1-1-2008

Design and implementation of programmable pipelined FIR filter in FPGA

Ganendran Narasingavel
Ryerson University

DESIGN AND IMPLEMENTATION OF PROGRAMMABLE PIPELINED FIR FILTER IN FPGA

BY

Ganendran Narasingavel
BASc, University of Ottawa, Dec 2000

An engineering project presented to Ryerson University in partial fulfillment of the requirements for the degree of Master of Engineering in the program of Electrical and Computer Engineering

Toronto, Ontario, Canada, 2008

Author's Declaration

I hereby declare that I am the sole author of this project.

I authorize Ryerson University to lend this project to other institutions or individuals for the purpose of scholarly research.

Gaṇendran Narasingavel

I further authorize Ryerson University to produce project by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Gaṇendran Narasingavel

Borrower's Page

Ryerson University requires the signatures of all persons using or photocopying this report. Please sign below, and give address and date.

Name	Address	Date	Signature

Abstract

Since the introduction of DSP blocks in commercial FPGAs such as Altera Stratix II and Xilinx Virtex II, DSP applications are increasingly being implemented on FPGAs. This project implements a pipelined digital FIR filter with programmable coefficients in an Altera Cyclone II FPGA. An automated test system is also constructed to verify the design. The project places equal emphasis on implementing a programmable FIR as well as building an automated test system. Also, we will evaluate the practicality of the design by comparing the design to the FIR IP core provided by Altera.

Acknowledgement

I would like to express my appreciation to my project supervisor Dr. Andy .G Ye for his patience and helpful suggestions throughout the life of this project. His knowledge and support throughout this project has made this work possible.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ASIC	Application Specific Integrated Circuit
BFM	Bus Functional Model
DA	Distributed Arithmetic
FIFO	First In First Out
DSP	Digital Signal Processing
DUT	Design Under Test
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GLS	Gate Level Simulation
HDL	Hardware Description Language
IDE	Integrated Development Environment
IP	Intellectual Property
LC	Logic Cell
LUT	Look Up Table
PLI	Programmable Language Interface
PLL	Phase Locked Loop
MAC	Multiply Accumulate
RAM	Random Access Memory
RTL	Register Transfer Logic
SOC	System On Chip
SOPC	System On Programmable Chip
UART	Universal Asynchronous Receiver/Transmitter

Introduction

1.1 Motivation

Traditionally, real-time DSP applications have been performed using either DSP processors or custom application specific integrated circuits (ASICs). DSP chips have the advantage of low design costs and flexibility but lack performance when DSP algorithms need parallel execution [1]. In particular, a single DSP processor generally has limited number of multiply-accumulate (MAC) unit and require many clock cycles to compute each output value [2,3]. To remedy this ASICs are used which achieved excellent performance; however, the cost associated with ASICs are impractical for moderate-volume applications. Furthermore, it is impossible to reprogram an ASIC once it has been manufactured. The recent advancements in FPGA technology deliver programmability, higher performance and low development cost required for DSP applications. In particular, fully parallel, pipelined DSP architectures implemented in an FPGA can operate at very high data rates, making FPGAs ideal for high-speed DSP applications. Consequently, DSP applications are increasingly implemented in FPGAs.

This project focuses on Digital FIR filters due to its wide range of practical applications. FIR filters are the fundamental building blocks in a wide range of DSP applications, such as waveform shaping, anti-aliasing, band selection and low-pass filtering. On the other hand, a filter's characteristic is determined by the value of its filter coefficients. Therefore, it is desirable and beneficial to be able to selectively program the coefficients of an FIR filter. Such programmable FIR filters are necessary in systems that must support multiple protocols and standards. As a result, the FIR filter in our design was made to be programmable.

1.1 Objective

The main objective of this project is to implement a 32-tap pipelined programmable Finite Impulse Response (FIR) filter on an Altera Cyclone II FPGA. The second goal of this project is to build a hardware platform to verify this FIR filter in system level. Emphasis is placed on studying different FIR architectures, FPGA realization of the FIR filter and building a test platform for hardware emulation. The project has several other objectives in terms of educational aspects: Gain experience in System On Chip (SOC) design, Interface Design and Embedded System Design.

1.2 Report Structure

The rest of this report is divided into four sections. Section 2 presents an overview of existing FIR filter architectures. Section 3 presents the design process including system specification, system architecture, system component, implementation details and FPGA realization. Section 4 will present the verification strategy used in this project. Section 5 will evaluate the practicality of our design by comparing it to an Altera IP. Finally a conclusion and references will be provided.

SECTION 2: Literature Survey

This section examines the intrinsic parallelism of the FIR algorithms, analyze suitable architecture alternatives from the previous work, and make architectural decisions.

2.1 Theory

FIR filter is the fundamental building block in Digital Signal Processing and communication systems. As a result, the whole system's performance may very well depend on FIR filter's performance. An FIR filter's action depends on its tap length and coefficient values. The output of an L tap FIR filter depends on the previous L input samples. The basic operation of a FIR filter is convolution of the input sequence by the corresponding filter coefficients.

The following equation represents a FIR filter operation:

$$y[n] = \sum_{l=0}^{l-1} h(l)\, x(l-n) \qquad Equation\ (1)$$

where $x(l)$ input samples, $h(l)$ is the filter coefficient and L is the tap length.

2.2 Related Work

There are three main FIR architectures namely Direct, Transposed and Serial Distributed Arithmetic (DA). A detailed comparison between these three types of architectures is presented in [4]. Sequential architectures share a single MAC resource making it very area efficient at the expense of performance. Although DA type of implementation is suitable for high-performance application, they do not take advantage of the DSP blocks embedded in the FPGAs. This is because DA architectures uses sequence of table look-ups, additions, subtractions, and shifts of the input data these operations efficiently map into Look Up Tables (LUT) but not into the embedded DSP blocks. As a result, we will only discuss about the Direct and Transposed architectures going forward.

2.2.1 Architecture Exploration

Parallel hardware must be used when high sampling rate applications are involved. The two most popular parallel FIR architectures are the Direct and Direct Transposed forms. Even though Direct-Form architecture has low fan out of the input signal and a constant register width in the delay line they have a larger critical path due to the multi-operand adder. On the other hand, the Transposed-Form has a shorter critical path but the fan out on the input port grows linearly with the number of taps. In addition, Transposed-Form requires increased number of registers to account for the increasing width of the sum [1]. Figures (1) and (2) shows a direct and transposed structure of an FIR filter. Direct-Form is a direct representation of the equation (1). The input sequence is shifted through a delay line, multiplied with the corresponding coefficients and accumulated.

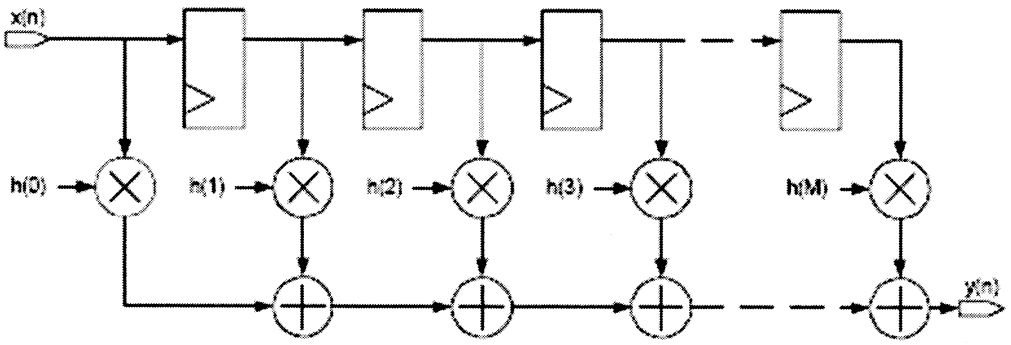


Figure 1: Direct form

Transposed-Form is obtained using the flow-graph-reversal theorem [1]. That is the Direct-Form is transposed by reversing the direction of all signal paths and interchanging the input with the output.

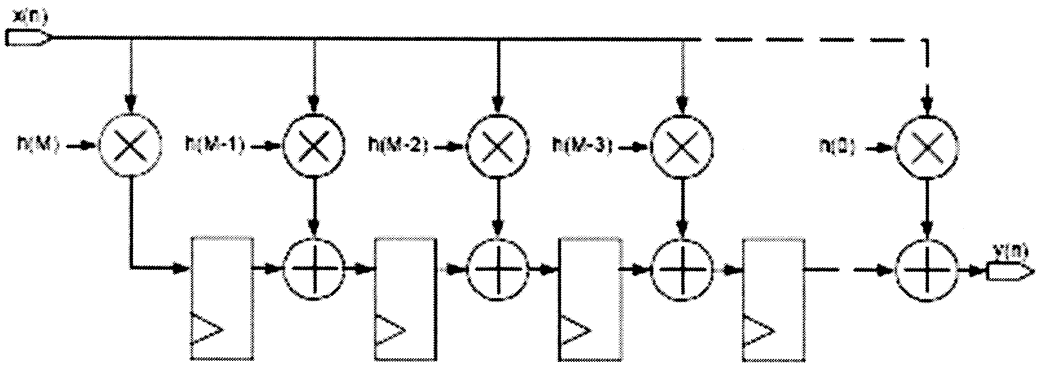


Figure 2: Transposed-Form

It is still arguable, which architecture achieves the highest performance, Direct-Form or Transposed-Form? According to [4], direct form architectures are suitable for area sensitive small filter orders while transposed structures are suitable for large filter orders. However, according to [1] replacing the multi-operand adder in the Direct Form with a pipelined binary adder tree will achieve similar performance as the Transposed-Form with fewer arithmetic resources. Therefore, a pipelined programmable FIR filter using Direct-Form and binary adder tree structure will be implemented in this project. One other point is that the binary adder tree is most efficient when the coefficients are in 2's power therefore a 32-tap filter is chosen for implementation. Figure 3 illustrates a 4th order pipelined FIR filter with binary adder tree.

