# AUTOMATED CONVERSION OF SYSTEMC FIXED-POINT DATA TYPES

Axel G. Braun, Djones V. Lettnin, Joachim Gerlach, Wolfgang Rosenstiel

*University of Tuebingen,*
*Wilhelm-Schickard-Institute for Computer Science,*
*Department of Computer Engineering,*
*Sand 13*
*72076 Tuebingen, Germany*
{abraun|lettnin|gerlach|rosenstiel}@informatik.uni-tuebingen.de

**Abstract** This article describes a methodology for the automated conversion of SystemC fixed-point data types and arithmetics to an integer-based format for simulation acceleration and hardware synthesis. In most design flows the direct synthesis of fixed-point data types and their related arithmetics is not supported. Thus all fixed-point arithmetics have to be converted manually in a very time-consuming and error-prone procedure. Therefore a conversion methodology has been developed and a tool enabling an automated conversion of SystemC fixed-point data types as well as fixed-point arithmetics has been implemented. The article describes the theory and transformation rules of the conversion methodology, their implementation into a tool solution, and its application in terms of an experimental case study.

**Keywords:** Fixed-Point; Data Type Conversion; SystemC; System Design; Hardware Synthesis.

## 1. Introduction

The increasing complexity of today's and future electronic systems is one of the central problems that must be solved by modern design methodologies. A broad range of functionality which should be adaptable to market requirements in a very fast and flexible way, and thereby a decreasing time-to-market phase are only some of the most important issues. There are several promising approaches like system-level modelling, platform-based design, and IP re-use, to face these problems. The economic success of a product highly depends on the power and flexibility of the design flow and the design methodology.

A very popular approach for system-level modelling is SystemC [14, 15]. SystemC is a C++-based system-level specification language that covers a

*Please use the following format when citing this chapter:*
Braun, Axel, G., Lettnin, Djones, V., Gerlach, Joachim, Rosenstiel, Wolfgang, 2006, in IFIP International Federation for Information Processing, Volume 200, VLSI-SOC: From Systems to Chips, eds. Glesner, M., Reis, R., Indrusiak, L., Mooney, V., Eveking, H., (Boston: Springer), pp. 55-72.

broad range of abstraction levels. The language is already a de-facto standard for system-level design. Several commercial design and synthesis products from Synopsys, Forte, Coware, Summit and others will be a foundation for system-level design flows. An important property of a design flow is that it must be free of gaps for all paths in a design cycle.

Algorithm design is usually based on floating-point data types. In a following step word lengths and accuracies are evaluated and fixed-point data types are introduced (e.g. for signal processing applications, etc.). The introduction and the evaluation process of fixed-point data types are well-supported by tools like FRIDGE (Fixed-Point Programming and Design Environment) [3, 6] or Synopsys CoCentric Fixed-Point Designer [11]. SystemC itself offers a very powerful concept of fixed-point data types [8]. In nearly all cases these types are not directly synthesizable to hardware. In common hardware design flows an additional manual conversion has to be applied before the synthesis process. Fixed-point data types are transformed to an integer-based data format containing virtual decimal points (which correspond to binary points in the digital world). The manual translation is very time-consuming and error-prone. To close this gap, it is necessary to develop an appropriate methodology for the conversion of those data types and arithmetics to a synthesizable integer format. In the following sections we present a solution for this step, which is based on SystemC fixed-point data types.

## 2. Fixed-Point to Integer Conversion

The basis of our methodology is a set of conversion rules. These conversion rules have been developed by analyzing traditional transformations of fixed-point arithmetics and operations, manually done by a designer. We considered variable declarations, assignments, basic arithmetic operations like summation, subtraction, multiplication, and division, as well as comparisons. In the description of the conversion rules of our methodology below, we will use the following definitions. A fixed-point number in binary representation can be described as follows:

$$fix(n,m) := \underbrace{d_{n-1}d_{n-2}\ldots d_0}_{IntegerPart} . \underbrace{d_{-1}\ldots d_{-(m-1)}d_{-m}}_{FractionalPart}$$

$$fix(n,m) := \sum_{i=-m}^{n-1} d_i \cdot 2^i$$

The corresponding data type in SystemC notation is `sc_fixed<n+m,n>`. A fixed-point number can be described using an integer data type without loss of information, if the decimal point will be shifted $m$ positions to the right. All fractional bit positions are now located in the integer part. The resulting integer number has the same word length as the fixed-point representation before. In the following we will also use the definitions below:

- Least Common Data Type (LCDT): The LCDT of two fixed-point variables $a$ and $b$ is a fixed-point data type, which has minimal width that can contain all possible values of $a$ and $b$ without a loss of accuracy.

$$LCDT(fix(n,m), fix(o,p)) := fix(max(n,o), max(m,p))$$

- Result Data Type (RDT): The RDT is the fixed-point data type, which can contain the result of an operation without loss of accuracy.

