ACCELERATION AND IMPLEMENTION OF A DSP PHASE-BASED FREQUENCY ESTIMATION ALGORITHM: MATLAB/SIMULINK TO FPGA VIA XILINX SYSTEM GENERATOR.

BY

KURT D. ROGERS

B.S., Binghamton University, 2001

THESIS

Submitted in partial fulfillment of the requirements for
the degree of Master of Science in Electrical Engineering
in the Graduate School of
Binghamton University
State University of New York
2004

Accepted in partial fulfillment of the requirements for
the degree of Master of Science in Electrical Engineering
in the Graduate School of
Binghamton University
State University of New York
2004

Mark Fowler_____May 19, 2004
Department of Electrical Engineering

Douglas Summerville_____May 19, 2004
Department of Electrical Engineering

Edward Mohring_____May 19, 2004
Department of Electrical Engineering

ABSTRACT

This paper utilizes a phase-based frequency estimation algorithm as a research vehicle to explore a new and improved software tool and design flow for the implementation of Matlab DSP algorithms on Xilinx FPGA's. The software tool, called System Generator for DSP, is essentially a hardware add on toolbox for Simulink, which is a software subset of Matlab. It offers a unique design solution whereby DSP algorithm design in Matlab, and hardware implementation on an FPGA, can essentially by executed in a single flow software environment. A thorough description of the Xilinx Virtex-II Pro FPGA architecture and associated software tools, with a special focus on the architectural information needed to understand multipliers and the implementation of DSP algorithms, is provided as background. The motivation for studying the System Generator design flow and the DSP theory of the phase-based frequency estimation algorithm is provided. The algorithm is implemented in Matlab code as a test comparison model for the Simulink design. A subsystem of the phase-based frequency estimation algorithm, the FFT front-end, is utilized as an example to provide a detailed explanation of Simulink and the two modes of the System Generator, software simulation and hardware co-simulation. Finally, an analysis of the implementation of the phase-based frequency estimation on the Xilinx FPGA via System Generator for DSP is provided, with special emphasis on the FFT front-end subsystem design.

ACKNOWLEDGEMENTS

Special thanks needs to be given first to Michael Fallat, a Xilinx dedicated Field Applications Engineer for Avnet. Without his support, much of the work with the Xilinx System Generator for DSP could not have been accomplished. Also, Dr. Mark Fowler of the ECE Department at Binghamton University needs to be acknowledged for his assistance with the phase-based frequency estimation algorithm DSP theory. A special thank you is also extended to the Electrical and Computer Engineering Department of Binghamton University, as well as to the Xilinx University Program (XUP). Without their generous donation to purchase the necessary hardware and software, none of the work presented in this paper could have been accomplished. Finally, a thank you is extended to Edmond Mohring, an adjunct professor in the ECE Department at Binghamton University, for serving as the advisor to this thesis research.

Table of Contents

List of Figures

List of Equations

1. INTRODUCTION

1.1 Background : Why FPGA's for DSP ?

Despite the many advancements in dedicated digital signal processing (DSP) microprocessors, the efficiency with which any DSP hardware implementation is able to execute the long standing metric of multiply and accumulate still remains to be the limiting factor in overall performance. The specialized and constrained RISC-like instruction sets that give DSP's such as Texas Instruments' 8 instruction deep VLIW TMS320C62xx family their increased efficiency, oddly enough also lead to their performance ceilings for DSP applications. First, it is incredibly difficult to produce efficient assembly and machine code for these specialized instruction sets, and thus DSP's require a powerful, specially optimized compiler (typically C). Second, and most important, is their inability to do multiply and accumulate functions in parallel. Despite the 8 instruction deep pipeline, the fact still remains that the data flow must pass through a pipeline that is only capable of completing one multiply and accumulate function every few clock cycles at best. This translates to calculating one sample of a 512-point Fast Fourier Transform (FFT) every several thousands to 10's of thousands of clock cycles, depending on the architecture [1]. These two limitations led designers in the early 1990's to look for other alternatives that might alleviate the overhead of and dependency of the compiler, and offer a more parallel approach.

At first, this search led designers down the path of Systems on a Chip (SOC) design. SOC's are very dense and fast Application Specific Integrated Circuits (ASIC's) that contain a central processor hard core (ex. IBM 405 PPC), surrounded by soft-core

algorithm accelerators, memory, soft-core peripherals, and I/O interfaces all attached to a central bus. These chips are expensive, designed for large production quantities, and are one-time-programmable. An example of a current large SOC market is the second-generation cellular phone.

Around the same time as the move to ASIC's, Field Programmable Gate Arrays (FPGA's) emerged in the engineering prototyping and emulation world. FPGA's are a re-programmable, SRAM based, VLSI platform that allow a digital hardware designer to almost instantly (compared to ASIC's) upgrade a design. FPGA's loose their memory when power is removed from the circuit, and thus must be programmed upon each power up, either through JTAG or a PROM. FPGA's contain internal units such as built in memory, registers, and multiplier blocks. As the number logic resources in FPGA's grew, DSP designers began to take advantage of the vast amount of parallelism offered by such re-programmable chips, at the cost of reduced clock speed in comparison with current dedicated DSP microprocessors. By 2000, FPGA's began to approach the gate density of ASIC's, and make significant increases in maximum clocking frequencies. However, ASIC's are still superior for overall density and speed. Nevertheless, the vast parallelism allowing for the execution of hundreds of multiply and accumulate (MAC) operations in parallel, combined with nearly instant re-programmability, steered the implementation of computationally complex DSP algorithms down the path of FPGA's.

Many companies specializing in DSP algorithm design and implementation utilize FPGA's as their primary means of prototyping. Often times they are utilized as the final design implementation choice over an ASIC platform in situations where power consumption is not a major concern. This is most evident in companies such as Lockheed

Martin who develop military applications that require cost effective rapid prototyping platforms that can be highly integrated with DSP software development. One of the most world-renowned software tools for DSP algorithm design is Matlab, produced by Mathworks. Many companies currently realize all of their high level DSP modeling in Matlab, including all of the necessary test harnesses. They then convert these algorithm models to a hardware description language, such as Verilog or VHDL, either manually or by using some algorithm specific conversion software. From here, the normal design flow is followed to synthesize and place and route the algorithm into a target FPGA of choice. The original test vectors used to test the algorithm in Matlab are generally applied in some fashion to test the algorithm in the actual hardware. Therefore, printed circuit boards (PCB's) housing the FPGA's must be designed with interfaces that will allow data to be transmitted to and from the PCB via a PC running Matlab. This interface requires additional hardware and software overhead for testing. The bottom line is that today's DSP algorithm and hardware designers have a desire and need to begin designing with Matlab, target FPGA hardware, and then complete their verification by reading test vectors back into Matlab from the real hardware to compare against their original software algorithm results.

In 2001, one of the leading FPGA companies, Xilinx, teamed up with IBM PowerPC (PPC) ASIC designers to develop a new VLSI design platform. The result was a new family of Xilinx FPGA's, called the Virtex II Pro (V-II Pro), built on Xilinx's Virtex family. Not only are these FPGA's faster and denser than previous programmable logic families from Xilinx or their competition, but they also integrate an IBM 405 PPC ASIC hardcore into the FPGA fabric. Along with the built in processor, Xilinx and IBM

offered a set of soft cores to accompany the processor, which were optimized for FPGA implementation, as well as a new set of software tools to easily integrate them into FPGA designs.

The integration of an ASIC hardcore into the programmable fabric of an FPGA is not only a great milestone from a VLSI technology standpoint, but it also provides an essential bridge between the SOC market, traditionally only capable of being implemented in an ASIC, and re-programmable logic chips. With the availability of a central processor, FPGA optimized I/O cores (ex. Ethernet), new software, and millions of FPGA gates allowing for user defined soft cores, Xilinx and IBM gave birth to a whole new market; the *re-programmable SOC*. Despite the advantages this offers to the general SOC market, there is one major draw back for DSP implementations; the 405 PPC does not contain a floating-point unit (FPU). Floating-point operations are essential to DSP algorithms. However, not all is lost. Since the user-defined SOC cores are programmed into the FPGA fabric, portions of a given DSP algorithm can take advantage of vast hardware parallelism to implement floating-point MAC operations, while the rest of the algorithm is executed on the CPU. Future generations of the Xilinx re-programmable SOC will no doubt contain a FPU CPU, presenting the first fully integrated system to physically emulate (DSP) algorithm trade offs and optimization. Therefore, the new re-programmable SOC (i.e. the Xilinx Virtex-II Pro) offers a unique and high tech platform to develop the ultimate DSP engine.

1.2 Motivation for high level, single flow, implementation of DSP algorithms

The current DSP algorithm to FPGA implementation flow must cross many boundaries, such at Matlab to HDL, HDL to the FPGA, and FPGA back to Matlab, while

attempting to maintain the consistency of the original algorithm. What if design-flow, design time, and design resources for implementing DSP algorithms in hardware could be seriously improved? Matlab is an interpretive language; it does not compile its code. Large DSP algorithms can therefore take many minutes, even hours to execute. Those engineers who are strictly research oriented would most certainly be interested in a new Matlab design flow that could speed up Matlab and Simulink algorithms via an FPGA hardware accelerator. Those engineers who are interested in optimizing the entire DSP algorithm to FPGA implementation flow would most certainly be interested in a new solution that packages the entire flow into one software tool, such as Simulink, which is part of Matlab. Finally, for engineers who wish to explore re-programmable SOC's to develop more efficient DSP engines, this design work would rely heavily on researching hardware and software tradeoffs between a PPC compiled algorithm and a soft-core algorithm accelerator. What if large DSP cores eventually targeted for an SOC embedded system could be tested at very high levels with assurance of timing and functional equivalency once they were implemented in the SOC as logic?

Xilinx and Mathworks have the solution to these two questions. In addition to creating the concept of re-programmable SOC's, Xilinx has developed a new tool capable of a high level, single flow, implementation of DSP algorithms on FPGA's. The tool, called System Generator for DSP, is integrated into Simulink, a tool within Matlab used for modeling and simulating dynamic systems. Xilinx worked closely with Mathworks to create an additional set of toolboxes for Simulink. The Xilinx FPGA tools include a bank of DSP and related logic cores, optimized for implementation on FPGA's, which can be programmed to the users specifications into any HDL design. These cores are created

with the Xilinx software tool called Core Generator. The new System Generator toolboxes extend the original capabilities of Simulink by integrating the cores from the Core Generator, allowing Simulink users to essentially implement logic designs at a high level. Interfacing these new hardware blocks with traditional Simulink blocks is possible, as well as interfacing to traditional Matlab files. Once the design is complete and simulated from within Matlab, the Xilinx specific blocks within the Simulink design can then be synthesized and placed and routed through the normal Xilinx FPGA flow. When this is complete, the Simulink algorithm can be simulated, with the Xilinx core functions literally being executed on an FPGA. This is accomplished through a JTAG cable connected to the JTAG interface on an FPGA development card and the parallel port on a PC. When the algorithm is executed, the tools automatically load the FPGA and run the simulation real-time on the hardware. The boundaries from Matlab to HDL to FPGA and back to Matlab are thus removed, thereby creating a high level, single flow methodology to implement a Matlab DSP algorithm on an FPGA.

1.3 **Thesis Objective and Paper Organization**

The main objective of this thesis is to implement a Matlab DSP algorithm on a Xilinx Virtex II Pro FPGA (XC2VP7) utilizing their new System Generator for DSP software. Therefore, this thesis is broken down into four smaller objectives: research, design, implementation, and analysis. The motivation for choosing to focus this thesis around System Generator for DSP is presented in chapter 4.2. As a DSP algorithm *research* vehicle, a phase-based frequency estimation algorithm was chosen. The motivation for this choice is presented in chapter 4.3. In order to aid in the discussion of the implementation and analysis of the algorithm, a lengthy analysis of DSP MAC

functions in hardware is presented in chapter 4.1, using the FFT as an example. Some knowledge of the Xilinx Virtex II Pro FPGA architecture is necessary to understand chapter 4.1, and this material is covered in chapter 2, focusing specifically on FPGA resources for MAC implementations. This chapter also explains the 405 PPC IBM Core Connect Architecture (re-programmable SOC). Chapter 4.4 discusses the theoretical *design* of the phase-based frequency estimation algorithm. The initial Matlab implementation of the algorithm is discussed in chapter 4.5, with a lead into Simulink in chapter 4.6.

The System Generator *implementation* portion of the thesis was by far the longest and most difficult portion, and consequently only the FFT front-end subsystem of the System Generator design was actually implemented in hardware. First, the Simulink interface & the Xilinx toolboxes that make up System Generator are presented in chapter 5.1. Some knowledge of the Xilinx FPGA software is necessary to understand chapter 5.1, and thus it is covered in chapter 3. System Generator is broken down into two modes: software simulation mode, and hardware co-simulation mode. The FFT front-end subsystem of the frequency estimation algorithm is used as an example, and the two modes are discussed in chapters 5.2 and 5.3, respectively.

The *analysis* of the implementation of the frequency estimation algorithm is discussed in chapter 6. First a brief description of the Xilinx and Avnet hardware development kit used to implement the design is given in chapter 6.1. Second, some general notes and key concepts about hardware implementations in System Generator are given in chapter 6.2, the most notable being the override with doubles feature explained in chapter 6.2.2, a key concept and advantage of System Generator. Third, the hardware

implementation of the phase-based frequency estimation (PBFE) algorithm is discussed in chapter 6.3, making note of any special difference between the Matlab software approach and the Simulink hardware approach. Since the PBFE algorithm was not fully implemented at the time this paper was written, analysis of the algorithm is broken down into 2 parts. Chapter 6.3.1 covers the FFT front-end subsystem, which was completely verified and implemented in hardware. The total logic resources utilization of the two FFT front-end approaches (64 and 512) is discussed, making references back to the FPGA architecture covered in chapter 2. Chapter 6.3.2 discusses the implementation and resource utilization of the portion of the PBFE algorithm that was completed.

To tie in the re-programmable SOC concept of the Xilinx Virtex II Pro with DSP algorithm optimization, some speculative thoughts are offered on how to proceed down this path are offered in chapter 7. Finally, the conclusion, chapter 8, offers a summary of why FPGA's are used for DSP implementation, why a high level, single flow implementation from Matlab to FPGA's is needed, final thoughts on the frequency estimation algorithm, advantages and disadvantages of the System Generator design flow, and a future direction for SOC and DSP algorithm development.

1.3.1 Joint Research, Design, and Implementation Effort

The research, design, and implementation of the information presented in this thesis are the result of a collaboration between a professional colleague [4] and myself. My colleague's professional experience includes a detailed knowledge of test structures and interfaces, an industry level understanding of hardware logic design utilizing Application Specific Integrated Circuits (ASIC's) and Field Programmable Gate Arrays (FPGA's), and a basic knowledge in digital signal processing (DSP). My professional

experience includes detailed knowledge DSP, FPGA & ASIC logic design, System on a Chip (SOC) design, including the IBM 405 PPC core connect architecture, and DSP algorithm implementation.

While our individual, focused research efforts for this work differ, the overall material covered is a result of our joint efforts. My colleague's individual *research* focuses on FPGA test philosophy, while my individual *research* focuses on phase-based frequency estimation DSP algorithm. These topics form the core chapters of each of our individual theses papers. Collectively, we also *researched* the Xilinx Virtex-II-Pro FPGA architecture, Xilinx FPGA software tools, Simulink and the Xilinx System Generator for DSP, and SOC algorithm implementation. While the *design* of the DSP phase-based frequency estimation algorithm is my work, the *implementation* of the phase-based frequency estimation algorithm using the System Generator, and some of the *analysis* of this implementation, were a joint effort. My colleague's thesis paper discusses in detail the research of Xilinx Virtex-II-Pro FPGA architecture, Xilinx FPGA software tools, and SOC algorithm implementation. My thesis paper discusses in detail the research of Simulink and the Xilinx System Generator for DSP, as well as the design and implementation of the phase-based frequency estimation algorithm using the System Generator, and the analysis of this implementation. The topic of FPGA test researched by my colleague is beyond the scope this thesis, and will not be included here. The reader is encouraged at this time to reference my colleague's [4] work for more information. Of particular interest is chapter 3.4 & chapter 7, which directly tie together the System Generator Hardware Co-simulation process with FPGA Test. Other topics covered in my colleague's thesis, such as the FPGA architecture and Xilinx software, are more relevant,

so a summary is provided with more of a DSP slant to aid in background discussion for this paper.

To keep the overall flow of each individual thesis consistent with the motivation for our joint research, the introduction and general outline of both my colleague's paper and mine are very similar. Therefore credit is given at this time to my colleague [4], acknowledging his 50% effort in writing and editing the introduction and general paper outline, as well as his 100% effort in researching and writing any information that is referenced from his thesis. At this time, the reader is once again encouraged to read and/or reference, *Analysis of Xilinx FPGA Architecture and FPGA Test: A Basis for FPGA Enhanced DSP Algorithm Acceleration and Development in Matlab/Simulink Via Xilinx System Generator* [4].

2. Xilinx Virtex-II Pro Architecture

In order to provide sufficient background information for the discussion of implementing DSP algorithms on Xilinx FPGA's, this chapter presents a thorough summary of the Virtex II Pro architecture. It focuses primarily on the architectural

information needed to understand multipliers and the implementation of DSP algorithms. The reader may reference [4], chapter 2, as an additional source.

2.1 CLB's & Slices

The Xilinx Virtex II Pro (V2P) FPGA architecture is basically the same as the Virtex II platform, with the exception that the V2P is shipped on 300 nm wafers with dies that are fabricated on 0.13um technology. These chips are SRAM based devices: that is, they do not retain their logical configuration once power is removed. Instead they contain an internal SRAM based configuration memory. Upon power up, application specific configuration data is loaded into the configuration memory, which is typically stored in an EEPROM (It can also be loaded via a PC through a JTAG boundary scan interface. See [4], chapter 3.4). The EEPROM communicates with the FPGA to facilitate a three phase loading sequence. First, the configuration memory is cleared, then the configuration data is loaded, finally followed by a start-up sequence that activates the logic (sequential release of clocks and control lines) [8].

Although FPGA stands for Field Programmable Gate Array, the Xilinx FPGA's at the highest level of abstraction are really more like a very dense array consisting of the 6 major building blocks shown in Figure 2-1 [6]. Configurable Logic Blocks (CLB's),

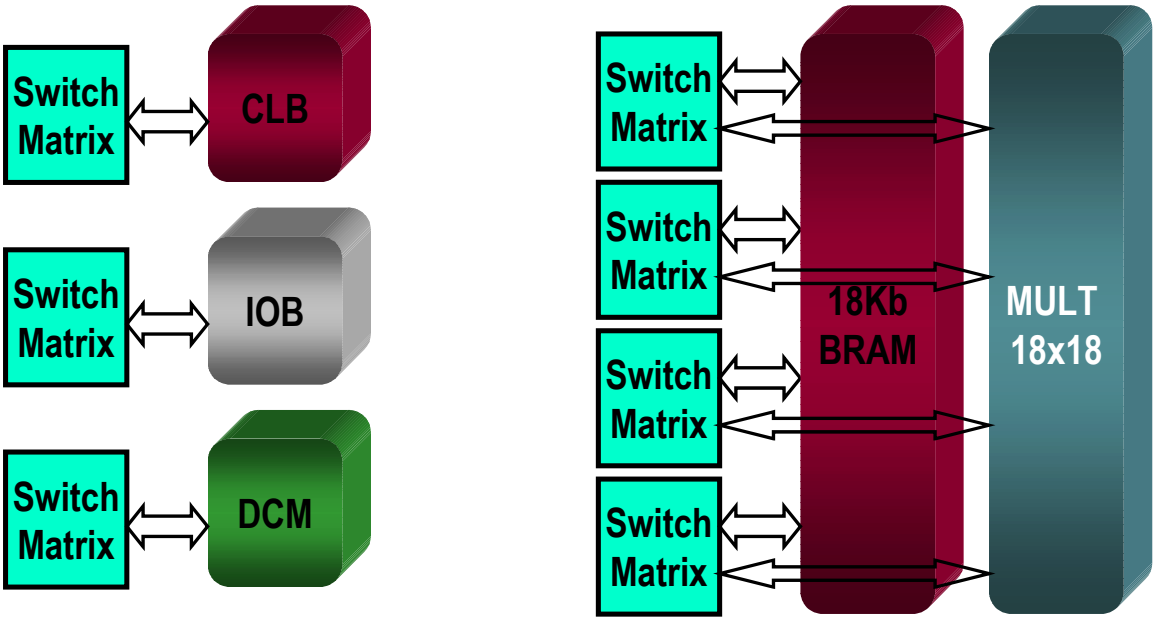


Figure 2-1 : Building Blocks of Virtex II Pro Array Architecture

Block RAM's (BRAM), Multipliers, Digital Clock Managers (DCM's), and standard and high speed I/O (IOB's), are all connected to each other through a fully buffered (SRAM controlled pass transistor) programmable Switching Matrix [11]. This switching matrix is programmed and controlled via the configuration data, known as a bit file, loaded into the configuration SRAM at power up. The CLB building blocks take up more than 75% of the area resources, and therefore each device within a the V2P family can be characterized by its CLB array size, with all of the other building blocks in relation to the CLB's, as depicted in Figure 2-2 [8]. The FPGA device that was supplied with the Xilinx development card that was utilized in the implementation portion of this research was the XC2VP7. This device has an array of 40x34, for a total of 1360 CLB's. As can be

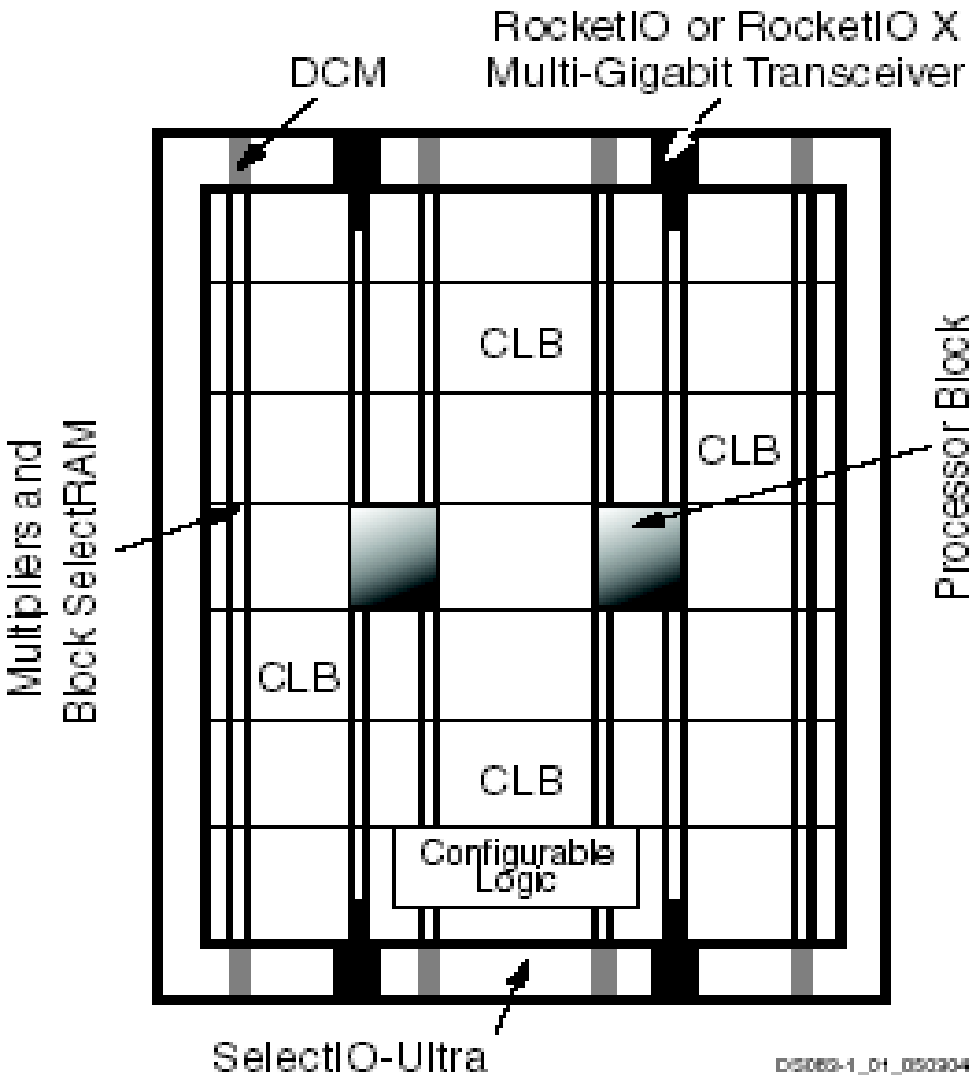


Figure 2-2 : Array Architecture of Virtex II Pro FPGA

seen from Figure 2-2, the Multipliers and Block RAM's are sandwiched in narrow columns between the CLB's. The XC2VP7 device has six such columns, containing a total of 44 multipliers and 44 Block RAM's. The maximum size V2P device goes up to 120x94 (11280) CLB's with 16 columns of 444 Block RAM's and 444 Multipliers. Notice in Figure 2-2 that the Digital Clock Manager blocks are placed at the top and bottom of each Block RAM/ Multiplier column; thus there are a total of 12 DCM's in the

XC2VP7 chip, with a maximum of 32 DCM's for the V2P family. The Rocket I/O Multi-Gigabit Transceivers are parallel to serial (and vice versa) embedded transceiver cores used for high-speed interfaces between multiple FPGA's, over a bus or back plane for example. Although these are very helpful to DSP designers who have need to parse a DSP algorithm across 2 chips and communicate quickly to keep processing real time data, neither the Rocket I/O nor the DCM's are the direct focus of discussion here, and thus the reader is referenced to [5] for further information on these topics. The last item in Figure 2-1 is the IBM 405 PCC, a hardware ASIC RISC processor. Although this is certainly a building block for the architecture, it really is an item that deserves separate attention, as it brings into the design a whole architecture of it own, with its own set of building blocks. Therefore, there will be a bit more discussion on the PPC later on in this chapter.

Clearly, the central building block of the V2P architecture is the CLB. Figure 2-3 [6] illustrates the construction of a single CLB. It consists of 4 slices, or sub-blocks, staggered into two columns, each with its own independent logic carry chain, as well a common shift chain connecting the staggered sets of slices. Each slice is connected to the programmable switch matrix such that each block may gain access to the IOB's, DCM's, BRAM's, Multipliers, and to other CLB's as Figure 2-1 illustrates. The fast connects

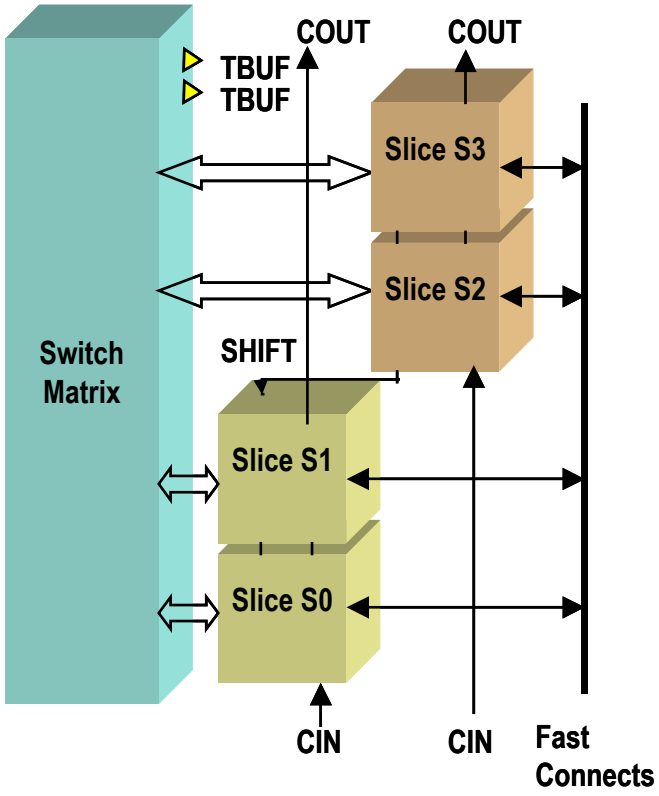


Figure 2-3 : Virtex II Pro Configurable Logic Block (CLB)

allow for quick local feedback within the CLB. Later on, the routing lines that connect CLB's will be explained.

It is at the slice level that the more common and smallest building blocks of the FPGA architecture are found. As Figure 2-4 [4] illustrates, each slice contains two 4-input function generators, two registers, and dedicated logic gates. The 4-input function generators can be configured as a 4-input look up table (LUT), 16 bits of distributed select RAM memory (RAM16), or as a 16-bit variable tap shift register (SLR16). In the distributed select RAM mode, the function generator can be configured as a 16, 32, 64, or 128-bit wide single port RAM address (uses 1, 2, 4, or 8 function

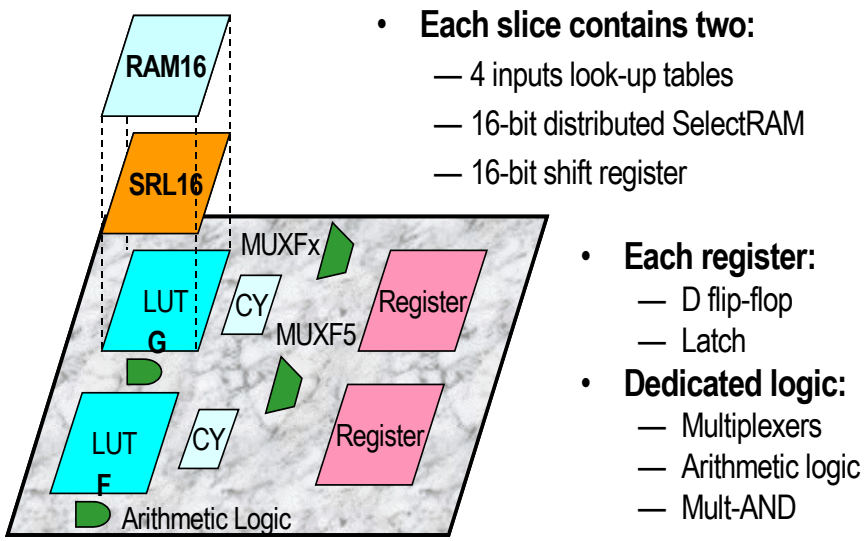


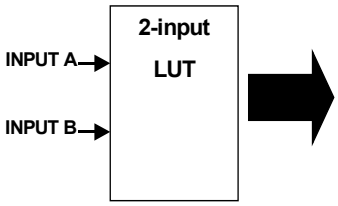
Figure 2-4 **:** Virtex-II Pro Slice

generators), or as a 16, 32, or 64-bit wide dual port RAM address (uses 1, 2, 4, or 8 function generators). Utilizing 4 or 8 function generators consumes either 2 or 4 slices within the CLB, respectively. The address lines to several distributed Select RAM's can be tied together and controlled by other combinational logic via the switch matrix, thereby creating small "distributed" chunks of memory, with each slice or CLB (depending on the width) being one address. Alternatively, the function generator can be configured to be a 16 to 256-bit wide ROM address. The 128 and 256-bit modes use 4 and 8 slices, or 1 and 2 CLB's, respectively. Therefore, the RAM's bit widths are configurable to within one CLB, however the ROM's are cascadable to implement wider memory.

In the 16-bit serial shift register mode, the function generators of up to 8 slices (2 CLB's) can be cascaded to create up to a 128-bit shift register with dynamic access to any bit in the chain. This makes the SLR16's a great resource for creating delay elements.

The dedicated shift chain illustrated in Figure 2-3 allows for fast connection between slices without going through the switch matrix.

The look up table (LUT) mode of the function generator is probably the most exhaustively used resource on an FPGA chip. Although there are some dedicated logic gates within a slice (described below), any combinational realization of a logic circuit in an FPGA inevitably becomes a massive array of LUT's. In general, whether it be a multiplexer, AND gate, OR gate, or some variation of these combined logic functions, LUT's are capable of implementing any arbitrarily defined Boolean function of 4 inputs. An example of how a 2-input combinational logic function would be coded into a LUT is given in Figure 2-5 [9]. Implementing Boolean functions in a look up table fashion is a



LOOK-UP TABLE						
INPUT ADDRESS	LOGIC FUNCTION IMPLEMENTED BY VALUE AT INPUT ADDRESS					
AB	AND	OR	NAND	NOR	XOR	XNOR
00	0	0	1	1	0	1
01	0	1	1	0	1	0
10	0	1	1	0	1	0
11	1	1	0	0	0	1

Figure 2-5 : Example of a 2-input Look Up Table (LUT) Realization

unique concept in that the combinational propagation delay is independent of the function implemented. Additionally, the MUXFx and MUXF5 shown in Figure 2-4 allow for the cascading of function generators in order to create combinational function of 5, 6, 7, or 8 inputs. Since all combinational logic functions are mapped to a table, it becomes difficult to use the old ASIC and CPLD metric of "gate count" to estimate the FPGA real estate necessary to implement any given design. Therefore, it is important to have a clear grasp on how LUT's are utilized in an FPGA, the total number in the device, and what sort of interconnections are available for routing between them.

The registers in Figure 2-4 can be either configured as a level sensitive latch, or a D-type flip-flop. These registers serve as local storage element (if needed) for all function generator configurations, as well as any directly implied registers. As opposed to ASIC designs where it is typically desirable to realize a logical solution with as few registers as possible, registering in FPGA design is an advantage, an actually necessary. For example, the direct routing between any two particular building blocks, say from a LUT in one CLB to a LUT in an adjacent CLB, could potentially cause a timing problem. Assuming a data path starts and ends with a register as depicted in the top portion of Figure 2-6 , the data would have to travel from register 1, through both LUT's (assuming in separate CLB's) and then finally to the 2^{nd} register. The propagation delay through these 2 LUT's may very easily exceed the system clock period for combinational trees surpassing 10 levels of logic at high frequencies. By adding a

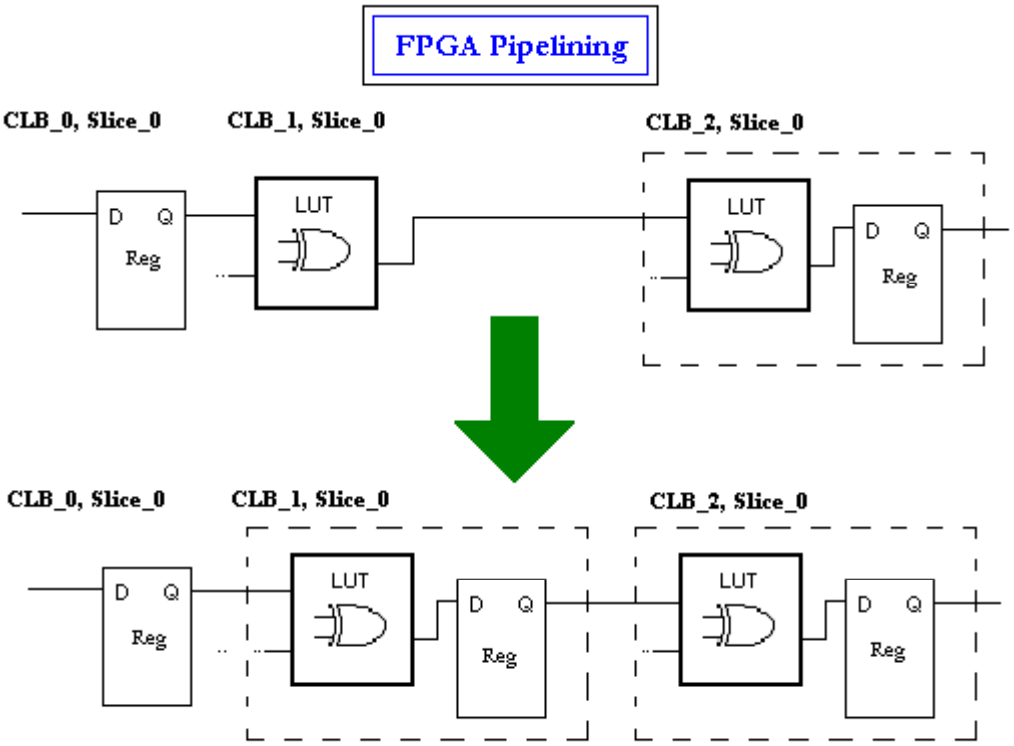


Figure 2-6 : FPGA Pipelining Example

pipelined register, that is placing a register in between two sets of combinational logic functions (the LUT's in this example) as in the bottom portion of Figure 2-6, the potential timing issue can be resolved. The original output result will be delayed by one clock period, however now the data only has to make it through one LUT within one clock period, whereas in the register-LUT-LUT-register data path, the data has to propagate through two LUT's within one clock period. This type of routing and propagation, known as pipelining, can usually be implemented quite easily in Xilinx FPGA's to allow

system clocks up to about 200 MHz. However, the speed gained by pipelining comes with an added cost; increased utilization of registers and routing resources. Thus, registers are the second most utilized resource in FPGA designs.

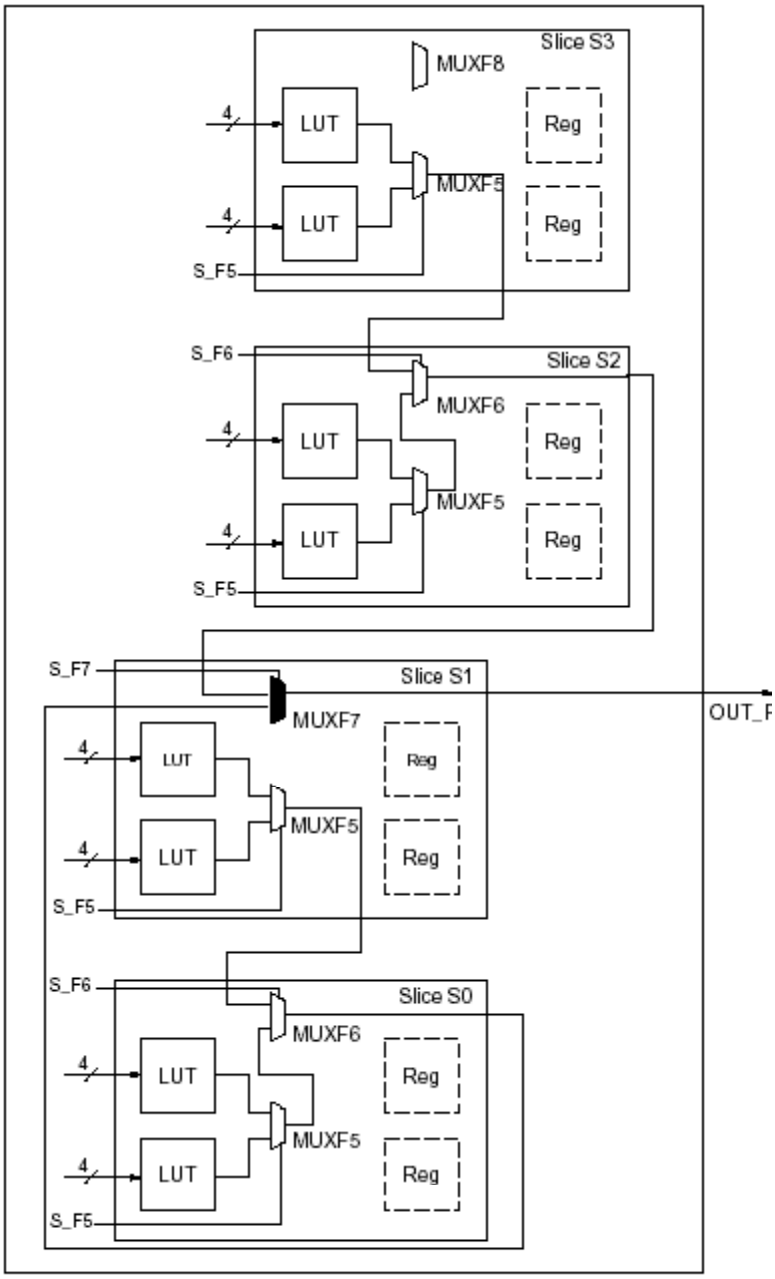


Figure 2-7 : 8:1 CLB Multiplexers Example

The last group of building blocks to discuss in Figure 2-4 are the dedicated logic gates, namely the multiplexers, arithmetic logic, and the Mult-AND. Any 2:1 mux can be implemented in the LUT's. Beyond that, the dedicated multiplexers in each slice can be used to create 4:1 (one slice), 8:1 (2 slices), 16:1 (1 CLB), or 32:1 (2 CLB's) multiplexers, broken down into groups of 2:1 muxes, with the first set of the tree being implemented in the LUT's. Figure 2-7 [8] gives an example of the full utilization of a CLB with the implementation of an 8:1 mux. As can be seen, the MUXF5 depicted in the slice of Figure 2-4 is used to create the first sets of 4:1 muxes, while the MUXFx (x = 6, 7, 8) allows for the expansion to 5, 6, 7, or 8 to one multiplexers. There is dedicating mux routing between adjacent CLB's, and thus to build a 16:1 mux, two 8:1 multiplexers are connected via MUXF7 of slice 1 in the first CLB, and MUXF8 of slice 3 in the second CLB, and so on for larger trees.

