We are IntechOpen,
the world's leading publisher of
Open Access books
Built by scientists, for scientists

6,900
Open access books available

186,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS

BOOK CITATION INDEX

INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Advanced Decimator Modeling with a HDL Conversion in Mind

Drago Strle and Dušan Raič

Additional information is available at the end of the chapter

http://dx.doi.org/10.5772/58242

1. Introduction

Today, many systems designers use software tools such as Matlab to model complex, mixed-technology systems prior to physically building and testing the system. These tools, along with their associated toolboxes provide an effective means for the initial modeling and simulation stages in a project. Such software tools also provide the means to extract information in a relevant format to aid the physical realization [see 1-3]. In this chapter we will describe the use of Simulink and its library blocks in conjunction with the HDL Coder tool in order to produce a HDL representation of the model that is, suitable for synthesis into digital logic hardware for implementation on devices such as FPGA and ASICs. We follow the idea of one single model, starting from the system level simulation to the final system integration and hardware implementation, possibly eliminating any designer interventions on the low (RTL) level. This aim can be achieved with the right concept right at the beginning of top level modeling, based on the knowledge of the tool background and its limitations.

Our presentation starts with a simple example, using an existing block from the DSP System Toolbox Library [4]. We then proceed with the redesign of the same block in order to gain more control over the implementation details that will allow fulfilling specific requirements that might be needed for system integration. The presentation is accompanied with practical Simulink and HDL Coder tips that we believe will help the reader to reproduce and possibly upgrade the presented work. The two designs are first compared for equivalence in the Simulink environment, and then again with the circuit simulation of the compiled and synthesized hardware. Finally, some proposals are given to upgrade models with more advanced features targeting the high-precision systems.

2. Modeling with a HDL coder in mind

The first surprise that a conventional digital designer is faced with when switching to the Simulink HDL tool is the lack of flip-flops and the associated clock and reset signals. Although there are ways to model the flip-flop functionality (e.g. D Latch, D Flip-Flop and J-K Flip-Flop blocks in the 'Simulink Extras/Flip Flops' library), this is not the right way to go since these models cannot be converted by the HDL coder tool. The answer lies in the high-level modeling approach that is intentionally made free of the low-level circuit details. It must be pointed out that Simulink models are basically graphical interfaces to mathematical operations [5]. Although block diagrams seem to represent connections of physical entities by electric wires, this is not the case since the signals represent mathematical variables. However, a close analogy makes it possible to translate the Simulink models into an equivalent hardware description, where signals turn to wires and blocks turn to library cells.

A HDL coder [6] inserts low-level control signals automatically, e.g. the system clock, the power-on reset, and the clock enable signals are not expected as part of the Simulink model. High-level modeling is released from these details, letting the designer concentrate on system-level considerations. However, the lack of controlling signals at the system model level may introduce some problems. For example, the lack of the output data enable signal on the model level prevents modeling of pipelined structures and data synchronization with other blocks in the system.

2.1. Single-rate vs. multi-rate model implementation

By definition, the CIC decimation filter that we use in our design example is a multi-rate system since it transforms the high-rate input signal into the low-rate output signal so that the decimation rate factor R is a constant, positive integer. The Simulink HDL Coder automatically creates all timing signals to drive the multi-rate system. However, the merging of the compiled code with the rest of the system is much easier and requires less hardware if all system components are based on a common, single base-rate clock. Sharing data between blocks with the appropriate handshaking details may become very complicated if the automatically generated multi-rate clock signals are to be matched with other parts of the system. From a system integration point of view it is therefore advantageous to model our 'demo' design filter as a single-rate system.

In Simulink the multi-rate systems can be easily modeled as enabled, single-rate systems by replacing the Unit Delay block with the Enabled Unit Delay block. By doing so the automatic handling of different sampling times becomes visible and accessible to the designer in the form of various 'enable' signals, added to the block diagrams. There is of course also a drawback with this approach since the block diagram gets more populated with additional signals, prone to design errors, and therefore is more difficult to maintain. Nevertheless, the single-rate modeling brings other advantages, like:

- The possibility to create programmable sample rate,

- Pipeline latency control with base-rate delays,

- Full control over the timing controller.

The automatically generated timing controller (e.g. tc.vhd) of the multi-rate model is built with a number of counters and registers that may be avoided in the centralized, system-wide timing, resulting in lower area and power. In the case of our decimating filter example we will follow the single-rate approach by creating the customized down-sampling block with the integrated down-sampling counter, which will replace the HDL Coder-generated timing controller of the multi-rate version.

Another rather important feature of Simulink models is the fact that pipelining options of the HDL block implementation cannot be simulated. Like timing control signals, the pipelining hardware is generated during the process of HDL compilation, which is separated from the simulation. At this point the Simulink operation is inconsistent; obviously, the pipelined functionality of HDL blocks is present in the data-base, otherwise the testbench generation would not be possible. On the other hand, the simulator ignores the inserted pipelines: the resulting signals cannot be viewed on the Scope or stored to the data file, preventing the application of powerful data analysis tools at the model level. The simulation of pipelined structures is made possible only after the compilation, using other tools in the design loop. From the point of view of consistent Simulink/HDL modeling it is therefore better to bring pipelining into the Simulink model and forget about HDL block implementation options. The latter consideration is one more reason to choose single-rate over multi-rate modeling.

2.2. Using library blocks in the the HDL Coder environment

Only a limited number of blocks from the large Simulink library are supported by HDL Coder. To get an overview of these blocks, enter `hdllib` from the Matlab prompt and create *HDL Demo library*; save the library for later reference and investigation. For example, in Bit Operations we find useful operators like Bit Concat, Bit Reduce, Bit Rotate, Bit Shift, and Bit Slice. Another interesting block is the generic counter model, suitable for HDL coder compilation. It may be investigated with the right-click on the HDL Counter block, selecting the 'Look under the mask' option.

A number of useful operators can be also found in other libraries. However, some blocks that seem to suit the logic design cannot be used with the HDL coder. The Pulse, Multiphase clock, Integer to Bit Converter blocks, for example, cannot be used. The Delay block must be replaced by the Unit Delay block. Some limitations apply in other blocks, e.g.:

- Trigger block: 'Show output port' must be OFF; the Trigger port must be driven by registered logic; outputs of the triggered system must have an initial value of '0'; the trigger type 'either' is not allowed – use 'rising' or 'falling'.

- Divide: use Product block from the Math Operations Library; rounding must be Zero or Simplest; saturation must be ON.

- Data Type Conversion: Double-> Fixed-point conversion is not possible.

Generally, the signed integers should be avoided and operations should be carried out on unsigned integers when logic operations are to be performed on VHDL standard logic vectors.

The key elements for hardware design are logic gates and registers. Registers must be modeled by the Unit Delay block from the 'Discrete' library. The Delay block (with the parameterized

number of delay cycles) cannot be used since its data structure is not supported for HDL conversion. Logic gates are modeled with blocks from the 'Logic and Bit operations' library. In the following we present some comments above the library blocks that will be most likely used in designs, intended for the HDL conversion.

To Workspace block: The best way to compare the logic simulation results of a reference design with the Simulink results is to store Simulink signals in the Matlab workspace. The recorded signal must be given a suitable name and the save format should be set to 'Array'. Arbitrary signal values at selected times can be printed out in the workspace using the Matlab expression signame(time_index). Before printouts, do not forget to apply the 'formal long' command to get full precision data.

Extract Bits block: Most frequently only one bit (MSB or LSB) is to be extracted. In such cases the best way is to specify bits to extract from the 'Range starting with MSB' or 'Range ending with LSB' and set 'Number of bits' to 1. The output scaling mode should be set to 'Treat bit field as an integer'. With these settings the output bit will be scaled to ufix1 which is suitable for more logic operations or for the conversion to the Boolean data format. Other bit positions or ranges can be extracted as well.

