A Graphical Approach to High-Level, HDL-Based VLSI Systems Design

Dr. James Davis - Knowledge Based Silicon Corporation, Columbia, SC, USA
Christopher Sheldon - VIZEF Limited, Bucks, UK

The New Realities of VLSI Systems Design

The increasing acceptance of synthesis technology to implement and optimize high-level circuit designs is redirecting designers to focus more on issues of product specification through high-level HDL-based definition. Synthesis tools are now available to implement efficient silicon for designs embracing complex algorithms, control structures and extensive datapath functions—all from a single HDL definition of the design intent.

This redirection of focus towards new levels of design abstraction is promising single chip designs of 100K gates to be created in the same time scale as designs of 10K chips using traditional structural design methods. This increase in design productivity is being driven by the competitive need to achieve shorter product life cycles and ever-increasing gate count potential within ICs. In addition, over the next ten years we can expect to see product life cycles of electronics-based products shrink to under two years with the gate counts of the individual ASICs populating the electronic systems approaching 1 million gates.

As you can see in Figure 1, this increasing gate count is mitigating an increase in overall design complexity, while decreasing product life cycles are mitigating ever shorter development cycles. To effectively manage the change being imposed by these factors, the productivity of electronic designers using HDL techniques must increase at a greater level than the three times over gate-level design being realized today.

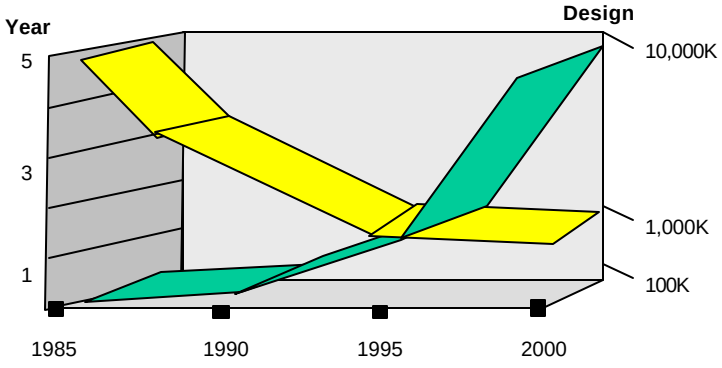


Figure 1. The Productivity Challenge

The business needs for greater designer productivity will demand the specification and detailed design of new chips in excess of 500K gates in around three to six months, compared to today's averages of six to nine months for 100 gates. Achieving these productivity increases will require a re-focusing of design activities around the issues of "How do I quickly and unambiguously specify my design objectives?" and away from "How do I implement my design objectives?"

Attaining larger designs more quickly, with greater success and according to schedule is not seen as a "device packaging" issue. It is seen more as an issue of managing design complexity. The solution discussed in this paper involves giving designers a well-proven graphical alternative to HDL text for capturing and manipulating high-level design information, an alternative that is more intuitive than text.

Creating Effective High-Level Specifications

The old expression "a picture is worth a thousand words" is indeed true. It has particular merit for designers of logic devices such as ASICs and FPGAs who have used graphical approaches in the past and who must now cope with the gain in abstraction, as well as the loss in design clarity and intuitiveness, associated with using HDL-based design methods coupled with logic synthesis.

To bring intuitiveness back to high-level design, there are many graphical approaches competing for designers' attention. There is also much confusion about the various levels of design abstraction, and which level is right to attack the high-level design problem. The answer depends on the practitioner. System architects will use abstractions that are suited to the problems they are trying to solve, whereas VLSI logic designers will use abstractions that allow them to solve their problems. These two domains are complementary, but not the same.

The Importance of the Right Abstraction

To support the paradigm shift to high-level design, new classes of graphical tools and methodologies are emerging that complement today's accepted HDL-based design processes. Some tools are optimized for high-level behavioral system definition, whereas others are optimized around the issues of defining the behavioral and functional implementation of a defined design architecture.

The importance of selecting the "right" abstraction cannot be stressed enough. There is generally confusion in this selection process, because the prominent HDL languages used today support modeling of a design at different levels of abstraction. Consequently, the available graphical tools that seek to assist the design practitioner also support modeling at different levels.

In order to select the appropriate level at which to model an application, and thereby aid in selecting the appropriate tools and methodologies, it is important to understand how different abstraction levels are related to each other. One way to view this is shown in Figure 2, which depicts the hierarchy of levels for design specification as practiced by systems architects, hardware designers, and software programmers.

