Hardware Architecture Modelling using an Object-oriented Method

Frédéric MALLET*, Fernand BOERI* Senior Member IEEE, Jean-François DUBOC**
*Laboratoire I3S, UPRES_A 6070 CNRS,Université de Nice-Sophia Antipolis,41 Bd Napoléon 3
06041 Nice Cédex France. Tel : (+33) 4 93 21 79 58 e-mail : fmallet@i3s.unice.fr, boeri@unice.fr
** VLSI Technology inc., 505 Route des Lucioles, Sophia-Antipolis 06560 Valbonne.
Tel : (+33) 4 92 96 11 81 e-mail : jean-francois.duboc@sophia.europe.vlsi.com

Présenté à : Euromicro'98 à Västeras, Suède, 25-27 Août 1998
Workshop : System Level Design : Architectures, methods and tools.

Hardware Architecture Modelling using an Object-oriented Method

Frédéric MALLET*, Fernand BOERI* Senior Member IEEE, Jean-François DUBOC**
*Laboratoire I3S, UPRES_A 6070 CNRS,Université de Nice-Sophia Antipolis,41 Bd Napoléon 3
06041 Nice Cédex France. Tel : (+33) 4 93 21 79 58 e-mail : fmallet@i3s.unice.fr, boeri@unice.fr
** VLSI Technology inc., 505 Route des Lucioles, Sophia-Antipolis 06560 Valbonne.
Tel : (+33) 4 92 96 11 81 e-mail : jean-francois.duboc@sophia.europe.vlsi.com

Abstract

The very high integration rate and the increasing complexity of digital hardware architectures and embedded applications lead designers to search for new tools and methods. In order to reduce the time-to-market it becomes essential to allow designers to evaluate performances of a given application with the targetted architecture very soon in the design phase. So we have decided to build a modelling simulation environment in order to evaluate the requisite number of cycles for processing a given application with a simple model of a digital hardware architecture.

Then, our main objective and the greatest part of our work is to describe this environnement with an example based on the Pine DSP and some classical digital signal processing applications : FIR, FFT butterfly, Viterbi's Butterfly.

1. Introduction

With VLSI incorporation, we aim at evaluating digital signal processor (DSP) performances in relation to specific applications. So we have decided to build a modelling simulation environment in order to evaluate the requisite number of cycles for processing a given application with a simple model of a digital hardware architecture. Then, we can design a new prototype architecture.

Our main objective and the greatest part of our work is to develop methods and tools to achieve this objective. In order to present results, we will expose the developed framework with an example based on the Pine DSP and some classical digital signal processing applications : FIR, FFT butterfly, Viterbi's butterfly. The application is presented in section 2.

With this objective we designed a generic object-oriented model for digital hardware architectures. This model has been designed using the Object Modelling Technique (OMT) and is presented in section 3.

Then, we designed a graphical interface with the Java language in order to optimize the use of this model. This interface is presented in section 4.

Finally, the obtained results and conclusions are presented in section 5.

2. The targetted application

2.1 Introduction

We choose the Pine which is the simplest DSP from VLSI, because it contains almost all techniques used in other processors and its behaviour is completly known. Its detailed documentation was done using the user's manuals [10] [5]. This processor is designed to compute efficiently applications with multiplication and accumulation sequences as FIR; so this classical transformation is well processed by the Pine.

Nevertheless, FFT butterfly and Viterbi's butterfly are very used but the Pine isn't very efficient for those transformations. Our aim was to validate the Pine model and to improve its design using those three test algorithms.

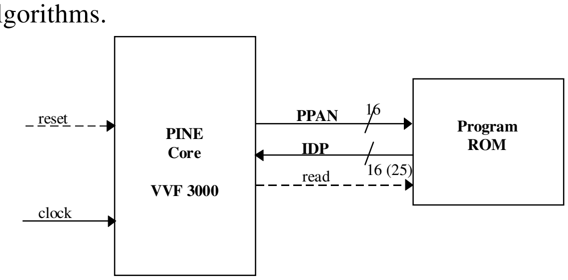

Figure 1 - PINE extended Core « combo »

PPAN : Program Address Bus
IDP : Program Data Bus

2.2 Modelled system overview

For our purpose, we present the following abstraction of the modelled system that is sufficient to illustrate our modelling method.