Rate Transition block is supported for HDL conversion only if both Data Integrity and Deterministic Data Transfer options are ensured. Consequently, the transition from the Downsample output data rate back to the base-rate will introduce additional output delay of one low-rate period. This means that the Downsample block output must be registered with the output-rate clock before being forwarded to any control blocks, like end-of-conversion pulse generation or output data handshaking. Therefore, these procedures can be only performed with considerable delay that may not be acceptable.

Downsample block: The output data rate is equal to its input data rate divided by the downsample factor K (also frequently referred to as the decimation factor R). The sample offset, D, which must be an integer in the range [0, K–1] allows the definition of output delay. The sample offset is specified in the output data rate period units. Therefore, the downsample output delay cannot be specified in base-rate period units.

CIC Decimation block: This block is based on three parameters: decimation factor (R), differential delay (M) and the number of sections (N). It allows three options for data type specification: user-defined register and/or fraction lengths, maximum precision and register pruning according to the desired output register length. The input word length is automatically determined from the block diagram. The model does not provide an 'output enable' pulse. The timing block is automatically generated (not visible at the model level).

Arithmetic Shift block: Input S in not supported for HDL; you must use the dialog box. The block ignores the Overflow and Rounding modes; it also does not check overflow or underflow. Right shift of unsigned numbers shifts '0' into the MSB position, while in the signed numbers the MSB bit is preserved. The numeric type of output variable is equal to the numeric type of the input variable.

The understanding of the Arithmetic Shift block is important since it can be used for register pruning in CIC Decimation filters. The HDL coder handles arithmetic shifts by automatic

adjustment of numeric data types. For example, if the fixed point variable of type sfix23 is pruned by the right shift for 3 bit positions the resulting data type will be set to sfix20_E3.

When dealing with scaled numbers in block diagrams, attention should be paid since shifting is implemented independently from scaling. Let us take as an example the number 162874 represented as fixed point data type fixdt(1,16,-7). The fraction '-7' means that the original value has been pruned by right-shifting for 7 bit positions, i.e. multiplied by $2^{\wedge}(-7)$ in order to store the approximated magnitude as a 16-bit signed integer. After shifting the stored value is thus equal to bitshift(162874,-7)=1272. The approximated value, reconstructed by the scaling factor $2^{\wedge}7$, is therefore equal to 1272*128=162816. If this scaled number is now, for example, shifted right by 6 more bit positions, the stored value would be bitshift(1272,-6)=19, representing the scaled value $19*(2^{\wedge}6)*(2^{\wedge}7)=155648$. The result is therefore not equal to 162874 or to 162816 as one might expect.
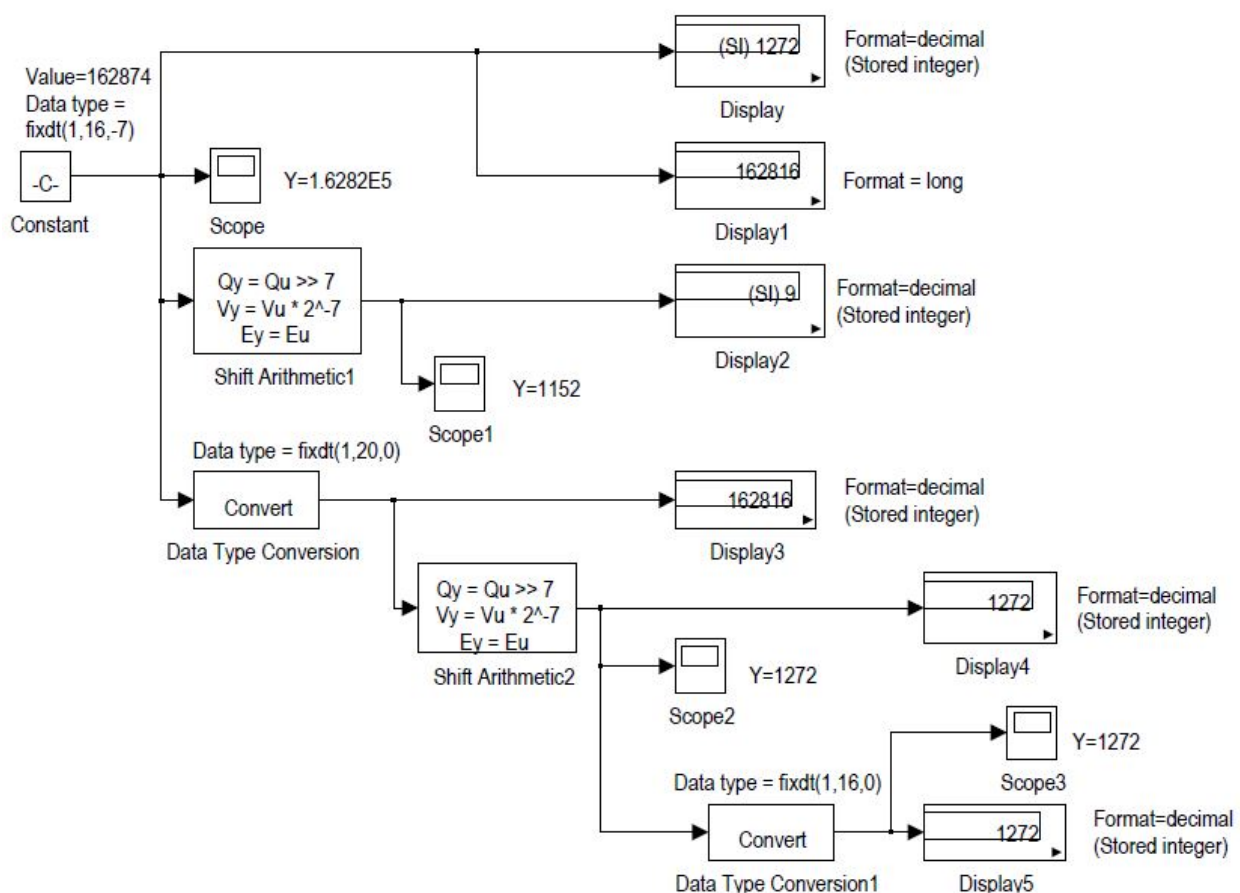

Figure 1. Scaling example.

A number of format options in the Display block allow presenting the signal also as the 'Stored integer' value. Conversely, the Scope block is based on the built-in 5-digit floating decimal point precision without this option, so one has to construct it from the (scaled) input value. Since shifting is independent from scaling we have to remove the scaling before we re-apply

it again to display the stored integer value. The Data Conversion block has to convert the input value into integer format without fraction, so the word length must accommodate the largest possible value. If we need the initial word length, one more data conversion to fraction-less data type is needed after shifting. The block diagram in Figure 1 illustrates the wrong (Scope1) and the correct methods (Scope2 and Scope3) to display the stored number of the above example.

2.3. HDL coder tips

General & model-related tips:

- If HDL Coder option is not visible try to run

```
hdlsetup('toplevel_model_name')
```

from the Matlab prompt.

- Use the 'ver' command to verify installed software tool components.

- Minimize sampling time definitions as much as possible; most blocks should use Ts=-1. Limit sampling time definitions to inputs and downsampling blocks. Rate transitions must be specified by Signal Attributes/Rate Transition or Signal Operations blocks (e.g. Down-sample). Delay, Unit Delay and Zero-Order Hold blocks cannot change the sample rate.

- Format models to display the sampling time and port data types. HDL conversion requires that model sampling times and data types are consolidated; to display sampling times, use: Format->Sample Time display-> Annotations (colors). To display data types, use Format->Port/Signal Displays->Port Data Types. Finally, use Edit-> Update model to see conflicting definitions.

- Use meaningful signal names. To improve node traceability in the compiled code it is advisable to replace the automatically indexed block names with meaningful node names (e.g. rename the block 'Unit Delay 2' to 'IIR_reg3').

HDL coder keeps Inport names on input signals. User-defined names of signals connected to Inports are ignored. However, signal names of signals connected to Outports may be user-defined.