One important difference between these design practices is that hardware design (unlike software design) demands that the designer note that hardware functions happen at specified intervals, are linked to explicit clocking cycles, and are always active unless explicitly deactivated. The need to consider these forms of hardware-specific design issues has a

direct on the class of design specification tools that can be effectively used to support the ASIC design process.

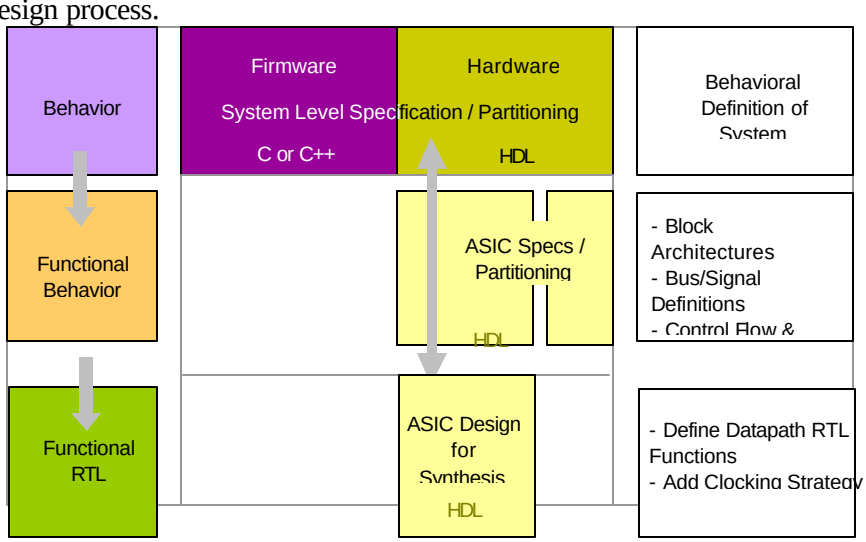


Figure 2. Design Specification Hierarchy

The Design Specification Hierarchy shown in Figure 2 highlights the steps involved in migrating an overall system specification/definition into a form suitable for today's RTL-based synthesis tools. Although the information indicated in this figure can be represented in many different ways, the relevant point is that there is a distinction between purely abstract, or "domain behavior," and system behavior as defined by the underlying implementation medium, referred to here as "functional behavior."

Many domains, such as Telecommunications or Computer Architecture, have high-level models that allow system designers to abstract and model the essential behaviors relevant in the domain. Invariably, these models are mathematical or computational in nature, relying on specific concepts and relationships associated with the domain to give them meaning. When taking such abstract descriptions to software, the transformation from abstraction to implementation is fairly smooth.

This is because software systems are basically layers of "virtual machines" built one on top of the other, where the system ultimately executes on some form of hardware. However, since software itself is nothing more than mathematical abstraction, software components can be effectively layered such that high-level, complex abstractions can be built on virtual representations at lower levels. These lower levels hide the complexity of the model and impose the mappings from high-level abstractions to their implementation at lower levels.

As shown in Figure 3, software-based systems take advantage of this layering to abstract away the underlying constraints of the real-world hardware environment. The transformations from application to implementation have been built up over many years, and are very smooth. However, when layers in this model are skipped, as is the case with ASIC and FPGA design, the direct transformation from software abstraction to hardware

implementation is not smooth. This becomes exacerbated as the size and complexity of the system abstraction gets larger.

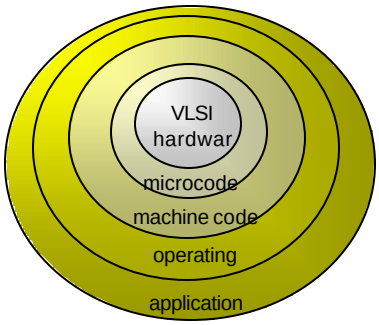


Figure 3. Systems Built on Virtual Machines

Thus, for mapping high-level abstractions directly to hardware, as in the case of designing VLSI custom logic using an HDL-based design approach, there are no intervening layers on which such high-level abstraction can run. If the intervening layers of a software abstraction are removed, a "conceptual mismatch" is present, making it difficult to effectively and efficiently map these higher-level abstractions to VLSI circuits.

