Predictable Success

Pain Killers for the Fixed-Point Design Flow

March 2010

Author

Holger Keding, Ph.D. Corporate Application Engineer,

Synopsys GmbH, Herzogenrath, Germany

Abstract

"How can something so trivial cause such a delay in your project?" This is one of the most frequently asked questions from engineers and managers who never had to suffer from the pains of converting a floating-point algorithm or system to a fixed-point representation. The latter becomes the golden reference and basis for the implementation in hardware or software. Floating-point implementations are rare because they consume more power, need more area, and cost more.

Most algorithms are developed using an abstract floating-point representation, making the quantization a mandatory step from the algorithm development phase towards an implementation. If the ugly details of algorithm quantization are understood then the dilemma that makes this phase so costly is known.

- Floating-point variables can be converted to integral values and use native data types of a computer (such as `int`, `long`, etc.) or customized data types based on those native types. Quantization operations have to be coded as shift or bit-masking operations, resulting in code that is error prone to write and hard to debug at the same time. A combination that can kill every schedule.
- The obvious alternative is to use libraries that offer generic fixed-point data-types for modeling and simulation. These fixed-point data types allow the user to explicitly define the word length and the number of integer bits. They also include convenient functions, operators and modes for quantization handling. Also, they allow for a smooth transition from a floating-point to a fixed-point system, as data types can be mixed without the need for explicit scaling whenever doing a quantization, and are easy to debug. This would be a perfect solution - except for the drastically decreased simulation speed. A fixed-point system using those libraries can easily simulate 10-50 times slower compared to the original floating-point design, which is not very well suited to produce results in time.

In addition, it is often not considered that every quantization is an application and implementation driven specialization of a more generic floating-point system. A floating-point algorithm is a flexible piece of IP that can be re-used in different systems. Creation or acquisition of floating-point IP can be considered as non-recurring engineering costs. In contrast, quantizing this floating-point IP does not produce a generic piece of fixed-point IP, since every application and implementation might impose different requirements on accuracy, area, cost, etc. Hence, the quantization step in a project must be seen as recurring engineering costs.

This article describes how advanced tools and methodologies can help with the challenges of fixed-point design, keeping the recurring engineering costs of the quantization process at a minimum. One crucial aspect is a simulation optimization technology that can in fact overcome the dilemma explained above.

Technical Introduction

There are many different approaches to transform an algorithm or system from a floating-point into a fixed-point representation, though the basic recipe is often the same.

1. **Specialization and simplification of the algorithm structure:** Many systems are initially created for algorithmic explorations, not taking any implementation aspects into account. They often contain *approximately ideal* algorithms, for example a low-pass filter that is implemented using a transformation into the frequency domain, a subsequent reset of all frequency components above the specified cutoff frequency, and a back transformation into the time domain. With a long FFT/IFFT length for the time-frequency transformations this becomes a good approximation of an ideal low-pass filter. Nevertheless, when it comes to implementation those approximately ideal algorithms are often replaced by less complex, easier to implement, and often easier to quantize algorithms such as an FIR or IIR filter. It is important to take such simplification decisions before quantizing the system. Otherwise, all effort put into this transformation will be lost. Also, quantizing an ideal frequency domain filter is much harder.

   Another aspect of specialization and simplification: algorithmic functions and blocks of a generic floating-point algorithm are often highly parameterizable. This renders the IP flexible and re-usable in different contexts and systems. For example an FFT might have an *fft_length* parameter – dependant on this parameter there is allocation of memory, calculations of scale factors, complex roots of unity, etc. Keeping all this flexibility for an implementation where the *fft_length* is actually known makes both the implementation costs and the quantization effort unnecessarily high. Costs and effort can be reduced by removing unnecessary generic parts, replacing them by constants or tables.

   Floating-point function calls are often found in the original algorithm, such as *sin(x) or cos(x)* function calls to calculate the sine or cosine of the argument x. In many cases the source code of the floating-point function is not available, since it is part of a standard C library. There are two options to quantize algorithms using such floating-point functions:

   - Leave in the call to the floating-point function and make sure that the argument and the result are quantized. For example the result of a sine function can be quantized with 8 bits and an integer word length of 2 and using rounding as quantization mode as follows:

     ```
     y = sc_fix(sin(x),8,2,SC_RND,SC_TRN);
     ```

     This is an easy way to get a bit-true specification of the result of the function. The big drawback of this type of quantization is that the final implementation of the *sin()* function is likely to be different from the floating-point algorithm on the simulation host. Many implementations will use a cordic algorithm, Taylor row / approximation, or a look-up table for the *sin()* or *cos()* function. This implies that even if the result of the operation has the same fixed-point format there may be slight differences in the numerical values. These differences are or at least should be harmless for the overall functionality of the algorithm, but they render the quantized algorithm with the floating-point functions useless as golden bit-true reference for the implementation.

   - The floating-point function can be replaced with a directly implementable construct, such as a cordic algorithm, a Taylor row, or a look-up table already in the floating-point description of the algorithm before starting the quantization process. This way the quantized algorithm can be implemented with the specified simplification in a bit-true way, and the quantized model can serve as a bit-true reference model for the implementation.