The dedicated addition arithmetic logic is very simple, while the dedicated Mult_AND gate is more complicated and will be covered in detail later on. For now, suffice to say that the Mult_AND gate improves the efficiency of small multipliers implemented in the slices. As for the addition arithmetic, there is one dedicated 2-bit XOR gate per slice, which combined with the fast look ahead carry chain (see Figure 2-3), provides one 2-bit full adder per slice. This configuration allows slice 0 and slice 1 to perform addition on a 4-bit number by sharing the carry chain. Note that slice 2 and 3 cannot operate on the same signal as slice 0 and 1, since there is no direct connection between the carry chains. In this manner, the maximum addition any one CLB can handle is a 4-bit addition on two *separate* signals. If 'a' and 'b' represent the 2-bit signals to be

summed, then the (a0 + b0) XOR function is implemented in the LUT, while the dedicated XOR gate in the slice adds the carry (Cin-1) from the previous addition stage. The output of the LUT also controls the carry mux (MUXCY) to either pass the carry bit on, set it to a 1, or a zero. Through a simple analysis, it can be seen that regardless of the carry value, if a0 & b0 are both zero, then the carry is always zero. Likewise, if a0 and

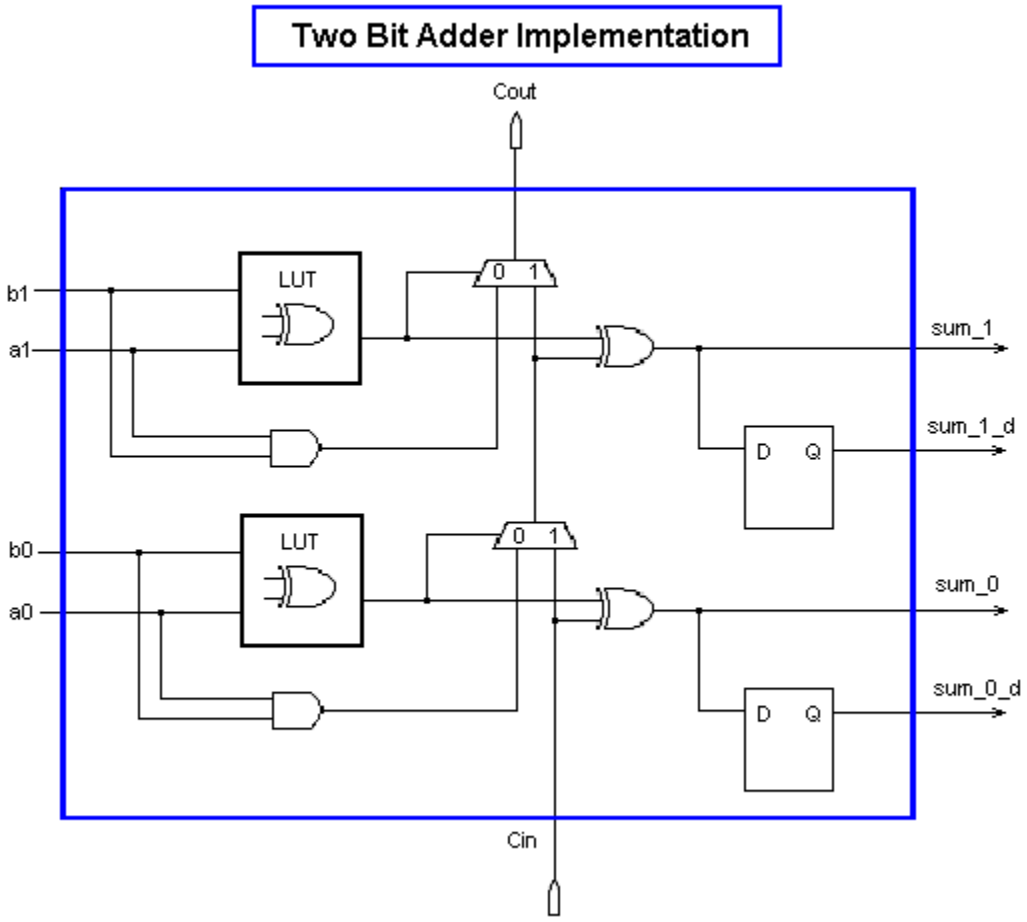


Figure 2-8 : V2P Slice implementation of a 2-bit adder

b0 are both one, then the carry is always one. If a0 and b0 are different however, then the new carry is equal to the old carry. This is the simple addition concept that allows for an extremely fast carry chain path. The beauty of this principle is that 2 birds can be killed with one stone; that is you need to add (a0 + b0) and you need to detect if a0= b0. Both of these are accomplished with an XOR function in the LUT. Additionally, in cases where a0 does equal b0, then it is necessary detect whether the carry should be set to zero or a one. This is accomplished by simply ANDing the 2 input signals together, as depicted in Figure 2-8. Although its not for multiplication in this example, later on it will be shown that this is the exact reason for the MULT_AND gate. Since the carry chains go up in the V2P FPGA, the (a1+b1+Cin-1) sum in this example must be executed in the slice above. In this manner, for a 2-bit addition, the LSB sum is executed in either slice 0 or 2, and the MSB sum is executed in slice 1 or 3 of a CLB.

Figure 2-9 [8] represents a logical schematic of the top half of a Virtex-II Pro slice depicted in Figure 2-4, which illustrates three of the five key building blocks of the V2P architecture; the function generators, the registers, and the dedicated XOR and carry chain for addition. The fourth key building block, the MUX5 and MUXFx, have already been depicted in Figure 2-7. Between this figure and the nearly complete schematic of the slice, just about all of the key architectural blocks have been illustrated in some fashion. The 5th key block is the MULT_AND, which will be explained in terms of how slices and CLB's implement multipliers near the end of this chapter. The only other key component of the slice to make mention of is the dedicated ORCY gate, which allows for quick sum of product (SOP) chains. Up to four products per LUT can be implemented,

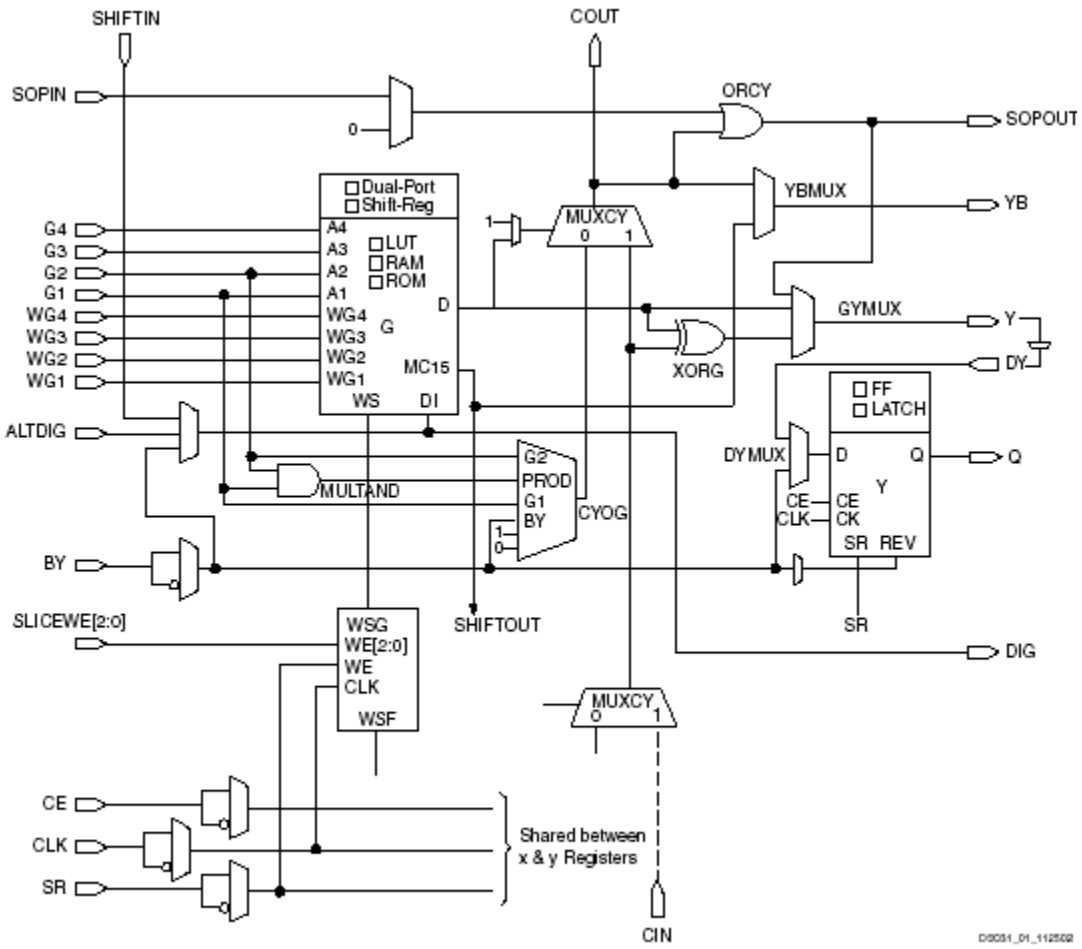


Figure 34: **Virtex-II Pro Slice (Top Half)**

Figure 2-9 : Virtex-II Pro Slice (Top Half)

allowing up to a 16-input AND gate per CLB. Only the top slices in a CLB have the SOP chain. In this manner products trickle up the carry chain from the lower slices to the upper slice of a CLB, while the sums propagate left to right across the chip.

The top portion Figure 2-10 [8] offers a summary of all the types of building blocks within a slice, including their maximum values. The bottom half of Figure 2-10

Slices	LUTs	Flip-Flops	MULT_ANDs	Arithmetic & Carry-Chains	SOP Chains	Distributed SelectRAM+	Shift Registers	TBUF
4	8	8	8	2	2	128 bits	128 bits	2

Device	CLB Array: Row x Column	Number of Slices	Number of LUTs	Max Distributed SelectRAM+ or Shift Register (bits)	Number of Flip-Flops	Number of Carry Chains(1)	Number of SOP Chains(1)
XC2VP2	16 x 22	1,408	2,816	45,056	2,816	44	32
XC2VP4	40 x 22	3,008	6,016	96,256	6,016	44	80
XC2VP7	40 x 34	4,928	9,856	157,696	9,856	68	80
XC2VP20	56 x 46	9,280	18,560	296,960	18,560	92	112
XC2VPX20	56 x 46	9,792	19,584	313,334	18,560	92	112
XC2VP30	80 x 46	13,696	27,392	438,272	27,392	92	160
XC2VP40	88 x 58	19,392	38,784	620,544	38,784	116	176
XC2VP50	88 x 70	23,616	47,232	755,712	47,232	140	176
XC2VP70	104 x 82	33,088	66,176	1,058,816	66,176	164	208
XC2VPX70	104 x 82	33,088	66,176	1,058,816	66,176	164	208
XC2VP100	120 x 94	44,096	88,192	1,411,072	88,192	188	240

Notes:
1. The carry-chains and SOP chains can be split or cascaded.

Figure 2-10 : V2P CLB Slice Summary & Device Configurations

moves up one level of hierarchy to the CLB architecture level. This table lists the CLB array configuration. Notice that if one uses the CLB array information to try and calculate out the total number of each type of building block with the FPGA, one does not arrive at the same values in the chart. This is because there are items such as the configuration SRAM that also take up room in the device, and therefore the resource values are slightly smaller than might be expected.

2.2 Routing Resources

As should be apparent from the above discussions on the building blocks of CLB's and slices, CLB's need to route, via the switch network, the output of all 4 slices to other CLB's and slices. Paths such as the carry and SOP chain are independent of this routing. The hierarchical routing is the mechanism that allows for the massive compilation of complex circuit designs (usually via a hardware description language (HDL)) into the fabric of the FPGA. For example, by giving FPGA software tools, called a synthesizer and place and route tool (see chapter 3), the target FPGA device, along with some basic clocking information, the tools can sort out how to connect a 16-bit multiplier

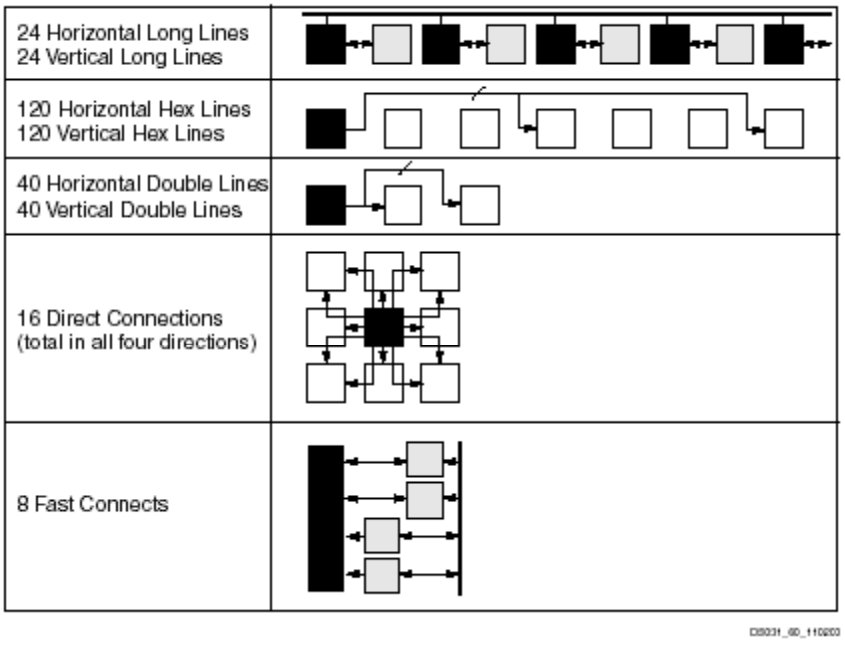


Figure 2-11 : Hierarchical Routing Resources

to a 16-bit adder and a 16-bit storage register (sometimes collectively called multiply and accumulate, or MAC). Figure 2-11 [8] illustrates how such components of a MAC, built out of the smaller building blocks in a slice, can be connected together across the FPGA. The CLB contains fast connects for quick routing between slices in a CLB. The direct connect lines allow connection between any neighboring CLB's, in all directions, including diagonally. Each CLB also has the option of routing signals to every other CLB in all four directions with the double connect lines. These lines can be driven at their endpoints, but have the ability to receive at the endpoints or at each skipped block as well. The hex routing lines allow CLB to route to every 3rd or 6th CLB in all four directions. The hex lines can only be driven one direction, and can receive at either the 3rd or 6th block. Finally, the horizontal and vertical long routing lines are bi-directional paths that span the entire FPGA.

There are also dedicated routing resources, three of which have been explained previously; the carry chain, SOP chain, and dedicated shift chain. Additionally, there are 2 tri-state drivers per CLB that can drive 1 of 4 on chip bus lines per CLB row. Most importantly, there are dedicated global clock routing nets throughout the FPGA. The device is broken up into four equal quadrants, and up to 8 different clock nets can be utilized per quadrant. The digital clock managers (DCM) shown back in Figure 2-2, are powerful clock managers that offer clock de-skew, frequency synthesis, and phase shifting of any externally or internally created clock. The combination of dedicated global routing to all synchronous elements such as RAM's and Registers with the powerful clock management options provides nearly unlimited clocking options.

However, depending on the type of the design, frequencies above 200 MHz in Virtex devices

generally become difficult to optimize, especially for multiplier and adder intensive designs such as DSP algorithms. In general, most designs will have only one or 2 clock domains. In this case, the routing works as follows. The clock comes into the device on one of 16 dedicated clock pins, through an input buffer, and then gets directed to the DCM's. From here, the DCM routes the signal to 1 of 16 global clock multiplexers. From here, the multiplexers distribute the clock on one of the 8 dedicated clock nets to each of the 4 quadrants. Finally, that clock net is fanned out into several rows, and each row can clock 16 CLB's in each direction (8 up and 8 down.)

2.3 18Kb Block RAM and 18x18 Dedicated Multipliers

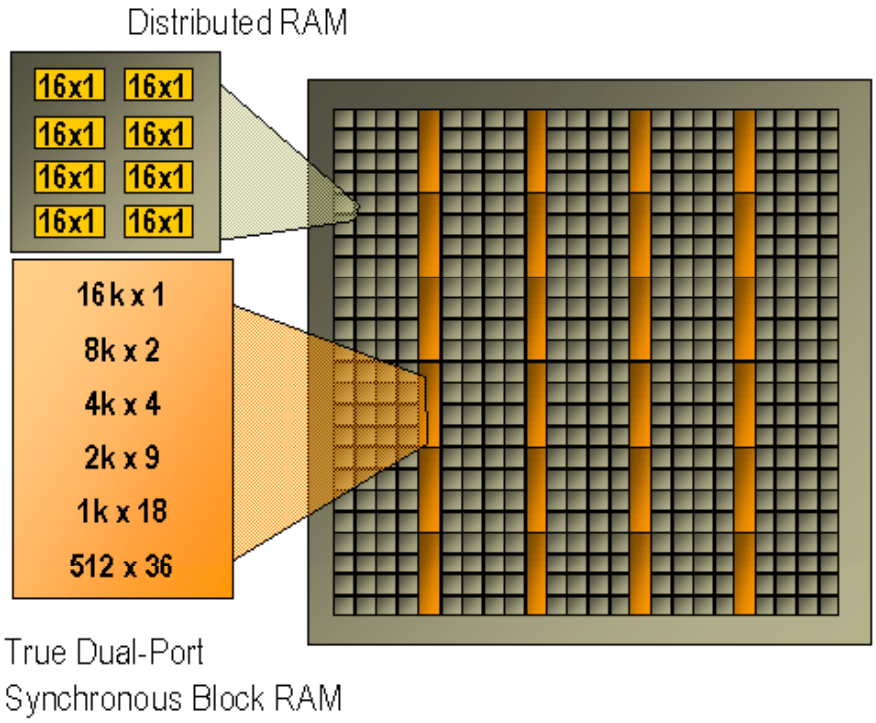
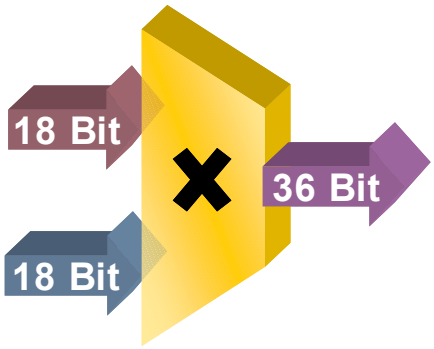


Figure 2-12 **:** Distributed and BRAM Location in V2P FPGA

The last 2 building blocks of Figure 2-1 and Figure 2-2 that have yet to be discussed are the 18x18 Block RAM's (BRAM's) and the 18x18 Multipliers. The Virtex-II Pro block SelectRAM+ is an 18Kb dual port RAM with independent synchronously clocked ports that access the same storage data. They can be configured in dual or single port mode. In single port mode, they can be programmed to be between 16Kx1 bit and 512Kx36 bit, with the most notable configuration being 1Kx18 bits. In dual port mode, the largest either port can grow to is 512x26 bits, and in this case the most notable configurations being those that are either 18 or 36 bits wide. The importance of the

notable configurations will be explained shortly. Figure 2-12 [4] illustrates both the Distributed RAM and the BRAM within the CLB array structure of the FPGA.



Signed Multiply Performance	
Virtex-II	
18 x 18	245 MHz
Virtex-II Pro	
18 x 18	300 MHz

Pipelined multiplier with registered inputs and outputs

Figure 2-13 : Virtex-II Pro Dedicated Multiplier

Figure 2-13 [4] illustrates a block diagram of how the dedicated 18x18 bit, 2's complement signed multipliers in the Virtex-II Pro architecture work. These multipliers are optimized for much higher speeds and much lower power consumption than multipliers implemented in the CLB's. As Figure 2-1 illustrates, an 18x18 multiplier and an 18Kb Block RAM share resources and are connected to 4 switch matrices. Therefore, a multiplier can be associated with a BRAM for quick access to memory, it can be used separately, or the BRAM can be used separately. However, if the multiplier is utilized, only 18 bits of the BRAM inputs can be used since the multiplier shares its inputs with

the upper data bits with of the BRAM. The reason that the 18 and 36 bit wide BRAM configurations are of special interests is that the sharing of interconnects between it and the multiplier has been optimized for 18 and 36 bit wide Block RAM resources feeding the multipliers. By utilizing a SelectRAM+ memory feeding a multiplier accompanied by an accumulator (that is an adder and a register) in the LUT's, DSP multiplier-accumulator (MAC) functions commonly found in digital filter bank and FFT implementations may be created.

2.4 Analysis of Dedicated Multiplier Implementation in the V2P

There will be additional explanation on the implementation MAC's in FPGA in chapter 4.1. However, in order to understand the advantages of executing MAC functions in FPGA's, it is beneficial to examine how multiplication is actually implemented in the hardware of an FPGA. The accumulate function has more or less been explained by

Figure 2-8, although it will be revisited many time through out the paper. Essentially, accumulate means to add and hold the value until the next product is calculate. However, due to parallelism, only some registering is necessary.

To begin understanding how multiplication works in hardware, it must be made clear that multiplication, in hardware logic or on paper, results in a growth in the number of digits needed to represent the result. This growth is proportional to the size of the inputs, and *must* take into account that each input *could* be at its maximum. Given a 3 bit signal 'a' and 4 bit signal 'b', the number of bits needed to represent the output of the

multiplication of (a x b) can be calculated as $\log_2(2^a \times 2^{b)})$, which reduces to a + b = 7 bits. For example, if a = 5_{10} (101_2) and b = 12_{10} (1100_2), then the resulting output, 60_{10}, should be represented with 7 bits as 011_1100_2. Although 60_{10} can be represented with only 6 bits, this would not be the general case. If 'a' was changed to 7_{10}(111_2), then the result would be 84_{10} (101_0100_2), which does require 7 bits. Thus the maximum range of each input (2^n) must be used to account for all possible products. However, this is not always the case, and this concept will be revisited in the discussion of signed numbers chapter 6.2.1.

Figure 2-13 illustrates that the dedicated Xilinx multiplier inputs are always assumed to be 18 bits each, and thus the product must be 36 bits. At first, this may seem like a lot of wasted resources if one only intends to build a system based off from 16-bit operations, for example. First, let's examine why this makes sense from a hardware implementation point of view, and then look at it from an algorithmic point of view.

The V2P multiplier blocks are Xilinx intellectual property; so they do not release data sheets that describe exactly how these multipliers function. However, using a simple example from elementary school, a general concept can be understood. Take for example the multiplication in base 10 of a=1,000 times b=1,001 = 1,001,000 in Figure 2-14. In

```
        1 0 0 0  Multipicand
    x   1 0 0 1  Multiplier
      ---------
        1 0 0 0  Intmediate prod 1  Multiplicand shifted 0 digits
      0 0 0 0    Intmediate prod 2  shift 1 digit, but zero
    0 0 0 0      Intmediate prod 3  shift 2 digits, but zero
  1 0 0 0        Intmediate prod 4  Multiplicand shifted 3 digits
---------------
  1 0 0 1 0 0 0  Product
```

Figure 2-14 : Base10/Base2 Multiplication Example

elementary school, we were taught how to multiply larger values by finding the product of the multiplicand and the first digit of the multiplier, then simply shifting over by one digit, and calculating the product of the multiplicand and the next multiplier digit, and so on. When there are no more multiplier digits, the sums of all the intermediate products are calculated. Although it was probably not explained to us as kids that we are really just executing a series of comparisons, shifts, and adds, this is however the underlying method for any hardware multiplier to obtain a product. The comparison is to check to see if bit 0 (LSB is bit zero here) of the multiplier is 1 or 0. If it's 1, place a copy of the multiplicand in the intermediate product. If it's a zero, place zero in the intermediate product. Now

shift over one multiplier digit and recheck it, and place the appropriate value (either multiplicand or zero) in the next intermediate product, but shifted to the left one digit. This process is repeated until the last multiplier bit is reached, and then all the intermediate products are summed. The example in Figure 2-14 was chosen since it serves as a multiplication example in both base 10 and base 2. The same exact procedure is followed in hardware to execute multiplication: compare, shift, and add.

Back in the introduction is was mentioned that one of reasons that FPGA's serve as superior DSP engines over the traditional DSP processors is mass parallelism. The parallelism comes from, in part, how FPGA vendors such as Xilinx create high-speed multipliers. In traditional microprocessors, the above explained elementary school multiplication procedure can be turned into a simple algorithm run on a CPU. Optimized, the CPU hardware only needs 4 main pieces of hardware to implement multiplication: an N-bit register and a 2*N-bit register, a logical shift right, a comparator (AND gate) and an adder. The N-bit register holds the multiplicand, and the 2*N register holds the intermediate product in the MSB's, and the multiplier in the LSB's. The comparator checks the LSB of the 2N register, and then adds the appropriate value (either multiplicand or zero) to the MSB of the 2N register, and then executes a logic shift right on the entire contents. This is repeated N times, and the resulting product is located in the 2N register. In a DSP microprocessor, this algorithm must be executed iteratively; that is, it can only work on one intermediate product at a time, since there is only one set of registers to hold (accumulate) data and one execution unit (the adder). For an 18x18 bit multiplier, it would take 18 iterations through the shift and add algorithm to execute the

multiplication of two 18-bit values. If each iteration took on the order of 5 clocks (assume and ideal 5 stage pipelined processor), and the clock speed was 100 MHz, then it would take 5*18 = 90 clocks * 10 ns = 190 ns to execute just one multiply. Many DSP algorithms use coefficient tables (Sine tables). If there are 256 entries in the table that must be multiplied, then it would take that DSP processor, at best case, 256*90 = 23040 clocks * 10 ns = 230.4 us to calculate all the products.

	{0000, (a3a2a1a0) AND (b0) }	Multiplicand ANDed with bit zero of Multiplier and shifted left 0
+	{000, (a3a2a1a0) AND (b1), 0 }	Multiplicand ANDed with bit 1 of Multiplier and shifted left 1
+	{00, (a3a2a1a0) AND (b2), 00 }	Multiplicand ANDed with bit 2 of Multiplier and shifted left 2
+	{0, (a3a2a1a0) AND (b3), 000 }	Multiplicand ANDed with bit 3 of Multiplier and shifted left 3
	xxxx_xxxx	**Product**

Figure 2-15 : Hardware Simplified Version of a 4-bit Multiplication

However, if the procedure of Figure 2-14 is analyzed from more of a hardware approach, the process becomes even simpler. In the case of the Xilinx multipliers, it is a given that the inputs are each 18 bits wide, and the product must be 36 bits. For simplicity, use the 4-bit example of Figure 2-14. If it is always known that the inputs are 4 bits and the product is 8 bits, then the process can be drastically simplified, as shown mathematically in Figure 2-15, and schematically in Figure 2-16. Since the input and

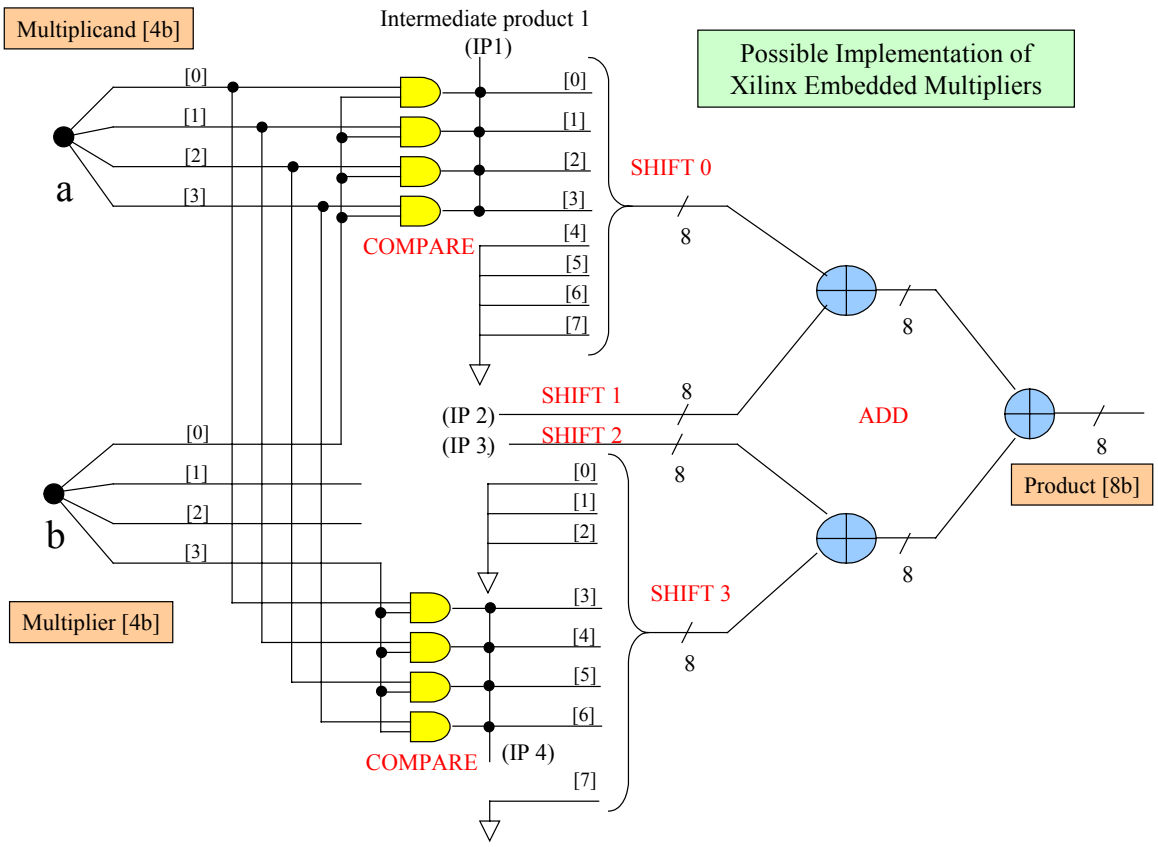


Figure 2-16 : Hypothesized implementation of V2P Dedicated Multipliers (4-bit)

output lengths are fixed, then the number of shifts is already known. However, no shift registers are required. The Multiplicand is simply compared (ANDed) with each bit of the multiplier. The results of the compares, the intermediate products, are shifted to the correct bit position within an 8-bit value by simply hard wiring them to correct location. The other 4-bits in the 8-bit value are set to zero. The four shifted 8-bit intermediate products are then summed. Since the largest value is only 7 bits, there is no need to be concerned with the carry, and thus 7 full adders and one half adder is needed per

summation block in Figure 2-16. A 4-bit implementation requires 2 stages of 8-bit adders.

Inside the multiplier, the Xilinx FPGA architects have access to the same transistor level circuitry that is available on an ASIC. Therefore, they are able to string together long chains of adders. In the case of the 18-bit Embedded Multipliers, 5 stages of 36-bit adders are daisy chained to calculate one 18x18 bit product. According to Figure 2-13, timing can be met up to about 300 MHz with 18-bit multipliers. However, this frequency value makes many assumptions. First, it implies that the both the input and output of the multiplier are registered. Further, the V2P dedicated multipliers have an option to place a pipeline register(s) somewhere in the middle of the 5-stage tree. Furthermore, it assumes that the there are no immediate multiplies that must follow this tree. This would grow the next product to 36+36 = 72 bits, and the next to 72+72 = 144 bits, and so on. Clearly this is not feasible, and the data must be clipped, or quantized, as will be discussed later on in the analysis portion of this research (see chapter 6.2.1). In addition, many DSP algorithms, such as digital filters, implement summation equations, which are basically multiply and add sequences. Looking at the bit growth issue combined with the parallel logic resources necessary to keep up with a series of 18 bit multiplies and adds, it should be clear that maximum clock rates for practical applications utilizing the 18x18 bit multipliers is probably more nearly between 100~200 MHz. Going back to the original example of a table of 256 18-bit filter coefficients, in the FPGA each 18-bit product can be calculated in one clock cycle. So instead of taking 190 ns per coefficient in the ideal 5-stage pipelined DSP CPU example, the FPGA can do it in 10 ns.

Furthermore, in the larger V2P devices, there are up to 444 multipliers. Therefore, it is possible to execute all 256 18-bit multiplications simultaneously in one clock cycle, with nearly 200 multipliers left over for other functions. Compare this to the 230.4 us for the DSP microprocessor. Despite the crude assumptions that were made in this example, the power of parallel processing offered by FPGA's should be clear.

2.5 **Analysis of CLB implemented Multiplier in the V2P**

Despite the speed and power gain of the dedicated multipliers, the CLB's do have efficient means to create multipliers as well. Recall the slices of Figure 2-4 have a building block called MULT_AND to implement slice based multipliers. However, how does Xilinx implement this affectively? If the traditional shift and add algorithm for multiplication were implemented in the CLB's, it would be very inefficient due to the large number of partial products that need to be together, exhausting the LUT's and adding timing delays for each one. However, just as the addition algorithm could be optimized and implemented in a different manner in the FPGA, so can the multiplication algorithm. If the multiplier is restricted to 2 bits, such that the result is either 0, 1, 2, or 3 times the multiplicand, all 4 of the products can be realized with just a shifter and an adder, as illustrated in the block diagram of Figure 2-17. The multiplexer, controlled by the 2-bit multiplier, selects the appropriate version of the multiplicand. Two times the multiplicand is obtained through the logical shift left. Three times the multiplicand is

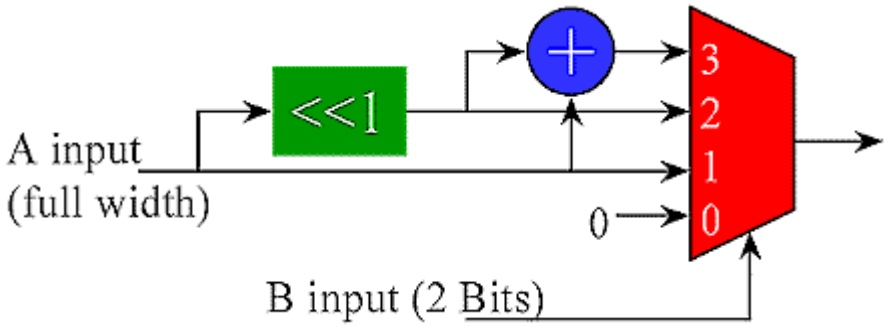


Figure 2-17 : Computed Partial Product Multiplier

easily obtained by adding the multiplicand to a left shifted version of itself (2x + 1x = 3x). Notice that this particular configuration does not have any limitation (within reason) to the number of bits in the multiplicand. This type of multiplier is known as a computed partial product multiplier. The savings in hardware with this approach is that all the 0, 1, 2, and 3x inputs to the multiplexer are exactly the same for all partial products, and thus one adder can be shared by all of the necessary partial products. (Note that shifts in hardware are actually implemented with simple wire routing and tying off the upper bits to the appropriate value of 1 or 0, depending on whether implementing a logical or arithmetic shift for signed/unsigned numbers, respectively).

The Xilnx Virtex-II Pro CLB's implement a computed partial product multiplier in very much the same way the 2-bit adder of Figure 2-8 works. Figure 2-18 illustrates the implementation of a Nx2 bit computed partial product multiplier in a V2P slice. Just like the adder implementation, the XOR gate in the LUT calculates the main part of the addition, and the dedicated XOR gate and carry chain multiplexers handle the carry. Just

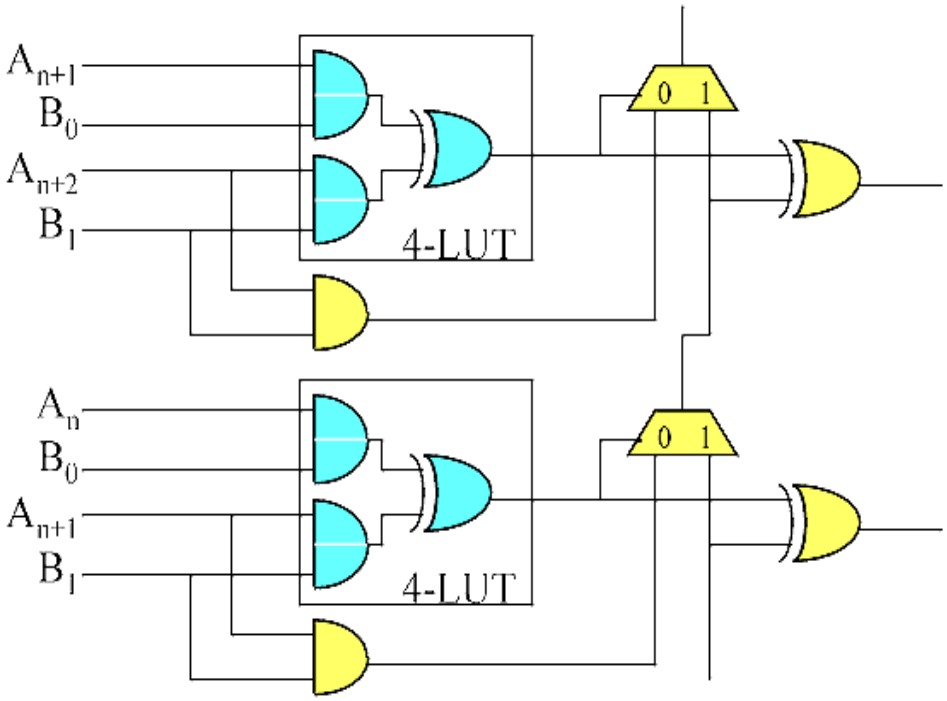


Figure 2-18 : Implementation of Nx2 Computed Partial Product Multiplier in a Slice

as in the adder, one of the inner terms of the LUT table needs to brought out to control the

carry chain multiplexer. It is the dedicated MULT_AND gate that allows for the optimization of both the addition and multiplication algorithms to be able perform 2 bit adds/multiplies within one slice. Notice that as you move up the carry chain, the 2-bit multiplier remains at the LUT inputs, but the multiplicand has been hardwired to the proper locations on the 4-input LUT's to create the proper shift. The LUT logic has been optimized to compute the partial products. In this case it is helpful to view the partial product output of each LUT as being the final summed up product, but without any carries propagated. The external XOR gate and carry chain make the final adjustment to each bit of the partial product to obtain the final correct product.

