

A Programming Environment for the Design of Complex High Speed ASICs

Patrick Schaumont Serge Vernalde Luc Rijnders Marc Engels Ivo Bolsens
IMEC vzw, Kapeldreef 75, B-3001 Leuven

Abstract

A C++ based programming environment for the design of complex high speed ASICs is presented. The design of a 75 Kgate DECT transceiver is used as a driver example. Compact descriptions, combined with efficient simulation and synthesis strategies are essential for the design of such a complex system. It is shown how a C++ programming approach outperforms traditional HDL-based methods.

1 Introduction

In this contribution, we present a programming environment based on C++ that supports simulation, verification and synthesis of complex high speed ASICs for digital telecommunications. It is part of a larger environment that targets an automated synthesis path from the Matlab algorithm level to the VHDL architecture level [8].

In order to introduce the requirements put on to such an environment, a recent design experience will be documented. The design consists of a digital radiolink transceiver ASIC, residing in a DECT base station (figure 1). The chip processes DECT burst signals, received through a radio frequency front-end RF. The signals are equalized to remove the multi-path distortions introduced in the radio link. Next, they are passed to a wire-link driver DR, that establishes communication with the base station controller BSC. The system is also controlled locally by means of a control component CTL.

The specifications that come with the design of the digital transceiver ASIC in this system are as follows:

- The equalization involves complex signal processing, and is described and verified inside a high level design environment such as Matlab.
- The interfacing towards the control component CTL and the wire-link driver DR on the other hand

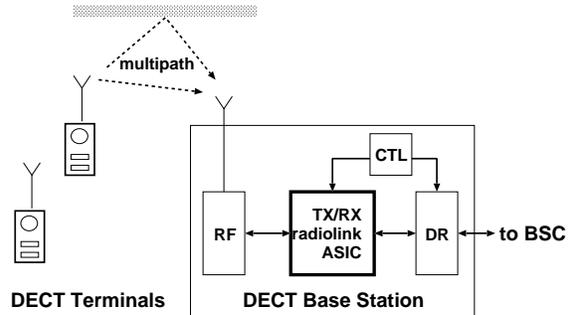


Figure 1: DECT Base Station Configuration

is described as a detailed clock-cycle true protocol.

- The allowed processing latency is, due to the real time operation requirements, very low: a delay of only 29 DECT symbols (25.2 μ secs) is allowed. The complexity of the equalization algorithm, on the other hand, requires up to 152 data multiplies per DECT symbol to be performed. This implies the use of parallel data processing, and introduces a severe control problem.
- The scheduled design time to arrive from the heterogeneous set of specifications to the verified gate level netlist, is 18 person-weeks.

The most important degree of freedom in this design process is the target architecture, which must be chosen such that the requirements are met. Due to the critical design time, a maximum of control over the design process is required. To achieve this, we use a programming approach to implementation, in which the system is modeled in C++. The object oriented features of this language allows it to mix high-level descriptions of undesigned components with detailed clock-cycle true, bit-true descriptions. In addition, appropriate object modeling allows the detailed descriptions to be translated to synthesizable HDL automatically. Finally, verification test-benches can be generated automatically in correspondence with the C++ simulation.

The result of this design effort is a 75 Kgate chip with a VLIW architecture, including 22 datapaths, each decoding between 2 and 57 instructions, and including 7 RAM cells. The chip has a 194 mm^2 die area in 0.7 μ CMOS technology.

The C++ programming environment allows it to obtain results faster than existing approaches. Related to register transfer design environments such as [10], it will be shown that C++ enables more compact, and consequently less error prone descriptions of hardware. High level synthesis environments [9] could solve this problem but have to fix the target architecture in beforehand. As will be described in the case of the DECT transceiver design, sudden changes in target architecture can occur due to hard initial requirements, that can be verified only at system implementation.

On the other hand, recent works in the research community propose to use high level languages as the specification for hardware synthesis [4, 1] and modeling [12, 13]. This makes us believe that a programming approach goes beyond the capabilities of traditional HDL.

In this paper, we will present our design environment as follows. First, the system machine model is introduced (section 2). This model includes two types of description: high-level *untimed* ones and detailed *timed* blocks (section 3). Using such a model, a simulation mechanism is constructed (section 4). It will be shown that the proposed approach outperforms current environments in code size and simulation speed. Following this, HDL code generation issues (section 5) and hardware synthesis strategies (section 6) are described. A summary of the contributions is given in 7.

