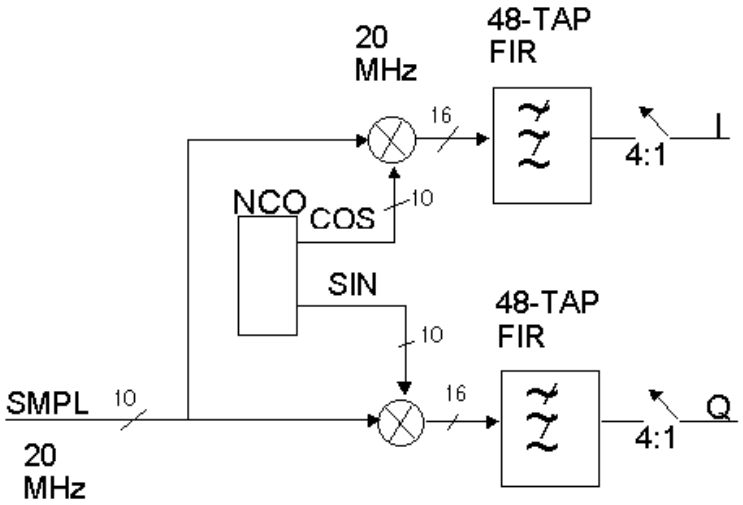
IMPLEMENTING a

DEMODULATOR

using the

XILINX CORE GENERATOR

Design Example: Quadrature Demod



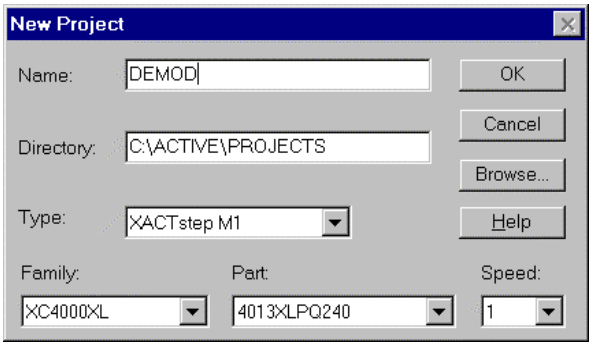
This demodulator consists of a channel demodulated with a Sine and Cosine function.
- The frequency is brought down mixing the input signal with a local oscillators (sin and cos) that are running at a frequency close to the channel.
The output of the mixer (multiply with a lower frequency sin wave) gives the sum and the difference of those 2 signals.
Ex: If the input we are interested in is 20MHz and the local oscillator runs at 21MHz, the output of the mixer will contain 1MHz and 41MHz.

- Since we are interested in the lowest frequency (1MHz), a low pass filter is used to eliminate the highest frequency. A high frequency means a high sample rate and processing high sample rates results in a lot of logic used. The low pass filter eliminates the high frequency (41MHz), making it possible to process the data at a lower sample rate. Lowering the sample rate is called decimation. In our case we decimate in the act of filtering - (this is represented by the 48-TAP 4:1 Decimate Filter).

Getting Started

The goal of this Lab is to implement a Demodulator using the Xilinx Core Generator in conjunction with Foundation.

You will first need to create a new Foundation Project:



You will then have to open the Xilinx Core Generator and set the Options as follows:

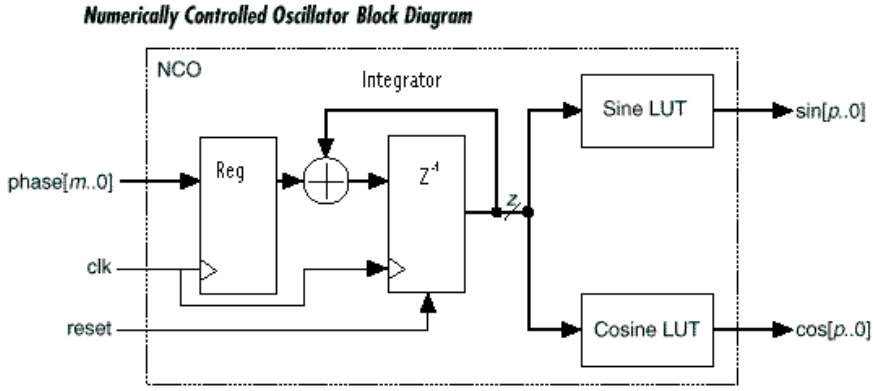
- Under Options -> Output Format, select XNF implementation Netlist and Foundation Schematic Symbol

  *Note1: Once you click on OK, a Warning will come up asking you to open the Foundation Schematic Editor. This is due to a bug with the symbol generator (*NET2SYM*) which will not generate the symbol unless the schematic editor is opened or iconized.*

  Note2: The symbol generator **does not work on Windows NT** *but will be fixed in F1.4. In the meantime, you will have to import the netlist from the schematic editor…*

- Under Options -> Systems Option…, update the Project Path to your design entry project Path (i.e. c:\active\proejcts\demod).
  All the other Path are automatically updated during the install so you do not have to modify them.