2.6 **PPC and IBM Core Connect Architecture**

The last building block of the V2P architecture is the IBM PPC hardcore. At the heart and sole of every SOC is a CPU, and in IBM's Core Connect Architecture approach, it's

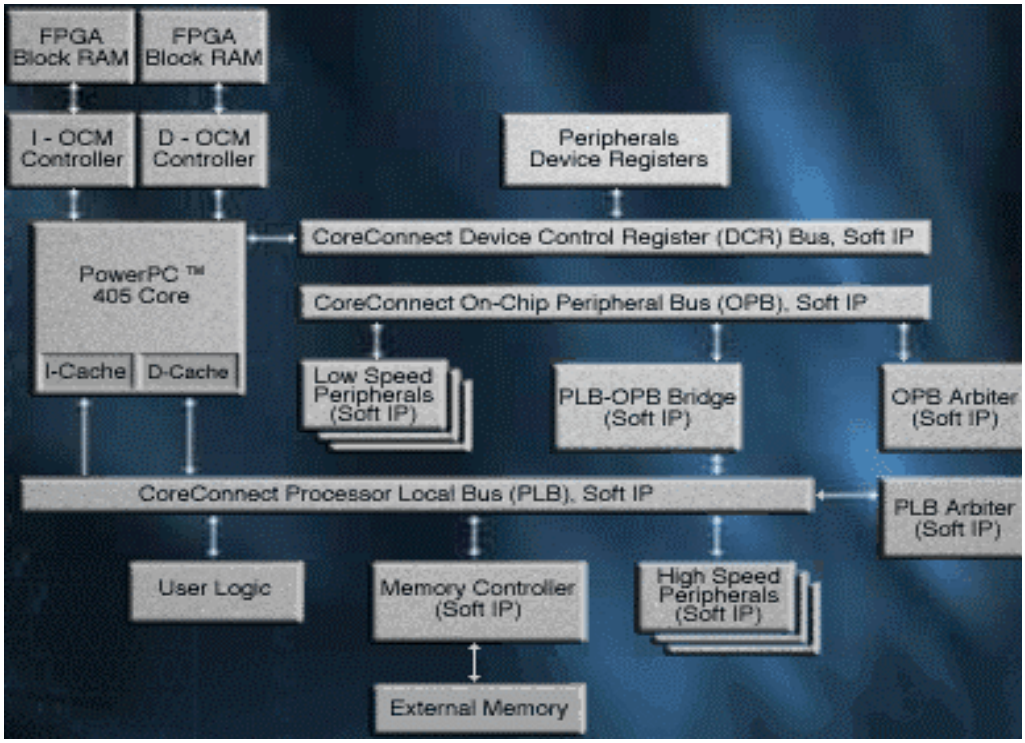


Figure 2-19 : PPC and IBM Core Connect Architecture

the 405 PPC, a RISC (load/store) processor, as shown in Figure 2-19. There are some key must have components in the architecture, and these components are called cores. The instruction and data cache of the PPC is connected to the processor local bus, or PLB. This bus is a 32 or 64 bit bus with separate read and write data paths that run at 133 MHz. To control this bus, there must be a bus arbiter core. There is also another bus,

called the off chip peripheral bus, or the OPB. This bus connects to the PLB through the PLB to OPB bridge core, runs at 33 MHz, and is also a 32/64 bit interface with separate read and write buses. To control this bus, there is an OPB arbiter core. In the IBM core connect, there are several other cores that can be connected to the buses, depending on the system needs. Lower speed peripheral cores, such as SDRAM controllers, Ethernet controllers, serial controllers, PCI controllers, etc, hang off from the OPB. The OPB in general is the interface to the outside world for a traditional SOC implementation. Higher speed cores, such as DDR memory, and most importantly, the user defined cores, hang off from the PLB. The whole point behind a SOC is for the design of a system algorithm that is both software and hardware intensive. Tradeoff studies can be done to find the optimal solution. The portion of an algorithm that runs faster and more efficient in hardware becomes a user defined core, hanging off the PLB. There is also one more bus, called the Device Control Register, or DCR bus. This bus is 32 bit bidirectional bus that connects all the cores that hang off from PLB to the processor via a ring-like chain. This bus is used to set up control registers in the user cores. Each core, including the PLB arbiter and PLB2OPB bridge, must have a DCR in and out port, and a dedicated DCR address for that core. If the address is for that core, it will respond; otherwise it passes the address and data on to the next core in the chain. The cores hanging off the OPB also have device control register that can be controlled via software from the PPC. However, they are not connected on the DCR chain. Instead, these control registers are written through the general memory map of the entire SOC (i.e. out the PLB and over the OCB)

The important concept to understand about the cores is the difference between a soft core and hard core. A hard core is a piece of intellectual property (IP) hardware

logic that has not only been predefined in terms of functionality and cannot be altered, but has also been physically placed inside the fabric (either ASIC or FPGA) of a chip. A soft core is also IP logic that is usually not open source, but this type of core has not been physically placed. It is more or less predefined functionally, but it is "soft" in that the system engineer may choose where and how to implement it in the fabric a the chip. The only core in the Xilinx FPGA that is a hardcore is 405 PPC itself; every other core mentioned is a soft core. Any of the necessary soft cores (such as PLB bridge) are generally included when a user buys the Embedded Development software package, the tool set for designing with the IBM Core Connect. (see chapter 3). There are many other commonly used cores that are available in the Xilinx IBM Core Library. These generally cost dollars per core to utilize, since they are IP.

There are a couple of cores in Figure 2-19 that have not been described yet. In order to make IBM Core Connect more efficient on the FPGA, Xilinx took the liberty of adding some additional cores, namely, the off chip memory controller (OCM's) that connect directly to the PPC core. These are not part of the original core connect architecture, but provide a fast, dedicated interface from the PPC to the Block RAM's.

3. Xilinx FPGA Design Flow and Software Tools

The basic flow for designing with most every FPGA vendor is illustrated in Figure 3-1. This flow diagram also illustrates the software tool or package used specifically in the Xilinx design flow. Xilinx owns and maintains a complete tool set for the entire FPGA design flow, some of which is in collaboration with 3rd party companies. Essentially, all of their tools are integrated under one umbrella called the Integrated

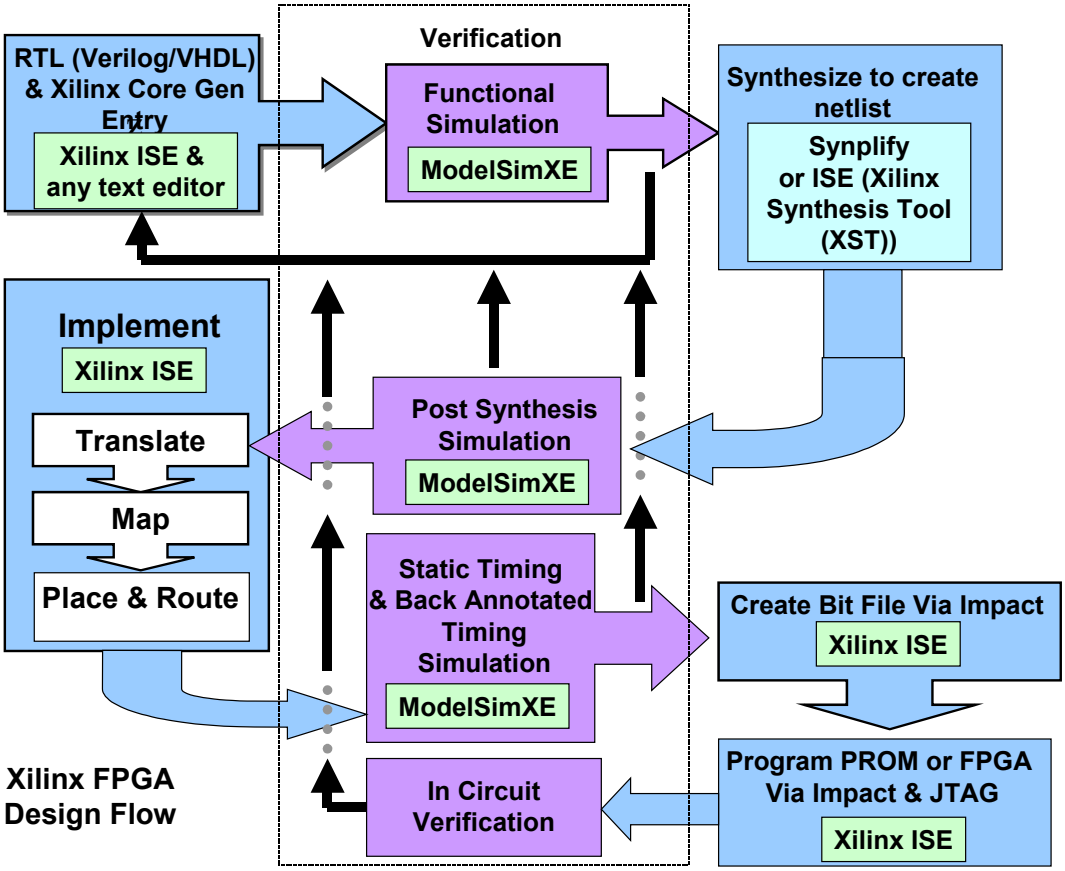


Figure 3-1 : Xilinx FPGA Design Flow

Software Environment (ISE) package.

Once the overall design and budget has been defined from the system level, the FPGA design flow can begin. Today, most designs begin with a hardware description language (HDL), either Verilog or VHDL, which can be entered using any basic text editor. Sometimes Verilog and VHDL is referred to as RTL, or register transfer logic. HDL's support mixed levels of abstraction, such as behavioral (ex. sum = a+b+cin), data flow (ex. sum = a XOR b XOR cin), and structural (ex. instantiation of pre-existing blocks or netlists). Multiple levels of abstraction can be built upon each other to create hierarchical designs. These abstraction levels allow for incremental and iterative refinement of the gate-level implementation of the system design. Included as part of ISE software suite is a tool called Core Generator. This is a GUI based tool that offers a designer parameterized logic cores that have been optimized (for area & speed) to be implemented on Xilinx FPGA's. This catalog of ready-made cores offers functions that range from simple counters, comparators, adders, and multipliers, to full system-level building blocks, such as memories, FIFO's, filters, and transforms. The structural abstraction level of HDL's allows logic designers to implement these cores right into the source code as a netlist. It is also possible to manually instantiate any Xilinx macros, like those listed in architecture description of chapter 2.

Once the source code has been complete, it needs to be functionally verified. Functional verification tests the design to determine if it is working as intended. First a test bench, which is a high level HDL stimulus file that wraps around a design to stimulate the inputs of a design, is created. Depending on the type of design, sometimes

these test benches are more advanced and also monitor the outputs of a design by providing additional logic to check the results, and thereby creating automated self-checking test cases. In either case, it is necessary to visually verify and debug the signals at all levels of hierarchy in the design on a wave simulator. A company called Model Technology (MTI) produces an HDL simulation and debug environment called ModelSim. If an HDL design is purely behavioral, the simulator will most likely be able to properly simulate the design. However, since the HDL source code may contain either cores generated by the Xilinx Core Generator, or lower level macros specific to the FPGA architecture, a set of Xilinx libraries that indicate how each of the higher level blocks should behave must be made available to the simulator. Xilinx solves this issue by working with Model Tech to produce their own version of ModelSim, called ModelSimXe (MXE), which includes all necessary Xilinx libraries to functionally verify any level of design abstraction for any Xilinx FPGA or CPLD. In addition it includes a text editor for HDL entry, as well as a source code syntax checker. ModelSim can be run as a stand-alone program, or it can be executed from within ISE.

During synthesis, the third block in Figure 3-1, the HDL source code, including any Xilinx specific macros and cores, are translated into a gate-level structural netlist. This netlist is then optimized for implementation on a Xilinx device. Synthesis tools contain a syntax checker as well. There are two synthesis tools used in the Xilinx FPGA design flow. As part of the ISE suite, Xilinx offers their own synthesis tool, simply called Xilinx Synthesis Tool, or XST. The ISE tools also support a 3rd party synthesis tool called Synplify, produced by Synplicity. This synthesis tool is an industry standard tool with

design libraries available to support nearly every major FPGA and CPLD platform. Although both tools essentially yield the same final result, Synplify is generally the synthesis tool of choice, especially if the FPGA design is a test platform for an ASIC implementation, which Synplicity also supports. The input file types to a synthesizer are either .V (Verilog) or .VHD (VHDL), with the output file type of Synplify being an .EDIF (electronic data interchange format) file, and the output file of XST being an NGC (native generic database) file. Since the netlist has not been mapped into Xilinx specific building blocks at this stage, synthesis tools cannot give accurate timing results in its timing and area log files (.srr file), only estimates. However, this information can be used to get a heads up on any really obvious timing problems.

Another output of a synthesizer is a Verilog or VHDL netlist (.vm or .vhm from Synplify) that can then be reentered back into the ModelSim simulator to verify that the output of the synthesis step equals the original functional simulation results. This is sometimes called post-synthesis simulation, or gate-level simulation. These additional output files are for functional simulation only and do not contain any timing information. Notice in Figure 3-1 that that the verification of a design is an iterative process that continues throughout the design flow. This design flow is very close to that of an ASIC design flow, and this step of the process is rarely skipped with ASICS. However, due to the quick re-configurable capability of FPGA's, this step is often skipped, especially with smaller FPGA designs.

The output of the Synthesis tool is then fed into the next stage of the design flow, which is called Implementation in the Xilinx flow, and is the core utility of

the ISE software suite. Before this step is executed, the user constraints file (UCF) is typically filled out. The most critical information in the constraints file is the pin locations for each I/O specified in the HDL design file, and the timing information, such as the system clock frequency. The constraints file also allows a designer to specify specific mapping of gates in the netlist to specific Xilinx blocks, as well as the placement of these blocks. Further it allows specific timing constraints on a per I/O basis for any critical timing paths. ISE contains a built in GUI, called PACE (pin and constrains editor), for the purpose of entering all the constraints. The Implementation step of Figure 3-1 reads in the constraints file, and consists of three major steps; translate, map, and place and route. The translate step essentially flattens the output of the synthesis tool into a large single netlist. A netlist in general is a big list of gates (typically NAND/ NOR) and macros (memory, Xilinx specific macros, etc), and is compressed at this stage to remove any hierarchy. The Map step groups the logical symbols in the flattened netlist into physical components specific to the target device, such as CLB's, slices, IOB's, Block RAM's, and dedicated Multipliers. The Place and Route step then places each of these physical components onto the FPGA chip and connects them through the switch matrix and dedicated routing lines (see chapter 2). Then, timing information is generated in log files that indicates both the propagation delay through each building block in the architecture, as well as the actual routing delay of the wires connecting the building blocks together. The ISE Implementation stage outputs a NGD (native generic database) file.

Just as the synthesis tools output a Verilog or VHDL simulation netlist, so do the ISE Implementation tools. However, this time these simulation files contain all of the

timing information that was generated in the Map and Place and Route stage of Implementation. These files can be used for 2 purposes. First they can be read back into the ModelSim simulator just as before. This is called back annotated timing simulation. The design is again check to see if it equals the original functional and post-synthesis simulations from earlier in the process. This type of simulation is much more time consuming and difficult, since all of the propagation and wiring delays are evident on each signal. Second, they can be use for static timing, that is, timing analysis that does not depend on stimulus to the design circuit. This step is crucial in ASIC design flows, and is also available in the FPGA design flow through a ISE tool called TRACE.

Notice once more the iterative verification process that leads back through the RTL entry and HDL simulator. Once the design had been fully verified with the correct timing, then the final bit file that will eventually be loaded into the FPGA configuration SRAM (see chapter 2) can be created. The generation of a bit file is also executed from the main ISE tool. Once the bit file has been created, another tool in the ISE suite called IMPACT is used to program either the FPGA directly, or a set of EEPROM's, through the JTAG interface. The standard IEEE.1149 JTAG (see [5], chapter 3.4) cable connects to a computer through the parallel port. For direct programming, the circuit card with the target FPGA device must be powered up and the bit file is loaded into the FPGA via the IMPACT software. If the EEPROM's are used to program the FPGA upon power up of the circuit card system, then IMPACT is use to create PROM configuration files from

the bit file. Then IMPACT is use to program the EEPROM's, which are also on the JTAG programming chain.

Once the FPGA is programmed, the in-circuit verification may begin. Typically, there is extensive additional test hardware and software development required in order to stimulate and capture results from the circuit card system housing the target FPGA. Again, this debug and verification process is iterative, and often requires cycling back to the RTL and functional simulation stages in order to fine tune the design.

4. DSP ALGORITHMS

4.1 Analysis of FFT MAC Functions In Hardware

Most DSP algorithms inevitably involve some sort of time to frequency transformation of a time-limited block of data. This transform can be accomplished in several ways, the most common of course being either the Discrete Time Fourier Transform (DFT) or a Filter Bank, and in the case of some algorithms such as MPEG Audio, both. The DFT is most often implemented using the Fast Fourier Transform algorithm. Matlab Filter Bank (FB) implementations, although typically modeled as a bank of overlapping frequency channels, are typically re-worked mathematically into some sort of equivalent, more efficient code-like algorithm. In either case, their mathematical representations generally break down to a multiplication inside of a summation; i.e. multiply and add. Thus, efficient implementation of DSP algorithms in hardware relies on the technology's ability to handle the multiply and add, or essentially the multiply and accumulate (MAC) function.

As an example, assume a given DSP algorithm needs to calculate an N-point DFT. First, to be clear, the "DFT is the 'mathematical entity' used to perform spectral analysis on a signal", while the "FFT is the 'computational entity' used to efficiently compute the DFT of a signal" [3]. The DFT of an N sample block of an input signal, x[n], where n = k = 0, 1, 2,…, N-1, is defined as

$$X(k) = \sum_{n=0}^{N-1} x[n] e^{-j 2\pi k n / N} \quad . \qquad (1)$$

Equation 1

To calculate just one DFT sample (i.e. one *X(k)*), it requires N multiplies and N-1 additions, or about N MAC operations. If N is large, say 512 samples, then direct computation of a single DFT sample is already resource intensive, never mind the entire computation, which is on the order of N^2 MAC operations. However, the Fast Fourier Transform algorithm was designed to reduce the computational complexity of the DFT, while maintaining *exactly* the same results. There are several different flavors of FFT algorithms, such as Radix-2, Radix-4, Chirp Fourier Transform, and Zoom FFT. For simplicity, a radix-2 FFT is used as an example. The Radix-2 can be executed as a decimation in time algorithm, or a decimation in frequency algorithm. The details here are beyond the scope of this discussion, but suffice to say there is no practical reason to choose one over the other, however, the decimation in time is most often used in text books an is chosen here as well.

The complexity of a standard Radix-2 FFT algorithm is defined as $N/2\ Log_2\ (N)$ multiplies and $N\ Log_2\ (N)$, or the required number of total MAC operations is on the order of $N\ Log_2\ (N)$. Compare this with the computational complexity, N^2, of the DFT above in Equation 1. So in the example of a 512 point FFT, it can be seen that the FFT reduces the computational complexity by a factor of $N^2/(\ N/2\ Log_2\ (N))$, or $512^2/(512/2*9) = 113.8$ for multiplies, and likewise by 56.9 for adds! The graph in Figure 4-1 shows the relationship between the DFT & FFT complexity for complex multiplies.

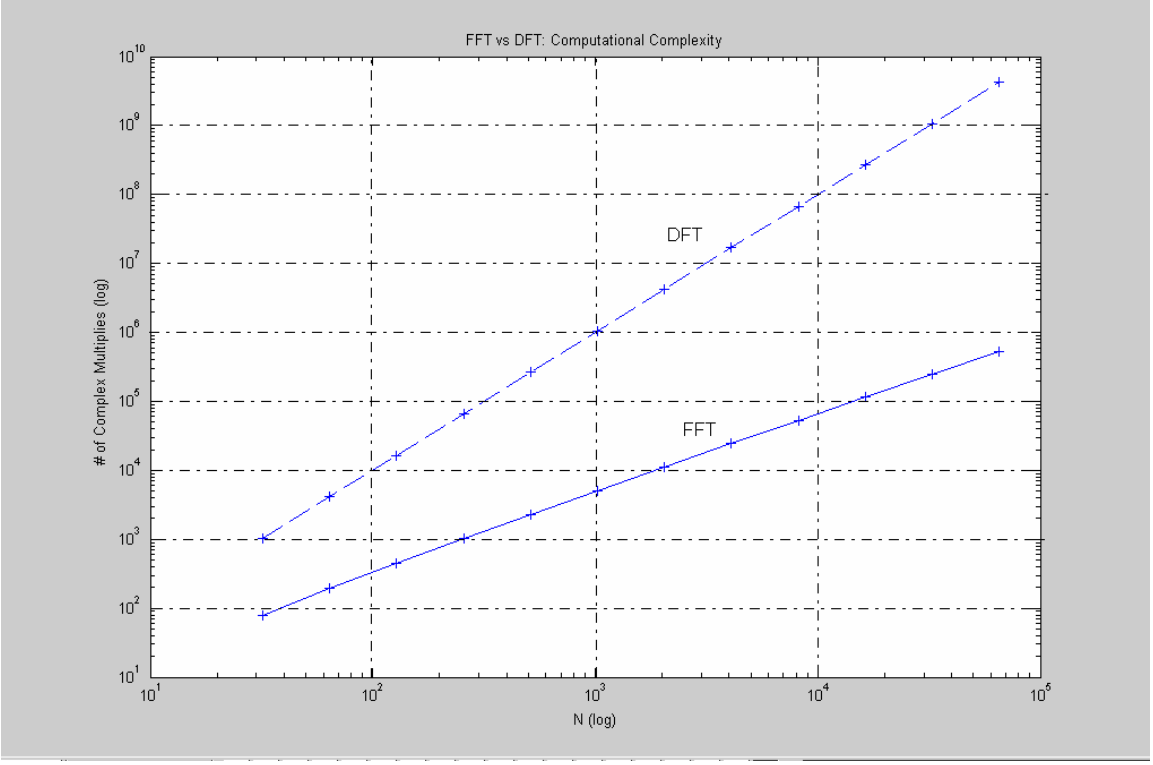


Figure 4-1: DFT vs FFT Complex Multiplies

As was mentioned in chapter 1, although DSP microprocessors run at high speeds and are fairly efficient DSP engines, most can only handle one calculation of a MAC function at a time. That is, typical DSP's only one have one floating point (FP) multiply unit. Figure 4-2 [2] shows a simplified data flow for a typical DSP microprocessor MAC operation. X[n] represents the n^{th} input sample, and CC[n] represents the corresponding n^{th} complex coefficient, such that the input takes on the form $CC[n]e^{-j2\pi n}$. Going back to the 512-point FFT, it can be seen from Equation 1 that it takes 512 MAC operations just to calculate one DFT output. Since the FP multiply unit is shared by all the instructions in

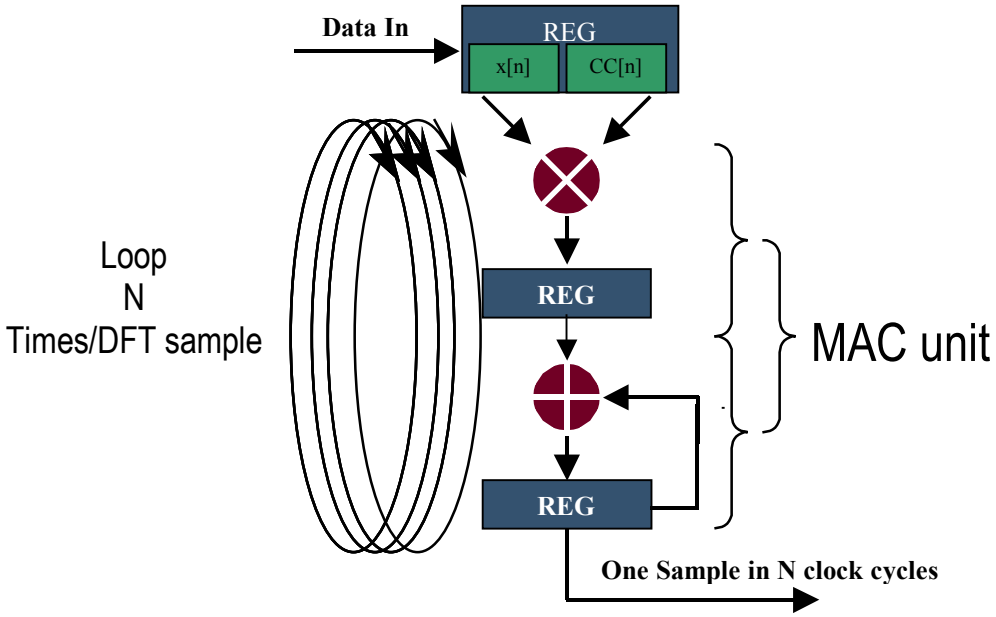


Figure 4-2: Single Cycle Multiplier

the processor, and the processors have sequential pipelined architectures, then each MAC operation must be calculated one clock cycle at a time. This does not imply however that a DSP microprocessor computes one DFT output sample in 512 clocks. Even if the FP multiply unit is pipelined such that you could execute several back to back multiply operations, the FP unit pipes are no where near 512 stages deep. Furthermore, there are more instructions, such load, store, and branch that have to be accounted for. Even in DSP's such as Texas Instruments' 8 instruction deep VLIW TMS320C62xx, the 8 parallel pipelines offer small gains when executing an algorithm that requires N^2 MAC's just to complete the spectral conversion for one N-block input sample. Therefore, the DSP microprocessors take on the order of 1000's, probably closer to 10,000's, clock

cycles just compute one DFT output, let alone the whole N-sample DFT. The FFT algorithm does not fair much better on a DSP microprocessor. Despite the reduction of the computational complexity of the FFT over the DFT illustrated by Figure 4-1, the sequential architecture and single FP execution unit of DSP microprocessors eventually limits efficient implementation of the FFT algorithm as well.

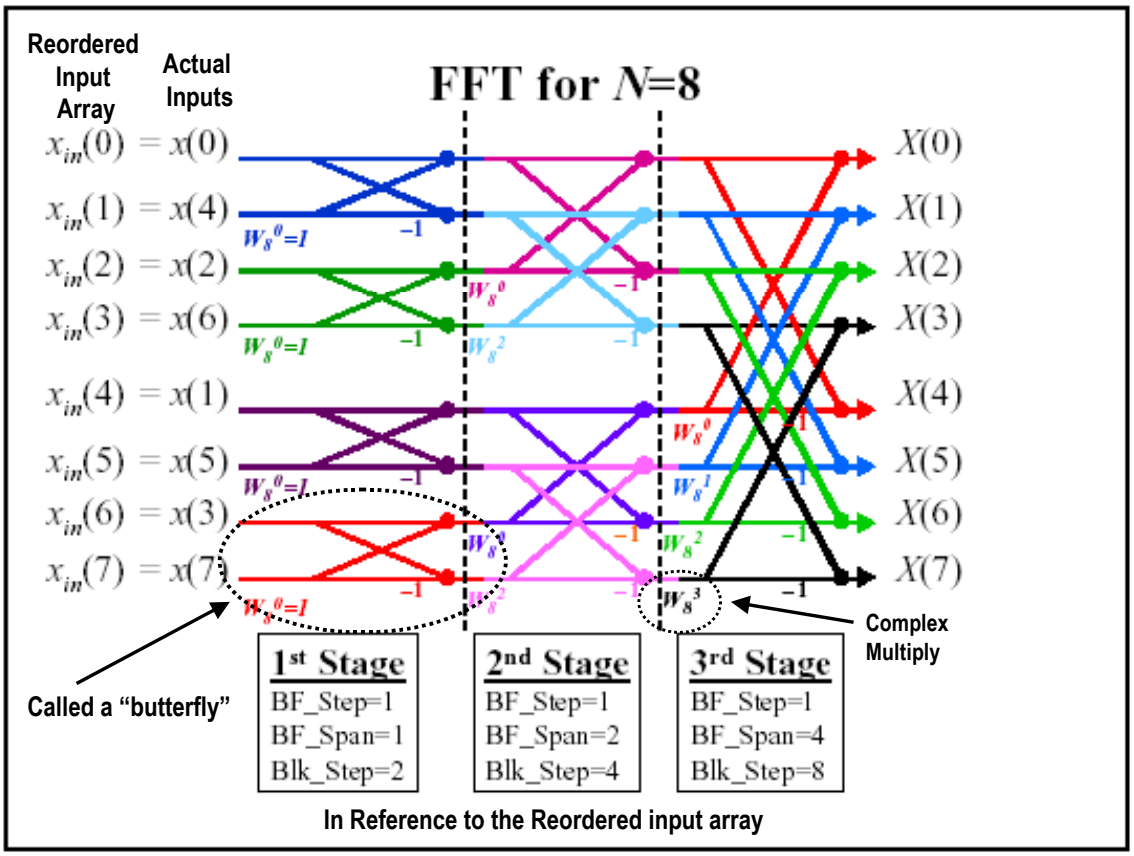


Figure 4-3 : N =8 Radix-2 FFT Butterfly Example

The FFT algorithm uses structures called stages and butterflies to implement the calculation of a DFT, as depicted in Figure 4-3 [3]. For an N-sample FFT, there are

$\log_2(N)$ stages, and each stage has N/2 butterflies. The input to the first stage is a reordered version of the N input samples, called bit reversal, and is simply calculated by executing a binary one's complement on the input indices. The output of each stage is N values, which are the input to the successive stage. Therefore, in order to save memory resources, once the first stage is through, the results can be resaved in the same locations as the N inputs. The butterflies are the structures within each stage that indicate which inputs get multiplied and summed together to form the output for that stage. The dots in Figure 4-3 represent additions or subtractions (multiply by -1). There is one complex multiply per butterfly, and thus there are N/2 complex multiplies per stage. The complex multiplies are called the twiddle factors, and are illustrated in Figure 4-4 for an N=8 point FFT, where $W_N = e^{-j2\pi n / N}$. The butterfly algorithm exploits a property of complex numbers in that the rotation by N/2 in Figure 4-4 [3] results in simply negating a previous twiddle factor. Hence the multiplication by –1 in Figure 4-3, and thus only the first N/2 twiddle factors are necessary for and N-point FFT calculation. A block is defined as a group of overlapping butterflies, and the number of reordered indices that the block spans is the block step (Blk_step). Finally, the number of indices each individual butterfly spans is called the butterfly span (BF_span). Each butterfly within a block receives a twiddle factor, starting at the top and working down. The progression of twiddle factors per block is 0x(N/Blk_step), 1x(N/Blk_step), 2x(N/Blk_step), 3x(N/Blk_step), etc. For example, the Blk_step of the second stage is 2. Since there are two butterflies per block in the second stage, then the two twiddle factors necessary are 0x(N/2) and 1x(N/2), or 0 and 2, giving $W_8^0 = 1$ and $W_8^2 = -j = -\pi/2$. This is illustrated in the 2nd complex unit circle of

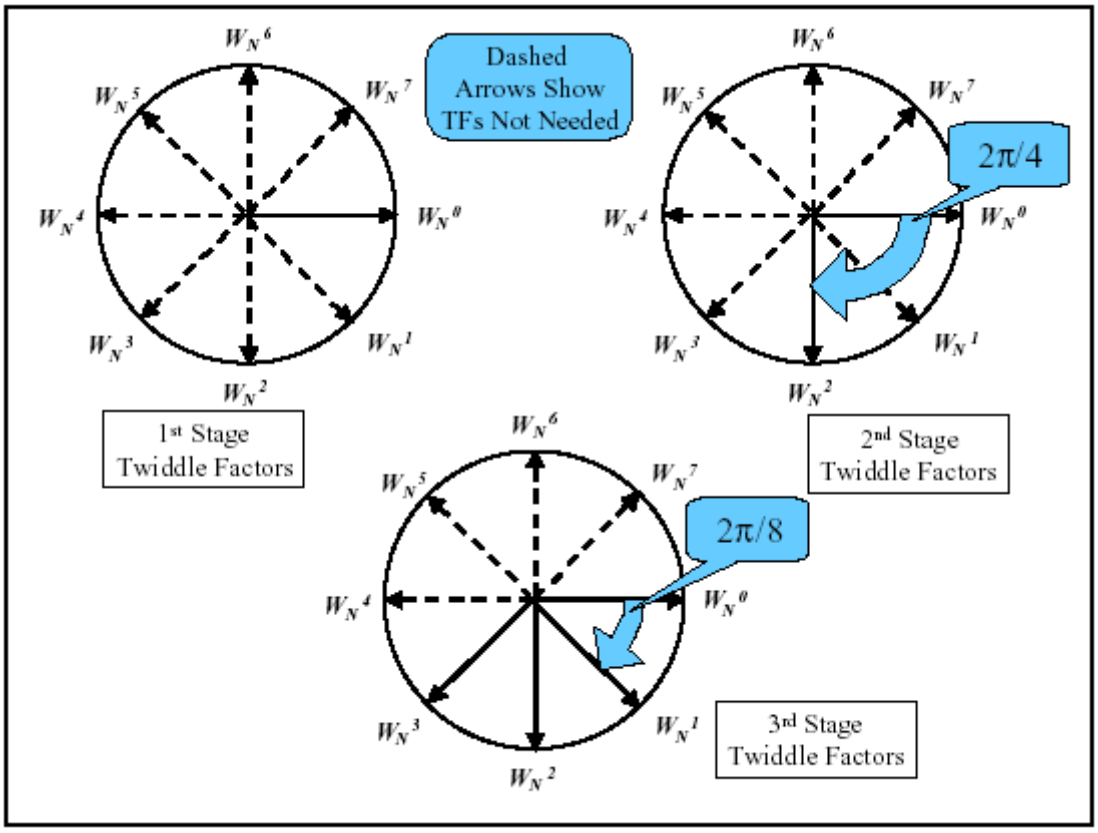


Figure 4-4 : Twiddle Factors for stages of N=8 FFT

Figure 4-4. These twiddle factors will be revisited in the discussion of the Radix-4 FFT algorithm in chapter 6.2.4.

The N outputs of the last stage are the final FFT = DFT outputs. In order to calculate one FFT output sample, all N calculations for each stage must be calculated in order, where as the DFT can calculate the result one sample at a time. Where the FFT makes its gain is that there are only N/2 multiplies per N samples (i.e. per stage), and with $\log_2(N)$ stages, results in N/2 $\log_2(N)$ complex multiples, compared to the N^2

complex multiples for the DFT explained earlier. Nevertheless, these N/2 complex multiples per stage must still execute sequentially in a processor, as shown in Figure 4-2. For the 512-

sample DFT, the FFT provided a gain in computational complexity of about 100. Therefore, if it is assumed that in a DSP microprocessor it took on the order of 10,000 clock cycles to calculate just one DFT output sample, then it can be seen that the FFT still takes about 1000 clock cycles (10,000/100) to compute just one FFT output. Again, note that the FFT cannot compute 1 output in the equivalent manner to the DFT; it must compute N output samples on a per stage basis. The explanation is offered from the viewpoint of one DFT sample (rather than the whole DFT result) in order to make an equivalent comparison between the DFT and the FFT hardware implementations. This method of comparison will be evident in a moment.

The discussion of the FFT algorithm thus far has been simplified a bit by viewing the input signal as a real input sequence and ignoring the fact that the twiddle factors are complex, simply calling it a "complex multiply". It is true that a received analog signal is generally a real signal, unless some previous processing has been executed. However, by the time the input sequence passes through the RF front-end and reaches the DSP processing, it is often a complex signal. There will be more discussion on this idea in chapters 4.4 and 6. The complex input signal of Figure 4-2 was given in the polar form of $CC[n]e^{-j2\pi n}$ in order to simplify the initial FFT algorithm explanation. However, recall that multiplication of signals in hardware essentially breaks down to shift and add, as has already been thoroughly discussed in chapter 2.4. Since it is easier to perform addition on complex values when they are in Cartesian form rather than Polar form, each complex

input in Figure 4-2 is typically represented in hardware using Cartesian format. Therefore, if the above discussion of FFT computational complexity is brought down to the hardware implementation level, there are actually four physical multiplies for each "complex multiply" discussed above. This doubles the previously discussed amount of clock cycles needed to calculate just one DFT or FFT sample in any DSP architecture that must share a MAC unit, as in Figure 4-2. In addition, extra logic must be generated to keep track of the real and imaginary parts, since with both complex coefficients and complex twiddle factors, multiplication by two imaginary terms ($j^2 = -1$) can occur, resulting in a real value. Given this relatively small amount of additional complexity, it becomes clear that DSP architectures with only one MAC unit suffer severe limitations in efficiency, especially as the number of samples, N, increases.

The solution to the limitations of sequential pipelined architectures that must share a MAC unit is massive parallel processing. Going back to the 512-sample DFT example once more, assume a real input sequence and once again lump the complex multiplies into one MAC operation for simplicity. As was described above for the DFT (Equation 1), it could potentially take on the order of 10,000 clock cycles for a DFT to compute just one sample, which requires executing N MAC operations. However, if there were enough hardware resources available, such as Look Up Tables (LUT's) for combination hardware, registers, adders, and especially multipliers, such that all N MAC operations and all N-1 addition operations could execute in parallel, then one sample could be calculated in 1 clock cycle! Figure 4-5 [2] illustrates this concept. With this type

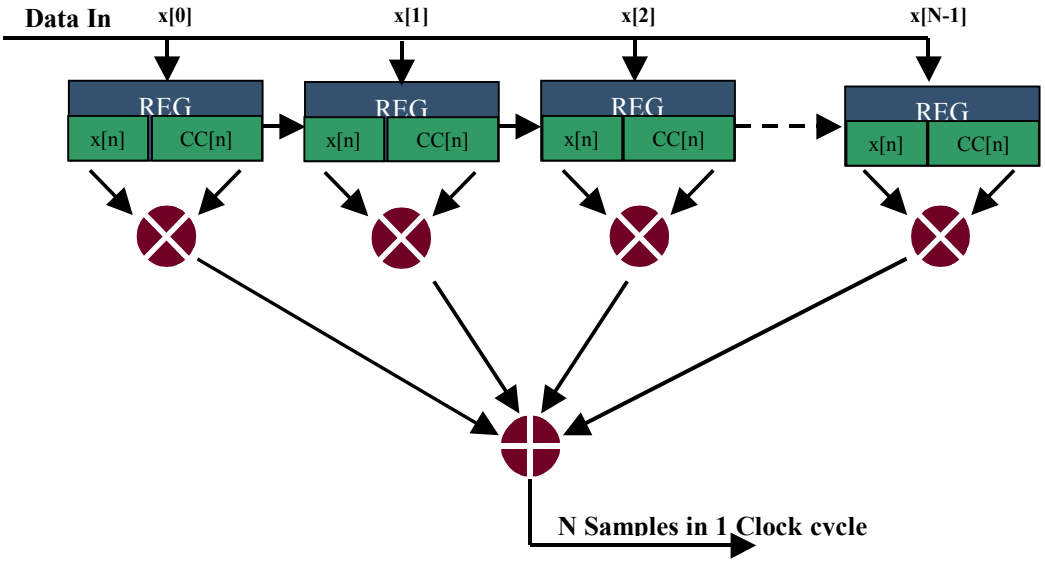


Figure 4-5: Parallel Multiplier

of MAC utilization, it is possible to calculate one DFT sample every clock cycle. Compare this with the 10,000 clock cycles in the typical DSP processor example. Likewise, and more importantly, the concept extends to the implementation of the FFT. In this case, as explained above, the equivalent analysis for the FFT respective to the DFT is to compare the calculation of the results for one FFT stage. Since there are N/2 complex multiples and N complex adds per stage, then N/2 parallel multipliers and N accumulate (add and register) units would be necessary to calculate the results for one stage in one clock cycle. Again compare this to the 1000's of clock cycles needed for FFT implementation on a DSP microprocessor.

