Arx: A Toolset for the Efficient Simulation and Direct Synthesis of High-Performance Signal Processing Algorithms

Klaas L. Hofstra and Sabih H. Gerez

Department of Electrical Engineering, Signals and Systems Group
University of Twente, The Netherlands
k.l.hofstra@utwente.nl

Abstract. This paper addresses the efficient implementation of high-performance signal-processing algorithms. In early stages of such designs many computation-intensive simulations may be necessary. This calls for hardware description formalisms targeted for efficient simulation (such as the programming language C). In current practice, other formalisms (such as VHDL) will often be used to map the design on hardware by means of logic synthesis. A manual, error-prone, translation of a description is then necessary.
The line of thought of this paper is that the gap between simulation and synthesis should not be bridged by stretching the use of existing formalisms (e.g. defining a synthesizable subset of C), but by a language dedicated to an application domain. This resulted in Arx, which is meant for signal-processing hardware at the register-transfer level, either using floating-point or fixed-point data. Code generators with knowledge of the application domain then generate efficient simulation models and synthesizable VHDL.
Several designers have already completed complex signal-processing designs using Arx in a short time, proving in practice that Arx is easy to learn. Benchmarks presented in this paper show that the generated simulation code is significantly faster than SystemC.

1 Introduction

In the last four decades many hardware description languages (HDLs) have been proposed, each having their strengths and weaknesses. HDLs serve one or more of the following goals: specification, formal verification, simulation, and synthesis. Ideally, one would use one and the same description language for all goals. In practice, however, the different goals are difficult to satisfy: code written in synthesizable VHDL (see e.g. [1]) will be much slower to simulate than code written in C, while code written in e.g. *SystemC* [2] in a style to optimize simulation speed is not likely to be synthesizable.

In practice, multiple HDLs are used to overcome this problem. C is often used to create a first executable model of the system to be designed. Such a model is *untimed* [3] in general. It has the advantage of fast execution and allows

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

extensive elaboration of the system's performance. Once the model has been refined to the point that a hardware architecture can be specified, the design is manually recoded in a synthesizable HDL, such as VHDL. This is a cumbersome and error-prone process.

This paper addresses the problem of conflicting language requirements and proposes a solution for the domain of register-transfer level (RTL) descriptions of signal processing algorithms. The solution consists of a specification language called Arx in combination with tools. These tools can convert designs written in Arx to either C code for simulation or VHDL code for synthesis. The generated C code is optimized for simulation speed while the VHDL code is optimized for synthesis. The proposed approach has the advantage that a thorough design-space exploration can be performed due to high simulation speed, while it is not necessary to rewrite the simulation code by hand in order to synthesize hardware.

This paper is organized as follows. Section 2 describes our requirements for a language for simulation and synthesis and reviews some existing languages. In Section 3 the Arx tools and language are introduced. Section 4 describes the C and VHDL code generators. The results of the simulation speed benchmarks are discussed in Section 5. In Section 6, some implementation results are discussed.

2 Languages for Simulation and Synthesis

In this section we present a list of requirements that we feel should be met for languages (and tools) for fast simulation and synthesis of signal processing algorithms. Subsequently, we review some existing design languages.

2.1 Requirements

The design goal of the Arx toolset is to simplify the hardware design for fixed-point signal processing algorithms. The toolset has been designed with the following requirements in mind:

1. High performance simulation must be possible because the signal processing algorithms typically require computation-intensive simulations.
2. The designs should be made at the register-transfer level. Synthesis from behavioral descriptions has the advantage of higher design productivity but results in implementations with an area overhead as well as suboptimal speed and power. This is due to the fact that the design is mapped on some predefined hardware template which limits the design freedom.
3. The entire language should be synthesizable, making it unnecessary to isolate the synthesizable language subset.
4. Fixed-point data types and fixed-point arithmetic operators must be supported because they are extensively used in signal processing algorithms.
5. The toolset must be text based, giving full design freedom to designers. Graphical toolsets limit the design freedom to the available building blocks and become cumbersome when designs are control dominated or have complex data paths.

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

6. In order to take advantage of the available highly optimized C compilers, the toolset should emit C code for the simulator instead of machine code.

