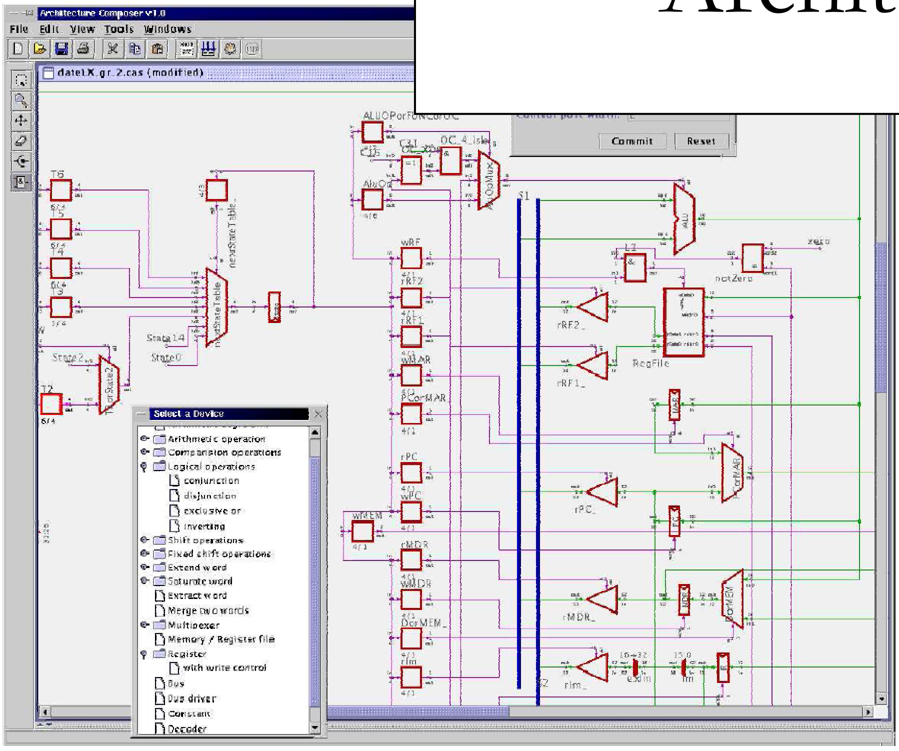
Simulation of SoC Architectures



Part I: Introduction to Modeling, Design and Simulation

I. Introduction

1. The push-pull effect

Modern telecommunication devices are changing rapidly. Not only must designers cope with new emerging technologies, they also have to deal with the needs of the market. A classical push-pull effect can be seen here. From one side, new applications and systems are getting more and more complex, pushing the designer. From the other harsh time-to-market constraints pull him.

The embedded system designer's productivity has to be increased. One answer is to take advantage of programmable architectures. A solution based on programmable architectures has two interacting parts: the hardware and the software. The designer's job is to distribute the complexity of his solution over these parts. On one side, a fully hardware solution (*hard-wired*) may offer great performances, but the overall cost of designing, creating and testing the silicon based solution would be too important. On the other hand a "complete" software solution offers all the ease of being as far as possible from the physical world, but may result in poor performances.

A tradeoff must be found, the designer's work will have to focus on both hardware and software. As a consequence, he will need an appropriate designing platform to be used during the iterative process of prototyping. With hardware involved, a *real* platform would be too expensive and hard to set up. Instead of that, it would be particularly interesting to study the behavior of the hardware components, model it using for example mathematical models, and imitate it as precisely as possible. Combining our *virtual* hardware with the software would need to lead to the same results as if we were working with physical hardware, but only with a fraction of the cost. This technique is called simulation.

2. Definitions

Some definitions need to be given.

Modeling is the activity of representing a possibly existing entity using "intellectual" tools. An example can be found in mechanics where mathematical models represent and describe precisely the movement of a rock swung from a catapult. Apart from mathematical, models may also be constructive. Constructive models are made of a possibly large number of logical statements (for example an *if … then* statement), the execution of those rules leads to a **simulation**.

Designing can be seen as a modeling of a not yet existing entity. Here, the intellectual representation serves as a prototype for a future production.

II. Machine and Hardware Description Languages, the LISA example

This part is based on the paper *"Modeling and Simulation Issues of Programmable Architectures"*, by A. Hoffman, A. Nohl, G Braun, O. Wahlen and H. Meyr, March 2001.

