## ZERO OVER HEAD LOOP FIR PROCESSOR

**Sourindu Chatterjee**

**Dr. Uwe Meyer Baese**

**Department of Electrical and Computer Engineering**

**Florida State University**

**Spring 2016**

## Brief Description

The reason for this project is to actualize the hardware specific strategy found in most DSP processors, ZERO OVER HEAD LOOPING towards the improvement of a FIR microprocessor. The design filter coefficients are continuous and supplied to the processor as a signal[5]. The design has a innovative approach to adaptive filtering thus providing a way to change the filter length in real-time during execution of different states. The input value are loaded in after the full convolution cycle and then again another convolution initiates. The key objective of the design is to provide a way for the microprocessor to have a single embedded multiplier for any number of filer coefficients.

## Definition Of The Problem

Well the execution of Zero Overhead Loop (ZOL) for different embedded DSP chip is different in perspective when comparison is drawn between hardware and software implementation of the same [1]. As discussed earlier this project is about usage of Zero Overhead Loop control in Hardware thus coded in Very Fast Hardware Description Language (VHDL) for the outline of a ZOL FIR microprocessor, such that the execution of cyclic ‘Multiply and Accumulate’ for each convolution takes one cycle for each execution and furnishes the final convolution result after n number of cycles where n is the number of coefficients in the FIR system. There is another challenge connected with the advancement of the FIR microprocessor[6], which is to make the loading of the coefficients real-time through an input signal. Well this problem is addressed and in addition to this there has been an enhancement to the design which enables the microprocessor to load in filter length or the number of coefficients through an input signal. This makes the design more robust, dynamic and adaptive [3].

## Solution to The Problem

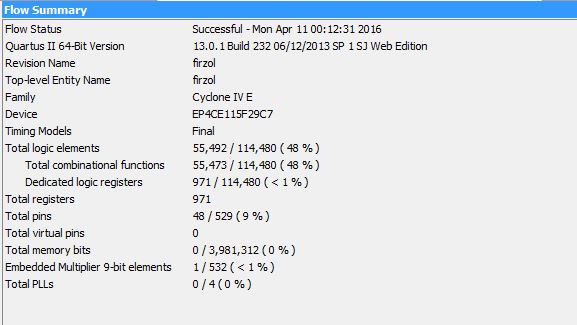
We realize that the most crucial mathematical statement in FIR microprocessor development is the Sum-Of-Product (SOP) evaluation, y[n] = ∑h[k] ∗ x [n−k] [6] [5], which can be likewise termed as convolution in the time domain of data sign x and impulse response h. This is the reason ZOL is paramount to the execution, as this bit of mathematical statement requires multiple Embedded Multiplier blocks in any Digital Signal Processors or FPGA`s. The convolution under ordinary microprocessors require and information to be executed in a loop over n number of times each time a piece comparison is made [6][1][3] inside a convolution phase.

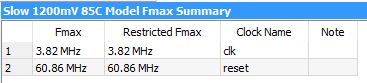
The issue identifying with ZOL is critical and now and again crucial to the proficiency of the DSP microchip. In this anticipate the assignment is to build up a FIR chip with ZOL support. This is completely genuine on the grounds that the vast majority of the tasks performed by DSP chip are on real-time where the cycles per convolution should be as less as could be expected under these dynamic circumstances, whether it is ALU performance or circling guideline the diminishment in execution time is essential and a basic component in deciding efficiency of the framework [4].

Thus the design of the ZOL Fir microprocessor is implemented with a soul target that the number of embedded multiplier blocks in the design should be equal to 1 and that the design should produce convolution output in n number of cycles where will be equal to the number of coefficients in the coefficient array.

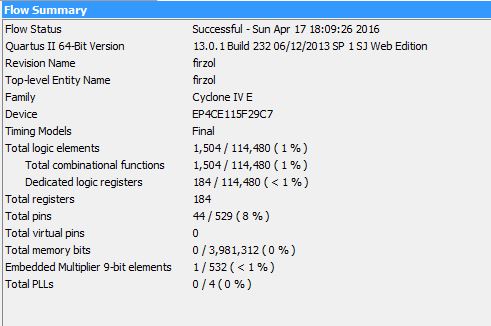
## Design Specification

This is for generic length of 50.





