CHALMERS

Executing Simulink models on FPGA in a LabVIEW environment

Master of Science Thesis in Integrated Electronic System Design

OLOF KINDGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, September 2009

Executing Simulink models on FPGA in a LabVIEW environment

Olof Kindgren

Examiner: Sven Knutsson

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden September 2009

Abstract

This paper describes the tools, methods and problems of converting Simulink models to IP blocks for FPGA and run them in the LabVIEW FPGA environment. The conversion from Simulink models is performed with Mathworks Simulink HDL Coder, Xilinx System Generator and by manually writing HDL code to investigate the different aspects of the workflow.

A proof-of-concept model is implemented with different parts converted with different methods. The analysis show that the automatic tools are useful for quickly implementing and verifiying DSP models. It is also noticed that the tools in many cases produce suboptimal code and in these cases hand-written code is the only option. The target platform, National Instruments compactRIO, is considered useful as there are high-level communication readily available, simplifying the integration with other components. The biggest problem with the RIO platforms is the lack of control a developer has, once the IP is inserted in the LabVIEW FPGA design flow.

Contents

List of Figures

1 Introduction

Traditionally there have been different design flows for developing systems for hardware and software. This forces the system designer to decide where to deploy the different parts of a system and define an interface between them early in the development process. Later on, these choices can turn out to produce a suboptimal solution. When the need arises to refactor a system between the different platforms, it has to be reimplemented for the new platform, using the old system as a reference. As the two platforms require different tool chains, design methodologies and engineering skills, a lot of manual development and verification might be needed. Lately there have been attempts to overcome this problem. The RIO[2] platforms from National Instrumens tries to bridge the gap between software and FPGA development by providing hardware units with FPGA, I/O, and a processor running a realtime OS. FPGA development for these platforms is incorporated in the LabVIEW design flow and high-level communication protocols between the components allow easier refactoring of a system between FPGA, realtime OS and a PC. External IP blocks can be integrated on the FPGA, allowing access to existing systems developed for FPGA.

One class of such systems especially well suited for FPGA is DSP applications. For developing DSP applications in software, Simulink is often used. Simulink is a data flow programming tool with a large number of function blocks and an environment for simulation and verification. The advantages of using high-level data flow programming to build DSP systems for FPGA has been recognized and several tools have been created to aid in the transition to FPGA.

The purpose of this master thesis is to examine some of the tools, methods and potential problems of converting existing Simulink models written for execution in software to HDL code, for implementation in the LabVIEW FPGA environment. The idea is to combine and benefit from the high speed and low latency of FPGA, the consistent programming environment of LabVIEW and the widespread use of Simulink models.

In *Section 2* the proposed workflow is presented, together with a few alternative methods, when applicable. *Section 3* describes a model that has been implemented as a proof-of-concept. This model, along with general observations is analyzed in *Section 4* and a summarized conclusion is found in *Section 5*, together with notes of what was left out in the writing of this thesis.

Figure 1 on page 5 shows an overview of the different formats that can be used and how to translate between them.

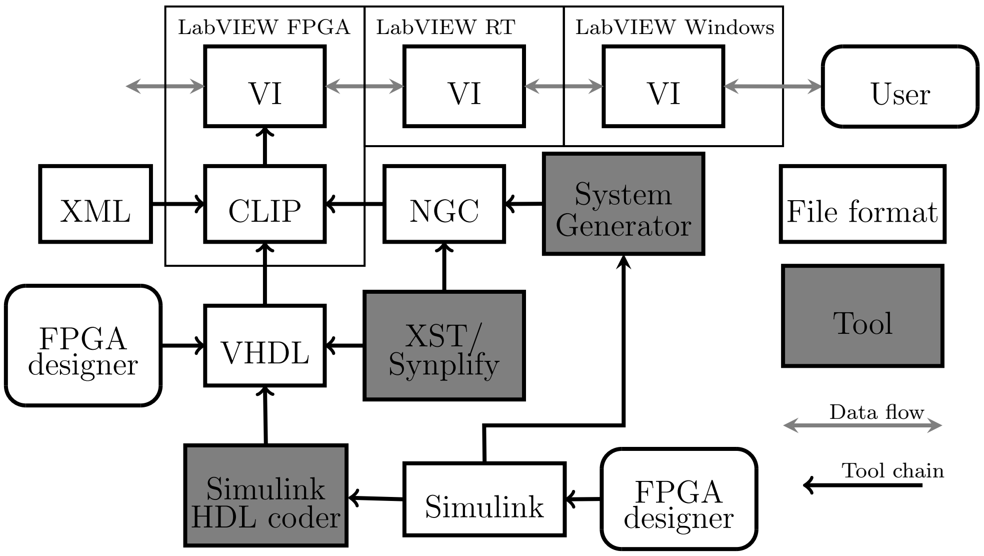

Figure 1: Toolchain overview

2 Workflow

Integrating a Simulink model written for execution in software into the LabVIEW FPGA environment requires several intermediate steps, and can be done in different ways with different tools. The ultimate goal, however is to have a net list or synthesisable HDL code which is functionally equivalent to the original Simulink model and can be imported into LabVIEW.

Every modification that occur to the model during the conversion may have an impact on function, chip area, precision and speed. It is therefore important to be able to verify the correctness of the model at all stages in the conversion. In some cases this can be done by testing for equivalence with the previous iteration. When changes to the model has to be made in a way that no longer makes it equivalent to the previous iteration, the model should instead be checked against the original specifications. An outline of the workflow is summarized in *Figure 2* on page 6.