Figure 1 presents the extended core and Figure 2 presents the core.

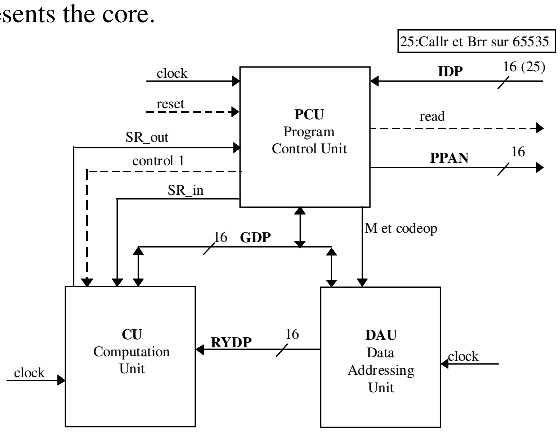

Figure 2 - PINE Core - VVF3000

GDP : Global Data Bus
RYDP : Y Data Bus
SR_in : from status registers
SR_out : Toward status registers
M : Modulo status register bits
codeop : code representing the function to be executed by DAU

3. The generic model

3.1 An object-oriented method

Object-oriented techniques are more and more used for designing software systems, but not for designing complex hardware systems like digital signal processors [9]. Meanwhile, hardware system problems are often closed to software system one's. Nevertheless, object-oriented techniques have several good properties (abstraction, hierarchy, inheritance, polymorphism, encapsulation).

Therefore, main objectives for hardware system designers are to increase component and model reusability with a lowest cost and a highest abstraction level. These objectives should be achieved using good properties from object-oriented techniques [7].

So, our idea is to adapt object-oriented techniques from software systems to hardware architectures and to construct adapted tools. With similar objectives, RASSP [6] and POLIS [4] projects introduced an object-oriented layout upon VHDL.

3.2 Object-Oriented VHDL

Let us note that other designers have looked for such a method based on object-oriented VHDL (OO-VHDL) [2] [1] [3]. Indeed, VHDL is a well-known very used normalised hardware description language and does not include all object-oriented mechanisms.

The RASSP program and the POLIS system are considered by some industrials like very completed but too much complex solutions. Indeed, they propose solution to any problems in a general way, but theirs frameworks become too much complex. We only aim at analyzing performances for a modelled architecture relatively to a specific application. Our proposed framework is more specific so it is smaller but more efficient.

Nevertheless, the proposed tool has to be integrated into a design process fixed by other industrial tools (analysis, design or synthesis tools). All of these tools use VHDL, so we should offer solution for the integration. Possible way to achieve this objective could be the translation from our model to VHDL or the encapsulation of a VHDL architecture into our material Components. A priori, this could raise some implementation problems even if there are lots of similarities between the two approachs.

We have not choosen OO-VHDL, because VHDL is a procedural specification and synthesis language with ADA-like hard types, so it contains lots of useless mechanisms for our purpose. Consequently, adding all object-oriented concepts to VHDL would have resulted into a too much loud solution. Moreover, this could raise technical problems [8].

It seems as if it is a good idea to choose a completly objected-oriented method specifically designed to performance analysis. So we have choosen to use the Object Modelling Technique (OMT) defined by James Rumbaugh as a design technique and Java as a development language. Let us consider our proposition.

3.3 Our proposition

We propose a generic object-oriented model for digital hardware architectures. This model allows us to generate simulable models. Before presenting you with it, we first illustrate it with a simplified presentation of the selected architecture (cf. Figure 1).

