

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

Ryan M. Monroe

Georgia Institute of Technology, Atlanta, GA 30332

Mentor: Robert Jarnot

Jet Propulsion Laboratory, Pasadena, CA 91109

A state-of-the-art digital Fourier Transform spectrometer 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 model the composition of Earth's upper atmosphere with regards to certain helpful and harmful chemicals, such as greenhouse gases and Ozone. The Microwave Limb Sounder (MLS) is an instrument designed to map the global day-to-day concentrations of key atmospheric constituents continuously.

One 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 more efficiently but also be processed directly once received. The present generation 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 massively superior bandwidth and 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 be converted into a digital format, processed through a Fourier Transform and finally accumulated to reduce 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.

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 usually poor.

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.

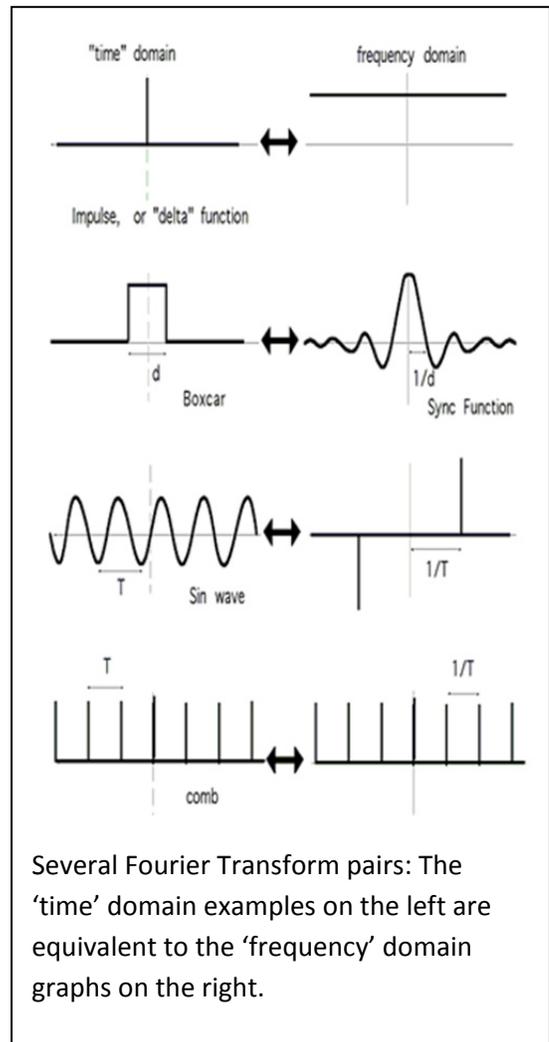
In the final stage, Bitstream Generation, the completed design is compiled into a bitstream which configures the FPGA directly. This stage is deterministic, and if the preceding stages completed successfully, this one typically will as well.

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.

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 $N \geq 128$. 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 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, performed similar work, creating a 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 FPGA board, produced by Nallatech.