2.1 Preparing the model

Only a subset of the function blocks and features in Simulink can be realized in FPGA. Therefore the first step to make the model ready for HDL generation is to identify incompatibilities. This includes:

- Using a fixed-step discrete time solver instead of continuous states to emulate clocked logic design.

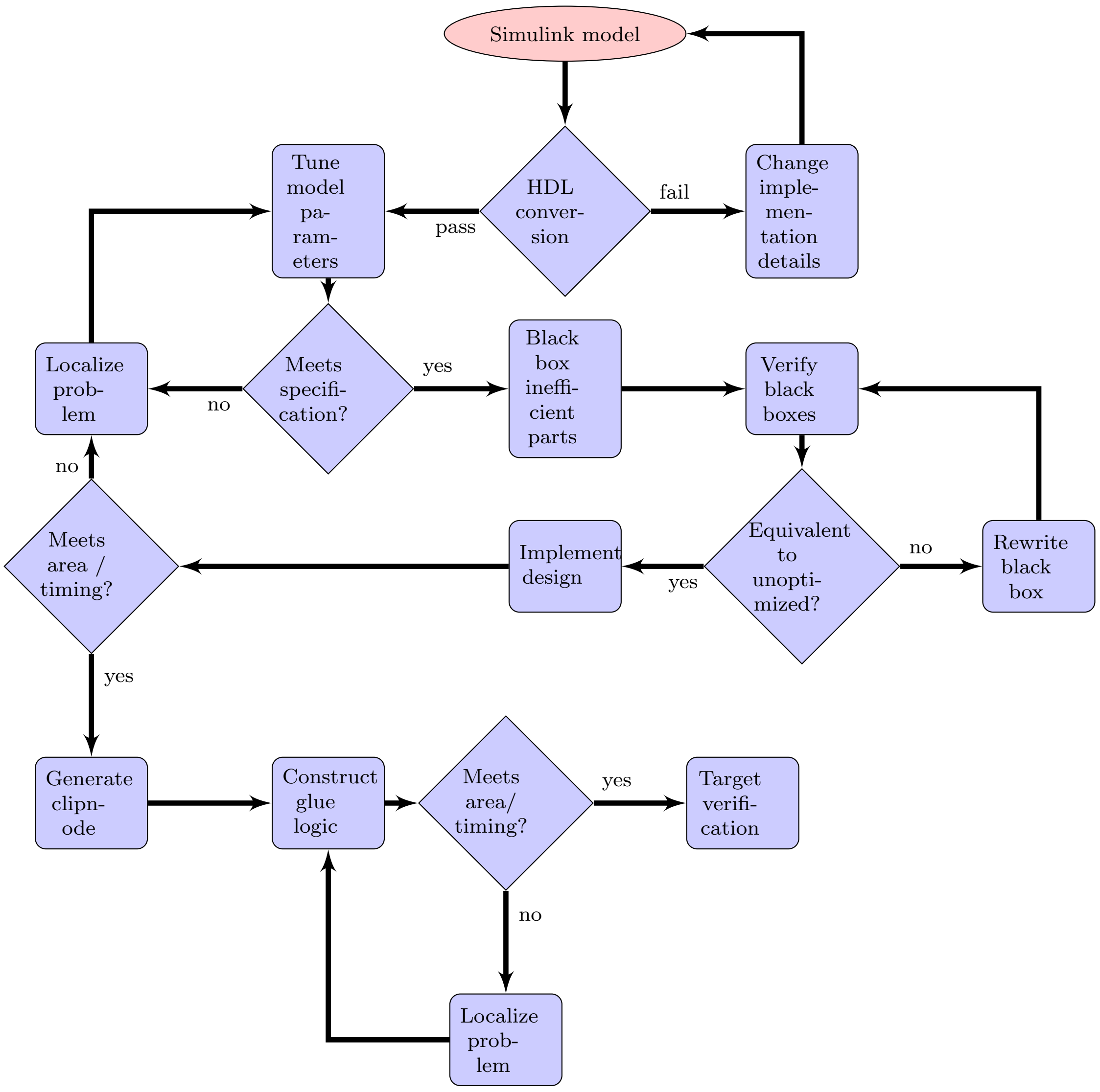

Figure 2: Workflow

- Replacing all floating point arithmetics with fixed-point.
- Only use blocks for which there exists a corresponding HDL implementation.

The model may not be able to directly use a fixed-step discrete time solver, which is a prerequisite for converting the model. In some cases models may need to undergo too drastic changes to be candidates for implementation in FPGA, while others can be used after some redesign.

The use of fixed-point arithmetics instead of floating point will in most cases introduce some precision loss. The amount of precision loss is related to the size of the fixed-point signal paths and the method used to round fixed-point numbers. This is a trade-off between silicon area and precision and must be taken into consideration.

Simulink HDL Coder maintains a list of blocks that have a HDL implementation, and any model can be checked against this list. The number of supported blocks increase for every version of Simulink HDL Coder, and at the time of writing, the list included most arithmetic operations, tools for filter design and support writing embedded M Code. Some blocks, such as graphical output or file I/O, have no equivalent implementation in FPGA and are therefore excluded. Other native Simulink blocks may not yet be ported and must therefore be built from more primitive blocks.

