Accelerating Fixed-Point Simulations Using Width Reconfigurable Hardware Architectures

Keyvan Shahin
Chair of Computer Engineering
Brandenburg University of Technology
Cottbus, Germany
keyvan.shahin@b-tu.de

Michael Huebner
Chair of Computer Engineering
Brandenburg University of Technology
Cottbus, Germany
michael.huebner@b-tu.de

***Abstract*— Digital Signal Processing (DSP) systems can be described using either fixed-point or floating-point for their numeric representations. Since the general computer CPU uses floating-point representation, software-based simulation tools which are used for modeling and simulation of the arithmetic operations in DSP systems, use floating-point representation as well. However, synthesizing customized hardware for fixed-point arithmetic operations for FPGAs or ASICs is more efficient compared to their floating-point counterparts. This necessitates a step to convert the representation of a floating-point simulated algorithm on MATLAB for example, to a fixed-point representation which is more suitable for hardware implementation. While former approaches for this conversion step have always been software-based, like on MATLAB itself, here a new approach has been introduced to show the possibility of accelerating it by using hardware width re-configurable designs.**

***Index Terms*—Fixed-point Simulation, DSP Systems, FPGA Acceleration, Reconfigurable Hardware Design**

I. Introduction

The procedure for implementing a DSP functionality on hardware consists of description of the functionality, designing the functionality in an environment with simulation capabilities, conversion of the numeric representations to a format which is suitable for hardware implementation, description of the functionality using a hardware-aware language and using appropriate tools for compiling and synthesizing the design based on the target platform [1]. Fig. 1 demonstrates this general flow. In general, development of a DSP algorithm starts with its functional description. For example how a novel forward error correction (FEC) code, like LDPC encodes the data at the transmitter side, and how it decodes the coded data to reproduce the original data at the receiver side, can be described mathematically.

Based on this description the designer can develop the algorithm and write a software code for example in MATLAB to test and simulate the design. For the mentioned FEC code example, this will be the generation of different data packages, encoding them, adding appropriate noise values to the coded data to simulate the effect of channel, decoding the noisy data and comparing the final results with the original data in order to measure the effectiveness of the FEC code.

Most DSP algorithms use floating-point arithmetic because of its development simplicity and its good numerical properties [2]. However, for numerous embedded systems, fixed-point arithmetic is preferred because of its benefits in terms of power consumption, area and architecture latency. Thus, a conversion from floating-point to fixed-point is required before the hardware implementation of the algorithm [1]. When the algorithm needs to be implemented on hardware platforms like FPGAs, SoCs, ASICs and etc. the numeric representation of all the variables should be converted from floating- to fixed-point. Floating-point and fixed-point representations are thoroughly explained on Section II.

After the integer and floating part widths for the variables in the code are figured out in the last step, the development of the hardware description of the algorithm can be done using these widths with the appropriate description language based on the target platform. Then the code can be compiled or synthesized and the outcome can be loaded onto the platform for the final result.

In this work, our focus is mainly on the floating-point to fixed-point conversion part of this flow. For a developer who is using MATLAB for developing and simulating an algorithm, there are certain toolboxes available for generating the fixed-point version of the code and running the simulation to measure the effect of reducing the bit-widths of each variable on the accuracy of the overall output of the algorithm. For a complicated project with a lot of different inputs, intermediate variables, and complex arithmetic functions, running the fixed-point simulation in cases, can take several weeks to complete. In many cases the variety of possibilities for inputs and different widths for the variables are so vast that the designers are forced to settle for a sub-optimal solution of the floating- to fixed-point conversion solution. Many DSP systems require an implementation of nonlinear datapaths on ASICs and FPGAs and by going for a sub-optimal and maybe constant width over the whole datapath, loss of efficiency in terms of area, power consumption and speed can be resulted. Especially, when going for mass production, the exhaustive search for the optimal widths for each intermediate variable becomes more crucial due to reduced cost of silicon area, higher speed of the end result, and the reduced power consumption. By using the approach which will be introduced in this paper, this exhaustive simulation will be accelerated using width reconfigurable hardware implementation of the DSP algorithms on an FPGA

platform.

