

Object-Oriented System Engineering: A Method for Managing VHDL Development

Eric Laquer
President Q Systems Inc.
Feasterville, PA 19053

Abstract

Object-Oriented System Engineering (OOSE) is a mature, proven system development methodology [1] that, when applied to VHDL-based hardware development programs, provides extraordinary improvements in quality and efficiency. The process offers intuitive, highly efficient graphical representations of complete systems at various levels of definition to illuminate otherwise opaque technical issues.

As with VHDL itself, OOSE constructs can be applied with equal ease at multiple levels of abstraction to document requirements, design and implementation. OOSE is a malleable framework designed to be tailored to meet a variety of development scenarios guiding the application of resources, methodologies and tools. It is an ideal platform for managing quality and maintaining development efficiency across many product generations.

This paper presents an introduction to OOSE as applied to challenges familiar to integrated circuit developers.

Introduction

Hardware development is a creative process, forming a very detailed and unambiguous design from a less detailed and more ambiguous requirement. To accomplish this transformation, engineers, by training and by experience, follow some step-by-step process that includes:

- 1) Preparation of specifications
- 2) Bottom-up and/or top-down design
- 3) Support manufacturing, marketing and customer constituencies.

OOSE offers a formal process for accomplishing these steps. It exists for the sole purpose of managing complexity and augmenting communication between designers, design teams and their constituencies. Documents produced during a particular OOSE development cycle can significantly reduce the effort required to produce future generations of similar products [1]. It offers greatest value for product developments likely to encounter changing requirements, various implementations and which involve complex technology.

Foundations

The process of developing high technology equipment is changing rapidly with the introduction of new tools and techniques every month. Most such tools are based on accepted design practices and paradigms. As such they improve productivity by applying computer power to more efficient manipulation of symbols already accepted as representative of real-world devices. OOSE, on the other hand, is based on the premise that the symbols and diagrams used today are inadequate representations of the high technology devices we are now capable of producing.

To produce more powerful representations, OOSE bases its constructs on immutable aspects of the development process that are unlikely to change dramatically in the future. Understanding these 'roots' and accepting that they are solid is essential to justifying the application of OOSE.

The question then becomes, how does OOSE presume to rest its existence on some bedrock of unchanging truth? The answer lies in the historical context of the development work now taking place in the relatively young fields of hardware and software development. Both fields , while rapidly progressing, have not yet

matured to the degree that other industrial processes have. This maturity is fundamental since it allows practitioners to produce high quality results reliably and efficiently. Two concepts emerge at the core of industrialization; the use of standardized components and the acceptance of abstract descriptions.

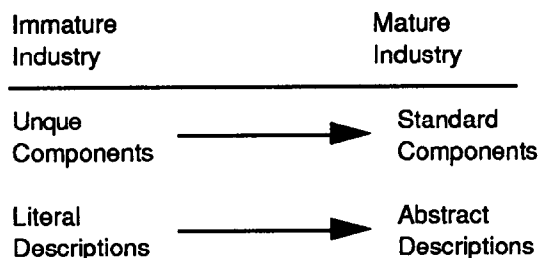


Figure 1 - Industrial maturity.

As an example of a mature industry, consider home construction. An individual buying a home accepts as output from his builder a set of plans that he accepts as a 'complete' description of the new home he is about to have built. All involved parties, including the banker, accept the plans as representative of the ultimate dwelling even though they represent only a high-level abstraction of the collection of nails, boards and shingles that will ultimately make up the house. Further, the plans call out standardized components like studs, sinks and water heaters. These components again are accepted as having value based on abstract representations and a shared acceptance that, when completed, the result will have value greater than the sum of the component parts due to its performance of abstract functions (ie. provide 2,000 square foot dwelling).

Getting back to our industry, hardware designers quickly grasped the power of standardized components and made extremely quick advancements based on TTL functional, electrical, and packaging standardization. Microprocessors, RAM and other VLSI further accelerated this industry, but did not result in an accepted, abstract way of representing a completed design.

Meanwhile, the software industry was following a somewhat different course. Higher-level abstractions were being applied, but they were built on the concepts of transformational

calculi, not of pre-packaged functionality. High level languages enable short-hand description of complex structures but do not, of themselves, offer the ability to package functions such that large-scale reuse is encouraged.