Basically, only conversions of those operations where both operands are fixed-point data types will be made (basic arithmetic operations). Constants or other operands must be signed by an explicit type cast to a fixed-point data type. The first step is the calculation of the LCDT of a (binary) operation (see Figure 1). The integer and the fractional part equal to the maximum width of the two operand's integer and fractional parts. Secondly, the word lengths and type of the result (i.e. RDT) are determined. For the preservation of accuracy, both operands are adapted to the word length of the result type before the intrinsic operation takes place. The result will then be reduced to the LCDT. This reduction is important to avoid a steady extension of the word length throughout the further conversion procedure (e.g. for multiplications).

## Declarations, Assignments, and Comparisons

Variables or attribute declarations of a fixed-point data type (`sc_fixed` and `sc_ufixed`) are directly converted to their appropriate SystemC integer data type (`sc_int` and `sc_uint`) of the same overall word lengths. A major prerequisite for assignments and comparisons is that both operands are exactly of the same data type. To ensure this, both operands are converted to the LCDT and then the comparison takes place. All C++ comparison operators can be converted directly. In case of a simple assignment (not combined with other operators like +=), the conversion has to adapt the data type on the right side to the type of the left side. We assume that combined assignments (e.g. +=) are split up in a basic three-address-format (e.g. `a+=b` is expanded to `a=a+b`). Return values of functions or methods are treated in the same manner as assignments: the returned value will be adapted to the type of the function respectively method.

## Summation and Subtraction

Summation and subtraction operations are relatively easy to convert. Both operators are treated identically. The integer type of the result's data type

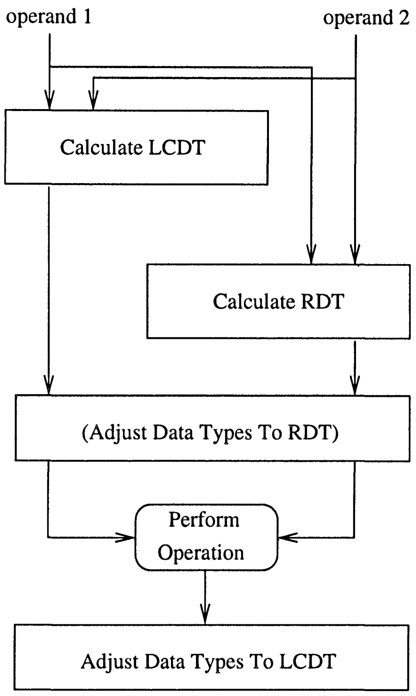


*Figure 1.* Conversion procedure

equals to the maximum of the integer parts of both operands. The part on the right side of the decimal point is converted in the same manner. In case of a summation or a subtraction an overflow is possible. Therefore the result type word length is enlarged by one bit:

$$fix(s,t) \pm fix(u,v) \rightarrow fix(max(s,u)+1, max(t,v))$$

For the conversion of a summation or subtraction of two fixed-point variables with different bit layouts, i.e. different widths of integer part and different widths on the right side of the decimal point, an adaptation has to be performed. The position of both variables has to be adjusted. If one of the operands has less bits on the right side, a shift to the left is necessary.

Figure 2 (left side) gives a short example for the summation of 2 numbers. Variable x has entirely 5 bits and contains 3 integer bits and 2 fractional bits, variable y has 6 bits and contains 1 fractional bit. The width of 7 bits is determined by the LCDT. In case of summation and subtraction here, the adaptation to the RDT is not needed and therefore kept. The conversion performs one left

```
//Fixed-point                //Integer base
sc_fixed<5,3> x;             sc_int<5> x;
sc_fixed<6,5> y;             sc_int<6> y;
...                          ...
x + y;                      (sc_int<7>)x + ((sc_int<7>y) << 1);
x - y;                      (sc_int<7>)x - ((sc_int<7>y) << 1);
...                          ...
```

*Figure 2.* Summation and subtraction example

shift of variable y to adjust the (virtual) decimal points (right side of Figure 2). Now the summation can be performed correctly.

## Multiplication and Division

During the conversion of a multiplication the word lengths of the integer parts of both operands are added. The same holds for the fractional part on the right sides of the decimal points. If there are differences between the word lengths of the fractional parts of the LCDT and RDT, the RDT must be adapted. As described above, both operands will be adapted to the width of the RDT before the multiplication will be performed.

$$fix(s,t) \cdot fix(u,v) \rightarrow fix(s+u, t+v)$$

For the conversion of a division operation the RDT will be calculated similarly to the multiplication, except of the word lengths of the fractional part. These word lengths will be subtracted instead of added. The resulting lower number of bits will be balanced with the integer parts. Instead of a data type adaptation after the operation, the dividend will be scaled. Due to this scaling, the dividend gets a new type, which must not under-run the word lengths of the LCDT. Consequently, the data type (word lengths) of the result also changes and must be adapted.

$$\frac{fix(s,t)}{fix(u,v)} \rightarrow fix(s+v, t-v)$$

In principle the number of bits in the fractional part is inherently defined by the layout of the fixed-point data type of the result. And this type is determined by the algorithm respectively hardware designer or even by a fixed-point evaluation tool in the previous design stage. This accuracy is exactly preserved during the conversions process.

