Summer 2011

WvFEv3, An FPGA-based general purpose digital signal processor for space applications

Brian Thomas Mokrzycki
University of Iowa

WvFEv3: AN FPGA-BASED GENERAL PURPOSE DIGITAL SIGNAL PROCESSOR FOR SPACE APPLICATIONS

by

Brian Thomas Mokrzycki

A thesis submitted in partial fulfillment
of the requirements for the Master of
Science degree in Electrical and Computer Engineering
in the Graduate College of
The University of Iowa

July 2011

Thesis Supervisors: Professor Thomas L. Casavant
Professor Jon G. Kuhl

Graduate College
The University of Iowa
Iowa City, Iowa

CERTIFICATE OF APPROVAL

MASTER'S THESIS

This is to certify that the Master's thesis of

Brian Thomas Mokrzycki

has been approved by the Examining Committee
for the thesis requirement for the Master of Science
degree in Electrical and Computer Engineering at the July 2011 graduation.

Thesis Committee: _____
Thomas L. Casavant, Thesis Supervisor

_____
Jon G. Kuhl, Thesis Supervisor

_____
William S. Kurth

To my Dad – If you had not been encouraging me to constantly test the boundaries of my capabilities I would have never become the person I am today. Thank you.

To my Mom – Your loving support and kind demeanor will continue to be sorely missed. Rest in peace.

I am now able to say “I’m finally finished”

Jorge Cham
Piled Higher and Deeper - ”Happenings outside”

ACKNOWLEDGMENTS

I have to thank my parents, who have always supported me, provided wisdom, and helped me overcome numerous obstacles throughout my life. I would have never been able to achieve what I have without you. From the very bottom of my heart, thank you.

I'd like to thank the entire UI Radio and Plasma Wave group. It's been six years since they offered me a wonderful opportunity to join the team and build space hardware destined for Jupiter (has it been that long?). Now I can see why you all laughed at me when I said "Oh, that's plenty of time."

Specifically I would like to thank William Robison, Terry Averkamp, William Kurth, and Don Kirchner of the Radio and Plasma Wave group. The last six years would have been far more difficult without you. Your confidence, wisdom, guidance, and example have taught me far more than you realize.

To Jaimee Carpenter, who has been encouraging me to finish my thesis for years. Your efforts to keep me focused, especially during these final months, have been crucial and have not gone unnoticed.

Bio::Neos Inc. founders Mike Smith and Steve Davis. Your friendship and guidance over the years has been vital. I respect you both very much and wish you a happy and fulfilling life.

I would also like to thank Dr. Thomas L. Casavant and Dr. Jon Kuhl for their support, feedback and input. Tom, you saw promise in me, treated me as a friend and stuck with me through all these years, thank you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

CHAPTER I

INTRODUCTION AND BACKGROUND

Introduction

The *Waves* instruments aboard the *Juno* and *Radiation Belt Storm Probe (RBSP)* spacecraft represents the next generation of space radio and plasma wave instrumentation developed by the University of Iowa's Radio and Plasma Wave group [1, 2]. The previous generation of such instruments on the *Cassini* [3] spacecraft utilized several analog signal-conditioning techniques to compress and condense scientific data. Compression techniques are necessary because the plasma wave instruments can often generate significantly more science data than can be transmitted using the narrow telemetry channel of the hosting spacecraft. The next generation of plasma wave instrumentation represents a major shift of analog signal conditioning functionality to the digital domain, drastically reducing the amount of power and mass required by the instrument while simultaneously further condensing scientific data, increasing the precision of plasma emission measurements, and adding flexibility.

The digital transition of *Waves* instruments relies heavily on available integrated circuit technologies capable of performing signal processing tasks in real time. Performance is not the only consideration, however, as the digital system must also operate in a space environment with no atmosphere, wide temperature variations, and radiation exposure for the lifetime of the mission. Architecturally speaking, the ideal solution would also be flexible enough to implement a wide variety of digital signal processing techniques for changing scenarios during space flight with the additional benefit of potentially using such a system in missions beyond *Juno* and *RBSP*.

An attractive solution to these goals is the use of a general-purpose digital signal processor that combines the programmatic approach of a traditional central processing

unit (CPU) with optimized circuitry/instruction set for signal processing. This approach has been used numerous times before in commercial products available in the consumer, military, automotive, and industrial sectors [4-7]. However, due to the rigors of space flight and qualifications set forth by the National Aeronautics and Space Administration (NASA) [8], these solutions are not adequate for reliable operation in a space environment. This has resulted in a lack of adequate options to address the needs of the radio and plasma wave instruments aboard the *Juno* and *RBSP* spacecraft.

The solution presented in this thesis is to utilize a low-cost radiation tolerant field programmable gate array (FPGA) that serves as a space qualified implementation platform for a custom designed general-purpose digital signal processor, called the *WvFEv3*. The design of the *WvFEv3* processor is unique among traditional FPGA implementations due to its generic processing flow, thus allowing a wide variety of algorithms to be implemented programmatically without the need to reprogram the FPGA during a mission. This approach addresses the performance and flexibility needs of the *Waves* instruments in its continuing goals to reduce mass and power while simultaneously increasing the precision and compression ratios of science products.

The realized *WvFEv3* processor has met and surpassed the requirements of the *Waves* instruments and now resides aboard the *Juno* and *RBSP* spacecraft, awaiting launch to their respective destinations.

Waves Instrument

The Physics and Astronomy Department at the University of Iowa (UI) has been studying naturally occurring magnetospheric phenomena for over fifty years. Space borne research first began at the dawn of the space race with instrumentation aboard the United State's first satellite, *Explorer I*, developed by Dr. James Van Allen and his team at the

UI. The 18-pound satellite was responsible for the first major discovery of the Space Age, the Van Allen radiation belts [9]. These torus regions of energetic particles are the result of the Earth's stable magnetosphere trapping charged particles in two distinct belts of radiation. The inner radiation belt is composed of high concentrations of energetic protons that are believed to be the result of beta decay of neutrons created by cosmic ray collisions with atoms in the upper atmosphere [10]. The outer radiation belt is believed to be largely made up of electrons produced by inward radial diffusion and local accelerations caused by the energy transfer of whistler mode plasma waves to radiation belt electrons [11]. Both of these dynamic regions of space produce large exposures of radiation that could impact space systems and the health of humans traveling through them.

One future mission, the *Radiation Belt Storm Probes (RBSP)* [12] will continue the investigation of the radiation belts to better understand the Earth's magnetosphere, processes that generate hazardous space weather and how these processes could impact space travel. *RBSP* is part of NASA's *Living with a Star* program and is composed of twin satellites that will travel along an elliptical orbit that repeatedly intersects both radiation belts throughout their mission. One instrument aiding this investigation is the UI's *Waves* instrument as part of the *Electric and Magnetic Field Instrument Suite (EMFISIS)* [2]. The *Waves* instrument will assistant in the science goals of *RBSP* by providing measurements to better understand plasma wave origin.