The concept of increasing the efficiency of the MAC intensive DSP algorithms with the use of massive parallel may seem elementary at first. However, it needs to be put into perspective. Typical processors have a bank of 32 or 64 working registers that are each 32 or 64 bits wide. Additionally, their memory units (cache) are on the order of mega-bytes. Compare this to the amount of resources needed to implement Figure 4-5. Assume that an input sequence has been digitally sampled and quantized with a 16-bit analog to digital converter (A/D). Therefore, each input sample (again assume real inputs and lump the twiddle factor multiplication into one "complex multiply) must be 16 bits. Using the DFT example and N = 512, this implies that 512, 16 bit registers, multipliers, and adders must be available in parallel. For many years, this type of hardware resource requirement was not available in an FPGA, crippled by both area and speed limitations. However, in the past few years, FPGA vendors such as Xilinx have leaped technology hurdles to make such intensive resources in re-programmable logic devices available to DSP engineers. However, the data must be loaded to the input registers, and stored from the output registers (not shown in Figure 4-5)) to memory. If this load/store process is to be executed in one clock cycle, then the required data bus bandwidth into memory would be 512x16 = 3070 bits. However, this value only takes into account a real input sequence and a lumped "complex multiply" The actual memory bandwidth needed is 2x3070 = 6140 bits wide, which is not feasible with today's memory. So once again, a bottleneck caused by lack of concurrent advancements to standard memory interfaces causes a bottleneck in a true "fully" parallel implementation. However, Xilinx has further developed their architecture to allow extremely fast access

to memory to help reduce this bottleneck. Together with mass parallelism, the FPGA's offer significant speed up advantages for real time processing.

The first move away from DSP microprocessors in order to take advantage of hardware MAC parallelism was a move into systems on a chip (SOC's). Up until recently, SOC's have been implemented in very high dense and high-speed application specific integrated circuits (ASIC's). An SOC is an embedded solution that contains a central processor hard core (ex. IBM 405 PPC), surrounded by soft-core algorithm accelerators, memory, soft-core peripherals, and I/O interfaces, all attached to a central bus. These chips are expensive, designed for large production quantities, and are one-time-programmable. An example of a current large market for this type of application is the second-generation cellular phones. This architecture is actually ideal, since hardware and software tradeoff analysis can be performed to determine which portions of a DSP algorithm run most efficiently compiled as a program on the embedded processor, and which portions of the code can be extracted to be accelerated by a hardware core attached to the processor. Typically, any portion of an algorithm that is MAC intensive runs more efficiently in hardware, while any portion that is loop intensive on control and decision making logic executes more efficiently on the processor. However, this is not always the case, and this concept will be revisited later on in chapter 7.

Despite the continued advancements of increased speed, increased logic density, and power reduction in FPGA's, DSP algorithms continue to push the limits of these devices as more a more leading edge designs are realized using the parallelism offered by implementing DSP algorithms in FPGA's. FPGA's, and even ASIC's, have a physical

limit to their mass parallel MAC implementations. Direct implementation of FFT's in today's FPGA's and ASIC's cannot generally exceed 1024 point FFT algorithms. This is due in part to the fact that essentially 2 multiplications paths must be maintained; that is the real and imaginary. One solution for FFT's beyond this point is to again share MAC resources between real and imaginary paths, which requires designing additional logic in the FPGA to buffer the data. Even if parallel MAC's are shared, the speed improvement of an FFT of greater than 1024 samples executed on an FPGA is still grand compared to the speed of the same FFT executed on a DSP microprocessors.

4.2 Motivation for high level, single flow, implementation of DSP algorithms

In the late 1990's, FPGA's began to approach gate counts and speeds of ASIC's, although ASIC's remain superior for both parameters. However, when FPGA's reached a semi-equivalent status, they became more attractive than ASIC's for design and development work, due to their re-programmability. With efficient massive parallel MAC structures, FPGA's quickly became the method of choice for most DSP algorithm implementations. However, the design flows for DSP algorithms and digital logic design via FPGA's did not develop in unison. The Xilinx FPGA design flow has already been discussed in chapter 3, and is illustrate by Figure 3-1. The DSP design flow is much shorter. One of the most world-renowned software tools for DSP algorithm design is Matlab, produced by Mathworks. Many companies researching and developing DSP, controls, and communication algorithms currently realize all of their high level modeling in Matlab, including all of the necessary test harnesses. In some cases, especially for real

time applications, these designers would utilize a subprogram of Matlab, called Simulink. Simulink is a block diagram approach (connect boxes together) to system modeling that allows for modeling of streaming data, multi rate systems, and other numerically application intensive areas.

The issue that arose was that there was a gap between the Matlab/Simulink design flow, and FPGA design flow, as is illustrated by Figure 4-6. Designers needed the

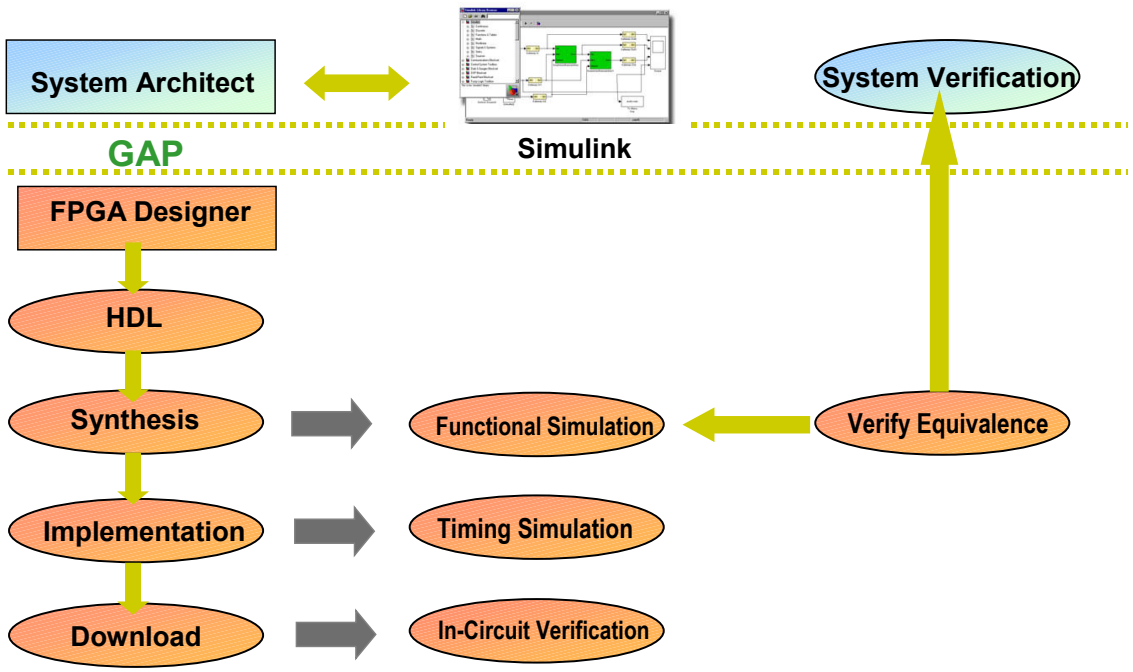


Figure 4-6 : Separation between DSP and FPGA Design Flows

parallel processing advantages offered by FPGA implementation. However, there was no software suite of tools that would allow a DSP algorithm designer to interface directly with an FPGA system architect. Furthermore, there was no streamlined process to verify equivalency of a hardware implemented DSP algorithm against the original test harnesses used in Matlab and Simulink.

To solve this separation in design flow issue, a new electrical engineering work force skill began to be requested by companies around the world who specialized in both DSP algorithm development and FPGA design. This skill was the ability to execute brute force translation of Matlab and C++ code into Verilog or VHDL. This knowledge required not only a broad experience in software languages, but also an experience with converting software constructs into parallel hardware constructs. Therefore, the current DSP algorithm to FPGA design flow, illustrated in Figure 4-7, has been pieced together

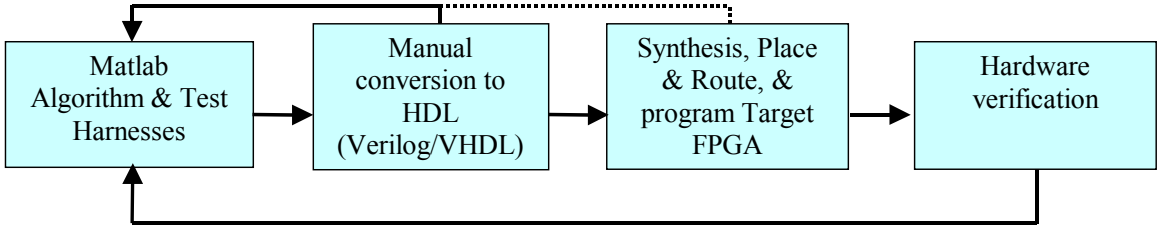


Figure 4-7: DSP algorithm to FPGA design flow

from two entirely differing design flows. First, the Matlab/Simulink design flows are carried out as normal. Then the algorithm models are typically broken down into smaller, easier to handle software functions that are more hardware friendly. From there, each function is converted to a hardware description language, such as Verilog or VHDL, either manually or by using some algorithm specific conversion script. Once the RTL entry has been completed, the normal FPGA design flow of Figure 3-1 is followed. The original test harnesses used in the verification of the Matlab algorithm are converted into HDL test bench format, and used to stimulate the logical design. The results from the simulation are generally saved into some type of matrices format and dumped to a file to be parsed by Matlab and compared against the original test results. Alternatively, the

HDL simulation results can be converted and compared against known results saved from Matlab test harnesses. Once functional equivalency at the functional verification stage is complete, the normal design flow is followed to synthesize and place and route the algorithm into a target FPGA of choice.

After the FPGA has been programmed, the real test harnesses used in the verification of the Matlab algorithm can be fed directly into the input of actual hardware system. This implies that a special interface must be developed to transmit data from the PC to a development card housing the FPGA, and then back again to the PC. In some cases, the actual intended input, such as an RF signal, is used to stimulate the hardware, however, the resulting data must still be transmitted back to Matlab for final verification of the results. The design, debug, and maintenance of this data acquisition test interface, typically controlled through a program such as Labview, is generally a very large effort.

The underlying theme demonstrated by Figure 4-6, Figure 4-7, and the above explanation, is that today's DSP algorithm designers and hardware implementation engineers have a need to begin and end their design flow in Matlab. However, from Figure 4-6, it should be clear that despite the brute force method of converting software languages to hardware languages, the current DSP algorithm to FPGA implementation flow must cross many boundaries. For example, the design flow illustrated in Figure 4-7 must convert from Matlab to HDL, HDL to the FPGA, and from the FPGA back to Matlab, while attempting to maintain consistency of the original algorithm. With so many different boundaries to cross, it becomes very difficult to verify 100% equivalency between the two design flows. There are 3 important concepts represented by the "gap"

in Figure 4-6; they are signed numbers, floating point to fixed-point conversion, and quantization. There is no easy method of dealing with these three concepts in hardware.

Logic implementations of any algorithm are by nature of the hardware, integer applications. Therefore, Matlab/Simulink algorithms that inherently execute 64-bit floating-point operations must be converted first to fixed-point, and then to integer representation. Since every signal is forced into an equivalent integer representation, both the range of the original integer and the precision of the decimal cannot be maintained at the same time. The situation becomes worse for signed (negative) values. The restrictive range leads to unintended quantization (from the DSP algorithm architect's point of view) of the signals. It is essentially this quantization that forces a gap between the 2 design flows and ultimately renders equivalent verification an extremely difficult task. These topics will be discussed in more detail in chapter 6.

What if design-flow, design time, and design resources for implementing DSP algorithms in hardware could be seriously improved? What if there was a tool set that would allow algorithmic design, conversion to HDL, FPGA implementation, and full functional and timing verification all in one design flow? What if the quantization issue could be analyzed on a per module basis, with the ability to model mixed floating point and fixed-point implementation in the same design, thereby exposing the quantization culprit? The Xilinx System Generator for DSP offers exactly this unique, high level, single flow implementation. System Generator, a joint effort between Xilinx and Mathworks, integrates pre-built Xilinx DSP logic cores into a new toolbox for Simulink, which again is part of Matlab. This integration brings Simulink designs to a whole new

level, allowing for algorithm development and logic design to essentially be implemented in the same step. This design can then be used to program the FPGA, and accelerate the

algorithm on the hardware, effectively executing hardware functional testing from Simulink. The results of quantization trade offs can be readily observed real time. The Xilinx System Generator, and its interface to the FPGA hardware, will be covered in detail in chapter 5.

4.3 **Motivation to study phase-based frequency estimation**

At the forefront of DSP algorithm advancements are those companies and researchers that fall into the category of DSP development for electronic warfare applications. One of the key components of DSP algorithms for electronic warfare applications, as well as many commercial applications, is the ability to accurately and efficiently implements frequency detection algorithms that have a resolution to within a fraction of a hertz. There are a few known techniques used to implement frequency detection, and one method is to use DFT processing in conjunction with phase calculations over a short period of time, known as phase-based frequency estimation. This is the algorithm that was chosen as the research vehicle to analyze DSP implementations on FPGA's via the System Generator for DSP design flow discussed above.

In most electronic warfare and other RF applications, the signal of interest is a bandpass signal that has been modulated with a carrier frequency in the 1~100 GHz range. In many cases, the bandwidth to be processed by the DSP front end is on the order

of 100 MHz, and could even be on the order of 1 GHz. As with most communication systems, the first steps in the DSP front end processing is signal detection. An algorithm that characterizes the signals parameters, namely frequency component(s), quite often follows detection. New and improved methods of performing frequency estimation are an ongoing focus of DSP research.

One straightforward method of determining the frequency components of a signal in a small bandwidth is to first compute the magnitude of the Discrete Fourier Transform (DFT), via the FFT. Then, a simple peak search algorithm to determine the frequency index locations of the sinusoidal spectra spikes, thus estimating the frequency, can be applied to the FFT result. In theory, this simple scheme would work if it were possible to calculate an infinite length FFT. However, from chapter 4.1, it is known that the maximum point FFT that can be calculated using an N-parallel MAC's in an FPGA is 1024. Assume that by the time an RF signal reaches an analog to digital converter (A/D), it is a band limited to 25 MHz, centered at 75MHz, and sampled roughly at 4x (102.4 MHz). Assume that the system needs to be able to detect a sinusoidal tone at 67.220450 MHz to within 100 Hz. The frequency range available from this spectral computation is from 0 to Fs, or equivalently from –Fs/2 to Fs/2, which is –51.2 MHz to +51.2 MHz. Assuming a 1024-point FFT (N_{FFT}) will be used to calculate the DFT of the signal, the 1024 DFT samples will be distributed evenly across the entire spectrum of 102.4 MHz. This results in a bin frequency spacing of F_S samples per second / N_{FFT} samples = 102.4 MHz/ 1024 = 100 KHz. Therefore, the DFT only allows a spectral resolution to within 100 KHz, whereas it is desired to detect to within 100 Hz, three powers of 10 too large! The two closest bin frequencies to the received 67.220450 MHz signal would be

67.200000 and 67.300000 MHz. Thus, the sinusoidal spectral peak falls to the low side of a frequency bin, and this sinusoid can only be accurately detected to within a bout 100 KHz, let alone trying to detect the 405 Hz portion of the signal. In this case, a simple peak detect algorithm would locate the input frequency at 67.200000 MHz, which his clearly off by 20,450 Hz. In some cases, it may be acceptable to estimate the frequency, or frequencies, in an input stream to within 100's or 1000's of hertz. However, for most RF and electronic warfare applications, it is desirable to achieve a much more accurate estimate of the frequency to within a few Hz, or maybe sub hertz. Clearly, if it is desired to estimate the frequency of an RF receiver capable of processing such wide bandwidths of data, this FFT alone will not suffice. The limiting factor, and an important DSP concept, comes down the ratio between the sampling rate and the maximum point FFT available by the hardware in the DSP front-end circuitry. If this ratio is large, then the resolution of the resulting spectral analysis will be course, and additional DSP processing must be executed to focus in on the target frequencies.

The above analysis does not make any mention of noise. Transmission channels, analog filters and amplifiers, A/D's, and to some degree digital electronics, all introduce noise into system. In many electronic warfare applications, the power level of the signal is relatively low compared to that of the noise; that is the SNR is inefficient, and the signal must be boosted in order to be processed. However, when the signal is boosted, the noise also gets amplified, causing many small sinusoidal like spikes superimposed on the signal of interest in the resulting DFT spectrum. In the simple frequency detection scheme outlined above, the addition of noise increases the difficulty in accurately detecting the actual correct peak of the sinusoid. For example in an extremely low SNR

scenario, for example tracking a GPS signal in the presence of a jamming signaling, there may be so much noise that the mathematical DFT spectrum contains several spikes within a 100 KHz bin frequency. This could easily force the simple FFT and peak search approach just mentioned above to estimate the frequency to be 67.300000 MHz instead of 67.200000, which is off by 79,550 Hz, a much larger deviation than without the noise. Therefore, the large gap between bin frequencies as a result of the F_S/N_{FFT} ratio renders the DFT and FFT approach to frequency estimation nearly useless on their own.

However, there are many more DSP properties that can be exploited to take advantage of the above DFT scheme to estimate frequency more accurately. In many DSP applications, especially from an analysis point of view, it is customary to represent frequency in radians per sample, rather than Hz, where -Fs/2 and Fs/2 Hz get mapped to $-\pi$ to π radians, respectively. Another parameter of the input signal that is often estimated in the DSP front end of an RF receiver is phase, which is measured in radians. If the phase of an input signal is measured over period of time (or blocks of N samples), and this phase changes per some linear time base (blocks of N samples), then the derivative of the phase with respect to time can be used to calculate frequency; that is, *frequency is equivalent to the time rate of change of phase.*

4.4 Theory of phase-based frequency estimation (PBFE)

Often by the time the input signal passes through the RF front end and reaches the DSP front end, the signal has been converted to a low pass equivalent signal (LPE), which is a complex signal by design. The premise of a complex versus a real signal to

process also makes for an easier explanation of the phase-based frequency estimation algorithm. So, assume that after sampling, a noiseless signal is converted to a LPE

complex sinusoidal signal, represented by

$$x[n] = e^{j(\theta_o n + \phi_o)}, \tag{2}$$

Equation 2

with frequency θ_o (rad/sample) and an initial phase ϕ_o (radians). Notice that this input signal has constant frequency and constant phase. The DFT of Equation 2 might be illustrated by something like Figure 4-8. As will be demonstrated, it must be assumed that the input signal maintains a constant frequency and phase over short period of time in order to effectively implement phase based frequency estimation. In other words, in order

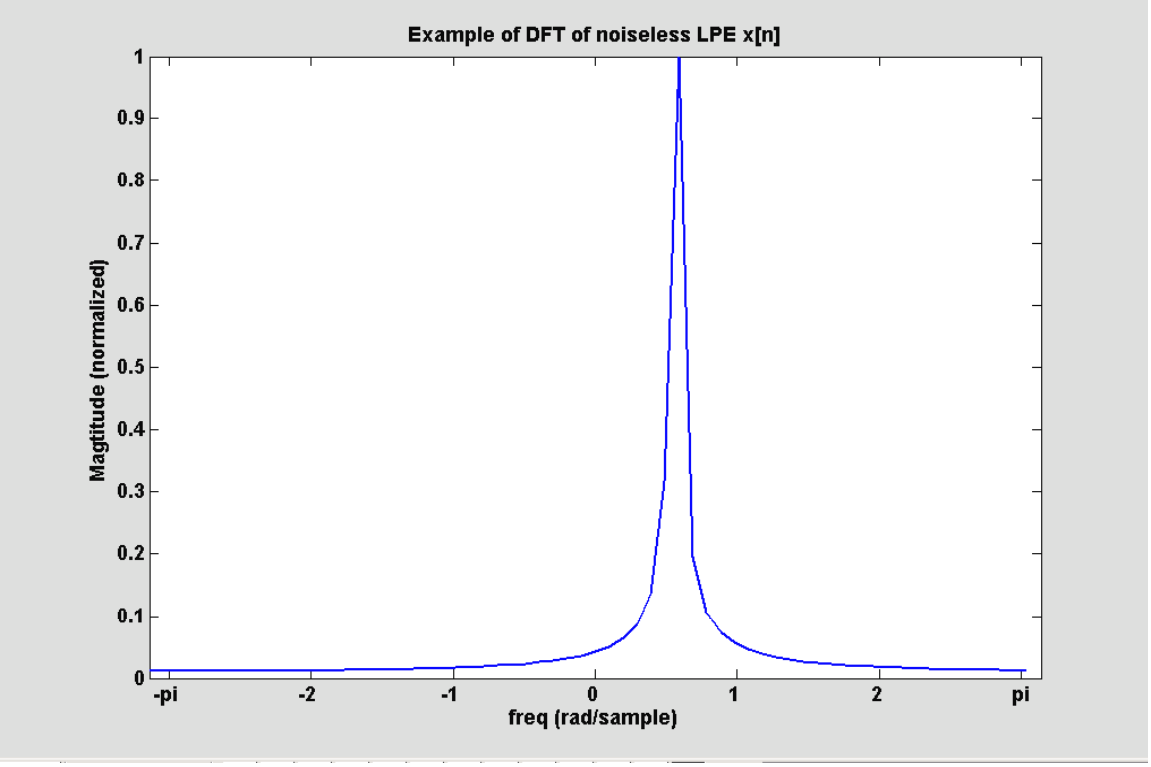


Figure 4-8: DFT noiseless LPE x[n]

to simplify the analysis, this discussion does not take into consideration more complex DSP input streams that may contain frequency and phase modulated signals. However, there is a phase modulation needed in order to calculate frequency, and this phase modulation is a phenomenon forced through DSP manipulation of the signal, as will be discussed shortly.

Assume that an N-point FFT will be used to calculate the DFT of Equation 2, and that enough samples of x[n] have been collected such that multiple, overlapping FFT's can be computed, as shown in Figure 4-9. Also assume for now (this will be revisited

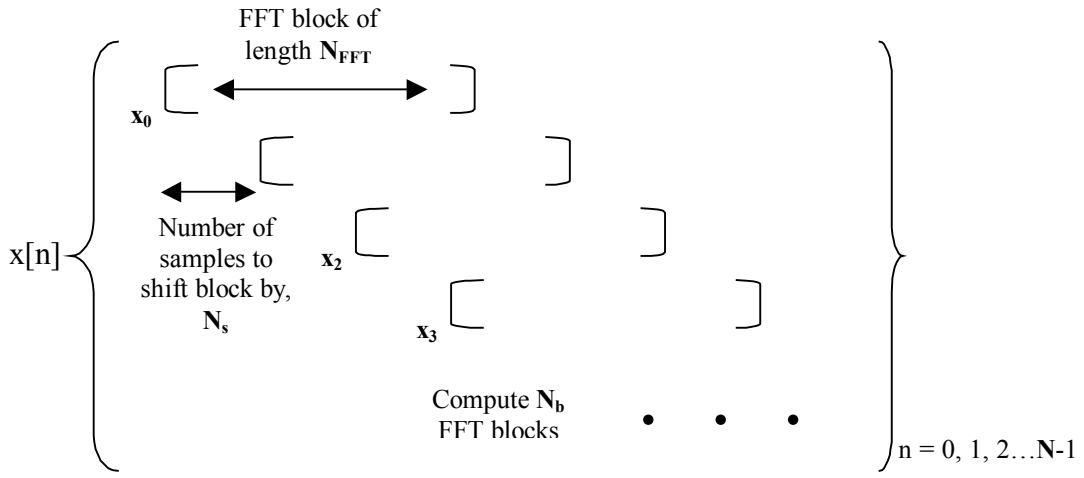


Figure 4-9: Sliding FFT Blocks

later) that the signal has been properly sampled, and that the ratio of the sampling rate to the FFT length (F_S/N_{FFT}) is suitable in order to avoid the resolution issue described in chapter 4.3. *x[n]* in Figure 4-9 is the LPE signal represented by Equation 2 with collected signal length *N*. N_{FFT} represents the number of samples in an FFT block, N_s represents the number of samples to shift each FFT block over by, and N_b represents the number of overlapping FFT blocks to compute. For each N_{FFT} block, the magnitude of the resulting DFT is searched to determine the maximum value, which in a noiseless case will locate the sinusoidal spectral spike to the nearest bin frequency, as in Figure 4-8. The phase angle at the peak is calculated, and then stored for later.

Recall, that the phase of the LPE signal represented by Equation 2 has been defined to be constant, at least over small time intervals. To understand how the technique of computing overlapping FFT blocks illustrated in Figure 4-9 introduces a "forced" change in phase from one block to the next, each block, $x_b[n]$, needs to be

rewritten in terms of the original signal, x[n]. Take FFT block $x_1[n]$ for instance. The data that has been collected begins at sample N_s and ends at sample ($N_s + N_{FFT} - 1$), such that

$$x_1[n] = x[n+N_s] = e^{j[\theta_o(n+N_s)+\phi_o]} = e^{j[\theta_o n+(\theta_o N_s+\phi_o)]}, \quad (3)$$

Equation 3

where $\theta_o N_s$ represents the induced "rate" of phase change from block $x_o[n]$ to $x_1[n]$, and $(\theta_o N_s + \phi_o)$ is the new phase angle, ϕ_1. This phase change is a result of two general DSP properties. First, from a general perspective, whenever you compute the DFT (or FFT) in Equation 1, the algorithm assumes that you are beginning at time zero. By shifting each block by N_s samples, a time shift is introduced, as in evident by Equation 3. The Discrete Fourier Transform time shift theorem is given by Equation 4,

$$x[n-m] \leftrightarrow X[k]\ e^{-j2\pi km/N}, \quad (4)$$

Equation 4

where X[k] in this case represents the N-point DFT result of any block, $x_b[n]$, calculated over the indices n = $[b*N_s , (b*N_s + N_{FFT} - 1)]$. According to the time shift theorem, a shift in the time domain (time is represented here by samples taken at evenly spaced delayed moments in time) is equivalent to multiplication by a phase term in the frequency domain. The N-point FFT computed in Figure 4-9 assumes that each N_{FFT}-point block begins at time zero, and that each result is *exactly* identical to the result that would have been obtained if the DFT of the entire x[n] sequence was calculated using an N-point (versus an N_{FFT}-point) FFT. However, the theorem of Equation 4 implies that each N-point FFT calculated does not *truly* result in the actual DFT since each N_{FFT}-point block

does not start at time zero. The magnitude of each block is not affected; however, there is an additional phase term that must be multiplied (note that multiplying by an exponential does not alter the magnitude, only the phase term) to each computed FFT *sample* in order for each N_{FFT} block output to be *exactly* mathematically equivalent to the N_{FFT}-point DFT. Keep in mind that this additional phase angle is not constant per block; it depends on *k*. The difference between the new phase angle ϕ_1 in Equation 3, and the initial phase ϕ_o, is $\theta_o N_s$. This term is the time rate of change of phase and is equivalent to the additional phase term, 2πkm/N, indicated by Equation 4. In other words, the time rate of change, $\theta_o N_s$, from Equation 3 equals the additional phase term, 2πkm/N, in Equation 4. It is this additional phase that makes the DFT of the time shifted blocks equivalent to the mathematical DFT of the entire original signal. The key concept of this whole analysis is the following. Since this phase term in *not* multiplied to each bock in the sliding FFT procedure of Figure 4-9, a phase difference between blocks is "forced" or "induced". Therefore, a time rate of change of phase, or more precisely a change in phase over a block of N_{FFT}-samples, is created by the sliding FFT block procedure. This phase difference can be utilized for frequency estimation. The F_S/N_{FFT} ratio issue is still present since the first part of the procedure is to execute an small point FFT on a large signal bandwidth. However, by using the standard bin frequency estimation, taking average of several phase calculations, and utilizing additional DSP mathematics, phase-based frequency estimation allows for much finer precision frequency estimation.

In general, then, it can be seen from Equation 3 that the new phase of any computed DFT block will be given by

$$\phi_b = N_s \theta_o b + \phi_o \ . \tag{5}$$

Equation 5

Clearly, this is a linear equation with slope = $N_S\theta_o$, as shown by Figure 4-10, where b = 0, 1, 2, … (N_b = 8) -1, $\phi_o = \pi/2$, and the slope $N_S\theta_o$ is arbitrarily set to 8 radians per block. Since the slope = $N_s\theta_o$ has units of radians per block, and there are N_{FFT}-samples per block,

$$\mathrm{N_s}\theta_o/\mathrm{N_{FFT}} = \mathrm{radians/sample} = \mathrm{frequency} = \theta_o \ . \tag{6}$$

Equation 6

From here, the final desired frequency in Hz is obtained through

$$\mathrm{f_o} = \mathrm{F_s}\theta_o/(2\pi) \ . \tag{7}$$

Equation 7

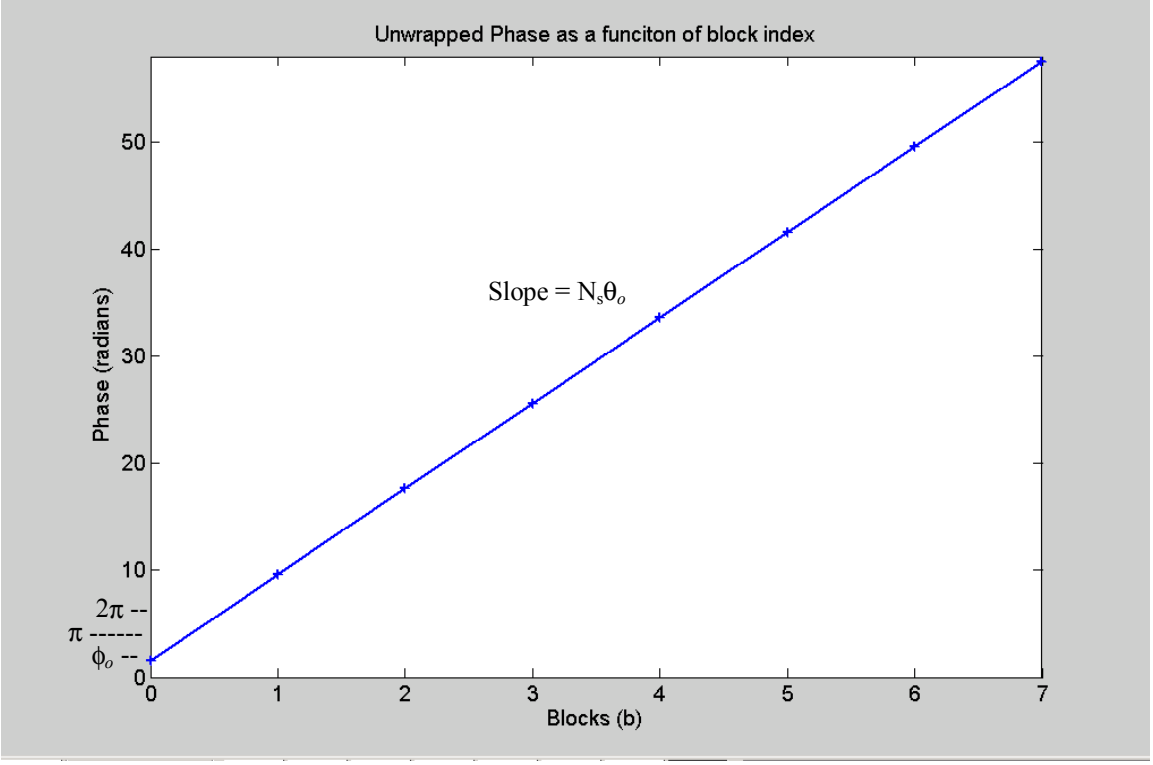


Figure 4-10 : Unwrapped Phase as a function of block index

It should be noted that if the LPE signal given in Figure 4-8 contained a negative frequency, then the slope in Figure 4-10 would be negative. Thus, since the sliding FFT blocks of Figure 4-9 induce a linear change in phase over time, the rate of phase change, or equivalently the derivative of Equation 5, results in the frequency of the sinusoidal input signal. Hence, frequency is the time rate of change of phase.

After some analysis of Figure 4-10, an issue with the phase-based frequency estimation should be apparent. As is alluded to on the 'y' axis, phase is cyclic on 2π, or equivalently $-\pi$ to π. The mathematical computation of the phase at the sinusoidal peak in

each FFT block of Figure 4-9 will always produce a result

$$\phi_b \in [-\pi, \pi) \ . \tag{8}$$

Equation 8

Therefore, Figure 4-10, represents the *desired* phase plot as a function of blocks, rather than the plot of the actual phases computed from the sinusoidal peak in each FFT block.

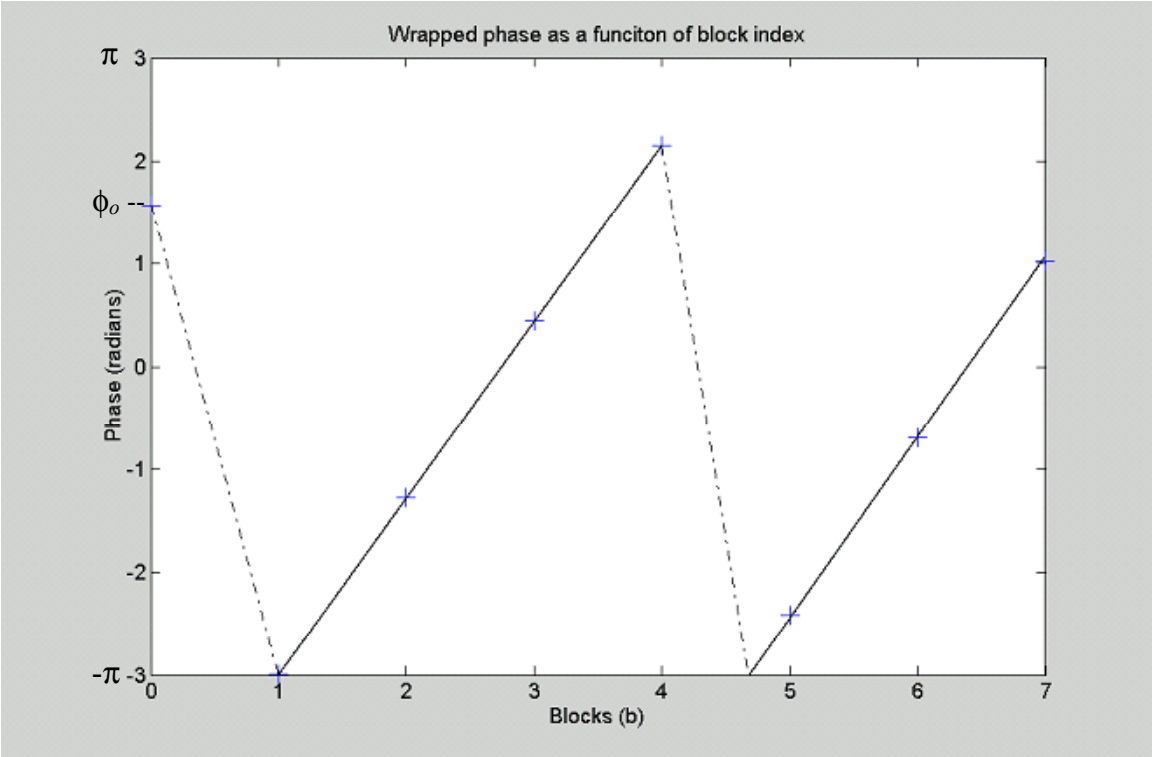


Figure 4-11 : Wrapped phase

The plot of the actual phase angles for each block would look something to the affect of Figure 4-11. Between Figure 4-10 and Figure 4-11 it should be evident that the number of times the phase component of each successive blocks may have wrapped around the

unit circle is not detectable through the normal computation of a phase angle. In order to calculate the slope and frequency, the computed phase angles in Figure 4-11 must be unwrapped to construct the linear plot illustrated by Figure 4-10, which is a DSP phenomenon is known as *phase un-wrapping*. This is another area of DSP that is an ongoing focus of research by many DSP designers who use phase-based algorithms.

There are a handful of methods to deal with this phase wrapping, and a common, simple solution is implemented in this research. However, before continuing with this explanation, a clarification must be made. If one researches phase wrapping/unwrapping, they are likely to find two different, but related, definitions. The first definition is as explained above in Figure 4-10 and Figure 4-11. The second definition deals with the manner in which phase is computed, which is strictly a mathematical issue. For example, given the complex number –a–jb located in the third quadrant of the unit circle, the phase angle may be computed with one of two results, depending on the tool used in the calculation. Theoretically, the phase angle is given by $\tan^{-1}(-b/-a)$. If calculated using arctangent, today's typical graphing calculator, for instance, would produce a phase angle lying in the 1st quadrant, since the negatives cancel out. Likewise, the complex number –a+bj located in the second quadrant would result in a phase angle in the 4th quadrant, since b/-a = -b/a. This can be thought of, in some sense, as "wrapping" the phase angle by $-\pi$ when ever the actual phase angle is not in quadrants one or four, such that the computed angle is referenced from the positive real axis. The arctangent algorithm in Matlab also produces the same results. However, Matlab offers another command explicitly designed to address this issue, automatically *unwrapping* by $+2\pi$, such that the resulting phase angles lie between $-\pi$ and π. Phase calculation algorithms commonly

used in DSP applications, such as the Cordic algorithm, typically employ routines to automatically adjust for the mathematical version of phase wrapping. The Cordic algorithm is implemented in the phase-based frequency estimation algorithm in this research paper, and will be explained in a bit more detail later.

Figure 4-11 has been constructed assuming that a method of calculating phase that

produces angles between $-\pi$ and π has been implemented, versus 0 to 2π. Unwrapping the phase angles computed from each FFT block can be accomplished by first setting a threshold of π. Then the resulting vector of phases is indexed, calculating the difference between each adjacent phase. If the absolute value of the difference exceeds the threshold of π, then an integer multiple of 2π is added to the second phase. For each violation, all of the phases following the violator must be biased by 2π, until the next violation, which will bias the next set of remaining phases again by 2π, and so on. In this manner, the linear solution of Figure 4-10 can be realized.

It may seem from Figure 4-10 that if the absolute phase difference over several blocks (3 or 4) does not jump over the threshold, then one should be able to estimate the slope, and thus the frequency, from just those few phase angles. However, it must be remembered that it has been assume that the LPE signal is noise free to simplify the explanation. In a system with a small SNR (high noise), phase points plotted in Figure 4-11 would not appear to be liner, and a few sample points would not be sufficient to estimate the slope. The phase unwrap algorithm must be applied such that many points are available to construct a least squares fit to estimate the linear function, and thus the frequency.

Using the method of unwrapping the phase explained above imposes some restrictions on the frequency estimation algorithm. Recall that Equation 5 represents the phase, ϕ_b, of the peak of the sinusoid for each block. If the absolute value (takes care of negative frequencies) of the phase difference, $|\phi_b - \phi_{b+1}|$, between any 2 successive blocks

exceeds 2π, there is no way of detecting the *additional* wrapping that has occurred. For example, say $|\phi_b - \phi_{b+1}|$ is actually equal to $2\pi + \Delta$. In this case the difference (ϕ_Δ) will be Δ rather than $2\pi + \Delta$, since the *mathematical phase wrap* issue explained above forces ϕ_b and ϕ_{b+1} to adhere to Equation 8. This indicates that for a given number of samples, N_s, to shift each FFT block by, there are multiple sinusoidal frequencies that will result in the same phase at the peak. In other words, if Equation 5 is substituted into $\phi_\Delta = |\phi_b - \phi_{b+1}|$, the result is

$$\phi_\Delta = N_s \theta_o \ , \tag{9}$$

Equation 9

which by Equation 6 results in the calculation of the estimated frequency. However, there is no way of detecting the difference between $N_S\theta_o$ and $N_S\theta_o + k*2\pi$, where k is an integer. An input signal containing a sinusoid with a frequency greater than 2π radians, then, would not be properly detected, and thus it appears that a limit of $\phi_\Delta < 2\pi$ per block has been established.

