HUNT ENGINEERING
Chestnut Court, Burton Row,
Brent Knoll, Somerset, TA9 4BP, UK
Tel: (+44) (0)1278 760188,
Fax: (+44) (0)1278 760199,
Email: sales@hunteng.co.uk
www.hunteng.co.uk
www.hunt-dsp.com

DSP with HERON modules that have FPGAs

v 2.3 T.Hollis 09-05-05

The HERON-FPGA and HERON-IO families are ranges of HERON modules with FPGAs, often combined with some interface capability. They may be used as processing elements, as interface elements, or as some combination of the two. They can implement simple, high-speed tasks very efficiently, and so are ideal for fast DSP processing – such as in wireless, RADAR or imaging; but they can also implement interface elements such as UARTS (RS232 etc), PWM generators or digital camera interfaces.

HUNT ENGINEERING provides a library of interface components, allowing developers to focus on the application-specific parts of the system. All of the module hardware should be accessed using parts from the library

This tutorial walks you thorough adding some Digital Signal Processing to a data stream, using the FPGA on a HERON-IO2V. The principles discussed can be applied to other FPGA based modules, but the bit streams and example programs supplied will obviously need to be changed accordingly.

History

Rev 1.0	first written
Rev 1.1	tidied and updated for ISE
Rev 2.0	changed to reflect 2nd generation support (VHDL), and HEART based carriers
Rev 2.1	minor documentation changes for HERON-IO2 Version 2
Rev 2.2	Core Generator now run from within Project Navigator, files are now for IO2V version2
Rev 2.3	updated to reference ‘Using Different Versions of ISE’ app note

Concepts

FPGA devices are programmable hardware, which can be configured to meet the needs of your system. As with a microprocessor, the function of the FPGA is controlled entirely by the program – and by the peripherals the FPGA is connected to.

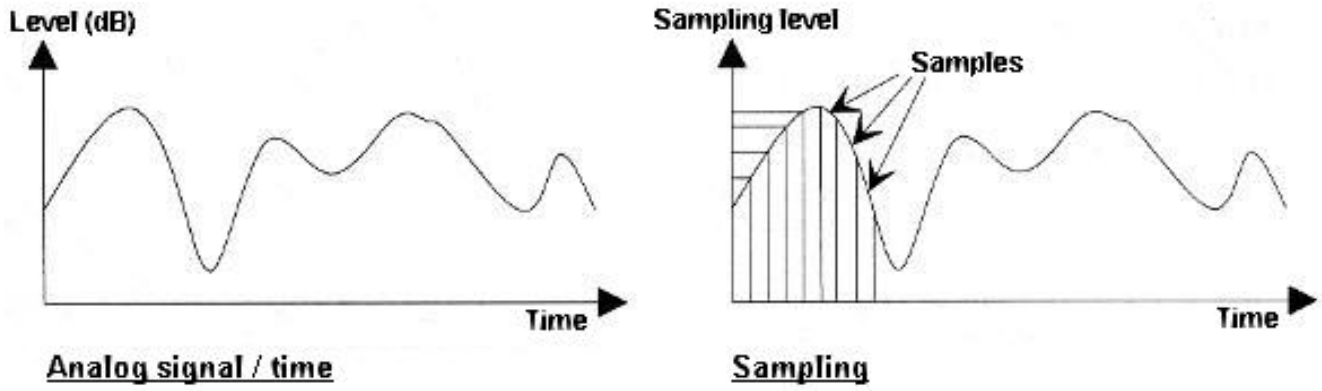
A HERON module has access to FIFOs for communicating with other modules in the system – there may be up to 6 input FIFOs and 6 output FIFOs. It may have additional interfaces, such as ADCs, or level-shifting buffers, allowing it to interface to the real world.

The Heron-FPGA family use Xilinx reconfigurable logic devices. These can be programmed at low level, but can also be programmed using high level blocks, such as FIR filters, FFT transforms and so on. This is the subject of this tutorial. It is assumed that you have already followed the "Starting your FPGA development" tutorial so know how to create and load bit streams onto the modules.

Digital Signal Processing

Digital Signal Processing generally involves sampling an analog signal, and using a Digital technique to process that signal.

The advantage of using Digital processing is that it removes the problems of temperature age and tolerance dependence that is typically associated with Analog processing.



When you combine this with a system that is programmable so that the same hardware can perform different tasks, the reasons to use DSP become stronger.

Typically DSP is trying to change the frequency content of a signal, using operations like filters along with sub-sampling etc. This is usually implemented in a processor device like the C6000 using a high level language like C.

Actually the action of sampling the signal already changed the frequency content of your signal, but we will not address that issue here.

When you look at Signal Processing Theory, algorithms are usually drawn as delays, adders and multipliers (See the next section).

FPGAs allow you to make designs using registers (delays) adders and multipliers. Most people initially worry about how they might program a simple function like a Digital Filter into an FPGA, as they do not know how to program in VHDL. Actually not make people know how to code a Digital Filter in C either, but they don't worry about that as they can get C callable libraries for functions like that.

Well actually you can for your FPGA design too. These "libraries" are generated by the "Core Generator" which is part of the standard Xilinx Tools.

This is what we will do in this lab.

Once we have made our filter we can compare it with one made on a DSP and immediately see the advantages of building such functions in an FPGA.

Filters

There are two main types of Digital Filters, Finite Impulse Response (FIR) and Infinite Impulse Response (IIR).

The IIR filter uses feedback, so requires careful selection of the number of taps, and the co-efficients to prevent oscillations of the filter output. These issues occur whatever processor you use to implement it.

