Audio Engineering Society

Convention Paper

Presented at the 126th Convention
2009 May 7–10 Munich, Germany

Implementing Audio Algorithms and Integrating Processor-Specific Code using Model-Based Design

Arvind Ananthan[1], Mark Corless[2], and Marco Roggero[3]

[1] The MathWorks, Inc., 3 Apple Hill Drive
Natick, MA 01760-2098, USA
Arvind.Ananthan@mathworks.com

[2] The MathWorks, Inc., Crystal Glen Office Centre,
39555 Orchard Hill Place, Suite 280, Novi, MI 48375, USA
Mark.Corless@mathworks.com

[3] The MathWorks GmbH, Friedlandstr 18
D-52064 Aachen, GERMANY
Marco.Roggero@mathworks.com

ABSTRACT

This paper explores implementation of an audio algorithm on a fixed-point embedded processor using Model-Based Design. Once the algorithm, a 3-band parametric equalizer in this example, is designed and simulated using a combination of scripting and graphical modeling tools, embeddable C-code is automatically generated from this model. This paper illustrates how algorithmic C-code generated from such a model in Simulink can be integrated into the parent stand-alone embedded project as a library and implemented on an Analog Devices Blackfin® 537 processor. It also elaborates how processor-specific C-callable assembly code can then be integrated into the model for both simulation and code generation to improve its execution performance on this processor.

1. INTRODUCTION

Implementing fixed-point audio algorithms on embedded processors is a challenging task that could involve many teams and many different steps. Apart from the time it takes to completely develop an algorithm, convert to and test it in fixed-point, manually hand-coding these algorithms into C and assembly for implementation on to embedded processors and, subsequently, verifying their executions with original designs takes up a significant portion of the development cycle. In this paper, we explore the code generation concepts to implement an audio algorithm

and integrate it into a real-time application on an embedded processor.

Engineers can describe algorithms using textual or graphical techniques to model mathematical equations, signal flow, and state machines. Model-Based Design helps bring together these different modeling paradigms for efficient system development while providing a framework to specify and explore functional behavior, implement these specifications through C-code generation, and continuously test and verify the design against requirements [1][2].

We first explore how a design could be passed on to software engineer for implementation on an Analog Devices Blackfin processor. We are assuming that the designer has already created a fixed-point model of a Parametric Audio Equalizer [8] that has been tested and verified for correct behavior through real-time simulation on a PC. We explain how to configure the model such that the code generated from the algorithmic subsystem can easily be integrated into a parent embedded project.

Next we explore integrating C-callable libraries with Model-Based Design. We create a custom block that wraps an optimized Blackfin C-callable assembly function to replace the original filter blocks in the model. We verify the performance of the optimized implementation with the original design through processor-in-the-loop (PIL) testing, which enables co-simulation between Simulink and an IDE (VisualDSP++).

Finally, we compare how different model configurations and design choices affect performance by profiling execution times for the generated code. Profiling the model for execution statistic such as memory footprint and processor utilization helps with not only identifying candidate subsystems in the design for such replacements with optimized libraries but also in quantifying the benefits of such replacement.

2. AUDIO ALGORITHM MODEL

The audio algorithm we used for our example in this paper is a 3-band parametric equalizer. Each band is a biquad filter which can be tuned by specifying three parameters – center frequency, bandwidth and the amplitude. Such filters allow a precise control of the effective magnitude response of the overall system desired by the audio engineer. One common application of these filters is in compensating for the acoustics of the cabin environment of a car during the calibration phase while the audio system is installed by the acoustic engineer.

An effective prototyping platform should be flexible enough to graphically specify signal flow and state logic, as well as enable the designer to textually specify the algorithm. We have used a combination of the script based technical computing language, MATLAB® and the graphical modeling environment Simulink® to develop this model of the 3-band parametric equalizer. A graphical user interface to tune the parameters of the equalizer was also created using MATLAB that helps in tuning this 3-bands in real-time by writing new filter coefficients to the MATLAB workspace and uploading them to the executing Simulink model. A screenshot of this model and the GUI is shown in Figure 1.