1. Programmable Architecture Overview

The most obvious programmable architecture is the personal computer. Users interact with **user applications** using for example a windowing interface. The user application, in turn, interacts with a "system" application, the **Operating System**. This Operating System provides all user applications with a set of instruction it can carry out, regarding for example manipulation. **POSIX** is an example of a standardization effort of that interface. Finally, the Operating System is the want who "talk" to the **hardware**. This hardware has again an interface of micro instructions it can carry out, called the **instruction set**.

Less complex systems can have only one application talking to the hardware, playing the role of both user and system application. In that case, it would be interesting to customize the instruction set to the needs of the application. ASIP *(Application Specific Instruction Set)* processors follow this idea.

From a designer point of view, the hardware/software optimal interaction can be created only using an appropriate set of simulation tools. The first step is to represent intellectually the behavior of the processor. That's what the LISA project tries to achieve is.

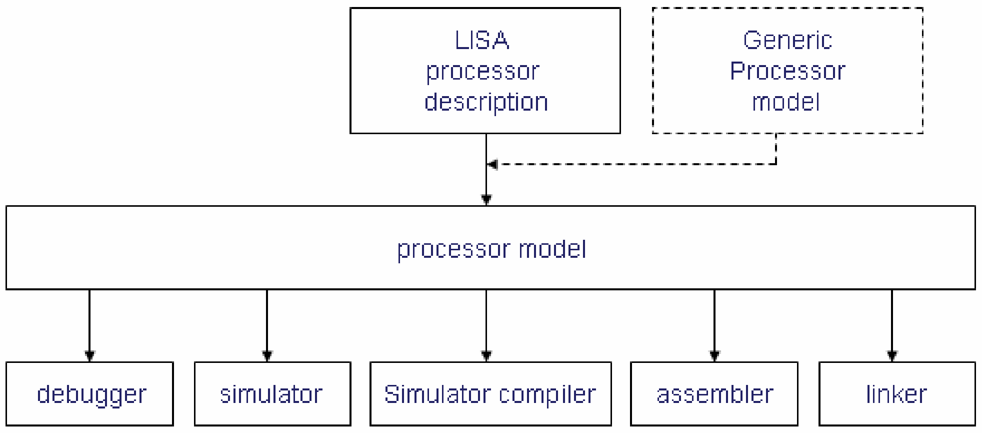
2. The LISA project

This project is conducted at the Aachen University of Technology in Germany.

Many languages were created to represent the processors architecture, but whereas VHDL or Verilog were created for processor designing purposes, a language was needed where all the fine electronic details were hidden and only the useful part was represented. That useful part, from an application point of view, is of course the Instruction Set. That's what LISA *(Language for Instruction Set Architecture)* was designed for.

Nevertheless, the LISA project goes far beyond that scope, and provides the designer with a whole set of tools called the *automatic tool-chain generation*. The idea is that the description of the processor can automatically generate a simulator of that processor, an assembler and linker to translate programs written in high-level languages into executable application, directly usable by the described processor.

A debugger gives a useful graphical interface to the designer, a kind of visual step-by-step overview console of the simulated execution.



The most important (and unique) part of this project is the use of a **simulator compiler**. An analogy can be drawn with programming languages. In interpreted ones, an analysis is done line by line during execution time, leading to poor performances. Compiled languages work on the initial *text* before execution to optimize once and for all the possibly many run times. Here, a highly customized compiler is specifically created for every processor / program / scenario, reaching simulation speeds in the order of 100K instructions per second.

3. Validation of the tools

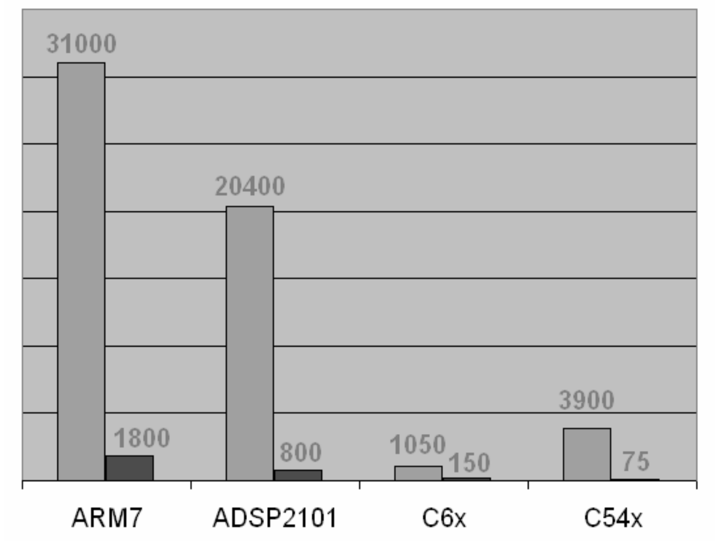
Ones the tools are set up, it is important to show they work properly. In the case of simulation, two main criteria have to be taken into account: the modeling capabilities and the efficiency.