The FIR is simpler, and has a block diagram of:

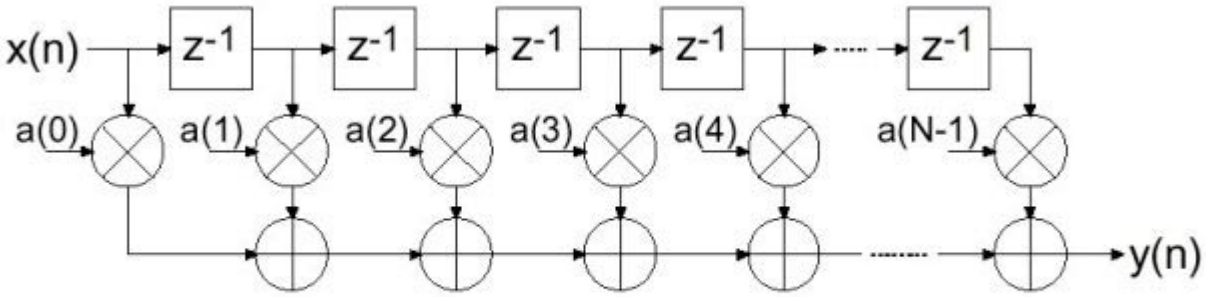


Figure 1: Conventional tapped-delay line FIR filter mechanization.

Which you can see can be simply made out of registers (Z-1) multipliers (X) and adders (+).

The response of the filter is governed by the number of taps n, and the value of the co-efficients a(0) to a(n).

Tutorial

This tutorial concentrates on changing your FPGA design to include a filter that is generated using the Xilinx Core Generator. To view the results of your changes you could use Code Composer running on a DSP module, or the hegraph program running on the host PC. As all users can configure their system to use the hegraph program, but only users that also have a C6000 HERON module could use Code Composer, we will describe the tutorial using the hegraph program.

If you prefer to use Code Composer for this, you can do so by adapting the instructions found in the HERON-IO2 Example2 instructions.

The supplied “answer” projects and bitstreams are for a HERON-IO2 Version2. If you have a HERON-IO2 Version1 you need to follow the same instructions but using the io2v1\transient_analysis(ex2) project.

Actually you can follow this example if you have a module that is different from the HERON-IO2 Version 2, but you will need to take your signals from another source. For example a HERON-FPGA module could take the filter input data from a FIFO. How to do this is not described here and is left as an exercise for the user. The principles of making a filter and applying it to your FPGA design will be the same.

If you have a HERON-IO2, and a HEART based carrier (like HEPC9) you can fit the module into any slot, but if you have an HEPC8 the HERON-IO2 must be in the first slot.

With a HEART based carrier you make the connections to the module yourself using heartConf (if you don't know how to do that then you should refer to the presentations and documentation about using HEART again).

With an HEPC8 the FPGA on the module uses FIFO #1 to communicate with the Host machine.

Now you can work through the Host example provided for example2 of the HERON-IO2. To do that follow "Getting Started" → "to start using FPGA modules and tools …" → "Examples and IP by module type" → "HERON-IO2" → "Transient Analysis/pattern Generation".

When you click on that link you open a document that describes the example and how to use it. If you have not already followed that document, follow it now.

What the document describes is loading the example2 bitstream for the HERON-IO2. This allows us to set patterns to be "played" on the DAC outputs at 100Mhz sample rate, and to digitise blocks of data using our A/Ds at 100Mhz sample rate.

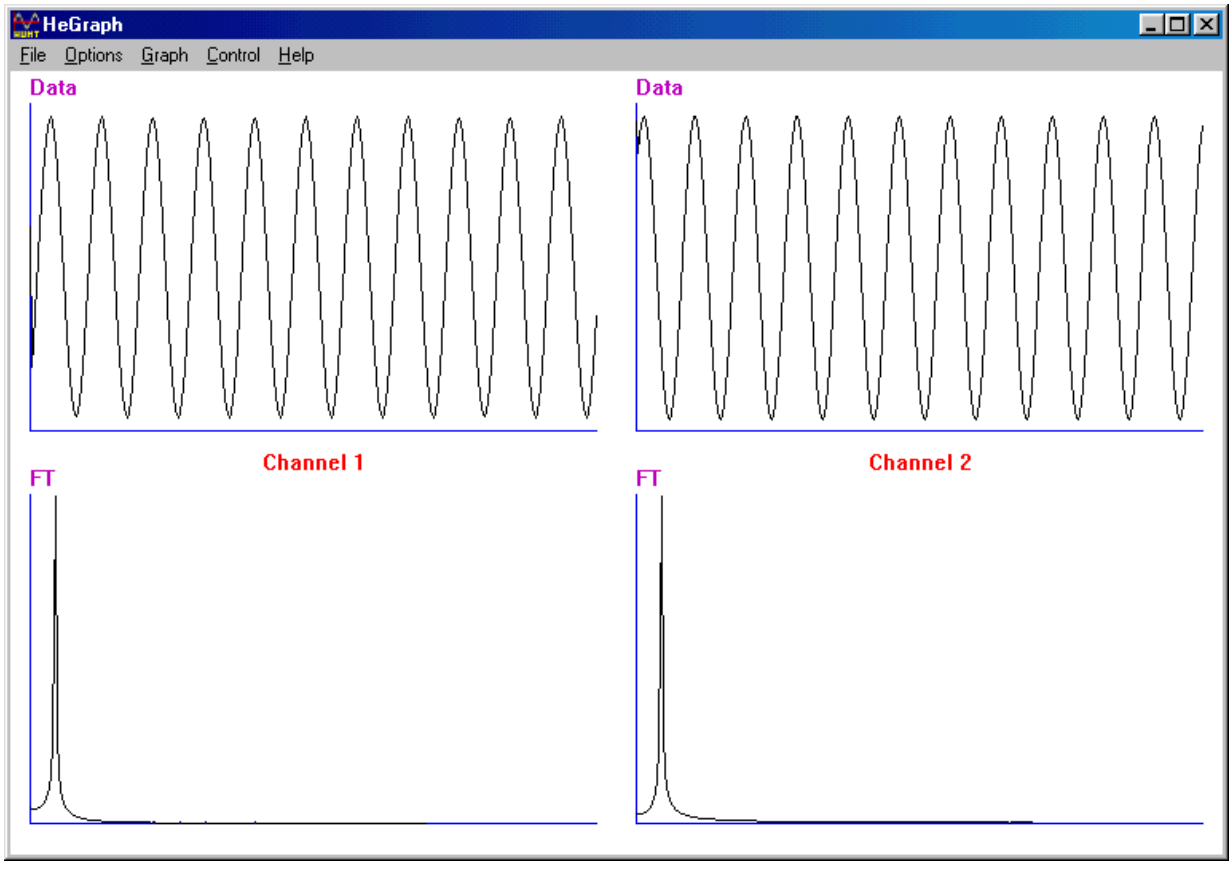
For this tutorial we will follow the "Host" part of that example.

