ECE532

Project Group Report

DIGITAL ADAPTIVE EQUALIZATION FOR SERIAL DATA COMMUNICATION

Boris Spokoinyi	(991360255)
Ryan Janzen	(991470065)

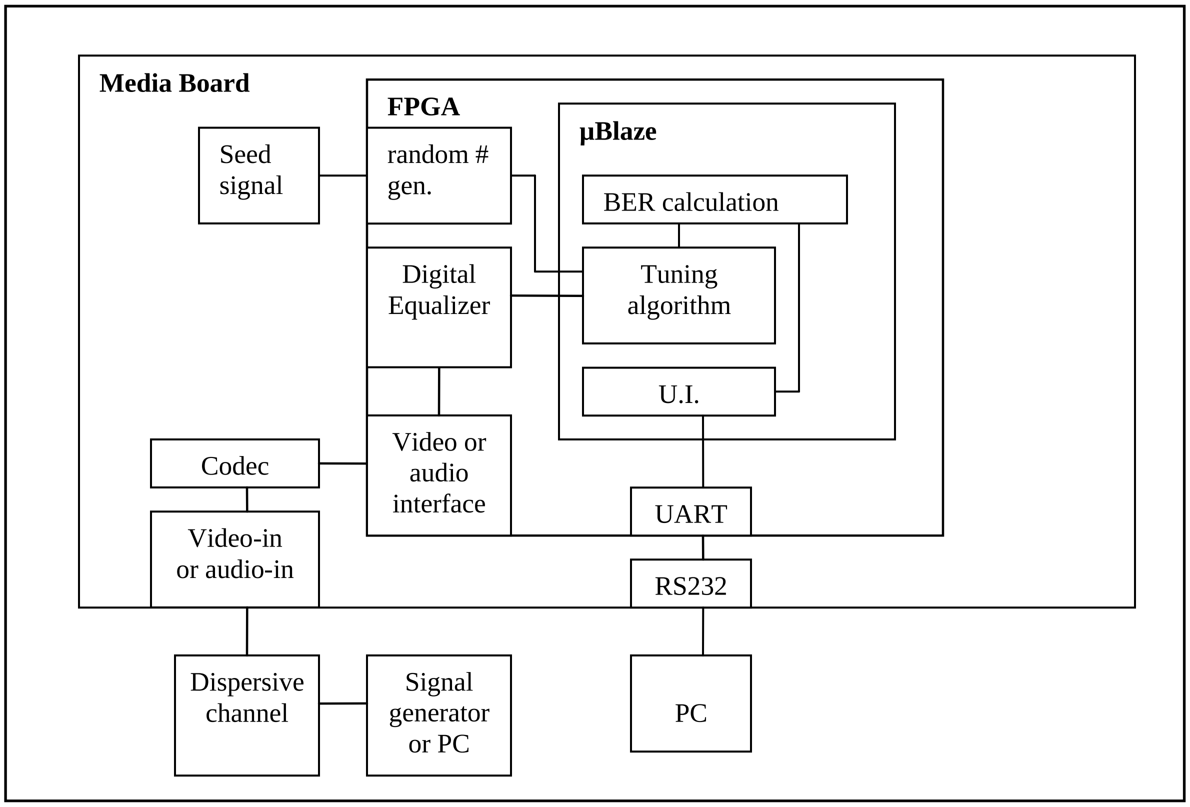
March 27, 2005

Overview

Initial Goals

The purpose of this project is to build an adaptive equalizer on FPGA to equalize asynchronous serial data that have been distorted by a certain channel. The serial data would travel from a RS232 port of PC, through a long cable, that introduces distortion and noise, then it would travel to audio input on Multimedia Board, get equalized using the weights from µBlaze, then the equalized data would be fed to µBlaze for error calculation. Using Genetic Algorithm the weights would be calculated by µBlaze based on the calculated error from difference in equalized and actual data.