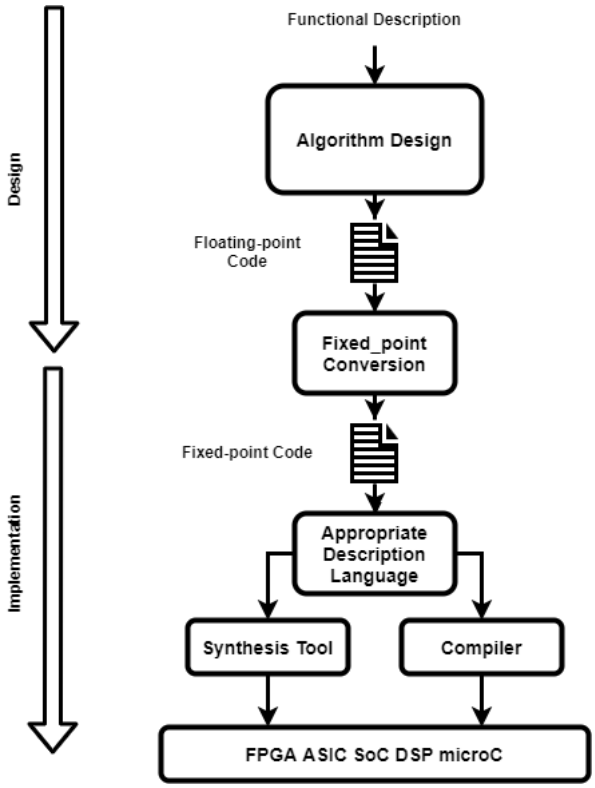


Fig. (1) Digital Signal Processing Hardware Development Flow

This article is a milestone for an approach to accelerate the fixed-point simulation of a general DSP algorithm, using an FPGA-based width reconfigurable hardware implementation of the algorithm. The rest of this paper is organized as followed. Section II describes the floating- and fixed-point numeric representations and their characteristics, and a common approach for converting a normal floating-point code in MATLAB to a fixed-point code. Section III describes our approach to accelerate the fixed-point simulation on hardware, the execution flow of the simulation on hardware, and the width reconfigurable hardware sub-modules designed to meet this end, and Section IV is a conclusion of the contributions of this work and describing the way forward to reach the goals of this project.

II. Floating-Point to Fixed-Point Conversion

The floating-point to fixed-point conversion process is an optimization problem [3] to find the data word-lengths. Searching for a solution to this problem, is a trade-off between the cost (area, power and speed) and the application quality degradation caused by the limited bit-widths of the fixed-point data types. In [4], [5] it is shown that the conversion process can take up to 30% of the total development time which motivated this work, in accelerating this process by using width-reconfigurable hardware implementation of the floating-point to fixed-point conversion. Thus this acceleration reduces the development time and decreases the time-to-market of the final product [1].

The accuracy which is observed at the outputs of a DSP system, depends on the bitwidths used to represent each intermediate variable in the system. In a general system, the accuracy is more sensitive to some variables than to others, hence the most efficient hardware implementation of an algorithm is one in which different bitwidths are used for representing different intermediate variables. There are mainly two class of approaches for finding the optimal bitwidths of the intermediate variables, namely analytical and bit-true simulation.

The analytical approach attempts to model quantization error statistically and expanded to specific linear time invariant (LTI) systems such as digital filters, FFT, etc. in these approaches many conditions were assumed such as data signals and error signals are not correlated and error sources are white and uniformly distributed [6]. Because of these assumptions, it has been demonstrated that bit-width optimization, even for LTI systems is NP-hard. Even though the analytical techniques often are faster than simulation-based approaches, they are less general since they are only applicable to LTI systems.

