2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION JAPAN

Quantization Methodology using Value Range Analysis

Shigetaka Nata, Siemens EDA Japan
Petri Solanti, Siemens EDA
SIEMENS

accellera SYSTEMS INITIATIVE

Agenda

What is Quantization

Chracteristics of Different Data Types

Quantization Process

Value Range Analysis

Catapult Value Range Analysis Feature

Validating Quantized Design

Conclusions

What is Quantization

The art of reducing data accuracy without losing algorithm performance

Quantization

Quantization is a process to reduce the data accuracy to the minimum, still reaching the algorithm performance requirements.

Usually, quantization means converting a floating point algorithm model to a fixed point model with optimal word lengths.

There are many comprehensive academic studies introducing different approaches to automate quantization process. Most of them are too complicated for normal project work.

Quantization methodologies are usually divided into two categories:

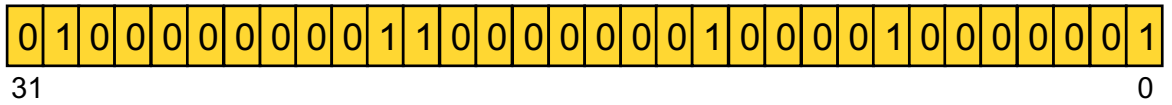
- Simulation based (dynamic) methods
- Analytical (static) methods

Characteristics of Different Data Types

Standard data types

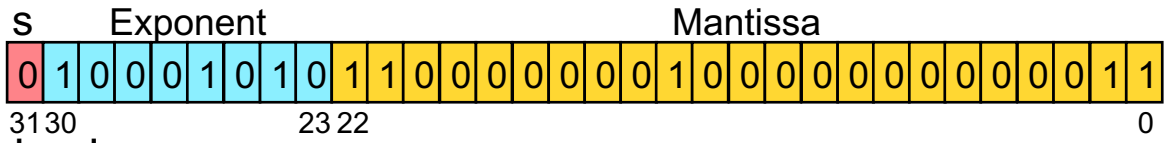
Whole number (integer) types

- Represents numbers without decimal part
- Implemented as a bit vector
- Constant step size throughout the value range
- No automatic saturation, in overflow case wrap around



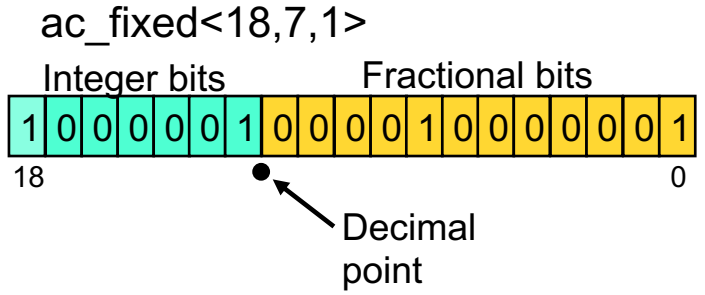
Floating-point types

- Represents both integer and decimal parts of numerical value
- Implemented as a struct with sign, exponent and mantissa
- Variable step size depending on exponent value
- Rounding scheme



Fixed-point types

- Represents both integer and decimal part of numerical value
- Implemented like a bit vector with a virtual decimal point
- Step size constant 2^{-frac_bits}
- Rounding and saturation schemes parameterizable



