University of Tennessee, Knoxville

TRACE: Tennessee Research and Creative Exchange

Masters Theses

Graduate School

5-2007

Optimization of Digital Filter Design Using Hardware Accelerated Simulation

Getao Liang
University of Tennessee - Knoxville

Follow this and additional works at: https://trace.tennessee.edu/utk_gradthes

Part of the Computer Engineering Commons

Recommended Citation

Liang, Getao, "Optimization of Digital Filter Design Using Hardware Accelerated Simulation. " Master's Thesis, University of Tennessee, 2007.
https://trace.tennessee.edu/utk_gradthes/301

This Thesis is brought to you for free and open access by the Graduate School at TRACE: Tennessee Research and Creative Exchange. It has been accepted for inclusion in Masters Theses by an authorized administrator of TRACE: Tennessee Research and Creative Exchange. For more information, please contact trace@utk.edu.

To the Graduate Council:

I am submitting herewith a thesis written by Getao Liang entitled "Optimization of Digital Filter Design Using Hardware Accelerated Simulation." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Donald W. Bouldin, Major Professor

We have read this thesis and recommend its acceptance:

Hamar Elhanany, Gregory D. Peterson, Mark A. Buckner

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Getao Liang entitled "Optimization of Digital Filter Design Using Hardware Accelerated Simulation." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Donald W. Bouldin

Major Professor

We have read this thesis
and recommend its acceptance:

Hamar Elhanany

Gregory D. Peterson

Mark A. Buckner

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the
Graduate School

(Original signatures are on file with official student records.)

Optimization of Digital Filter Design Using Hardware Accelerated Simulation

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Getao Liang

May 2007

Acknowledgments

I would love to express my thanks to every one who has helped me obtain my Master of Science degree in Computer Engineering.

First, I would like to thank my major advisor, Dr. Donald W. Bouldin, for his inspirational teaching, encouraging guidance and warmhearted support. Second, I would like to thank Dr. Mark Buckner for introducing me the great concept of DE Optimization with hardware accelerated simulation. Third, I would like to thank Dr. Itamar Elhanany for his instructive tutoring in the class of packet switching. And I also want to thank Dr. Gregory D. Peterson for his great lecture on hardware verification two years ago.

Special thanks to my friends and coworkers, who has been around me when I was striving on this work. Without anyone of them, this work can not be done. Especially, I want to thank Harriette Spiegel for her patience and enthusiasm in reviewing my work.

Heartfelt thanks to my family and my beloved.

Abstract

The goal to this research was to develop a scheme to optimize a digital filter design using an optimization engine and hardware-accelerated simulation using a Field Programmable Gate Array (FPGA). A parameterizable generic digital filter, which was fully implemented on a prototyping board with a Xilinx Virtex-II Pro xc2vp30-7-ff896 FPGA, was developed using Xilinx System Generator for DSP. The optimization engine, which actually is a random candidate generator that will eventually be replaced by a differential evolution engine, was implemented using MATLAB along with a candidate evaluator and other supporting programs. Automatic hardware co-simulations of 100 candidate filters were performed successfully to demonstrate that this approach is feasible, reliable and efficient for complex systems.

Table of Contents

List of Figures

I. Introduction

1. Overview

Nowadays, digital systems are becoming faster, more powerful, more complicated and more capable, so designing circuits for these systems is becoming more and more difficult. Firstly, increasing system complexity and functionality means more goals and constraints have to be considered at the same time. These goals sometimes conflict with each other. Take system complexity and power consumption for example. As circuit complexity increases the ability to minimize area and reduce power becomes increasingly difficult [1]. Secondly, due to the limitation of design tools, solutions are only optimized for certain objectives out of all the objectives that the designer wants to achieve. In another words, the tools only provide locally optimized solutions instead of globally optimized ones. Thirdly, complex systems always have more design parameters which are tightly interacting with each other, making it even harder for designers to ignore the negative effects on others when changing one or some of the parameters.

Though electronic design automation (EDA) tools are becoming more and more popular in the hardware industry and have benefited from the availability of high performance computing, there are some innate weaknesses that deteriorate the performance when simulating a hardware design with a software simulator. First of all, the software simulation cannot reflect the actual transistor behavior which occurs in the hardware. Secondly, low-level simulation using software executes very slowly, which is totally unacceptable to most digital system designers. That is because shortening the time-to-market is one of the most efficient ways to produce more profit.

2. Goals and Contributions

The major goal for this work was to establish an efficient framework for the optimization of digital filter design using an optimization engine and hardware-accelerated simulation using a Field Programmable Gate Array (FPGA). This framework can be divided into two parts: the hardware and the software.

For the hardware, a parameterizable generic digital infinite impulse response (IIR) filter was developed using Xilinx System Generator (Sysgen). The Sysgen model for this generic filter was designed to be reconfigurable, which means that the filter can be reconfigured anytime by data that are input to this model. The filter design was verified by comparing the results produced from this model to the theoretical results. After it was validated to be fully functioning, the filter design was then implemented in a Xilinx Virtex-II Pro FPGA on the prototyping board.

In the software domain, all programs were developed using MATLAB. Mainly three major functions were created, including the random candidate generator that will eventually be replaced by a real differential evolution optimization engine when it is ready, the filter evaluator and a program that coordinates the data exchanging between the software and the hardware.

As soon as the hardware and the software were verified to be working appropriately, hardware co-simulation was performed.

II. Background

In this chapter, background information about different simulation approaches, digital filters and the algorithm of Differential Evolution will be covered before detailed discussion of the design.

1. Simulation Approaches for Hardware Design

Software simulation is widely used by designers when they are developing hardware systems. Software simulation refers to using a high performance computer to simulate the activities which occur inside of a hardware design. Software simulation is very helpful for verification and debug purposes. The most widely used tool for software simulation of FPGA designs is ModelSim by Mentor Graphics, which provides a comprehensive simulation and debug environment. ModelSim offers simulation support for multiple languages, including VHDL, Verilog and SystemC. Besides behavior level simulation, there are simulations on other levels. For example, transaction level simulation utilizes abstracting communications to minimize the number of simulation events. By doing so, simulation time can be reduced significantly, which makes it possible for simulation of a complex system on the system level.

Hardware co-simulation has been used primarily for the verification of hardware/software systems, such as an embedded system [2]. Co-simulation can be also defined as manipulation of simulated hardware with software. With hardware co-simulation, the verification can benefit from both high-speed executions by the hardware part and the flexibility of computation from the software part. Thus, hardware co-simulation is highly recommended nowadays for the simulations of complicated systems or embedded systems.

2. Digital Filters

Based on the existence of feedback loop, digital filters can be divided into two categories: convolution filters and recursive filters. The following sections will discuss them in detail.

a. Convolution Filters - FIR

Convolution filters can be also called finite impulse response (FIR) filters because their response to an impulse signal ultimately becomes zero. Figure 2.1 illustrates the structure of a 2nd order FIR filter. From the diagram, it is evident that the structure of an FIR is very straightforward so that it is really easy to be implemented in both software and hardware. Since there is no feedback loop for an FIR filter, direct realization of it requires many resources. But at the same time, due to the lack of internal feedback, a FIR filter is inherently stable. In the frequency domain, the phase response for a FIR filter is a linear function of the frequency. Thus, no phase distortion exists.

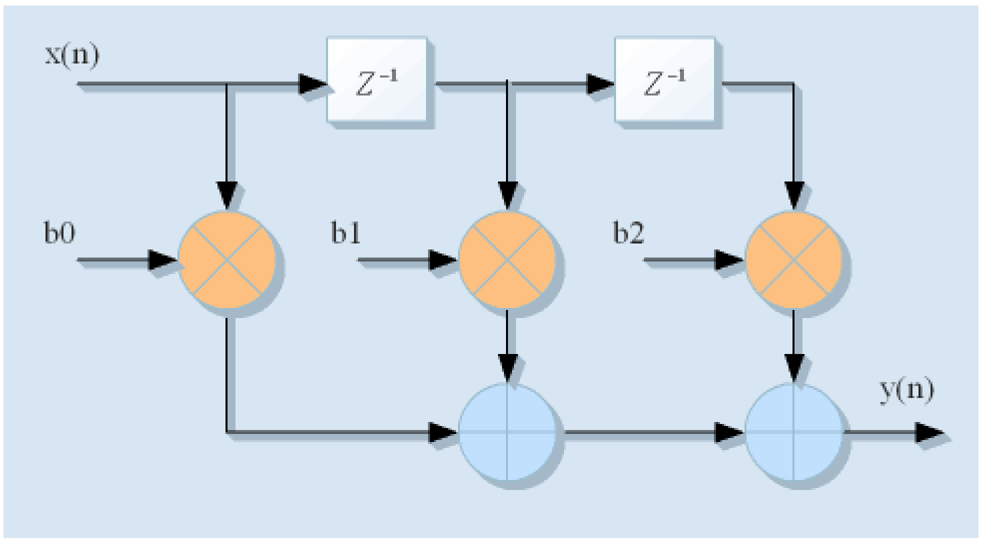


Figure 2.1 – Structure of a 2nd Order FIR Filter

There are two forms of representation that can be used to describe a specified FIR filter. One is called a difference equation, which defines how the input signal is related to the output signal. From the equation, the output of current time interval is the sum of the products of current input and some delayed inputs.

$$y(n) = b_0 \cdot x(n) + b_1 \cdot x(n-1) + ... + b_{N-1} \cdot x(n-N+1)$$

And the other is called a transfer function, which is used to describe the filter in Z domain.

$$H(z) = b_0 \cdot z^0 + b_1 \cdot z^{-1} + ... + b_{N-1} \cdot z^{1-N} = \sum b_n \cdot z^{-n}$$

b. Recursive Filters - IIR

Recursive filters can be also called infinite impulse response (IIR) filters. Figure 2.2 shows the structure of an IIR filter in Direct Form I. Compared to the FIR filters, IIR filters are more complicated concerning the structure since IIR filters require internal feedback and use one or more output signals as inputs. Due to the nature of recursion, the output will never become zero and that is why it is called infinite. Since the structure is more complex than that of a FIR filter, it will be harder for a designer to design and implement one. Unlike FIR, IIR has a non-linear phase response, which means there will be phase distortion. Because of the feedback loop, the response becomes unstable. But with careful design, it can be designed to be stable. Rounding errors are not compounded by summed iterations. IIR filters are much closer to the analog models and better for non-standard filter realization. A high-order FIR filter can be represented by a low-order IIR filter.

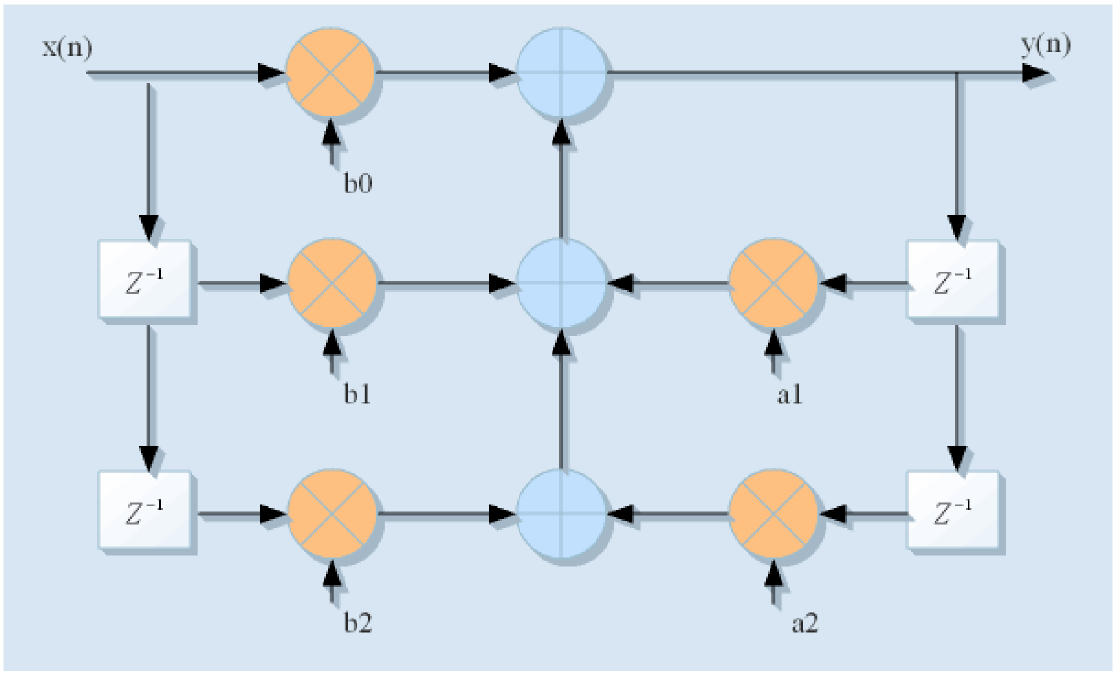


Figure 2.2 – Structure of a Second order IIR Filter in Direct Form I

Just like the FIR filter, there are two forms of representation that can be used to describe an IIR filter. One is called a difference equation, from where we can see that some of the previous outputs are used to calculate the current output.

$$y(n) = b_0 \cdot x(n) + b_1 \cdot x(n-1) + \ldots + b_M \cdot x(n-M)$$
$$-a_1 \cdot y(n-1) - a_2 \cdot y(n-2) \ldots - a_N \cdot y(n-N)$$

And the other is called a transfer function, which is used to describe the filter in Z domain.

$$H(z) = \frac{b_0 \cdot z^0 + b_1 \cdot z^{-1} + \ldots + b_M \cdot z^{-M}}{1 + a_1 \cdot z^{-1} + \ldots + a_N \cdot z^{-N}} = \frac{\sum b_m \cdot z^{-m}}{1 + \sum a_n \cdot z^{-n}}$$

Actually, the structure of an IIR can be used to describe a FIR filter. When all the coefficients to multiply with outputs are set to be zero, then this is a FIR filter. This is obvious when we look at the picture. If all the coefficients are set to be zero, then only the left part remains and it is the FIR filter.

c. General Filter Structure

A filter of any order can be described as a cascade of biquad filters. This equation shows there are two structures that can be used to construct a high-order filter. The left part of this equation is called direct form and the right part is called SOS form, which is short for second-order section.

$$H(z) = \frac{b_0 \cdot z^0 + b_1 \cdot z^{-1} + \ldots + b_n \cdot z^{-n}}{1 + a_1 \cdot z^{-1} + \ldots + a_n \cdot z^{-n}} = g \cdot \prod_{K=1}^{K} \frac{b_0 + b_{k,1} \cdot z^{-1} + b_{k,2} \cdot z^{-2}}{1 + a_{k,1} \cdot z^{-1} + a_{k,2} \cdot z^{-2}}$$

Compared to the direct form, SOS form has a lot of advantages. First of all, one simple design of a second-order filter can be used to handle all kinds of filters. That will be great for the optimization of hardware implementations. Because only one instantiation of a second-order filter with some support circuits for loop control are required to implement a filter design of any order. More importantly, filters in this structure are less sensitive to quantization and overflow. However, there is a not-so-important disadvantage. In order to implement a specified order, using the SOS form requires more coefficients than using the other one.

d. Target Filter

The target filter used in this work was a graphics codec for sample rate reduction. This filter is promulgated by Radio Communication Sector of International Telecommunication Union (ITU-R) in 1985. Figure 2.3 depicts the tolerance schemes for magnitude and group delay for the target filter.

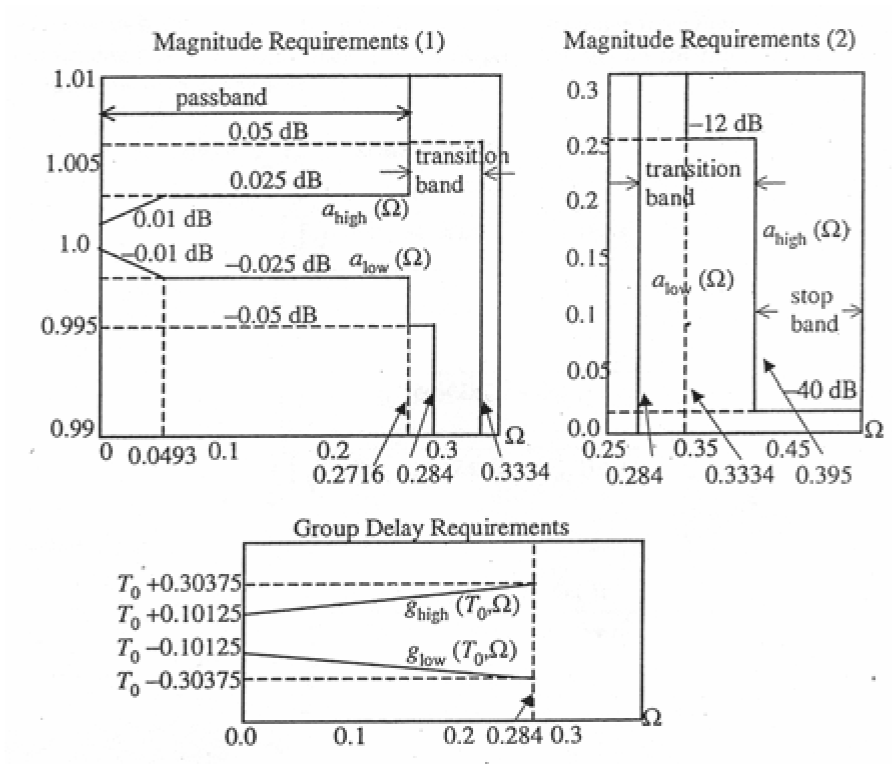


Figure 2.3 - Tolerance Schemes for Magnitude and Group Delay [9]

3. Differential Evolution

Differential Evolution is a very simple population-based, direct-search evolution algorithm for global optimization. Basic steps involved in this algorithm can be described as below, which is also illustrated in Figure 2.4.

1) Treat parameters as a vector of n elements;
2) Randomly construct initial vector population;
3) Use vector differences for perturbing the vector population;
4) Compare the results of another random vector and the relocated one, and keep the better one;
5) Repeat and make all the vectors converge to a global optimized position.

