NASA's Jet Propulsion Laboratory
Pasadena, California 91109-8099

Technical Support Package

Broad-Bandwidth FPGA-Based Digital Polyphase Spectrometer

NASA Tech Briefs
NPO-48352

National Aeronautics and
Space Administration

Technical Support Package

for

Broad-Bandwidth FPGA-Based Digital Polyphase Spectrometer

NPO-48352

NASA Tech Briefs

Innovative Technology Assets Management
JPL
Mail Stop 202-233
4800 Oak Grove Drive
Pasadena, CA 91109-8099

E-mail: *iaoffice@jpl.nasa.gov*

National Aeronautics and Space Administration

Broad Band Digital Radiometers

Principal Investigator: Robert F Jarnot (382F)
Sharmila Padmanabhan (382F), Ryan M Monroe (GIT, SURF student)
Zachary W Pannell (389D)

Project Objectives:

1. Determine why high performance FPGA-based DSP designs created using the standard Matlab/Simulink/System Generator tool flow perform very poorly once FPGA utilization exceeds ~60% of available resources, and find solutions to this problem
2. Implement a 3 GHz bandwidth, 1024 channel, digital spectrometer in a Berkeley ROACH platform (Xilinx XC5VSX95T FPGA and interleaved National Semiconductor ADC083000 3 Gsps ADCs) to support upcoming microwave radiometer requirements
3. Design and implement a low power, rad hard, ASIC with multiple 1 GHz, 2 bit ADCs and cross-correlators to improve ASIC design skills at JPL, and to provide a proof of concept test chip for GeoSTAR

FY10/11 Results:

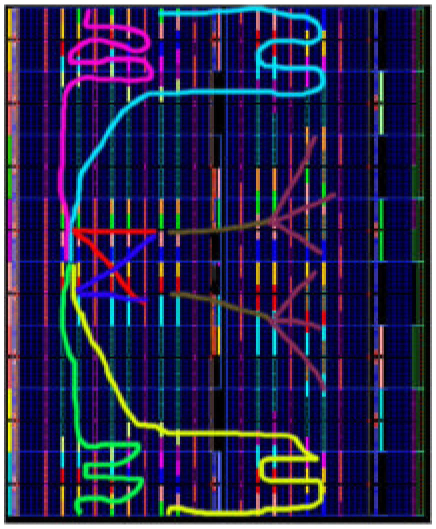
•Multiple reasons for poor performance were determined, including:
- Current synthesis tools do not make effective use of critical FPGA blocks such as DSP48s (multiply-accumulate), using either the multiplier *or* the adder (but never both), and most often ignoring the DSP blocks completely in favor of logic-based equivalents
- Poor layouts, which result in poor timing, higher power consumption than necessary, and an inability to use the dedicated inter-DSP signal routing. The poor layouts also lead to significant clock speed degradation as FPGA utilization increases beyond ~60%

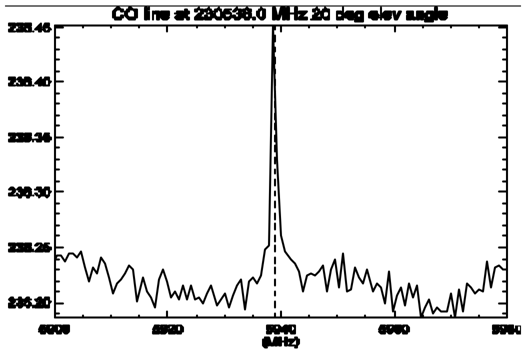
•Solutions:
- Instantiate key blocks (such as DSP48s) and set their operating modes explicitly
- Manually floorplan designs to make effective use of dedicated inter-DSP48 routing, to provide short data paths between key elements (DSP48s and Block RAMs), and to force overall data flow in the FPGA to follow vendor-provided guidelines
- Modify designs and layouts of key structures (such as FFT butterflies) to make efficient use of FPGA resources
- Provide pipelining as necessary to meet timing requirements for 'obstinate' nets

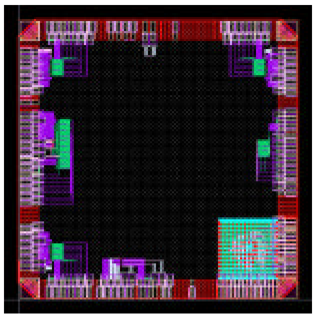
•Accomplishments:
- An 8192 channel, 8 tap, digital polyphase spectrometer with 3 GHz signal bandwidth, successfully tested in the field. See figures below for an example of field data
- This design uses over 90% of the FPGA DSP resources, and operates at 375 MHz (with 8 way parallelism), showing that high FPGA resource utilization and high clock speed are not mutually exclusive goals given appropriate insight and adequate design effort
- The ASIC design effort has resulted in a prototype low-power ASIC meeting the original performance goals. This design is currently in fabrication. The test ASIC layout is shown below

Benefits to NASA and JPL:

•Broad band, high resolution, digital spectrometers are essential for future instruments such as SMLS on the Decadal Survey GACM mission, and Heteracam on SOFIA (32 pixel submm array), and this work has shown the path for their development, demonstrating a state of the art digital spectrometer
•High performance, low power, ASIC back-ends are essential for future synthetic aperture radiometers such as GeoSTAR, and this work has made significant progress in bringing the necessary design skills to JPL
•The transition to digital back-ends for radiometers will bring additional advantages in terms of science data quality, calibration, cost, mass, size, power consumption, stability and manufacturability – i.e. more competitive designs







Figures. Manually constrained high level FFT data flow paths in the 3 GHz bandwidth spectrometer are shown at the left. Two radix-4 FFTs are at the top and bottom of the chip, with their outputs fed to a direct form radix-16 FFT in the center. The center panel shows data (CO emission at 230.538 GHz) taken at Table Mountain Observatory with this spectrometer. The right hand panel shows the 45 nm SOI CMOS ASIC designed at JPL, currently under fabrication. Chip packaging and testing will take place in FY12.

Report: Wideband Spectroscopy: The design and implementation of a 3 GHz, 2048 channel digital spectrometer. URS224289.

National Aeronautics and Space Administration
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, California

www.nasa.gov

Wideband Spectroscopy: the design and implementation of a 3 GHz bandwidth, 8192 channel, polyphase digital spectrometer

Ryan M. Monroe

Georgia Institute of Technology, Atlanta, GA 30332

Mentor: Robert Jarnot

Jet Propulsion Laboratory, Pasadena, CA 91109

A family of state-of-the-art digital Fourier Transform spectrometers has been developed, with a combination of high bandwidth and fine resolution unavailable elsewhere. Analog signals consisting of radiation emitted by constituents in planetary atmospheres or galactic sources are downconverted and subsequently digitized by a pair of interleaved Analog-to-Digital Converters (ADC). This 6 Gsps (giga-sample per second) digital representation of the analog signal is then processed through an FPGA-based streaming Fast Fourier Transform (FFT), the key development described below. Digital spectrometers have many advantages over previously used analog spectrometers, especially in terms of accuracy and resolution, both of which are particularly important for the type of scientific questions to be addressed with next-generation radiometers. The implementation, results and underlying math for this spectrometer, as well as potential for future extension to even higher bandwidth, resolution and channel orthogonality, needed to support proposed future advanced atmospheric science and radioastronomy, are discussed.

I. Introduction

With present concern for ecological sustainability ever increasing, it is desirable to accurately measure and model the composition of Earth's upper atmosphere with regards to certain helpful and harmful chemicals, such as greenhouse gases, Ozone, and pollutants. The Microwave Limb Sounder (MLS) on the Aura Spacecraft is an instrument designed to map the global day-to-day concentrations of key atmospheric constituents continuously.

An important component in MLS is the spectrometer, which processes the raw data provided by the receivers into frequency-domain information which can not only be transmitted to the ground more efficiently, but also be processed directly once received. The present generation mainstream spectrometer in use is fully analog: the present goal is to include a fully digital spectrometer in the next generation sensor. A digital spectrometer would offer considerably superior bandwidth and uniform resolution, while suffering considerably less leakage from side-lobes, and providing a lower cost, more stable and manufacturable solution.

II. Background

In a digital spectrometer, incoming analog data must first be converted into a digital format, processed through a tapped finite impulse response (polyphase FIR) filter, a streaming Fourier Transform, and finally accumulated to reduce data rate and the impact of input noise. While the final design will be placed on an Application Specific Integrated Circuit (ASIC), the building of these chips is prohibitively expensive (over a million dollars): every possible effort must be made to ensure the design is fully functional before sending the chip to manufacturing. To that end, we are constructing our design on a Field Programmable Gate Array (FPGA), which will allow us to prototype and fully test our design before sending it to fabrication, and readily test new ideas (such as RFI detection and enhanced sideband separation).

Field Programmable Gate Arrays are special chips designed with a large amount of logic and interconnect wiring on board. The interconnect is both programmable and extremely flexible, enough that any logical design could, in principle, be made and placed on an FPGA. Unlike regular computer software, designing for an FPGA is really programming the hardware itself, which is both very different from writing conventional software, and poses a special set of problems: for instance, when designing for software, instructions are executed sequentially. In hardware, every 'instruction' is processed simultaneously, with the result being presented to the next instruction on the subsequent clock cycle. For this reason and others, it is extremely difficult to send test vectors to the simulation of a hardware-based design. Once a design is completed on an FPGA, several steps must be processed sequentially in order to build the design into a version which is functional on the actual chip. These steps, in order, are 'Synthesis', 'Translate', 'MAP', 'Place and Route' and finally 'Bitstream Generation'.

In the synthesis stage, the Hardware Description Language (HDL) code is compiled into a logical netlist, which describes the operations required to transform the input data into the form required of the output data. This stage is hardware independent, and many optimizations may be made in order to improve performance or hardware consumption for the later (less flexible) stages.

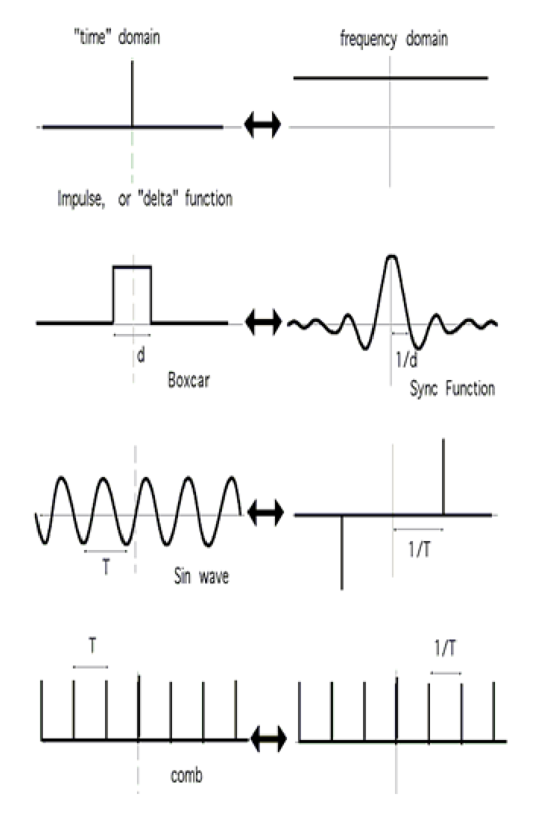
In the translate stage, the design is converted from the hardware-independent logical netlist into a physical netlist, which describes the physical hardware on the target FPGA which will process each logical transformation. Because different hardware is available on each chip, and even similar hardware on different chips may have varying functionality, this step is specific to the class of chips being targeted.