Quantization Process

Quantization process

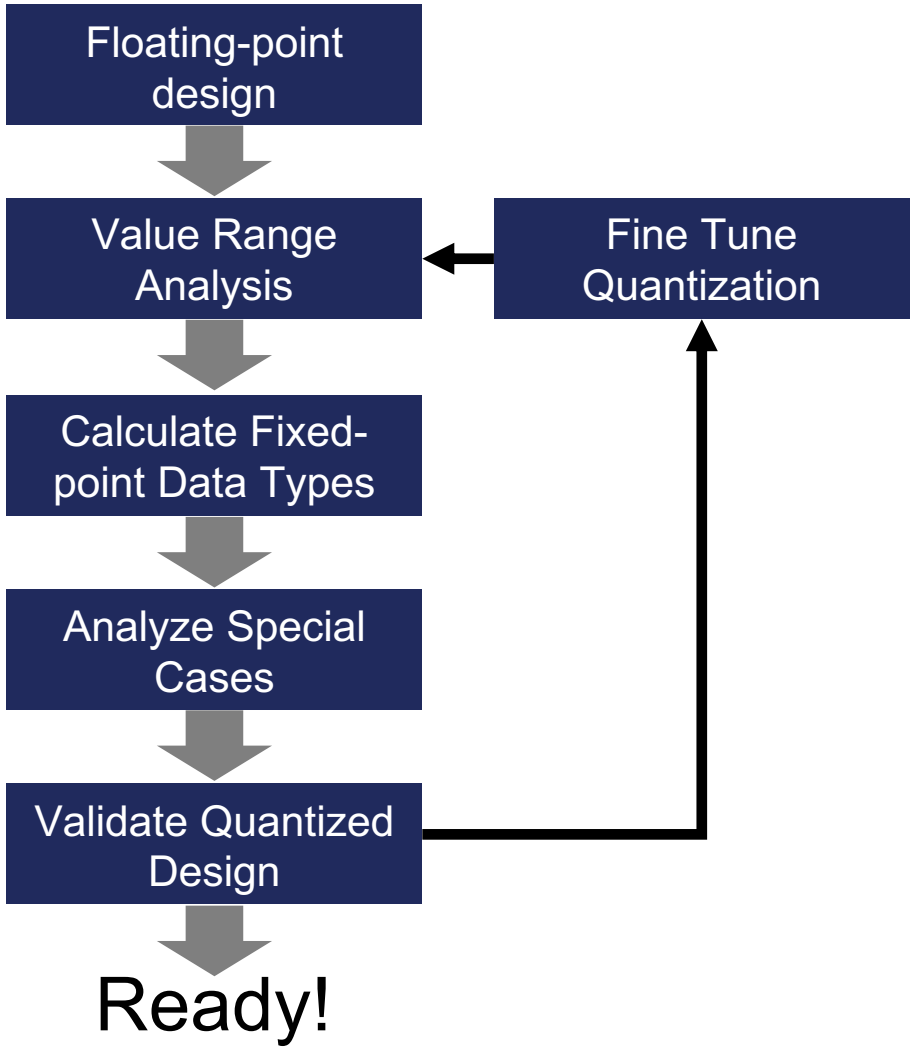
Quantization is an iterative process with multiple steps

Simulation based quantization

- Value range analysis of all signals and variables in the design
- Calculating fixed-point data types based on signal values
- Analyzing special cases (constants, coefficients, etc.)
- Validating quantized design
- Fine tuning quantization

Analytical Method

- Analyzing input and output data types
- Calculating word length based on arithmetic operations
- Validating quantized design
- Fine tuning quantization



Quantization process phases

Quantization process has three disconnected phases:

- Pre-quantization
- Coarse quantization
- Bit utilization analysis

Pre-quantization

- Defining word lengths and value ranges of inputs and outputs
- Easy to do with pseudo-quantization in simulation

Coarse quantization

- Define all internal bit widths using value range analysis method
- Provides a reliable number of integer bits, but only coarse estimate of fractional bits