Four different commercial architectures will be modeled by LISA: ARM7, ADSP2101, TMS320C54x and TMS320C62x. It is to be noted that the validation needs to be done in modeling, not designing mode. Each of those architectures has very specific characteristics regarding for example very large instruction word length (VLIW), or multiple instruction words. The LISA architecture permits, in a reasonable amount of time, to model those four architectures.

The final *test* for LISA would be the performance. As we are talking about simulation, performance would be the number of instructions or cycles the simulator can carry out in one second. As a comparison, the efficiency of the official simulator hipped with the processor are given. We can see the performances are better by one or two degrees of magnitude. This is really the strong part of LISA, and is totally due to the use of compiled simulators.

In thousands of instructions / cycles per second.

The left bar represents the performance of the LISA simulator; the right simulation uses the official simulator.



4. Interest of the paper

Together with the Ptolemy project, the LISA project gives us a very clear insight view of the importance and possible uses of simulation assisted design. What's more, the use of compiled simulation is a very clever way of dealing with the traditional performance issues.

Nevertheless, it would have been useful to have a more explicit presentation of the syntax and semantics of the LISA languages.

Part II: Modeling and Simulation of Embedded Processors Using Abstract State Machines

III. Architecture/compiler co-design

1. The needs

As described in the introductory part, today's designers need to cope with both efficiency and time-to-marked constraints. Whereas programmable architectures provide the flexibility, their primary drawback is their performance. To counter this problem, special purpose optimized processors are created: the ASIP family. In *Application Specific Instruction set Processors*, the instruction set (the programmable interface the architecture provides) is customized to match the needs of each specific application as well as possible.

2. The process of architecture/compiler co-design

The scope of the design process in enlarged to both hardware and software, these two components forming one block. The big designing challenge is where to put the complexity of the calculation to be done. On one hand, a complete hard-wired solution gives excellent results as far as the performance is concerned, but the cost of development, creation and testing may be enormous. In this solution, the instruction set will be very complex, while the application will be restricted to the strict minimum. On the other hand, a software based solution will transpose the complexity from processor to application. Only a simplified instruction set will be needed. The tradeoff between hardware and software has to be found.

In order to explore the many possibilities during the iterative process of prototyping, the most effective solution is the use of a simulator. Not only should it be able to monitor the simulated environment, it should be able to create a complete development environment with a description of the hardware only. This is called Architecture/Compiler co-design.

3. Related work

Three research groups can be pointed out working on architecture /compiler co-design. They all try to achieve the same goal, but of course present many differences. The first difference in the speed they may work at, that is to speed the simulation of compiling speed. What's more, they will all use different models, mathematical or not, who will be able to model in an optimal way only a part of the range of architectures. It is to note that the results each solution provide, especially the execution times presented, have to be taken with care. An important phase of the validation of a simulator is to compare the execution time on the simulator and the real hardware.

The first research group is LISA, at the University of Aachen in Germany. It has been presented in Part I of this document. The main specificity of the proposed simulation environment is the use of a compiled simulator. Exactly as in programming languages, a compiled solution has much better performances as an interpreted counter part.

The specificity of the CASTEL project is that its goal is not directly to provide a simulation environment, but focuses more on the modeling part. Indeed, the tool presented lets us use a VHDL representation of the hardware as the register transfer level, and creates a mathematical model of the data path using extended finite state machines.

Finally, the last research group presented here is the one closest to the BUILDABONG project we will present. The EXPRESSION framework lets us design an architecture graphically using V-SAT, translate it automatically into an

EXPRESSION model, which in turn can generate both a compiler and a simulator.

IV. Abstract State Machines

1. The mathematical tool

In mathematics, the ensemble of entities one is working on is called the universe. Inside that universe, a function lets us transform one of those entities into another, letting us "move". A complete structure is achieved if we also add relations, that is to say operators that let us compare two entities. Without relations, that is to say with only functions, the structure is called and algebra.