Connect the inputs to the outputs of the HERON-IO2 so O/PA drives I/PA, and O/PB drives I/PB. This is Ok as the outputs have a 2V peak to peak output which is the same as the input levels.

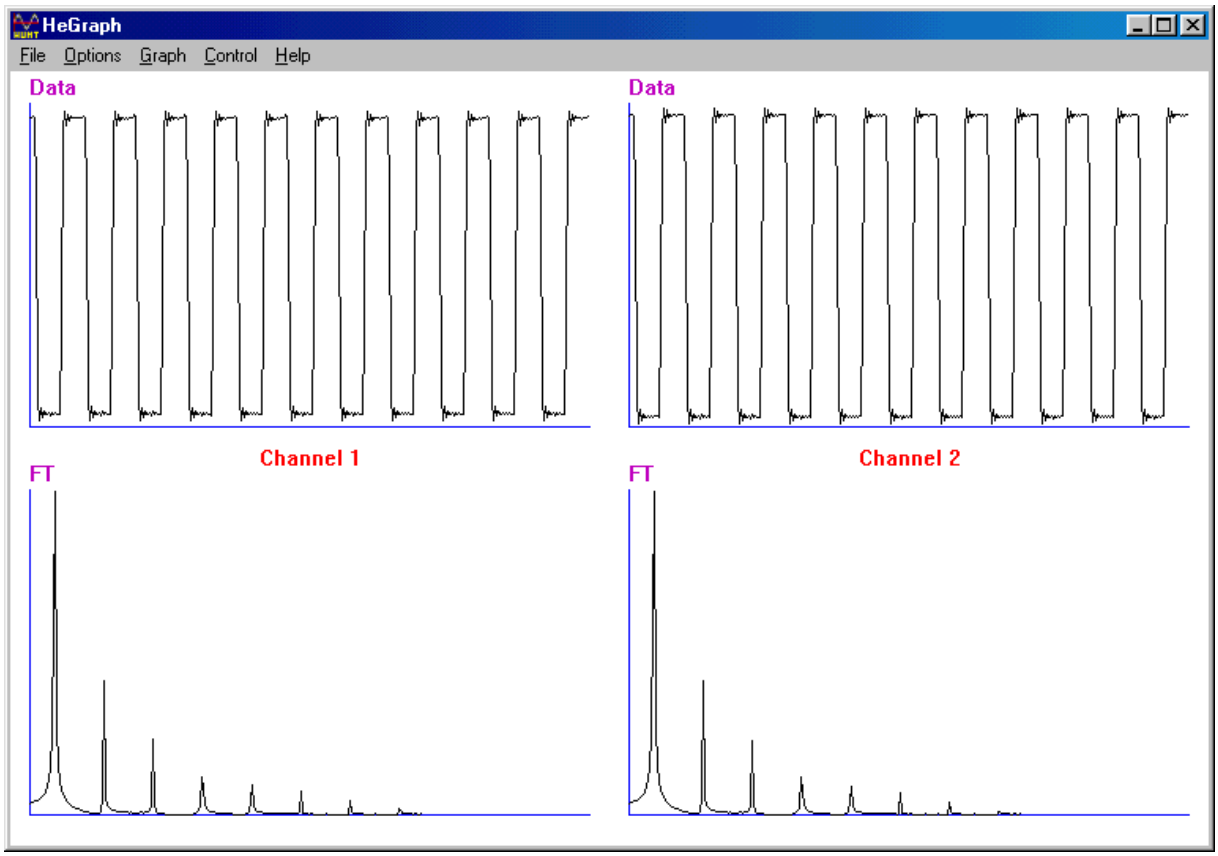
Load the example2 bitstream into your module and start the host application "hegraph" found in the \fpga\io2v1\example2\host directory of the CD. You can get here easily by pressing the "files" link next to the "Transient Analysis/pattern Generation" link on the CD menu program. Then select the "host" directory.

Select "File→Start" and the outputs should have a sine and cosine wave on them, which can be seen in the input graphs.

Use "Graph → Channel1 FFT → on" and "Graph → Channel2 FFT→on" to display the frequency spectrum of the signals. This is almost a single vertical line, as the sine-wave contains a single frequency. Actually the sampled nature of the output and input will introduce some impurities, but they are small as the frequency of the signal is very far from the 100MHz sample rate. What you should see is :-



Now use the “Control → function2” menu to switch the outputs to square-wave. You should see :-



Notice that the signal is not an exact square-wave. This is because a perfect square-wave has an infinite number of harmonics, or frequencies. Our sampling has removed some of these harmonics that are outside of our analog bandwidth.