2. **Identify key objects for a quantization of the algorithm:** The fixed-point format chosen for these key objects determines the quantization loss, i.e. the difference in algorithmic performance in comparison with the floating-point representation. Typical key objects are state variables or arrays, (filter) coefficients, accumulators, and the input and output ports.
3. **Collect statistics for key objects:** For the key objects the statistical characteristics need to be determined, such as the value range, histograms, or similar data. The value range of a floating-point variable will indicate the *integer word length (iwl)*, i.e. the number of bits on the left side of the binary point of the final fixed-point format for this variable. It is important to run simulations with sufficient and relevant stimuli. To gain the required data needed to instrument the algorithm code. This instrumentation will record and collect the required data during simulation runs. Once this data is collected the required *iwls* can be calculated for the instrumented variables. This step often requires a manual instrumentation of the code and a well organized system to store the collected data and associate it with the corresponding variable or code location.
4. **Determine required precision for key objects:** After the *iwls* for all key objects are known, find the required *precision, or word length (wl)* for each key object. The required precision can only be defined using a quality criterion for the overall algorithm performance, such as a bit error rate (BER) or similar. The reference is always the behavior of the floating-point algorithm, and the constraints, such as allowed BER, is in most cases set by a standard document or company internal field studies. The only reliable way to find out about the required word length *wl* for an object is to use a fixed-point casting statement or a fixed-point data type for this object, where the known *iwl* is hard-coded and the overall *wl* is a parameter and can be varied. This way the user can run several simulations with different word lengths and create a *Quality criterion vs. wl diagram*. The latter enables a dependable decision to be made on what word length is needed for this key object. In most design flows this represents the most painful step due to the dilemma explained in the previous section. A decision between hard-to-write-and-debug, but fast simulating integral data types, or easy-to-use but slow simulating fixed-point data types needs to be made. The recommended order of quantizing key objects is from global to local, i.e. starting with the top-level signals/ports. This provides fixed interfaces for the connected blocks which also enables multiple engineers to work on the quantization of the connected blocks in parallel.
5. **Determine fixed-point formats for remaining objects:** Once all the key variables are quantized there will be a few other objects, e.g. local or temporary variables, in the code that are still in floating-point format. Their required fixed-point format is determined by the key objects and thus can be calculated in most cases. If it can't be calculated, additional simulations may be needed to determine *iwl* and *wl* as explained in step 3 and 4.

Details of this recipe may vary, depending on the implementation target – for example software implementations sometimes get away with using the available register/memory width of the processor as *wl* for all key objects – but even that only saves a bit of effort in step 4, while the other steps are still mandatory. Especially the manual, error prone and time consuming steps 3 and 4 make the quantization process a rather painful experience.

System Studio Solution

System Studio comes with a bundle of pain killers for the quantization process, consisting of unique simulation optimization techniques and a set of well organized utilities to ensure an efficient design flow. Many of those utilities are specific to the quantization process, while others are of more general nature, but very useful for this design flow, too.

Data Type Support

System Studio supports the SystemC fixed-point data-types as specified in the IEEE 1666 standard. `The sc_fix and sc_fixed<>` data-types can be parameterized with a word length *(wl)*, an integer word length *(iwl)*, a quantization mode (such as rounding, truncation, etc.), and an overflow mode (such as wrap-around, truncation, etc.). Compared to native integral data types, the SystemC data types enable the following

- The word lengths can be as large as required.
- The word length *wl* can be different from the integer word length *iwl*, so they do not require any up scaling when converting floating-point values to fixed-point types, or other adaptation code when mixing floating-point values and fixed-point values. This is a tremendous advantage not only when writing the code, but also when analyzing/debugging the algorithm, as all value ranges are preserved. The value of π in the debugger will still be in the range of 3.14 and not 12868 because it happens to be represented by an integer with 12 bits.
- Casting, such as overflow handling (wrapping, clipping, etc.) and quantization (truncation, rounding, symmetrical rounding, etc.) does not need to be modeled using bit-masking, shifts, or other basic operations, but selection is to be made from a large set of convenient overflow handling and quantization modes for a type or operation.
- In addition, all fixed-point parameters, (integer) word length and casting modes can be changed in the simulation, so it is easy to evaluate the effects of different (integer) word lengths or casting modes on the quality of the algorithm.
- Operators on the IEEE 1666 fixed-point data types are also defined and make the types very efficient for modeling – e.g. there is no need for manually adjusting the location of binary point / scaling before an addition, since this is done automatically.
- System Studio also supplies special support for debugging code with SystemC data types – anybody who ever had to deal with a debugger spitting out all the guts of a SystemC class instead of simply printing the value of a variable will highly appreciate this feature.

Fast Fixed-Point Simulation

Using the fixed-point types of the SystemC library for simulation comes at a very high price in terms of simulation speed. The reason is the inherently generic nature of a data type library. Internal operations such as alignment of binary points or allocation of containers for intermediate results have to be executed at runtime.

This is where the optimizations in System Studio benefit the user. These optimizations allow the use of the SystemC fixed-point data types for modeling. System Studio parses the entire algorithm and translates the SystemC data types back to native data types of the host machine, determines the location of the binary point in advance, and optimizes the number of shift and cast operations, etc.

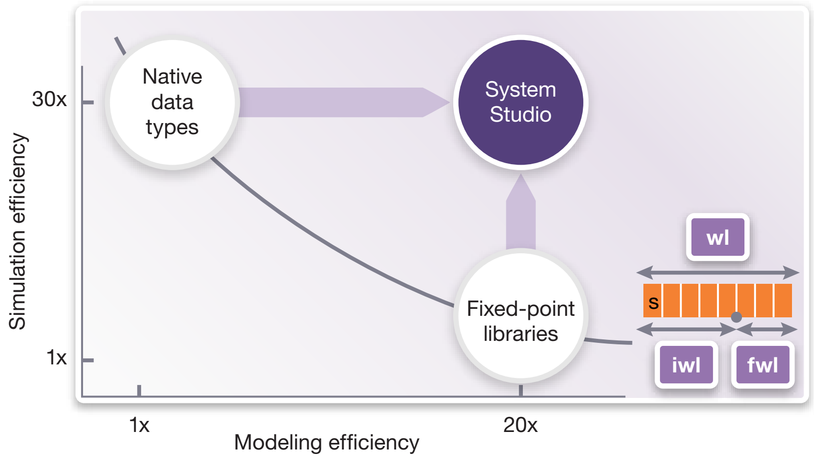


Figure 1: System Studio allows for both, efficient modeling and fast simulation

This leads to a significant speedup of the simulation, the same simulation speed is realized as if the algorithm was coded manually using integral data types plus shift and bit-mask operations for fixed-point modeling. This means that System Studio overcomes the dilemma mentioned in the beginning: High modeling efficiency is achieved by using the SystemC fixed-point data types, and at the same time achieves the speed of a native data type based simulation, as illustrated in Figure 1. Instead of 30 times slower simulations on average, compared to floating-point, simulations are back in the range of 1-3 times overhead only (depending on the type and number of quantization/overflow operations, the size of the data types, etc.).

Code Instrumentation and Statistics Data Collection

As outlined in item 3 in the Technical Introduction section one of the first steps of the quantization process is to collect statistical data such as the value range of a variable, or a histogram. In System Studio there is no need to manually instrument the code to achieve this –simply pick the ports, variables, or arrays in the design that need to be monitored and the appropriate data will be collected and processed automatically. Examples for the automatically calculated data by System Studio are the required *iwl* of a variable, the mean value, the variance, or a histogram.