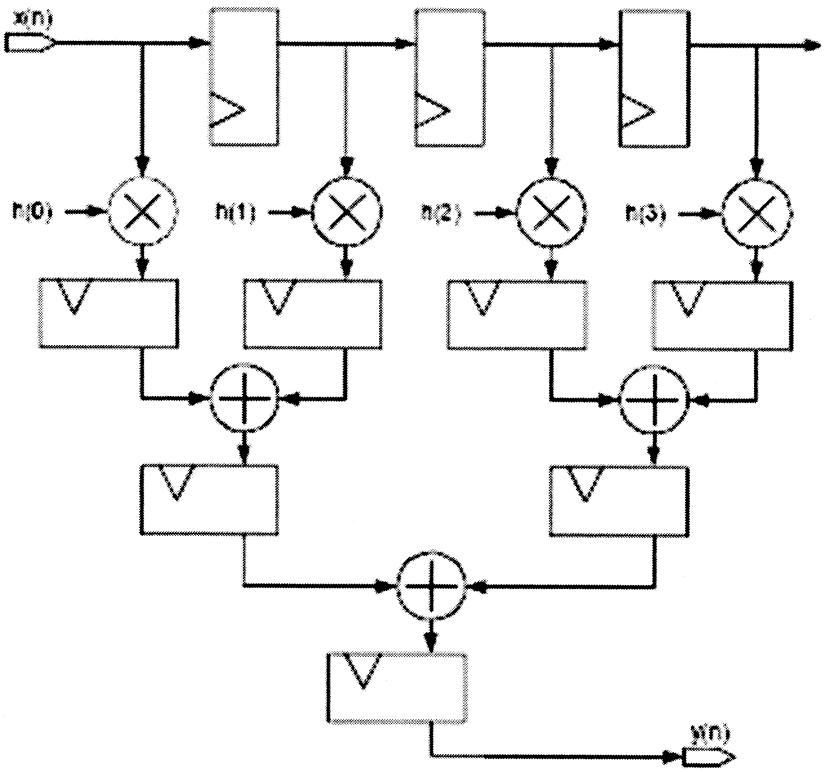


Figure 3: 4th order direct form pipelined FIR with binary adder tree

SECTION 3: Design & Implementation

3.1 System Overview

Our system is primarily a programmable pipelined FIR filter realized in an Altera Cyclone II FPGA (from here on this FIR filter module will be referenced as FILTER). This FILTER module is then integrated into an overall system as shown in Figure 4. The FILTER module communicates with the microprocessor for reloading its coefficients. The FILTER receives its input samples from an On-Chip cache memory. The FILTER module receives its clock from a Phase Locked Loop (PLL) module and communicates to the outside world through a Universal Asynchronous Receiver/Transmitter (UART). All inter module communications were handled by the Altera's Avalon Bus.

3.1.1 Design Specification of the Overall System

First, processor gives the following inputs to the FIR module: a) starting address of the input samples b) total byte count of the input samples. Next the processor issues a START execution command and then polls for the first output to be ready. As soon as the START is issued the FIR begin reading the input samples from the On-Chip RAM and calculates the results. As soon as the first result becomes valid, the processor begins to read the results at its own speed. Once all the input data have been processed the FIR informs the processor by issuing a DONE execution command. The FIR cannot be reconfigured while it is in the middle of processing a batch of input samples, except through a system reset. In case the FIFO becomes full, the FIR core halts its execution and stop reading in any more new input.

The overall system should satisfy the following design requirements:

1. At every clock edge the system should be able to feed in a new input sample and produces a new output sample after the latency period.
2. A data hand-off unit should be available to hand-off the data from the FIR's clock domain to the processor's clock domain.
3. A master interface should be available to retrieve the samples from the memory location specified by the processor.
4. A slave interface should be available to receive the coefficients from the processor and to transfer the results back to the processor.

5. FIR should be able temperately halt its operation as soon as the FIFO overflow. Moreover, the FIR should be able to resume its operation from where it left off. (I.e. the pipeline should resume without any additional latency).

3.2 System Architecture

The overall system implemented for testing the FIR differs from most of the conventional designs. In this project the test platform was implemented as a System on Chip (SOC) for design reuse purposes. For example, this test platform can be used to verify another custom design by replacing the FILTER unit with the new design. Of course the new design should have the interface to communicate with the Avalon Bus.

The overall test platform is pictured in Figure 4. As we can see, the target SOC is composed of a processor (NIOS II), a memory, a clock generator (PLL), a communication medium (Avalon Bus) and our FILTER unit. In this project the FILTER is composed of a FIR, a master interface, a slave interface and a FIFO. The master interface can initiate data transfers from the on chip memory. The slave interface can respond to the bus read/write within one clock period. Each of these components will be explained in detail in the following section.

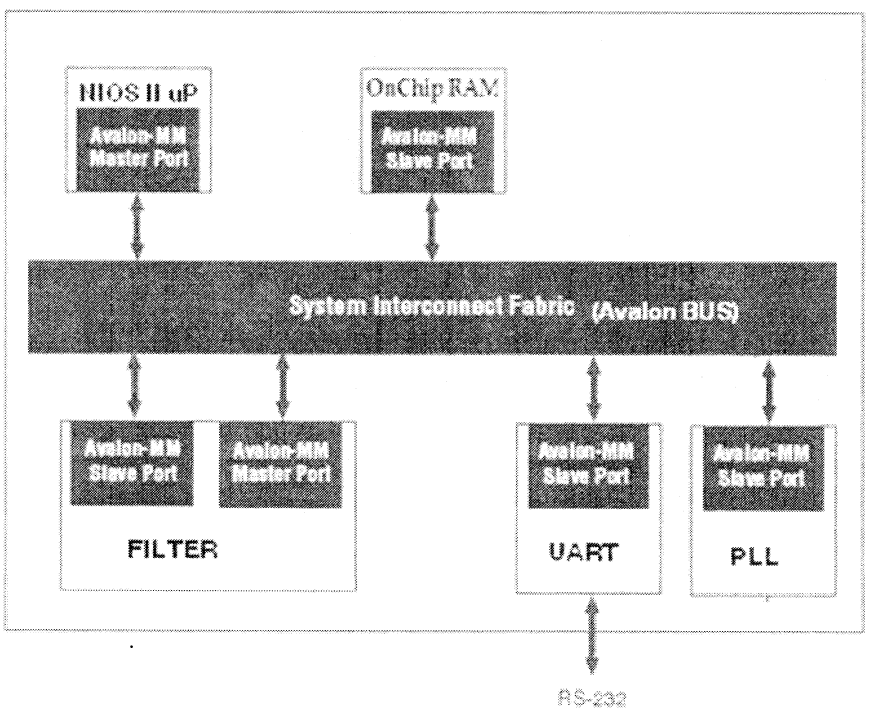


Figure 4: Overall System Architecture

3.2.1 Component Description

FILTER

This component will handle the FIR calculations of the input samples. This is a custom-logic mainly composed of 2 modules: FIR calculation core and FIFO. The details are depicted in Figure 7.

The FIR core is implemented as shown in Figure 3, utilizing a binary adder tree. Since the FIR core is running on a faster clock than the rest of the design, data transfer between modules need to be properly synchronized. Consequently, a simple data-hand off mechanism is implemented using a FIFO.

The data-hand of mechanism functions as follows: As soon as a result is ready the FIR will hand over the result to the FIFO and keep on calculating the next result without worrying if the result has been passed to the processor. The FIFO will hold the result until the processor is ready to read. If the FIFO becomes full the FIFO will inform the master interface to stop reading anymore new samples from the memory. If the processor try to read from and empty FIFO the FIFO will instruct the slave interface to issue a wait request to the processor.

Processor

The component will handle the following two main tasks: (a) Loading the filter coefficients, (b) Controlling the test-automation.

Altera's NIOS II/s soft-core processor with 16kb-instruction cache was used in our design. NIOS II processor is a general-purpose RISC processor core. Refer to [7] for more details.

On-Chip RAM

This component will store and deliver the input samples to the FILTER. A 20kb, 32-bit width single I/O port RAM block was used in our design.

PLL

This component will be used to build the clock for the FIR core. The FIR core clock is currently set to 2 times of the master clock.

UART

This component will handle the communication between the SOC and the outside world. In particular, to send calculated FIR results to the host computer.

Avalon Bus

This component will be the system bus which connects all the components described above.

3.3 Design Constraints and Choices

This section states the assumptions and the choices we made in building the FILTER and the SOC platform.

3.3.1 Polling vs. Interrupts

In order to communicate the completion of the FILTER calculations to the Processor, polling was implemented in our design as opposed to the preferred Interrupts technique. Polling was chosen for its simplicity.

3.3.2 Software Reset

Once the START command is given the FILTER cannot be stopped, unless the FILTER has finished processing the entire samples. The only way to get around this is by doing a hard reset.