Bit utilization analysis

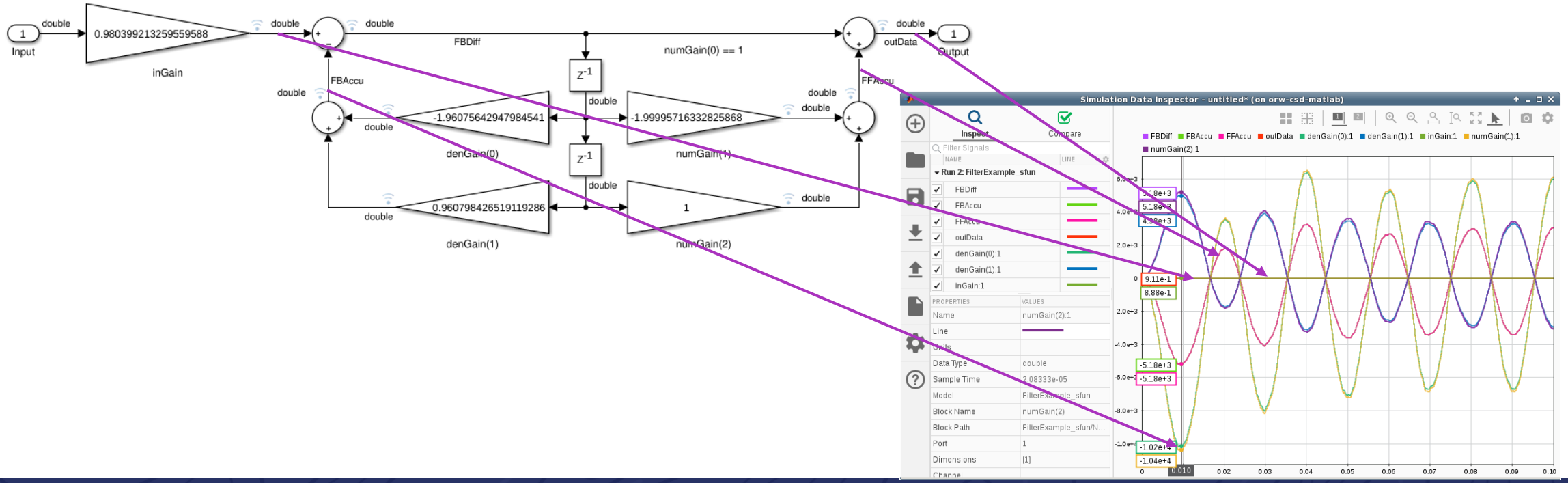
- Bit range analysis (overflow/underflow detection)
- Fixed-point bit utilization analysis (unused bits)

Why helper tools are needed

Value ranges in the design are not always predictable

- Feedback architectures may have 10.000x signal level differences
- Differences are dynamic based on e.g. signal frequency, cumulative offset, etc.

Value range analysis with floating-point or wide range fixed-point types is necessary



Value Range Analysis

Value Range Analysis methods

Dynamic (simulation based) methods

- Collect value range data of all assignments during simulation
- Store all assigned values for each variable into a vector and post process it later
- Gives reliable data for all nodes in the complete design
- Requires instrumentation of the model
- QoR depends heavily on the stimulus quality