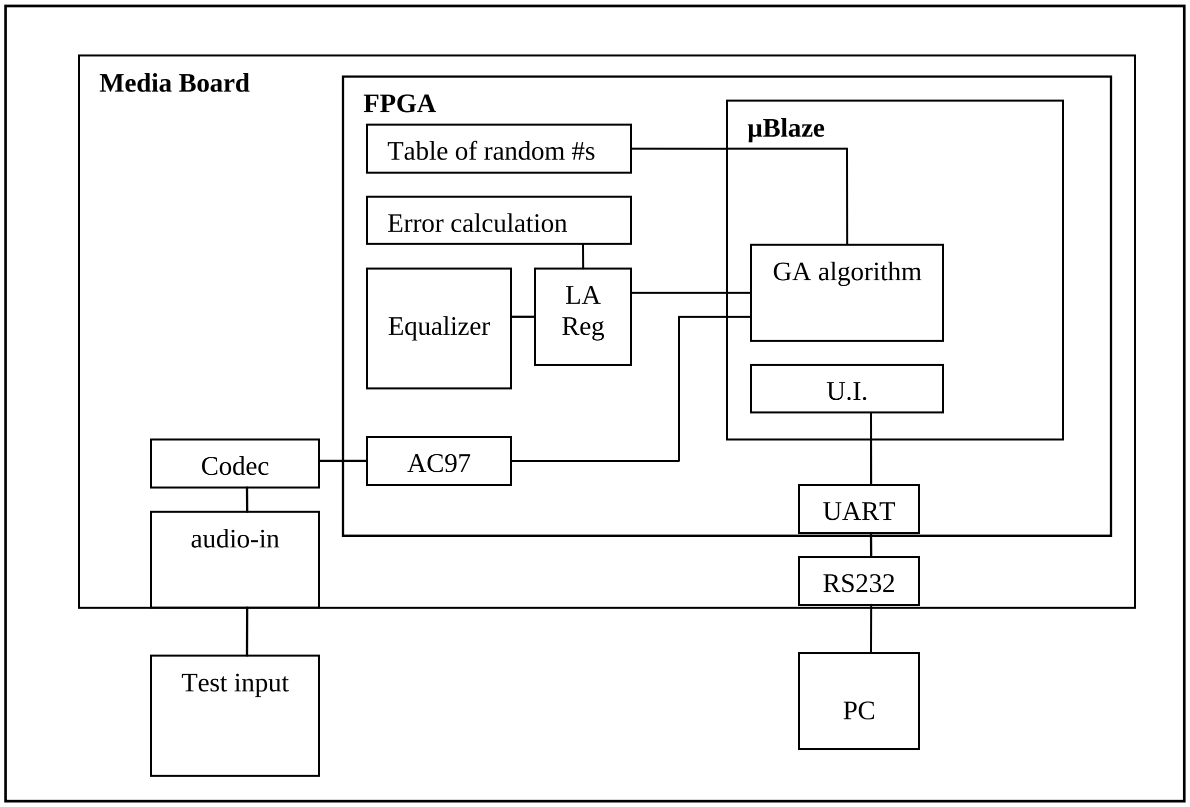
Initial block diagram



Modified Goals

In final design the error is calculated in hardware instead of µBlaze. Random number generator is replaced by a lookup table with a pre-computed pseudorandom sequence (PRS) of numbers. A second, 8-bit PRS is hard-wired in the error calculation logic. Logical Addressable Register (LAregister) is added to make equalizer's registers accessible from OPB bus. Audio input is used only for verifying that we can read from microphone input from µBlaze. The actual data, original and distorted, is pre-computed and fed from µBlaze to the equalizer. In future the data could be fed from microphone through AC97 core as it was intended.

Modified block diagram



IP blocks used

- MicroBlaze
- AC97 core, from one of previous year's projects
- Xilinx GPIO core
- Xilinx UARTlite core
- Custom "equalizer" core
- Custom "LAregisiter" core

Outcome

Due to time constraints we did not have time to integrate µBlaze GA code with equalizer core and LAregister core. By itself, µBlaze GA code worked well for a simple error surface, where error is the sum of all weights squared. A complete convergence was observed, for this simple case, in about 5s with 27MHz clock. AC97 core worked well from µBlaze: samples were successfully recorded into the memory from microphone input. The equalizer was successfully simulated in Quartus and later in ModelSim; it correctly equalizes a sample pre-coded low-passed PRS sequence using static weights.

