

# A Language and Toolset for the Synthesis and Efficient Simulation of Clock-Cycle-True Signal-Processing Algorithms

Klaas L. Hofstra, Sabih H. Gerez\* and David van Kampen\*  
University of Twente, Department of Electrical Engineering,  
Signals and Systems Group,  
P.O. Box 217, 7500 AE Enschede, The Netherlands  
Phone: +31 53 489 2773 Fax: +31 53 489 1060  
E-mail: k.l.hofstra@utwente.nl

*Abstract*—Optimal simulation speed and synthesizability are contradictory requirements for a hardware description language. This paper presents a language and toolset that enables both synthesis and fast simulation of fixed-point signal processing algorithms at the register-transfer level using a single system description. This is achieved by separate code generators for different purposes. Code-generators have been developed for fast simulation (using ANSI-C) and for synthesis (using VHDL). The simulation performance of the proposed approach has been compared with other known methods and turns out to be comparable in speed to the fastest among them.

*Keywords*—hardware description languages, simulation, synthesis

## I. 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 extensive elaboration of the system's performance. Once the model has been

\*The main affiliation of Sabih Gerez is with SiTel Semiconductor, Design Center Hengelo, The Netherlands. David van Kampen's contributions to this work were part of his internship at SiTel Semiconductor.

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. This solution consists of a single specification language called *Arx*, in combination with tools that process designs written in this language. The tools build an internal representation and map this internal representation to external formats in an optimal way. The C-generation tool expands the compact *Arx* format into C code that executes efficiently: fixed-point data types are e.g. mapped on the `int` data type where possible, *shadow* variables are introduced to distinguish the input and output of a register, etc. In short, all kind of detail that a user would add manually to a simulation model, is automatically generated. In a similar way, the VHDL-generation tool incorporates expert knowledge about synthesis when converting *Arx* into synthesizable VHDL.

The proposed approach has the advantage that a thorough design-space exploration can be performed due to the optimized simulation speed, while it is not necessary to rewrite the simulation code by hand in order to synthesize hardware.

RTL in the context of *Arx* should be understood in a wide sense. It distinguishes between *registers* and *wires (or variables)* and assumes the presence of an implicit clock that controls the register updates (support for such a distinction could already be found in early HDLs such as MoDL [4]). First versions of a design can be written with a minimal number of registers and need not be clock-cycle true. The refinement of the design leads to a careful placement of registers in the signal flow that eventually results in a clock-cycle

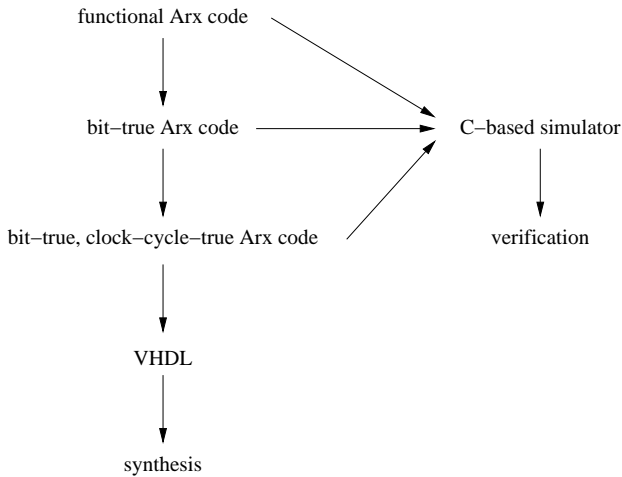


Fig. 1. *Arx* workflow.

true specification.

This paper is organized as follows: in Section II the tools and language are introduced. The next section describes the problem of modeling RTL in C. Section IV provides more details about the implementation of the tools. Benchmark results are presented in Section V. Section VI concludes the paper.

## II. THE *Arx* TOOLSET

The language and toolset enable a successive refinement methodology, i.e. a design can be first described in a high-level description and iteratively refined into a synthesizable description. At each level 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 1 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 non floating-point. Once the conversion to fixed-point is completed, the generated simulator is bit-true but not clock-cycle-true. The next refinement step transforms the design into RTL code. Given this RTL description, the tools can generate a fast bit-true and clock-cycle-true simulator and VHDL code for synthesis.

