc. A number of relations are defined in this thesis which bridge these differences. These relations are not guaranteed to be complete. There seems to be no formal bound on the difference in input/output behavior that is deemed acceptable in design practice. The relations defined in the thesis are based on my own experience and intuitions of architectural design. The framework developed can be augmented with new relations, if necessary.

Just as there do not seem to be bounds on acceptable differences in input/output behavior, there also do not seem to be absolute equivalence classes of string functions. If there

were absolute equivalence classes, a verification program could possibly be written which takes only the implementation and the specification as inputs, without any information as to how they are related. Since there do no seem to be absolute equivalence classes, taking only circuits as inputs seems to be impossible. A user of the procedures presented in this thesis is required to state precisely which (subscripted) relation he intends the implementation and the specification to have. The procedures then automatically verify that relation.

Recently a number of basic procedures have been introduced, based on BDD representations and implicit enumeration techniques, which efficiently perform certain verification tasks. These procedures are not directly applicable to the verification of industrial designs, since they do not directly address relevant correctness requirements, and because they cannot be directly applied to large circuits. The basic procedures can be used indirectly, however, if sufficient, or necessary and sufficient conditions for an overall correctness requirement can be derived, which are verifiable by the basic procedures. The overall correctness requirement should be a relevant one, such as the validity of a string function relation, and the verification of the (necessary and) sufficient conditions should be applicable to large circuits. If the (necessary and) sufficient conditions can be generated automatically for a given class of implementation/specification pairs, the overall correctness requirement could be verified automatically for that class.

The research direction outlined above has been partially explored in this thesis. I first developed a procedure which automatically verifies an arbitrary composition of five primitive string function relations (under certain assumptions). The assumptions are that multiple occurrences of β have modulo counters as H-functions, that parallel implementations have a prefix preserving serial equivalent, and that encodings on parallel vectors can be decomposed into encodings on the components of the vector. The procedure consists of modifying the specification and the implementation circuits, and then applying an FSM equivalence check. It works for arbitrary implementation/specification pairs (under the specified assumptions), and implements a sound and complete proof method for verifying string function relations. However, since it is practical for small to medium-sized circuits only, it constitutes itself a lower-level procedure to be used in a broader verification strategy. Section 4.5 hints at a

possible candidate for such a broader verification strategy, which is perhaps suited for the verification of microprocessor designs against previous generation designs.

The goal of automatically verifying relevant correctness conditions for large designs was attained for particular classes of implementation/specification pairs with an event-based compositional verification method. The method allows for the verification of microcoded and array processors against signal flow graphs or dependency graphs, under the assumption that the implementation performs the same set of operations as the specification. It is also assumed that the specification has a sufficiently fine granularity, so that operations in the specification correspond to register transfers in the implementation. Under these assumptions the overall correctness condition can be decomposed into the combinational correctness of the data path modules in the implementation, and the validity of a number of CTL formulae with respect to the controller. Each formula expresses conditions on events in the implementation, such as register transfers. These conditions are demonstrated to be sufficient for correctness. The method can be applied to large circuits because state space traversal is applied to the controller only.

Two realistic industrial designs were verified with the event-based compositional method. Both of them are microcoded processors. The first design is a recursive filter consisting of three cascaded second-order stages followed by a line equalizer. The data path consists of 3 14-bit ALU's and 30 14-bit registers. The controller contains 14 latches and 58 output lines, and initiates 70 register transfers for each new input sample. Verification of this design took roughly 2 minutes on a SUN SPARC-Station 2 and revealed several errors, which could be traced back to bugs in the synthesis system.

The second circuit is an echo cancellor, the specification of which contains if-then-elses and iterative loops. The data path consists of one ALU, one multiplier, two ACU's (all with word lengths of 14 bits), two RAM's (with 128 12-bit words and 256 14-bit words respectively) and 39 registers. The controller contains 20 latches and 95 output lines, and initiates 144 register transfers for each new input sample. Verification took 37 minutes and revealed again several errors. Examples of errors include the corruption of values in registers between production and consumption, incorrect jumps, violated precedence constraints,

and insufficient delay between decision making operations and register transfers generating status bits for the decision making – the minimal delay corresponds with the number of pipeline stages in loops around the controller and the data path.

The event-based compositional method was also applied to the verification of systolic array designs against dependency graphs.

There are several opportunities for further work. First, more work can be done on the verification of circuits with globally timed control. The verification of CTL conditions derived in this thesis can be accelerated due their particular form. This will make it easier to check larger designs than were verified so far. Continued experimental work is needed to see if the verification of the CTL conditions, which are sufficient but not necessary for correctness, produces false negatives in practical situations. Work remains to be done also on verifying the β-relation between an expanded and an original signal flow graph. This constitutes mainly a verification check across high-level memory management (i.e. the design of arithmetic for computing RAM addresses – low-level memory management involves the selection of registers for communicating signals).

A second broad topic for further research is the verification of designs with locally timed control, such as microprocessors, along the lines of the verification strategy outlined above (i.e. the reduction of an overall behavioral correctness requirement, such as the validity of a behavioral relation, to efficiently verifiable conditions). Another possible topic for further work is the adaptation of the method described in this thesis to software verification. The verification of CTL conditions can be applied to compiled (binary) code to check for consistency with a signal flow graph type specification.

Chapter 7

Appendix A: Proofs

7.1 Lemmas leading to Theorem 4.6

Lemma 4.2 *Given* F, K :

$(\exists K \mid F\ \delta_{k,l}\ K\ \wedge\ K\ \beta_{H,n}\ G) \Leftrightarrow (\exists K' \mid F\ \beta_{H \circ k^{-1^*},n}\ K'\ \wedge\ K'\ \delta_{k,l}\ G)$

Proof:

$F\ \delta_{k,l}\ K'\ \wedge\ K'\ \beta_{H,n}\ G$

$\Leftrightarrow F \circ k^* = l^* \circ K\ \wedge$

$Relevant(K(x), R_{0\uparrow n} \circ H(x)) = G(Relevant(x \downarrow 1..(|\, x \mid - n), H(x \downarrow 1..(|\, x \mid - n))))$

$\Leftrightarrow Relevant(l^{-1^*} \circ F \circ k^*(x), R_{0\uparrow n} \circ H(x)) =$

$G(Relevant(x \downarrow 1..(|\, x \mid - n), H(x \downarrow 1..(|\, x \mid - n))))$

$\Leftrightarrow l^{-1^*}(Relevant(F \circ k^*(x), R_{0\uparrow n} \circ H(x))) =$

$G(Relevant(x \downarrow 1..(|\, x \mid - n), H(x \downarrow 1..(|\, x \mid - n))))$

$\Leftrightarrow Relevant(F \circ k^*(x), R_{0\uparrow n} \circ H(x)) =$

$l^* \circ G(Relevant(x \downarrow 1..(|\, x \mid - n), H(x \downarrow 1..(|\, x \mid - n))))$

$\Leftrightarrow$ (Let $y = k^*(x)$)

$Relevant(F(y), R_{0\uparrow n} \circ H \circ k^{-1^*}(y)) =$

$\underbrace{l^* \circ G \circ k^{-1^*}}_{K'}(Relevant(y \downarrow 0..(|\, y \mid - n), H(y \downarrow 0..(|\, y \mid - n))))$

$\Leftrightarrow \exists K' \mid F\ \beta_{H \circ k^{-1^*},n}\ K'\ \wedge\ K'\ \delta_{k,l}\ G$ □

Lemma 4.3 *Given F, K where F is m-serializable; given combinational encodings k and l which (to the extent that they operate on m-parallel vectors) can be computed by applying combinational functions k^s and l^s to the individual elements in the vector:*
$(\exists K \mid F\ \delta_{k,l}\ K\ \wedge\ K\ \gamma_m\ G) \Leftrightarrow (\exists K \mid F\ \gamma_m\ K\ \wedge\ K\ \delta_{k^s,l^s}\ G)$

Proof:

$F\ \delta_{k,l}\ K\ \wedge\ K\ \gamma_m\ G$

$\Leftrightarrow F \circ k^* = l^* \circ K\ \wedge\ K \circ C^m = C^m \circ G$

$\Leftrightarrow {l^{-1}}^* \circ F \circ k^* \circ C_m = C_m \circ G$

$\Leftrightarrow F \circ k^* \circ C_m = l^* \circ C_m \circ G$

$\Leftrightarrow F \circ C_m \circ k^{s*} = C_m \circ l^{s*} \circ G$

$\Leftrightarrow F \circ C_m = C_m \circ \underbrace{l^{s*} \circ G \circ {k^{s-1}}^*}_{K'}$