2.2 Generating HDL code

Starting out from a Simulink model there are three proposed ways to turn this into HDL. The whole model can be implemented using one method or different methods can be used for subsets of the model. The three methods analyzed here are:

- **Convert with Simulink HDL coder**
  Mathworks Simulink HDL Coder[3] is a tool for converting a model built in Simulink directly to VHDL or Verilog, and is aimed at rapid prototyping. All native Simulink blocks however are not supported by Simulink HDL Coder and the model must therefore be built using the supported subset of blocks. To automate the process Simulink HDL Coder can analyze a model and inform the designer of what needs to be done. If all criterias are met, the model can be translated to HDL. If changes were made to the model it should be compared with the original model by cosimulating them inside of Simulink. In addition to the generated code, Simulink HDL Coder can also provide test benches

with stimuli to aid in the HDL verification process. Using Simulink HDL Coder is assumed to be the preferred way of generating HDL code in this report, since it requires the least effort when dealing with existing Simulink models. The other methods are primarily used as complements when using Simulink HDL coder is not applicable.

- **Build an equivalent Xilinx System Generator model**
  Xilinx System Generator[8] provides a library of Simulink blocks that are designed for implementation in Xilinx FPGA. Using only these blocks, Xilinx System Generator can synthezise the model into netlists with an optional HDL wrapper. The blocks on the Simulink block diagram are mostly implemented as Xilinx IP blocks, such as those available from Xilinx Core Generator. Xiling System Generator also generates a large amount of VHDL macro functions to handle data conversion between the blocks. As only System Generator blocks are supported in this workflow, a Simulink model must be manually converted from using native Simulink blocks. For verification of model correctness, the two models can be cosimulated inside of Simulink to check for equivalence.

- **Write equivalent HDL code**
  By analyzing the behaviour of the model, HDL code representing an equivalent model can be written manually. In order to test the equivalence of the HDL implementation with the Simulink model, Simulink HDL Coder can provide test benches for ModelSim and a few other logic simulators. These test benches contain pregenerated input and output vectors based on the original model to be used as stimuli. This is meant as a last resort, as the emphasis of this paper is on automated conversion methods.

2.3 Optimizing HDL code

Even if the generated HDL code can be synthezised and meets the specifications, the design might be suboptimal for FPGA implementation. Many of the default implementations of the Simulink blocks are optimized for low latency, to mimic their software equivalents, which generally have no latency. This can however have a negative impact on speed and area when a FPGA is the target. A test build of the generated code is therefore needed to ensure that the code meets these requirements.

Should the code not satisfy the demands for area and speed the problem should be localized. There are then several ways of improving the code, while still taking advantage of the automatically generated test benches and the Simulink environment.

2.3.1 Tuning the model

Many arithmetic blocks grow linearly or even exponentially depending on the size of the bus width of the input and output data. For a DSP application there can be much to gain by minimizing the needed bus width in the signal chain, as a too wide bus will use up excessive resources without adding any benefit. Narrowing down the bus too much will have a negative impact on SNR[1] due to rounding errors, and can also effect the functional requirements on the design. The SNR can often be related to the bit width as 6dB per extra bit[6].

A different approach is to completely change the implementation of some block. Complex blocks generally have more options to choose from, but even a simple function such as an adder can have several implementations, each focused on small area, high clock speed or low latency. There are also more generic parameters to take into consideration, such as providing synchronous, asynchronous or no reset at all to a block.

Simulink HDL coder provides a way to specify these parameters in a control file, which is consulted when the HDL code is generated. The parameters can be set for individual blocks, classes of blocks or whole subsystems. Parts of a commented control file is included in *Listing 5* on page 23. Using System Generator these parameters are instead specified directly in the blocks used.

2.3.2 Black-boxing

If changing bit widths or tweaking model parameters won't give the desired results, or if there is no feasible way to implement a part of the design with Simulink HDL Coder or System Generator, that part of the model can be black-boxed. Black-boxing a subset of a model prevents the tool from generating HDL code for that part. Instead it can be replaced with a netlist or external HDL code which is compiled with the rest of the model after synthesis. The black box can still contain a Simulink model which is used for simulating the behaviour inside of Simulink.

[1]Signal to Noise Ratio. Defines the amplitude of a signal compared to the background noise

2.4 Integrating with LabVIEW

The LabVIEW software environment is a dataflow programming language for software development for many targets, including Mac and PC as well as several embedded platforms. The user interface together with a block diagram containing functional code is called a VI (Virtual Instrument), as it originally was made to emulate desktop instruments such as oscilloscopes and signal generators.

This concept has also been expanded to generate FPGA load files for the RIO platforms from a block diagram. The RIO platforms are a series of hardware units from National Instruments containing a FPGA and I/O. Several of these systems also contain a CPU running a realtime OS and onboard memory, effectively turning it into an embedded computer with a dedicated FPGA for I/O handling.

The generated load file contains the code generated from the block diagram and a wrapper which LabVIEW uses internally to handle all FPGA I/O. The I/O consist both of communication with the host system for the FPGA, and any I/O modules that may be present on the selected platform. This prevents the developer from having direct access to the I/O, and instead relying on higher-level LabVIEW defined communication protocols. The intention of this is to reduce the amount of custom protocols and glue logic that needs to be written.