3.3.3 Pipeline Flushing

This design does not flush out the pipeline at the end because the pipeline was not flushed a system reset is required before processing the next batch of data in order to clear all the pipeline registers.

3.4 Design Details

The design of the overall system consists of hardware design, software design and hardware software interface design.

3.4.1 Hardware Design

For each functional block illustrated in Figure 7, a behavioral model was developed in Verilog. In our Verilog model the data and coefficient widths were parameterized for future study purpose which will be later discussed in section 5. An automated test-bench was developed for each module and verified before integration. All modules were synthesized using Altera's Quartus II.

3.4.1.1 FIR Filter Design

A behavioral model of the FIR was implemented in Verilog utilizing the FIR architecture chosen from the literature survey (Figure 3). A block diagram of the FIR with the inputs and outputs are shown in Figure 5.

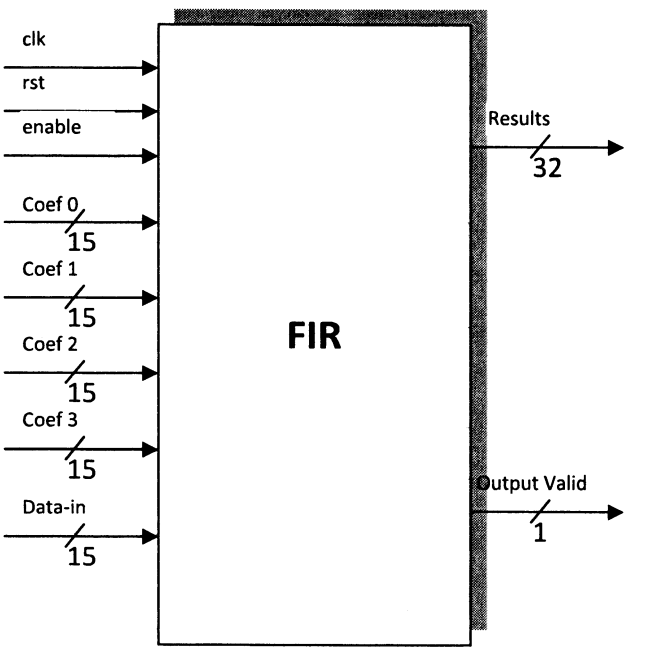


Figure 5: Block Diagram of the FIR

The task of calculating FIR was partitioned into four main modules as follows:

1) Shift Register
2) Multiplier
3) Adder Tree
4) pipeline controller

Only the multiplier circuitry was coded into a separate module. The pipeline controller circuitry was implemented using a counter circuitry which was initialized with the latency value and counted down to zero. The other two categories are straight forward and need no further explanation.

3.4.1.2 FIFO Design

A behavioral HDL was developed for a synchronous FIFO. The data width and the depth of the FIFO were parameterized. A block diagram detailing the input and the outputs of the FIFO is given in Figure 6.

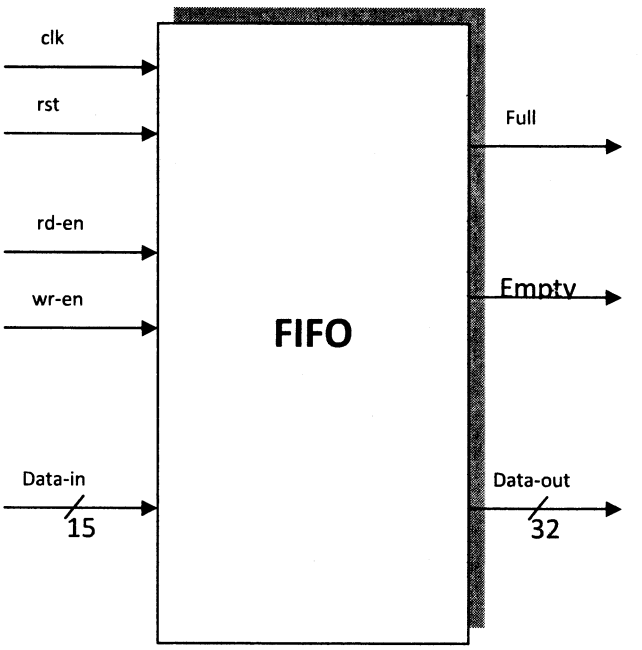


Figure 6: Block Diagram of FIFO

3.4.1.3 Interface Design

The interface has tow functionalities: a) It provides a mechanism to transfer data between system bus and the FIR b) It controls the functionality of the FIR. An Avalon-MM Master Interface and an Avalon-MM slave interface were used in this design. The behavioral model of the master and slave interface was adapted from [8] and modified to suit the needs of our project. These modifications will be detailed shortly. First, for the completeness of this report the characteristics and the signals of the Avalon-MM master and slave interface are reproduced here from reference [8].

Master Interface:
The Custom FIR's Avalon-MM Master port has the following characteristics:

- It is synchronous to the Avalon-MM master clock interface.
- It initiates master transfers to the system interconnect fabric.

Signal Name in HDL	Avalon-MM Signal Type	Width	Dir	Notes
Avm_m1_address	Address	32	Out	Byte Address aligned on word boundary.
~~Avm_m1_byte_enable~~		4	Out	Enables specific byte lanes on ports greater than 8 bits
avm_m1_read_n	Read_n	1	Out	Active low read request signal
Avm_m1_readdata	Readdata	32	In	Uni-directional data
Avm_m1_waitrequest	Waitrequest	1	In	Forces master port to wait until the system interconnect fabric is ready to proceed with the transfer
Addr_reg	Slave_port signal	32	In	Deliver the starting address of the data samples
Len_reg	Slave_port signal	16	In	Deliver the byte count of the total sample to process.
Go	Slave_port signal	1	In	Initiate the read transfer
Read_busy	Slave_port signal	1	Out	Avoid accepting new jobs till finish processing
FIFO_FULL	Signal from FIFO	1	In	Force the master to stop reading in any new samples. This will force the data_ready to low as well. Which will halt the FIR
Data_in_ready	FIR module signal	1	Out	Enables the FIR module
Data_in	FIR module signal	32	OUT	Input Sample for FIR calculation

Table 1: Avalon-MM master port Input/Output signals [8]

Modifications to the Master Interface:

In Table 1 the omission and the addition of the interface signals are indicated by strikethrough font and the underlined fonts. Particularly, the master interface was modified to handle FIFO overflow. This requirement translates in to the following two tasks:

1. Master port should stop reading new samples when the FIFO is full.
2. Master port should stop the execution of the FIR when the FIFO is full.

Slave Interface

The slave port handles the simple read and writes transfers to the slave interface registers. The register map is shown in Table 3. The slave port has the following characteristics:

- Synchronous to the Avalon-MM clock interface
- Readable and writeable.
- Zero wait states for writing and one wait state for reading.
- No setup or hold restrictions for reading and writing.
- Uses native address alignment, because the slave port is connected to registers rather than a memory device.

Signal Name in HDL	Avalon-MM Signal Type	Width	Dir	Notes
Avs_sl_address	Address	6	in	Byte Address aligned on word boundary.
Avs_sl_read_n	Read_n	1	In	Read request input
Avs_sl_write_n	Write_n	1	In	Write request input
Avs_sl_chipselect_n	Chipselect	1	In	Chip-select to slave port. Slave port ignores all other signals unless it is selected
Avs_sl_readdata	Readdata	32	In	Uni-directional read data
Avs_sl-writedata	Writedata	32	In	Uni-directional write data
Avs_sl_waiterequest	Waiterequest	1	Out	Forces the system interconnect to wait reading from the FIR module.
FIFO_Empty	Signal from FIFO	1	In	Force the slave port to generate a waitrequest.

Table 2: Avalon-MM slave port Input/Output signals

Modification to the Slave Interface:

The following modifications are done to the slave interface:

1. Registers were included to store the reloaded coefficient values.
2. The address width is increased to be able to handle the increased register space.
3. An additional waitrequest and FIFO_Empty signals were added to handle the FIFO underflow.

3.4.1.4 Top-level Design

The top-level instantiates the FIR, FIFO, Avalon-MM master and slave interfaces. Figure 7 depicts the overall block level diagram of the FILTER. The FIFO module acts as the data hand-off module. The data-handoff module has an overflow/underflow prevention mechanism by communication with both master and slave interfaces. It can be seen from Figure 7 that the FIFO informs the emptiness to the slave interface and its fullness to the master interface. The FIFO module in our design has a depth of eight. However, the behavioral code of the FIFO is parameterized thus it can be set to any desired depth.