In the MAP stage, the physical hardware is placed throughout the chip. Placement is a very computationally "hard" problem, because not only is the task of finding the optimal placement for a given physical netlist NP-Complete, but the solution space is huge. In addition, finding a good placement has a gigantic impact on the final performance of the chip. All the placed hardware must eventually be connected together using programmable interconnection paths, and hardware blocks which are too distant from each other will have a considerably longer interconnect delay, leading to a slower clock rate. As a consequence, modern toolsets use heuristics to predict placement solutions which will be "good". Because these tools are imperfect, the results for densely packed hardware, such as that which we are targeting, can be inconsistent and are usually poor compared to what a human can achieve.

In the Place and Route (PAR) stage, signals are routed between the placed hardware (the 'Place' in the name is now a misnomer, as that process is managed in 'MAP'). Like MAP, this is a computationally hard problem, and depends heavily on the results produced by MAP.

In the final stage, Bitstream Generation, the completed design is compiled into a bitstream which configures the FPGA directly. This stage is deterministic, and if the preceding stages completed successfully, this one typically will complete without any issues as well.

The Fourier Transform is a mathematical technique which transforms data between the 'time domain' and the 'frequency domain'. The time domain represents signals as magnitudes with respect to time: it is the form in which humans most commonly interpret information. The frequency domain, however, represents signals as their magnitude and phase, with respect to frequency. The frequency domain is useful for analyzing signals with many different spectral components, because their amplitudes can be viewed directly, even when the time domain signals are intermixed and hard to notice. When used in a discrete setting, the Fourier Transform becomes the Discrete Fourier Transform (DFT), which has very similar properties.



Several Fourier Transform pairs: The 'time' domain examples on the left are equivalent to the 'frequency' domain graphs on the right.

The DFT, happens to be extremely slow, requiring O(N^2) operations for an N-point DFT. This is unacceptable when trying to run the algorithm quickly with N >= 128. In 1966, two mathematicians named Cooley and Turkey released the Fast Fourier Transform, which allows the FFT to be computed in O(N * log(N)) by taking advantage of "divide and conquer" methodology. A variant of this algorithm is used in our proposed digital spectrometer.

Unfortunately, the frequency response of each output channel from the FFT is actually rather poor. The response sags near the edge of the channel, and output channels respond to some frequencies near their channel but outside the intended pass-band. By adding a low-pass filter before the FFT, this frequency response can be improved greatly. This combination of FFT and low-pass filter is known as a

Polyphase Filter Bank (PFB). Adding these filters is often challenging, however, because quality filters are hardware expensive.

Assisting us in the design of this spectrometer is the Center for Astronomy Signal Processing and Electronics Research (CASPER), who collaborate to produce tools used for the rapid production of DSP tools. Their toolset allows for the automated generation of the hardware descriptions of several prominent DSP algorithms. Their tools also process the generated model through all the steps listed above, all the way to bitstream generation with the press of a single button.

III. Objectives

The objective of this internship was initially fairly ill-defined, because it was impossible to know what was practical until some progress had been made on basic improvements. Over a previous semester another intern, Suraj Gowda (UCB), performed similar work, creating an FFT-based digital spectrometer with 512 channels which performed admirably. One goal of the current internship was to expand on this design by improving the output resolution (in both amplitude and frequency) and maximum number of accumulations. Additional goals included adding a PFB filter to the design and making the firmware compatible with an alternative, more compact FGPA/ADC board, produced by Nallatech.

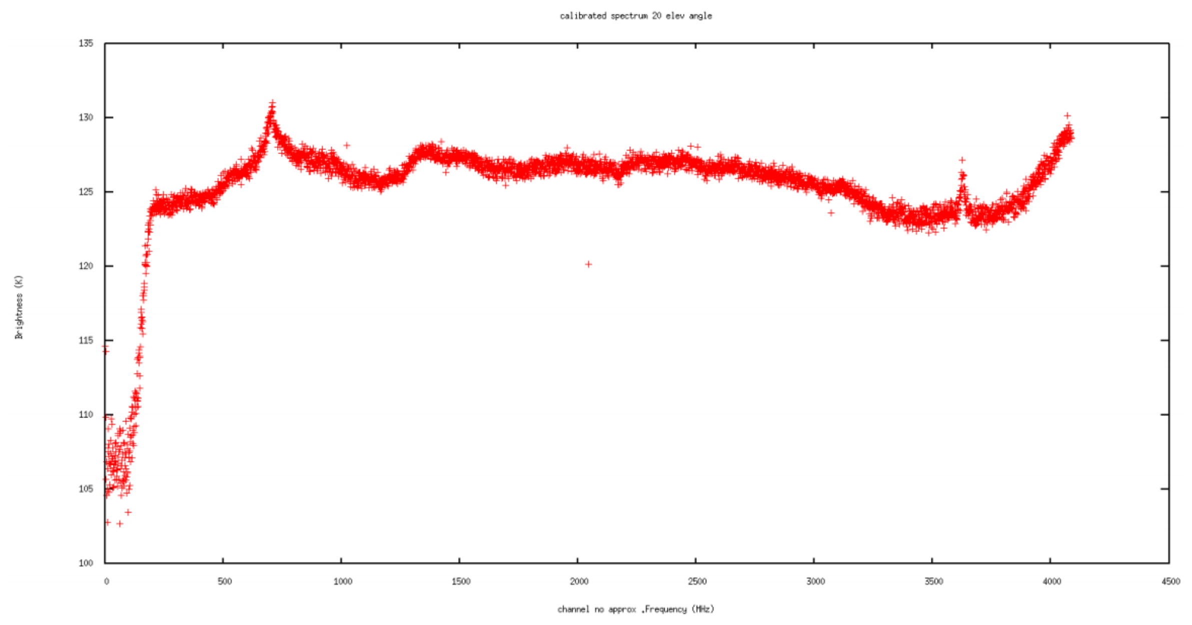
IV. Approach

A. Enhance the present spectrometer

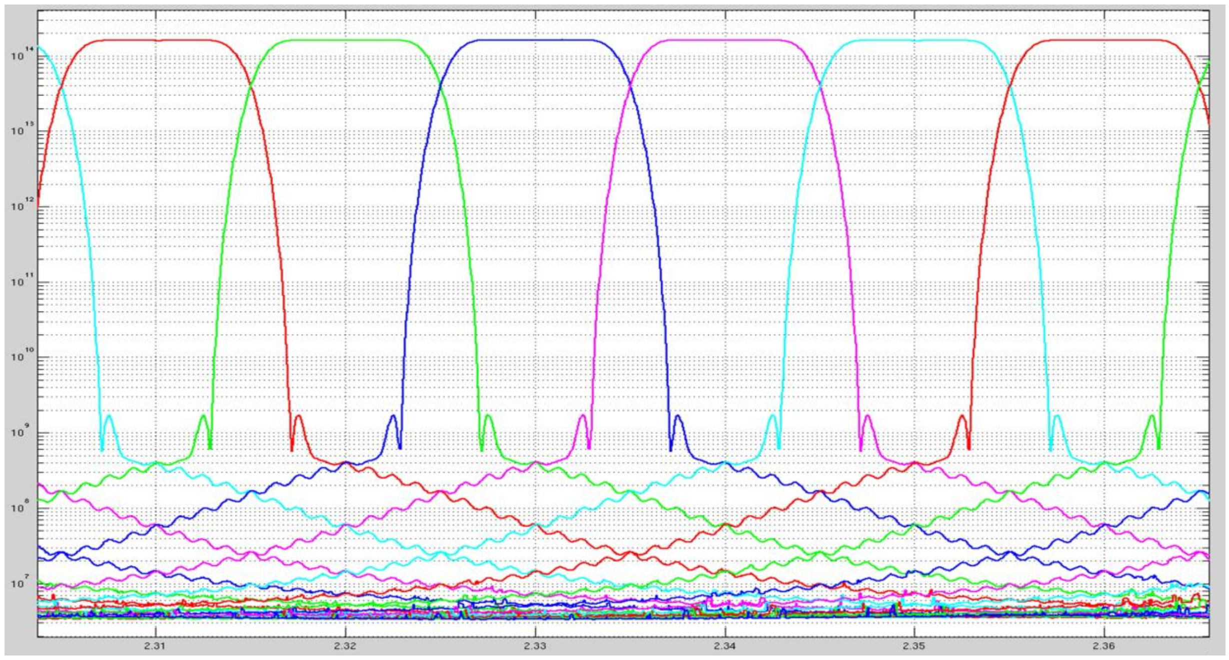
Prior to the start of my internship, Suraj Gowda had already designed a working spectrometer. My task was to continue his work and further improve the tool. The first task was to improve the logical design of the algorithm by replacing inefficient fabric operations for multiply and add operations with DSP48E blocks of the Xilinx Virtex 5 FPGA, which are well designed to manage math-heavy DSP operations. In addition, extra storage elements were added to the end of the design in order to allow superior output resolution and accumulation lengths.

Achieving a working spectrometer was a great struggle. While the tools are innovative, they are also still in their infancy. Changing any parameters in CASPER Library blocks caused the hardware contained within those blocks to be corrupted in an unpredictable fashion. In addition, simulation often took nearly an hour, resulting in a segmentation fault (thus crashing MATLAB) about half of the time. Because of these problems, the MATLAB design was placed on hold for a period of about a month, while the design was re-created directly as hardware in Xilinx ISE (the FPGA vendor design environment). Ultimately, however, that design was discarded in favor of a return to the original Simulink design. Using the knowledge gained from my experience in ISE, I returned to the MATLAB/Simulink design and proceeded production from there. After locating a flaw in the constraints for the design, a working spectrometer was achieved and efforts to improve the spectrometer continued.

As more aggressive spectrometer designs were created, designing the hardware to run at a sufficiently high clock rate became progressively more difficult. These issues were mitigated by duplicating hardware and adding (or removing) latency as necessary. The floorplanning of the design was changed dramatically from the original provided by Suraj, which proved inefficient for larger designs.



Spectrometer output, showing actual atmospheric data near 230 GHz captured from Table Mountain using an SIS (superconducting) microwave radiometer. The large peak to the left represents the Ozone line, while the smaller peak to the right is a television station.



Measured channel frequency response from 8192 channel spectrometer

IV. Results and Discussion

The final spectrometer designed (as of the writing of this paper) is an 8192-channel polyphase FFT implementation. Designed with additional output capacity, the spectrometer has superior frequency resolution, dynamic range and accumulation length when compared to previous versions. An alternate, dual-polarization 1.5 GHz, 4096 channel spectrometer is available as well, and also a 3 GHz, 4096 channel version for applications requiring reduced output data rate. All designs are capable of accumulating for hours, several orders of magnitude above what is required (this may actually be considered a design flaw which will be addressed in the next generation: accumulation time can be traded for superior amplitude resolution). The image shown on the previous page represents actual output recorded from Table Mountain Observatory (TMO), using a previous generation (lower resolution/higher noise) spectrometer design.

In addition, a further improved spectrometer with double the frequency resolution, a Polyphase-FIR filter frontend, and substantially reduced noise has been successfully simulated and is presently in the final stages of development. When finished, it will, to our knowledge, be the best spectrometer developed on Virtex-5 hardware in the world with regards to bandwidth as well as spectral resolution. These results are an order of magnitude superior to the capabilities of the analog spectrometers we presently use.