Lab 1: Numerically Controlled Oscillator (NCO) Implementation



Functional Description

The NCO function contains sine and cosine look-up tables (LUTs) that perform the following functions:

$\text{Sin}(n) = \sin(2\pi n/\text{N})$
$\text{Cos}(n) = \cos(2\pi n/\text{N})$

Where: n = Address input to the LUT
N = Number of samples in the LUT
$\text{Sin}(n)$ = Amplitude of sine wave at $(2\pi n/\text{N})$
$\text{Cos}(n)$ = Amplitude of cosine wave at $(2\pi n/\text{N})$

Incrementing n from 0 to N-1 causes the LUT to output one complete cycle of amplitude values for the sine and cosine functions. The value $2\pi n/\text{N}$ represents a fractional phase angle between 0 and 2π. The time (t) required to increment n from 0 to N-1, is the period of the sine and cosine waveforms produced by the NCO. The LUT address increments once each system clock cycle by an amount equal to the phase input. The LUT address, or phase angle, is accumulated and stored in the integrator (also called phase accumulator register). The register's output is used to address the sine and cosine LUTs.

The frequency (f) of the system clock (f_{CLK}) is fixed. Therefore, the frequency of the sine and cosine waves is:

$$F = 1/t = f_{CLK} * \text{phase[m..0]} / 2^m$$

Modules Generation

The parameters are the following:

- 12-bit Input phase,
- 20-bit Accumulator,
- 64 samples sin/cos table
- 10-bit Outputs

1- Generate the Input Phase Register

From Coregen, select the REGISTER module under LogiCORE -> Basic Elements. Double click on the Icon (left side of the module name) and enter a *Component Name* and a *Data Width* of 12.
Generate.

2- Generate the Integrator

From Coregen, select the INTEGRATOR module under LogiCORE -> Math. Double click on the Icon and enter a *Component Name*, *Input Data Width* of 12 and *Output Data Width* of 20.
Generate.

3- Generate the SIN/COS Table

From Coregen, select the Sine-Cosine Look-Up Table module under LogiCORE -> DSP -> Building Blocks. Click on the **Spec** Icon (Green Icon located on the top left corner) to bring up the Sine-Cosine Look-Up Table data sheet and answer the following questions:

Q1) Are all the values that represent the Sine Wave Stored in the ROM Look-Up Table? If not, how much of it is stored?

Q2) How many CLBs is a 64 words by 10 bits Sine Table going to use?

Q3) How many clock Latency?

Q4) Do you need to generate two separate modules for the Sine Table and the Cosine Table used in the NCO?

Q5) What Sine Wave are you getting if the Ctrl signal is:

ctrl=1 :
ctrl=0 :

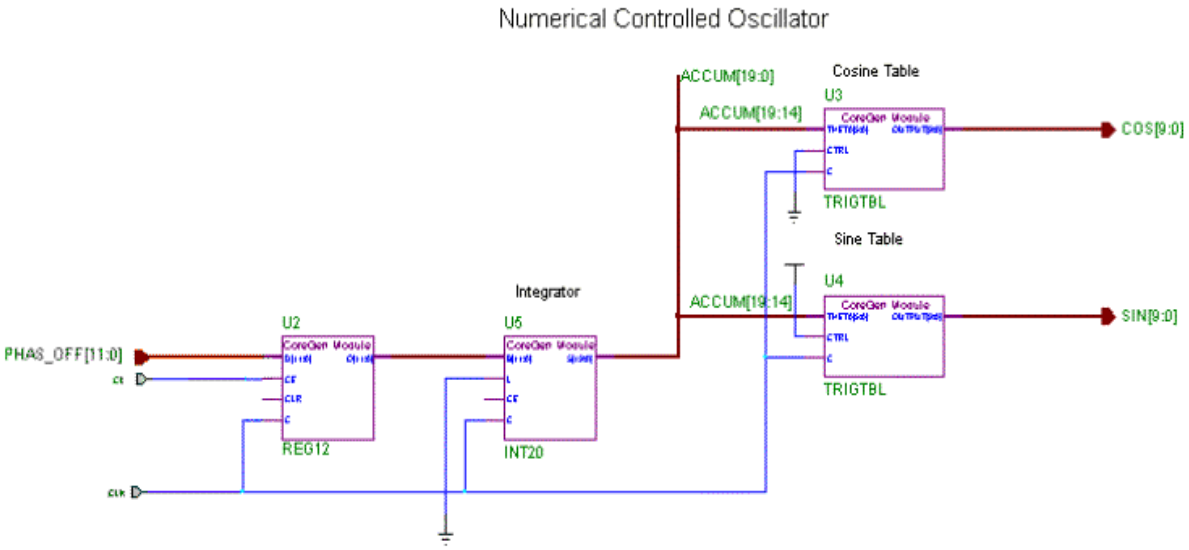
Bring up the SIN & COS Look-Up Table GUI and enter a *Component Name*, *Address Width* of 6 and *Output Data Width* of 10.
Generate.