The goal was to perform all these conversions automatically without any manual assistance. Therefore a tool, which will be described more detailed in the following section, has been developed.

## 3. Automated Fixed-Point to Integer Conversion Tool

The basic criteria for the development of the conversion tool, called **FixTool** [4], were an arithmetical correct application of the conversion rules described above, the preservation of parts of the code, which contain no fixed-point data types and arithmetics, including the code formatting, and finally the automated generation of directly synthesizable SystemC code.

The conversion is structured into four different stages: 1) parsing of the SystemC [7] code; 2) scope analysis and data type extraction; 3) analysis of all expressions in the code; 4) generation of the converted and adapted integer-based code. Figure 3 depicts the basic flow within the FixTool application.

At first the SystemC source code is analyzed syntactically and a complete parse tree is generated. The analysis within this stage cannot be restricted to SystemC keywords only or to keywords related to the fixed-point data types. Besides SystemC-specific syntax, the analysis must cover also C++ syntax [1]. In the code generation stage, the tool inserts e.g. brackets into the SystemC code. This procedure must preserve the evaluation order of expressions given in the original code. Therefore it is a fundamental prerequisite to be able to recognize SystemC (C++) expressions correctly. The parse unit can handle context-free grammars. The grammar itself is combined of C++ and SystemC, but is specified separately, so it can be modified and extended very easily. The parse tree will be the major repository of all information collected in the following analysis steps.

The second stage of the tool analyzes variable declarations and attributes. A very important issue is the scope of variables respectively classes. The FixTool annotates these scopes in detail. Data types (numerical data types) and the related scope information of variables or classes are the basis for the correct conversion. Both, data types and scope information are integrated into the nodes of the parse tree for the following processing stages.

The third phase analyzes all expressions found in the parse tree by evaluating the data types and the scopes of the partial expressions. The final type of a certain expression is evaluated by applying the related grammar rules and the data type information to the expression.

The fourth stage of the tool replaces the fixed-point arithmetics code parts and inserts appropriate integer-based code. These insertions are mainly type casts and shift operations as described in Section 2. Fixed-point variable declarations will be mapped to the appropriate SystemC integer data types. As mentioned above, one of the criteria for the development of the tool was that the original structure and layout of the code will be preserved if possible. It is very important for a designer to be as familiar with the converted code as with the original fixed-point-based model. Therefore the fourth stage does not generate independently new code, it rather uses a replacement strategy for pre-

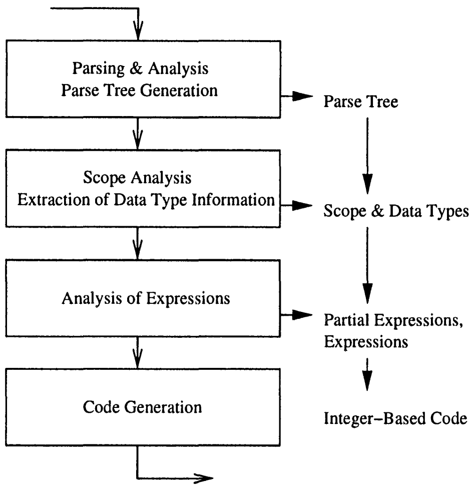


*Figure 3.* Conversion stages

cisely located code fragments. To be able to handle these code fragments, the parse tree also contains dedicated layout information of the source code file.

The conversion of entire projects consisting of several files requires the usage of a special project file. The project file specifies all source files and their corresponding header files. Similarly to the `make` tool, FixTool only converts files if they have changed since the last conversion run. Figure 4 shows a sample project description, which specifies the dependencies of different source modules (`source1.cpp`) with their related header files (`header1_1...`).

```
source1.cpp {header1_1.h ...
             header1_n.h source1.cpp}
source2.cpp {header2_1.h ...
             header1_m.h source2.cpp}
...
```

*Figure 4.* Project file

## 4. Simple Example

In this section we demonstrate the conversion methodology by applying it to a SystemC model that calculates square roots according to Newton's algorithm. The specification is using SystemC fixed-point data types and has been converted using the FixTool. A cutout of the original SystemC code, the calculation function, is shown in Figure 5.

```
void newton() {
  const sc_ufixed<1,1> null = sc_ufixed<1,1>(0);
  const sc_fixed<26,6> delta = sc_fixed<26,6>(0.00001);
  sc_uint<5> param    = in;
  sc_fixed<26,6> c    = sc_fixed<26,6>(param);
  sc_fixed<26,6> diff = sc_fixed<26,6>(0);
  sc_fixed<30,12> x0  = c;
  sc_fixed<30,12> x1  = c;

  do {
    sc_fixed<26,6> div = (sc_ufixed<2,2>(2) * x0);
    if (div == null) {
      x0 = null;
      break;
    }
    x0   = x1;
    x1   = x0 - (x0 * x0 - c) / div;
    diff = x0 - x1;
    if (diff<null) diff = sc_fixed<2,2>(-1) * diff;
  } while(diff>delta);
  out = x0;
}
```