$\Leftrightarrow \exists K' \mid F \circ C^m = C^m \circ K'\ \wedge\ K' \circ {k^{s-1}}^* = {l^{s-1}}^* \circ G$ □

Lemma 4.4 *Given F, K :*
$(\exists K \mid F\ \beta_{H,n}\ K\ \wedge\ K\ \epsilon_{IDC}\ G) \Leftrightarrow (\exists K' \mid F\ \epsilon_{IDC_{H,n}}\ K'\ \wedge\ K'\ \beta_{H,n}\ G)$
where $IDC_{H,n}$ contains strings x such that
$Relevant(x \downarrow 1..(\mid x \mid - n), H(x \downarrow 1..(\mid x \mid - n))) \in IDC.$

Proof:

$F\ \beta_{H,n}\ K\ \wedge\ K\ \epsilon_{IDC}\ G$

$\Leftrightarrow Relevant(F(x), R_{0\uparrow n} \circ H(x)) =$
$K(Relevant(x \downarrow 1..(\mid x \mid - n), H(x \downarrow 1..(\mid x \mid - n))))\ \wedge$
$(\neg(\exists y \in IDC \mid y \preceq x) \Rightarrow K(x) = G(x))$

$\Leftrightarrow \neg(\exists y \in IDC_{H,n} \mid y \preceq x) \Rightarrow$
$(Relevant(F(x), R_{0\uparrow n} \circ H(x)) =$
$G(Relevant(x \downarrow 1..(\mid x \mid - n), H(x \downarrow 1..(\mid x \mid - n)))))$

$\Leftrightarrow \exists K' \mid (\neg(\exists y \in IDC_{H,n} \mid y \preceq x) \Rightarrow F(x) = K'(x))\ \wedge$

$$Relevant(K'(x), R_{0\restriction n} \circ H(x)) = G(Relevant(x \downarrow 1..(|\, x \,| - n), H(x \downarrow 1..(|\, x \,| - n))))$$

□

Lemma 4.5 *Given F, K where F is m-serializable:*
$(\exists K \mid F\ \epsilon_{IDC}\ K\ \wedge\ K\ \gamma_m\ G) \Leftrightarrow (\exists K' \mid F\ \gamma_m\ K'\ \wedge\ K'\ \epsilon_{E_m(IDC)}\ G)$
where $E_m(IDC) = \{E_m(x) \mid x \in IDC\}$

Proof:

$F\ \epsilon_{IDC}\ K\ \wedge\ K\ \gamma_m\ G$

$\Leftrightarrow (\ \neg(\exists y \in IDC \mid y \preceq x) \Rightarrow F(x) = K(x)\ )\ \wedge\ K \circ C_m = C_m \circ G$

$\Leftrightarrow \neg(\exists y' \in E_m(IDC) \mid y' \preceq x') \Rightarrow (F \circ C_m(x') = C_m \circ G(x'))$

$\Leftrightarrow \exists K' \mid F \circ C_m = C_m \circ K'\ \wedge\ (\neg(\exists y' \in E_m(IDC) \mid y' \preceq x') \Rightarrow K'(x') = G(x'))$ □

Lemma 4.6 *Given F, K :*
$(\exists K \mid F\ \delta_{k,l}\ K\ \wedge\ K\ \epsilon_{IDC}\ G) \Leftrightarrow (\exists K' \mid F\ \epsilon_{k^*(IDC)}\ K'\ \wedge\ K'\ \delta_{k,l}\ G)$
where $k^*(IDC) = \{k^*(x) \mid x \in IDC\}$

Proof:

$F\ \delta_{k,l}\ K\ \wedge\ K\ \epsilon_{IDC}\ G$

$\Leftrightarrow F \circ k^* = l^* \circ K\ \wedge\ (\neg(\exists y \in IDC \mid y \preceq x) \Rightarrow K(x) = G(x))$

$\Leftrightarrow \neg(\exists y \in IDC \mid y \preceq x) \Rightarrow (F \circ k^*(x) = l^* \circ G(x))$

$\Leftrightarrow \exists K' \mid\ (\ \neg(\exists y' \in k^*(IDC) \mid y' \preceq x') \Rightarrow F(x') = K'(x')\ )\ \wedge\ K' \circ k^*(x) = l^* \circ G(x)$ □

Lemma 4.7 *Given F, K :*
$(\exists K \mid F\ \beta_{H,n}\ K\ \wedge\ K\ \zeta_{ODC}\ G) \Leftrightarrow (\exists K' \mid F\ \zeta_{ODC_{H,n}}\ K'\ \wedge\ K'\ \beta_{H,n}\ G)$
where $ODC_{H,n} = (\{ODC^{i,m}_{H,n}\}, Special_{H,n})$ *with* $ODC^{i,m}_{H,n} = \{x \mid Relevant(x, R_{0\restriction n} \circ H(x)) \in ODC_i \wedge |\, x \,| = m\}$ *and* $Special_{H,n}(ODC^{i,m}_{H,n})$ *having in the relevant positions the elements of* $Special(ODC_i)$*, and 0's elsewhere.*

Proof:

$F\ \beta_{H,n}\ K \quad \wedge \quad K\ \zeta_{ODC}\ G$

$\Leftrightarrow Relevant(F(x), R_{0\uparrow n} \circ H(x)) =$

$K(Relevant(x \downarrow 1..(|\ x\ | - n), H(x \downarrow 1..(|\ x\ | - n)))) \ \wedge$

$Subst_{ODC} \circ K = Subst_{ODC} \circ G$

$\Leftrightarrow Subst_{ODC}(Relevant(F(x), R_{0\uparrow n} \circ H(x))) =$

$Subst_{ODC} \circ G(Relevant(x \downarrow 1..(|\ x\ | - n), H(x \downarrow 1..(|\ x\ | - n))))$

$\Leftrightarrow Relevant(Subst_{ODC_{H,n}}(F(x)), R_{0\uparrow n} \circ H(x)) =$

$Subst_{ODC} \circ G(Relevant(x \downarrow 1..(|\ x\ | - n), H(x \downarrow 1..(|\ x\ | - n))))$

$\Leftrightarrow \exists K' \mid Subst_{ODC_{H,n}} \circ F(x) = Subst_{ODC_{H,n}} \circ K'(x) \ \wedge$

$Relevant(K'(x), R_{0\uparrow n} \circ H(x)) =$

$G(Relevant(x \downarrow 1..(|\ x\ | - n), H(x \downarrow 1..(|\ x\ | - n))))$ □

Lemma 4.8 *Given F, K where F is m-serializable:*

$(\exists K \mid F\ \zeta_{ODC}\ K \ \wedge \ K\ \gamma_m\ G) \Leftrightarrow (\exists K' \mid F\ \gamma_m\ K' \ \wedge \ K'\ \zeta_{E_m(ODC)}\ G)$

where $E_m(ODC) = (\{E_m(ODC_i)\}, E_m(Special)$ *with* $E_m(ODC_i) = \{E_m(x) \mid x \in ODC_i\}$

and $(E_m(Special))(ODC_i) = E_m(Special(ODC_i))$

Proof:

$F\ \zeta_{ODC}\ K \quad \wedge \quad K\ \gamma_m\ G$

$\Leftrightarrow Subst_{ODC} \circ F = Subst_{ODC} \circ K \quad \wedge \ K \circ C_m = C_m \circ G$

$\Leftrightarrow Subst_{ODC} \circ F \circ C_m = Subst_{ODC} \circ C_m \circ G$

$\Leftrightarrow Subst_{ODC} \circ F \circ C_m = C_m \circ Subst_{E_m(ODC)} \circ G$

$\Leftrightarrow \exists K' \mid F \circ C_m = C_m \circ K' \quad \wedge \quad Subst_{ODC} \circ F \circ C_m = C_m \circ Subst_{E_m(ODC)} \circ G$

$\Leftrightarrow \exists K' \mid F \circ C_m = C_m \circ K' \quad \wedge \quad Subst_{E_m(ODC)} \circ K' = Subst_{E_m(ODC)} \circ G$ □