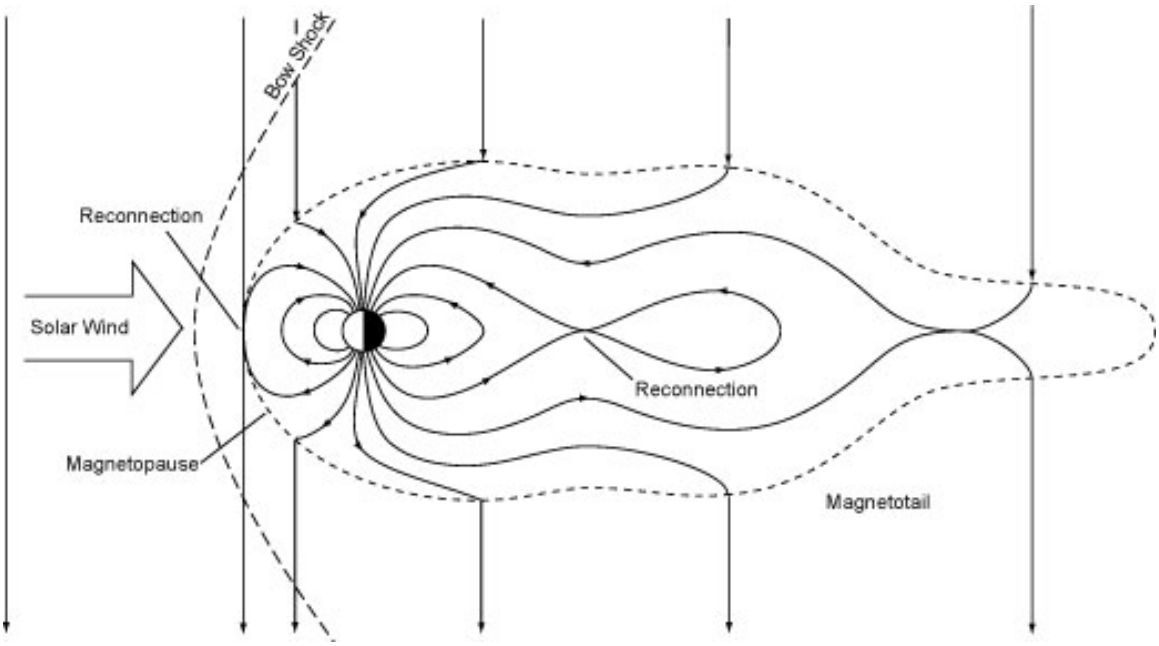


Figure 1-1: Simple illustration of the magnetosphere and a geomagnetic tail reconnection event.

The radiation belts are not the only natural phenomenon occurring as the result of plasmas trapped by the magnetosphere. The *aurora borealis*, more commonly known as the northern lights, are thought by some to be the result of geomagnetic tail reconnection events that accelerate particles along magnetic field lines toward the magnetic poles of Earth. In the case when the accelerated particles have sufficient velocity parallel to the magnetic field, the physical location of the magnetic reflection point is pushed into the ionosphere resulting in the bombardment of the ionosphere's gas molecules with charged particles. The result is the emission of light from these molecules in an assortment of colors known as the *aurora borealis*. This effect has also been observed at several other planets in the solar system including Jupiter and Saturn.

Another future NASA mission, *Juno*, will provide the first possible direct measurements of these auroral regions at Jupiter. After its launch in 2011, the *Juno* spacecraft will cruise to Jupiter over a five-year period and reach orbit along a highly

elliptical orbital path that will pass directly over both poles of the planet. Its primary science objective is to characterize the formation and internal structure of the planet through the measurement of multiple natural phenomenon [13]. The *Waves* instrument will aid in this objective by providing radio and plasma wave measurements of the auroras and other plasma wave features, allowing for a better characterization of Jupiter and its polar magnetosphere [1].

Measuring electromagnetic emissions in space plasmas over a wide range of frequencies allows for an understanding of the basic properties of the plasmas and perturbations within them that occur as a result of a wide range of interactions with the charged particles comprising the plasma. For example, measuring emissions at two fundamental frequencies, the electron plasma frequency and the electron cyclotron frequency, provides information about the density of the plasma and the magnetic field in which it is embedded. Measurements of these fundamental frequencies as well as emissions at other frequencies are received and conditioned by the radio and plasma wave instrument through on-board antennas and receivers. In recent implementations of the instrument, the signal is compressed by either transforming the signal to a coarse spectrum using analog circuitry or by compressing a digitized representation of the measured waveform. In either case, compression is a necessary operation due to the high data volume the instrument can generate relative to the narrow telemetry link of the spacecraft.

The next generation of radio and plasma instrumentation aboard the *Juno* and *RBSP* spacecraft, known as the *Waves* instrument in each instance, represents a significant shift of signal processing functionality towards the digital domain when compared to previous incarnations found on *Voyager 1 & 2, Geotail,* and *Cassini* [3, 14-16]. One of the most effective methods for reducing the data requirements for transmitting science data is through spectral analysis. Previous incarnations achieved this by utilizing analog circuitry to transform the received signal into a coarse power

spectrum. The same result can be achieved in the digital domain by utilizing a digital processor and the Fourier transform. Several benefits are realized when moving to a digital solution; mass and power needs of the instrument would be reduced while simultaneously increasing the precision of the resultant compressed spectrum. Furthermore, a digital processing solution can also be utilized to implement further signal processing techniques such as binning and averaging of multiple spectrums to further compress science data.

The cornerstone of the analog-to-digital processing transition is based upon the well-understood time to frequency domain transform called the Fourier transform. The *Waves* instrument will make heavy use of this operation and any performance advantages among implementations is highly beneficial. One of the most commonly used Fourier transform implementations found in digital systems is the radix-4 complex FFT or fast Fourier transform which extensively exploits the use of common partial products between point computations to drastically reduce the number of arithmetic operations. A radix-4 complex FFT dataset does make an assumption about the size of the input vector where it must conform to a base-four number, i.e. 4, 16, 64, 256, etc.

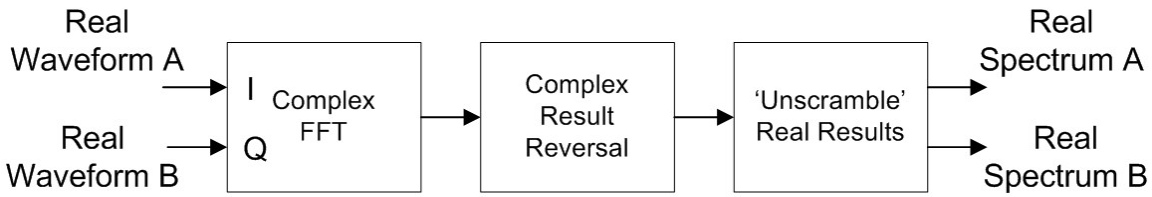


Figure 1-2: Dual real FFT processing flow

To further reduce arithmetic complexity, a second assumption can be made if the signal input is real only, allowing for the simultaneous transformation of two real signals with a single complex FFT operation. In this case the performance of the real FFT operation can

be nearly doubled by modifying the input of the complex FFT operation to include two real signals, one in the real part and the other in the complex part of the input, seen in Figure 1-2. Once the complex FFT operation is complete, a post-processing step is required to 'unscramble' the real results from one another [17]. The operations needed to perform signal-processing techniques in the digital domain are all well understood; the only requirement is a digital processing system capable of performing these operations.

Field Programmable Gate Arrays

Space qualified integrated circuits provide a key technology for enabling the transition of functionality from the analog domain to the digital domain by providing high-speed logic components for implementing digital processing systems. The specific nature of the space environment exposes integrated circuits to non-traditional environmental extremes in temperature, vacuum, and radiation exposure. High-energy ions or electro-magnetic radiation striking a sensitive node in an integrated circuit, such as a flip-flop or memory cell can result in a state change of the logic element called a single event upset (SEU). SEUs can cause a processor to exhibit peculiar behavior, as the state of operation is no longer valid leading to unpredictable actions. One part of qualifying an integrated circuit for spaceflight requires the design to be capable of mitigating SEUs through gate design and tolerating SEU effects through logic design.

Field programmable gate arrays (FPGAs) provide one possible technique for implementing integrated circuit designs for spaceflight. FPGAs are a flexible integrated circuit containing programmable logic components and a hierarchy of reconfigurable interconnects allowing the logic components to be connected in a wide variety of configurations. The programmable 'sea of gates' allow a designer to implement anything from a simple logic circuit, to a sequential circuit, to more complex functions such as a

microprocessor. A radiation tolerant FPGA incorporates additional features into the gate design of the logic components to detect and mitigate SEUs asynchronously [18].

The design and implementation of an FPGA circuit begins with the definition of the intended function in a high-level hardware description language (HDL) such as VHDL or Verilog. The HDL provides a definition of the functionality intended, whether it's a bus interface, a state machine, or a processor pipeline. The HDL by itself is only useful for performing a high-level simulation of the intended design and must be transformed several times before it's loadable into an FPGA device. The first step, called synthesis, is the transformation of high-level constructs in HDL to specific resources the FPGA provides. These can either be basic logic gates such as AND, OR, and XOR gates or higher-level structures such as 1-bit adders or 2-input multiplexers.

