International Journal of Mechanical, Industrial and Aerospace Sciences
ISSN: 2517-9950
Vol:14, No:9, 2020

Seamless MATLAB® to Register-Transfer Level Design Methodology Using High-Level Synthesis

Petri Solanti, Russell Klein

***Abstract*—**Many designers are asking for an automated path from an abstract mathematical MATLAB model to a high-quality Register-Transfer Level (RTL) hardware description. Manual transformations of MATLAB or intermediate code are needed, when the design abstraction is changed. Design conversion is problematic as it is multidimensional and it requires many different design steps to translate the mathematical representation of the desired functionality to an efficient hardware description with the same behavior and configurability. Yet, a manual model conversion is not an insurmountable task. Using currently available design tools and an appropriate design methodology, converting a MATLAB model to efficient hardware is a reasonable effort. This paper describes a simple and flexible design methodology that was developed together with several design teams.

***Keywords*—**Design methodology, high-level synthesis, MATLAB, verification.

I. Introduction

MATLAB is the tool of choice in most algorithm development projects. Its built-in matrix operations, comprehensive toolboxes and advanced visualization capabilities provide a superior platform for complex algorithm development. Yet, the final implementations of the algorithms often require either a full hardware (HW) or software (SW) implementation with HW acceleration. The path from a floating-point MATLAB model to a working silicon is long.

The classical methodology, writing RTL code manually starting from the abstract floating-point MATLAB model, is very painstaking and error prone. Not only re-coding the algorithm from a sequential program to a HW architecture, but also maintaining the integrity between the very different models is difficult. The gap between the abstractions is too large for traditional tool flows. Usually such a project needs multiple intermediate models and a multi-disciplinary team with several engineers.

A new, simple design methodology is needed to improve the productivity and quality of the design. It should tackle both the design abstraction and validation problems and provide a flexible framework for different design needs. Also the SW implementation, HW/SW partitioning and late changes from HW to SW and vice versa must be taken in account.

The purpose of this study was to find a simple workflow that minimizes the number of required models, can be mastered by one or two engineers and maintains the design integrity throughout the process.

A major part of the study focused on handling the model abstraction. Designers with different backgrounds have different approaches to handle this. HW designers think in terms of clock cycles, registers and signals, whereas algorithm developers deal with matrix operations, toolbox functions and function parameters. SW developers like object orientation and complex data structures. At least one of these three abstraction levels must be discarded.

A good common denominator for all appeared to be the abstraction level of High-Level Synthesis (HLS) C++ that is syntactically close enough to MATLAB, but has also HW related features like arbitrary length data types and bit operations that give HW designers the possibility to influence the details. Embedded SW is usually written in C/C++, so all involved designer groups are taken into consideration. RTL code generated by HLS is supposed to be correct by construction.

HLS alone does not solve the problem, but it enables a design methodology with higher abstraction level throughout the process. This methodology is introduced by using a Direction of Arrival (DOA) Estimator model as an example [1].

II. Design Methodology

The proposed workflow consists of five steps:

A. Model analysis and partitioning to HW and SW
B. Rewriting the HW part functionality in HLS C++
C. Validating C++ model functionality in MATLAB
D. Quantizing and fine tuning HLS C++ model
E. Exploring different HW architectures using HLS

A. Model Analysis

There are many studies describing very comprehensive methodologies for model analysis and HW/SW partitioning. Some of them are targeted to reconfigurable systems [2], [3] and some other are more generic [4], [5]. These methodologies are usually too complicated for real-life design projects. In most cases, the model analysis can be reduced to interface traffic and resource requirement estimations that can be done in MATLAB.

The main purpose of a HW accelerator is to speed up the processing and decrease the processor load. Partitioning analysis should find the optimal interface between HW and SW and isolate the functionality of the accelerator. Many times the MATLAB model does not have a clear interface between the testbench and algorithm itself.

Interface traffic is a critical metric in HW/SW systems.

P. Solanti is with Mentor Graphics Deutschland, Munich, 80634, Germany (e-mail: petri_solanti@ mentor.com).
R. Klein is with Mentor Graphics, Wilsonville, OR 97070 USA (e-mail: russell_klein@mentor.com).

Massively parallel number crunching HW is idle, if the interconnect passing the data between the processes is congested. This is a common mistake that can be avoided by careful analysis.

In a MATLAB model the data traffic requirements can be estimated by analyzing the variable sizes and sample rates. The bigger the array, the more bandwidth and data transfer time is required. This can influence the efficiency of the acceleration by leading to unnecessary idle times. The data transfer time depends also on the interconnect type and interface architecture, so a rough overview of the desired HW architecture should be available at this time.