2.2 Language Overview

VHDL was originally designed as a powerful simulation language [4] meant to have an event-driven simulation engine. Such a simulation approach offers the possibility to model hardware at various levels of abstraction but is less suitable for computation-intensive simulations of signal-processing systems. The overhead involved with run-time scheduling of events can cause a considerable increase in the simulation time. Synthesis from VHDL was introduced later. It required the definition of a *synthesizable subset* [1] as not all language constructs could be associated with hardware in a direct way. In synthesizable VHDL, one is practically forced to use the standardized `std_logic` data type and types derived from it. This 9-valued data type is very useful to trace design errors as it contains special values for uninitialized and undefined signals. This nice property, however, also contributes to slowing down simulations.

General-purpose programming languages have been combined with libraries or extensions for hardware simulation and synthesis. A prominent example of this approach is SystemC. SystemC is a C++ class library that provides functionality for system-level design. Abstraction levels range from RTL to the system-level. A synthesizable subset has been defined to enable hardware design in SystemC. The definition of such a subset is required because of the great scope of SystemC, i.e. constructs required for high-level modeling do not map to the RTL domain. The great scope and flexibility of SystemC are its strenghts. However, when the scope of a design is limited to the domain of RTL, we feel that a simple domain-specific language is better suited for the task. Such an approach has the advantage of having a clean language of which all features can be used at wish. The effort to learn the SystemC library and the synthesizable subset is as high as or higher than the effort to learn a small dedicated language [5].

Synchronous languages [6] combine deterministic concurrency with synchrony (i.e. time advances in lockstep with one or more clocks). Lustre [7] and Esterel [8] are examples of such languages that have been successfully used for hardware simulation and synthesis [9]. Arx is also a synchronous language but differs from the ones mentioned because it combines concurrent statements and sequential statements in a way similar to VHDL. In Arx, unlike the other synchronous languages, registers are explicitly instantiated. Because Arx had been specifically designed for RTL design, the notion of synchrony is linked to the concept of registers.

3 The Arx Toolset

3.1 The Arx Language

Arx enforces a synchronous design style. The clock is implicit in the design and is only apparent through the registers. In order to guarantee correct behavior,

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

systems written in Arx are not allowed to contain zero-delay loops. Arx has two types of data objects: *registers* and *variables*. When a register object is declared, its reset value must also be specified. Assignments to registers are concurrent while assignments to variables are sequential.

A design is implemented as a set of *components* and *functions*. Conceptually, components operate concurrently. Only components can contain registers. Functions are purely combinational. Both can be parameterized with *generic* types and constants. Generic types enable an efficient refinement methodology by allowing reuse of the same code in, for example, floating point and various fixed-point implementations. Functions can also be declared 'external', which means that the function is implemented in the target language (currently C or VHDL). This provides the designer with extensive library support at the algorithmic verification level. When the design is further refined into a synthesizable system, the external functions are replaced with synthesizable native Arx functions.

Table 1. Arx data types.

type	description
bit	1 or 0
bitvector	vector of bits
boolean	*true* or *false*
integer	integer values
real	floating-point
signed	signed fixed-point
unsigned	unsigned fixed-point

The available data types include floating-point and fixed-point. Table 1 presents a list of all Arx data types. Additionally, users can define type aliases and enumerated types. For both `signed` and `unsigned` data types the number of bits is specified in the same way as the SystemC `sc_fixed` data type [10], i.e. `signed(wl,iwl)` and `unsigned(wl,iwl)`, where `wl` denotes the total word length and `iwl` denotes the number of integer bits. Arx supports all the *overflow* and *quantization* modes defined in SystemC [11].

Arx supports a large set of arithmetic operations. The conditional statements `if` and `case` and the loop statement `for` are also included in the language. Arrays can be constructed and individual bits or slices of vectors can be selected with special operators.

Figure 1 shows Arx code for an accumulator with generic types. The hardware block diagram is depicted in Fig. 2. This example also shows the difference between registers and variables. The value of r at line 19 refers to the current value of the register because it occurs on the right-hand-side of the assignment expression. When registers are referenced at the left-hand-side of an assignment, such as at line 18, they refer to the register value for the next clock cycle.