So, by the end of the 1980's, the software and hardware industries were simply struggling to accomplish the gains made by the other. Software engineers clamored for the encapsulation and packaging offered by object-orientation and hardware engineers reached for the transformational power of hardware description languages (HDL) driving logic synthesizers.

| | 1980s | 1990s |
|----------|----------------------|---------------------------|
| Hardware | VLSI Components | HDL Synthesis |
| Software | High-Level Languages | Object-Oriented Packaging |

Figure 2 - Two industries mature.

This gross simplification was done in order to expose the fundamental reasons certain tools exist. This context is well addressed within the framework of OOSE [1] in terms of the relationship between architecture, methods, processes and tools but will be discussed no further here.

OOSE origins were the result of a hardware engineers struggle to manage the exploding complexity of a telecommunications software-intensive project. As such, it not only has natural appeal to hardware engineers, but also addresses directly the issues involved in managing changes to the process itself. In an industry in which markets and tools are subject to rapid change, quality must be maintained via a process that is less time variant. OOSE is intended to be the least time variant element of an organization's development process.

VHDL Context

If OOSE is the most stable element of an ideal development process, what is the least? Without a doubt, to any professional who has

managed a conversion process, the most volatile elements of a design are the most literal, detailed descriptions of structure. To make the point more strongly, consider the difference in value between machine executable software versus source code. In hardware development, the transient value of physical mask sets (for example gate-array metal layer layouts) are a more difficult starting point for redesign, and therefore have less long-term value than the more abstract descriptions from which they were produced. Most designers recognize the desirability of capturing higher-level representations at each stage of a product's development and VHDL designers might maintain many [2], as shown in figure 3.

| Deliverable | Utility |
|--------------------------|---------------|
| Top-level schematics | Design Reuse |
| Behavioral - VHDL | Documentation |
| Register Transfer - VHDL | Synthesis |
| Structural - VHDL | Manufacturing |
| Testbench - VHDL | Test |

Figure 3 - Early deliverables have utility

OOSE encompasses not only the stages of development involved directly with VHDL source code, but also the earlier steps at which the design takes its most basic form. Towards the end of this discussion, we will revisit the stages where VHDL source exists . Examples of how the code can directly (and potentially automatically) be derived from earlier, more abstract representations will be left to another paper.

Model Building

OOSE is based on the systematic series of steps during which an idea becomes a fully implemented and tested system. The process is broken into major phases. Each phase delivers at its conclusion a model representing the entire system performing its required functions. Figure 4 shows the progression.

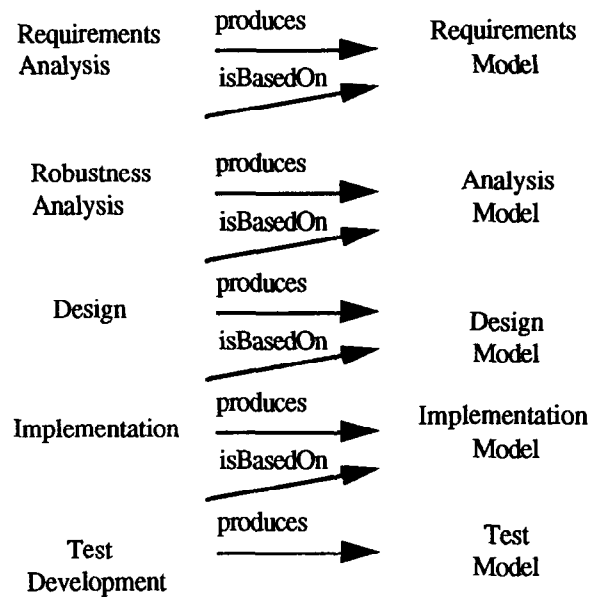


Figure 4 - Model Building

For VHDL developers, Figure 5 offers a map from OOSE to familiar documents.

| OOSE | VHDL |
|----------------------|---------------------------|
| Design Model | Top-level schematics |
| | Behavioral - VHDL |
| | Register Transfer - VHDL |
| Implementation Model | Structural - VHDL |
| Test Model | Testbench - VHDL |
| | Test programs and vectors |

Figure 5 - OOSE to VHDL map.