Estimating resource requirements of a MATLAB code segment or a function has two simple options. One is the number of operations in the code that can be estimated by counting all arithmetic and comparison operations multiplied by the loop iteration count in the MATLAB code. This becomes problematic when the algorithm uses toolbox functions with no source code available.

Another simple method is to run MATLAB simulation with timestamps to see how the execution time is split between the code segments. This method is not as accurate as the operation count, but it gives a good estimate and it also includes the toolbox functions.

The HW partition should be selected so that it can be parallelized to get higher throughput and the data traffic does not become a bottleneck. The best accelerator efficiency can be reached in code segments that have a fixed dataset processed multiple times in the loop that can be unrolled in HLS.

In addition to the interface and resource analysis we must analyze the required data types. A floating-point MATLAB model does not have any type definitions. All variables are double by default, whereas in synthesizable HLS C++ the variables have arbitrary length integer or fixed-point data types that are optimized for the value range of each variable.

In the model analysis phase, the variable value ranges are analyzed to assess which variables can be grouped together with the same datatype to reduce the number of required type definitions. These are maintained in a separate header file to enable smooth transition between floating-point and fixed-point later in the workflow. In this phase all datatypes are set to floating-point – either in double or ac_ieee_float64, which is a double equivalent in the open source Algorithmic C (AC) datatype [6].

B. Rewriting the HW Part Functionality in HLS C++

This part of the flow is the real model conversion. Writing a C++ model with the same functionality as the original floating-point MATLAB model can be a simple syntax conversion, if the original model is using only basic operations and no matrix arithmetic like the main loop of the polyphase filter example in Figs. 1 and 2.

MATLAB code with matrix operations or toolbox functions requires some more background work, because the functionality is hidden in the MATLAB language or library model. Open source HLS C++ implementations of matrix multiplication and several mathematical functions are available on the internet [7], which reduces the workload with the first design. Carefully written templated C++ class can be used as library IP in future projects.

```
if (stepCntr == 0)

    FIR1_SR = [in, FIR1_SR(1:6)];
    FIR2_SR = [in_SR(1), FIR2_SR(1:6)];
    FIR3_SR = [in_SR(2), FIR3_SR(1:6)];

    for i = 1:7
        tmpFIR10ut(:) = tmpFIR10ut + (FIR1_SR(i) * Coeffs1(i));
        tmpFIR20ut(:) = tmpFIR20ut + (FIR2_SR(i) * Coeffs2(i));
        tmpFIR30ut(:) = tmpFIR30ut + (FIR3_SR(i) * Coeffs3(i));
    end

    tmpOut(:) = tmpFIR10ut + tmpFIR20ut + tmpFIR30ut;
end
```

Fig. 1 MATLAB implementation of polyphase filter main loop

```
if (stepCntr == 0) {

    SHIFTLOOP: for (int j=6; j>=0; j--) {
                  FIR1_SR[j] = (j>0) ? FIR1_SR[j-1] : in;
                  FIR2_SR[j] = (j>0) ? FIR2_SR[j-1] : in_SR[0];
                  FIR3_SR[j] = (j>0) ? FIR3_SR[j-1] : in_SR[1];
    }

    FIRLOOP: for (int i=0; i<7; i++) {
                tmpFIR10ut = tmpFIR10ut + (FIR1_SR[i] * Coeffs1[i]);
                tmpFIR20ut = tmpFIR20ut + (FIR2_SR[i] * Coeffs2[i]);
                tmpFIR30ut = tmpFIR30ut + (FIR3_SR[i] * Coeffs3[i]);
     }

    tmpOut = tmpFIR10ut + tmpFIR20ut + tmpFIR30ut;
}
```

Fig. 2 HLS C++ implementation of polyphase filter main loop

Writing a missing function manually is like writing any helper function in a SW project. Interface, class hierarchy and datatype definitions are made in a similar way. Using templates to parameterize the data types, array dimensions and iteration counts helps in the next design steps and makes the model reusable.

When the helper functions are available, the main MATLAB code can be converted to C++. Some MATLAB constructs like built-in index loops and persistent variable definitions must be rewritten. Other language structures are close to C.