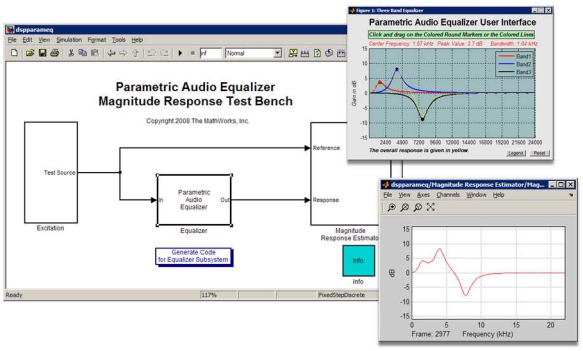

Figure 1: Simulink model of a 3-Band Parametric Audio Equalizer

This fixed-point model was created from the floating point version of the "Parametric Audio Equalizer" Simulink model [8] tested thoroughly on a PC in real-time. We assume the desired behavior has been achieved in simulation after the necessary model elaborations and iterative testing as described in [1], and use this fixed-point model as a starting point for the implementation steps described in this paper.

3. GENERATING CODE FOR A VISUALDSP++ PROJECT

Automatic code generation enables designers to quickly deploy their ideas to hardware to continue verifying performance on a real-time embedded system. When using automatic code generation tools during the development process, designers typically generate code from an algorithm model then integrate this code into the parent project. For example, Real-Time Workshop Embedded Coder can be used to generate C-code from a Simulink model. Specifications can be added to the model to customize the generated code to ease the code review and integration process. The generated files can then be hand integrated into an existing project [6]. Embedded IDE Link extends these capabilities and provides additional optimization, verification, execution profiling capabilities, and code integration features. In this section, we'll take advantage of the Embedded IDE Link feature to automate creation of a VisualDSP++ library to ease integration of the generated code into a pre-existing parent VisualDSP++ project.

In the following sections we will demonstrate how to configure a model to generate C-code and how to integrate the generated code into the parent project. We applied the following steps to accomplish this task:

1. Prepare the parent VisualDSP++ project
2. Specify the data interface in the model
3. Generate a Blackfin library from the model and profile execution
4. Integrate the generated library into the parent project

3.1. Prepare the Parent Visual DSP++ Project

We began with an audio "pass-through" VisualDSP++ project for an Analog Devices Blackfin processor. This stand-alone project implements the ADC and DAC device driver code for a Blackfin BF537 EZ-Kit board. Building and executing this project on the BF537 EZ-Kit board implements a direct feed through of the input audio through the processor and to the output. Integrating an algorithm code, automatically generated or manually hand coded, into this framework allows the engineer to quickly and easily deploy their algorithm in real-time while verifying its performance using real-world audio signals.

The purpose of this project is to verify that we can pass audio through the part. Also, this project configures the ADC/DAC such that its gathers frames of data in a double buffered model, with the frame (buffer) size specified as a parameter that can be configured by the user before compilation. The audio data from the double buffers were originally written to two separate variables corresponding to Left and Right channels. We modified this project such that both the left and right channel input data are concatenated and indexed off a single variable each for input (*frame_in*) and output (*frame_out)*. We did this as the algorithm we are integrating is developed as a frame based model in Simulink, which can handle multi-channel data – in our case, the stereo signal is dealt as a single 2-D signal in the model, which would correspond to a single variable in the generated code, with Left channel data followed by right channel.

We also had to match the frame size defined in the project to that of the model. An excerpt from the parent project is shown in Figure 2.

```
for (k=0; k<NUM_SAMPLES; k++)
{
    // convert 24-bit integer input to 16-bit fract16 format for proce
    frame_in[k] = (fract16)(iChannel0LeftIn[i+k] >> 8);
    frame_in[NUM_SAMPLES+k] = (fract16)(iChannel0RightIn[i+k] >> 8);

}
    // do frame-based processing here
    //dspparameq_fixPt_BF537_step();

for (k=0; k<2*NUM_SAMPLES; k++)
{
    //Feedthrough
    frame_out[k] = frame_in[k];
}

for (k=0; k<NUM_SAMPLES; k++)
{
    // convert 16-bit fract16 back to 24-bit and send it to output
    iChannel0LeftOut[i+k] = (unsigned int)frame_out[k] << 8;
    iChannel0RightOut[i+k] = (unsigned int)frame_out[k+NUM_SAMPLES] <<
}
```

Figure 2 Excerpt from audio pass-through Analog Devices VisualDSP++ project

3.2. Specify the Data Interface in the Model

In order for the generated code to tie into the feed-through VisualDSP++ project, the input and output signals of the algorithm model need to be configured to have the same names as the variables in the parent project, namely *frame_in* and *frame_out*, as seen in Figure 3.