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

```
01: component acc
02:     T_io        : generic type
03:     T_sum       : generic type
04:     clear       : in bit
05:     data_in     : in T_io
06:     data_out    : out T_io
07:
08: variable
09:     sum         : T_sum
10: register
11:     r           : T_sum = 0
12: begin
13:     if clear == 1
14:         sum = data_in
15:     else
16:         sum = r + data_in
17:     end
18:     r = sum
19:     data_out = r
20: end
```

Fig. 1. Arx code for an accumulator example.

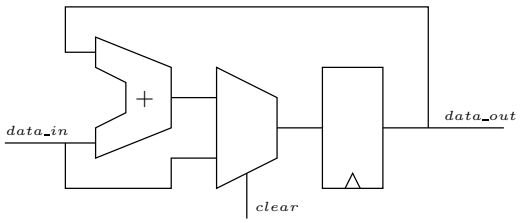

Fig. 2. Block diagram of the accumulator example.

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

3.2 The Workflow

The Arx language and toolset enable a stepwise refinement design methodology that starts with a high-level description and iteratively reaches an optimized synthesizable description. At each stage of the design, the tools can generate C code for a simulator. This simulator can be used for high-speed verification and evaluation of the design and algorithms.

Figure 3 shows the typical Arx workflow. The high-level system description is used for algorithmic verification. At this level all data types, including floating point, are allowed. The generated simulator is neither bit-true nor clock-cycle-true. Subsequently, the design is further refined into a bit-true description that restricts the data types to fixed-point. At this step, the simulator is bit-true but not clock-cycle-true. The next refinement step transforms the design into RTL code. RTL in the context of Arx amounts to distinguishing between *registers* and *wires (or variables)* and assuming the presence of an implicit clock that controls the register updates. The refinement of the design leads to a careful placement of registers in the signal flow that eventually results in a clock-cycle true specification. Given this RTL description, the tools can generate both a fast bit-true and clock-cycle-true simulator as well as VHDL code for synthesis.

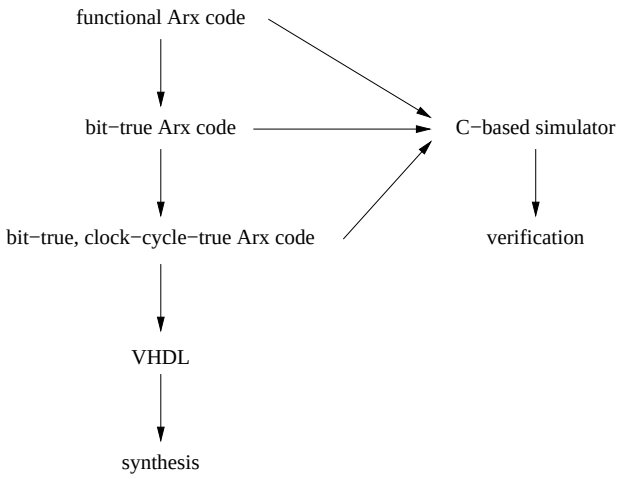

Fig. 3. Arx workflow.

4 Code Generation

4.1 Fixed-point Data Types for C Generation

For fast simulation of fixed-point arithmetic, the efficient mapping of fixed-point data-types on the host machine is crucial. SystemC offers two different fixed-

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

point class implementations. The limited-precision implementation maps the fixed-point data-types on the 53 mantissa bits of the native C++ floating point type. This restricts the size of fixed-point to 53 bits. Another disadvantage of this approach is that fixed-point types with a small number of bits, are mapped on 64-bit floating-point numbers which require extra memory bandwidth. The other option offered by SystemC is the unlimited fixed-point implementation that is based on concatenated data containers.

We chose to implement the fixed-point mapping in a similar way as described in [12]. All fixed-point values are mapped on the native machine word-size, which is 32-bit or 64-bit for most general purpose processors. If a fixed-point type exceeds this word-size, the type is mapped on a number of concatenated words.