Description of IP Blocks & Other System Segments

µBlaze code: GA algorithm

Introduction to Genetic Algorithm based equalization

The goal of equalizer is to minimize error between a correct signal and an equalized one. Adaptive equalization is used in cases where the channel can change from place to place or with the passage of time. There are basically two major ways to adaptively update weights of an equalizer. One possible way is to adjust the weights using a gradient descent approach, such as LMS, and another way is do random search. Gradient descent is a fast way to find a weight vector corresponding to the minimal error but it only gives a suboptimal solution and has performance problems when there is a lot of noise. A random search by itself is a very slow method but it is guaranteed to eventually converge to the optimal solution. Genetic Algorithm is a sort of combination of random search and gradient descent. The gradient descent is performed through averaging (crossbreeding) of weights between different weight vectors (individuals) and keeping only the best performing ones. Random perturbations (mutations) are introduced to the weights to perform random search, usually near the local minima, so that optimal solution is eventually found.

Code description `GA.c`

The GA code consists of several functions for each of the Genetic operations: crossbreeding, mutation, picking the best individuals, and calculating error for each individual based on the error provided by the equalizer core.

Crossbreeding `void crossbreed()`

This part of GA algorithm is a major mechanism by which the solution converges to a local minimum. In this implementation, crossbreeding is done by having two imbedded loops: one selects first parent and the other selects the second parent. A parent-to-parent crossbreeding happens with certain probability, which in this case is implemented by taking a random number from PRS table and seeing if it exceeds a certain threshold.

Mutation `void mutate()`

Mutation is necessary so that the solution eventually finds an absolute minimum on the error surface. Mutation is performed on each individual except the best individuals (elite). Mutation is not performed on the elite because if it was then the best solution would be lost.

Calculating errors `void calcErrs()`

Error is calculated at each iteration for each individual in the population. The weight vector of each individual is transferred to LAregister's *register_portCoeffI* along with 16 bits of a distorted data to *register_portSignalI*. After this the software loops until a status register, *register_portStatusI*, is read from LAregister indicates that the equalization and error calculation is done in the Equalizer core. The error is read from *register_portErrorO* port of LAregister. µBlaze also reads the equalized data from LAregister so it can be output for comparison. This procedure is repeated for each individual.

For testing of GA algorithm a simple test error calculation is performed by simply squaring and adding all weights of a weight vector. The GA in this case finds a weight vector that has all weights of zero.

Sorting to find the Elite `void findElite()`

A simple sorting of the population is performed to find out the two best performing individuals (elite). A simple algorithm is employed, where an individual with minimum error is searched through the population and is then placed at position 0, then the next best individual is found in similar fashion and placed at position 1.

Equalizer IP Core   v.1.00b

equalizer_v1_00_b
Authors: Ryan Janzen, Boris Spokoinyi
2005.03

Dynamic Equalization of communication data received from a noisy or distorting channel. The core implements a delay-tap-line, where all the delayed versions of the input signal are multiplied by set coefficients, and summed together to form an output signal. The coefficients can be set as a matched filter, matched to the shape of the transmitted pulses, to combat noise. They can also be set to form a filter which undoes the effect of a distorting channel's impulse response. (The latter is used in this project.)

Key features:

- Dynamic Equalization using 8-Delay-Tap Line
- Binary Threshold Detection and Output bit sequencing
- Compatible with Frame-based communication schemes, with beginning of each data frame containing a pre-determined Pseudo-Random Sequence (PRS) of bits for frame synchronization and error rate detection.
- Error Rate Detection

In this project, the coefficients are controlled dynamically by the Genetic Algorithm. The error rate calculation is fed back into the GA. These two exchanges of data take place once every data frame.