C. Validating C++ Functionality in MATLAB

One of the major problems that verification engineers are facing is the mismatch between the MATLAB reference model and the RTL implementation. In the traditional design flow, RTL designers make their own interpretation of the specification when they implement the RTL. Verification engineers build their own predictor based on the specification and end up debugging both RTL and MATLAB models, because the results do not match. To avoid this problem, the HLS C++ model must be validated against the MATLAB model. This can be done by using the MATLAB external C interface (mex) [8] and instantiating the C++ model as a mex function into the MATLAB testbench parallel to the original device under test (DUT). This might require an additional adaptation layer, if the original model is vector based and the C++ function has a streaming interface. Functionality is validated by comparing the outputs of both models.

The mex wrapper can be created manually or automatically with the wrapper generator of the HLS tool. The wrapper file must be modified if the signals in the DUT interface are

changed. Switching the port data types between floating-point and fixed-point does not require wrapper modification, because the MATLAB mex API does not support MATLAB fixed-point data types. Conversion from double to fixed-point type must be done in the adaptation layer or in the wrapper itself.

Test coverage is typically analyzed in the RTL verification. The same methodology can be applied in the C++ level to ensure that the testbench is testing the full implementation. High-level verification (HLV) provides the line and decision coverage data at the C++ level to ensure that the MATLAB test suite covers all corner cases handled by the C++ code.

D. Quantizing and Fine Tuning the HLS C++ Model

Fixed-point analysis and conversion, aka quantization, is a critical task in HW design. Every additional bit increases the design size, propagation delay and power consumption and missing bits increase noise or may cause an overflow.

Quantization methodologies are well known. There are many theoretical studies about quantization, but a fairly basic methodology [9] is suitable for the most designs. Signal-to-noise-ratio (SNR) based methodologies [10] give much better accuracy, but usually the effort is far too high compared to the benefit.

There are two categories of quantization methodologies: analytic and simulation based. Usually both methodologies are needed for fixed-point analysis of a system.

Analytic quantization methodology analyzes the number of required bits based on the input bit width and operation type. Addition, for example, adds one integer bit to the output word length to avoid an overflow in any case. More advanced methodologies also provide quantization noise modeling. The analytic method is independent of the testbench quality and gives reliable, but usually very conservative results.

Simulation based quantization analysis extracts the peak absolute values and non-zero minimum absolute values or minimum non-zero difference between two consecutive samples during the simulation. This requires a pre-quantized input that limits the minimum step size. The fixed-point integer and fractional bit widths can be analyzed from these results. This methodology is dependent on the stimulus quality and often gives too conservative fractional bit widths, but is in most cases the better starting point.

When the minimum and maximum absolute values are available, the required fixed-point integer and fractional word lengths can be calculated with nextpow2 function or with log2 function using Excel or a pocket calculator.

$$N_{intbits} = ceil(log_2(maxval)) + sign$$
$$N_{fracbits} = floor(log2(minval))$$

With this information, the fixed-point type definitions can be made in the type definitions file. The same type names must be used for both floating-point and fixed-point types and the type definitions must be separated with #ifdef macro to fixed-point and floating-point segments.

Depending on the data types used, the fixed-point syntax can be different. For the ac_fixed<> data type [6] the syntax is

ac_fixed<Totalbits, Integerbits, Signedness>

When the fixed-point type definitions are made, the C++ model can be switched to fixed-point mode, a new mex wrapper can be generated and compiled and the fixed-point C++ model can be simulated in MATLAB.

E. Exploring Different HW Architectures with HLS

Once the fixed-point C++ code is validated against the MATLAB reference model, it can be loaded into the HLS tool and synthesized with different constraints to explore the performance, power consumption and area for FPGA or ASIC technology targets. There is no need to touch the C++ code, unless some coding style issues prevent reaching the desired HW architecture. In such cases, the C++ code must be modified and validated again against the MATLAB model.

III. Design Example

A. Model Analysis

The example design, a DOA estimator, is a single file MATLAB model having no clearly defined DUT. The comments in the code segment in Fig. 3 indicate that the covariance matrix calculation could be the first DUT operation. Yet, matrix x is a 10x200 complex matrix, whereas the NN is only an 8x10 complex matrix. Furthermore, the for-loop consumes more than 80% of the simulation time, so limiting the DUT to the outer for-loop is a good compromise between data traffic and acceleration efficiency.

```
x=x+awgn(x,snr);%Insert Gaussian white noise
R=x*x'; %Data covarivance matrix

[N,V]=eig(R); %Find the eigenvalues and eigenvectors of R

NN=N(:,1:M-P); %Estimate noise subspace
theta=-90:0.5:90; %Peak search

for ii=1:length(theta)
    SS=zeros(1,length(M));
    for jj=0:M-1
        SS(1+jj)=exp(-j*2*jj*pi*d*sin(theta(ii)/180*pi)/lambda);
    end
    PP=SS*NN*NN'*SS';
    Pmusic(ii)=abs(1/ PP);
end
```

Fig. 3 DUT section of the MATLAB model

Closer analysis of the inner for-loop reveals that the SS vector values are only dependent on the loop counters and constants d, theta and lambda, thus, the SS values are constant. In HW they can be mapped to a ROM to make the HW simpler and smaller. The SS values can be collected in a matrix during the MATLAB simulation and stored into an ASCII file for the C++ conversion.