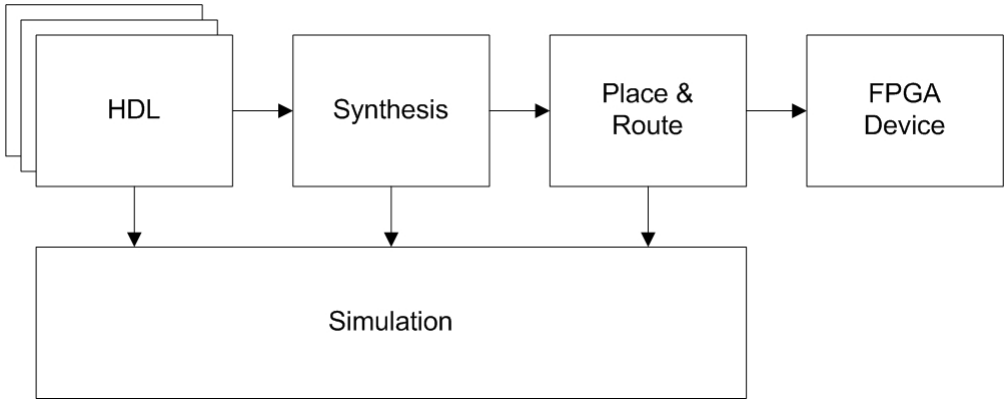


Figure 1-3: FPGA Design flow methodology

Additionally, the synthesis step also connects these resources together to form a net-list that is logically equivalent to the high-level HDL description. The final step of the design flow concludes with the 'place and route' of the synthesized net-list. Each resource defined in the net-list is given a location on the FPGA die and then connected together using the FPGA routing infrastructure. Once this process is complete a programming file

representing the original HDL design is generated and can be used to program the specific FPGA family and model selected during the design flow.

FPGA designs are typically referred to as intellectual property cores (IP cores), which are further sub-categorized into three groups, soft IP cores, firm IP cores, and hard IP cores. A soft IP core is a logic design that is described as HDL source and still needs to run throughout the entire design flow before being FPGA loadable. A soft IP core represents the most flexible description of the design as it can be targeted for a wide variety of FPGA devices; this is analogous with high-level software source code. In the next level down, the firm IP core is represented by the net-list generated post synthesis. This definition can be implemented in any device in one FPGA family; this would be comparable with assembly code for a particular processor family. Finally, a hard IP core is the product created post 'place and route' and represents a hardwired implementation of the design that is specific to a particular FPGA family and model. The hard IP core representation is most similar to a pre-compiled software executable where it is very difficult to reverse engineer the design.

The verification of an FPGA design can be done at several levels throughout the design flow. The lowest fidelity verification is called functional verification; this is performed when the high-level HDL description of the design is simulated in a logic simulator, such as ModelSim or Active-HDL. This level of verification only verifies that the HDL representation logically represents the function intended. Physical effects such as wire delays, pad delays and gate delays are not included at this level of simulation. To achieve this level of detail the design must first pass all the way through the design flow to 'place and route'. At this point enough information is known about the gates and interconnects that delays between various components can be calculated. Since this level of simulation most accurately represents the physical device it is called physical simulation. Once functionality has been verified in physical simulation a device may be programmed and unit level testing is used to continue the verification effort. The most

rigorous of these efforts is thermal vacuum testing where the device is expected to operate flawlessly while exposed to extreme temperature variations under vacuum.

CHAPTER II

PROBLEM STATEMENT

Digital signal processing (DSP) aboard the *Juno* and *RBSP* space exploration satellites presents a number of unique constraints upon performance, flexibility, and space qualification where no previously known solution is suitable. To resolve this issue a novel approach is taken; the design and implementation of a general-propose digital signal processor targeted for a radiation tolerant field programmable gate array (RT-FPGA).

Several commercial solutions exist for digital signal processing (DSP) applications with varying levels of performance and flexibility. The use of an off-the-shelf general-purpose central processing unit (CPU, provides a high-level of flexibility through the use of an instruction set to program a wide variety of application programs. Because such a CPU is general-purpose in nature, it can be used to implement a wide variety of application programs including DSP algorithms. An assortment of space qualified CPUs exist for both FPGAs and application specific integrated circuits (ASICs) [19, 20], however these CPUs typically have inadequate clock rates or instruction flow inefficiencies that limit DSP performance when compared to contemporary CPUs. As a result, they cannot meet the minimum performance requirements of the *Waves* instrument aboard the twin *RBSP* satellites where spectral structure is expected on 30 millisecond time scales. Due to this constraint, the DSP sub-systems in each of these instruments require a processor capable of calculating, at minimum, ninety-six real 1024-point Fourier transforms per second in single precision floating-point format, or one every 10.4 milliseconds. No known space qualified CPU is capable of a performance high enough to meet this requirement.

Application specific logic (ASL) blocks provide another commercial option that can meet the minimum performance requirement of *RBSP*. However, their relative inflexibility when compared to CPUs creates additional obstacles associated with integration and flight operations. A DSP ASL performs a specific DSP task quickly and efficiently through the use of a custom circuit inside an FPGA or ASIC [21]. Due to the highly specialized nature of ASLs they are incapable of performing any other operation than the specific one they were designed to perform. This typically is not an issue unless it's difficult or infeasible to simulate the exact operational environment in which the ASL is to be deployed, such as with satellites exploring uncharted regions of space. These unknowns may lead to a situation where the alteration a DSP operation(s) would be beneficial or even critical; for instance, adaptively cancelling solar panel switching noise. Although FPGAs are reprogrammable and could potentially alter DSP operations, the environment of space flight typically restricts this activity resulting in an ASL implementation that is rigid and static. This static property makes ASL implementations an unattractive solution for this application.

ASIC designers have already faced the issues of flexibility and provided a solution through the design of a general-purpose DSP processor (DPU). The DPU marries the programmatic control of a CPU with specialized instructions and circuits as seen in ASLs. This solution represents a compromise between performance and flexibility; a DPU sacrifices a portion of the flexible features of a CPU to gain additional DSP performance closer to that of an ASL. As seen in Figure 2-1, a vendor supplied DPU in an ASIC would provide the lowest cost, highest performance acceptable solution, but no space qualified part was available during the design phase for the missions supported by this thesis work. A DPU implemented in an FPGA would provide the next logical solution, but no development effort is known previously to this work that has addressed this possibility -- space qualified or otherwise.

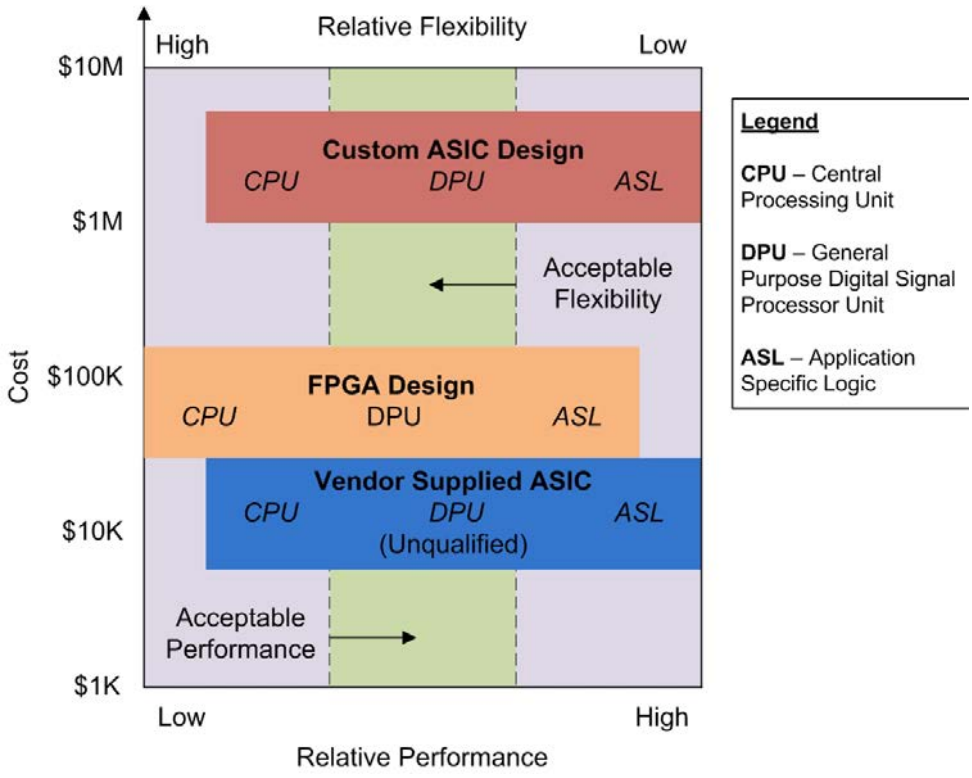


Figure 2-1: Relative distribution of DSP solutions based on cost, performance, and flexibility