Lemma 4.9 *Given F, K :*

$(\exists K \mid F\ \delta_{k,l}\ K \ \wedge \ K\ \zeta_{ODC}\ G) \Leftrightarrow (\exists K' \mid F\ \zeta_{l^*(ODC)}\ K' \ \wedge \ K'\ \delta_{k,l}\ G)$

where $l^*(ODC) = (\{l^*(ODC_i)\}, l^*(Special))$
with $l^*(ODC_i) = \{l^*(x) \mid x \in ODC_i\}$ *and* $(l^*(Special))(ODC_i) = l^*(Special(ODC_i))$

Proof:

$F\ \delta_{k,l}\ K \quad \wedge \quad K\ \zeta_{ODC}\ G$

$\Leftrightarrow F \circ k^* = l^* \circ K \quad \wedge \quad Subst_{ODC} \circ K = Subst_{ODC} \circ G$

$\Leftrightarrow Subst_{ODC} \circ l^{-1^*} \circ F \circ k^* = Subst_{ODC} \circ G$

$\Leftrightarrow l^* \circ Subst_{ODC} \circ l^{-1^*} \circ F \circ k^* = l^* \circ Subst_{ODC} \circ G$

$\Leftrightarrow Subst_{l^*(ODC)} \circ F \circ k^* = Subst_{l^*(ODC)} \circ l^* \circ G$

$\Leftrightarrow Subst_{l^*(ODC)} \circ F = Subst_{l^*(ODC)} \circ \underbrace{l^* \circ G \circ k^{-1^*}}_{K'}$

$\Leftrightarrow \exists K' \mid Subst_{l^*(ODC)} \circ F = Subst_{l^*(ODC)} \circ K' \quad \wedge \quad K' \circ k^* = l^* \circ G$ □

Lemma 4.10 *Given* F, K :
$(\exists K \mid F\ \epsilon_{IDC}\ K \ \wedge\ K\ \zeta_{ODC}\ G) \Leftrightarrow (\exists K' \mid F\ \zeta_{ODC}\ K' \ \wedge\ K'\ \epsilon_{IDC}\ G)$

Proof:

$F\ \epsilon_{IDC}\ K \quad \wedge \quad K\ \zeta_{ODC}\ G$

$\Leftrightarrow (\neg(\exists y \in IDC \mid y \preceq x) \Rightarrow (F(x) = G(x))) \quad \wedge \quad Subst_{ODC} \circ K = Subst_{ODC} \circ G$

$\Leftrightarrow \neg(\exists y \in IDC \mid y \preceq x) \Rightarrow Subst_{ODC} \circ F(x) = Subst_{ODC} \circ G(x)$

$\Leftrightarrow \exists K' \mid Subst_{ODC} \circ F = Subst_{ODC} \circ K' \quad \wedge \quad \neg(\exists y \in IDC \mid y \preceq x) \Rightarrow K'(x) = G(x)$ □

For the first transitivity lemma (the one for the β-relation), we need another lemma first. Let mod_m be a function which returns a value of 1 at cycles $1 + mN$, $N = 0, 1, 2, \ldots$, and a value of 0 at the other cycles. Since this function does not depend on the input string, I write $Relevant(x, mod_m)$ instead of $Relevant(x, mod_m(x))$. Furthermore, I write R_n instead of $R_{0\uparrow n}$.

Lemma 4.11 $Relevant(Relevant(x, R_{n_1} \circ R_{m_1'} \circ mod_{m_1}), R_{n_2} \circ R_{m_2'} \circ mod_{m_2})$
$= Relevant(x, R_{n_1 + m_1 n_2'} \circ R_{m_1' + m_1 m_2'} \circ mod_{m_1 m_2})$

Proof:

$Relevant(x, R_{n_1} \circ R_{m_1'} \circ mod_{m_1})$ retains of x the characters at cycles $n_1 + m_1' + 1 + m_1 N$, $N = 0, 1, 2, \ldots$. If we apply *Relevant* again, with filtering function $R_{n_2} \circ R_{m_2'} \circ mod_{m_2}$, we retain the characters at cycles $n_1 + m_1' + 1 + m_1(n_2 + m_2') + m_1 m_2 N$, $N = 1, 2, \ldots$. This can be rewritten as $(n_1 + m_1 n_2) + (m_1' + m_1 m_2') + 1 + m_1 m_2 N$. These are exactly the relevant cycles picked out by $R_{n_1+m_1 n_2} \circ R_{m_1'+m_1 m_2'} \circ mod_{m_1 m_2}$ □

Lemma 4.12 *Given* F, K :
$(\exists K \mid F\ \beta_{H_1,n_1}\ K\ \wedge K\ \beta_{H_2,n_2}\ G) \Leftrightarrow F\ \beta_{H_3,n_3}\ G$
where $H_1 = R_{m_1'} \circ mod_{m_1}$, $H_2 = R_{m_2'} \circ mod_{m_2}$, $H_3 = R_{m_1'+m_1 m_2'} \circ mod_{m_1 m_2}$, *and* $n_3 = n_1 + m_1 n_2$.

Proof:

$F\ \beta_{H_1,n_1}\ K\ \wedge\ K\ \beta_{H_2,n_2}\ G$

$\Leftrightarrow Relevant(F(x), R_{n_1} \circ R_{m_1'} \circ mod_{m_1}) = K(Relevant(x, R_{m_1'} \circ mod_{m_1})$
$\wedge\ Relevant(K(y), R_{n_2} \circ R_{m_2'} \circ mod_{m_2}) = G(Relevant(y, R_{m_2'} \circ mod_{m_2})$

$\Leftrightarrow$ (Let $y = Relevant(x, R_{m_1'} \circ mod_{m_1})$)
$Relevant(Relevant(F(x), R_{n_1} \circ R_{m_1'} \circ mod_{m_1}), R_{n_2} \circ R_{m_2'} \circ mod_{m_2})$
$= G(Relevant(Relevant(x, R_{m_1'} \circ mod_{m_1}), R_{m_2'} \circ mod_{m_2})$

$\Leftrightarrow$ (Lemma 4.11)
$Relevant(F(x), R_{n_1+m_1 n_2} \circ R_{m_1'+m_1 m_2'} \circ mod_{m_1 m_2}) =$
$G(Relevant(x, R_{m_1'+m_1 m_2'} \circ mod_{m_1 m_2})$

$\Leftrightarrow F\ \beta_{H_3,n_3}\ G$ □

The remaining transitivity lemmas are trivial.

Lemma 4.13 $(\exists K \mid F\ \gamma_{m_1}\ K\ \wedge K\ \gamma_{m_2}\ G) \Leftrightarrow F\ \gamma_{m_1 m_2}\ G$

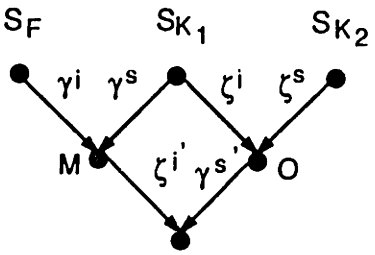
Lemma 4.14 $(\exists K \mid F\ \delta_{k_1,l_1}\ K\ \wedge K\ \delta_{k_2,l_2}\ G) \Leftrightarrow F\ \delta_{k_1 \circ k_2, l_1 \circ l_2}\ G$

Lemma 4.15 $(\exists K \mid F\ \epsilon_{IDC_1}\ K\ \wedge K\ \epsilon_{IDC_2}\ G) \Leftrightarrow F\ \epsilon_{IDC_1 \cup IDC_2}\ G$

Lemma 4.16 $(\exists K \mid F\ \zeta_{ODC_1}\ K\ \wedge K\ \zeta_{ODC_2}\ G) \Leftrightarrow F\ \zeta_{ODC_{1,2}}\ G$
where $ODC_{1,2} = (\{$*union of the elements of* $C_i \mid i \in I\}, Special_{1,2})$ *in which* $(C_i)_{i\in I}$ *are the equivalence classes of the transitive closure of the relation "is not disjoint with" in* $ODC_1 \cup ODC_2$*, and with* $Special_{1,2}$ *coinciding with* $Special_1$ *whenever possible, and coinciding with* $Special_2$ *otherwise.*

4.2 Eliminating Intermediate Circuits (Theorem 4.7)

Eliminating S_{K_1}



Construct transformations $\zeta^{i'}$ and $\gamma^{s'}$ such that

$$\zeta^{i'} \circ \gamma^i(S_F) = \gamma^{s'} \circ \zeta^s(S_{K_2}) \Leftrightarrow S_F\ (\zeta_{ODC'} \circ \gamma_m)\ S_{K_2}$$