HDL coder keeps signal names defined in the library blocks (the lowest level); if a number of identical blocks is invoked the names are appended with _x. Toplevel signal names are ignored. In order to allow user-defined signal names in the toplevel block diagram the custom library blocks should not define any signal names.

HDL usage restrictions & implementation details

- The HDL Coder requires Fixed-Point Toolbox

- Maximum word size for fixed-point numbers is 32 bits (Simulink limitation). The maximum number of vector elements is 2^32.

- Multitasking is not supported; the model must be configured for Single-Tasking. Single-rate and multi-rate models are supported within the single-tasking operation.

- The HDL Coder requires equal sampling time between each two blocks. The Rate Transition block (Signal Attributes Library) must be applied when this condition is not met.

- Switch, Manual Switch, and Multiport Switch (Signal Routing Library) cannot select between signals operating at different sample rates. Signals must be brought to (the fastest) common sampling rate, using the Rate Transition block, which may introduce additional registers and timing delay.

- An enabled subsystem may not represent the toplevel for HDL code generation. Outputs in an enabled subsystem are coded so that they come from the bypass selector with the select input connected to the Enable port.

Fixed point arithmetic

Multiplication or division is always performed on fixed-point variables that have zero biases. If an input has a nonzero bias, it is converted to binary-point-only scaling before the operation. If the result is to have a nonzero bias, the operation is first performed with temporary variables that have binary-point-only scaling. The result is then converted to the data type and scaling of the final output.

There are three fixed-point scaling expressions in Simulink:

- Unspecified scaling: fixdt(signed, word_length)

- Slope& bias scaling: fixdt(signed, word_length, slope, bias)

- Bin.point-only scaling: fixdt(signed, word_length, fract_length)

The binary-poin-only scaling should be used for the HDL coder. Matlab uses fixed-point structure coding in the form

- sfix'no.of_bits'_En'negative_fixed_exponent', or

- sfix'no.of_bits'_E'positive_fixed_exponent'.

Attention must be paid when Matlab structure coding is translated into Simulink expressions, since the negative fixed exponent corresponds to the fraction length (shift right to get integer result) and positive fixed exponent corresponds to the negative fraction length (shift left to get integer result). For example:

- fixdt(1,16,8)==sfix16_En8 (signed, 16 bits word length, 8 bits fract. length)

- fixdt(1,52,-25)==sfix52_E25 (signed, 52 bits word,-25 bits fraction length)

(a negative fraction means that the result has to be multiplied by 2^25)

Matlab always works in double precision (unless you are using the Symbolic Math Toolbox), but output display can be changed with the "format" command. This is useful when we are checking testbench results from the Matlab prompt. FORMAT may be used to switch between different output display formats as follows:

```
FORMAT            Default. Same as SHORT.
FORMAT SHORT      Scaled fixed point format with 5 digits.
FORMAT LONG       Scaled fixed point format with 15 digits.
FORMAT SHORT E    Floating point format with 5 digits.
FORMAT LONG E     Floating point format with 15 digits.
FORMAT SHORT G    Best of fixed or floating point with 5 digits.
FORMAT LONG G     Best of fixed or floating point with 15 digits.
FORMAT HEX        Hexadecimal format.
FORMAT RAT        Approximation by ratio of small integers.
FORMAT COMPACT    Suppress extra line-feeds.
```

HDL testbench generation

In order to generate the testbench code we need a toplevel testbench block diagram consisting of the stimulus generator and DUT. The testbench compiler runs the simulation and stores input signals into the appropriate data structure; additionally it generates assertions to check DUT output signals. The HDL Coder/Test Bench option is therefore disabled (gray-out) if the testbench block diagram does not define the DUT input and output signals. The HDL code for DUT is generated separately from the testbench code. Attention must be paid to the Solver Stop time to consider the necessary simulation_cycles; long simulation times generate large input/output data tables that may need prohibitive computer time to be loaded in the logic simulator.

```
sim_cycles=(Stop_time-Start_time)/Simulink_base_rate
simulated_time=sim_cycles * testbench_clock_period
```

The testbench code begins with two packages

```
PACKAGE *_tb_pkg IS... functions & procedures... END *_tb_pkg;
PACKAGE BODY *_tb_pkg IS... END *_tb_pkg;
PACKAGE *_tb_data IS... constants... END *_tb_data;
PACKAGE BODY *_tb_data IS... END *_tb_data;
```

Architecture rtl begins with toplevel component definition, followed by the Component Configuration Statements. Pay attention when compiling the testbench with the DUT synthesized (Verilog) netlist. As Verilog is case sensitive you may get the compilation error

```
Port "x" is declared in component "xx" but is not a formal port in entity "xx"
```

The message is reported because the case sensitive Verilog netlist ports are not found. In this case it might help disabling the Component Configuration Statement of the DUT netlist.

Reset cycles are added automatically to the simulation time (they needn't be considered in the model configuration-> solver end time).

Clock, clock-enable, and reset signals are forced. Clock timing is independent from the model; clk_high and clk_low times are defined in the HDL coder/Test Bench Configuration window. A hold time is applied to the reset and input data signals, using the VHDL statements 'WAIT FOR clk_hold' and 'AFTER clk_hold'.

Error detection does not stop the simulation; errors are counted.

Output data are strobed one cycle after the change; each output change is strobed only once. Strobing time is defined by the output signal sampling time. The first check is applied after power-on reset when the output data is in the power-on state to check the reset condition.

Sample time output signals are automatically added on the toplevel as clock enable output ports. The number of clock enable outputs is equal to the number of different sampling times applied to output signals. Details about Output Signal / Clock Enable / Sample Time details are given in the toplevel source code header.

3. Reference design

Let us assume that we need the decimation filter for a typical sigma-delta analog-to-digital converter. For the purpose of this chapter we are more interested in the design flow rather than filter parameters, therefore we decided to choose a small version of the popular CIC filter structure [7-9]. We will assume a low number of stages, a low sampling frequency and will not deal with register pruning to keep the design simple, while devoting more attention to the implementation technique that will allow us to build more advanced features according to specific system requirements. The latter are left to the interested reader as a continuation of this work.

The beauty of Simulink is that we find the answer right away in the DSP System Toolbox Library under Filtering/Multirate filters. The 'CIC Decimation' block does it all for us, we only have to set mask parameters according to given propositions. As we aim at the sigma-delta converter application we assume the signed 2-bit integer data on the filter input, representing the bit-stream data from the sigma-delta modulator. We set the following filter parameters:

- f_{ovs}-the oversampling frequency, equal to f_{clk}; f_{ovs}=512 kHz

- B_{IN}-the input data bit width; B_{IN}=2

- R-the sample rate change implemented by the filter; R=128

- M-the comb filter differential delay; M=1

- N-the number of stages in the filter; N=3

The decimation period of the filter is given by

$$T_{dec} = R/f_{ovs} = 250\,us \tag{1}$$

The gain at the output of the final stage of a CIC decimation filter is calculated as

$$Gain = (RM)^N = (128)^3 = 2097152 \tag{2}$$

The maximum bit width in a decimation filter occurs at the first integrator register:

$$Bmax = \text{ceil}\left(N \, Log_2 \left(RM\right) + B_{IN}\right) = \text{ceil}\left(3 \log 2 \left(128\right) + 2 = 23 \text{ bits}\right) \qquad (3)$$

In the full-precision mode all registers following the first register must support the maximum accumulation produced by the first integrator section. According to the known CIC filter properties the overflow in the integrating sections is prevented because

1. The filter is implemented using two's complement arithmetic that wraps around from the most positive to most negative number representations.

2. The range of numbers supported by the bit width of the filter is greater than the maximum value expected at the output of the composite CIC filter.

We start our design by the construction of the testbench model where the reference design (ref_dec) will be simulated and later compared with the demo (demo_dec) design, described in the next section. Both designs are entered as subsystems, the demo_dec being an empty subsystem for the time being. Configuring the designs as subsystems is the best way to select them for the HDL conversion. The testbench presented in Figure 2 applies the unit step input from the Constant block which defines the value (+1) and the sampling rate (1/f_{ovs}). The filter base-rate is automatically inherited from the same source.