There are several factors which collectively result in a lack of high quality digital spectrometer designs. One major contributing factor is the distinct lack of developers who are knowledgeable in both

DSP algorithms and FPGA design. While creating firmware is possible for a developer who lacks either skill, the results will typically be inferior due to either a lack of mathematical optimization or inefficient use of hardware. In addition, the tools which allow the design of these spectrometers are of surprisingly poor quality: for large sized models we are designing, they crash consistently or change important component attributes arbitrarily.

In order to mitigate those problems, I applied several techniques to segregate complicated parts of the design. These techniques have been outlined in previous sections of this paper, and will be elaborated in the attached (informal) appendix, which describes the techniques in excruciating detail.

VII. Conclusions and Future Work

My endeavor in this internship can be considered a complete success. The tool created can be easily regarded as top-of-the-line, and will likely be used by several teams around the world.

Future work will involve adding features to support these teams, as well as striving to improve noise characteristics further. Adding an additional accumulator as well as supporting logic will allow the accumulator to automatically select between saving output data to one of two accumulators (signal vs reference), or discard the data entirely. This will allow users who need to take advantage of a mechanical chopper to take advantage of my firmware as well, opening its use to another family of potential 'customers'.

This tool is just a stepping stone towards future, even more ambitious projects. Plans to make an 8 GHz bandwidth spectrometer taking advantage of the same technology used for this device are already being made. Finally, efforts are presently being made to interface this design to a compact Nallatech board, which consumes less power and can be more readily used in remote locations and demanding environments, such as an upcoming high altitude aircraft mission flight planned for October 2011.

Acknowledgments

This work was carried out at the Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the SURF program, and the National Aeronautics and Space Administration.

Appendix 1: Optimizing the Spectrometer

The goal of this project was to produce a high resolution with (ideally) excellent channel orthogonality. Because the FPGA chip the design is being tailored to has limited resources, and adding resolution and pre-filters are both hardware-expensive, considerable efforts were made to maximally streamline the algorithm. As the design became larger, more extreme measures were required in order to meet timing. These will be discussed in the following sections.

First-Generation Efficient Butterflies

Since the foundation of most modern FFTs is the butterfly, it was essential that this block was optimized properly. A butterfly design consuming six DSP48Es was implemented and tested, which proved acceptable. This was managed by first computing b*w, by using the standard technique of computing [b*w]_re = b_re*w_re - b_im*w_im and [b*w]_im = b_im*w_re + b_re*w_im. Using the PCOUT and PCIN ports on the DSP48s allowed for savings in fabric, as well as additional DSPs.

After computing b*w, one must find a+b*w and a-b*w. Four real add/subtracts are required to perform these two complex arithmetic operations. Since DSP48E adders are 48 bits wide, while the argument values are only 18 bits wide, it is practical to convert a DSP48E block into two 24-bit adders. Thanks to support by Xilinx for this functionality, the feature can be achieved simply by applying a certain setting and presenting the pairs of arguments to the top and bottom 24 bits of the adder, respectively.

Second-Generation Efficient Butterflies

A UC-Berkley PhD student and former JPL intern, Suraj Gowda, further improved the butterfly by reducing the number of DSP48E's to five. He instead computes a+bw = ((a_re - b_im*w_im) + b_re*w_re) + j((a_im + b_im*w_re) + w_re*b_im), and a-bw = 2a - (a+bw). Unfortunately, the benefit of this modification is limited in many areas due to several other hardware bottlenecks, and therefore has not yet been fully implemented.

Sync management

The original Casper design uses a naive implementation for the 'Sync' signal, which travels with the same latency as the data path, acting as a 'valid data warning' signal and resetting coefficient counters as well as the other logic needed to run the FFT. Throughout the original design, at any place where there were multiple 'sync' signals being released and later multiple signals collected and used, the design would drop all but one of the output signals, and duplicate the one signal which was passed, to allow it to serve the next stage in the design. While this does provide the same logical functionality, Xilinx place and route algorithms do a poor job at managing the fanout and routing considerations of that one sync pulse. In particular, they tend not to duplicate the given register, instead simply placing the one instance equidistant to all the recipient hardware. In many locations, where there are up to eight or sixteen destinations for that pulse, occasionally spread to different sides of the chip, which results in extremely poor timing. Two mitigating solutions must be used situationally to minimize the impact of this deficit.

In situations where there are multiple drivers and multiple receivers, it is essential that each driver's signal is actually routed to the pieces of hardware which will ultimately be placed closest to it. This may involve adding extra logical ports to the Simulink model of the hardware to be generated.

In situations where there are several distant destinations for the sync pulse, a binary tree must be added before the area of the design which the sync pulse must serve. Extreme care must be taken that the sync latency remains identical to the data latency, or increased noise and strange errors will ensue. In some cases, such as the beginning of the entire algorithm, the precise latency of the sync pulse is immaterial, because it is not yet coordinating any logic before the beginning of the design. Be aware, however, that in the event that further logic is later added in front of the present beginning of the design, that the binary delay tree will have to be removed to maintain latency integrity.

Hardware duplication

On larger designs, the growing bit-width of various counters, as well as the increased routing demand imposed by the added logic makes meeting timing much more challenging. One of the main driving forces behind the timing problems is wide fanout: counter bits which previously drove just 6 pieces of hardware now drive 11 devices or more. In order to meet timing, it is often necessary to duplicate any counters which are addressing several devices, an act which also simplifies placement greatly.

Naming conventions

After building the netlist, larger designs will also have to be floorplanned to meet timing. Because Xilinx System Generator appends randomized prefixes to the hierarchy in order to ensure name uniqueness, finding or identifying hardware elements can become difficult. In order to make identifying hardware elements as simple as possible, I strongly recommend maintaining consistent naming conventions, such as identifying any DSP48E blocks which are serving as multipliers with a 'dsp48_mult' prefix.

On Hardware Implementation of Simulink Designs

The Virtex-5 FPGA has several idiosyncrasies which must be attended to in the use of their designs, especially regarding System Generator. Those will be discussed here.

All delays implemented using Xilinx System Generator blocks are implemented using one SRL16 per bit. A single SRL16 block is an FPGA hardware element which supports delaying a single bit up to 16 clock cycles. This has two important consequences: First, after adding a single delay element to a location in hardware, you may add up to 15 more at no additional hardware cost. Second, if you are trying to improve routing, adding a multi-cycle delay will not improve the routing of your process because those multiple delays will typically be implemented as a single SRL16. If multiple latencies are needed for routing concerns, one must use several delays of once clock cycle a piece explicitly. In addition, a delay with a latency of ‘1’ is implemented as a single ‘D-Flip Flop’, which (according to Xilinx) has a lower setup time than an SRL16. Keep that in mind in the use of any adjustments for timing-based optimizations

If you are using DSP48E blocks with the PCOUT/ PCIN functionality, be aware that long (4 or longer especially) chains of DSP48E blocks will interfere with placement and routing, and will probably require manual placement for good results. Breaking the blocks up into groups of 4 or fewer may be prudent.

DSP48E blocks provide a power-inexpensive tool for performing math operations, when compared to fabric. Using every DSP48E block on the chip will consume between 2 and 3 watts. If you are trying to conserve power, be sure to mark any add/subtract only blocks as such in the Simulink settings, as this will reduce power consumption considerably.

Block RAMs (distributed FPGA RAM elements) are implemented as 18 bit wide, 1024 element deep memories. Keeping these values in mind will allow the designer to take maximum advantage of the hardware. Because Xilinx allows their memories to be used in a half-width / double-depth mode, wider memories are implemented as numerous RAMB18s, each serving a fraction of the required bits. For instance, an 18 bit wide, 2048 element deep memory will be implemented as two 9 bit wide, 2048 element deep memories.

Optimizing the Polyphase Filter Bank

One disadvantage to a FFT is the considerable leakage to adjacent channels when an input frequency does not lie perfectly in the center of a channel. Adding a Polyphase Filter Bank (henceforth abbreviated 'PFB') allows for the reshaping of the FFT channels into more 'ideal' shapes with evener pass bands, weaker side lobes and a steeper and deeper roll off. In a naive implementation of a PFB filter with N taps, each of the simultaneous inputs will require one multiply per tap (N multipliers), as well as one add and one delay for each tap past the first (N-1 adds and N-1 delays, implemented in Block RAM). It will also require a coefficient block for each tap (N Block RAMs). Adds and Multiplies are both implemented in DSP48E blocks, while both delay elements and coefficient storage are managed by Block RAM blocks. Therefore, we can add the costs of these respective components. Consequently, each simultaneous input to the filter bank will require <2N-1> DSP48E blocks and <2N-1> Block RAM blocks. For input vectors longer than 1024 elements, the Block Ram consumption is increased dramatically. This means that it is difficult to make a PFB filter with more than four taps for spectrometers with 4096 channels or fewer, and almost impossible to make any PFB at all for larger spectrometers. The following section discusses techniques explored to improve the efficiency of the PFB.

Halving adder consumption

Re-using the same technique for halving the required number of DSP48E blocks used for small adds, the number of DSP48E blocks used in the system can be reduced to <N + ceil((N-1)/2)> with no further hardware cost.

Removing adders completely

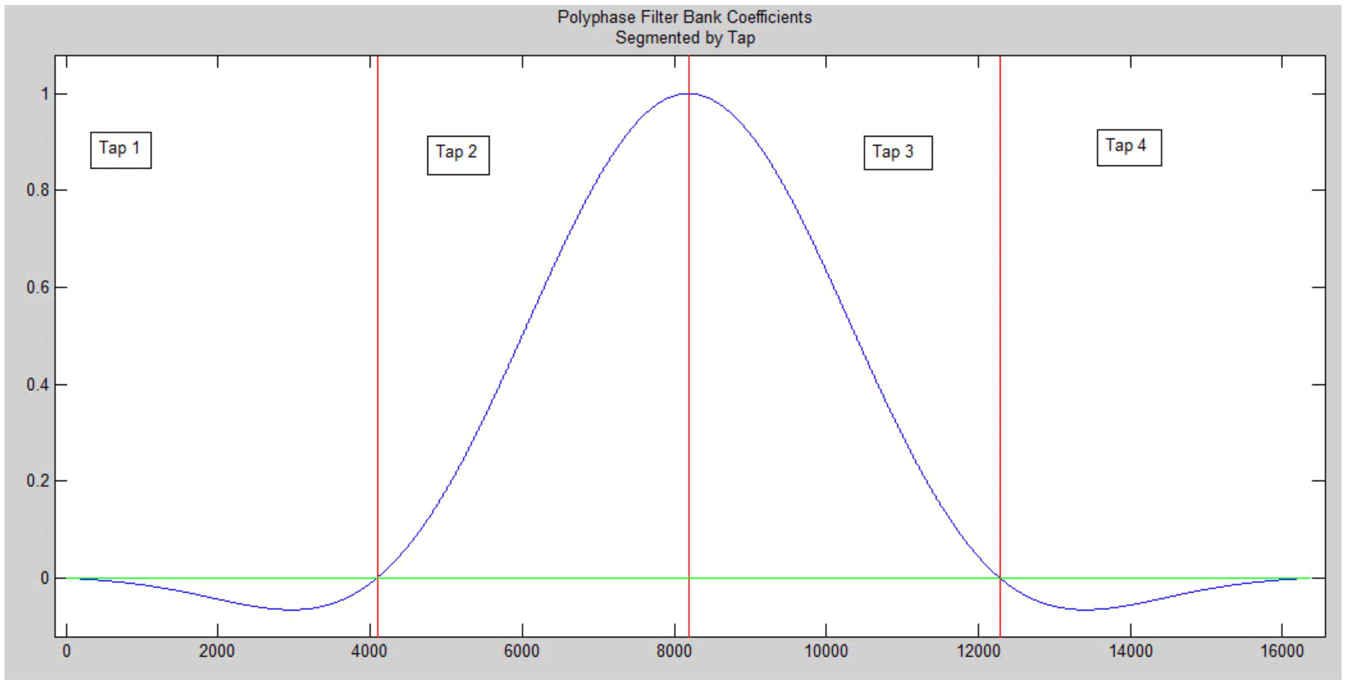
For a small additional price, however, the adders can be completely removed. Because each DSP48E block supports a multiply followed by an add, the PCOUT ports of the first multipliers can be attached to the input ports of subsequent taps. If the input data and coefficient blocks are each delayed by an additional clock cycle, the final accumulated result will be provided by the output of the final tap. For larger designs, this will impose a large fabric cost in the form of SRL delay components. Half of these can be removed by simply moving the coefficient delays to the reset port of the counter which addresses the coefficient BRAM, which negates the necessity of delaying the output ports of the coefficient blocks. This reduces the DSP48E cost of the PFB filter to <N>