The bit-true simulation method has been extensively used recently since they can handle non-LTI systems as well as LTI systems. This method requires large simulation time, which has been a bottleneck of this approach. Our goal in this work is to accelerate this method using hardware and removing this bottleneck. In this simulation, the integer bitwidths and fractional bitwidths are optimized in two separate phases. In the first phase, all the possible combinations of the inputs are fed to the system, and the maximum and minimum values for each intermediate variable will be observed. Knowing these values, the required number of bits that are needed to represent the integer part of each variable in order to avoid overflow will be calculated. In the second phase, the simulation runs by observing the effect of changing the number of bits representing the fractional part of each intermediate variable and how it effects the output accuracy. Since this method was slow, for bigger designs it is usual to decrease the numbers of simulations by sacrificing the optimality of the answer [6]. This way, it is possible to find a set of sub-optimal fractional widths for the intermediate variables without testing for all the possible solutions.

In [7] a process has been introduced to methodically, to convert floating-point MATLAB code to fixed-point. Our work is a hardware-based conversion, based on a simplified version of this method, hence this method will be briefly explained here. In this method, the conversion is divided into two main steps, the integer width calculation and the fractional width calculation. The first step tries to find the ranges of the intermediate variables and the second step finds the smallest number of bits which is required for the fractional parts of each variable while keeping the resulting error within an acceptable range.

For the dynamic range calculation, the simulation runs all the inputs on the system and observes the MIN and MAX

values for each of the variables. These MIN and MAX values show the dynamic range for each variable based on all the inputs that are being tested. Based on these, the number of bits that are required for the integer part of each variable will be known. after this point, the integer widths of the fixed-point MATLAB code will be set to the calculated values for the rest of the simulation.

Some definitions which are needed to go through with this method are as follows. The resulting error of fix pointing the code will be

$$e = outdata_{float} - outdata_{fixed} \tag{1}$$

The Error Metric (EM) is defined as

$$EM\% = norm(e)/norm(outdata_{float}) * 100 \tag{2}$$

The fixed-point character of each variable is shown by the quantizer numbers that are set for it in MATLAB code and hence

q_i is the i_{th} quantizer (for the i_{th} variable)

q_i is shown as [wordlength fractionallength] or [IW_i+FW_i, FW_i]

In which IW_i and FW_i are the integer and fractional widths of the i_{th} variable.

The precision optimization part of the fixed-point conversion in this work is divided into coarse optimization and fine optimization. In the coarse optimization, we search for a sub-optimal solution and a width for all the variables which is adequate for reaching the desired accuracy. After that we use fine optimization to find more optimal solution for each individual variable. Assuming that the largest acceptable error is shown by EM_{max}, the coarse optimization has four steps:

- we choose three mark points, L, H, and M. L is set to the lowest precision to start with for example 0 bits. H for highest number for example 32 bits and M would be set as the average of H and L.
- EM is calculated for the M width.
- If $EM_M < EM_{max}$, it means the average number of bits or case M, is enough and we can search in the interval L, M now, thus we replace M by H, otherwise M is not enough and we search in the interval M, H by replacing M by L.
- We repeat steps 2 and 3 until M is equal to either L or H. We call EM_M at this point EM_{coarse}.

As was mentioned, the coarse optimization finds a sub-optimal precision which is equal for all the variables and it will be the starting point for the fine optimization to find individual fractional widths for each variable. Starting from coarse optimization helps by providing a good starting point for the fine optimization. Without coarse optimization the fine optimization would take a long time to reach the results. The fine optimization is done in these steps:

