ADIUVO
ENGINEERING AND TRAINING, LTD.

Magical Math

Adam@adiuvoengineering.com

Objective

The objective of this session are:

- Introduction to applications using maths in programmable logic
- The difference between fixed and floating point maths
- Why programmable logic is often more suited to fixed point maths
- How do we work with fixed point maths in programmable logic
- What are the rules of fixed point maths in programmable logic
- How can we implement complex algorithms and filters using fixed point maths
- The benefits of using AMD Vitis™ High Level Synthesis to implement algorithms
- Leveraging Matlab / Simulink and AMD Vitis™ Model Composer to implement math solutions.

Why We Need to Do Math in FPGA

Programmable Logic enables accelerated applications

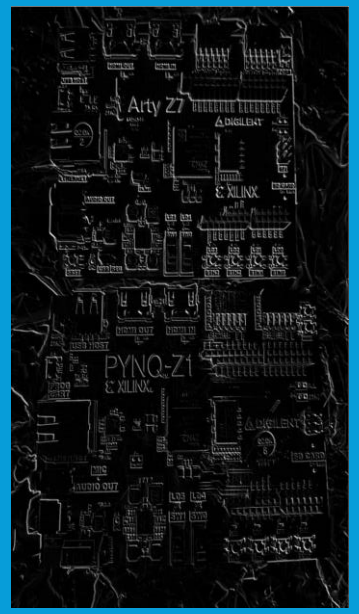
- **Image Processing** – Noise removal, edge detection, filtering
- **Radar Processing** – Signal generation, reply processing
- **Signal Processing** – Signal filtering & manipulation
- **Robotics** – End Effector positioning, Navigation
- **Control Systems** – Kalman Filtering, PID Loops
- **Motor Control** – Servo Motor, DC Motor Control

All these applications require the ability of FPGA to do maths and implement algorithms

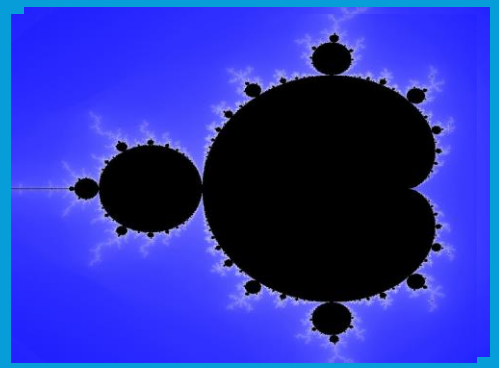
Example Applications

Background Removal & Substitution

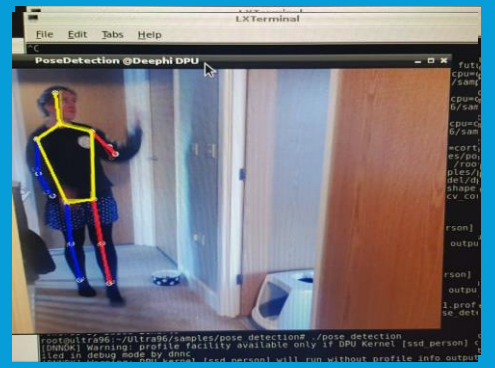
Edge Detection



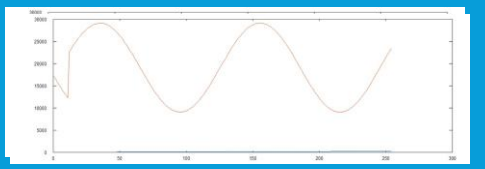
Fractals



AI/ML



Signal Processing



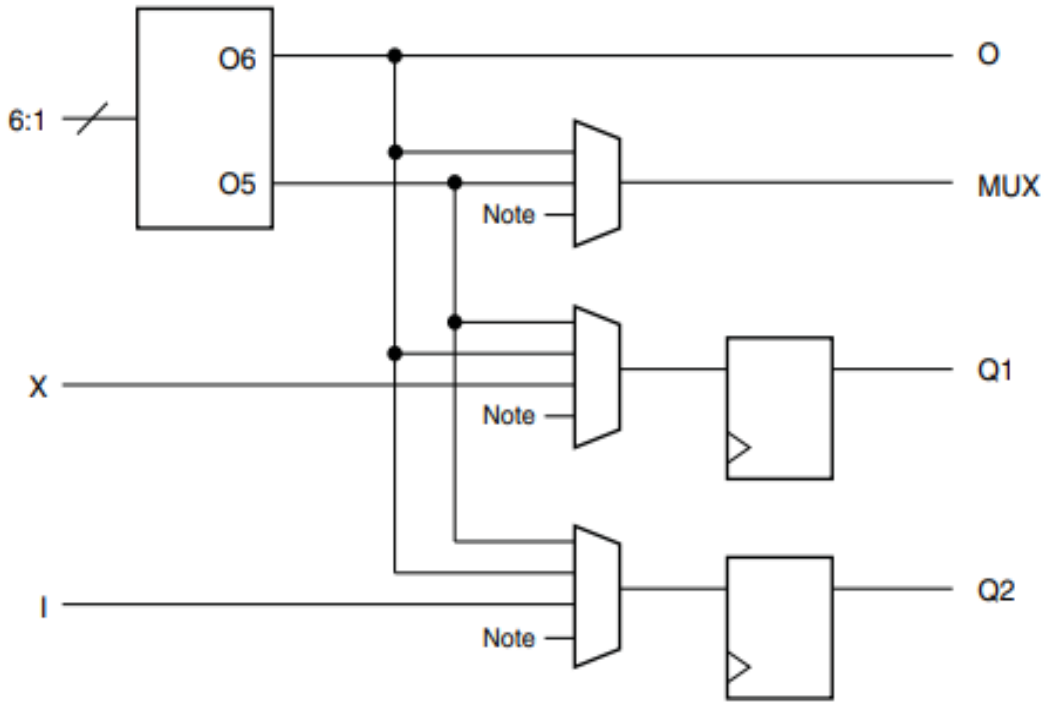
FPGA Architecture

FPGA are register and logic rich

Configurable Logic Blocks contain:

- Registers
- Look Up Table
- Distributed RAM
- Carry Mux

Logic resources are the basic building blocks of our algorithms. It is where we implement our mathematical algorithms.



DSP Elements

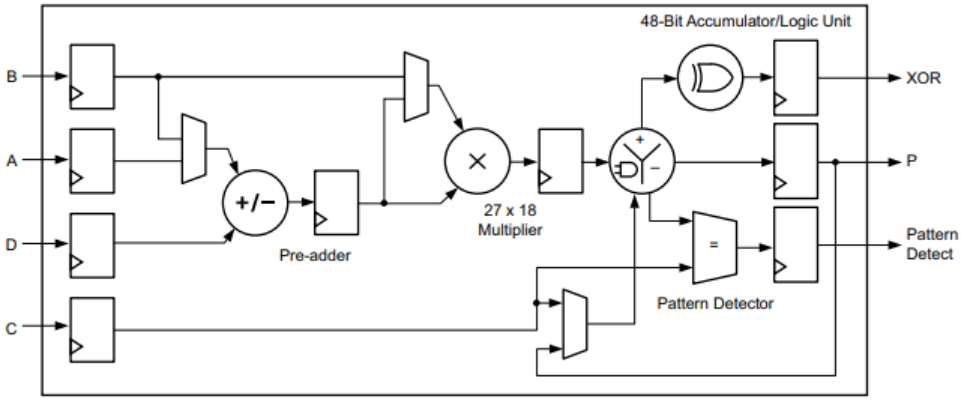
Implementing math directly in logic is costly

- Resources – increased resources
- Performance – reduced performance

Manufactures to address this include dedicated DSP elements (e.g., DSP48)

Capable of doing 48 bit Multiply accumulator

- Pre-Adder
- Multiply
- Accumulator
- Can do advance things SIMD – More later!



FPGA Maths

So far, we have looked at Logic & DSP elements, but they all have one thing in common ?

They are all ideal for the implementation of fixed-point solutions.

What is the difference between fixed point and floating-point numbers ?

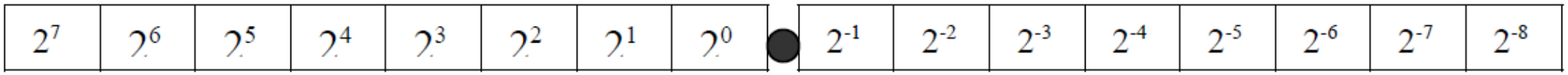
Fixed Point Number

Fixed-point representation maintains the decimal point within a fixed position allowing for straight forward arithmetic operations.

The major drawback of fixed-point representation is that to represent larger numbers or to achieve a more accurate result with fractional numbers, a larger number of bits are required.

A fixed-point number consists of two parts called the integer and fractional parts.

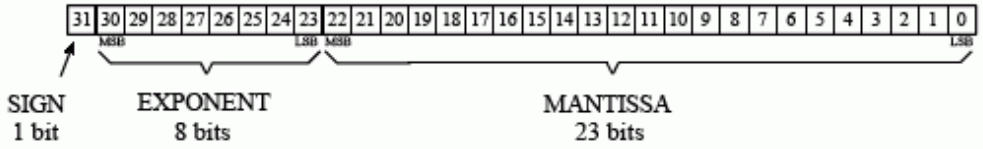
But in programmable logic Fixed Point Maths can be very fast



Floating Point Number

Floating point representation allows the decimal point to float to different places within the number depending upon the magnitude.

The floating-point number is standardized by an IEEE / ANSI Standard 754-1985 the basic IEEE floating point number



Example 1

0 00000111 11000000000000000000000

\+ 7 0.75

$$+1.75 \times 2^{(7-127)} = +1.316554 \times 10^{-36}$$

Example 2

1 10000001 01100000000000000000000

\- 129 0.375

$$-1.375 \times 2^{(129-127)} = -5.500000$$

FIGURE 4-2
Single precision floating point storage format. The 32 bits are broken into three separate parts, the sign bit, the exponent and the mantissa. Equations 4-1 and 4-2 shows how the represented number is found from these three parts. MSB and LSB refer to "most significant bit" and "least significant bit," respectively.