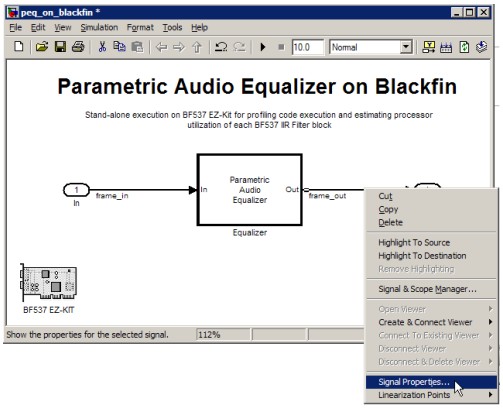

Figure 3 Simulink algorithm model for integrating into VisualDSP++ project showing steps for configuring the 'Signal Properties'

To accomplish this task of tying in the generated code, we specify the 'Storage class' parameter of the input and output signals as *Imported Extern* using the signal properties dialog as seen in Figure 4. This setting assumes that the parent project will declare the memory for these variables and the generated code just accesses them.

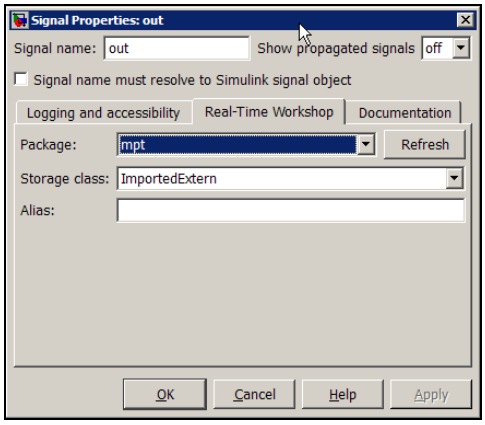

Figure 4 Selecting 'Imported Extern' storage class for output signal line

Also notice that in Figure 4 we set the 'Package' parameter to *mpt*. Module Packaging Technology (MPT) enables us to effect certain customizations in the generated code. For example, we can customize the comments that are inserted in the generated code, separate out the generated filter code as a header file, and specify the location of variables in the target memory. Partitioning coefficients into separate files enables other software components to access this data. For example, in a deployed application, the software engineer could schedule another software component to modify these variables at runtime before they are used by the main calling routine in the generated algorithm code.

We applied some of the MPT features to the coefficient variable specification in order to define and declare coefficient variables in separate source and header files (*biquad_coeffs.c* and *biquad_coeffs.h*) as shown in the in Figure 5. We also customized the parameter MPT object and the model to insert comments in the generated header files that corresponds to the design parameters of that filter [8].

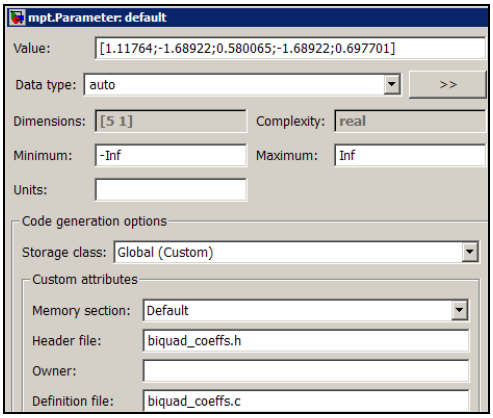

Figure 5 MPT Parameter object settings dialog

At this point, using the model we have elaborated thus far, we have generated ANSI C-code that is portable and could be hand integrated into a larger project [6].

3.3. Generate a Blackfin Library from the Model

In this section we will automate creation of a VisualDSP++ library. This library can then be included in the parent project and the calling function could be directly referenced after declaring the function header in this project. This approach minimizes the manual steps needed to include the generated code [5]. We will also customize the code generation process to make use of fixed-point intrinsic functions for the Blackfin processor [12].

The *Target Preferences* block, from the Embedded IDE Link library, provides access to the processor hardware settings required to generate a VisualDSP++ project. This block also provides the ability to define custom memory banks and placement of code and data sections into memory. We added a Target Preferences block to our model and selected the appropriate processor and session settings corresponding to the BF537 EZ-Kit board as shown in Figure 6.

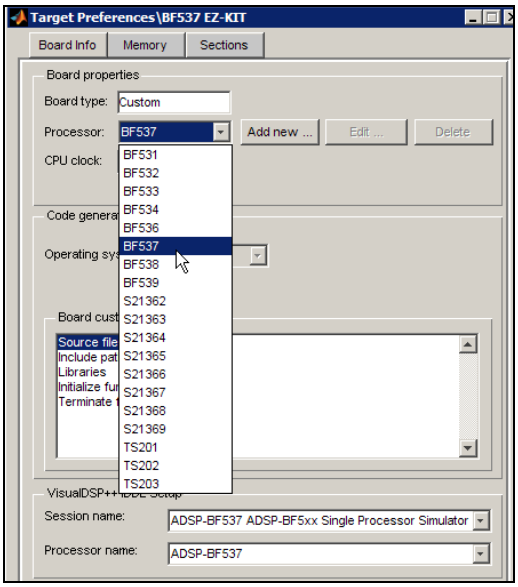

Figure 6 Target Preference block for configuring the processor and hardware settings