Using an effective graphical design methodology helps designers model their application in terms of concepts that abstractly represent the important elements of the design description. The benefit of selecting the right graphical approach is that it helps eliminate the "conceptual mismatch" that is inherent in the design process when designers attempt to bridge too great a gap in levels between highest level abstraction and VLSI custom logic implementation.

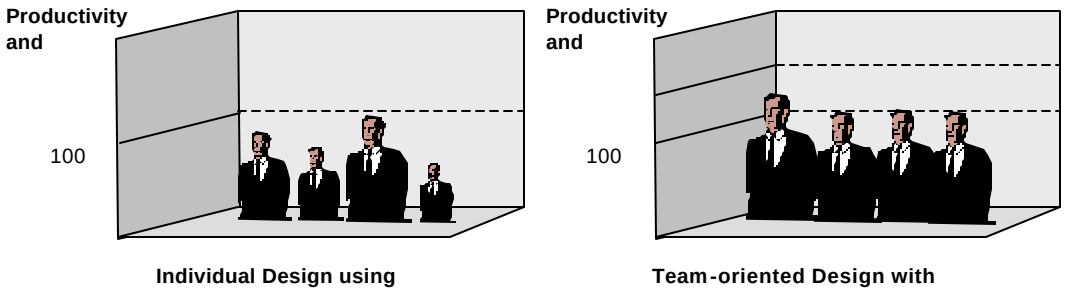
Improving Design Team and Performance

The tools and methodologies that address the complexity problem of the ASIC/FPGA design community are those that optimize the bottom two elements of the specification hierarchy shown in Figure 2. ASIC designs may be members of a broader team, including systems architects and designers. Whereas the architects and systems designers may be responsible for specifying a system's behavior at its highest and most abstract levels, it is up to the ASIC and FPGA designers to effectively realize the core VLSI components of those systems.

A logic designer's efficiency is best leveraged by the application of design tools that automate the migration of this high-level "system" design intent directly into synthesis. The designer needs tools that take a predetermined architectural understanding of the design intent and put it into a form that creates "correct-by-construction" synthesizable RTL code. In addition, such tools must write HDL code in a way that produces efficient circuits as a result of synthesis (in terms of area, speed, power, fitting a target device, etc.).

High-level ASIC/FPGA design tools and methodologies need to get good hardware designers to realize synthesizable designs as effectively as, or very close to, those realized by

the company's best HDL programmers. Designs today are usually realized by teams ranging in size from 4 to 20 engineers. The design efficiency of such a team is "gated" by the capabilities of the "least efficient" designer—i.e., the "lowest common denominator"—in the project design team. In many teams, the best hardware designer is often not the best HDL



programmer, and the best programmer is seldom the best hardware designer.

Figure 4. Leveling the Playing Field

As indicated in Figure 4, individual members of a design team will perform at different levels of competency. This basic measure of competency in a design team population could be plotted under a statistical Bell Curve. It follows that not all designers on the team (or in a given company) will have the same competency in HDL-based design.

The goal of facilitating team-oriented design is to "level the playing field," giving all members of a design team access to tools and methodologies that enable them to reach higher levels of design productivity and performance in HDL-based design approaches. However, such tools should not inhibit the performance of those in the team that exceed the norm.

Graphical specification and analysis tools supporting HDL-based design methods overcome the fundamental problem of variability of capabilities in HDL programming. Using such graphical tools, HDL code becomes the result of the design definition and can be considered the vehicle for communicating the design between synthesis and simulation, not the primary vehicle for expressing the hardware design intent itself.

This has a fundamental impact on design productivity. Designs created this way are:

- Easier to define, since designers focus on the important functional aspects first
- More readily understood, since graphics are used rather than text
- Easier to verify and validate, since behaviors are easier to follow as graphics
- Better documented for design review, since there is no ambiguity of meaning

An Effective High-Level Methodology

Creating effective high-level design specifications for VLSI custom logic design applications involves selecting the right level of abstraction and adopting an approach that can be easily applied across a team of designers. We next examine what constitutes an effective methodology for realizing the benefits of high-level custom logic design.

Partitioning a design

An effective high-level design methodology for VLSI-based systems should be built around the principle that designs consist of a series of time-dependent control actions interacting with, or operating on, data values or combinatorial logic functions to define the design intent. In digital systems, these concepts are embodied in the notions of the **control path** and **datapath**. Memory is also an essential component for modeling complete VLSI-based systems. Figure 5 depicts how elements of a design can be viewed.