- Start by setting $EM_{fine} = EM_{coarse}$
- Define and calculate EM_i as $EM(FW_1, FW_2, ..., FW_i + 1, ..., FW_n)$, n being the total number of variables. This is the error metric, if the i_{th} variable had 1 more fractional bit.
- Define $DEM(i) = EM_{fine} - EM_i$. This array shows the effect of adding 1 fractional bit to the i_{th} variable on the error metric compared to the current error metric. As was discussed the precision of the final output has different sensitivity to the bitwidths of different variables. When we calculate $DEM(i)$ for different variables from 1 to n, the larger $DEM(i)$ shows that the output precision is more sensitive to the addition of 1 fractional bit to the i_{th} variable.
- In the array we choose the smallest value for example $DEM(j)$ with the condition that FW_j is not 0 (if a variable's fractional width is already 0 there is no room for reducing it). This j_{th} index corresponds to the variable for which its width reduction has the least impact on the overall precision of system output.
- Define and calculate EM^j as $EM(FW_1, FW_2, ..., FW_j - 1, ..., FW_n)$. There will be two possibilities here. If $EM^j \leq EM_{max}$ it means reducing the j_{th} variable's fractional width by 1 bit does not nullify the maximum desired error threshold and this reduction can be enforced. The widths will be updated to $(FW_1, FW_2, ..., FW_j - 1, ..., FW_n)$ and the new EM_{fine} will be set to EM^j. We go back and do the last 3 steps again.

After these procedures are over, a set of widths for integer and fractional parts of the of the variables are ready. Even though this method saves a lot of time by avoiding an exhaustive search into all the possible widths, the software based true bit simulation still consumes a long time for complex DSP systems. In the next section we will explain our approach to mimic this procedure on hardware to accelerate the fixed-point conversion speed.

III. Hardware-based conversion

As it was described, when a DSP algorithm is to be implemented on hardware, we ought to find the minimum integer and fractional width for each intermediate variable and still maintain a desirable accuracy for the final output, so that the resulting hardware implementation is as optimized as possible, consuming least amount of resources and running at the highest speed. Even thought it is possible to find these intermediate variable widths using the software-based fixed-point simulation described in Section II, in practice, it can be a very time consuming task to run all the possible inputs and observe the output validity and accuracy for each set of widths for intermediate variables. Going back to the LDPC codes for an example, if one wants to have an LDPC code implementation, after the design phase is over in MATLAB for example, the widths of the intermediate variables should be figured out. The LDPC codes themselves are not the focus of this article but they are interesting as an example here since their bit error rate can reach very small numbers; hence in order to test their error correction capabilities, a huge number of random input samples need to be generated and run through