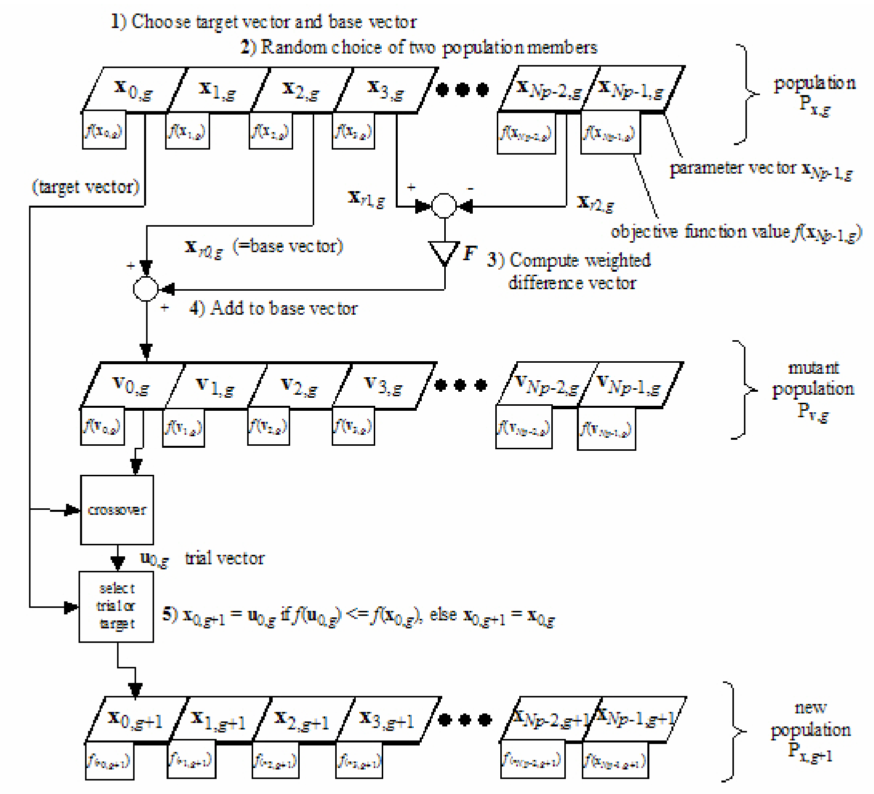

1) Choose target vector and base vector
2) Random choice of two population members
population $P_{x,g}$
parameter vector $x_{Np-1,g}$
objective function value $f(x_{Np-1,g})$
(target vector)
$x_{r1,g}$
$x_{r2,g}$
$x_{r0,g}$ (=base vector)
F
3) Compute weighted difference vector
4) Add to base vector
mutant population $P_{v,g}$
crossover
$u_{0,g}$ trial vector
select trial or target
5) $x_{0,g+1} = u_{0,g}$ if $f(u_{0,g}) <= f(x_{0,g})$, else $x_{0,g+1} = x_{0,g}$
new population $P_{x,g+1}$

Figure 2.4 – Illustration of Differential Evolution Algorithm [10]

III. System Overview

1. System Requirements Analysis

As discussed in previous chapters, the goal for this project was to decrease the simulation time for a large-scale design of a digital system. Furthermore, it was also important to examine the actual performance of the design when it was implemented as working hardware. Thus in order to achieve both objectives, the whole system design required consideration of two domains: software and hardware. Figure 3.1 illustrates the main system blocks, as well as their respective responsibilities. Obviously, the functions of the software modules are to pass test candidates to and collect feedback from the device-under-test (DUT), while the hardware takes the responsibility of performing the filter functions described in the design specifications.

For the hardware part, the major task was to implement a specified digital circuit design, which is controlled by the parameters fed from the software program. The target digital design, used in this project to evaluate the performance of the accelerating scheme, is basically a digital filter. Thus, the hardware design should be

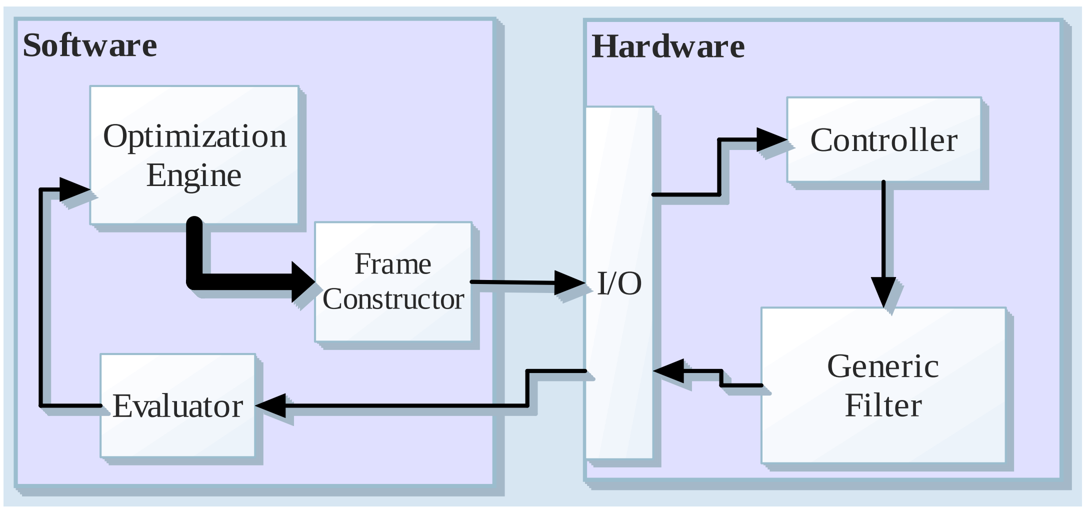


Figure 3.1 – System Block Diagram

a parameterized generic filter, and can be configured as any IIR, FIR or IIR+FIR mixed filter. For the purpose of giving more flexibility to the optimization engine, the filter design should accept various candidates of different numbers of taps, as well as bit-width setting of these input data. Also, the design should allow for acceptance of new filter candidates at any moment. To test the filter's responses, only one single set of data should be used as the test string for all the candidates, such that the outputs of all filters under the test are analogous and comparable. While the set of test data is the same in any case of simulation, any single test number should reveal independence or randomness to others of the test sequence. That is because the arbitrary property of test vectors is crucial to the comprehensive evaluation of a digital filter. Otherwise, the results gained from those data might not be universal to all possible cases. Of course, there should be an I/O interface for the hardware to communicate with the software program.

Compared to the requirements of the hardware, those of the software design are much more straightforward. There are two essential functionalities that the programs should handle. One is an optimization engine to achieve the optimized design of the filter by iterations of generating, perturbing and re-calculating operations. The selection processes of better candidates, completed within the optimization engine, are mainly based on the results from the performance evaluator, the other key component. The job of the evaluator is to calculate the frequency response and group delay from the simulation outputs, compare the results to the target specifications, and finally to generate a cost value for each candidate, which indicates the quality of the design. In order to feed coefficients into the hardware implementation, a communication protocol was made and a program to construct the input sequences was necessary.

2. System Level Design

According to the analysis of requirements, both the hardware and software portions can be divided into several subsystems, each of which is accountable for performing certain parts of the tasks. The partition of the system into modules can help to simplify the design process, as well as the debugging. For the purpose of communication between the software program and the FPGA, there must be data exchange existing between them during system operations. Before considering detail designs of each domain, it is necessary to make up a protocol that describes what and how the exchange works.

a. HW/SW Interface

Concerning the hardware part, the data swap means transferring in and from the FPGA. Two kinds of signals needed to be exchanged: coefficients and filter responses. The former is what the hardware needed in order to be configured it as the designated candidate, and the latter is the result after the operation of the filter. Because the output timing is not predictable due to unforeseen input signals, an additional control signal was required to facilitate informing the software with the status of the FPGA operations.

So there are totally three I/O ports needed for the FPGA to cooperate successfully with the software program. Figure 3.2 illustrates the types and directions of these signals within the system. Since all of the calculations of the filter coefficients are performed using floating-point, the data must be converted to fixed-point before they can be recognized by the digital design. After numerous experiments, it was discovered that, for better accuracy, it is best to convert the floating-point coefficients to signed 24-bit fixed-point with 20 bits for the fraction. Since the width of signal is also one of the aspects that this project should optimize, a wider dynamic range of data width should be implemented. Considering both FPGA utilization and performance, the signal types were set to be fix_32_29 for coefficients and fix_34_29 for data output. The extra two bits for *Data_OUT* were for the possible carryout bits added after arithmetic operations within the filter. Obviously, a Boolean type of signal is enough for the control signal, which is used to indicate the completion of the simulation.

In order to configure a filter, three parameters are needed including gain, number of SOS sections, and coefficients in SOS form. Concerning the arbitrariness of the data of signal *Coef_IN* in both incoming time and data value and their exclusive properties, there must be some strategy to indicate the start of a new set of candidate coefficients. Figure 3.3 shows the structure of the framed coefficients of one filter candidate. The first element, *start-flag*, indicates the start of a set of input sequences, which can be used to reset and initialize the system, while the UID (unique identifier) is a timestamp for the particular candidate. Since the UID is a unique value assigned to each candidate when created, it helps the filter controller to determine whether or not to refresh the memories with the incoming coefficients. The other components are the parameters to describe each individual candidate, and will be discussed more in the following sections.

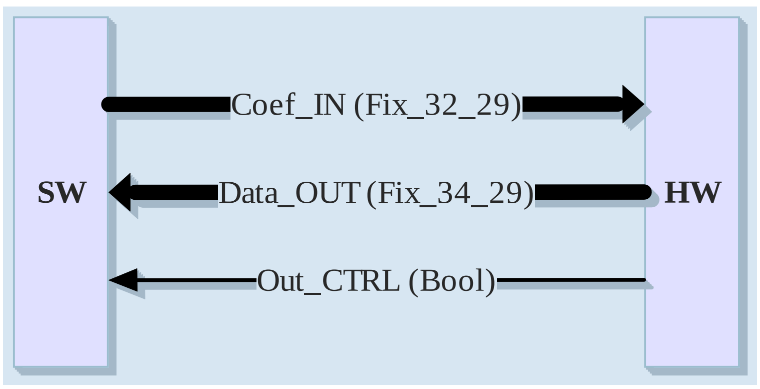


Figure 3.2 – Signal Types and Directions

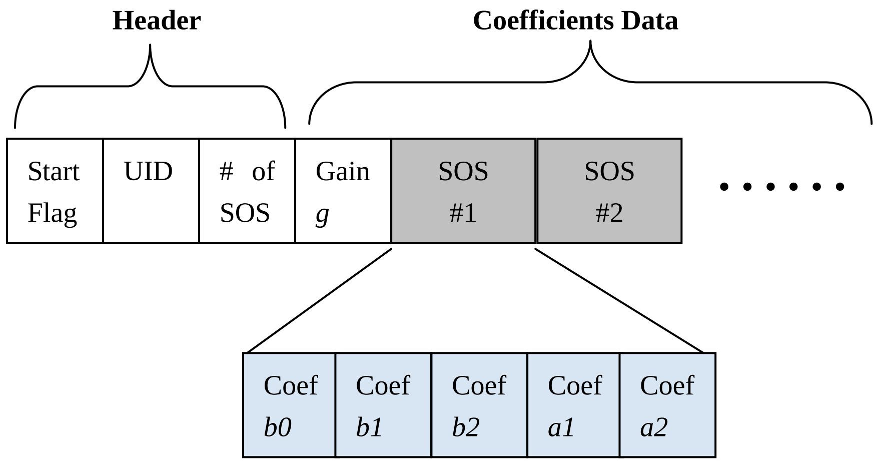


Figure 3.3 – Structure of Signal *Coef_IN*

b. Hardware Sub-system Design

As mentioned in Chapter II, any digital filter design can be presented as a cascade of second-order sections (SOS) of IIR. So as to implement a generic digital filter, which can perform as any filter with maximum order of 20, a cascade of 10 biquad filters was required. There are two ways to design such a structure in hardware. One is to instantiate 10 biquad modules and then connects the output port of one to the input of another. This scheme is so straightforward that very little consideration has to be made for the peripherals circuits for control and support purpose. However, the cost for this simplicity is huge FPGA utilization, which means a waste of FPGA space and power consumption. So a better plan was to use sophisticated control circuits to manipulate a loop of 10 biquad operations by using only one single instantiation. It causes more delay, but costs 1/10 the space of the first scheme.

To give a better visual demonstration of this scheme, a flow chart of the hardware design is given as Figure 3.4 which details the process of reusing one instantiation of a biquad filter to form a cascaded filter. From the chart, the hardware sub-system can be roughly divided in to six modules, which are listed below. Figure 3.5 illustrates the relationship and data flow between these modules.

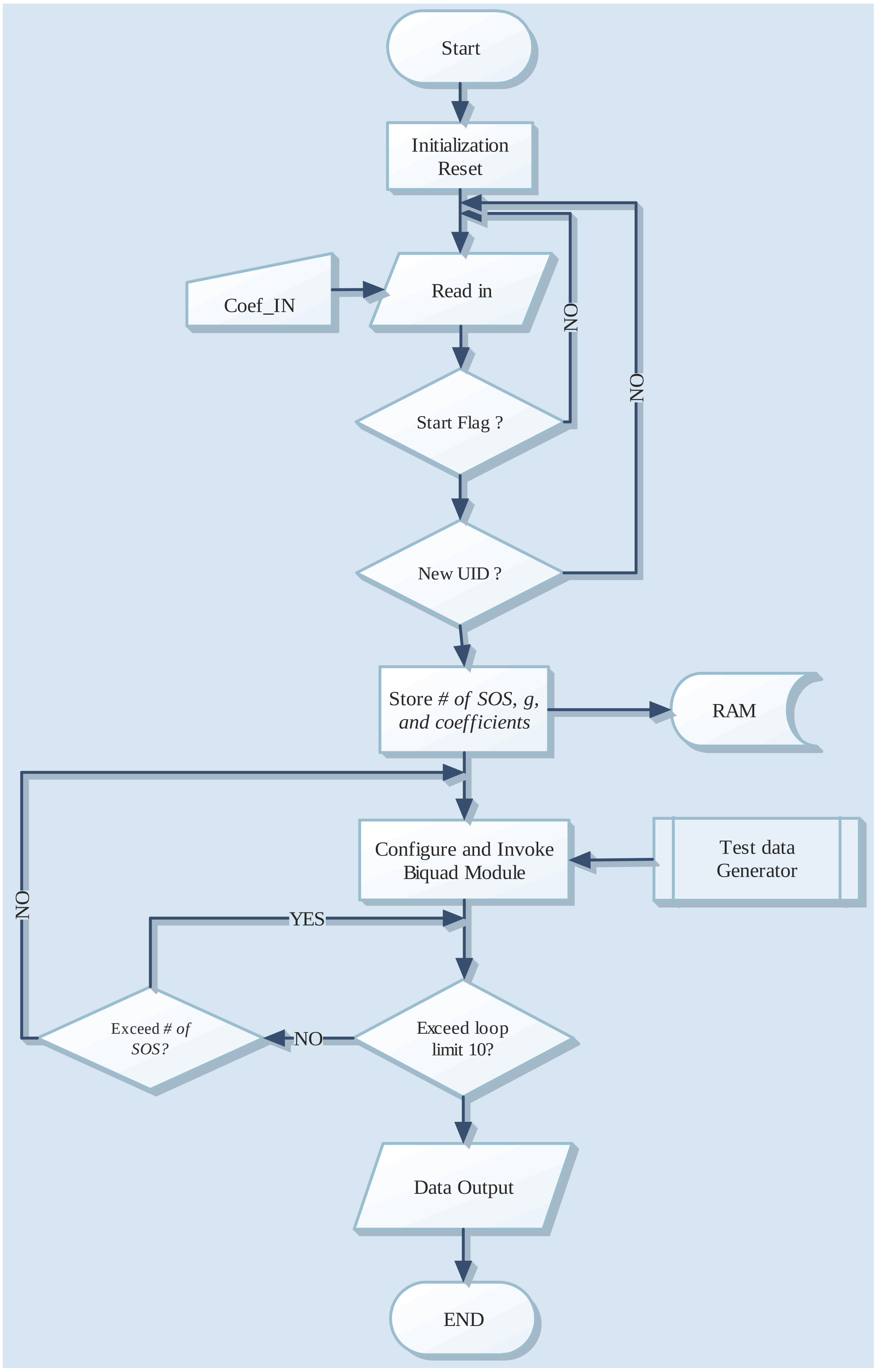


Figure 3.4 – Flow Chart of HW Design

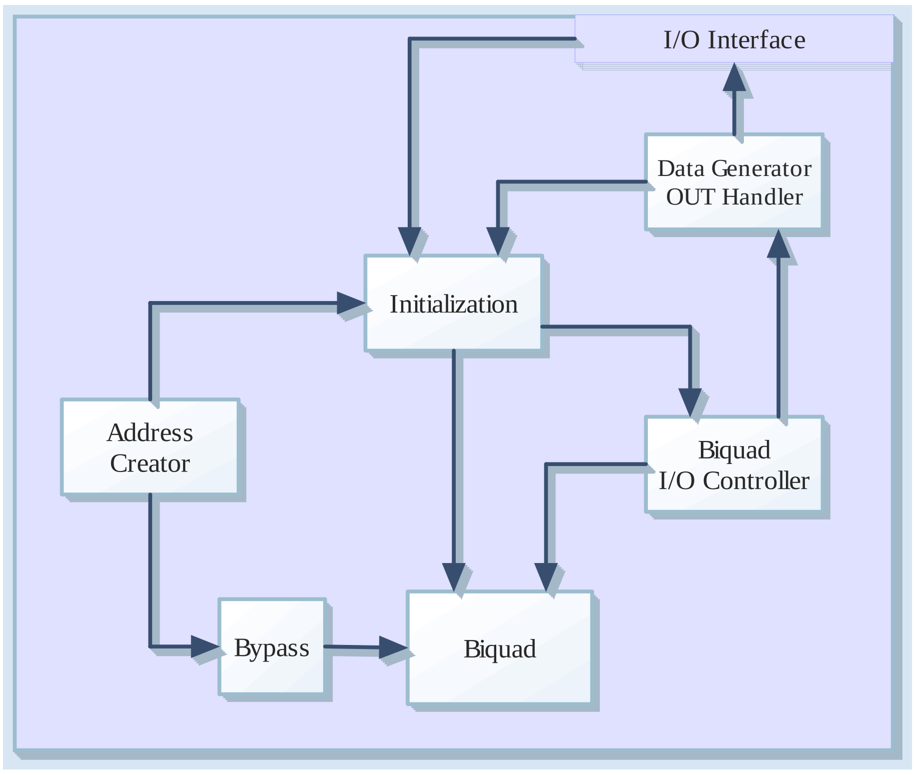


Figure 3.5 – Data Flow of HW Modules

- Initialization
- Data Generator and Output Handler
- Address Creator
- Biquad I/O controller
- Bypass
- Biquad

As indicated by the name, the *Initialization* module is in charge of resetting the system into its original status and preparing it for the following operations. Besides, it also has two important responsibilities associated with input signals. One is to examine the incoming data, recognize the data head, and generate the necessary control signals which are used to enable, reset or trigger other circuit components. The other task is to preload coefficient data into memory once only for each new candidate. Thus the module consumes less power consumption but provides more driving ability for the downstream circuits.

Data Generator Module is responsible for generating a pseudo-random vector sequence fed into the filter as the signal to be filtered. This module also functions as an output cache and generates an enable signal that handles the coordination of data exchanging with software. *Bypass Module* actually is a group of circuits used to discard input signals into the *Biquad* when the coefficients for that SOS section are undefined. The direct result of this module is saving unwanted power consumption by reducing as much electronic switching within the *Biquad* as possible.