However, as it turns out, the actual limit is $\phi_\Delta < \pi$ radians per block. By analyzing the sliding FFT block approach of Figure 4-9 against the theory the DFT-based

filter banks (see [3]), it can be understood why this is so. The details here are a bit beyond the scope of this research. Suffice to say that each frequency bin of the DFT blocks in the sliding FFT block approach can be thought of as a filter bank (FB) channel, and that the shifting of each block by N_s samples is equivalent to a decimation factor of N_s on each FB

channel. From the rules of multi-rate processing, it is known that in order to decimate by N_s, the frequencies in the signal of interest must lie within $-\pi/N_s < \theta < \pi/N_s$. After decimation, the limit becomes $-\pi < N_s\theta < \pi$, or equivalently,

$$| N_s\theta_o | = | \phi_\Delta | < \pi . \tag{10}$$

Equation 10

By substituting Equation 10 into Equation 6 and Equation 7, one arrives at

$$| f_o | < F_s/(2N_s) , \tag{11}$$

Equation 11

or equivalently,

$$N_s < F_s/(2|f_o|) . \tag{12}$$

Equation 12

Equation 11 and Equation 12 indicate that there is a severe limitation to the phase-based frequency estimation algorithm described thus far. This limitation puts a restriction on either the maximum frequency capable of being detected for a given FFT block shift, or the maximum FFT block shift for a desired frequency input range. Figure 4-12 is a 3-D

surface plot illustrating Equation 11, for positive frequencies only, with sampling rates ranging from 0 to 1 GHz, and FFT block shifts from 0 to 32 samples. All axes are shown

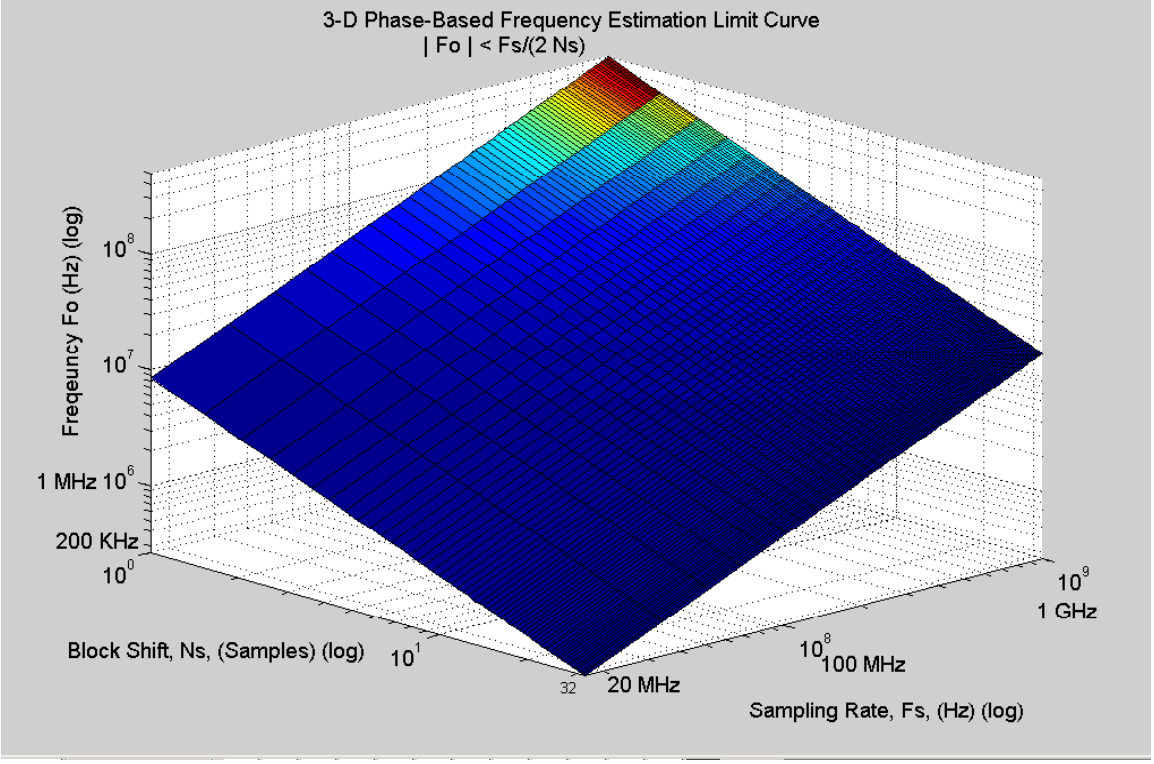


Figure 4-12 : 3 – D Phase Based Frequency Estimation Limit Curve

on a log scale. Notice that the estimated frequency, f_o, increases linearly with the sampling rate, F_s. For the phase- based frequency algorithm implemented in Simulink discussed later on, the sampling rate was fixed at 51200 Hz. Figure 4-13 illustrates the frequency estimation limit curve for this fixed sampling rate. Notice that N_s = 16 samples (choose closest power of 2 → 2^4) is about the maximum amount of shift allowed before the frequency that can be detected becomes too small relative to the sampling rate. This

results in a minimum detectable frequency of 1600 Hz, with F_S = 51200 Hz. Since the estimated frequency tracks linearly with the sampling rate as shown by Figure 4-12, then

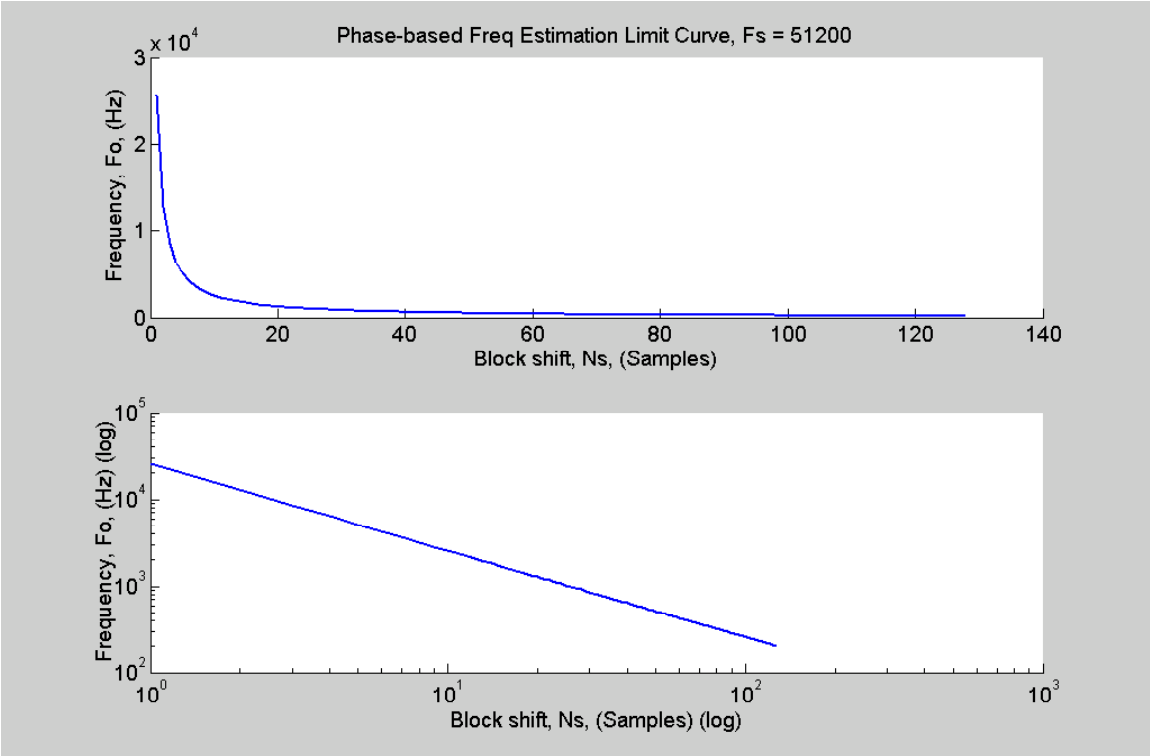


Figure 4-13 : Phase Based Frequency Estimation Limit Curve

this *block shift limit* of N_s = 16 samples roughly holds for all sampling rates, since the frequency bandwidth of interest generally scales with F_s. If the signal is a bandpass signal modulated or frequency shifted up to some higher center frequency as in the RF example of chapter 4.3, then the maximum frequency f_o in this case is corresponds to a maximum deviation from center frequency, within the frequency band of interest.

4.5 Matlab Implementation of Phase-Based Frequency Estimation Algorithm

Although the target software implementation tool was Simulink using the Xilinx System Generator, the design of the phase-based frequency (PBFE) algorithm began in Matlab. At the beginning of the learning curve, little was understood about System Generator, and therefore it was necessary to build up a comparison model using familiar software. The Matlab code was used to become familiar with the basic frequency estimation concepts that were discussed above, such as the issue of phase unwrapping and the frequency estimation limit curve.

The flow chart for the Matlab PBFE algorithm is illustrated in Figure 4-14. The algorithm essentially follows the exact flow of Equation 2 through Equation 12, and

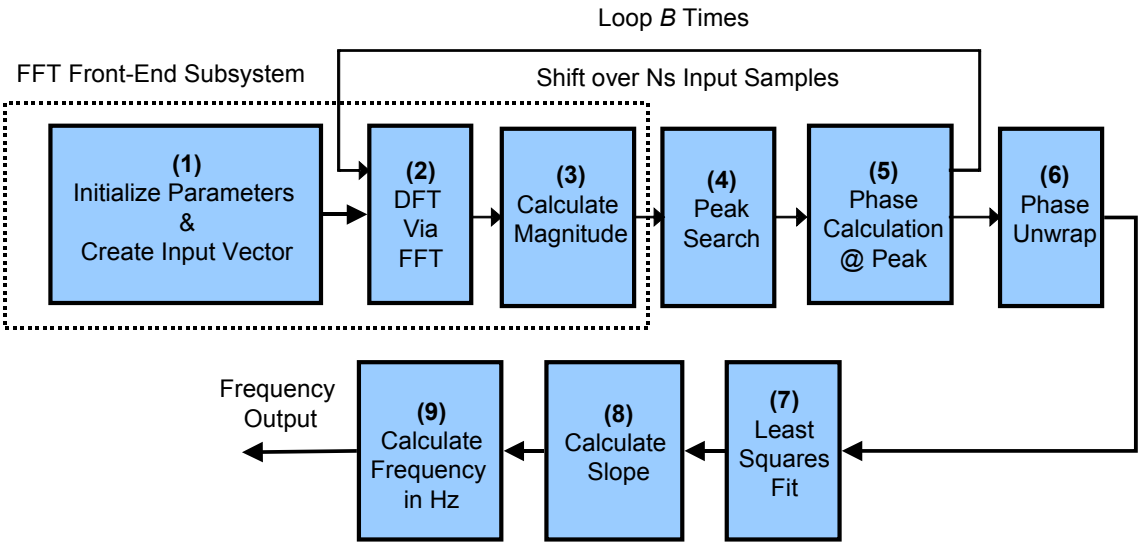


Figure 4-14 : Matlab Flow Chart of PBFE Algorithm

Figure 4-8 and Figure 4-13 outlined in chapter 4.3. First, all of the necessary parameters are initialized. The frequency estimation limit curve of Equation 11 and Equation 12 indicate a restriction on either the maximum sinusoidal frequency, or the maximum shift amount. In order to minimize the amount of data processing over time, it is desired to maximize the amount of samples to shift by. Therefore, N_S was set at 16, thereby restricting the frequency depending on the sampling rate. The FFT length was chosen to be 512, and thus in order to have an even DFT spectral resolution of 100 Hz, a sampling rate of 51200 was chosen. Based on Equation 11, this restricts the input frequency to be between 0 and 1600 Hz, as previously discussed. The last major parameter is the number of FFT blocks, and effectively the number of phase values, to calculate. It was determined that for this noiseless experiment, 10 blocks would be sufficient.

The implementation of most of the steps in Figure 4-14 are fairly straightforward in Matlab, as there are built in commands to handle nearly each step. For example, the FFT, phase unwrap, and least squares fit are all built in commands. However, as will be discussed later on in chapter 6.3, the implementation of such blocks in hardware is not so straightforward.

The Matlab code that implements the flow chart of Figure 4-14 is provided in appendix 10.1. The code was executed for several input frequencies between 0 and 1600 Hz. It was also executed with frequencies greater than 1600 Hz to verify theory described

in chapter 4.3. As expected, the theory was proven to be accurate, and the frequency estimation limit curve is correct. The output of each experiment was simply a 1x1

frequency vector, and thus there are no results to be illustrated for this portion of the design. Again, the Matlab code was constructed as a reference for the System Generator design and servers only as a base model, rather than the target design.

4.6 **Simulink**

The algorithm defined in Figure 4-14 can be fully specified with the version of Simulink that comes with Matlab. The DSP Blockset toolbox is need for such things as the built in FFT. The DSP Blockset toolbox is an optional package that must be ordered when Matlab is purchased. It allows a designer to implement in Simulink any DSP function that can be used in basic Matlab code. Simulink has the ability to run Matlab files from within. Simulink designs can also be stimulated from within a Matlab file. Variables, vectors, and matrices can be passed from Simulink to the Matlab workspace, and vice versa, during simulation. In this manner, the traditional means of creating a "setup parameter" file in Matlab to control user design parameters at a high level can be maintained. A top-level Matlab parameter file can then be used to initiate a Simulink design, passing the necessary parameters from the workspace to Simulink. This also provides a nice means of capturing the results from Simulink back into vector format, and then plotting the data in a traditional Matlab sense in order to monitor either the final output, or intermediate results, such as the FFT outputs.

However, once an algorithm is fully specified and verified in the current version of Simulink, the design stops there. To continue implementing the design in an FPGA, the algorithm still must be converted into HDL by hand. There are pros and cons to using the Simulink approach over entire implementations in Matlab. Until recently, this choice most likely came down to familiarity of each respective tool and the personal preference of a designer. However, now that Xilinx and Mathworks joined forces to develop the Xilinx System Generator toolboxes for Simulink, there are definite advantages to working entirely in Simulink.

5. Simulink & Xilinx System Generator

5.1 Simulink Interface & Xilinx Blockset

As was discussed above in the chapter 3 on the FPGA design flow and software tools, Xilinx has created a tool called Core Generator. Again, Core Generator is a GUI based tool that offers a designer parameterized logic cores that have been optimized (for area & speed) to be implemented on Xilinx FPGA's. This catalog of ready-made cores offers functions that range from simple counters, comparators, adders, and multipliers, to full system-level building blocks, such as memories, FIFO's, filters, and transforms. Mathworks and Xilinx recognized the needs of DSP and other like algorithm designers who were implementing their designs in FPGA's to begin and end their design flow with Matlab, as Figure 4-7 suggests. These two partners also recognized that this design flow could be improved. Xilinx and Mathworks collaborated to create a new set of Simulink toolboxes that allow an engineer to implement both algorithm design and logic design in essentially the same step.

The Simulink interface is a GUI, drop, drag, and connect software tool, similar in nature to another popular data acquisition tool, Labview. The main interface consists of 2 windows, the library browser and workspace, as shown in Figure 5-1. The library window contains the catalog of all of the Simulink tool boxes. The catalog entries that

are circled in Figure 5-1 comprise the new Xilinx toolboxes. As shown in the figure, any Simulink block that has an 'X' represents a Xilinx block, versus a general Simulink

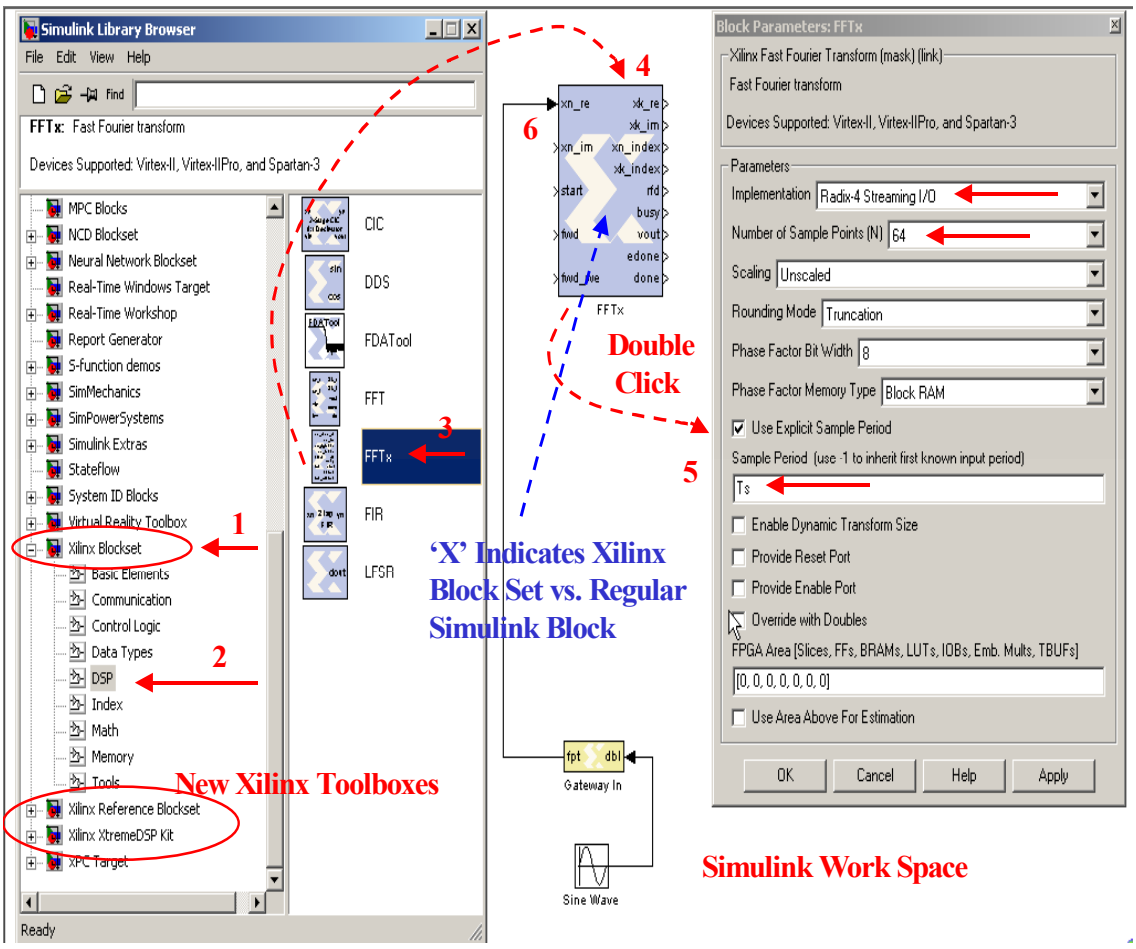


Figure 5-1: Simulink Interface

Block. The significance is that the Xilinx blocks are the only blocks that can eventually be synthesized, placed and routed, and programmed into the FPGA. This will be discussed in more detail shortly. To begin constructing a design, first choose a catalog topic, such as the Xilinx Block set, shown by step 1 in Figure 5-1. Within each catalog topic, there are several sub-topics, for example DSP functions, illustrated by step 2 in the

figure. Once you click on the DSP sub-topic, several Xilinx blocks will be shown in the right hand block window, such as the FFT and FIR blocks, shown by step 3. To begin

using the FFTxx block in a new design for example, you simply click and drag the FFTxx block from the library browser to a blank Simulink workspace file, illustrated by step 4. Step 5 is to configure the FFT block, which is done by double clicking on the new block in the workspace window. This action produces a pop-up box, the block parameter window, with all the user configurable block parameter options for the FFT. For example, the 2 important options for the FFT would be the implementation field, which chooses between a radix-2 and radix–4 algorithms, and the number of samples field, which specifies the number FFT samples, such as 64. The most critical parameter to take care of on each and every block in the Xilinx set is the sample period. This field has units of seconds, and will be discussed at several points throughout the paper. Finally, step 6 is to pull in additional building blocks, whether they are other Xilinx System Generator blocks or other regular Simulink blocks, and wire the FFT block up appropriately. In this case, Figure 5-1 illustrates a sinusoid generator from the regular Simulink library, connected through a floating point to fixed-point gateway, and then finally wired the real input on the Xilinx FFT core.

Recall that the Xilinx blocks are essentially a Simulink version of all the cores in the Xilinx Core Generator. A convenient feature of the block parameter window is the help button, which first links to the Simulink help files for each of the blocks. Once there, the help file for any system-level building blocks have a link to the Core Generator data sheet. Just as if you were purchasing an AND gate from a manufacturer and using

the data sheet to wire up your breadboard, this data sheet is extremely important for two reasons. First, it provides detailed information on the functionality of all I/O's to the core. Secondly, and probably most important, is that the data sheets have full functional timing information, as well as any timing information relative to each FPGA target device (ex. XC2VP7). Since System Generator integrates high level DSP algorithm design with FPGA implementation, it is now necessary to understand the timing of each Xilinx block in order to assure proper algorithm results, at both the Simulation level, and at the Hardware Co-simulation level.

5.2 **Software Simulation Mode & FFT Front-End Design Example**

Designing with Simulink is broken down into two simulation modes; software simulation mode, and HDL or Hardware Co-Simulation mode. In order to explain HDL and Hardware co-simulation, it is easiest if a Simulink design example is demonstrated for the Software simulation mode first. Figure 5-2 illustrates a complete DSP Simulink design that can be targeted for an FGPA. This system is the FFT front-end sub-system for the phase-based frequency estimation algorithm. Simulink designs can be broken down into levels of hierarchy, just as an HDL design can. These levels of hierarchies are called sub-systems, or sheets. To help understand the functionality of this subsystem, refer to the phase-based frequency estimation flow chart illustrated in Figure 4-14. In step two of the algorithm, an N_{FFT}-point FFT of b overlapping blocks of a digital input sequence is computed, where each overlapped block is shifted by some amount of samples, N_s. Again, Figure 4-9 illustrates a visual representation of this process. Recall

that N_s must be less than or equal to 16, as per the DSP theory discussion surrounding Figure 4-12 and

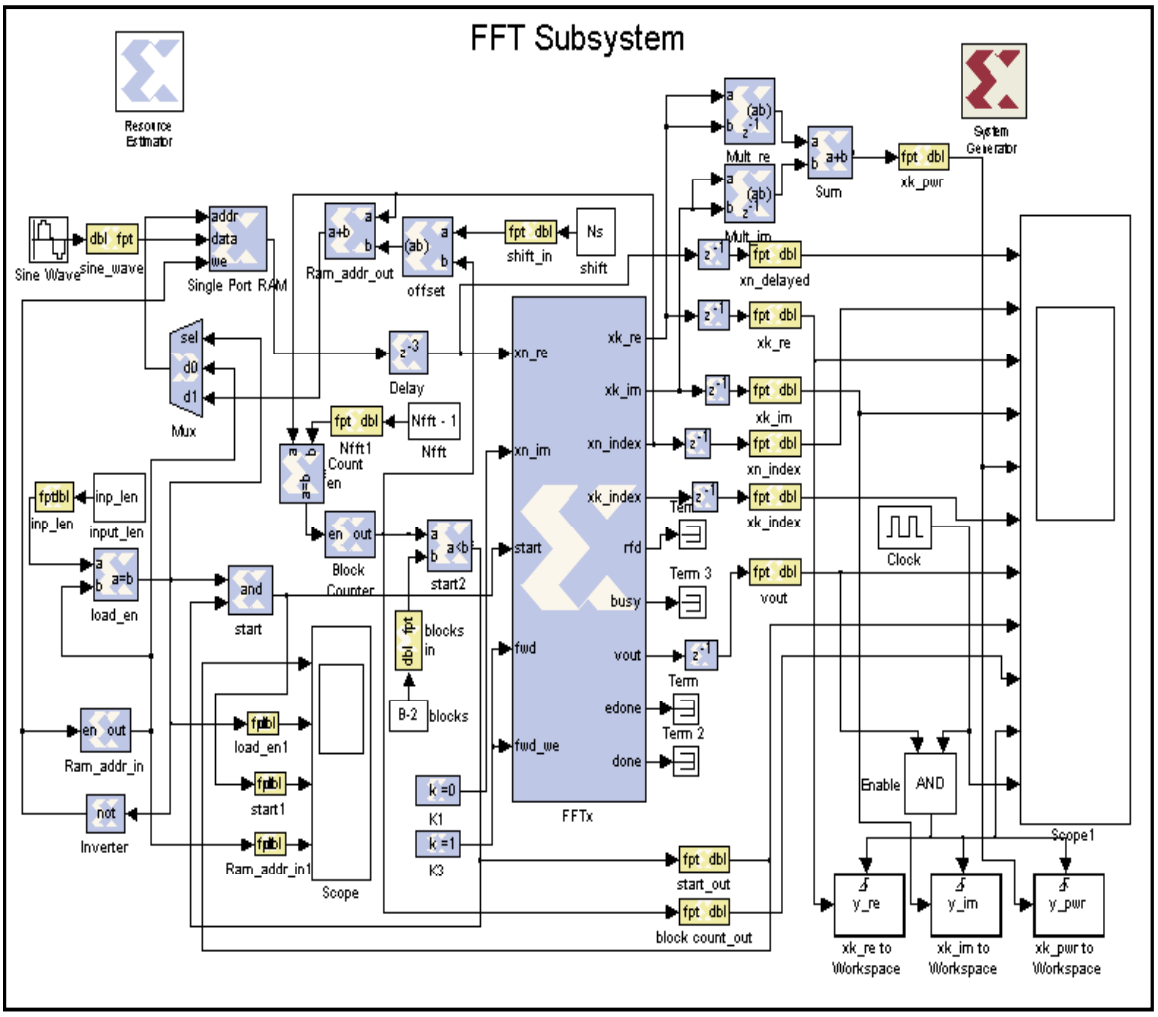


Figure 5-2 : FFT Front-End Design Example

Figure 4-13 in chapter 4.4 This task is collectively called the FFT front-end subsystem for the frequency estimation algorithm in this research. The input to the FFT subsystem is a real, single frequency sinusoid. The sinusoidal generator uses the frequency, Fo1

obtained from a Matlab script file, to load a RAM with an a sinusoidal input sequence of length "inp_len", as shown in Figure 5-3 (explained shortly). Constants, fed from the Matlab workspace, control two address counters. The first increments as the RAM fills up. Once this is complete, a mux selects the other address counter, which is controlled by the index output of the FFT core. This counter creates addresses to the RAM such that the sliding window input sequence of Figure 4-9 is fed into the FFT core. The main outputs of this subsystem are a $b*N_{FFT}$ length real vector, containing the real portions of each FFT calculation stored back-to-back, and a $b*N_{FFT}$ length imaginary vector, containing the imaginary portions of each FFT calculation stored back-to-back. More detail is provided on this subsystem in chapter 6.3.1.

First, note the types of Xilinx blocks used in the FFT front-end subsystem. There are basic logic design elements, such as a counter, comparators, an adder, multipliers, and a register (unit delay). Additionally, note the two important blocks near the top of Figure 5-2, the System Generator block and the ModelSim block. *The System Generator block is critical*; it must be present in any Simulink design in order to use any Xilinx block. Additionally, this block is the unit that configures the system for Hardware co-simulation, while the ModelSim block configures the system for HDL co-simulation. These two modes will be discussed in chapter 5.3. A third type of Xilinx block in this system is the FFT and ROM blocks, which as described above, are some of the full system level building elements provided by the Xilinx Core Generator software tool. The last type of Xilinx block in this design example is the gateway, which converts from double precision floating-point format to fixed-point format, or vice versa. These blocks are important, as they are the blocks that allow the interface between any Xilinx

hardware synthesizable unit and any regular Simulink block. Think of the gateways as the list of all inputs and outputs (I/O's) to the FPGA. As was mentioned back in chapter 4.2, Matlab and Simulink operate in either a 32 or 64 bit double precision format, while operations in actual hardware (i.e. an FPGA) must be executed in an integer representation of fixed-point. Therefore, the gateway blocks effectively act like programmable quantizers. The remaining blocks, such as the constants, clock, AND gate, and triggers to Matlab workspace, are all original Simulink blocks. They are used in this design to create a trigger, much like a trigger on a logic analyzer, to capture the results back to Matlab so that they can be plotted and visually verified. The scope is probably the most critical block in debugging a Simulink design. This block will be discussed in detail shortly.

Once the blocks are connected together and the Simulink system timing configurations are properly set, the design can be debugged and simulated. This can occur in one of two ways. The most direct way is by simply clicking the run button in the Simulink main workspace window. If there are errors in the design, such as incomplete wire connections, data type mismatches, or data width mismatches, an error screen will appear allowing the designer to link directly to the block causing the error. The other simulation option is to initiate a Simulink design by invoking a function call from within a Matlab script file, which is the procedure used to design and test the FFT front-end subsystem of the frequency estimation algorithm. The script file used in this case begins by initializing the user-defined parameters already discussed in step one of the frequency estimation algorithm of Figure 4-14. Once initialized from a script file, the variables are available in the Matlab workspace until otherwise cleared. Once they are available in the

workspace, the variables are also available to be accessed from a Simulink design, as is illustrate in the FFT front-end subsystem simulation flow of Figure 5-3.

For example, referring back to the block parameters of Figure 5-1 (right window), some of the parameters can be set by using variable names that exist in the Matlab

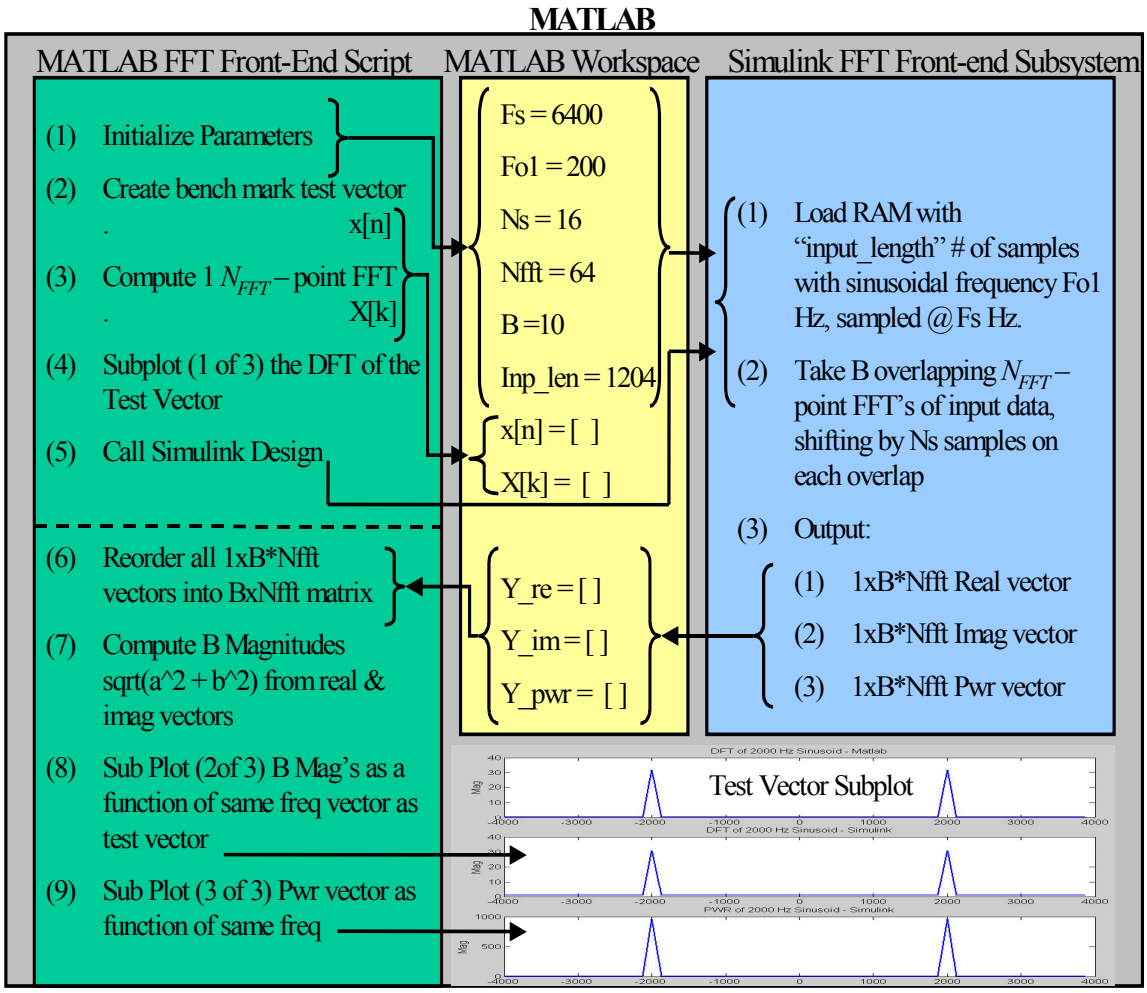


Figure 5-3 : FFT Front-End Simulation Flow

workspace. For example, the Simulink constant blocks (i.e. the ones without an 'x') 'N_{FFT}', 'shift', and 'blocks' in Figure 5-2 can be initialized from variables on the Matlab

workspace, rather than manually updating them each time a change such as sampling frequency is needed. As mentioned earlier, the most critical block parameter is the sample period, $T_s = 1/Fs$. Simulink actually runs off a system clock that is used to derive all other necessary clocks. Simulink typically automatically derives any implied clocks from the system clocks, which is generally set to the global sample rate. Systems that have more than one sampling rate (multi-rate systems) or any feed back loops are a bit more complicated. The sample period parameter of each block can be set to a fixed value, it can be inferred from a previous block, or it can be set to a variable that exists on the Matlab workspace. The initial blocks at the beginning or left side of a design (think of them as simulation time = 0 sec) must be either set to a fixed sample period or to a Matlab workspace variable. Some examples in Figure 5-2 are the Xilinx constant blocks, the counter, and the gateways in. In the FFT front-end subsystem, there is one global sampling rate, and this is achieved by setting the sample period parameter on the 'initial' blocks to $1/F_S$, which is initialized by a Matlab script in Figure 5-3. . Each Xilinx block that has inputs leading from the output data paths of these initial (time zero) blocks can derive or infer its sampling period automatically from the previous block by placing a –1 in the sample period field. An example is the FFT block. In this manner, the system clock (sampling rate) for an entire design can be reconfigured by changing just one variable, F_S, in a Matlab script. Alternatively, the Simulink system clock rate can be controlled from the Simulink simulation parameters, on a global basis, as well as on a sheet by sheet (or subsystem) basis. There is more discussion on sampling periods in chapter 6.2.5.

For the original design and debug of the FFT front-end sub-system, the user-defined parameters from step one of the Matlab algorithm of Figure 4-14 were set to F_{o1} = 200, N_s = 16, N_b=10, and N_{FFT} = 64, as is illustrated in Figure 5-3. The values of $F_{o1}$was chosen somewhat arbitrarily. N_s was initialized to 16 in accordance with the results of Figure 4-13, and therefore F_S was calculated from Equation 11, which equals 6400. For the purpose of the frequency estimation algorithm research in this paper, calculating the phase over 10 blocks is sufficient, thus N_b=10. N_{FFT} was chosen to be 64 samples rather than the target of 512 in order to reduce the run time of the Simulink simulation. Although dynamic FFT lengths are possible with the FFT core, this feature was not implemented, and thus the FFT length is fixed. To set up a test case to compare the results of the Simulink design against, the FFT front-end Matlab script in Figure 5-3 executes the first three steps of the frequency estimation algorithm of Figure 4-14. The results of the N_{FFT}-point DFT blocks were plotted on a Matlab subplot to compare to the output of the Simulink design, which is shown in the top portion of Figure 5-5 and will be discussed after the Simulink design is analyzed.

Once the Matlab script, available in appendix 10.2, finishes executing the test vector plot, the FFT front-end Simulink subsystem is invoked, utilizing the user defined parameters already existing on the Matlab workspace to initialize all of the necessary block parameters previously discussed. The data flow begins at time zero with the sinusoid generator loading up the RAM with the input vector. The RAM is loaded with sinusoid of frequency F_{o1}, obtained from the Matlab workspace, of length 'input_lengnth' by 16 bits wide. Again calling on Figure 4-9, it can be seen that the maximum number input samples needed for the constraints of the parameters is $N_s*b + N_{FFT}$ = 16*10 + 64 =

224 samples. Moving to the next power of 2 gives 256, and thus a minimum of 256 memory addresses are needed for the RAM. However, the actual target for phase based frequency estimation algorithm is a 512-point FFT, and thus $N_s*b + N_{FFT} = 16*10 + 64 = 672$. The next power of 2 is 1024. Therefore the RAM was initialized with a holding capacity of 1024x16 for growth up to

the intended design specifications. The binary point for the FFT inputs must be set to N-1 = 15 bits, thus allowing FFT inputs only in the range of –1 +1. The key difference between the Matlab code and the Simulink code is that the System Generator FFT block is a streaming FFT function. This means that it expects a continuous stream of data, contrary to the Matlab code, which takes a block of data, calculates a 64-point FFT, and then stores the result into a row of a 10x64 matrix. In the Simulink design, the RAM must continuously stream out data according to a sliding address window. Since the RAM is word (16 bits) addressable, an indexing scheme was constructed using a counter and two comparators to shift the starting point of the memory address over the proper amount of samples (addresses) in order to capture the correct N_{FFT} block of input samples, as demonstrated by Figure 4-9. Remember, a 'sample' here is in reference to a 16-bit value, and is located in a single memory address.