Data Type Switching

Many experienced teams try to keep the maintenance effort for their IP as low as possible by avoiding unnecessary duplication. Keeping both the floating-point and the fixed-point representation of an algorithm in one model is one way to achieve this. System Studio supports this with conditional type parameters. For example, a type parameter can be set to a fixed-point type in case a top-level Boolean parameter ***FxSim*** is set to `true`, and otherwise set to floating-point types – see Table 3 for an example. This allows using the same source code for both the floating-point and fixed-point simulation.

Helpful Features

There are numerous other helpful features in System Studio that enable an efficient design flow for a quantization. To name a few:

- Using System Studio’s Tcl based scripting interface easily launches and controls different simulations with different word length parameters to explore the fixed-point design space of the algorithm or post-process simulation results.
- Using System Studio’s support for parallel simulation can fan out different iterations, i.e. simulation with different parameter sets, to different machines. The results are synchronized and collected automatically in a central location. This also helps the user to get to results much faster.

Tutorial Example

As a demo and tutorial for a System Studio based fixed-point design flow the carrier synchronization / QPSK receiver example that is also shipped with the training material in the System Studio releases is used. Starting with version 2009.12-SP1 of System Studio the single steps explained in this tutorial are also available in the library ***lab_advanced_fixed_point*** in the training material. Read through this tutorial section and look up the details in this training library, or try out the steps explained here to get some hands-on experience.
The library ***lab_advanced_fixed_point*** itself contains the original floating-point version of the design and the sub-libraries ***step1*** through ***step5*** contain intermediate states of the quantization process. The README files in these sub libraries contain additional information on modifications or simulation results.

The Algorithm

The core part of the QPSK receiver frontend is a root raised cosine filter. In the floating-point version of the example, the model ***RRCosine*** was used from the ***models/filter*** library in System Studio. It basically consists of an FFT, a filtering stage in the frequency domain, and a re-transformation to the time domain using an IFFT. After the RRC filter comes a phase error correction, consisting of a phase error detection, a loop filter (NCO) providing the angle for a ***Rotation*** block that actually compensates the phase error. Finally, a QPSK demodulation block is followed by decoding blocks (gray, differential). The latter decoding blocks are not of interest here, since they operate on integral values already and do not need to be quantized.

Specialization and Simplification of the Algorithm Structure

By using the example system, the different types of specialization and simplications will be described:

- *Implementation alternatives*, i.e. the change of subsystem from an ideal algorithm into a more implementation friendly alternative.
- *Implementation refinements*, i.e. removing generic parts of algorithms.

Implementation Alternative: the Root Raised Cosine Filter as FIR Filter

In this step the root raised cosine filter will be changed from an ideal frequency domain filter into a much less complex FIR implementation. With an overall error criterion (BER<5*10-6) this simplifies the algorithm and implementation significantly and delivers good results at the same time. In a first step, the filter design tool QED in System Studio is used to find a set of suited filter coefficients – the parameterization of the ***RRCosine*** block can be used directly to obtain the parameters used to calculate the filter coefficients:

- FIR filter, Filter Type: RootRaisedCosine
- Frequency Mode: Hertz
- Symbol Rate: 1
- Rolloff Factor: 0.25
- Sampling Rate: 4
- Passband Ripple: 0.1dB
- Stopband Ripple: 30dB
- Type/Windowing: Rectangular
- Number of Coefficients: 17

This results in floating-point coefficients stored in the file `step1/float_coeffs.dat.` The ***FIR*** filter model from the library ***filter*** can be used as a starting point to implement this filter. Using the System Studio profiler (or any other) it becomes visible that the computational complexity of the receiver filter (***rx_filter***) is reduced significantly (a quarter of the ***RRCosine*** block), while the results in terms of overall BER are still the same. The modifications described above can be reviewed in library ***lab_advanced_fixed_point/step1***.

Implementation Refinement: simplify the Root Raised Cosine FIR block

Next, the ***FIR*** filter model from the library ***filter*** can be copied into a local library and simplified as permitted by this specific application. The major changes to the FIR filter model in the ***library lab_advanced_fixed_point/ step2*** are:

- Removal of the different alternatives for coefficient storage
- Inlining of the coefficients as local array instead of reading them from a file
- Introduction of additional data type parameters, such as ***Taccu***, to be able to use different fixed-point types for specific key variables such as the accumulator that typically uses a larger number of bits

Implementation Alternative: look-up tables instead of trigonometric floating-point functions

Another example for implementation alternatives is the mapping of floating-point functions such as `sin(x), cos(x), arctan2(x,y)` to look-up table implementations, that are cheaper to implement and that can be quantized in a bit-true way.

The original phase error compensation uses the ***Rotation*** block of the ***channel*** library, implementing the following operation:

```
OutData = InData*polar(1.0,-InPhase);
```

This can be re-written as

```
OutData = InData * complex(cos(-InPhase),sin(-InPhase));
```

The `cos()` and `sin()` functions can be replaced by look-up tables, e.g. using the models ***CosTable*** and ***SinTable*** from the ***basemath*** library. These models have a parameter ***TableSize*** that can be used to determine the required size of the look-up table, e.g by running different iterations of the simulation with different table sizes and recording the effect on the BER.

Tip:
Especially in fixed-point designs care needs to be taken with the size of look-up tables. The look-up table as such already represents a quantization of the output values. If in addition the output data type has different quantization steps, i.e. the value of the least significant bit (MSB), is not aligned with the look-up table values, and the input data to the look-up table is quantized the look-up table can introduce significant errors. In case of doubt, first select a rather large value for the table size, and try to reduce it later on when the exact word lengths of the input and output signals are known.

`The arctan2(x,y) (or arg(c)` with a complex valued c) function can also be coded as a look-up table, though the implementation is not as straightforward as in the `cos()` case: the `arctan2()` function takes two arguments and uses the sign of `x` and `y` to determine the quadrant of the resulting angle, while using `arctan(x/y)` to determine the angle itself. Since `r=x/y` is not bounded a non-linear look-up table for `r` is needed, that maps the input values `r=x/y` to the angle values. The model ***ArcTan2Table*** from the ***basemath*** library can be used to implement this lookup table. Like the ***CosTable*** model it also has a ***TableSize*** parameter to explore what table size is needed to obtain acceptable algorithmic performance. After these transformations all elements of the receiver can be implemented in a bit-true way, in other words the actual quantization of the algorithm can be started.