As can be seen, the end of the process is already well handled by 'traditional' application of existing VHDL technology

Requirements Analysis

The collection and documentation of requirements was the original charter for VHDL but has received inadequate attention. Historically more projects fail as the result of specification ambiguities than fail as a result

of inaccurate implementations. OOSE offers a systematic way of handling requirements throughout a product's lifecycle.

The process of requirements specification writing is pragmatically done by engineers familiar with the referenced standards and with specific experience (therefore prejudices) as to appropriate ways that the ultimate product will be internally structured. OOSE requirements analysis involves exactly the same process, only an effort is made to make prejudices and standards references explicit and traceable. This can actually reduce the work involved since creating references to existing descriptions is easier than duplicating those details in a new document.

Requirements specifications can be the most stable entities relative to time and can serve a great many functions in addition to directing development towards fulfillment of the correct system. As products evolve and those functions are revised, the requirements model provides a map for the development team to an appropriate set of revisions that fulfills the new requirement and that maintains as much of the integrity of the original design as possible.

To begin a requirements model, the system must be described in terms of its relationship to external entities. Figure 6 shows a block diagram of a system that communicates through two busses and a serial port.

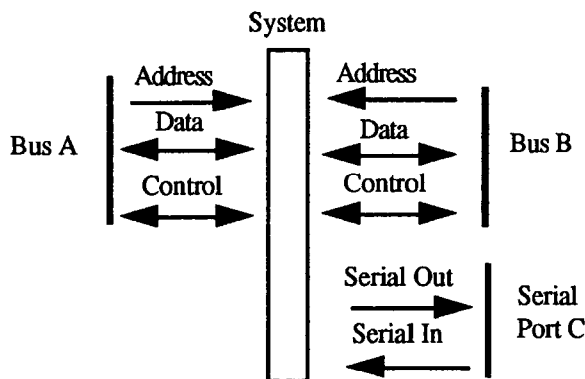


Figure 6 - System in its environment

OOSE includes the concept of an ACTOR being some external entity that can initiate actions within a given system. Actors are abstractions of real-world structures therefore they hide (or, a better term, encapsulate) the details. Figure

7 shows the same system with the interface details hidden.

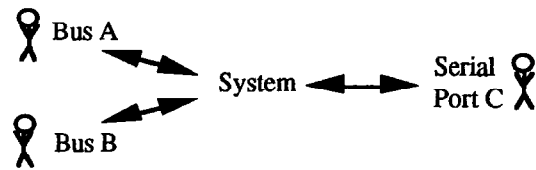


Figure 7 - Interfaces are 'Actors'.

Without reference to the specific intricacies of each interface, the specific scenarios describing the desired system performance are listed. This set of scenarios should not be extraordinarily long, nor should it omit important functions required during the entire product lifecycle. For example, if JTAG full-scan testability is required, it should be included as a single scenario in the initial requirements model. OOSE refers to these scenarios or protocols as 'USE-CASES'. Figure 8 shows a set of them that describe the system

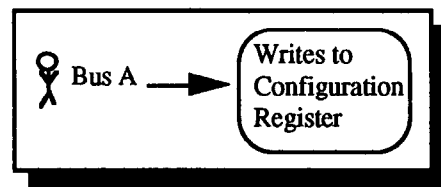


Figure 8 - 'Bus A' writes to 'Configuration Register'.

The USE-CASE diagrams contain brief, structured descriptions of transactions. The structuring, applied systematically and naturally associates rich semantics with the simple syntax of the descriptions. The sentences should use nouns and verbs which are unambiguous and are tied to application-specific meaning and to each other. To describe semantic relationships, software developers have come to use entity-relationship diagrams. Figure 9 shows a brief example.

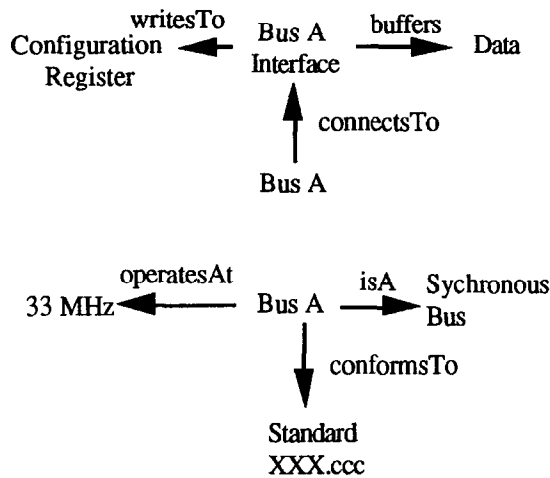


Figure 9 - Entities and relationships established

A balance is struck between brevity and detail, between casualness and rigor, and between abstractness and utility. Useful models refer to tangible things, known to developers of similar systems built in the past. Of course if we are to produce innovative new products, we want to be able to deviate from old architectures. Unfortunately, any useful requirements model implies something of the internal structure of the ultimate design.