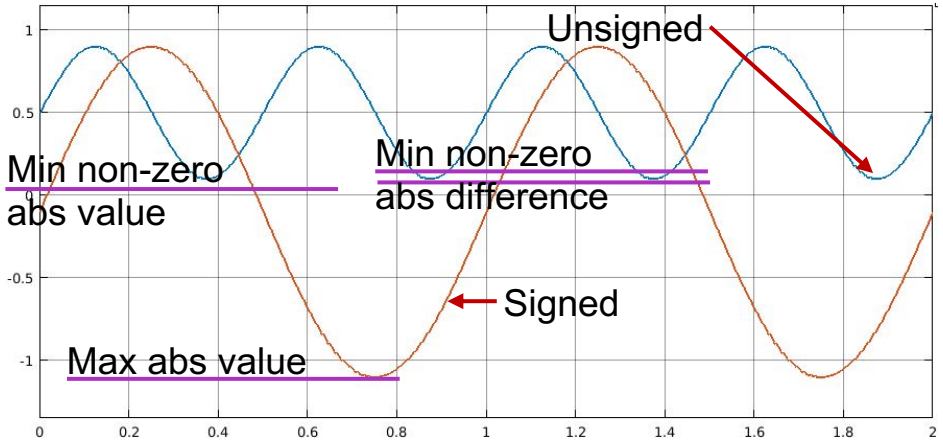
Static (analytical) methods

- Analyze values based on input value range → Pre-quantization required
- Calculate max and min values based on behavior of quantized signals in arithmetic operations
- Not suitable for analyzing a complex design
- Error prone in designs with feedback, e.g. IIR filter
- Works well for feed-forward type systems, e.g. FIR filter

What are the relevant metrics?

To determine the required number of integer and fractional bits in the fixed-point type we need

- Maximum absolute value of the signal assigned to a variable
  → Number of integer bits required to handle the value without overflow
- Minimum non-zero absolute value
  → Minimum value to set a fraction bit
- Minimum non-zero difference between two samples
  → Minimum value that toggles a bit
- Signedness of the signal
  → Adds one bit to the integer part



With these metrics the fixed-point analysis provides attributes for almost lossless implementation

Final quantization requires careful analysis of allowed noise figure and potential saturation

Simulation-based Value Range Analysis

Analysis process should begin with a floating-point or wide range fixed-point (e.g. 64-bit) types

1. Pre-quantization of inputs and outputs using **Pseudo-Quantization Method**
2. In large hierarchical designs, pre-quantization of the major hierarchical blocks individually
3. Analysis of algorithm performance with quantized I/O
4. Instrumentation of all assignments in the design
5. Simulation of instrumented model
6. Max/min value analysis of the simulation data

Pre-quantization limits the step size of the input signal and gives better decimal bit estimates, e.g.

- Unquantized input signal results in one node a minimum difference of 1.26175e-04 → 12 decimal bits
- Input signal quantized to 6 decimal bits results in the same node diff 0.0153187 → 6 decimal bits

Static Value Range Analysis

Calculating values in the output of an operation based on known input values.

- Assigning max and min values to operands and calculating output values through the signal chain
- Inputs and outputs must have limited bit widths

Example: Accumulator in 32 tap FIR filter : Input<16,2,true>, Coeff<16,0,true>, Output<16,2,true>

- Maximum absolute value:
  - Calculate sum of absolute values of the coefficients = 1.3657
  - Multiply the sum by maximum input value
  - Result is the maximum worst case value the accumulator can reach = 2.7314 signed → 3 integer bits
- Minimum non-zero value
  - Multiplier output has automatic type of <32,2,true>, which has 30 fractional bits
  - Output is reduced to <16,2,true> → 14 of 30 fractional bits will be used: 2^{-14} = 0,000061035
  - Multiplier output is truncated
    → uniform distributed error (0.5 bits) accumulates 32 times => 32 * 0.5 * 2^{-14} = 0,00097656
  - To reach resolution of 14 fractional bits in output, accumulator must represent 2^{-14} / 16 = 0,00000381 → 19 fractional bits

Value Range Analysis Feature

New Catapult Value Range Analysis feature

Value Range Analysis feature integrated in ac_fixed data type since 2023.1 release

Analyzes multiple metrics during simulation execution

- Minimum and maximum values
- Signedness
- Minimum absolute fractional value
- Activation count (number of assignments to the variable)
- Overflow count
- Minimum quantization error

Feature must be activated with compiler flag **`-DAC_FIXED_VRA`**

```
$ g++ -g -std=c++11 -DAC_FIXED_VRA -I$MGC_HOME/shared/include -I.
<filename>.cpp -lbfd
```