3.3.1. Simulation context For simulating the Pine, we need some external components (reset, clock, ROM) in addition of the Pine core. These components constitute the simulation context for Pine (cf. Figure 3).

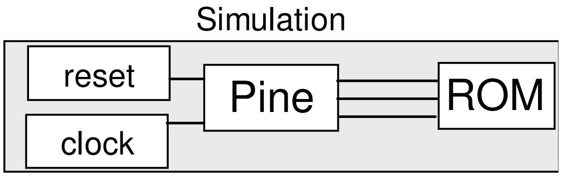

Figure 3 - Simulation context for Pine

For representing this context we constructed a Java application named Simulation. In this class, each component (reset, clock, Pine, ROM, Signal) is represented by another Java class. We just need to instantiate these classes and to make links between components as shown by the following source code.

```
class Simulation extends Frame {
public static void main(String arg[]) {
    Pine pine = new Pine();
    ROM mem = new ROM();
    Actuator rst = new Actuator("reset");
    Actuator clk = new Actuator("clock");
    Signal reset = new Signal();
    reset.linkToPort(rst.getPort("out"));
    reset.linkToPort(pine.getPort("reset"));
    Signal clock = new Signal();
    clock.linkToPort(clk.getPort("out"));
    clock.linkToPort(pine.getPort("clk"));
    Signal ppan = new Signal();
    ppan. linkToPort(pine.getPort("ppan"));
    ppan. linkToPort(mem.getPort("addr"));
    Signal idp = new Signal();
    idp. linkToPort(pine.getPort("idp"));
    idp. linkToPort(mem.getPort("out"));
    Signal read = new Signal();
    read. linkToPort(pine.getPort("readInstr"));
    read. linkToPort(mem.getPort("read"));
}}
```

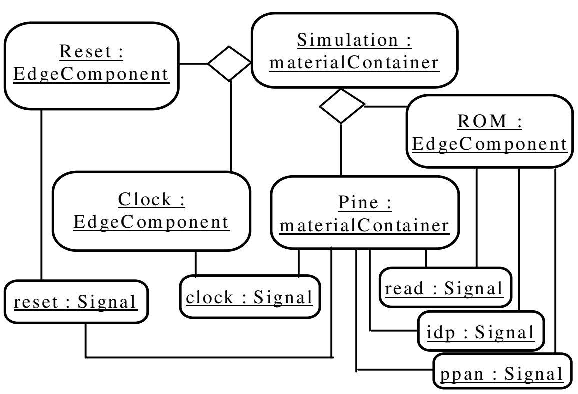

Figure 4 - OMT static object diagram

This context is also described using the OMT static object diagram (cf. Figure 4).

3.3.2. Basic classes for the model It appears that we need to define several classes in order to represent different parts of the modelled system. So we consider digital hardware architectures as **material component**s.

The description of these components can be a behavioural description or a structural description. In the example above, we distinguish three material components, one with a structural description (Pine) and two with a behavioural description (Actuator, ROM). The communication with other components is done across some **port**s. These ports are connected by **signal**s which transport different **value**s. These values can be specialised into several sub-classes (assembly instructions, integers, events, bit values, hardware levels).

In the example, the ROM component can easily be represented with a behavioural description language. In the other hand, the Pine component should be represented with a structural description.

Components whose description is behavioural are called elementary material components and constitute the leafs in our hierarchy. Other components are represented with the **materialContainer** class.

3.3.3 The material container According to this, in what follows we give a rough sketch of the current Java implementation for the Pine component which is a material container.

```
class Pine extends materialContainer {
public Pine() {
    /* variables declaration ... */
    /* components instantiation */
    addMaterialComponent(new PCU());
    addMaterialComponent(new CU());
    addMaterialComponent(new DAU());
    /* signal connections */
    addSignal(GDP = new Signal());
    GDP. linkToPort(pcu.getPort("gdp"));
    GDP. linkToPort(cu.getPort("gdp"));
    GDP. linkToPort(dau.getPort("gdp"));
    /* others .... */
}}
```

A material container only serves to encaspulate material components and signals; it allows a structural description for material components. These components could either be elementary material components or material containers. So the materialContainer class has to inherit from the materialComponent class.