Identify Key Objects for a Quantization of the Algorithm

The key objects for a quantization are relatively easy to find:

- The interface elements, i.e. input/output ports and floating-point parameters.
- The key variables – including arrays of the leaf-level PRIM models: Key variables are central elements of an algorithm. Their accuracy, i.e. fixed-point format significantly impacts algorithmic performance of the model. Their fixed-point format cannot be derived from other formats or elements – in contrast to temporary variables and intermediate results. Examples for such key elements would be the accumulator or the coefficients of the FIR model for the matched filter of the receiver.

Tip:
To ease the quantization process key elements should be visible during the simulation, i.e. they should be state variables declared in the header section of the PRIM model in System Studio, so you can query their state and collect statistics. In case you have key variables that are declared locally in the `main_action()` – such as an accumulator of a filter - you should modify the model and promote these key variables to states.

Collect Statistics for Key Objects

To collect statistics and have System Studio calculate the required integer word length *(iwl)* for each key element from the collected data is straightforward. Select the instances containing the key objects in the design and pick ***Highlight Instance in Hierarchy Tree*** from the context menu. This will open the design's ***Hierarchy Tree*** window, with the selected element highlighted, as shown in Figure 2.

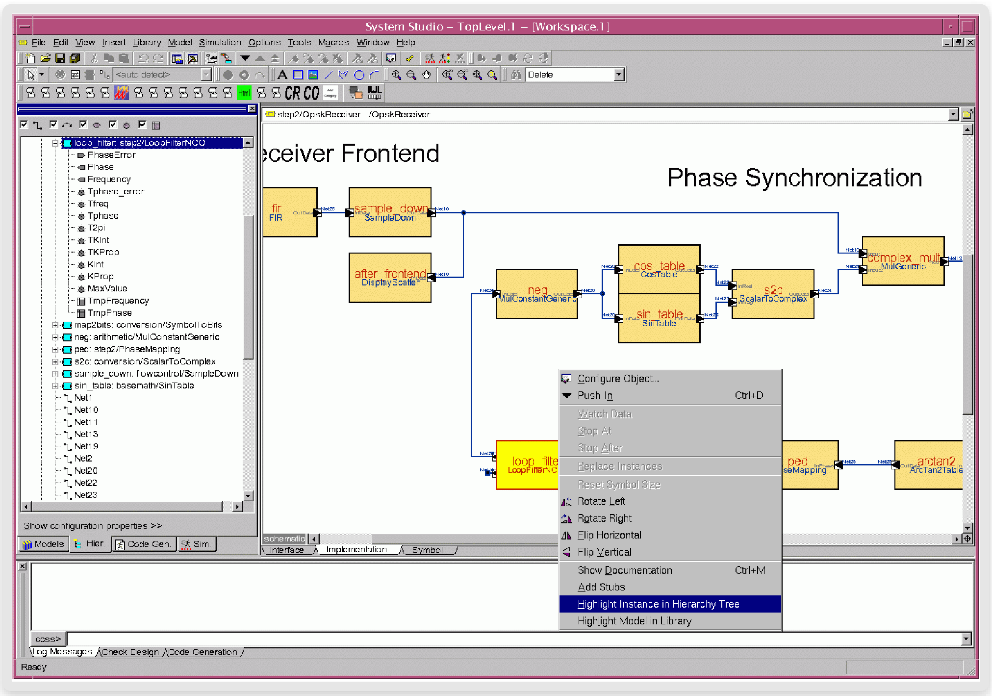
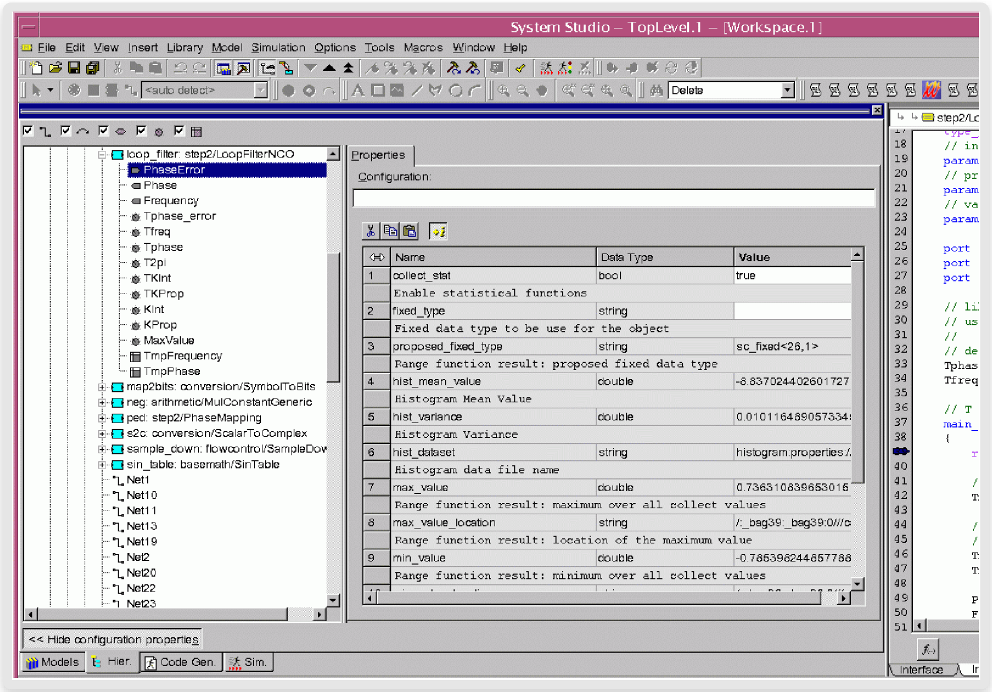


Figure 2: Selecting an instance in System Studio's design hierarchy tree

Clicking on the ***Show Configuration properties*** line at the bottom of the ***Hierarchy Tree*** window will open the properties page for the hierarchy elements, and enter `true` in the ***collect_stat*** field for all key elements identified above. During the next simulation run System Studio will generate an object configuration database `(.ocd)` file in the simulation directory that can be merged into the hierarchy tree as follows:

- In the ***Hierarchy Tree*** window, select ***Configuration Properties>Merge***… from the context menu (right mouse button), and browse for the `design_properties.ocd` file in the activation/iteration directory of the simulation, and click OK.
- The fields of the ***Configuration Properties*** dialog will now be filled in with the collected data, as shown in Figure 3. The most important data for the quantization are the minimum and maximum values, as well as the proposed fixed-point format derived from this. Note that the calculated fixed-point parameters are conservative, i.e. provide a fixed-point format that can hold the same information as the floating-point representation. If, for example, the proposed fixed-point format for the ***PhaseError*** input port of the ***loop_filter*** is `sc_fixed<26,1>` typically only the integer word length *(iwl=1)* and the signedness from the proposed format is kept. The overall word length *(wl=26)* can usually be reduced significantly without introducing too many quantization errors, so in a later optimization step a compromise between costs of implementation and algorithmic performance can be found. In addition to the statistics and formats, for every selected element System Studio generates a histogram that can be viewed using System Studio's data visualization tool DAVIS.
- If there are different sets of test vectors for the application, it is important to run a significant number of these sets and merge all collected data into the System Studio data base, which can be done by repeating the merging described in item 1 for all iterations with different input test vectors. Only this way will make sure the collected data reflects the requirements of the application.



System Studio – TopLevel.1 – [Workspace.1]
File Edit View Insert Library Model Simulation Options Tools Macros Window Help
<auto detect>
Delete
<< Hide configuration properties
Models
Hier.
Code Gen.
Sim.
Interface

- loop_filter: step2/LoopFilterNCO
  - PhaseError
  - Phase
  - Frequency
  - Tphase_error
  - Tfreq
  - Tphase
  - T2pi
  - TKInt
  - TKProp
  - KInt
  - KProp
  - MaxValue
  - TmpFrequency
  - TmpPhase
- map2bits: conversion/SymbolToBits
- neg: arithmetic/MulConstantGeneric
- ped: step2/PhaseMapping
- s2c: conversion/ScalarToComplex
- sample_down: flowcontrol/SampleDow
- sin_table: basemath/SinTable
- Net1
- Net10
- Net11
- Net13
- Net19
- Net2
- Net20
- Net22
- Net23

Properties

Configuration:

	Name	Data Type	Value
1	collect_stat	bool	true
	Enable statistical functions		
2	fixed_type	string	
	Fixed data type to be use for the object		
3	proposed_fixed_type	string	sc_fixed<26,1>
	Range function result: proposed fixed data type		
4	hist_mean_value	double	-8.837024402601727
	Histogram Mean Value		
5	hist_variance	double	0.010116489057334
	Histogram Variance		
6	hist_dataset	string	histogram:properties:/
	Histogram data file name		
7	max_value	double	0.736310839653015
	Range function result: maximum over all collect values		
8	max_value_location	string	/:_bag39:_bag39:0///c
	Range function result: location of the maximum value		
9	min_value	double	-0.785398244857788
	Range function result: minimum over all collect values		

Figure 3: Configuration Properties dialog after merging back simulation results

Tip:
Before introducing fixed-point formats into the design it is a good idea to modify its structure to be suited for a fixed-point implementation. In a floating-point design the number of data types used is naturally small since most ports/signals and variables are of type `float or double`. In a fixed-point design this changes, so it makes sense to introduce additional type parameters. As an example, in a FIR filter use a type parameter ***Taccu*** for the accumulator, ***Tsample*** for the input/output data, and a ***Tcoeff*** for the coefficients. This way the quantization process and the subsequent mapping of the quantized algorithm to hardware or software becomes easier.

While some of these additional type parameters will only affect the internal computations in a model, others will have more global effects, such as the data types for the input and output ports. Model ports that are connected via signals need to be of the same data type. Instead of setting the type parameter values for the connected models in a DFG individually, it is more convenient and avoids inconsistencies to use hierarchical type parameters. Examples are the type parameters ***Tsample***, ***Tphase***, etc. in the design ***step2/CarrierRecoveryTop*** – as also illustrated by the screenshot in Figure 4.

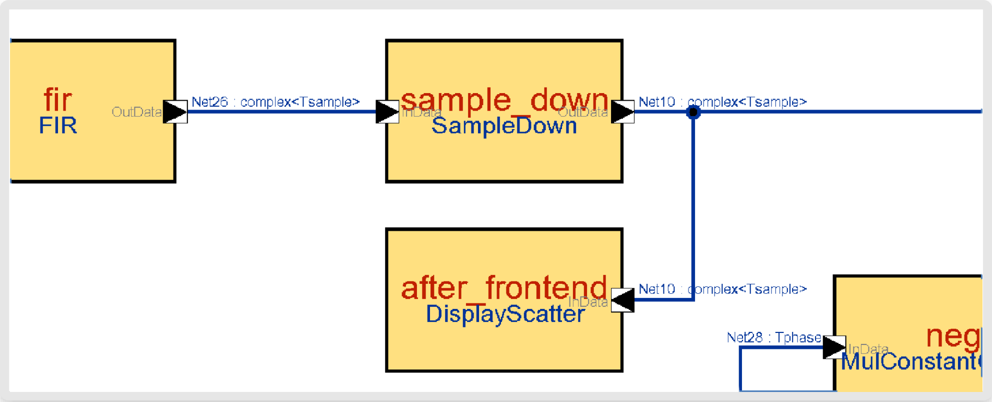


Figure 4: Using hierarchical type parameters to ensure type consistency in a DFG

Tip:

SystemC data types come in 2 flavors: `sc_fix[ed]` and `sc_fixe[ed]_fast`. The former can be used for arbitrary word lengths, while the latter uses a member of type double to store the values, i.e. is restricted to 52 bits. If this limitation is acceptable to you, then you should use the `sc_fix[ed]_fast` types. They are roughly a factor of 2 times faster than the unlimited precision `type sc_fix[ed]`. The `suffix_fast` is a bit misleading, though, as you can clearly see from the benchmark results in section Benchmarking Results.