The verification of the demo filter output is simple: the unit-step input must come out as the integer value representing the Gain, which we check on Scope1. The rest of the testbench is not needed for this initial experiment; we will use it later to verify the demo design by comparing it to the reference design. The simulation time must set to at least 4*T_{dec} (in our case ~1 ms) to stabilize the numerical output. The circuit simulation presented in a later section will be run even longer to get an average power consumption value.
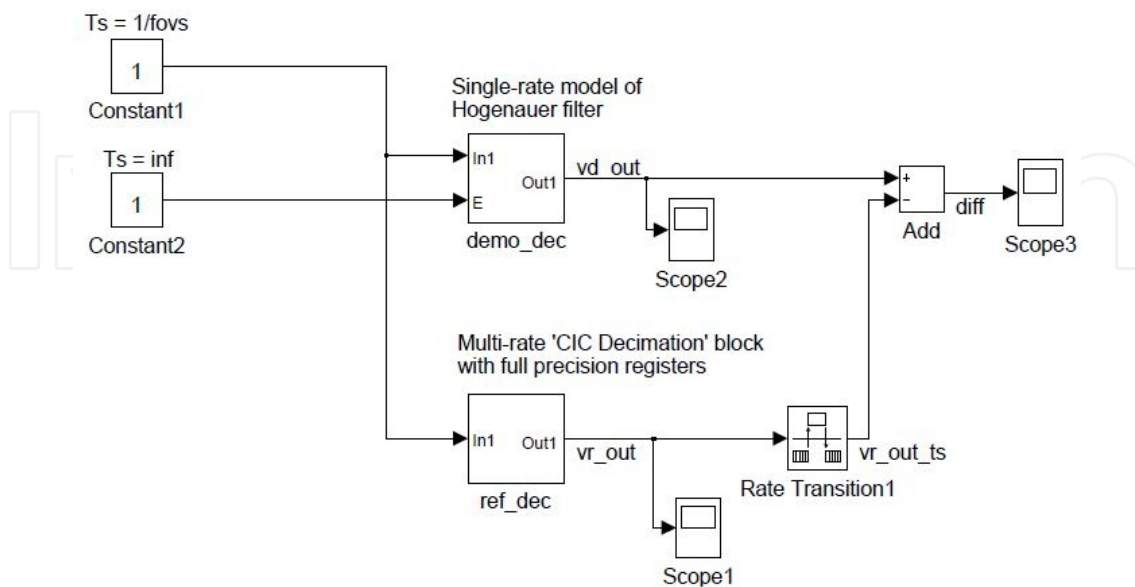

Figure 2. Unit-response testbench

The reference design is presented in Figure 3. It consists of a single masked subsystem (CIC Decimation block) taken from the library and configured for multi-rate operation with dialog parameters R=128, M=1 and N=3.
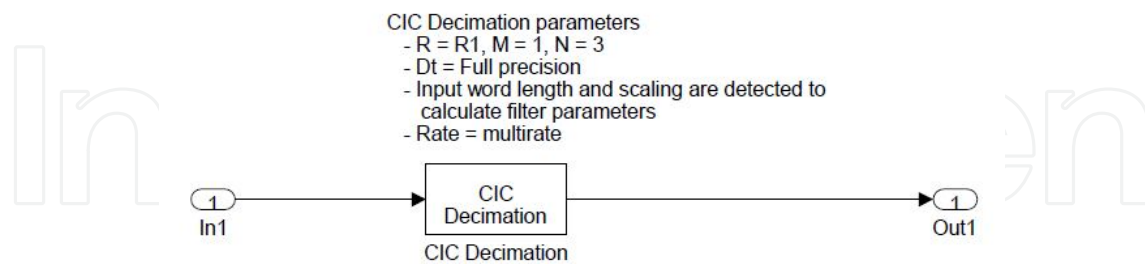

Figure 3. Reference decimator design (ref_dec).

The HDL code generation starts from the Model Explorer by opening the active configuration. Under the HDL Code Generation tab we select 'ref_dec' as the target subsystem and follow the options menu to set the conversion parameters.

There are some mandatory details that must be satisfied to enable the HDL Coder compilation. Under the Solver tag you must set the start time (=0) and stop time, according to the design. Some other settings must be set as follows:

- Solver options: Type=Fixed-step, Solver=discrete, Step size=default (auto).
- Periodic sample time constraint=default (Unconstrained).
- Tasking mode for periodic sample times=Single Tasking.
- Automatically handle rate transition=Off.
- Higher priority value means task priority=Off.
- Diagnostics - Multi-task rate transition=Error.
- Diagnostics – Single task rate transition=Error.

After successful compilation we proceed to the testbench generation which allows us to replicate the testbench simulation with the logic simulation tool.

4. Demo design

If we do some investigation of the HDL code generated for the reference design we can see that the internal structure of our CIC Decimation block puts the down-sampling stage at the end of the last comb section, like the structure presented in Figure 4. This topology spares some hardware. If the system allows additional delay of one base-rate clock cycle at the filter output the down sampler switch can be eliminated so that the down sampler evolves into a simple

output register. However, like in the structure presented in Figure 4, this is the worst case solution from the point of view of power consumption because of the direct high-frequency switching data path from the integrating section through all comb section adders to the filter output register.
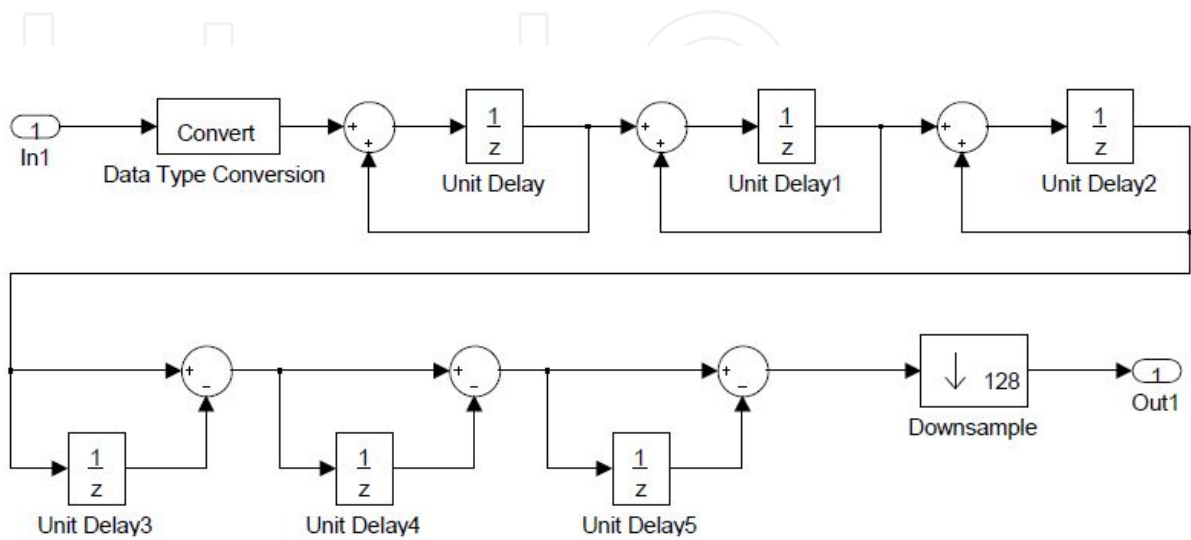

Figure 4. CIC Decimation block internal structure, as used in the reference design.

According to Nobel identity [8] the down sampler can be placed either between the last integrating section and the first comb section, or at the end of the last comb section as in Figure 4. In our demo design we will therefore place the down sampler after the last integrating section, following the Hogenauer filter topology [10] presented in Figure 5. Moreover, we will build our own down sampler block so that it will contain the timing counter and allow single-rate modeling.