Additional abstraction to remove all such dependencies raises the model to a pedestal from which nearly all participants in the development process feel excluded. One outstanding feature of OOSE is that it does not interfere with the natural process most designers follow. It is totally natural for experienced designers to have prejudices based on previous design experiences and the process of building a Requirements Model can be built firmly on the existing foundations. Following versions of the system model might revise these 'first cut' decompositions for specific, well-documented reasons and thus most potential innovations are possible derivations from a reasonably well-done Requirements Model.

In any case, the 'invented' internal entities have to behave together such that the system as a whole behaves as it must (according to its use-case descriptions). To insure that a thorough understanding has been captured, an INTERACTION DIAGRAM is created for each

use-case. Figure 10 shows a set of such diagrams that describe how the system fulfills each use-case requirement.

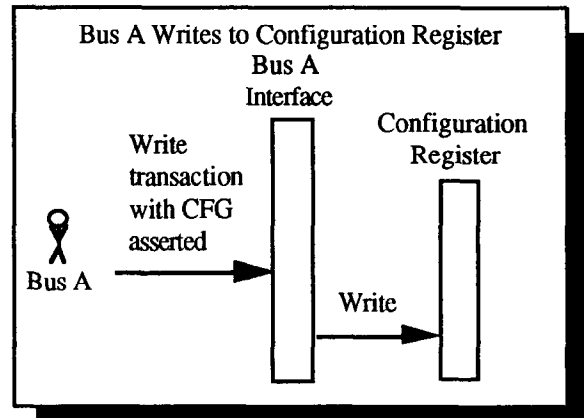


Figure 10 - A set of interaction diagrams showing support for all functions.

When we refer to the system as being 'defined' by the Requirements Model, we mean that the scope of the system has been delineated and a foundation has been established for the rest of the development program. Most importantly, the model is capable of serving as a communications medium between different teams who must coordinate the following efforts. This is a key aspect of any non-trivial engineering effort that requires more than one individual to complete. In those cases where the effort is small and individual, the process is justified by the longer-term utility of having some level of development documentation available for maintenance purposes.

Analysis Model

The requirements model could lead directly to an Implementation Model if the size of the project is relatively small, if the developers of the requirements model are all experienced users of the development tools required to complete the project and if the targeted architecture of the system is well understood and considered adequate. The analysis model provides an opportunity to improve the quality of the design by manipulating the hierarchic boundaries imposed during the requirement model phase. Analysis at this time can be extremely cost-effective because simple to perform changes can dramatically effect the

effort required to support the entire lifecycle of the system.

Changes to the hierarchy are made to either improve the performance of downstream development tools, to enable resource sharing lowering the cost of the delivered system, or to structure the system such that future requirements and implementation changes can be more efficiently performed.

Downstream development tools are typically driven by constraints. In order to operate those tools, therefore, the constraints imposed by the requirements must be mapped to the internal blocks that perform the system functions. The complexity of that mapping is a large determinant of not only the effort that will ultimately be required to implement the system, but also the quality of the final product. This statement is intuitive to users of synthesis tools and the difficulty of synthesizing improperly structured designs is considered 'experience'.

Experience can lead to explicit or implicit application of design rules. To explicitly state one such rule:

Synchronous => Good

Asynchronous => Bad

This truth may not, however, survive the test of time (no pun intended). Difficulties managing clock networks operating at hundreds of megahertz across millions of transistors in power-limited applications casts some serious doubts about what 'truism' will apply a few years from now. Since we are attempting to build structures with long-term value, and since the first models produced are intended to have the longest term utility, it is unwise to impose such edicts too soon. The analysis model is the time to balance pragmatism and rigor.

In our view, a more universal rule is

Requirements Constraint

= mapped to one =>

Block Constraint

Blocks should be delineated such that external constraints are mapped one-to-one them.

