# A Tutorial on Digital Design Derivation Using DRS *

Bhaskar Bose[†], M. Esen Tuna, and Venkatesh Choppella

Derivation Systems, Inc.
5963 La Place Court, Suite 208
Carlsbad, CA 92008, USA
[†]bose@derivation.com

**Abstract.** This paper presents a tutorial on digital design derivation using DRS. The DRS system is an integrated formal system for the design of verified hardware. The underlying approach employs a derivation methodology in which a series of correctness preserving transformations are applied to high-level specifications in order to synthesize hardware descriptions. In this paper, we sketch the key steps in the derivation of an example circuit. The example illustrates several aspects of DRS and serves as an introduction to the derivational paradigm of synthesis.

## 1 Introduction

DRS (Derivational Reasoning System) is an integrated formal system for the development and analysis of verified digital hardware systems. The system provides design engineers with the ability to synthesize verified hardware from high-level specifications. DRS integrates an executable specification language, a polymorphic type inference system and a powerful derivation engine with existing theorem provers and logic synthesis tools to provide a formal framework for design. The derivation methodology has been applied to several non-trivial designs including a 32-bit general purpose microprocessor [1], and a fault-tolerant clock synchronization circuit [6].

## 2 Design Derivation

The primary mode of reasoning in DRS is *derivation.* Derivation is a form of mathematical proof that deals with correct-by-construction reasoning [3, 4, 5]. A series of correctness preserving transformations are used to derive an implementation from a specification. A key advantage of this approach is that it obviates the need for post-factum verification. This reduces the goal of design verification to proving the correctness of the transformations. The correctness of the transformations is established independently when the transformation is

* Research reported herein was supported, in part, by The National Aeronautics and Space Administration.

developed, rather than by the designer. The designer simply applies the transformation to a design to achieve some desired goal. In addition, derivation provides the designer the ability to explore the design space while maintaining the rigorous integrity of the specification. In this respect, while deductive (conventional theorem prover based) verification attempts to formalize a particular design, derivation attempts to formalize the design process.

## 3 Derivation of a Fibonacci Sequence Generator

DRS is an interactive formal system. This interaction provides the designer with direct control of the design process. In this respect, the system resembles a proof-checker in the sense that it automates the algebra needed for circuit synthesis, but requires interactive guidance to perform a derivation. The user interacts with the system through a series of transformation commands that is used to refine the specification from an abstract description to a concrete implementation. In DRS, a sequence of transformations is applied to an initial specification defining a ***derivation path*** towards an implementation satisfying an intended set of design constraints. Design tactics and constraints imposed by the designer sketch a complex design space with many possible paths between specification and implementation. In practice, however, the derivation path has distinct phases.

Consider the example of a Fibonacci sequence generator. The behavioral specification is obtained from the iterative definition of the Fibonacci function

$$\begin{aligned} fib(n) &= g(n,1,1) \ \textbf{where} \\ g(x,y,z) &= \textbf{if } lt?(x,2) \textbf{ then } y \textbf{ else } g(dec(x), z, add(y,z)). \end{aligned}$$

The specification is an abstract algorithmic description that defines the functionality of the circuit by specifying a sequence of operations and control decisions. The specification describes what operations must occur, but not how they are implemented in hardware.

The first phase in the derivation is to apply a series of transformations at the behavioral level. A class of transformations, called *behavioral transformations*, manipulate the behavioral specification. These transformations usually involve manipulating control and architecture in a tightly integrated relation. Some examples include transformations to achieve a desired scheduling of operations and transformations to move operations between control and architecture. The above specification indicates that the operations ***add*** and ***dec*** are executing in parallel. For the sake of the example, let us assume that our final hardware implementation is constrained to use a single arithmetic logic unit. In this phase the ***dec*** and ***add*** operations are serialized so that they may be combined into a single logic unit during structural refinement later in the derivation. This serialization (over time) is achieved by introducing an intermediate function $h$, and splitting the calls to ***dec*** and ***add*** across the calls to $g$ and $h$ resulting in

$$\begin{aligned} g(x,y,z) &= \textbf{if } lt?(x,2) \textbf{ then } y \textbf{ else } h(dec(x), y, z) \\ h(x,y,z) &= g(x, z, add(y,z)). \end{aligned}$$