I/O Ports

Port	Dir.	Width	
`clk`	I	1	External clock
`reset_b`		1	External ~reset
`coeffArrayIn`		64	8x Coefficents, 8-bit each. First is coeffArrayIn[7:0]
`signalIn`		16	Sample of received analog communication signal
`statusIn`		8	(desc. below)
`statusOut`	O	8	(desc. below)
`statusOutIsValid`		1	(Can enable an external latch)
`dataBlockOut`		8	Sequence of output bits, equalized and thresholded corresponding to 8 samples at 2nd half of frame
`dataBlockIsValid`		1	(Can enable an external latch)
`errorRate`		16	Cumulative analog error calculation for PRS section of frame. Unsigned
`errorRateIsValid`		1	(Can enable an external latch). Error available before end of frame.
`signalOut`		32	Equalized analog sample (optional). Signed. Flow controlled by statusIn

During the first part of each frame, the equalizer output is compared with a hard-coded 8-bit pseudorandom sequence (PRS) of bits which correspond to actual known bits which are to be sent by the transmitter at the beginning of the frame. The purpose is to have an indication of the error rate between transmitter, distorting channel, equalizer, and output bit sequence. Each equalized sample, which is a quantized analog value, is subtracted from the corresponding bit in the PRS. The discrepancy's absolute value is accumulated on an errorRate register. After the 8-sample sequence is complete, plus an additional 4-sample delay which accounts for the EQ latency, the errorRateIsValid bit is asserted.

The remainder of each frame contains the actual communication data. The equalizer output samples are threshold-detected (>0 or <0) to interpret a communication bit. Bits are sequenced into a byte, which is the data output of the frame.

The errorRate is made available as soon as possible during each frame, so that the software has as much advance time as possible to read it and calculate new coeff's, which are needed at the beginning of the next frame.

Timing: Data segmentation in 1 frame.

Coefficient Data in.			
	Frame – 16 Samples, 1 for each bit. Samples from distorted channel written one at a time to core.		
	Pre-determined PRS (8 bits)	Data (8 bits)	
		Error total out	
			Byte out

Interface protocol:

- SW writes coeffArray.
- Assert statusIn[2] (3rd bit from right) to 1. "Start of frame"
- Write signalIn, which is the a sample of a single symbol.
- Assert statusIn[0] to 1
- Wait for statusOut[0] to go to 1. (can tie this to an interrupt, in LAregister)
- Clear statusIn[0] to 0, and write new signalIn.
- Assert statusIn[0] to 1.
- ...so on for rest of input symbol samples. 16 in total.
- At end of frame of signalIn's, statusOut[2] will go to 1. "End of frame"
- Read output data byte and error.
- Clear statusIn[2] to 0 to acknowledge.
- Repeat all, for next frame

Status Registers:

statusIn [7:0]	**Control bits** (active high)
0	Sample available (cycle 0,1 for next sample)
1	
2	Frame active ~Data out acknowledge
3	
4	
5	
6	
7	

statusOut [7:0]	**Status bits** (active high)
0	Sample complete
1	Error complete Error out valid
2	Frame complete Data out valid
3	Truncated sample # counterSampleBlock[2:0]
4	
5	
6	State – Top level state machine
7	

Hardware compatibility versions:

- 1_00_a Self-contained input/output data registers
- 1_00_b Asynchronous inputs/outputs, assuming external latches. Designed for opb_LAregister_v2_00_a