*Figure 5.* Fixed-point-based Newton example

The resulting converted code (see Figure 6) shows the mapping of the SystemC fixed-point data types to SystemC integer data types. E.g. the constant `delta`, which is of type `sc_fixed<26,6>`, a fixed-point type with 6 bits integer width and 20 bits fractional part will be inserted. This constant is converted in a `sc_int<26>` type, the decimal point is only virtual. The underlying Newton's algorithm is quite simple, but it uses subtraction operations as well as multiplications and divisions and therefore gives a good example for demonstrating the methodology.

A comparison of this very simple example shows the changes introduced during the conversion process. The changes made in a manual translation are similar. It is evidently clear that the manual conversion is time-consuming and error-prone. The manual conversion of more complex algorithms is therefore very difficult and often messing. Optimized fixed-point data type word lengths are often softened during the debugging after a manual conversion process. Especially algorithms containing calculations with very different data type layouts (very different integer and fractional part word lengths) are critical for manual conversion. If certain variables are not local to some expressions, but rather used in the entire module in different expressions, it is also very difficult to do a manual conversion step by step. This requires additional type casting and data conversion effort during a stepwise conversion of the code.

```
void newton() {
  const sc_uint<1> null  = sc_uint<1>(0*1);
  const sc_int<26> delta = sc_int<26>(0.00001*1048576);
  sc_uint<5> param    = in;
  sc_int<26> c    = sc_int<26>(param*1048576);
  sc_int<26> diff = sc_int<26>(0*1048576);
  sc_int<30> x0  = (((sc_int<30>)(c))>>2);
  sc_int<30> x1  = (((sc_int<30>)(c))>>2);

  while(true) {
  do {
    sc_int<26> div = ((sc_int<26>)((((sc_int<30>)
          (((((sc_uint<32>)(sc_uint<2>(2*1))))*
          (((sc_int<32>)(x0))))))))<<2));
    if (div == (((sc_int<26>)(null))<<20)) {
      x0 = (((sc_int<30>)(null))<<18);
      break;
    }
    x0 = x1;
    x1 = ((sc_int<30>)(((((sc_int<32>)(x0))<<2) -
          ((sc_int<32>)(((((sc_int<52>)
          (((((sc_int<32>)(((sc_int<30>)
          (((((sc_int<60>)(x0)))*(((sc_int<60>)
          (x0))))>>18))))<<2)-(((sc_int<32>)
          (c))))))<<20) /(((sc_int<52>)
          ( div)))))))>>2));
    diff = ((sc_int<26>)((x0 - x1)<<2));

    if (diff<(((sc_int<26>)(null))<<20))
      diff = ((sc_int<26>)(((((sc_int<28>)
             (sc_int<2>(-1*1)))) *
             (((sc_int<28>)(diff))))));
  } while(diff>delta);
  out = ((sc_int<26>)((x0)<<2));
}
```

*Figure 6.* Converted Newton example

These models can only be handled efficiently using an automated fixed-point to integer conversion. The automated conversion process of the Newton's algorithm example shown in Figure 4 takes 0.96 seconds. In opposition to that, the manual conversion of this code may take several hours of work, including the correction of conversion errors. The simulation of both, the original fixed-point-based model and the converted integer-based model, leads to the same output values. Figure 7 shows the example output for the fixed-point-based calculation of square roots starting from 0 up to 9.

The corresponding output of the converted model using integer arithmetics is shown in Figure 8. The integer numbers in the output protocol are representing bit patterns containing a virtual decimal point with 6 bits on the left and 20 bits on the right, e.g. $3145728_d = 1100000000000000000000_b$. The virtual

```
       SystemC 2.0 --- Dec 11 2001 15:28:26
  Copyright (c) 1996-2001 by all Contributors
              ALL RIGHTS RESERVED
sqrt(0) = 0
sqrt(1) = 1
sqrt(2) = 1.414211273193359375
sqrt(3) = 1.732051849365234375
sqrt(4) = 2
sqrt(5) = 2.2360687255859375
sqrt(6) = 2.449493408203125
sqrt(7) = 2.645748138427734375
sqrt(8) = 2.828426361083984375
sqrt(9) = 3
```