Reducing delay element Block Ram consumption

Each Block Ram element contains space for up to 1024 elements, as well as two ports which can be used for both reading and writing independently. For designs with vectors of 512 and fewer elements (spectrometers with 4096 or fewer channels), the second ports on these Block Ram elements can be used for a second delay element, saving half of the Block Ram hardware for 'free'. This reduces Block Ram consumption to <ceil((N-1)/2)> (designs with vectors of 512 elements or fewer) and <N-1> (designs with vectors of 1024 or more). By cleverly grouping input signals and using the optimal number of bits per port, a further 10% savings can be incorporated. This final optimization adds the additional constraint of requiring input pairs to be placed in a monotonically increasing or decreasing manner.

Halving coefficient Block Ram consumption

Consider the impulse response of the PFB filter: It is a windowed <sin(x)/x> function. When centered on zero, this is an even symmetrical function. In practical implementations, however, this is shifted such that the first element of the filter is at time t=0. As a consequence, a naive implementation of the PFB filter stores each coefficient twice: for a N-element filter (1-indexed), elements 1 and N are identical, as are 2 and N-1, et cetera. In a simple design with only one simultaneous input, this means that the final tap's coefficients are just the initial tap's coefficients read in reverse order.



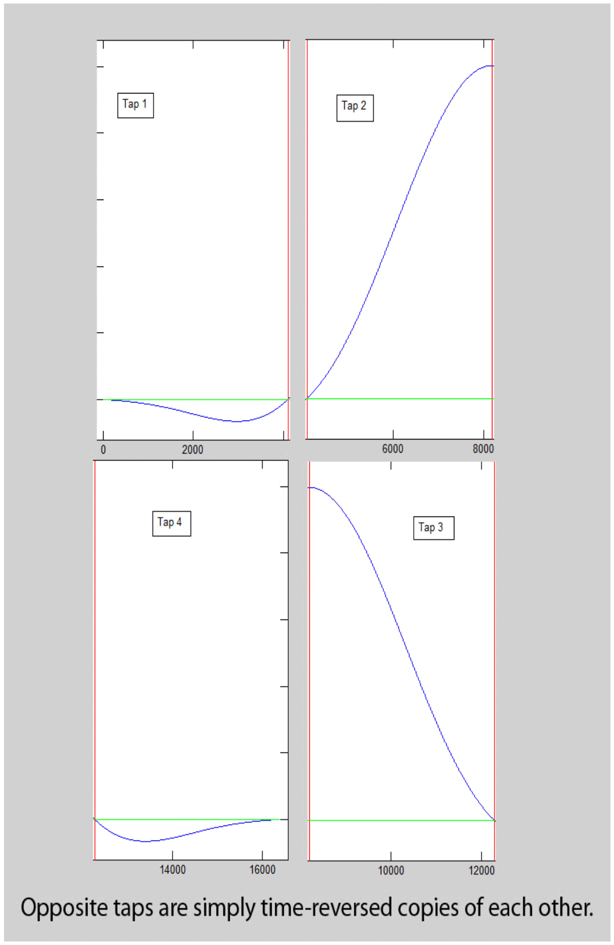
Four-tap Polyphase Low-Pass Filter, split by taps

Unfortunately, this technique is complicated when the input signals are presented simultaneously. When multiple signals are presented simultaneously, the coefficient values for the second (duplicate) half of the spectrum are available only on a different simultaneous input. If there are X simultaneous inputs, the Nth input will find its duplicate coefficients on the X-N'th simultaneous input. Consider the first coefficient on the first tap of the first simultaneous input (the very first coefficient in the PFB): this will be equivalent to the final coefficient on the last tap of the last simultaneous input (which will hold the very last coefficient in the PFB). If there are an odd number of taps, no savings will be available for that tap, since it will not have a duplicate available elsewhere (that data will be contained within the second half of the same Block Ram). This means that the modification is best applied to PFB's with even numbers of taps.

This technique has serious implications regarding the hardware efficiency of the PFB. It will reduce the number of coefficient Block RAMs to <ceil(N/2)>, without any additional hardware requirements. Since the taps will have to be paired in an inconvenient manner, however, there may be routing costs imposed by using this design. It is my professional opinion that these challenges will be worth the hardware benefits which are provided.

Summary (PFB optimizations):

Using these four techniques together results in a much more streamlined PFB implementation. Collectively, they reduce the cost of an even-tap-count PFB from <2N-1> DSP48Es and <2N-1> Block RAMs down to <N> DSP48Es and (512 length vector or smaller) <N+1> or (1024 length vector or greater) <3N/2 - 1> Block RAMs. Practically, this equates to a 50% increase in PFB taps for designs with vector lengths of 1024 or greater, and 100% increase in PFB taps for designs with vector lengths of 512 or less. Clearly, this allows for designs with excellent channel orthogonality.



Block RAMs can be saved by reading coefficient Block RAMs in reverse order.

Optimizing the FFT

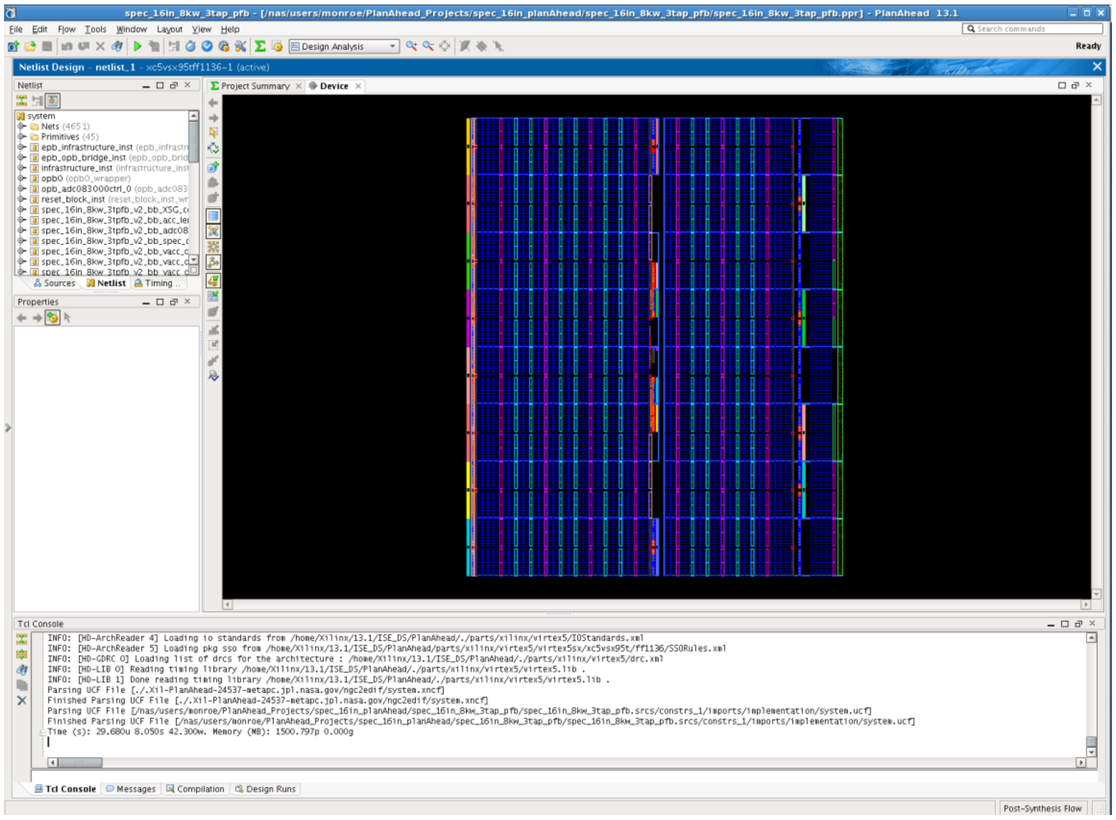
Like the PFB, the FFT can be aggressively optimized to improved performance. By using an optimized butterfly design, about one out of every 6 DSP48Es can be saved. In addition, using dual-port memories to store both the real and imaginary components of the FFT coefficients in a single block allows for further savings of 50% of the coefficient memories. Finally, the same memory sharing technique used in the PFB delay elements can be used to conditionally save further memory. Collectively, the optimizations save about 33% of block memory hardware, as well as 17% of the DSP48Es.

Developed library blocks

All the optimizations described above are implemented in a set of 'blocks', analogous to those found in the CASPER block. Given a set of parameters, these blocks will automatically draw the desired algorithm. They also include several further optional optimizations. These allow for coefficient sharing, which reduces hardware consumption in exchange for increased routing constraints. Optional reduced coefficient sets are also made available. The documentation for the blocks, which can be found in the 'monroe_library', can be found in appendix 2.

Floorplanning the Design

The tools provided by Xilinx to automatically place and route designs perform incredibly poorly when given extremely crowded designs, or designs which are dominated by routing-heavy math operations. By passing physical constraints to the Xilinx tool flow, these results can be improved greatly.



```
INFO: [HD-ArchReader 4] Loading io standards from /home/Xilinx/13.1/ISE_DS/PlanAhead/./parts/xilinx/virtex5/IOStandards.xml
INFO: [HD-ArchReader 5] Loading pkg sso from /home/Xilinx/13.1/ISE_DS/PlanAhead/parts/xilinx/virtex5/virtex5sx/xc5vsx95t/ff1136/SSORules.xml
INFO: [HD-GDRC 0] Loading list of drcs for the architecture : /home/Xilinx/13.1/ISE_DS/PlanAhead/./parts/xilinx/virtex5/drc.xml
INFO: [HD-LIB 0] Reading timing library /home/Xilinx/13.1/ISE_DS/PlanAhead/./parts/xilinx/virtex5/virtex5.lib .
INFO: [HD-LIB 1] Done reading timing library /home/Xilinx/13.1/ISE_DS/PlanAhead/./parts/xilinx/virtex5/virtex5.lib .
Parsing UCF File [./.Xil-PlanAhead-24537-metapc.jpl.nasa.gov/ngc2edif/system.xncf]
Finished Parsing UCF File [./.Xil-PlanAhead-24537-metapc.jpl.nasa.gov/ngc2edif/system.xncf]
Parsing UCF File [/nas/users/monroe/PlanAhead_Projects/spec_16in_planAhead/spec_16in_8kw_3tap_pfb/spec_16in_8kw_3tap_pfb.srcs/constrs_1/imports/implementation/system.ucf]
Finished Parsing UCF File [/nas/users/monroe/PlanAhead_Projects/spec_16in_planAhead/spec_16in_8kw_3tap_pfb/spec_16in_8kw_3tap_pfb.srcs/constrs_1/imports/implementation/system.ucf]
Time (s): 29.680u 8.050s 42.300w. Memory (MB): 1500.797p 0.000g
```

Planahead Interface

The following sections will describe general rules used to produce 'good' results.