Report for each variable is printed to stdout at the end of the simulation

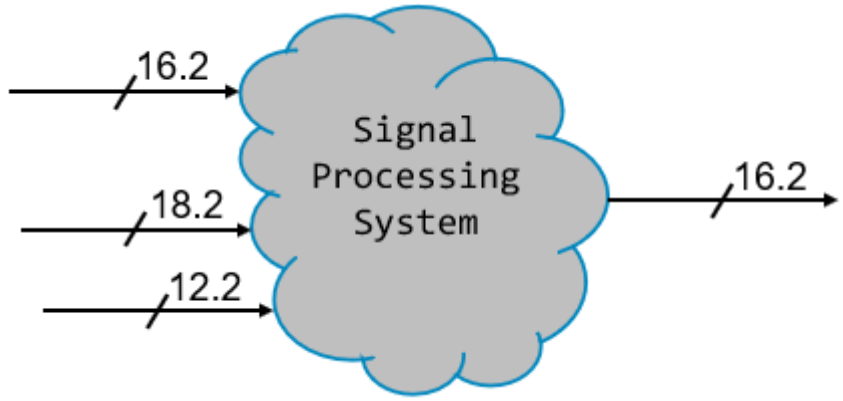
Report can be directed to .csv file by setting an environment variable

```
$ setenv AC_FIXED_VRA_OPTS '-f<filename>'
```

Preparing design for initial value range analysis

Pseudo-quantize all input signals to DUT

- Bitwidth analysis should be done by algorithm developers
  - Changes the functional characteristics of the system
  - Must fulfill system-level requirements
- Usually done in the testbench with local variables, the values of which are assigned to the DUT input variables



Quantize the DUT output variable to the specified data type

- Needed for loss-of-precision and overflow analysis

Declare all internal variables to something big enough

- **`ac_fixed<64,32,true>`** is a good starting point
- If peak values are known, number of integer bits can be reduced

```
typedef ac_fixed<64,32,true> iir_coeff_t;
typedef ac_fixed<64,32,true> iir_in_t;
typedef ac_fixed<64,32,true> iir_accu_t;
typedef ac_fixed<IIR_OUT_WIDTH, IIR_OUT_INT_BITS, true, AC_TRN_ZERO, AC_SAT> iir_out_rs_t;
typedef ac_fixed<64,32,true> iir_out_t;
```

DUT internal variable types

Catapult Value Range Analysis report

Value Range Analysis report contains a separate data segment for each variable

- Original declaration and value range it can handle
- Observed value range and minimum fractional value
- Signedness
- Fractional bits range
- Value change count
- Overflow count
- Minimum quantization error and RMS
- Proposed new type declaration
- File name and line number of variable declaration
- Call stack

```
------------------------------------------------------------
DECLARATION: ac_fixed<16,2,true,AC_TRN,AC_WRAP>
  Available Value Range (min -2 : max 1.99994)
  Observed Value Range  (min -1.71369 : max 1.71387 : min_frac 0.000210514)
  Signed                = true
  Fractional bits       = 13..18
  Value Change Count    = 1000
  Overflow Count        = 0
  Smallest Quant Err    = 0.000001
  RMS Quant Err         = 0.000000
  Modified declaration  = ac_fixed<15,2,true,AC_TRN,AC_WRAP>
  CALL STACK:
    tb_iir_filter.cpp:54 'main'
 /home/projects/Quantization/src_IIR_demo/tb_iir_filter.cpp:54
------------------------------------------------------------
```

Analyzing the results

Original declaration of interface variables is **`ac_fixed<16,2,true>`**

- Modified declaration should have the same. Otherwise, the stimulus or quantization may not be correct.

Value change count should be a large number. Small value indicates inaccurate measurement.

Overflow should not take place in the initial simulation with **`ac_fixed<64,32,true>`**

Variables that belong logically together, can be grouped to the same type.

- IIR gain parameters and tmpFF*/FB* variables. The largest word length should be used.

Number of calculated integer bits is the second template parameter of modified declaration

Number of calculated fractional bits is the first template parameter minus number of integer bits.