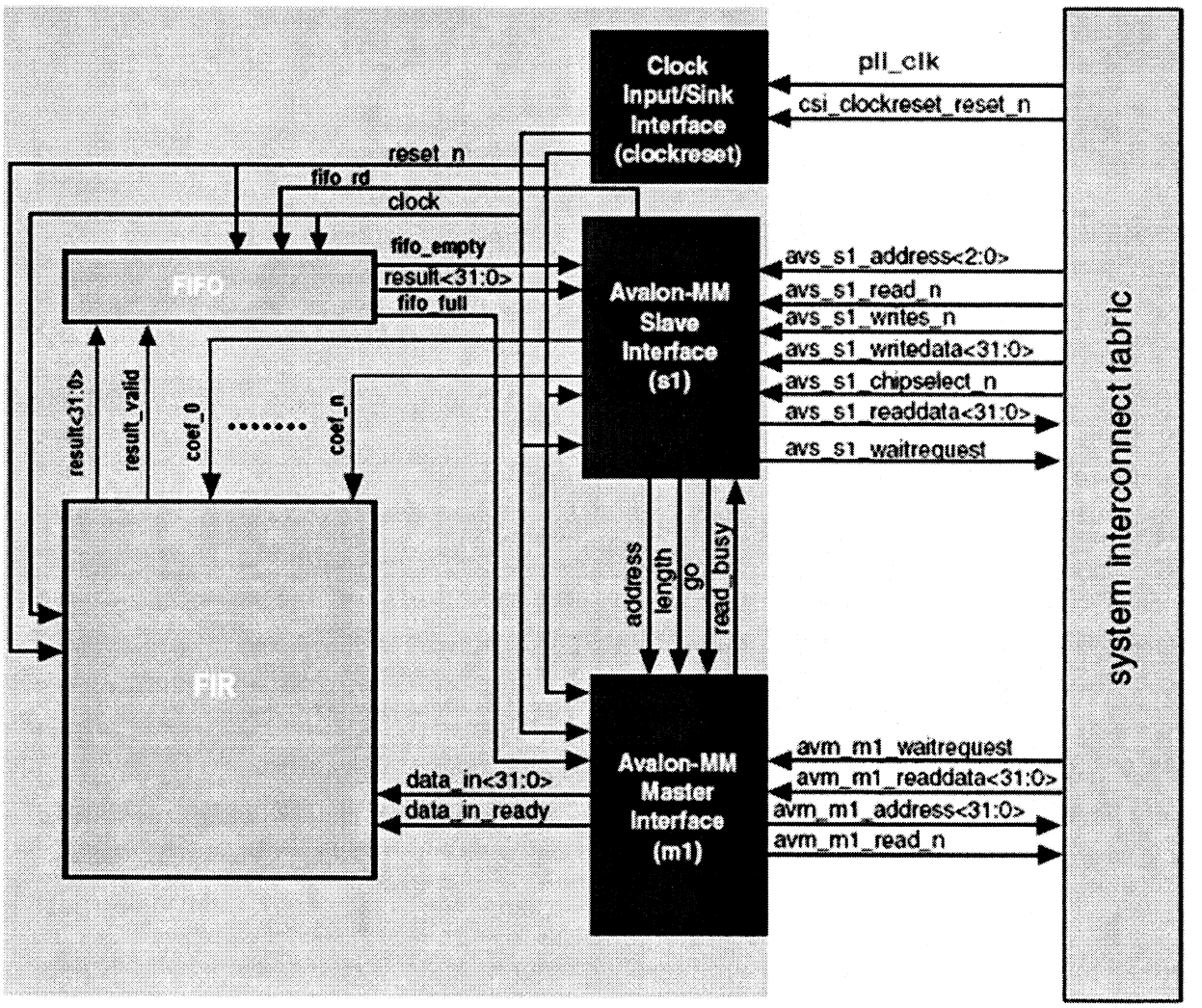


Figure 7: Top level block diagram

3.4.2 Software Design

The operation of the FILTER is controlled by the NIOS II processor. To facilitate this, a set of hardware registers were included in the slave interface. It is through these registers the hardware and the software communicates. The software configures the FILTER by writing values into its configuration registers. The software obtains information from the FILTER by reading the status and result registers.

The layout of the register map is shown in Table 2. In order for the software to understand the variables stored in the hardware registers NIOS II processor include two macros IORD and IOWR. Using these two basic macros another set of more readable device access macros were defined according to the register layout shown in Table 4. (See appendix C, fir_reg.h).

3.4.2.1 Address Mapping

The table below shows the address mapping of the Filter Interface

Register Name	Offset	Access	Description
Address	0x00	RW	32-Bit start address of the input samples
Length	0x04 +04	RW	16-bit byte count of the total data sample
Control	0x08 +08	RW	Bit [0] is the GO bit which instructs the FIR to begin execution. Bit [7:1] is reserved.
Coefficient 0	0xOC1 +12	RW	16-bit coefficient
Coefficient 1	0x10 +16	RW	16-bit coefficient
Coefficient 2	0x18 +20	RW	16-bit coefficient
Coefficient 3	0x1C +24	RW	16-bit coefficient
Reserved	0x20-38	--------	Reserved for future enhancements. E.g. more taps
Result	0x3C +60	RO	32 –bit result of the FIR calculations
Status	0x40 +64	RO	Bit [0] is the BUSY bit to indicate that the FIR still haven't finish processing the batch of data specified in the "length register". Bit [1] is the DONE bit to indicate that the FIR finish processing and ready for new batch of data. Bit [2] is the OUT_VALID bit indication the processor when the output becomes valid after latency period.

Table 3: Address Map

3.4.2.2 Register Layout

The table below shows the layout of the registers.

Offset	31 16	15 2	1	0
0x00	Address			
0x04	Reserved	Length		
0x08	Reserved			GO
0x0C	Reserved	Coefficient 0		
0x10	Reserved	Coefficient 1		
0x18	Reserved	Coefficient 2		
0x1C	Reserved	Coefficient 3		
0x3c	Result			
0x40	Reserved		OUT_VALID	DONE	BUSY

Table 4: Register Layout

3.4.3 System Integration Details

To simplify the process of attaching Altera IPs to a system interconnect, Altera provides a tool named SOPC Builder, which takes care of the bus interface signals, bus protocol, as well as any other interface issues. The crucial point to note here is that the SOPC Builder cannot automatically generate the interface logic for custom-cores, unless they are built using the standard Avalon Bus signals. This detail was already taken care of during the design phase by naming the signals with the appropriate interface name as well as the correct signal type (refer to Table 1). For example, in the signal name "*Avs_s1_address*" s1 indicates that it is a slave interface signal and the word "*address*" indicates that the signal is of type address. Once the custom core was made SOPC compatible it was instantiated into the SOC System as shown in Figure 4.

Once all the necessary connections were made in the SOPC builder GUI, a Verilog HDL files defining the overall system was generated. This HDL file was then simulated in ModelSim to verify the correctness of the design. The next section describes our verification strategy in detail.

SECTION 4: Verification

The process of verifying the design consumed more time than the design itself. This section describes our verification approach and the test environment used in the verification process. The objective of this section is not only to test out design, but also to automate the testing process.

4.1 Module-Level Verification

Verifying the design started with the module level simulation. First, self-checking module-level test benches were designed and simulated to exhaustively exercise the features in that module. Once the entire module tests were completed, modules were integrated and verified.

4.1.1 RTL Simulation

RTL simulation was done using the ModelSim student version software. First, unit simulation was carried out; then adjacent units were integrated and simulated. Strictly speaking, each individual module and the top-level module were simulated with a self-checking testbench. The necessary reference module was also developed either in Verilog or C++ to facilitate self checking. As an example the self checking process implemented to simulate the FIFO is illustrated in Figure 8.

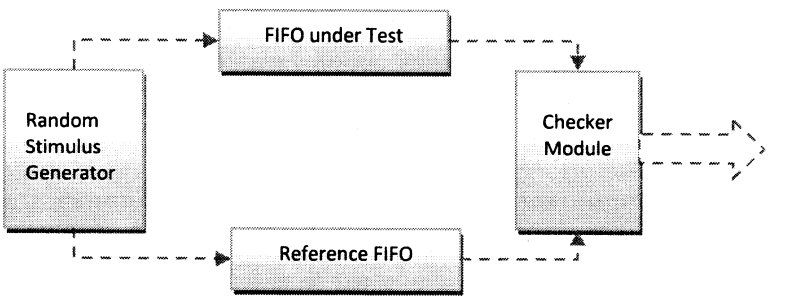


Figure 8: Self-Checking architecture

4.2 Top-Level Verification

Once all the filter components were integrated the top-level was simulated. Figure 9 illustrates the block diagram of the testbench used for top-level simulation. The testbench consists of a Bus Functional Model (BFM) to simulate the READ and WRITE task of the microprocessor, a Verilog ROM-Model to provide the input samples and a PLI routine to generate the expected output results. The intended checker unit was not completed. The design details of the BFM and Programming Language Interface (PLI) routines are described in the following subsections.

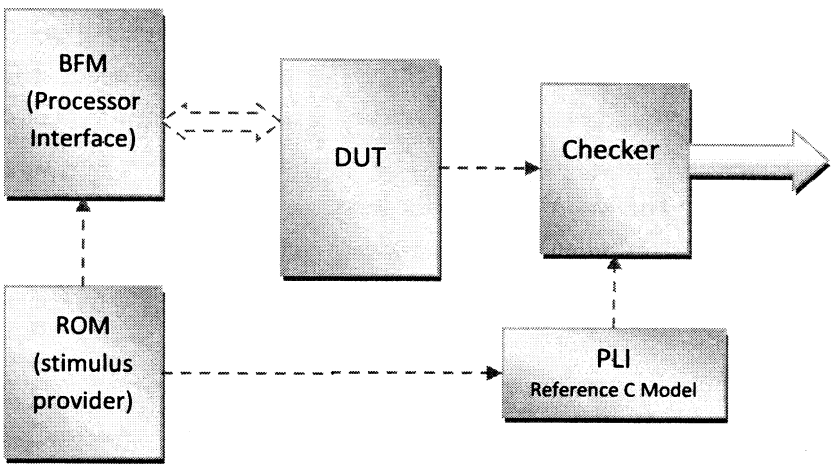


Figure 9: Top-Level TestBench Block Diagram

4.2.1 Bus Functional Model Design

BFM is a simplified model that reflects only the I/O behavior of a device (in our case a processor) without modeling its internal details. Consequently, a BMF is designed to imitate the READ and WRITE operation of the NIOS II microprocessor. This BFM was then used to verify the interaction between the FILTER and the microprocessor. Figure 10 shows a block diagram of the BMF designed for this simulation. Again this BFM is designed only to co-op with the Fundamental Slave READ/WRIT transfers of the Avalon Bus.