2 System Machine Model

Due to the high data processing parallelism, the DECT transceiver is best described with a set of concurrent processes. Each process translates to one component in the final system implementation.

At the system level, processes execute using data-flow simulation semantics. That is, a process is described as an iterative behavior, where inputs are read in at the start of an iteration, and outputs are produced at the end. Process execution can start as soon as the required input values are available.

Inside of each process, two types of description are possible. The first one is a high level description, and can be expressed using procedural C++ constructs. A firing rule is also added to allow data-flow simulation [7, 6].

The second flavor of processes is described at register transfer level. These processes operate synchronously to the system clock. One iteration of such a process corresponds to one clock cycle of processing.

For system simulation, two schedulers are available. A data-flow scheduler is used to simulate a system that contains only untimed blocks. This scheduler repeatedly checks process firing rules, selecting processes for execution as their inputs are available.

When the system also contains timed blocks, a cycle scheduler is used instead. The cycle scheduler manages to interleave execution of multi-cycle descriptions,

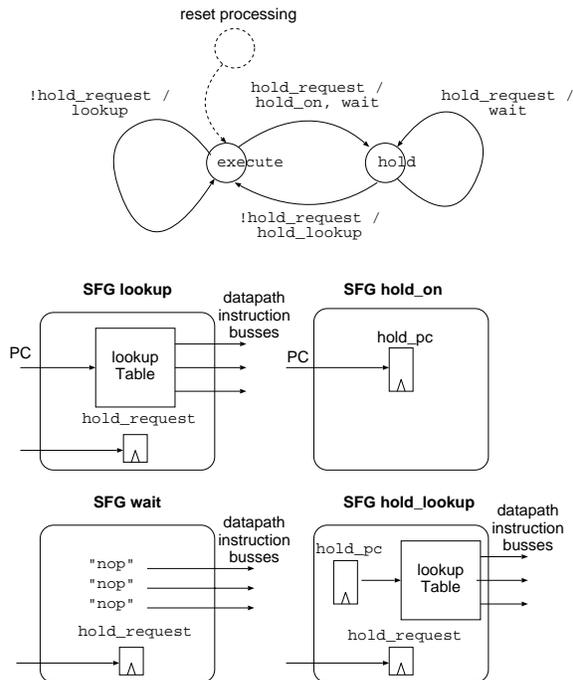


Figure 2: Cycle True Model Example

but can incorporate untimed blocks as well. The ability to mix both high level descriptions and detailed clock cycle true descriptions is essential in maintaining an executable system specification at all times. A detailed description of the cycle scheduler operation is given in section 4.

In the following section, we will focus on the modeling of detailed, timed descriptions, and motivate why the proposed model is at the right level for a telecommunications ASIC.

3 Cycle-true Descriptions

Detailed process descriptions reflect the hardware behavior of a component at the same level of the implementation. To gain simulation performance and coding effort, several abstractions are made.

- Finite Wordlength effects are simulated with a C++ fixed point library. It has been shown that the simulation of these effects is easy in C++ [5, 11]. Also, the simulation of the quantization rather than the bit-vector representation allows significant simulation speedups.
- The behavior is modeled with a mixed control/data processing description, under the form of a finite state machine coupled to a datapath. This model is common in the synthesis community [2]. In high throughput telecommunications circuits such as the ones in the DECT transceiver ASIC, it most often occurs that the desired component architecture is known before the hardware description is made. The FSM/DP model works well for these type of components.

Sig Class

```
class sig {
    Value value;
    char *name;
public:
    sig(Value v);
    sig operator +(sig v);
    virtual Value simulate();
    virtual void gen_code(ostream &os);
};

sig sig::operator +(sig v) {
    sigadd add;
    add.left = &v;
    add.right = this;
    return add;
}

Value sig::simulate() {
    return value;
}

sig::gen_code(ostream &os) {
    os << name;
}
```

Derived Operator Class

```
class sigadd : public sig {
    sig *left;
    sig *right;
public:
    Value simulate();
    void gen_code(ostream &os);
};

Value sigadd::simulate() {
    return left->simulate() +
           right->simulate();
}

sigadd::gen_code(ostream &os) {
    os << left->cg()
       << " + "
       << right->cg();
}
```

Figure 3: C++ SFG construction class

The two aspects, wordlength modeling and cycle true modeling, are available in the programming environment as separate class hierarchies. Therefore, fixed point modeling can be applied equally well to high level descriptions.