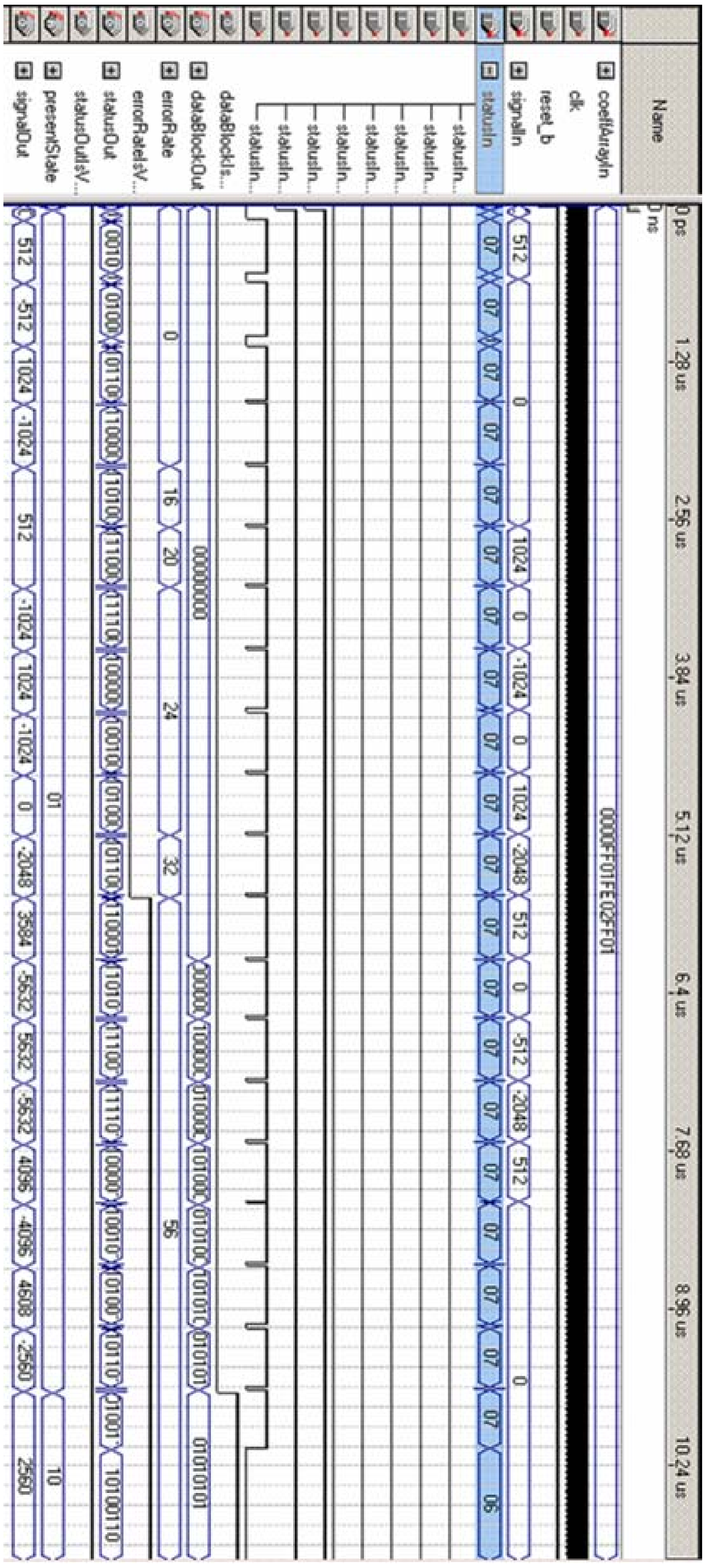
The core is written in Verilog, with the following modules:

`signalproc.v`	Top level HDL of core. Instantiates equalizer and errorChecker. Implements threshold detection (slicing) of equalized signal, and sequences the resulting binary data. Contains state machine.
`equalizer.v`	Dynamic delay-tap-line equalizer. Uses embedded multipliers. Contains state machine.
`errorChecker.v`	Analyzes EQ output signal during first part of frame, comparing samples to an internally hardcoded PRS, accumulating an analog error total.
`MULT18X18.v`	Xilinx instantiation of Virtex II embedded multiplier
`signExtend_8_18`	Vector bit expansion, from 8bit to 18bit, with sign extension.
`signExtend_16_18`	Vector bit expansion, from 16bit to 18bit, with sign extension.