The central issue is that at the time of design, no space qualified DSP solutions existed that were able to meet the performance and flexibility goals described. A novel, cost-conscious approach would be to utilize a radiation tolerant FPGA to implement a general-purpose DSP processor. This would provide a solution that concurrently addresses the constraints of performance, flexibility, and space-worthiness. Presented in this thesis is the *WvFEv3* processor, *a* general-purpose DSP processor targeted for Actel's RTAX2000 radiation tolerant FPGA.

CHAPTER III

METHODS

System Architecture

The complexity of autonomously managing tasks, telemetry, and system health in space systems is substantial. Often, the simplest solution is to utilize a general-purpose processor coupled with a real-time operating system to form the foundation of the system. This solution provides the flexibility to alter instrument operations during flight and a hardware interface to supplement the system with additional resources, such as memory devices, serial interfaces, data collection systems and co-processors. The *WvFE SoC co-processor* is one such example; it extends the host processor's capabilities with additional resources to perform data collection and DSP related tasks.

Seen in Figure 3-1, the *WvFE SoC co-processor* is not just the *WvFEv3* processor core itself, but a complete 'system on a chip' (SoC) comprised of several cores and interfaces. Storage for application code, waveforms, and processed products is provided via an independent local memory; architected in a manner to isolate the memory bandwidth needs of the co-processor from the host processor's bus. The host processor is able to read and write to the *WvFE SoC co-processors* internal configuration registers and the entirety of local memory via the host bridge. Finally, the analog-to-digital (A/D) controllers in the co-processor provide waveform-capturing capabilities on tight time boundaries.

Scaling of the system to provide additional computational resources is possible due to the bus architecture of A/D subsystem and the independent local memories for each co-processor. Figure 3-1 shows that the co-processor configuration can be replicated several times until a limitation is either found in the bandwidth of host processor bus or the A/D subsystem bus. In this configuration, the host processor interfaces to each co-

processor as a separate address mapped resource, and since all co-processors are identical any of them may be selected to perform the needed operations.

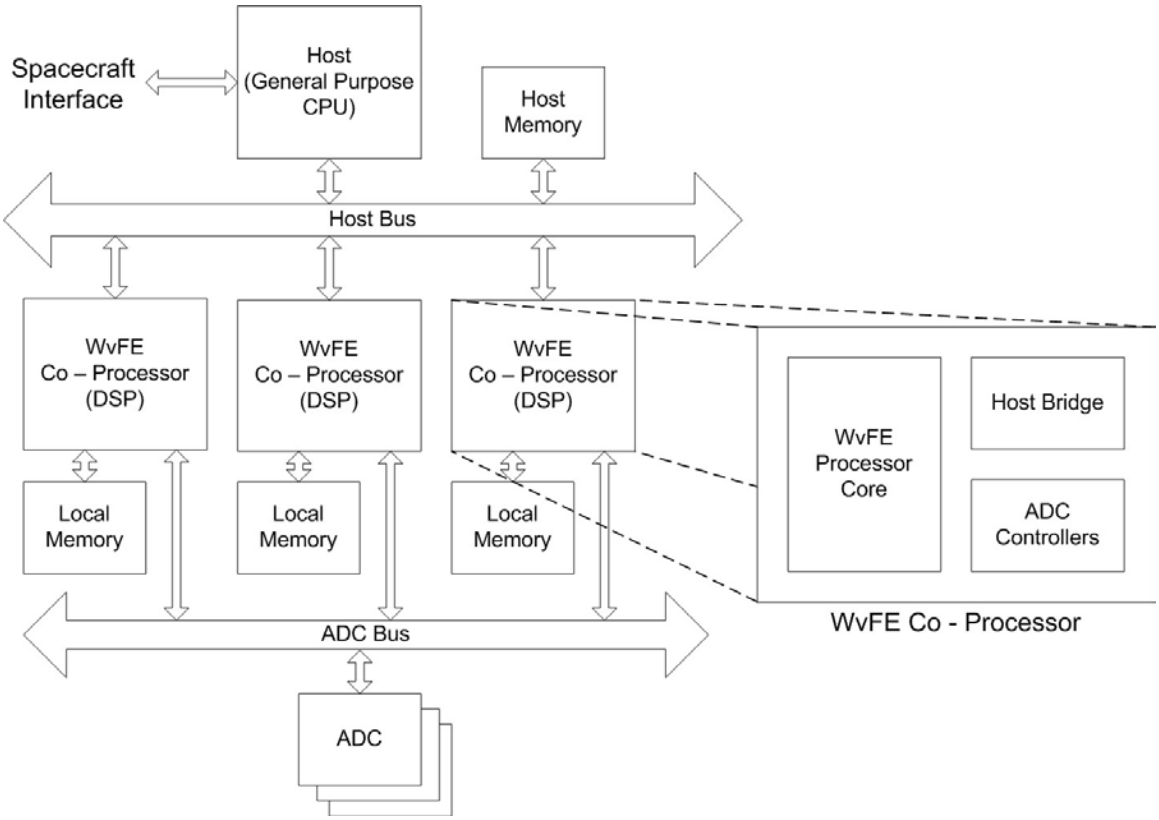


Figure 3-1: Architectural description of a multi-processor DSP system with *WvFE SoC co-processors*

From this point forward, the emphasis of discussion will be placed upon the *WvFEv3* processor core. Specific details, such as those related to the peripheral cores and interconnections of the *WvFE SoC co-processor* architecture is only described when needed.

WvFEv3

The *WvFEv3* is a synthesizable, programmable processor soft IP core with an emphasis on performing DSP related tasks quickly and efficiently in IEEE 754 single precision floating point. The term *WvFEv3* refers to 'Waves FFT Engine' and represents a relic of convention that emphasizes a misnomer of the architectures' capabilities, as the design is far more capable than performing just FFTs. For this reason the production release is referred to as *WvFEv3,* representing the version of release and a shortening of the 'Waves FFT Engine' convention. Superseding version two, the design of the *WvFEv3* further optimizes the instruction set architecture (ISA) and microarchitecture while also implementing fault tolerant features. In additional to these optimizations, the secondary goal of the ISA and microarchitecture is to keep gate resources minimal when it does not adversely affect performance, due to resource limitations of the FPGA target.