Why Fixed Point

Less complex to implement in logic

Enables a faster solution

Can be more power efficient

FPGA are register rich!

No standard for implementation but, Q format is popular

- Q15 – 15 fractional bits
- M,N – M integer bits and N fractional bits

Number Schemes

Fixed point numbers need to represent positive and negative numbers

- **Sign and Magnitude** - Utilises the left most bit to represent the sign of the number (0 = positive, 1 = negative) the remainder of the bits represent the magnitude. BUT! Positive and Negative Numbers.

- **Ones Complement** – Same unsigned representation for positive numbers as Sign and Magnitude representation. However, for negative numbers the inversion (ones complement) of the positive number are used. Requires end around carry for subtraction – Added complexity.

- **Twos Complement** – Positive are represented in the same manner as an unsigned numbers. While negative numbers are represented as the binary number you add to a positive number of the same magnitude to get zero

Twos Compliment

A negative twos complement number is calculated by first taking the ones complement (inversion) of the positive number and then adding one to it. The twos complement number system allows subtraction of one number form another by performing an addition of the two numbers. The range a twos complement number can represent is given by:

(2n-1) to + (2n-1 – 1)

One method we can use to convert a number to its twos complement format is to work right to left leaving the number the same until the first one is encountered, after this each bit is inverted.

Fixed Point

How many bits do we need to represent my value ?

$$\text{Integer Bits Required} = Ceil\left(\frac{LOG_{10}\text{Integer\_Maximum}}{LOG_{10}2}\right)$$

For example, the number of integer bits required to represent a value between 0.0 and 423.0

$$9 = Ceil\left(\frac{\text{LOG}_{10}423}{\text{LOG}_{10}2}\right)$$

Fixed Point

How do we work out fractional bit ? Trade off between bit length and accuracy

To store the number 1.45309806319x10-4

Multiply by 2^16 1.45309806319x10-4 * 65536 = 9.523023

Can only store 9 in the FPGA registers.

9/65536 = 1.37329101563x10-4

Significant loss of accuracy, how can we address this?

Fixed Point

We can obtain a more accurate result by scaling the number up by a factor of 2 that produces a result of between 32768 and 65535 therefore still allowing storage in a 16-bit number

268435456 * 1.45309806319x10-4 = 39006.3041205

Stored number therefore 1.45308673382x10-4 (39006/ 268435456)

Number is formatted as Q28 or 1,28

Fixed Point Rules

Fixed Point Arithmetic does have some rules which must be followed.

- **Addition** – Decimal points must be aligned
- **Subtraction** – Decimal points must be aligned
- **Division** – Decimal Points must be aligned
- **Multiplication** – Decimal points do not need to be aligned

Fixed Point Result Sizes

Operation	
A + B	Max(A'left, B'left) + 1 downto Min (A'right, B'right)
A - B	Max(A'left, B'left) + 1 downto Min (A'right, B'right)
A * B	(A'left + B'left) + 1 downto A'right + B'right
A / B - Unsigned	A'left - B'left downto (A'right + B'right) -1
A / B - Signed	(A'left - B'left) +1 downto A'right + B'right

Implementing in VHDL

Two options:

Numeric Standard (pre VHDL 2008)

- Unsigned
- Signed
- Need to keep track of decimal point – Range of number from X downto 0
- Quantisation needs to be performed by developer
- No in-built checking, requires design to check correct sizing / overflow etc

Fixed Point (VHDL 2008)

- Ufixed
- Sfixed
- Decimal point located between the 0 and -1 bit
- Inbuilt checking to ensure correct sizing of results

Fixed Package

Integer bits are represented in the range MSB down to 0

Fractional bits are represented in the range -1 down to LSB

SIGNAL example : ufixed(2 DOWNTO -3);

Which represents the vector of 000.000 allowing for a range of 0.0 to 7.875

To help initialise signals, variables and constants in our algorithm we can use the to_ufixed and to_sfixed, these can be used with integers, real, ufixed, sfixed and std_logic_vectors.

ADIUVO

ENGINEERING AND TRAINING, LTD.

Fixed Point Example

ADIUVO
ENGINEERING AND TRAINING, LTD.

Implementing Complex Functions

What about more complex math

How would I implement the following functions

- Sine / Cosine / ArcTan
- SineH / CosH / ArcTanH
- Square Root
- Exponential
- Ln

Taylor / Maclaurin Series? Look Up Table ?

But how do we achieve performance ?

CORDIC Algorithm

CORDIC (COordinate Rotation DIgital Computer) algorithm invented by Jack Volder for B58 Program

Deployed in first scientific calculator HP35

Shift and Add algorithm which can be used to implement transcendental functions.

No dedicated Multiplier required

CORDIC Algorithm

Three configurations - Linear / Hyperbolic / Circular

Each mode has two modes – Rotation / Vectoring

Enable a range of complex math's functions to be implemented.

Configuration	Rotation	Vectoring
Linear	Op Y = X * Y	Op Z = X / Y
Hyperbolic	Op X = CosH(X) Op Y = SinH(Y)	Op Z = ArcTanH
Circular	Op X = Cos(X) Op Y = Sin(Y)	Op Z = ArcTan(Y) Op X = SQR(X^2 + Y^2)