As an illustration of cycle true modeling, a part of the central VLIW controller description for the DECT transceiver ASIC is shown in figure 2. The top shows a Mealy type finite state machine. As actions, the signal flow graph descriptions below it are executed. The two states `execute` and `hold` correspond to operational and idle states of the DECT system respectively. The conditions are stored in registers inside the signal flow graphs. In this case, the condition `hold_request` is related to an external pin.

In state `execute`, instructions are distributed to the datapaths. Instructions are retrieved out of a lookup table, addressed by a program counter. When `hold_request` is asserted, the current instruction is delayed for execution, and the program counter PC is stored in an internal register. During a hold, a `nop` instruction is distributed to the datapaths to freeze the datapath state. As soon as `hold_request` is removed, the stored program counter `hold_pc` addresses the lookup table, and the interrupted instruction is issued to the datapaths for execution.

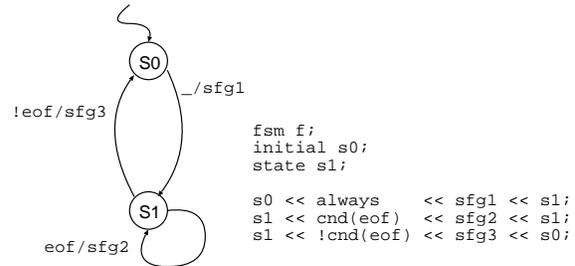


Figure 4: C++ FSM description

We next discuss the C++ objects that allow us to capture clock cycle true behavior.

3.1 Signals and Signal Flow Graphs

Signals are the information carriers used in construction of a timed description. Signals are simulated using C++ `sig` objects. These are either plain signals or else registered signals. In the latter case the signals have a current value and next value, which is accessed at signal reference and assignment respectively. Registered signals are related to a clock object `clk` that controls signal update. Both types of signals can be either floating point values or else simulated fixed point values.

Using operations, signals are assembled to expressions. By using the overloading mechanism as shown in figure 3, the parser of the C++ compiler is reused to construct the signal flow graph data structure.

A set of `sig` expressions can be assembled in a signal flow graph (SFG). In addition, the desired inputs and outputs of the signal flowgraph have to be indicated. This allows to do semantical checks such as dangling input and dead code detection, which warn the user of code inconsistency.

An SFG has well defined simulation semantics and represents one clock cycle of data processing.

3.2 Finite State Machines

After all instructions are described as SFG objects, the control behavior of the component has to be described. We use a Mealy-type FSM model to do this.

Again, the use of C++ objects allow to obtain very compact and efficient descriptions. Figure 4 shows a graphical and C++-textual description of the same FSM. The correspondence is obvious. To describe an equivalent FSM in an event driven HDL, one usually has to follow the HDL simulator semantics, and for example use multi-process modeling. By using C++ on the other hand, the semantics can be adapted depending on the type of object processed, all within the same piece of source code.

3.3 DECT Transceiver Architecture

The FSM model described above was used to construct an ASIC architecture shown in figure 5. It consists of a central (VLIW) controller, a program counter controller and 22 datapath blocks. Each of these are modeled with the combined control/data processing

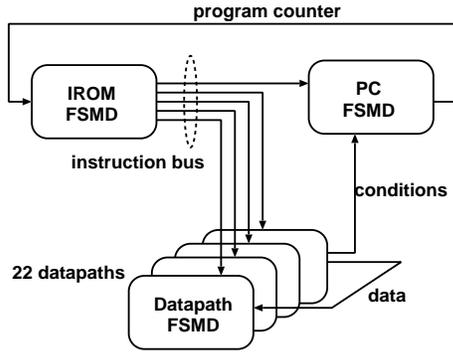


Figure 5: DECT Transceiver System Architecture

shown above. They exchange data signals that depending on the particular block, are interpreted as instructions, conditions or signal values. By means of these interconnected FSMD machines, a more complex machine was constructed.

It is now motivated why this centrally controlled architecture was chosen. For the DECT transceiver, there is a severe latency requirement. Originally, a data-flow target architecture was chosen which is common for these types of telecommunications signal processing. In such an architecture, the individual components are controlled locally and data driven. However, the extreme latency requirement (29 DECT symbols) required the introduction of global exceptions. In a data driven architecture however, such global exceptions are very difficult to implement. This is far more easy in a central control architecture, where they take the form of a jump in the instruction ROM. Because of these difficulties, the target architecture was changed from data driven to central control.