The harmonics of the signal can clearly be seen in the Frequency domain shown on the graphs at the bottom.

What we want to do now is to make an FIR that will modify one of those channels.

Filter requirements

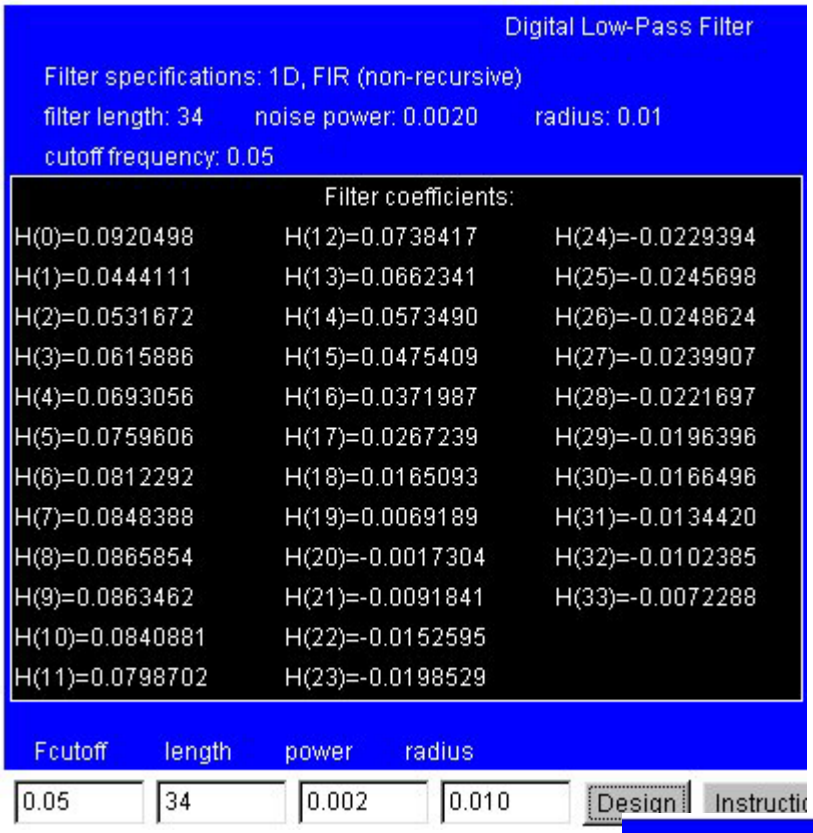
Our exercise is to design a low pass digital FIR filter to be fitted inside the Xilinx. The first stage in the design is to decide what the characteristics of the filter should be, for this example it will be a low pass filter with a cut off frequency of approximately 5 MHz. This frequency has been chosen as the square wave generated using the D/A converters, with 16 samples high and 16samples low will have a frequency of 3.125MHz, with the 100MHz D/A output sample clock. The harmonics of the square wave will fall into the stop band of the filter and so be attenuated, but the fundamental will be un-attenuated in the pass band so the output will be close to a sine wave.

Filter Co-efficients

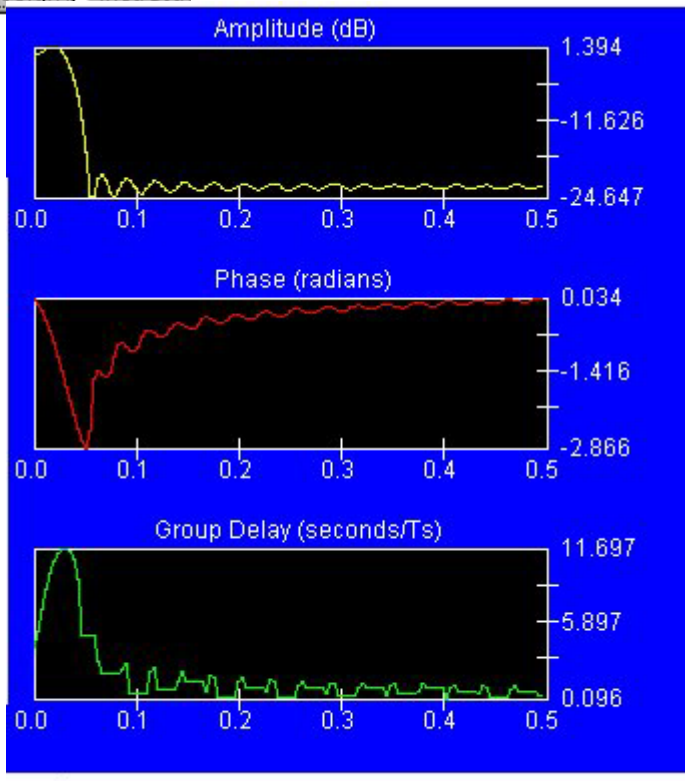
There are software packages to evaluate the coefficient values required for different frequency responses, the one we have used here is free online software from Nauticom, and it can be found through the HUNT ENGINEERING web site under the filters section. From the index page of this web site the filter design in this example is found by selecting “The Band-Select FIR Filter Design” then “FIR Low-Pass Filter 1”, the filter parameters can then be entered. The cut off frequency required is 5MHz which when normalised to the converters sample clock frequency of 100MHz is

(5/100) = 0.05, a filter length of 34 taps has been chosen to give a relatively sharp cut off. Selecting "Design" will cause the required filter coefficients to evaluated, and also give the resulting amplitude, phase, and group delay plots. By changing the filter parameters and pressing "design" the effect of varying the parameters can be seen, for example reducing the filter length to say 20 causes the pass band and cut off to become less well defined. The power and radius values are for fine tuning the filter and have been left at their default values.