SystemC uses operator overloading to implement arithmetic operations on fixed-point data types. The Arx code generation tools generate speed optimized code for each individual operation. We do not use global optimization to reduce the number of shift operations as is proposed in [12] because the authors indicate that the gains are minimal.

4.2 Scheduling for the C Backend

A hardware design is conveniently modeled by a set of parallel communicating processes. Conceptually, an Arx component corresponds to such a process. For simulation, the Arx tools convert this concurrent process model to a sequential program by appropriately scheduling the processes. Process scheduling can be done dynamically by a run-time scheduler or statically by a compiler. Because of the synchronous nature of the Arx language, the C code generator tools can use static scheduling. Contrary to static scheduling, dynamic scheduling as used in SystemC incurs a run-time overhead. FastSysC [13] is a replacement for the SystemC simulation engine that uses acyclic scheduling instead of dynamic scheduling. This scheduling approach reduces the run-time overhead of the scheduler. FastSysC is designed for cycle-level simulators and only supports a subset of the SystemC syntax. We analyse the impact on the simulation performance of these three scheduling methods in Sect. 5.

The first step in the Arx C generator is to instantiate all components. Subsequently, the component hierarchy is flattened (i.e. component boundaries are removed). Next, constant values are propagated and loops are unrolled. After that, the scheduling phase begins. For each clock cycle the Arx simulator needs to compute the new value of each register and all top-level outputs, based on the current register values and all top-level inputs. Hence, the current value of all registers and all top-level inputs are at the start of the data-flow graph, and the next value for all registers and all top-level outputs are end points of the graph. Because the Arx language is synchronous and zero-delay cycles are not permitted, the graph will not have cycles. Therefore, the scheduling is straightforward. Currently, Arx only supports a single clock but static scheduling of synchronous programs with multiple clocks is possible (see e.g. [14]).

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

4.3 Generated C Code

The C code generator creates code for a C++ class with two member functions: `run` and `reset`. The `run` function has the same inputs and outputs as the top-level component of the design. This function should be called for every clock-cycle of the simulation. The `reset` function assigns to each register its reset value. A SystemC wrapper is automatically created to enable easy integration with the SystemC Verification Library.

There are a number of differences between SystemC code and the C code generated by Arx that have an impact on simulation performance. Contrary to Arx C code, SystemC code uses virtual methods which result in a run-time overhead. Another difference is the number of function calls. Because all the Arx simulation code is located in the `run` function, only a single function call is executed for the simulation of a single cycle. For the SystemC simulator the number of function calls is a lot higher and depends on the number of components. Because the Arx simulation code is not spread out across multiple functions in different files, the C compiler can use more aggressive optimizations.

4.4 VHDL Code Generation

The VHDL code generator maintains the component hierarchy of the original design. Fixed-point data types are mapped on the `signed` and `unsigned` data types defined in the `ieee.numeric_std` package. An optimized VHDL package has been developed for the implementation of the supported overflow and quantization modes. The testbench created for the verification of the C code can be reused for verification of the VHDL provided that the VHDL simulator has an interface for C/SystemC co-simulation.

5 Benchmark Results

Two FIR filter implementations have been implemented to compare the simulation performance of Arx, SystemC and FastSysC. The first design (FIR1) implements an unfolded version of a 16-tap FIR filter. This implementation is modeled as a single component and processes one sample per clock cycle. The code for the Arx and SystemC versions of the filter are shown in Fig. 4 and Fig. 5 respectively (notice the compactness of the Arx code). The FastSysC version of the code differs only slightly from the SystemC version and is therefore not shown.

The second design (FIR2) is also a fully unfolded 16-tap FIR filter, However, in this design every adder, multiplier and register is modeled as a separate component. Therefore, instead of one filter component, this version has 47 components (16 multipliers, 15 adders and 16 registers) that all need to be scheduled. Both FIR1 and FIR2 implement exactly the same filter. Hence, any difference in simulation performance can be attributed to the scheduler.

