

Chip Layout Structural Design Flow with C++ Class Interfaces

Authors

Abstract

From iccad call for paper, the abstract should contain:

1- state out clearly and precisely what is new

2- point out to significant results

for 1: what we do is use the ICSP/C++ interfaces to specify structural information and guide the functional decomposition on the structure.

for 2: it enables layout predictability early in the design stage and eases functional decomposition

for 2 also: it points out that new tools are needed to do this

In this paper, we present a system engineering methodology built upon the definition and implementation of object relationship. We show how class interfaces can be used for structural layout representation throughout all design abstraction levels.

With our design process, the engineer uses an extended UML (Unified Modeling Language) notation, Scenic and ICS class interfaces and our design CAD programs suite.

This paper describe the use of ICSP C++ class interfaces to specify structural information and to guide our tool throughout the functional decomposition on the specified structure.

In this paper, we describe our methodology and the implementation of a prototype.

1 Introduction

write a nice intro here

Complexity management

The usage of high abstraction design language language to address the complexity issue has been introduced in [1]

Usual traditional design flow for IC design

To be able to address structural layout predictability at the specification level, there is a need for a methodology paradigm shift

-functional specification

-task structuring

-architectural exploration

-structural mapping

-functional decomposition on the structural elements

[1]

acceptably accurate layout preview at the high level of abstraction

Many approaches have been proposed to manage the complexity of modern designs. Object oriented methodology have been proposed in [2] [3] [4]. Mechanisms has been explored in [5]

...Languages issues...

Subset of the object oriented methodology for hardware design has been implemented with an academia approach in [6] [7], and in the industry [8] [9] [10] [11]. Also, many companies now offer design tools to support this methodology CoWare [12], CynApps [13], C Level Design [14] and SystemC[15]

However, designers use these the same way they use HDLs: they model the hardware part of the design at the register transfer level, or wire level. In particular, extensive use of object oriented mechanisms is restricted. Usually, this is because these features are difficult to conceptualize, simulate and especially to synthesize. If a designer uses a language, it should use its mechanism, and not use it in a VHDL fashion.

The language is NOT the only issue for productivity: the process plays important role.

It does not really matter which programming language is used for SOC modeling when the basic paradigms are the same as in each other language. Each object oriented language has more or less the same features. They are different by the speed of execution of a program, their availability, their interoperability and minor syntactic details. A good language should have the necessary semantics to specify concurrency, timing and communication, and thus should be able to be automatically translated to another language without any changes in the functionality of the specification. The language choice will have to be made by each company individually, according to the common interfaces between the tools they choose for their design process. In our case, we will use C++ with our ICSP interfaces, with are extensions of SystemC.

This paper describe the use of ICSP C++ class interfaces to specify structural information and to guide

our tool throughout the functional decomposition on the specified structure.

The Incidence-Composition-Structure Project propose a methodology, class interfaces and a suite of CAD tools to address structural decomposition, functional-to mapped structure decomposition and layout estimation at a high level of abstraction..

Once the methodologies are exposed, we should build CAD tools to support each steps, and use the appropriate formats

once all languages and tools have been presented, do an array presenting what paradigms are supported by which tools

Iterative behavioral exploration, to extract data flow and structural regularity,

2 Structure, Behavior and Geometry

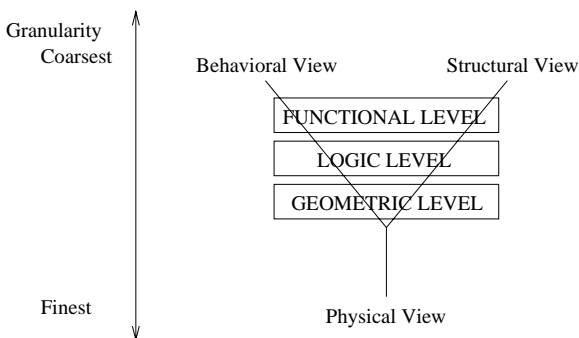


Figure 1: Y Chart

Object oriented design is useful for modeling microelectronic systems by supporting structure as a key concept at the conceptual level. This methodology is used to describe the system in terms of class and object structures. It is very easy to go from one level of abstraction to the other, since behaviors and components are encapsulated behind interfaces.