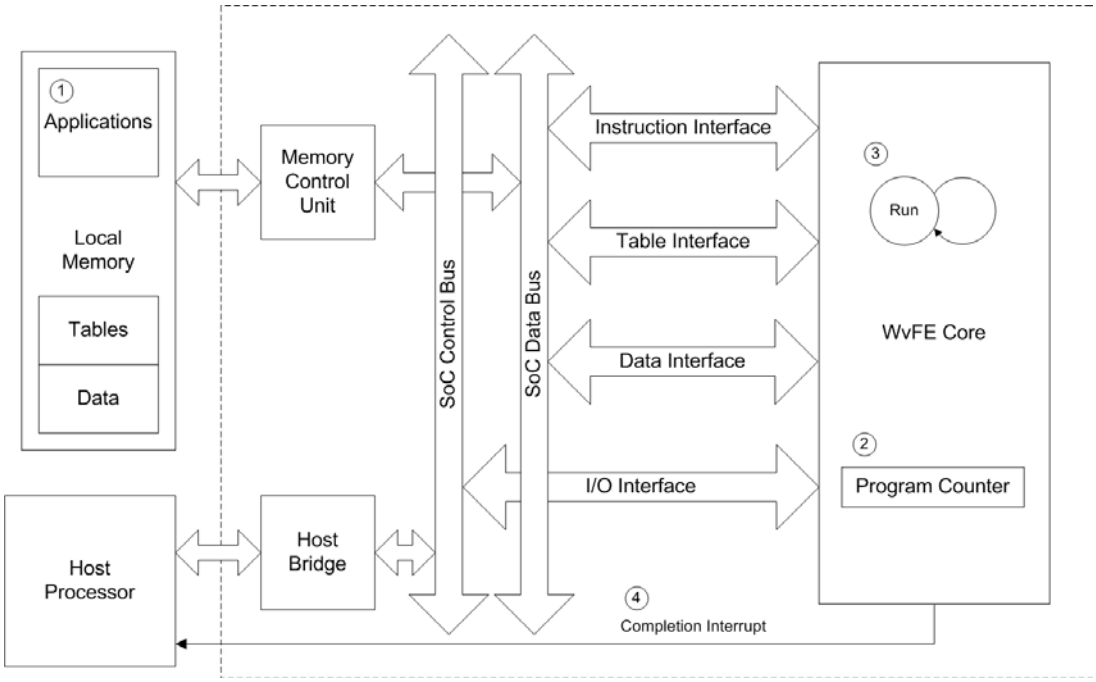


Figure 3-2: System architecture of the *WvFE SoC co-processor*

The *WvFEv3* processor is a modified Harvard architecture machine, which logically uses several address spaces for instructions, data, input/output, and 'virtual tables'. Each address space utilizes a separate interface for fetching data, therefore increasing the total amount of potential data bandwidth available to the processor core. This configuration also makes implementing a caching structure more convenient as each interface can utilize a different caching scheme depending on its unique accessing patterns. In the physical implementation, however, the various interfaces are time multiplexed to a single local memory using external arbitration logic due to pin limitations of the physical package.

Application code for the *WvFEv3* is simple in nature and similar to that of general-purpose processors. Applications contain a series of consecutive instructions performing small steps to achieve a larger function. These functions can then be performed serially using calls from a higher-level main function; such as performing an A/D capture, then a FFT on the capture, then binning the results. Executing an application typically follows these operations:

Table 3-1: Series of operations needed by the host processor to configure the *WvFE SoC co-processor* for program execution

Step	Operation
1	The host processor loads application code into *WvFEv3* local memory
2	The host processor writes the start address of the loaded application to *WvFEv3's* program counter
3	*WvFEv3* begins fetching and executing instructions until a halt instruction or a fault is encountered
4	Upon completion, the *WvFEv3* signals the host processor

Since the *WvFEv3* is considered a co-processor, there has been no attempt to incorporate features in the ISA to provide operating system support, such as interrupt handling or context switching.

The next sections of this chapter will discuss details of the instruction set architecture and microarchitecture in more detail, followed by a discussion of instruction flow considerations.

Instruction Set Architecture

The *WvFEv3* instruction set is an example of an explicitly parallel instruction computing (EPIC) paradigm, utilizing a very long instruction word or instruction bundle to group multiple instructions in an attempt to exploit instruction level parallelisms. The EPIC paradigm gives the programmer or compiler complete control over how instructions are allocated to the various functional units inside the *WvFEv3* core, reducing complexity during instruction dispatch, but at the cost of added complexity on the programmer or compiler.

An instruction bundle contains two instructions that are packed into one fixed 64-bit word, one is allocated to the control unit (CU), fixed at 42-bits long, and the second is allocated to the floating-point unit (FPU) using the remaining 22-bits. Each instruction pair is executed lock step and in parallel by each of the logical execution units. Each unit, the CU and FPU, operate on a unique 'reduced instruction set computing' (RISC) like sub instruction set tailored to the types of operations they are responsible for. Each of the sub-ISAs perform register to register operations; the CU ISA incorporates full access to all sixty four 32-bit registers in the register file using 6-bit address fields (R_d, R_a & R_b), where the FPU ISA has limited visibility by only allowing access to the upper 32 registers with 5-bit address fields (R_z, R_x, & R_y). Since the sub-ISAs share access to the

register file, they are not mutually exclusive data-wise, and scheduling of sub-instructions must adhere to certain rules to avoid hazards. Other fields in the instruction bundle include a control unit operation code (C_{op}), a floating point unit operation code (F_{op}), a virtual table address (V_t), and an 18-bit or 24-bit immediate constant (Imm_{18}, Imm_{24}). For a complete listing and description of all instructions and formats for both execution units please refer to Appendix A.

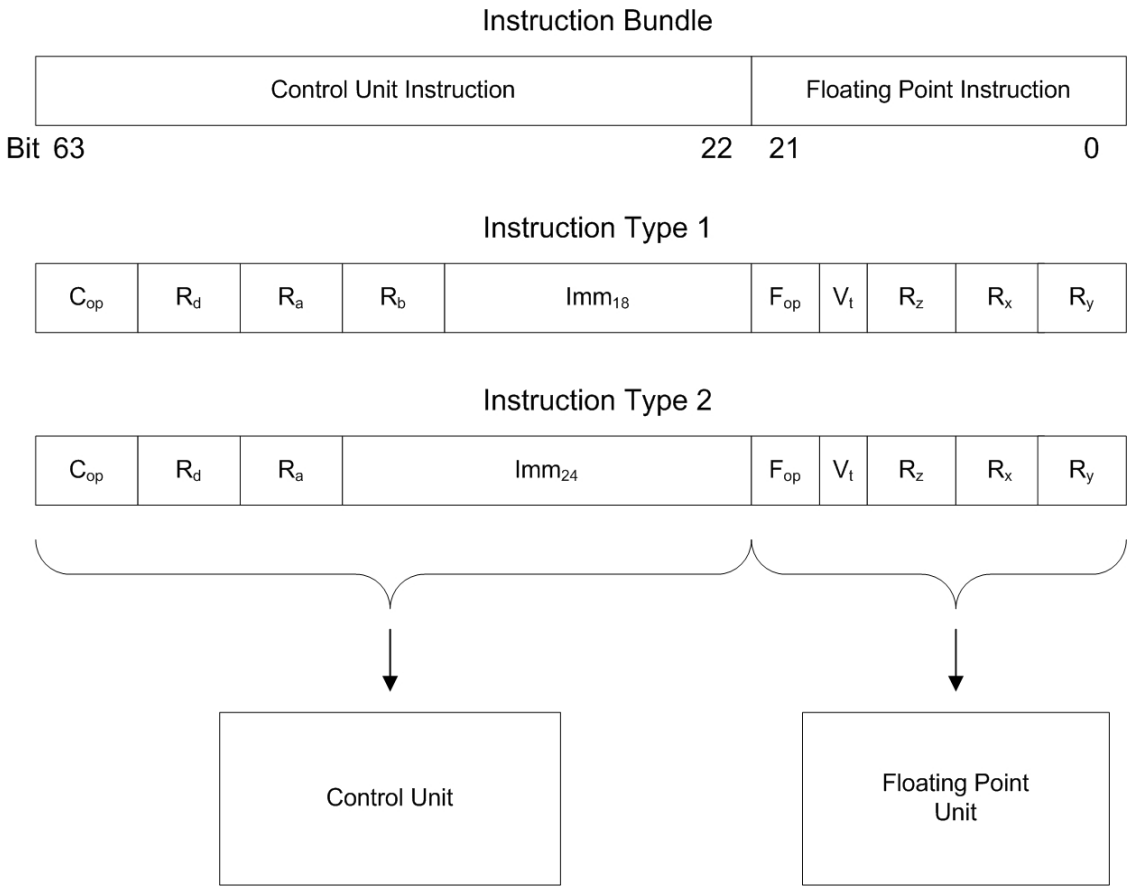


Figure 3-3: Description of *WvFEv3* Instruction bundle formats

Control Unit ISA

The control unit is a logical execution unit that performs integer based arithmetic and programmatic operations. Its primary purpose is to move data in and out of the processor, in an attempt to keep the FPU unit performing useful operations. To this end, the CU ISA incorporates store/load instructions, various integer arithmetic instructions, bitwise operators, program control flow instructions, and a handful of special butterfly address calculation operations specifically designed for FFT execution. Additionally, the CU ISA also implements instructions for approximating reciprocal and square roots in floating point, as seed values to Newton-Raphson convergence functions.