With the values we have chosen you will see:-



For the coefficients, and then these graphs



ons

Once the required filter response has been obtained, the software package gives a set of coefficients which are all real numbers with a maximum value in this example of 0.0920498. The filter that we are going to fit into the Xilinx will be generated be the "Core Generator" which will require a file containing the coefficients in integer form ranging from +2047 to –2047 for 12bit signed coefficients.

To convert the coefficients from the filter design software package to the integer values required, we make the largest co-efficient become 2047, and all the other values are multiplied by 2047/(original largest value) then rounded to the nearest integer value.

The Xilinx "Core Generator" uses an ASCII text file with a .coe extension, further information about the format of this file can be found via the "Core Generator" FIR filter page.

In our example this file becomes :-

```
radix=10;

coefdata=2047,988,1182,1370,1541,1689,1806,1887,1925,1920,1870,1776,1642,1473,1275,
1057,827,594,367,153,-38,-204,-339,-441,-510,-546,-552,-533,-493,-436,-370,-
299,228,-161;
```

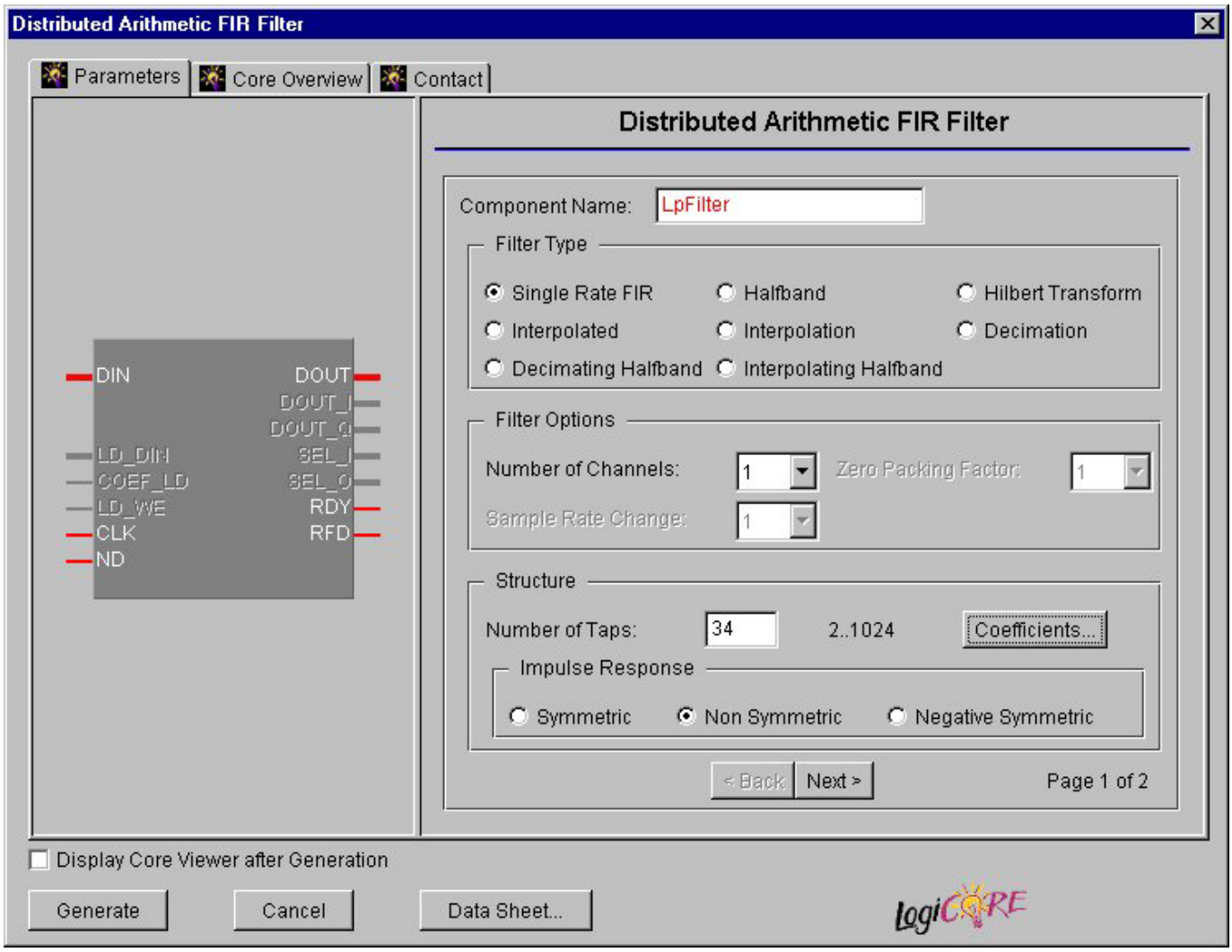
This file can be found in the directory \fpga\tutorials\dsp_with_fpga on the HUNT ENGINEERING CD

Generating the filter

The Core Generator is run from within the Xilinx Project Navigator, so copy the 'Transient analysis(ex2)' and 'Common' folders for the IO2V2 from the CD as described in the 'Getting Started with FPGA' application note.

Start the Project Navigator and open the '../Transient analysis(ex2)/ise/ex2_io2v2.ise' project that has been copied from the CD. Select Project → New source then select 'CoreGen IP' from the options and in the 'file name' box enter 'lpfilter' , select 'next' then 'finish' to start the Core Generator.

In the Core Generator the Distributed Arithmetic FIR Filter section of the Core Generator is located by selecting: Digital Signal Processing → Filters→ Distributed Arithmetic FIR Filter. This will bring up the parameters page for the filter Core Generator with the following options:



*Component Name. This is the name of the filter core once it has been generated.