From a suitable behavior description, DRS automatically builds an abstract *structural* description, denoted by a set of recurrence equations, representing an initial estimation of architecture:

$$\begin{aligned} w &= \mathbf{reg}(\mathrm{g}, \mathit{select}(\mathit{status}, \mathrm{g}, \mathrm{h}, \mathrm{g})) & \mathit{status} &= [w, \mathit{lt?}(x, 2)] \\ x &= \mathbf{reg}(n, \mathit{select}(\mathit{status}, x, \mathit{dec}(x), x)) & \mathit{rdy} &= \mathit{and}(\mathit{equal?}(w, \mathrm{g}), \mathit{lt?}(x, 2)) \\ y &= \mathbf{reg}(1, \mathit{select}(\mathit{status}, y, y, z)) & \mathit{ans} &= y \\ z &= \mathbf{reg}(1, \mathit{select}(\mathit{status}, z, z, \mathit{add}(y, z))) \end{aligned}$$

where

$$\mathit{select}([s, p0], v0, v1, v2) = \mathbf{case}\ s \quad \mathrm{g} : \mathbf{if}\ p0\ \mathbf{then}\ v0\ \mathbf{else}\ v1 \quad \mathrm{h} : v2$$

The block diagram denoting the initial structural description is shown in Figure 1. ( ⫤▷ denotes the select function. For clarity only the inputs to the selector are shown, and the status signal is omitted. ⎍ represents a register unit. )

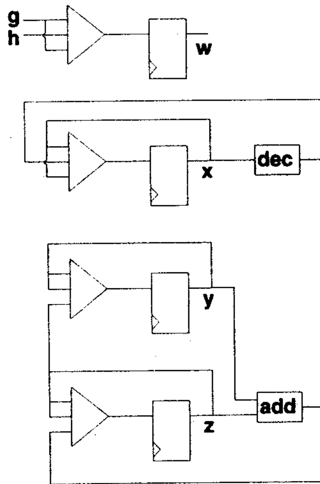


**Fig. 1.** Initial Structural Description

The structural description defines the components in a circuit and their connectivity. The specification expresses logical behavior and physical organization, but does not address electrical characteristics of the circuit. Timing is coordinated by storage elements whose behavior in turn is governed by an external synchronizing clock. Each variable now denotes an infinite sequence of values over time. $\mathbf{reg}(v, S)$ denotes the sequence of values $< v, S^0, S^1, \ldots >$ where the "reg" function introduces a delay and is interpreted as a register. The construction guarantees that given a particular sequence of input events, the structural

description produces the same output-event sequence as does the original behavior specification.

The second phase in the derivation is to refine the structural description to an architecture. A class of transformations, called ***structural transformations***, decomposes the description into a system of modules encapsulating signals as coprocesses, isolating components of the specification for verification, mapping to existing hardware components, or further algebraic refinement. In the example, the architecture is refined by subsuming the *dec* and *add* operations by a single component. The transformation is valid since these two operations do not occur simultaneously, as imposed by the earlier serialization of these two operations. The results of factoring *dec* and *add* are the synthesis of an abstract component,

$$\begin{aligned} alu(inst, op_a, op_b) = \mathbf{case}\ & inst \\ & \mathrm{nop} : ? \\ & \mathrm{dec} : dec(op_a) \\ & \mathrm{add} : add(op_a, op_b) \end{aligned}$$

and the derivation of four equations for the signals *alu_out*, *inst*, *op_a*, and *op_b* to communicate with the factored component. The ? symbol denotes a *don't care* value. The original occurrences of *add* and *dec* are replaced with the output of the factored component. The resulting system of equations is