The remaining two functional code lines make up the accelerator core, which still has several operations. The first two multiplications are vector by matrix multiplication. The last multiplication is a dot product of two 10 element vectors. Opening the statement to three individual lines in Fig. 4 helps to understand the operations and to convert the statement to C++.

```
PP=SS*NN*NN'*SS';  % NN: 10Rx8C complex double
SSNN = SS*NN;  % 1Rx8C complex double
SSNN_NNtick = SSNN*NN';  % 1Rx10C complex double
mexPP = SSNN_NNtick * SS';  % complex scalar
```

Fig. 4 Opened matrix multiplication statement

The last remaining MATLAB code line can be rewritten in the form:

$$Pmusic(ii) = 1 / abs(PP);$$

This is a trivial operation that can be implemented with library elements, so it can be left out of the analysis at the moment.

Now it is time to extract the required data types. The max/ min value analysis of the variables is shown in Table I.

TABLE I
MAXIMUM AND MINIMUM VALUES OF MATLAB VARIABLES

Variable	Maximum value	Minimum value
NN	0.5854	0.0034330
SS	1.0000	0.0001196
SSNN	2.1474	n.a.
SSNN_NNtick	1.3352	n.a.
mexPP	9.9315	n.a.

Based on this analysis, only variables SS and SSNN_NNtick can be grouped to the same datatype. Because the difference to SSNN is one bit up and to NN one bit down, we group all four variables together. This enables better reuse of multipliers in low and medium concurrency implementations. Data types for output variable mexPP and the accumulator in the matrix multiplier must be defined separately. In addition to the base data types, the complex types must be defined as well. The resulting header file in Fig. 5 also has the fixed-point macro to switch between float and fixed later.

```
//#define FIXED_POINT
#ifndef FIXED_POINT // ac_ieee_float types work in Catapult 10.5
typedef ac_ieee_float64 data_out_t;
typedef ac_ieee_float64 mtx_in_t;
typedef ac_ieee_float64 mac_out_t;
#else
// ac_fixed typedefs here
#endif

// Complex types are primitive type independent
typedef ac_complex< mtx_in_t > mtx_in_cpl_t;
typedef ac_complex< mac_out_t > mac_cpl_t;
typedef ac_complex< data_out_t > data_out_cpl_t;
```

Fig. 5 Type definition file after initial type analysis

The last part of the analysis phase is to specify the required helper functions. From the MATLAB model in Fig. 4 we can extract the required matrix operations. The first multiplication is a vector by matrix. In the second multiplication, the matrix NN is transposed. The third multiplication is a dot product, where the right operand is transposed too. From this information we can define a class hierarchy needed to implement all required functions. Fig. 6 illustrates the class structure.

Complex multiplication for direct form and complex conjugate has a different operation between the products. Programming both variants into the multiplier function enables the matrix transpose to be done on-the-fly. The multiply-accumulate function should detect automatically if the data type is scalar or complex and select between the C++ multiply operation and complex multiplier function. Vector dot product and vector by matrix multiply functions use this class.

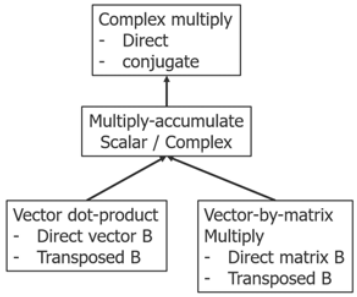

Fig. 6 Class hierarchy of complex matrix multiplier class

B. Model Conversion

After the analysis phase, we have a clear list of functions to be implemented and the data types needed in the system. The first step is to convert the matrix operations to C++.

The first class to implement is the complex multiply class. Principally, this could be done by using the C++ multiply operator, but the requirement of doing complex conjugation prevents us from using it. The implementation of the class is straightforward, defining an if-statement to switch between normal and conjugate B modes. A template to parameterize the data types and conjugate mode is needed to enable instantiation of the block with different data types and operating modes.

