Digital Hardware Design
An Object-Oriented Approach

By : Tarak G. Modi
Advisor : Dr. Wells

Object-oriented modeling and design is a new and revolutionary way of approaching and solving complex problems. This methodology is based on the concept of using models organized around real world concepts. This approach has gained great popularity among the software engineering community in recent years, and is nowadays the de-facto standard for software design in all fields. In this paper we will try to demonstrate that this methodology can also be successfully applied to hardware design, and that objects are in fact inherent to hardware design.

Fundamental Object-Oriented Design Characteristics

All object-oriented based designs have fundamental characteristics inherent to the object-oriented design methodology. Although it is possible for designs based on other methodologies to exhibit some of these same characteristics, they are innate to object based systems.

1. Abstraction

Abstraction is the process of focussing on different levels of detail of the same system/subsystem. For example, during the initial analysis phase of the system we are more concerned about what the system should do rather than how it does it. This is the highest level of abstraction. As the development process advances, the level of abstraction goes from higher to lower, until finally the designer is faced with the actual implementation of the system (The actual C ++ program, or the circuit design). Proper use of abstraction allows the same system model to be used for analysis, high and low level design, and documentation. Consider a computer. To the layman, a computer appears to be a magic box whereas to an application oriented user, it appears to be a useful tool. These are simply different views, or different levels of abstraction, of the same computer.

2. Encapsulation

Also known as "Information Hiding", it basically refers to the process of separating external interfaces from internal implementation details. Encapsulation prevents a system from becoming so interdependent and intermingled that a single small change has rippling effects on the entire system. Encapsulation allows different, independent portions of the design to be modified which may be for the purpose of fixing a bug, improving performance, or for porting reasons.

3. Reusability
Object-oriented design highly encourages the principle of reusability. By properly understanding, and recognizing repetitive portions of a design, a single object model may be reused again each time. Once again, reusing the same object model gives the added advantages of abstraction and encapsulation.

Object-Oriented Methodology

The object-oriented methodology enforces design dicipline. It forces the designer to spend more time and energy during the design phase of the system, before actually implementing it; sort of like think twice before you speak. The fact is that object-oriented design is a conceptual process, independent of software engineering, or for that matter, any other area of specialization. Its purpose is to serve as a medium for specification, analysis, documentation, and interfacing. It helps both developers and customers form and communicate abstract concepts clearly and concisely.

As described above, the object-oriented methodology consists of building a model of the application and then adding the actual implementation details during the design of the system. The methodology may be logically divided into the following stages of development :

1. Analysis
The problem statement is very carefully analyzed, and a requirements list is prepared. Often times, the system analyst must work closely with the requester (customer) to understand the problem, because problem statements are rarely complete. From the initial set of requirements, a set of derived requirements must be prepared. It is the purpose of the analyst to make the problem statement more precise and to expose ambiguities and inconsistancies. The problem statement should not be considered unchangeable but should serve as the foundation to which to build upon. The analyst now builds a preliminary design model. At this stage the model is concise: a precise abstraction of *what* rather than *how*. There is no scope for implementation details at this point. This model is presented to other system analysts where it is reviewed, criticized, and refined.

2. System Object Design
The model is now subdivided into logical and physical subsystems based on the proposed architecture. This subdivision is made on a functional and behavioral basis. Each subsystem encompasses aspects of the system that share some common properties such as same basic functions, proximity of physical location, or the same basic hardware. For example, a robotic paint stripping system might consist subsystems for vision, mapping, paint removal, etc. A subsystem may also be divided into other subsystems, and thus a heirarchy of subsystems may be obtained. Various performance issues are now considered, optimized, and trade-offs are made. At this point, details are added to the subsystems and a design model based on the analysis model is prepared. Issues such as compatibility and interfacing between the subsystems, and the integration of each independent subsystem into a fully functional system are considered. Subsystems are defined so that most

interactions are within the subsystems, rather than across the subsystems. This reduces dependancies, and facilitates future modifications and maintenance. Each of these subsytems are representative of a system object.

3. Implementation
Next, each individual system object is developed and simulated. Each individual system object may be developed by separate and independent teams, as long as the behavior adheres to the object model developed during the second phase of the development process. Finally, all the individual system objects are integrated into one system. The system must now undergo a rigorous testing and debugging process, and its performance compared to the initial design requirements. Depending on the results, the various subsystems are modified (or, tweaked) and the desired performance is achieved. The additional effort during the analysis and the system object design phase pays off during the implementation phase by minimizing system object dependencies, and hence facilitating the tweaking process.

Using VHDL To Implement System Obect Models

VHDL is an acronym for VHSIC Hardware Description Language (VHSIC itself is an acronym for Very High Speed Integrated Circuits) that can be used to model simple to complex digital systems. VHDL is often quoted to be acronym for Very Hard Description Language. Granted, VHDL is a very verbose language with many complex constructs that have complex semantic meanings, and may be initially difficult to understand. A subset of VHDL may be, however, quickly learned and understood, and is an effective tool to that can be used to implement system objects.