### A. The *Arx* Language

The language *Arx* is still in development. It combines elements from C and VHDL. *Arx* supports all arithmetic operations of VHDL. The conditional statements `if` and `case` and the looping statement

TABLE I  
*Arx* DATA TYPES.

type	description
boolean	<i>true</i> or <i>false</i>
integer	integer values
real	floating-point numbers
signed	signed fixed-point
unsigned	unsigned fixed-point

`for` are also included in the language. Arrays can be constructed and individual bits or slices can be selected with special operators. The design goal of the language is to simplify the hardware design for signal processing algorithms. All state transitions in the design must be synchronous, which makes it impossible to construct latches. This limitation allows a simple data-flow style design description and enables fast simulations. Globally asynchronous locally synchronous (GALS) designs can still be created by embedding the design in a C-based framework such as SystemC.

*Arx* has two types of data object: *registers* and *variables*. Assignments to registers are concurrent while assignments to variables are sequential. When a register object is declared, its reset value can be specified, otherwise it defaults to zero. While the simulator only supports synchronous resets, the VHDL generator supports both synchronous and asynchronous resets.

The available data types are listed in table I. Additionally, users can define their own 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 [5]: `signed(wl, iw1)` and `unsigned(wl, iw1)`, where `wl` denotes the total word length and `iw1` denotes the number of integer bits. The code below shows how to declare a register and a variable with signed fixed-point data type (2 integer bits and 4 fractional bits). The declared register has a non-zero reset value.

```

register signed(6,2) my_reg(reset=1.75);
variable signed(6,2) my_var;
  
```

*Arx* supports a subset of the many *overflow* and *quantization* modes of SystemC [6]. The reason to limit the number of supported modes is a practical one. Implementing new modes is almost trivial. For unsigned data types  $MIN = 0$  and  $MAX = (2^{wl} - 1)2^{iw1-wl}$ . For signed data types  $MIN = -2^{iw1-1}$  and  $MAX = (2^{wl-1} - 1)2^{iw1-wl}$ . The supported overflow modes, with their SystemC equivalent between parenthesis, are:

- **wrap** (SC\_WRAP) Wrap around, redundant MSB bits are discarded. This is the default overflow mode.
- **sat** (SC\_SAT) Saturate, positive overflow yields *MAX* and negative overflow yields *MIN*.
- **satsym** (SC\_SAT\_SYM) Saturate symmetrical, positive overflow yields *MAX* and negative overflow yields  $-MAX$ .

The supported quantization modes are:

- **trunc** (SC\_TRN) Remove redundant bits. This is the default quantization mode.
- **rnd** (SC\_RND) Rounds by adding the MSB of the removed bits to the remaining bits.

*Arx* is a strongly typed language, i.e. values of one data type can't be assigned to values of another type without conversions. An assignment does not implicitly convert the type of the right-hand side to the type of the left-hand side because this hides possible loss of precision. Hence, the following code will produce a type error because the type of the right hand side is `signed(7,3)` (the addition increases the number of integer bits) while the type of the left hand side is `signed(6,2)`:

```
variable signed(6,2) a, b, c;
a = b + c;
```

In these situations, casts must be used to convert the type of an expression. The code below shows how to cast the result of the addition to the correct type with the overflow mode set to saturation:

```
a = cast signed(6,2,sat,rnd) ( b + c );
```

If one of the operands of a two-operand operation is a constant, the constant is automatically converted to the type of the other operand. When this leads to loss of precision, a warning is generated.

Instead of converting a bit-sequence to another bit-sequence as is done with casts, it is sometimes necessary to just interpret a bit-sequence differently. The following code shows how to reinterpret a signed fixed-point value as an integer:

```
variable signed(8,1) a;
variable integer b;
a = 0.5;
b = reinterpret integer ( a );
```

The bit pattern corresponding to the value of 0.5 ("01000000") is reinterpreted as the integer value 64.