Determine Required Precision for Key Objects

The next step is to change the data type of the key objects from a floating-point type to a fixed-point data type with a parameterizable word length *wl* so multiple simulations with different word lengths can be done. The effects of different precisions of a specific variable on the overall quality criterion can be explored, resulting in a BER vs. word length similar to the graph shown in Figure 5.

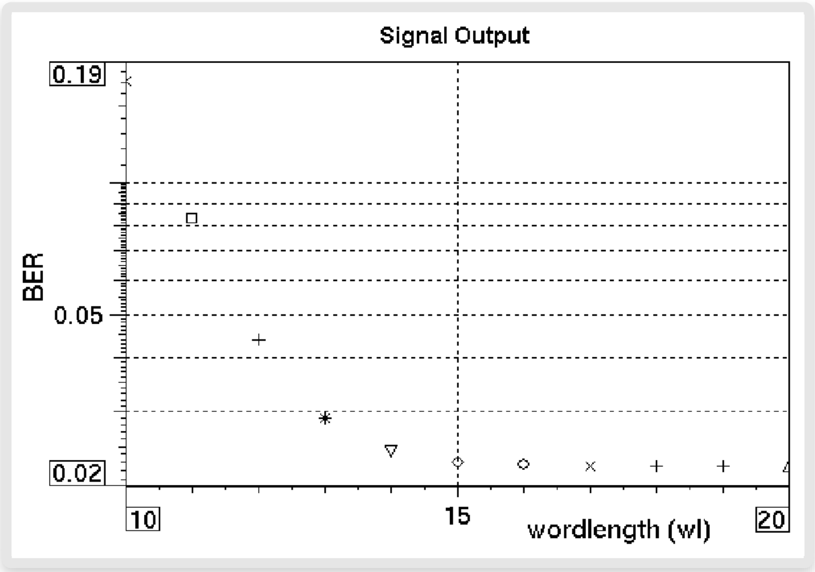


Figure 5: Example of a word length vs. BER graph in DAVIS

A very efficient way to explore the effect of different word lengths on the BER is to run scripted parameter iterations in System Studio, using a top-level parameter `wl` that is used in a hierarchical type parameter of the **QPSKReceiver**:

Parameter Name	*Data Type*	*Value*
Tsample	*type_param*	*sc_fix_fast(wl,3,SC_RND, SC_WRAP)*

Table 1: Using a word length parameter wl to explore the effect of different word lengths on the BER

This parameter can be set from a simulation control file, i.e. a script that launches and controls different iterations of the same simulation executable with different parameter values set. The following example of a simulation control file launches a row of iterations with the word length parameter wl set from 4 to 16, but a relatively small number of samples to be simulated:

```
set_value NumSamples        1e6
for {set wl 4} {$wl <= 16} {incr wl 1} {
  set_value wl $wl
  run_iteration
}
```

Due to the relatively small number of samples - definitely not enough to get sufficient statistics for an expected BER in the range of 10-6 – the simulations are relatively short and quick feedback will be provided for which word lengths it makes sense to start longer simulations. From the following simulation output it can be seen that for word lengths below 6 the BER is definitely too high, so it would make sense to try longer simulations for word lengths between 6 and 10, for example:

```
netbatch: job 6 has been submitted to grd system with id 4638550
netbatch: job 7 has been submitted to grd system with id 4638551
netbatch: job 8 has been submitted to grd system with id 4638552
netbatch: job 9 has been submitted to grd system with id 4638553
netbatch: job 10 has been submitted to grd system with id 4638554
wl = 6 : BER: 4.4e-07
wl = 7 : BER: 0.0
```

For a number of longer simulations with 108 samples it is advisable to use System Studio's Netbatch utility to submit different iterations for parallel execution on a compute farm, using load job scheduling systems such as SunGrid Engine (aka Gridware) from Sun Microsystems or LSF from Platform Computing. To use parallel iterations select the load sharing system – in this example Gridware was used – in the StartOptions dialog, and adapt the SCF file to submit the simulations in parallel in a first loop iterating over the word lengths and then wait for the results of the parallel iterations to return in a second loop. Examples for such SCF files using Gridware are the files `CarrierRevoveryTopNB.scf` in the libraries ***step3*** and ***step4*** of the ***lab_advanced_fixed_point library***. The following shows the main sections:

```
set_value NumSamples        1e8
set simname $env(SIM_NAME)
…

set wl_min 6
set wl_max 10

for {set wl $wl_min} {$wl <= $wl_max} {incr wl 1} {
  set_value wl $wl
  nb_submit_iteration $wl
}

for {set wl $wl_min} {$wl <= $wl_max} {incr wl 1} {
  puts -nonewline "wl = $wl : "
  nb_catch_iteration $wl
  set biterrors($wl) [show_value /$simname/ber/biterrors]
  set counter($wl) [show_value /$simname/ber/counter]
  puts "BER: [expr $biterrors($wl) / double( $counter($wl) )]"
}
```

In the shell all the jobs can be seen to be submitted immediately, and as the jobs complete, the results will be printed:

```
netbatch: job 6 has been submitted to grd system with id 4638550
netbatch: job 7 has been submitted to grd system with id 4638551
netbatch: job 8 has been submitted to grd system with id 4638552
netbatch: job 9 has been submitted to grd system with id 4638553
netbatch: job 10 has been submitted to grd system with id 4638554
wl = 6 : BER: 4.4e-07
wl = 7 : BER: 0.0
```

From the results above it is seen that for the fixed-point format of the type ***Tsample*** a word length larger than 7 does not improve the BER any further. Note that the other data types are still floating-point. Hence, choosing the least word length – in this case 7 – is not recommended, since the number of remaining elements to be quantized is still large. With every bit that is cut-off additional errors can be introduced that will accumulate as the number of quantized elements increases. This typically leads to a scenario where the first quantized elements have a short word length, while those that are quantized later have a significantly larger word length to cope with the large quantization errors introduced earlier. It is more advisable to spend at least one or two extra bits for the word length. This usually keeps the overall word lengths smaller. After the word length is chosen the parameter ***wl*** can be replaced in the type parameter with the hard coded value:

Parameter Name	***Data Type***	***Value***
Tsample	***type_param***	***sc_fix_fast(8,3,SC_RND, SC_WRAP)***

Table 2: Swap the word length parameter wl for the hard coded word length 8

Continue with this process for the other ports/variables of the design: first run short exploration runs to find a suited word length interval and then send off longer simulations to get precise results. Once more than 2-5 fixed-point variables are used in the design it is highly recommended to use System Studio's fast fixed-point optimization that can speed up the simulation significantly. Simply check the ***Fast fixed-point*** box in the ***Code Generation*** tab as shown in Figure 6. This optimization maps the SystemC fixed-point data types to native integral data types of the host machine and thus reduces the number of run-time tasks and storage management. The more fixed-point types are used in the design and the less mixed fixed/floating-point code is used, the higher the impact of code optimizations -see also the benchmarking results presented in the Benchmarking Results section.