Today, the goal is to isolate and minimize asynchronous behaviors so that implementation-specific metastability issues can be easily addressed and so that synthesis engines based on static analysis of the circuit can be most effectively operated. Can a partitioning be made that meets the immediate requirements imposed by experience and tools, yet allows for later innovations, such as the use of asynchronous design practices throughout the IC?

Any partitioning capable of meeting both these requirements would isolate interface timing (synchronous or asynchronous) from internal functions. With the environmentally constrained elements isolated, the internal blocks could be redefined as asynchronous without disturbing the system's interface implementation.

If the environment is synchronous in nature, the interfaces would be subject to the full range of synchronous circuit performance criteria (ie. skew, setup, hold, etc.) while the interior blocks might use handshaking to synchronize activities.

To provide illustration of a specific example of the dramatic effect that hierarchy decisions have on the ultimate difficulty of implementation, additional details must be introduced to our model. To do this, the previously introduced interaction diagram (figure 10) is enhanced with a more literal interpretation of time extending down the diagram. and with specific time-based constraints indicated with vertical arrows. This format for interaction diagrams is integral to the remaining stages of the system development process. These diagrams are the essential link between high level abstraction and practical implementation. The ability to trace specific named entities from the requirements model, through to specific lines of VHDL and specific transistors on a chip is provided by interaction diagrams containing progressively more and more detail.

As can be seen in figure 11, the clock rate of Bus A has propagated to the 'Configuration Register' block such that the performance of the data buffers in 'Bus A Interface' is critical.

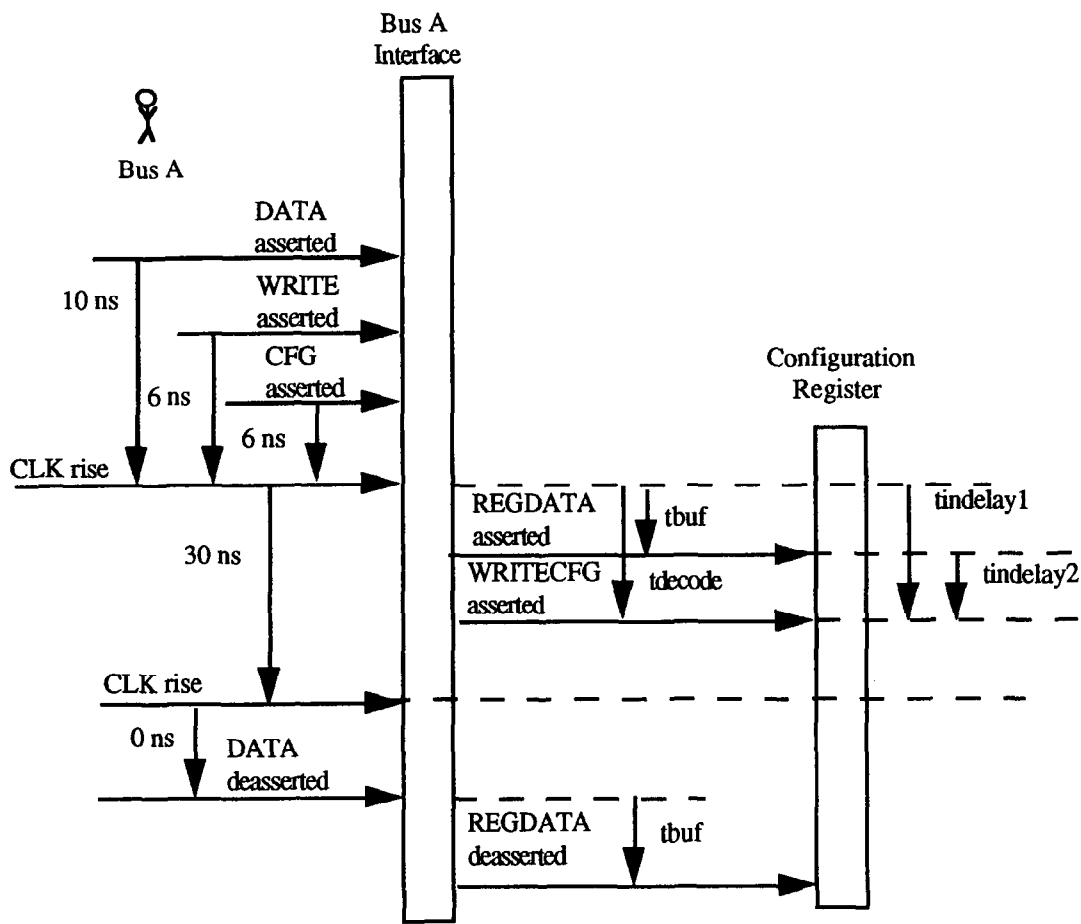


Figure 11 - 'Bus A' writes to 'Configuration Register'