3.3.4. Elementary material components We said before there are components with a simple behaviour like the ROM component. The ROM code is not very interesting to illustrate our behavioural model so the following Java code presents the description of an edge-triggered register (cf. Figure 5) with an asynchronous reset.

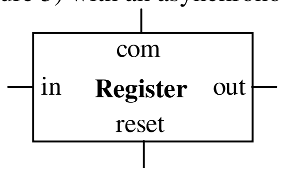

Figure 5 - Edge-triggered Register

```
class Register extends EdgeComponent {
    public Register () {
        addRisingEdgePort("com");
        addFallingEdgePort("reset");
        addPort("in"); addPort("out");
    }
    void report(Value v, String sender) {
        if (sender.equals("reset"))
            value = new IntValue(0);
        else if (sender.equals("com"))
            value = getPort("in").read();
        emit("out", value);
}}
```

The EdgeComponent class inherits from the materialComponent class. It describes material components the behaviour of which is commanded when an edge event occurs on one of the ports. Each port can be responsible for the component sensitivity. Considering the Register component case, the builder declares 'in' and 'out' ports as unsensitive ports, 'com' port as sensitive on rising edge event and 'reset' port as sensitive on falling edge event. So, when an event occurs on a sensitive port, the report method of the component is executed.

3.3.5. Using a library The preceding example shows how we can use existing components and how we can create new components. After the study of several examples of digital architectures, it appears some components which are often used. These components have been implemented and added to a library of standard components (multiplexer, ALU, register, multiplier, memory). Moreover, in order to construct the model of the Pine processor more components were needed (decoder, PC, DAU, PCU, CU). So we give some terminal components which can be reused because they are objects inheriting from the materialComponent class.

But for each new studied system, the user will have to construct its own elementary component library.

3.4 The generic object-oriented model

Until now we have shown how we can deduce some properties for digital hardware architectures. In a more general way, we studied digital architectures and we deduced some general properties most of which are described in section 3.3. These properties have been modelled using the Object Modelling Technique (represented in Figure 6).

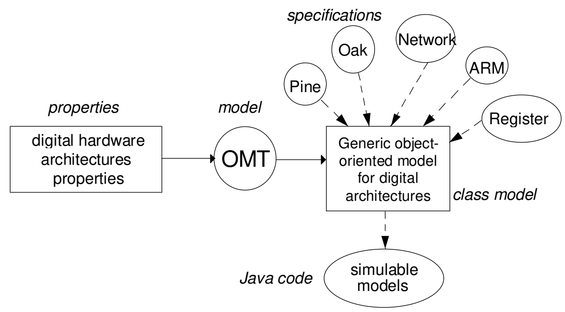

Figure 6 - Representing genericity with OMT

Following this method, we construct a generic object-oriented model for digital hardware architectures. This model is presented by the Figure 7 and contains every deduced properties.

Using this unique model we are able to construct simulable models from specifications of each particular architecture. The application of this generic model to a specific architecture can be boring so we have built a graphical interface ables to apply it in a semi-automatic way.