An Abstract State Machine (ASM) is defined as the association of a finite vocabulary and a finite set off n-ary functions over that vocabulary. It is to note that a state of an ASM is al algebra. What's more, the ASM is completely defined with an initial state S_0 and a set of transition rules P. A transition rule is nothing more than a "if … then" statement, where the possibly executed rule is an update rule, the condition a Boolean valued expressions (an explicit relation can not be expressed, as we are working in a algebra).

As for the operational semantics, the ASM can be seen as an initial state on which a set of transition rules is applied iteratively. The ASM will end up in a terminal state where the transitions rules will have no effect (the terminal state is detected after two consecutive transformations leading to the same state).

2. Modeling processors using ASMs

The granularity of the processor model we are looking for is described as at least RTL (Register Transfer Level) and cycle accurate. A register transfer is physically conditioned by for example the value of special mode registers, or the state of special purpose bits. As a consequence, a register transfer pattern will only be executed if a register transfer condition is met, leading to a classic if … elsa . These guarded register transfer patterns are naturally modeled by ASMs.

3. Advantages

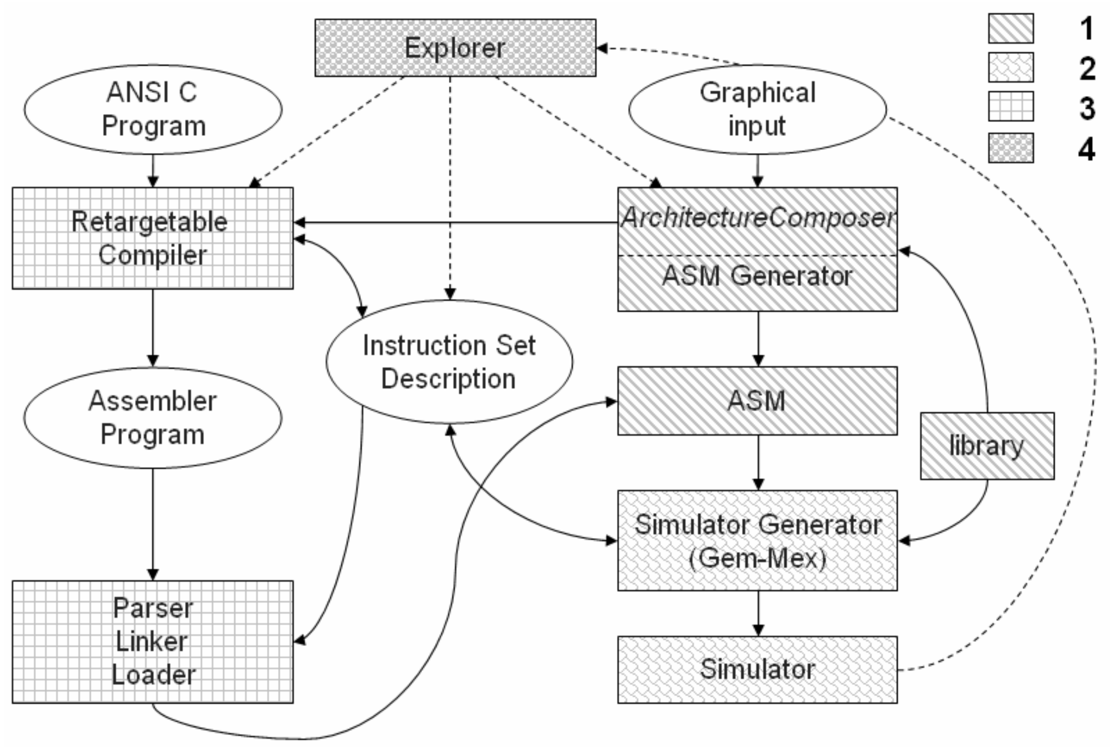
Apart from the fact that ASMs and a processor described at the RTL have the same intellectual representation, using ASMs to model architectures present some other advantages. The ASM representation is relatively short (200 lines for the complete ARM7 description) and easy to read. The model is cycle accurate and granular at the RTL. Even though the paper studied list the simulation speed as an advantage, neither values nor comparisons are given. Finally, the language called XASM used to describe the ASM include in its native syntax the call of external C libraries. This will be most helpful for supporting irregular arithmetic operations on arbitrary large word-lengths.

