

Object Oriented ASIC Design

Charles F. Shelor
Shelor Engineering
3308 Hollow Creek Rd
Arlington, TX 76017-5346

Abstract

This paper describes the author's experience using VHDL based logic synthesis with an object oriented methodology in the development of ASICs. Application of this methodology results in designs that are produced more quickly, designs that are capable of being reused, and designs that are more likely to work the first time.

Introduction

Today's electronics market is highly competitive and fast paced. Sometimes the viable selling period of a product is only a few months. This puts tremendous pressure on the designers of electronics to complete their designs in a very short schedule. ASICs in these designs usually generate the critical path of the schedule. The ASIC fabrication process is relatively long and rather expensive. Thus it is imperative that the ASIC design is accomplished **rapidly** and **correctly**.

There has been a significant amount of discussion in conferences and trade journals about top-down design of ASICs. However, these discussions don't provide much detail in how to actually perform top-down design. The object oriented methodology described in this paper provides some details in one approach used in the generation of a top-down design.

The methodology presented in this paper was generated with the overriding goal of **maximizing engineering productivity** in the development of ASIC based electronics. The development period addressed in the paper begins with initial system requirements and continues through system delivery. The methodology reduces design time by stressing

reuse of design components. Second pass silicon cost and schedule is averted by emphasizing correct first pass silicon. Integration time is reduced by using well tested components communicating through controlled interfaces.

The first section of this paper provides a very cursory introduction to object oriented methods. The paper then presents the methodology in the sequence appropriate for application to a project. This sequence is requirements definition, design, implementation, test, integration, fabrication, and system integration. The quantity and types of resources needed in each of the project phases is also discussed.

Object Oriented Methodology

The methodology used in this paper was derived from Principles of Object-Oriented Analysis and Design by James Martin, published by Prentice Hall, Englewood Cliffs, NJ, 1993. Martin defines an *object* as any thing, real or abstract, about which we store data and those operations that manipulate the data. A greatly simplified explanation of an object oriented process is the decomposition of an object into smaller component objects. Objects with similar characteristics are collected into groups, normally referred to as *classes*, during the decomposition process. The keys to a successful object oriented approach are (1) recognition of objects, (2) selection of appropriate object classes, and (3) maintaining a distinction among the different views of the objects.

Recognition of an object is sometimes very easy. The objects of an automobile include doors, engine, wheels, seats, radio, transmission. Sometimes the

object may have the same name as the function that it performs. An OR gate is an object. The ORing of two signals is a function. An object should always be a noun in a sentence. Use a proposed object as a noun in a variety of sentences. If the sentences are awkward or do not make sense you have not selected an object. 'I painted the *door* of my car.' 'Tom replaced the *tires* last month.' 'I painted the *steel* of my car.' 'Tom replaced the *vinyl* last month.' Although there is a lot of steel and vinyl in the automobile object and its component objects, steel and vinyl are not objects themselves.

Recognizing commonality among objects is not difficult. Trying to categorize objects into a distinct set of classes can seem virtually impossible at times. The problems arise when object A shares features with object B; object C shares features with object D and object E shares some features with A and others with D, while being quite different from B and C. The front doors and rear doors of an automobile are obviously very similar. (Some differences could be whether or not the windows roll down completely, existence of a child proof interlock, existence of rear view mirrors.) The fenders and quarter panels are objects similar to each other. Should the trunk and hood be placed in the *doors* class or the *panels* class or a new class generated? They open like doors but do not have windows. The trunk can be locked with a key but not the hood. People do not enter and exit through the hood or trunk. (That is in normal operational conditions. Entry through the hood or trunk could be accomplished in an exception processing mode.) This is where engineering judgement must be applied. Sometimes the purpose of the overall system can help. (A glass installation facility would put trunk and hood in the *panels* class.) Sometimes object classification may need to be deferred until the component objects are analyzed.

Once objects have been defined they must be described. Most of the time an object can be described in 3 distinct ways. The *structure* view indicates its composition. This is the collection of smaller objects that are assembled to form the current object. Some software authors use the term *context* to represent composition. They might document the structure with an Object Class Diagram. This easily maps to a structural architecture in VHDL.

The *behavior* view describes what the object does and the states that it can achieve. This view of an object is very similar to what is developed when using a pure functional paradigm. Many object oriented methods use state charts or state transition diagrams to document the behavior view of an object. This easily maps into a behavioral architecture in VHDL.

The third view of an object is its interface with other objects. This describes types, directions, and rates of data communication among the objects. This view is usually documented with object interaction diagrams or dataflow charts. The VHDL entity and component features capture the static interface of this view but do not require a description of the rates of interaction. Placing this information in comments will complete the interface description.

Requirements Definition

The requirements definition phase of the development is the most critical step in determining the chances of success or failure of the project. Establishing unrealistic requirements will force the design to be overly complex. This results in increased design time, a riskier design, and pushes the implementation tools and technology to their limits. Ambiguous requirements create the potential of the designer misunderstanding the intent of the product. This could result in an ASIC that operates as designed but fails to meet the system requirements. Inadequate requirements can also lead to functional ASICs that fail to meet the *true* system requirements. Requirements can also be too constraining. This limits the choices of the designer and can prevent the use of available design components or the use of special features in the implementation technology.