The down sampler presented in Figure 6 can be parameterized and put in the user library, using the down sampling factor R as a mask parameter. The remaining parameters are calculated in the mask initialization process:

- CountLen=ceil(log2(R));

- CountLim=R-1;

The decoded zero-state (Ph0) and the last state (Phl) counter signals provide the control for output data synchronization with the rest of the system. Since all signals run with the base-rate clock it is easy to extend the model with the required type of data synchronization on the filter output, using data buffering and/or handshaking protocols. Figure 7 illustrates the down sampler timing with control signals, counter state and filter output at the time when filter output reaches the final unit-response output value.
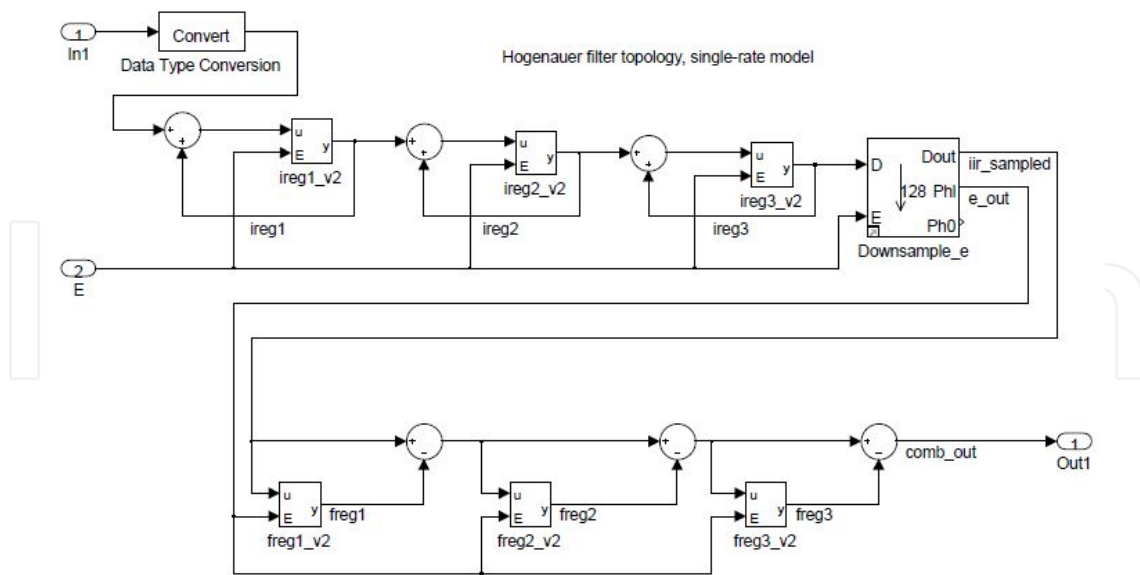
Figure 5. Decimator demo design (demo_dec).

The verification of the demo design is possible by inspecting of the difference between filter outputs. So, the waveform on Scope3 in Figure 2 shows zero difference in the span of the whole simulation time. Since we are comparing waveforms with different time bases we must convert the reference output to the base-rate timing, using the Rate Transition block. The latter must be set with the disabled data integrity checkbox; otherwise the inserted delay prevents the comparison.
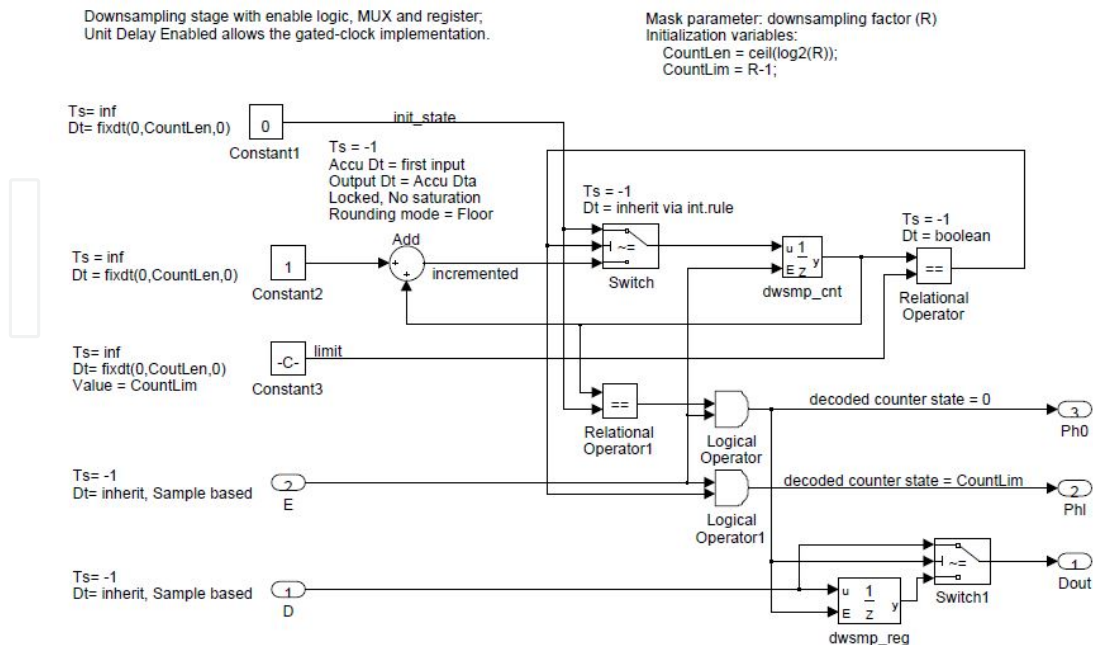

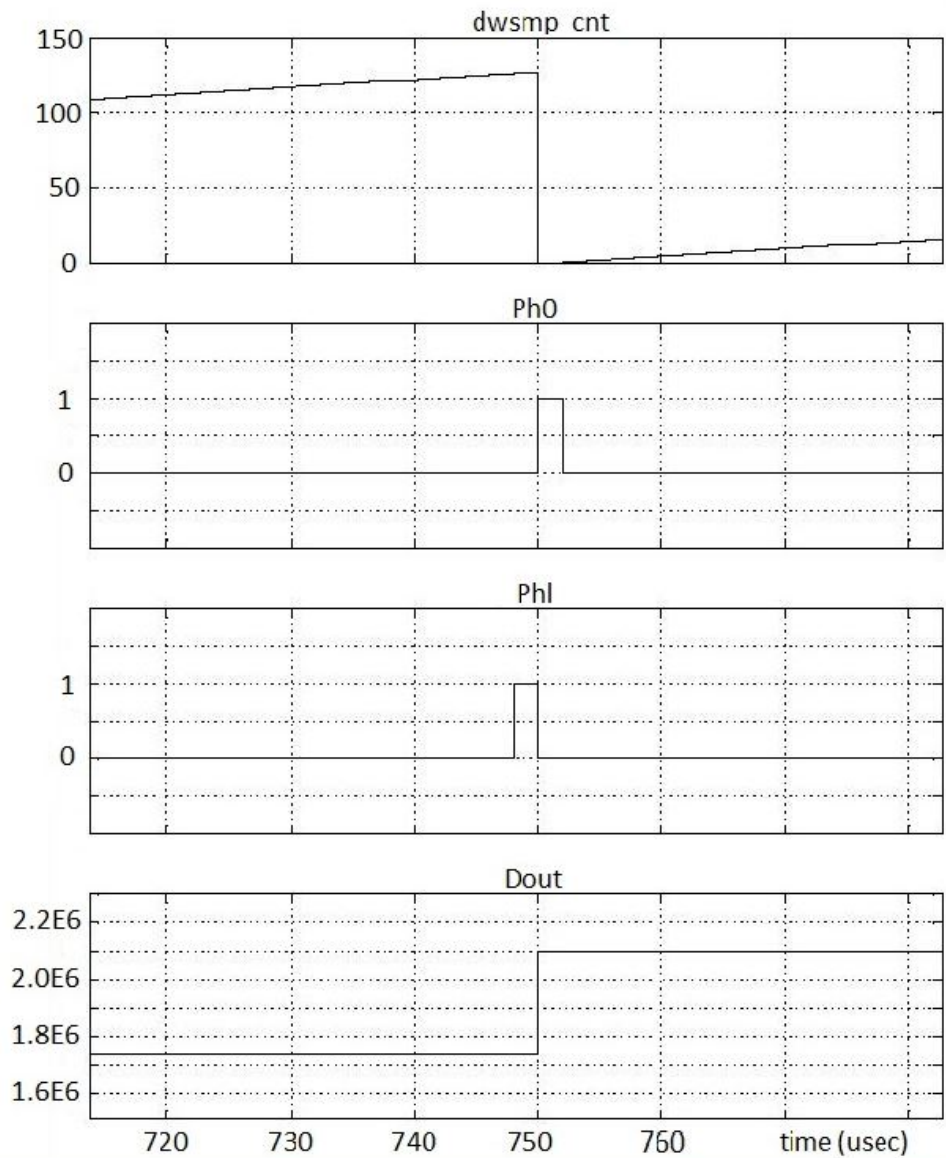**Figure 6.** Custom made down-sampling block with built-in counter (Downsample_e).