The Blackfin processor supports intrinsic functions for saturated fixed-point arithmetic which provide superior performance to writing equivalent routines in ANSI C. Real-Time Workshop Embedded Coder can be configured to generate calls to optimized fixed-point math routines using a Target Function Library (TFL) [11]. TFL provides the ability to control function and operator replacements in the generated code. One or more function replacement tables define the target-specific implementations of math functions and operators. The Embedded IDE Link provides function replacement tables for Analog Devices processors. In our model, we specified a TFL table for Blackfin 53x as shown in Figure 7.

Finally, we configured the model to directly create a VisualDSP++ library by using the 'Archived Library' build option. An excerpt from generated code – highlighting the TFL replacement is shown in Figure 8.

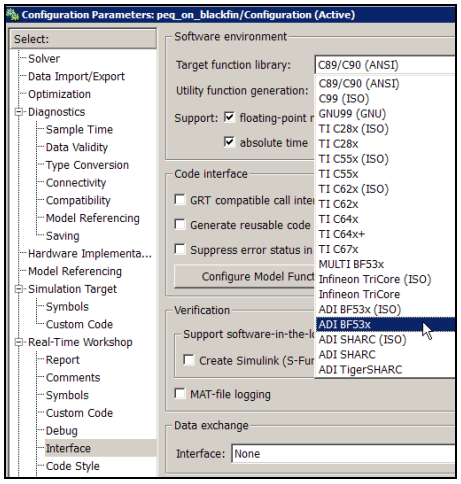

Figure 7 Selecting the TFL for Blackfin 53x

```
/*
 * operation:         s32 + s32
 * saturation:        Yes
 * rounding mode:     N/A
 */
inline int32_T bf53x_add_s32_s32_s32_sat(int32_T a, int32_T b)
{
  return add_fr1x32(a, b);
}
```

Figure 8 Target Function Library (TFL) code replacements for 16-bit saturation operations on Blackfin

Before we integrate this library into our parent project, it's desirable to verify that this library fits in terms of its execution performance. The execution profiling capabilities of the Embedded IDE Link allows a designer to determine the processor cycles taken to execute this library along with the maximum stack usage. This helps identify areas of optimization before the final integration steps are undertaken. We automated the profiling of the generated code and verified that the performance of this library was within acceptable bounds. We go into the details of this profiling process later in this paper.

3.4. Integrate the Generated Library into the Project

We manually added the library built from the previous steps to the project. Before building this project, it is necessary to include the header file generated in this process that declares the calling function as extern.

The library has two important points of entry: an initialization function and a step function. The initialization function should be called during the startup function of the parent project. State variables are initialized in this routine. The step function is typically called during the periodic execution of the algorithm, and hence, has to be integrated into that part of the code serviced by a timer or interrupt service routine.

The routine to be invoked from this library to execute the parametric equalizer algorithm is the function call *dspparameq_fixPt_BF537_step()* - this was used within the calling function in the feed-through project as shown in Figure 9. This function also needs to be declared in one of the header files in the project [5].

```
for (k=0; k<NUM_SAMPLES; k++)
{
    // convert 24-bit integer input to 16-bit f
    frame_in[k] = (fract16)(iChannel0LeftIn[i+k
    frame_in[NUM_SAMPLES+k] = (fract16)(iChanne

}

// do frame-based processing here
dspparameq_fixPt_BF537_step();


for (k=0; k<NUM_SAMPLES; k++)
{
    // convert 16-bit fract16 back to 24-bit ar
    iChannel0LeftOut[i+k] = (unsigned int)frame
    iChannel0RightOut[i+k] = (unsigned int)fram
}
```

Figure 9 VisualDSP++ Project showing the function call invoking the parametric equalizer sub-function

To avoid having to manually open the parent project and recompile each time we changed the model, we could create a MATLAB script to automate the compilation of the library as well as linking and downloading it into the parent project when we generate code from the model.

4. CREATING CUSTOM BLOCKS TO INTEGRATE PROCESSOR-SPECIFIC CODE

Once the generated code has been integrated into the parent project and ran successfully on the target processor, it's useful to gather the execution statistics of this algorithm such as memory footprint, and processor utilization. This will help the designer identify areas of improvement in the algorithm.