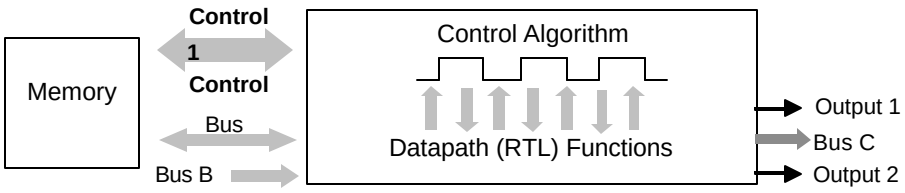


Figure 5. Partitioning a Block into Control Path, Datapath and Memory

The control path sequences and schedules the actions that enable the transformation of data in the datapath. The collection of these time-ordered operations in the datapath gives the design its overall behavior. The control path functions are generally modeled using the Finite State Machine (FSM) model, in which actions are progressed in time from one state to the next in sequence with a clock. State changes are established in concert with datapath functions that perform logical operations.

Introduction to flowdiagrams

The notation that is highlighted in this paper is the **flowdiagram**. The flowdiagram notation and methodology is an extension to the original Algorithmic State Machine (ASM) chart, developed in the 1970s at Hewlett-Packard. The major extensions to the ASM chart for flowdiagrams are in the areas of supporting a full semantic language and the macro-function components. With flowdiagrams, a designer can model both control path behavior and the sequencing and scheduling of datapath operations, according to either synchronous or asynchronous events.

The control path is modeled as either a FSM or a Stored Program Machine (SPM). SPMs are the basis for creating complex designs exhibiting program-based behaviors, such as those found in microprocessors. The datapath of a design is modeled relative to the control sequencing as a Register Transfer (RT) model. The RT model represents the datapath

operations as a sequence of assignment operations that are ordered and scheduled according to their placement on the flowdiagram.

Comparing two graphical methodologies

There are two distinct graphical methodologies for supporting this generalized approach to design representation for VLSI custom logic design: state (or "bubble") diagrams and flowdiagrams. Both approaches claim to improve designer productivity.

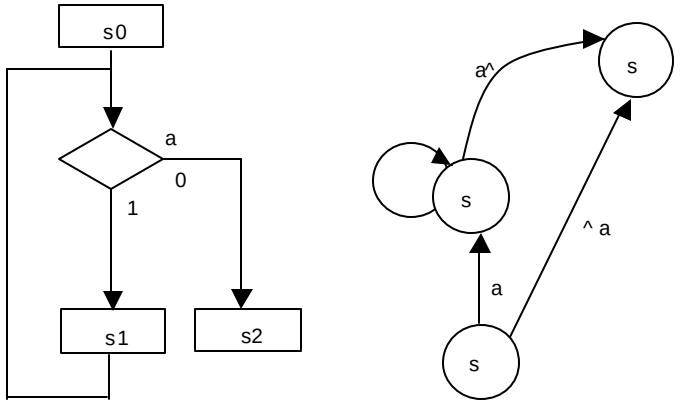


Figure 6. Flowdiagram and Equivalent Bubble Diagram

As shown in Figure 6, some similarities exist in the graphical depiction of state machines between flowdiagrams and bubble diagrams, but there are also many differences. The strengths of the ASM-based flowdiagrams notation over the more traditional bubble diagram are summarized as follows:

- **It is easier to model large control path description with flowdiagrams**. State "bubble" diagrams get messy if there are several states, because you must draw each state transition separately. The diagrams are inherently unorganized. Even if the state diagrams supports hierarchy, only a limited amount of information (about 3 to 4 states) can be modeled efficiently in a state diagram before the complexity makes it difficult to read. On the other hand, flowdiagrams can model greater complexity more easily. You can break a flowdiagram across multiple sheets, making it easier to read. Flowdiagrams also support hierarchy, reducing the complexity of a diagram by hiding details at a lower level.

- **Flowdiagrams more accurately capture the flow of control** Many complex designs have control flow diagrams that are difficult to understand. A state diagram cannot show the flow of control, since it does not order the state transitions or represent the movement of states in an intuitive fashion. Because it organizes the state actions in an algorithmic form, a flowdiagram makes it easier to see the flow control. Using a flowdiagram, you can see the sequencing of states, the order in which they are visited, and the priorities of the transition and output conditions. Flowdiagrams force the designer to completely specify all information, leaving no ambiguity.