Figure 7. Downsample_e timing detail (vertical axes represent signal integer values).

5. Circuit simulation results

After HDL conversion of the reference and demo designs we proceeded to the synthesis and circuit simulations to compare circuit features. Our synthesis has been based on the TSMC 0.25um CMOS technology, using the ARM standard digital library and Synopsys Design Compiler tool. Gated-clock technology was applied to enable low-power operation. Accordingly, the circuit's physical layout with clock tree synthesis and parasitic extraction has been completed with Cadence tools. Finally, the back-annotated Verilog netlist have been simulated

with the NanoSim tool on the transistor-level to provide realistic circuit parameters. Power consumption was measured for the step-response input, using the setup from Figure 2 with fclk=512 kHz and Vdd=2.5 V. We have applied equal loads of one inverter gate on all output data bits, again to assure realistic circuit application conditions. The simulation time was set to ~16 decimation periods so that the average power supply current reached a practically stable value.

The comparison of the two circuit implementations is presented in Table 1 and in Figure 8. We can see that circuit implementation parameters are very similar, with the exception of power consumption and average supply current, which differ by more than 30%. This is in accordance with initial assumptions, presented at the beginning of the demo design section.
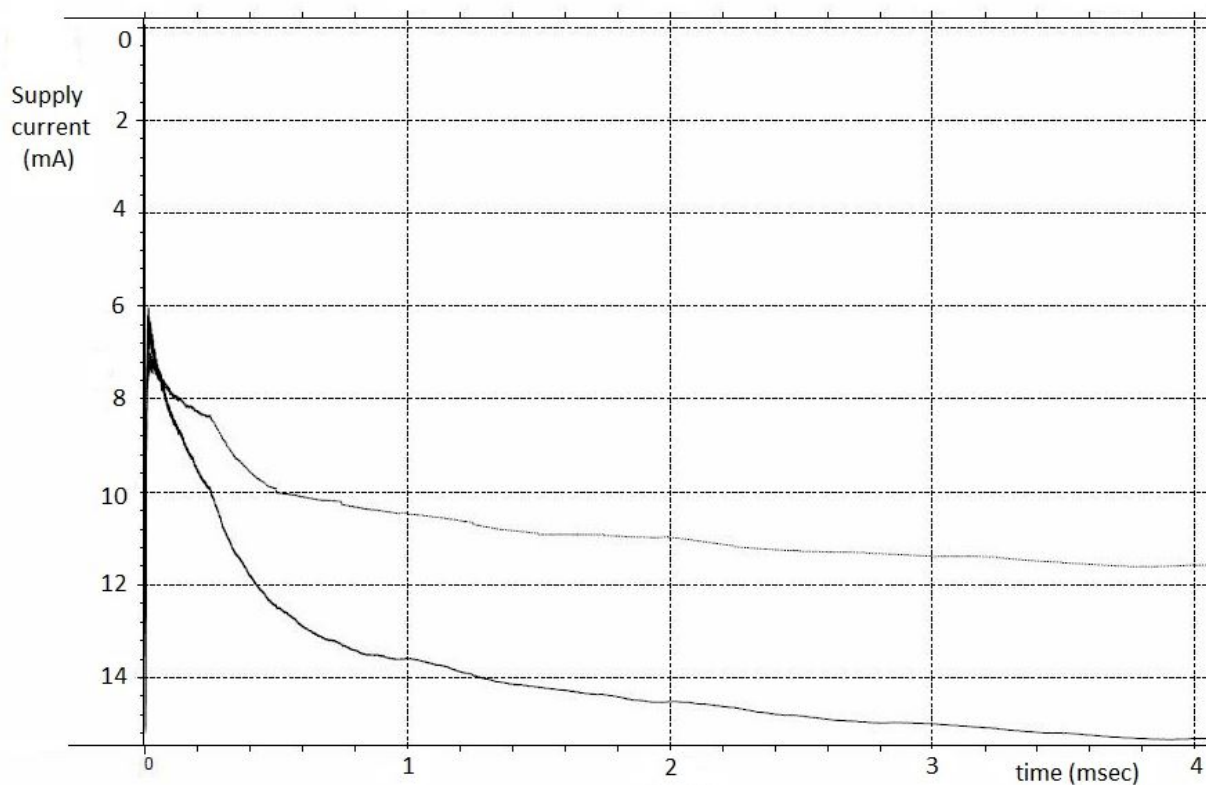

Figure 8. Comparison of the simulated average power supply currents (upper curve: demo design, lower curve: reference design).

Design	Gate-level Leaf cell count	Physical Design area [um]	Circuit power simulation [uW]
reference	487	315 x 315	38.270
demo	466	315 x 315	28.928

Table 1. Comparison of two circuit implementations.

6. Advanced design

6.1. Programmable decimation rate

As we have already mentioned, single-rate modeling gives more control over timing and allows one to model structures that are not available in standard Simulink libraries. One such example is the programmable decimation rate filter, presented in Figure 9. Here we have chosen the very simple architecture of a one-stage decimator in order to have more room to present the concept. The down-sampling stage structure is explained separately in Figure 10. It consists of a counter with selectable last-state signal, implemented by the 'Multiport switch' block from the Simulink Signal Routing library. The 'win' input allows for the selection of deliberate decimation rates in the form R=2^win. The example in Figure 8 assumes 3-bit unsigned integer value on the 'win' input, allowing selecting from seven different down-sampling rates.
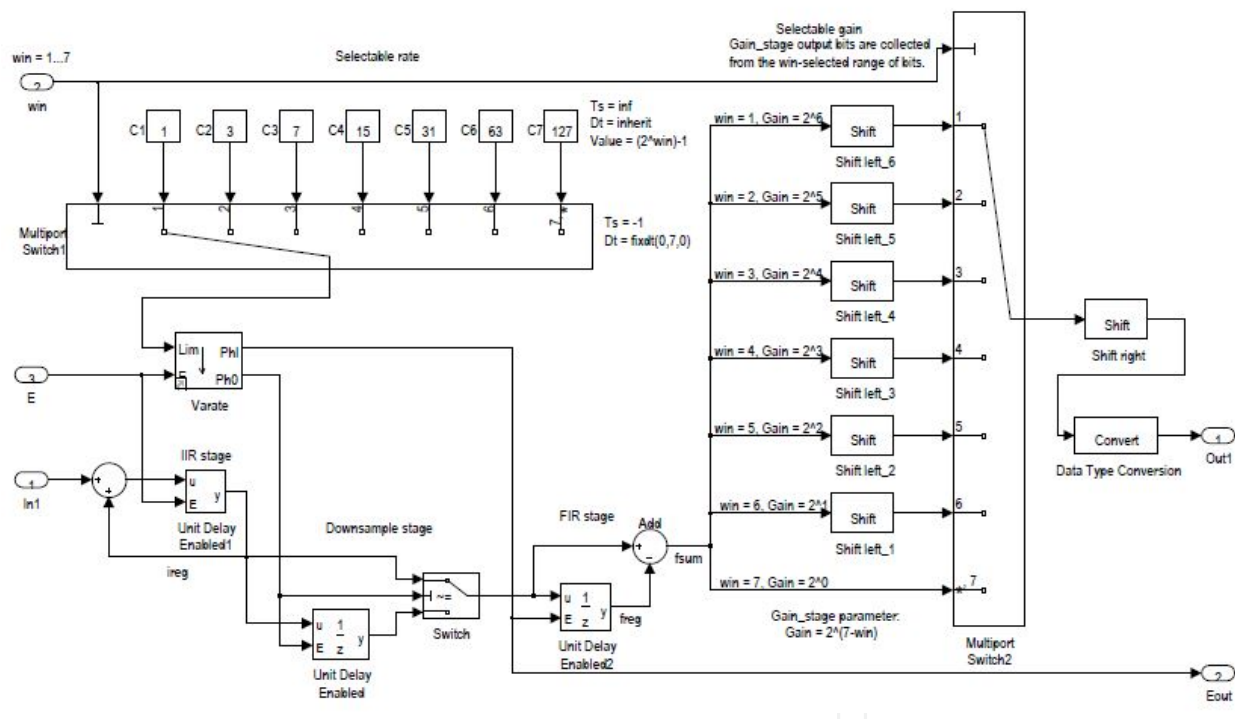