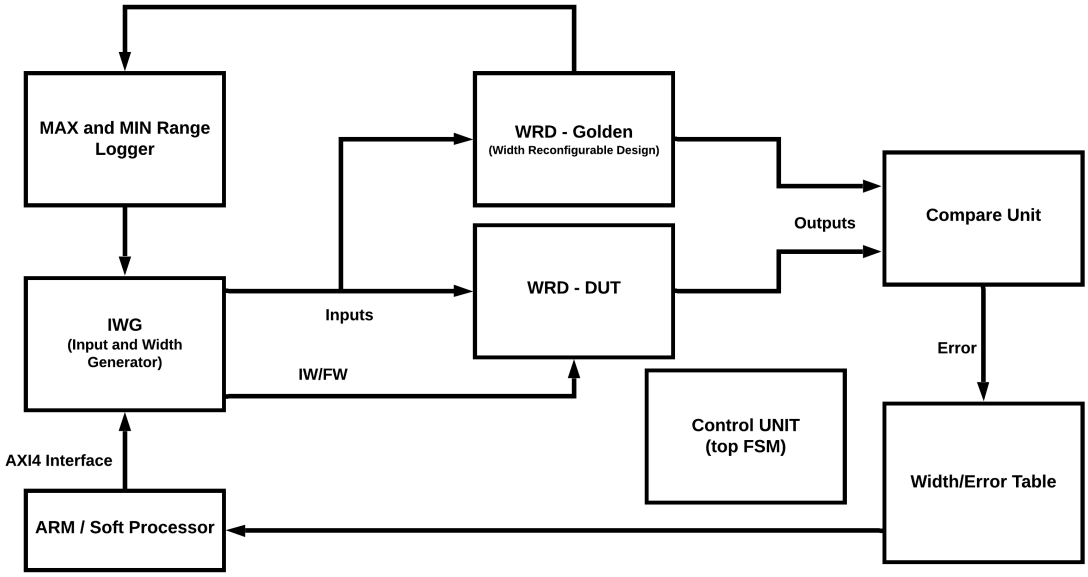


Fig. (2) Hardware-based Fixed-point Simulation Architecture

the algorithm for each set of variable widths. This fixed-point simulation is still possible in MATLAB, but it takes a very long time to execute since there are many variables present in this algorithm and the width for each of them needs to be tuned individually in order to reach the optimal implementation on hardware.

A. Architecture and Flow

Fig. 2 demonstrates the architecture used in our approach that will be implemented on an FPGA platform to do the fixed-point simulation of a custom DSP algorithm. This architecture uses two instances of the implemented design. one of these instances has constant maximum width values and is serving as the accurate instance with golden output and the other instance's variables widths can be changed at run time. we call these instances Golden and Design Under Test (DUT). The Input and Width Generation module (IWG) has FSM loops to set the widths for DUT and for each set of widths it feeds all the inputs to both instances so that their outputs can be compared with each other. This comparison is done by the Comparison Unit and the results of this comparison is written to a table with the corresponding widths sets and sent to the ARM core on the Zynq FPGA for the user information and further analysis in the future.The Min and Max Range Logger keeps the maximum and minimum values of all the variables in the first phase for finding out the dynamic range of each of them. The Control Unit is the FSM responsible for the flow of the simulation. It starts the simulation, controls when the IWG needs to generate the next set of widths or the next set of inputs, when the outputs of WRD instances are valid for error comparison and the flow of the simulation phases to find the variable integer widths or fractional widths.

The simulation starts by the code on the ARM processor. It sets some parameters for the simulation and commands the start. After the start command is sent, the Control Unit takes control. The steps are very much similar to the software-based fixed-point simulation flow which was discussed in Section II. The simulation starts by finding out the dynamic range for the variables. In this step, the WRD-DUT is turned off and the IWG module scans through all the input values and sends them to the WRD-Golden which has the maximum widths for the integer and fractional parts. A logger keeps the maximum and minimum of every variable during this input scan. At the end of scanning the inputs, the maximum and minimum values for all the variables are sent to the IWG. Based on that, the IWG will know what is the required integer bit width for each variable.

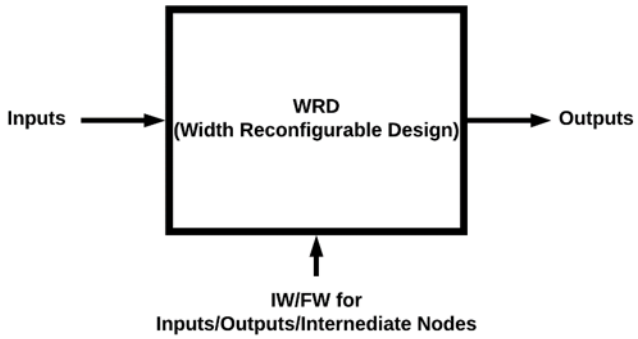


Fig. (3) Width Reconfigurable Design

In the second phase of the simulation, the WRD-DUT will be turned on and the IWG will put the integer widths for WRD-DUT as constants that were calculated in the first phase