- **Flowdiagrams model multi-way branch conditions**. State diagrams can only model binary decisions for state transitions or Mealy outputs. Flowdiarams can model binary and multiway branch conditions using Case constructs, making it more natural to model complex algorithms.

- **Flowdiagrams can model both control path and datapath**. Using a patented approach to modeling datapath through a macro-function library, you can capture both the flow of control and the sequencing and scheduling of the data path operations. This allows flowdiagrams to capture more of the design specification than state diagrams. The state diagram only models control behavior, using the Finite State Machine model. The flowdiagrams incorporates two models—the Finite State Machine model and the Register Transfer model—unified into a single representation.

Graphical Modeling using the Flowdiagram Methodology

Definition of a flowdiagram

The ASM-style of design has been commercialized in the Nimbus product. The Nimbus product uses the *flowdiagram* notation, an extension to the basic ASM chart. Flowdiagrams have constructs for visualizing state machine-oriented control, as well as datapath and memory operations. An example is shown in Figure 7.

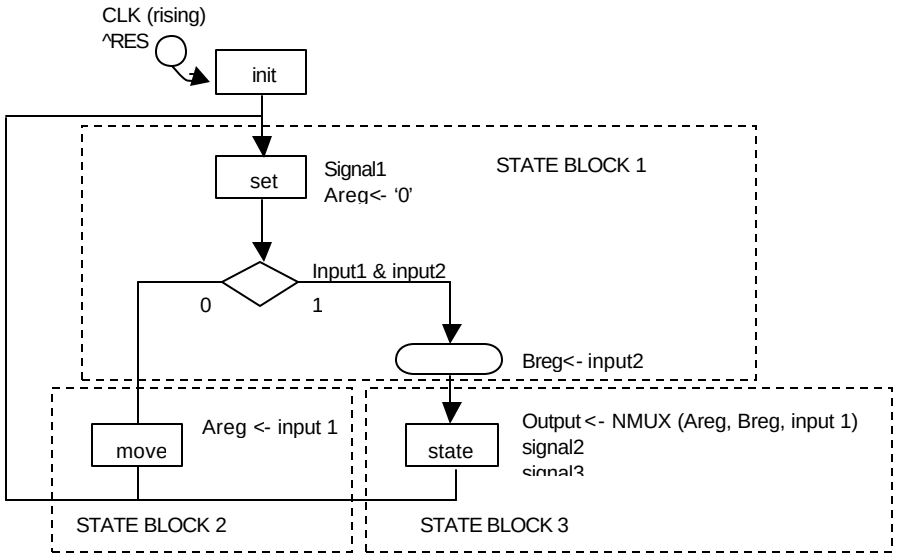


Figure 7. State Block Structure of a Flowdiagram]

In flowdiagrams, the rectangles represent *states of the system*. The diamonds represent *binary decisions*, the outcomes of which determine next state and conditional outputs. The ovals represent *conditional outputs*, dependant on both present state and input conditions. Expressions to the right of the flowdiagram objects express the actions to be carried out (for states and conditional outputs) or the conditions of interest. Branching and

conditional outputs can either be gated by binary "if-else" decisions or multivalued "case" decisions.

Specification of datapath in a flowdiagram

In Nimbus, the basic state machine specification can be augmented to express complex datapath actions. The designer expresses datapath assignments and assertions in terms of a well-proven Register Transfer model. This model is "expression oriented," in that a designer specifies datapath intent by defining assignment expressions representing the various stages of the datapath sequence that are under the control of the state machine.

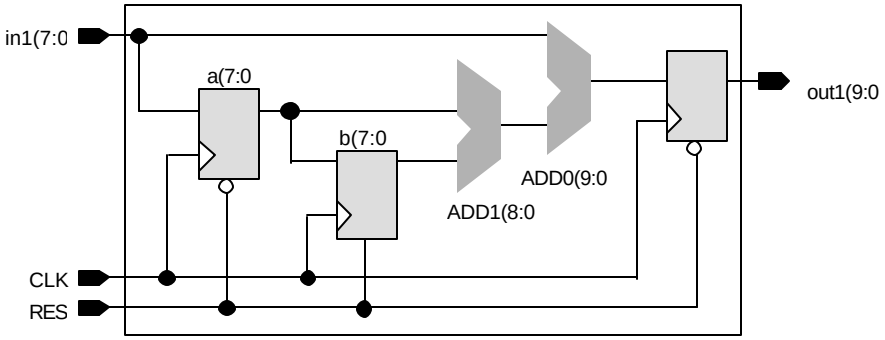


Figure 8. Structure of Datapath Description in Flowdiagrams

The flowdiagram methodology, as used in Nimbus, takes advantage of a library of generic digital macro-function operators to implement the datapath operations. Boolean logic operators, arithmetic operators, steering function operators, and register operators are defined.