The control unit's role also extends to managing the state of the processor core; enumerated states include *idle*, *run*, *sleep*, and *error*. Most of these states are self-explanatory, with the exception of *sleep* state, invoked by the 'sleep' instruction. It represents a low power processor state, much like the *idle* state, but is polling for an external event to 'wake' it up. An event is synonymous with an asynchronous interrupt, but the response from the processor is primitive in nature, it simply transitions from *sleep* to the *run* state and continues executing where the program counter left off. The *sleep* state is useful when the *WvFEv3* processor has set up an external activity, such as an A/D capture, and must wait for the data to become available to execute the next task.

Floating Point Unit ISA

As is obvious from its name, the FPU performs floating-point arithmetic. The multiply accumulate operation, being particularly well suited to DSP-related tasks, forms the foundation from which supplemental operations are derived. Addition, subtraction,

multiplication, and multiply-decrement are all implemented through the toggling of operands into the fused multiply accumulate block via op-code decoding.

In addition to floating-point operations the FPU ISA also incorporates a unique feature for accessing non-registered data, called virtual tables. A virtual table is a method of address translation to blocks of local memory with the goal of increasing data throughput to the FPU and reducing the memory requirements on local memory for storing pre-computed functions, cyclic ones in particular.

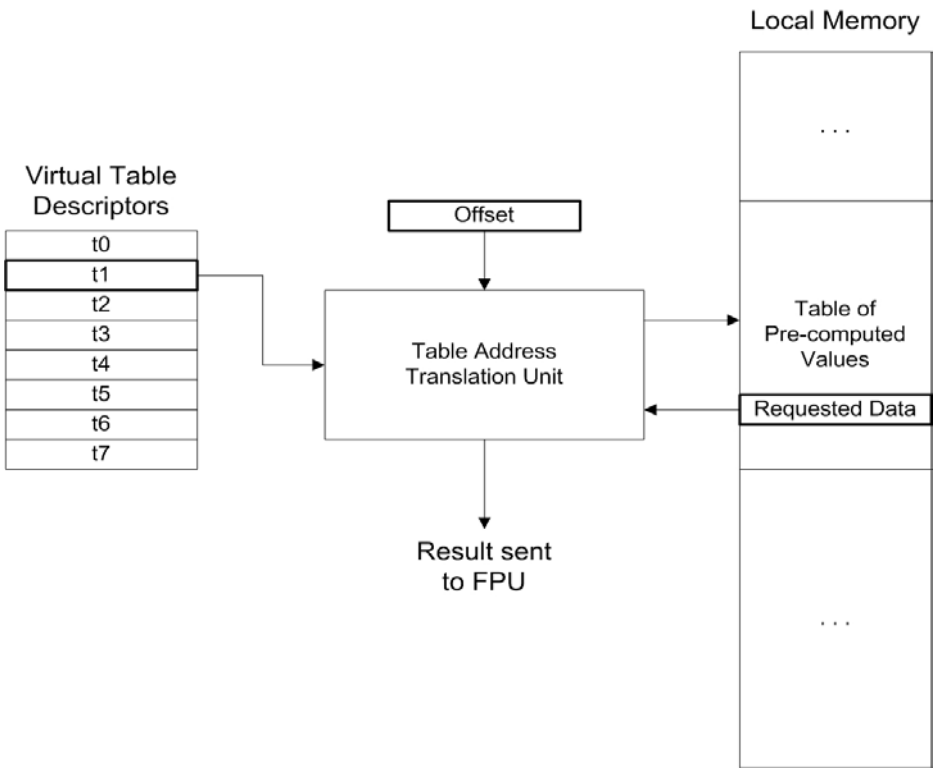


Figure 3-4: A logical description of the virtual table translation unit

At the heart of the virtual table concept is the table address translation unit or TAU. The TAU reads one of the eight virtual table descriptors and an offset value referenced by the executing instruction. It then performs a translation on the offset to

fetch the requested piece of data from local memory and apply any transformation on the data. The virtual table descriptors provide location, size, and orientation of the physical buffer being referenced and whether the fetched data needs to be altered. Additionally, in the case of cyclic functions, such as (co)sine, the descriptor also holds function polarity and fractional representation of the physical buffer. To illustrate the concept of virtual tables, imagine a full sine wave that is segmented into four equal portions. Any segment is equivalent to any other, just inverted or rotated, and in essence the entire sine wave can be represented by just one segment if the appropriate rotations and inversions are applied. This is exactly the function of the TAU, to perform the conversions necessary to represent cyclic functions in a fraction of the traditional memory requirements.

Microarchitecture

The *WvFEv3* microarchitecture is a direct extension of the duality imposed by the ISA specification. It employs two execution flows, one representing the control unit and the other representing the floating-point unit, each operating on their respective sub-ISAs. Detailing each flow further is a discussion of the pipelined design, followed by a description of the caching architecture, and finally considerations in instruction flow.

Concurrent Dual Pipelines

As can be seen in Figure 3-5, the microarchitecture is centered about the register file, with a single pipeline beginning with instruction fetch that eventually diverges into two execution flows. The left side represents operations defined by the CU ISA and the right side represents operations defined for the FPU ISA. Also illustrated is the mismatched depth of each pipeline, CU operations are typically simpler in nature,

requiring fewer stages than the FPU. Physically, the CU is a four-stage pipeline, and the FPU is a seven-stage pipeline resulting in two instructions executing every machine cycle.

Instruction execution for both units begins when the next program counter is calculated in the instruction pre-fetch stage. The program counter is presented to the instruction interface to fetch the needed instruction. The interface is 64 bits wide resulting in one instruction bundle fetch per transaction. The fetched instruction bundle is passed to the operand pre-fetch stage, which decodes the instructions and presents up to six possible register addresses to the register file. Finally, the register file fetches the appropriate data and routes it to the next stages of the divergent pipelines.

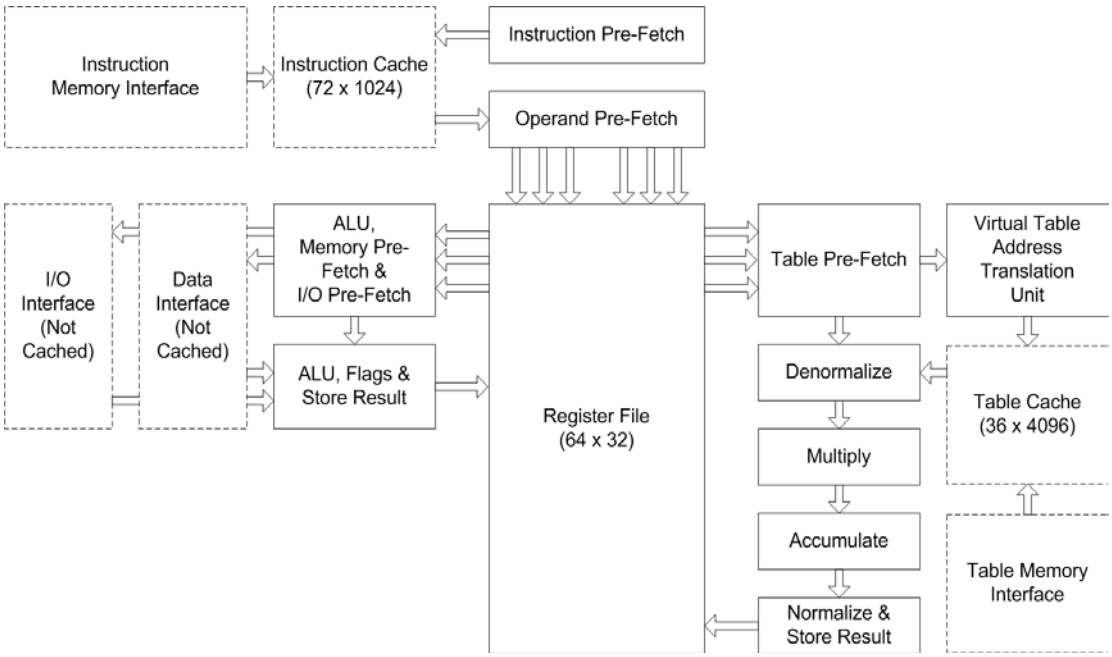


Figure 3-5: Microarchitecture of the pipelined *WvFEv3* processor

The control unit continues execution in the next stage by calculating addresses for memory and I/O accesses, or performing the first stage of a more sophisticated integer

operation. The following stage finishes the interface access or integer operation and stores the result, if needed, back to the register file.