We have $\gamma^s(K) = R_{0\uparrow m-1} \circ K$ and $\zeta^i(K) = R_{0\uparrow p} \circ Subst_{ODC} \circ K$.
Take $\gamma^{s'} = \gamma^s$ and $\zeta^{i'}(K) = R_{0\uparrow p} \circ Subst_{ODC'} \circ K$
where $ODC' = (\{ODC'_i\}, Special')$ with $ODC'_i = \{(0 \uparrow m-1).x \mid x \in ODC_i\}$ and $Special'(ODC'_i) = (0 \uparrow m-1).Special(ODC_i)$.
I first prove the left-to-right implication. Given $\zeta_{i'}(M) = \gamma_{s'}(O)$, I have to prove the existence of a string function K such that $\gamma^s(K) = M$ and $\zeta^i(K) = O$.
There exists a K such that $R_{0\uparrow m-1} \circ K = M$, since $M = \gamma^i(F)$, implying that output strings of M do not depend on the last $m-1$ input values, and that the first $m-1$ output values are 0. Obviously $\gamma^s(K) = M$. I still have to prove $\zeta^i(K) = O$.
$\zeta_{i'}(M) = \gamma_{s'}(O)$

$\Rightarrow R_{0\uparrow p} \circ Subst_{ODC'} M = R_{0\uparrow m-1} \circ O$

$\Rightarrow R_{0\uparrow p} \circ Subst_{ODC'} \circ R_{0\uparrow m-1} \circ K = R_{0\uparrow m-1} \circ O$

$\Rightarrow R_{0\restriction p} \circ R_{0\restriction m-1} \circ Subst_{ODC} \circ K = R_{0\restriction m-1} \circ O$

$\Rightarrow R_{0\restriction m-1} \circ R_{0\restriction p} \circ Subst_{ODC} \circ K = R_{0\restriction m-1} \circ O$

$\Rightarrow R_{0\restriction p} \circ Subst_{ODC} \circ K = O$

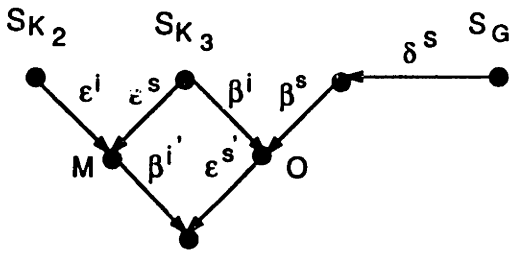
$\Rightarrow \zeta^i(K) = O.$

To prove the right-to-left implication, it suffices to show that the following identity holds (for arbitrary K):

$$\zeta^{i'} \circ \gamma^s(K) = \gamma^{s'} \circ \zeta^i(K)$$

This can be done with exactly the same symbolic manipulations as the ones shown above.

Eliminating S_{K_3}



Construct transformations $\beta^{i'}$ and $\epsilon^{s'}$ such that

$$\beta^{i'} \circ \epsilon^i(S_{K_2}) = \epsilon^{s'} \circ \beta^s \circ \delta^s(S_G) \Leftrightarrow S_{K_2}\ (\delta_{k,l} \circ \beta_{H.n} \circ \epsilon_{IDC})\ S_G$$

We have $L(\epsilon^s(K(x))) =$ if $(\exists y \in IDC \mid y \preceq x)$ then 0 else $L(K(x))$ (Recall that L extracts the last character from a string.)

and $L(\beta^i(K(x))) =$ if $L(R_{0\restriction n} \circ H(x))$ then $L(K(x))$ else 0

Take $\epsilon^{s'} = \epsilon^s$ and $\beta^{i'} = \beta^i$.

I first prove the left-to-right implication. Given $\beta_{i'}(M) = \epsilon_{s'}(O)$, I have to prove the existence of a string function K such that $\epsilon^s(K) = M$ and $\beta^i(K) = O$.

Take K such that $L(K(x)) =$ if $(\exists y \in IDC \mid y \preceq x)$ then $L(O(x))$ else $L(M(x))$.

Obviously $\epsilon^s(K) = M$. I still have to prove $\beta^i(K) = O$.

$\beta_{i'}(M) = \epsilon_{s'}(O)$

$\Rightarrow$ if $L(R_{0\uparrow n} \circ H(x))$ then $L(M(x))$ else 0 = if $(\exists y \in IDC \mid y \preceq x)$ then 0 else $L(O(x))$

$\Rightarrow$ if $L(R_{0\uparrow n} \circ H(x))$ then $L(K(x))$ else $0 = L(O(x))$

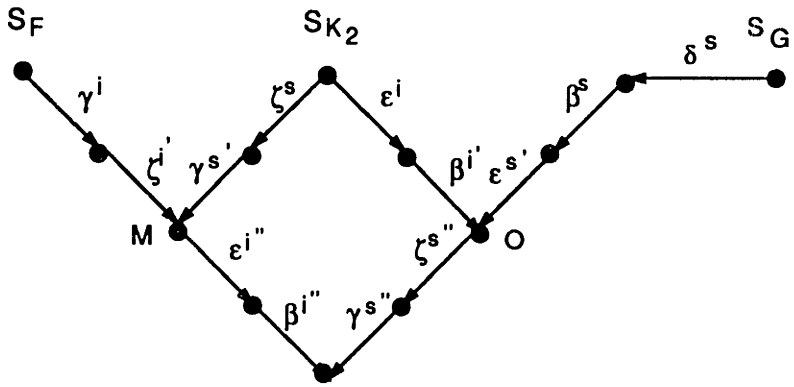
$\Rightarrow \beta^i(K) = O$.

To prove right-to-left implication, it suffices to show that the following identity holds (for arbitrary K):

$$\beta^{i'} \circ \epsilon^s(K) = \epsilon^{s'} \circ \beta^i(K)$$

This identity is obvious.

Eliminating S_{K_2}



Construct transformations $\beta^{i''}$, $\epsilon^{i''}$, $\gamma^{s''}$ and $\zeta^{s''}$ such that

$\beta^{i''} \circ \epsilon^{i''} \circ \zeta^{i'} \circ \gamma^i(S_F) = \gamma^{s''} \circ \zeta^{s''} \circ \epsilon^{s'} \circ \beta^s \circ \delta^s(S_G) \Leftrightarrow S_F \ (\delta_{k,l} \circ \beta_{H,n} \circ \epsilon_{IDC} \circ \zeta_{ODC} \circ \gamma_m) \ S_{K_2}$

We have $\gamma^{s'} \circ \zeta^s(K) = R_{0\uparrow m-1} \circ R_{0\uparrow p} \circ Subst_{ODC} \circ K$

and $L(\beta^{i'} \circ \epsilon^i(K(x)))$ = if $L(R_{0\uparrow n} \circ H(x))$ then (if $(\exists y \in IDC \mid y \preceq x)$ then 0 else $L(K(x)))$ else 0

Take $\gamma^{s''} = \gamma^{s'}$, $\zeta^{s''} = \zeta^s$, $\beta^{i''} = \beta^{i'}_{H,n+m+p-1}$ and $\epsilon^{i''} = \epsilon^i_{IDC''}$ where $IDC'' = \{(0 \uparrow m+p-1.x) \mid x \in IDC\}$.

I first prove the left-to-right implication. Given $\beta^{i''} \circ \epsilon^{i''}(M) = \gamma^{s''} \circ \zeta^{s''}(O)$, I have to prove the existence of a string function K such that $\gamma^{s'} \circ \zeta^s(K) = M$ and $\beta^{i'} \circ \epsilon^i(K) = O$.

Take K such that $R_{0\uparrow p} \circ Subst_{ODC'} \circ R_{0\uparrow m-1} \circ K = M$. This is possible because $M = \zeta^{i'} \circ \gamma^{i}(S_F)$. This leads to

$\gamma^{s'} \circ \zeta^{s}(K)$

$= R_{0\uparrow m-1} \circ R_{0\uparrow p} \circ Subst_{ODC'} \circ K$

$= R_{0\uparrow p} \circ Subst_{ODC'} \circ R_{0\uparrow m-1} \circ K$

$= M$

K can still be varied in accordance with ODC. This is exploited in the remainder of the proof.