IV. Approach

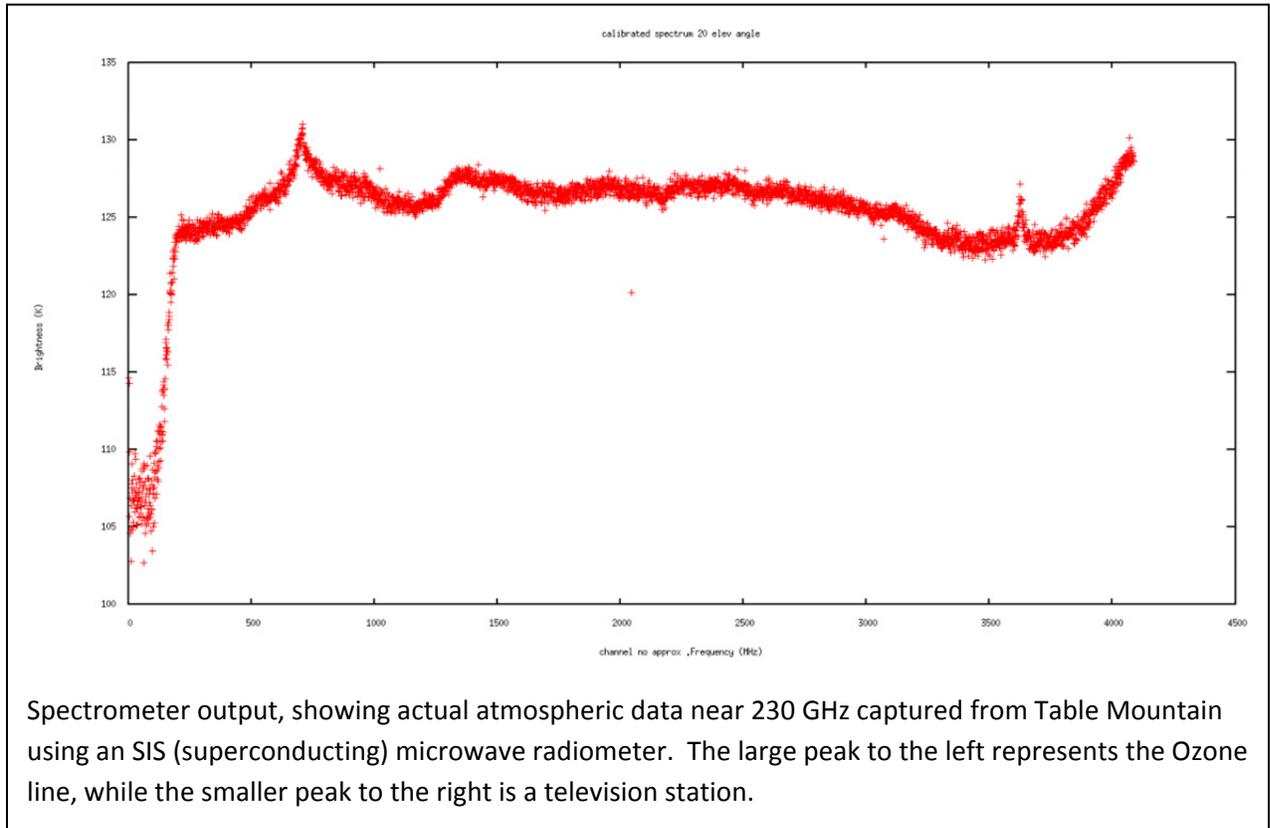
A. Enhance the present spectrometer

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 causes the hardware contained within those blocks to be corrupted in an unpredictable fashion. In addition, simulation often takes 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.

Efforts were also made to make the same hardware function on the Nallatech board, which is more compact, requires less power and has a more streamlined interface. Unfortunately, despite several simulations and attempted solutions, the board remains nearly as inscrutable as when our efforts began.



IV. Results and Discussion

The final spectrometer designed (as of the writing of this paper) is a 4096-channel implementation. Designed with additional output capacity, the spectrometer has superior frequency resolution, dynamic range and accumulation length when compared to previous versions. A higher order, 8192-channel spectrometer is nearing completion as well. Both designs are capable of accumulating for hours, several orders of magnitude over 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 above represents actual output recorded from table mountain, using a previous generation (lower resolution/higher noise) device.

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 in future, even more ambitious projects. Plans to make an 8 GHz 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 balloon flight planned for September.

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 is of a limited size, 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-Berkeley 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 even 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 $\langle 2N-1 \rangle$ DSP48E blocks and $\langle 2N-1 \rangle$ 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 $\langle N + \text{ceil}((N-1)/2) \rangle$ with no further hardware cost.