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

```
component fir
    T_io        : generic type
    T_sum       : generic type
    T_coeff     : generic type
    data_in     : in T_io
    data_out    : out T_io
constant
    N           : integer = 16
    coeff       : array[N] of T_coeff = { ... }
register
    delay       : array[N] of T_io = 0.0
variable
    prod        : T_io
begin
    delay[0] = data_in * coeff[N-1]
    for i in 1:N-1
        prod = data_in * coeff[N-1-i]
        delay[i] = convert(T_sum, prod + delay[i-1])
    end
    data_out = delay[N-1]
end
```

Fig. 4. Arx code for the unfolded FIR filter.

The FIR designs have been benchmarked with both fixed-point and floating-point data types. The floating-point version uses `float` for SystemC and FastSysC and `real` for Arx. All operations are floating-point and there are no type conversions. In case of the fixed-point version, all data types are signed 10-bit fixed-point. The result of the addition operation is saturated in order to stay within the 10-bit range. There is no FastSysC fixed-point version because we are only interested in the performance of the scheduler of FastSysC.

In order to benchmark the performance of the SystemC fixed-point data types separately from the SystemC scheduler, a special version of the fixed-point FIR1 benchmark was made that does not use the SystemC scheduler. Instead of using signals for input and output, this implementation uses function parameters.

The SystemC implementations use the synthesizable subset of SystemC, i.e. they use `SC_METHOD` and signals for communication between logic blocks. In order to reduce the simulation overhead, internal signals are declared as normal local variables and not as SystemC signals. Version 2.1 of the SystemC library and version 1.1 of the FastSysC simulator were used for the benchmarks.

The simulation times for the benchmarks are summarized in Table 2. The results have been scaled relative to the fastest simulation. The results show that the simulation times for the Arx versions of FIR1 and FIR2 are identical in the floating-point case and very close for the fixed-point data types. This shows that there is no extra scheduling overhead for the Arx simulator when the number of components is increased from 1 (FIR1) to 47 (FIR2).

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

```
SC_MODULE(fir) {
    sc_in<bool> clock;
    sc_in<bool> reset;
    sc_in<T_io> data_in;
    sc_out<T_io> data_out;

    T_coeff coeff[NR_TAPS];
    T_sum delay_reg[NR_TAPS];
    T_sum delay_nxt[NR_TAPS];

    SC_CTOR(fir)
    {
        SC_METHOD(update);
        sensitive_pos << clock;
        coeff[0] = ...
        ...
    };

    void update()
    {
        int i;
        T_io input, output, prod;

        if (reset.read()) {
            for (i=0; i<NR_TAPS; i++) {
                delay_reg[i] = 0;
            }
        }
        else {
            input = data_in.read();

            /* compute next values */
            delay_nxt[0] = (T_io)(input * coeff[NR_TAPS-1]);
            for (i=1; i<NR_TAPS; i++) {
                prod = (T_io)(input * coeff[NR_TAPS-1-i]);
                delay_nxt[i] = (T_sum)(prod + delay_reg[i-1]);
            }

            /* update registers */
            for (i=0; i<NR_TAPS; i++) {
                delay_reg[i] = delay_nxt[i];
            }
        }
        output = delay_reg[NR_TAPS-1];
        data_out.write(output);
    }
}
```

Fig. 5. SystemC code for the unfolded FIR filter.

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

Table 2. Simulation time results.

simulator	data type	FIR1	FIR2
Arx	floating-point	1.00	1.00
SystemC	floating-point	23.5	201
FastSysC	floating-point	2.63	45.3
Arx	fixed-point	2.19	2.46
SystemC	fixed-point	467	1263
SystemC (without scheduler)	fixed-point	385	-

In case of the floating-point SystemC benchmarks, the simulation time for FIR2 is about 8 times longer than the simulation time for FIR1. This shows that the overhead of the dynamic scheduler increases when the number of components increases, which is to be expected. For the FastSysC benchmarks the increase in simulation time between FIR1 and FIR2 is roughly a factor of 17. This relative increase is larger than that for the SystemC benchmarks, but both FIR1 and FIR2 floating-point FastSysC simulation times are respectively 9 and 4 times faster than their SystemC counterparts. Compared to the results for the floating-point Arx benchmarks, the FastSysC benchmarks are 2.6 and 45.3 times slower while the SystemC benchmarks are 23.5 and 201 times slower. We can conclude that the FastSysC simulator is faster than the SystemC implementation but slower than the Arx simulator. The Arx simulator is one to two orders of magnitude faster than the SystemC versions, depending on the number of components in the simulation.