*Figure 7.* Fixed-Point model output

```
       SystemC 2.0 --- Dec 11 2001 15:28:26
  Copyright (c) 1996-2001 by all Contributors
              ALL RIGHTS RESERVED

sqrt(0) = 0
sqrt(1) = 1048576
sqrt(2) = 1482912
sqrt(3) = 1816192
sqrt(4) = 2097152
sqrt(5) = 2344688
sqrt(6) = 2568484
sqrt(7) = 2774272
sqrt(8) = 2965820
sqrt(9) = 3145728
```

*Figure 8.* Integer-based model output

*Table 1.* Results

| | Fixed Point | Integer |
|---|---|---|
| Simulation Speed | 0.53 sec. | 0.03 sec. |
| Synthesizable | no | yes |
| Conversion Effort (Manual.) | > 2 hours | |
| Conversion Effort (FixTool) | 0.96 sec. | |

decimal point is located at position 20 (seen from the least significant bit) and therefore the represented number is $11.00000000000000000000_b = 3.0_d$. A simulation run for the fixed-point-based version takes 0.53 seconds, whereas the integer version takes 0.03 seconds for 100 square root computations[1]. The conversion results for the Newton's example are summarized in Table 1.

*Table 2.* Synthesis results.

| | |
|---|---|
| Combinational area: | 17677.474609 |
| Noncombinational area: | 1644.525757 |
| Net Interconnect area: | 61395.625000 |
| Total cell area: | 19321.390625 |
| Total area: | 80717.625000 |

This automatically generated integer-based code has been taken directly into the CoCentric SystemC Compiler flow without any additional modification for hardware synthesis. Parts of the result are shown in Table 2.

In addition to the simple Newton's algorithm example above, we also applied the tool to a more complex design of a hand prosthesis control unit [5]. The control unit is based on artificial neural networks, which contain subunits for the signal-processing of nerve signals and classification algorithms based on Kohonen's self-organizing maps (SOM) [9]. All computations in the nerve signal recognition stage and the signal classification module are originally based on floating-point arithmetics. In the refinement process of our SystemC design flow, the appropriate accuracies have been determined and fixed-point data types have been introduced. Finally, this fixed-point-based model was automatically converted. A comparison with the fixed-point version of the control unit shows that all output values of both simulations are exactly identical.

For comparison purposes an integer-based version of the prosthesis control unit has also been implemented manually. This implementation took several days, whereas the automated conversion was done in minutes for the entire design. This shows that the automated conversion will drastically save time. Especially against the background of the refinement philosophy of SystemC and the practical work in a design flow, this becomes important: All changes and modifications concerning the algorithm or the word lengths in a fixed-point-based model have to be propagated to the integer-based modelling level.

The FixTool ensures that the (synthesizable) integer version of a design is basically up-to-date at any time of the refinement process. It allows to evaluate and modify the fixed-point version of a model instead of the integer version (see Figure 9). In the fixed-point version of a model, the designer can easily exercise full control over the bit-layout and parameters of fixed-point data types.

Similar to the Newton's algorithm example described above, parts of the prosthesis control unit have already successfully been taken through the same hardware synthesis design flow to hardware.

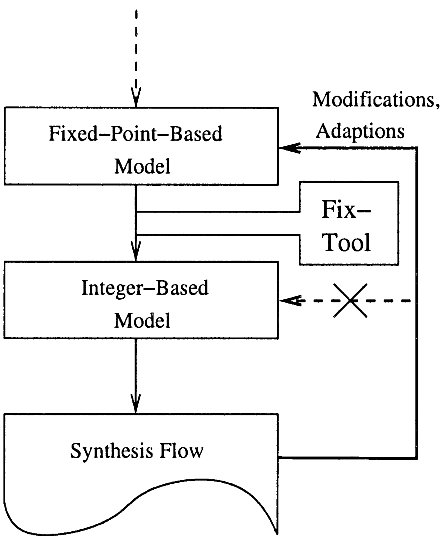


*Figure 9.* FixTool design flow

## 5. FixTool Application in a Real Case Study: Electrocardiogram Classification System

In addition to the last section, where a simple example of conversion of the Newton's algorithm was presented, we also have the objective to exemplify the application of the FixTool with a more complex design corresponding to the classification of electrocardiogram (ECG) signals. This system aims to present a new proposal in order to integrate the signal acquisition and the automatic ECG data classification in a single chip (e.g., SoC), close to the hospital patient [2]. Therefore, an IP (Intellectual Property component) was designed using the connectionist model (e.g., Artificial Neural Network - ANN) of the Artificial Intelligence area, which is capable to solve complex non-linear problems in many different areas.

ANNs are usually non-linear adaptive systems that can learn how to map input/output functions based on an input vector set. The adaptation process happens through well-defined rules that compose the training phase of the neural network. These training rules adapt the free parameters (e.g., synaptic weights) until the moment that the neural network achieves the pre-established stop criteria (for instance, minimum error or maximum number of epochs). After the training occurs, the free parameters should be fixed on the neural network in order to verify it in test phase [9].