When building a block diagram, the developer has access to many LabVIEW function blocks, such as integer and fixed-point arithmetics, signal generation and analysis tools and many of the built-in dataflow design patterns. In addition to the built-in function blocks, there is also a system called CLIP[5] to include external IP blocks. CLIP allows developers to access the top-level entity of a IP block as if it was a built-in function block. To insert a CLIP into a LabVIEW project an XML description file is used to provide information about the IP to LabVIEW. The XML file describes the following:

- Connection between HDL entity names and LabVIEW signal names
- Which HDL entity signals that should be used for clock and reset.
- The frequency the CLIP should be compiled for
- Files that should be included when building the CLIP
- Name of the top entity

Clock and reset signals are handled transparently in LabVIEW. All other signals are passed to the VI and can be used to connect the CLIP to other

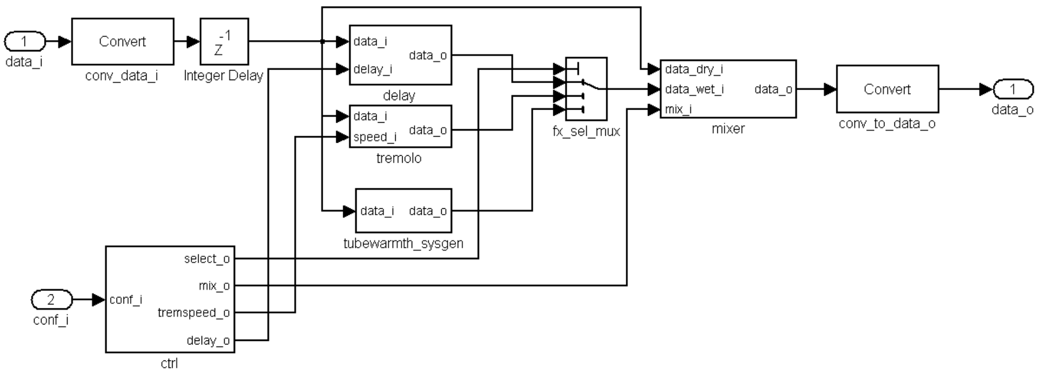

Figure 3: FPGAFX structure

LabVIEW blocks. This makes it possible for seamless integration with the native block diagram units. To compile and build a load file, LabVIEW FPGA uses the Xilinx ISE toolchain.

3 Implementation

In order to perform analysis on the toolchain, a DSP model is created. The constructed model to be analyzed is built with the intent of processing audio signals. The reason for choosing an audio signal is the ability to analyze the behaviour of the system by listening to the processed signal through speakers in addition to viewing the waveform on an oscilloscope. The implemented model, called FPGAFX, contains three sub blocks that adds audible effects on the input signal. These three blocks are an echo block that delays the source signal by a specified amount of time, a tremolo block that multiplies the source signal with a slow changing sinus carrier with configurable frequency, and a tubewarmth block that emulates the effect of a signal passing through a vacuum tube. A fourth block, called mixer, blends the processed signal and the unprocessed signal. A fifth block contains a simple interface for setting parameters in the other blocks and choose which of the three effects that should be applied to the source signal. Together, these blocks make up a multieffect unit for audio processing. The resource usage and implementation details of the three main effect units are chosen to be diverse enough to make it possible to analyze different aspects of the workflow. The structure of the model is shown in *Figure 3* on page 11, and all blocks are explained in greater detail in this chapter

The target for the model implementation is the Spartan 3 FPGA contained in a compact RIO 9074. The compact RIO is a stand-alone hardware

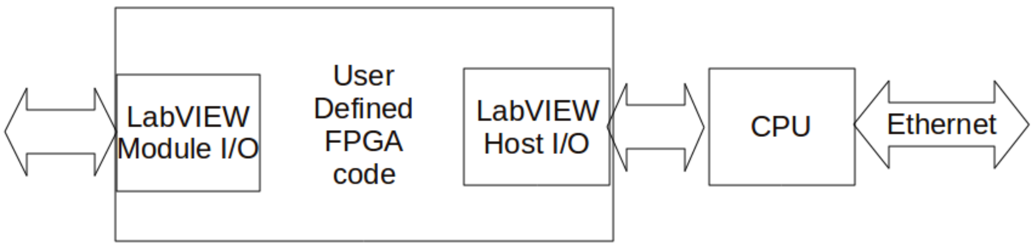

Figure 4: Compact RIO overview

unit with a FPGA situated between pluggable IO modules and a 400MHz PPC CPU running a real-time OS. The I/O modules used in this setup consists of a NI cRIO-9215 A/D converter and a NI 9263 D/A converter. The CPU runs a VxWorks OS with an application to monitor and control the FPGA. The load file for the FPGA consists of both HDL code generated from LabVIEW block diagrams and external HDL code imported as a CLIP, that make up the user defined parts of the FPGA. During the building of the load file, LabVIEW also inserts code for I/O handling, as seen in *Figure 4* on page 12

3.1 Test environment

As seen in *Figure 5* on page 13, the model recieves its input from an analog source via a NI cRIO-9215 A/D converter. The processed signal is sent to a PC over ethernet or directly to a NI 9263 D/A converter.

The FPGA uses the $f_{sys} = 40MHz$ onboard clock available on the compact RIO platform, and a common sampling rate, which is a fraction of f_{sys} for the A/D and the D/A converters. As audio is the intended input, a sampling rate of $f_{sys}/1814 \approx 22050Hz$ is used for the system. The word length of the system is 16 bits.

The model under analysis is packaged as a CLIP called FPGAFX and is placed between FIFOs to provide protection from jitter in communication both with the I/O module and the CPU.

The analog output is connected to an oscilloscope and a speaker to provide both visual and audible feedback.

Configuration of the FPGA parameters are controlled from the CPU on the compact RIO.