Additional Functions

Tan = Sin / Cos

TanH = Sinh / Cosh

Exponential = Sinh + Cosh

Natural Logarithm = 2 * ArcTanH note 1

SQR = $(X^2 – Y^2)^{0.5}$

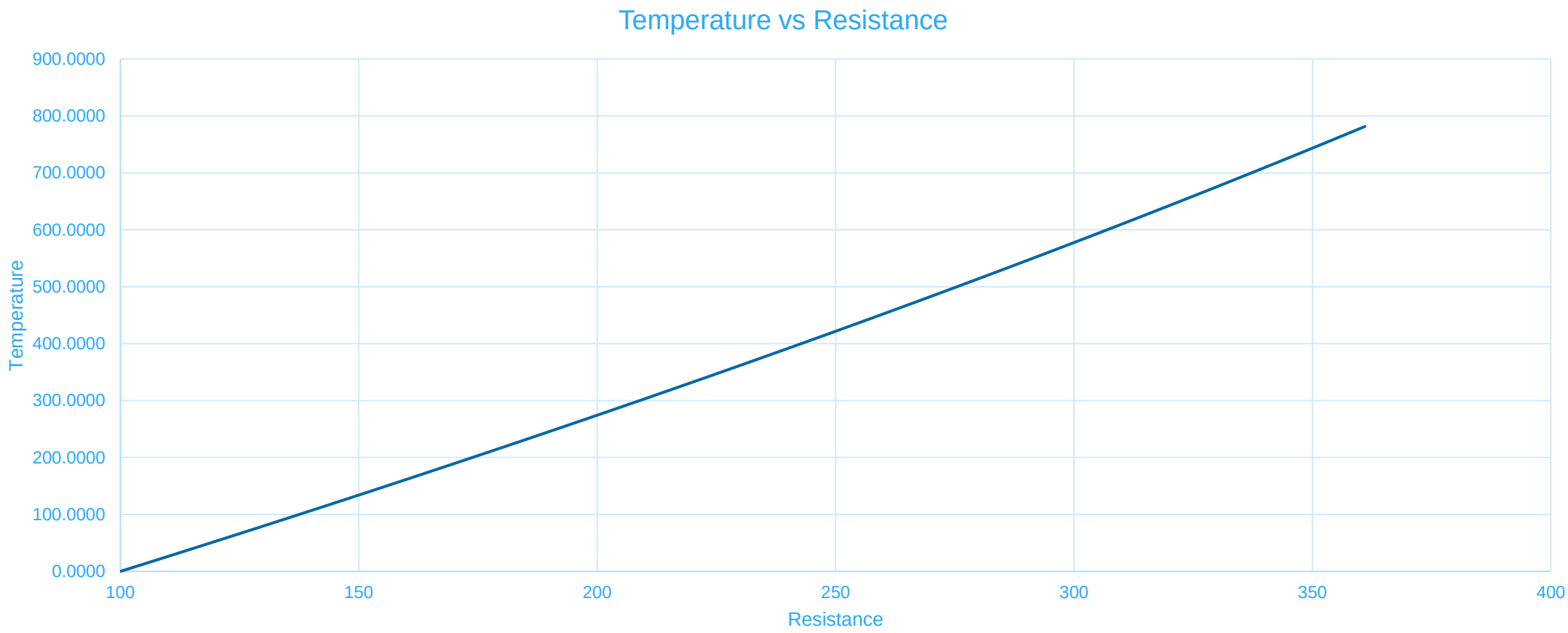
Complex Algorithm

How would you implement the following algorithm

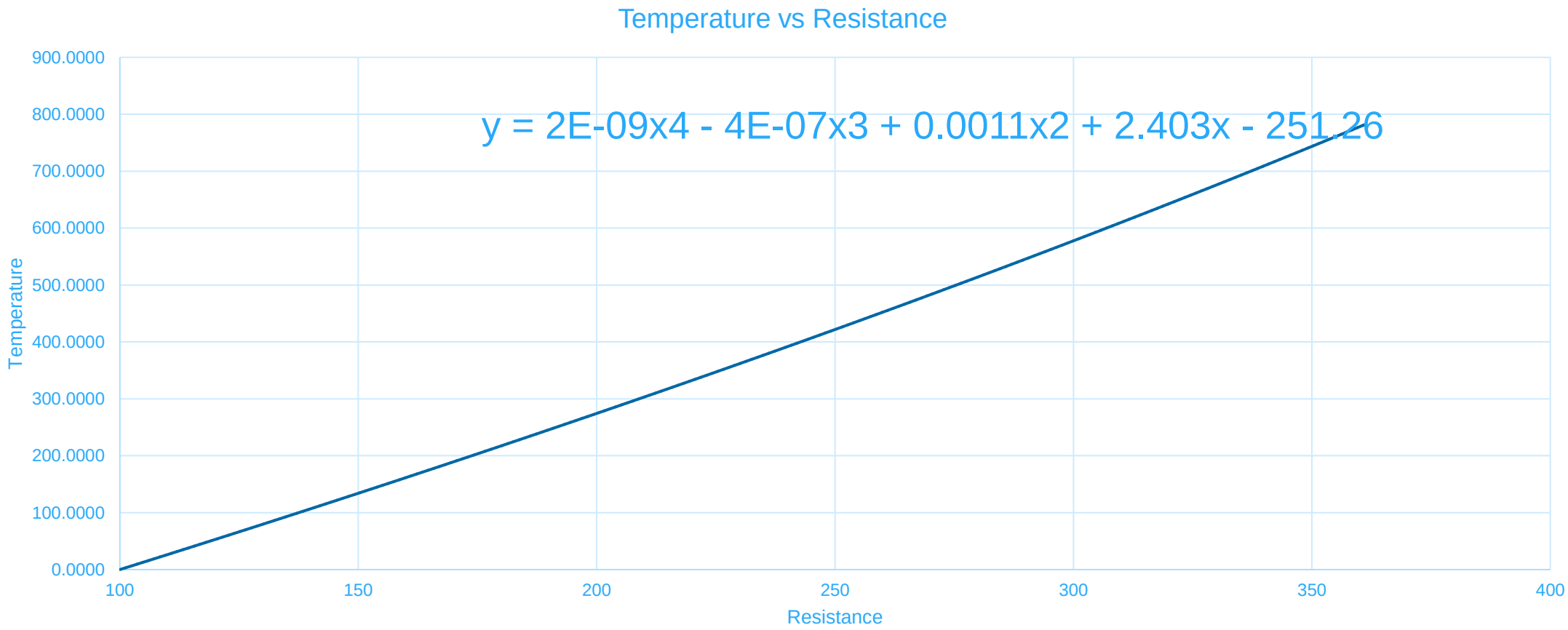
$$t = \frac{-R_0 \times a + \sqrt{R_0^2 \times a^2 - 4 \times R_0 \times b \times (R_0 - R)}}{2 \times R_0 \times b}$$

Typical Platinum Resistance Thermometer conversion equation used in industrial applications

Complex Algorithm



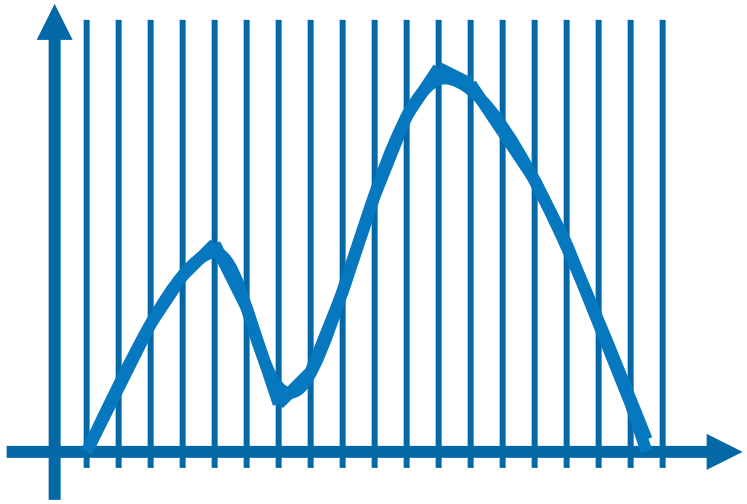
Polynomial Approximation



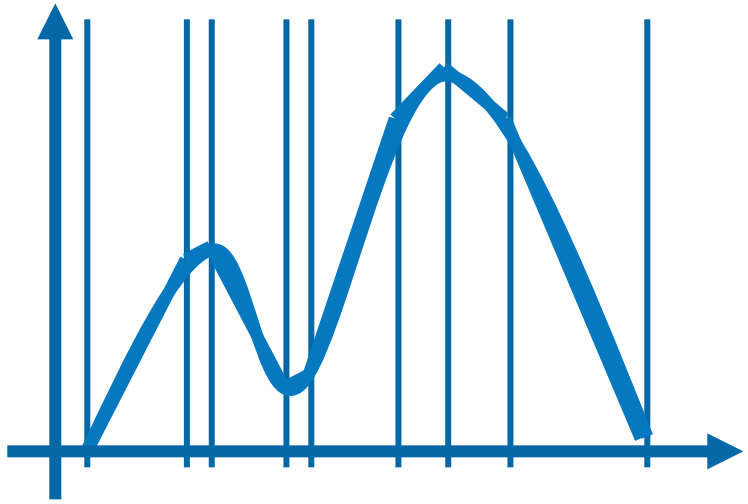
Polynomial Approximation

Leverage FPGA DSP rich environment – Addition and Multiplication easy to do in FPGA especially as we now know how!

If Accuracy is difficult with one overall polynomial equation – Segment it to several elements



Uniform segmentation



Non-uniform segmentation

ADIUVO
ENGINEERING AND TRAINING, LTD.

Complex fixed example

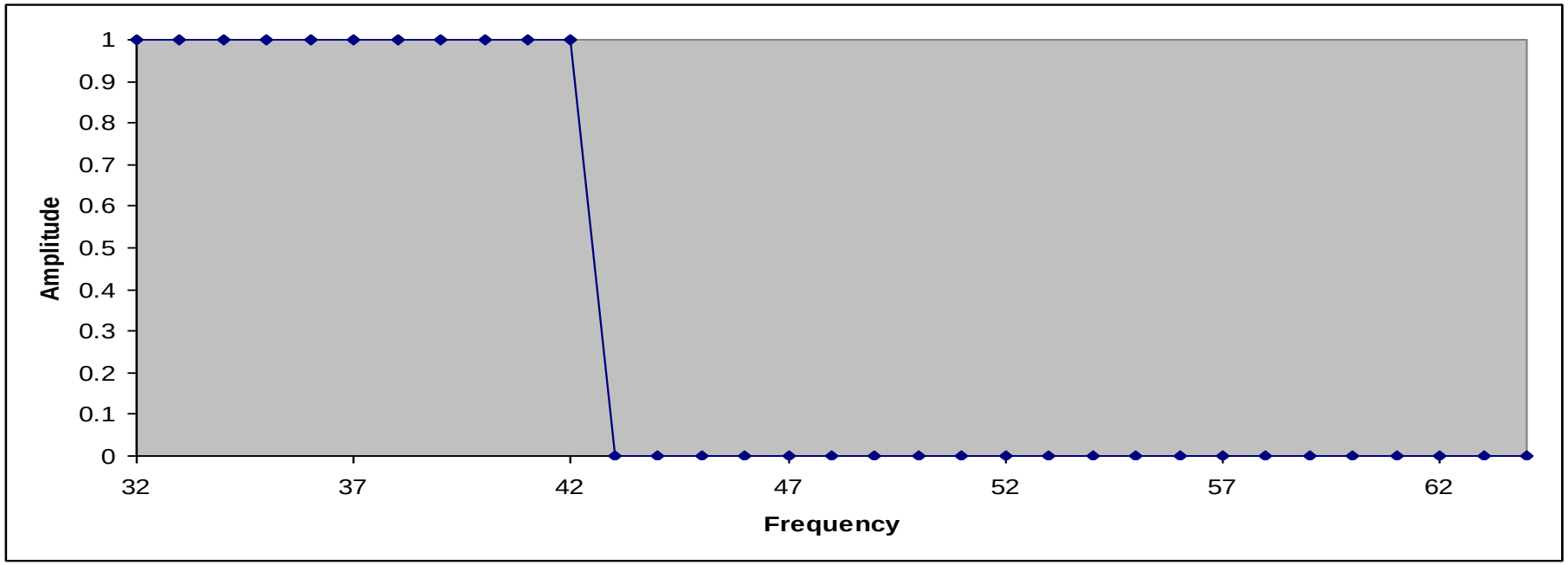
ADIUVO
ENGINEERING AND TRAINING, LTD.

Filters

What about signal processing

Finite Impulse Response Filters – Leverage the Multiple Accumulate Capability

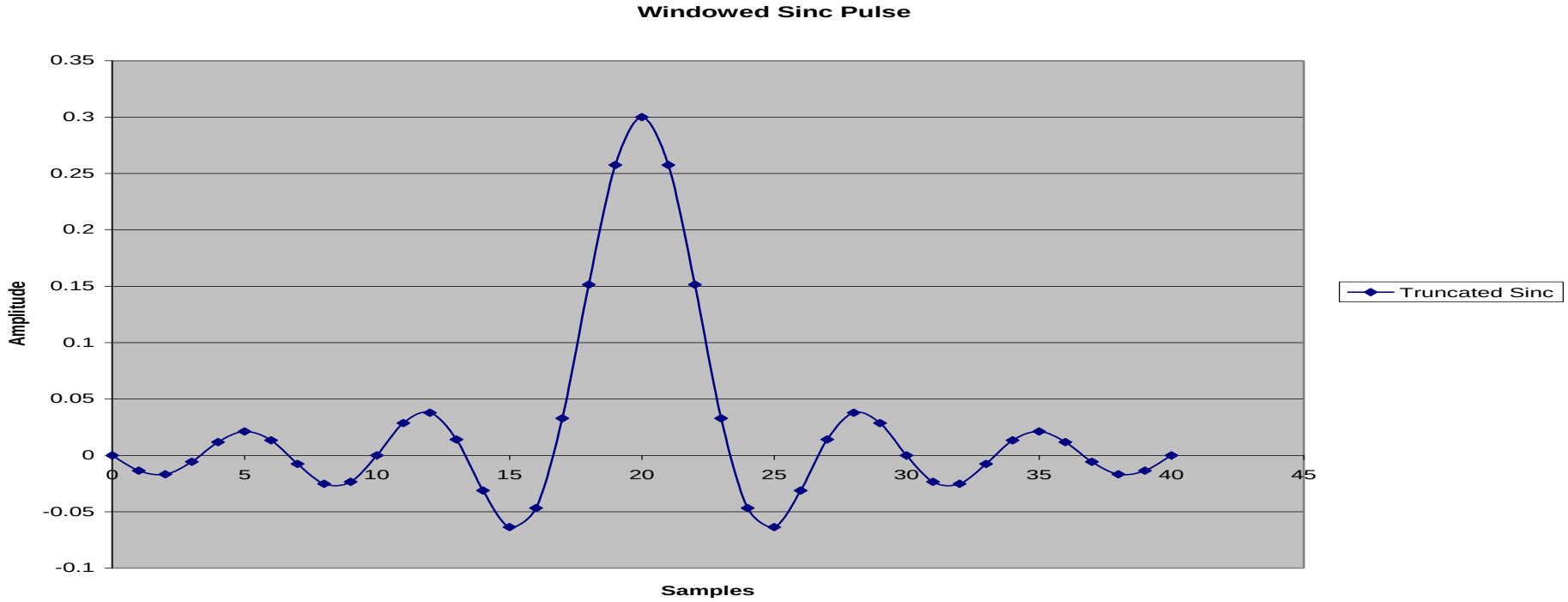
Assume an ideal filter in the frequency domain – Brick Wall