It is possible to add more detailed structural information as the architectural and technological decisions are made. This process is repeated until designers reach a good mix between structure and behavior at the proper level of abstraction, which can produce predictable and satisfactory implementation results.

To enable this process, we need a notation to capture the specification. We chose a mix of schematic and UML notation which is a language for specifying, designing, visualizing and documenting the system..

In this section, we will look at the interfaces available for the specification of the hardware components, and their relations to each other. We will then take a look at interfaces to express the physical layout information that we want to have throughout the design process. Once we know these interfaces, we will elaborate about design flow changes to specify hardware components.

2.1 Structural Specification

The goal of system conceptualization is to enable the designer to commit a design into a formal or semi-formal description that can be later used as documentation as well as input to verification tools for system correctness. This form of system conceptualization is an integral part of an important system design methodology.

We start with system level design exploration using the specify-explore-refine paradigm [16]. This methodology enables the writing of precise specifications and manual design space exploration and refinement. To close the gap between conceptual exploration and architectural exploration [17], we seek the capability to capture layout structural information at the highest levels of abstraction. This means not only identifying the entities in a design, but actually identifying which blocks will compose the layout.

In the case of industrial designs, the designers are in one of these two cases:

1. we do not know what the layout will look like
2. The designer knows what blocks he will have in the layout

2.2 Functional Decomposition

One of the most important aspects of the classes modeling is the characterization of the relationships between the entities in a design. Let us look at the major kinds of relationships we usually find in a model, and talk about their meaning for a hardware implementation.

(cut down this enumeration to the minimum needed)

1. *Generalization*: From a base class, we can derive specialized classes, through inheritance. These specialized classes inherit all interface elements from the superclass: the data structures, and member functions. The subclass's interface is said to conform to the superclass's interface, i.e. an object from the subclass is able to substitute an object from the superclass. We also say

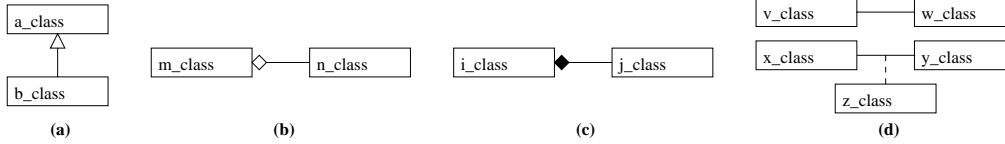


Figure 2: UML examples for relationships between classes: (a) generalization (b) composition (c) aggregation (d) association

that a superclass is a generalization of its subclasses. If we look at Figure 4(a), the superclass is *a_class*, and the subclass is *b_class*. For a hardware implementation, this means that a component could be replaced by another component which has the same port interfaces.

2. *Aggregation*: one object owns another one. The owner has logical control on the ownee. This is logical encapsulation, and expresses behavior hierarchy. For a hardware implementation, for instance, it may correspond to communication between two entities in a master-slave style. On Figure 4(b), *m_class* aggregates *n_class*.
3. *Composition*: one object owns and contains another one. A stronger form of aggregation. The owner has the control over the invocation and liveliness of the ownee. Hard encapsulation, expresses structural component hierarchy. For the hardware implementation, it may correspond to component hierarchy in a design. On figure 4(c), *j_class* is a component of *i_class*. This may represent, for instance a CPU datapath that physically contains an ALU block.
4. *Association*: represents conceptual relationships between classes. In the Figure 4(d), *v_class* has an association with *w_class* and *z_class* is an association class, which allows to add attributes, operations to the association. From an hardware perspective, it corresponds to communications, through either signals or channels.

Use polymorphism to distribute the functionality

2.3 Incidence-Composition-Structure Project

Give structural directive at the higher design abstraction.

Tools need to be redesign.