Design Implementation

Go into the Foundation Schematic editor and select *Update Libraries* from the File menu. This will add the three new cores that you just generated into the Symbols Toolbox.

Reminder: If you are on Windows NT, the automatic symbol generation doesn't work so you need to select "Import Netlis"t from the Option Menu and import each Netlist. The symbol will be generated with Pins instead of Busses so you should edit the symbol and modify it.

Connect the 4 Cores as shown below. Note that the upper 6 bits of the output of the integrator are driving the Sine-Cosine Table.



Save this sheet as NCO and under *Hierarchy*, select *Create Macro symbol from Current Sheet* in order to generate a symbol.

Simulation

The goal is to run two simulations with a different phase_offset value and compare the amplitude values that you get on the Sine and Cosine outputs. Put some Probes on the Phas_off[11:0], CE and CLK inputs as well as on the THETA inputs of each Trigonometric Tables and their SIN[9:0], COS[9:0] outputs.

S1) Load a phase_offset of 100 (hex) and clock the NCO until you start seeing some Sine and Cosine amplitude values. It is necessary to clock the NCO until the output of the Integrator is high enough to start driving the Theta input of the Sin/Cos table. Look at the Sin and Cos LUT outputs and save the 8 first values on the table below. The 8th value should be the same for COS and SIN, can you explain why?

S2) Now, rerun the simulation doubling the phase_offset value (load 200 (hex)). Compare the Sin and Cos outputs with the previous simulation and you should see that only every other amplitude values are now displayed. Can you explain why?

You are Done with Lab 1.

Lab 2: Multipliers

This Lab will give you a chance to generate 3 types of multipliers. The first one comes straight out of Coregen since it meets the requirements we have to implement the Demodulator. The second one has nothing to do with the Demodulator design but will show you how you can speed up the Coregen multiplier by combining several of them together. Finally, the third example will show you how to implement a Complex multiplier.

1- 10x10 Multiplier

This multiplier will be used in the top level Demodulator design so you do not have to change the Foundation nor the Coregen settings at this point.
From Coregen, go under LogiCORE -> Math -> Multipliers. Three types of multipliers are currently available: Constant Coefficient Multipliers, Speed-Optimized Multipliers and Area Optimized Multipliers.

Using the Data sheets available for this modules, answer the following questions:

Q1) 8x8 Non-Pipelined KCM

Number of CLB used?

Q2) 8x8 Pipelined KCM

Number of CLB used?
Number of Clock Latency?

Q3) 8x8 Speed Optimized Multiplier

Number of CLB used?
Number of clock Latency?
Performance in a –1 part?

Q4) 8x8 Area Optimized Multiplier

Number of CLB used?
Number of clock Latency?
Performance in a –1 part?

Q5) Compare the performance and the CLB count of a 8x16 and a 16x8 area optimized multiplier.

Which one is faster?
Which one is bigger?

Q6) All the cores generated with Coregen are relatively placed. All the variable multipliers (NxM) are built to fit in a rectangular or square matrix of N rows by M (or M-1) columns.

What is the shape of a 8x16 multiplier?
What is the shape of a 16x8 multiplier?

Conclusion: The Short and Fat multiplier runs faster but is bigger than the Tall and Thin one. This should be taken in consideration when entering the A and B width of the multiplier.

It is now time to **Generate** the 10x10 multiplier for the Demodulator design.