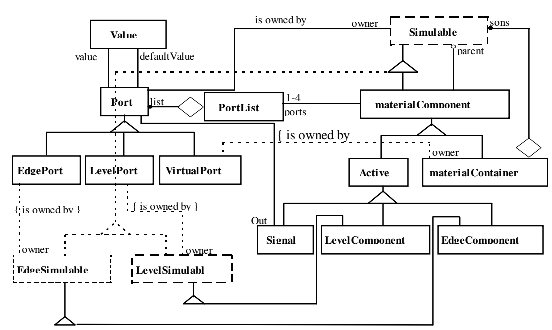

Figure 7 - Generic object-oriented model

4. The framework

4.1 Objectives

The framework aims at allowing hardware architecture designers to make the structural description of the selected digital hardware architecture and to have a given application simulated by the architecture model. Components with behavioural description has to be constructed and compiled separately. With the graphical interface, the designer can add all components, make links between them, add input/output components and start the simulation. He can proceed in an incremental way and choose an adapted abstraction level. The graphical interface can also be used to edit an existing material container by removing or replacing some of the components or links. The modelled architecture could be mono-core or multi-core. The simulated architecture is supposed to be modelled using the proposed generic object-oriented model.

4.2 Applying the model

The Figure 8 shows the graphical interface of the framework.

When the designer want to model a new architecture, he firstly has to identify all elementary material components, to implement their behaviours in Java and to add them to a new library using the "custom" button of the library bar. Most of the needed components should yet be present in the standard library. Then he can build a new material Container with the "Module" menu. He just has to drag and drop existing components from the library bar to the editing zone. If he thinks that the abstraction level is not adapted, he just has to add a new level by inserting a new material container.

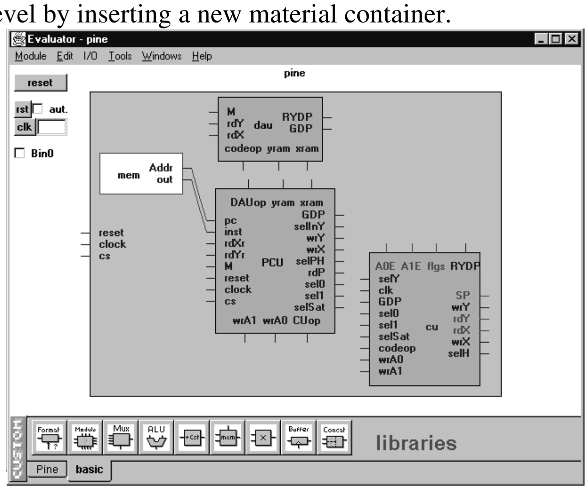

Figure 8 - graphical interface

With the mouse, the designer can also add some new signals or ports to the material containers.

4.3 Describing the targetted application

If the architecture has to process an application, it contains in its design an instruction memory. When the designer model a processor he has to describe the instruction-set of the targetted processor. This could be done in any language from binary description to assembly language or high-level language.

The graphical interface only permits designers to construct every components of the architecture. If the designer wants the model to use a high-level description language, he has to describe in detail the instruction-set and the mean by which the architecture will interpret the application.

We will present now, two possibilities that we have tried, but each designer can use its own method.

4.3.1. Application binary description For a very simple processor, we have described the entire application with a binary description. The instruction memory was loaded with data (Values) corresponding to binary instructions, and the decoder was described with all logic gates. The controller was sending read events to memory and the program counter was generating the right address. When the sequence of bits (instruction) comes to the decoder, all logic gates send the right data values on the data path and generate the right control signals for other components.

4.3.2. Application assembly description For the Pine model, we did not have modelled all logic gates constituting the decoder, because it contains thousand of gates. So we decided to use an assembly description for applications.

We have implemented an elementary material component which is an abstract model of the instruction decoder. This component read instructions from the instruction memory and generate all control and data values (cf. Figure 9). It can be considered as a micro interpreter for assembly instructions.

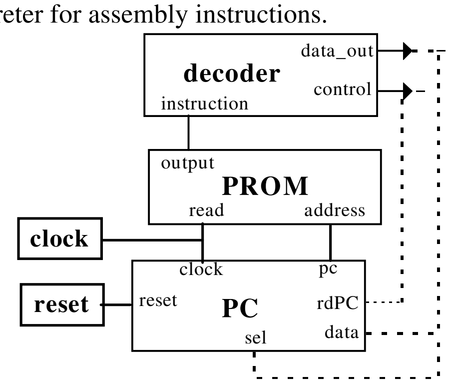

Figure 9 - decoding instructions

So it has been defined an Instruction class which inherits from the Value class. This instruction is transmitted as a normal Value by a Signal. Several subclasses (ALU_Instruction, Branch_Instruction, Address_Instruction) of the Instruction class have been defined in order to model all the Pine processor instruction-set. Each addressing mode has been modelled too. For each of the three selected algorithms, the instruction memory has been loaded with an assembly program as shown by the memory viewer included in the framework (cf. Figure 10).