What this shows is that the interface conditions that constrain the implementation of 'Bus A Interface' are not isolated to that block but are instead further constrained (t_{buf} and t_{decode}) before being passed to 'Configuration Register'. The implementor of 'Configuration Register', upon understanding the implications of this seemingly innocent design decision, might, understandably, feel the design task unreasonably taxing, if not impossible.

As a goal, the designer might target a classic synchronous design architecture that applies a combinatorial function to inputs and latches the outputs [2]. Figure 12 shows this 'ideal' gate-level design.

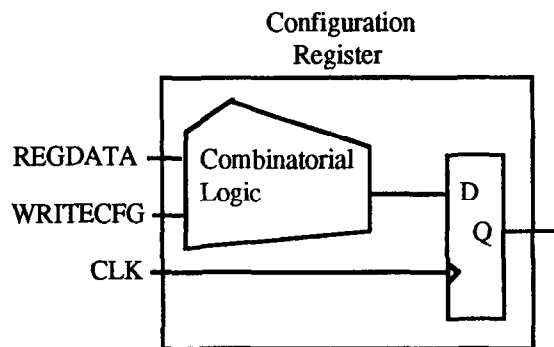


Figure 12 - Desirable implementation.

To properly design (or synthesize) this, the input delays on REGDATA and WRITECFG must be known. Of course this value won't be fully determined until the 'Bus A Interface' is implemented. Now the design hierarchy has imposed the requirement that these blocks be cooperatively developed and that, for all time, the implementation of one depends on the

implementation of the other. What we have done is produce a somewhat complex sequence of dependencies that must be accurately traced throughout the lifecycle of the product. Changes to clock rates, functionality, or implementation technology for this design will all encounter additional effort and risk as a result of these dependencies.

Robustness analysis, in general, incorporates quality improvements by altering the hierarchic boundaries of the components to make individual blocks more independent. This will have the effect of reducing time to implement one generation of the design and to

reduce the costs of producing revised designs in the future.

Complex interdependencies can be managed by high-end tools if, and only if, the process of applying the tool is intimately understood and if an appropriate bottom-up and/or top-down synthesis strategy is chosen and applied [5].

This particular situation could be simplified by combining the configuration register into the 'Bus A Interface' block as shown in figure 13. This eliminates the need to pass constraints between blocks during synthesis and will lessen the cost to incorporate later changes.