2- Speed up the Area Optimized Multiplier

Requirements: **14x14 signed** multiplier running at **75MHz** in a XC4000E-1 part.

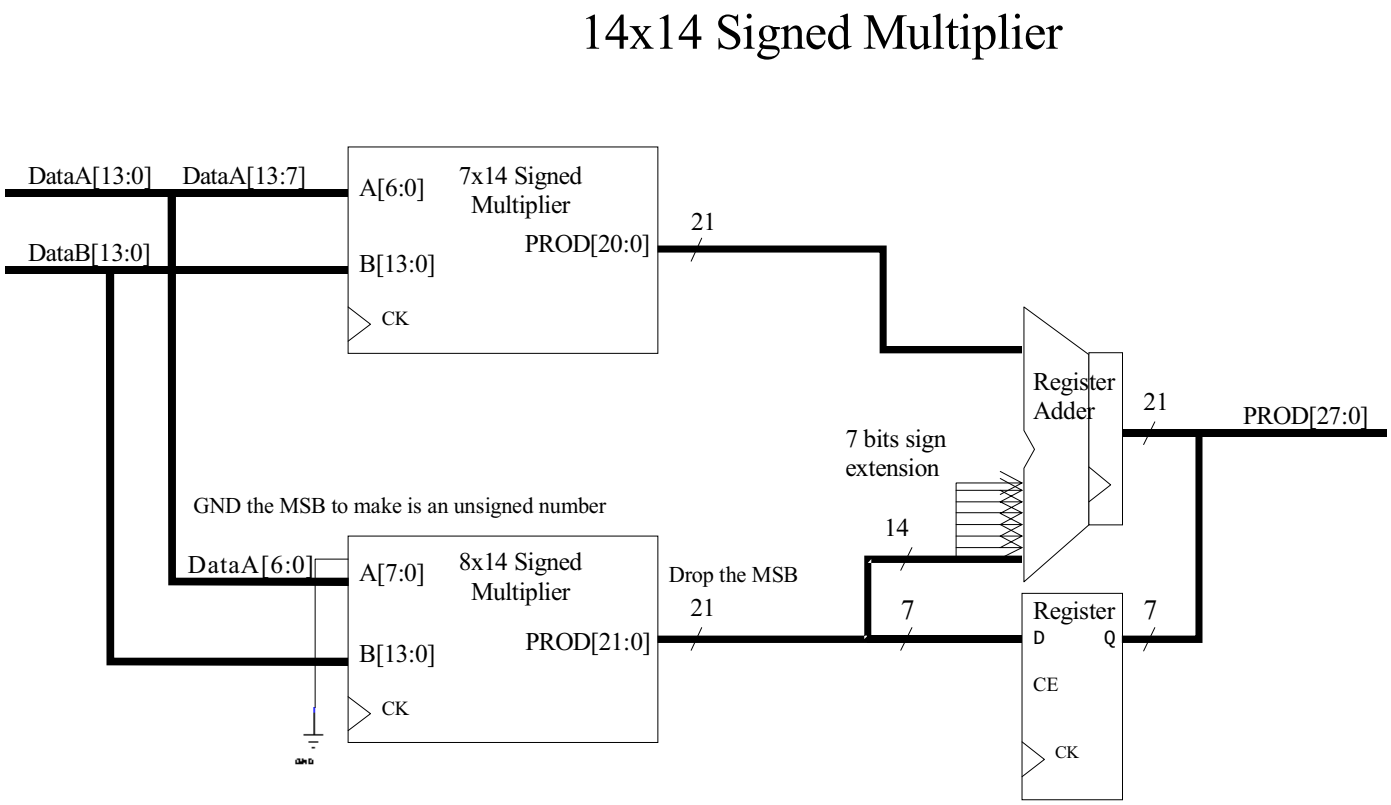
Design Overview

A regular 14x14 multiplier runs at **62MHz** and uses **173 CLBs** (see Data Sheet). In order to increase the speed of the Area Optimized Multiplier provided in the Xilinx Core Generator, it is possible to use two multipliers in parallel. Each multiplier processes one half of the data and their outputs are added together to produce the full resolution result.

Q1) What is the performance and the CLB count of a 7x14 multiplier?

Note: If splitting one of the input in half doesn't give you the performance required, it is possible to split it in 3 or more, in which case 3 or more multiplier would run in parallel.

Each of the modules needed to implement this design are parameterizable cores available in the Xilinx Core Generator. There are four basic components to the filter: The Upper half multiplier, the lower half multiplier, the output adder and the output register.