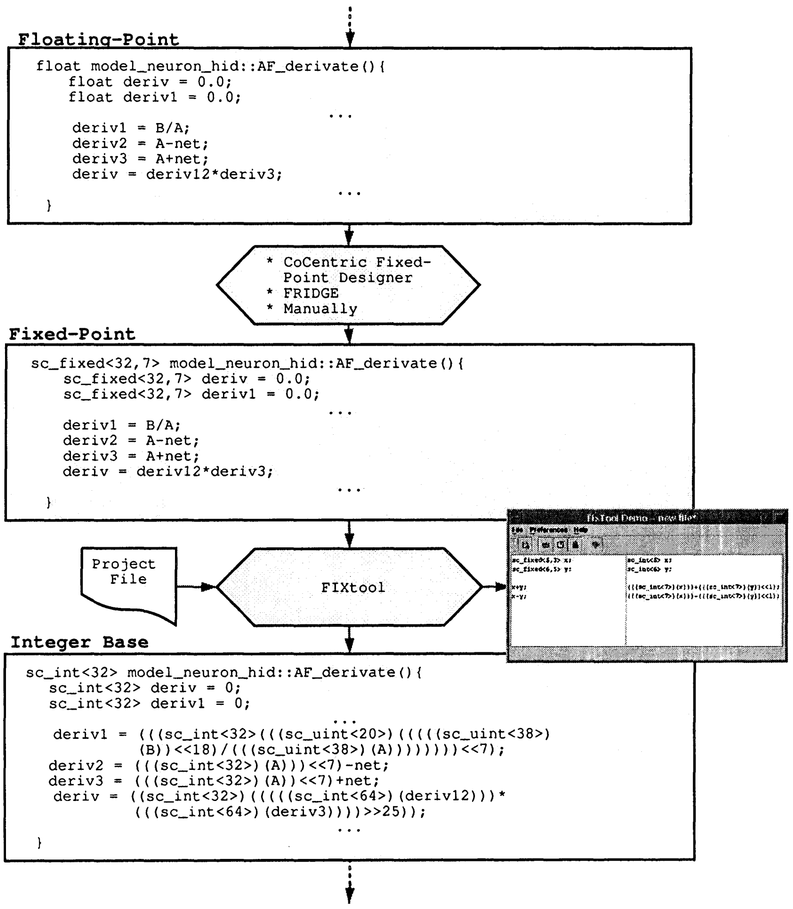
This electrocardiogram classification system consists of a multilayer perceptron (MLP) neural network, which is composed of parallel units (i.e., neurons) responsible to process and to classify the electrocardiogram based on the well known backpropagation algorithm [9]. The system is initially described on floating point arithmetic to capture the requirements of the system. In Figure 10, we present the design flow to convert floating-point arithmetic to integer-based arithmetic in the function AF-derivate(), which is responsible to derive the error signal to each neuron of the network. The system is initially simulated to achieve the specifications requirements. After that, the floating-point to fixed-point translation is applied for the determination of the word length and its precision. The FRIDGE and CoCentric Fixed-Point Designer are examples of tools for automating this process. Also, it can be done manually, in which it is necessary to explore the system in several stages of simulation to extract variable value ranges and their precision in number of bits.

Based on fixed-point system representation, the translation to integer base can be started. This process using FixTool can be done in two modes: a) using a GUI (Graphical User Interface), where some parts of the code can be copied to the application and the translation will occur automatically; b) using a project file, where all the source files and their corresponding headers are specified. This project file is executed in the shell console and then all the system will be translated from fixed-point to integer base.

In Figure 10 we can see also that not only the variables and the arithmetic expressions can be translated, but also the function declarations with their arguments. It is possible to figure out also how the result data types are determined. The explicit type cast is inserted to adapt the result of the expression to the capacity range of the variables.

Having the integer base description of the system, it is necessary to perform the simulation to verify if the system attends the same characteristics as before. Another fact that should be pointed out is the possibility to improve the simulation speed with the integer-based representations. This process was observed based on the number of epochs[2] that the neural network is trained. As is presented in Table 3, the simulation speed based on integer arithmetic can be around 2 times faster, compared to fixed-point representation. This slower simulation speed occurs due to the higher complexity transformations used to simulate the fixed-point representation.