The multiply-accumulate class takes two vectors as input parameters and calculates their dot product. Principally it is just doing multiply-accumulate in the for-loop, but the multiply operation is different for scalar and complex variables.

```
template<typename Tin, typename Taccu, typename Tout, int VSize, bool conjugate>
class multiply_accumulate_class
{

private:
    complex_multiply_class<Tin, Taccu, Tout, conjugate > ComplexMult;

public:
    multiply_accumulate_class() { }

    Tout CCS_BLOCK (multiplyAccumulate)(Tin vec1[VSize],
                   Tin vec2[VSize]   )
    {
        Taccu accu;

        accu = 0.0;

        MAC:for (int idx=0; idx<VSize; ++idx)
        {
        // If data type is ac_complex, call optimized complex multiplier
        // Complex multiplier conjugates values, if transpose is selected
           if (is_ac_complex<Tin>::value) {
              accu += ComplexMult.Product(vec1[idx], vec2[idx]);
           }
           else {
              accu += vec1[idx] * vec2[idx];
           }
        }

        return accu;
    }
};  // multiply_accumulate_class
```

Fig. 7 Implementation of type agnostic multiply-accumulate class

Automatic complex type detection requires a specialized template that returns true if the template parameter is ac_complex. This is equivalent to the MATLAB iscomplex() function. Fig. 7 shows the complete implementation of the multiply-accumulate class. This level of configurability is not required in this design, but the same class library can be reused in other designs too.

Implementation of the dot product function is easy. It is just passing the template parameters to the instance of multiply-accumulate class and calling the function. This class could also be replaced by a direct call to multiply-accumulate.

Vector by matrix multiplication has more complexity. It has two for-loops that scan through the matrix and pass two vectors to the multiply-accumulate function. In case of on-the-fly transpose, this function switches the indices of the matrix. With this feature we can use one memory for the matrix and transpose it on-the-fly, when it is necessary

Now the background work is done and the top-level function can be implemented. Because we decided to leave the reciprocal operation out of the scope in this phase, only the matrix operations are implemented. The SS vector is mapped to an input parameter that is passed from MATLAB model to the function.

```
void CCS_BLOCK (musicTest) (Tin   NN[MTX_ROWS][MTX_COLS],
            Tin   SS[MTX_ROWS],
            Tout  &Out                    )

{

   Tin SSNN[MTX_COLS];
   Tin SSNN_NNT[MTX_ROWS];

   Tout PP;

   // Implement Matlab statement PP=SS*NN*NN'*SS';
   Mult_10Cx10R8C.CrossProduct(SS, NN, SSNN);
   Mult_8Cx10R8C_T.CrossProduct(SSNN, NN, SSNN_NNT);
   dotProd_10x10.DotProduct(SSNN_NNT, SS, PP);

   Out = PP;
}
```

Fig. 8 Top-level matrix test function implementation

Implementation of the matrix test function in Fig. 8 has only the three matrix multiply function calls. The data types are ac_ieee_float64 corresponding to the double type in MATLAB.

C. Validating C++ Functionality in MATLAB

The design can be loaded into the HLS tool for mex wrapper generation, if it is supported by the tool. Alternatively the mex wrapper can be created manually.

The generated mex function is instantiated into the MATLAB model. The modified testbench in Fig. 9 has a separate vector for the mexPP results that can be easily compared against the original results. Direct comparison inside the for-loop would also be possible, but the vector approach allows graphical analysis of the results later, when the model is quantized and the fixed-point effects are tested against the floating-point MATLAB reference model.

```
%    PP=SS*NN*NN'*SS';
    % Implement previous statement as 3 individual statements
    % to make it easier to map them to function calls
    SSNN = SS * NN;  % 1Rx8C complex double
    SSNN_NNtick = SSNN * NN';  % 1Rx10C complex double
    PP = SSNN_NNtick * SS';  % complex scalar

    mexPP = complex(1e-32,1e-32);
    % Call HLS C++ function
    mexPP = mfun_musicclasswrappermusicTest(NN,SS);

    PPvec(ii) = PP;
    mexPPvec(ii) = mexPP;

    Pmusic(ii) = abs(1/ PP);
    mexPmusic(ii) = abs(1/mexPP);
end
```

Fig. 9 MATLAB testbench with mex function

MATLAB simulation with the floating-point mex function should generate identical results. The logarithmic scale in Fig. 10 does not show any difference between the signals.