of the simulation. Then it uses the same algorithm that was described in Section II for software-based simulation, to set the fractional bits widths and scans through the inputs at for each set of widths. When the whole input span is fed to the WRDs for each set of widths, the error between WRD-Golden output and WRD-DUT can be calculated. The way to calculate the error depends on what the tested functionality is. This must be taken into account that this article is a description of an approach for hardware-based fixed-point simulation rather than a fixed architecture for every possible DSP algorithm. For example when the goal is to implement something which has some memory elements by which the output depends on both the current input and the past inputs, like the example of an FIR filter, the error or output difference between WRD-DUT and WRD-Golden needs to be calculated after the whole span of inputs are fed to the WRDs and the array of outputs are calculated for the whole span, but when the output is only dependant on the current input, like the example where we want to implement a memory-less trigonometric function like a sinus function, each individual output between the two WRD modules can be compared with each other for every single input independent of the inputs before and after it. We will go deeper into the WRD architecture to describe our approach in implementing a hardware implementation of a custom mathematical functionality, with the ability to change the bit-widths of its intermediate variables at run-time.

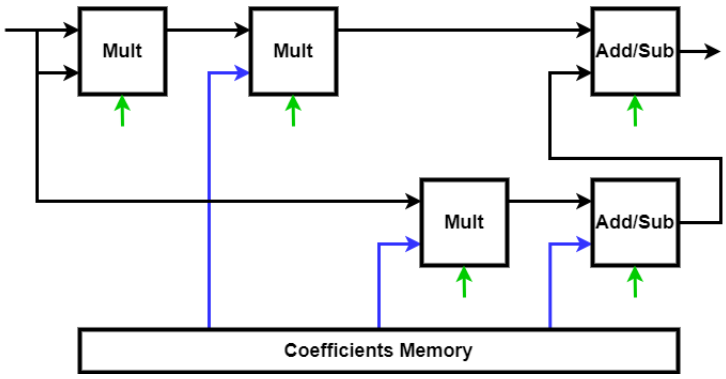
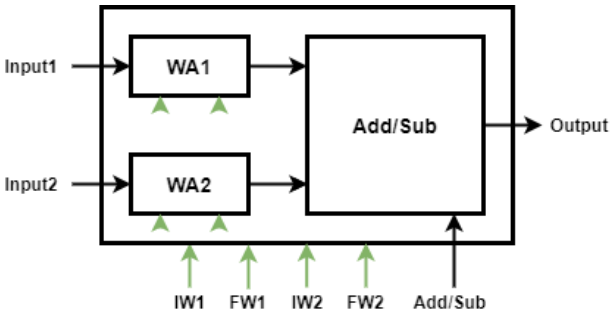


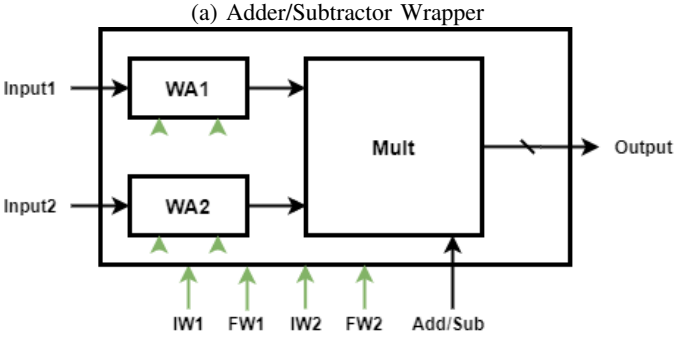
Fig. (4) Polynomial Approximation Example