A design is partitioned in modules. Each design has one top-level module called *top*, that contains all other modules of the design. Modules can be parameterized with generics. Generics can be types and constant values. This makes it easy to make, for example, a single FIR implementation with parameterized number of taps and data types. An example of a second

---

**Listing 1** *Arx* code for a second order IIR filter with generic types.

---

```
// module declaration
module iir
<
  // parameters
  type T_ACC,
  type T_DATA,
  type T_COEFF
>
(
  // interface
  data_in in T_DATA,
  data_out out T_DATA
)

// declare registers and variables
variable T_COEFF a1, a2, b0, b1, b2;
register T_ACC z1, z2;
register T_DATA z3;
variable T_DATA y;
variable T_ACC a1p, a2p, b0p, b1p, b2p;

a1 = 0.1;
a2 = 0.2;
b0 = 0.3;
b1 = 0.4;
b2 = 0.5;

b0p = cast T_ACC( b0 * data_in );
b1p = cast T_ACC( b1 * data_in );
b2p = cast T_ACC( b2 * data_in );

y = cast T_DATA( z2 + b0p );
a1p = cast T_ACC( y * a1 );
a2p = cast T_ACC( y * a2 );

z1 = cast T_ACC( b2p + a2p );
z2 = cast T_ACC( cast T_ACC( b1p + a1p ) + z1 );
z3 = y;
data_out = z3;
end

// top-level module
module top
<
  // parameter with default value
  type T_DATA = signed(16,1,sat,rnd)
>
(
  // top-level interface
  in0 in T_DATA,
  out0 out T_DATA
)

// module instantiation
iir iir1
<
  // pass parameters
  T_ACC = signed(24,4,sat,rnd),
  T_DATA = T_DATA,
  T_COEFF = signed(10,1,sat,rnd)
>
(
  // connecting interface
  data_in = in0,
  data_out = out0
);
end
```

---

order IIR design is shown in Listing 1. This is just an example and not a real design.

### III. C MODELING OF RTL

#### A. Fixed-Point Data Types

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

---

**Listing 2** A|RT-C code for a “toy” example.

---

```
void reg_incr(int d_in, int& d_out)
{
#pragma OUT d_out
  static int local_reg = 0;
  int local_nxt;

  local_nxt = d_in;
  d_out     = local_reg + 1;
  local_reg = local_nxt;
}
```

---

advantage 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 [7]. 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. Because our tool generates code, it can generate optimized code for each individual operation in the *Arx* code. This method does not incur the run-time overhead of operator overloading. Currently, we do not use global optimization to reduce the number of shift operations as is proposed in [7].

### B. Register Modeling

There exist many ways to model the registers of an RTL design in C. If a hardware module corresponds to a C function, the use of `static` variables within the function effectuate that their value is preserved from one function call to the next. This style is used by the commercial *A|RT Builder* tool that has reached an end-of-life status [8]. Listing 2 shows a toy design that consists of a register followed by an incremter written in the A|RT style. The intended hardware block diagram is depicted in Figure 2.

If SystemC is used to model hardware at the register-transfer level, one has many choices for the coding style. One could e.g. use a *static* or *synchronous* data-flow style [9] where modules interact through FIFOs (first-in first-out buffers) [2] and each invocation of the model (read inputs, execute process, write outputs) corresponds to a single register update. One could also opt for a synthesizable style that is closer to the hardware and declares the clock and reset signals explicitly. A possible implementation of

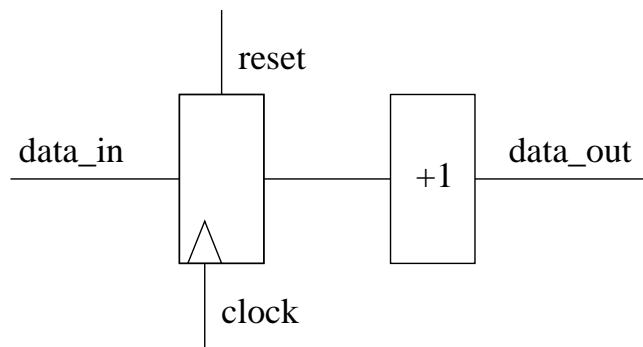


Fig. 2. Block diagram of the “toy” example.

---

**Listing 3** SystemC code for the “toy” example.

---

```
SC_MODULE(toy)
{
  //Declarations of input/output ports
  sc_in<bool> CLOCK;
  sc_in<bool> RESET;
  sc_in<int> d_in;
  sc_out<int> d_out;

  //Declaration of a register
  sc_signal<int> local_reg;

  SC_CTOR(toy)
  {
    SC_METHOD(reg_update);
    sensitive << CLOCK.pos();
    sensitive << RESET.pos();

    SC_METHOD(increment);
    sensitive << local_reg;
  }

  void reg_update();
  void increment();
};

void toy::reg_update()
{
  if(RESET.read())
    local_reg = 0;
  else
  {
    local_reg = d_in.read();
  }
}

void toy::increment()
{
  d_out.write(local_reg+1);
}
```

---

the “toy” example coded in this style is given in Listing 3.