$\beta^{i''} \circ \epsilon^{i''}(M) = \gamma^{s''} \circ \zeta^{s''}(O)$

$\Rightarrow$ if $L(R_{0\uparrow n+m+p-1} \circ H)(x)$ then (if $(\exists y \in IDC'' \mid y \preceq x)$ then 0 else $L(M(x))$ else 0

$= L(R_{0\uparrow m-1} \circ R_{0\uparrow p} \circ Subst_{ODC} \circ O(x))$

$\Rightarrow$ if $L(R_{0\uparrow n+m+p-1} \circ H)(x)$ then

(if $(\exists y \in IDC'' \mid y \preceq x)$ then 0 else $L(R_{0\uparrow p} \circ Subst_{ODC'} \circ R_{0\uparrow m-1} \circ K(x))$)

else 0

$= L(R_{0\uparrow m-1} \circ R_{0\uparrow p} \circ Subst_{ODC} \circ O(x))$

$\Rightarrow$ if $L(R_{0\uparrow n} \circ H)(x)$ then

(if $(\exists y \in IDC \mid y \preceq x)$ then 0 else $L(Subst_{ODC} \circ K(x))$)

else 0

$= L(Subst_{ODC} \circ O(x))$

$\Rightarrow$ K can be chosen such that

if $L(R_{0\uparrow n} \circ H)(x)$ then

(if $(\exists y \in IDC \mid y \preceq x)$ then 0 else $L(Subst_{ODC} \circ K(x))$)

else 0

$= L(O(x))$

$\Rightarrow \beta^{i'} \circ \epsilon^{i}(K) = O.$

To prove the right-to-left implication, it suffices to show that the following identity holds (for arbitrary K):

$$\beta^{i''} \circ \epsilon^{i''} \circ \gamma^{s'} \circ \zeta^{s}(K) = \gamma^{s''} \circ \zeta^{s''} \circ \beta^{i'} \circ \epsilon^{i}(K)$$

This can be done with the same symbolic manipulations as above.

4.3 Transitivity of the β-relation (Theorem 5.1)

Recall Theorem 5.1.

Theorem 5.1 $F\ \beta_{R_{n'} o mod_n; i_1 \ldots i_I; o_1 \ldots o_O}\ L\ \wedge\ L\ \beta_{R_{m'} o mod_m; j_1 \ldots j_J; k_1 \ldots k_K}\ G$

$\Leftrightarrow\quad F\ \beta_{R_{n'+nm'} o mod_{nm}; S_1; S_2}\ G$

where $S_1 = i_1 + nj_1, \ldots, i_I + nj_1,\ \ldots\ , i_1 + nj_J, \ldots, i_I + nj_J$

and $S_2 = o_1 + nk_1, \ldots, o_O + nk_1,\ \ldots\ , o_1 + nk_K, \ldots, o_O + nk_K$

Proof:

$F\ \beta_{R_{n'} o mod_n; i_1 \ldots i_I; o_1 \ldots o_O}\ L\ \wedge\ L\ \beta_{R_{m'} o mod_m; j_1 \ldots j_J; k_1 \ldots k_K}\ G$

$\Leftrightarrow Relevant(F(x), R_{o_1} \circ R_{n'} \circ mod_n) \underline{\times} \ldots \underline{\times} Relevant(F(x), R_{o_O} \circ R_{n'} \circ mod_n)$

$= L(Relevant(x, R_{i_1} \circ R_{n'} \circ mod_n) \underline{\times} \ldots \underline{\times} Relevant(x, R_{i_I} \circ R_{n'} \circ mod_n))$

$\wedge\ Relevant(L(y), R_{k_1} \circ R_{m'} \circ mod_m) \underline{\times} \ldots \underline{\times} Relevant(L(y), R_{k_K} \circ R_{m'} \circ mod_m)$

$= G(Relevant(y, R_{j_1} \circ R_{m'} \circ mod_m) \underline{\times} \ldots \underline{\times} Relevant(y, R_{j_J} \circ R_{m'} \circ mod_m))$

$\Leftrightarrow$ (Let $y = Relevant(x, R_{i_1} \circ R_{n'} \circ mod_n) \underline{\times} \ldots \underline{\times} Relevant(x, R_{i_I} \circ R_{n'} \circ mod_n)$

$Relevant(Relevant(F(x), R_{o_1} \circ R_{n'} \circ mod_n), R_{k_1} \circ R_{m'} \circ mod_m) \underline{\times} \ldots$

$\underline{\times} Relevant(Relevant(F(x), R_{o_O} \circ R_{n'} \circ mod_n), R_{k_1} \circ R_{m'} \circ mod_m)$

$\underline{\times} \ldots \underline{\times}$

$Relevant(Relevant(F(x), R_{o_1} \circ R_{n'} \circ mod_n), R_{k_K} \circ R_{m'} \circ mod_m) \underline{\times} \ldots$

$\underline{\times} Relevant(Relevant(F(x), R_{o_O} \circ R_{n'} \circ mod_n), R_{k_K} \circ R_{m'} \circ mod_m)$

$= G(Relevant(Relevant(x, R_{i_1} \circ R_{n'} \circ mod_n), R_{j_1} \circ R_{m'} \circ mod_m) \underline{\times} \ldots$

$\underline{\times} Relevant(Relevant(x, R_{i_I} \circ R_{n'} \circ mod_n), R_{j_1} \circ R_{m'} \circ mod_m)$

$\underline{\times} \ldots \underline{\times}$

$Relevant(Relevant(x, R_{i_1} \circ R_{n'} \circ mod_n), R_{j_J} \circ R_{m'} \circ mod_m) \underline{\times} \ldots$

$\underline{\times} Relevant(Relevant(x, R_{i_I} \circ R_{n'} \circ mod_n), R_{j_J} \circ R_{m'} \circ mod_m))$

$\Leftrightarrow$ (Lemma 4.11)

$Relevant(F(x), R_{o_1 + mk_1} \circ R_{n'+nm'} \circ mod_{nm}) \underline{\times} \ldots$

$\underline{\times} Relevant(F(x), R_{o_O + mk_1} \circ R_{n'+nm'} \circ mod_{nm})$

$$
\begin{aligned}
&\underline{\times} \ \cdots \ \underline{\times} \\
&\mathit{Relevant}(F(x), R_{o_1+mk_K} \circ R_{n'+nm'} \circ \mathit{mod}_{nm}) \ \underline{\times} \ \cdots \\
&\underline{\times} \ \mathit{Relevant}(F(x), R_{o_O+mk_K} \circ R_{n'+nm'} \circ \mathit{mod}_{nm}) \\
&= G(\mathit{Relevant}(x, R_{i_1+nj_1} \circ R_{n'+nm'} \circ \mathit{mod}_{nm}) \ \underline{\times} \ \cdots \\
&\qquad \underline{\times} \ \mathit{Relevant}(x, R_{i_I+nj_1} \circ R_{n'+nm'} \circ \mathit{mod}_{nm}) \\
&\qquad \underline{\times} \ \cdots \ \underline{\times} \\
&\qquad \mathit{Relevant}(x, R_{i_1+nj_J} \circ R_{n'+nm'} \circ \mathit{mod}_{nm}) \ \underline{\times} \ \cdots \\
&\qquad \underline{\times} \ \mathit{Relevant}(x, R_{i_I+nj_J} \circ R_{n'+nm'} \circ \mathit{mod}_{nm}) \\
&\Leftrightarrow F \ \beta_{R_{n'+nm'} \circ \mathit{mod}_{nm}; S_1; S_2} \ G
\end{aligned}
$$