Upper Half Multiplier: Used to multiply the upper half of DataA time DataB. Both DataA and DataB's MSB represent the sign bit in this case so the multiplier to generate is a 7x14 Signed multiplier.

Lower Half Multiplier: Used to multiply the lower half of DataA time DataB. DataA's MSB (7^{th} bit) in this case is NOT a sign bit and hence the input A should be treated as an Unsigned data. Since the Core Generator is not currently able to generate an UNSIGNED x SIGNED multiplier, it is necessary to add an extra bit to the lower half of DataA and ground it. The multiplier to generate is now a 8x14 Signed multiplier.

Lower bit register: Used to register the shifted data. In this example, the register to generate is of 7 bits.

Registered Adder: There are two important things to remember in regards to adding the partial results together. The first is that the incoming data carries signed information and is therefore in twos complement form. The second is that the upper half of the data is 7 bits more significant than the lower half. This relative positioning must be recreated when adding the partial results. An efficient way to obtain the final result is to shift the less significant result 7 places to the right before adding. Notice that when this occurs the least significant bits in the lower significant result have nothing to add, so we can remove and save them in a register.

```
              1110..110010110
+ 10101100110..11
```

Removing the 7 lower bits gives:

```
              1110..11
+ 10101100110..11
```

These numbers, however, are both in twos complement form, and carrying sign data in their most significant bit. To maintain the sign value for both numbers it is necessary to sign extend the less significant result 7 places. To sign extend, the most significant bit (now bit 20) is replicated onto newly created bits 21,22,23,24,25,26 and 27. This gives:

```
  1111111 1110..11
+ 10101100110..11
```

The purpose for this is to have the sign bit line up.

Q2) Does sign extending the number changes the value?

Q3) How many CLBs is this faster multiplier going to require?

Q4) What is the limiting performance path? What can be done?

Q5) Create a new Foundation Project and implement this design.

Q6) Simulate to make sure that it is correctly implemented. Remember that you are dealing with signed number (2's complement format).

3FFF x 3FFF = –1 x –1 = 1 = 0001
1FFF x 1FFF = 8191x 8191 = 67092481 = 3FFC001
3FFF x 0001 = –1 x 1 = –1 = FFFFFFF

1. Complex Multiplier

Complex multipliers are often used in DSP application…

The implementation we are the most familiar with is the following:

(A + jB) (C + jD) = AC – BD + j(AD + BC)

where the Real side is Re = AC – BD
and the Imaginary is Im = AD + BC

This Complex multiplier requires 4 Multipliers, 1 subtractor and 1 adder which is pretty expensive as far as CLBs.

Q1) How many CLBs would be required to implement a 14x14 complex multiplier using this implementation?

By decomposing this equation differently, we can achieve a much more efficient way to implement this multiplier:

(A + jB)(C + jD) = AC – BD + j(AD + BC)

Take:

x = a (c + d) = ac + ad
y = b (c – d) = bc – bd
z = d (a + b) = ad + bd

Then:

x – z = ac + ad – ad – bd = ac – bd = **Re**
y + z = bc – bd + ad + bd = bc + ad = **Im**