Figure 9. Programmable decimation rate filter.

The trick around the programmable down-sampling stage is that it should be usually followed by the programmable gain stage to compensate the variable decimator gain. In our case this feature is implemented by another Multiport Switch block for the selection of the appropriate gain, implemented by the 'Shift Arithmetic' blocks from the Simulink Logic and Bit Operations library. The final output is then supplemented by another, common gain stage and the conversion to the desired output data type.
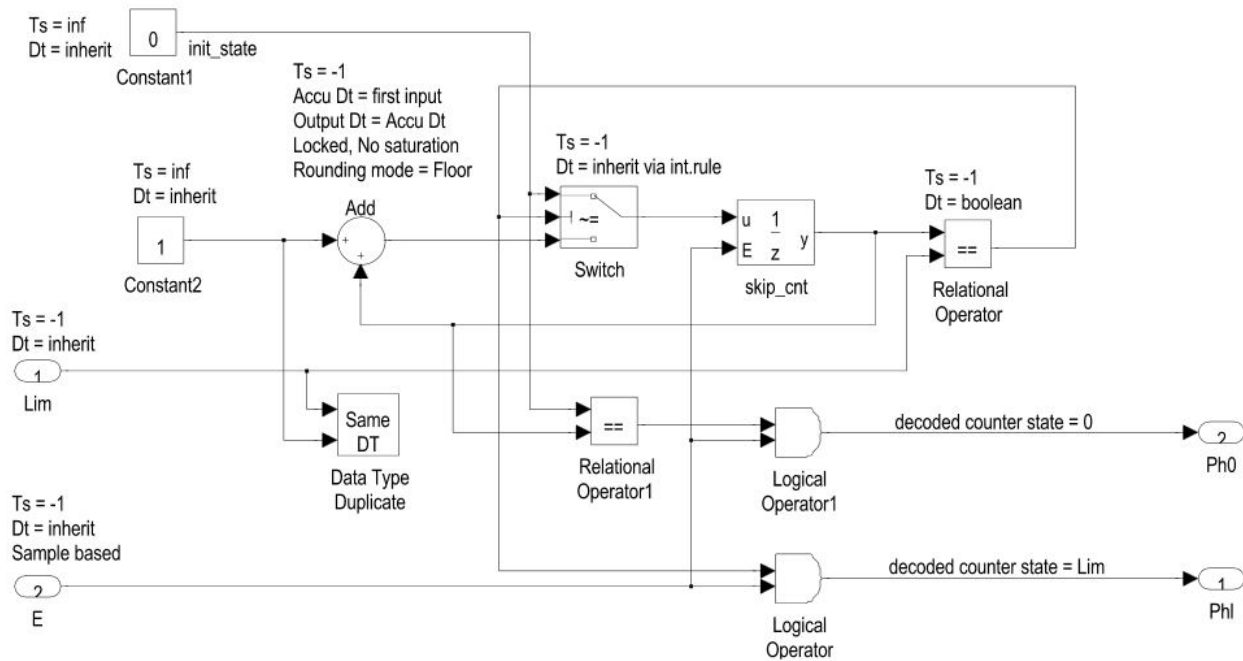
Figure 10. Variable decimation counter block (Varate).

6.2. Unbiased rounding

The results of an arithmetic operation often extend beyond the word length of the data bus and the question arises how to best throw away bits when necessary. Three alternatives are commonly encountered to pare down the result in order to fit the register or bus-width requirements [11].

Truncation of less significant bits systematically underestimates the values; e.g. 1.49 becomes 1.4 when less significant bits are removed.

Rounding pushes a result downwards if the bits to be deleted are less than half of the full-scale LSB, and upward if more than half-scale. For example, 1.44 becomes 1.4 and 1.47 becomes 1.5 when rounded to two digits.

Unbiased rounding rounds up half the time and down the other half. This is usually done by rounding towards an even number and relaying on the random distribution of even and odd numbers.

If only the MSH of a data set is to be kept, rounding is preferable to truncation, and unbiased rounding is preferable to rounding. The uround (unbiased-rounding) algorithm is simple: perform the rounding, and then detect if the LSH is '100…0', signifying that the LSH started out at "half-scale". The uround logic rounds this event up half the time and down half the time. The decision may be based on the evaluation of the LSB of the MSH (for example, add 1 to the LSB of the MSH if it is 1 or 0 if it is 0, respectively).

One possible way to implement the uround algorithm in Simulink is presented in Figure 11. The model can be put in the custom library, using WLin and WLout as mask parameters, while WLstrip is calculated as WLin-WLout in the mask initialization process.

Tables 2 and 3 illustrate paring of the data word y with initial word length WLin and the reduced word length WLout. Four different techniques are compared by calculating the averaged sum S and the mean number value M. To prevent negative results for y=28…31 the uround adder must not be configured for the wrap-around arithmetic. The condition where the uround logic adds MSH(0) to the result is marked with '*'. The sum S is calculated as S=Sum all (y / (2^(WLin-WLout)-1)), while the mean number value M is calculated as M=S/(2^WLin).

Val.	b5	b4	b3	b2	b1	b0				
y		MSH			LSH		Truncate	round with LSH(msb)	uround with MSH(0)	Matlab round (y/8)
0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	0	0	0	0
2	0	0	0	0	1	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0	0
4	0	0	*0	1	0	0	0	1	*0	1
5	0	0	0	1	0	1	0	1	1	1
6	0	0	0	1	1	0	0	1	1	1
7	0	0	0	1	1	1	0	1	1	1
8	0	0	1	0	0	0	1	1	1	1
9	0	0	1	0	0	1	1	1	1	1
10	0	0	1	0	1	0	1	1	1	1
11	0	0	1	0	1	1	1	1	1	1
12	0	0	*1	1	0	0	1	2	*2	2
13	0	0	1	1	0	1	1	2	2	2
…										
…										
28	0	1	*1	1	0	0	3	4	*4	4
29	0	1	1	1	0	1	3	4	4	4
30	0	1	1	1	1	0	3	4	4	4
31	0	1	1	1	1	1	3	4	4	4
S=62							S=48	S=64	S=62	S=64
M=1.9375							M=1.5	M=2.0	M=1.9375	M=2.0

Table 2. Positive numbers example (WLin=6, WLout=3).