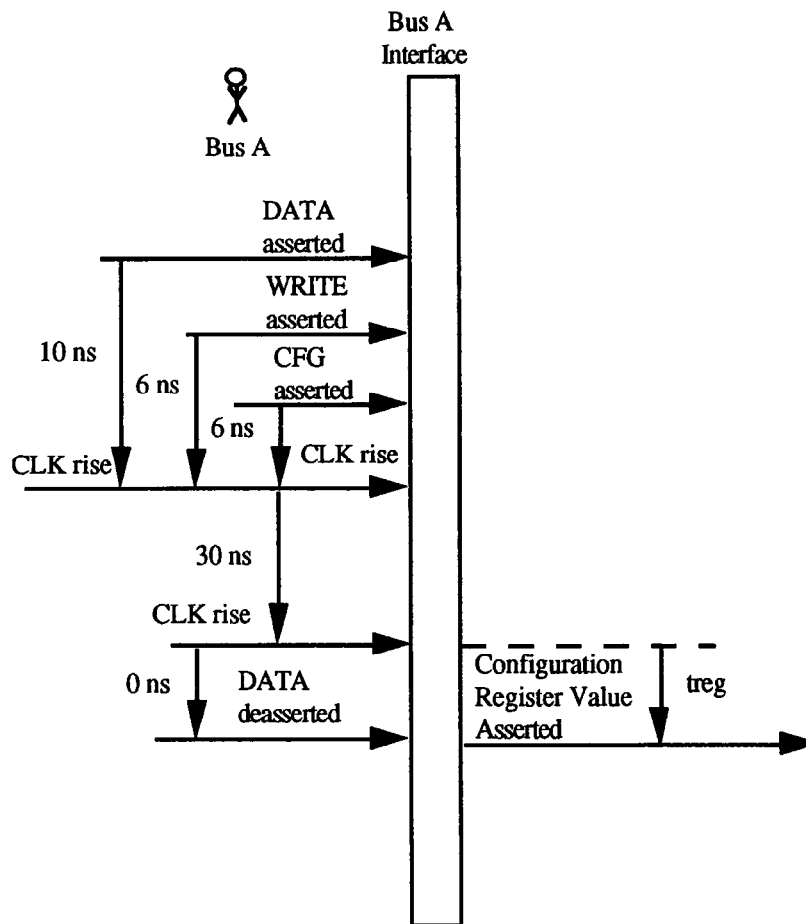


Figure 13 - 'Bus A' writes configuration data to 'Bus A Interface'.

This is only one possible way to decouple the external and internal timing constraints. The configuration value could be registered by the 'Bus A Interface' block and then registered again by the 'Configuration Register' block. This adds the incremental cost of latches for the DATA and would increase the latency of changes to the configuration information by a clock. Only greater knowledge of overall system requirement could reconcile what approach is appropriate. This illustrates a key point:

There is no free lunch

Efforts to impose rules of segmentation will impact the size and speed of a design. The modeling process described here does not prescribe a set of rules that has to be blindly applied to all designs in the absence of careful weighing of tradeoffs. The models do, however, structure the process of design such that there is a time and a place for each appropriate effort whether it be the directed to long term or short-term goals.

Imposing new hierarchic boundaries to protect the integrity of a model in the face of requirements changes (at some unpredictable time in the future) might at first be considered a frivolous luxury to those engineers thinking in terms of meeting already difficult performance criteria. This consideration is much more familiar to those engineers developing improved versions of previously accepted chips. The market comes to accept the documented and undocumented features of a design and will reject a component that does not behave (or misbehave) identically. Effort , or even silicon, can be a wise investment when the entire lifecycle of a product is considered.

Design Model

The Design Model is intended to apply the constraints of the specific implementation environment. This obviously involves more detail including names of individual signals and timing parameters. This process is especially familiar to designers using FPGAs at the boundaries of their capabilities. Careful pipelining to divide functions across more than

one clock boundary might be needed to satisfy demanding timing for one implementation that might not be required for another.

Implementation Model

When applied to software, the Implementation Model is said to be writing source code and developing the processes required to convert it to executable code. For hardware developers using VHDL tools, the source code could have potentially been started far earlier as support for even the first Requirements Model. This difference of approach is made possible by VHDL's ability to represent system behaviors, register-transfer behaviors, as well as structural interconnections of components [3].

In any case, a design process must end with the creation of a description of adequate detail that a system can be constructed. The Implementation Model is the description from which the system is built.

Test Model

Again, VHDL offers a great deal of flexibility in regards to test. It is possible, and usually desirable, to develop a Test Model in parallel with the rest of the system. This has the advantage of bringing the test process to maturity in parallel with the design. This can reduce the costs of debugging hardware 'in-circuit' with operational software. This subject deserves greater study and will hopefully be further developed in the future.

Future Work

The method described in this paper is one which has potential application in any number of design environments. We see the opportunities as a range extending from pragmatic, immediate uses of the framework to incrementally improve small segments of the development process to the development of information modeling schemas capable of managing engineering work-flow in a seamless, intuitive way.

We incrementally apply the principles to our everyday work gaining constant improvements for ourselves and our customers.

References

[1] Ivar Jacobson, Object-Oriented Software Engineering : A Use Case Driven Approach, Addison-Wesley, 1992.

[2] Douglas E. Ott & Thomas J. Wilderotter, A Designer's Guide to VHDL Synthesis, Kluwer Academic Press, 1994.

[3] Douglas L. Perry, VHDL, McGraw Hill, 1991.

[4] Donald E. Thomas & Philip R. Moorby, The Verilog Hardware Description Language, Kluwer Academic Press, 1995.

[5] Pran Karup & Taher Abbasi, Logic Synthesis Using Synopsis, Kluwer Academic Press, 1995.