A definite advantage of SystemC is that it provides a simulation engine and the modeling semantics necessary for flexible system-level design. From a performance point of view, however, one pays a price for the simulator overhead. Data on this overhead can be found in Section V. Because of this overhead the *Arx* code generation uses a style more like A|RT.

In the C code generated by the *Arx* tools, registers are modeled as global variables. A separate function is generated for the handling of a reset. This function sets all global register variables to their respective reset values. At the start of the simulation function a copy is made of the register values. We call these

---

**Listing 4** *Arx* generated C code for the “toy” example.

---

```
int reg;

void reset(void)
{
    reg = 0;
}

void sim(const int d_in, int& d_out)
{
    int shadow_reg = reg;
    reg = d_in;
    d_out = shadow_reg + 1;
}
```

---

copies shadow values. When a value is read from a register in the original *Arx* code, code for a read from the shadow value is generated. A write to a register is translated to a write to the actual register value. Listing 4 shows the code for the example in the style of the *Arx* code generator.

#### IV. TOOL IMPLEMENTATION

A C-code generating backend targeting fast simulation has been fully implemented and a VHDL generating backend for synthesis is in development.

The lexer and parser for *Arx* have been implemented with the parser generator ANTLR (ANother Tool for Language Recognition) [10]. Given a grammar description, ANTLR is able to generate code for lexers, parsers and tree-parsers in Java, C#, C++ or Python. We chose Python[11] because it is well suited for rapid development.

The parser generated by ANTLR creates an *abstract syntax tree* (AST) which can then be parsed by tree-parsers creating new ASTs. Actions can be embedded in the grammars, i.e. specific code is executed when a rule matches. These actions can modify the returned AST. The translation from *Arx* to C and VHDL is implemented by several stages of tree transformations. The tree-parsers for the final stages emit the target code.

The C code generator flattens the module hierarchy. This results in a single C function that needs to be called once per clock-cycle. Additional functions are generated for the allocation and freeing of memory for persistent storage. These functions should be called at the beginning and end of the simulation respectively. For the handling of the reset, a separate function is generated as described in section III-B.

#### V. SIMULATIONS

As a proof of concept, two typical signal processing kernels have been implemented:

- **FIR filter** A 64 taps transpose form FIR filter.
- **IIR filter** A second order IIR filter.

Implementations of these kernels written in *Arx*, SystemC and A|RT style C have been benchmarked in order to quantify the effects of the different fixed-point implementations and register modeling approaches. Version 2.1 of SystemC was used. The implementations are listed below:

- **Floating-point** A floating-point reference implementation of the algorithms using the A|RT-C style of register modeling.
- **A|RT C sc.fixed** A|RT C implementation that used the SystemC `sc_fixed` data type for fixed-point arithmetic.
- **A|RT C integer** An implementation written in A|RT style C in which the `sc_fixed` data types have been replaced by the `int` data type by an in-house conversion tool.
- **SystemC sc.fixed** This implementation is written in SystemC and uses the `sc_fixed` fixed-point data type.
- **SystemC integer** This is a converted version of the previous benchmark using the `int` data type.
- **Arx** Implementation of the algorithm written in *Arx*.

The conversion of the `sc_fixed` benchmarks into `int` equivalents were performed by an adapted version of the `fixed2int` tool [12]. The original version replaced `sc_fixed` by the `sc_int` in order to replace synthesizable code. This version aims at simulation speed-up.

All of the implementations, are bit-true and clock-cycle-true except for the floating-point one that is only clock-cycle true. The SystemC models have been coded using a variant of the style presented in Listing 3. In order to reduce the simulation overhead, only the module ports are declared as signals (`sc_in`, `sc_out`). The internal signals are not declared as `sc_signal< <type>>` but directly as `<type>`.

The fixed-point FIR and IIR implementations have been tested with two sets of overflow and quantization modes. The kernels FIR 1 and IIR 1 combine wrapping (`wrap`) and truncation (`trunc`). Saturation (`sat`) and rounding (`rnd`) is used for kernels FIR 2 and IIR 2. The simulation times for the benchmarks are summarized in Table II. The results have been scaled relative to the simulation time of the floating-point code.