In order to compare the fixed-point data type implementations of SystemC and Arx, we compare the Arx fixed-point FIR1 benchmark with the SystemC FIR1 benchmark without scheduler. The simulation time results show that the Arx version is two orders of magnitude faster than the SystemC implementation without scheduler.

6 Implementation Results

Arx has successfully been applied to a number of designs, including an LDPC decoder, a CDMA equalizer [15] and a MIMO MMSE equalizer. The goal of efficient simulation has been met for these designs. The VHDL generated by Arx was successfully mapped on an experimental fast prototyping setup with an FPGA PCI board and also synthesized using modern ASIC standard-cell libraries. The simulation efficiency of Arx has made it possible to explore larger parts of the design space than usual.

7 Conclusions

In this paper we have briefly introduced the design language Arx and our workflow for simulation and synthesis. Arx has been created to enable synthesis and

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

fast simulation of signal processing algorithms based on a single design description. For synthesis this description is transformed into VHDL, while C code is generated for fast simulation.

In order to benchmark the simulation speed of the generated C code, two FIR implementations have been realized using Arx, SystemC and FastSysC. The benchmark results show that the Arx simulator is faster than both the SystemC and FastSysC simulators. When we compare the fixed-point synthesizable implementations written in Arx and SystemC, the Arx simulators are two orders of magnitude faster than the SystemC versions. This speed advantage is achieved through a faster implementation of the fixed-point arithmetic and the use of static scheduling.

References

1. Rushton, A.: VHDL for Logic Synthesis, Second Editon. John Wiley and Sons (1998)
2. Groetker, T., S. Liao, G.M., Swan, S.: System Design with SystemC. Kluwer Academic Publishers (2002)
3. Jantsch, A.: Modeling Embedded Systems and SoCs, Concurrency and Time in Models of Computation. Morgan Kaufmann, San Francisco (2004)
4. Lipsett, R., Schaeffer, C., Ussery, C.: VHDL: Hardware Description and Design. Kluwer Academic Publishers, Boston (1989)
5. Edwards, S.A.: The challenges of hardware synthesis from C-like languages. In: Proceedings of the International Workshop on Logic and Synthesis (IWLS). (2004)
6. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. In: Proceedings of the IEEE. Volume 91. (2003) 64–83
7. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.A.: Lustre: a declarative language for real-time programming. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, New York, NY, USA, ACM Press (1987) 178–188
8. Berry, G.: The Foundations of Esterel. MIT Press (2000)
9. Rocheteau, F., Halbwachs, N.: Pollux, a Lustre-based hardware design environment. In Quinton, P., Robert, Y., eds.: Conference on Algorithms and Parallel VLSI Architectures II, Chateau de Bonas (1991)
10. Black, D., Donovan, J.: SystemC: From the Ground Up. Kluwer Academic Publishers, Boston (2004)
11. OSCI: SystemC version 2.0 user's guide, update for SystemC 2.0.1. `http://www.systemc.org` (2002)
12. Keding, H., Coors, M., Lüthje, O., Meyr, H.: Fast bit-true simulation. In: DAC '01: Proceedings of the 38th conference on design automation, New York, NY, USA, ACM Press (2001) 708–713
13. Garcia Perez, D., Mouchard, G., Temam, O.: A new optimized implementation of the SystemC engine using acyclic scheduling. In: Design Automation and Test in Europe, DATE'04. (2004)
14. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Transactions on Computers **36**(1) (1987) 24–35

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.

15. van Kampen, D., Hofstra, K.L., Potman, J., Gerez, S.H.: Implementation of a combined OFDM-demodulation and WCDMA-equalization module. In: Proceedings of the Workshop on Circuits, Systems and Signal Processing, ProRISC. (2006)

International Conference on High Performance Embedded Architectures and Compilers, Ghent, Belgium, January 2007.
Preprint version. Official version available through SpringerLink.