B. Width Reconfigurable Design

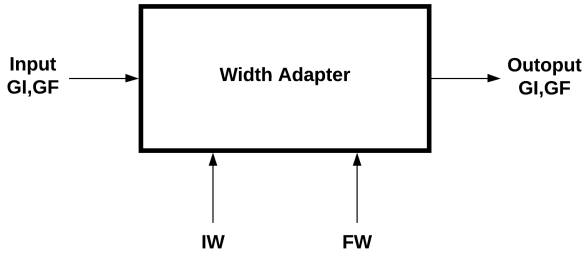
This subsection describes the approach that was taken to implement a mathematical functionality, with the ability to change the bit-widths of the intermediate variables after the implementation of the logic on the FPGA and during the run-time. This enables the designed hardware function to receive the widths as inputs, so that the IWG can run the fixed-point simulation on the design by changing these bit-widths.(Fig. 3) For each design, we define a maximum for the number of bits that we desire to provide for the integer and fractional parts of each variable. We call these numbers Global Integer width (GI) and Global Fractional width (GF). These numbers are the constants for all the variable widths in the WRD-Golden instant and limit the span for the bit-widths of the WRD-DUT instant variables during the fixed-point simulation. In contrast, we refer to the reconfigurable changing widths for each variable as Integer Width (IW) and Fractional Width (FW) of that variable. So the IW and FW for each variable can change from 0 to GI and GF.



(a) Adder/Subtractor Wrapper



(b) Multiplier Wrapper



(c) Width Adaptor

Fig. (5) Building Blocks

Considering the fact that implementing any custom design on hardware with the ability of bit-width re-configurability using a general method is impossible, we took the approach to use the piece-wise polynomial approximation representation of the functions. Of course the functions that are inherently implemented using only adders and multiplication do not need to be converted to be represented by piece-wise polynomial approximation. In this way we break the function into basic adder/subtractors and multipliers which makes implementing a bitwidth reconfigurable design possible. When a function is broken down to only adder/subtractors and multipliers, what is needed to be designed is width reconfigurable addition and multiplication modules. Consider the example where we have a function which we want to represent as a simple second

```
62    Gen1: for sCount in 0 to cIWGlobal+cFWGlobal-1 generate
63         O (sCount) <= I (sCount) when ((sCount<OutIW+cFWGlobal) and (sCount>cFWGlobal-OutFW-1))  else
64                       I(OutIW+cFWGlobal-1) when sCount >= OutIW+cFWGlobal  else
65                       '0';
66     end generate Gen1;
```

Fig. (6) Width Adaptor VHDL Code

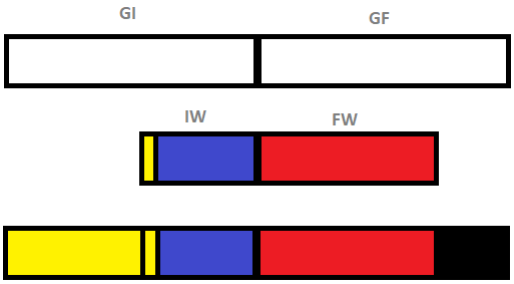


Fig. (7) Width Manipulation Inside The Width Adaptor

degree polynomial. From the implementation point of view, we will have a look up table to hold the coefficients, three multipliers, and two adder/subtractors (Fig. 4).

We notice that even when the goal is to find individual widths for each module, they have to be connected together to form the functionality. In order to deal with this situation, we used a wrapper for each of the multiplier and adder/subtractor modules. The wrapper itself has inputs and outputs with integer and fractional widths set to GI and GF. This ensures that the modules can be connected to each other easily and without the necessity of using additional logic between them. Inside the wrapper, we need to implement a multiplier or an adder/subtractor which can be re-configured at run-time to operate with different input bit-widths. In our approach, we used a combination of a constant maximum width (GI+GF) adder/subtractor or multiplier, and something that we call the Width Adaptor (WA) to transform each input, as if it had the desired (IW and FW) bit-widths.

Fig. 5a shows the way the Adder/Subtractor Wrapper combines a normal Add/Sub module with GI,GF widths for inputs and two instances of Width Adaptor to re-configure the effective widths of the inputs to be processed by the normal Add/Sub module. The same architecture is seen on Fig. 5b for the Multiplier Wrapper. The normal Add/Sub and Multiplier modules work with the constant width inputs (GI, GF) and as can be seen in Fig. 5c the Width Adaptor itself does have the input and output, both with GI, GF widths. What actually happens inside the WA is that it makes the output from the input in a way as if it has less bits representing its integer and fractional parts. Even though the output physical width is still GI and GF for its integer and fractional parts, the contents of the output are manipulated as it will be described shortly, so that it has the effective widths of IW and FW.