The presence of numerous calculations, states transitions, and memory accesses within a generic filter requires the system to have an effective control scheme which handles all the complex exchanging and transiting procedures. *Module Address Creator* is one of the important parts of this control scheme. Its duty is to arrange and generate different address signals used for corresponding purposes, such as maintaining correct status of all the memory components for each individual round of the *Biquad*. In this project, the only the instantiation of the *Biquad* is shared by all the SOS sections so different input sources or output destinations might be used for different sections. Another controlling module, *Biquad I/O Controller*, is utilized to handle this situation. Instructed by the address signals, it selects the correct input signal to the *Biquad* and routes its output to the exact destination. The key module of this system is the *Biquad*. The only objective of this module is to implement a second-order IIR filter that can be configured by parameters input from the outside.

c. Software Sub-system Design

Figure 3.6 depicts the flow chart of the software design. From the chart, two main loops are compulsory in order to find the most optimized filter candidate. Within the loops, apparently there major operations are performed repeatedly. According to this attribute, the software system is partitioned into four parts, one main program and three functions, which are listed as

- Main Function
- Optimization Engine
- Candidates Framer
- Result Evaluator

The main function takes care of most of the setting, control and communication tasks shown on the flow chart. First of all, it has to set the constants and initialize parameters. Then, *Main Function* starts two loops and calls corresponding functions at appropriate moments. In order to perform co-simulation with the hardware, *Main Function* is also responsible for invoking the hardware model and controlling the simulation.

The function *Optimization Engine* generates an initial population of filter candidates the first time it is called. When called again, it performs a re-calculation and provides a population for the next generation based on evaluation of the candidates' performance. When a set of coefficients of a candidate is ready, the function *Framer* is called to generate a data stream that complies with the protocol discussed in the last section, by adding a header in front of the coefficients and inserting some zeros for synchronization purposes. When the simulation of one candidate is finished, function *Evaluator* analyzes the outcome from the filter. By comparing the frequency response and group delay of filter candidates and the ITU-CCIR standards, a corresponding score is given to each candidate to indicate the degree of its match with the target filter.

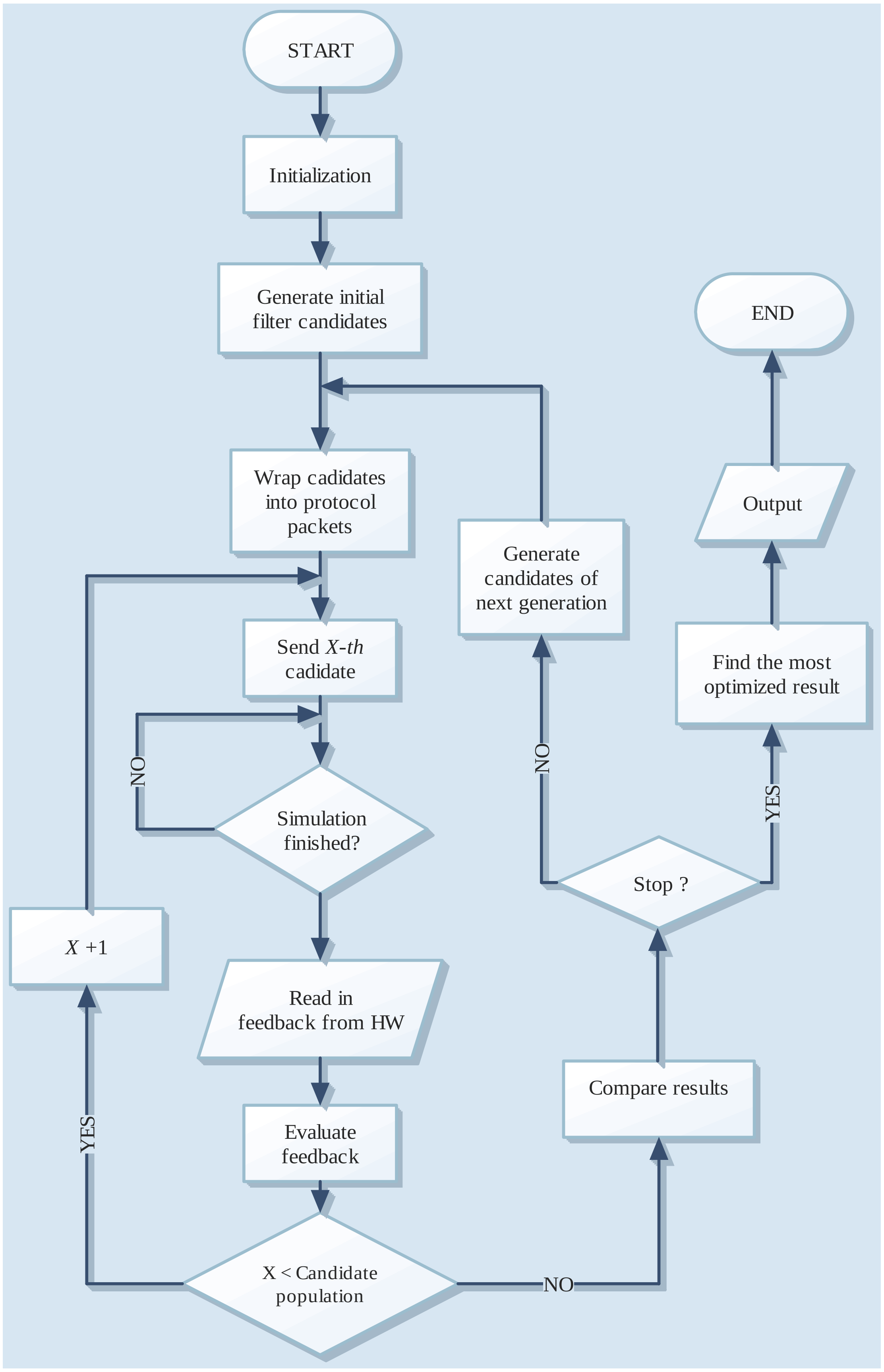


Figure 3.6 – Flow Chart of SW Design

3. Platforms for System Design

When choosing design platforms for this project, a few important facts must be taken into consideration. First, the optimization engine requires many complex matrix operations and intricate array rotations. Second, the process of filter analysis involves frequency transformation for digital data and other related advanced algebra functions. In addition, the system is constructed by different sub-systems located on two separate domains. Last but not least, the software program runs co-simulations with a physical version of the hardware design implemented on an FPGA chip. So in reality, this system demands a feasible and efficient development environment in which the hardware and software modules can be co-designed, co-debugged, and co-verified. The design platforms used for this project were MATLAB 7.0.1 R14 with Simulink from MathWorks, and System Generator 8.1 for DSP from Xilinx.

a. The MathWorks MATLAB® and Simulink®

MATLAB, short for Matrix Laboratory, is an innovative highly-integrated numerical computing and programming environment provided by The MathWorks. With the exceptional ability of matrix manipulation, data analysis and algorithm development, MATLAB has been a valuable scientific application tool for researchers, especially in the fields of digital signal processing, communications and engineering computation. Featuring a vast collection of array functions and the ability of flexible-but-simple matrix operations, MATLAB provides a well-integrated platform for developing and implementing a high-performance optimization engine for designing digital systems. Besides benefiting from the extensive frequency analysis and domain transfer functions, as well as other built-in or add-on data processing toolboxes, analysis of the performance of each design candidate can be easily created and evaluated.

As an advanced supplement to MATLAB, Simulink is a powerful platform for multi-domain modeling, simulating, and analyzing for dynamic systems. Unlike the script-based design within MATLAB, Simulink utilizes comprehensive block libraries customizable for specialized applications. Moreover, Simulink uses an interactive graphical interface environment for building system models as block diagrams. The hierarchical models give users the advantages of developing, prototyping and exploring a complicated system design using different approaches. Through the support of more than 300 third-party solutions, designs across applications and industries can be easily conducted at a low cost of both time and resources. With the above benefits and the tight integration with MATLAB, Simulink was the first choice for the development platform for this project.

b. Xilinx System Generator™ for DSP

Xilinx System Generator (Sysgen) for DSP (digital signal processing) is a MATLAB/Simulink-based signal processing modeling and designing tool for high-performance digital systems. Sysgen provides Xilinx blocksets that contain functions for different purposes in the DSP area, allowing engineers to design, simulate and implement complicated DSP systems optimized for Xilinx FPGAs. Given the environment of high integration with MATLAB and Simulink, designers can easily develop complicated digital circuits, which may be difficult to be described using a hardware description language (HDL), such as control circuits, by combining the imported MATLAB functions. Besides, HDL modules can also be imported into a Sysgen model. Designers gain the facility or ability to create individual modules using a desirable developing tool. But the debugging and verification of the entire system can be performed in an integrated environment, saving design time and resources.

System level modeling is only one aspect of Sysgen's capability. The others, which are relevant in this project, include automatic generation of HDL code mapped to the Xilinx FPGA and hardware co-simulation. All of the code generated by Sysgen from the system level model can be synthesized and implemented in a Xilinx FPGA.

The hardware implementation can be brought back to the original system design for verification purpose. This kind of simulation is called a “FPGA-in-the-loop” hardware co-simulation, which utilizes the fast processing capability from a hardware core to accelerate the simulation process. It can also be used to verify and analyze the actual hardware implementation of the Sysgen model. Thus, Sysgen provides engineers a sophisticated platform for developing, simulating and implementing bit-true or cycle-true models for DSP systems.

IV. Designs of Hardware and Software Modules

In this chapter, information on how each module was developed from conception to implementation will be discussed in detail. Within the discussion of designing the individual modules, introduction to some important concepts which are necessary to fully understand the mechanism of the component will also be covered.

1. Designs of Hardware Modules

As described in Chapter III, hardware modules for this project were developed under the integrated environment of Simulink and Sysgen. Instead of text-based coding, Sysgen/Simulink provides to the designers a graphic interface for programming, which allows users to design systems by the method of arranging and interconnecting components or sub-systems built with provided basic blocks or advanced intellectual property (IP) blocks from a third party. Throughout the whole design process, the creator does not have to understand the mechanism working inside any of the blocks, because these blocks are black-boxes to the designer. What the designer needs to consider, when using a block, is the inputs, outputs and, most importantly, the parameters that determine how the specific block performs its tasks.

The hardware part of system can be divided into six modules. During the discussion of system level design, the system model and its data flow have already been described in Figure 3.5 in Chapter III. Now, to demonstrate how a system model is constructed within Simulink/Sysgen environment, the actual Sysgen model for this project is given in Figure 4.1. Considering the different purposes the system needs to serve, a Sysgen design can be dissected into three sections: the Simulink section, the input/output gateways, and the hardware synthesizable section.

The major task of the Simulink section, which is colorless in Figure 4.1, is handling all the input and output communications between the Simulink/Sysgen model and the MATLAB workspace. The Simulink source on the left is used to fetch candidate data from the MATLAB workspace to drive the whole system, while the Output block on the right is responsible for returning results from the model back to the workspace such that they are accessible to the software counterpart. Besides, controlling the simulation and data exchange is also an important job of this section. This gives the users more flexibility in designing a simulation and cooperating Simulink model with MATLAB scripts. For this system, the hardware portion is configured to send an enable signal to the Simulink to identify the end of a successful processing of the set of test data, thereby enabling output of the results and pausing of the simulation.

Digital systems are operated on fixed-point numbers only, while the MATLAB/Simulink models operate on double floating-point values. So there must be an intermediary between the Simulink and synthesizable hardware sections to convert data into the target format. The In/Out gateways, colored in light yellow on the picture, play the role of this kind of intermediary. In Sysgen the method describing a fixed-point number is defined as a 3-portion notation, first the type of the fixed point number (FIX/UFIX, for signed/unsigned), then the width of the number and then the position from the LSB (least significant bit) of the decimal point. For example, *Gateway_in_coef* converts the candidate's coefficients from the format of double into that of fix_32_29, which means a 32 bits signed fixed-point number with 29 bits for fraction. As shown next to the input and output ports of the gateway blockset, *Gateway_out_data* turns the processed data form fix_34_29 back to double, and *Gateway_out_EN* changes a Boolean type of enable signal into double for further operations by MATLAB programs. Besides, the gateways also represent the input and output of the generated HDL top-level entity and the pins of the device.

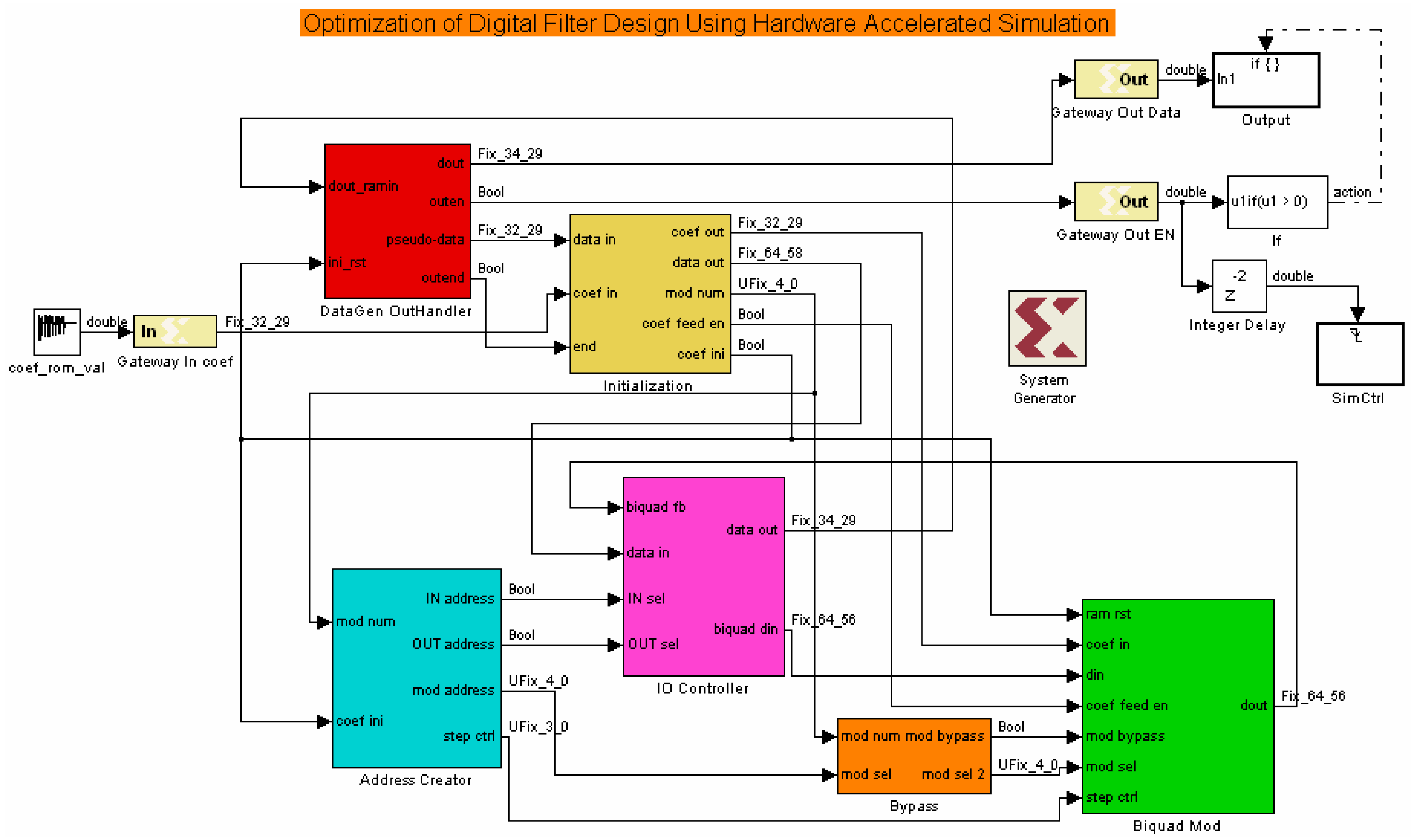
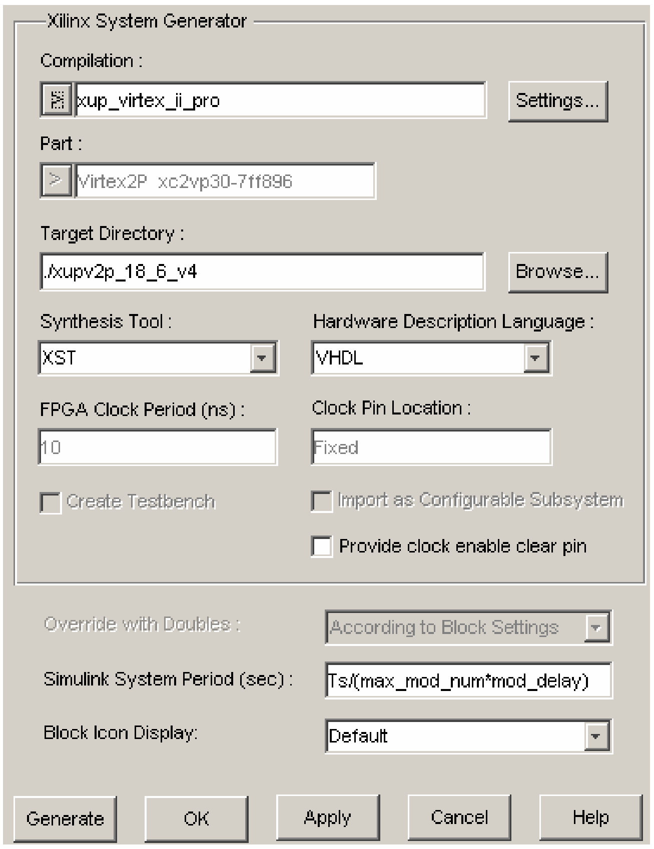


Figure 4.1 – Simulink/Sysgen Model


Xilinx System Generator

Compilation : xup_virtex_ii_pro

Part : Virtex2P xc2vp30-7ff896

Target Directory : ./xupv2p_18_6_v4

Synthesis Tool : XST

Hardware Description Language : VHDL

FPGA Clock Period (ns) : 10

Clock Pin Location : Fixed

Create Testbench

Import as Configurable Subsystem

Provide clock enable clear pin

Override with Doubles : According to Block Settings

Simulink System Period (sec) : Ts/(max_mod_num*mod_delay)

Block Icon Display: Default

Figure 4.2 – Properties of Sysgen Token

The rest of the colored blocks in the picture together are called the hardware synthesizable section, which can be compiled, synthesized and implemented in a Xilinx FPGA. Each color block represents one module, which can be viewed in detail by double-clicking it. The solid arrowed lines between the subsystems stand for inner communications between modules and their directions. Any Sysgen model must have at least one System Generator token presented which is used to configure the important properties of the Simulink/Sysgen model, including Simulink system period, target part and FPGA clock period, as shown in Figure 4.2. The Simulink system period represents the smallest sample period of the system and all other sample rates must be integer multiples of the system period. In its hardware counterpart, this parameter also defines the system clock that drives the design. Combined with the FPGA, the CLK period also reflects the timing constraints that are used to gain desired timing performance when implementing a design. Detailed information on implementation of the design will be covered in Chapter IV. The following section of this paper will talk about the design process of each of the following six hardware modules:

- Initialization
- Data Generator and Output Handler
- Address Creator
- Biquad I/O Controller
- Bypass
- Biquad

a. Initialization Module

The structure of the initialization module is illustrated in Figure 4.3. The main purpose of this circuit is to initialize the system and to prepare incoming data for further processing. Before exploring the actual design, brief explanations of the I/O ports will be given to help better understanding of this subsystem.