The ICSP class interfaces are built on top of the Scenic (now SystemC) class library. This class library has two layers:

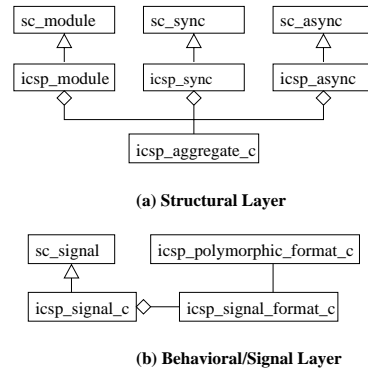


Figure 3: ICSP class interfaces

3 Our Methodology

We will present our methodology in a fashion targeted towards digital logic designs.

3.1 Steps to follow

- *Identify concurrent synchronous processes*. For a synchronous design, these will form the major blocks of the design. By connecting these blocks together, the identification of the data format for the interfaces will be needed. This is the first step to establish the structure of the system. These components will have *sc_sync* interfaces and will be on the floorplan. The behavior may be general, and dependent on what data is fed to the blocks
- *Identify data flow patterns and polymorphic behavior* Object-oriented design provides powerful behavior and structure decomposition methods. Inheritance can be used to minimize duplication of descriptions by promoting transparent shared behavior through the superclass interface. This step is an important part of the process; the identification of a generalized interface for the operations of the datapath of the system. The behavior is invoked and propagated throughout