PlanAhead

The Xilinx tool provided to add constraints is known as "PlanAhead". This is a graphical interface which displays the architecture of the chip and allows the user to input constraints and floorplan as necessary. The tool also allows the user to analyze previous implementation results.

Floorplanning by Partition Blocks

Partitioning is the most general way to assign constraints. Any components assigned to a P-Block will be ultimately be placed somewhere in that P-Block during the tool flow. P-Blocks may be drawn graphically using the "Draw P-Block" tool. Once a P-Block is drawn, you may assign components to a P-Block by selecting them, right clicking and selecting "assign". You may view the consumption of P-Blocks in the properties window while the P-Block is selected. P-Blocks are most easily selected using the "Physical Constraints" window, which is not available by default and must be selected from the "Window" menu.

Caveats on P-Blocks

It is necessary that each P-Block contain enough resources to support all the hardware contained within (hardware assigned to a P-Block, but manually placed elsewhere does not count). Failure to do so will cause the "Percent in Use" metric in the property window to exceed 100% and the design will produce an error during implementation. Solve this problem by removing hardware from the P-Block or increasing the size of the P-Block

By default, each assignment to a P-Block is produced as a separate constraint. Since System Generator does not produce hierarchical designs, very large numbers of components must be designed to a P-Block in order for them to prove useful. Xilinx tools take a very long amount of time to read large numbers of constraints, so this is not recommended. Anecdotally, placing 50,000 elements into various P-Blocks took over seven hours to implement.

The answer to this problem is to use wildcards: all Xilinx tools support the '?' wildcard, which may be replaced with any single character, and the '*' wildcard, which may be replaced with any other string. If these are listed in instance, port or clock names, the constraint will apply to any components which match that wildcard string.

Hand-Placing Components

Hardware may also be placed manually. By selecting the "Make BEL Constraint" or "Make Site Constraint" tools, components may be forced to specific locations. These placements typically supersede any other constraints on the components. To place a "Site" or "BEL" constraint, locate the piece of hardware to be constrained, select either tool and drag the hardware to the desired location on the chip. It is important to realize that since "Site" and "BEL" constraints are both absolute and fixed, they must be extremely well placed, or the final output will have absolutely terrible timing results.

General Floorplanning Instructions

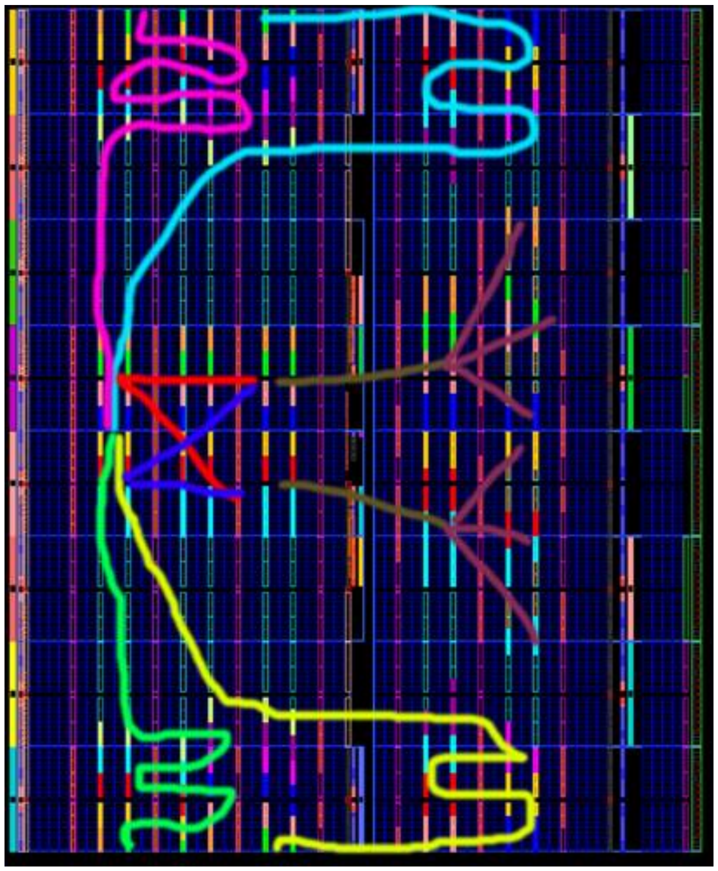
Before starting floorplanning, allow the tools to generate a naive implementation automatically. This will allow you to determine how distant the design is from meeting your timing requirements. After running an initial implementation run, load your project into PlanAhead and import your Placement and Timing results in order to get an inkling of the actual hardware consumption imposed by the chip. It is now time to begin floorplanning your design. I recommend printing out several copies of your chip architecture and drawing your floorplans before actually implementing it. Once a high level floor plan is conceived, begin floorplanning the most regular sections. These are not only the regions which are easiest to floor plan, but often the ones which Xilinx tools perform worst at automatically placing. After building an adequate floor plan, open your <project_name>/XPS_Roach_Base/data/system.ucf file and append your component placement constraints onto the end of the file (it is important NOT to append any port or timing constraints, as they may cause errors or override the (correct) constraints provided by

BEE_XPS). Afterwards, re-run BEE_XPS, selecting only the EDK/ISE/Bitgen option. In the event that the process fails in only seconds, it is possible that ISE interpreted the entire design as "up to date", concluded that your design still did not meet timing, and returned the same results as your previous run. If you believe this is the case, re-run BEE_XPS on IP Creation up to (but not including) EDK/ISE/Bitgen. This will reset your results. You may now update your system.ucf file and run EDK/ISE/Bitgen.

If your design still does not meet timing, begin by re-importing your updated placement and timing results. Identifying troublesome paths is made easiest by selecting them, right clicking and highlighting them. Only the actual failing paths will be highlighted. Use these to analyze the causes of your timing failure. Repair any flaws in the original floor plan, and constrain more elements if necessary. In the case that the placed DSP48Es and BRAMs appear good, but the design still fails timing, use P-Blocks to restrict the fabric logic to local areas on the chip.

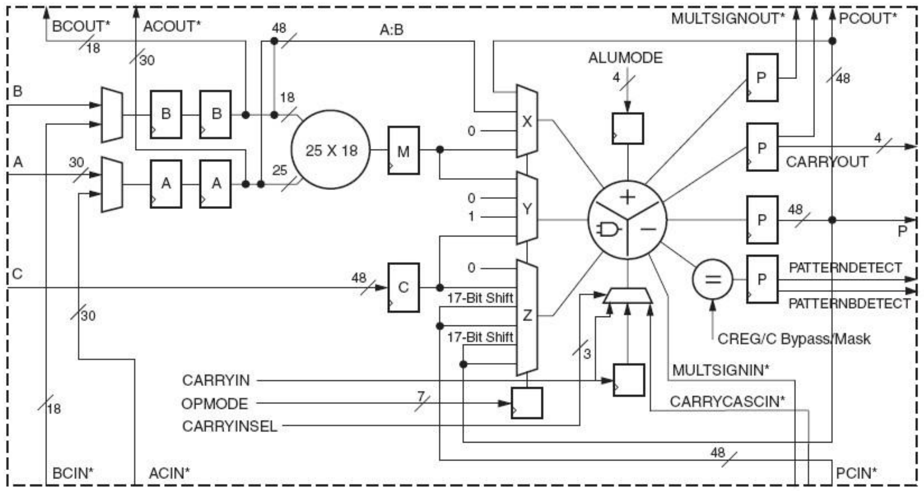
General Floorplanning Advice

Special efforts should be made to pay attention to routing, which is a "hidden resource". Remember that every signal which is generated must be passed through an invisible routing matrix, which has finite resources. While the availability of this routing is uniform throughout the chip, routing demands are typically far greater near the center of the chip. This means that good designs will try to distribute signals across the edges of the chip, as well as through the middle. Anecdotally, it appears that vertical and horizontal routing resources are somewhat independent. It appears to cost relatively little to route a signal through a routing-dense environment when most of the already-present signals are travelling east-west, and the new signals are travelling north-south.


Actual Spectrometer Floorplan, data-paths annotated. Planning datapaths in advance is crucial.

Be sure to consider the locations of the input and output ports of the chip when floorplanning. The signals must inevitably travel through these ports. In the ROACH 1, the ADC I/O ports are all situated on the West side of the chip, while the Software Registers and other BEE_XPS I/O logic is found on the East side of the chip.

In addition, both the Block RAM and DSP48E blocks have their input ports on the West side of the components, and the output ports on the East side of the component. Due to both the ROACH port locations and the internal Block RAM / DSP48E ports, it is advisable for the majority of designs to use a predominately West-to-East design.



DSP48E slice diagram. Note that fabric signals enter from the left and exit to the right, indicating a West-to-East datapath.

Regarding Block Slices

BRAM and DSP48E components appear to have an extremely weak driving potential. Placing their destination components nearby or buffering the DSP's output has a very substantial result on the maximum running speed of the design. DSP48E components appear to suffer more severely from this problem than BRAMs.

DSP48Es are designed such that the 'P' port of a given chip (the 'P' port is on the right side of the component) lines up bitwise perfectly with the 'C' port on the DSP48E block immediately to its left (the 'C' port is on the left side of the component). Placing so connected components in this manner will produce extremely routing-efficient designs. Note that this trick is only advantageous if there are not intervening BRAMs or other gaps in the chip between the adjacent DSP48E components (a small amount of fabric, however, is acceptable).

BEE XPS Toolflow

The steps of the BEE_XPS toolflow and their (apparent) actions are as such:

1. Update Design (runs all scripts on all CASPER blocks in the design, appears to be optional if a simulation has just been run). Very slow.

2. Design Rules Check (checks for BEE_XPS block and XSG block, ensures there are no extra 'gateway in' and 'gateway out' blocks)

3. Xilinx System Generator (runs System Generator. Produces VHDL equivalent to the model and synthesizes it). Very slow.

4. Copy Base Package (overwrites the contents of <model_name>/XPS_Roach_Base> with <casper_library/xps_lib/XPS_Roach_Base)

<<The remaining steps are performed in the Xilinx XPS tool>

5,6,7. IP Creation/Synthesis/Elaboration (adds CASPER yellow blocks as pcores into the XPS project, configures and synthesizes them.

8. Software generation (parses the included constraints files and adds constraints as necessary). Options 5-8 are poorly understood.

9. EDK/ISE/Bitgen (synthesizes and implements the finished design. Generates placement, timing and bit files into XPS_Roach_Base/Implementation.