Removing adders completely

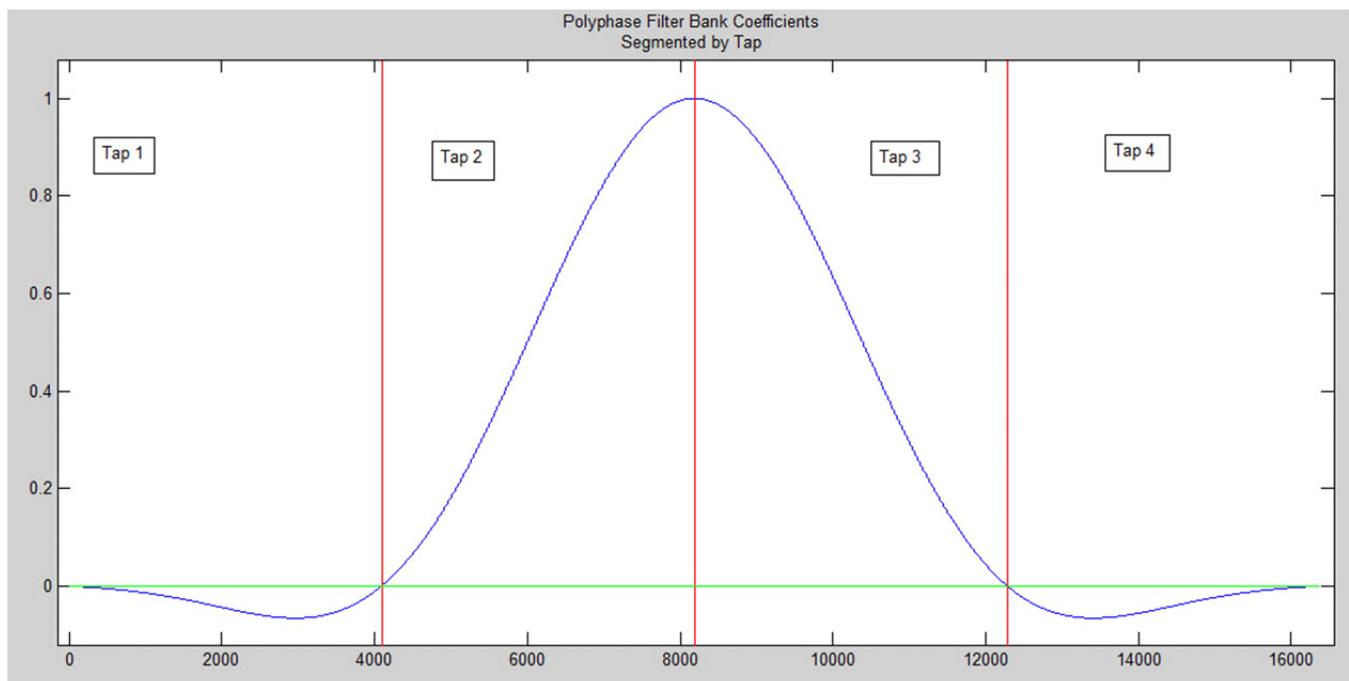
For a small additional price, however, the adders can be completely removed. Because each DSP48E block supports a multiply followed by an add, the PCOUT ports of the first multipliers can be attached to the input ports of subsequent taps. If the input data and coefficient blocks are each delayed by an 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 $\langle N \rangle$

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 $\langle \text{ceil}((N-1)/2) \rangle$ (designs with vectors of 512 elements or fewer) and $\langle N-1 \rangle$ (designs with vectors of 1024 or more).

Halving coefficient Block Ram consumption

Consider the impulse response of the PFB filter: It is a windowed $\langle \sin(x)/x \rangle$ 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