The FSMD machine model allowed to reuse the datapath descriptions and only required the control descriptions to be reworked. This architectural change was done *during* the 18-week design cycle, which shows the flexibility of using C++ as modeling mechanism.

4 The Cycle Scheduler

Whenever a timed description is to be simulated, a cycle scheduler is used instead of a data-flow scheduler. The cycle scheduler creates the illusion of concurrency between components on a clock cycle basis.

The operation of the cycle scheduler is best illustrated with an example. In figure 6, the simulation of one cycle in a system with three components is shown. The first two, components 1 and 2, are timed descriptions constructed using `fsm` and `sfg` objects. Component 3 on the other hand is described at high level using a firing rule and a behavior. In the DECT transceiver, such a loop of detailed (timed) and high level (untimed) components occurs for instance in the RAM cells that are attached to the datapaths. In that case, the RAM cells are described at high level while the datapaths are described at clock cycle true level.

The simulation of one clock cycle is done in three phases. Traditional RT simulation uses only two; the first being an evaluation phase, and the second being

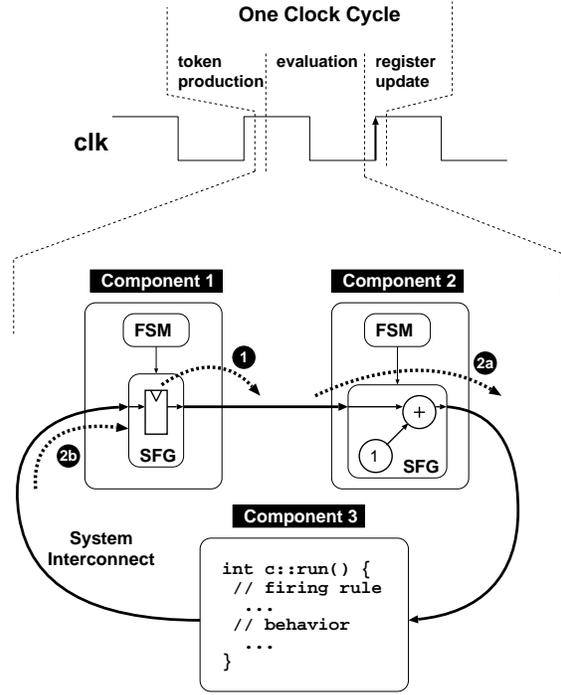


Figure 6: Cycle Scheduler Operation

a register update phase.

The three phases used by the cycle scheduler are a token production phase, an evaluation phase and a register update phase. The three-phase simulation mechanism is needed to avoid apparent deadlocks that might exist at the system level. Indeed, in the example there is a circular dependency in between components 1, 2, and 3, and a data-flow scheduler can no longer select which of the three components should be executed first. In data-flow simulation, this is solved by introducing *initial tokens* on the data dependencies. Doing so would however require us to devise a buffer implementation for the system interconnect, and introduce an extra code generator in the system.

The cycle scheduler avoids this by creating the required initial tokens in the token production phase. Each of the phases operates as follows.

- 0) At the start of each clock cycle, the `sfg` descriptions to be executed in the current clock cycle are selected. In each `fsm` description, a transition is selected, and the `sfg` related to this transition are marked for execution.
- 1) **Token production** phase. For each marked `sfg`, look into the signal flow graph, and identify the outputs that solely depend on registered signals and/or constant signals. Evaluate these outputs and put the obtained tokens onto the system interconnect.
- 2a) **Evaluation** phase (case a). In the second phase, schedule marked `sfg` and untimed blocks for execution until all marked `sfg` have fired. Output tokens are produced if they are directly dependent on input tokens for timed `sfg` descriptions, or else if they are outputs of untimed blocks.

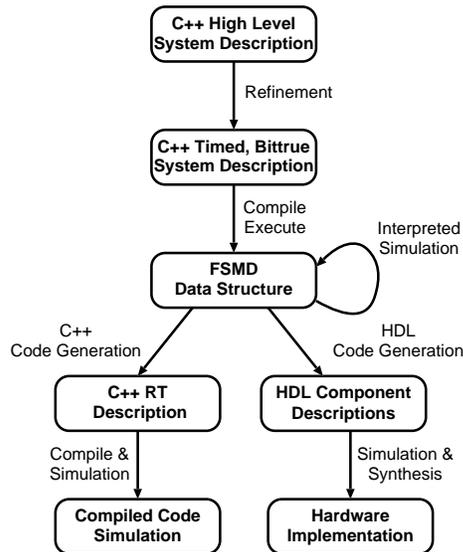


Figure 7: Code Generation and Simulation

- 2b) **Evaluation** phase (case b). Outputs that are however only dependent on registered signals or constants will not be produced in the evaluation phase.
- 3) **Register update** phase. For all registered signals in marked `sfg`, copy the next-value to the current-value.