This is for generic length of 7



## Code Procedure & Flow

The project is coded in VHDL code composed and arranged with Quartus II programming. The simulations are done with ModelSim. The utilization of Matlab is likewise there to create channel coefficients set and determine the correctness of the output.

The design can be broadly classified into five sections: reset, start, mac, trans & done. The proceedings of these sections will be briefly described.

**reset** : In this section the system first enter it is the default section at the beginning of the program. The “**reset**” signal is an exernal asynchronous reset and sets all registers and register arrays to ‘0’. There is also an internal reset signal, “**rst**” associated with section which gets triggered the filter length “**L\_in**” or the coefficient input switch “**Load\_c**” gets changed from previous values. After completion of these state the design enter the synchronous section.

**start** : This stage basically sets the base for the whole execution cycle to proceed. The primary aim of this section is loading of input coefficients and input data and placing them is respective integer register arrays. This state depends on coefficient loading signal “**Load\_c**” and data loading control signal “**Load\_x**”.

**trans** : This stage is there to make the input of the next data feasible without hampering the existing output of the processor. This makes use of the input loading switch “**Load\_x**” to load the new input data. These stage also shift the previous values one position to fit in the newest data in the stream.

**mac** : This is the most important stage in the execution cycle. This is the section where the ZOL is implemented in order to get the best performance out of any hardware in which this processor will be implemented. In this stage the product of each elements in the coefficient array and data array is computed and the accumulator “**d**” is modified in each clock cycle to display the current accumulated value. At the end of each “**mac**” stage this accumulated value is transferred to a register “**a\_temp**” which is then furnished to the output “**y\_out**”. These stage which in the middle of MAC execution cycle does not care to check any changes in the parameter, only at the end of the execution these checks are resumed.

**done** : These stage signifies the end of one convolution cycle and the control remains in this stage until any changes to switching signals like “**L\_in**”, “**Load\_x**” or “**Load\_c**”. On the onset of such a change the control moves to ‘trans” state in case of change in “**Load\_x**” or “**start**” state in case of changes to other switching signal.

Code Segment

PACKAGE n\_bit\_int IS -- User defined type

SUBTYPE S15 IS INTEGER RANGE -2\*\*14 TO 2\*\*14-1;

TYPE STATE\_TYPE IS (start, trans, mac, done);

END n\_bit\_int;

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

USE ieee.std\_logic\_arith.ALL;

USE ieee.std\_logic\_signed.ALL;

LIBRARY ieee\_proposed;

USE ieee\_proposed.fixed\_float\_types.ALL;

USE ieee\_proposed.fixed\_pkg.ALL;

USE ieee\_proposed.float\_pkg.ALL;

LIBRARY work;

USE work.n\_bit\_int.ALL;

ENTITY firzol IS

GENERIC (W1 : INTEGER := 9; -- Input bit width

W2 : INTEGER := 18;-- Multiplier bit width 2\*W1

W3 : INTEGER := 19;-- Adder width = W2+log2(L)-1

W4 : INTEGER := 11;-- Output bit width

L : INTEGER := 7 -- Filter length

);

PORT(

clk : IN STD\_LOGIC; -- System clock

reset : IN STD\_LOGIC; -- Asynchronous reset

Load\_x : IN STD\_LOGIC; -- data input/loading switch

Load\_c : IN STD\_LOGIC; -- coefficients input switch

L\_in : IN INTEGER RANGE 1 TO L; --The actual number of coefficients

x\_in : IN STD\_LOGIC\_VECTOR(W1-1 DOWNTO 0);-- System input

c\_in : IN STD\_LOGIC\_VECTOR(W1-1 DOWNTO 0);-- Coefficient data input

y\_out : OUT STD\_LOGIC\_VECTOR(W3-1 DOWNTO 0));--System Output result

END firzol;

ARCHITECTURE fpga OF firzol IS

SUBTYPE SLVW1 IS STD\_LOGIC\_VECTOR(W1-1 DOWNTO 0);

SUBTYPE SLVW2 IS STD\_LOGIC\_VECTOR(W2-1 DOWNTO 0);

SUBTYPE SLVW3 IS STD\_LOGIC\_VECTOR(W3-1 DOWNTO 0);

TYPE A0\_L1SLVW1 IS ARRAY (0 TO L-1) OF SLVW1;