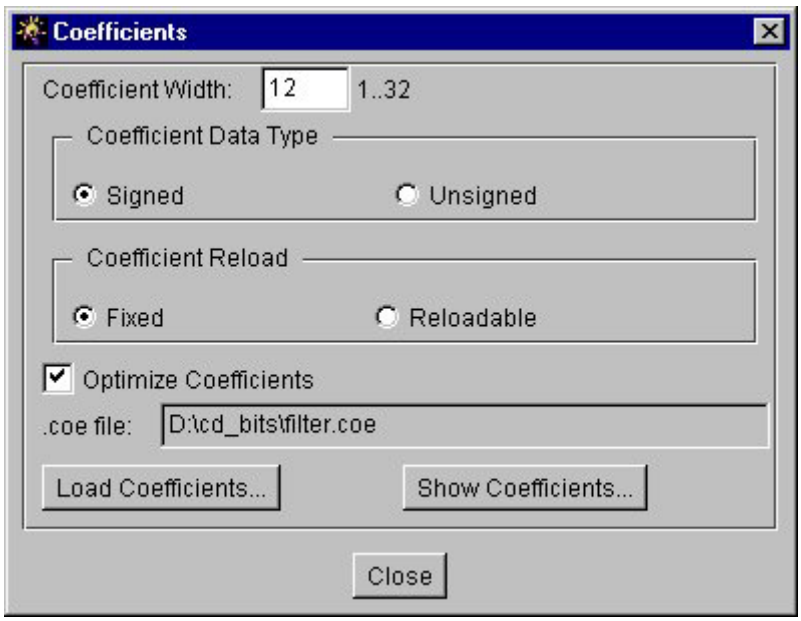
*Filter Type. The required filter in this example is a single rate filter, this has the same sample rate at its input as at its output. Information on the other options can be obtained by clicking on the Data Sheet button at the bottom of the parameters page.

*Filter Options. The filter required for this example is a single channel filter.

*Structure. The number of taps for this filter is 34 as defined by the Nauticom filter design software.

Looking at the coefficient values, which are the Impulse Response of the filter, they can be seen to be Non Symmetric.

By clicking on the Coefficient button a Coefficients Window will come up:



-Coefficient Width. The data in the coefficients file has been normalized to 12 bit signed.

-Coefficient Data Type. Signed.

-Coefficient Reload. In this example the coefficients are fixed.

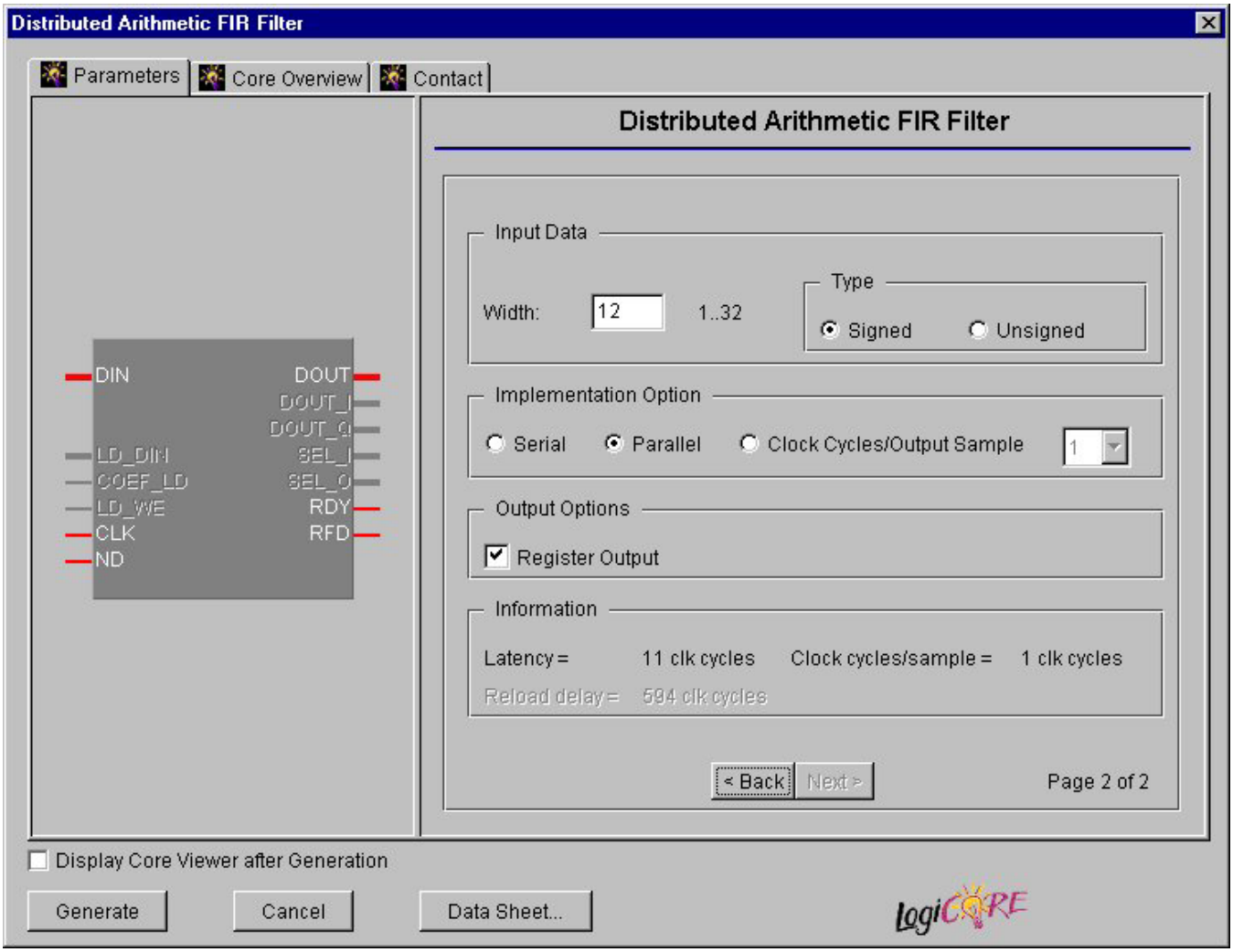
-Optimize Coefficients. Selecting this optimises the amount of FPGA logic fabric employed by the core for the look up tables.