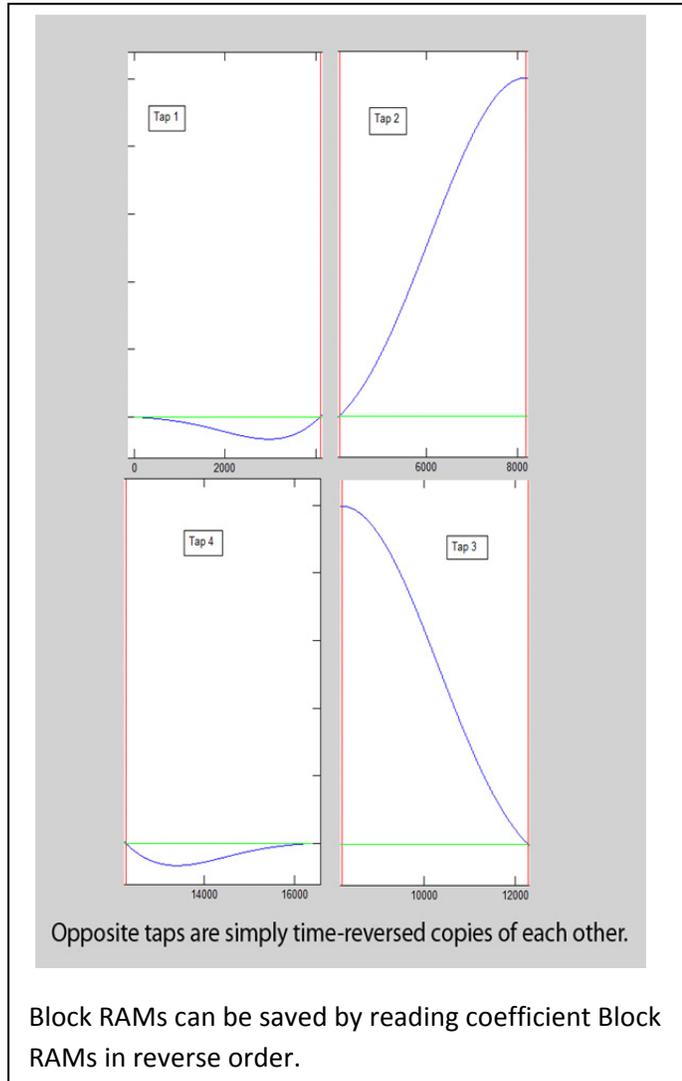


Four-tap Polyphase Low-Pass Filter, split by taps

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.

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 $\lceil N/2 \rceil$, 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.



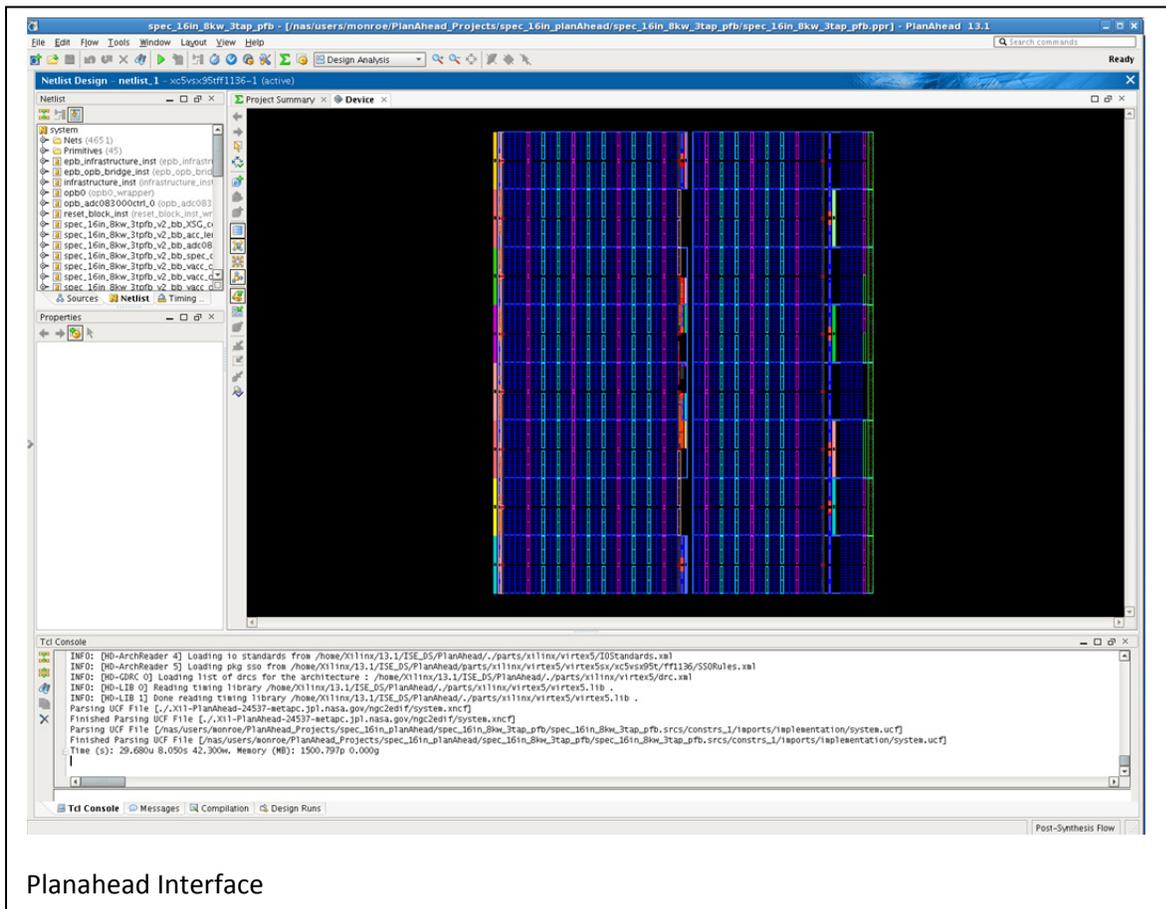
Summary (PFB optimizations):