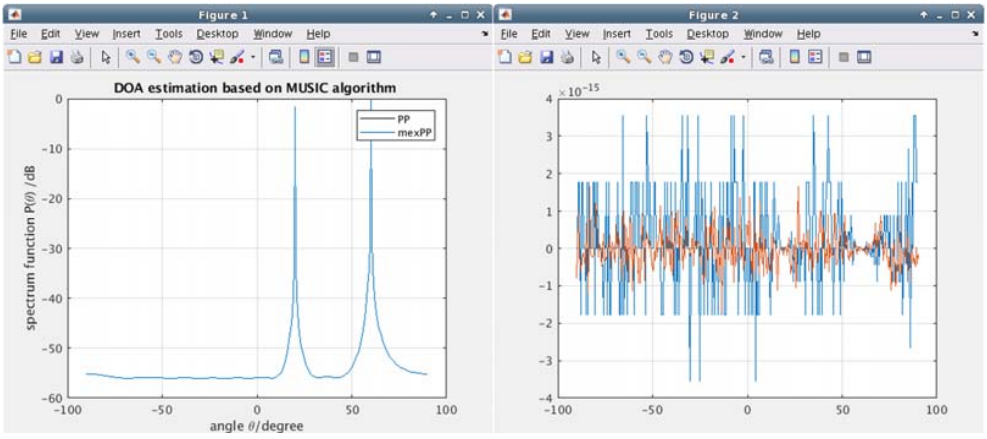

Fig. 10 Comparison of results MATLAB vs. floating-point C++

The difference view in Fig. 10 shows deviation in the level of 10-15, which is normal for floating-point operations in this scale. The mantissa of the IEEE double type is only 52 bits and doing multiply and add operations in a different order can cause this difference.

D. Quantizing and Fine Tuning the HLS C++ model

Once the floating-point behavior is correct, the design can

be converted to fixed-point. In this particular case it is useful to quantize the matrix operations first. Division and square root operations might cause unpredictable effects, so it is better to analyze the safe operating conditions for the matrix arithmetic separately.

Using the maximum and minimum values in Table I we can calculate the fixed-point parameters for the three data types. Table II shows the integer and fraction bit requirements for these values.

TABLE II
FIXED-POINT WORD LENGTHS FOR INTERNAL VARIABLES

Variable	Maximum	Int bits	Minimum	Frac bits
NN	0.5854	0 + sign	0.0034330	9
SS	1.0000	1 + sign	0.0001196	14
SSNN	2.1474	2 + sign	n.a.	
SSNN_NNtick	1.3352	1 + sign	n.a.	
mexPP	9.9315	4 + sign	n.a.	

SSNN requires 3 integer bits. Because we have only one test dataset available, at least one bit of additional headroom should be given. We have no simulation values for the accumulator available, so the data type must be analyzed statically. Both inputs of the multiplier have 4 integer bits, so with 8 integer bits overflow is excluded. Multiply-accumulate (MAC) functions have 8 or 10 MAC operations, which need a fully parallel implementation of an adder tree with 4 stages. Each stage increases the number of integer bits by one. Because the multiplication needs with the given maximum values only 5 integer bits, 8 integer bits should be a safe value for the accumulator too. Data output needs 5 integer bits, but we start with 8 bits.

Because we use only one input data type, the number of fractional bits must be calculated for the smallest minimum value of all variables. SS requires 14 fractional bits. This is the number to be used for all variables.

The fixed-point type definitions are

ac_fixed<18,4,true> for the input type
ac_fixed<22,8,true> for accumulator and output

When these type definitions are added into the header file and the mode is switched to fixed-point by defining the FIXED_POINT macro, a new mex file must be generated or the hand coded mex wrapper must be modified to do the right type conversions and a new mex object must be compiled.

Now we can change the fixed-point word lengths to test how the design works with different fractional lengths. Because the ac_fixed type has the number of integer bits explicitly defined, changing the total word length automatically changes the number of fractional bits.

By simulating the design with different word lengths and storing the data into the MATLAB workspace, the influence of the word length can be easily visualized.

In Fig. 11 the undermost curve is the floating-point MATLAB result. Other curves represent different fixed-point word lengths in the order of legend. The knee point of the word length is 20 bits. With a higher number of bits the results remain about the same, but below the 20 bits the quality of the results decreases rapidly. The estimated 18 bits is still acceptable, but not functionally optimal. A 20 bit input word length would be optimal for the matrix multiply part of the design, but this is not the whole truth. The reciprocal of the magnitude is still missing.