-Load Coefficients. To load the Coefficient file generated from the Nauticom filter design software.

-Show coefficients. Displays a list of the Coefficients loaded.

-Close. Closes the Coefficients window.

On the Distributed Arithmetic FIR Filter window click on the Next> button, this brings up the second page of parameters.



*Input Data. The input data is 12 bit signed data from the A/D converters

*Implementation Option. In this example we use a parallel implementation to give a single output sample per clock cycle.

*Output Options. The output sample value is registered. This puts a clock cycle delay in the output value but the output value is held between output samples.

*Information. This shows the latency and output sample rate for the design selected.

Once the Filter Design Parameters have been selected the Core Generator can be set to generate the Core by clicking on the Generate button. Depending on the speed of the machine used this may take some time, in the order of one hour.
When it has completed it has generated the files:-

`lpfilter.edn`	the netlist of the filter core
lpfilter.vho	the templates to use in your design (see next section)
lpfilter.vhd	the simulation model for the core
lpfilter.mif	the coefficients and other settings for the simulation.

Exit the Core Generator by File → Exit.

Once the Core Generator has completed the 'lpfilter' Core the symbol can be used in the Example2 project. The 'lpfilter' core should now be displayed in the 'Sources for project' window below the Xilinx device type.

Open the file user_ap2.vhd by double clicking on it in the 'Sources for project' window. Then select Edit → Language Templates and select CORGEN\VHDL\lpfilter which will open the template file to allow the component declaration and instantiation to be copied to the user_ap2 file.

The component declaration:-

```
component lpfilter
          port (
          ND: IN std_logic;
          RDY: OUT std_logic;
          CLK: IN std_logic;
          RFD: OUT std_logic;
          DIN: IN std_logic_VECTOR(11 downto 0);
          DOUT: OUT std_logic_VECTOR(27 downto 0));
end component;
```

must be added to the user_ap2.vhd in the components section found under the line :-

```
architecture EXAMPLE2 of USER_AP is
```

Then the instantiation :-

```
your_instance_name : lpfilter
          port map (
                    ND => ND,
                    RDY => RDY,
                    CLK => CLK,
                    RFD => RFD,
                    DIN => DIN,
                    DOUT => DOUT);
```

Should be added to the section below the line

```
--        <<<<<<<    INSERT YOUR CODE HERE     >>>>>>>>>
```

This instantiation needs to be edited so that it is used in your design.

The output of the filter is a signed 28bit word, for this example only a 12 bit signed output is required. The output level of the filter will depend on the co-efficients. By experimenting with the co-efficients of this example we have found that using bits 25-15 plus the sign bit gives an output that is just slightly smaller than the input. Different co-efficient sets will mean that different output bits from the filter should be used.

In our case, the 12 bit output bus is made up of the sign (filter output bit 27) together with filter output bits 15 → 25.

To do this we need to make a new bus which we can call FILTER_OUT.

We define it as

```
signal FILTER_OUT : std_logic_vector(27 downto 0);
```

Then where we define the drivers for the ADC_IN bus, we need to change the bottom 12 bits from

```
ADC_DIN(11 downto 0)  <= ADC_A;
```

To

```
ADC_DIN(10 downto 0)  <= FILTER_OUT(25 downto 15);
ADC_DIN(11)  <= FILTER_OUT(27);
```

The 12 bit input bus is linked to the output of the A/D converter HE_ADC_A which is called ADC_A

The clock for Filter is the same as the A/D converter sample clock so there is one sample per clock. The New Data (ND) input to the filter symbol identifies when data should be clocked in and in this example this is on every clock, so ND should be held asserted. The Ready For Data (RFD) output from the filter symbol indicates when the filter is ready for data, with the parallel implementation it takes data on every sample clock cycle and so is always asserted. Linking the RFD output to the ND input insures that data is taken on every clock cycle.

This means we end up editing the instantiation to be :-

```
 myfilter : lpfilter

            port map (
                      ND =>noname,
                      RDY=> open,
                      CLK => SCLK_G,
                      RFD => noname,
                      DIN => ADC_A,
            DOUT => FILTER_OUT);
```

The new signal "noname" is defined as "`signal noname : std_logic;`"