□

.Chapter 8

Appendix B: SILAGE Description and Expanded SFG of an Echo Cancellor

8.1 SILAGE Description of "echo"

```
#define W_IN  fix<12,11>
#define WL    fix<14,13>
#define WFIR  fix<14,12>
#define NTAPS 128

func main (farendinput, line : W_IN) error : W_IN =
begin
  alfa = WL(0.0078125);
  alfacompl = WL(0.9921875);

  /*
    initialize all delays
  */
  av_farendinput@@1 = 0.125;
  av_line@@1 = 0;
  error@@1 = 0;
  coef[0]@@1 = 0;
  (i : 1 .. NTAPS-1) ::
  begin
    farendinput@@i = 0;
    coef[i]@@1 = 0;
  end;
```

```
/*
  verify whether you have to update the coefficients
*/
abs_error_del = if (error@1 > 0) -> error@1
                || -error@1
                fi;

abs_farendinput = if (farendinput > 0) -> farendinput
                  || -farendinput
                  fi;

abs_line = if (line > 0) -> line
           || -line
           fi;

av_farendinput = WL(alfacompl * av_farendinput@1) +
                 WL(alfa * abs_farendinput);

av_line = WL(alfacompl * av_line@1) +
          WL(alfa * abs_line);

halfav_farendinput = av_farendinput >> 1;

K_error = WL(WL(alfa / WL(av_farendinput * av_farendinput)) * error@1);

disable = ((av_farendinput < WL(0.0043945312)) |
           (av_line > halfav_farendinput) |
           (abs_error_del < W_IN(0.0625)));
/*
  calculate the coefficients and convolution
*/
coef[0] = if (disable) -> coef[0]@1
          || coef[0]@1 + WL(K_error * farendinput)
          fi;

fir[0] = WFIR(coef[0] * farendinput);

(j : 1 .. NTAPS-1) ::
begin
  coef[j] = if (disable) -> coef[j]@1
            || coef[j]@1 + WL(K_error * farendinput@j)
            fi;
  fir[j] = fir[j-1] + WFIR(coef[j] * farendinput@j);
```

```
  end;
  conv = fir[NTAPS -1];

  error = W_IN(WFIR(line) - conv);
end;
```