The evaluation phase of the three-phase simulation is an iterative process. If a preset amount of iterations have passed, and there are still unfired components, then the system is declared to be deadlocked. This way, the cycle scheduler identifies combinatorial loops in the system.

5 Code Generation and Simulation Strategy

The clock-cycle true, bit-true description of system components serves a dual purpose. First, the descriptions have to be simulated in order to validate them. Next, the descriptions have also to be translated to an equivalent, synthesizable HDL description.

In view of these requirements, the C++ description itself can be treated in two ways in the programming environment. In case of a *compiled code* approach, the C++ description is translated to directly executable code. In case of an *interpreted* approach, the C++ description is preprocessed by the design system and stored as a data structure in memory.

Both approaches have different advantages and uses. For simulation, execution speed is of primary importance. Therefore, compiled code simulation is needed. On the other hand, HDL code generation requires the C++ description to be available as a data structure that can be processed by a code generator. Therefore, a code generator requires an interpreted approach.

We solve this dual goal by using a strategy as shown in figure 7. The clock-cycle true and bit-true description of the system is compiled and executed. The description uses C++ objects such as signals and finite

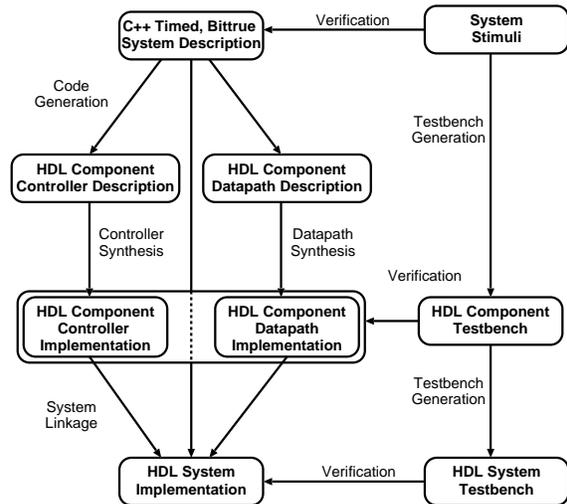


Figure 8: Hardware Synthesis Strategy

state machine descriptions which translate themselves to a control/data flow data structure.

This data structure can next be interpreted by a simulator for quick verification purposes. The same data structure is also processed by a code generator to yield two different descriptions.

- A C++ description can be regenerated to yield an application-specific and optimized compiled code simulator. This simulator is used for extensive verification of the design because of the efficient simulation runtimes.
- A synthesizable HDL description can also be generated to arrive at a gate-level implementation.

The simulation performance difference between these three formats (interpreted C++ objects, compiled C++, and HDL) is illustrated in table 1. Simulation results are shown for the DECT header correlator processor, and also the complete DECT transceiver ASIC.

The C++ modeling gains a factor of 5 in code size (for the interpreted-object approach) over RT-VHDL modeling. This is an important advantage given the short design cycle for the system. Compiled code C++ on the other hand provides faster simulation and smaller process size than RT-VHDL.

For reference, results of netlist-level VHDL and Verilog simulations are given also.

6 Synthesis Strategy

Finally, we document the synthesis approach that was used for the DECT transceiver. As shown in figure 8, the clock-cycle true, bit-true C++ description can be translated from within the programming environment into equivalent HDL.

For each component, a controller description and a datapath description is generated, in correspondence with the C++ description. This is done because we rely on separate synthesis tools for both parts, each one optimized towards controller or else datapath synthesis tasks.

Design	Size (Gates)	Type	Source Code (# lines)	Simulation Speed (cycles/sec)	Process Size (Mbyte)
HCOR	6K	C++ (interpreted obj)	230	69	3.8
		C++ (compiled)	1.7K	819	2.7
		VHDL (RT)	1.6K	251	11.9
		VHDL (netlist)	77K	2.7	81.5
DECT	75K	C++ (interpreted obj)	8K	2.9	20
		C++ (compiled)	26K	60	5.1
		Verilog (netlist)	59K	18.3	100

Table 1: Performances of interpreted and compiled approaches

For datapath synthesis, we rely on the Cathedral-3 back-end datapath synthesis tools [3], that allow to obtain a bit-parallel hardware implementation starting from a set of signal flow graphs. These tools allow operator sharing at word level, and require run times less than 15 minutes even for the most complex, 57-instruction datapath of the DECT transceiver.