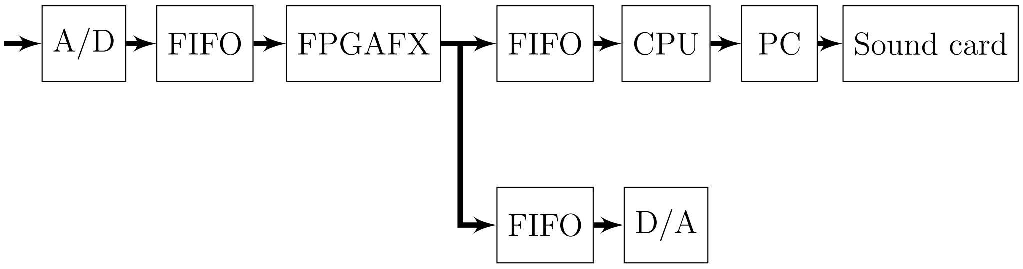

Figure 5: Test environment

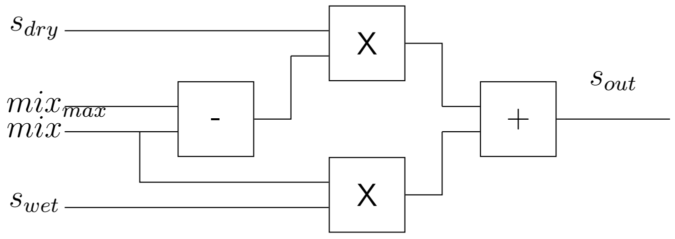

Figure 6: Mixer

3.2 Mixer

The purpose of the mixer is to mix two audio sources, in this case the unprocessed (dry) signal with the processed (wet) signal. Assuming the mix signal has a range $[0 : mix_{max}[$ the mixed output signal can be described as $s_{out} = s_{wet} \cdot mix + s_{dry} \cdot (mix_{max} - mix)$.

For the fixed-point implementation mix is implemented as a 8-bit unsigned number without a decimal part, and therefore $mix_{max} = 1$. Using $mod((2^n - 1) - mix, 2^n) = mod(2^n - 1 + \overline{mix} + 1, 2^n) = \overline{mix}$ the equation can be simplified to $s_{out} = s_{wet} \cdot mix + s_{dry} \cdot \overline{mix})$. The original implementation is shown in *Figure 6* on page 13

3.3 Delay

The delay model shown in *Figure 7* on page 14 is implemented as a chain of delay elements, each delaying the signal k cycles. The signal is tapped off between every shift register, and a multiplexer selects which tap to output, thereby controlling the delay time. The algorithm can be described as $s_{out}[t] = s_{in}[t - k \cdot delay \cdot T_s]$.

The maximum delay time is limited by the available FPGA resources that can be used for the delay chain, and was found to be around 12000 elements, resulting in a maximum delay of $\approx 0.5s$ for the given sampling rate. The

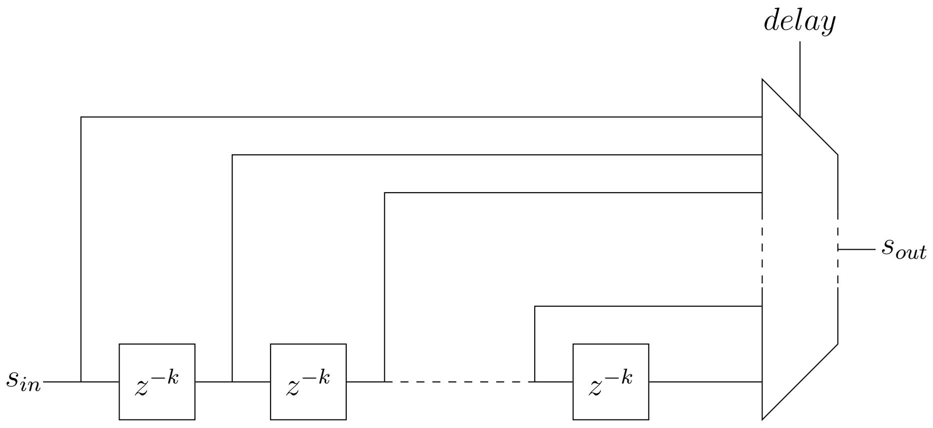

Figure 7: Delay

delay parameter, controlling the multiplexer size was set to 32 as a trade-off between resource usage and sufficiently fine-grained control over the delay. Since we want the k parameter to be a integer multiple of 16, for reasons explained in Section 4.1, this was set to 384.

3.4 Tremolo

The tremolo effect is accomplished by amplitude modulating the signal with a low frequency carrier, usually a sine wave or a saw tooth. The speed of the tremolo is controlled by changing the frequency of the carrier as in $s_{out][t]} = s_{in}[t] \cdot carrier[i]$, where $i[t] = mod(i[t-1] + speed, size_{LUT})$. The carrier signal is implemented as a precalculated look-up table containing a full period sine wave. The value of the parameter speed is added to the current address of the look up table, to control the frequency of the sine wave. The block diagram is shown in *Figure 8* on page 15

3.5 Tubewarmth

The purpose of the Tubewarmth model is to emulate the effects on a signal that is passed through a vacuum tube. Vacuum tubes are nonlinear and begin to saturate as the amplitude of the input signal increases. This is commonly used to introduce harmonic distortion in music [1]. Emulating a tube can be a very complex operation, and it is therefore common to use a simpler model ignoring frequency related characteristics, and instead concentrating only on the amplitude relationships of the input and output.