Declaration	Location	Variable	Value Change Count	Overflow Count	Modified Declaration	Allowed Min	Allowed Max	Min	Max	MinFrac	Signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	tb_iir_filterconst_blk:38	IIR_DenGain[2	2	0	ac_fixed<7,2,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-1,96077	0,960785	0,0392303	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	tb_iir_filterconst_blk:39	IIR_NumGain[	3	0	ac_fixed<17,2,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-1,99997	1	3,05E-05	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	tb_iir_filterconst_blk:42	IIR_InGain	1	0	ac_fixed<6,0,false,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	0,980392	0,980392	0,0196075	unsigned
ac_fixed<16,2,true,AC_TRN,AC_WRAP>	tb_iir_filterconst_blk:54	inData	1000	0	ac_fixed<15,2,true,AC_TRN,AC_WRAP>	-2	1,99994	-1,71369	1,71387	0,000210514	signed
ac_fixed<16,2,true,AC_TRN,AC_WRAP>	tb_iir_filterconst_blk:60	outData	1000	0	ac_fixed<15,2,true,AC_TRN,AC_WRAP>	-2	1,99994	-0,7688	1,01282	0,000183105	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	IIR_class.h:28	delayLine[2]	2002	0	ac_fixed<25,14,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-5863,2	6101,95	0,000687914	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	IIR_class.h:42	inData	1000	0	ac_fixed<14,2,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-1,71375	1,71387	0,000244141	signed
ac_fixed<16,2,true,AC_TRN_ZERO,AC_SAT>	IIR_class.h:44	outData	1000	0	ac_fixed<15,2,true,AC_TRN_ZERO,AC_SAT	-2	1,99994	-0,76883	1,01286	0,000218097	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	IIR_class.h:45	tmpFBAccu	2000	0	ac_fixed<27,14,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-6102,44	5862,66	0,000192473	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	IIR_class.h:46	tmpFFAccu	2000	0	ac_fixed<29,14,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-6102,37	6101,95	3,93E-05	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	IIR_class.h:47	tmpFBDiff	2000	0	ac_fixed<25,14,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-5863,2	6101,95	0,000687914	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	IIR_class.h:48	tmpInGainOut	1000	0	ac_fixed<15,2,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-1,68014	1,68026	0,000198969	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	IIR_class.h:49	tmpFBGainOu	2000	0	ac_fixed<29,15,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-11964,5	5862,66	6,22E-05	signed
ac_fixed<64,32,true,AC_TRN,AC_WRAP>	IIR_class.h:52	tmpFFGainOu	2000	0	ac_fixed<28,15,true,AC_TRN,AC_WRAP>	-2,15E+09	2,15E+09	-12203,7	6101,95	0,000231935	signed

Simulating with modified types and analyzing results

When the variable declarations are modified, the design can be analyzed with simulation again.

Overflow count indicates that the variable doesn‘t have enough integer bits.

If modified declaration has less fractional bits that original, the feeding variable may be too short

- outData should have 14 fractional bits, but in modified declaration it has only 12
- outData = tmpFBDiff + tmpFFAccu;
- Both variables have only 13 fractional bits. 15 bits are needed to deliver 14 fractional bits with rounding.