4.4 Simulation

When the architecture and application description have been done, the simulation can immediatly be started without any compilation phase. Indeed, the designer can send clock or reset events using the input/output components (cf. 4.4.1). The clock events can be sent manually in order to debug or to look in detail each step of the simulation. They can also be sent automatically until the entire application is executed. The tool does not allow to verify that execution times are good, it just permits designers to verify that the design of the architecture does not forbid the execution of the application with respect to constraints described in terms of number of cycles. If results are not satisfying, the designer can immediatly add or replace a component and measures the modification consequences.

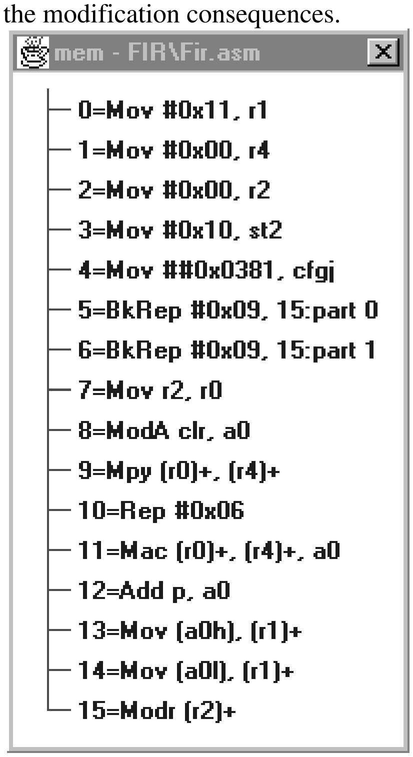

Figure 10 - instruction memory with FIR

He can also change the abstraction level by replacing a component with a behavioural description with a new material container.

4.4.1. The Input/Ouput components We can make a difference between input and ouput components. Input components generate events towards the modelled architecture and output components display some static results. We will describe now, each input/output component.

- Input : Field used to get all digital type inputs.
- Actuator : When this button is clicked, a LOW value followed by a HIGH one are sent on the output port.
- Bin : This is a check button. When it is checked, HIGH value is maintained on the output port. When it is not checked, a LOW value is maintained.
- Clock : This entry is composed by a **'rst'** button, a **'clk'** button, a **'aut.'** button and a counter field. The **'rst'** button make a reset of the clock counter (not the clock component). When **'aut.'** check button is not checked, this is the normal mode, when it is checked this is the automatic mode. The **'clk'** button send ONE edge event (LOW then HIGH) on the clock port in the normal mode. In the automatic mode, this button starts or pauses the automatic sent of edge events on the clock port each 250 ms.
- Sensor : Display *Value*s as plain text. Digital values are laid out in a decimal base format.
- Hexa sensor : Idem than sensor, but digital values are laid out in an hexadecimal base format.

4.4.2. Automatic Java code generation The constructed model can be saved and add to a library. The Java code corresponding to the material container is automatically generated by the framework and has to be compiled. By example, the code present in section 3.3 has been generated with the interface.

4.4.3. Observing results and tools Several tools have been developed and included in the framework.

The **console** window (Figure 11) displays last executed instructions with comments and error messages.