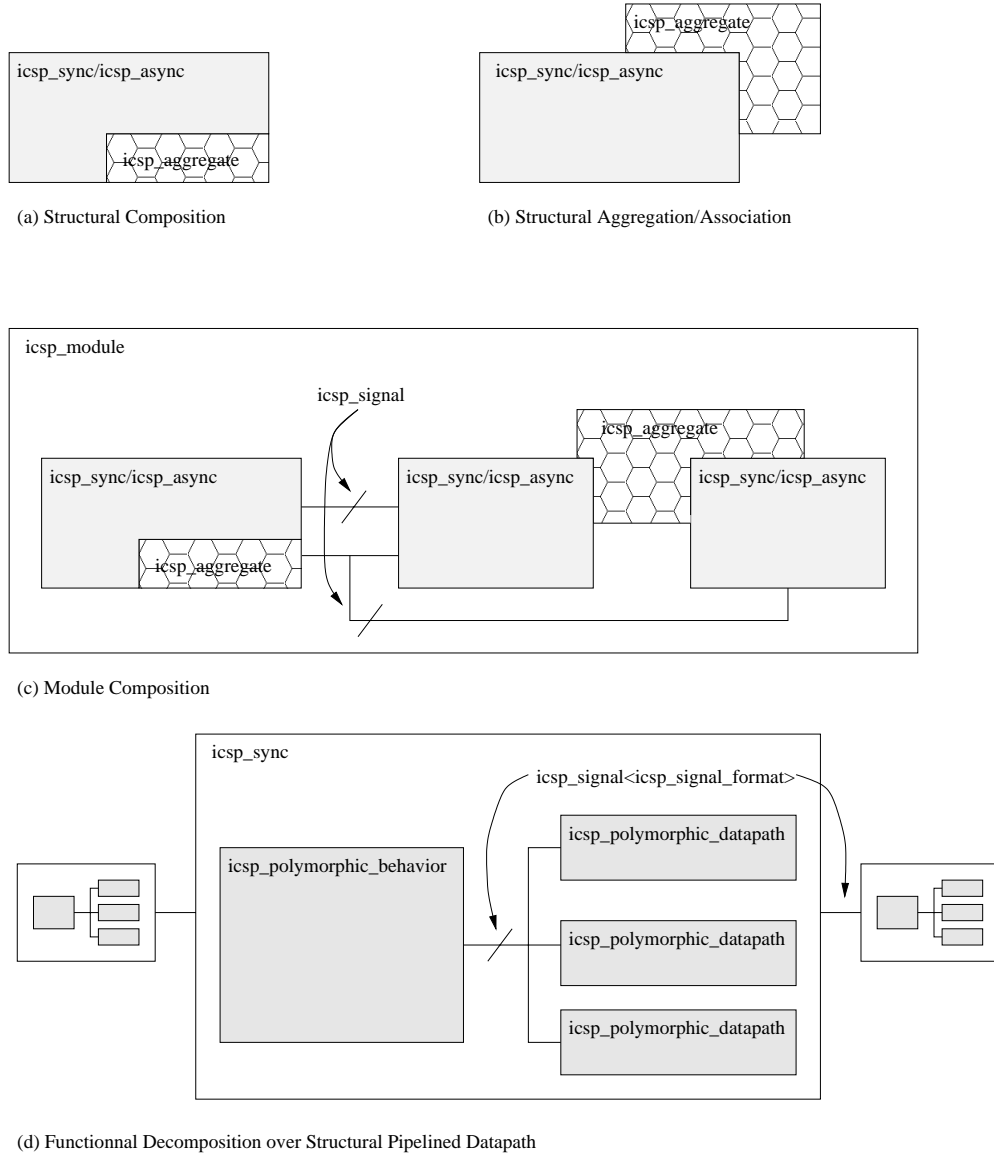


Figure 4: Structural Steps

the synchronous stages by propagating a reference to the superclass from process to process. The superclass has several virtual methods that are designed to be called by each stage it will reach. We call this mechanism datapath design and functionality propagation through polymorphism.

- *Identify asynchronous processes.* For a synchronous design, these processes consist of the control signals between the synchronous stages. These components have the *sc_async* interfaces, and will be on the floorplan. Note that a full asynchronous design is also possible, with this

polymorphic behavior propagation.

- *Identify control logic.* These components are glue logic between the processes. They are signal format, and control words. They do not have a Scenic interface, since they will be flattened in some process.
- *Build object diagram.* Once the designer is satisfied with his class models, he can build the object diagram to express the object instances, their location on the floorplan, and the ports and wire information. He will change the Scenic interfaces to ICSP interfaces to do so.

4 Case Study: A DLX Pipeline

(replace all `sc_*` references by `icsp_*` references)
(this section needs to be reworked as the steps are refined)

In this section, we present a case study for our design methodology with the conceptualization of a DLX compatible processor pipeline [18]. The model needs to be cycle accurate in the execution of the instructions, and byte code compatible with the DLX encoding. We will describe the process of the implementation step by step. Once the requirements are clear, we can proceed with the building of the class diagram.

4.1 Step 1: Identify concurrent synchronous processes

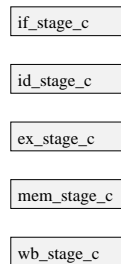


Figure 5: Basic class diagram for the DLX pipeline

These are the major blocks of the design that should be present and recognizable at the layout level

We have first done a class model of the stages of the pipeline. The stages of the pipeline, are shown on Figure 5. The stages are specialized from the `sc_sync` interface. All stages are synchronous processes, which get activated on each clock high tick. We also have specified classes for the format of the data for the communication between classes. Let us not commit now to the choice of the communication mechanism, but rather focus on behavior encapsulation.

Note that the `ex_stage_c` class is composed of one `alu_c` class. This ALU has no Scenic interface, which means that its methods are directly called by its owner, the EX stage. This means that the ALU is combinational logic; its input and output are registered by the EX stage.

Next step is to add the external structural component to the design. In this case, we have data and program memory and a register file. The rectangle for these classes is dotted in the figure, because they are not in the DLX pipeline classes. The reader can also note that we added arrows to the association

lines, which correspond to the direction of the data in the communication. These are slight changes from the standard UML conventions. Usually, an arrow indicates navigability or responsibility. In some extent, the arrow has a data or control responsibility.

4.2 Step 2: Identify data flow patterns and polymorphic behavior

(the fig for this step will be reworked)

We have been through the major structural blocks of the pipeline, let us now look at the needed behavior structuring to be able to execute an instruction. In this design, we will propagate a reference to the generalized instruction class; but it will refer a specialized instruction class.