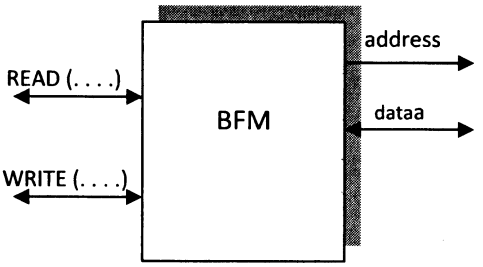


Figure 10: Block Diagram of BFM

4.2.2 Programmable Language Interface design

Simply said, PLI is a mechanism to invoke a C-language function from a Verilog simulator. To verify the top-level a reference model of the FIR was implemented in C (See appendix C, calculate_fir.c). Now the task is to make the Verilog simulator understand the C model and properly execute it using Verilog simulator. Once it has been done the C function becomes a PLI routine. In order for the simulator to understand the existence of the PLI routine the compiled PLI routine was linked with the existing binary of the simulator. Further details can be found in [10].

4.2.3 Gate-Level timing Simulation

The RTL simulation can only verify the functionality of the design. RTL simulation doesn't have any concept of timing. In order to verify if the timing requirements are met Gate-Level Simulation (GLS) was performed.

After synthesis, a gate-level net-list was generated using Quartus II. This gate-level net-list contains information about the incurred delay through each cell in a net-list. This information is then used to verify if the design met the worst case timing.

4.3 System Level Verification

The whole system was simulated to verify the data paths only. The entire system was emulated in hardware because the simulation was extremely slow. NIOS II IDE and DE2 board was used for hardware emulation. A test program was written in C-language to facilitate the hardware emulation. The following sub sections describe these tow process in detail.

4.3.1 Hardware Emulation

Emulation was an important part of the overall verification because it enhances test coverage far beyond what was possible with RTL simulation alone. There were two main motivations for doing emulation. First, emulation helps to identify the corner cases missed in simulation. Second, emulation was much faster than the RTL simulation, providing the twofold benefit of shorter run time and increased coverage.

4.3.1.1 Emulation setup (Test Environment)

The overall SOC (Figure 4) was synthesized on to a Cyclone II FPGA. The Design Under Test (DUT) was exercised through the NIOS II IDE running on a host-PC. Figure 11 below illustrates our Emulation environment.

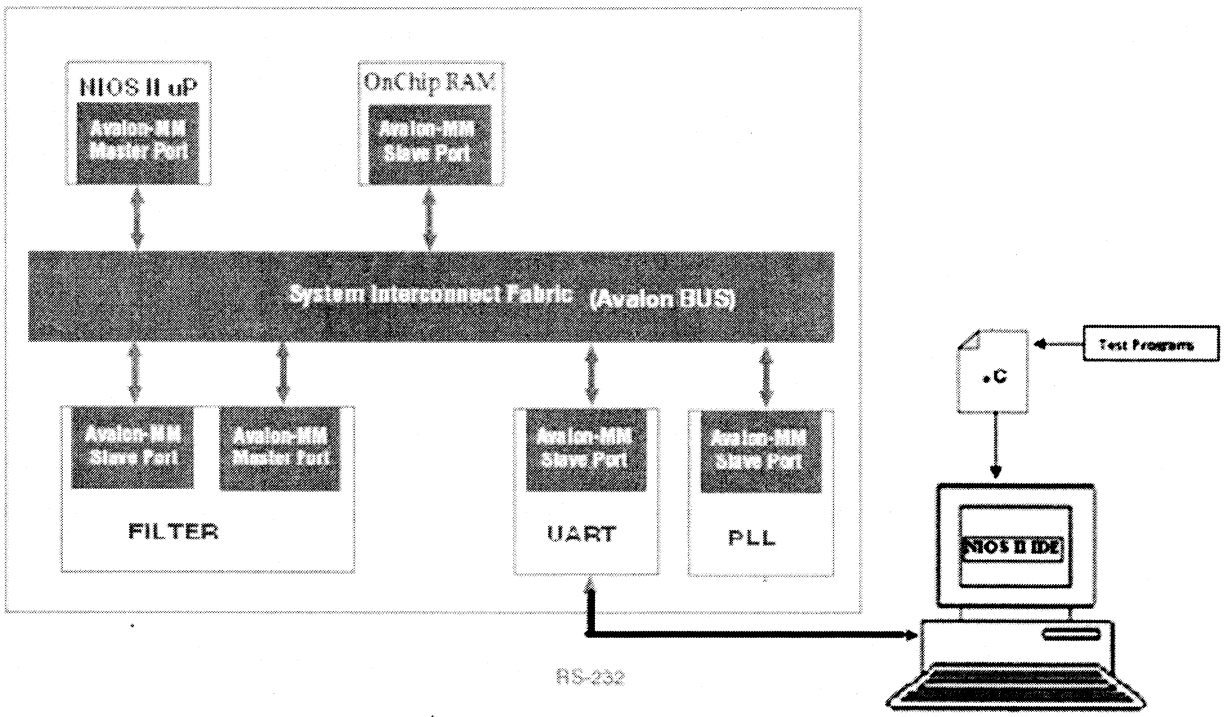


Figure 11: Emulation Setup

For emulation a test program was written in C-language to exercise the DUT. The NIOS II IDE was used for compiling the test program for the NIOS II processor's instruction set. The compiled program was downloaded and executed in the NIOS II processor. The output of the DUT was viewed using the NIOS II IDE.

The IOWR (I/O write) and IORD (I/O read) macros were used to access the registers in the slave interface. At compile time these macros expand to the appropriate assembly instruction to access the hardware registers in the device. Each of these macros uses a base address to identify the component and offset to identify the appropriate register. Once the software finished configuring the device registers (using IOWR) the DUT start its execution and write the results into the result register. Finally the software read these results (using IORD) from the result register and displays it in the NIOS II IDE. These results were then manually varied by comparing it with the outputs of the reference model.

SECTION 5: Design Assessment

This section evaluates the merits of our design by comparing it to an Altera IP core. A reference design was created with similar constraints for this analysis.

5.1 Reference Design Realization

Altera's FIR Compiler v7.2 was used to parameterize and generate a Verilog HDL of the reference design. A detail description of the FIR Compiler can be found in [12]. In order for a fair comparison between the designs the following properties of the Altera IP was set to reflect our custom design (FILTER):

- Coefficient Bit Width <15>
- Input Bit Width <15>
- Output Bit Width <15>: *Three least significant bits were truncated.*
- Pipeline Level <1>
- Coefficient Reload <yes>
- Clock to compute <1>: i.e. *an input data is processed every 1 clock period and a new output data is generated every clock period.*

Once the Verilog code was obtained from the FIR compiler the code was synthesized, analyzed and compiled. From the Analysis & Synthesis report following matrices was collected: maximum clock speed, estimated total logic elements, total registers and the total number of LUTs. These results are summarized in Table 5 below.

Design	Total LC	LUTs Only LC	Register Only LC	DSP Block Usage (18x18)	*f*Max (MHz)
Altera IP	3,549	1626	621	32	125
Custom Logic	3,666	332	1360	32	150

Table 5: Summary of the Results

Performance Increase relative to the Altera IP $= \frac{(150-125)}{125} \times 100 = 20\%$

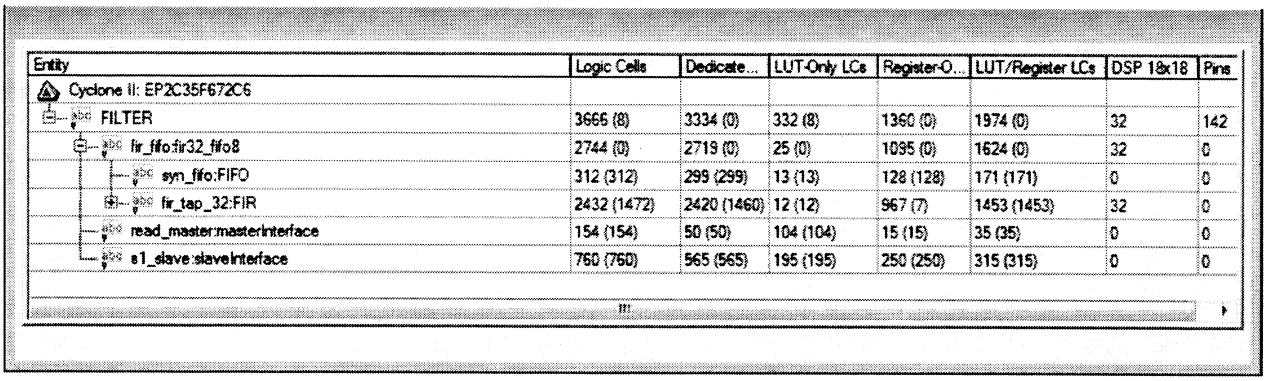
Resource Usage Increase relative to the Altera IP $= \frac{(3666-3549)}{3549} \times 100 = 3\%$

As we can see from Table 5, our design can be synthesized with a 150 MHz clock while Altera's IP can only constrained for a maximum of 125 MHz clock. This gives our design a 20% performance increases.