- DATA_IN – This is the pseudo-random test data from *DataGen*, which are used to evaluate the candidate.
- COEF_IN – This is the incoming data from MATLAB/Simulink, including the coefficients of the filter candidate and controlling header, all of which are grouped as a frame obeying the established protocol.
- END – This is a signal from the *DataGen* module identifying the end of one successful run when it is active.
- COEF_OUT – This is the set of cached coefficients that are sent to the memory block in the *Biquad* module.
- DATA_OUT – The test data which are multiplied by normalization factor *g*.
- MOD_NUM – This is the total number of SOS sections for the candidate.
- COEF_FEED_EN – The enable signal for feeding coefficients into the *Biquad* when the data is prepared and ready.
- COEF_INI – An initialization signal for all modules to reset their RAMs.

Defined by their different purposes, three parts can be identified from the structure diagram. The upper portion of the diagram describes the circuits to examine the incoming data frame. According to Figure 3.3, a complete packet should contain a start flag, followed by a UID, the number of SOS sections and coefficients of each SOS. With the help of a pre-set special value, which serves as the start flag, the initialization module determines the start of an incoming data packet. Following recognition of the flag, the system will enable a register to store the next value, the UID of this data packet. Then the same value will be compared to the one registered from the last frame. If the two are equal, it means the latter one actually is a redundant version of the previous one. This mechanism prevents the hardware implementation from wasting power to refresh the cache memory for an identical set of coefficients. It significantly reduces the power consumption, which is one of the crucial criteria for evaluating the performance of an FPGA design. Once a new UID is received, the module will refresh the UID register, store the values for *g* and the number of SOS sections required to describe the candidate filter, and send a trigger signal to activate the caching of the coefficients into a RAM. There is a converter right before the output port MOD_NUM. Since the value of MOD_NUM is used to control the number of *Biquad* module loops, it has to be converted to an unsigned integer from a signed number with fractions. All the delays in the circuit are necessary for the synchronization between different components in the circuit. They are also helpful to increase the driving ability and to reduce the probability of having timing issues after realization. Because it is difficult to understand the state transfers by examining the graphic interface within Simulink/Sysgen, a state diagram is given in Figure 4.4, to explain how these procedures are conducted.

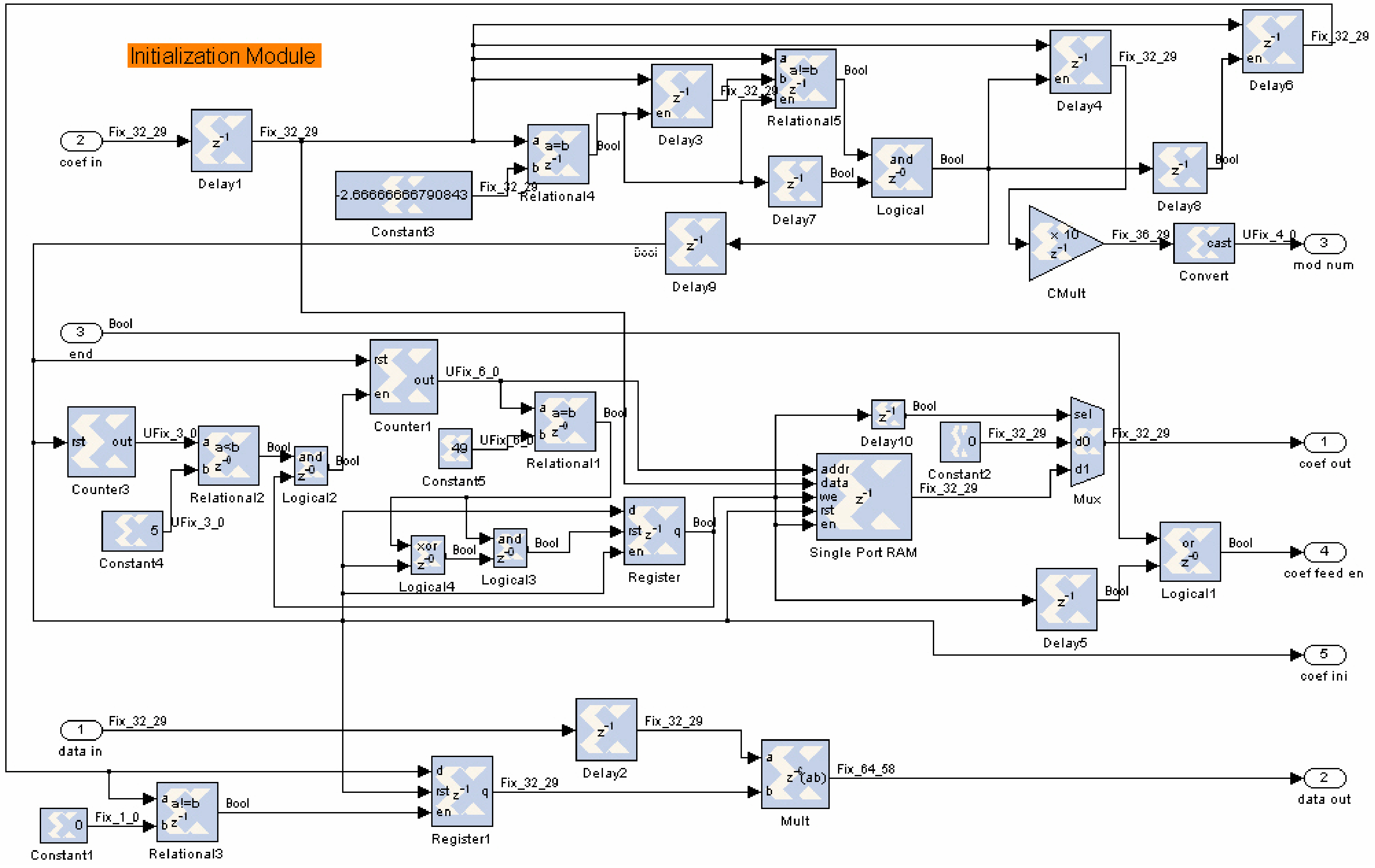


Figure 4.3 – Structure of the *Initialization Module*

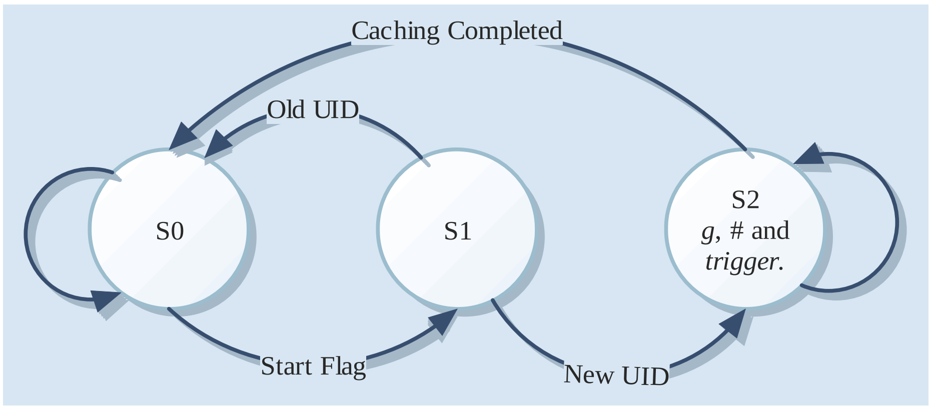


Figure 4.4 – State Diagram of Input Examining Circuit

The middle part in the diagram describes the circuits to separate the coefficients data from the complex incoming packet and to cache them into a RAM. Recalling from Figure 3.3, the incoming data is constructed of control signals and coefficients. In order to be compatible with the *Biquad* module timing, three dummy values are stuffed right after the five coefficients of each SOS section. This is because a successful run of the *Biquad* module takes eight clock cycles. Actually, during the early developing period, two methods were proposed to solve this problem. One was the “value-stuffing” described below, and the other was adding an extra and complicated control circuit to handle the synchronization between modules. For an FPGA design, extra circuits mean higher costs of area, power and delay. After comparing the costs, complexities and performances from simulations of the prototypes using the two proposed methods, the later one was selected.

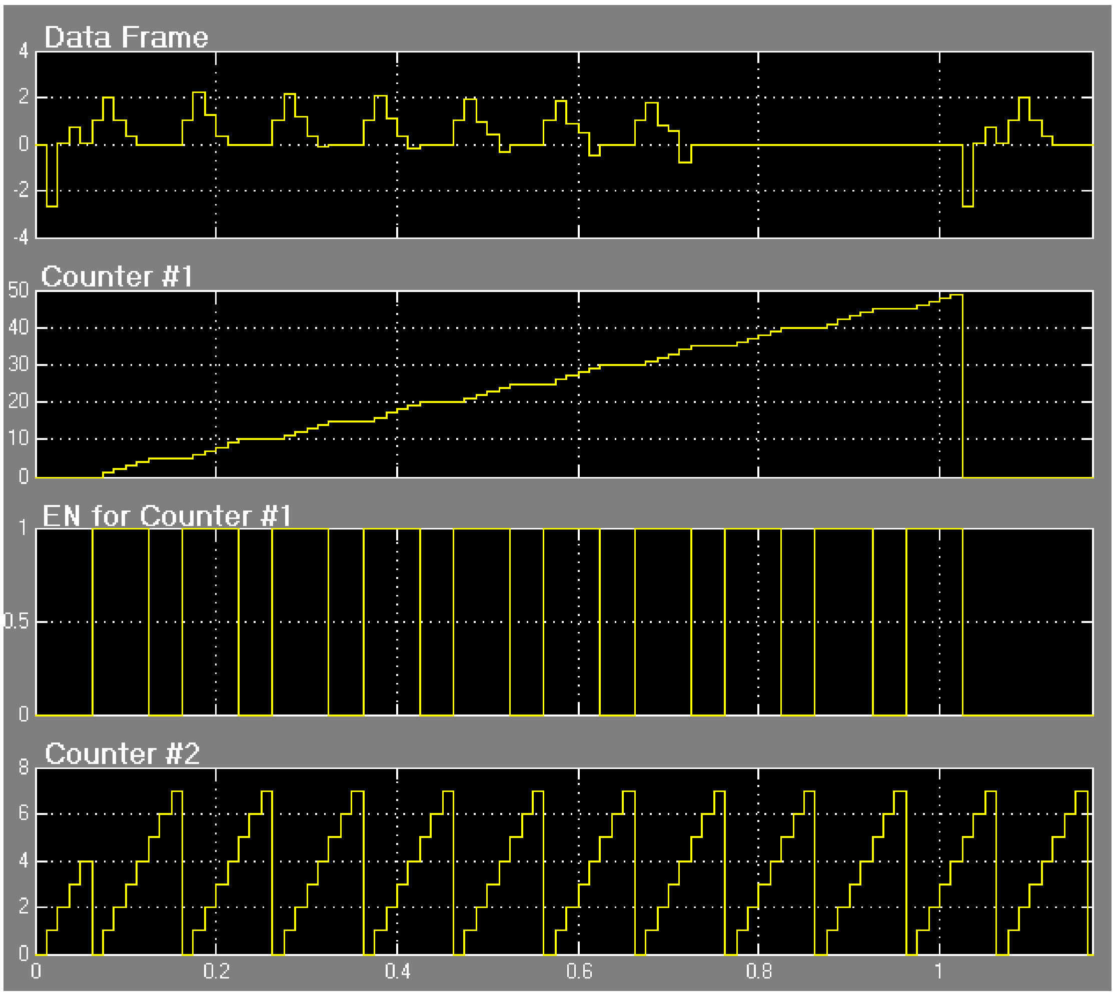


Figure 4.5 – Waveforms of Two Counters

To facilitate deleting the dummy values when storing coefficients into a single port RAM, a control circuit consisting of two counters and some associated logical components was needed. The graphic in Figure 4.5 shows how the two counters cooperate together to select the desired data from the packet. First of all, counter#2 counts from 0 to 7 repeatedly. An enable signal for counter#1, repeatedly counting from 0 to 49, is driven high whenever the value of counter#2 is less than 5. Only when the enable signal is active does counter#1 advance its value. The output from counter#1 is used as the address signal when accessing the RAM. Then with this scheme, coefficient data for at most ten SOS sections can be stored in the RAM of size 50x32 bits. In order to save as much power as possible, several small circuits were added to make sure the writing to and reading from the RAM occurs only once for each new case, such as when several logic expressions are applied to drive the EN, WE and RST ports of the RAM blockset. An important lesson was learned when developing the model. If the contents in the RAM are not cleared before the arrival of a new candidate, the calculations of the new run will be disturbed greatly by the remainders, even though the value of them is very small. So for insurance purposes, a MUX was added to make sure no remainder was sent out just in case the RAM was not reset correctly. At the same time of caching the coefficients, a feed enable signal was set active to inform the *Biquad* module to load the coefficients. This is a pipeline process, which greatly helps to improve the design's performance.

The circuit in the lower part of the diagram performs a simple task, multiplying the test data with *g*, the normalization factor or gain from the transfer function. Attention had to be paid to the property of latency when using the MULT cores. If the latency is set to be too low, the timing constraint for this circuit might be impossible to be reached when implementing the design. As the bit-width of the number waiting to be processed grows, the value of the latency should increase accordingly. In this case, the MULT latency is six clock periods for a full precision multiplication of two fix_32_29 numbers.

b. Data Generator and Output Handler

Figure 4.5 illustrates the architecture of the module of data generator and output handler. The major tasks that this module can perform include up-sampling the reset/initialization signal to make it multi-rate compatible, generating pseudo random vectors as the test data for filter candidate, and caching processed data for output purpose. There are 2 input ports and 4 output ports.

- DOUT_RAMIN – This is the input port for the result data that are ready for output caching.
- INI_RST – The initialization/reset signal.
- DOUT – Data output port, through which data will be send back to the

software part for further evaluations.

- OUT_EN – This is the signal informing Simulink to save the output data as a matrix in MATLAB workspace.
- PSEUDO_DATA – This is the test vector for the evaluations of the filter candidates.
- OUTEND – The active of this signal means the end of an evaluation.

In order to fully understand the design of this module, one key initial concept must be explained clearly. This design is a multi-rate system, which means there is more than one sample rate or clock source in action. For this particular system, two sample rates are applied, one of which is the rate for the test vectors, and the other for the imported data from MATLAB. As discussed above, an IIR filter with any transfer function can be represented as a cascade of SOS sections. For one successful filter operation, one single datum must be processed by all the SOS sections one by one. Thus all the calculations required must be finished in one input sample period. This demands that the sample rate for the data that are used for theses calculations be integer times faster than that of the input data. For example, in this system the incoming data rate is *max_mod_num*mod_delay (10*8)* times faster than the rate of the test vectors. On the other hand, if a signal is input into a circuit which is running under a different sample rate, it has to be up-sampled or down-sampled to the same rate as the target.

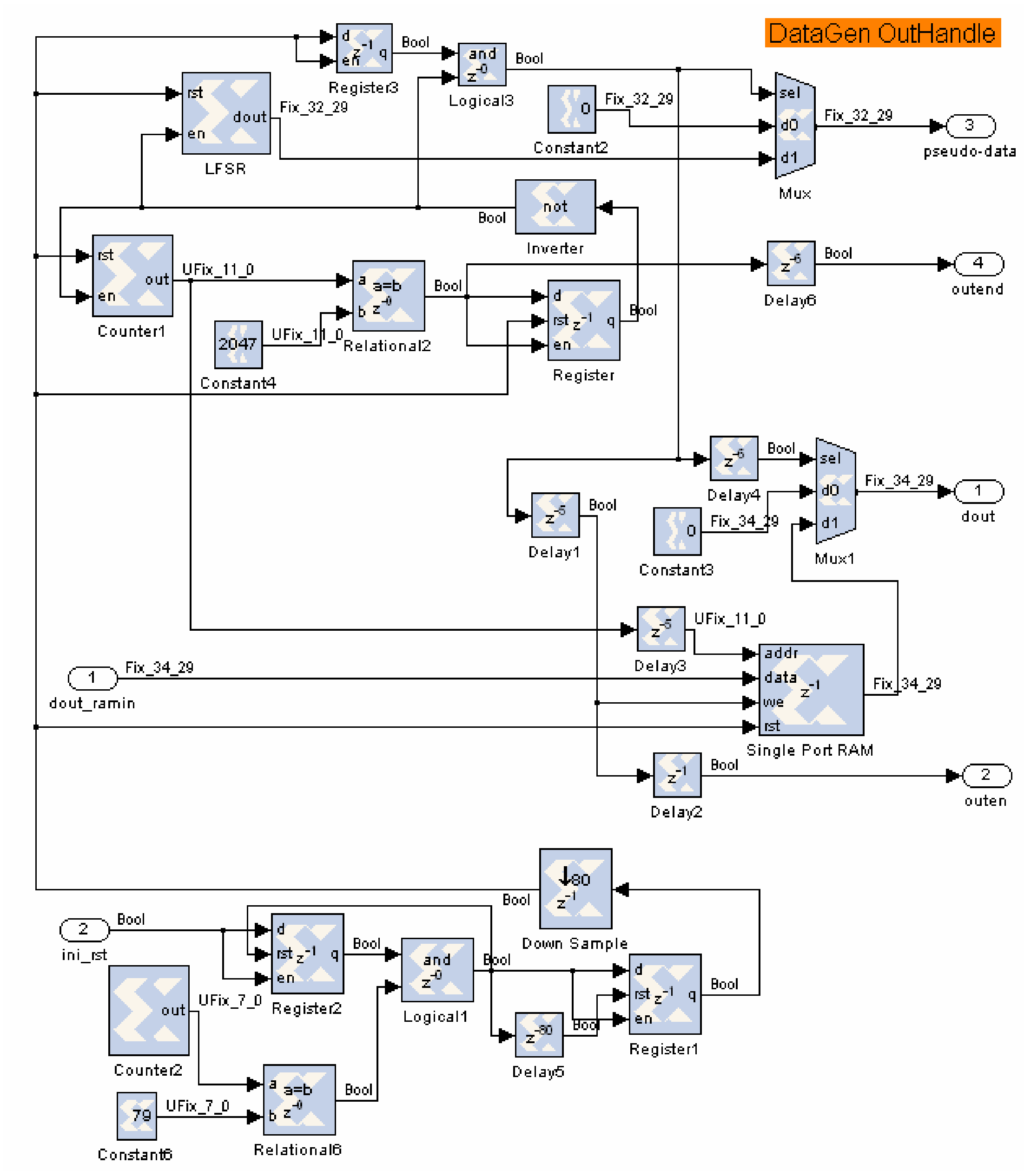


Figure 4.6 – Structure of the *DataGen Module*

To start the generation of test vectors, an active initialization signal must be received from the initialization module. Before this INI_RST signal, whose sample period is T_s can be used to stimulate the generator, a down sample blockset is utilized to change its period in $T_s/80$. Though a down-sample block can be configured to sample either the first or the last value of the frame, the later one is most efficient when it is implemented on an FPGA. Its implemented representative in the hardware domain is a D flip-flop, which samples the input data at the end of the frame, and outputs the value for the duration of next frame. Since the *ini/rst* signal will occur in any interval within one frame, a circuit was developed to make sure that, once an *ini/rst* signal is received, it will be registered and occurs at the end of that particular frame. This circuit is represented in the lower part of the module diagram, and its effect is illustrated in the waveforms in Figure 4.7.