10. JTAG Download (unused in ROACH builds)

Failing Timing

If your design does not meet timing, the design will error out before building a bit file. Often a design will function even if it fails a static timing analysis. If you are prepared to take this risk, follow these instructions to override the timing error: open up an instance of Xilinx XPS (command 'xps'). Select 'open recent project / browse for more projects'. Navigate to <model_name>/XPS_Roach_Base/'. Your project will be the only file. Open up 'Project Options', locate the 'Treat Timing Closure Failure as Error' setting and disable it. Close the options dialog and exit XPS. Re-run BEE_XPS under EDK/ISE/Bitgen. Your project will probably still fail to meet timing, but a bit file will still be generated. **Your mileage may vary**.

Appendix 2: Monroe_library Documentation

Hey all,

I've been pretty busy over here in Pasadena. I've built myself a bit of a personal library on top of the xBlocks scripting language. Please feel free to take a look!

Please note that while these blocks have been tested in a general sense, I do not believe that they are up to the rigor we need to release them to the CASPER community. While they're probably largely functionally correct, I haven't tested every reasonable combination of parameters, and my checking for valid inputs is still a bit spotty.

In addition, while these blocks can use much less hardware than the stock casper blocks, they have not been performance tested. There are many parameters to adjust at the top level, and I'm not quite sure what values are most significant/useful.

Finally, Andrew commented on these blocks being inter-compatible with the stock casper ones and I will have to clarify: that is probably not going to happen. The xBlocks method is very different from the stock blocks, and the drawing functions are called quite differently. I think it would be best if they were separate blocks in the casper library, or if existing software support for the old blocks was maintained, but the new blocks were the only ones that showed up in the "library browser"

Using the monroe_library

In order to use the current version of the monroe_library, extract the files included to any folder and add both the 'monroe_library' folder and the 'monroe_library/xblock_helper_functions' folder to your path.

All the blocks listed below can be found in the monroe_library.mdl file, which is the only way to access monroe_library blocks at the moment.

I have included two copies of one of my spectrometers (redesigned with my new blocks) as a demonstration. They can be found in the monroe_library/demonstration folder. The base design runs at full bit width (but shares coefficients between butterflies wherever possible), while the 'lessbits' design uses less bits / coefficients. For the 'lessbits' design, pol1 is set to use fewer coefficient values in the FFT Direct, while pol0 is set to have 9 bits through the entire datapath (data and coeffs). They may be tested by opening the model, running the simulation, and using the included script to unpack the results. You may have to change the offset at the top of the file, depending on which design/settings you use.

Neither design is intended to be built to hardware using these libraries yet.

Known issue

Sometimes, when changing the parameters of a block, it does not immediately re-draw. I don't know why this is, but if you right-click on the block and choose 'look under mask', it will re-draw.

Know that this library has been developed and tested entirely in Matlab r2010b, with Xilinx ISE 13.1.

Index of blocks

(top level-ready blocks are green)

Misc

oneSync
counter_limited_fast
bit_reverse
cram
uncram

Delays

delay_tree_z^-6
bulk_delay
sync_delay_fast
delay_bram_fast
double_delay_bram_fast
double_delay_bram_external_counter
double_delay_distrib (unsupported)
multi_delay_bram_fast (super proud of this block)

Math

dsp48e_mult
dsp48e_mult_add_pcin
dsp48_negate_dual

Polyphase Filter Bank Logic

pfb_coeff_gen_dual
fitst_tap_improved
middle_tap_improved
last_tap_improved
pfb_fir_real

Biplex FFT Logic

twiddle_stage1
twiddle_stage2
twiddle_cheap
biplex_stage
coeff_gen_dual
biplex_coeff_muxsel_gen
biplex_4x_unscr
unscr_tail
FFT_Biplex_4x

Direct FFT Logic

coeff_gen_dual
direct_coeff_gen
fft_direct_stage
FFT_direct

Draw times for blocks potentially superseded by the monroe_library.

Block	Time (casper_library)	Time (monroe_library)	Notes
Biplex FFT	6 minutes, 55 seconds	12.5 seconds	FFTSize = 10
Direct FFT	28 minutes, 26 seconds	24.8 seconds	FFTSize = 4

Polyphase- Filter Bank	32.8 seconds	31.6 seconds	n_sim_inputs = 4
			n_taps = 6

Note that for the direct FFT, the mask script for each butterfly was run five (!!) times. It's possible that my library is out of date and the present casper library version draws up to 5x faster

The biplex FFT also ran its scripts several times, which may be why it's so slow.

I'm not going to document each and every of the blocks here (but I will someday). Here's a thing or two about how to use my blocks:

(any 1-d array can be replaced with a single integer, which will be interpreted as an appropriately- sized array composed entirely of that integer)

pfb_fir_real		
Parameter	Format	Bounds
PFB Size	Integer	1 <= x <= 25
# Simultaneous Inputs	Integer	1 <= x <= 6
# Taps	Integer	x <25
Input Bit Width	Integer	1 <= x <= 25 (recommended max 18)
# Coefficient Bits	Integer	1<= x <=18

Output Bit Width	Integer	x >0
Window Function	Pull-Down	
Channel Width	Real	Positive
Downscale at end	Integer	-1 = default; else, downscale by 2^x
Delay at end?	Boolean	<0 or 1>
Use Cheap Sync Hardware	Boolean	<0 or 1>
Autoplacement mode?	Pull-Down	<not active yet>
Optimize for Autoplacement?	Check-Box	<not active yet>
Bram Latency <coefficients>	Integer	<2 or 3>
Bram Latency <delays>	Integer	<2 or 3>
Register delay counters?	Boolean	<0 or 1>

Comments:

I have a half-finished auto-placing constraints generator for this block. I'm going to be leaving JPL soon, so I probably won't finish it for another month or two.

Settings for auto-placement mode are presently ignored

The 'Register delay counters' setting is probably fine at 0, but I have not tested it. Setting it to '1' may improve performance, but will be extremely hardware-expensive.

Optimizations:

Used dual-port memory for the coefficients, saving ½ the coefficient BRAMs.

Connected all the FIR DSP48E's into one PCIN/PCOUT chain, which removes the need for an adder tree at the end. Saves almost ½ the DSP48Es. Adds one datapath- delay element and one counter -delay element per stage (except the first).

Used the 2-stage pipeline the DSP multipliers to remove the need for the second delay in both the counter and the datapath.

Made a highly optimized self-drawing block for multiple delays. Uses the minimum number of BRAMs for a given number of delay bits. Variable gains in efficiency, based on filter parameters, but typically saves 45%-55% of the BRAMs and ½ the delay counters.

The above multi-delay block uses pipelined counters for high-speed, but drops down to simple (fabric-cheaper) counters when the vector-length is short enough.

Optionally uses a cheaper sync-delay, which uses a SRL to delay mod(totalDelayLen, vectorLen) cycles instead of the full pfb latency. This is good if it is ok for the sync to be offset by several vectors from the start of the PFB

FFT_Biplex_4x		
Parameter	Format	Bounds
FFT Size	Integer	1 <= x

Input Bit Width	Integer	1 <= x <= 24 (recommended max 18)
# Coefficient Bits	Integer Array(FFTSize-2)	1<= x <=18
Output Bit Width	Integer	x >0
Inter-Stage Bit Widths	Integer Array (FFTSize-1)	1 <= x <= 24 (recommended max 18)
Shift after n-th stage?	BooleanArray (FFTSize)	<0 or 1>
Coefficient BRAM Limit?	Integer	x >0
Delay BRAM Limit?	Integer	x >0
Register Coefficients?	Boolean	<0 or 1>
Stage Pre-delay?	Boolean	<0 or 1>; Recommend 0
Stage Mid-Delay	Boolean	<0 or 1>; Recommend 1
Stage Post-Delay	Boolean	<0 or 1>; Recommend 1
BRAM Latency (FFT)	Integer	<2 or 3>
Bram Latency (Unscrambler)	Integer	<2 or 3>
reorder data-in Latency	Boolean	<0 or 1>

Comments:

Bit Growth FFT: To make a 10-stage FFT which grows from 8 bits to 18, simply do the following:

Input Bit width:	8
inter-stage-bit-widths:	9:18
output bit width:	18

You can also do coefficient bit-growth, but I don't recommend using anything other than 9 or 18 bits for coefficients, as there's little benefit to using values in between.

I'm not quite sure if carrying 24 data bits has any benefit over using only 22 or 23. I'll test this later

I have removed the option to change the down-converts at each stage to use rounding or saturation. These were *very* fabric expensive, and in my opinion, not worth it. Especially because they make my design more complicated and draw more slowly.

Optimizations:

Used dual-port BRAMs for coefficients, halving coefficient bram usage for stages 3-10. Also saves on counters.

Moved all the coefficient generation and mux-select generation to a single coefficient/muxSel generation block, so that in the future we can share those values between several biplex FFTs.

Used dual-port BRAMs for delays, halving bram delays for delays of 512 cycles or fewer. Also saves on counters.

Used Suraj'es 5-DSP butterfly trick (thanks Suraj!) to reduce DSP use by 16%

Also connected stray DSPs with their PCIN/ PCOUT ports. They don't actually use the data bussed through the port, but Xilinx infers it as a useful connection, and places them in a column anyways! I'm pretty proud of this :-)

Made several fabric optimizations, which won't all be listed here.

Instead of using a separate counter for the Muxes in each biplex stage, I made a single counter, and used progressively fewer bits on each stage.

There was another counter driving the mux in stage 2: drove this with an inverted (and delayed) version of the mux signal already being delivered to stage 2.

In the unscrambler:

Consolidated the two early reorders, and shared the BRAM. Also made a special bit-reverse block, removing the need for a 0-latency distributed memory.

Likewise replaced the distributed memory-(being addressed with an upcounter) with a down-counter, saving more hardware

replaced the delays with half-hardware double-delay blocks

Replaced the complex conjugate blocks with half-hardware equivalents

Duplicated two counters that kept the design running below 375 MHz

To Do:

Add a feature to limit the number of BRAMs being used for a given biplex stage (or, equivalently, add an adjustable ceiling to the number of brams) this is pretty easily implemented, but didn't make it into this test. This feature is already implemented in the FFT_Direct, but didn't quite make it into this release.

(if there is demand) Add a feature to share coefficients and mux-selects between biplex FFTs. Perhaps make a fft_biplex_x8 block for it.

Add an option to remove the PCIN-connection between the A-BW DSP and the rest of the butterfly. This feature is helpful for unconstrained designs, but it actually hurts for hand-placed designs or RLOC-ed designs.

Improve the error-checking and warnings for bad input parameters. I think my blocks might be a bit hard to use right now.

(semi-near future): Add RLOCs to constrain each butterfly into its own box. I have a pretty well thought out plan for how this will happen, including the exact values the constraints should take. I won't be starting this until everyone's perfectly happy with the unconstrained block though.

^^^ doing this will change a lot of things. Many of the parameters that are presently settable will no longer need to be, as many delays are only there because bits of the butterflies are placed far from each other. I imagine we can save massively on fabric by doing this, and then use BRAMs for most of the delays and coefficients (since they're going to be well placed) to save even more on fabric.