FIR Filter

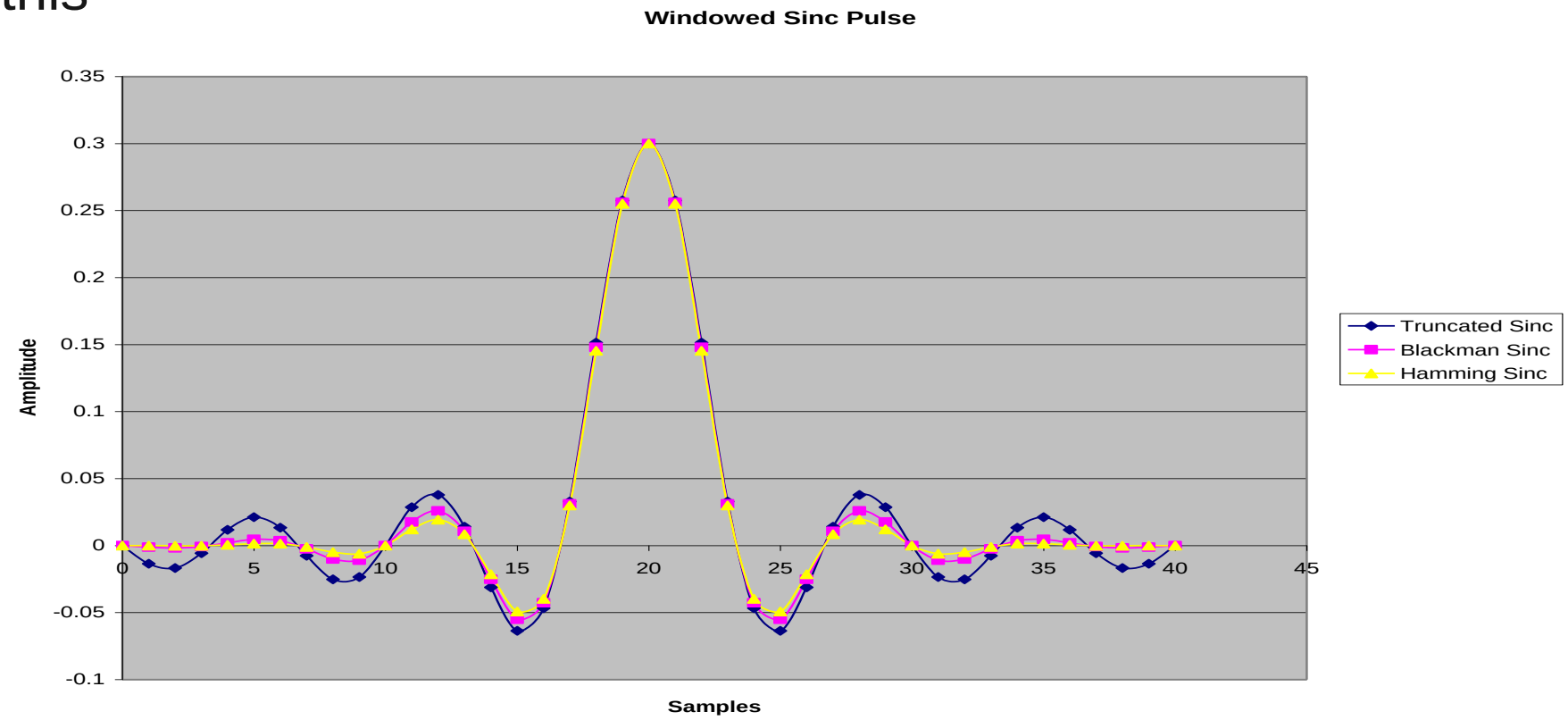
IFFT of the brick wall filter gives us the Windowed Sync Pulse