On the other hand, our design uses slightly larger resources than the Altera IP. Both designs use the same number of dedicated multipliers. However, our design uses 117 more logic cells than the Altera's IP. The following section gives a detail break down of the resource consumption of both designs. Then tries to reason why our design consumes more recourse compare to the Altera IP.

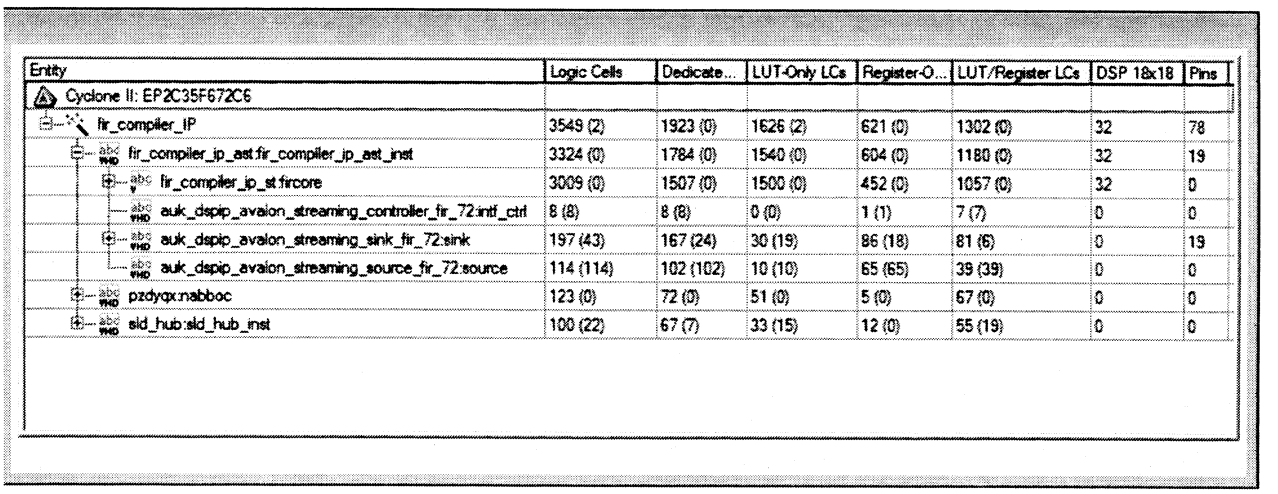
5.2 Comparison of Results

The tables 6 and 7 give a detail breakdown of the resource usage.



Entity	Logic Cells	Dedicate...	LUT-Only LCs	Register-O...	LUT/Register LCs	DSP 18x18	Pins
Cyclone II: EP2C35F672C6							
FILTER	3666 (8)	3334 (0)	332 (8)	1360 (0)	1974 (0)	32	142
fir_fifo:fir32_fifo8	2744 (0)	2719 (0)	25 (0)	1095 (0)	1624 (0)	32	0
syn_fifo:FIFO	312 (312)	299 (299)	13 (13)	128 (128)	171 (171)	0	0
fir_tap_32:FIR	2432 (1472)	2420 (1460)	12 (12)	967 (7)	1453 (1453)	32	0
read_master:masterInterface	154 (154)	50 (50)	104 (104)	15 (15)	35 (35)	0	0
s1_slave:slaveInterface	760 (760)	565 (565)	195 (195)	250 (250)	315 (315)	0	0

Table 6: Resource usage break-down of the Custom Logic



Entity	Logic Cells	Dedicate...	LUT-Only LCs	Register-O...	LUT/Register LCs	DSP 18x18	Pins
Cyclone II: EP2C35F672C6							
fir_compiler_IP	3549 (2)	1923 (0)	1626 (2)	621 (0)	1302 (0)	32	78
fir_compiler_ip_ast:fir_compiler_ip_ast_inst	3324 (0)	1784 (0)	1540 (0)	604 (0)	1180 (0)	32	19
fir_compiler_ip_st:fircore	3009 (0)	1507 (0)	1500 (0)	452 (0)	1057 (0)	32	0
auk_dspip_avalon_streaming_controller_fir_72:intf_ctrl	8 (8)	8 (8)	0 (0)	1 (1)	7 (7)	0	0
auk_dspip_avalon_streaming_sink_fir_72:sink	197 (43)	167 (24)	30 (19)	86 (18)	81 (6)	0	19
auk_dspip_avalon_streaming_source_fir_72:source	114 (114)	102 (102)	10 (10)	65 (65)	39 (39)	0	0
pzdyqx:nabboc	123 (0)	72 (0)	51 (0)	5 (0)	67 (0)	0	0
sld_hub:sld_hub_inst	100 (22)	67 (7)	33 (15)	12 (0)	55 (19)	0	0

Table 7: Resource usage break-down of the IP Core

As it can be seen from the detail break down of Tables 6 and 7, the custom logic consumes slightly large number of logic cells (11% of the total) than the Altera IP core (10% of the total).

The crucial difference is that the custom logic uses a large number of dedicated registers (43% -increases) than the IP core. This is due to the fact that in our design the coefficients coming in from the NIOS II processor were registered in the slave interface (Figure 7 & Table 4). The IP core however reload the coefficients directly form a memory.

Another observation is that both our FIR calculation core and Altera's IP used 32, 18x18 dedicated multipliers. However, our FILTER core used only 2432 logic cell while Altera's FIR core used 3009 logic cells.

The other main different between these two designs are the custom logic uses the Avalon memory mapped interface and the IP core uses the Avalon streaming interface.

5.3 Future Work

In our present work, if we wanted to cascade multiple FIR modules an additional adder tree should be implemented as shown in Figure 12. Moreover, a pipeline controller (not shown in the figure) will also be required. The presence of the adder tree and the pipeline controller forbids the design from scaling as we desire. Remedying this will be our future work.