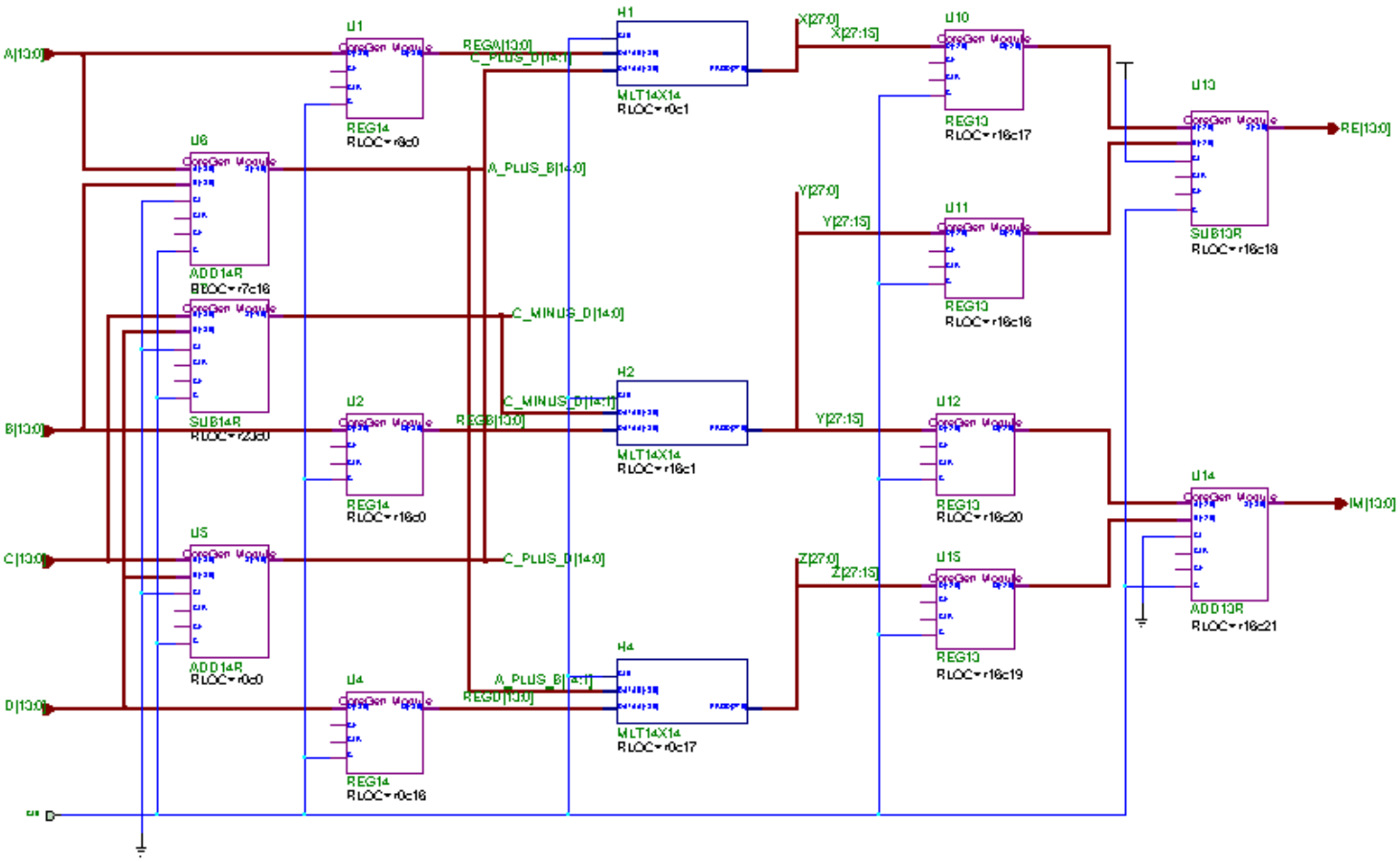
In summary:

Re = x –z = a(c + d) – d(a + b)
and
Im = y + z = b(c - d) + d(a + b)

This Complex Multiplier can now be implemented using 3 Multipliers, 3 adders and 2 subtractors.

Q2) How many CLBs are now used for a 14x14 complex Multiplier with this new approach?

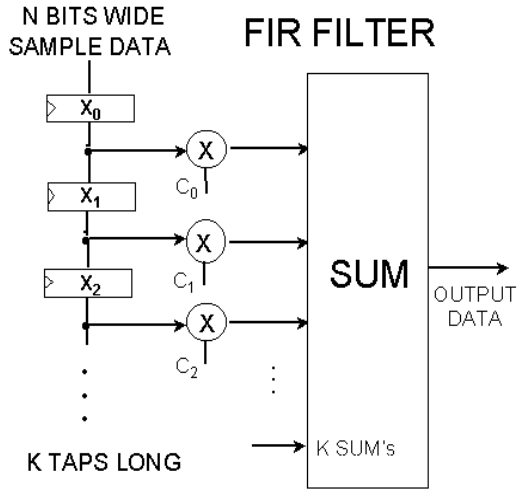
Design Implementation



You are done with Lab 2!

LAB 3: Finite Impulse Response (FIR) filter

The goal of this Lab is to understand how to generate a FIR filter using the Core Generator and how to simulate it.



Functional Description

The input signal x(n) is a series of discrete values obtained by sampling an analog waveform. In this series x(0) corresponds to the input value at t=0, x(1) is the value after on sampling period, x(2) is the value after 2 sampling period, and so on. Each set of registers is a time delay of one sampling period.
x(n-1) is the value of x(n) one time period before now, i.e., the previous input.