In digital systems, all complex datapath functions are built up of these functional primitives. The complex datapath descriptions of a VLSI design can be constructed using the basic set of such macro-function primitives. Boolean functions are the standard AND, OR, NOT, etc. Arithmetic functions include ADD, SUB, etc. Steering logic functions include Multiplexors and Decoders. Data shifting is realized with operators such as Shift Left, Shift Right, etc., and bus manipulation is supported with operators for Concatenation and Rotation.

When the designer implements a datapath assignment from this library of macro functions, the selected functions are automatically scaled to the bus dimensions in the design file. Rule checking is implemented to ensure bus widths and signal conditions are compatible in the context of the selected operator, and that timing and metastability constraints are not violated. This approach eliminates the need to implement free-form code for expressing lower-order RTL operations and creates a design representation that is only correct-by-construction, but will produce well-structured code for synthesis. Since the generation of effective and efficient RTL-level HDL code for synthesis is a primary requirement of such design tools, the benefits of the powerful pre-optimized datapath macro-functions referenced above cannot be overstated.

Relating flowdiagram behavior to design structure

The Nimbus tool environment supports the design definition for synthesis process through a wide range of complementary functions beyond the basic ASM-oriented flowdiagram concepts described above. As indicated in Figure 9, it is possible to use other graphical represention to facilitate capturing the design intent for purposes of managing design complexity.

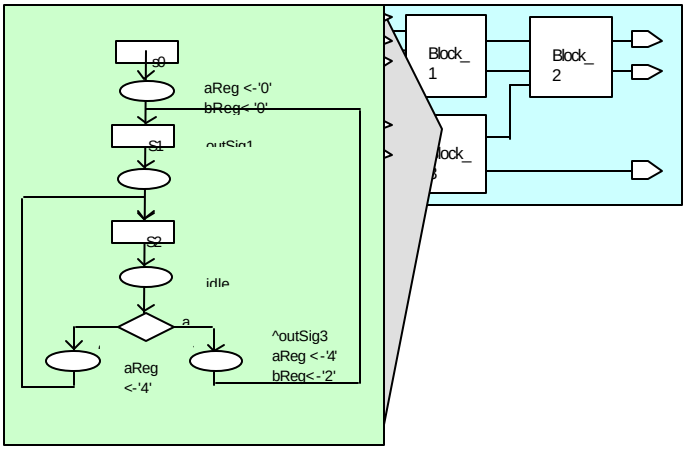


Figure 9. Combining Function, Structure and Behavior in Multiple Views

The design decomposition can be managed from a hierarchy of design "blocks" that embrace one or more flowdiagram design threads. The distinction between block diagrams and flowdiagrams is that flowdiagrams capture the functional behavior of the system components, whereas the block diagram captures the structural relationships between the various hierarchically decomposed modules. Interactive communication between hierarchical design blocks and multiple design threads is allowed, enabling the designer to support and clearly differentiate structural and behavioral hierarchy within the context of a single composite design representation.

Code Generation from Graphical Specifications

Since the objective of using high-level graphical tools is to produce efficient code for synthesis into VLSI custom logic devices such as ASICs or FPGAs, the code generation aspects of the tool used to create the design definition is critical to its success in the real world.

The important elements needed in the HDL code generation process for effective high-level design are:

- Efficiency of code for purposes of logic synthesis
- Semantic and syntactical accuracy of code for differing vendor-specific simulation and synthesis environments

- Ease of relating generated HDL code to the original graphical representation, allowing easy modification

- Flexibility of HDL coding style to meet above hardware technology constraints

Many of these objectives are interrelated. There are large differences in the performance of synthesis tools among different vendors. A key point when considering the use of graphical design approaches is whether the high-level specification of the design is reasonably independent of coding style issues. This separation of design intent from coding style allows a designer to focus on capturing the design intent, ensuring that the design is functionally correct, before considering how the code should be written for the particular synthesis tools, target device type, or coding practice. The latter issues should be available at the push of a button.