The ripples extend to infinity and never settle to zero



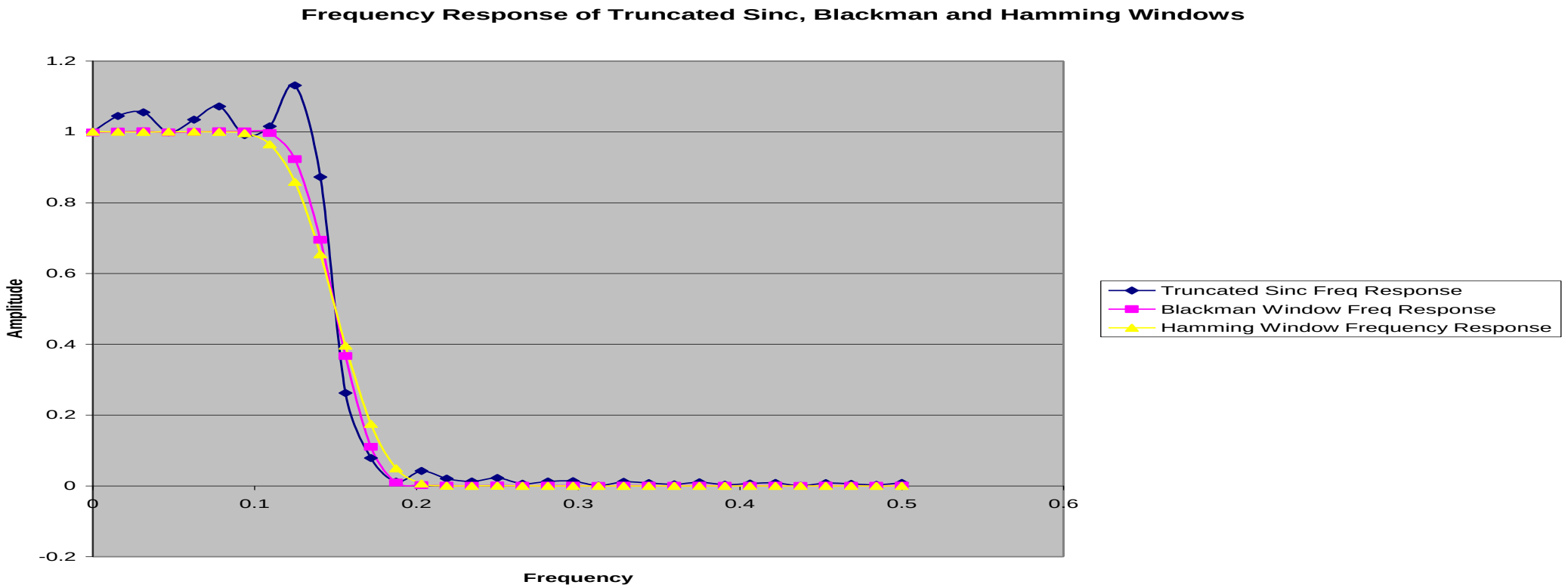
FIR Filter

Truncating the impulse response gives us ripples – Windowing helps address this

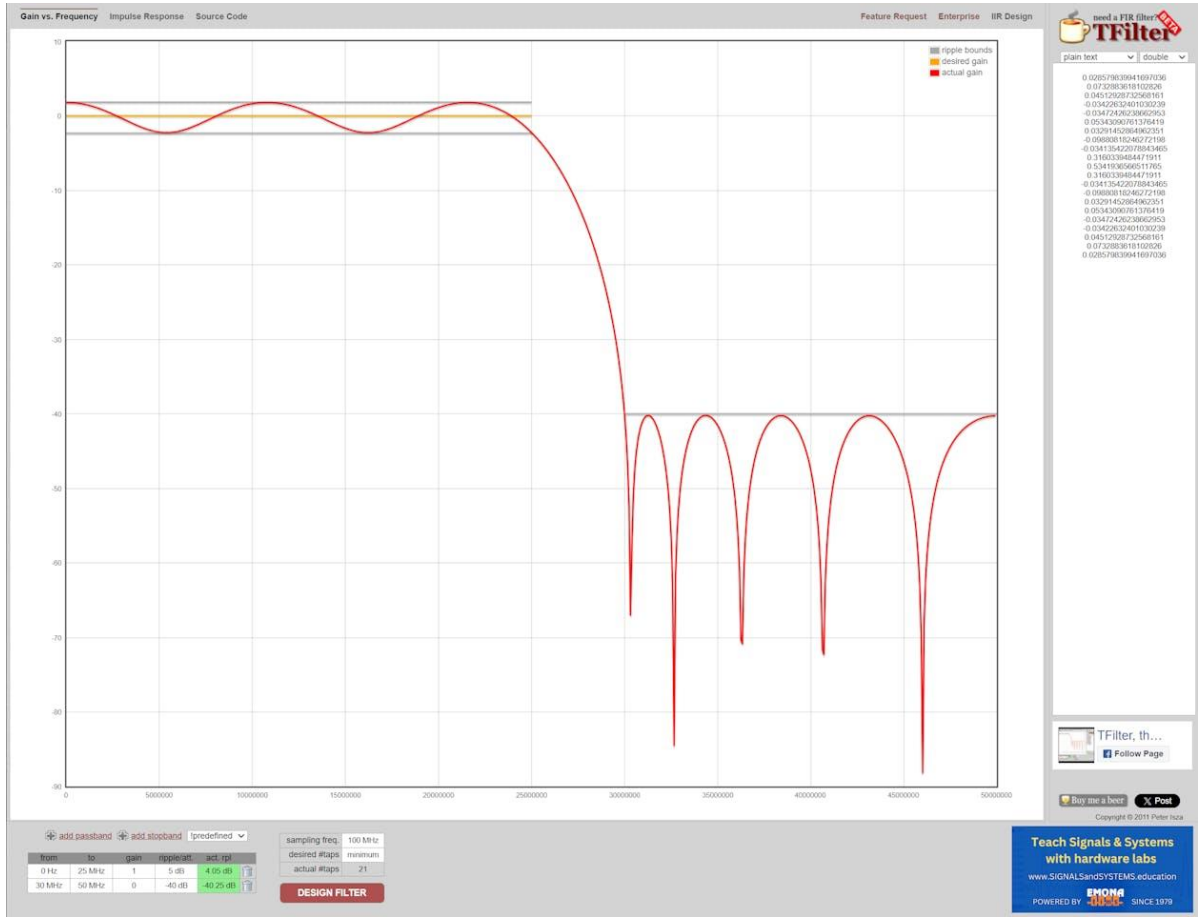


FIR Filter

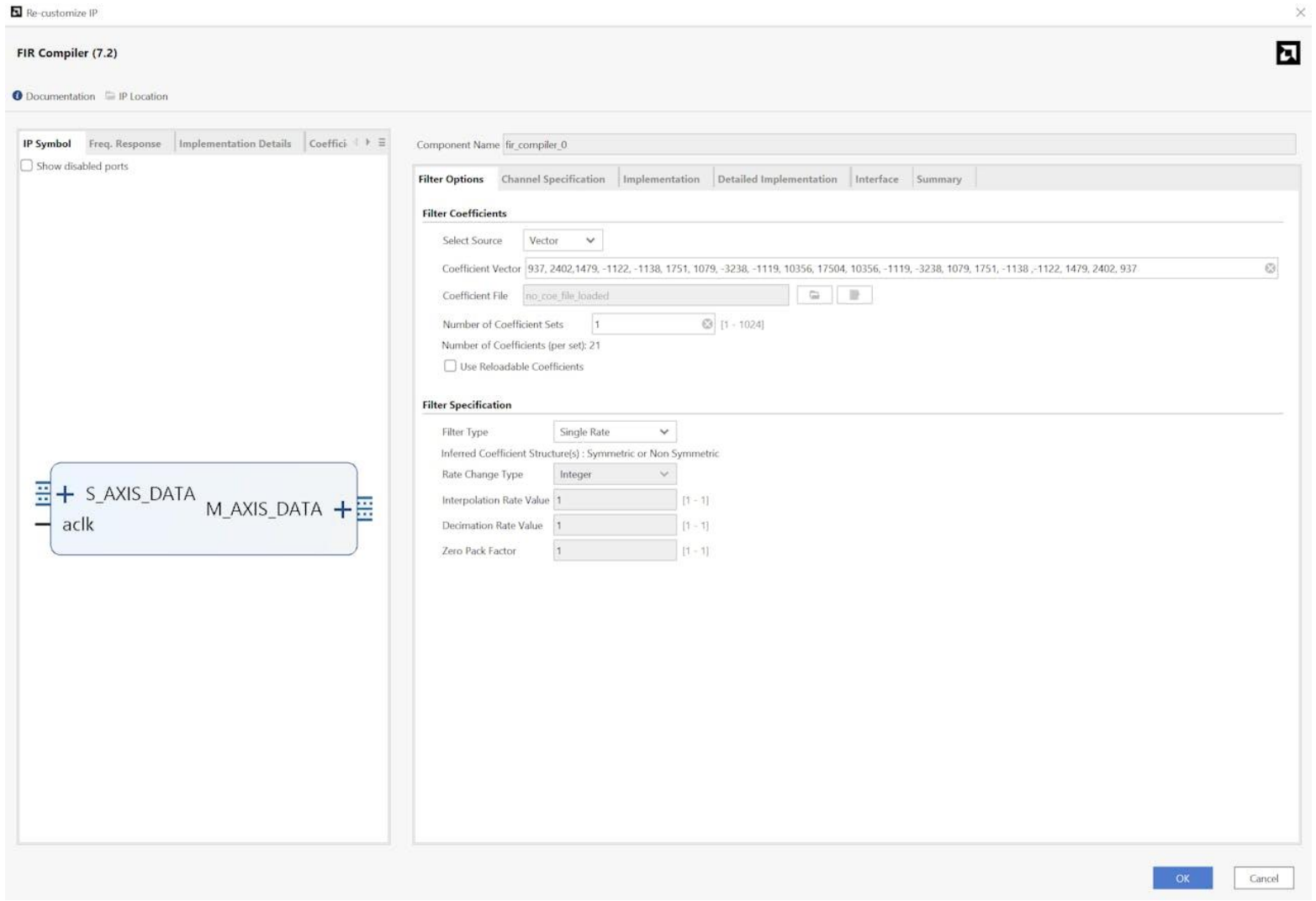
Filter Response is improved with window



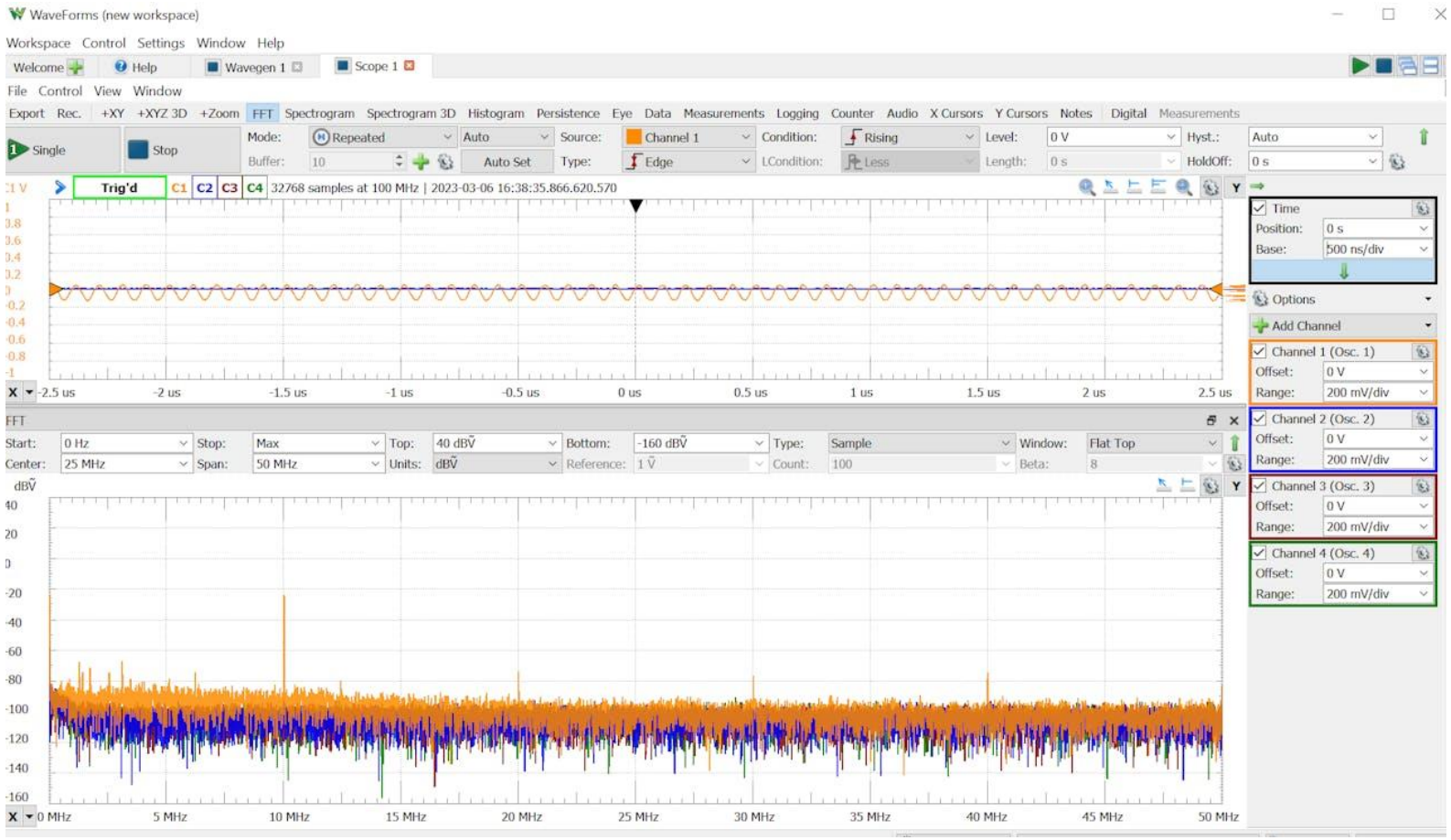
Filter Design Tools



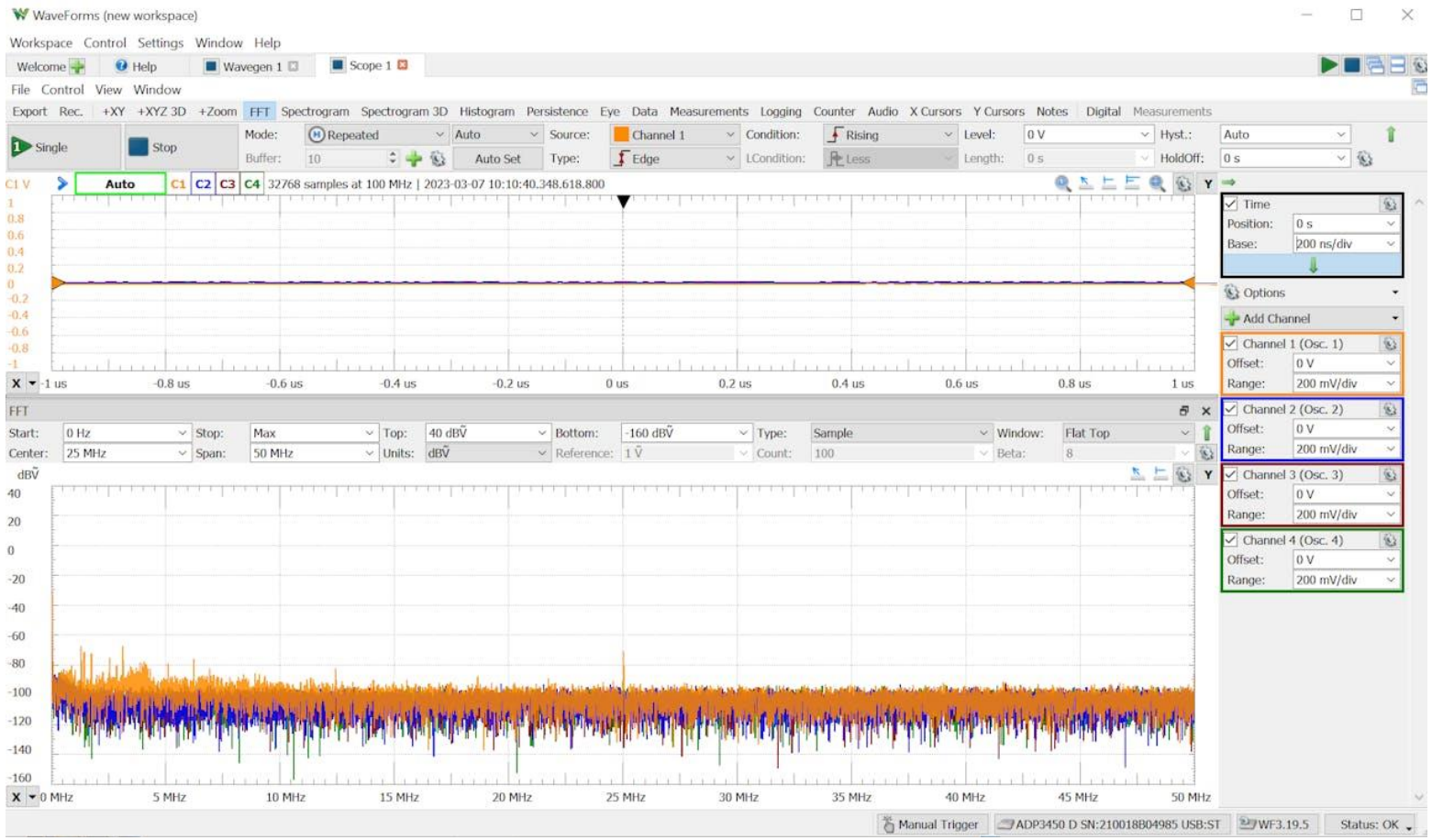
Implementing



Results 10MHz



Results 25 MHz



ADIUVO

ENGINEERING AND TRAINING, LTD.

High Level Synthesis

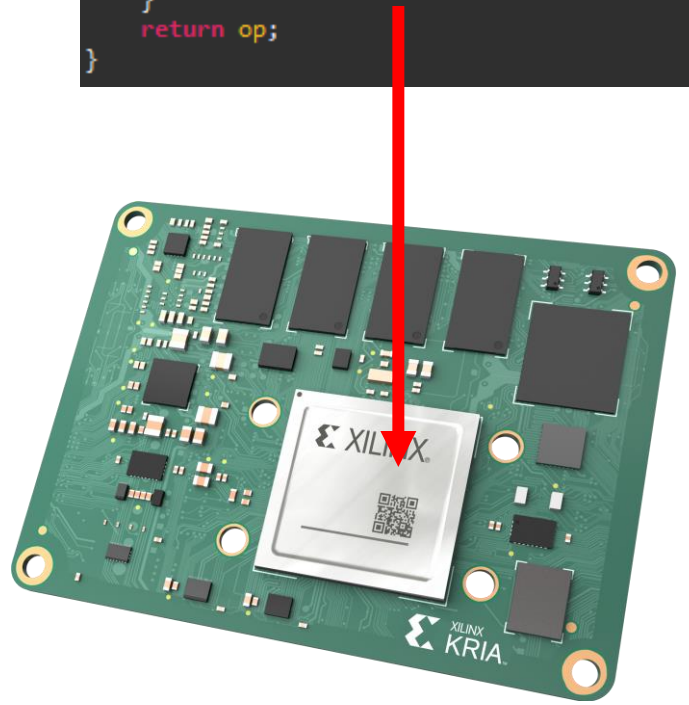
What is HLS

High Level Synthesis (HLS) enables generation of RTL modules from higher level language such as C, C++, OpenCL