In the example shown above, the output signal y(n) is always a combination of the last K input samples. Each of the samples is multiplied by a coefficient, C_R, to give:

$$y(n)= C_0.x(n) + C_1.x(n-1) + C_2.x(n-2)\ldots\ C_k.x(n-k)$$

Notion of Symmetry

A Filter is said **Symmetric** when its coefficient match the following pattern:

$C_0 = C_k$, $C_1 = C_{k-1}$, $C_2 = C_{k-2}$, $C_3 = C_{k-3,\ldots..}$ i.e: for a 10 Taps filter: $C_0 = C_9$, $C_1 = C_8$, $C_2 = C_7$ ...

Now, the equation becomes: $y(n)= C_0.[x(n) + x(n-k)] + C_1.[x(n-1) + x\ (n-k-1)] + C_2.[x(n-2) + x(n-k-2)]\ldots$

Negative symmetry is the following:

$C_0 = -C_k$, $C_1 = -C_{k-1}$, $C_2 = -C_{k-2}$, $C_3 = -C_{k-3,\ldots..}$

Negative symmetry causes phase shift in the data of 90 degrees

Non Symmetric is when none of the above applies.

Module Generation

From Coregen, Select the SDA Single CHANNEL module under LogiCORE -> DSP -> Filters -> FIR Filters -> Serial Distributed Arithmetic -> SDA FIR Filter single-Channel.

The parameters are the following:

- 8-bit data in, signed
- 12 Taps
- 10-bit coefficient width
- Symmetric
- Maximum output width (21)

Q1) How many CLBs is this filter going to use?

Q2) What is the maximum Sample Rate if it is clocked at 70MHz?

Before you can generate the filter, it is necessary to enter the coefficients using a Coefficient (*.COE) file. Create your own coefficient file based on the example below, or by modifying the one found under c:\coregen\wkg\sdafir.coe. The syntax and parameters of the COE file are also explained in the online documentation under the Help -> Help Topic menu.

********** COE file Example **********

```
compname=FIR12T8B;
taps=12;
radix=10;
inputwidth=8;
outputwidth=21;
coefwidth=10;
symmetry=symmetric;
data= 1, 2, -3, -4, 5, 6;
```

****

The coefficients are listed after the Data parameter which should always be placed last in the COE file and are in increasing time order.

Double click on the “SDA FIR Filter Single Channel” entry to reveal the SDA FIR Filter parameterization window. Click on the “Load coefficients” button and specify the example.coe file shown above. Inspect the various fields on the window to ensure that they have all taken-on the values from the .COE file.
Verify that the coefficients have been read correctly by clicking on “Show Coefficients” and finally click on **Generate**

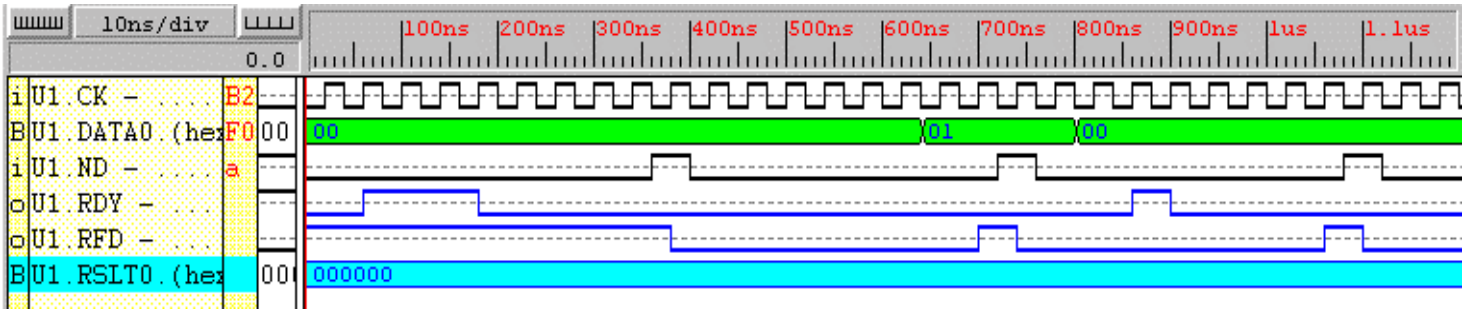
Simulation

Load the symbol into the Foundation Schematic Editor and attach some Nets and Busses to it, or attach the simulation probes directly on the symbol pins. Since the "Cascade option" was not selected, the pins labeled SINF, SINR, SOUTF and SOUTR are unused and can be safely ignored. If you are not familiar with the pinouts, please check the table below.