If the designer is aware of the lower level technology for implementing the design at the code generation stage, several code output options should be available to improve the effectiveness of the design created. State machine encoding styles should be set, clocking strategies should be flexibly determined, asynchronous datapaths should be left as wires or implemented with latches.

These detailed design decisions are essential aspects of improving the efficiency of the overall design process, and need to be carefully reviewed when determining the class of tools used for design specification. A good design concept will always remain just that if there is no cost effective and productive way of implementing it in silicon. The key to all graphical tools used to support design definition of ASICs and/or FPGAs is how well they traverse the hierarchy of design decomposition from concept to final silicon.

Today's state-of-the-art in synthesis demands that rigorous and explicit definitions of the design in terms of control sequencing and datapath scheduling. The flowdiagram methodology, and the Nimbus tool set that supports this methodology, is particularly adept at accommodating these needs, while allowing high levels of design definition for complex VLSI custom logic-based designs to be cost effectively implemented.

Summary

A complete high-level graphical system design toolset must accommodate many independent and yet interrelated issues to ensure real world complex design constructs are handled correctly, and in a manner conducive to improving design efficiency and quality.

To effectively provide real value in the design process, the tools must also be capable of representing the design intent in a natural form, as though the designer were describing the design behavior to a colleague at a design review. Any approach must allow to design to conceptually "map" between three different design levels at the same time: the high-level description, the HDL description, and (often) the gate-level description. Flowdiagrams allow the designer to easily map between these three levels naturally.

No tool can eliminate the need for the engineer to produce a meaningful and correct design specification. They can, however, make dramatic differences in enabling original design intent to be accurately and efficiently implemented into working silicon. To realize this goal, a high-level design environment must provide a sound methodology in which a designer can easily, efficiently and unambiguously express a design intent, support capabilities in the tool that enforce a "correct-by-construction" design process, and allow the design to explore the "space" of possible design solutions easily so that changes to a high-level design description can be made quickly.

The key functions needed in a graphical design capture and analysis toolset can be summarized as follows:

1. Moore and Mealy FSM modeling techniques with explicit graphical representation
2. Flow chart oriented conditional decision branching constructs
3. Easy definition/editing of buses/signals (types, initialization, values, and widths)
4. Support for generic scalable datapath functions
5. Structural and behavioral hierarchy with "push and pop" through design levels and views (Block, Behavior, HDL Code)
6. Support for parallel and concurrently active design flows
7. Interactive graphical simulation of design and creation of test benches and metrics for measuring functional coverage.
8. Optimized RTL level code generation for target synthesis/simulation environment

References

1. S. Tanenbaum, Structured Computer Organization, Prentice-Hall, Inc., 1976.
2. C. R. Clare, Designing Logic Systems Using State Machines, McGraw-Hill, 1973.

   Clare's text is the original treatise on Algorithmic State Machine (ASM) methodology, based on his use of ASM notation at Hewlett-Packard.

3. William I. Fletcher, *An Engineering Approach to Digital Design*, Prentice-Hall, Inc., 1980.

   Introduces the use of "flow diagrams" to model high-level control specifications. Compares flow diagram notation to the use of state diagrams, encouraging the use of both notations for advanced design problems. One of the best books on general logic design; quite comprehensive.

4. John P. Hayes, *Computer Architecture and Organization*, McGraw-Hill, Inc., 1978.

Uses flowcharts for describing and analyzing high-level control behaviors for computer architectures. Also describes the basic register transfer (RTL) notation similar to that used in flowdiagram expression language. Establishes relationship between control flow and data path sequencing from a methodology perspective.

5. John P. Hayes, *Digital Logic Design*, Addison Wesley Publishing Co., 1993.

The most up-to-date treatment of ASM-style methodology. Integrates control and data path design into single methodology, using ASM-style flow charts. Also uses state diagrams, and compares state diagrams to ASM charts. Prefers using ASM flowcharts for discussion of advanced design topics. Best description on register-level design, relating it to both HDL-based design and use of ASM-style methodology.

6. David Protheroe, *Design of Logic Systems*, Chapman Hall, 1992.

Recent influential British text on the use of ASM-style methodology in systems design applications.

7. Herbert Taub, *Digital Circuits and Microprocessors*, McGraw-Hill. Inc., 1982.

Comprehensive book on design topics. Uses flow diagrams and state diagrams equally in the text, but focuses flowdiagrams on higher-level specification of design behavior because of their algorithmic description capability. Also introduces register transfer notation similar to flowdiagrams.