The floating-point unit execution follows a similar path, beginning with a table address calculation. The fetched table values are passed, along with operands from the table pre-fetch stage to the next stage. The denormalize, multiply, accumulate, and normalize stages all follow, representing a complete floating point multiply-accumulate operation. Addition, subtraction, and multiply all follow the same pipeline flow, the only difference being how operands are selected and manipulated. The final stage is shared between the normalization step and write back of the result to the register file.

Cache Architecture

The *WvFEv3* microarchitecture implements several interfaces for accessing data; the instruction, table, data, I/O interfaces. Due to the pipelined nature of the design, it is now theoretically possible to request multiple memory locations per machine clock. The gains in performance can only be attained when data is provided to the processor core in a timely manner, otherwise the processor must stall and wait for the data to become available. To alleviate the demand on local memory, a local cache is used on the instruction and table interfaces for data items that are repeatedly accessed.

Accessing data items from the table or instruction interface is typically done in a spatially localized manner. The table interface, for instance, will generate physical addresses for a virtual table that could be up to 4096 consecutive values in size. While the instruction interface, in areas of high repetition (common for DSP), will execute code out of a series of consecutive locations in memory. Since the data accesses are well understood for the applications intended, a simple but appropriate caching scheme is adopted, a direct-mapped cache.

The instruction cache (I-cache) is implemented as a 64-bit by 1024 entry direct-mapped cache utilizing a block size of 128 entries. This allows up to eight different segments of consecutive code to be cached concurrently. The table cache, logically shown in Figure 3-6, is larger at 32-bit by 4096 entries and a block size of 512, again this allows for eight different segments of consecutive table data to be cached concurrently, reflecting the eight virtual table descriptors available.

The FPGA fabric contains static RAMs that are used to implement the caching architecture; these internal static RAMs are not protected against SEU effects like the external local memory. The external local memory is hardened enough against radiation that SEUs will be very rare in Earth and Jovian orbits [22].

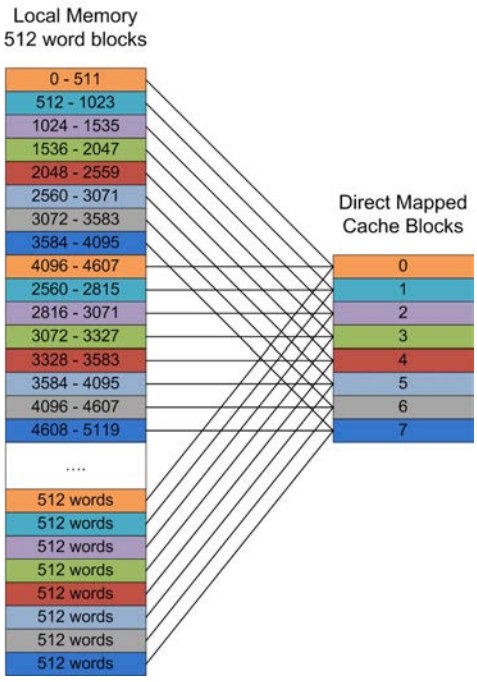


Figure 3-6: Direct-mapped memory block caching scheme

To effectively utilize the FPGA's static RAMs for cache, a parity check and reload scheme was adopted. When the processor requests a line of cache, pre-computed parity bits are verified and if a mismatch is encountered, the processor is stalled while the cache re-fetches the affected line from local memory. In this way, the processor can continue executing without having to fault due to an SEU in the cache structure.

Hazards and Programming

Increasing performance through a pipelined microarchitecture, as seen in a previous section, can drastically increase performance but comes at the cost of added complexity when scheduling and executing instructions due to various hazards. To minimize the impact of these hazards, several attempts have been made to partially alleviate their effects during execution flow.

The first of these hazards is due to the pipeline's multi-stage separation between operand fetching and the writing back of results, leading to hazards known as 'read after write' (RAW) hazards. A RAW hazard is the result of data dependent operands in an instruction being read before these dependencies have been fulfilled by the write back of results from previous instructions in the program flow. To avoid this hazard the dependent instruction must read its operands after the results write back from previous instructions. The CU has effectively two stages separating the operand fetch and write back stage of the result, leading to a two slot exposure to a RAW hazard. The FPU suffers the same issue, except the operand fetch and write back stages are separated not by two stages but by five stages, leading to a five-slot exposure. To partially alleviate the exposure to RAW hazards, result forwarding is implemented in the register file to route the results being written back directly to requesting instructions in the operand fetch stage. This cuts out the cycle needed to commit the result to the register file before

reading it back, resulting in a RAW exposure reduction in the CU to 1 slot and in the FPU to 4 slots. Avoiding the rest of the RAW hazards is the responsibility of the programmer. These exposed slots should be filled with useful instructions or if not possible, NOPs, as seen in Figure 3-7.