The FIR benchmark is more computationally intensive compared to the IIR (64 taps vs. 2 internal delays). The effects of replacing the fixed-point data types by integer equivalent code is therefore clearer for the FIR. A general conclusion is that directly sim-

TABLE II  
SIMULATION SPEED RESULTS

kernel	floating-point	SystemC sc_fixed	SystemC integer	A RT sc_fixed	A RT integer	<i>Arx</i>
FIR 1	1.00	73.18	1.87	122.48	0.83	0.82
FIR 2	1.00	72.53	2.75	127.83	1.61	1.38
IIR 1	1.00	25.30	4.37	41.18	0.44	0.49
IIR 2	1.00	25.52	4.29	41.66	0.56	0.51

TABLE III  
SIMULATION-TIME DISTRIBUTION (IN %) FOR THE IIR 2  
BENCHMARK

kernel	SystemC sc_fixed	SystemC integer	A RT sc_fixed
SC fixed	76.13	0.00	90.11
SC sim.	16.10	89.30	0.00
Rest	7.77	10.70	9.90
total	100.00	100.00	100.00

ulating with the fixed-point data types leads to high overhead (as high as 2 orders of magnitude). These results are similar to those reported in [7]. As expected, the “direct” C implementations in the A|RT-C style or the *Arx* style are equally efficient.

The table also shows that the SystemC simulator has a high overhead as well. This is more visible in the IIR benchmark as its I/O signals, which trigger the simulator events, change relatively more often than in the FIR benchmark.

The conclusions have been confirmed by *profiling* the benchmarks. The profiling results for the IIR 2 benchmark are shown in Table III. The table shows the percentage of the CPU time spent on fixed-point calculations, SystemC simulator functions and all other computations.

## VI. CONCLUSIONS

In this paper we have briefly introduced the design language *Arx* and our tools and workflow for simulation and synthesis. A single design description written in *Arx* is processed by the tools and code is generated depending on the selected backend. The C generating backend optimized for simulation speed has been presented in detail. As a proof of concept we have implemented two typical signal processing algorithms, and compared the simulation performance with implementations written in SystemC and A|RT style C.

The benchmarks show that the right choices have been made for the simulator generated by the *Arx* tools, as far as simulation speed is concerned. The next development step for *Arx* is the completion of the VHDL backend designed for synthesis.

## ACKNOWLEDGMENT

Sabih Gerez would like to thank Søren Rievers of SiTel Semiconductor for his contributions to the discussions on C-based hardware design.

## REFERENCES

- [1] A. Rushton. *VHDL for Logic Synthesis, Second Edition*. John Wiley and Sons, 1998.
- [2] T. Groetker, G. Martin S. Liao, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.
- [3] A. Jantsch. *Modeling Embedded Systems and SoCs, Concurrency and Time in Models of Computation*. Morgan Kaufmann, San Francisco, 2004.
- [4] J. Smit, B.J.F. van Beijnum, S.H. Gerez, R.J. Mulder, and L. Spaanenburg. The MoDL hardware design system. In M. Barbacci and C.J. Koomey, editors, *CHDL 87, Computer Hardware Description Languages and Their Applications*, Amsterdam, 1987.
- [5] D. Black and J. Donovan. *SystemC: From the Ground Up*. Kluwer Academic Publishers, Boston, 2004.
- [6] SystemC Version 2.0 User’s Guide, Update for SystemC 2.0.1. <http://www.systemc.org>, 2002.
- [7] Holger Keding, Martin Coors, Olaf Lüthje, and Heinrich Meyr. Fast bit-true simulation. In *DAC ’01: Proceedings of the 38th conference on Design automation*, pages 708–713, New York, NY, USA, 2001. ACM Press.
- [8] ARM. <http://www.arm.com/>.
- [9] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.
- [10] ANTLR. <http://www.antlr.org>.
- [11] Python. <http://www.python.org>.
- [12] S.H. Gerez and J. de Zoeten. A conversion tool for the synthesis of SystemC fixed-point data types by the CoCentric SystemC Compiler. In *Synopsys User Group Conference, SNUG Europe*, Munich, Germany, May 2004.