V. The BUILDABONG project

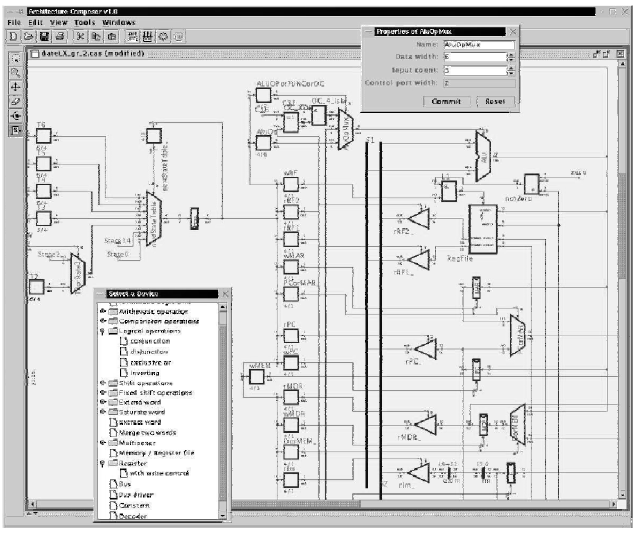
1. General view : the development phases

The BUILDABONG project is a very ambitious project as it should provide a complete architecture/compiler co-design framework. Because of this importance, the development has been divided in four phase, currently phase 2 is completed and phase 3 is on his way. The basic idea is to provide the system with a graphical representation of an architecture using the *ArchitectureComposer*, having is translated in XASM code and generate both a compiler and a simulator. Finally, an explorer will provide the designer with a user-friendly interface to steer the designing.

A graphical overview of the development phases is presented on the next page.



2. The graphical editor

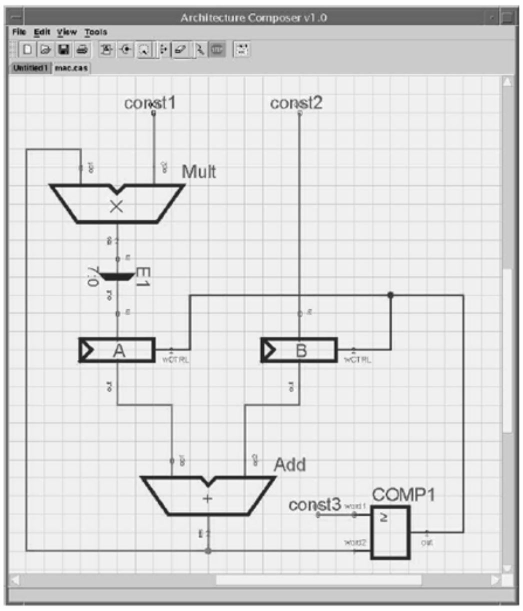


The first module the designer will have to work with is the *ArchitectureComposer*, or graphical Editor. A screenshot is presented on the right. A library of customizable components used as graphical building bricks fastens the overall editing process.

This system has a serious drawback if you keep in mind that every simulator environment (like EXPRESSION) uses his own graphical editor and hardware description language. It would be interesting to build a module capable of importing directly hardware descriptions in widely used HDLs (Hardware Description Languages) as VHDL or Verilog.

3. automatic XASM-code generation

After the architecture is entered in the framework, it has to be described using ASM, and the associated XASM code has to be generated. To illustrate the translation algorithm, a simple example is given here. We will study the different phases.