This implementation is based on the TAP Tubewamth plugin [7], written

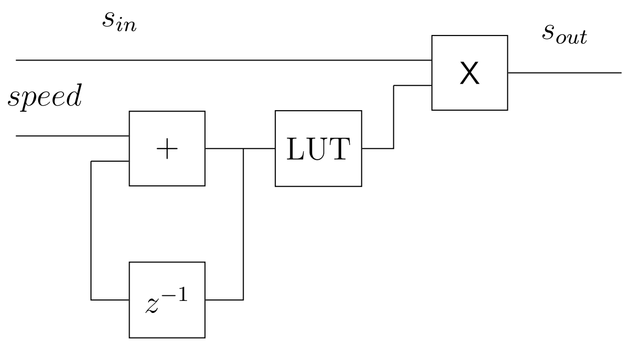

Figure 8: Tremolo

in C. It has been converted into a Simulink model and modified to be better suited for FPGA implementation. The original C code is available from `http://sourceforge.net/projects/tap-plugins` under GPL. The single largest modification comes from using fixed values for blend and drive, and thereby being able to use precalculated values for the internal parameters. Also fixed-point arithmetic is used instead of the original floating point implementation.

3.6 Control

The control block is a interface for setting parameters in the FPGAFX system. To minimize the amount of control objects on the front panel, only a single 16 bit vector is used to enter data, write enable and address. *Table 1* on page 15 describes the bit assignment. The address space is described in *Table 2* on page 15

	Write enable	**Address**	**Data**
Bits	15	10-8	7-0

Table 1: Structure of the configuration vector

Address	**Name**	**Range**	**Description**
0	select	0-2	Selects which effect to use 0=Delay 1=Tremolo 2=Tubewarmth
1	mix	0-255	Controls balance between dry and wet signal
2	tremspeed	0-255	Controls the speed of the tremolo effect
3	delay	0-31	Controls the delay time of the echo effect

Table 2: Address space of the Control component

4 Analysis

During the conversion of FPGAFX several complications were brought to attention. Some of these were solved by a slight modification of the model, or by changing implementation parameters. Others prevented the conversion from being executed automatically and required manual intervention to solve. This section describes the specific problems and solutions for the FPGAFX model, giving a context for some of the general problems that can arise in the conversion process.

4.1 Delay

The delay model consists primarily of memory capable elements. The number of elements grow linearly with the length and width of the delay chain. As there are several different types of memory capable elements in an FPGA, such as RAM, flip-flops and shift registers, different implementations can produce vastly different results regarding resorce usage.

By default Simulink HDL coder implements the Integer Delay block as *Listing 2* on page 16 which is a hardware independent behavioural description of a shift register with an asynchronous reset signal. Using XST from ISE 9.2, which is bundled with LabVIEW 8.6.1, this is synthesized as flip-flops. The required resource usage on a Spartan 3[4] is $width \cdot depth$ flip-flops. With two flip-flops per slice this requires at least $width \cdot depth/2$ slices.

Setting the ResetType parameter to none as in *Listing 1* on page 16 in the control file, produces the code in *Listing 3* on page 17. This gives freedom to the synthesis tool to use the hardware shift registers available in the FPGA fabric. Still, XST first recognizes this as flip-flops and then spends a long time optimizing before implementing this as a mixture of flip-flops and shift registers of varying length.

```
c.forEach('fpgafx/fpgafx/delay',...
  'hdldefaults.IntegerDelayHDLEmission', {'ResetType', 'none'})
```

Listing 1: Excluding the reset signal from Integer Delay

By black-boxing and explicitly instantiating chains of SRLC16E components, which are native 16 bit shift registers in the Spartan 3 architecture, both compile time and resource usage are drastically reduced, with the downside of losing the hardware independence. When the hardware is known, this implementation is considered the best. The final implementation is illustrated in *Listing 4* on page 23 and uses $width \cdot depth/16/2$ slices.

```
--data_i : input  std_logic_vector(WIDTH-1 downto 0)
--data_o : output std_logic_vector(WIDTH-1 downto 0)
```

```
--data   : signal array(0 to DEPTH-1) of std_logic_vector(WIDTH-1 downto 0)
process (clk,rst)
begin
  if rst = '1' then
    data <= (others => (others => '0'));
  elsif rising_edge(clk) then
    if ce = '1' then
      data <= data(DEPTH-2 downto 0) & data_i;
    end if;
  end if;
end process;
data_o <= data(DEPTH-1);
```

Listing 2: Behavioral description of a shift-register with reset

```
--data_i : input  std_logic_vector(WIDTH-1 downto 0)
--data_o : output std_logic_vector(WIDTH-1 downto 0)
--data   : signal array(0 to DEPTH-1) of std_logic_vector(WIDTH-1 downto 0)
process (clk,rst)
begin
  if rising_edge(clk) then
    if ce = '1' then
      data <= data(DEPTH-2 downto 0) & data_i;
    end if;
  end if;
end process;
data_o <= data(DEPTH-1);
```

Listing 3: Behavioral description of a shift-register without reset

The resource usage of the different cases is summarized in *Table 3* on page 17

	Available	**SRLC16E**	**Default**	**No reset**
Number of Slices	20480	7700	35457	17200
Number of Slice Flip Flops	40960	20	55153	23275
Number of 4 input LUTs	40960	15358	30434	28868

Table 3: Resource usage of the delay block with different implementations

4.2 Tremolo

When implementing the Tremolo block in Simulink HDL Coder, the original idea was to use the built-in Sine lookup table, which takes advantage of quarter-wave symmetry. This was not possible due to a flaw in the HDL generation that prevented a signed and an unsigned number in the internal representation of the Sine lookup table block. Instead an ordinary lookup table was used. In this case this was not a problem since the differences in resource usage was relatively small. This, however, highlights some of the unexpected problems that one can come across, and have to work around, in the conversion stage.