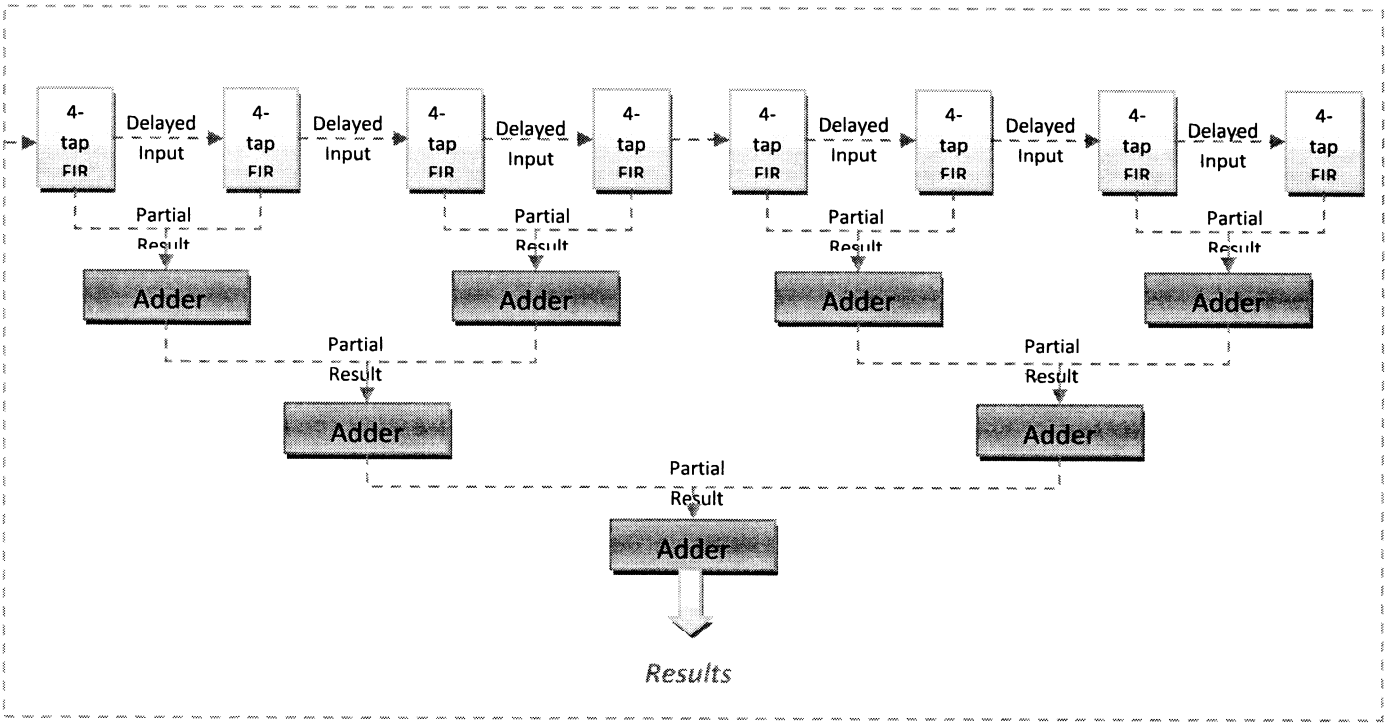


Figure 12: Example of cascading multiple FIR modules

Conclusion

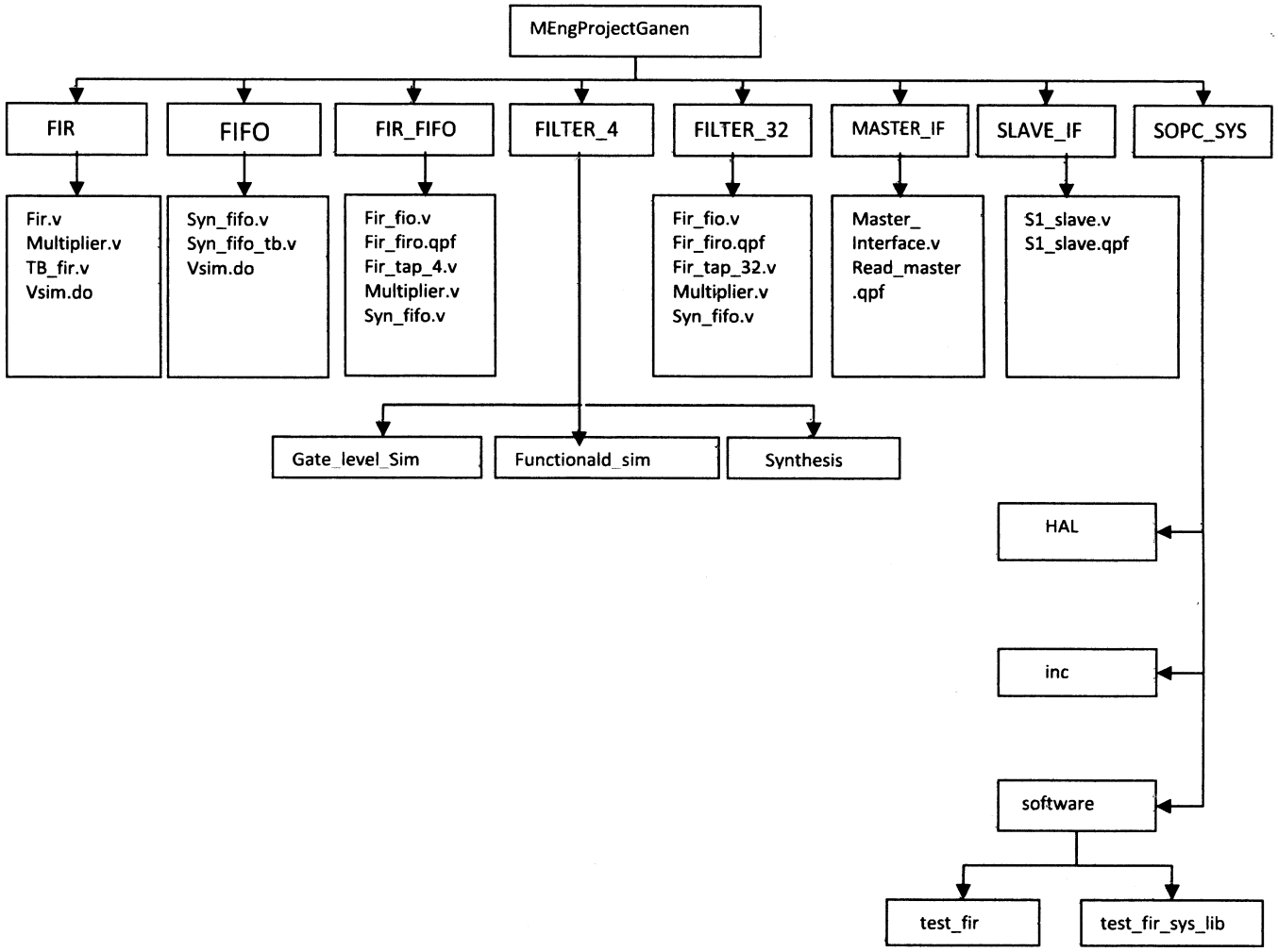
In this project we implemented a 32-tap pipelined programmable FIR filter that adopts the direct form architecture. Our custom design can be used in higher sampling rate applications than the Altera IP. It was observed that, with our assumptions, our design can achieve a 20% higher clock rate with only 3% area penalty.

In addition, a reusable hardware emulation platform was designed to verify the design. This emulation platform can be used to verify other custom logics in system-level with minimal modifications to the Avalon-Interface.

Due to timing limitation the effects of those results summarized in Table 5 were not studied for varying filter sizes (i.e. increasing tap lengths). It will be an interesting future study.

Appendix

A. Design File's Directory Structure



The design files, test cases, and the corresponding simulation scripts are arranged as shown in the above diagram. Each folder contains the corresponding behavioral HDL, testbench, and the simulation script. For synthesis, a separate directory is created and this directory includes a Quartus II project file as well. These files are with our research group and it can be obtained by contacting Dr. Andy G. Ye.

B. Explanation of the directory structure and design files

File/Directory Name	Description
/FILTER_4	Contains HDL files and Run scripts for GateLevel Simulation, Functional Simulation, and Synthesis
/Gate_level_sim	Contains the altera_library mapped netlist of the 4-tab FIR and the simulation file
TB_top	Gatelevel Simulation Testbench for the Top-level design
vsim.do	ModelSim simulation script
top.vo	gate-level netlist of the top-level design
pli_calculate_fir.dll	executable file of the PLI routine
/QuartusII_Synthesis	Contain all the HDL file and the Quartus II project for synthesizing the FIILTER_4
top.qpf	Quartus II project which specify the working directory and the design involved design files.
fir_tap_4.v	This file contain fir calculation core
syn_fifo.v	This file contain the FIFO logic
fir_fifo.v	fir and the FIFO modules put together
read_master.v	This file contain the logic for the Avalon-MM read master interface
sl_slave.v	This file contains logic for reading and writing to the FILTER_4 (Software registers) and the slave interface to the Avalon_MM bus.
top.v	This is the top-level of the FILTER_4
TB_top.v	This is the top-level testbench

Files and scripts in the Functional Simulation directory are similar to that of Quartus II Synthesis Directory. The next page will describe the SOPC SYSTEM directory.

File/Directory Name	Description
/SOPC_SYSTEM	This directory Contains HDL files of the whole system, header files defining the low-level hardware interface and C program to test the FILTER's hardware and software.
/inc	Sub-directory includes header files defining the low-level hardware interface
fir_reg.h	Defines the macros to access registers in the FILTER component
/software	Contains the test program to exercise FILTER's H/W & S/W
/test_SOPC_system	Contains NIOS II IDE project
/test_my_fir.c	test program write samples to the on chip ram, configure the filter and read back the results and display it on the consol.
/test_SOPC_system_syslib	System Library needed for the NIOS II IDE project

C. Software Source Code

fir_reg.h

```
/************************************************************************
*
* License Agreement
*
* Copyright (c) 2007 Altera Corporation, San Jose, California, USA.     *
* All rights reserved.                                                  *
*
*
************************************************************************/

/************************************************************************/
* Modified by N.Ganen to fit the custom component FIR filter          *
* march 2008                                                          *
/************************************************************************/


#ifndef __FIR_REGS_H__
#define __FIR_REGS_H__

#include <io.h>

/* Basic address, read and write macros. */

#define IOADDR_MY_FIR_ADDR(base)          __IO_CALC_ADDRESS_NATIVE(base, 0)
#define IORD_MY_FIR_ADDR(base)                  IORD(base, 0)
#define IOWR_MY_FIR_ADDR(base, data)            IOWR(base, 0, data)

#define IOADDR_MY_FIR_LENGTH(base)        __IO_CALC_ADDRESS_NATIVE(base, 1)
#define IORD_MY_FIR_LENGTH(base)                IORD(base, 1)
#define IOWR_MY_FIR_LENGTH(base, data)          IOWR(base, 1, data)

#define IOADDR_MY_FIR_CTRL(base)          __IO_CALC_ADDRESS_NATIVE(base, 2)
#define IORD_MY_FIR_CTRL(base)                  IORD(base, 2)
#define IOWR_MY_FIR_CTRL(base, data)            IOWR(base, 2, data)

#define IOADDR_MY_FIR_COEF_0(base)        __IO_CALC_ADDRESS_NATIVE(base, 3)
#define IORD_MY_FIR_COEF_0(base)                IORD(base, 3)
#define IOWR_MY_FIR_COEF_0(base, data)          IOWR(base, 3, data)

#define IOADDR_MY_FIR_COEF_1(base)        __IO_CALC_ADDRESS_NATIVE(base, 4)
#define IORD_MY_FIR_COEF_1(base)                IORD(base, 4)
#define IOWR_MY_FIR_COEF_1(base, data)          IOWR(base, 4, data)

#define IOADDR_MY_FIR_COEF_2(base)        __IO_CALC_ADDRESS_NATIVE(base, 5)
#define IORD_MY_FIR_COEF_2(base)                IORD(base, 5)
#define IOWR_MY_FIR_COEF_2(base, data)          IOWR(base, 5, data)

#define IOADDR_MY_FIR_COEF_3(base)        __IO_CALC_ADDRESS_NATIVE(base, 6)
#define IORD_MY_FIR_COEF_3(base)                IORD(base, 6)
#define IOWR_MY_FIR_COEF_3(base, data)          IOWR(base, 6, data)
```

```
#define IOADDR_MY_FIR_RESULT(base)          __IO_CALC_ADDRESS_NATIVE(base, 14)
#define IORD_MY_FIR_RESULT(base)                IORD(base, 14)

#define IOADDR_MY_FIR_STATUS(base)          __IO_CALC_ADDRESS_NATIVE(base, 15)
#define IORD_MY_FIR_STATUS(base)                IORD(base, 15)


/* Masks. */

#define MY_FIR_CTRL_GO_MSK                      (0x1)
#define MY_FIR_STATUS_DONE_MSK                  (0x2)
#define MY_FIR_LENGTH_MSK                       (0xFFFF)
#define MY_FIR_RESULT_MSK                       (0xFFFF)

/* Offsets. */

#define MY_FIR_CTRL_GO_OFST                     (0)
#define MY_FIR_STATUS_BSY_OFST                  (0)
#define MY_FIR_STATUS_DONE_OFST                 (1)

#endif /* __FIR_REGS_H__ */
```