8.2 Expanded SFG of "echo" (Excerpts)

The following are excerpts from the expanded SFG description for the echo cancellor in a local IMEC format.

```
{ Code generated by:              }
{ Jack the Mapper, version 3.0.159 of the IMEC-II-Jack, 27-aug-1991 (may30) }
{ Atomics interface version 3.0. (17 feb 1989). }
{ Mon Nov 18 10:49:23 MET 1991                   }


PROGRAM echo(no_arch, no_prec, make_scanpad, do_racecar, no_cond, alap_const, no_folding)

INIT BEGIN
  FOR j23 := 1..127 FLAG j37 HOLDS
  BEGIN
    j36[j23+1]:reg_file_1_acu_0 <-
      j36[j23]:reg_file_1_acu_0
        | core_acu_0 = inca,
          mod_acu_0 = modulo[13,13],
          bus_2 = acu_0,
          mux_1_acu_0 = pass[0,1] <0> %1;

    j37[j23]:status_reg_11_acu_0 <-
      j36[j23]:reg_file_1_acu_0
        | core_acu_0 = inca,
          mod_acu_0 = modulo[13,13] <0> %2;

    j25[j23]:reg_file_2_bgram_0 <-
      j22@1:reg_file_1_acu_1,
      j36[j23]:reg_file_2_acu_1
        | core_acu_1 = sub,
          mod_acu_1 = modulo[6,13],
```

```
          bus_1 = acu_1 <0> %3;

    ...

    j14[-j23]:ram_3_bgram_0 <-
      j24[j23]:reg_file_1_bgram_0,
      j25[j23]:reg_file_2_bgram_0
        | core_bgram_0 = write <0> %9;

  END; { FOR j23 }

  j36[1]:reg_file_1_acu_0 <-
    #1:reg_file_1_fgrom_0
      | bus_3 = fgrom_0,
        mux_1_acu_0 = pass[1,1] <0> %10;

  j42:reg_file_2_bgram_1 <-
    j38@1:reg_file_1_acu_1,
    j51:reg_file_2_acu_1
      | core_acu_1 = add,
        mod_acu_1 = modulo[7,13],
        bus_1 = acu_1 <0> %11;

  j41:reg_file_1_bgram_1 <-
    #0:reg_file_1_fgrom_0
      | bus_3 = fgrom_0,
        mux_1_bgram_1 = pass[1,1] <0> %12;

  ...

  j49@1:reg_file_1_alu_0 <-
    #0:reg_file_2_alu_0
      | core_alu_0 = passb,
        sh_alu_0 = upsh[0,14],
        shmux_alu_0 = up,
        bus_6 = alu_0,
        buf_1_alu_0 = write,
        mux_1_alu_0 = pass[0,7] <0> %20;

END; { INIT }
BEGIN { TIME loop }
  j1:io_buf_reg_4_outpad_0 <-
    j1:pipe_latch_1_outpad_0
```

```
    | core_outpad_0 = write <0> %21;

j1:pipe_latch_1_outpad_0 <-
  j2:reg_file_1_alu_0,
  j3:reg_file_2_alu_0
    | core_alu_0 = sub,
      sh_alu_0 = upsh[0,14],
      shmux_alu_0 = up,
      bus_6 = alu_0,
      buf_1_alu_0 = write <0> %22;

j2:reg_file_1_alu_0 <-
  j4:io_buf_reg_1_inpad_0
    | core_inpad_0 = read,
      bus_5 = inpad_0,
      mux_1_alu_0 = pass[3,7] <0> %23;

j3:reg_file_2_alu_0 <-
  j5[128-1]:reg_file_1_alu_0
    | core_alu_0 = passa,
      sh_alu_0 = upsh[0,14],
      shmux_alu_0 = up,
      bus_6 = alu_0,
      mux_2_alu_0 = pass[0,6] <0> %24;

FOR j7 := 1..127 FLAG j8 HOLDS
BEGIN
  j6[j7+1]:reg_file_1_acu_0 <-
    j6[j7]:reg_file_1_acu_0
      | core_acu_0 = inca,
        mod_acu_0 = modulo[13,13],
        bus_2 = acu_0,
        mux_1_acu_0 = pass[0,1] <0> %25;

  j8[j7]:status_reg_10_acu_0 <-
    j6[j7]:reg_file_1_acu_0
      | core_acu_0 = inca,
        mod_acu_0 = modulo[13,13] <0> %26;

  j5[j7]:reg_file_1_alu_0 <-
    j5[j7-1]:reg_file_1_alu_0,
    j9[j7]:reg_file_2_alu_0
      | core_alu_0 = add,
```

```
      sh_alu_0 = upsh[0,14],
      shmux_alu_0 = up,
      bus_6 = alu_0,
      buf_1_alu_0 = write,
      mux_1_alu_0 = pass[0,7] <0> %27;

j9[j7]:reg_file_2_alu_0 <-
  j10[j7]:reg_file_1_mult_0,
  j11[j7]:reg_file_2_mult_0
    | core_mult_0 = mult,
      bus_4 = mult_0,
      mux_2_alu_0 = pass[4,6] <0> %28;

IF ((NOT (j45)
    OR NOT (j48))
 OR NOT (j33)) THEN
BEGIN
  j10[j7]:reg_file_1_mult_0 <-
    j18[-1,j7]:ram_3_bgram_1,
    j19[j7]:reg_file_2_bgram_1
      | core_bgram_1 = read,
        bus_6 = bgram_1,
        mux_1_mult_0 = pass[2,3] <0> %29;

  j10[j7]:reg_file_1_bgram_1 <-
    j18[-1,j7]:ram_3_bgram_1,
    j19[j7]:reg_file_2_bgram_1
      | core_bgram_1 = read,
        bus_6 = bgram_1,
        mux_1_bgram_1 = pass[0,1] <0> %30;

END ELSE
BEGIN
  j10[j7]:reg_file_1_mult_0 <-
    j20[j7]:reg_file_1_alu_0,
    j21[j7]:reg_file_2_alu_0
      | core_alu_0 = add,
        sh_alu_0 = upsh[0,14],
        shmux_alu_0 = up,
        bus_6 = alu_0,
        mux_1_mult_0 = pass[2,3] <0> %31;

  j20[j7]:reg_file_1_alu_0 <-
```

```
      j18[-1,j7]:ram_3_bgram_1,
      j19[j7]:reg_file_2_bgram_1
        | core_bgram_1 = read,
          bus_6 = bgram_1,
          buf_1_bgram_1 = write,
          mux_1_alu_0 = pass[0,7] <0> %32;

    j21[j7]:reg_file_2_alu_0 <-
      j44:reg_file_1_mult_0,
      j11[j7]:reg_file_2_mult_0
        | core_mult_0 = mult,
          bus_4 = mult_0,
          mux_2_alu_0 = pass[3,6] <0> %33;

    j10[j7]:reg_file_1_bgram_1 <-
      j20[j7]:reg_file_1_alu_0,
      j21[j7]:reg_file_2_alu_0
        | core_alu_0 = add,
          sh_alu_0 = upsh[0,14],
          shmux_alu_0 = up,
          bus_6 = alu_0,
          mux_1_bgram_1 = pass[0,1] <0> %34;

END;

j30[j7]:reg_file_2_bgram_1 <-
  j38@1:reg_file_1_acu_1,
  j6[j7]:reg_file_2_acu_1
    | core_acu_1 = add,
      mod_acu_1 = modulo[7,13],
      bus_1 = acu_1 <0> %35;

j19[j7]:reg_file_2_bgram_1 <-
  j29:reg_file_1_acu_1,
  j6[j7]:reg_file_2_acu_1
    | core_acu_1 = add,
      mod_acu_1 = modulo[7,13],
      bus_1 = acu_1 <0> %36;

j11[j7]:reg_file_2_mult_0 <-
  j14[-j7]:ram_3_bgram_0,
  j15[j7]:reg_file_2_bgram_0
    | core_bgram_0 = read,
```

```
        bus_3 = bgram_0,
        mux_2_mult_0 = pass[5,5] <0> %37;

  j18[0,j7]:ram_3_bgram_1 <-
    j10[j7]:reg_file_1_bgram_1,
    j30[j7]:reg_file_2_bgram_1
      | core_bgram_1 = write <0> %38;

  j15[j7]:reg_file_2_bgram_0 <-
    j22@1:reg_file_1_acu_1,
    j6[j7]:reg_file_2_acu_1
      | core_acu_1 = sub,
        mod_acu_1 = modulo[6,13],
        bus_1 = acu_1 <0> %39;

  j6[j7]:reg_file_2_acu_1 <-
    j6[j7]:reg_file_1_acu_0
      | core_acu_0 = passa,
        mod_acu_0 = modulo[13,13],
        bus_2 = acu_0,
        mux_2_acu_1 = pass[2,2] <0> %40;

END; { FOR j7 }

j6[1]:reg_file_1_acu_0 <-
  #1:reg_file_1_fgrom_0
    | bus_3 = fgrom_0,
      mux_1_acu_0 = pass[1,1] <0> %41;

j5[0]:reg_file_1_alu_0 <-
  j12:reg_file_1_mult_0,
  j13:reg_file_2_mult_0
    | core_mult_0 = mult,
      bus_4 = mult_0,
      mux_1_alu_0 = pass[5,7] <0> %42;

j13:reg_file_2_mult_0 <-
  j17:io_buf_reg_1_inpad_0
    | core_inpad_0 = read,
      bus_5 = inpad_0,
      mux_2_mult_0 = pass[3,5] <0> %43;

j16:reg_file_2_bgram_0 <-
```

```
  j22@1:reg_file_1_acu_1
    | core_acu_1 = inca,
      mod_acu_1 = modulo[6,13],
      bus_1 = acu_1 <0> %44;

j13:reg_file_1_bgram_0 <-
  j17:io_buf_reg_1_inpad_0
    | core_inpad_0 = read,
      bus_5 = inpad_0,
      mux_1_bgram_0 = pass[1,1] <0> %45;

IF ((NOT (j45)
    OR NOT (j48))
 OR NOT (j33)) THEN
BEGIN
  j12:reg_file_1_mult_0 <-
    j18[-1,0]:ram_3_bgram_1,
    j26:reg_file_2_bgram_1
      | core_bgram_1 = read,
        bus_6 = bgram_1,
        mux_1_mult_0 = pass[2,3] <0> %46;

  j12:reg_file_1_bgram_1 <-
    j18[-1,0]:ram_3_bgram_1,
    j26:reg_file_2_bgram_1
      | core_bgram_1 = read,
        bus_6 = bgram_1,
        mux_1_bgram_1 = pass[0,1] <0> %47;

END ELSE
BEGIN
  j12:reg_file_1_mult_0 <-
    j27:reg_file_1_alu_0,
    j28:reg_file_2_alu_0
      | core_alu_0 = add,
        sh_alu_0 = upsh[0,14],
        shmux_alu_0 = up,
        bus_6 = alu_0,
        mux_1_mult_0 = pass[2,3] <0> %48;

  j27:reg_file_1_alu_0 <-
    j18[-1,0]:ram_3_bgram_1,
    j26:reg_file_2_bgram_1
```

```
      | core_bgram_1 = read,
        bus_6 = bgram_1,
        buf_1_bgram_1 = write,
        mux_1_alu_0 = pass[0,7] <0> %49;

  j28:reg_file_2_alu_0 <-
    j44:reg_file_1_mult_0,
    j13:reg_file_2_mult_0
      | core_mult_0 = mult,
        bus_4 = mult_0,
        mux_2_alu_0 = pass[3,6] <0> %50;

  j12:reg_file_1_bgram_1 <-
    j27:reg_file_1_alu_0,
    j28:reg_file_2_alu_0
      | core_alu_0 = add,
        sh_alu_0 = upsh[0,14],
        shmux_alu_0 = up,
        bus_6 = alu_0,
        mux_1_bgram_1 = pass[0,1] <0> %51;

END;

j33:status_reg_6_alu_0 <-
  j34:reg_file_1_alu_0,
  j35:reg_file_2_alu_0
    | core_alu_0 = sub,
      sh_alu_0 = upsh[0,14],
      shmux_alu_0 = up <0> %52;

j45:status_reg_6_alu_0 <-
  j46:reg_file_1_alu_0,
  j47:reg_file_2_alu_0
    | core_alu_0 = sub,
      sh_alu_0 = upsh[0,14],
      shmux_alu_0 = up <0> %53;

...

j13:reg_file_2_alu_0 <-
  j17:io_buf_reg_1_inpad_0
    | core_inpad_0 = read,
      bus_5 = inpad_0,
```

```
      mux_2_alu_0 = pass[2,6] <0> %135;

IF j73 THEN
BEGIN
  j70:reg_file_2_mult_0 <-
    j13:reg_file_2_alu_0
      | core_alu_0 = negb,
        sh_alu_0 = upsh[0,14],
        shmux_alu_0 = up,
        bus_6 = alu_0,
        mux_2_mult_0 = pass[2,5] <0> %136;

END ELSE
BEGIN
  j70:reg_file_2_mult_0 <-
    j13:reg_file_2_alu_0
      | core_alu_0 = passb,
        sh_alu_0 = upsh[0,14],
        shmux_alu_0 = up,
        bus_6 = alu_0,
        mux_2_mult_0 = pass[2,5] <0> %137;

END;

j73:status_reg_6_alu_0 <-
  j13:reg_file_2_alu_0,
  j59:reg_file_1_alu_0
    | core_alu_0 = sub,
      sh_alu_0 = upsh[0,14],
      shmux_alu_0 = up <0> %138;

j4:reg_file_2_alu_0 <-
  j4:io_buf_reg_1_inpad_0
    | core_inpad_0 = read,
      bus_5 = inpad_0,
      mux_2_alu_0 = pass[2,6] <0> %139;

IF j74 THEN
BEGIN
  j72:reg_file_2_mult_0 <-
    j4:reg_file_2_alu_0
      | core_alu_0 = negb,
        sh_alu_0 = upsh[0,14],
```

```
        shmux_alu_0 = up,
        bus_6 = alu_0,
        mux_2_mult_0 = pass[2,5] <0> %140;

  END ELSE
  BEGIN
    j72:reg_file_2_mult_0 <-
      j4:reg_file_2_alu_0
        | core_alu_0 = passb,
          sh_alu_0 = upsh[0,14],
          shmux_alu_0 = up,
          bus_6 = alu_0,
          mux_2_mult_0 = pass[2,5] <0> %141;

  END;

  j74:status_reg_6_alu_0 <-
    j4:reg_file_2_alu_0,
    j59:reg_file_1_alu_0
      | core_alu_0 = sub,
        sh_alu_0 = upsh[0,14],
        shmux_alu_0 = up <0> %142;

  j46:reg_file_1_mult_0 <-
    j52:reg_file_1_alu_0,
    j53:reg_file_2_alu_0
      | core_alu_0 = add,
        sh_alu_0 = upsh[0,14],
        shmux_alu_0 = up,
        bus_6 = alu_0,
        mux_1_mult_0 = pass[2,3] <0> %143;

  j46:reg_file_2_mult_0 <-
    j52:reg_file_1_alu_0,
    j53:reg_file_2_alu_0
      | core_alu_0 = add,
        sh_alu_0 = upsh[0,14],
        shmux_alu_0 = up,
        bus_6 = alu_0,
        mux_2_mult_0 = pass[1,5] <0> %144;

END. { FOR time }
```

Bibliography

[1] F. Van Aelten. *Efficient Verification of VLSI Circuits Based on Syntax and Denotational Semantics.* MS Dissertation, MIT, RLE Technical Report 546, 1989.