If a bottleneck is identified, the designer may make a change at the algorithmic level in the model or in some cases may want to integrate processor-specific C-callable optimized routines. The designer could manually replace parts of the generated code with this custom code, but such an approach breaks the link between the generated code and the original model. This makes it difficult to reuse the original models throughout the development process. Maintaining a link between the model and implementation code is an important aspect of Model-Based Design that enables continuous verification of the design throughout the process. To maintain this link, designers can create custom blocks which call out to these optimized routines in the generated code.

There are a variety of ways to create custom blocks in Simulink which supports both simulation and code generation [10][13][11][14]. In the following sections, we provide an overview of the workflow we applied to create a custom filter block. This custom block calls the optimized Blackfin IIR filter *iirdf1_fr16* in the generated code. We will focus on this workflow at a high level, and publish the results we obtained in the next section. We applied the following workflow to create this block.

1. Create a block for code generation to call the processor-specific code
2. Specify a functionally equivalent block for simulation
3. Create a mechanism to switch between simulation and code generation blocks
4. Verify the simulation and code generation behavior

4.1. Create Code Generation Block

The Blackfin IIR function we have chosen to integrate is *iirdf1_fr16*. This function prototype for *iirdf1_fr16* is shown in Figure 10.

```
typedef struct
{
   fract16 *c;     /* coefficients                  */
   fract16 *d;     /* start of delay line           */
   fract16 *p;     /* read/write pointer            */
   int k;          /* 2*number of stages + 1        */
} _iirdf1_fr16_state;

#define iirdf1_init(state, coeffs, delay, stages) \
    (state).c = (coeffs); \
    (state).d = (delay);  \
    (state).p = (delay);  \
    (state).k = (2*(stages)+1)

#pragma linkage_name __iirdf1_fr16
    void iirdf1_fr16 (const fract16 _input[],
                      fract16 _output[], int _length,
                      _iirdf1_fr16_state *_filter_state);
```

Figure 10 *iirdf1_fr16* function prototype

Note that the coefficients are passed as an array of *fract16* data type within a structure. For the Blackfin, the *fract16* data type is defined as a 16-bit signed integer and represents a fixed-point number with a fraction-length of 15 bits. We created a utility function to convert filter coefficients designed in MATLAB to an array of integer values which can be passed to *iirdf1_fr16*.

We used the Legacy Code Tool (LCT) to automate creation of a block which will specify code generation behavior. The LCT is a MATLAB script-based tool to wrap custom C-code or C-callable code into Simulink for both simulation and code generation [7]. An excerpt of this M-code we used is shown in Figure 11.

Because the *iirdf1_fr16* routine is written in assembly code for the Blackfin processor, this code can not be compiled for the Simulink simulation environment. If we had functionally equivalent ANSI C-code for the *iirdf1_fr16* routine, we could have configured the LCT to compile the ANSI C-code for simulation and make a call out to *iirdf1_fr16* in the generated code. Since we did not have a functionally equivalent piece of ANSI C-code available, we created a "dummy" function which just passed the input to the output. Hence, the resulting Simulink block will pass the signal in simulation, but call out to the *iirdf1_fr16* routine in the generated code.

```
%% Define function prototypes
def = legacy_code('initialize');
def.SFunctionName = 'iirdf1_fr16_sfun';
def.InitializeConditionsFcnSpec = [...
    'void iirdf1_init('...
        'iirdf1_state_fr16 work1,'...          % state
        'int16             p1[],'...           % coeffs[]
        'fract16           work2[size(u1,1)],'...  % delay []
        'int32             p2)  ');            % stages
def.OutputFcnSpec = [...
    'void iirdf1_fr16('...
        'fract16           u1[],'...           % input[]
        'fract16           y1[size(u1,1)],'... % output[]
        'int32             size(u1,1),'...     % length
        'iirdf1_state_fr16 work1[1])'];        % iir_state_fr16
```

Figure 11 Excerpt from LCT script to specify code generation behavior

4.2. Create Simulation Block

In this section we will describe how we specified simulation behavior for the custom block. Ideally, the simulation behavior should be equivalent to the behavior of the generated code compiled and executed on the target processor. If some deviation is acceptable, then it is important to identify, quantify, and verify where these deviations exist. This is especially true when creating a library block used by multiple engineers. Ensuring accurate simulation and target behavior enables engineers to detect many design errors in the simulation environment before implementing the design in hardware - one of the key benefits of Model-Based.