4.3 Tubewarmth

The tubewarmth model uses several arithmetic functions such as multipliers, adders and a square root. The square root especially has several alternative implementations that vary in area, speed, accuracy and latency.

Using the tubewarmth block created from the Simulink HDL coder model was not possible to do on the Spartan 3 FPGA, due to insufficient resources. Aside from using a large amount of the available logic, the tubewarmth block uses all available 18x18 multipliers, leaving no resources to the LabVIEW generated HDL code.

As a first attempt to solve this, the logic for generating the internal parameters was replaced with precalculated constants. Also, the internal signal paths were made as narrow as possible without losing noticable audio quality. Investigation of the RTL schematics and build logs revealed that the square root blocks was responsible for the largest part of the resource usage. Simulink HDL coder uses the multiply/add algorithm and produces combinatorial code. This has the advantage of being the closest to a model written for software, but does not incorporate well into the pipelining methodology used in FPGA/ASIC design. It also uses a lot of chip area.

The solution to this was to black-box the Tubewarmth sub block and instead build an equivalent model using Xilinx System Generator. Models created with Xilinx System Generator is targeted at a specific FPGA, and can therefore take advantage of the hardware resources in a more efficient way. Also, many of the implementations are based on iterative CORDIC algorithms, which can save considerable amounts of logic.

For reference, both implementations were synthesized for Spartan 3 outside of the LabVIEW environment. The relevant resource usages are summarized in *Table 4* on page 19

4.4 CLIP

The integration of the model as a CLIP has some limitations. Most of the limitations are intentional to easily fit the CLIP into the LabVIEW design flow, and to hide low level FPGA behaviour.

- CLIP cannot access FPGA pins directly

	Available	**HDL Coder**	**System Generator**
Number of Slices	20480	10173	2354
Number of Slice Flip Flops	40960	107	3852
Number of 4 input LUTs	40960	18914	3856
Number of 18x18 Multipliers	40	40	15

Table 4: Resource usage for different implementations of the Tubewarmth model. Notice that the HDL Coder implementation uses less Flip Flops due to its combinatorial implementation

- Only Boolean, U8, U16 and U32 can be used for passing data
- Not possible to use constraint files
- Not possible to change build parameters
- No way to analyze the RTL code generated from the block diagram

The restriction on data types forces in some cases the designer of the designer to create a wrapper to cast top entity signals to one of the supported formats.

The inability to use constraint files can be problematic in cases where the FPGA designer want to use multi cycle paths or ignore timing constraints completely. Together with the inability to control build parameters, this leaves the build process to use the settings provided by LabVIEW. If a compilation fails because of a timing violation, there is not much a designer can do more than redesign the IP, and for example use pipelining on signal paths where applicable.

When the HDL code from the components on the block diagrams is generated, these files are encrypted and sent directly through the ISE toolchain. This prevents any analysis and profiling of the LabVIEW-generated code, and the only way to see the final design is through the FPGA Editor. This, however is a time consuming way to reverse-engineer an unknown design.

5 Conclusions

The ability to use existing Simulink models or do rapid prototyping from scratch with an integrated test environment can speed up development. Simulink HDL coder and Xilinx System Generator are helpful tools but they both have limitations, that prevents the process from becoming fully automatic. If one wishes to use Simulink for models targeted to FPGA implementation, this should be considered early in the design flow to minimize manual intervention as much as possible. Still, the ability to black box some parts of the design is a necessary feature because of the limitations Simulink HDL Coder currently contains.

With Simulink HDL Coder, implementation of many blocks fails to take advantage of dedicated hardware resources. In the delay model, the SRLC16E primitives were far from optimally utilized when synthesized with XST. This could be compared with other synthesis tools such as Synopsis Synplify pro, but as there was no access to other tools this was considered to be outside the scope of this thesis.

Another problem with Simulink HDL Coder is the choice of block implementations that highlight the inherent differences between software and hardware modelling. Many designs require non-uniform step sizes which prevent them from being used, at least directly, as a clocked design. The models that can use fixed step-sizes most probably contain blocks that need one or none cycles to execute, after which their state is updated to the next iteration. An equivalent HDL implementation often produces large combinatorial logic nets, which is badly suited for FPGA. This can be seen in the Tubewarmth model where the square root block and the multipliers were so large that they consumed five times as many LUTs as the System Generator implementation.

A fundemental reason for this is that Simulink HDL Coder provides no built-in handshaking mechanism telling the blocks when the previous block is done calculating and the next are ready to input data. LabVIEW FPGA and System Generator uses handshaking, which allows the blocks to consume several cycles while still maintaining the data.

There were also more subtle failings of the HDL generation in Simulink HDL Coder , such as the inability to use the sine lookup table. Limitations such as these hinders the workflow when working with Simulink HDL Coder.

On the other hand, Xilinx System Generator requires a manual conversion to the supported blocks. This means that an extra conversion step is introduced, requiring another round of verification. But once this is done, the close integration with the rest of the Xilinx platform and the behind-the-scenes knowledge makes this a powerful tool for resource-efficient implementations. The downside in this case is that it is not as easy to implement in an existing design as Simulink HDL coder, and it is directly targeted towards Xilinx hardware.

Both tools are primarly used for DSP programming. Other types of systems may still be easier to implement by hand, and it is therefore a good thing that there are ways to incorporate external HDL Code in the design flow.