Most projects with which I am familiar have made several bad errors in the requirements definition phase. Many of these result from not having the appropriate hardware and software personnel involved in the early requirements definitions. Requirements errors are not normally discovered until system integration and can easily result in a reduction in initial system functionality and a second ASIC fabrication.

In a short development cycle there is never enough time to allocate to any of the phases. Requirements definition usually gets the most severe schedule cuts. Designs in many projects are started well before the requirements are completed. (Isn't the definition of concurrent engineering, certainly the essence of it anyway, the simultaneous development of requirements and design?) Requirements definition must continue until it is complete and not allowed to stall because design has been started.

One approach that can assist the co-development of requirements and design is the use of a common methodology. An object oriented requirements analysis combined with object oriented hardware and object oriented software developments will help reduce requirements errors.

The requirements definition phase should be staffed with the systems, hardware, and software senior personnel. The hardware and software personnel should have a thorough grasp of what the final system is expected to accomplish and understand the implementation technology being used. Requirements definition should account for about 10% of the schedule and budget. The requirements definition phase concludes with the final release of the requirements documents for the various objects of the system. These documents should be reviewed by senior and staff level personnel from other projects to ensure completeness and applicability.

Design

The design of the hardware is an iterative decomposition of the objects defined in the preceding step into smaller objects. The first iteration takes all of the objects defined during the requirements phase and partitions each of the hardware objects into its components. Each resultant object is decomposed into smaller objects. This process continues until *atomic* objects are defined. An atomic object is one that cannot be logically broken into smaller objects. Automotive examples of atomic objects are steering wheel, window, windshield, piston, brake pad, headlight, and tire. Gates and flip flops might be considered atomic objects for electronics. However, the use of logic synthesis allows higher level components to be the atomic objects in hardware design. A state machine, registered counter, parity generator, or address decoder represent atomic objects during the design phase.

The challenge during the design phase is recognition of a good decomposition relative to a poor decomposition. The following factors should be considered during the decomposition of an object into the next level of hierarchy. (1) Clear relationship between the system composition and the resultant objects. (2) Interaction among the resultant objects is minimized. (3) Complexity of the resultant objects should be roughly equal. (4) The structure should be easily comprehended by a skilled engineer. (5) The object is capable of being implemented in the target technology. (6) Potential reuse of the object in future projects. (7) Testability of the object as a standalone component.

(1) System relationship. The objects should be described in the terminology of the system. Obvious system components should not be split into sections simply to satisfy a partitioning mandate. It would not be appropriate to partition a steering wheel into spokes, rim, and hub.

(2) Interaction. Attempt to minimize the interactions between objects while maximizing the activity within an object. Software refers to this as minimum coupling maximum cohesion. A partitioning that contains few ports would be preferred over one that has many ports. Attempt to keep any high frequency signals or analog signals in a single object or minimum number of small objects. Minimum interaction makes understanding the composition easier, makes testing the composite easier, reduces the global interconnect wiring, and reduces the sensitivity to skew and noise.

(3) Complexity equivalence. One of the purposes of the partitioning is to make the system understandable to current and subsequent designers. A partitioning that includes the automobile engine and the battery strap components would be absurd. A partitioning that shows a SCSI bus controller and a pair of flip flops for status light control would not be balanced. One approach to managing complexity level is to have a miscellaneous component that collects all of the very simple items into a level of hierarchy.

(4) Comprehension. Psychological studies have shown that the average person comprehends 5 pieces of information at a time. Most senior and lead engineers can be considered above average. Therefore, any single level of abstraction should be decomposed into 6 to 8 component types. The actual number and types of objects will be determined by the natural organization of the system. If an abstraction can only be logically decomposed into 2 pieces, you might consider 'pushing it up' into the prior level replacing the single component with its 2 pieces. Similarly when an object decomposes logically into 14 components, the current object might be replaced with 2 objects containing 7 components each.

(5) Implementation. If the current object can be directly implemented then it should be considered an atomic object and the partitioning process should be halted. Partitioning too far overly constrains the implementation. The final partitioning step should be accomplished within a VHDL architecture. Thus, a process statement, a conditional assignment statement, a generation statement, a concurrent procedure call, a block statement, and a component instantiation statement would all be implementations of an atomic object.

(6) Reuse. The design phase is critical to future productivity gains by reuse. Each object should be examined for application to future projects. Sometimes a small amount of programmability can convert a specialized single use object into a general purpose reusable object.

(7) Testability. The easier an object is to test, the more thoroughly it can be tested. Totally testing the object in a standalone mode means that internal verification of the object at higher levels is not required. This also impacts reuse as you would only want to reuse an object that was thoroughly tested.

The design phase will require approximately 20% of your schedule and budget. You will want senior and lead persons performing the design. The output of the design process will be a series of design documents. A typical project might have a design document for each board and a document for each ASIC. The document sections should reflect the partitioning hierarchy. A document for an automobile might contain section 1 for engine, section 2 for body, section 3 for transmission, etc. Section 1.1 for block, section 1.2 for piston, section 1.3 for camshaft, etc. A separate section or document for common components used throughout the design can be used to reduce replication of information.