As described in the previous section, if we had functionally equivalent ANSI C source code for the *iirdf1_fr16* routine, we could have directly applied the LCT to create a block to specify both the final simulation and code generation behavior. In many cases, a designer will only have the processor-specific code available and does not have the resources or desire to rewrite the code in ANSI C. It is often easier to specify the algorithm behavior using a variety of techniques within the Simulink environment including using existing blocks or writing Embedded MATLAB code [15].

For our example, we used the *Biquad Filter* block from the Signal Processing Blockset to specify simulation behavior. The Biquad block supports specification of fixed-point attributes. We specified the data types to represent the filter internals similar to that of the Blackfin IIR library.

It is important to note that the configuration for the Biquad block will provide behavioral results which are numerically very close (in the order of 10^{-4}), but not identical to the *iirdf1_fr16* routine. This is because although we can configure the Biquad block to be a fixed-point Direct Form 1 structure, the actual implementation of the *iirdf1_fr16* is slightly different. For example, the *iirdf1_fr16* routine expects the denominator coefficients to be the negative of the denominator coefficients used by the Biquad filter.

We chose to use the Biquad filter block because it was quick to configure and create a block for which we deemed the simulation performance as adequate. If we required bit-true simulation performance, we could exactly specify the fixed-point mathematics of the *iirdf1_fr16* using low level Simulink blocks or by writing Embedded MATLAB code.

4.3. Automate Block Selection

In the previous sections we described how to create two blocks. The first block acted a pass-through for simulation and calls the *iirdf1_fr16* routine in the generated code. We will refer to this as the "Code Generation Block". The second block acts as filter whose simulation response is very close to that of the *iirdf1_fr16* routine. We will refer to this as the "Simulation Block." In this section we will describe how we automated selection between these blocks for simulation and code generation.

To accomplish switching between the "Simulation Block" for simulation and the "Code Generation Block" during code generation, we leveraged the RTW Environment Controller block. The RTW Environment Controller block is provided with the product Real-Time Workshop and enables designers to specify different behavior for simulation and code generation.

Finally, we created a Simulink library block which contained the above elements. This library block and it's implementation using the "Code Generation Block, "Simulation Block" and RTW Environment Controller block are shown in Figure 12.

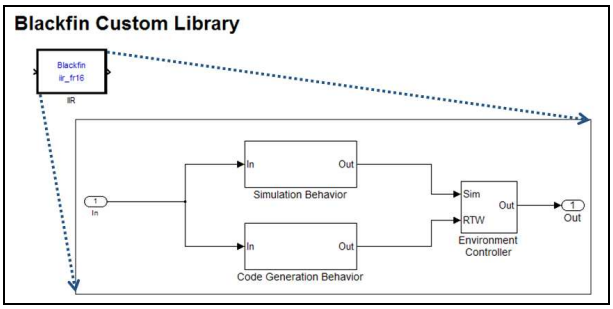

Figure 12 Example Simulink library block defining Simulation and Code Generation behavior

4.4. Verify the Simulation and Code Generation Behavior

One of the key benefits of Model-Based Design is the ability to ensure accurate simulation and target behavior. This enables engineers to detect many design errors in the simulation environment well before implementing the design in hardware. In the following sections, we'll verify the behavior of the *iirdf1_fr16* custom block on the Blackfin using the processor-in-the-loop (PIL) testing.

A typical verification task involves exporting test vectors from the host simulation environment and importing them into the target integrated development environment. PIL automates this task by enabling co-simulation between the Simulink model and VisualDSP++ IDE. For each simulation step, the Simulink model drives the execution of the VisualDSP++ project to feed test data, execute the algorithm on the processor, and pull back the processed results for comparison. A conceptual view of PIL testing can be seen in Figure 13.

Using this approach we were able to verify that the output of this Parametric Audio Equalizer running on the processor gave expected results compared to the output of the Simulink Biquad filter when fed in the same input vectors from Simulink [9].

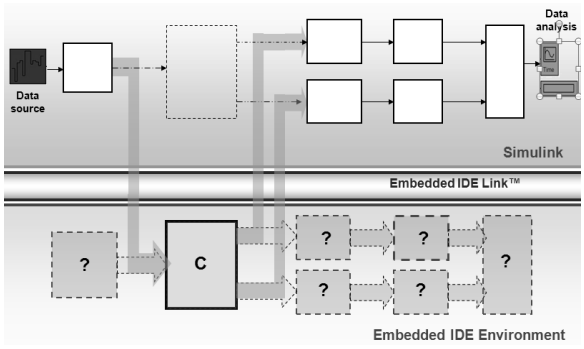

Figure 13 A conceptual view of processor-in-the-loop testing

5. REAL-TIME EXECUTION PROFILING

Execution profiling, stack profiling, and RAM/ROM analysis are common techniques to verify that the generated code meets resource requirements on the target. RAM/ROM usage can be obtained from the memory map file generated during the build process. Embedded IDE Link enables automation for collecting and reporting execution time and maximum stack usage. Based on analysis of the resource usage statistics, the designer can gauge whether the performance of the model at different stages of the elaboration process is acceptable or not. Typically during this elaboration, the designer will trade off behavioral performance versus resource usage [1]. The designer could also stop the elaboration process early on if the performance within are acceptable processor utilization limits. Thus, over design of the systems could be avoided.