The way that the WA manipulates the input, so that it has the effective widths of IW and FW is by using sign extension and changing the least significant bits to zero. As it is depicted in Fig. 5c, the WA has the input and output with the physical widths of GI, GF. The effective width of the input is dictated by the preceding blocks and what they have done to this data, however the WA manipulates the data presented by the input and uses IW and FW to trim it into output. How this trimming is done is depicted in Fig. 7. Since we are using the 2's complement representation, we keep the IW-1 bits from the integer part of the input and will extend the bit representing the sign to fill the rest of the whole GI width. For the fractional part, the bits that are less significant than the target FW bits, should be deleted, and we do that by replacing them with 0s as if we do not have any information about those bits. The VHDL code for this process is shown in Fig. 6. Line 63 assigns the part that is similar in the input and output, line 64 is responsible for the sign extension, while line 65 puts the bits that are less significant than the desired FW width to zero.

Since all the physical widths of intermediate modules' inputs and outputs in this design are based on the global widths (GI, GF), which are set before the synthesis and implementation of the hardware on the FPGA, connecting the modules together is made possible, while by introducing the Width adaptor and the concept of effective bit-widths, the ability for the design to be width reconfigurable is added which makes this architecture ready for execution of the hardware-based fixed-point simulations.

IV. Conclusion and Future Work

In this article we had an overview of the flow in digital signal processing hardware design and the fixed-point simulation step to find the suitable bit-widths of the intermediate variables in the algorithm which is required when migrating from software code to hardware implementation. Considering that previous approaches used software-based fixed-point simulation and the long run-times of that approach, the main contribution of this work can be summarized as the introduction of a hardware-based approach, to use a width reconfigurable hardware architecture for accelerating the fixed-point simulation.

While this article describes the flow and the architecture of the modules in this approach, right now we are working to implement various examples of mathematical and DSP algorithms and use this approach to do the fixed-point simulation on them. Using these examples, we will try to write scripts for automatically generating the human readable VHDL codes for generating the whole architecture for as many algorithms as possible and come up with numerical results to show the execution time efficiency of this approach in coparison with software-based approaches.

REFERENCES

[1] Nehmeh, Riham. (2017). Quality Evaluation in Fixed-point Systems with Selective Simulation.

[2] Nehmeh, Riham and Banciu, Andrei and Michel, Thierry and Rocher, Romuald. (2014). A Fast Method for Overflow Effect Analysis in Fixed-point Systems. Conference on Design and Architectures for Signal and Image Processing, DASIP. 2015. 10.1109/DASIP.2014.7115608.

[3] Shi, Changchun and Brodersen, Robert. (2003). An automated floating-point to fixed-point conversion methodology. 2. II - 529. 10.1109/ICASSP.2003.1202420.

[4] Ren, S.-J and An, J.-P and Bu, X.-Y and Xu, Z. and Zhang, X.. (2016). Design of MB-OFDM UWB system frequency synchronization. 36. 1283-1288. 10.15918/j.tbit1001-0645.2016.12.014.

[5] T.Hill. AccelDSP Synthesis Tool Floating-Point to Fixed-Point Conversion of MATLAB Algorithms Targeting FPGAs. White papers, Xilinx, april 2006.

[6] Roy, Sanghamitra and Banerjee, Prith. (2005). An algorithm for trading off quantization error with hardware resources for MATLAB-based FPGA design. Computers, IEEE Transactions on. 54. 886- 896. 10.1109/TC.2005.106.

[7] Roy, Sanghamitra and Banerjee, Prithviraj. (2004). An algorithm for converting floating-point computations to fixed-point in MATLAB based FPGA design. 484-487. 10.1109/DAC.2004.240395.