TYPE A0\_L1SLVW2 IS ARRAY (0 TO L-1) OF SLVW2;

TYPE A0\_L1SLVW3 IS ARRAY (0 TO L-1) OF SLVW3;

SIGNAL x\_count\_ini : INTEGER RANGE -1 TO L := 0; -- counting register for input data array

SIGNAL d\_out : S15 := 0; -- temporary accumulator result in each cycle

SIGNAL a\_temp : SLVW3; -- final result of the accumulator variable

SIGNAL x\_ar\_out : A0\_L1SLVW1 ; -- array for data input

SIGNAL c\_ar : A0\_L1SLVW1 ; -- Coefficient array

SIGNAL p : A0\_L1SLVW2 ; -- Product array

SIGNAL state : STATE\_TYPE; --state type variable

SIGNAL rst : STD\_LOGIC := '0'; --Internal reset

BEGIN

St : PROCESS(clk, reset, c\_in, x\_in, Load\_x, Load\_c, state, p, L\_in, rst)

VARIABLE L\_current : INTEGER := L\_in; -- actual number of coefficients

VARIABLE d : INTEGER := 0; -- temporary accumulator result in each cycle

VARIABLE x\_ar : A0\_L1SLVW1; -- array for data input

VARIABLE x,c : SLVW1; -- Just to make the Multipliction.

VARIABLE c\_count : INTEGER RANGE -1 TO L := 0; -- counting variable for coefficient array.

VARIABLE count : INTEGER RANGE -1 TO L := 0; -- counting variable for MAC state.

VARIABLE x\_count\_ini : INTEGER RANGE -1 TO L := 0; -- counting variable for data input array.

BEGIN ------> Load data or coefficients

IF reset = '1' OR rst = '1' THEN

rst <= '0';

L\_current := L\_in;

x := (OTHERS => '0');

c := (OTHERS => '0');

count := 0;

c\_count := 0;

x\_count\_ini := L\_current-1;

state <= start;

FOR K IN 0 TO L-1 LOOP

EXIT WHEN K = L\_current;

c\_ar(K) <= (OTHERS => '0');

x\_ar(K) := (OTHERS => '0');

END LOOP;

ELSIF rising\_edge(clk) THEN

CASE state IS

--start state

WHEN start =>

IF Load\_c = '1' THEN

c\_ar(c\_count) <= c\_in; -- Store coefficient in register.

IF c\_count = L\_current-1 THEN

c\_count := 0;

ELSE

c\_count := c\_count+1;

END IF;

END IF;

IF Load\_x = '1' THEN

IF x\_count\_ini = -1 THEN

x\_count\_ini := L\_current-1;

END IF;

x\_ar(x\_count\_ini) := x\_in; -- Get one data sample at a time.

x\_count\_ini := x\_count\_ini-1;

END IF;

IF L\_current /= L\_in THEN

L\_current := L\_in;

rst <= '1';

state <= start;

ELSIF Load\_x = '0' AND Load\_c = '0' THEN

x\_count\_ini := L\_current-1;

state <= trans;

END IF;

x\_ar\_out <= x\_ar; -- Store input data in register.

--trans state

WHEN trans =>

IF Load\_x = '1' THEN

-- Get one data sample at a time

FOR K IN 0 TO L-2 LOOP

EXIT WHEN K = L\_current-1;

x\_ar(x\_count\_ini-K) := x\_ar(x\_count\_ini-K-1);

END LOOP;

x\_ar(0) := x\_in;

END IF;

x\_ar\_out <= x\_ar;

IF L\_current /= L\_in THEN

L\_current := L\_in;

rst <= '1';

state <= start;

ELSE

d := 0;

state <= mac;

END IF;

--MAC state

WHEN mac =>

x := x\_ar(count);

c := c\_ar(count);

d := d + CONV\_INTEGER(c\*x);

count := count+1;

d\_out <= d;

IF count = L\_current THEN

a\_temp <= CONV\_STD\_LOGIC\_VECTOR(d,19);

d := 0;

count := 0;

IF L\_current /= L\_in OR Load\_c = '1' THEN

L\_current := L\_in;

rst <= '1';

state <= start;

ELSIF Load\_x = '1' THEN

x\_count\_ini := L\_current-1;

state <= trans;

ELSE

state <= done;

END IF;

ELSE

state <= mac;