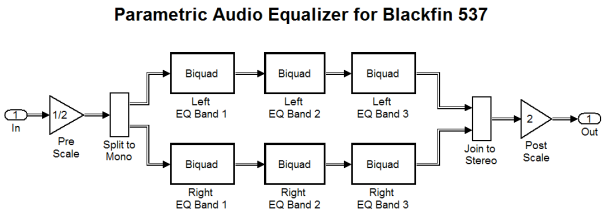

Figure 14 Screen shot of parametric audio equalizer for generating ANSI C-code

In this section, we focus on collecting profile execution times for different model configurations of the three-band Parametric Audio Equalizer. A screenshot of the Parametric Audio Equalizer model from which we generated C-code is shown in Figure 14. This model is configured to operate on 16-bit stereo audio inputs containing 512 samples per channel sampled at 48 kHz; hence the base rate is 10.67 msec. The primary components of the algorithm are a Pre Scale, six Biquad filters and a Post Scale. Filter coefficients and states used 16-bit fixed-point data types.

We configured the model to generate code for a variety of settings. We started with portable ANSI C-code and explored the effect of wrapping versus saturation. We then enabled TFL and configured it to generate Blackfin processor-specific intrinsic functions for fixed-point saturated arithmetic. We explored the effect of inlining and function call with this setting. Finally, we replaced the Biquad filter block with the custom iirdf1_fr16 block (described previously) and profiled the results. The results of these tests are shown in Table 1.

We first generated processor-independent portable ANSI C-code and profiled it to show the baseline performance for the filter function first using wrap and then using saturate. Designers prefer ANSI C-code if portability and platform independence is a key requirement. This way the code can be easily recompiled for deployment to different environments. The execution time for the Parametric Audio Equalizer on the Blackfin processor when configured to use wrap was around 60 µsec. When the same model was configured to use fixed-point saturation arithmetic, the generated code took almost 400 µsec to execute.

As we can see from the execution number, saturation can be an expensive operation if implemented in ANSI C on a DSP. Typically, better execution performance is obtained by using wrap instead of saturating arithmetic. However, this can also make the task of algorithm development harder. Often, designers can take advantage of processor-specific intrinsics to leverage the hardware features and get better performance without sacrificing the fidelity of the design obtained when using saturation. In this case, it's desirable to have a single model specification and leverage any fixed-point intrinsic capabilities of the processor. TFL enables the designer to create a single model and swap processor optimizations in and out for deployment to different processors. Using TFL, we were able to reduce the execution time of the algorithm (using saturation arithmetic) from 400 µsec to 120 µsec – a 70% improvement with just a change in the model settings.

Table 1 Comparison of Execution Times for Variants of the Parametric Audio Equalizer

Implementation	Biquad Filters		Execution Time** (µsec)
	Wrap / Saturate	Inline / Function	
ANSI C	Wrap	Function	60.106
	Saturate	Function	398.117
Blackfin 53x C-intrinsic*	Saturate	Function	121.587
		Inline	108.004
ASM *iirdf1_fr16*	Saturate	Function	87.813

** Using Target Function Library (TFL) for integrating Blackfin 53x C-intrinsic for fixed-point operations such as saturation*

*** Base rate of the code was 10.66 msec. Processor utilization in % is obtained by dividing the execution time by this base rate. Product version used for code generation and measurements is R2009a.*

We could get better numbers if we were to inline the code for the Biquad filters instead of using re-entrant functions. This requires trading off program memory size. Thus, when program memory space is a limitation, using re-entrant function calls can appreciably reduce program memory utilization. Using inlined code, we were able to improve the performance of our algorithm to about 108 µsec.

Finally, we investigated the performance gains we could achieve by integrating the Blackfin specific *iirdf1_fr16* filter as a custom block as described in the previous section. As described in the previous section, this block is functionally similar, but not identical used in the previous examples that generated ANSI C-code. Using the optimized *iirdf1_fr16* filter block, we achieved a performance of 88 µsec. By creating a block that is optimized for this processor, the model is no longer portable but achieves the best execution time for saturated arithmetic.