The last phase of the design flow is the synthesis of the system. The hardware neural network design specified in SystemC was validated on a Xilinx SPYDER-VIRTEX-X2E board. This board contains one XCV2000E FPGA of the VirtexE family (i.e., equivalent to 2.000.000 logics ports) and one CPLD responsible for controlling the 32 address/date bits of the local bus. In addition, it contains SRAM memory blocks that operate synchronously. The frequency of the system can be selected up to 33MHz.



```
float model_neuron_hid::AF_derivate(){
    float deriv = 0.0;
    float derivl = 0.0;
                                   ...
    derivl = B/A;
    deriv2 = A-net;
    deriv3 = A+net;
    deriv = derivl2*deriv3;
                                   ...
}
```

```
sc_fixed<32,7> model_neuron_hid::AF_derivate(){
    sc_fixed<32,7> deriv = 0.0;
    sc_fixed<32,7> derivl = 0.0;
                                   ...
    derivl = B/A;
    deriv2 = A-net;
    deriv3 = A+net;
    deriv = derivl2*deriv3;
                                   ...
}
```

```
sc_fixed<5,3> x;          sc_int<5> x;
sc_fixed<6,5> y;          sc_int<6> y;

x+y;                      (((sc_int<7>)(x)))+(((sc_int<7>)(y))<<1);
x-y;                      (((sc_int<7>)(x)))-(((sc_int<7>)(y))<<1);
```

```
sc_int<32> model_neuron_hid::AF_derivate(){
   sc_int<32> deriv = 0;
   sc_int<32> derivl = 0;
                                   ...
    derivl = (((sc_int<32>(((sc_uint<20>)(((((sc_uint<38>)
             (B))<<18)/(((sc_uint<38>)(A))))))))<<7);
   deriv2 = (((sc_int<32>)(A)))<<7)-net;
   deriv3 = (((sc_int<32>)(A))<<7)+net;
    deriv = ((sc_int<32>)(((((sc_int<64>)(derivl2)))*
             (((sc_int<64>)(deriv3))))>>25));
                                   ...
}
```

*Figure 10.* From floating point to integer base

Figure 11 shows the tool chain to perform the synthesis process of the system originally described in SystemC. Initially, the specified system could be analyzed and verified by CoCentric System Studio [13, 10] and DAVIS visualization environments. After its functional verification, the floating-point to fixed-point data type conversion was performed using SystemC fixed-point types. The refinement of the fixed point date types to integer base was performed by the FixTool.

*Table 3.* Simulation speed improvement[3].

| Epochs | Floating(sec.) | Fixed(sec.) | Integer(sec.) |
|---|---|---|---|
| 1 | 2.56 | 6.15 | 3.06 |
| 10 | 26.33 | 62.56 | 30.25 |
| 100 | 257.80 | 629.02 | 304.09 |

The *CoCentric SystemC Compiler*, is capable to synthesize integrated RTL and behavioral modules in logic gates level (EDIF) or HDL RTL descriptions. In this work, an EDIF description was obtained and applied to the synthesis tools of the Xilinx platform. This process is composed by mapping to the desired prototyping board. The automatic synthesis was completed after the generation of FPGA configuration file.

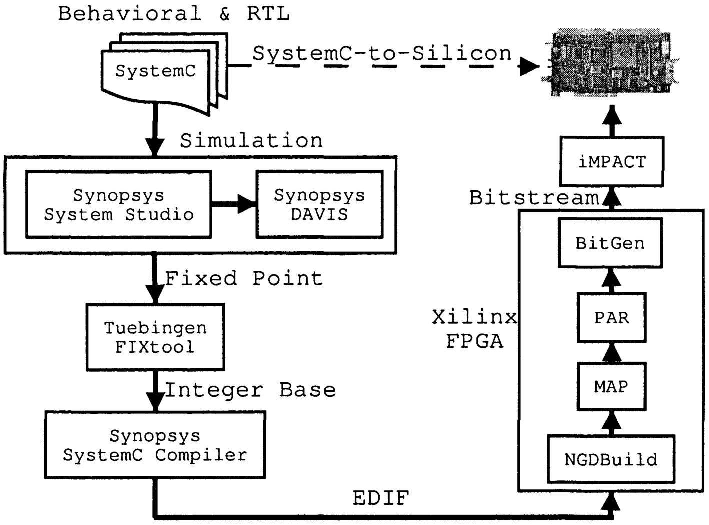


*Figure 11.* Synthesis process toolchain: from SystemC to Silicon.

Besides the synthesis of the electrocardiogram classification system, the classic XOR problem[4] was also synthesized so as to compare synthesis of a complex system (e.g., ECG) with a simpler one (e.g., XOR). These two examples are non-linear complex systems, which use neural network architecture with at least two layers. Thus, two ANNs were synthesized having the following architectures:

- ECG - 7 inputs / 5 neurons in hidden layer / 1 neuron in output layer;
- XOR - 3 inputs / 2 neurons in hidden layer / 1 neuron in output layer;

The synthesis results on the Xilinx platform are presented in Table 4. The first two columns present the elements and the maximum amount of them available in this FPGA. The last four columns present the synthesis results of the main system integrated with its testbench. The presented results also consider the memory block synthesis of both systems (ECG and XOR). One can notice that the ECG classification system occupied 90% of the present slices$^5$. The XOR system is simpler because it has a smaller number of: neurons, interconnections and amount of local registers to store the synaptic weights. In both systems three I/O blocks (IOB) are used and also an input for reset signal and two outputs to indicate neural network training process.