END IF;

--done state

WHEN done =>

IF L\_current /= L\_in OR Load\_c = '1' THEN

L\_current := L\_in;

rst <= '1';

state <= start;

ELSIF Load\_x = '1' THEN

x\_count\_ini := L\_current-1;

state <= trans;

END IF;

x\_ar\_out <= x\_ar;

END CASE;

END IF;

END PROCESS St;

y\_out <= a\_temp;

END fpga;

## Simulations

1. This is the first of the simulations where we can see that the filter length is depending on a control signal “L\_in” and is not using the generic length of the filter. Also the loading of data and coefficints in respective arrays is showcased. The MAC cycle for ZOL implementation is evident where in order to furnish the first output the processor takes 3 number of cycles which is equal to the dynamic filter length. Also a second input data is processed by switching the “Load\_x” signal and we can a set of convolution cycle to get the final output of “-20”.
2. Here the processor after furnishing a set of MAC cycles is subjected to a change in the coefficient array length and as evident the control returns to the start stage and begins execution again.
3. In this simulation a change in the dynamic filter length is processed but no change in the coefficients and data switches thus furnishes an output of ‘0’.
4. Here there has been a change in the control signal “Load\_c” and also a input of coefficient “9” but no change in data array is evident.
5. This is the final Simulation, where we see that there has been a change in the filter length determining switch “L\_in” from 3 to 4. Then there is again initiation of “start” state and an impulse response of the filter coefficient which furnishes the last input coefficient “7” to the output.

## Simulation Files .do

1. The first do file firzolfinalv1.do

set project\_name "firzol"

vlib ieee\_proposed

vcom -work ieee\_proposed fixed\_float\_types\_c.vhdl

vcom -work ieee\_proposed fixed\_pkg.vhd

vcom -work ieee\_proposed float\_pkg\_c.vhdl

vlib work

vcom $project\_name.vhd

vsim work.${project\_name}(fpga)

########## Add I/O signals to wave window

add wave -divider "Simulation by Sourindu Chaterjee"

add wave -divider "Control Signals:"

add wave clk reset Load\_x Load\_c L\_in

add wave -divider "Input Data:"

radix -decimal

add wave x\_in c\_in

add wave -divider "Outputs:"

add wave -radix decimal c\_ar x\_ar\_out d\_out

add wave -color Blue y\_out

add wave state

######### Add stimuli data

force clk 0 0ns, 1 50ns -r 100ns

force reset 1 0ns, 0 50ns

force Load\_x 0 0ns, 1 50ns, 0 350ns, 1 460ns, 1 760ns, 1 770ns, 0 860ns

force Load\_c 0 0ns, 1 50ns, 0 300ns

radix -decimal

force L\_in 3 0ns

force x\_in 0 0ns, 1 50ns, 2 150ns, 3 250ns, 4 350ns, 0 450ns, 4 550ns

force c\_in 0 0ns, -3 50ns, -2 150ns, -1 250ns

########## Run the simulation

run 2500ns

wave zoomfull

configure wave -gridperiod 10ns

configure wave -timelineunits ns

1. Now the defining part of the do files is the same only the “signal force” part will vary thus will be furnishing that, firzolfinalv2.do