6. CONCLUSION

During initial stages of development of audio algorithms for embedded applications, designers often apply modeling tools to specify algorithms using graphical and textual techniques. Designers then simulate these models to explore the behavior of the algorithm. Once the behavior of these models is verified, designers can configure the algorithmic model to generate C-code which can be integrated into an embedded application. In this paper we demonstrated a technique to configure the model to generate code which can be called within the embedded application, as well as a technique to integrate existing C callable code into the modeling and code generation environment.

Specifically, we demonstrated how to configure a model of a 3-band parametric equalizer algorithm to generate C-code which can be integrated into the Analog Devices VisualDSP++ embedded development environment. To demonstrate this process, we automated creation of a VisualDSP++ library project for the parametric equalizer model and integrated this library into the parent feed-through project for an Analog Devices Blackfin BF537 processor.

Subsequently, we discussed how the designer can use execution profiling tools to identify areas for further optimization in the design. Such profiling could measure both memory footprint as well as processor utilization.

We then described how to integrate optimized C-callable libraries for specific sub-components, such as the IIR Biquad filters. We chose the optimized C-callable assembly routine for Blackfin processor that ships with VisualDSP++ as an example replacement, and also showed the resulting improvements in processor utilizations as a result of this replacement. We also compared the execution performance of the different variants of the audio parametric equalizer model using the profiling techniques described earlier.

Finally, to verify that the replacement with optimized C-callable libraries did not introduce any unexpected deviations in the results of the original algorithm, we detailed the technique of processor-in-the-loop testing to verify the behavior of the parametric equalizer (using the optimized library calls) executing on the DSP with the original simulation model by comparing their outputs when using the same input test vectors.

7. REFERENCES

[1] Corless, Mark and Ananthan, Arvind, "Model-Based Design of Fixed-Point Filters for Embedded Systems", Society of Automotive Engineers World Congress 2009-01-0150, April 2009

[2] Philips Consumer Lifestyle (Philips) Develops One-Piece Surround Sound System with MathWorks Tools, The MathWorks, April 2008: www.mathworks.com/company/user_stories/userstory17418.html

[3] Corless, Mark, and Ananthan, Arvind, "PC Based Prototyping of Audio Applications using Model-Based Design", AES 36th International Conference, Dearborn, Michigan, USA, 2009

[4] Real-Time Workshop Embedded Coder, Module Packaging Technology, The MathWorks: http://www.mathworks.com/access/helpdesk/help/toolbox/ecoder/mpf/f260641.html

[5] Ananthan, Arvind, "Integrating Simulink Model with VisualDSP++ Project," November 2008: www.mathworks.com/matlabcentral/fileexchange/22243

[6] Donovan, Mike, "Deploying Simulink Designs on Your DSP: An Accelerated Approach to Custom Implementation," MATLAB Digest, January 2006.: http://www.mathworks.com/company/newsletters/digest/2006/jan/simdsp.html

[7] Simulink, Legacy Code Tool, The MathWorks: http://www.mathworks.com/access/helpdesk/help/toolbox/simulink/sfg/bq4g1es-1.html#bq4g1es-3

[8] Signal Processing Blockset, Demos, Audio Parametric Equalizer: '*dspparameq.mdl*', in R2009a release of MATLAB.

[9] Ananthan, Arvind, "Integrating Blackfin specific optimized IIR Library into Simulink", April 2009: http://www.mathworks.com/matlabcentral/fileexchange/23236

[10] Corless, Mark and Reddy, Vinod, "Integrating Your Processor-Specific Code with Model-Based Design of Embedded Systems," GSPx 4th International Signal Processing Conference, October 2006.

[11] Real-Time Workshop Embedded Coder, Target Function Library, The MathWorks: http://www.mathworks.com/access/helpdesk/help/toolbox/ecoder/ug/brc_o1j-1.html

[12] Blackfin Compiler and C/C++ Library Manual, Analog Devices Inc.,

[13] Fielder, Jon, "Optimized Infineon TriCore Simulink Blocks for use with Link for TASKING": http://www.mathworks.com/matlabcentral/fileexchange/14069

[14] Real-Time Workshop, Target Language Compiler: http://www.mathworks.com/access/helpdesk/help/toolbox/rtw/tlc/bp6j4co.html

[15] Zarrinkoub, Houman, "Embedded MATLAB, Part 1: From MATLAB to embedded C", DSP Design Line: http://www.dspdesignline.com/howto/207800773