$$\begin{aligned} w &= \mathbf{reg}(\mathrm{g}, select(status, \mathrm{g}, \mathrm{h}, \mathrm{g})) & status &= tuple(w, lt?(x, 2)) \\ x &= \mathbf{reg}(n, select(status, x, alu_out, x)) & op_a &= select(status, ?, x, y) \\ y &= \mathbf{reg}(1, select(status, y, y, z)) & op_b &= select(status, ?, ?, z) \\ z &= \mathbf{reg}(1, select(status, z, z, alu_out)) & inst &= select(status, \mathrm{nop}, \mathrm{dec}, \mathrm{add}) \\ rdy &= and(equal?(w, \mathrm{g}), lt?(x, 2)) & alu_out &= alu(inst, op_a, op_b) \\ ans &= y. \end{aligned}$$

The third phase of the derivation is to introduce a lower-level representation. A class of transformations, called ***projection transformations***, introduces a lower-level representation. In the example, the architecture is still abstract in the sense that signals represent integer values. To obtain a concrete binary description, these signals are instantiated with bit-vectors of appropriate width. Type declarations are used to project each variable, constant, and operator to a binary representation. For instance, the projection of

$$y = \mathbf{reg}(1, select(status, y, y, z))$$

to a binary representation of three bits is rewritten as

$$\begin{aligned} y_0 &= \mathbf{reg}(\mathrm{T}, select(status, y_0, y_0, z_0)) \\ y_1 &= \mathbf{reg}(\mathrm{F}, select(status, y_1, y_1, z_1)) \\ y_2 &= \mathbf{reg}(\mathrm{F}, select(status, y_2, y_2, z_2)). \end{aligned}$$

The constant 1 and signal $z$ are also projected to their respective equivalent bit representations. At this stage the entire design is at the binary level. Subsequent

transformations impose a logical and physical ordering on the design to map to a particular target technology. Algebraic transformations provide a powerful approach since the massive restructuring and decomposition necessary in reorganizing the design represent purely syntactical manipulations. Ultimately, this formal development produces a hierarchy of boolean subsystems, which are then partitioned into synthesizable subsystems. These boolean subsystems are then passed to logic synthesis tools to generate hardware realizations.

## 4 Conclusion

The DRS system is based on the philosophy that design is a reasoning process that involves analysis, deduction and generation. Although, derivation is the primary mode of proof in DRS, the system integrates with existing verification systems at several levels. In DRS, verification is necessary to establish the correctness of the specification and representations. In addition, design optimizations that are more easily handled using either mechanical theorem proving techniques or model checking are employed throughout the derivation. For example, suppose we wanted to substitute the derived *alu* specification with an efficient technology dependent implementation. Fully automatic OBDD [2] verification techniques would be sufficient to establish the equivalence between the derived *alu* and its optimized implementation.

The idealized design environment consists of multiple formal systems with secure interaction between them. Proofs in one system are interpreted as valid in another, eliminating the need to re-validate proofs across system boundaries. With this approach the designer is able to employ the most effective tool for a particular design context without sacrificing confidence in the correctness of the design.

## References

1. Bhaskar Bose. *DDD-FM9001: Derivation of a Verified Microprocessor*. PhD thesis, Indiana University, December 1994.
2. Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. In *IEEE Transactions on Computers*, volume C-35, pages 677–691, August 1986.
3. R. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24:44–67, 1977.
4. Steven D. Johnson. *Synthesis of Digital Designs from Recursion Equations*. The MIT Press, Cambridge, 1984.
5. Steven D. Johnson. Manipulating logical organization with system factorizations. In M. Leeser and G. Brown, editors, *Hardware Specification, Verification and Sythesis: Mathematical Aspects, Lecture Notes in Computer Science*, volume 408, pages 260–281. Springer, Berlin, 1989.
6. Paul S. Miner, Shyamsundar Pullela, and Steven D. Johnson. Interaction of formal design systems in the development of a fault-tolerant clock synchronization circuit. In *13th Symp. on Reliable Distributed Systems*, October 1994.