(distant future): fully auto-placing biplex blocks, given that the block is either:
a. attached directly to one of the ADCs, or
b. attached directly to a PFB (which is attached directly to one of the ADCs)

FFT_Direct		
Parameter	Format	Bounds
FFT Size	Integer	1 <= x <= <larger_fft_size>
Larger FFT Size	Integer	<fft_size><= x
Input Bit Width	Integer	1 <= x <= 24 (recommended max 18)
# Coefficient Bits	Integer Array(FFTSize-2)	1<= x <=18
Inter-Stage Bit Widths	Integer Array (FFTSize-1)	1 <= x <= 24 (recommended max 18)
Output Bit Width	Integer	x >0
Delay Input	Boolean	x >= 0; Recommend 1
Inter-stage Delays	Integer Array (FFTSize-1)	x >= 0; Recommend 1
Delay Output	Boolean	x >= 0; Recommend 1
Add Sync Tree to Input?	Boolean	<0 or 1>
Register Coefficients?	Boolean	<0 or 1>
Coefficient Grouping Array	<read below>	<read below>
Coefficient Step Size	Integer Array (FFTSize)	x >= 0; see below
Shift after n-th stage?	BooleanArray (FFTSize)	<0 or 1>

Comments:

I actually haven't tested the new direct FFT for anything other than FFTSize=3. Worth considering.

Bit Growth FFT: Like the biplex FFT, this also supports bit growth. Coefficients too (but again, I recommend 9 or 18 bit coefficients.

Again, I'm not quite sure if carrying 24 data bits has any benefit over using only 22 or 23. I'll test this later

As in the biplex FFT, options to change the down-converts at each stage to use rounding or saturation have been removed.

Add Sync Tree to Input:

The high fanout for sync on the first stage of the direct FFT caused a bit of a performance bottleneck. By adding a binary tree to the input, this problem is resolved. Using said binary tree restricts the FFTSize to 6 (as if anyone's actually going to use one that large). If this actually becomes a problem, we can extend the binary tree to be larger. Remember that unused parts of this tree will be pruned by the synthesizer.

Coefficient Group Array (AKA Butterfly Coefficient Sharing):

For the first several stages of the direct FFT, the coefficients for adjacent butterflies are the same. Hardware can be saved by using a single coefficient generator for many butterflies. This setting allows you to implement that.

The coefficient grouping array allows you to build 'groups' of butterflies, which will all be serviced by the same coefficient generator. There are no error checks regarding 'safe' arrays (which will not be delivering the wrong coefficients to a butterfly), so make sure your array is correct! A coefficient group array is composed of FFTSize number of rows, each 2^(FFTSize-1) wide. Each row represents a stage (top row is the first stage, second row is the second, etc). In a given row, butterflies with the same group number will receive their coefficients from the same piece of hardware. Group numbers must span from '1' to 'N', where N is the number of groups desired. Having the same group number for butterflies on different stages is OK; stage numbers are only considered within that given stage. If an empty array is provided (looks like this: []), the default casper-equivalent will be used.

Examples of group arrays for a FFT_Direct of FFTSize=3:

<identical to the stock casper design, maximum hardware consumption>

1	2	3	4
1	2	3	4
1	2	3	4

<also identical to the stock casper design, maximum hardware consumption>

4	3	2	1
3	2	1	4
1	2	4	3

<Absolute minimum hardware consumption>

1	1	1	1
1	1	2	2
1	2	3	4

(this will cause low hardware consumption, but high fanout and routing costs will limit clock rate)

<bad array; delivers incorrect coefficients to butterflies... first row OK, 2nd and 3rd broken>

1	1	1	1
1	2	1	2
1	1	2	2

<invalid array; incorrect size (causes error)>

1	1	1	1
1	2	1	4
1	1	2	2
1	2	3	4

<invalid array; incorrect size (causes error)>

1	1	1	1	1
1	1	2	2	3
1	2	3	4	5

Coefficient Step Array (AKA 'discard half the coefficients'):

For large FFT Sizes, having the full vector of coefficients is somewhat overkill. In these circumstances, it can be advantageous to reduce the size of the vector by a factor of 2^N, saving memory in exchange for increased noise. For manually floor-planned or auto-placed designs, it is recommended that a good coefficient group array is used to minimize hardware consumption first. Each value above zero that an integer receives in this array will half the number of coefficients used. Please consider that there are no hardware gains from reducing the number of coefficients below 2^9 (which can fit in a single BRAM18). Examples are shown below.

Imagine a 16-element coefficient array, to be used in a direct-fft. The full coefficient list looks like this:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

These coefficients will be presented to the butterfly in order, from 0 to 15, repeatedly. This is what you would expect with the stock casper blocks. It is also what happens in my block if the step array is a '0' for that stage.

If the array is incremented to a '1' for that stage, the coefficients stored now would look like this:

0	2	4	6	8	10	12	14

And they would be presented to the butterfly in this pattern:

0	0	2	2	4	4	6	6	8	8	10	10	12	12	14	14

Because adjacent coefficients will be similar, this crudely emulates the original coefficient vector. As you can see, considerable memory savings can be achieved in this manner.

Optimizations:

The optimizations used here are largely cannibalized from the biplex FFT. There was not quite as much to optimize away from this design.

I used the same cheaper butterflies as in the biplex FFT. Thanks again, Suraj!

I took the mandatory delays at the beginning and end of the improved butterfly and folded them together in the outside of the direct FFT. The first delay element for each stage also goes here. This means that it is highly recommended to have at least one delay between each stage of the FFT_Direct, since it only adds hardware to 1/4th of the datapath, and will greatly simplify placement.

To Do:

Add an option to remove the PCIN-connection between the A-BW DSP and the rest of the butterfly. This feature is helpful for unconstrained designs, but it actually hurts for hand-placed designs or RLOC-ed designs.

Allow for a 'biplex direct FFT' block/option, which shares coefficient values across two direct FFTs

(near-ish future) As before, add butterfly-wise RLOC constraints

Appendix 3: Spectrometer Documentation

Dual Polarization, 4096 Channel, 1.5GHz Spectrometer on ROACH

User Manual

Overview:

This design accepts input from two ADCs (running at no more than 3GSPS each) and computes a 4096-point, 8-tap PFB FFT on the data. It subsequently accumulates the data and releases it to the user via the shared BRAM interface.

Usage:

In general, this design follows most of the conventions of CASPER spectrometers. There are a few important features to note:

Sync Pulse: Because this design is meant to be used with very long accumulation lengths, the sync pulse is fully managed within the design. There are no user-available software registers to manage the sync generation.

Starting/Stopping accumulation: By default, the design will not automatically accumulate. In order to start an accumulation, set the 'start_acc' software register to '1' and reset it back to '0'. This can also be managed via a GPIO pin, as mentioned below.

Continuous accumulation: By setting the 'cont_acc' software register to '1' while an accumulation in in progress, the spectrometer will continuously accumulate new vectors and update the shared BRAMs accordingly. This was originally designed as a 'development' feature and was never intended for general use, so use at your own risk.

Accumulation Length: The design features a configurable accumulation length, via the software register 'acc_len'. Accumulations of up to 2^32 vectors long are supported, but output values are subject to overflow. A conservative worst-case estimate gives a maximum accumulation length of 200,000 (about 0.5 seconds). A noise signal can run for considerably longer. Please note that the vector accumulator does not reset on changes in accumulation length. As a consequence, you should always restart the accumulation after reducing accumulation lengths, or your design will appear to be locked up (it will, in fact, be running through the full 2^32 vectors)

GPIO pins: Several GPIO pins have been conscripted as an easier way to send signals between a computer controlling the ROACH and external hardware. GPIO port A has been assigned to 'Output', and has pins 0 through 5 (inclusive) available for use. Setting software register 'to_gpio_a' with an appropriate bit pattern control those pins. Likewise, GPIO port B has been assigned as an 'Inport' register. Since two pins are already in use for the accumulation control, only three pins are available for input: 2, 3 and 7. Inputs will appear on 'from_gpio_b'. Note that there is a plastic header covering pin B-7 on my board; I did not test this pin, use at your own risk.

<value to assign to 'to_gpio_a'> = (a0) + 2*(a1) + 4*(a2) + 8*(a3) + 16*(a4) + 32*(a5)
<value appearing on 'from_gpio_b'> = (b2) + 2*(a3) + 4*(a7)

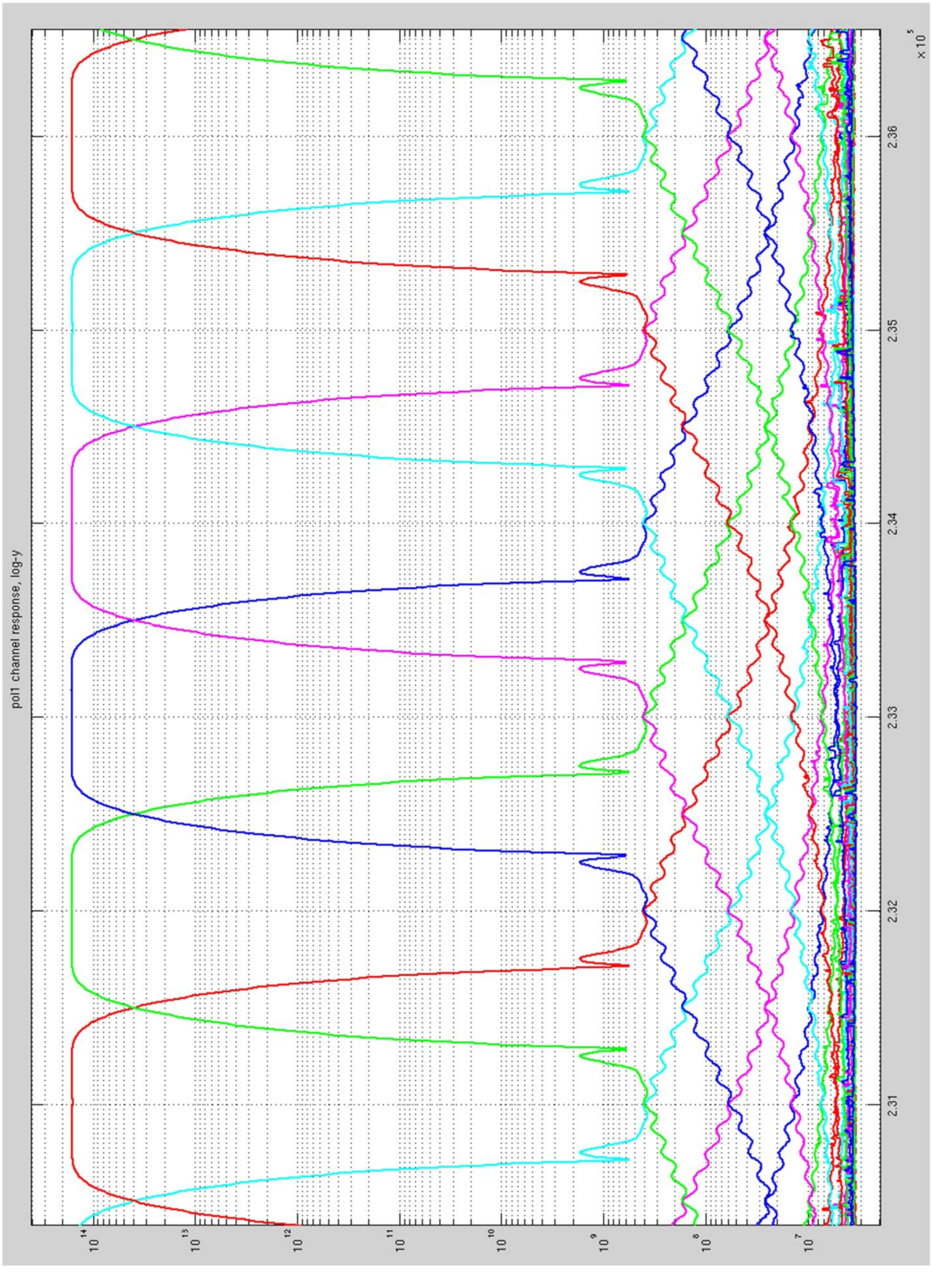
Output Format:

At the end of the design, the 4096 channels of each spectrometer are broken up into four blocks of 1024 channels each. (thus, eight blocks total for both polarizations).
The accumulated values are stored as 48-bit, unsigned integers. The upper 32 bits are stored in a separate BRAM for each block, while the lower 16 bits of a given block are shared with another block for efficiency purposes. The sample python script collects all 48 bits, but if acquisition time is of the essence, the lower bits could be ignored for a small increase in noise.