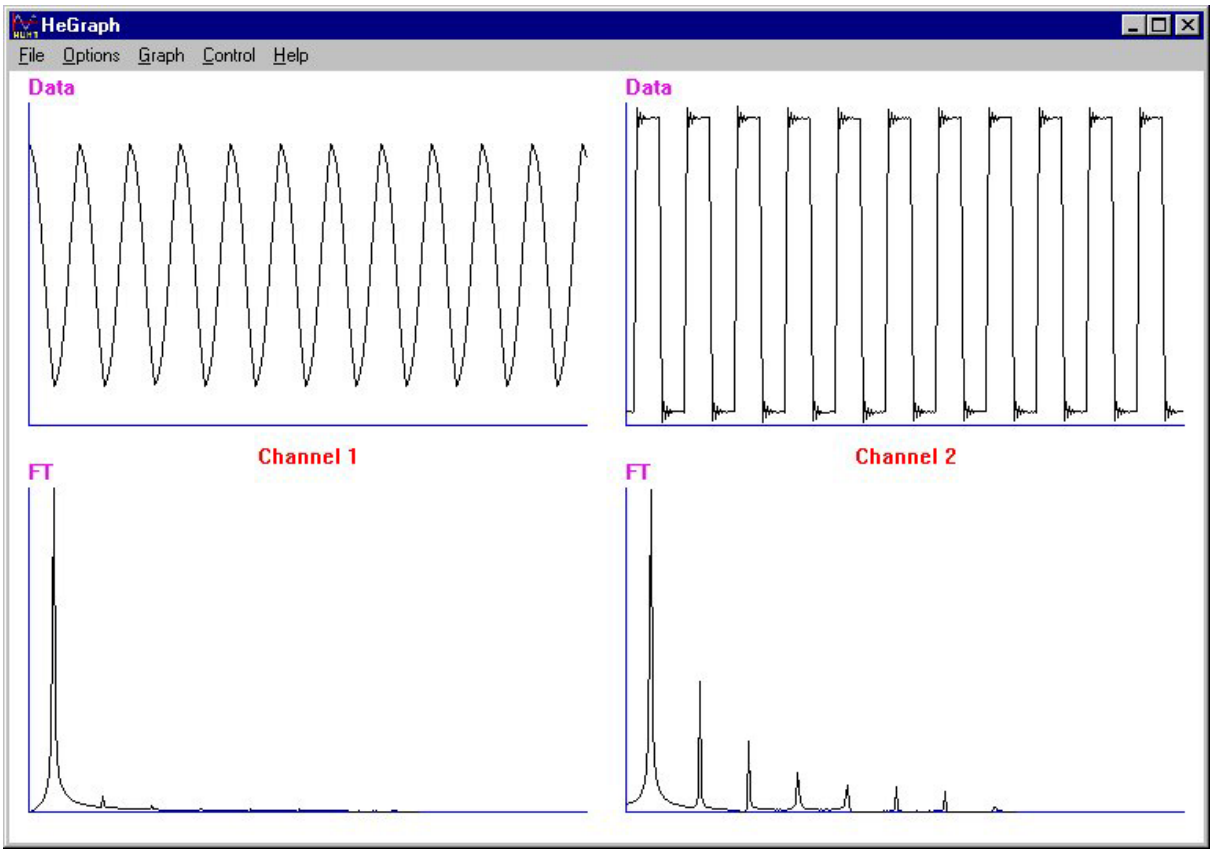
Make a bitstream with that project, and load it onto your HERON-IO2 and start the host application "hegraph" found in the \fpga\io2v2\example2\host directory of the CD.

Select "File→Start" and the outputs should have a sine and cosine wave on them, which can be seen in the input graphs.

Use "Graph → Channel1 FFT → on" and "Graph → Channel2 FFT→on" to display the frequency spectrum of the signals. The signal on channel A will be slightly smaller because the filter has scaled the signal.

Select the squarewave outputs ("Control → function2").

Now you should see:-



The data from channel A is now apparently a sinewave, but as before the A/D is capturing the same square wave that can be seen on channel B.

Look at the Frequency plot, and you can see that the harmonic frequencies have been removed by the filter, leaving just the main frequency.

Simulation of FPGA

The example2 that is used to derive this tutorial comes with simulation models that allow you to provide text files for the ADC inputs, and it generates a text file output from the HERON FIFOs.

There is a separate application note that covers simulation in more detail, but for this tutorial you have added a Core generator block. The netlist that is used to generate this block cannot be simulated directly, so the lpfilter.vhd and lpfilter.mif files are used.

To do this you need to edit the simulation script simu.do to add then line

```
vcom -93 ../Src/LPFILTER.VHD
```

on the line above the user_ap2.vhd. The lpfilter.mif file needs to be copied to the sim directory to be properly found when simulating.

For more details on simulation please see the application note on simulating HUNT ENGINEERING FPGA examples.

Comparing the filter with one on a DSP

The most obvious thing about our filter is that we have made it on a module that has twin A/Ds connected directly to the FPGA. In our Example these are clocked at 100Mhz, so we are using 400Mbytes of data every second.

The FPGA on the HERON-IO2 also has two DACs running at similar rates, and an input and output FIFO interface that can each transfer 100Mwords/sec. That totals 1.6Gbytes/sec.

A DSP cannot read this amount of data even when it is using DMA, and not performing any processing.

As we have seen, a filter uses multipliers and adders, which is the limit of the processing speed when you use a DSP. A DSP like the C6000 has two multipliers so can perform two multiplies per clock. This means that the maximum speed of a filter routine will be 0.5cycles per tap of an FIR filter. Our example uses 34 taps for the FIR, so a DSP will take 17 cycles to process each sample.

A C6201 runs at 200Mhz, so could operate at a sample rate of about 11Mhz.

Our FPGA is running the same filter at 100Mhz, and could run faster. What's more we could add the filter to the other channel of input, and have them both run at 100Mhz.

For the FPGA, more taps for the filter simply use more logic and run at the same speed. The sample rate of a DSP running the filter would reduce according to the number of taps.

So it's clear that at least for some DSP tasks using an FPGA gains great advantages for your system – and it wasn't hard was it!

Problem Solving

If you have problems in making your filter work, we have provided projects that are the "result" of this lab. They are for provided for the latest version of ISE, just like the examples that they started from. If you are using a different version of ISE to the projects provided on the CD then please refer to the application note 'Using Different Versions of ISE'.

Alternatively, if you use another Synthesis tool refer to the separate application note 'Using non ISE development tools'.

There are also bit streams provided so that you can see the effect of a working filter.

These are all found in the \fpga\tutorials\dsp_with_fpga on the HUNT ENGINEERING CD.