Once the generated IP is in the LabVIEW FPGA environment it is easy to incorporate it in the LabVIEW design flow, but there are also more potential problems. LabVIEW FPGA 8.6.1, which is the current version at this time of writing, uses Xilinx ISE 9.2 as a backend for building the FPGA load file. ISE 9.2 is a worse performer than ISE 10.1, and as ISE 11 is recently released, it is even more dated. Also, what LabVIEW FPGA gains in simplicity, it also loses in control. There is no way to specify constraints or setting parameters for the Xilinx tool chain. Every timing violation makes the build process fail. Together with the inability to analyse the LabVIEW generated code, due to the encryption of the HDL files, this makes building for a LabVIEW supported FPGA targets a trial-and-error process, when the limits are pushed. Still, LabVIEW FPGA is a powerful platform for rapid prototyping, and with the CLIP system, designs can easily be moved to stand-alone FPGA with a different I/O layer if needed.

The toolchain tested is very useful for implementing DSP systems from Simulink on FPGA. The simulation and verification tools are also of great help, as it minimizes the amount of redundant test benches that needs to be written, which also mimimizes the need to verify the verification environment more than necessary. There are however some rough edges to this, and the ability to use hand-written HDL code is still required. The toolchain will not replace the manual work as cutting edge designs need the options to finetune constraints and analyze every part of the design. On the other hand, this is not the target application. The tools are developed to give system designers and LabVIEW programmers the possibility to use FPGA in their design flow. Experienced FPGA programmers can then provide them with high performance IP cores that are black boxed for LabVIEW.

During the writing of this paper I also learned about Synplify DSP, with a scope similar to that of System Generator and Simulink HDL Coder. Unfortunately there was no access to the tool.

References

[1] Merlin Blencowe, *Chapter 1: The common cathode, triode gain stage*, `http://www.freewebs.com/valvewizard1/Common_Gain_Stage.pdf`.

[2] National Instruments Corporation, *About RIO Technology*, `http://www.ni.com/fpga/rio.htm`.

[3] The MathWorks Inc, *Simulink HDL Coder product overview*, `http://www.mathworks.com/products/slhdlcoder`.

[4] Xilinx Inc, *Spartan-3 FPGA Family Data Sheet*, `http://www.xilinx.com/support/documentation/data_sheets/ds099.pdf`.

[5] National Instruments Corporation, *Importing External IP into LabVIEW FPGA with the CLIP Node*, `http://zone.ni.com/devzone/cda/tut/p/id/7444`.

[6] Khalid Sayood, *Introduction to data compression*, 2000, Academic press.

[7] Tom Szilagyi, *TAP Tubewarmth*, `http://tap-plugins.sourceforge.net/ladspa/tubewarmth.html`.

[8] Xilinx Inc, *System Generator for DSP*, `http://www.xilinx.com/tools/sysgen.htm`.

A Code listings

```
entity srlc16e_vec is
  generic (width : integer := 20);             --Data path width
  port (
    d   : in  std_logic_vector(width-1 downto 0);
    ce  : in  std_logic;
    clk : in  std_logic;
    q   : out std_logic_vector(width - 1 downto 0));
end srlc16e_vec;

architecture rtl of srlc16e_vec is
  --Declare the native 16 bit shift registers of a Spartan 3
  component SRLC16E
    port (
      d   : in  std_logic;
      ce  : in  std_logic;
      clk : in  std_logic;
      a0  : in  std_logic;
      a1  : in  std_logic;
      a2  : in  std_logic;
      a3  : in  std_logic;
      q   : out std_logic;
      q15 : out std_logic);
  end component;
begin
  --Generate a 20 bit wide Shift registers.
  g_srl_w: for i in 0 to width - 1 generate
    i_srl : SRLC16E port map(d(i),ce,clk,'1','1','1','1',open,q(i));
  end generate g_srl_w;
end rtl;
```

Listing 4: Hardware specific description of a shift-register

```
function c = fpgafx_ctrl

c = hdlnewcontrol(mfilename);

c.generateHDLFor('fpgafx/fpgafx');

%Don't generate HDL code for delay model. This block will be replaced
%by hand written HDL code
c.forEach('fpgafx/fpgafx/delay',...
      'built-in/SubSystem', {}, ...
      'hdldefaults.SubsystemBlackBoxHDLInstantiation',...
      {'AddClockPort', 'on', 'ClockInputPort','clk',...
      'AddClockEnablePort', 'on','ClockEnableInputPort','ce'...
      'AddResetPort','off',...
      'InlineConfigurations','off'});
```

```
%Don't generate HDL code for tubewarmth model. This block will be replaced
%by a model created in Xilinx System Generator
c.forEach('fpgafx/fpgafx/tubewarmth_sysgen', ...
    'built-in/SubSystem', {}, ...
    'hdldefaults.SubsystemBlackBoxHDLInstantiation',...
    {'AddClockPort', 'on', 'ClockInputPort','clk_1',...
    'AddClockEnablePort', 'on', 'ClockEnableInputPort','ce_1',...
    'AddResetPort','off',...
    'InlineConfigurations','off'});

%General options. Only a few are shown as example
c.set( ...
        'AddInputRegister',             'on',...
        'AddOutputRegister',            'on',...
        'AddPipelineRegisters',         'off',...
        'CheckHDL',                     'on',...
        'ClockEnableInputPort',         'ce',...
        'ClockEnableOutputPort',        'ce_out',...
        'FIRAdderStyle',                'linear',...
        'TargetLanguage',               'VHDL');
```
Listing 5: Control file for FPGAFX model