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 $\langle 2N-1 \rangle$ DSP48Es and $\langle 2N-1 \rangle$ Block RAMs down to $\langle N \rangle$ DSP48Es and (512 length vector or smaller) $\langle N+1 \rangle$ or (1024 length vector or greater) $\langle 3N/2 - 1 \rangle$ 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.

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.



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.

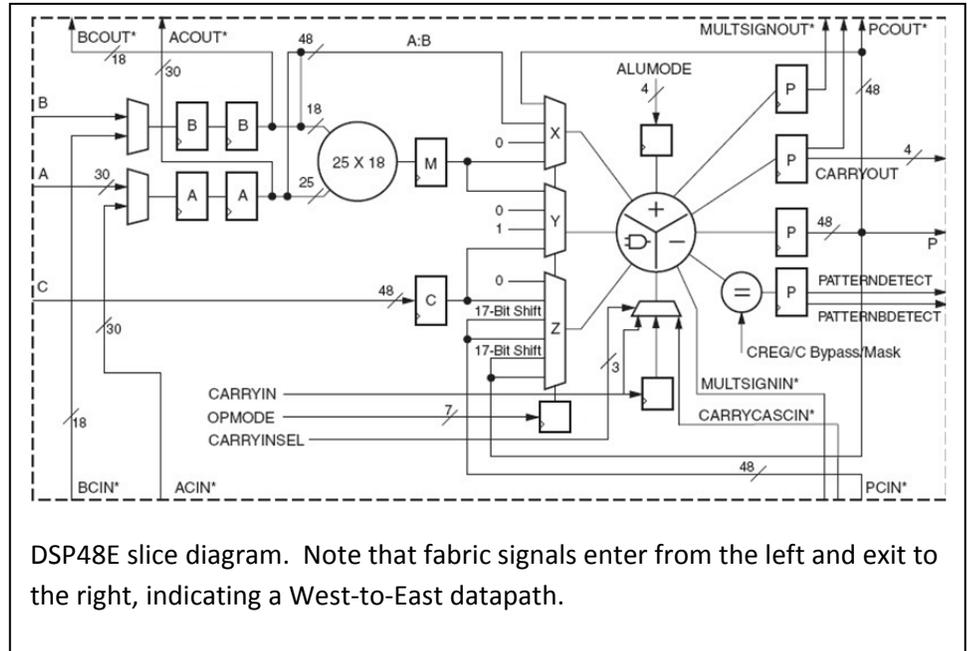
Be sure to consider the locations of the input and output ports of the chip when



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

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.



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.