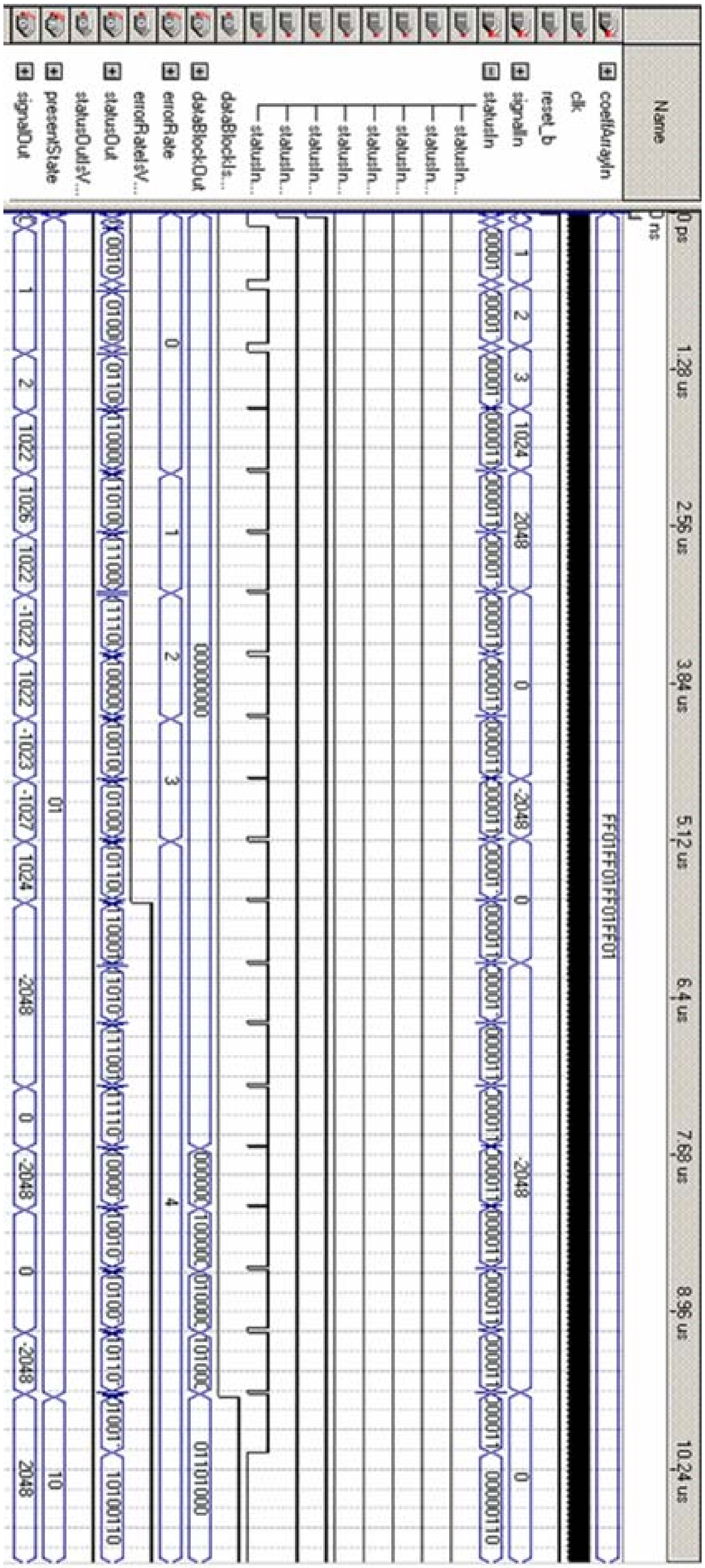
“Frame” also referred to as “block” within code.

The hardware design logic was verified. Simulation waveforms are shown as follows.

Simulation 1



Simulation 2



Logical Addressable Register IP Core v. 2.00a

opb_LAregister_v2_00_a
Authors: Ryan Janzen, Boris Spokoinyi

LAregister makes data transfer possible between the µBlaze and another custom core (EQ in this case), using memory-addressable hardware registers. This custom core was generated by the XPS Create/Import Peripheral Wizard to include OPB addressable register capability, and then modified so that the registers could be interfaced with hardware ports. It is intended that these ports be connected to corresponding ports in the custom core of interest, using XPS/MHS.