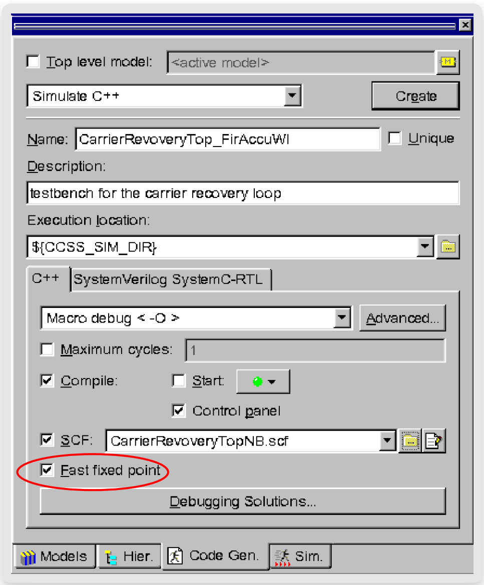

☐ Top level model: <active model>
Simulate C++ — Create
Name: CarrierRevoveryTop_FirAccuWl ☐ Unique
Description:
testbench for the carrier recovery loop
Execution location:
${CCSS_SIM_DIR}
C++ | SystemVerilog SystemC-RTL
Macro debug < -O > — Advanced...
☐ Maximum cycles: 1
☑ Compile: ☐ Start:
☑ Control panel
☑ SCF: CarrierRevoveryTopNB.scf
☑ Fast fixed point
Debugging Solutions...

Figure 6: Fast fixed-point optimization option

Tip:

When quantizing data types you have the choice of using different quantization modes, i.e. different ways of removing the least significant bits (LSBs). The most common are rounding (`SC_RND`) and truncation (`SC_TRN`), where `SC_TRN` is the default quantization mode for the SystemC fixed-point types. The disadvantage of truncation is that you always introduce a negative quantization error, since LSBs are cut off, which in both are cut off, unsigned and two's complement signed representation, carry positive values. This means that an originally bias free signal becomes biased. Using rounding removes this bias, but rounding is considered more expensive, since it involves an extra addition. Instead of always picking truncation due to the costs, it is recommended though to closely analyze the costs of rounding. In many cases the rounding is actually not done at runtime, such as the quantization of the constant coefficients of the FIR filter. If rounding (SC_RND) is picked here, the results will improve, but there is no additional cost associated with this. Even a final hardware or software implementation will only contain the already quantized coefficients without any rounding operation required.

Tip:

The quantization of a complex system is an iterative approach, introducing many non-linear effects that often have puzzling effects. In many situations it is very helpful to:

1. Document intermediate results and decisions on which (integer) word length to pick for the elements in the design.
2. Have the opportunity to revert changes or return to a prior status known to be correct.
3. Have an easy way to compare the algorithmic performance of the (partially) quantized algorithm against a floating-point representation of the same system.

To achieve these goals the following practices have proven to be very helpful:

1. When running long simulations exploring the effect of the word length of a certain element on a quality criterion, use a simulation name that you can recognize when generating the simulation from the ***Code Generation*** panel. For example pick the simulation name `CarrierRecovery_RecFirCoeffWl` for iterations over the word length parameter for the coefficients of the FIR filter in the Receiver – the generated simulation directory will have the same name. After the simulation, the simulation directory should not be removed as it contains most valuable information such as the simulation results, the used simulation control file reflecting the used parameter values, etc. In case of doubt it is useful to be able to document and/or review the results and settings.
2. Use a revision control system such as *CVS* or *ClearCase* to check relevant intermediate results, e.g. after the quantization of all signals/ports of the receiver, use recognizable tags or labels for these check-ins, and provide meaningful check-in comments. This way you will be able to revert unwanted changes or test against known-to-work prior states of the design in case something goes wrong. You can achieve the same by copying the design after each relevant intermediate step, but this is typically less convenient.
3. You can keep your models 'convertible' in System Studio, i.e. have them switch from a fixed-point implementation to a floating-point implementation. This way one source for both the fixed-point and floating-point implementation is needed, thus avoiding inconsistencies as you refine a model in the quantization process. The easiest way is to use conditional type parameters, by defining a top-level Boolean parameter (e.g. ***FxSim***) and using this parameter in the ***Configure Object*** dialog to set the actual type parameter values for an instance, e.g.:

Parameter Name	***Data Type***	***Value***
Tsample	***type_param***	***FxSim?sc_fix_fast(8,3,SC_RND, SC_WRAP):float***

Table 3: conditional type parameters for switching from fixed-point to floating-point simulation

Quantization of Intermediate Results

After all key objects have been quantized the application is almost bit-true. Remaining non-quantized intermediate results are e.g. division operators. There are also intermediate results, that are already bit-true, but using a larger word length than required, thus increasing the cost of an implementation. For most operations - apart from division - the word length (wl), integer word length (iwl), and fractional word length (fwl, with wl =iwl+fwl) can be derived that can hold every bit of the result (for signed a and b):

- *a ± b = c: iwl(c) = max(iwl(a),iwl(b)) +1; fwl(c) = max(fwl(a),fwl(b))*
- *a * b = c: iwl(c) = iwl(a)+iwl(b); fwl(c) = fwl(a) + fwl(b)*