Value	b5	b4	b3	b2	b1	b0	Truncate	round with LSH(msb)	uround with MSH(0)	Matlab round (y/8)
y		MSH			LSH					
-32	1	0	0	0	0	0	-4	-4	-4	-4
-31	1	0	0	0	0	1	-4	-4	-4	-4
-30	1	0	0	0	1	0	-4	-4	-4	-4
-29	1	0	0	0	1	1	-4	-4	-4	-4
-28	1	0	*0	1	0	0	-4	-3	-4*	-4
-27	1	0	0	1	0	1	-4	-3	-3	-3
-26	1	0	0	1	1	0	-4	-3	-3	-3
-25	1	0	0	1	1	1	-4	-3	-3	-3
-24	1	0	1	0	0	0	-3	-3	-3	-3
-23	1	0	1	0	0	1	-3	-3	-3	-3
-22	1	0	1	0	1	0	-3	-3	-3	-3
-21	1	0	1	0	1	1	-3	-3	-3	-3
-20	1	0	*1	1	0	0	-3	-2	-2*	-3
-19	1	0	1	1	0	1	-3	-2	-2	-2
-18	1	0	1	1	1	0	-3	-2	-2	-2
-17	1	0	1	1	1	1	-3	-2	-2	-2
-16	1	1	0	0	0	0	-2	-2	-2	-2
-15	1	1	0	0	0	1	-2	-2	-2	-2
-14	1	1	0	0	1	0	-2	-2	-2	-2
-13	1	1	0	0	1	1	-2	-2	-2	-2
-12	1	1	*0	1	0	0	-2	-1	-2*	-2
-11	1	1	0	1	0	1	-2	-1	-1	-1
-10	1	1	0	1	1	0	-2	-1	-1	-1
-9	1	1	0	1	1	1	-2	-1	-1	-1
-8	1	1	1	0	0	0	-1	-1	-1	-1
-7	1	1	1	0	0	1	-1	-1	-1	-1
-6	1	1	1	0	1	0	-1	-1	-1	-1
-5	1	1	1	0	1	1	-1	-1	-1	-1
-4	1	1	*1	1	0	0	-1	0	0*	-1
-3	1	1	1	1	0	1	-1	0	0	0
-2	1	1	1	1	1	0	-1	0	0	0
-1	1	1	1	1	1	1	-1	0	0	0
S=-66							S=-80	S=-64	S=-66	S=-68
M=-2.0625							M=-2.5	M=-2.0	M=-2.0625	M=-2.125

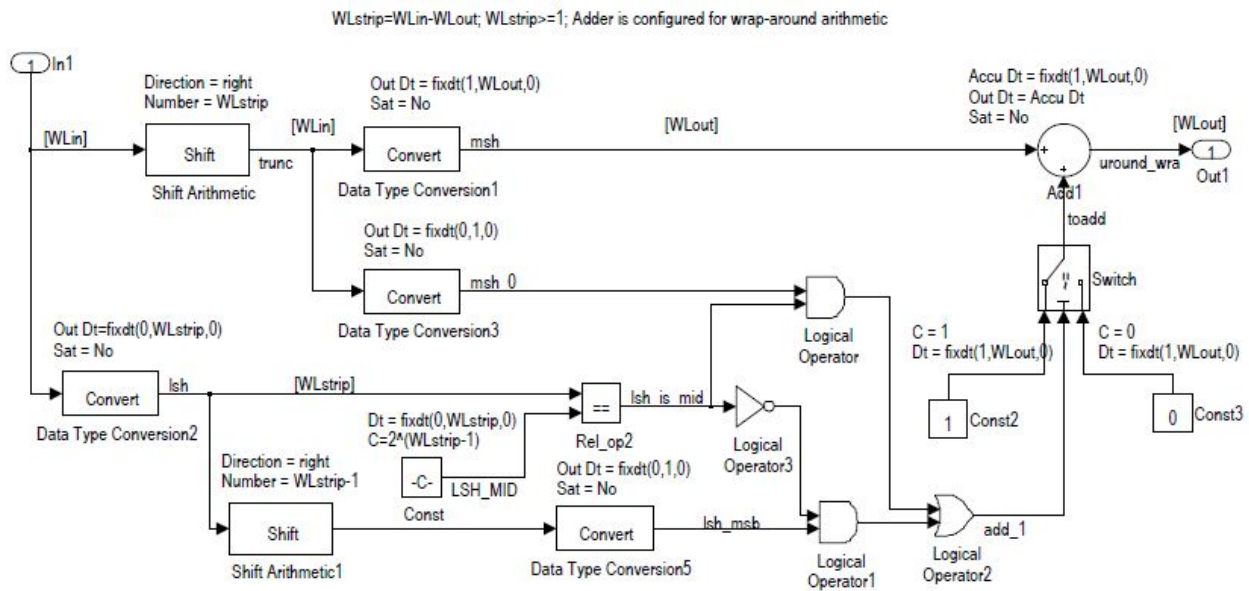**Table 3.** Negative numbers example (WLin=6, WLout=3).

WLstrip=WLin-WLout; WLstrip>=1; Adder is configured for wrap-around arithmetic

Figure 11. Unbiased rounding block example.

7. Conclusion

This chapter deals with modeling, simulation, and synthesis of a digital module for mixed-signal applications using a high-level Matlab/Simulink model on a high hierarchical level. More specifically, it presents an example design and the modeling details of a 3-stage CIC decimation filter suited for automatic conversion to the synthesizable HDL data base, using the HDL coder tool. The aim was to enable automatic procedures to finish the job, without any designer manipulation of the generated code on the low (HDL) level. Considerable care has been taken to provide practical instructions to cope with some tool-specific requirements. A single-rate modeling approach has been used to enable further modeling of specific operating modes and/or data handshaking procedures. Using the proposed modeling technique, one can speed up the implementation, improve the reliability, assure system integrality, and provide comprehensive documentation by maintaining one single, high-level model throughout the whole design effort.

Appendix A: Glossary of acronyms

ARM   A commercial provider of integrated circuit components

ASIC   Application-specific integrated circuit

Cadence   A commercial provider of the integrated circuit design tools

CIC	Cascaded integrator-comb
CMOS	A specific integrated circuit production technology
DSP	Digital signal processing
DUT	Device under test
FPGA	Field-programmable gate array
HDL	Hardware design language
LSB	Less significant bit
LSH	Less significant half
MSB	Most significant bit
MSH	Most significant half
RTL	Register transfer level
Synopsys	A commercial provider of the integrated circuit design tools
TSMC	A commercial silicon foundry
Verilog	A specific HDL language-
VHDL	A specific HDL language

Acknowledgements

Author details

Drago Strle[*] and Dušan Raič

*Address all correspondence to: drago.strle@fe.uni-lj.si

University of Ljubljana, Faculty for Electrical Engineering, Ljubljana, Slovenia

References

[1] Grout, I.A. Modeling, simulation and synthesis: From Simulink to VHDL generated hardware. 5th World Multi-Conference on Systemics, Cybernetics and Informations, SCI 2001.

[2] Mauderer A., Oetjens J.H. System-level design of mixed-signal ASICs using Simulink: Efficient transitions to EDA environments. EETimes; 2012. http://www.eetimes.com/General/PrintView/4373903

[3] Erkkinen T., Breiner S. Automatic Code Generation–Technology Adoption Lessons Learned from Commercial Vehicle Case Studies. The MathWorks; 08AE-22; 2007.

[4] MathWorks. DSP System Toolbox Reference. The MathWorks; 2012.

[5] MathWorks. Simulink User's Guide. The MathWorks; 2012.

[6] MathWorks. HDL Coder User's Guide. The MathWorks; 2012.

[7] Meyer-Baese U. Digital Signal Processing with Field Programmable Gate Arrays. Springer; 2001.

[8] Harris F.J. Multirate Signal Processing for Communication Systems. Prentice Hall; 2004.

[9] Newbold R. Practical applications in digital signal processing. Prentice Hall; 2013.

[10] Hogenauer E.B. An Economical Class of Digital Filters for Decimation and Interpolation. IEEE Transactions on Acoustics, Speech and Signal Processing, ASSP-29(2); 1981. 155–162.

[11] Higgins J.H. Digital signal processing in VLSI. Prentice Hall; 1990.