######### Add stimuli data

force clk 0 0ns, 1 50ns -r 100ns

force reset 1 0ns, 0 50ns

force Load\_x 0 0ns, 1 50ns, 0 350ns, 1 460ns, 1 760ns, 1 770ns, 0 860ns

force Load\_c 0 0ns, 1 50ns, 0 300ns, 1 1400ns, 0 1500ns

radix -decimal

force L\_in 3 0ns

force x\_in 0 0ns, 1 50ns, 2 150ns, 3 250ns, 4 350ns, 0 450ns, 4 550ns

force c\_in 0 0ns, -3 50ns, -2 150ns, -1 250ns

1. The file firzolfinalv3.do

######### Add stimuli data

force clk 0 0ns, 1 50ns -r 100ns

force reset 1 0ns, 0 50ns

force Load\_x 0 0ns, 1 50ns, 0 350ns, 1 460ns, 1 760ns, 1 770ns, 0 860ns

force Load\_c 0 0ns, 1 50ns, 0 300ns

radix -decimal

force L\_in 3 0ns, 4 1400ns

force x\_in 0 0ns, 1 50ns, 2 150ns, 3 250ns, 4 350ns, 0 450ns, 4 550ns

force c\_in 0 0ns, -3 50ns, -2 150ns, -1 250ns

1. The file firzolfinalv4.do

######### Add stimuli data

force clk 0 0ns, 1 50ns -r 100ns

force reset 1 0ns, 0 50ns

force Load\_x 0 0ns, 1 50ns, 0 350ns, 1 460ns, 1 760ns, 1 770ns, 0 860ns

force Load\_c 0 0ns, 1 50ns, 0 300ns, 1 1400ns, 0 1600ns

radix -decimal

force L\_in 3 0ns

force x\_in 0 0ns, 1 50ns, 2 150ns, 3 250ns, 4 350ns, 0 450ns, 4 550ns, 9 1550ns

force c\_in 0 0ns, -3 50ns, -2 150ns, -1 250ns, 9 1550ns

1. This is the final simulation file fizolfinal.do

####### Add stimuli data

force clk 0 0ns, 1 50ns -r 100ns

force reset 1 0ns, 0 50ns

force Load\_x 0 0ns,1 50ns,0 350ns,1 460ns,1 760ns,1 770ns,0 860ns,1 1500ns,0 1950ns