As mentioned, the scope block is the key to debugging a Simulink design while in Simulation mode. Figure 5-4 illustrates the scope debug results for the FFT front-end subsystem of Figure 5-2. The scope is essentially a graphical version of a logic design simulator. It graphs outputs as a function of the system clock (sampling rate), however, it has the capability to show values along a 'y' axis as well, rather than in just binary form like ModelSim (see chapter 3). The key inputs and outputs to the FFT core are namely

the input, *x[n]* and its corresponding index, *n*, the real and imaginary output components, *x_r[k]* and *x_i[k]* respectively, and their respective index, *k*, as well as the *start* and *valid out (vout)* signals. These key signals are illustrated in the screen shot of the scope of Figure 5-4 for a given simulation run with the user defined block parameters

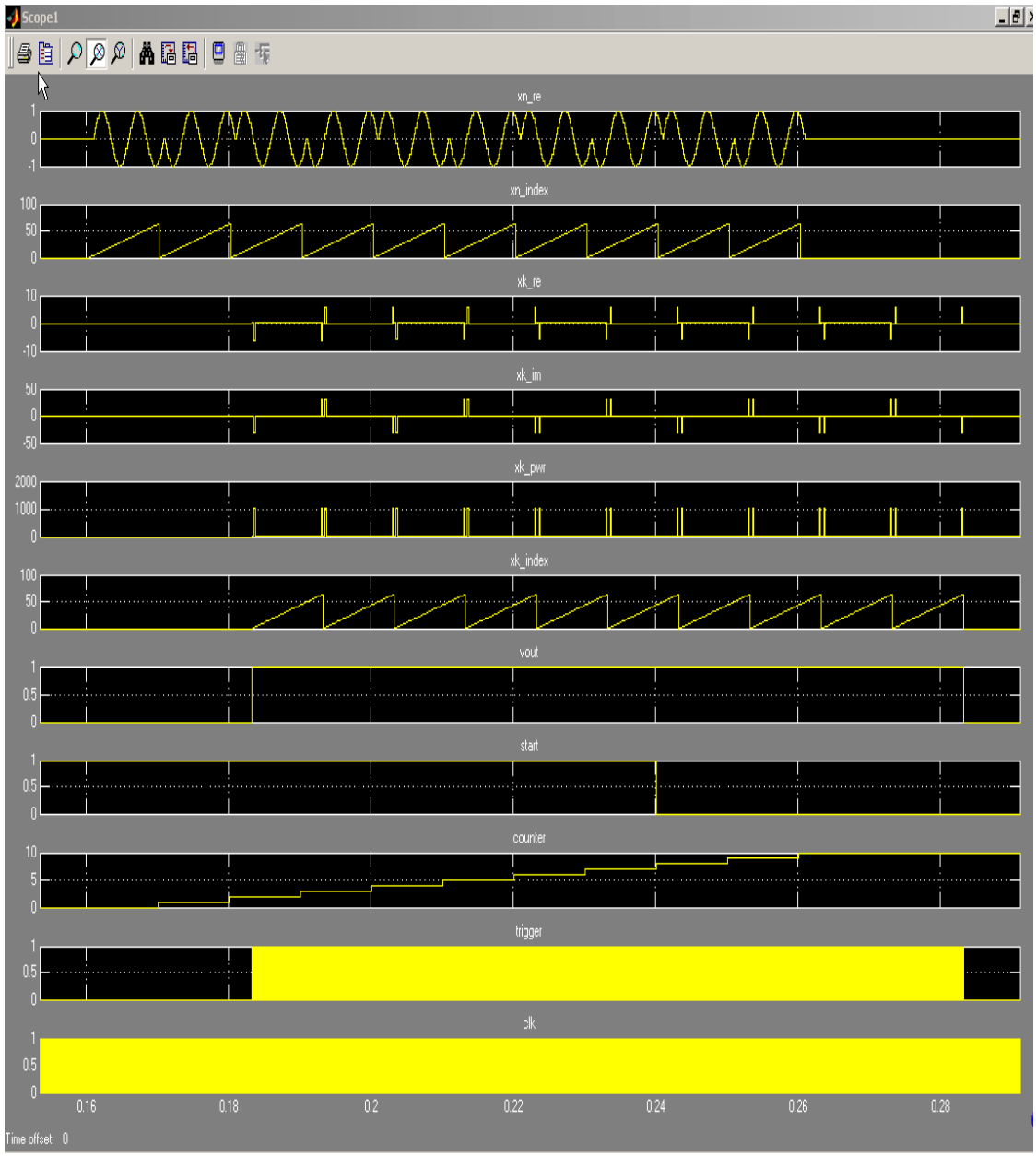


Figure 5-4 : Scope: FFT Front-End Subsystem Debug

configured as shown in Figure 5-3. The FFT core begins to process streaming data when the start signal is activated. Once the start signal is active, the FFT core automatically generates an input index counter for a designer to utilize. This counter is four clock cycles ahead of when the data actually needs to arrive at the FFT input to allow for any delay in retrieving the data from memory or registers. The input counter was used to create the sliding address window to properly index into the RAM. The valid out (vout) signal indicates when the first real and imaginary pair is available at the FFT core output, and is in sync with the *k* index output. There is a delay through the FFT core, depending on the size of the core (i.e 64 samples versus 512). For $N_{FFT} = 64$, the delay is about 0.003 seconds (simulation time), which equates to about 2.25 inputs blocks; in other words, the first output sample of the first FFT block is seen after 2.25 input blocks have been applied to the FFT inputs. This observation is useful, since in the function call to invoke the Simulink file in step five of Figure 5-3 the simulation run time must also be specified. Once the initial delay is known, the total simulation time can be calculated. The FFT core will continue to process back to back FFT frames (i.e. streaming data) as long as the 'start' signal is active, which can be removed when the block (frame) counter reaches $b = 10$ in this case. Notice that the expected spikes in the DFT of a real, single frequency sinusoid are displayed as a function of time in the scope, rather than frequency. Also notice that these spikes are generally graphically viewed on a frequency index ranging from $-\pi$ to π, as in Figure 4-8. However, the output samples of an FFT algorithm are actually calculated on the range of 0 to 2π, and thus the normally negative spikes are

spectrally shifted by 2π radians from a typical plot. Keeping this in mind during debug allows for a good indication of correct FFT results from a time-based simulation.

The valid out signal is also used in the FFT front-end subsystem as the trigger for the "To workspace" Simulink blocks. The valid out signal is ANDed with the system clock to create a trigger, as depicted in Figure 5-2, to send the results of the b = 10 FFT calculations back to the Matlab workspace. The trigger is illustrated graphically in the bottom of Figure 5-4. The real and imaginary results, as well as a pseudo power result, are sent to the Matlab works space in variables called y_re, y_im, and y_pwr, respectively. To be sure that the timing between the real and imaginary output of the FFT are synchronized when returning to the Matlab workspace, a pseudo power calculation was implemented by squaring the real and imaginary parts using multipliers, and then adding them together. The word pseudo power is used here since the squaring and summing procedure constitutes only part of a Power Spectral Density function, and therefore does not truly represent power.

The original Matlab script file that kicked off the whole simulation has access to the output variables as soon as the function call to Simulink returns control, as shown in Figure 5-3. The Simulink simulation for the FFT subsystem takes about 0.016 simulation seconds to calculate all b= 10, 64-point FFT blocks. Once control is returned, the Matlab script calculates the magnitude of the signal from all 10 FFT frames from the real and imaginary components. As mentioned in the explanation of the DSP theory, this analysis does not include noise. Thus all 10 FFT blocks are expected to give the same results. The results for the all three subplots from the above discussion of the FFT front-end subsystem and corresponding Matlab test vector are illustrated in Figure 5-5. The top

plot is the result of the Matlab code that created the test vector. The middle graph shows the output of the B= 10 FFT block from the Simulink design. The bottom plot illustrates the pseudo power. Cleary, all spikes are at a frequency of 200 Hz, and the FFT front-end

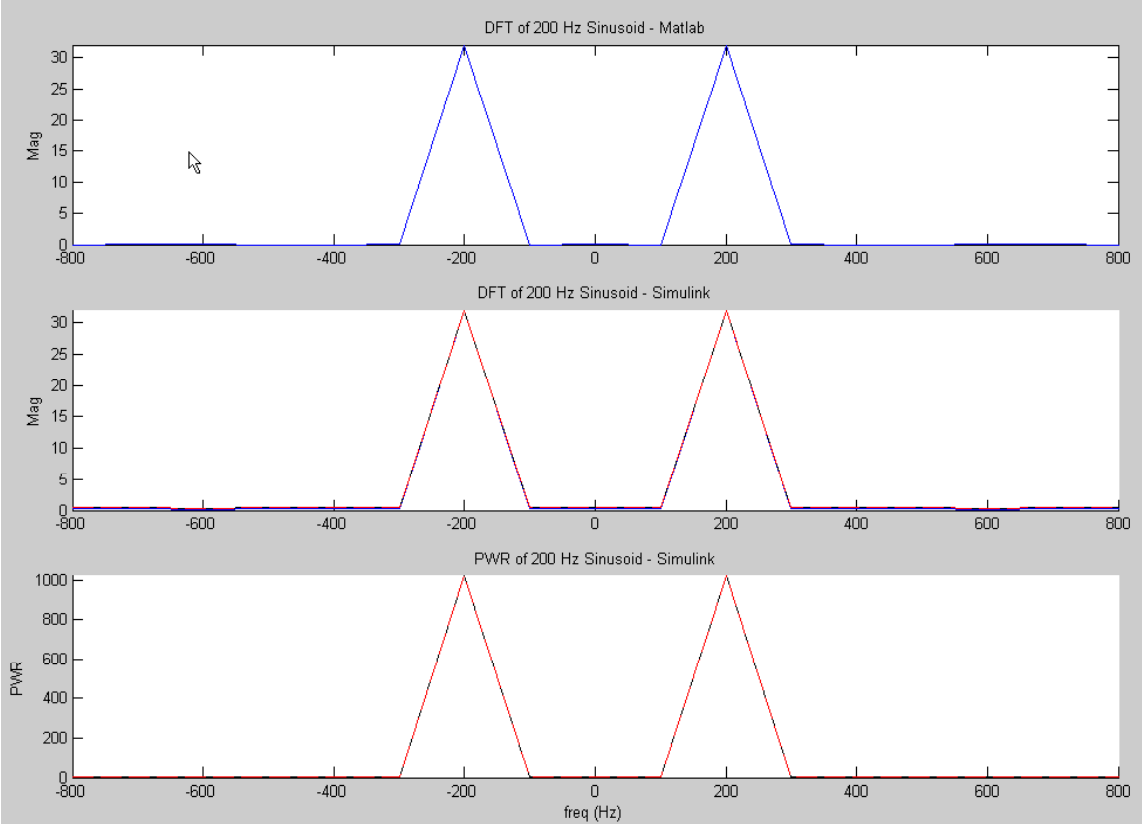


Figure 5-5 : Simulation Results: Comparison of Simulink to Matlab

subsystem to the frequency estimation algorithm Simulink design works exactly as expected.

Given that the input is a single sinusoid, this comparison and verification process may seem rather futile at first. However, 3 main issues had to be solved before forward

progress could be carried out on the rest of the phase-based frequency estimation algorithm. The first is that the Simulink FFTxx core expects a constant stream of data, while the Matlab FFT function is deigned to work on a block of data, which works well in the sliding FFT blocks procedure of Figure 4-9. In order to make the Simulink FFTxx core work in this finite block manner (versus streaming), much additional control logic was required to create a sliding window address counter, as demonstrated in the design example of Figure 5-2. Secondly, since the Xilinx gateway blocks and the FFT core perform double precision floating-point to fixed-point conversion and vice versa, and the issues of quantization and rounding had to be analyzed. Again, these parameters are modified through the block parameters GUI discussed above in chapter 5.1. Finally, and most critically, the timing issues had to be solved. Assuring proper setup of the sampling rate for each Xilinx block is critical. Further, passing streaming data from the FFT core back to the Matlab workspace was also a timing challenge. Since this subsystem is the key to the whole phase-based frequency estimation algorithm, it was critical to be sure that this design was producing precisely the expected results across a wide range of frequencies and sampling rates. Once the FFT subsystem was verified for the simpler and smaller user-defined parameters discussed above resulting in Figure 5-5, it was then verified at the original frequency estimation algorithm target parameters. Again, these parameters are $F_s = 51200$, $F_{o1} = 1600$, $N_s = 16$, $N_b=10$, and $N_{FFT} = 512$, as explained in the theory of the frequency estimation limitation curve of Figure 4-13. Since the System Generator is a relatively new tool and design flow, just the FFT front-end subsystem task proved to be challenging and essentially consumed most of the allotted time for

implementation. Thus, the full phase-based frequency estimation algorithm was never fully designed, simulated, or verified.

5.3 HDL & Hardware Co-Simulation Mode

Once a Simulink design is simulated to a designer's satisfaction in software mode, one or both of the Co-simulation flows can be begin. Figure 5-6 gives an overview

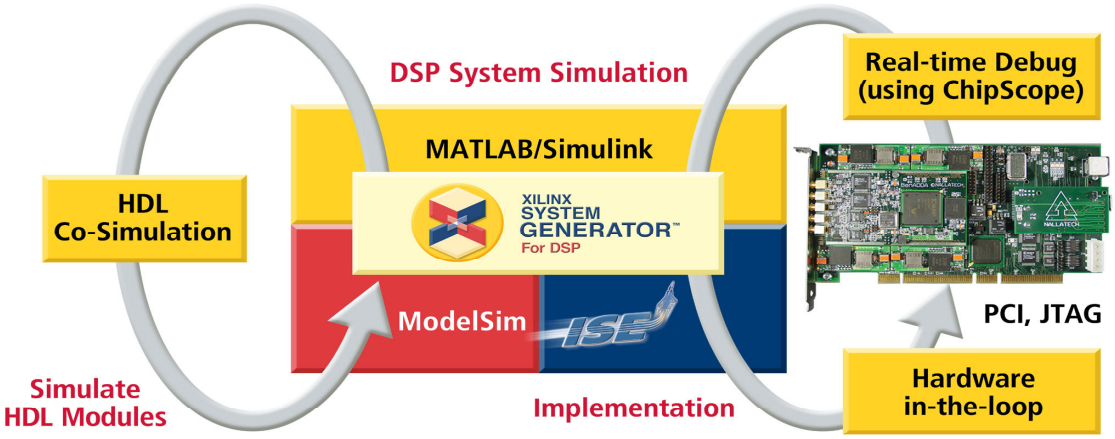


Figure 5-6 : DSP System Simulation

of Co-simulation. The left loop is known as HDL Co-Simulation, while the right loop is known as Hardware Co-simulation. These concepts are 2 completely different flows, with different objectives. HDL Co-Simulation is implemented using yet another software tool, adding a 3rd layer of software to System Generator. In the software mode, System Generator is collectively composed of Matlab and Simulink constructs. HDL simulation mode adds ModelSim, an industry standard, functional and timing verification simulation

tool for logic design, which was discussed in chapter 3. The objective of this flow is to allow HDL designs (RTL) to be included into a Simulink design as a black box in order to increase the functional capabilities of Simulink. A VHDL or Verilog file can be parsed,

creating a Matlab file that defines its parameters. Then a black box can be integrated into the Simulink design, using the Matlab configuration file to link the black box to the source HDL file. The ModelSim co-simulation block allows for the auto generation of a test bench for the HDL black box. At simulation time, ModelSim can be invoked, and the test bench and HDL black box will be simulated on the waveform viewer, using the stimulus provided by Simulink inputs to the black box. The outputs of the HDL black box are also simulated, and passed back into the Simulink design.

Hardware Co-simulation flow adds a fourth layer of software to the pile, as well as actual hardware, such as development circuit cards and FPGA's. The Hardware Co-Simulation flow first utilizes the Xilinx ISE and Xilinx Core Generator (see chapter 3) to synthesize, place and route, and generate a single FPGA programming bit file for all of the System Generator Xilinx blocks in a Simulink design. This results in a new System Generator block, called the JTAG Co-Sim block. The new block is used to replace all of the original Xilinx Simulation blocks (i.e. all the blocks targeted for hardware) in the design. Once this is complete, one end of a JTAG cable is connected to the parallel port of the PC, and the other end is connected to the JTAG port on a development card housing an appropriately chosen Xilinx FPGA. Now, when the Simulink design is executed, the new JTAG Co-Sim block causes the bit file to be loaded into the FPGA through the JTAG cable, as in normal programming mode (see chapter 3). The difference

here from the normal design flow is that after the FPGA is programmed (about 2 seconds), the entire Simulink design that was originally simulated using Matlab software is now run in real time on the actual FPGA hardware! Data is sent to and from the

FPGA, serially, through the JTAG cable. Despite the serial interface, this process is extremely fast. The FPGA hardware has the ability to utilize the parallelism concepts discussed in the DSP chapter, thus allowing for massive speed up of large designs.

The two simulation flows of Figure 5-6 are controlled via the 2 Co-simulation blocks shown in Figure 5-7. Doubling clicking on each block respectively provides the co-simulation GUI parameters illustrated in Figure 5-7. The HDL Co- simulation block

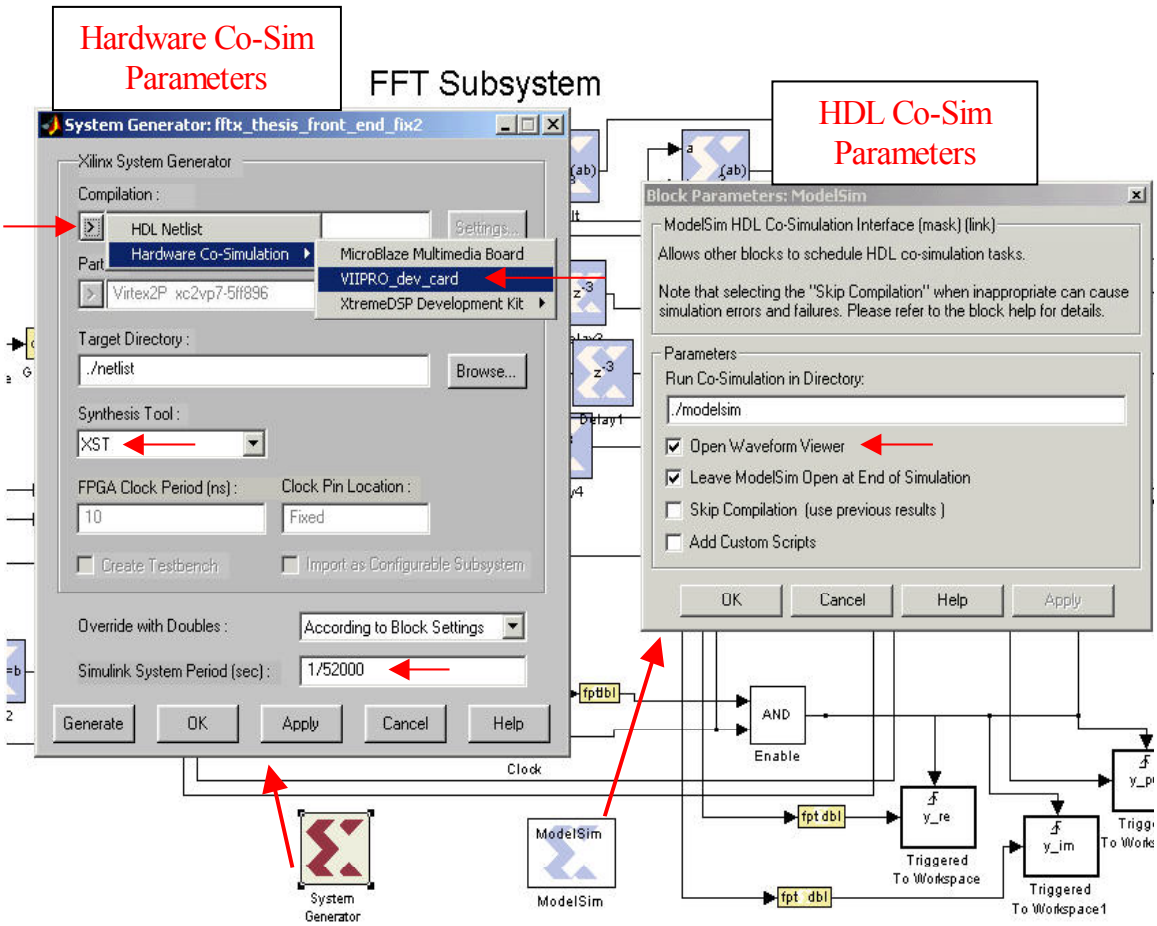


Figure 5-7 : Co-Sim Block

gives such options as the ModelSim directory to store the creation of the test bench and other simulation files, as well as the option to open the ModelSim waveform viewer. The ModelSim Co-Simulation block was not utilized in the implementation of the algorithm for the design in this paper, and thus any further discussion is out of scope for this research.

The Hardware Co-simulation block, however, was utilized in the implementation of the FFT front-end subsystem. To do so requires the setup up of four System Generator files. These are *the* files that glue together the Simulink, Xilinx Core Generator, and Xilinx ISE software packages into one, top-level, single design flow. The files define such parameters as the target FPGA device, the FPGA system clock speed and clock pin, and any other FPGA constrains that may be necessary. The Xilinx System Generator for DSP software package provides a template of these files, as they are unique to each design and target device. Once the System Generator files are set up to glue together all of the necessary designs, software, and hardware, then the Xilinx System Generator block can be opened, as illustrated in the left GUI of Figure 5-7. In the Simulation mode, this block is simply necessary to allow the System Generator Xilinx blocks to interface properly with the regular Simulink blocks. To move into hardware co-simulation mode, the JTAG-Co-simulation block mentioned earlier must be created via the System Generator block. First the target device must be chosen from the compilation option (top Figure 5-7.). Then the synthesis tool, either Synplicity or XST (see chapter 3), is

selected. Finally, and most importantly, the Simulink System period (i.e. the software simulation system clock used back in simulation mode) must be set properly, and in this case controlled by the Fs parameter in the Matlab script file. Note that this is *not* the same clock speed at which the hardware will be running. Once again, the hardware clock speed is controlled in the System Generator setup files, and is defined by the clocks available on the target circuit card (normally these are picked to meet DSP system constraints, such as A/D sampling).

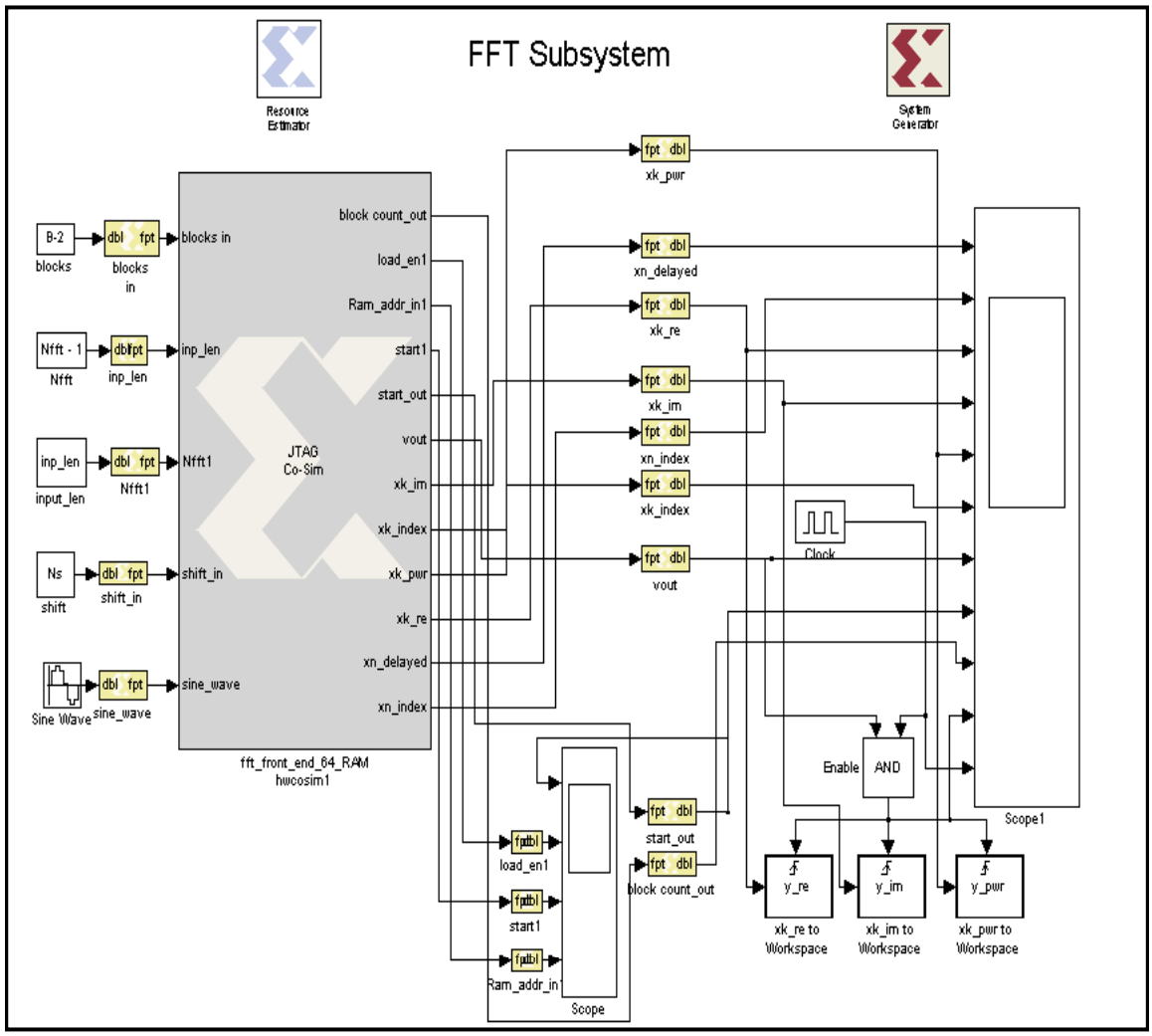


Figure 5-8 : HW Co-Simulation Design Example

Once the Hardware Co-simulation is setup, the generate button in the lower left corner of Figure 5-7 kicks off the synthesis, place and route, and bit generation of the System Generator blocks targeted for hardware implementation. Behind the scenes, Pearl scripts link the Simulink design to Xilinx ISE and the Xilinx Core Generator. The process time is dependent on the size of the Simulink design to be targeted, and is also highly dependent upon the speed of the PC on which the software is executing.. Once the bit file has been created, a "black box", called the JTAG Co-Sim block, appears in a new window. This block represents every Xilinx block illustrated in Figure 5-2 that has been implemented in Hardware. The original Xilinx blocks of Figure 5-2 must be removed, and the JTAG Co-Sim block must be put in their place. Then this new system must be saved as new file. Figure 5-8 depicts the hardware Co-Simulation version of the FFT front-end sub-system after the substitution of the JTAG Co-Sim block has been made. Note how each Xilinx gateway block becomes an input or an output to the JTAG block. Also note the regular Simulink blocks that are left behind and do not change.

Now the design is ready for Hardware Co-Simulation mode. Up until this point, hardware was not required; it was an entirely a software flow. Figure 5-9 illustrates the setup that is needed before JTAG Co-simulation of Figure 5-8 can be executed. The parallel IV Cable, available through Xilinx, must be connected up the parallel port on a PC and a JTAG port on a circuit card. The parallel IV JTAG cable gets power from the PS2 port on the PC. The power supply gets plugged into a standard AC 120V wall socket. There is a surface mount LED next the FPGA, and this will become solid once the power up sequence has completed. At this time, the hardware is ready to run Co-

simulation. The same Matlab script file of Figure 5-3 that was executed to initialize the user defined

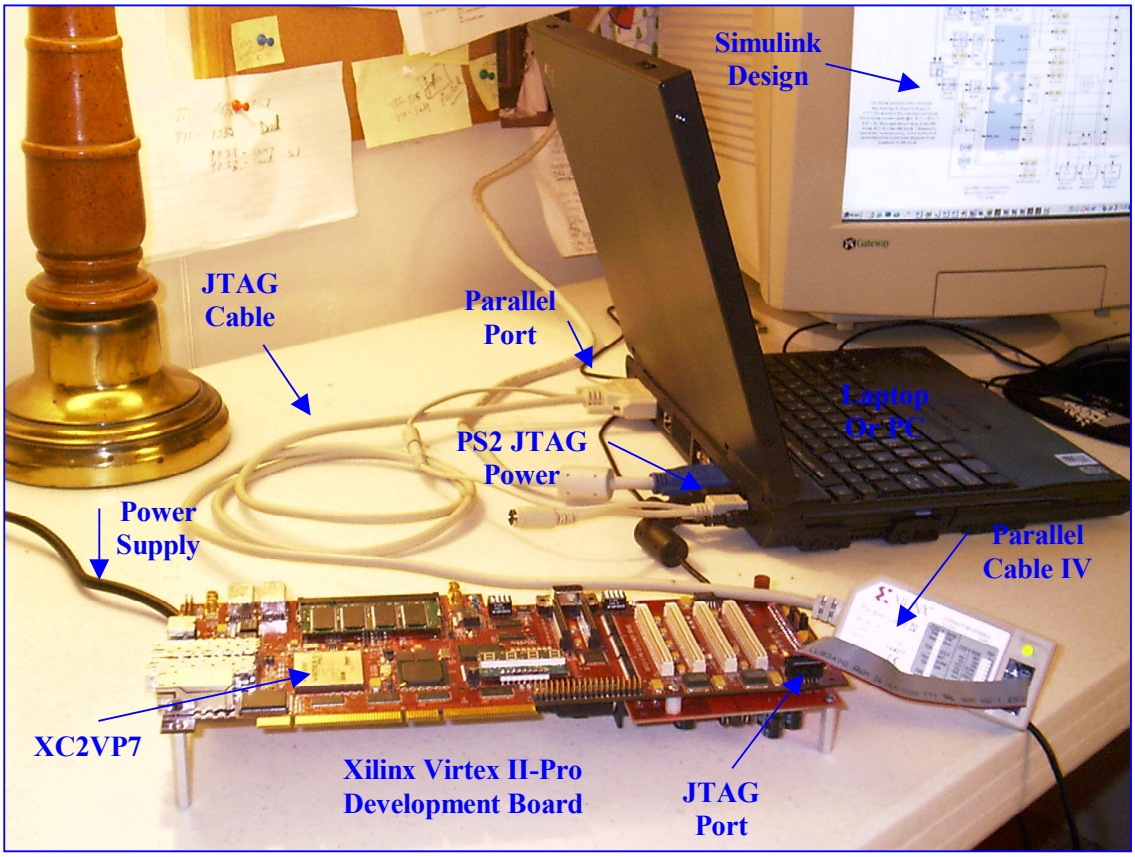


Figure 5-9 : Hardware Co-Simulation Setup

parameters and invoke the software mode of the Simulink design can now be used to invoke the JTAG Co-Sim design of Figure 5-8. The JTAG Co-Sim block must linked to the FPGA bit file. Again, this was produced when the generate button in System Generator block was initiated, which executed synthesis, place and route, and bit file generation, just as the traditional FPGA flow of Figure 3-1 illustrates.

If the design was implemented correctly, the results of the Scope and the Matlab subplots should be nearly identical to Figure 5-4 and Figure 5-5. Once the results from software simulation mode are considered acceptable, and the System Generator setup files are properly configured, the results of the Hardware Co-simulation flow almost always match the software simulation flow results exactly. This was true in the case of the FFT front-end subsystem, and therefore there is no need to replicate Figure 5-4 and Figure 5-5. The concept that that the Software simulation mode results almost always exactly equals the hardware co-simulation results is what makes this tool so unique. A Matlab DSP algorithm designer/ FPGA designer can work out the high level DSP functional problems, while at the same time debug nearly all of the low level hardware issues, such as quantization, before the design is ever implemented on the hardware.

6. Analysis of Frequency Estimation Implementation in Hardware

6.1 Hardware Development Board

In order to research the implementation of DSP algorithms on FPGA's, a development board was needed. The software tools that have been discussed thus far cost thousands of dollars. A decent development card ranges from $500 to $2000. Fortunately, Xilinx has a program called the XUP, or Xilinx University Program, which offers short-term licenses to Universities for research purposes. Given the breadth and depth of the research proposed, much of the software was donated. At this time, the XUP needs to be acknowledged for supplying the Xilinx ISE and System Generator for no charge. Synplicity, who makes the Synplify synthesis tool, also has a university program and offered their tool at an extreme discount. Xilinx and their vendors, however, do not offer discounts on their hardware. Therefore, the Electrical and Computer Engineering (ECE) department at Binghamton University was approached about the research presented in this paper, and asked to put forth the money for a development board. At this time, the ECE Department of Binghamton University needs to be acknowledged. The research could not have been completed without the support of all of the aforementioned parties.

The choice of the development board, at the time, was based primarily on one need: the embedded 405 PPC core (see chapter 3). Initially, the thesis research was

intended to follow more closely the path of implementing a DSP algorithm in an SOC environment on an FPGA. However, as many times happens, thesis objectives become altered during the research stages, as was the case here. The development system lacked an easy method to transmit and receive data for design purposes without serious overhead in time to design an interface (ex. PCI and Ethernet cores). The Xilinx System Generator for DSP was originally investigated as a means of communication between Matlab (where the DSP design started) and the hardware. As should be evident from the previous chapters, mastering the System Generator was not a short and straightforward

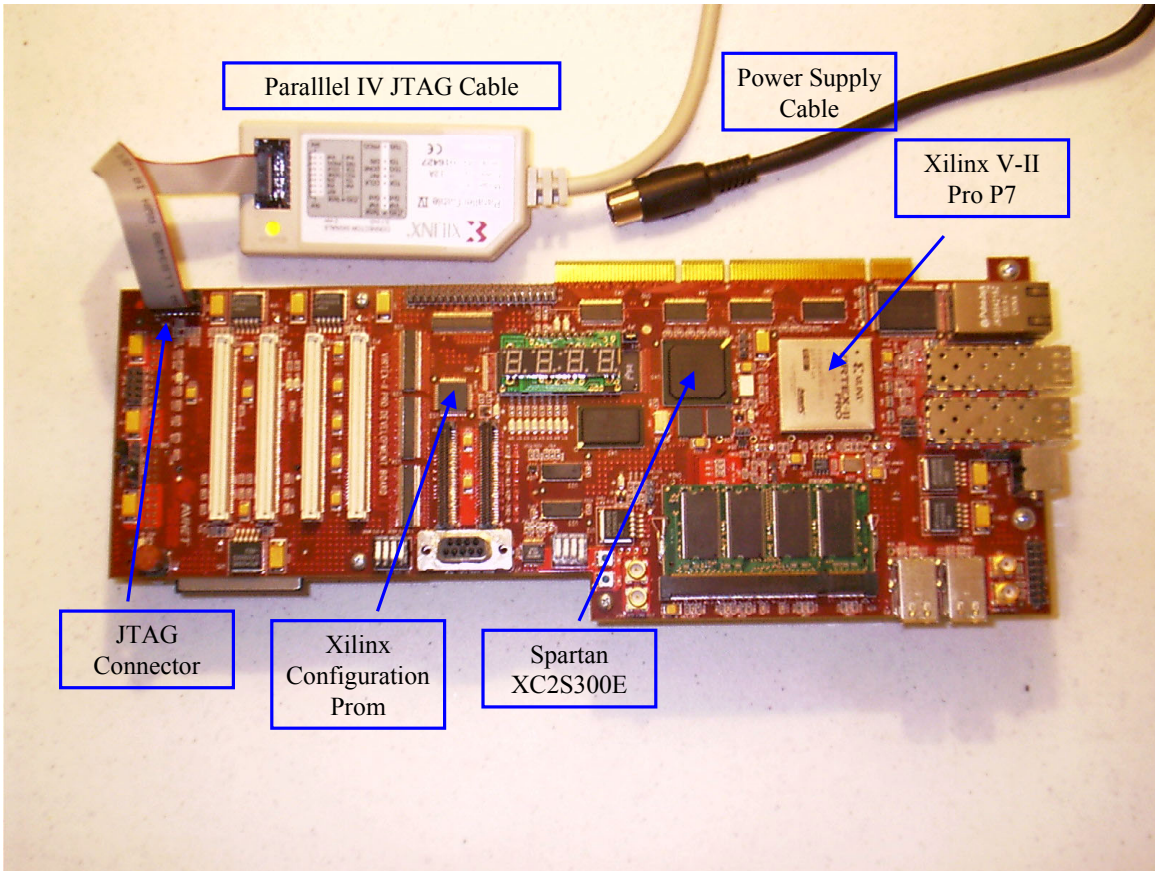


Figure 6-1 : Xilinx Virtex II-Pro Development Board : XC2VP7

task. It became a full time effort, and thus switched the intended thesis path from and SOC focus to an advanced DSP implementation focus.

The only complete development board offered at the time of purchase that housed an FPGA with an onboard IBM 405 PPC is shown in the Figure 6-1. Xilinx does not generally design and develop circuit cards. This is left up to their vendors. One such vendor is Avnet, who designed the V-II Pro development card in Figure 6-1. The FPGA is the Xilinx Virtex II-Pro family, and the chip chosen was the XC2VP7 (called the P7). The package is a Flip Chip fine-pitch BGA (1.00 mm pitch), with 896 total I/O, and 396 user I/O. This is the smallest Virtex II-Pro chip that houses a processor hard core. There were larger chips available, however they did not meet the allotted budget. As illustrated in Figure 6-1, the V-II-Pro development board contains many extras. The main I/O include Ethernet, Firewire, USB, serial, and PCI. To handle the PCI bus, a smaller FPGA, the Spartan XC2S300E, contains the PCI bus arbiter. Since the System Generator for DSP is essentially a closed loop system, there was no need to utilize any other features of the development card other than the JTAG and V-II Pro FPGA. System Generator can however be set up through the configuration files mentioned in chapter 5.3 to communicate with any I/O pin that interfaces with any other system on the circuit card. However, this was not necessary for this research, and thus brings discussion on the Avnet development circuit card to a close.

6.2 General Notes About Hardware Implementation and System Generator

There are some general key mathematical concepts that need to be presented concerning implementation of DSP algorithms in hardware, especially with the System

Generator for DSP. These concepts include signed numbers, double precision floating point, fixed point, rounding, quantization, overflow, and scaling. Further, there are some more key concepts specific to the System Generator that are related to the mathematical key concepts, such as the capability to override any floating point to fixed point conversion with doubles, why the Radix-4 FFT is more efficient, sampling rates and synchronous clocking, and finally, the impact of utilizing high level IP on FPGA design resources. Understanding these key concepts is crucial. It is exactly these concepts that make System Generator such a powerful tool.

6.2.1 Signed Numbers, Double to Fixed-point, Quantization & Overflow

Recall the multiplication example of Figure 2-14 back in chapter 2.4. For simplicity, this example was given using unsigned numbers. However, DSP algorithms typically must be able to work with negative values, since the input to a system is generally a real (vs. complex) values sinusoid that can take any value in the set of real numbers. Working with signed numbers adds additional responsibility to a logic designer. Signed numbers are represented in twos complement format using the same exact vectors and sequence of 1's and 0's as for unsigned numbers. There is no extra logic present in an FPGA or any other general PLD that accounts for keeping track of signed values. Further, there are not generally any separate routing paths specific for signed implementation. Therefore, as far as the hardware is concerned, singed and unsigned numbers are the same. It is left up to the Verilog, VHDL, Matlab, Simulink, etc, designer to be sure these values are handled properly, and this is called sign extension.

For example, if a 4- bit (ex. 1001) value and an 8-bit value (ex. 0001_0010) are to be multiplied together, then both of the values must be sign extended to 8+4 = 12 bits (see multipliers in chapter 2.4). By default, most HDL synthesis tools assume each signal is an unsigned value, and automatically sign extends the 4-bit value using 0's to 0000_0000_1001, and the 8-bit value to 0000_1001_0010. However, if either signal is actually a signed number, then they both must be properly sign extended. If the 4-bit value is negative and the 8-bit is positive for example, then the correct sign extensions would be 1111_1111_1001 for the 4-bit number, and 0000_1001_0010 for the 8-bit number. In other words, any negative number represented in twos complement with more bits than is necessary will simply have an extra string of ones, whereas a positive value represented with extra bits has a string of extra zeros. In the shift and add scheme explained in Figure 2-14 of chapter 2.4, the shift will either be a logical shift (unsigned) or an arithmetic shift (signed), which means fill in with either zeros or ones, respectively. This must be done per partial product in order to properly handle signed values. Since the FPGA and logic in general does not have the means to predict signed/unsigned, or the means to automatically sign extend even if it was known, the designer must be sure that each and every computation is handled properly. The simple solution is to sign extend no matter what. However, since each signal can be different in length, this quickly becomes cumbersome to automate, and generally requires attention on a per operation basis. Further, it may not always be necessary to sign extend every multiplication operation to the full length of $N_A + N_B$. The bit width and gain (magnitude) of an analog to digital sampler is presumably known for most DSP systems. Therefore, the system architect can generally calculate the maximum value necessary for all operations to occur with no

overflow ahead of time (Overflow is discussed shortly). Based on this maximum value, the maximum number of bits needed for all signals in at least a portion of a DSP can be estimated, which will most likely be less than the sum, $N_A + N_B$, of every pair of signals multiplied together.