Of course, SW engineers still consider these low-level languages.

HLS offers several benefits for the development of Signal / Data / Image processing algorithms

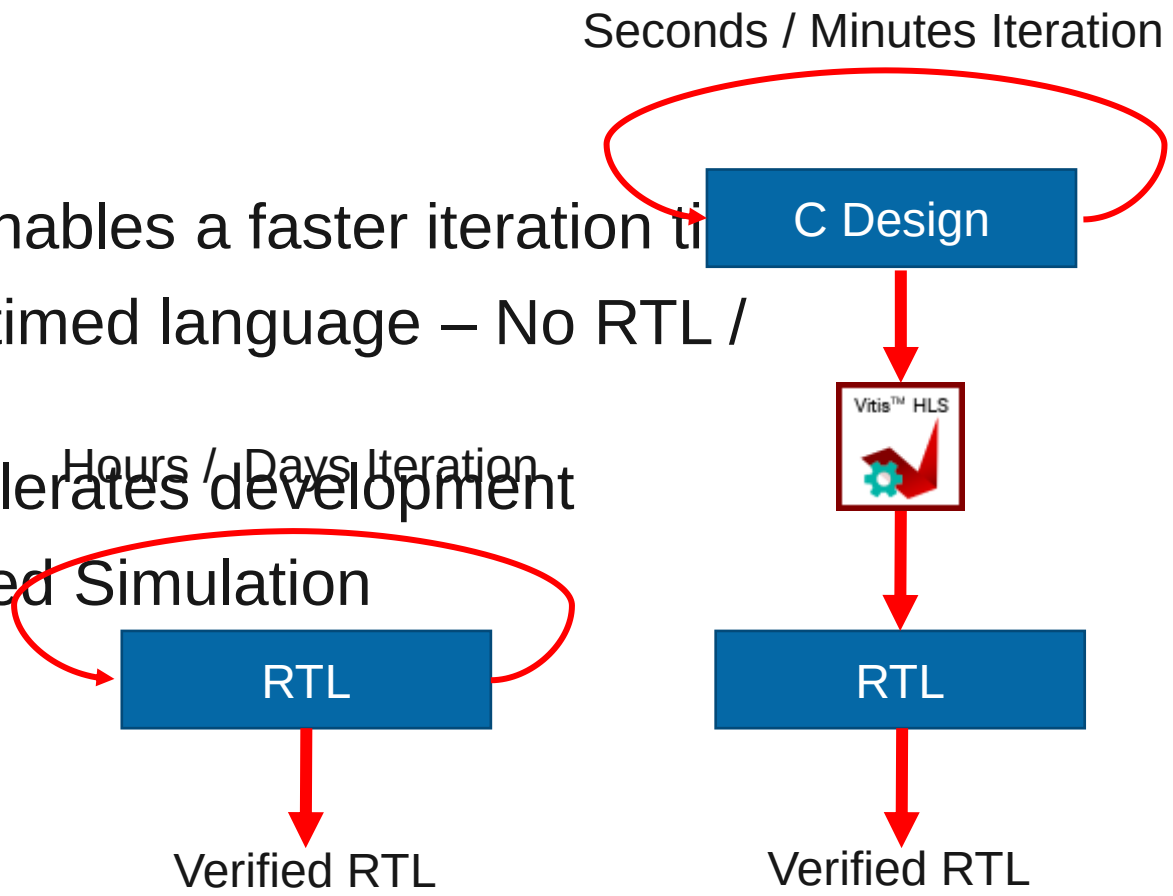
```
    error = set_point - sample;

    p = error * KP;
    i = i_prev + (error  * ts * KI);
    d = KD * ((error - error_prev) / ts);

    op = p+i+d;
    error_prev = error;
    if (op > pmax)  {
        i_prev = i_prev;
        op = pmax;
    }else{
        i_prev = i;
    }
    return op;
}
```



HLS Benefits

Developing in Higher Level language enables a faster iteration ti

- Development Time decreased as untimed language – No RTL / Behavioral level
- Increased level of abstraction – accelerates development
- Verification time is reduced as untimed Simulation