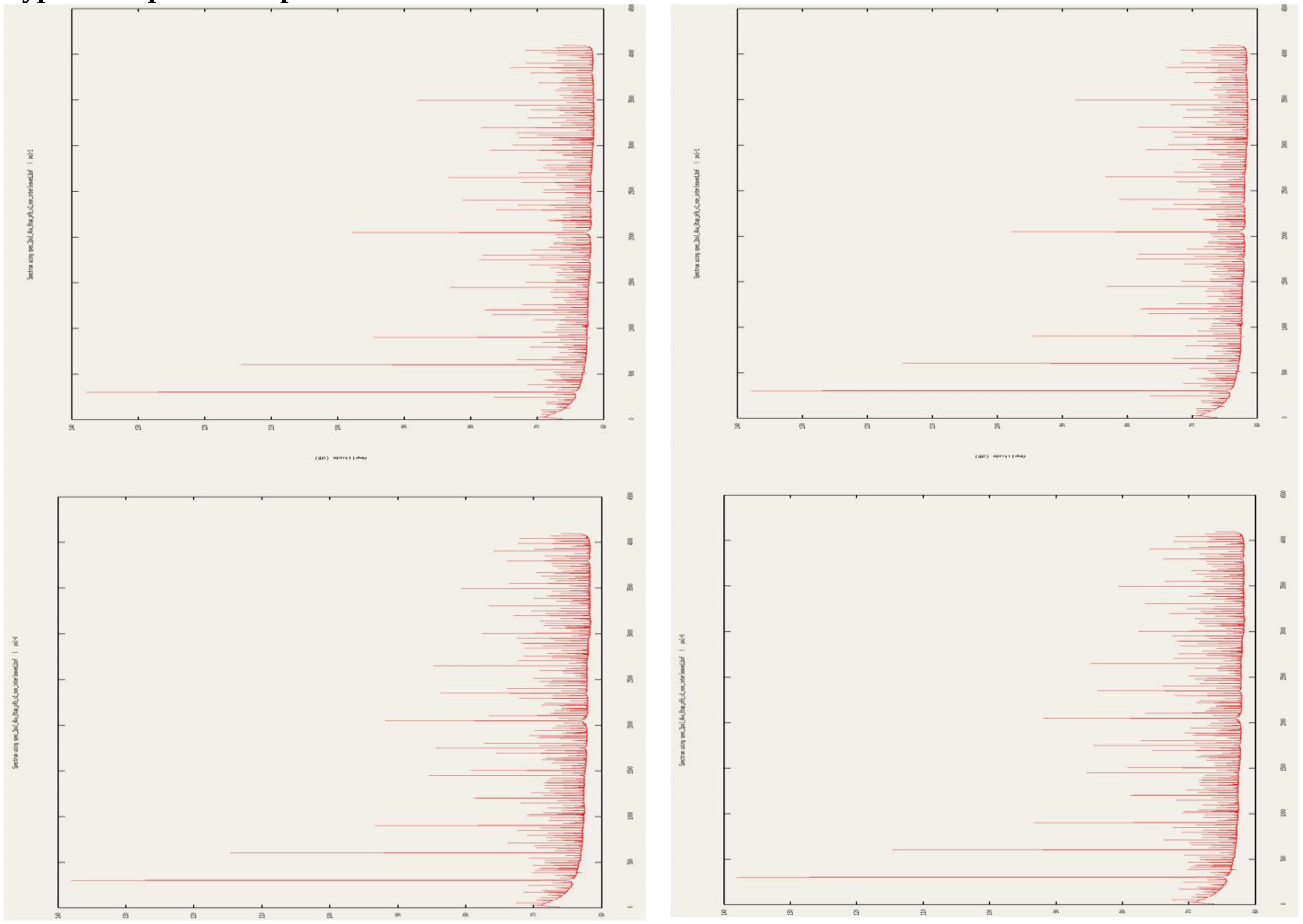
GPIO Pin	Purpose
A-0	to_gpio_a[0]
A-1	to_gpio_a[1]
A-2	to_gpio_a[2]
A-3	to_gpio_a[3]
A-4	to_gpio_a[4]
A-5	to_gpio_a[5]
B-0	start_acc
B-1	cont_acc
B-2	from_gpio_b[0]
B-3	from_gpio_b[1]
B-7	from_gpio_b[2]

Channel shapes, spectra, etc:

Below is a plot of several channels frequency responses plotted together. This was generated by sweeping a signal generator across the channel's frequencies in fine increments. The first sidelobe is very small, and has a frequency response of -50dB, relative to the passband. The second sidelobe is located in the middle of the adjacent channel, and has a frequency response of -55dB, relative to the passband.


pol1 channel response, log-y
10^{14}
10^{13}
10^{12}
10^{11}
10^{10}
10^{9}
10^{8}
10^{7}
2.31
2.32
2.33
2.34
2.35
2.36
$\times 10^{5}$

Typical output from spectrometer:



Known issues:

Reducing the accumulation length has a substantial chance to cause the accumulator to lock up for the next 3hr, 15min. Doing so is not recommended. Workaround: re-program the fpga when starting a shorter accumulation. This issue can be resolved in a future release, if there is sufficient demand.

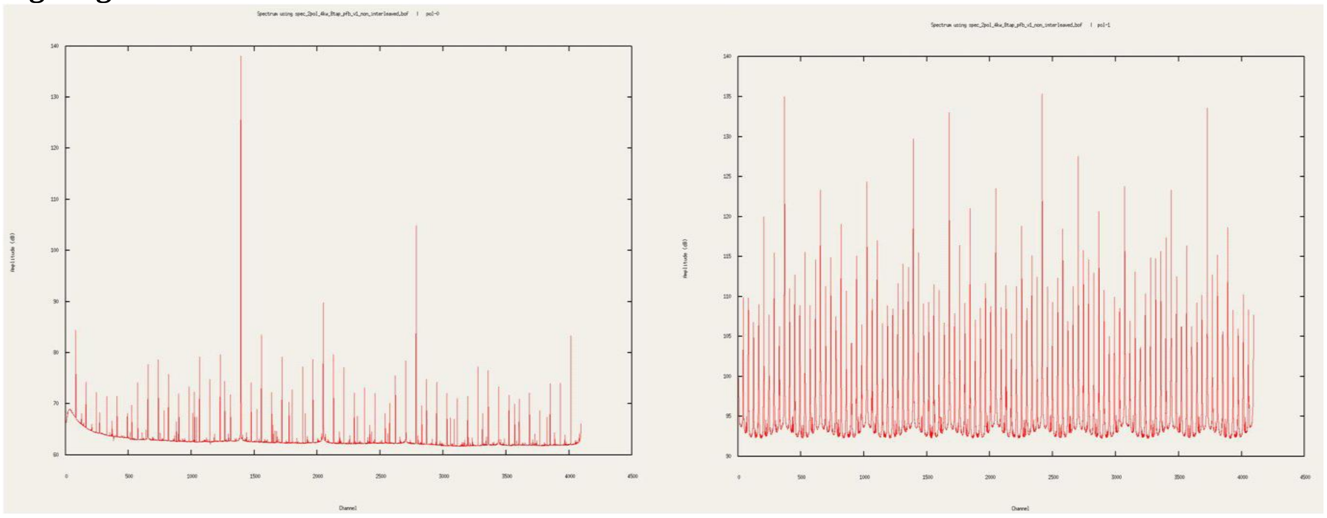
It is worth noting the presence of harmonics in the spectrum of our output: this appears to be caused by the ADCs used in our setup (National Semiconductor 083000s). The first harmonic is generally 25dB to
35dB below the level of the desired output.

In addition, channel 2048 on both pols is corrupted due to (in my opinion) the gain/offset mismatch between the two internal ADCs on each 083000. Those channels must be ignored.

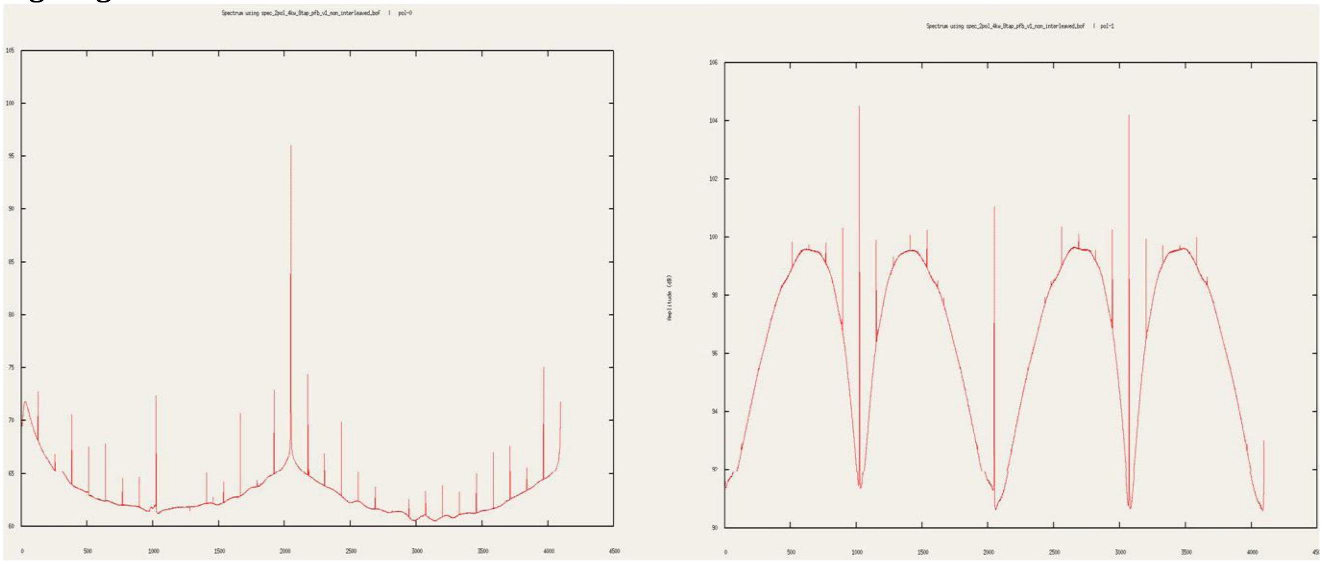
Approximately 1/5th of the time upon programming the FPGA, polarization 1 is corrupted beyond use. The reasons for this are unclear, but reprogramming the FPGA usually remedies the problem.
Polarization 0 always functions normally. Below are plots showing typical spectra with a corrupted
polarization 1.

Pol0 is on the left; Pol1 on the right.

Signal generator on:



Signal generator off:


Spectrum using spec_2pol_4kw_8tap_pfb_v1_non_interleaved.bof | pol-0
Spectrum using spec_2pol_4kw_8tap_pfb_v1_non_interleaved.bof | pol-1
Amplitude (dB)