The design phase should produce a set of entity declarations. The structural architectures that combine the components can also be generated during this phase. Writing these structures is one way of incorporating the younger engineers into the project. Comments embedded in the entity declarations can be used as the documentation for this level of the design. Describing the envisioned architecture in comments prevents this information from being lost and may provide beneficial insights when the implementation is performed. This is also a good way to establish and document timing and sizing goals for each component in the design. Another good use of comments is the discussion of how the object should be tested. This should include unique test conditions; test cases that have historically caused problems for circuits of this type; and test methods that might reduce the test times.

The designs should be reviewed by the senior personnel and other lead personnel on the project. Special attention should be placed on determining if there are common objects among the different designs. Other goals of this review are the identification of potentially reusable objects or questioning why particular existing objects were not used.

Implementation

Implementation of the design is the generation of the VHDL code for input to the logic synthesis tool. Each object should be synthesized as soon as the code is written. This ensures that the generated code is acceptable to the synthesis tool and that the target

technology can implement the desired behavior. The synthesis results should be examined to verify that it meets the object timing and size requirements. New synthesis users are often surprised at the large number of gates that are generated by some VHDL structures. (Especially when a 32-bit counter is generated for an unconstrained integer used as a 0 to 9 cyclic counter; or the 32-bit ALU that is inserted in place of the 4-bit increment function that was intended!) The new users may also be surprised at the long delays of some synthesized logic paths. (Especially the implied priority logic sequence when using conditional signal assignments or if-then-elsif constructs for address decoding!)

After reviewing the synthesis output, each object should be thoroughly tested before the next object is implemented. If modifications were needed to complete the tests, the VHDL should be resynthesized. A quick check of the synthesis output should be performed to ensure nothing surprising happened as a result of the changes.

Implementation is performed by the junior engineers with reviews and guidance by the senior engineers. Approximately 20% of the schedule and budget should be allocated to implementation. The VHDL source listings represent the output of this phase.

Test

Testing of each object occurs immediately after implementation. Good test cases are more likely to be generated in this manner since the implementation details are 'fresh' in the minds of the implementers. All tests should be performed using the test bench capabilities of VHDL. This makes the testing more portable among VHDL simulators than using simulator command files. The application of regression testing is enhanced by VHDL drivers and test benches. Reuse of the object is enhanced by having an associated test bench to allow the reusers to study the object in operation before determining if they can actually use it or not.

The testing is accomplished by the same junior engineers that performed the implementation. The senior engineers should review the test cases and test circuits to ensure that they adequately tested the implementation. This effort is documented by the test case VHDL code and simulation output files. Testing will require approximately 20% of the schedule and budget.

Integration

The integration phase refers to the collection of lower level objects into their composite parent objects. Each integrated object should be tested in a manner similar to the atomic or leaf objects. The use of VHDL test cases, stimulus generators, and test benches makes the testing more portable and provides regression test capabilities.

The final level of integration may include interfaces to other components of the system. This could include a VHDL to C interface to allow operational software written in C to interact with the VHDL simulation. This also might be accomplished by using ASIC emulation (such as the Enterprise from QuickTurn Systems or a breadboarded FPGA emulation.)

The biggest reason why 98% of all ASICs pass their functional vectors yet only 50% operate correctly in the system is inadequate testing with system stimulus. Consider a moderately complex ASIC with 50,000 gates. This ASIC probably includes around 2,000 flip flops and about 100 input signals. 2^{2000} is 1.15×10^{605} which represents the number of internal states that the ASIC can achieve. 2^{100} is 1.268×10^{30} which represents the number of input conditions that can be applied. The product of those two numbers represents the number of possible combinations of states and inputs. (1.45×10^{635}) All possible combinations can be generated within 4.8×10^{616} centuries at 1 GHz input rate!

Being realistic and assuming that most of the combinations are independent generates the following: 100 flip flops determine ASIC states, 1900 used for datapath operations; 28 inputs control operation, 64 data inputs plus 8 parity inputs used for data. The number of possible combinations is now only the product of 2^{100} and 2^{28} or 2^{128} . Only 1.13×10^{20} centuries is now required for exhaustive testing at 1 GHz. Thus, the engineer cannot possibly even come close to exhaustive testing with a few million vectors. The only way to adequately test a system of this complexity is via system emulation.

This testing will be accomplished by the engineers that are now moving from junior to senior status with some assistance from the original senior engineers. This phase will account for about 15% of the schedule and budget.

Fabrication

Engineering support to the ASIC vendor during place and route operations is placed in the fabrication phase. Performance of detailed, back annotated timing analysis is also conducted during the fabrication phase.

These tasks will be conducted by the same engineers that completed the integration task. This phase will require about 5% of the schedule and budget.

System Integration

The system integration phase of the project is the operation of the system with the ASICs assembled in the circuit cards. This phase is supported by the same engineers used in the integration phase. They will consume the remaining 10% of the schedule and budget. This phase concludes with a party celebrating first pass operation, on time delivery, and on budget completion.