For an efficient data generator, the test vectors that are used to evaluate all possible candidate designs must meet the following requirements. First, the vectors must be exactly identical in both values and sequence every time they are processed by different candidates. It is important because the evaluation results are not comparable or consistent if variant test vectors are used to estimate the performances of different

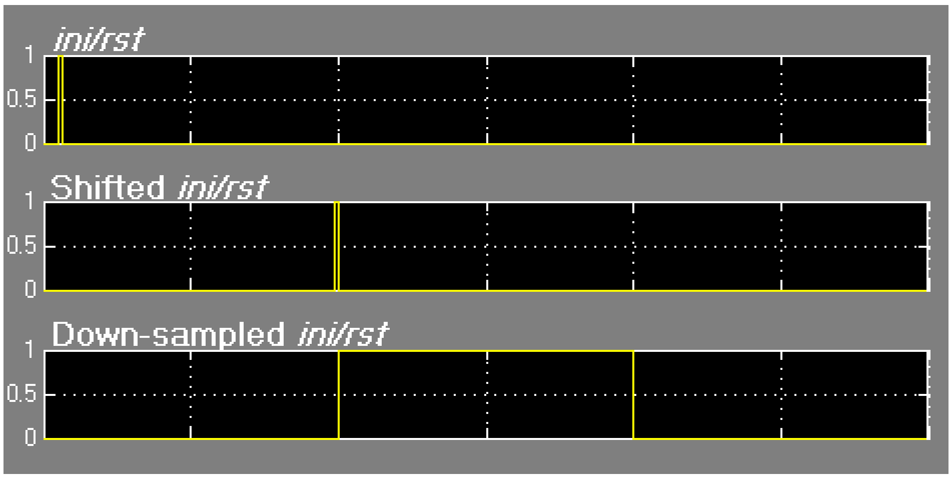


Figure 4.7 – Result of Down-Sampling

designs. Second, there cannot be any statistical pattern existing within the generated sequences. Random numbers are widely used by engineers and scientists to test their applications. However, there is no any way to produce true randomness with current technologies, because the deterministic algorithms that are used to generate the data imply the outcome is not truly random. Pseudo-random numbers, which appear to be statistically random for most practical purposes, were acceptable for this project. Third, the sequence generation must be at a really high speed, but the generator circuit should be easily implemented on the hardware.

In the digital system industry, a linear feedback shift register, LFSR, is generally applied for fast generation of test vectors with both deterministic and random properties. The format of test numbers used in this project is fix_32_29, so a 32-bit LFSR (linear feedback shift register) device is employed to act as a pseudo-random number generator. The LFSR must be a maximal one, in order to ensure the test vectors do not repeat during one simulation. The appropriate taps for a maximum-length LFSR counter in XNOR form is 1, 2, 22 and 32. A graphic illustrating the feedback structure of this maximal LFSR is presented in Figure 4.8. With this configuration, the LFSR blockset can generate at most $2^{32}-1$ test vectors before repeating itself. After many trial simulations, the number of test vectors for a candidate was set to be 2048. The upper part of the diagram shows the data generating

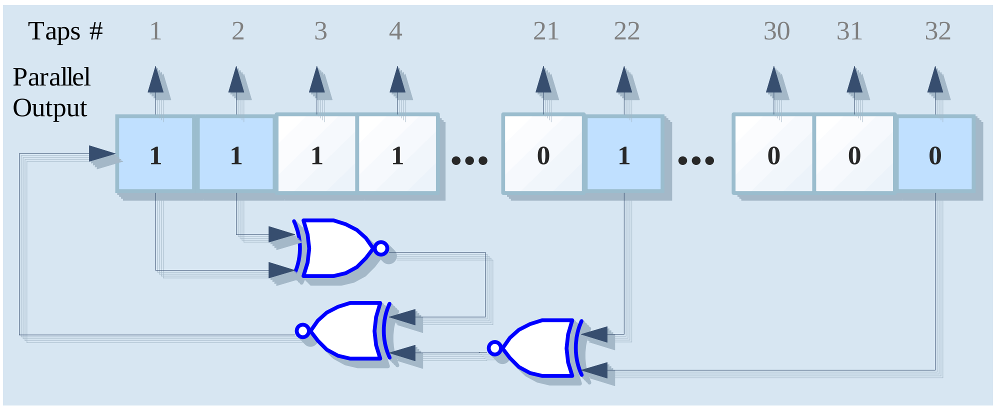


Figure 4.8 – Feedback of a 32-bit LFSR

circuit. Keep in mind that this part of the circuit is running at the lower rate, so the output pseudo-random data should be up-sampled before being used by others.

The remaining item shown in the diagram is the circuit to caching output data, where a 2048x34 single port RAM is used. It will be very helpful when the system is configured as a “Free Running”. More information will be covered later.

c. Address Creator

Address Creator Module is designed to produce all kinds of address signals that are used for the RAM accesses within *Biquad Module* and for the selections of corresponding I/O data for different SOS stage. From the structure diagram shown in Figure 4.9, 2 inputs and 4 outputs can be identified.

- MOD_NUM – This is the total number SOS sections the candidate has
- COEF_INI – A signal indicating the start of a new round.
- IN_ADDRESS – This is used to choose the data source for the Biquad Module of different stage.
- OUT_ADDRESS – This is the signal to determine whether an output from the Biquad Module should be sent to the output memory or not.
- MOD_ADDRESS – This is the address signal for RAM operations within Biquad Module
- STEP_CTRL – this is the signal directing the Biquad Module to transfer data from the one tap to the next tap. Detailed information will be discussed in the section for Biquad Module.

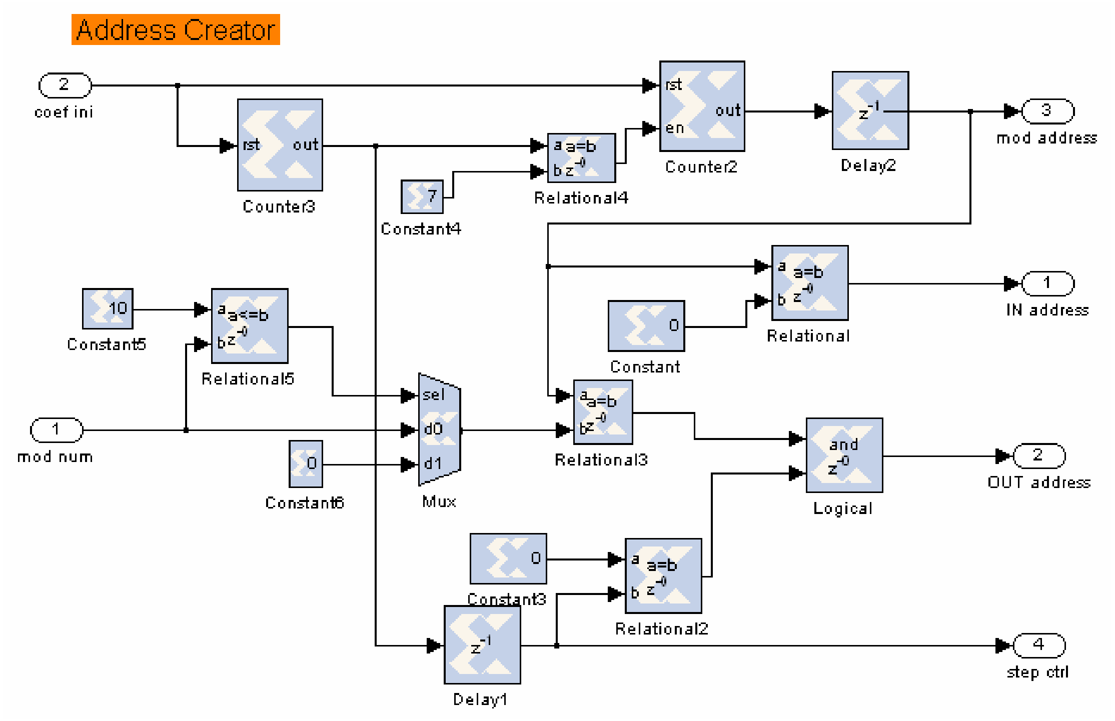


Figure 4.9 – Structure of the *AddGen Module*

The following section will discuss the algorithms that are used to generate all the addresses mentioned above, as well as the considerations when designing those algorithms. The address signals are all instructed by the variables from two counters. One counts from 0 to *max_mod_num*-1(9), indicating which stage the Biquad Module is operation on. The other one counts from 0 to *mod_delay*-1(7), suggesting the current step of the running Biquad Module. Thus the output from the first counter after one unit delay is sent to output port MOD_ADDRESS, and the output from the second one is used as the step control signal for the Biquad Module. Figure 4.10 is given to illustrate the connection between these two key concepts.

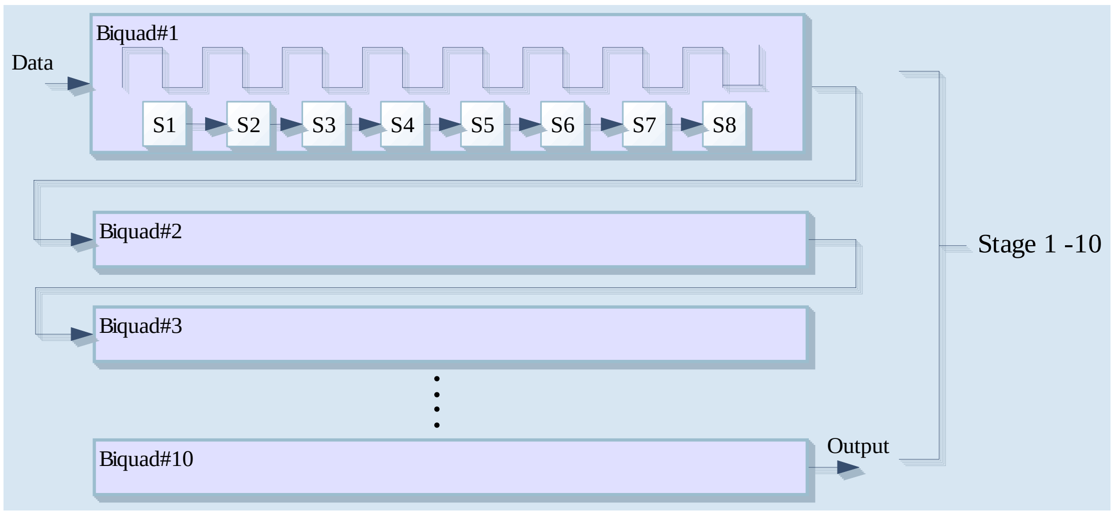


Figure 4.10 – Relationship between Stage and Step

In this design, Biquad Module is designed to play the role of a second order filter. Any high order IIR filter can be represented by a cascade of several second order filters. In order to simulate the operations of a high order filter, the Biquad Module must be executed serially for several times for processing one test datum. For example, 7 cascaded biquad filters is equivalent to a 14th order filter. The term cascade means that the executions of the 7 biquad filters are not simultaneous but sequential. Excepting the first one, the input for each biquad is the output from previous one. The signal input into the first *Biquad* is the pseudo random test numbers, and the output from the last *Biquad* should be routed to the system output port. So the algorithm in the form of pseudocode for generating the signals of IN_ADDRESS and OUT_ADDRESS is given below. A figure of waveforms for these signals is also given in Figure 4.11.

Limited by the scheme, the system can only simulate a filter of up to 20th order, which means the design will not perform correctly if the *mod_num* is large than 10. From the diagram, a circuit is applied to prevent any further operations for candidates beyond scale.

```
IF new candidate THEN
   INI stage_counter, step_counter
   INI in_address == FALSE, out_address == FALSE
   READ mod_num
   IF step_counter == mod_delay-1 THEN
      Stage_counter = stage-counter + 1
   END IF
   IF stage_counter == 0 THEN
      in_address = TRUE
   ELSE
      IF stage-counter == mod_num THEN
         IF step-counter == 0 THEN
            out_address = TRUE
         END IF
      END IF
   END IF
```

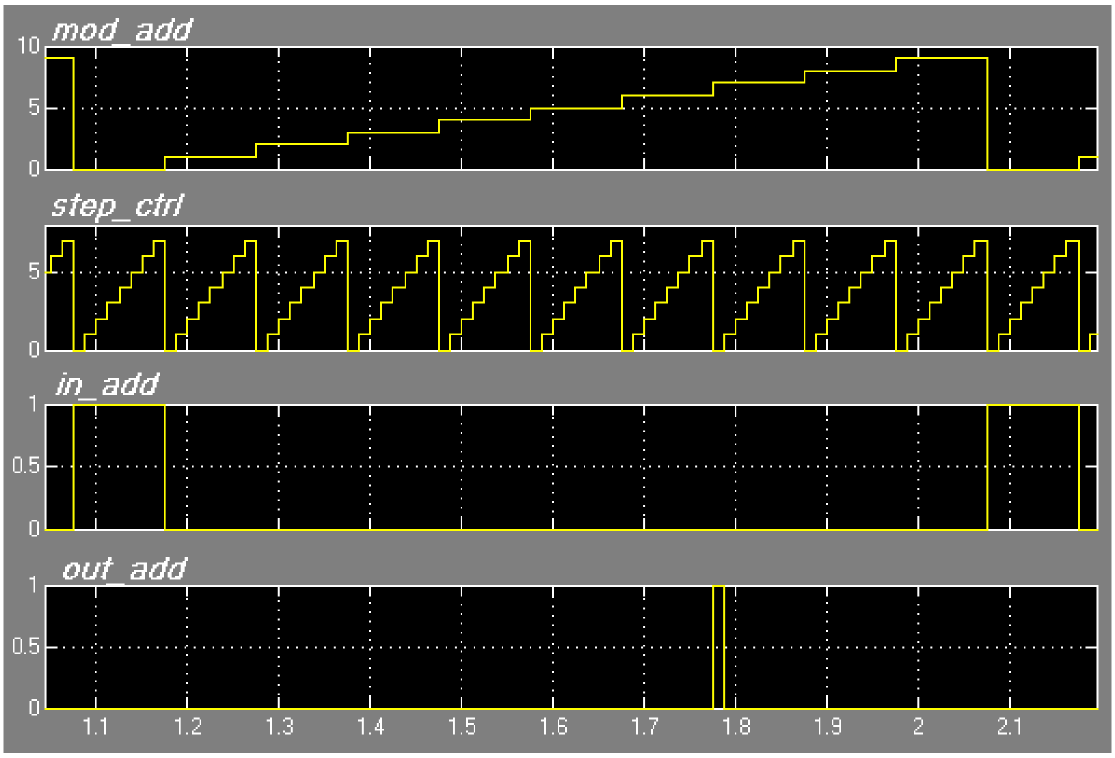


Figure 4.11 – Waveforms for 4 address signals

d. Biquad I/O Controller

The *Biquad I/O Controller Module* is in charge of the selection of the correct input source for the *Biquad Module* executed in a different stage. Furthermore, after a test datum is processed, registering the final result output from the biquad filter of the last stage is also an important task of this module, the architecture of which is indicated in Figure 4.12. In total, four inputs and two outputs are depicted in the diagram.

- BIQUAD_FB – This is one of the two input sources for the biquad filter of the next stage. It is actually the results fed back from the currently executing biquad filter.
- DATA_IN – This is another input source, which is the pseudo random number generated by DataGen Module.
- IN_SEL – This is used to select the correct input.
- OUT_SEL – This is used to control the outputting of the final result.
- BIQUAD_IN – Signal that is fed into the Biquad Module.
- DATA_OUT – Final results that are ready for system output.

Two data sources are to be selected as the input for the *Biquad Module*, depending on the current stage on which the simulated filter is working on. One is the test data from the generator and the other is the feedback result from the biquad filter on the last stage. A MUX device with two input ports is applied to do the job, whose select signal is from the AddGen module. From the diagram, a converter, which converts the test vectors from fix_64_58 to fix_64_56, is posited before the input port of the MUX for test numbers. The reason for this arrangement is to ensure that the format of the two sources is the same. Details on how the formats for these signals are determined will be presented in later sections. As described before, the data generator is running on a different sample rate than the other circuits, so the data must be up-sampled before they can be used by the *Biquad* module. Within Sysgen, the sample period of a device with multiple input ports can be derived from the input signal with the shortest period. For this reason, there is no up-sample blockset occurring in the circuit.

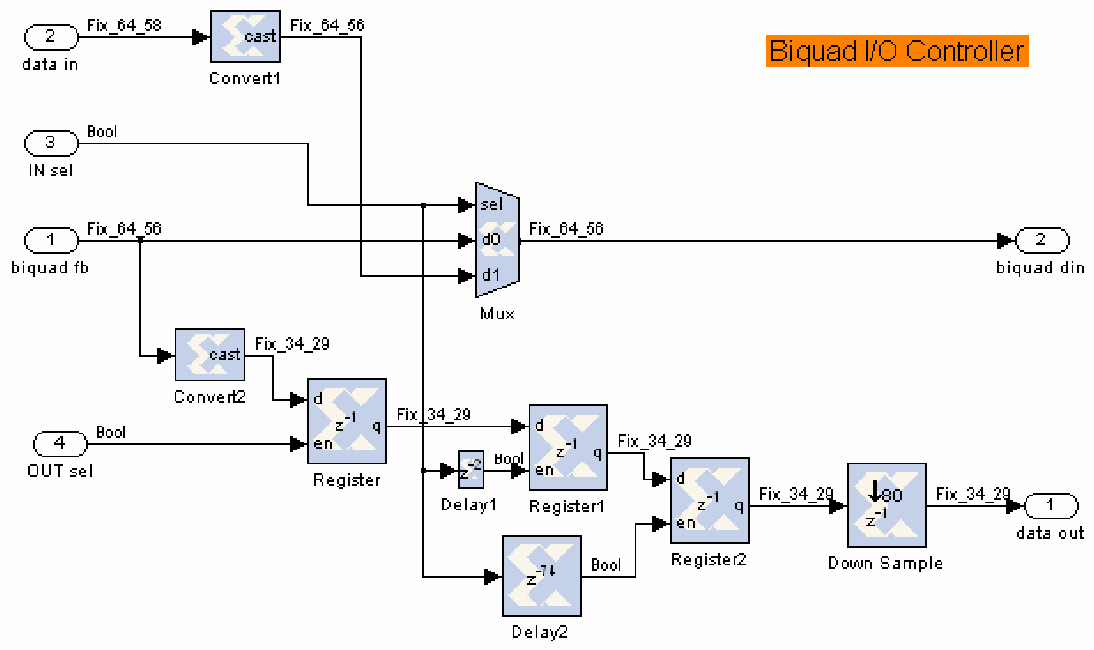


Figure 4.12 – Structure of *IO Controller Module*

Besides serving as one of the inputs, the result signal from the *Biquad* module is also the final result of the filter candidates. An enable signal from AddGen module is used for the register to cache the final result at the correct moment. An 80x down-sample block is employed to ensure that the sample rate of the final results is the same as that of the test data. As discussed in Chapter III, the format of the final output from the hardware is designed to be fix_34_29, to which the feedback data are converted by the converter before the register in the diagram. The circuit in between the register and the down-sample component is specially developed to ensure that the output timing for each candidate is identical. During continuous simulations, this scheme is very helpful, since the incoming data packet for a new candidate can be received at any moment within the down-sampling frame. For example, if the *ini/rst* signal, indicating a new candidate, occurs in the last interval, the result after the down-sample device would be output in the following frame, which might cause a problem when evaluating the performance of the filter.

e. Bypass Module

There are two input ports and two output ports for the *Bypass* module. The descriptions for those ports can be found in the discussions for the other modules. Before an efficient scheme was developed to reset or to clear the memories and registers within this design, this module was very powerful and important for the same purpose. In the final version of the design, this module has been further simplified, as illustrated in Figure 4.13. The *mod_sel* signal is compared to *mod_num* first. If the former is smaller than the latter, the current stage, at which the filter is running, is needless. Then a bypass signal is presented to the *Biquad* module to suppress the needless operation. Thus the power consumption for this design can be decreased significantly for filter candidates with lower order.