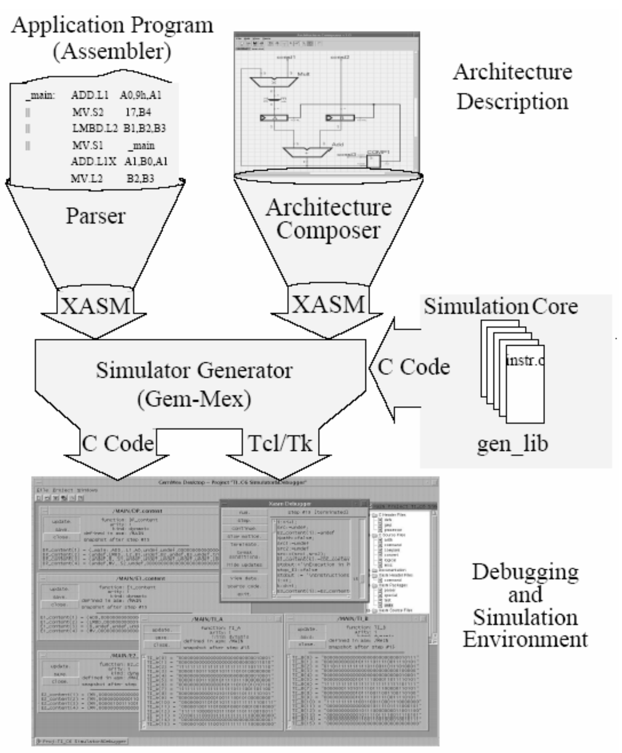


```
     asm MAIN is
(1)  [ use cpucore
     ┌ derived function Mult_res ==
     │       c_mult (Add_res, const1_out, 8, COMPLEMENT_2)
     │ derived function Add_res ==
     │       c_add (A_out, B_out, 8, COMPLEMENT_2)
     │ derived function E1_out ==
     │       c_extract (Mult_res, 16, 0, 8)
(2)  │ derived function COMP1_out ==
     │       c_gteq (const3_out, Add_res, 8, COMPLEMENT_2)
     │ function A_out
     │ function B_out
     │ function const1_out
     │ function const2_out
     └ function const3_out
     ┌ init
     │     A_out := "00000000"
     │     B_out := "00000000"
(3)  │     const1_out := "00000010"
     │     const2_out := "00000001"
     │     const3_out := "00001000"
     └ endinit
     ┌ if COMP1_out = "1" then A_out := E1_out
(4)  │ endif
     │ if COMP1_out = "1" then B_out := const2_out
     └ endif
     endasm
```

At phase (1), what could be seen as a C-type "include" is used. The *cpucore* package is invoked, letting us use more complex operators as *c_mult*, *c_add*, *c_extract* or *c-gteq*. At (2), a certain number of functions are declared. They refer to the components of the circuit, as well as the links between them. At phase (3), the sequential elements are initialized. Are considered sequential all the elements that contain information, e.g. registers, the *Mult* component is not sequential so does not need to be initialized. Finally, the phase (4) defines the guards of the register transfer patterns.

4. Generating the simulator

The last part of the architecture built by now by the BUILDABONG team is the simulator generation. Here, the XASM code representing the architecture is brought together with that representing the Application Program and with the use of Gem-Mex, the debugging and simulation environment appears.



```
_main:  ADD.L1   A0,9h,A1
        MV.S2    17,B4
        LMBD.L2  B1,B2,B3
        MV.S1    _main
        ADD.L1X  A1,B0,A1
        MV.L2    B2,B3
```

Two essential questions arise from this way of working. How do you create the XASM code of the Application Program? Is XASM an efficient way to represent Application Program code? The answer to the first question is patience, as phase 3 of the BUILDABONG project aims at creating a compiler that will be able to translate the program into assembler language and XASM representation. The answer to the second question is less obvious. Indeed, ASM has been proven a good modeling language for processor architecture, but not for Assembler Application Program Code. The answer should be given in the next paper about BUILDABONG phase 3.

5. Future Work

As depicted in the previous paragraph, phases 3 and 4 of the BUILDABONG project still have to be finalized. Whereas phase 4 is mainly about user-friendliness and concise presentation of the processors inner state, I suspect phase 3 to be the most challenging part of the project. Indeed, lots of questions arise, as the compiler will have to be customizable to meet different needs as power use or Real-Time constraints.

VI. Critical view over the paper

Even though a certain number of architecture/compiler co-design frameworks already exist, BUILDABONG is very interesting due to the fact it uses a formal mathematical representation, the Abstract State Machines. What's more, the structured project planning gives us a clear view of the work (to be) done, as provides us with already useable tools. Nevertheless, a serious drawback of the overall system is the use of a non-standard hardware description language. The description of a complete processor may take as much time as the analysis of the simulation results, which is not acceptable. As a consequence, it would be interesting to be able to import for example VHDL representations. What's more, it would be very useful to create in phase 3 a parametrable compiler, to take into account specific needs as battery life or real-time constraints. Finally, as good as the model is in a simulation environment, it is vital to perform comparison tests between an existing architecture and its model. The results of this type of tests should be presented in the next paper of BUILDABONG phase 3 and/or 4.