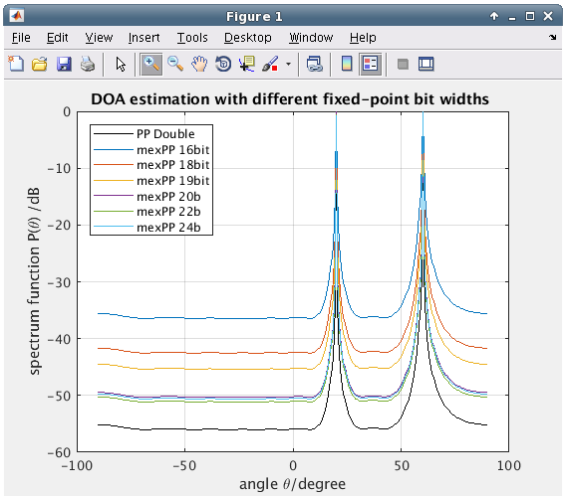

Fig. 11 Fixed-point word length influence to matrix multiply results

The final implementation of the outer for-loop of the MATLAB model requires the const array for the SS values and a for-loop with 361 iterations. The SS vector must be copied from the const array to a local variable that can be passed as a parameter to two matrix multiply functions.

The abs function caused a surprise. The function call was different for the ac_ieee_float64 and ac_fixed data types. It was replaced by separated square and square root operations to enable data type switching without conditional statements in the functional C++ code. The synthesis results are the same for both implementations, so the problem is only cosmetic. The final top-level implementation is shown in Fig. 12.

```
SCANLOOP: for (int ii=0; ii<NUM_ITER; ii++)
{
    SSCOPY: for (int j=0; j<MTX_ROWS; j++) {
       SS[j] = SSMatrix_361x10[ii][j];
    }

    // Implement Matlab statement PP=SS*NN*NN'*SS';
    Mult_10Cx10R8C.CrossProduct(SS, NN, SSNN);
    Mult_8Cx10R8C_T.CrossProduct(SSNN, NN, SSNN_NNT);
    dotProd_10x10.DotProduct(SSNN_NNT, SS, PP);

    // Implement Matlab statement Pmusic(ii)=1/abs(PP);
    PPabs = (PP.real() * PP.real()) + (PP.imag() * PP.imag());
    ac_math::ac_sqrt(PPabs, sqrtOut);
    divOut = ac_math::ac_reciprocal_pwl<Tdiv>(sqrtOut);

    OutVec.data[ii] = divOut;
}  // End of SCANLOOP
```

Fig. 12 Top-level C++ code

The fixed-point analysis for the reciprocal statement was made using a static analysis method.

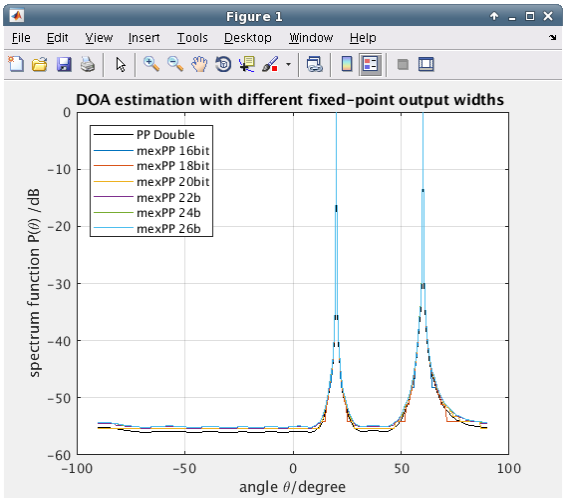

Fig. 13 Reciprocal output with different matrix multiply bit widths

The simulation results in Fig. 13 show that the matrix multiplication word length has almost no influence on the results, as long as it is at least 16 bits. The division output data type determines the height of the peak. With all matrix multiply word lengths up to 24 bits the notch value is zero, which causes a divide by zero condition in the reciprocal function. This is handled inside the function that returns the maximum possible value of the output data type. With 15 integer bits the peak value is closest to the floating-point value and works for the other values too.

E. Exploring Different HW Architectures with HLS

When the fixed-point behavior fulfills the requirements, different HW implementations can be explored with HLS. The synthesis tool is driven by constraints that control loop transformations, memory architecture and pipelining. The same C++ code can result in HW implementations from minimum resource, long latency to fully parallel, minimum latency architectures on different target technologies.

If the desired architecture cannot be reached, there might be coding style issues that prevent synthesis from accessing some resources in parallel. C-code problems can also be analyzed with the HLS design checker tool or with conventional SW analysis tools like lint or Valgrind. Sometimes the design is simply too large to fit into the target FPGA device or too complex to reach the latency requirements. In such cases the algorithm must be revised.