*Table 4.* Synthesis process summary.

| XCV200E | Max | ECG | % | XOR | % |
|---|---|---|---|---|---|
| Slices | 19200 | 17391 | 90 % | 5509 | 28 % |
| Block RAMs | 160 | 93 | 58 % | 1 | 1 % |
| Slice Flip-Flop | 38400 | 5441 | 14 % | 1858 | 4 % |
| 4 input LUT | 38400 | 31911 | 83 % | 10064 | 26 % |
| IOB | 404 | 3 | 1 % | 3 | 1 % |
| GCLKs | 4 | 1 | 25 % | 1 | 25 % |
| GCLKIOBs | 4 | 1 | 25 % | 1 | 25 % |
| CLK Freq. MHz | 33 | 7 | - | 16 | - |

## 6. Conclusion

We have described a methodology including all steps for the conversion of SystemC fixed-point data types and the related arithmetics into SystemC integer data types and adapted integer-based arithmetics. An implementation of the conversion tool FixTool allows the automated generation of integer-based designs out of fixed-point-based models. The tool covers all basic arithmetics like summation, subtraction, multiplication, and division. During the conversion process the structure of the original model code is preserved as far as possible in order to allow the designer to stay familiar with the design code at any time. The conversion results can be passed directly to a hardware synthesis tool, demonstrated by two examples.

The FixTool closes the gap in a system-level design flow e.g. between design tools, their floating-point-to-fixed-point features [12, 13], and hardware synthesis tools. It can avoid labour-intensive and error-prone manual conversion procedures within the SystemC refinement process.

Currently, further adaptations for a tight integration into the design flow are made. This also includes optimizations of the generated code in order to get an efficient adaptation to the synthesis tool, e.g. an optimized splitting of complex arithmetical expressions containing several divisions. These optimizations are mainly not related to the core conversion methodology, but to special requirements of the synthesis tools and the design flow beneath.

## Notes

1. SystemC 2.0 on a SunBlade 100 at 500 MHz.

2. Epoch is the presentation of the whole N vectors of the training set to artificial neural network input layer.

3. The simulations were executed in a Sun-Blade-100, with 1536 Mbytes of RAM memory and with 2201 Mbytes of Swap memory.

4. The XOR problem has four input vectors (0,0), (0,1), (1,0), (1,1). The first and the fourth belong to the class 0 and consequently the second and the third belong to the class 1.

5. One CLB is composed by two *slices*.

## References

[1] B.Stroustrup. *The C++ Programming Language (Special Edition). 2000.* Addison Wesley. Reading Mass. USA., 2000.

[2] D.Lettnin/A.Braun/M.Bodgan/J.Gerlach/W.Rosenstiel. Synthesis of embedded systemc design: A case study of digital neural networks. In *Design, Automation and Test in Europe Conference and Exhibition (DATE04)*, 2004.

[3] H.Keding/M.Coors/O.Luetje/H.Meyr. Fast bit-true simulation. In *38. DesignAutomation Conference (DAC)*, 2001.

[4] J.Freuer. *Entwurfsprozess einer Handprothesen-Steuerung in SystemC.* Diplomarbeit, Universitaet Tuebingen, 2002.

[5] M.Bogdan. *Signalverarbeitung biologischer Nervensignale zur Steuerung einer Prothese mit Hilfe kuenstlicher neuronaler Netzwerke.* Dissertation, Universitaet Tuebingen, 1998.

[6] M.Coors/H.Keding/O.Luetje/H.Meyr. Integer code generation for the ti tms320c62x. In *International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2001.

[7] Open SystemC Initiative. *Functional Specification for SystemC 2.0*, version 2.0-q edition, March 2002.

[8] Open SystemC Initiative. *SystemC User's Guide*, version 2.0 edition, 2002.

[9] S.Haykin. *Neural Networks: A Comprehensive Foundation.* New Jersey:Prentice-Hall, USA., 1999.

[10] Synopsys, Inc. *CoCentric SystemC Compiler Behavioral User Guide*, version 2000.11 edition, March 2001.

[11] Synopsys, Inc. *CoCentric Fixed-Point Designer User Guide*, version 2002.05 edition, 2002.

[12] Synopsys, Inc. *CoCentric System Studio Reference Manual*, version 2002.05 edition, June 2002.

[13] Synopsys, Inc. *CoCentric System Studio User Guide*, version 2002.05 edition, June 2002.

[14] T.Groetker/S.Liao/G.Martin/S.Swan. *System Design with SystemC.* Kluwer Academic Publishers, Boston/Dodrecht/London, 2002.

[15] W.Mueller/W.Rosenstiel/J.Ruf. *SystemC Methodologies and Applications.* Kluwer Academic Publishers, Boston/Dordrecht/London, 2003.