Controller synthesis on the other hand is done by logic synthesis such as Synopsys DC. For pure logic synthesis such as FSM synthesis, this tool produces efficient results. The combined netlists of datapath and controller are also post-optimized by Synopsys DC to perform gate-level netlist optimizations. This divide and conquer strategy towards synthesis allows each tool to be applied at the right place.

During system simulation, the system stimuli are also translated into test-benches that allow to verify the synthesis result of each component. After interconnecting all synthesized components into the system netlist, the final implementation can also be verified using a generated system test-bench.

7 Conclusions

In this paper, we illustrated the use of C++ for system synthesis by the design of a DECT transceiver. As systems grow more complex, designers tend to rely on higher levels of modeling to increase design cycle speed. The presented programming environment allows to use this higher level of modeling for both simulation and synthesis. The description style for synthesis is a behavioral register transfer model. The writing of HDL is avoided through code generation from C++. The design environment is also lightweight, and uses only a C++ compiler and library to do the system capture and simulation. Finally, as the C++ library is generic, it is currently being reused for several demonstrator designs done, including an upstream cable modem, an image compressor and a wireless LAN modem.

Acknowledgements

The authors wish to thank Francky Catthoor and Hugo De Man from IMEC for the constructive discussions and review by the writing of this paper. Also, the design efforts of Radim Cmar from IMEC were invaluable in the construction of the DECT transceiver. Marc Engels is a senior research assistant of the Belgium National Fund for Scientific Research. This work was carried out under the Flemish Government Impulse Program for Information Technology (IT-ISIS).

References

- [1] G. Dejong, B. Lin, C. Verdonck, S. Wuytack, and F. Catthoor. Background memory management for dynamic data structure intensive processing systems. In *Proc ICCAD 95*, pages 515 – 520, November 1995.
- [2] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High Level Synthesis*. Kluwer Academic Publishers, 1992.
- [3] W. Geurts, F. Catthoor, S. Vernalde, and H. De Man. *Accelerator Data-Path Synthesis for High-Throughput Signal Processing Applications*. Kluwer Academic Publishers, 1996.
- [4] R. K. Gupta and S. Y. Liao. Using a programming language for digital system design. *IEEE Design and Test of Computers*, pages 72 – 80, April-June 1997.
- [5] S. Kim, K. Kum, and W. Sung. Fixed-point optimization utility for c and c++ based digital signal processing programs. In *Workshop on VLSI Signal Processing '95*, pages 197–206. Osaka, November 1995.
- [6] R. Lauwereins, M. Engels, M. Ade, and J.A. Perperstraete. Grape-2: A tool for the rapid prototyping of multi-rate asynchronous dsp applications on heterogeneous multiprocessors. *IEEE Computer*, 28(2):35 – 43, February 1995.
- [7] E. A. Lee and D. G. Messerschmidt. Static scheduling of synchronous data flow programs for digital signal processing. *Transactions on Computers*, C-36(1):24 –35, January 1987.
- [8] P. Schaumont, S. Vernalde, M. Engels, and I. Bolsens. Synthesis of multi-rate and variable rate digital circuits for high throughput telecom applications. In *Proc. EDTC 1997*.
- [9] Synopsys Inc., 700 E. Middlefield Rd, Mountain View, CA 94043. *BC User's Manual*.
- [10] Synopsys Inc., 700 E. Middlefield Rd, Mountain View, CA 94043. *DC User's Manual*.
- [11] M. Willems, V. Bursgens, H. Keding, T. Grotker, and H. Meyr. System level fixed-point design based on an interpolative approach. In *Proc. 34th Design Automation Conference*. Anaheim, CA, 1997.
- [12] J-S Yim, Y-H Hwang, C-J Park, H Choi, W-S Yang, H-S Oh, I-C Park, and C-M Kyung. A c-based rtl design verification methodology for complex microprocessor. In *Proc. 34th Design Automation Conference*. Anaheim, CA, 1997.
- [13] V. Zivojnovic, S. Pees, and H. Meyr. Lisa - machine description language and generic machine model for hw/sw co-design. In W. Burleson, K. Konstantinides, and T. Meng, editors, *VLSI Signal Processing IX*, 1996.