The example design was synthesized with Catapult HLS using Nangate 45nm CMOS target technology with 3 different implementations.

Solution	Latency...	Latency...	Throug...	Throug...	Total Ar...	Slack
solution.v1 (new)						
music_class_wrapper.v1 (extract)	136075	1360750.00	136099	1360990.00	42297.51	1.95
music_class_wrapper.v2 (extract)	38262	382620.00	38268	382680.00	185662.70	0.22
music_class_wrapper.v3 (extract)	**4331**	**43310.00**	**4334**	**43340.00**	**891552.80**	**0.00**

Fig. 14 HLS Simulation results of the example design

The results in the Fig. 14 show the latency range from 136075 down to 4331 clock cycles. Because the main for-loop runs 361 iterations, the latency of one iteration varies between 377 and 12 clock cycles. Each iteration does 170 complex multiplications. The minimum resource implementation uses 4 multipliers and the minimum latency version uses 390.

An interesting case is the second implementation with 38262 clock cycles latency (102 cycles/iteration). It uses 39 multipliers and fits into a smaller FPGA or FPGA SoC. The acceleration factor is still reasonable compared to a SW implementation on an ARM core.

IV. Conclusions

A seamless, high abstraction level design flow from abstract floating-point MATLAB model to RTL can be implemented efficiently by using HLS. Automated RTL code generation and verification removes the need for maintaining the RTL model, which raises the overall abstraction level of the design process.

The model functionality is converted from MATLAB to C++ using floating-point data types to enable designers to focus on the functionality without fixed-point effects. Easy switching between floating-point and fixed-point makes validation of functional changes easier.

The automated validation link between MATLAB and C++ and HLV enables continuous regression testing between the two models and early verification of the design at the C++ level.

The proposed methodology has been developed together with several design teams and tested by using multiple designs. The whole process can be mastered by one engineer, who knows both MATLAB and HW design. An ideal design team has one MATLAB specialist, one RTL designer with MATLAB knowledge and one verification engineer.

Each design is different and requires a workflow that fits a particular project. This design methodology is a framework that can be applied to different needs of the projects.

The example design was converted from the original MATLAB model to synthesizable C++ code in one day including the implementation of the matrix multiply class.

V. Future Work

Even though the presented methodology is proven to work, there are still many details that need to be improved. Easy switching between the floating-point and fixed-point modes requires identical mathematical functions for both data types. In addition to that, a comprehensive library of open source HLS C++ classes for vector and matrix operations is needed to speed up the conversion process.

References

[1] H. Tang, "*Examensarbete : DOA estimation based on MUSIC algorithm*", Linnéuniversitätet Kalmar Växjö, 2014

[2] M.Baleani, F. Gennari, Y. Jiang, Y. Patel, R. K. Brayton, A. Sangiovanni-Vincentelli, "*HW/SW Partitioning and Code Generation of Embedded Control Applications on a Reconfigurable Architecture Platform*", CODES+ISSS '04, 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, 2004.

[3] J. Noguera, R.M. Badia, "*HW/SW Codesign Techniques for Dynamically Reconfigurable Architectures*", IEEE Transactions on Very Large Scale

Integration (VLSI) Systems, vol. 10, no. 4, August 2002.
[4] H. Youness, A. Hussein, A. Mahfoz, "A new hardware/software partitioning technique", Tenth International Conference on Computer Engineering & Systems (ICCES), Dec. 2015
[5] S. Banerjee, N. Dutt, "*Efficient Search Space Exploration for HW-SW Partitioning*", CODES+ISSS '04, 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, 2004.
[6] Mentor Graphics, "*Algorithmic C (AC) Datatypes*", Reference manual, 2020, https://github.com/hlslibs/ac_types/blob/master/pdfdocs/ac_datatypes_ref.pdf
[7] Open source HLS libraries, https://hlslibs.org/
[8] The MathWorks Inc., "*C++ MEX Applications*", Matlab reference manual, https://www.mathworks.com/help/matlab/cpp-mex-file-applications.html
[9] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde and I. Bolsens, "*A Methodology and Design Environment for DSP ASIC Fixed Point Refinement*", DATE 1999 Conference
[10] D. Menard, R. Rocher, O. Sentieys, "*Analytical Fixed-Point Accuracy Evaluation in Linear Time-Invariant Systems*", IEEE Transactions on Circuits and Systems, vol. 55, no. 10, November 2008