[2] M. Arnold and J. Ousterhout. Lyra: A New Approach to Geometric Layout Rule Checking. In *Proceedings of 19th ACM/IEEE Design Automation Conference*, pages 530–536, 1982.

[3] C. Bamji. *Graph-based Representations and Coupled Verification of VLSI Schematics and Layouts.* PhD Dissertation, MIT, 1989.

[4] C. Bamji and J. Allen. GRASP: A Grammar-Based Schematic Parser. In *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pages 448–453, June 1990.

[5] M. Blum, A. Chandra, and M. Wegman. Equivalence of Free Boolean Graphs Can Be Decided Probabilistically in Polynomial Time. In *Information Processing Letters*, volume 10, pages 80–82, March 1980.

[6] I. Bolsens. *Electrical Correctness Verification of MOS Digital Circuits using Expert System and Symbolic Analysis Techniques.* PhD Dissertation, K.U.Leuven, 1989.

[7] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[8] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. MIS: A Multiple-Level Logic Optimization System. In *IEEE Transactions on Computer-Aided Design*, volume 6, pages 1062–1081, November 1987.

[9] A. Bronstein. *MLP: String-Functional Semantics and Boyer-Moore Mechanization for the Formal Verification of Synchronous Circuits.* Ph.D Dissertation, Stanford, STAN-CS-89-1293, 1989.

[10] A. Bronstein and C. Talcott. Formal Verification of Pipelines based on String-Functional Semantics. In *Formal VLSI Correctness Verification, VLSI Design Methods-II*, pages 349–366. North-Holland, 1990.

[11] R. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.

[12] R. Bryant. Extraction of Gate Level Models from Transistor Circuits by Four-Valued Symbolic Analysis. In *Proceedings of the International Conference on Computer Aided Design*, pages 350–353, November 1991.

[13] R. Bryant. On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication. In *IEEE Transactions on Computers*, volume 40, pages 205–213, Februari 1991.

[14] J. Burch. Using BDDs to Verify Multipliers. In *Proceedings of 1991 International Workshop on Formal Methods in VLSI Design*, January 1991.

[15] J. Burch, E. Clarke, K. McMillan, and D. Dill. Sequential Circuit Verification Using Symbolic Model Checking. In *Proceedings of the 27^{th} Design Automation Conference*, pages 46–51, June 1990.

[16] H. Busch and G. Venzl. Proof-Aided Design of Verified Hardware. In *Proceedings of 28^{th} ACM/IEEE Design Automation Conference*, pages 391–396, 1991.

[17] L. Claesen. SFG-Tracing: a Methodology for the Automatic Verification of MOS Transistor Level Implementations from High Level Behavioral Specifications. In *Proceedings of the 1991 International Workshop on Formal Methods in VLSI Design*, 1991.

[18] E. Clarke and O. Grumberg. Research on Automatic Verification of Finite-State Concurrent Systems. In *Annual Reviews of Computer Science*, volume 2, pages 269–290, 1987.

[19] A. Cohn. A Proof of Correctness of the VIPER Microprocessors: The First Level. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71. Kluwer Academic Publishers, 1988.

[20] O. Coudert, C. Berthet, and J. C. Madre. Verification of Sequential Machines Using Boolean Functional Vectors. In *IMEC-IFIP Int'l Workshop on Applied Formal Methods for Correct VLSI Design*, pages 111–128, November 1989.

[21] S. Devadas. Optimizing Interacting Finite State Machines Using Sequential Don't cares. In *IEEE Transactions on Computer-Aided-Design*, volume 10, pages 1473–1484, December 1991.

[22] S. Devadas and K. Keutzer. An Automata-Theoretic Approach to Behavioral Equivalence. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 30–33, November 1990.

[23] S. Devadas, K. Keutzer, and S Malik. Delay Computation in Combinational Logic Circuits. In *Proceedings of the International Conference on Computer Aided Design*, pages 176–179, November 1991.

[24] S. Devadas, H-K. T. Ma, and A. R. Newton. On the Verification of Sequential Machines at Differing Levels of Abstraction. In *IEEE Transactions on Computer-Aided Design*, volume 7, pages 713–722, June 1988.

[25] D. Dill. *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* ACM Distinguished Dissertations, The MIT Press, Cambridge, Massachusetts, 1988.

[26] D. Dill and E. Clarke. Automatic Verification of Asynchronous Circuits Using Temporal Logic. In *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 127–143, December 1985.

[27] E. Allen Emerson. Temporal and Modal Logic. In *Handbook of Theoretical Computer Science; Volume B: Formal Models and Semantics*, pages 995–1072, 1990.

[28] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man. An Efficient Microcode Compiler for Application Specific DSP Processors. In *IEEE Transactions of Computer-Aided Design*, volume 9, pages 925–937, September 1990.

[29] M. Gordon. *Hardware Verification by Formal Proof.* Technical Report No. 74, Computer Laboratory, University of Cambridge, Cambridge, England, August 1985.

[30] P. Hilfinger. A High-Level Language and Silicon Compiler for Digital Signal Processing. In *Proceedings of the Custom Integrated Circuits Conference*, pages 213–216, May 1985.

[31] W. Hunt. *FM8501: A Verified Microprocessor.* University of Texas, Austin, Tech. Report 47, 1985.

[32] J. Jain, J. Bitner, D. Fussell, and J. Abraham. Probabilistic Design Verification. In *Proceedings of the International Conference on Computer Aided Design*, pages 468–471, November 1991.

[33] J. Joyce. Formal Verification and Implementation of a Microprocessor. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 129–157. Kluwer Academic Publishers, 1988.

[34] S. Y. Kung. *VLSI Array Processors.* Prentice-Hall, Englewood Cliffs, N. J., 1988.

[35] F. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann, San Mateo, California, 1992.

[36] H. De Man. Personal Communication, November 1991.

[37] M. McFarland and A. Parker. An Abstract Model of Behavior for Hardware Descriptions. In *IEEE Transactions on Computers*, volume C-32, pages 621–636, July 1983.

[38] P. McGeer and R. Brayton. Efficient Algorithms for Computing the Longest Viable Path in a Combinational Network. In *Proceedings of the* 26^{th} *ACM/IEEE Design Automation Conference*, pages 561–567, June 1990.

[39] M. Rem, J. van de Snepscheut, and J. Tijmen Udding. Trace Theory and the Definition of Hierarchical Components. In *Proceedings of the* 3^{rd} *CalTech Conference on VLSI*, pages 225–239, March 1983.

[40] K. Sabnani. An Algorithmic Technique for Protocol Verification. In *IEEE Transactions on Communications*, volume 36, pages 924–931, August 1988.

[41] K. Sabnani, A. Lapone, and M. Uyar. An Algorithmic Procedure for Checking Safety Properties of Protocols. In *IEEE Transactions on Communications*, volume 37, pages 940–948, September 1989.

[42] R. Spickelmier and A. R. Newton. Connectivity Verification Using a Rule-Based Approach. In *Proceedings of the Int'l Conference on Computer-Aided Design*, pages 190–192, November 1985.

[43] H. Touati, H. Savoj, B. Lin, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit State Enumeration of Finite State Machines Using BDD's. In *Proc. of Int'l Conference on Computer-Aided Design*, pages 130–133, November 1990.

[44] D. Verkest, L. Claesen, and H. De Man. On the use of the Boyer-Moore theorem prover for correctness proofs of parameterized hardware modules. In *IMEC-IFIP Int'l Workshop on Applied Formal Methods for Correct VLSI Design*, pages 405–422, November 1989.

[45] D. Verkest, L. Claesen, and H. De Man. Special Benchmark Session on Formal System Design. In *IMEC-IFIP Int'l Workshop on Applied Formal Methods for Correct VLSI Design*, pages 75–76, November 1989.

[46] S. Ward and R. Halstead. *Computation Structures.* The MIT Press, Cambridge, Massachusetts, 1990.

[47] D. Weise. *Formal Multilevel Hierarchical Verification of Synchronous MOS VLSI Circuits.* PhD Dissertation, MIT, A.I. Technical Report 978, 1986.

[48] T. Witney. *A Hierarchical Design Rule Checker.* MS Dissertation, California Institute of Technology, 1981.