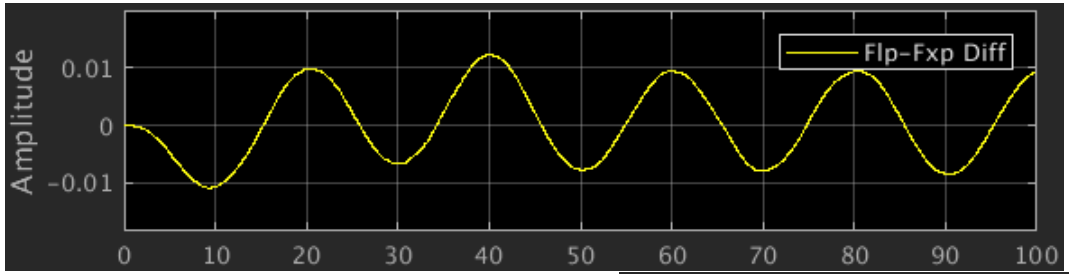
Declaration	Location	Variable	Value Change Count	Overflow Count	Modified Declaration
ac_fixed<18,2,true,AC_TRN,AC_WRAP>	tb_iir_filter.cpp:38	IIR_DenGain[2]	2	0	ac_fixed<7,2,true,AC_TRN,AC_WRAP>
ac_fixed<18,2,true,AC_TRN,AC_WRAP>	tb_iir_filter.cpp:39	IIR_NumGain[3]	3	0	ac_fixed<17,2,true,AC_TRN,AC_WRAP>
ac_fixed<18,2,true,AC_TRN,AC_WRAP>	tb_iir_filter.cpp:42	IIR_InGain	1	0	ac_fixed<6,0,false,AC_TRN,AC_WRAP>
ac_fixed<16,2,true,AC_TRN,AC_WRAP>	tb_iir_filter.cpp:54	inData	1000	0	ac_fixed<15,2,true,AC_TRN,AC_WRAP>
ac_fixed<16,2,true,AC_TRN,AC_WRAP>	tb_iir_filter.cpp:60	outData	1000	0	ac_fixed<14,2,true,AC_TRN,AC_WRAP>
ac_fixed<27,14,true,AC_TRN,AC_WRAP>	IIR_class.h:28	delayLine[2]	2002	0	ac_fixed<27,14,true,AC_TRN,AC_WRAP>
ac_fixed<16,2,true,AC_TRN,AC_WRAP>	IIR_class.h:42	inData	1000	0	ac_fixed<15,2,true,AC_TRN,AC_WRAP>
ac_fixed<16,2,true,AC_TRN_ZERO,AC_SAT>	IIR_class.h:44	outData	1000	0	ac_fixed<14,2,true,AC_TRN_ZERO,AC_SAT>
ac_fixed<27,14,true,AC_TRN,AC_WRAP>	IIR_class.h:45	tmpFBAccu	2000	356	ac_fixed<28,15,true,AC_TRN,AC_WRAP>
ac_fixed<27,14,true,AC_TRN,AC_WRAP>	IIR_class.h:46	tmpFFAccu	2000	364	ac_fixed<28,15,true,AC_TRN,AC_WRAP>
ac_fixed<27,14,true,AC_TRN,AC_WRAP>	IIR_class.h:47	tmpFBDiff	2000	0	ac_fixed<27,14,true,AC_TRN,AC_WRAP>
ac_fixed<16,2,true,AC_TRN,AC_WRAP>	IIR_class.h:48	tmpInGainOut	1000	0	ac_fixed<15,2,true,AC_TRN,AC_WRAP>
ac_fixed<27,14,true,AC_TRN,AC_WRAP>	IIR_class.h:49	tmpFBGainOut	2000	356	ac_fixed<28,15,true,AC_TRN,AC_WRAP>
ac_fixed<27,14,true,AC_TRN,AC_WRAP>	IIR_class.h:52	tmpFFGainOut	2000	364	ac_fixed<28,15,true,AC_TRN,AC_WRAP>

Validating quantized design by simulation

Validating algorithm against the original floating-point model

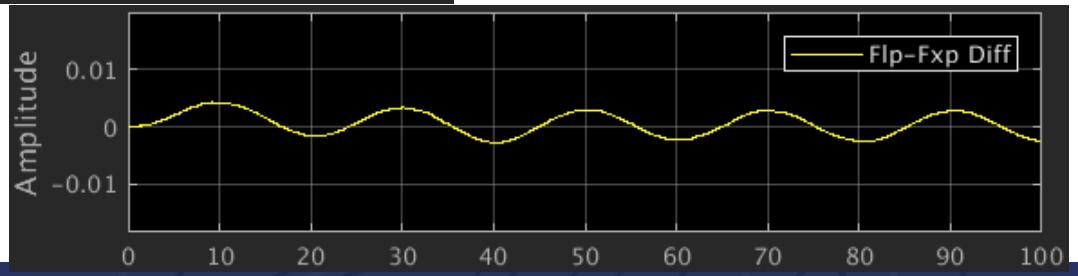
- Quantization effects can be seen in simulation results, e.g.
  - Signal-to-Noise ratio
  - Bit error rate
  - Difference between floating-point and quantized outputs
- Algorithm may not work in certain circumstances