force Load\_c 0 0ns, 1 50ns, 0 300ns, 1 1500ns, 0 1950ns

radix -decimal

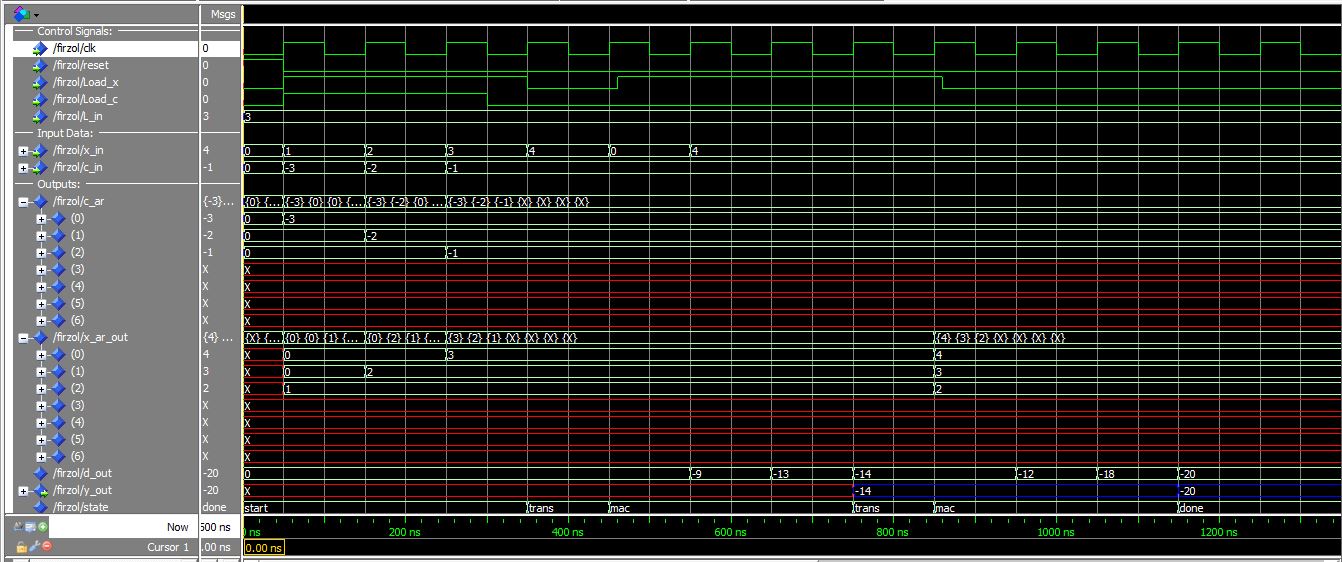
force L\_in 3 0ns, 4 1400ns

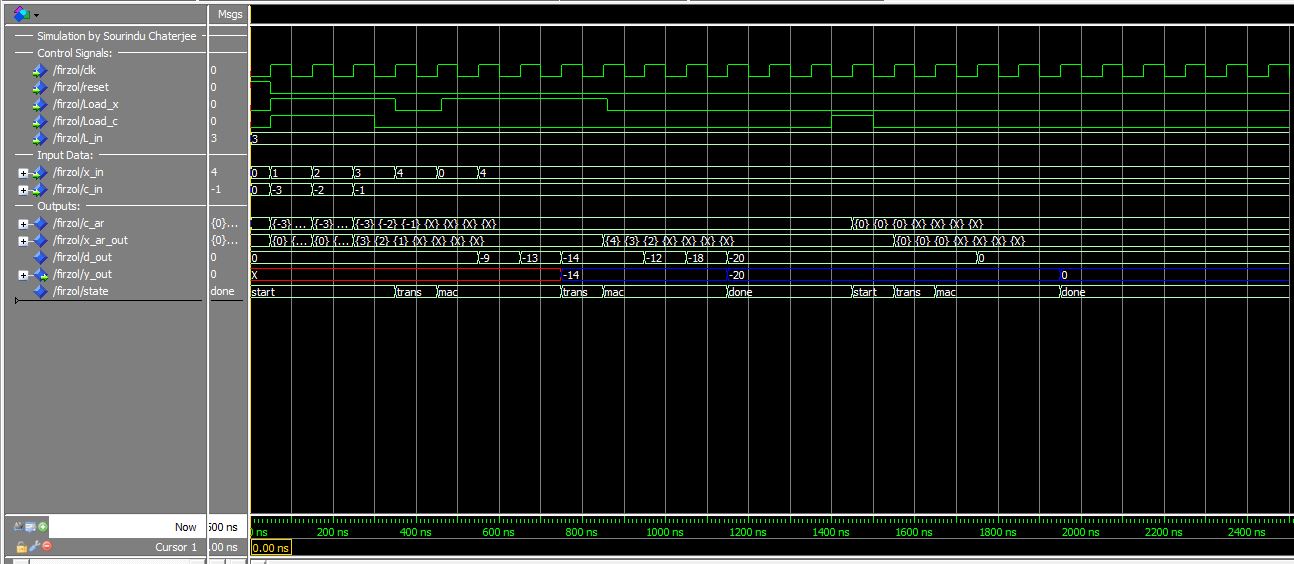
force x\_in 0 0ns, 1 50ns, 2 150ns, 3 250ns, 4 350ns, 0 450ns, 4 550ns, 1 1550ns, 0 1650ns

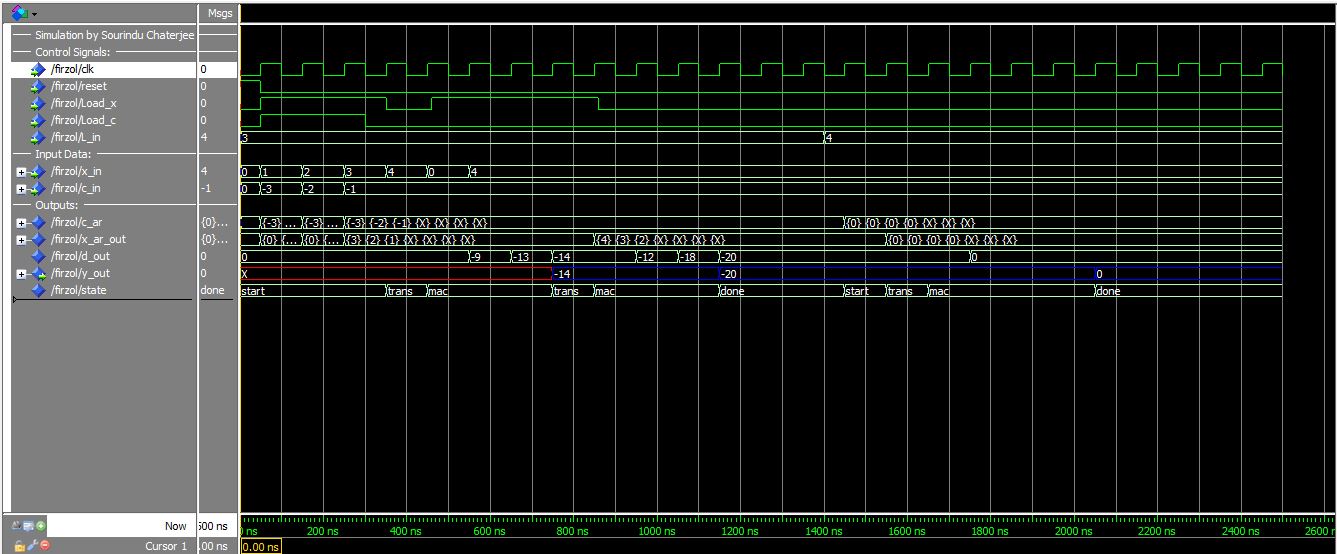
force c\_in 0 0ns, -3 50ns, -2 150ns, -1 250ns, -5 1550ns, 8 1650ns, 3 1750ns, 7 1850ns