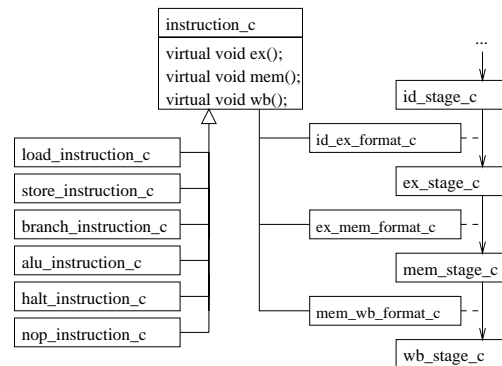


Figure 6: Polymorphic DLX instruction hierarchy

Looking at Figure 6 we see the `instruction_c` class interface, with the virtual methods `ex()`, `mem()` and `wb()`. These are invoked by the EX, MEM and WB stages respectively. The specialized instruction classes (such as an `alu_instruction_c`) re-implement the methods, to have the right behavior.

In this approach, we used inheritance to handle the division of the EX, MEM, and WB stages to invoke the polymorphic behavior of an instruction. At these stages, a virtual method is called at the instruction level interface, and the call is “forwarded” to the specialized class. The overloaded virtual methods of the implemented instruction behaves appropriately given its type, for example, an `alu_instruction_c` will perform a call to the ALU at the EX stage, while a `load_instruction_c` would not call to the ALU.

4.3 Step 3: Identify asynchronous processes

(the fig for this step will be reworked)

Now, the pipeline is capable of propagating the instruction behavior through its stages. However, it needs some control processes to orchestrate data movement through the datapath.

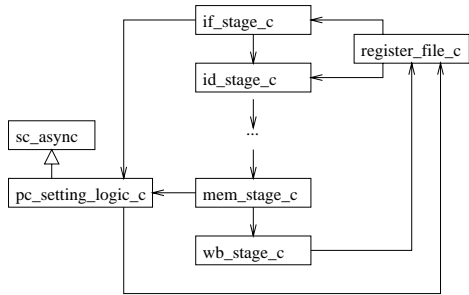


Figure 7: Asynchronous processes in DLX pipeline

Figure 7 shows the addition of an asynchronous process to update the program counter. It is generalized from the *sc_async* Scenic class, and triggered by changes on its input. This module should have the same physical properties as a synchronous process.

4.4 Step 4: Identify control logic

(the fig for this step will be reworked)

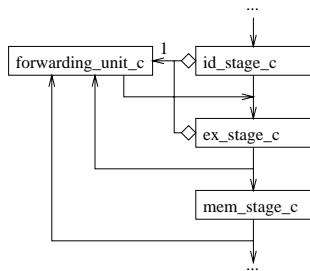


Figure 8: Control logic example in DLX pipeline

Figure 8 presents the mechanism for data forwarding in case of data dependencies between the instructions. Looking at the diagram, we see that the *forwarding_unit_c* class is aggregated in both the ID and EX stages, and that to perform its functionality, it receives the registered values from the inter stages registers. In this case, there should not be two entities of the forwarder because it is aggregated twice. The needed methods of this entity should be flattened in the stages. It is not the same method that is called by the ID and the EX stages. So, this should become asynchronous logic inside the entities. Then, it does not need an ICSP interface to have floorplanning information.

4.5 Step 5: Object Model

(the fig for this step will be reworked)

Once we reach a good compromise between structure and behavior, we can proceed to the object diagram, to express physical floorplanning information. At this time, non-ICSP classes with structural view should be converted to ICSP classes to be able to use the object view.

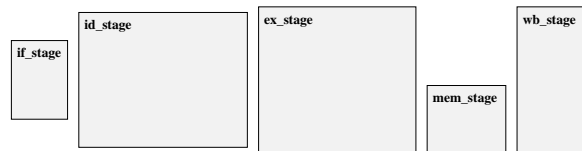


Figure 9: Simplified object diagram for the DLX pipeline

Figure 9 presents the object specification format. On it, we specify the port placement, and it shall be reflected automatically in the partially ordered sets for the ICSP port specification by our front end. The designer, with the help of the tool, should be able to estimate the area and study the feasibility of a design. It should also be possible to force these values, and add constraint and priorities.