As has been mentioned throughout this paper, Matlab and Simulink use double format to represent signals in simulation. A double is an N= 64-bit twos complement floating-point value. Since the decimal point can "float" anywhere to right of the sign bit, this allows for values in the range of $+/- 2^{N-1} = 2^{63} = +/- 9.2233\ldots \times 10^{18}$, with a fractional resolution of $2^{-(N-1)} = 2^{-63} = 1.0842\ldots \times 10^{-19}$ (the 64^{th} bit is the sign bit). This is an impressive range, and quite adequate for DSP algorithms. However, this sort of precision is simply not attainable in hardware, including FPGA's. Floating-point values are generally represented in the IEEE 754 floating-point standard format. This format is a normalized (no leading zero) scientific notation. It has a bit for the sign bit, a register for the significand, and a register for the exponent. The generic form for the IEEE representation is $(-1)^S \times (1 + \text{significand}) \times 2^E$, where S is the sign bit and E is the exponential value. The number of bits used to represent the exponent and significand depend heavily on the platform (i.e. processor) and the software. As explained above, Matlab and Simulink allow both the significand and the exponent bit lengths to vary in order obtain both maximum range and precision from the 64 available bits.

Not only is there no means of automatically detecting and executing sign extension in logic, there is also no extra hardware overhead for implementing the IEEE floating point. In fact, all values in logic must be represented in a fixed-point equivalent integer-like format, as will be explained shortly. However, the conversion from floating-

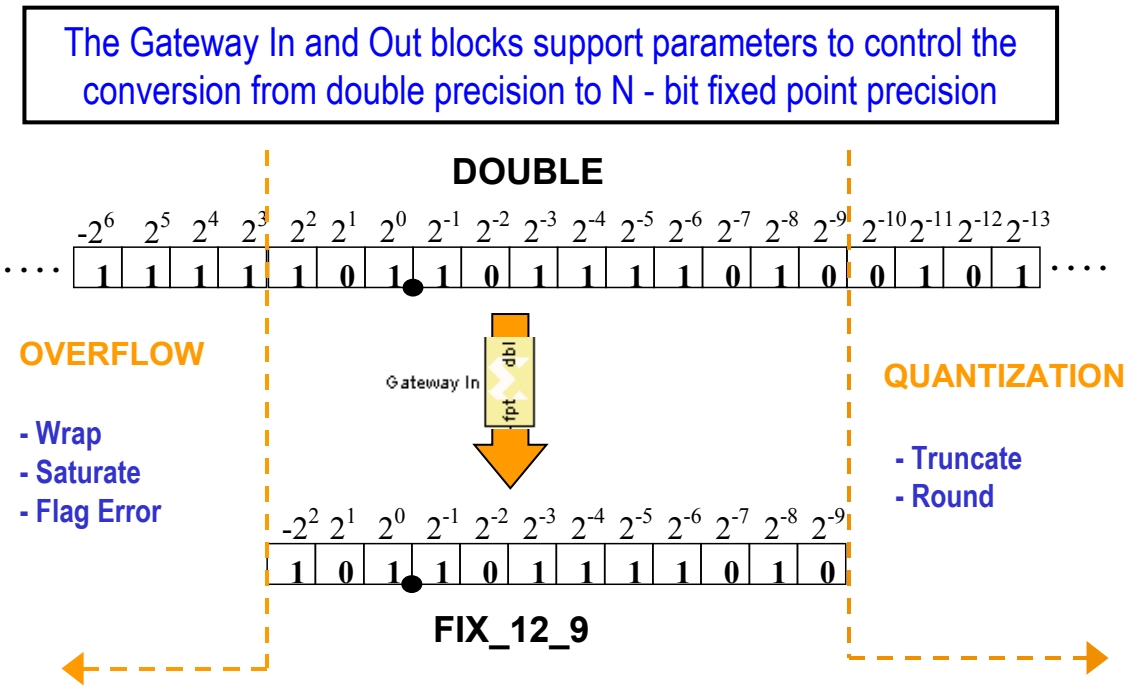


Figure 6-2 : Xilinx Gateway Double to Fixed-Point Conversion

point to fixed-point is not a loss less transition. Two phenomena can occur, called overflow and quantization. In order for Simulink to simulate a System Generator design, signals must be converted before they reach any of the Xilinx blocks dedicated for hardware. The gateways blocks mentioned in chapter 5.2 provide this conversion. However, they first convert from double precision floating- point to a fixed-point representation, as illustrated in Figure 6-2. If a real value is represented in a binary format with a decimal rather than in the IEEE floating-point format as in the top diagram of Figure 6-2, then conversion to fixed-point is conceptually straightforward. In Simulink, there are 2 types; unsigned fixed (Ufix) and signed fixed (Fix). As illustrated in the figure, the notation for a signed fixed-point value is *FIX_total bits_binary point.* In

this case, total number of bits is 12, and the binary point is 9, leaving 2 bits for the integer, plus the sign bit.

Quantization, shown in the right side of Figure 6-2, always occurs in a floating-point to fixed-point conversion, since there must be some limit set to the number of decimal places that are kept for hardware representation. Simulink and System Generator provide two options, either truncate or round, to handle quantization, as illustrated in

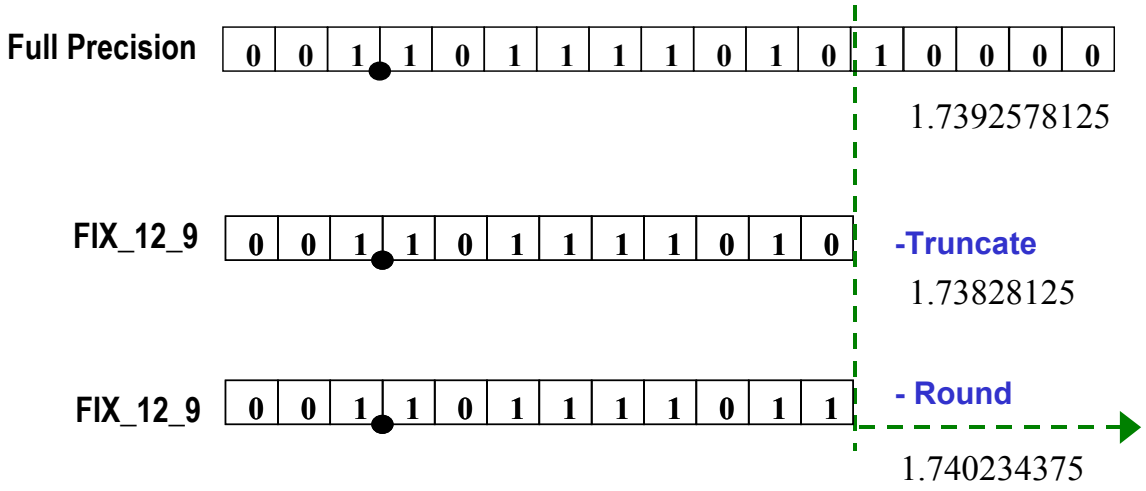


Figure 6-3 : System Generator General Block Parameter Options for Quantization

Figure 6-3. Truncation is as simple as discarding the bits to the right of the most significant bit. Rounding rounds to the nearest representable value, or away from zero if the value is midway. In either case, it's a form of rounding. Truncation is equivalent to

rounding toward negative infinity. Rounding toward the nearest representable fractional value depends on the sign. If it is a positive value, then the midpoint of the next fractional value is added to the existing value. For example, if the floating-point value is 4.565, and

the fixed-point binary point is to be one, then a value of .050 (mid-point of a binary point of 2) would be added to the floating-point value before truncating. On the other hand, if the signal is negative, then the maximum positive fractional value of next binary point to the right of the target must be added. For example, if the floating-point value is –4.565, and the desired fixed-point binary point is again one, then .099 (positive maximum of binary point of 2) must be added. By adding the appropriate rounding values, it forces the fractional parts that are not going to be truncated to proper value. In other words, the rounding must occur before truncation. Notice that the rounding shown in Figure 6-3 is not a base-10 round, but rather a base-2 round.

Overflow, shown in the left side of Figure 6-2, only occurs when the floating-point value coming into a Xilinx gateway lies outside the range of the fixed-point representation set for that input. Recall, gateways are equivalent to FPGA I/O. The issue of overflow and quantization is not a concern for FPGA outputs, that is, for a Xilinx fixed-point to floating point gateway, since the floating-point representation is always more than adequate to cover the range of any fixed-point representation. However, for the Xilinx floating-point to fixed-point gateway, a designer can choose to saturate the results, wrap the result, or produce an error flag, as illustrated in Figure 6-4. Saturation forces the

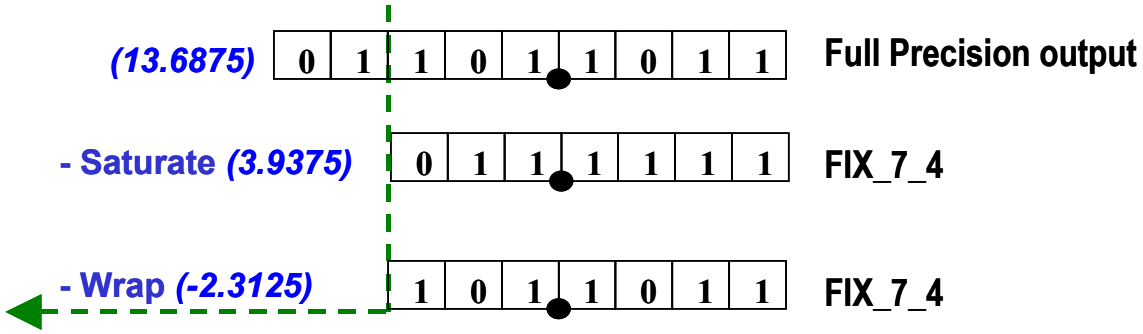


Figure 6-4 : System Generator General Block Parameter Options for Overflow

fixed-point value to the largest positive or negative representation. Wrap is the opposite of truncate for quantization; any bits beyond the most significant bits are discarded. The third option is to have Simulink generate a flag (a warning), which is a simulation only option; a physical logical warning signal is not created for the hardware to monitor, since there are no additional resources available in the FPGA to handle the warning. Notice in Figure 6-4 the large dynamic range between the saturation and wrapping options for overflow. Clearly, neither option is desired, as both results re very far from the actual number. In fact, wrapping allows a positive value to suddenly become negative, creating all kinds of strange results. As mentioned above in the discussion on the implementation of signed numbers, a system architect can generally strike a balance between not sign extending to the absolute maximum, while still avoiding overflow. However, as the size and complexity of the DSP algorithms grow, it becomes increasingly difficult for a system architect to anticipate the range of all possible results from potentially millions of multiplication operations, especially when the input signal to the A/D varies over a large range, as in the case of electronic warfare mentioned in chapter 4.2. As will be discusses shortly, one of the key advantages of System Generator is the ability to easily analyze the effects of quantization and overflow.

As has been eluded to, there is yet another conversion that must be made in order to actually proceed from System Generator simulation mode, to hardware co-simulation mode. Signals in hardware are represented as integer values. HLD's such as Verilog and

VHDL do not provide a simple solution for handling real numbers. In general, there are only integer vectors and integer matrices (memory) available in logic. However, converting a FIX_7_4 fixed point value into an HDL compatible representation, for example, is as simple as creating a 7-bit signal. In other words, just remove the binary point. The hard part is that the logic designer must keep track of where the binary point is. This is straight forward, until multiplication with signals of differing binary points is executed across the entire FPGA design. The task again lies on the shoulders of the HDL designer, and becomes very tedious.

6.2.2 Override with Doubles capability: KEY CONCEPT and ADVANTAGE

The Xilinx gateways perform double precision floating-point to user specified fixed-point conversion, and vice versa. Each Xilinx block that exists between the gateways, that is, each block targeted for hardware, is simulated with a fixed-point format, rather than a floating-point format. However, probably the most important advantage that the System Generator for DSP offers, and a very key concept to take away from this research, is that this data format conversion can be switched on and off. The general block parameters have a feature called *override with double*s. This feature allows for any gateway, Xilinx block, sub-system (hierarchical module), or the entire design to be simulated in double precision floating-point format. The fixed-point conversion can be systematically turned back on at strategic locations in the design in order to help pinpoint a sub-system or Xilinx block that is the source of any overflow and/or quantization error. The concept that System Generator allows for such a scheme is novel in itself. However,

once the ailing sub-system or design block has been identified and corrected, the design can be synthesized and place and routed to create a JTAG-Cosim block (see chapter 5.3), and immediately be verified in hardware. Since the results of the simulation nearly always match that of the hardware, the designer should be able to bring complete functional and timing closure to a design in this manner.

So how does System Generator handle keeping track of the binary point and pass this information to the Xilinx Synthesis and place and route tools when its time for actual implementation in hardware? This is precisely the advantage of simulating the design in both floating-point and fixed-point. First, if it is so desired or necessary, the entire design can be simulated in floating-point. Once the design has been functionally simulated to a designer's satisfaction, one, some, or all of the blocks targeted for hardware can be switched over to simulate in fixed-point. It may take several iterations to fine tune each block in the system until the user defined fixed-point settings result in acceptable amounts of quantization error and no overflow issues through the entire system. Once these values are set, System Generator stores the fixed-point information for the Synthesizer and place and route tool to utilize in creating the logic. Inside the logic of the FPGA, all calculations are performed in the integer vector fixed-point equivalent (i.e. the same number of bits as fixed-point, but with an implied binary point). System Generator also stores the fixed-point information for use by the gateways out, or the FPGA outputs. Here, the number of binary points of each multiply operation on a given output path is kept track of from beginning to end, thus allowing for the proper conversion from vector (hardware) format to fixed-point, and then to finally to floating point representation at the end of the signal path (the sink). The systematic analysis and debug of quantization and

overflow errors, followed immediately by hardware verification in the Simulink environment with accurate conversion of the output data back to floating-point, provides a DSP architect and FPGA designer with a very powerful, high level single flow tool for one stop design and verification.

6.2.3 Scaling and FFT's

Referring back to the radix-2 8-input FFT example of Figure 4-3, if the inner structure of the of butterflies are examined more carefully, some interesting observations can be made. In each butterfly, there is essentially one complex multiply, one addition, and one subtraction. Since the absolute value of any twiddle factor is one, multiplying by a twiddle factor within a butterfly cannot cause overflow at the end of a stage. However, the overall butterfly operation *can* cause overflow from the addition/subtraction. A method sometimes employed to reduce overflow is to do a one-time normalization of inputs by N. However, if the absolute values of both inputs to the butterfly operation are forced to be less than 0.5, then overflow at the butterfly level can be eliminated. There are two common approaches implemented in FFT algorithms to eliminate overflow in this manner. The first approach is to multiply all N inputs (see Figure 4-3) to each butterfly by 0.5. Since there are $\log_2 N$ stages, the N outputs would be scaled by $0.5^{\log2(N)}$, and thus can be rescaled back to normal at the end of computation. This method is preferred over normalization, since it makes better use of the natural capabilities to execute base-2 multiplication and division. In other words, multiplication by 0.5 (division by 2) is a simple right shift in hardware. However, thinking about it from

a shifting perspective, there is still a significant loss of bit precision in this scheme. For example, if N = 512, then 9 bits of accuracy would be lost.

In the second scaling scheme, overflow detection is inserted at each butterfly. If overflow is detected, the array inputs to the current FFT stage are multiplied by 0.5, and the ailing overflow is adjusted. A counter, *cntr*, is used to keep track of the number of time overflow occurred, and therefore the FFT outputs are scaled by 0.5^{cntr}. This scheme is referred to as block floating point, and provides variable scaling depending on the input sequence.

The FFT cores in the System Generator allow for no scaling, scaling by 1/N and other various dynamic (i.e. different per FFT stage) scaling factors, and block floating point. Again, the FFT radix that was used in the FFT front-end sub-system was the radix-4, which does not support block floating point scaling. In order to implement the scaling options, it requires a vector of scaling values that get registered at the beginning of each FFT frame. The elements of the vector are the scaling factor for each stage of the N-point FFT, and since the there $\log_2(N)$ stages, there are $\log_2(N)$ scaling factors needed. The allowable scaling values per stage are a shift right of 0, 1, 2, and 3 bits, indicating a multiplication by 1, 0.5, 0.25, or 0.125. Unfortunately, at the time of that this thesis was submitted, the scaling option on the FFT front-end sub-system had not been implemented, and therefore there is no further analysis on the impact of scaling on the resulting FFT spectrum output from the FFT front-end subsystem.

6.2.4 Radix – 4 FFT

There is another interesting observation that can be made by examining the inner structure of the butterflies of the 8-point FFT example of Figure 4-3. The butterfly configuration for a 4-point FFT (i.e. radix-4) can be implemented without any multiplication at all! Looking at the top two blocks of stage one and the top block of stage 2 of Figure 4-3, it can be seen that three of the four twiddle factors for these two stages are equal to one. The fourth twiddle factor is $W_8^2 = -j = -\pi/2$, as was previously explained in chapter 4.1. The multiplication of a complex number a+jb by –j gives b- ja, which is simply swapping the real part with imaginary and the imaginary part with negative of the real. This sort of operation is very easily implemented in hardware by hard wiring the results at this stage to automatically perform the swap. Using this concept, a 4-point FFT can be calculated using 8 additions and no multiplies! Therefore, if larger point FFT's are built out of 4-point FFT butterflies, then the overall efficiency of the FFT can be improved. In the case of the 8-point FFT of Figure 4-3, there are $N/2\log_2(N)$ complex multiplies = 12. However, the scheme just descried can be used to compact the first two stages of Figure 4-3 into two 4-point FFT's. Now, there are no complex multiplies needed in the first two stages, thus leaving only the 4 complex multiplies in the final stage. Therefore, the performance gained in complex multiplications by using a radix-4 to compute an 8-point FFT is 4/12 = 1/3, or about 33%. Further reduction can also be obtained in noting that there is always a twiddle factor after the 4-point FFT stages that is equal to –j, and thus can also be hardwired to improve performance. In general, the number of complex multiplies for the *Radix-4* FFT becomes $N/2\log_2(N) – N$, since there are N/2 multiplies per stage, and the first 2 stages of multiplies are removed in the radix-4 implementation. There is one caveat to the radix-4

FFT; to use it the number of samples in the input frame must be a power of 4 (i.e. 4, 8, 64, 256, 1024, etc). The FFTxx core in System Generator allows for FFT's of length 64 to 16384. In order to implement the other radix-2 FFT lengths of 128, 512, etc, the FFTxx cores combines both the methods of radix-4 and radix-2, thereby still maintaining significant performance gain in complex multiplies where ever possible. As mentioned before, 1024 is about the parallel limit for most 16-bit FFT's, and therefore any length above this most likely shares multiplier and memory resources.

6.2.5 Sampling Rates and Clocking in Hardware.

One of the most important concepts needed to design efficiently with System Generator for DSP is that of sampling. As mentioned back in chapter 5.2, each Simulink block has a sample period field that indicates how often the blocks function will be calculated and the results outputted. The units for the sampling period for all blocks are in seconds. Most blocks can derive their sampling rate from the block feeding it by inserting a –1 in that field. However, blocks such as a gateway in or a source (i.e. sinusoid) must have their sampling periods explicitly set. Again, parameters from the Matlab workspace can be used here rather than absolute values. As is the case with any DSP application, the smallest sampling rate (largest sampling period) is constrained by the Nyquist Theorem ($F_S > 2\ f_{MAX}$). For the FFT front-end subsystem design, there was only one sampling rate needed. However, in more complicated designs, such as multi-rate systems that utilize up samplers and down samplers, there will be several different sampling rates. In this case, the greatest common divisor (GCD) among all sampling rates

is used as the global system sampling rate. This can be manually calculated and entered into the System Generator block (see left side of Figure 5-7). However, System Generator will automatically calculate this at simulation time. System Generator also provides formatting options and design blocks that allow a designer to view the sampling period of any one or all signal nets. The sampling period of every other block in the design must be an integer multiple of system clock. Often, the collective sampling rates across a design are normalized to the system clock. This simplifies the conversion of the sampling rates/periods to a hardware implementation in the FPGA.

The sampling period defined for each block directly relates to how that block will be clocked in the actual hardware in the FPGA. Every System Generator design block receives the same *synchronous* system clock. The Avnet development card (see chapter 6.1) utilized in the implementation of the FFT front-end subsystem had two available system clock inputs to the FPGA; 100 MHz and 33 MHz. As will be explained shortly, it was necessary to utilize the 33MHz clock due to timing limitations in the 512-point FFT implementation. During the generation sequence from simulation mode to hardware co-simulation mode (see chapter 5.3), System Generator creates clocking circuitry called xclockdriver.vhd. This logic is essentially composed of a counter and some comparator logic that creates an appropriate clock enable (CE). A one-to-one correlation is drawn between the 33MHz FPGA system clock and the System Generator GCD sampling period. Since each System Generator block has a sampling rate defined as an integer multiple of the GCD system sampling rate, then conversion of each simulation sampling rate to an actual hardware sampling rate becomes a straightforward ratio. The CE from the xclockdriver.vhd. logic is asserted for the appropriate multiple of FPGA system clock

cycles in order to achieve the desired equivalent sampling rate at each respective block. For example, suppose the System Generator GCD sampling period is 1/2000, and block 'A' in the design uses a sampling period of 1/1000, which is exactly twice the sampling period, or half the rate. Therefore, in the hardware, each block would receive a 33MHz clock, and the clock enable (CE) on block 'A' would be asserted for two clock cycles in order to obtain the desired rate of one half, which in this case happens to be 16.5 MHz. By normalizing the simulation sampling rates, it becomes quick and easy to convert each sampling rate in a design to its equivalent real hardware frequency, despite the actual clock oscillator value.

6.2.6 Effects of high level IP on design resources

Every intellectual property (IP) core in System Generator has an associated HDL wrapper that links it between Simulink and hardware, as illustrated in Figure 6-5 [4]. As has been previously explained, these wrappers extend the core's functionality and simplify the interfaces by providing GUI's. The GUI allows parameters such as the number of bits, binary point, overflow, quantization, etc, to be programmed by the designer. This system

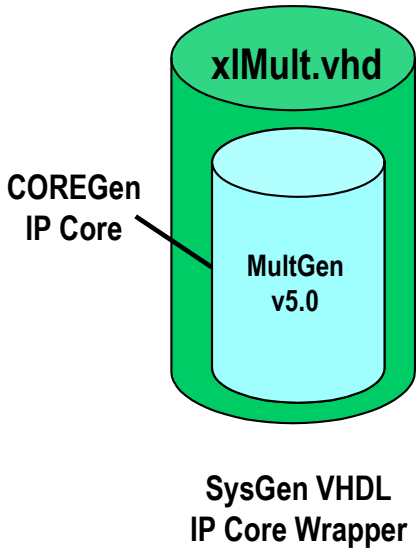


Figure 6-5 : System Generator HDL IP Core Wrapper

level abstraction is very powerful and allows for quick experimentation, however, it comes at a cost in hardware. For example, recall that for quantization the options are to truncate or round, and that for overflow the options are to wrap or saturate. Again, truncation and wrapping require no additional hardware. However, if saturation or rounding is selected in the GUI parameters, extra logic is required such as full adders (rather than half) and additional control logic. The scaling option of the FFT core also requires additional logic. Recalling the pipeline discussion in chapter 2.1 and Figure 2-6, some of the block parameters allow for built in programmable latency (i.e. pipelining). This is implemented with a shift register in the distributed SRAM of the LUT's. Further, some blocks perform implicit conversion of signals, such as the unsigned to signed, sign extension, and zero padding. These cores all require additional hardware to implement their functions. Further, some cores allow the designer to choose between different implemention options, such as using Distributed RAM versus Block RAM for memory.

This choice can not only affect the timing due to type of memory, but also the routing delays required to gain access to the specified resource. A designer must keep in mind that with each additional parameter that squeezes and optimizes each core and each signal, more and more logic resources in the FPGA are being utilized. Design trade offs must be made in order to strike a compromise between the desired results, hardware utilization, and inevitably, timing.

6.3 **Analysis of Hardware Implementation of the PBFE Algorithm**

At the time this research paper was written, the phase-based frequency estimation (PBFE) algorithm was not completely implemented in Simulink. The theoretical research combined with the design, implementation, and verification of the FFT front-end subsystem captured the all of the allowable time scheduled for this research. Therefore, the design is presented here in its current unfinished state. Referring back to the Matlab algorithm flowchart of Figure 4-14, steps 7, 8, and 9 were never completed, and thus there will be no discussion for those steps. The design was implemented up to step 6, phase unwrapping, however the verification of steps 4 through 6 was not completed. Thus, any results presented beyond the FFT front-end subsystem are to some extent speculation and not official.

Figure 6-6 illustrates the top-level design of the phase-based frequency estimation algorithm in Simulink. As can be seen, there are levels of hierarchies, called subsystems,

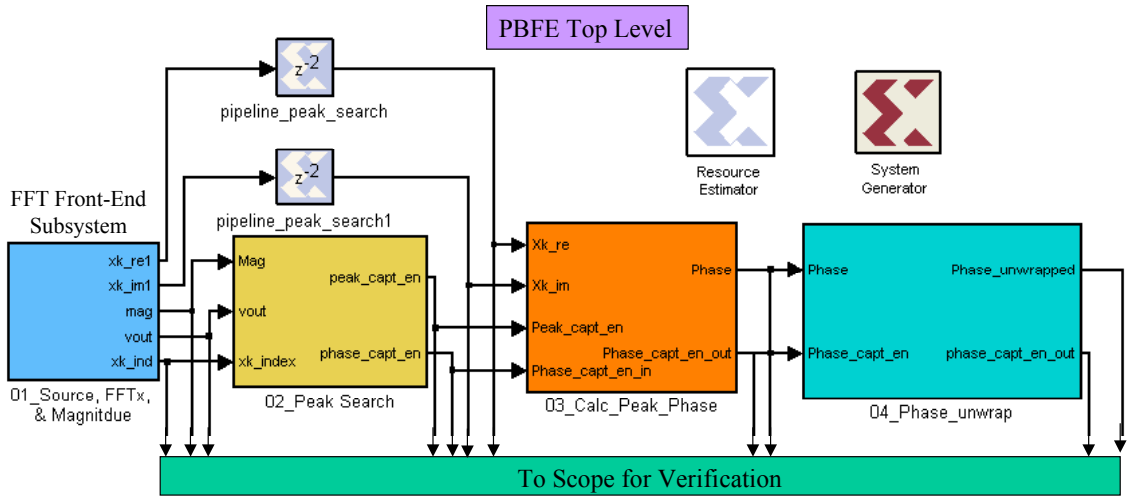


Figure 6-6 : PBFE Algorithm System Generator Implementation

just as there are in an HDL design. Since Simulink is a block diagram or schematic like entry tool, Figure 6-6 servers as a suitable flow chart for the System Generator implementation of Matlab PBFE algorithm presented back in Figure 4-14. As shown, the PBFE top level consists of four subsystems. The FFT front-end subsystem implements the storage of the source vector, the FFT, and the magnitude calculation. This subsystem is described in detail shortly. The next 3 subsystems, the peak search, phase calculation, and phase unwrap, follow exactly with steps 4, 5, and 6 of Figure 4-14. Since this part of the design was never officially verified, these 3 subsystems will be discussed together in chapter 6.3.2.

There are some considerable differences between the software implementation of the frequency estimation algorithm and the hardware implementation, namely the loop. As was briefly mentioned in chapter 5.2, the System Generator FFTxx core is designed to handle a continuous stream of input data. The Matlab algorithm of Figure 4-14 is a loop-based algorithm that works on static data, meaning it takes a block of data, operates on it,

stores it, and then comes back to get another block of data This sort of loop-based approach can be implemented in hardware at the expense of a lot of memory resources (ROM's, RAM's, and registers). However, this approach is generally not desirable or achievable in logic. Instead, DSP algorithms like the PBFE algorithm must be reconfigured to be hardware friendly, allowing for continuous streaming calculations at each step of the process. Any data that is needed at a later stage of the design is simply pipelined the appropriate number of clock cycles until it is needed, as illustrated in Figure 6-6. The total delay through the peak search subsystem is 2 clock cycles. The outputs of the FFT front-end subsystem are needed at the inputs to the both peak search and the peak phase calculation subsystems, and thus the latter inputs must be pipelined by 2 clock cycles. Further details on how the Matlab algorithm was reconfigured for hardware implementation are given in the next two sections.

6.3.1 Analysis of FFT Font End Hardware Implementation

The most critical subsystem is the FFT front-end, which is why so much time was spent on its design and verification. Obtaining the correct timing of the magnitude of the FFT outputs was crucial to the verification of the latter subsystems of the PBFE algorithm. This is why the FFT front-end script of Figure 5-3 created to compare and simulate the FFT front-end subsystem. The first task was to create a memory storage element that could accept any frequency sinusoid in order to eventually test a range of frequencies to verify the frequency estimation limit curve. First, the memory needed to be loaded with a vector of sufficient length. However, unloading the memory was a bit more challenging. Since the FFT core expects streaming data, that is, back-to-back DFT

frames (1 frame = 1 DFT block), the memory had to be unloaded in accordance to the sliding DFT block scheme of Figure 4-9. The FFT core itself has 3 key control signals; start, x(n)_index out, and valid out (vout). Once the memory is loaded, the start signal is asserted on the FFT, which causes the x(n)_index signal output to increment from 0 to $N_{FFT} - 1$, repetitively, until the start signal is removed. This built in incrementor served as the basis of the counter logic to force the RAM to output a signal to the FFT core that is equivalent to the sliding DFT blocks of Figure 4-9.

The second task was calculating the magnitude of the DFT result. This could have been done in Matlab, since the results were being sent there anyway, however, it was needed for the PBFE algorithm anyhow. Further, it assisted in the debug and verification of the FFT front-end, namely the capturing of the results. The third and final task was to output results back to the Matlab workspace for comparison against the original Matlab algorithm. The output of the FFT core is back-to-back DFT's in a single stream. The valid out (vout) signal from the FFT and a clock was used to achieve this. The vout is asserted when the first valid DFT sample of the first frame is present, and de-asserted when the last valid DFT sample of the last frame is present. The timing of the FFT core and all other control signals can be very tricky. By putting the magnitude calculation in the FFT subsystem, it was easier to debug. The source signal was a sinusoid, so it was very clear both on the scope and in the Matlab comparison sub-plots when the real and imaginary signals were either misaligned, or completely wrong altogether, since the magnitude

calculation required the squaring and summing of the real an imaginary samples. The FFT front-end subsystem has already been discussed in chapter 5.2, and can be viewed in

Figure 5-2. The only difference between Figure 5-2.and the actual subsystem is the input and output ports, which will be explained shortly.

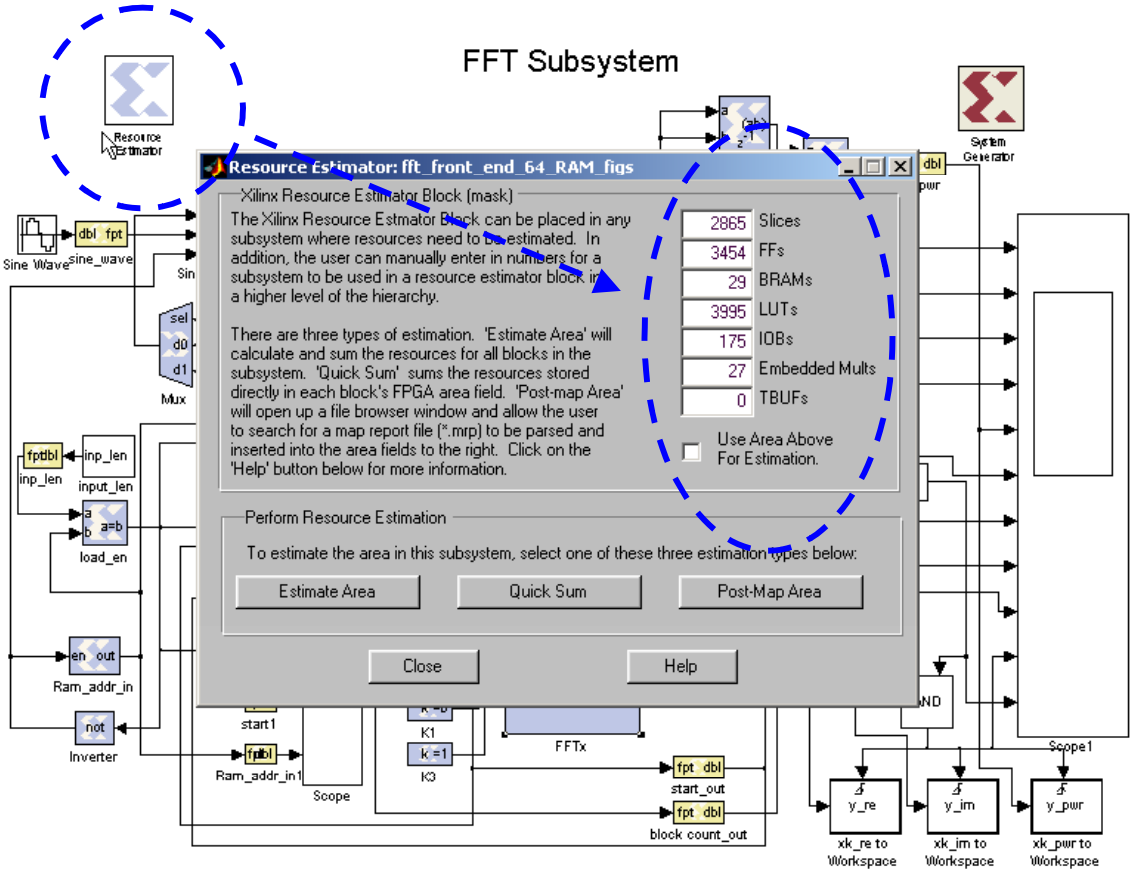


Figure 6-7 : Resources: 64-point FFT Front-End

Another useful Simulink block in the Xilinx toolbox that was not discussed in chapter 5.2 is the Resource Estimator. This block is illustrated in Figure 6-7. When this block is included in a design, there are three estimation choices as shown in the bottom of the GUI. Of the most useful is the estimation area option. By clicking this, the FPGA architectural resources needed to implement the design in hardware are estimated so that

a designer may get insight into the size of the FPGA device that is needed. Figure 6-7 shows the resources needed for the 64-point FFT case, which was used to design and debug the FFT subsystem. The table in Figure 6-8 illustrates the resources available in the

Table 1: Virtex-II Pro FPGA Family Members

Device	RocketIO Transceiver Blocks	PowerPC Processor Blocks	Logic Cells[1]	CLB (1 = 4 slices = max 128 bits)		18 X 18 Bit Multiplier Blocks	Block SelectRAM+		DCMs	Maximum User I/O Pads
				Slices	Max Distr RAM (Kb)		18 Kb Blocks	Max Block RAM (Kb)		
XC2VP2	4	0	3,168	1,408	44	12	12	216	4	204
XC2VP4	4	1	6,768	3,008	94	28	28	504	4	348
XC2VP7	8	1	11,088	4,928	154	44	44	792	4	396
XC2VP20	8	2	20,880	9,280	290	88	88	1,584	8	564
XC2VP30	8	2	30,816	13,696	428	136	136	2,448	8	644
XC2VP40	0[2], 8, or 12	2	43,632	19,392	606	192	192	3,456	8	804
XC2VP50	0[2] or 16	2	53,136	23,616	738	232	232	4,176	8	852
XC2VP70	16 or 20	2	74,448	33,088	1,034	328	328	5,904	8	996
XC2VP100	0[2] or 20	2	99,216	44,096	1,378	444	444	7,992	12	1,164

Notes:
1. Logic Cell = (1) 4-input LUT + (1)FF + Carry Logic
2. These devices can be ordered in a configuration without RocketIO transceivers. See Table 3 for package configurations.
3. -7 speed grade devices are not available in Industrial grade.

Figure 6-8 : Virtex-II Pro Resources by chip.

XC2VP7 FPGA on the Xilinx development card. As mentioned in chapter 2, the most critical resources are the LUT's, Multipliers, Block Rams, and registers. Note that the table in Figure 6-8 only indicates the number of slices. However, from chapter 2.1, it is known that there are 4 slices per CLB, and each slice has 2 LUT's and 2 registers. Thus, there are 4,928/4 = 1232 CLB's, and 4928*2 = 9856 LUT's and registers in the XC2VP7 FPGA. Compare the resources used by the 64-point FFT front-end subsystem of

Figure 6-7 to the total resources available. Roughly 60% of the available slices are utilized. About 40% of the LUT's are in use, and about 35% of the registers are in use. 29 out of 44 Block Ram's are utilized, and 27 of 44 embedded multipliers are utilized. Recall that the Multipliers and Block RAM's share resources. Since 27 + 29 is greater than 44, some of the multipliers must be utilizing the Block RAM's. The individual resources used by each block can be viewed by looking at the block parameters. The FFT core uses 18 of the 27 Multipliers, and 27 of the 29 Block Ram's. The other 9 Multipliers are used by the multiplier cores shown in Figure 5-2. The RAM used to store the input signal utilizes two Block Ram's. Therefore, the 18 multipliers and 18 Block RAM's in the FFT core share resources. Clearly, the FFT core utilizes the largest amount of resources, as should be expected. Already it can be seen the FFT subsystem itself, without any additional hardware for the frequency estimation algorithm, utilizes roughly 50% of the P7 FPGA resources.

Compare this with the 512-point FFT subsystem of Figure 6-9. Notice that the number of Block RAM's and Embedded Multipliers did not increase when the FFT size was increased from 64 to 512. This would seem to indicate that Xilinx is reusing some of the resources in the core, and that the FFT core is *not* an entirely parallel implementation. As mentioned in chapter 5.2, there is a delay through the core, as is evident by Figure 5-4. Although probably a very efficient algorithm, this is a prime example of the effects of high level IP and higher system abstraction on a design. If a designer absolutely needs to

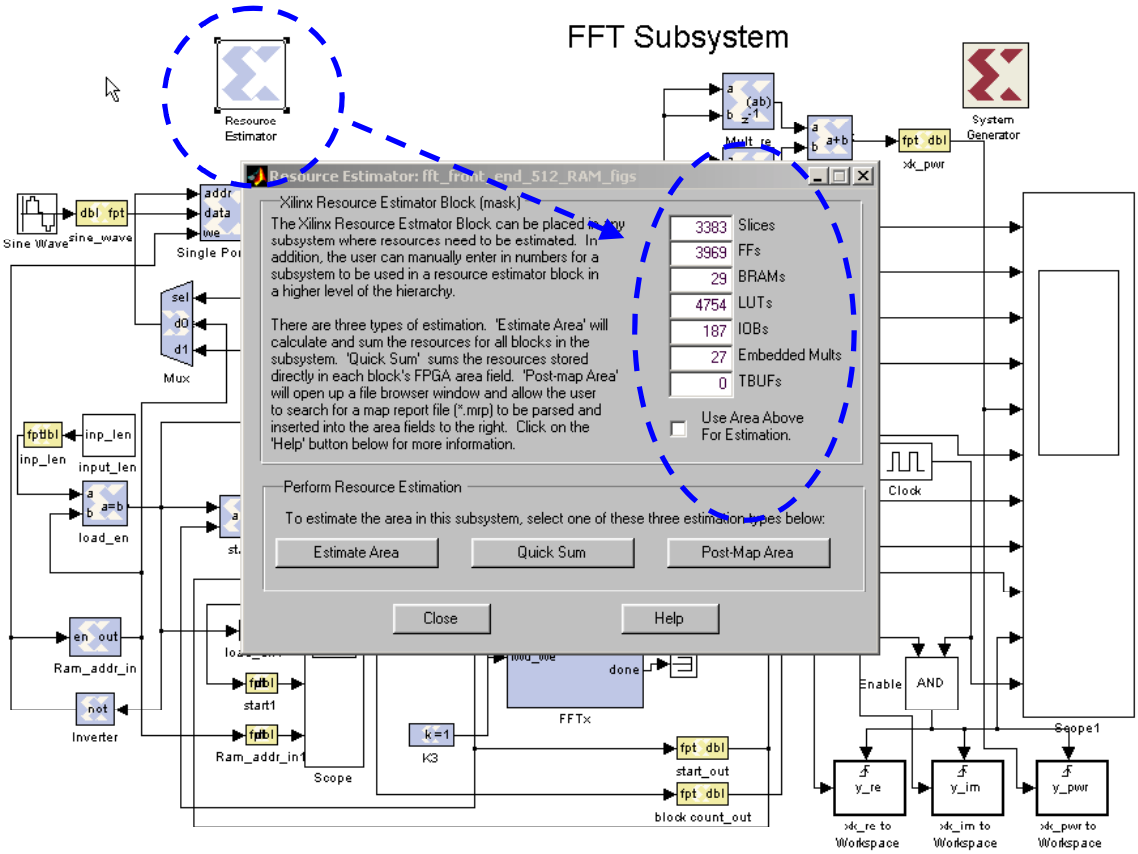


Figure 6-9 : Resources: FFT Front-End, 512 case

have a 512-point FFT execute in one (or very few) clock cycles, then more knowledge of this core would be required. However, since it is an IP wrapper, further details are often not available, and full parallelization is probably not possible, thus forcing a designer to implement his/her own FFT core.