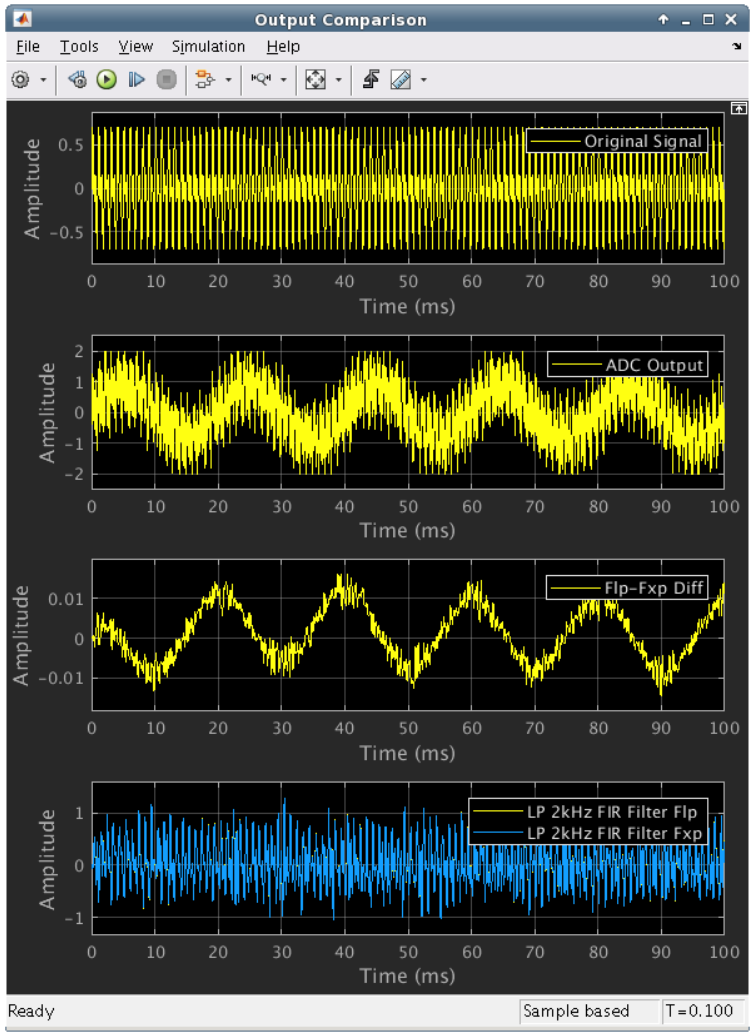
This step can be used to optimize constant input word lengths, e.g. coefficients



18-bit IIR coefficients

21-bit IIR coefficients





Conclusions

Conclusions

Quantization is a crucial part of hardware design process.

- Optimization of design area and power consumption
- Managing noise level, offset and overflow behavior of the algorithm implementation

Value Range Analysis based quantization methodology with static analysis of special cases is a simple and robust way to analyze the needed fixed-point word lengths for any types of digital designs

Catapult Value Range Analysis feature makes quantization easy

- Automatic instrumentation of all fixed-point variables
- Captures quantization information during simulation run and proposes fixed-point datatype for each variable individually
- Analyzes required number of integer bits and a range for number of fractional bits

Fixed-point effects must still be analyzed with the original testbench to ensure the algorithmic performance of the fixed-point design

Questions?