For a division *a / b = c* the integer word length of *c* can be huge, if the values of *b* are very small, so it is hard to calculate the fixed-point format, even if the value ranges of a and b are known. The safest and most general way would be to split up an example expression from the **LoopFilter** like:

```
TmpPhase = TmpPhase – MaxValue* floor(TmpPhase/MaxValue + 0.5);
```

into two lines with an explicit intermediate result, using its own state variable (let's name it `phase_by_2pi`) that can be traced in System Studio simulations, so find out the range and integer word length (iwl) of the expression:

```
phase_by_2pi = TmpPhase/MaxValue;
TmpPhase = TmpPhase - MaxValue * floor(phase_by_2pi + 0.5);
```

After the *iwl* is known insert an explicit cast statement to limit the result of the division to a word length parameter *wl* and run simulations with different values for wl:

```
phase_by_2pi = sc_fix(TmpPhase/MaxValue ,wl,2);
TmpPhase = TmpPhase - MaxValue * floor(phase_by_2pi + 0.5);
```

In most cases this additional effort is not required though. In this example MaxValue is a constant (2π), just turn it into a multiplication with MaxValue^{-1}, and in addition the overall expression must have a similar word length and integer word length as the variable ***TmpPhase*** of type ***Tphase. Tphase*** has been already quantized, so it is assumed that using the same fixed-point type for the result of the division yields reasonable results:

```
Tphase phase_by_2pi = TmpPhase/MaxValue;
TmpPhase = TmpPhase - MaxValue * floor(phase_by_2pi + 0.5);
```

Such assumptions have to be verified with an additional simulation run, and in this example it turns out that in fact the BER does not change if the intermediate result uses the same fixed-point representation as the final result.

Similar refinements of intermediate results can also be done for additions, subtractions, and multiplications. The results of SystemC fixed-point operations are always carried by the container data type `sc_fxval` that can hold every bit of the operation. How many bits are really used can be calculated and then it can be tried to reduce the number of bits to reduce costs or reflect the available hardware resources. Taking the expression:

```
TmpPhase = TmpPhase - MaxValue * floor(phase_by_2pi + 0.5);
```

With ***Tphase*** = `sc_fix(8,4)` and ***T2pi*** = `sc_ufix(6,3)` can be rewritten to an equivalent statement using explicit fixed-point formats as:

```
TmpPhase -= sc_fix(MaxValue*sc_fix(4,0,phase_by_2pi,SC_RND,SC_WRAP),9,3);
```

Leave this explicit bit-true notation as the golden reference for the implementation or try to reduce the used word lengths even further. In additions/subtractions the same format for both operands can usually be picked, i.e. in this case try to reduce the word length of the second operand:

```
Tphase tmp_prod = MaxValue * sc_fix(4,0,phase_by_2pi,SC_RND,SC_WRAP);
TmpPhase = TmpPhase – tmp_prod;
```

Again, it has to be verified that reductions of the word length do not have any (significant) impact on the quality criterion such as the BER.

Benchmarking Results

In case the *fast fixed-point* optimization was not enabled as shown in Figure 6 for one of the fixed-point simulations a drastic difference in simulation speed will be noted between the floating-point simulation and the non-optimized SystemC implementation using the IEEE 1666 fixed-point types.

System Studio offers the fast fixed-point code optimization to overcome this difference. To quantify the advantage obtained from this optimization in System Studio, a benchmark suite was created comparing the complete carrier recovery simulation with 108 samples as floating-point version, including floating-point versions of the transmitter, channel, and receiver with the same simulation, but with a fixed-point version of the receiver.

For the benchmark a RedHat Enterprise Linux 5.3 machine with 64-bit Intel Xeon processors running at 2666 MHz was used.

	Floating-Point [sec (min:sec)]	*Fast Fixed-Point [sec (min:sec)]*	*Fixed-point [sec (h:min:sec)]*	
64-bit mode	*88 (1:28)*	*107 (1:47)*	*using sc_fix_fast types*	*3328 (0:55:28)*
			using sc_fix types	*5901 (1:38:21)*
32-bit mode	*107 (1:47)*	*184 (3:04)*	*using sc_fix_fast types*	*12664 (3:31:04)*
			using sc_fix types	*24691 (6:51:31)*

Table 4: Benchmark comparing floating-point, System Studio's fast fixed-point, and non-optimized fixed-point code

The results show that on a 64-bit machine System Studio's optimized fixed-point simulation runs by a factor of 31 faster than the same simulation using the original SystemC fixed-point types. It also runs only 22% slower than the floating-point reference, which is mainly due to the additional cast, shift, or rounding operations contained in the fixed-point code. In 32-bit mode the differences are even larger, i.e. the fast fixed-point code runs 69 times faster as the non-optimized fixed-point code, and 72% slower as the floating-point reference.

For completeness the unlimited-precision `sc_fix` types that can also be used for word lengths above 52 bits was also benchmarked. Though in this case, long words were not utilized, i.e. using the limited-precision `sc_fix_fast` types would be sufficient, this additional data point gives a good impression on the performance difference between these two types. Please note, that in System Studio's optimized fast fixed-point code all fixed-point types, limited and unlimited-precision, are mapped to integer types if the word length and integer word length are known at code generation time. Hence, the generated fast fixed-point simulation code in this example is the same, regardless whether limited or unlimited-precision types in the models are used.

These numbers impressively demonstrate the value of this simulation optimization. The simulation results for complex designs are obtained either after the lunch-break when using System Studio's fast fixed-point optimization or, in the non-optimized mode, only next day (at best).

SUMMARY

In this article the challenges of different fixed-point design flows were presented, such as using native integral data types of the host machine to represent fixed-point data, or to use dedicated fixed-point types along with their operations and casting modes. The advantages and disadvantages of these flows were discussed, and how System Studio's fast fixed-point optimization manages to combine the best of these two approaches to a solution that offers both, high simulation speed and excellent support for modeling and debugging was shown.

In a tutorial example, that is also available with System Studio's training material as reference, the key steps turning a floating-point algorithm into a fixed-point representation were illustrated. In addition, the tutorial section also contains useful tips on good practices or how to avoid typical pitfalls in fixed-point design.

Finally the results of the tutorial example were benchmarked to provide some quantitative data on the simulation speed advantage that System Studio offers, showing that System Studio's fast fixed-point simulation run by an impressive factor of 31-69 times faster compared to the data types of the reference IEEE 1666 SystemC library, and only 22-72% slower as the floating-point version of the same system. All this demonstrates how System Studio can help making the fixed-point design flow faster and more reliable and thus to avoid delays in projects.