The DLX pipeline code consists of approximately 2000 lines of C++ code much of which can be automatically generated from the UML front end [19]. The actual body of the user code, which are the entry functions, is around 200 lines of code.

4.6 Synthesis

We have a tool Tool to translate it to vhdl.

How do we synthesize the polymorphic behavior? Refer to the technical report [?]

DLX Design	Area	Clock Period (ns)
RTL VHDL	31353	24.91
ICSP/C++	xx	xx

5 Discussion

Interoperability is an issue, whatever which solution is chosen. Results from one step of the design process needs to be able to be used as input by the EDA tools for the next step. We see many new languages and tools, but is it the process that is not clear enough? Why isn't there one solution from the one presented above that permit a team of designer to go from the specification to the tape out of a SOC, and be able

to do efficient specification and architectural exploration, and be able to have enough predictability in the design?

new tools are necessary, or at least more entry format and no structural flattening at synthesis level.

6 Conclusion

reasonably accurate structural prediction early in the design phase addresses ... and helps reduce design iteration at the layout phase, thus improves predictivity.

Design structure is not just conceptual, but also an implementation need.

References

- [1] Rajesh K. Gupta and Stan Y. Liao. Using a programming language for digital system design. *IEEE Design & Test of Comp.*, pages 72–80, April-June 1997.
- [2] Sanjaya Kumar, James H. Aylor, Barry W. Johnson, and Wm. A. Wulf. Object-oriented techniques in hardware design. *Computer*, July 1994.
- [3] Wayne Wolf. Object-oriented cosynthesis of distributed embeded systems. *ACM TODAES*, 1(3):301–314, July 1996.
- [4] Wolfgang Nebel and Guido Schumacher. Object-oriented hardware modeling - where to apply and what are the objects? In *Proceedings of DAC*. ACM/IEEE, 1997.
- [5] Tommy Kuhn, Wolfgang Rosenstiel, and Udo Keschull. Description and simulation of hardware/software systems with java. In *Proceedings of DAC*, 1999.
- [6] D. Verkest, J. Cockx, F. Potargent, G. Jong, and H. De Man. On the use of c++ for system-on-chip design. In *Computer Society Workshop on VLSI*. IEEE, April 1999.
- [7] S. Vernalde, P. Schaumont, and I. Bolsens. An object oriented programming approach for hardware design. In *Computer Society Workshop on VLSI*. IEEE, April 1999.
- [8] Patrick Schaumont, Serge Vernalde, Luc Rijn- ders, Marc Engels, and Ivo Bolsens. A program- ming environment for the design of complex high speed asics. In *Proceedings of DAC*. ACM/IEEE, 1998.
- [9] Kazutoshi Wakabayashi. C-based synthesis ex- periences with a behavior synthesizer, "cyber". In *DATE*, 1999.
- [10] David A. Burgoon. Achieving concurrent engi- neering for complex subsystem design: The case for hardware functionnal modeling using c. In *DesignCon*, 1998.
- [11] David A. Burgoon, Edward W. Powell, Lief J. Sorensen, and John A. Sundragon Waitz. Next generation concurrent engineering for complex dual-platform subsystem design. In *DesignCon*, 2000.
- [12] CoWare N2C home page: <http://www.coware.com>.
- [13] CynApps Tool Suite home page: <http://www.cynapps.com>.
- [14] C Level Design Tool Suite home page: <http://www.cleveldesign.com>.
- [15] SystemC home page: <http://www.systemc.org>.
- [16] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong. *Specification and Design of Em- bedded Systems*. PTR Prentice Hall, 1994.
- [17] Rajesh K. Gupta. Timing-driven system design. In *Computer Society Workshop on VLSI*. IEEE, April 1999.
- [18] John L. Hennessy and David A. Patterson. *Computer Organization and Design: The Hard- ware/Software Interface*. Morgan Kaufmann, 1994.
- [19] Rajesh Gupta Vivek Sinha. Design visualization and entry using structural and functional enti- ties: Yaml. Technical Report TR-xx-99, Univer- sity of California, Irvine, 2000.