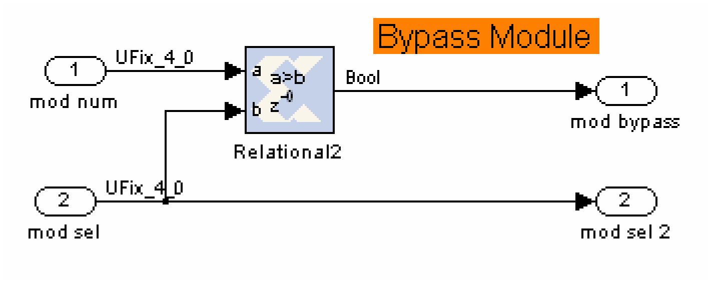


Figure 4.13 – Structure of *Bypass Module*

f. Biquad Module

The *Biquad* module is the most important subsystem of the entire project. In fact, all the filtering processes of a high-order IIR filter are actually realized by the combined operations of this key module and other supporting modules. The core mission of it is to perform as a second-order IIR filter. From the design plan, each execution of the module costs exactly eight sample periods. For each incoming test datum, the *Biquad* module will be executed ten times consecutively for the simulation of a filter with taps of up to 20. Figure 4.14 indicates the structure diagram of the *Biquad* module, which has seven inputs but only one output.

- RAM_RST – This is the signal to reset the memory elements when a new set of coefficients is coming. But keep in mind that, for a RAM the reset only applies to where the address is pointing to when the reset signal is high.
- COEF_IN – This is the signal that contains the coefficients of a candidate.
- DIN – This is test data signal.
- COEF_FEED_EN – Active of this signal informs the RAMs to refresh their current contents to reflect a new incoming candidate.
- MOD_BYPASS – This is the signal to notify the module to suppress operations of RAMs, Mults and Adders. It is set to be negative-active.
- MOD_SEL – This is the address signal indicating the current stage that the biquad is working on
- STEP_CTRL – This is the signal denoting the current step/status of the *Biquad*.
- DOUT – This is the signal for outputting the result from a 2nd order filter.

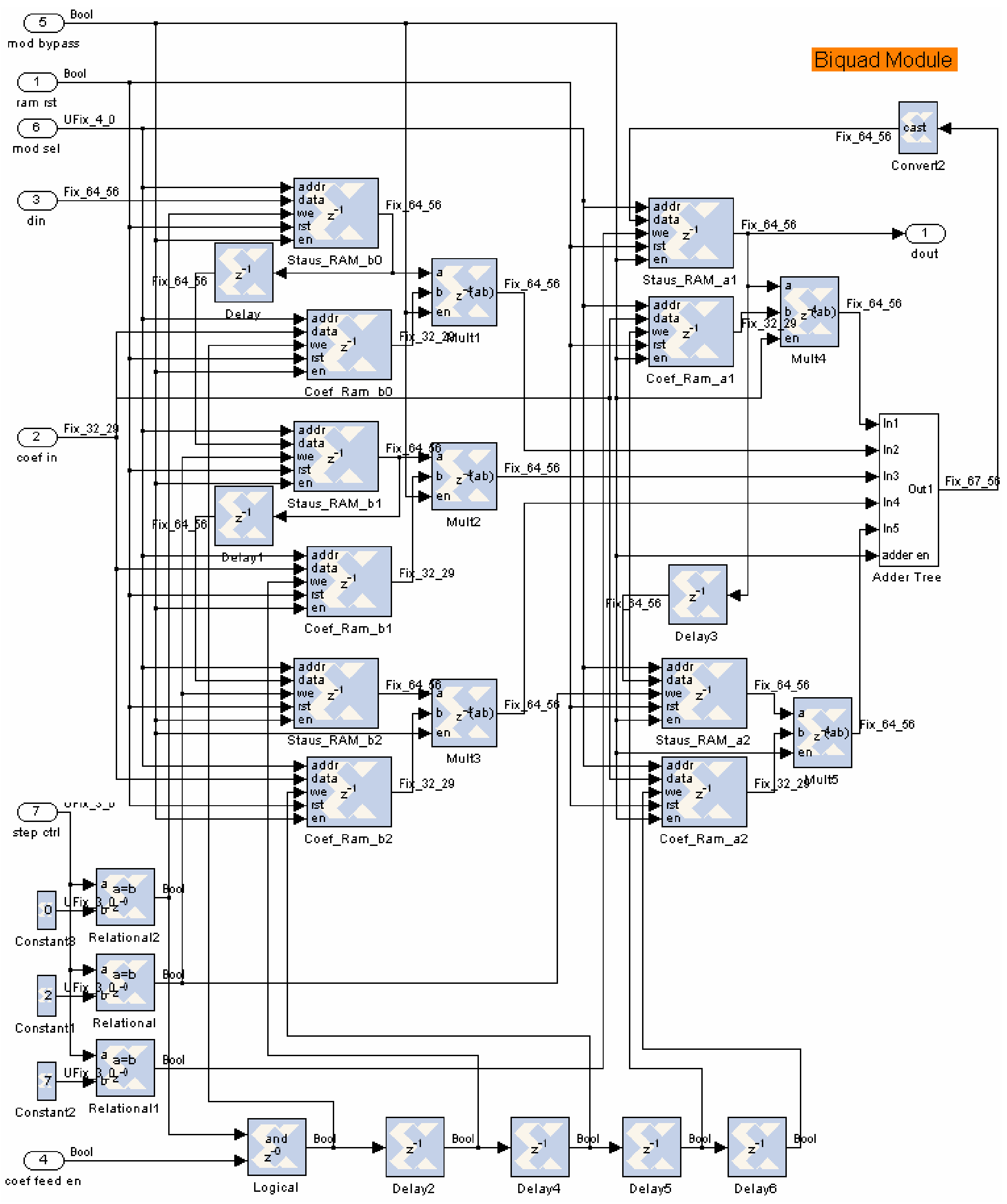


Figure 4.14 – Structure of *Biquad Module*

As discussed in the introductory chapter, an IIR filter can be represented by two equivalent forms: Direct Form I (DFI) and Direct Form II (DFII). In this project, the biquad filter is implemented in DFI. Though the structure of the former seems slightly more complex than that of the later, it is the most straightforward method. Another consideration of the selection is the delay cost, an essential one that affects the performance of the implemented hardware. In the early developing period, implementations using both forms were developed. All blocksets using the same configurations of device delay, the module delay of processing one number for the DFI implementation is one sample period smaller than that of the DFII form. In other words, in this case, that is 12.5% faster. Figure 4.15 illustrates the how the biquad filter processes one datum within eight sample periods. Take the *b0* tap as an example. In the first step, the input signal is written to the Status_RAM_b0, whose latency is set to 1. So at the beginning of the 2nd step, the signal is presented on the output port of the RAM, and at the same time, coefficient *b0* is also ready. The multiplication takes four sample periods, which means the result will be ready at the beginning of the 6th step. Then the result is input to the adder tree, whose operations cost two periods. By the end of the last step, the final result is written to Status_RAM_a1.

Basically, a second-order IIR filter contains three feedforward coefficients and two feedback coefficients, so it can be implemented by a circuit constructed by four delays, ten registers, five multipliers and one adder tree with five input ports. The design of the adder tree, a diagram of which is shown in Figure 4.16, was developed to require less power, delay and area of the FPGA. In this system the *Biquad* modules are supposed to be used for multiple successive times for one test datum. Each SOS section on different stages has its own coefficients and the current values on different taps. These status values must be fully recovered when the filter is working on the same stages during processing of the subsequent test vector. Thus instead of registers, ten RAMs are applied to maintain the status for all SOS sections of the different stages. The signal *mod_sel* is used as the address for accessing RAMs to retrieve the correct values for the current stage.

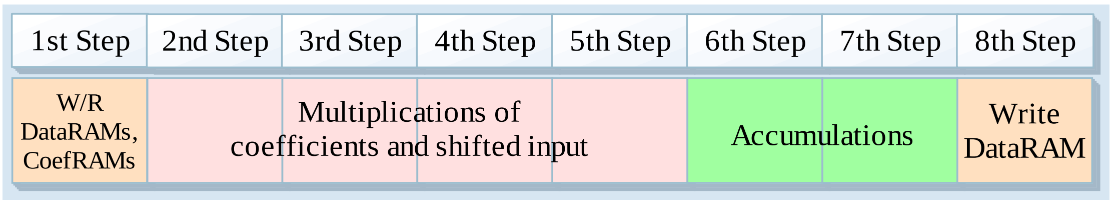


Figure 4.15 – Operations of One Biquad Filter within 8 Sample Periods

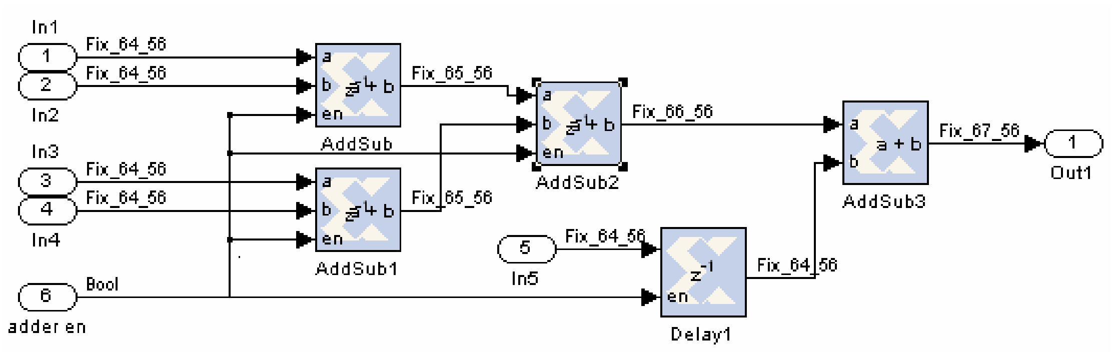


Figure 4.16 – Adder Tree

An IIR filter is very sensitive to quantization, due to the existence of feedback. For the purpose of achieving calculation results as accurately as possible, all the adders and multipliers are configured to full precision. Again, because of the feedback loop, a plan was established to ensure the consistency in terms of formats for the input signal and the feedback. After multiplication with the *g*, the format of the test signal changes to fix_64_58, which is also capable of handling the outcome from the multiplication with a coefficient. However, after the adder tree, at least two bits are needed to be added considering the carry. Considering the trade-off between precision and performance, two LSBs are truncated and the binary point is moved two bits to the right, resulting in the format of both input test signal and feedback signal being fix_64_56.

The control circuits located on the lower part in the diagram are used to control the write-enable ports of all the RAMs. Directed by the signal of *step_ctrl*, the circuit on the bottom-left controls the data exchanging between taps. Three key steps are utilized to handle the transfers of data between the Status_RAMs, which are 1st step, 3rd step, and 8th step (Value of 0 in step_ctrl denotes 1st step). During the first period, an active signal is sent to the write-enable port of Status_RAM_b0, allowing it to store the incoming datum. During the last step, a signal enables Status_RAM_a1 to cache the final result after one successful biquad filtering. Except for the two Status_RAMs mentioned above, all other three Status_RAMs refresh their contents with the data from their nearest up-river counterpart during the third period. Consequently, after eight sample periods, or one biquad filter operation, all the data on different taps are shifted by one unit delay, and they are ready for the next round of calculation. By sending write-enable signals, the circuit on the bottom is in charge of instructing the Coef_RAMs to grab their corresponding coefficients from the input stream at the correct moment.

2. Designs of Software Modules

First, while the hardware portion mainly performs the simulations of filter candidates themselves, the software is responsible for generating, evaluating and optimizing the candidates. Second, the communications between hardware and software are also handled by the software scripts. Finally, the software is also in charge of controlling of the simulation activities, such as starting, pausing and stopping. Four software modules will be discussed in the following section, including one main MATLAB script and three functions.

a. iir_main.m

The operations of this project are started by running this main script. Pseudo-code for this program is list below.

```
INI global parameters
FOR iteration = 1 to iter_num
   CALL iir_DE_fcn.m RETURNING candidates
   FOR population = 1 to popu_num
      CALL iir_coef_frame_fcn.m RETURNING data frame
      IF model unopened THEN
         OPEN module
      END IF
      START simulation
      WHILE simulation running
         WAIT
      END WHILE
      READ results
      CALL iir_eva_fcn.m RETURNING evaluation result
   END FOR
END FOR
OUTPUT result
CLOSE model
```

First, the main script defines the global parameters for the whole system, which can be divided into two categories based on their native properties. The parameters belonging to category one can be varied any time during the simulation, even after the system is implemented on an FPGA. For example, the variables in this category include the number of iterations, the number of population and the initialization parameters that define the optimizer. Parameters in the other category can only be varied for the Sysgen model before the hardware realization, which means the model has to be realized on a hardware core once their values have been changed. The parameters to define the Sysgen model are belong to this category, such as signal formats for test data and coefficients, system sample rate, number of test vector. A special binary sequence, which is *0xAAAAAAAA* in hex form, is constructed by this script to serve as the start flag used for synchronization with the hardware model. Due to its significantly artificial pattern, the possibility of a coefficient having the identical value is extremely small, making it a good start indicator.

Once the global variables are determined, the program enters its first level loop, which is controlled by the number of iterations of simulations. Then the function of the optimizer will be called to generate the first generation of candidates. Since the differential evolution engine was not yet available for embedding into this project, a low-pass IIR filter generator, iir_coef_gen_fcn.m, was used as a substitute. Next, the program enters its second level loop, within which the simulation for each candidate of this generation is executed. The function of the data frame constructor is then called upon to generate a data packet, which contains the filter's coefficients, based on the protocol designed for the communication between hardware and software. The following step is to invoke the simulation for this particular candidate and start clocking. The program will keep tracking the status of the simulation. Once the simulation is paused indicating the completion of a simulation, the program will grab the results from the hardware part and evaluate them by calling the function of iir_eva_fcn.m. The loops continue until the preset conditions are met. After all the designated operations are finished, the program will output the cost value of the most optimized result and the time cost for each simulation.

b. iir_coef_gen_fcn.m

This is a substitute for the DE optimizer before it is ready for embedding into the system. This program is pretty simple and straightforward. The nature of this module actually is a filter generator that can generate an IIR filter with arbitrary cut-off frequency and taps. Of course, a few constraints must be applied to these parameters to match some limitations of the hardware design. The MATLAB source of the function is listed below. From the code, it is clear that the normalized cut-off frequencies for the generated candidates are bounded between 0.2 and 0.8 and the numbers of taps for the candidates are limited from 2 to 19. A MATLAB built-in function, `butter()`, is used to generate a Butterworth digital IIR filter from the parameters.

```
function [b, a] = iir_coef_gen_fcn();

%%% Generate random parameters
filter_freq = rand*0.6+0.2      % Low-pass cutoff freq.
filter_order = floor(rand*17)+2  % Number of filter number

%% Calculation of Coef.
[b,a] = butter(filter_order,filter_freq); % Generate coef
```

c. iir_coef_frame_fcn.m

Frame constructor is a MATLAB function developed to construct data packets which are used to configure the hardware model form simulating the candidate filters. The protocol used to build the packets has already been discussed in previous sections, so this one will discuss more on how MATLAB functions are used to complete the task. The start indicator created by the main program will be included in the frame as the first element, followed by a UID, which is unique for each generated candidate. In order to reflect the moment when the data packet for a candidate is generated, a timestamp is used as the seed when generating a random number. MATLAB function `clock()` is used to obtain the current timestamp, and `rand()` is called to generate the UIDs for the candidates using their own timestamp as the seed.

Since the filter candidates generated by `iir_coef_gen_fcn.m` are represented by its transfer function, the coefficient data have to be converted to an equivalent second order section representation before they can be recognized by the hardware model. The conversion between these two forms is completed by MATLAB function `ft2sos()`. After these preparations are accomplished, all that left is to place the variables into their designed positions within the frame.

d. iir_eva_fcn.m

There are two major missions for this software module. One is to analyze the result data back from the hardware model in frequency domain. Another one is to evaluate the performance of the filter candidate by comparing the analysis results to that of the target filter, which in this project is the ITU-CCIR graphics codec.

In order to analyze the frequency response of the filter candidate, the output signal must be transformed from the time domain to the frequency domain. In digital signal processing, the transformation is performed by the Discrete Fourier Transform (DFT). MATLAB provides many built-in functions that can be used to compute the Fourier transform on discrete signals. In this design, the algorithm of Fast Fourier Transform (FFT) is employed. Before performing the transform, the data are divided into sections, each of which contains 512 elements. For achieving a higher accuracy, each section has an overlap of 256 elements with both its previous and next sections. It is not practical to perform the FFT on the data of a section directly because the signal is finite and is not continuous on its boundaries. Direct operations of the FFT on them will introduce unwanted frequencies, so before the FFT operation, a window function should be applied on the data. The MATLAB built-in function `blackman()` is used in this module to generate a 512-point symmetric Blackman window. This window function will then be applied to all the sections before performing 512-point FFT on them. After changing the data in frequency domain, the magnitude response and the group delay of the filter candidate can be easily obtained.

The criterion of evaluating a filter candidate is the value of its cost function, which reflects the degree that the candidate meets the requirements constrained by the specifications of the target filter. The cost function is constructed following the tolerance schemes for the magnitude and group delay of the graphic codec described in the background chapter. The magnitude response and the group delay of the filter candidate are used to calculate the cost value for this particular candidate by the established cost function. A candidate with a lower cost value is a better design. Once the cost values for the candidates are computed, the results are sent to the optimizer as a guideline for generating a new generation of candidates.

V. Hardware Implementation

In this chapter, detailed information of the implementation of the hardware design on a Xiilinx FPGA prototyping board will be covered. Before the discussion of implementation, a basic introduction to XUP (Xilinx University Program) board, which is the hardware platform for this project, will also be given for a better understanding of the concept of hardware co-simulation.

1. Introduction to XUP Development Board

In this project, the hardware platform, which is used for the final implementation of the hardware design, is the Xilinx University Program Virtex-II Pro Development System, or the XUP board for short. The XUP board is a powerful, multipurpose and low-cost system, which consists of a high performance Virtex-II Pro FPGA with PowerPC cores and a comprehensive collection of supporting components, such as on-board Ethernet device, serial ports and AC-97 audio codec. Figure 5.1 presents a block diagram of the XUP board.