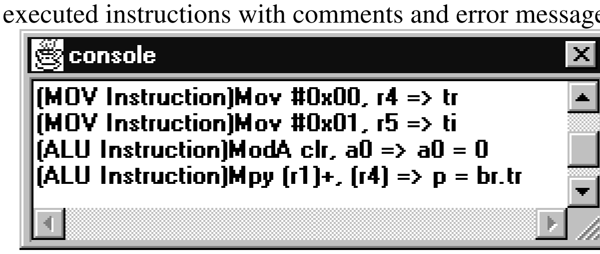

Figure 11 - console screen

At any time, the number of cycle needed to process the application is laid out into the clock input component.

The **memory viewer** shows the content of each memory during the simulation. It can display data memories as well as instruction memories (cf. Figure 10).

With the **register viewer**, the designer can have at any time the state of each register during the simulation.

With the **port status viewer**, the designer knows the value present on a selected port. Using this tool, he can determine when a data path is obstructed by following the path used by a specific data.

In the next release of the tool, an historic of each event will be displayed as a wave form.

5. Results and conclusion

With the proposed method and the graphical framework we have modelled the selected architecture (Pine). The simulation with the selected applications has given exactly right results according to real tests. This is not surprising because of the knowledge we have of the architecture. But we think that with a totally unknown architecture the designer can have realistic estimations of real results. And especially, the designer will be able to improve in an interactive way its architecture and to make quick modifications on its design. As an evidence, the simulation does not allow designer to make any proof, but it gives a good support to improve architectures.

Moreover, good object-oriented properties are conserved and useful:

- **inheritance :** A component can easily inherits from another one. By example, the RAM inherits from the ROM, or the accumulator inherits from the register.
- **polymorphism :** Very simply used. It permits not to duplicate source code the software maintenance is made easier. So the model can be upgraded in an easier way.
- **encapsulation :** Each materialComponent is independent from the others and a function can be encapsulated into a material container. Then the reusability is better because we can reuse a function just inserting a material container. Moreover, once the function is done, external components only see the external ports, they are not interested in the internal description. So the user can choose the abstraction and modelling level for each block in the hardware architecture.

In a future work, we will try to model an entirely asynchronous network with this method.

6. References

[1] *Object-Oriented High-Level Modeling of System Components for the Generation of VHDL Code.*
Karlheinz Agsteiner, Dieter Monjau, Sören Schulze. 0-8186-7156-4/95. IEEE, 1995.

[2] *A proposed Design Objectives Document for Object-Oriented VHDL.*
David L.Barton, Jean Michel Berge. The RASSP Digest - Vol. 3, September 1996.
http://rassp.scra.org/newsletters/96sep/news_18.html.

[3] *Object Oriented Extensions to CHDL, The LaMI proposal.*
Judith Benzakki, Bachir Djaffri. IFIP 1997. Chapman & Hall. p. 334-347.

[4] *A Framework for Harware-Software Co-Design of Embedded Systems : POLIS.*
http://www-cad.eecs.berkeley.edu/Respep/Research/hsc/abstract.html.

[5] *DSP Training.*
VLSI Technology Inc.

[6] *The Rapid prototyping of Application Specific Signal Processors (RASSP) Program.*
http://eti.sysplan.com/ETO/RASSP.

[7] *Object-Oriented Techniques in Hardware Design.*
Sanjaya Kumar, James H.Aylor, Barry W.Johnson, Wm. A. Wulf. Computer, Juin 1994, p. 64-70.

[8] *Inheritance Concept for Signals in Object-Oriented Extensions to VHDL.*
Guido Schumacher, Wolfgang Nebel. Proceedings of the EURO-DAC'95 with EURO-CHDL '95. IEEE Computer Society Press 1995.

[9] *Object-Oriented VHDL Provides New Modeling and Reuse Techniques for RASSP.*
Dr. Sowmitri Swamy, Vista RASSP Program Manager. The RASSP Digest - Vol. 2, No. 1, 1st. Qtr. 1995
http://rassp.scra.org/newsletters/95q1/news_6.html.

[10] *VVF3000 DSP Core user's manual r2.0.*
VLSI Technology Inc.