Test_fir.c

```
/************************************************************************
* Copyright (c) 2007 Altera Corporation, San Jose, California, USA.     *
* All rights reserved. All use of this software and documentation is     *
* subject to the License Agreement located at the end of this file below.   *
************************************************************************/

/************************************************************************
 * This is a simple C program that exercises the FILTER_4 component by
 * writing to the onchp Ram  with test data and then configuring the
 * FILTER_4 to read back the calculated FIR results as well as
 * the status regusters of the FILTER_4 using the IOWR and IORD
 * as defined in the fir_regs.h file.
 *
 * Modified by Ganen 2008 March */

#include <stdio.h>
#include <stdlib.h>
#include ".\inc\fir_regs.h"
#include "system.h"
#include <alt_types.h>

//TOP_INST_BASE
// ONCHIP_MEM_BASE


// write sample values into onchip ram
int set_buf_val( alt_u32* buffer, int length, alt_u32 val )
{
  int ret_code = 0;
  alt_u32 test_data = val;
  int offset=0;


  while (offset <= length )
  {
        printf( "TEST DATA := %d  \n", test_data);
        *(buffer + offset) = test_data;
        if( *(buffer+offset) != test_data )
        {
              ret_code = -1;
        }
        for (int i = 0; i < 100; i++)
        {
              if (i<9) test_data = i*100; // 0,100,200,300
              else test_data = (i + 92); // 101,102,103 ...
        }
        offset++;
  }
  return( ret_code );
}
```

```
/* This program points the checksum component at a small buffer and
 * computes the checksum. */

int main()
{
  // Point the buffer at the base of the onchip ram.
  // base of onchip ram defined in system.h file "ONCHIP_MEM_BASE"
  alt_u32* buf = (alt_u32*) ONCHIP_MEM_BASE;
  alt_u64 addr_reg, length_reg, contorl_reg, coef0, coef1, coef2, coef3;
  alt_u32 mem, fir_result;

  int len = 170; // set the length to 170
  int coef_0 =1; // initialize the coefficients
  int coef_1 =2;
  int coef_2 =4;
  int coef_3 =6;
  int status;
  int result;
  int i;


  // Set the buffer to all 0xF0's.
  printf( "Writing to test memory.  \n");
  if( (set_buf_val( buf, len, 0x0002 )<0) )
  {
    printf( "Error:  Could not pre-set buffer at %d.\n", (int) buf );
    return( -1 );
  }


  // IORD and IOWR macros setup in alter_avalon_checksum_regs.h and io.h
  // read memory values


  for (i=0; i<30; i++){
        mem = IORD(ONCHIP_MEM_BASE,i);
        printf( "memory:= %d\n", mem);
  }

  // Store the address (must be 32-bit word aligned address).
  printf( "Writing to address register.  \n");
  IOWR_MY_FIR_ADDR( TOP_INST_BASE, ONCHIP_MEM_BASE );

  printf( "Reading from address register.  \n");
  addr_reg = IORD_MY_FIR_ADDR(TOP_INST_BASE);
  printf( "address register read success: %x\n", addr_reg);

  // Store the length in bytes (up to a 16-bit value).
  printf( "Writing to length register.  \n");
  IOWR_MY_FIR_LENGTH( TOP_INST_BASE, len );
  length_reg = IORD_MY_FIR_LENGTH(TOP_INST_BASE);
  printf( "Length register read success: %d\n", length_reg);

  printf( "Writing to coef registers.  \n");
  IOWR_MY_FIR_COEF_0(TOP_INST_BASE, coef_0);
  IOWR_MY_FIR_COEF_1(TOP_INST_BASE, coef_1);
  IOWR_MY_FIR_COEF_2(TOP_INST_BASE, coef_2);
```

```
  IOWR_MY_FIR_COEF_3(TOP_INST_BASE, coef_3);

  printf( "READING coef registers.  \n");
  printf("coef_0:= %d\n",IORD_MY_FIR_COEF_0(TOP_INST_BASE));
  printf("coef_1:= %d\n",IORD_MY_FIR_COEF_1(TOP_INST_BASE));
  printf("coef_2:= %d\n",IORD_MY_FIR_COEF_2(TOP_INST_BASE));
  printf("coef_3:= %d\n",IORD_MY_FIR_COEF_3(TOP_INST_BASE));


  // Tell it to "go".
  printf( "Writing to go bit in control register.  \n");
  IOWR_MY_FIR_CTRL( TOP_INST_BASE+MY_FIR_CTRL_GO_OFST, MY_FIR_CTRL_GO_MSK);

  //Polling loop waiting for the component to be done.
  printf( "Polling for DONE bit in status register. . .  \n");
  status = IORD_MY_FIR_STATUS( TOP_INST_BASE );
  while( !(status & MY_FIR_STATUS_DONE_MSK) )
  {
        status = IORD_MY_FIR_STATUS(TOP_INST_BASE);
        fir_result = IORD_MY_FIR_RESULT(TOP_INST_BASE);
        printf ("fir result := %d\n", fir_result);
  }
  printf( "Done bit asserted, exiting polling loop.  \n");

  printf( "Done reading fir result \n", (int) result );

  return 0;
}
```

calculate_fir.c

```
// THIS IS THE C MODEL OF THE FIR USED FOR THE PIL ROUTINE
// This code was compiled using in Microsoft visual C++ 2008
#include "veriuser.h" // this is the header file for the verilog simulator

// fir calculating function
int firFilter( int coefPtr[], int dataPtr [], int y[])
{
      for (int n = 0; n < 99; n++) {
            int sum=0;
            for (int k = 0; ((n-k)> -1) &&( k< 4); k++){
                  int a=((n-k)<0)?0:(n-k);
                  sum +=  dataPtr[a]*coefPtr[k];
            }
            y[n] = sum;
      }

      return 0;
}

extern "C" __declspec(dllexport) PLI_INT32 CALCULATE_FIR()
{
      io_printf("\n\n**** Wellcome to the wonderful world of PLI ****\n");

      int coef [4] ={1,2,4,6};
      int data [99] ;
      int y [99];


      //initialize data array

      for (int i = 0; i < 100; i++){
            if (i<9) data[i] = i*100;
            else data[i] = (i + 92);
            //printf("%f , %0f\n",(i%4),data[i]);
            }

      firFilter (coef, data,y);

      for (int i = 0; i < 30; i++){
      io_printf("FIR_RESULT[%d]:=%0d\n",i,y[i]);
      io_printf("\n\n\t End of PLI\n");
      io_printf("*********************************************************\n");
      io_printf(" Manually check the simulation results  \n");
      io_printf("*********************************************************\n");
      return 0;
}

extern "C" __declspec(dllexport) s_tfcell veriusertfs[] = {
      // this is the simulator specific part, this is for modelsim
      {usertask, 0, 0, 0, CALCULATE_FIR, 0, "$CALCULATE_FIR"},
      {0} // last entry must be 0
};
```

References

[1] J.M. Pierre Langlois, "Design and Implementation of High Sampling Rate Programmable FIR Filters in FPGAs," *Circuits and Systems, 2006 IEEE North-East Workshop on* , vol., no., pp.37-40, June 2006

[2] Bilsby, D.C.M.; Walke, R.L.; Smith, R.W.M., "Comparison of a programmable DSP and a FPGA for real-time multiscale convolution," *High Performance Architectures for Real-Time Image Processing (Ref. No. 1998/197), IEE Colloquium on* , vol., no., pp.4/1-4/6, 12 Feb 1998

[3] Altera, "FPGA vs. DSP Design Reliability and Maintenance" pp.1-4, 2007

[4] AccelChip, Inc., "Filter Design Methods for FPGAs" pp. 1-10, 2004.

[5] R. Petersen and B. Hutchings, "An Assesment of the Suitability of FPGA-based Systems for Use in DSPs", in Lecture Notes in Computer Science, n°975, pp.293-302, Springer-Verlag, Berlin: 1995.

[6] Baumgartner, D.; Rossler, P.; Kubinger, W., "Performance Benchmark of DSP and FPGA Implementations of Low-Level Vision Algorithms," *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on* , vol., no., pp.1-8, 17-22 June 2007

[7] Altera, " Nios II Processor Reference Handbook", pp.1-232, 2007

[8] Altera, "Developing Components for SOPC Builder", Quartus II 7.2 Handbook, Volume 4, pp.1-32, 2007

[9] Les Mintzer, "FIR Filters with Field-Programmable Gate Arrays", Journal of VLSI Signal Processing, vol. 6, pp. 119-127, 1993

[10] Principles of Verilog PLI, Kluwer Academic Publisher, 1999, ISBN 0-7923-8477-6

[11] Chi-Jui Chou, Satish Mohanakrishnan, and Joseph B. Evans, "FPGA Implementation of Digital Filters", Proc. Int. Conf. Signal Proc. Appl. & Tech. (ICSPAT'93), 1993.

[12] Altera, "FIR Compiler User Guid", pp. 1-86, 2007