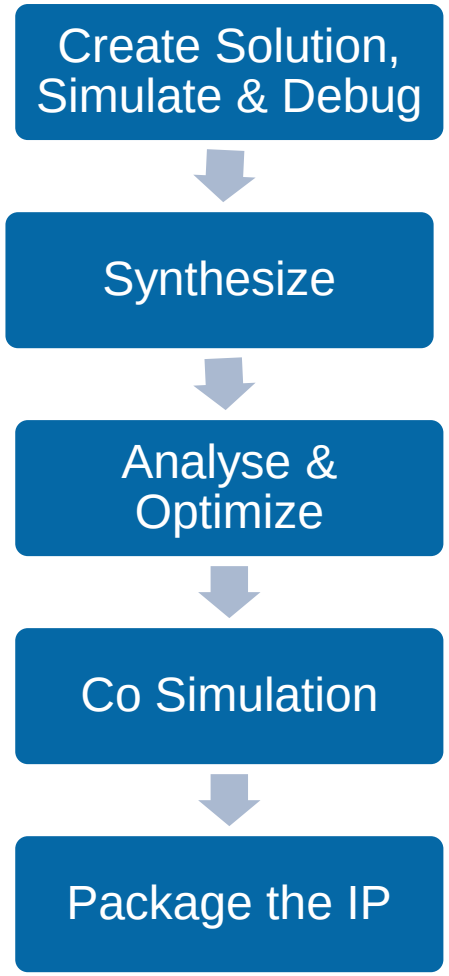


Creating HLS Solutions

Software written for CPUs and software written for FPGAs is fundamentally different

1. Not all C constructs can be synthesised
2. Learn about synthesizable C/C++ coding styles.
3. Need to focus on correct micro architecture
    1. Understand the producers and consumers
    2. Decompose the algorithm into small section which interconnect
    3. Understand throughput required for each element to achieve overall performance goals
4. Learn how to interpret the design reports

HLS Flow



Create the C Module and Test Bench – Leverage Libraries where possible
Verify algorithm performance in SW

Convert C Module to RTL

Optimize Interfaces and Performance of synthesized implementation

Simulation of the RTL module against the C Test Bench

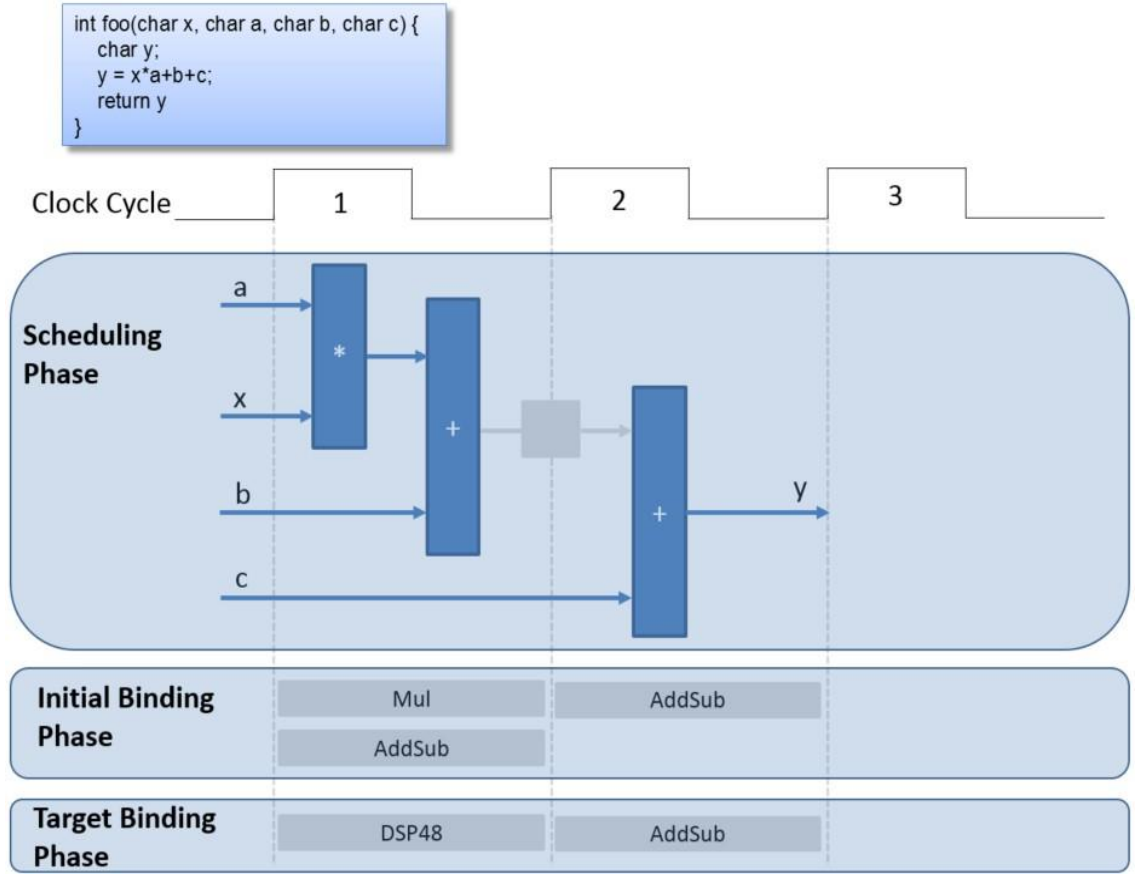
Package for use in Vitis (XO) or Vivado (Xact IP)

Untimed to Timed

Scheduling
- Determines which operations occur during each clock cycle

Binding
- Determines which hardware resource implements each scheduled operation

Control logic extraction
- Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design



HLS Math

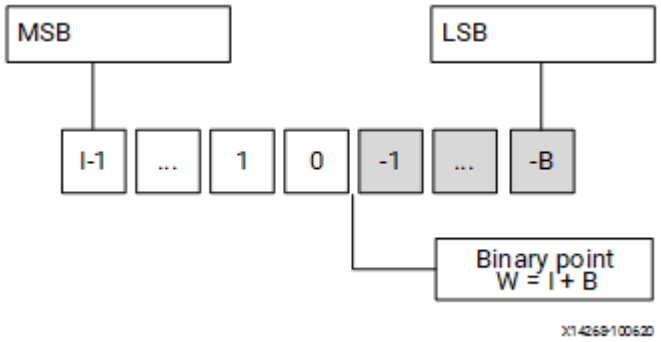
HLS is excellent for accelerating development time of algorithms.

As such there are several features we can leverage including

- Arbitrary precision data types – integer and fixed-point data types provided in ap_[u]int / ap_[u]fixed
- HLS_Math.h - library which provides functions implemented in cmath, provides floating- and fixed-point support.
- Domain specific HLS libraries including DSP, and Vision etc

HLS Math

ap_[u]fixed format



Identifier	Description	
W	Word length in bits	

I	The number of bits used to represent the integer value, that is, the number of integer bits to the *left* of the binary point, including the sign bit.
	
	When I is negative, as shown in the example below, it represents the number of *implicit* sign bits (for signed representation), or the number of *implicit* zero bits (for unsigned representation) to the *right* of the binary point. For example,
	
	`ap_fixed<2, 0> a = -0.5;  // a can be -0.5,`
	
	`ap_ufixed<1, 0> x = 0.5;  // 1-bit representation. x can be 0 or 0.5`
	`ap_ufixed<1, -1> y = 0.25;  // 1-bit representation. y can be 0 or 0.25`
	`const ap_fixed<1, -7> z = 1.0/256;  // 1-bit representation for z = 2^-8`	

Q	Quantization mode: This dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result.	
	ap_fixed Types	Description
	AP_RND	Round to plus infinity
	AP_RND_ZERO	Round to zero
	AP_RND_MIN_INF	Round to minus infinity
	AP_RND_INF	Round to infinity
	AP_RND_CONV	Convergent rounding
	AP_TRN	Truncation to minus infinity (default)
	AP_TRN_ZERO	Truncation to zero
O	Overflow mode: This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result.	
	ap_fixed Types	Description
	AP_SAT [1]	Saturation
	AP_SAT_ZERO [1]	Saturation to zero
	AP_SAT_SYM [1]	Symmetrical saturation
	AP_WRAP	Wrap around (default)
	AP_WRAP_SM	Sign magnitude wrap around
N	This defines the number of saturation bits in overflow wrap modes.	

ADIUVO

ENGINEERING AND TRAINING, LTD.

HLS Example

ADIUVO
ENGINEERING AND TRAINING, LTD.

Matlab Simulink

Matlab / Simulink Example

Example to convert voltage measured by negative Temperature Coefficient Thermistor into temperature.

Development will use Simulink HDL coder modules.

Two Stages

First determine resistance of NTC thermistor

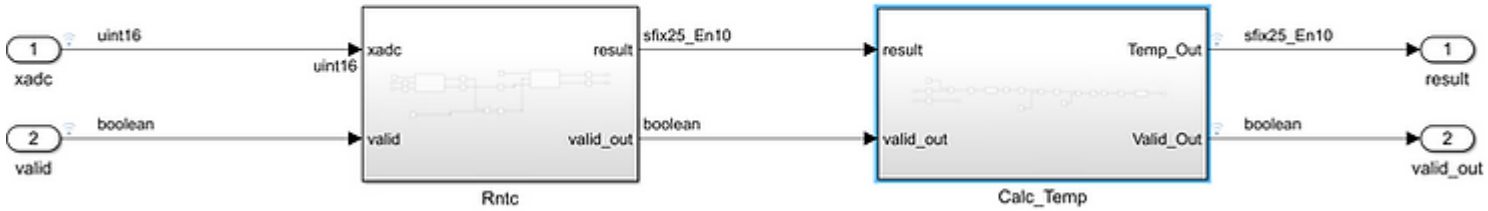
$$Rntc = \frac{Rseries}{(\frac{adc\ resolution}{adc\ value} - 1)}$$