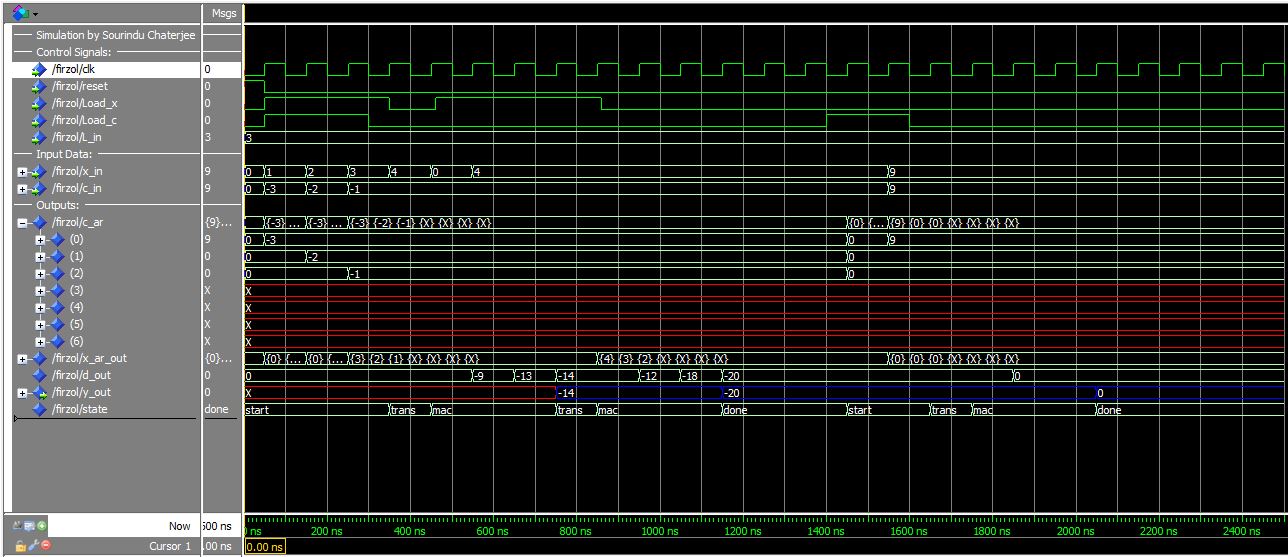
## Simulations Outputs

The outputs are in the same order as the above described segments











## TIME AND WORK SCHEDULE



QuatrusII, ModelSim and Matlab software would be used for doing the project work.

## Conclusion

This has been a challenging project and I am really grateful to my professor Dr. Meyer Baese for helping me with this endeavor. Well though ZOL implementation is successful in the MAC stage of the microprocessor execution cycle, further development to the design can be made by implementing ZOL in other stages of the design.

## REFERENCES

[1] U. Meyer-Baese: Digital Signal Processing with Field Programmable Gate Arrays (Springer, 4th edition, 2014).

[2] Zero-overhear loop controller for implementing multimedia algorithms

Nikolaos Kavvadias and Spiridon Nikolaidis,Section of Electronics and Computers, Department of Physics  
Aristotle University of Thessaloniki.

[3] Effective Exploitation of a Zero Overhead Loop Buffer

YuhongWang, David Whalleyt, Department of Computer Science, Florida State University

Gang-RyungUh, SanjayJinturkar, Chris Bums, Vincent Cao, Lucent Technologies,Allentown

[4] Application of Zero Overhead Loop and Address Generator in General Purpose Computing

Jinyung Namkoong.

[5] EEL 5722 DSP with FPGA lecture Notes.

[6] http://microchip.wikidot.com/dsp0201:zero-overhead-loops