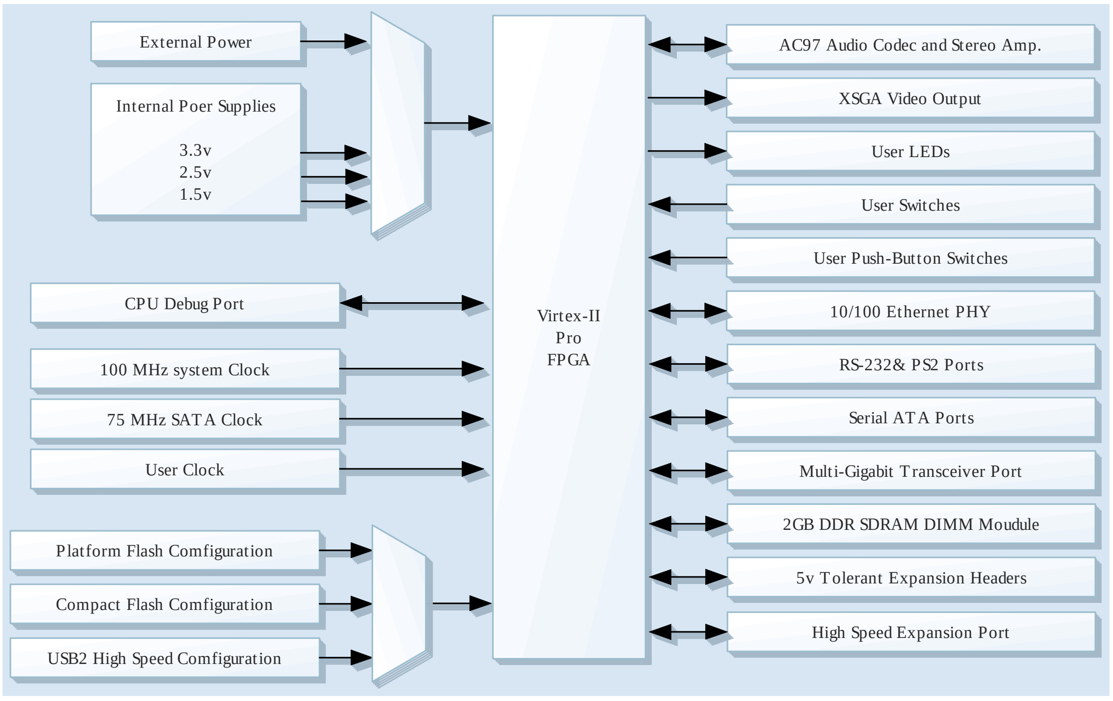

External Power
Internal Poer Supplies
3.3v
2.5v
1.5v
CPU Debug Port
100 MHz system Clock
75 MHz SATA Clock
User Clock
Platform Flash Comfiguration
Compact Flash Comfiguration
USB2 High Speed Comfiguration
Virtex-II Pro FPGA
AC97 Audio Codec and Stereo Amp.
XSGA Video Output
User LEDs
User Switches
User Push-Button Switches
10/100 Ethernet PHY
RS-232& PS2 Ports
Serial ATA Ports
Multi-Gigabit Transceiver Port
2GB DDR SDRAM DIMM Moudule
5v Tolerant Expansion Headers
High Speed Expansion Port

Figure 5.1 – Block Diagram of XUP Development System

The Virtex-II Pro FPGA chip included in the XUP board is xc2vp30, which provides 13969 slices, 428KB distributed RAM, 2448KB Block RAM, 136 Mults and 2 PowerPC RISC cores. The 100MHz system clock provided by the XUP board can facilitate the performance of the hardware implementation of a complicated system. Another important feature the XUP board provided is that the system offers several methods for the configuration of the FPGA. The FPGA can be configured by the data from either the internal flash memory or external ports, such as the embedded USB2 high speed interface. With the help of the USB interface, the Sysgen can perform the hardware-in-the-loop co-simulation.

2. Implementation of Hardware Model in a FPGA

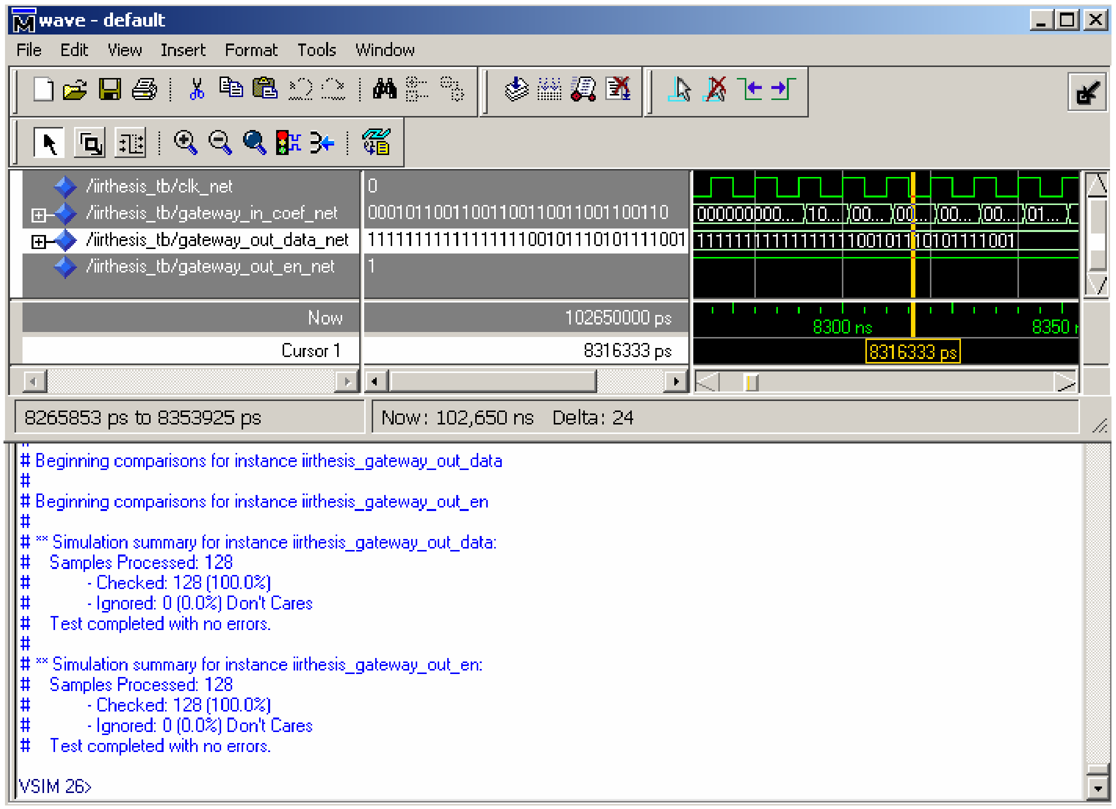
The hardware realization from a Sysgen model to a hardware core can be divided into two phases. During phase 1, the net-list files for the model are automatically generated by Sysgen. In phase 2, all the downstream processes, including building

NGD files, mapping, PARing and generating bitstream files for FPGA configuration, will be performed on the net-list generated in phase 1. As discussed in the beginning of Chapter IV and shown in Figure 4.1, System Generator Block is the key component that controls how the compilation and simulation should be handled. The block specifies all the important parameters necessary for the implementation, such as compilation type, target device and synthesis tools. For this project, the target part is xc2vp30-7ff896, and Xilinx XST is used as the synthesis tool. In order to achieve the highest performance, the Simulink system period is set to be 100 MHz.

a. HDL Net-list and Pre-synthesis Simulation

Sysgen offers automatic generation of two types of net-lists: HDL and NGC. The difference between these two types is the resulting files. Details on NGC will be covered in next section. The result of the HDL net-list generation is a collection of VHDL and EDIF files. The HDL net-list is used most often in designing a system. The automatic HDL net-list generation for this design takes about 20.2 seconds. Three VHDL files are generated: one for the design, one for the test-bench and one for the clock wrapper, which drives the clocks and clock enables. A project file for the design is also generated, allowing the design to be imported into Xilinx ISE environment for further development.

To ensure the automatically generated HDL net-list to be the exact HDL counterpart of the original Sysgen model, verification must be performed before synthesizing the VHDL design or any further actions are taken. The test-bench entity can be used as a wrapper that input the stimuli to the HDL design from the data files generated by Sysgen, and the results compared from the HDL design with those obtained in the Sysgen simulation. Figure 5.2 illustrates pre-synthesis simulation of the HDL within ModelSim and the corresponding waveforms for the system I/O ports. By comparing the results from the two models, the generated net-list was proved to be working 100% accurately as its Sysgen model does with no error. Once the verification



```
/iirthesis_tb/clk_net               0
/iirthesis_tb/gateway_in_coef_net   00010110011001100110011001100110
/iirthesis_tb/gateway_out_data_net  11111111111111111001011101011111001
/iirthesis_tb/gateway_out_en_net    1

Now       102650000 ps
Cursor 1  8316333 ps

8300 ns   8350 ns   8316333 ps

8265853 ps to 8353925 ps    Now: 102,650 ns   Delta: 24
```

```
# Beginning comparisons for instance iirthesis_gateway_out_data
#
# Beginning comparisons for instance iirthesis_gateway_out_en
#
# ** Simulation summary for instance iirthesis_gateway_out_data:
#    Samples Processed: 128
#         - Checked: 128 (100.0%)
#         - Ignored: 0 (0.0%) Don't Cares
#    Test completed with no errors.
#
# ** Simulation summary for instance iirthesis_gateway_out_en:
#    Samples Processed: 128
#         - Checked: 128 (100.0%)
#         - Ignored: 0 (0.0%) Don't Cares
#    Test completed with no errors.

VSIM 26>
```

Figure 5.2 – Pre-synthesis Simulation with ModelSim

procedure was passed with a positive result, the implementation can be advanced to phase 2.

b. Synthesizing the HDL Net-list

After the HDL code has been verified to be valid, the subsequent process is the synthesis of the HDL files to the Register Transfer Level (RTL) net-list. One of the two types of RTL net-lists, NGC or EDIF, can be produced depending on which synthesis tool is involved, XST or EDIF. Since XST was selected for this project, the RTL net-list is in the form of NGC. The NGC net-list is a standalone binary net-list file that contains both the logical and constraint information for the design, such that the net-list can be used as a complete system or as a module of a larger system. Though Sysgen can invoke XST to synthesize the HDL files automatically, the XST command line that is actually executed behind the scene is listed below.

```
run -ifn xst_IIRThesis.prj -ifmt mixed -ofn
IIRThesis_cw.ngc -ofmt NGC -p xc2vp30-7ff896 -ent
IIRThesis_cw -keep_hierarchy NO -iobuf YES -bus_delimiter
() -top IIRThesis_cw -hierarchy_separator / -
report_timing_constraint_problems warning -
register_balancing no -iob Auto -uc ./IIRThesis_cw.xcf -
write_timing_constraints yes
```

It took 142 seconds for the XST to synthesize the hardware design. Useful estimates can be obtained from the synthesis report, such as the minimum slack time, which is 7.845 ns in this case. Table 5.1 shows the device utilization estimation after synthesis.

c. Building Xilinx Native Generic Database

The next process in the implementation flow is using NGDBuild to combine the synthesis result, core net-lists, black-box net-lists and constraint files together, and then to reduce all the components to NGD primitives. Before the NGD file is output, a logic Design Rule Check (DRC) will be performed to verify the converted design. The command used to build the NGD file is listed below.

Table 5.1 – Estimation of Device Utilization Summary after Synthesis

Device: xc2vp30ff896-7	**Used**	**Total**	**Percentage**
Slices	6799	13696	49%
Slice Flip Flops	12765	27392	46%
4-input LUTs	12813	27392	46%
as logic	12788	-	-
as shift registers	25	-	-
Bonded IOBs	68	556	12%
BRAMs	15	136	11%
GCLKs	1	16	6%

```
ngdbuild -p xc2vp30-7ff896 -nt timestamp -intstyle xflow
IIRThesis_cw.ngc IIRThesis_cw.ngd
```

d. Mapping the logical design to a Xilinx FPGA

After a NGD file is created, which contains the logical description of the design, it is input to MAP. Based on the information contained in the NGD file, MAP maps the logic into Xilinx components, such as IOBs and CLBs, in the target device. During the mapping, MAP removes all unused components and nets existing in the logic design. MAP runs a physical DRC on the mapped design before outputting the final result in the form of a Native Circuit Description (NCD) file, which is a physical representation of the design. The command executed to map a logic design to a physical design is listed below.

```
    map -o IIRThesis_cw_map.ncd -intstyle xflow -timing -ol
    high IIRThesis_cw.ngd IIRThesis_cw.pcf
```

Table 5.2 indicates the actual device utilization summary after the design is mapped to a Xilinx Virtex-II Pro xc2vp20 FPGA. Comparing Table 5.1 and Table 5.2, it is clear that both the total numbers of occupied slices and flip-flops have decreased slightly, due to the removal by MAP of useless components.

e. Placement, Routing and TRCE

Mapped NCD and PCF (Physical Constraints File) can be used as inputs for PAR to place and route the physical design on the target device. There are two considerations when performing the placement and routing: cost and timing. Cost-based PAR is performed based on the final cost calculated from several factors, which are assigned weighted values. Timing-driven PAR pays more attention to the timing constraints of the design. The most important resulting files after PAR include a placed and routed NCD file and a PAD file, which contains the final assignments of I/O pins on the FPGA. The PAR for the design of this project took a little more than 16 minutes. From the PAR report, the minimum absolute time slack is only 0.009ns. And Figure 5.3 illustrates the actual placement of this design on a Virtex-II pro FPGA. The command used to complete the tasks of placement and routing is listed below.

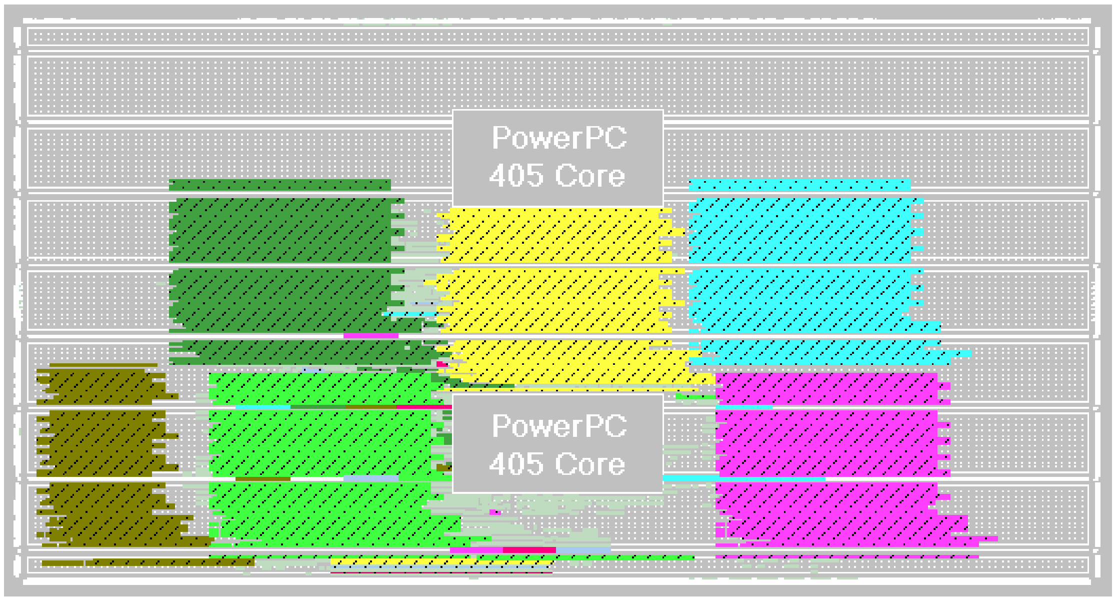


Figure 5.3 – Placement of Design on a Virtex-II Pro FPGA

Table 5.2 – Device Utilization Summary after Mapping

Device: xc2vp30ff896-7	Used	Total	Percentage
Occupied Slices	6646	13696	48%
Slice Flip Flops	12316	27392	44%
4-input LUTs	12436	27392	45%
as logic	12353	-	-
as a route-thru	58	-	-
as shift registers	25	-	-
Bonded IOBs	68	556	12%
Device: xc2vp30ff896-7	**Used**	**Total**	**Percentage**
Block RAMs	15	136	11%
GLKs	1	16	6%
PPC405s	0	2	0%
RPM macros	6	-	-
Total Gates for Design	1239553	-	-
Additional JTAG Gates	3264	-	-

```
par -w -ol std -intstyle xflow IIRThesis_cw_map.ncd
IIRThesis_cw.ncd IIRThesis_cw.pcf
```

Post-PAR Trace is also performed to analyze the routed NCD file. Its main purpose is to find out the paths with the worst slack time. A TRCE report was generated and revealed that the minimum time slack is 9.991ns and the maximum path delay is 6.076 ns for the actual layout of the design on an FPGA.

f. Generating BIT file

The last step for implementing a hardware design on an FPGA is generation of the bitstream file, which is used to configure a FPGA. Using the information provided by the fully elaborated NCD file, BitGen is called to create the binary file for FPGA configuration. The command used is as below.

```
bitgen -l -w -m -intstyle xflow iirthesis_cw.ncd
```

After the bit file is generated, the Sysgen model is ready to be deployed on the XUP board with a Xilinx Virtex-II Pro FPGA.

VI. Simulations and Results

In this chapter, both software simulation and hardware co-simulation with FPGA in the loop will be presented, and their results will also be discussed. Verification of appropriate operations is conducted before starting the simulation for the Sysgen model.

1. Software Simulation

In this section, “software simulation” means that the Sysgen model of a hardware design is simulated by a software simulator, which in this case is Simulink/Sysgen.

a. Verification of Sysgen Model

Before the Sysgen model can be recognized as a generic filter that can be configured to be an IIR or a FIR filter with taps less than 20, elaborated verification must be conducted.

The most practical approach to verify a user-created Sysgen model is comparing it to a well-established model that is already proven to be correct. Here for this project, the Simulink block Transfer_Fcn_Direct_Form_II was chosen to be the reference model. This block can implement a Direct Form II realization of a transfer function specified by the coefficients, which is pretty much the intention of this developed Sysgen model. By comparing the results from a model against those processed by the reference model in a simulation with identical conditions, verification for a Sysgen model can be accomplished. For this purpose, a test environment was constructed within Simulink, whose structure is illustrated in Figure 6.1. From the graphic, it is clear that the output signals from both systems are comparable in both time domain and frequency domain.