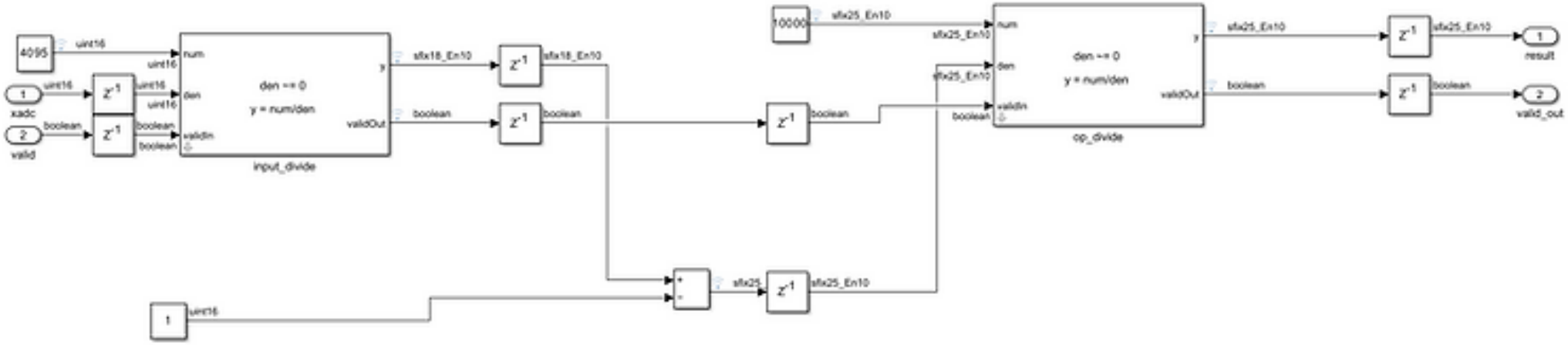
Second determine the temperature

$$\frac{1}{T} = \frac{1}{T0} + \frac{1}{Beta} * \ln \frac{Rntc}{Rnom}$$

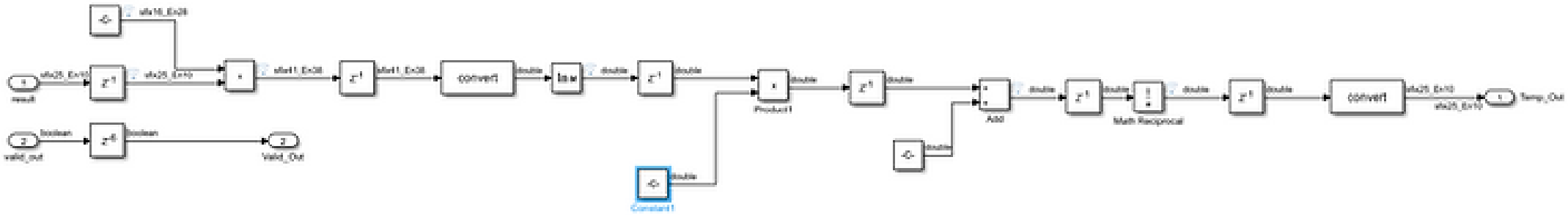
Matlab / Simulink Example

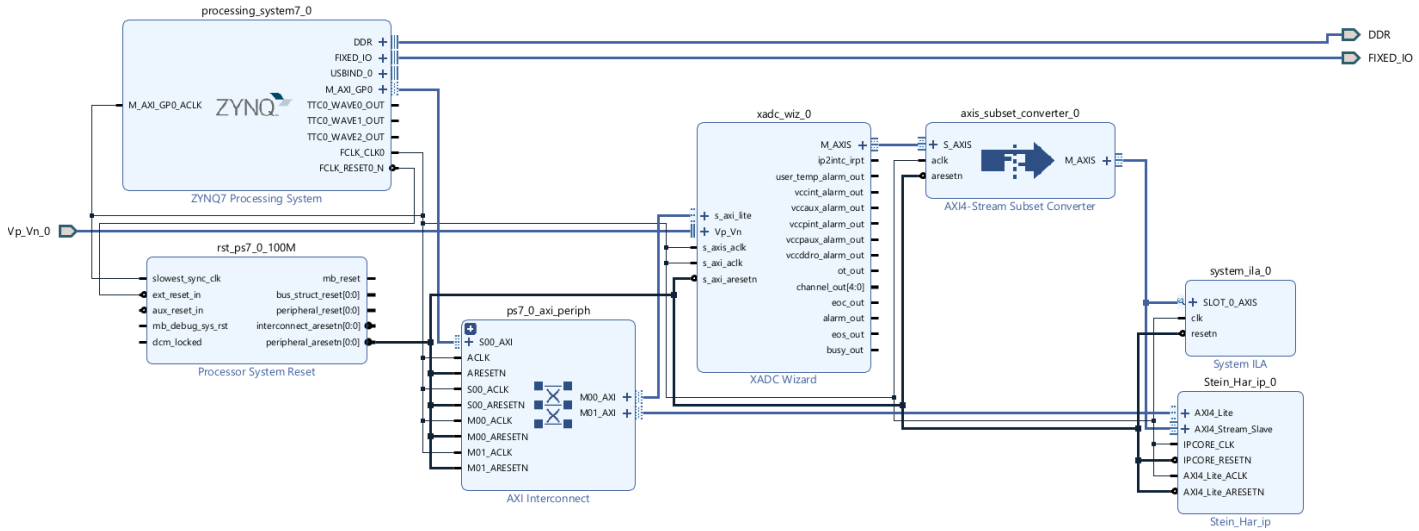
Create two modules as per the stages outlined previously



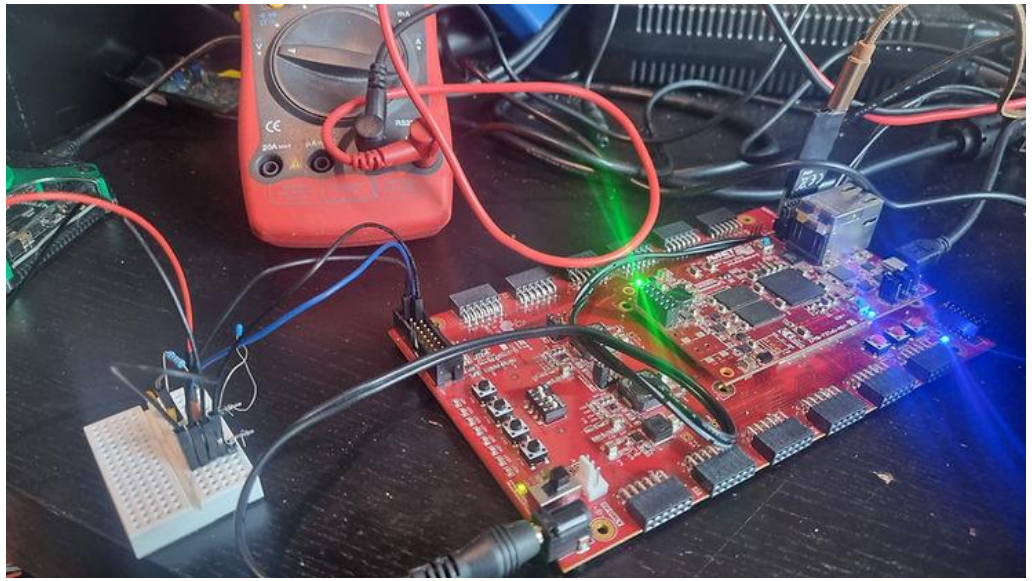


Matlab / Simulink Example

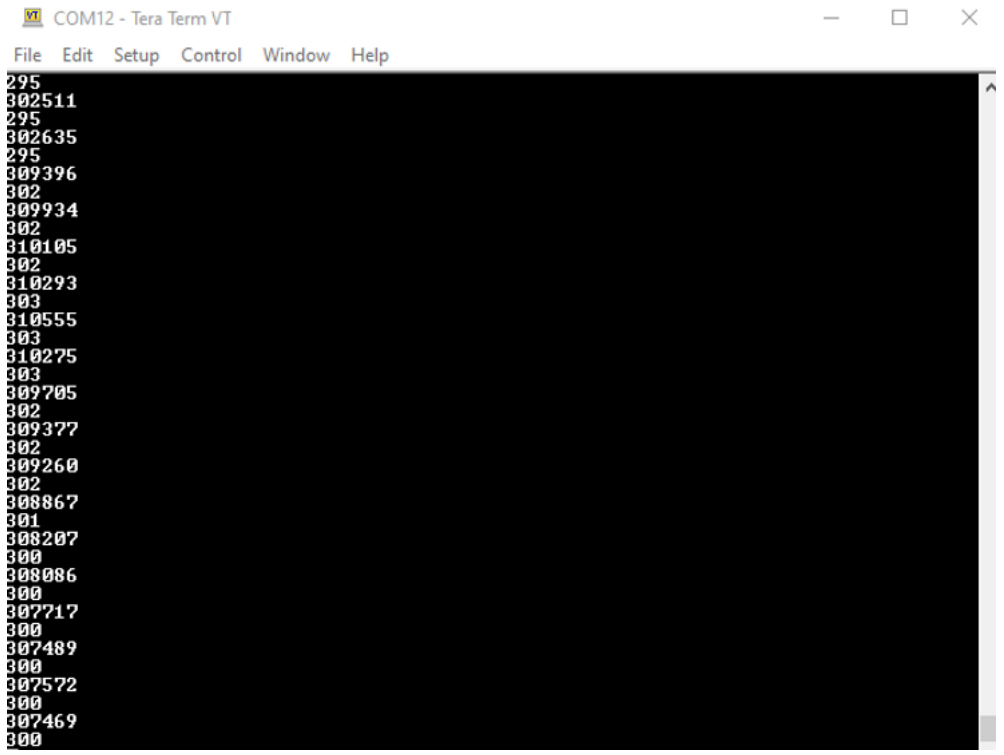




Matlab / Simulink Example



Hardware testing with NTC shows the functionality is as required the temperature is correctly shown in Kelvin



Summary

- Implementing Math functions and algorithms in FPGA is not as challenging as it might appear.

- There are several different ways to implement algorirhtms as presented
  - HDL
  - HLS
  - Matlab / Simulink

- Each has their own pros/cons – HLS is excellent for rapid acceleration of algorithm development and aligns often with C based models

ADIUVO
ENGINEERING AND TRAINING, LTD.

Questions

ADIUVO

ENGINEERING AND TRAINING, LTD.

www.adiuvoengineering.com

info@adiuvoengineering.com