To implement a system object model, VHDL provides five different primary constructs (called design units):

- Entity Declaration
- Architecture Body
- Configuration Declaration
- Package Declaration
- Package Body

Together, these five constructs provides a powerful combination for implementing system objects. The entitiy delcaration describes the external view/interface of the object, for example, the input and output signals. The entity declaration may also be used to specify certain default parameters (called generics) inherent to the system object, such as delay times. The term entity and system object model will be used interchageably in the following discussion. The architecture body consists of the actual implementation of the entity. A single entity may have many architecture bodies. This is due to the fact that the same entity may be viewed in different ways. As long as the external interface does not change and the end results are the same, the actual implementation of the entity is irrelevant to the system design (although certain implentations may be more efficient than others, where efficiency is a loose term). The configuration declaration actually determines which view of the entity, i.e. which architecture body is actually picked up during simulation and synthesis. The package declaration and body design units may be used to define system wide parameters. C++ programmers will probably identify similarities with C++. The entity declaration along with the architecture body is similar to a class and the class implementation (methods) in C++. Package declarations and package bodies are similar to header files. Classes in C++ are interfaced with by using public class variables or public class methods which basically define the class interface. In VHDL, the entities are interfaced with by the interface ports or, signals defined in the entity declaration.

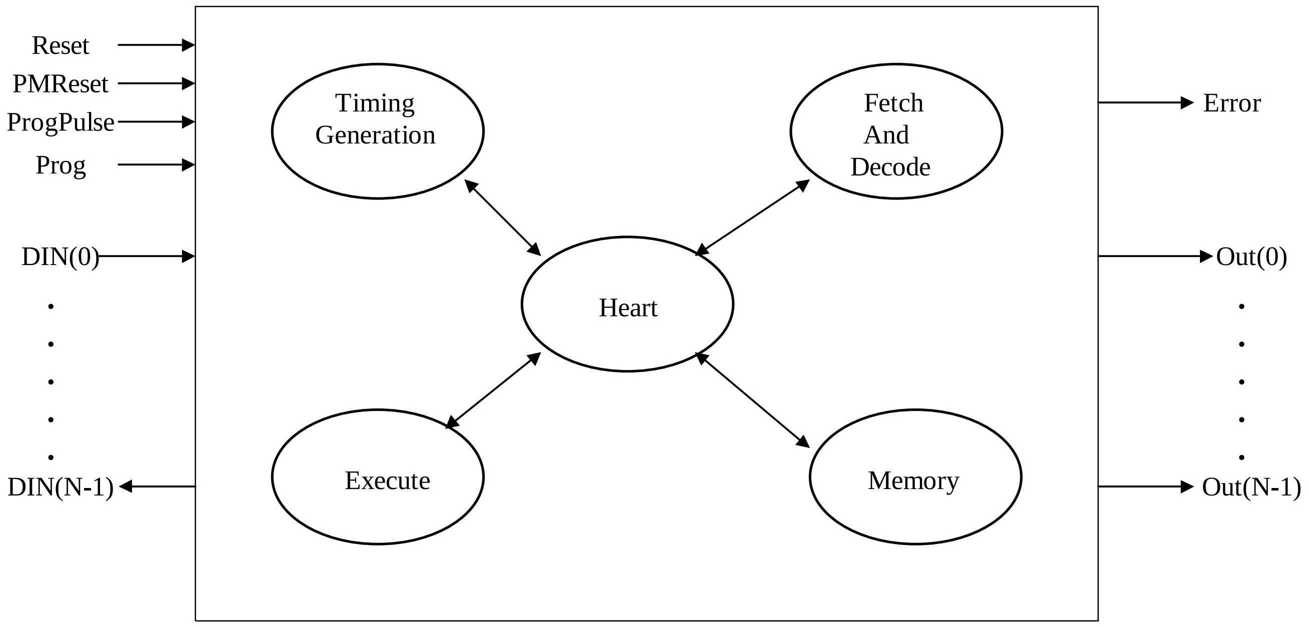
An Example

Let us consider the implementation of a very preliminary microprocessor with a limited instruction set and on-chip RAM. The first step would be to determine the instruction set required. This would also dictate the minimum bit width (N). The amount of on-chip RAM required must also be known. As seen in the VHDL source code defining these parameters in a package allows us to make changes and implement them just by recompiling the source code. Next, the inputs and outputs of the microprocessor are determined. A clock (CLK) input is used by the microprocessor to generate the internal timing for the different operations. A reset (Reset, active high) input resets the microprocessor. A separate program memory reset (PMReset, active high) is provided to reset the memory, so that the same program may be used over and over again. A program pulse (ProgPulse, positive edge triggered) input along with a program enable (Prog, active high) input is used to program the memory. The data inputs (DIN(0) to DIN(N-1)) are set to the data store in the memory each time the program pulse is given with program mode enabled. The system outputs consist of an error (Error) output indicating an error condition such as an invalid instruction, and outputs (Out(0) to Out(N-1)).

Once all the requirements (or, most of them) have been determined, the system object models are defined. A system may be defined by different combinations of object models. In our case the microprocessor has been divided into five system objects :

- Timing Generation
- Fetch And Decode
- Execute
- Memory
- Heart

In the VHDL source code each system object is represented by an entity and an architecture body. The timing generation system object gets the external clock signal, and generates the states t0 to t3. The fetch and decode system object fetches the next instruction from the program memory, and decodes it during the t0 state. The execution system object executes the decoded instruction during the t3 state, and also fetches any immediate operands if required. The memory system object keeps track of the current memory location to be accessed, and updates the program counter during every t1 state and also during t3 if necessary. It is also responsible for loading an external program



The Microprocessor : System Objects

when the program mode is enabled and program pulse is received. The heart system object instantiates all the components (when entities are used by another entity they are called components). and coordinates all the activities. This action is similar to a parent object constructing objects.

Conclusion

Reduced development time is not one of the selling points of object-oriented design methodology. In fact, the development time may exceed that with other conventional methodologies. Object-oriented methodology aims at achieving a superior, well thought design which promotes future reuse and reduced downstream errors and maintenance.

References

1. Object-Oriented Modelling and Design.
2. A VHDL Primer