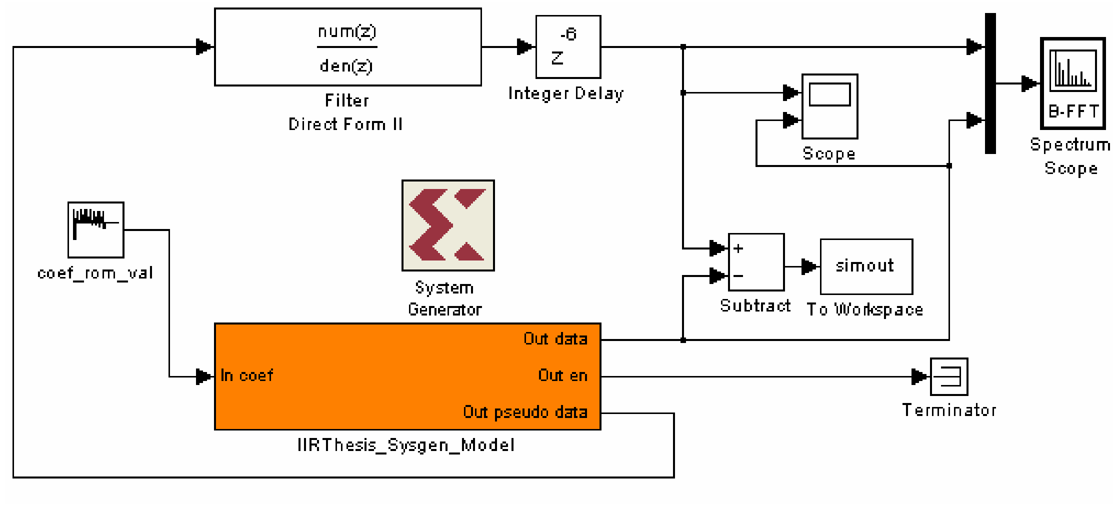


Figure 6.1 – Simulink Environment for Verification

In order to gain more general result, four classic filter designs were used to verify the Sysgen model, a 14th-order Butterworth high-pass filter with cutoff frequency of 0.5, an 18th-order Butterworth low-pass filter with cutoff frequency of 0.6, a 6th-order Chebyshev Band-pass filter with frequency range from 0.4 to 0.6 and a 10th-order elliptical band-stop filter with frequency range from 0.3 to 0.5. Table 6.1 shows the max difference and the mean of differences between the results from the theoretical model and the Hardware Model. The small differences between the theoretical result and the simulation result for all the four test filters indicate the Sysgen model can perform as a re-configurable IIR filter correctly. Figure 6.2 illustrated the spectrums of the output data from both model simulating four test filters. From the figures, two curves are almost overlapped, which means these two models produce almost identical frequency response to the test data.

a. Simulation and Result

Once the Sysgen model is verified, software simulation of the Sysgen model can be started by executing the MATLAB script `iir_main.m`. The detailed processing flow has already been discussed before. In this section, more attention will be put on the cost value for each candidate and the time it cost to simulate one candidate in

Table 6.1 – Max Difference and Mean of Differences

between Theoretical and Simulation Results.

	High-pass	**Low-pass**	**Band-pass**	**Band-stop**
Max Difference	2.4140×10^{-5}	2.4576×10^{-5}	8.1037×10^{-8}	4.5946×10^{-6}
Mean of Differences	6.6464×10^{-6}	6.5471×10^{-6}	2.2921×10^{-8}	1.3858×10^{-6}

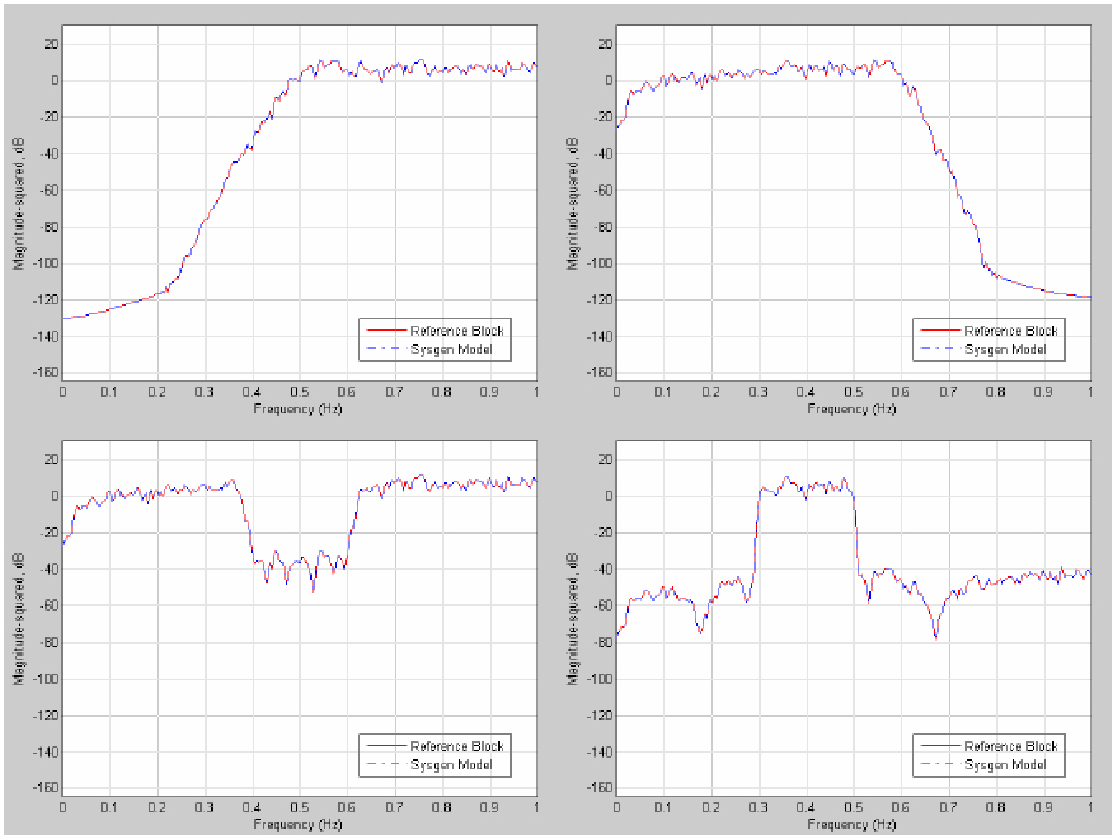


Figure 6.2 – Spectrum of Simulations Results from Theoretical and Sysgen Models

software. Every time a filter designed is simulated and its results are evaluated by the `iir_eva_fcn.m`, a MATLAB figure will be prompted up showing the magnitude response and group delay of previously simulated candidate. Figure 6.3 presents one of the prompted graphics. The blue line defines the lower boundary, while the line in yellow defines the upper one. If only the candidate's curve falls in between these boundaries, it gets no penalty, which means no cost value is accumulated. Any point outside that bounded area will cause an increase of the cost value by a weighted value. The final cost value will be sent to optimizer engine for generating a better generation of candidates.

The average time cost for a successful software simulation of the candidate is about 19.4714 seconds.

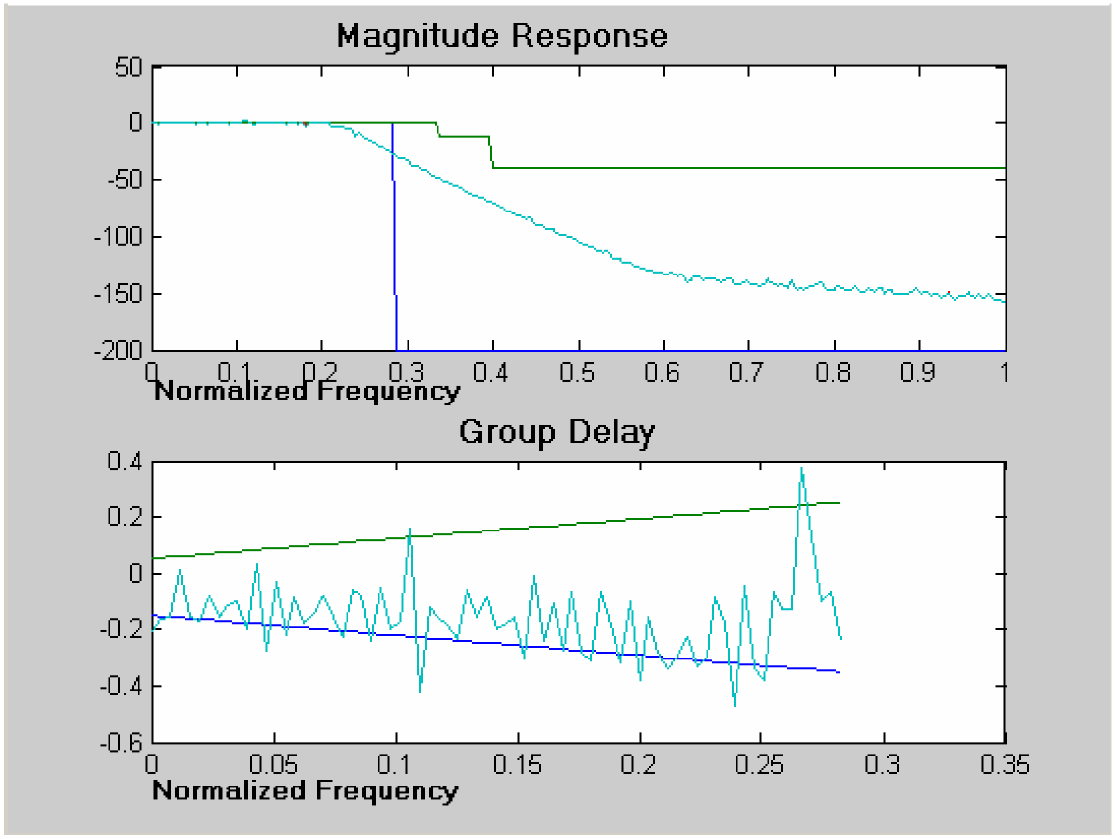


Figure 6.3 – Magnitude Response and Group Delay of a Filter Candidate

2. Hardware Co-Simulation with FPGA-in-the-Loop

a. Basic Introduction to HW Co-Simulation

After the hardware model's successful implementation in a FPGA, hardware co-simulation can be performed to accelerate the simulation process. In order to do so, the model must be compiled again using the "Hardware co-simulation" as compilation type. At this time, an extra interfacing circuit, which allows Sysgen to communicate with the implemented design using a physical interface between the computer and the hardware platform, was added to the original design. The new model did not have to be verified, because during the implementation, the verification has already been done. As soon as the bitstream file is produced, a new hardware co-simulation block is also created. By easily replacing the whole Sysgen model with this new block, it's now ready to perform the accelerated co-simulation with FPGA-in-the-loop. Figure 6.4 shows the how the model looks like after the replacement.

b. HW Co-Simulation Clocking

There are two modes for the System Generator to obtain synchronization with its associated FPGA hardware design, single-step mode and free-running mode. In single-step mode, instead of using the fast internal system clock, the hardware receives its clock signal from the software simulation. In other words, hardware activities inside the FPGA are all controlled by Simulink via the physical interface

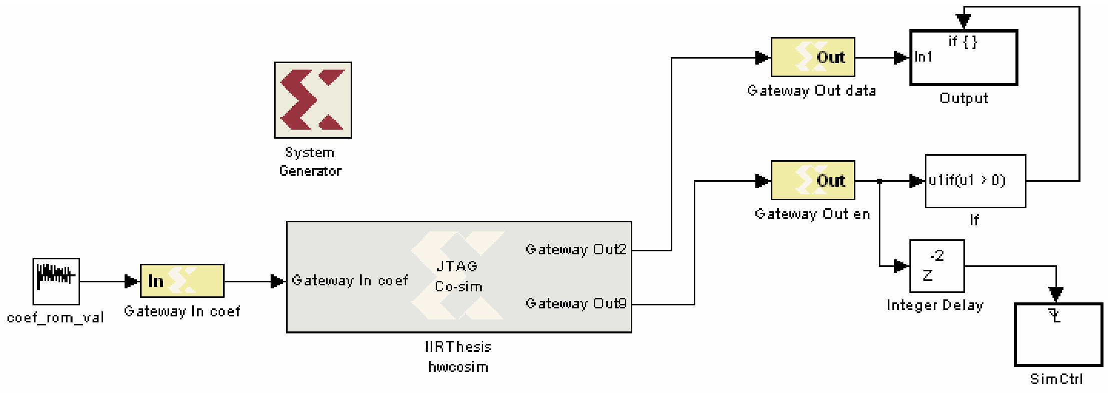


Figure 6.4 – Hardware Co-Simulation Block

between the computer and the hardware platform. Thus the hardware co-simulation operating in this mode is bit-true and cycle-true to the original design. On the other hand, the limitations from slow software simulating and communication latency deteriorate the performance achieved by the hardware. The hardware can still improve the simulation significantly as long as the limitations are negligible when compared to the great improvement achieved by hardware simulation. In free-running mode, the hardware co-simulation is running by internal clock signal, making the simulation no longer synchronized with the software simulation, but thousands times faster than that in single-step mode. The co-simulation in this mode is widely applied for streaming applications

By executing the `iir_main.m`, the simulations involved in the Sysgen model are not running by the software simulator any more. Instead, all the activities that occur in the Sysgen model are now performed by real hardware components, which are running at a very high speed. When operating in single-step mode, the hardware co-simulation took an average time of 25.0509 seconds for a successful run for one candidate. But it took only 0.9407 seconds, which is a significant improvement, to perform the same mission when the co-simulation block is configured to be clocked by internal CLK signal. Though the simulation of the hardware design is accelerated considerably, the problem left is that, the Simulink can no longer capture all data that is output from the FPGA, due to the asynchronous communications between the software simulator and the hardware platform. Unlike the BER (bit error rate) measurement system for encoder/decoder [], in which the software only cares for a few output data, the integrity of the whole data output from the hardware is vital to the evaluating accuracy of the filter candidate. If only the evaluator, which is so far implemented as a software program, could be migrated into the hardware design. That would be the best solution, because only one number, the cost value, is necessary to be fed back to the software for the simulation of one candidate.

VII. Conclusions

This work was originated from an innovative idea proposed by Dr. Buckner and his fellow researchers, who are developing a method for the design and optimization of large scale digital circuits using the combined power of an optimizer (Differential Evolution) and hardware-accelerated simulation. Following the basic concept, a simplified framework was established for optimizing the design of a target digital filter within the MATLAB environment. The framework includes two portions: software and hardware. The software part was developed quickly and is used to generate and optimize generations of candidates for a digital filter. Exploiting from the ability to be reconfigured on-the-fly, the implemented hardware design was employed to perform accelerated simulation for the candidate designs. The results from the hardware simulation in turn were used by the optimization engine to refine the next generation of candidates. Experimental results proved the framework was properly constructed and works as predicted. Though a part of its outcome is not perfect, this framework can be improved and extended for larger projects.

1. Summary

The first step of this project was developing a parameterizable generic digital IIR filter within System Generator. This generic filter has the following capabilities. First, it is reconfigurable, which means the model can read in coefficient data from the software, and reconfigure itself to the filter that is defined by the incoming coefficients. Second, no re-compiling is needed after the model is reconfigured to reflect a new candidate design. Last, the hardware implementation of this model can be used for hardware co-simulation with FPGA-in-the-loop.

Once the Sysgen model for the generic filter was finalized, behavior verification was conducted against a theoretical model, which was widely tested and recognized to be valid. In this work, a Simulink built-in model was selected as the reference model. By analyzing the results for both models from numerous simulations, the created model was verified. Supporting circuits were developed right after the verification of the filter model. These circuits are used to control the communications between the hardware and the software. In order to achieve consistency among simulations, a pseudo-random number generator was added to the hardware design. Its purpose is to provide identical test data for all the simulations.

After the hardware model was developed, the corresponding software programs were created within MATLAB. At the same time, a protocol for data exchanging between the hardware model and software programs was also established. Major functions include data frame constructor and candidate evaluator. These programs were demonstrated to allow auto-regenerating coefficients based on the feedback from the hardware without a human in the loop.

Then, the Sysgen model was implemented in a Xilinx Virtex-II pro FPGA. A pre-synthesis simulation was performed on the automatically generated HDL net-list to make sure the generated code was identical to the original design.

Once every thing was ready, hardware co-simulation with the FPGA-in-the-loop was performed. Co-simulation in single-step mode was tested on the XUP development system.

2. Future Work

Hardware co-simulation is too slow when running in single-step mode so the most important and highest priority task for future work is modifying the system to be capable of working in the free-running mode. A possible modification of the existing model would be to change the figure of merit for the optimization to a combination of power, delay and area consumption for the FPGA. Another possibility is adding an extra input port for the hardware model to allow the inputting of test data from external sources, such as an analog-to-digital converter. It would also be desirable to migrate all of the software modules to the hardware design including the evaluator or even the DE optimization engine. The ultimate goal is to create a System-on-a-Chip (SoC) that contains everything in this project which can accelerate the optimization of a circuit.

List of References

[1] M. Buckner, “A Method for the Design and Optimization of Digital Circuits Using Differential Evolution and Hardware Accelerated Simulation,” 2006.

[2] V. S. Lin, R. J. Speelman, C. I. Daniels, E. Grayver, and P. A. Dafesh, “Hardware Accelerated Simulation Tool (HAST),” *The Aerospace Corporation*, 2005.

[3] Mentor Graphics,
http://www.mentor.com/

[4] R. Quinnell, “Designing Digital Filters”, TechOnLine, 2006

[5] K. Nagappa and F. Harris, “On the Most Efficient M-Path Recursive Filter Structures and User Friendly Algorithms to Computer Their Coefficients,” *Software Defined Radio Technical Conference*, 2005

[6] V. Ingle and J. Proakis, *Digital Signal Processing Using Matlab*, BK&DK, 1997

[7] C. Chen, *Digital Signal Processing – Spectral Computation and Filter Design*, Oxford University Press, Inc., 2001

[8] M. Lutovac, D. Tosic and B. Evans, *Filter Design for Signal Using MATLAB and Mathematica*, Pearson Education, Inc., 2001

[9] R. Storn, *Advanced Topics In Computer Science Series - New ideas in optimization*, McGraw-Hill Ltd., UK, 1999

[10] R. Storn, “Differential Evolution Homepage,”
http://www.icsi.berkeley.edu/~storn/code.html

[11] The MathWorks
http://www.mathworks.com/

[12] The MathWorks, Inc., *Getting Started with MATLAB*, 2006

[13] The MathWorks, Inc., *Using Simulink*, 2006

[14] The MathWorks, Inc., *Simulink Reference*, 2006

[15] Xilinx, Inc., *User’s Guide for Xilinx System Generator for DSP*, 2006

[16] Xilinx, Inc., “Application Note: Hardware Acceleration of 3GPP Turbo Encoder/Decoder BER Measurements Using System Generator,” 2006

[17] S. Golomb, *Shift Register Sequences*, Holden-Day, Inc., 1967

[18] Xilinx, Inc., “Application Note: Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators,” 1996

[19] E. Boutillon, J. Danger and A.Ghazel, “Design of High Speed AWGN Communication Channel Emulator”, *Analog Integrated Circuits and Signal Processing*, 34, 133–142, 200

[20] Xilinx, Inc., *XUP Virtex-II Pro Development System Hardware Reference Manual*, 2005

[21] Xilinx, Inc., Xilinx Software Documentation

[22] Xilinx, Inc., Xilinx ISE 8.1i Manual

[23] B. Dhillon, “Optimization of DSSS Receivers Using Hardware-in-the-Loop Simulations,” The University of Tennessee, 2005

Vita

Getao Liang was born in Guangdong, China. He obtained his Bachelors of Science degree in Electrical Engineering from South China University of Technology in 2003. He started his graduate study in fall of 2004 at The University of Tennessee, Knoxville. He is expected to obtain his Master of Science Degree in Computer Engineering in spring of 2007