In the 512-point FFT case, 69% of the slices, 40% of the registers, and 48% of the LUT's are in use, which is approximately a 10% increase in the non-dedicated resources from the 64-point FFT. One might estimate that the 512-point FFT front-end subsystem

utilizes approximately 60% of the entire FPGA. Since the original Matlab phase-based frequency estimation (PBFE) algorithm specified a target FFT of length 512, then this leaves about 25~30% of the FPGA available for the rest of the PBFE algorithm. The reason that the full 40% is not available is for timing consideration due to routing. Recall the XC2VP7 device has both a 100 MHz clock and a 33 MHz clock available. When the 512-point FFT front-end subsystem was targeted for hardware via the System Generator, it failed timing, forcing the use of the slower 33 MHz clock. The more logic that gets packed into an FPGA chip, the tougher it becomes for the place and route tools wire up all the mapped components and meet timing. Therefore, it is more or less an FPGA design rule of thumb to leave some breathing room and not max out the available resources.

6.3.2 Analysis of remainder of PBFE Algorithm Implementation

The three subsystems following the FFT subsystem had to be designed to receive a continuous stream DFT samples. The peak search subsystem is illustrated in Figure 6-10. Note that each subsystem is linked to the top-level design and other subsystems via the input and output ports. To fine the sinusoidal spectral peak of each DFT block, a simple scheme of $(i > i\text{-}1)$ is implemented. That is, the magnitude from the FFT front-end is delayed by one clock cycle, and then the current and delayed version are sent into a greater than comparator. The valid out signal from FFT core is used to enable the comparator. As long as the current sample is larger, the peak capture enable will be asserted. This is used in the next subsystem, phase calculation. The x(n) index from the

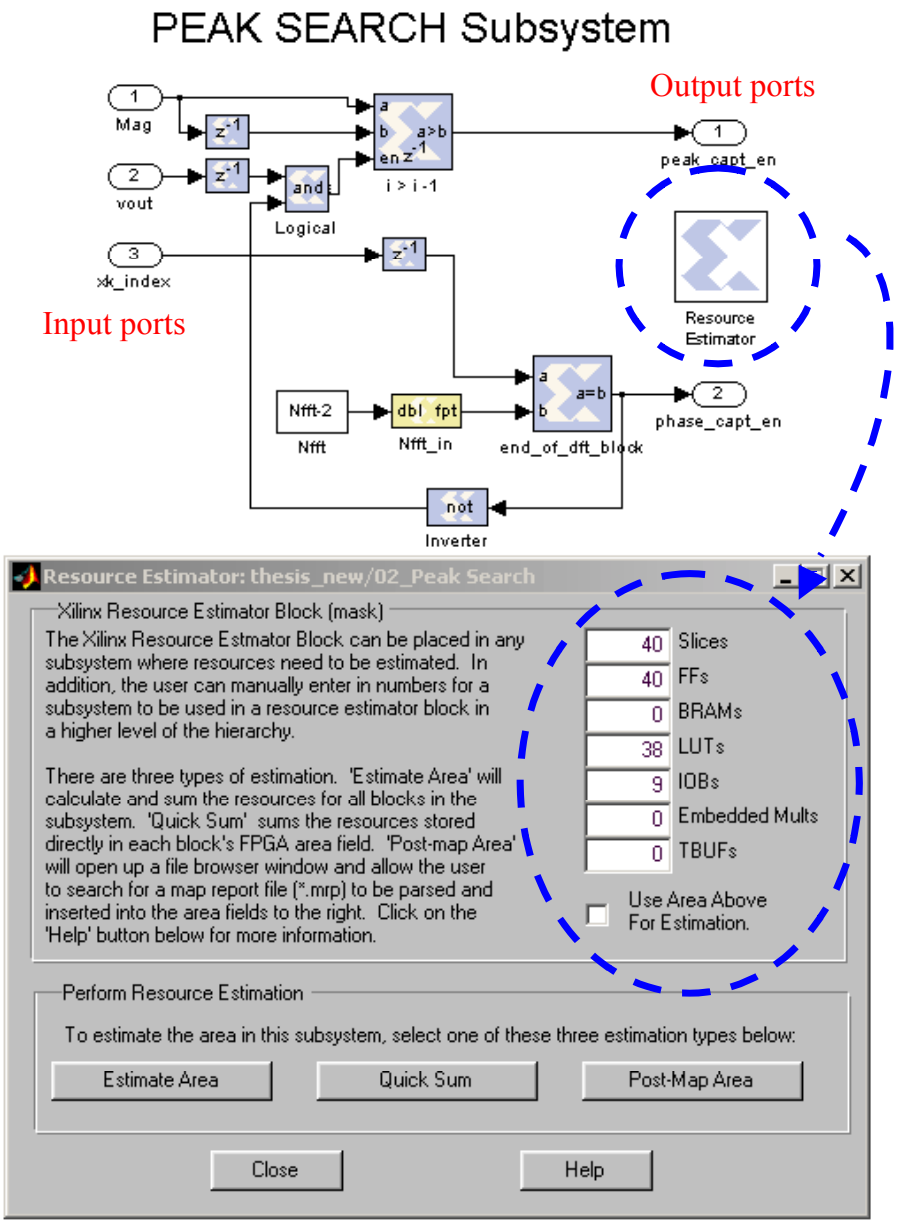


Figure 6-10 : Peak Search Subsystem of PBFE Algorithm

FFT core is also used to indicate the end of a DFT frame, which creates the phase capture enable, also utilized by the phase calculation subsystem. As can be seen in Figure 6-10, the peak search subsystem uses up minimal resources.

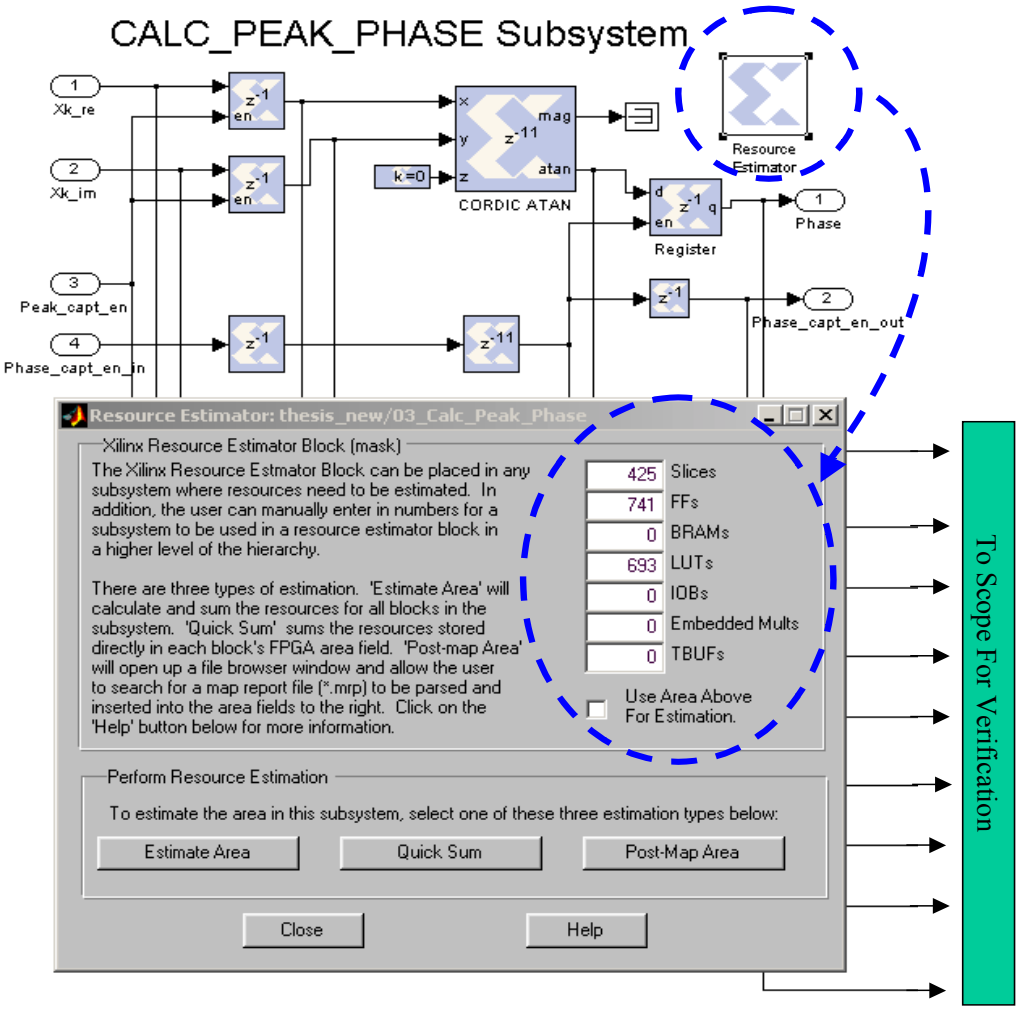


Figure 6-11 : Calculate Peak Phase Subsystem of PBFE Algorithm

Figure 6-11 illustrates the subsystem that calculates the phase at the sinusoidal spectral peak of each DFT block. This subsystem uses the pipelined real and imaginary outputs from the FFT core mentioned earlier. The peak capture enable signal from the peak search subsystem is used enable the registers. The output of the registers is fed into

the Cordic core. The Cordic core is a Xilinx IP core used to calculate trigonometric values, and has a latency of 11 clocks cycles. The output of the Cordic is a phase value in radians, in the range of $-\pi$ to $-\pi$, as mentioned in the theory of chapter 4.4. When the phase capture enable is active, the output of the Cordic core is registered. Note that the peak capture enable must be pipelined 11 clock cycles in order to line up with output of the Cordic core. In this manner, as the DFT output samples proceed from negative frequency to positive frequency, the greater than comparator of the peak search system will stop at the most positive peak, thus deactivating the peak capture enable. The last pair of DFT outputs latched into the first set of registers in the calc_peak_phase subsystem will be the real and imaginary values at the peak. Once the frame has ended, the phase capture enable stores the actual peak phase at the register on the output of the Cordic. Since there are N_{FFT} sample outputs, the phase capture enable signal will pulse every N_{FFT} clock periods, once the valid out signal from the FFT core is active. Since the system clock rate is Fs, then the equivalent phase calculation clock rate is $Fs/N_{FFT.}$ Given that this is an integer multiple of the system clock, it should be no problem for System Generator to implement. A note should be made, however, that the peak search method presented here is rather simple. A more sophisticated method, such as a quadratic fit over the sinusoidal spectral peaks, would no doubt be necessary in the presence of noise. Of course, such a mathematical implementation would require a significant amount of additional resources. As can be seen from Figure 6-11, the Cordic core uses up a considerable amount of slice resources.

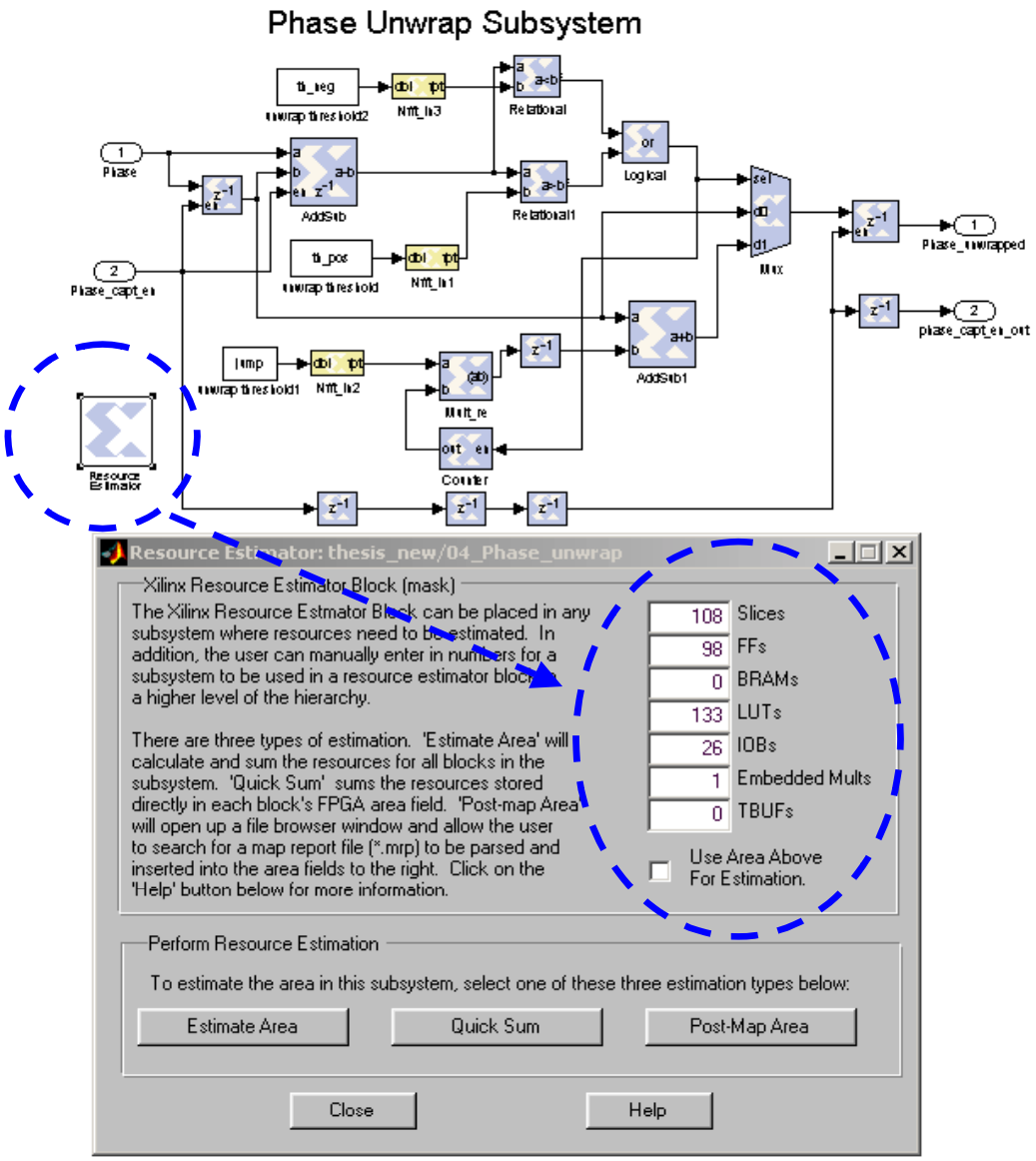


Figure 6-12 : Phase Unwrap Subsystem of PBFE Algorithm

Figure 6-12 illustrates the final subsystem that was completed, the phase unwrap. This subsystem takes the calculated peak phases and phase capture enable from the calculate peak phase subsystem as inputs. Then, the phase unwrapping scheme described

in chapter 4.4 and depicted in Figure 4-11 is implemented. The only notable difference is that the negative and positive search paths for the +π and -π constant thresholds must be handled separately. The jump constant is equal to 2π, as the theory of Figure 4-11 indicates. Note that the phase capture enable signal, at a rate of Fs/N_{FFT} Hz becomes the driving control signal for the system, by simply applying the appropriate amount of pipelining. As Figure 6-12 indicates, the phase unwrap subsystem uses up minimal

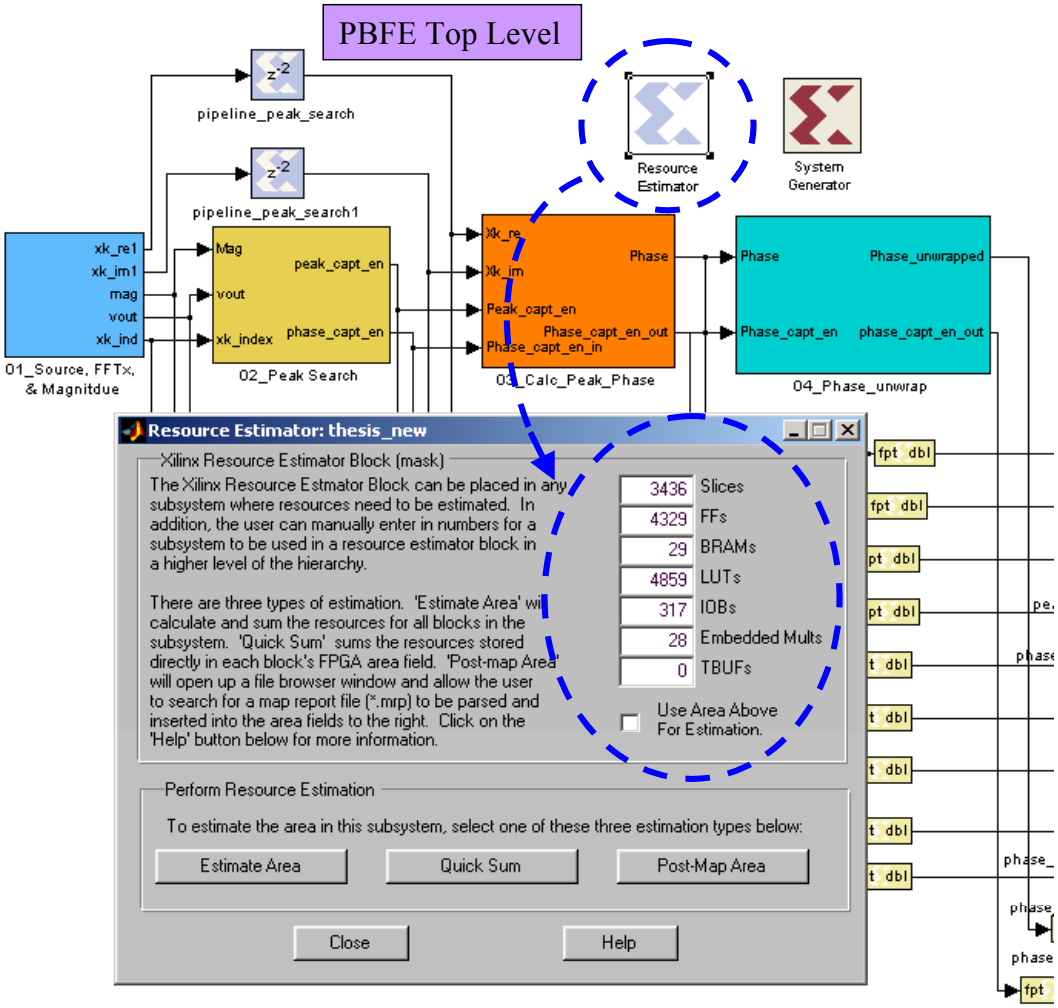


Figure 6-13 : Resource Utilization of PBFE Algorithm

resources compared to that of the FFT front-end, however note that one additional embedded multiplier has been used.

Figure 6-13 illustrates the total resources utilized by the incomplete PBFE System Generator implementation. The embedded Multiplier and Block RAM status is essentially the same as that of the FFT front-end subsystem utilization. Roughly 70% of the slices have been utilized, 44% of the registers, and 50% of the LUT's have been used. Again, the only steps of the Matlab algorithm of Figure 4-14 that were not completed are the least squares fit of the phases, the calculation of the slope, and the algebra of Equation 11. Therefore, with the current FPGA utilization, there should be no issue fitting the entire phase-based frequency estimation algorithm into XC2VP7 FPGA device and meeting timing at the 33 MHz clock rate.

7. SOC Implementation of DSP Algorithm

Recall the System on a Chip (SOC) discussion from chapter 2.6. Xilinx FPGA's provide a terrific re-programmable SOC platform for the analysis of hardware and software tradeoffs within an algorithm. System Generator for DSP can be used to help design, debug, and optimize DSP algorithms targeted for an SOC application. For example, in the phased-based frequency estimation flow chart of Figure 4-14, there are clearly some steps that may be more efficiently implemented in hardware, while other steps that may be more efficiently implemented with software. If the FFT algorithm is to be implemented in a strictly parallel manner, then it will without a doubt execute faster in hardware. However, steps such as phase unwrap and least squares fit may actually be better suited to execute on a processor, since they require floating point comparisons.

The one draw back to the 405 PPC in the Xilinx chips is that it is *not* a floating-point processor. Therefore, complete software/hardware tradeoffs of DSP algorithms in the Xilinx SOC environment are not fully realizable. There may always be trade off gains between hardware and software, however, the non floating-point processor is a hindrance. Floating point operations could be executed on the 405 PPC in the same manner that they are in hardware; that is, they could be converted to fixed-point and then finally to an integer vector representation of fixed-point. However, the same issue of quantization and overflow returns, leaving this option no better than implementing in the design completely in hardware. One solution might be to create user defined IP core that connects to the Processor Local Bus (PLB) and handles floating-point operations. In the

near future, it is expected that Xilinx will replace the PPC core with a float-point processor. Once this occurs, then Xilinx FPGA's will have an ideal platform to study tradeoffs between hardware and software implementation of DSP and other like algorithms.

Once the initial analysis is made on which portions of an algorithm may be better suited for hardware, the System Generator can be utilized to design, implement, and verify the hardware core functionality before it is ever put into the SOC architecture. In this manner, the functionality and timing of the main guts of the core are already verified before the SOC tradeoff study begins. The only logic that must be added are the PLB and DCR (see chapter 2.6) controls to interface with IBM core connect architecture. Further integration of the System Generator and the SOC design tools is currently in progress.

8. Conclusion

FPGA's offer mass parallelism for implementing DSP algorithms, specifically for implementing multiply and accumulate functions. In any application that requires real time processing, such as electronic warfare applications, parallel MAC implementations are needed to speed up the hardware. The Xilinx Virtex-II Pro FPGA architecture provides many system resources for automatic or implicit implementation of DSP MAC functions. Today's DSP and FPGA designers have an increasing need to begin and end their design flow in Matlab. The current approach of manually converting software to HDL, implementing it on an FPGA, and then attempting to compare the hardware results to the software results, must cross many error prone boundaries. In order to assure equivalency between a DSP algorithm designed in Matlab and one implemented in hardware, it is useful to have a high level, single flow, software tool. The Xilinx System Generator for DSP, which is essentially a hardware add on to Simulink in Matlab, offers a unique solution. System Generator allows for DSP algorithm design and hardware implementation to be executed in essentially the same step.

The advantage to System Generator is that a designer has the ability to very quickly analyze the key issues that cause errors in translating a design from Matlab to an FPGA. These key issues are quantization and overflow. A designer has the ability to override an entire design with doubles, and then systematically turn on the fixed-point conversion to each subsystem or even design block until the quantization or overflow error is found. Once fixed, it can then be immediately tested in hardware. This concept is an extremely powerful feature that is simply not attainable with manual conversion of

algorithms. Further, many features, such as the best type of overflow or quantization technique to use, analysis of binary points, and the effects of scaling in FFT's can also be quickly analyzed.

System Generator does have a few drawbacks, however. As mentioned above, there is a penalty to system level abstraction: it may not always yield the best area utilization compared to a HDL design. Although System Generator handles multiple clock domains from a DSP implementation (i.e. multi-rate systems), it is not well suited for general multiple clock domains, especially those that are asynchronous. Further, many of today's DSP designs for electronic warfare applications require the resources of several of the largest FPGA's currently available, not just one chip. System Generator is not specifically designed to handle such large-scale systems. However, each subsystem in the Simulink design can be targeted to a different FPGA and individually verified in hardware, while the rest of the system is implemented in Simulink. So there are ways around the some of the limitations.

Unfortunately, the phase-based frequency estimation (PBFE) algorithm was never completely implemented in hardware. The real advantage of implementing such an algorithm in System Generator for DSP was never realized. Once the basic noiseless algorithm was verified, the main idea would have been to apply a signal with noise, and then experiment with all of the key advantages listed above, such as quantization, overflow, binary point, scaling, etc. However, the FFT front-end subsystem that was implemented successfully on the FPGA hardware served as a perfect research vehicle to explore the advantages and disadvantages of System Generator for DSP. From an SOC application, the PBFE algorithm is an ideal candidate, for two reasons. One, the

algorithm has easily identifiable steps that can be either implemented in hardware or software for trade off analysis. Second, all of the steps in the PBFE algorithm can be implemented with System Generator to easily verify the functionality and timing before the logic is ever placed into the more complex SOC architecture.

There are many avenues of future research that can continue from this work. For example, the although it was shown in Equation 11of chapter 4.4 that the PBFE algorithm is bounded, there are further theoretical (mathematical) DSP "tricks" that can be played to expand the capability of this algorithm. Also, it was mentioned in chapter 4.4 that part of the theory behind the frequency estimation limit curve comes from DFT-based filter bank analysis. This entire system could be implemented with filter banks and a decimation scheme (multi-rate). Lastly, one could continue this research work in the realm of re-programmable SOC (i.e. SOC' s on Xilinx FPGA's rather than ASIC's) to explore hardware and software tradeoffs, using the System Generator to verify the cores to be implemented.

The research presented here covers a vast amount of FPGA background material, design skills, and theoretical knowledge. The Xilinx System Generator for DSP in general aids in converging the gap that once existed between system level DSP architects and hardware designers. The System Generator forces any designer that comes in contact with it to dramatically broadens his/her knowledge and skill set. Some have claimed that System Generator is a tool that reduces the number of engineers needed to design and verify a complete DSP system. However, I proclaim that the System Generator for DSP offers a high level, single design flow that can increases the throughput of DSP designs within a company, while at the same time assuring equivalency between software and

hardware; that is, nearly 100% equivalency between a Matlab algorithm and the same algorithm implemented in hardware on an FPGA.

9. References

[1] J.L. Hennnesy & D.A. Pattterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufmann Publishers, 2003.

[2] Xilinx University Program, Presentation Material & lab files, *2003 Digital Signal Processing with FPGA's Workshop,* Online. Available: http://www.xilinx.com/univ.

[3] Dr. Mark Fowler, "Digital Signal Processing Class Notes", *ECE Department, Binghamton University, NY,* Fall 2002.

[4] 2003 Digital Signal Processing with FPGAs Workshop, *Xilinx University Program, Presentation Material & lab files,* Online. Available: http://www.xilinx.com/univ

[5] C.A.Wisknesky, "Analysis of Xilinx FPGA Architecture and FPGA Test: A Basis for FPGA Enhanced DSP Algorithmic Acceleration and Development in Matlab/Simulink via Xilinx System Generator," *Binghamton University, State University of New York, Master Thesis*, 2004

[6] M. Fallat, "Virtex-II Technical Design Solutions", *Xilinx FAE Presentation,* 2002.

[7] M. Fallat, "Platform for Programmable Logic", *Xilinx FAE Presentation,* 2003.

[8] Virtex-II Pro and Virtex-II Pro X Platform FPGA's : Complete Data Sheet. Online. Available: http://www.xilinx.com

[9] Virtex-II Platform FPGA's : Complete Data Sheet. Online. Available: http://www.xilinx.com

[10] Andraka Consulting Group, Inc, *Multiplication in FPGA's.* Online. Available: http://www.andraka.com/dsp.htm

[11] The Design of an SRAM-Based Field Programmable Gate Array – Part I: Architecture. Online. Available: http://www.eecg.toronto.edu/~pc/research/publications/lego1.tvlsi99.pdf.

[12] B.P. Lathi, *Signal Processing and Linear Systems*, Berkley-Cambridge Press, 1998.

[13] B. Porat, *A Course In Digital Signal Processing*, John Wiley & Sons, Inc., 1997.

10. Appendix

10.1 Matlab Code for Phase-Based Frequency Estimation Algorithm

```
%%%%%%%%%%%%%%%%%%%%%%%%
% Kurt D. Rogers
% Copy Right 2004
% Thesis Algorithm
% Phase-Based Frequency Estimation Comparison Model
%       - Experiment with Phase Unwrapping
%       - Frequency Estimation Limit Curve
%%%%%%%%%%%%%%%%%%%%%%%%

clear
close all
% clc

% --- Change these variables only ---
Fs = 51200; % Hz
Nfft = 512; % length of FFT
    % - results in Bin = 51200/512 = 100 Hz.
Fo1 = 1213; % Hz %14137
%Fo2 = 19183; % Hz
Fo2 = 0; % Hz (stick to 1 freq for now)
phi1 = 0; %radians
phi2 = pi/2; %radians (set to pi/2 so second signal has result of 0)
wind_type = 1; % 0 for rectangle, 1 for han, 2 for ham
N_shift = 16; % samples
    % note: use N - Nfft)/N_shift = integer, solve for N_shift to find a
    % starting value that comes at to the end of the vector  (12 works with
    % these numbers)
N_blocks = 10; % # of FFT's blocks to take data of
plot_matrix = [0; % FFT plot
               0; % raw phase plot
               0; % unwrapped phase plot
               0];% linear phase plot
% ------------------------------------

%--- Create a test vector---
Ts = 1/Fs;
t = 0:Ts:1-Ts; % create 1 seconds worth of data --> # of samples = Fs
y = cos(2*pi*Fo1*t + phi1) + cos(2*pi*Fo2*t + phi2);
```

```
% --- Obtain Freq transform ----------
freq = -Fs/2:Fs/Nfft:Fs/2 - Fs/Nfft;
%for (i = 1:N_shift:(length(y) - Nfft)) % use this to end up exactly at end of tets vector
for (i = 1:N_shift:N_shift*(N_blocks-1)+1)
    if (wind_type == 0)
        y_w = y(i:i + Nfft -1);
    elseif (wind_type == 1)
        y_w = y(i:i + Nfft -1).*(hanning(Nfft).');
    else
        y_w = y(i:i + Nfft -1).*(hammning(Nfft).');
    end

    Y((i-1)/N_shift +1, :) = fftshift(fft(y_w, Nfft));
        % note: don't do magnitude here like normal, since need to find
        % phase later.
end

% --- Plot the output of the N_block FFT's ---
% create colors to loop on
colors = [ 'b  '; 'g  '; 'r  '; 'c  '; 'm  '; 'y  '; 'k  '; 'b--'; 'g--'; 'r--'; 'c--'; 'm--'; 'y--'; 'k--' ];
if (plot_matrix(1))
      figure(1)
    for (j = 1:length(Y(:,1)))
        subplot(2,1,1)
        plot(freq, abs(Y(j,:)), colors(j,:) )
        hold on
        subplot(2,1,2)
        plot(freq, angle(Y(j,:)), colors(j,:) )
        hold on
    end
    subplot(2,1,1)
    ylabel('Magnitude')
    title('DFT of each of 10 blocks')
    subplot(2,1,2)
    xlabel('Freq (Hz)')
    ylabel('Phase')

end
```

```
% --- Find magnitude, peak, & phase ---
% get magnitude

for (k = 1:length(Y(:,1)))
    Y_mag(k,:) = abs(Y(k,:));
end

% Find peak index and Real & Imag vales of peak

% % for (i = 1:length(Y_mag(:,1)))
% %     for (j = 2:length(Y_mag(1,:)) - 1)
% %         if ( (Y_mag(i,j-1) < Y_mag(i,j)) & (Y_mag(i,j) > Y_mag(i,j+1)) )
% %             index_peak(i) = j;
% %             peak(i) = Y(i,index_peak(i));  % note this indexes Y, not Y_mag to get R & Im
% %             peak_phase(i) = angle(peak(i));
% %         end
% %     end
% % end

f_z=(Nfft/2)+1;   % index of DFT's zero frequency bin
for (i = 1:length(Y_mag(:,1)))
    [Ymax,index]=max(Y_mag(i,f_z:end));   % find max over only positive frequencies
          %%% if you switch to complex sinusoids you'll have to find max over ALL bins
    index_peak(i)=index+Nfft/2;
    peak(i)=Y(i,index_peak(i));
    peak_phase(i)=angle(peak(i));
end


% --- Phase unwrap ---------------------------------

%create plotting index for phase (function of FFT block number)
block_index = 1:length(peak_phase);

% plot raw phase
if (plot_matrix(2))
    figure(2)
    plot(block_index, peak_phase/pi, '-+') % normalized to Pi.
    title('Raw Phase')
end
```

```
% unwrap the phase
peak_phase_unw = unwrap(peak_phase);

% plot the unwrapped phase
if (plot_matrix(3))
    figure(3)
    plot(block_index, peak_phase_unw/pi, '-+')
    title('Unwrapped Phase')
end

% do least squares fit on unwrapped phase
p = polyfit(block_index, peak_phase_unw, 1);

% create new linear phase
phi_b = p(1)*block_index + p(2);

% plot the linear phase
if (plot_matrix(4))
    figure(4)
    plot(block_index, phi_b, '-+')
    title('Linear Phase')
end

% calculate the frequency in radians
theta = abs( p(1)/N_shift );

% calculate the frequency in Hz.
f = theta*Fs/(2*pi)
```

10.2 Matlab FFT Front-End Script For System Generator Execution

```
% ------------------------------------------------------------------------
% FFT_front_end_script.m
%
% This script is the FFT front end subsystem to the phase-based frequency
% estimation algorithm.  It creates a real, single frequency sinusoidal
% test vector. It takes B, overlapping Nfft-ponint FFT blocks, based on the
% the user defined input paramters. The results are subplotted for
% comparison with an equuivalent Simulink design. The Simulink design is
% invoked and given a simulaiton run time basd on the user defined
% parameters and pretermined delay through the FFT core (see Simulink
% design). The results of the Simulink design are then read back,
% reformatted, and plotted on the subplot for comparison.
%
%Eq 1: Resolution = Fs/Nfft
%Eq 2: Freq limit curve --> abs(Fo) < Fs/(2*Ns)
%
% Knowns : Fo, Nfft, Ns
% Solve for Fs
% ------------------------------------------------------------------------

clear
close all
clc

% ------------------ Change these variables only ----------------------
Fol = 2000;
Nfft = 512; % length of FFT
Ns = 16;

phil = 0; %radians
A =1; % Amplitude
B = 10; % # off FFT blocks to take
cosim = 0; % set this to one to run hardware co-sim, 0 for normal software sim
inp_len = 1024; % input length (Max needed is 16*10+512 = 672.
th_pos = pi;
th_neg = -pi;
jump = 2*pi;
clock = 33000000; %Hz
% ---------------------------------------------------------------------
```

```
% ------------------------------------------------------------------------
% Create a test Vector & Matlab Comparison Plot
% ------------------------------------------------------------------------

% Calculate Actual Fs based on Freq Est Limit Curve
Fs = 2*Ns*Fol;

% Caclulate Spectral resolution
res = Fs/Nfft;

% --- Create Test Vector ---
Ts = 1/Fs;
t = 0:Ts:(Nfft -1)*Ts; % Create Nfft samples
y = A*cos(2*pi*Fol*t + phil);

% --- Create Matlab version of DFT for comparison ---
freq = -Fs/2:Fs/Nfft:Fs/2 - Fs/Nfft;
Y = fftshift(fft(y, Nfft));
figure(1)
subplot(3,1,1)
plot(freq, abs(Y), '-o')
%xlabel('freq (Hz)')
ylabel('Mag')
title(['DFT of ', num2str(Fol), ' Hz Sinusoid - Matlab'])
AXIS([-Fs/8 Fs/8 0 max(abs(Y))])
```

```
% ------------------ Invoke Simulink Design -------------------------------
% Comment from here to the end in order to NOT run Simulink file from the
% script --> i.e. it will just set up the user defined varibles.
% -------------------------------------------------------------------------

 if (Nfft == 64)
     if (cosim == 1)
         sim('fft_front_end_64_RAM_cosim', ( Nfft*Ts*(9/4) + B*Nfft*Ts + Nfft/2*Ts ));
     else
        % must add about .2 sec to fill up RAM now
         sim('fft_front_end_64_RAM', ( Nfft*Ts*(9/4) + B*Nfft*Ts + Nfft/2*Ts  + .5));
    end
elseif (Nfft == 512)
     if (cosim == 1)
         sim('fft_front_end_512_RAM_cosim', ( Nfft*Ts*(9/4) + B*Nfft*Ts + Nfft/2*Ts ));
     else
%          sim('fft_front_end_512', ( Nfft*Ts*(9/4) + B*Nfft*Ts + Nfft/2*Ts ));
         sim('fft_front_end_512_RAM', ( Nfft*Ts*(9/4) + B*Nfft*Ts + Nfft/2*Ts  + .5));
     end
 end

% -------------------------------------------------------------------------
% Read back "B" results from the Simulink (from Workspace)
% -------------------------------------------------------------------------

for (j = 1:B)
    Y2(j,:) = (fftshift(y_re(1+(j-1)*Nfft:Nfft*j) + i*y_im(1+(j-1)*Nfft:Nfft*j))).';
                                    % NOTE: must use .' here, since array is complex
%     Y2(j,:) = (y_re(1+(j-1)*Nfft:Nfft*j) + i*y_im(1+(j-1)*Nfft:Nfft*j)).';
    Y_pwr2(j,:) = (fftshift(y_pwr(1+(j-1)*Nfft:Nfft*j)))';
end
```

```
% ------------------------------------------------------------------------
% Plot "B" Simulink Results for Comparison
% ------------------------------------------------------------------------

colors = [ 'b  '; 'g  '; 'r  '; 'c  '; 'm  '; 'y  '; 'k  '; 'b--'; 'g--'; 'r--'; 'c--'; 'm--'; 'y--'; 'k--' ];
for (j = 1:length(Y2(:,1)))
    subplot(3,1,2)
    hold on
    plot(freq, abs(Y2(j,:)), colors(j,:) )
    subplot(3,1,3)
    hold on
    plot(freq, Y_pwr2(j,:), colors(j,:) )
end
subplot(3,1,2)
ylabel('Mag')
title(['DFT of ', num2str(Fol), ' Hz Sinusoid - Simulink'])
AXIS([ -Fs/8 Fs/8 0 max(abs( Y2(1,:) ) ) ])
subplot(3,1,3)
xlabel('freq (Hz)')
ylabel('PWR')
title(['PWR of ' , num2str(Fol), ' Hz Sinusoid - Simulink'])
AXIS([-Fs/8 Fs/8 0 max(Y_pwr2(1,:))])

% ------------------------------------------------------------------------
```