CORE Signal Pinout

Signal	Signal Direction	Description
DATA	Input	Parallel Data Input – N-bits wide
CK	Input	Bit rate clock
ND	Input	New Data, active high to indicate that the next rising edge of the clock will cause data to be loaded into the parallel-to-serial converter (PSC).
RFD	Output	Ready For Data – active high when the last data bit is about to leave the PSC and a new data word may be applied and loaded into the filter on the DATA input.
RSLT	Output	Parallel Data Out – N+x bits wide
RDY	Output	Result Ready – active high when the RSLT data is available

The Sample Data is loaded in Parallel on the raising edge of the clock when ND (New Data) is high. New Data should be asserted for ONE clock cycle after RFD (Ready For Data) goes high. In order to initialize the internal counter in the SDA FIR Filter, it is necessary to run the clock for several cycles (as many as the input bit width) before loading a new sample.



There are two ways to simulate this Filter and easily verify that it is functioning properly:

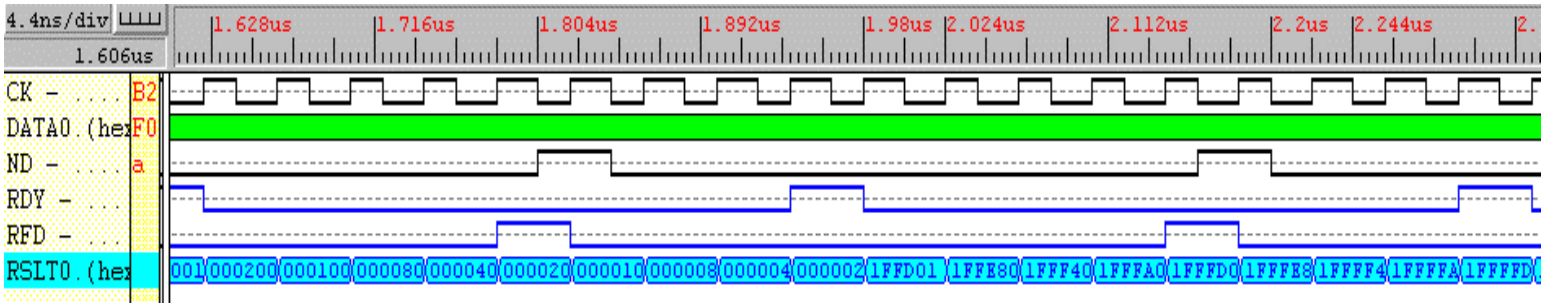
1) **Impulse response**

An impulse is a non-zero value surrounded by all zeros on either side. This non-zero value is one sample wide.
For the impulse response, you will be sending an impulse of 01\h on the Data Input and verify that all your coefficients come out.

If we look at the FIR Filter equation: $y(n)= C_0.x(n) + C_1.x(n-1) + C_2.x(n-2)\ldots C_k.x(n-k)$

The output of the filter should be the following:

$y(0)= C_0.\mathbf{01} + C_1.00 + C_2.00\ldots C_k.00 = C_0$
$y(1)= C_0.00 + C_1.\mathbf{01} + C_2.00\ldots C_k.00 = C_1$
$y(2)= C_0.00 + C_1.00 + C_2.\mathbf{01}\ldots C_k.00 = C_2$
….
$y(k)= C_0.00 + C_1.00 + C_2.00\ldots C_k.\mathbf{01} = C_k$



Simulate and verify that the Coefficients appear on the output when the RDY signal (ReaDY) is high.

2) **Step response**

In the step response simulation, the sample is a non-zero value which remains unchanged for as many samples as number of Taps. The ouptut in this case should be a sum of coefficients.

If we look at the FIR Filter equation: $y(n)= C_0.x(n) + C_1.x(n-1) + C_2.x(n-2)\ldots C_k.x(n-k)$

The output of the filter should now be the following:

$y(0)= C_0.\mathbf{01} + C_1.00 + C_2.00\ldots C_k.00 = C_0$
$y(1)= C_0.\mathbf{01} + C_1.\mathbf{01} + C_2.00\ldots C_k.00 = C_0 + C_1$
$y(2)= C_0.\mathbf{01} + C_1.\mathbf{01} + C_2.\mathbf{01}\ldots C_k.00 = C_0 + C_1 + C_2$
….
$y(k)= C_0.\mathbf{01} + C_1.\mathbf{01} + C_2.\mathbf{01}\ldots C_k.\mathbf{01} = C_0 + C_1 + C_2 + \ldots + C_k$

End of Lab3.

End of the Coregen Lab!

Thank you.