	Software interface	Hardware interface
I registers "inputs to the external core"	Software writes a bytes/word to the appropriate memory address. Read-back is possible.	Data is continuously available on the ports `register_port<name>I`

O registers
 	
"outputs from the external core"
	Software reads a bytes/word from the appropriate memory address.
	Write-back is disabled and has no effect.
		External core strobes output vector on
		`register_port<name>O`
		External core asserts
		`register_port<name>O_latch`
		Data is latched every bus clock cycle when this signal is high.

The design presented here has been customized for the EQ core. Data ports have been given names and widths that correspond with EQ ports.

The LAregister core contains 32 registers, each one 32-bits wide. There are ports which give the EQ core access to register data, and these ports are to be connected to the EQ core ports in XPS/MHS. "I" ports are to be written to by software and are "inputs to the EQ core". "O" ports are readable, but not writable, by software and are "outputs from the EQ core". Within the LAregister HDL, O ports are inputs and I ports are outputs.

Upon including the core in a system in XPS, address ranges are to be set up as one contiguous block. In this case, ports and their address locations are as follows:

Addr. Byte offset	Wired port, "I"/"O"	bits	Direction	Internal reg
0 ~ 255	register_portCoeffI	2040	bus->port	slv_reg0 & slv_reg1 & slv_reg2 & slv_reg3 & slv_reg4 & slv_reg5 & slv_reg6 & slv_reg7
1016	register_portStatusI	8	bus->port	slv_reg30(0 to 7)
1018~9	register_portSignalI	16	bus->port	slv_reg30(16 to 31)

1020	register_portStatusO	8	port->reg->bus	slv_reg31(0 to 7)
1021	register_portDataO	8	port->reg->bus	slv_reg31(8 to 15)
1022~3	register_portErrorO	16	port->reg->bus	slv_reg31(16 to 31)
	Latch enable signals:			
-	register_portStatusO_latch	1	port->reg	-
-	register_portDataO_latch	1	port->reg	-
-	register_portErrorO_latch	1	port->reg	-

The core has a VHDL design, with the following modules:

`opb_LAregister.vhd`	Top level HDL of core. Originally generated by Create Peripheral Wizard. Modified to contain i/o ports as above.
`user_logic.vhd`	Originally generated by Create Peripheral Wizard. Modified to contain i/o ports which are connected to registers in the entity, as in the above table. Particularly for O ports, the registers were given enable signals which are connected to the external _latch signals.

Test Procedure, independent of external core

- In XPS/MHS, after inserting the core and setting memory addresses, add the ports.
- In MHS, hard wire the "O" ports (port-->reg-->bus) to random 0's and 1's.
```
# Simulated data from Verilog core, to be latched to reg.
PORT register_portStatusO = 0b10001011
```
- Hard wire the "latch" signal for each “O” port - some to "0", some to "1". This will cause only certain “O” registers to continually latch the port data. The others will remain initialized at zero.
- In software, or XMD, try read from the logical addresses you have set for these registers.
- When the software writes to the "I" ports, that data should appear on the output port wires you have access to. Eventually, connect those wires to another core.

OPB Interrupt controller v1.00c

This component is provided with EDK 6.3.

Embedded Multipliers Virtex II – MULT18X18

8 hard multipliers are instantiated from the EQ core. Each has two 18-bit inputs and a 36-bit output, for signed, twos-complement numbers. They are treated as asynchronous and take 1 clock cycle for the operation to complete.

Description of Design Tree

- uBlaze source code (GA algorithm): file **GA.c**
- Equalizer core: folder **pcores\equalizer_v1_00_b**
- LAregister core: folder **pcores\opb_LAregister_v2_00_a**
- LAregister software drivers: folder **drivers\ opb_LAregister_v2_00_a**

Resource Requirements

- Xilinx Multimedia Board
- PC for serial data transmission and equalization verification