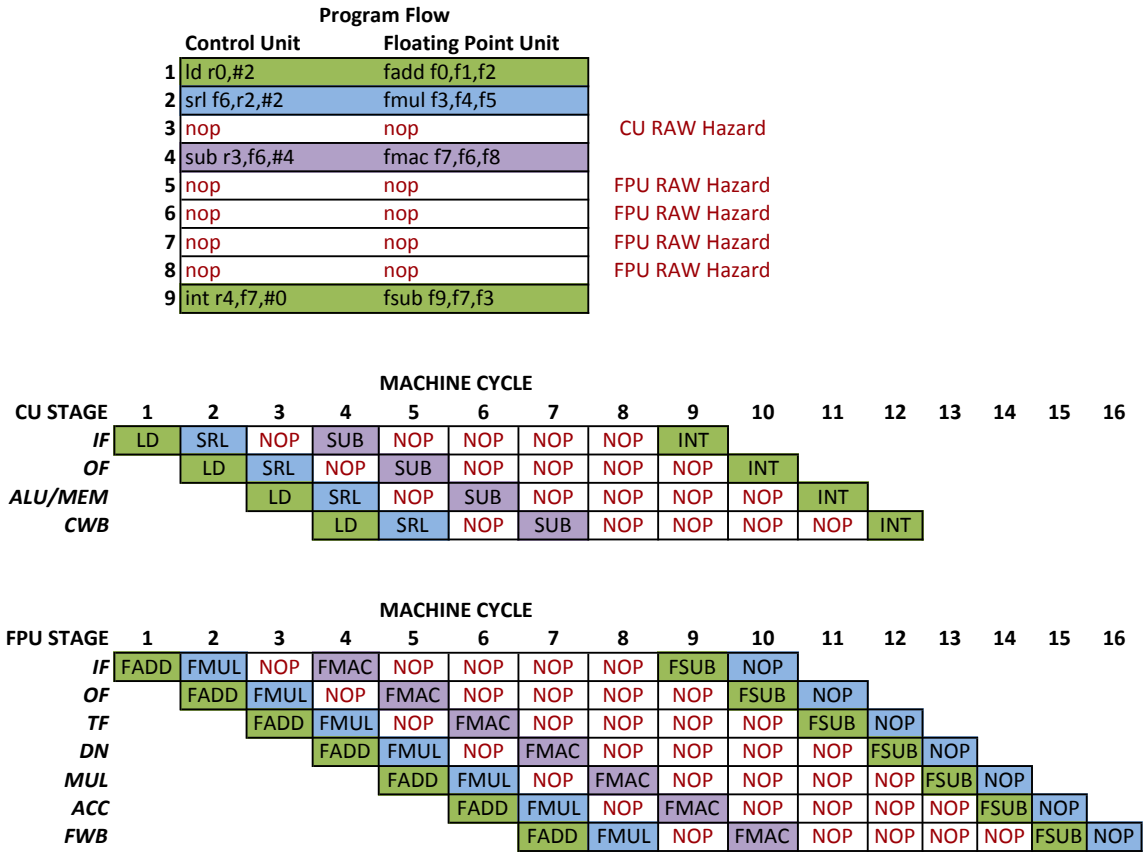


Figure 3-7: Read after write hazards of the *WvFEv3* pipelines

Control flow instructions such as calls, returns, and branches present their own set of unique challenges. In the case of a call instruction, the data needed to compute the new program counter is encoded in the instruction bundle, but this does not become available until the after the instruction fetch stage. This means that the pipeline has fetched one invalid instruction directly after the call instruction; this instruction is automatically

invalidated. The return function induces a larger penalty since the return address has been saved on the stack and must be fetched from local memory before the new program counter can be loaded. Due to the additional delay, additional instructions must be invalidated that follow the return instruction.

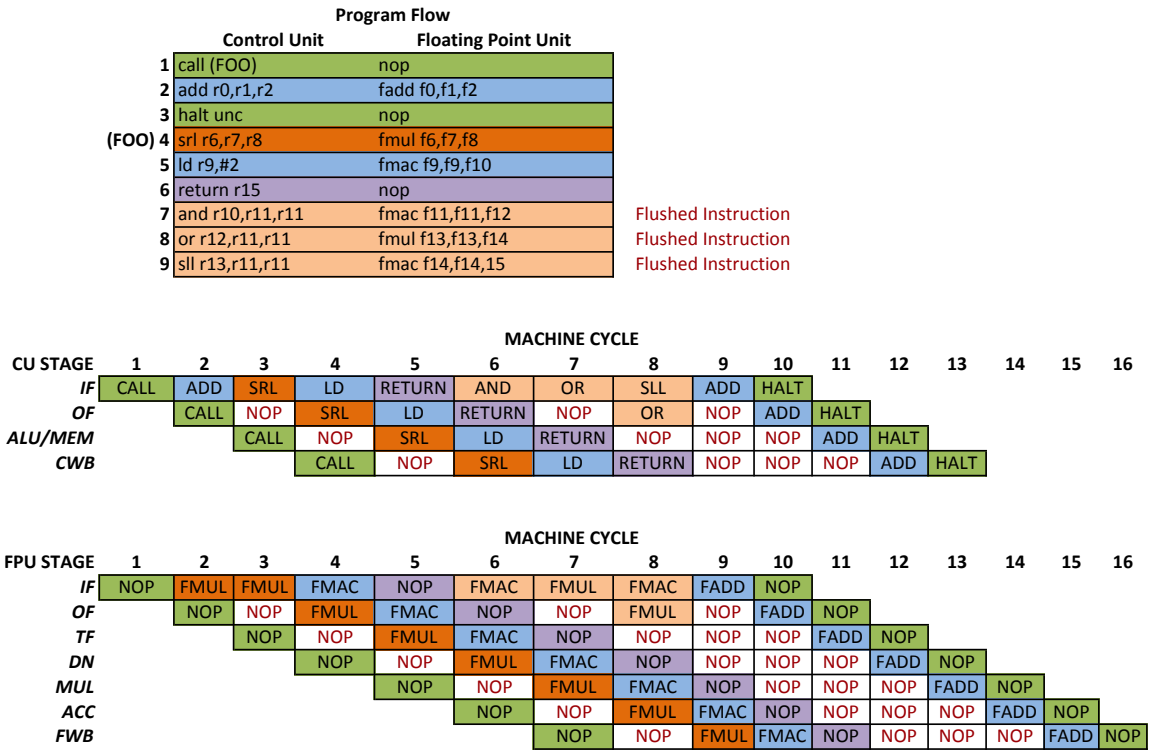


Figure 3-8: Flushed instructions from function calls and returns

A branch in a pipelined design is often an expensive operation due to the unknown state of the branch until a late stage in the pipeline. The processor must stall or induce NOPs until the condition of the branch is resolved, degrading performance. In the case of the *WvFEv3* processor, a branch would induce three machine clocks of wasted cycles and could, in the case of tight loops, degrade performance significantly. Branch

prediction, a technique used to guess which execution path should be taken, can be used regain a portion of the lost performance by filling the wasted slots with, hopefully, useful instructions. A successfully predicted branch will only incur one pipeline bubble due to the branch address being unavailable during instruction fetch.

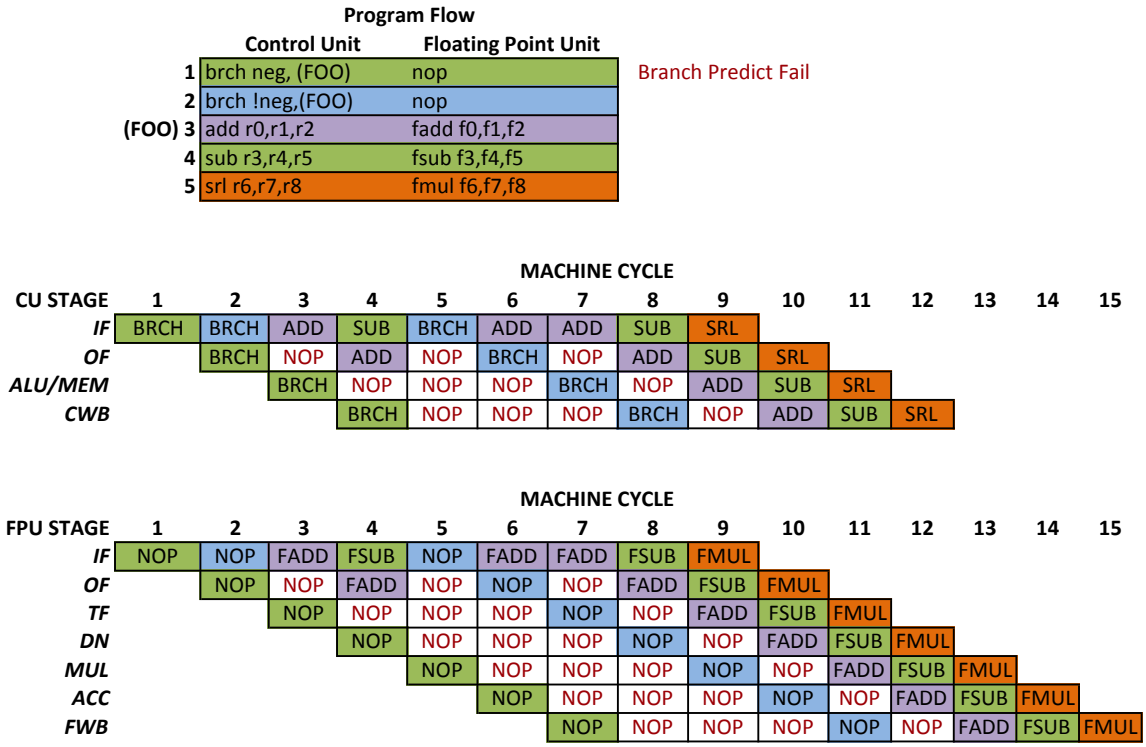


Figure 3-9: Flushed instructions from branch prediction

In the case when a predicted branch is incorrect, the instructions fetched from the false path must be flushed from the pipeline and the alternate and correct program flow taken. The *WvFEv3* utilizes a simple 'always taken' branch prediction scheme due to the repetitive nature of the target DSP applications

The mismatched pipeline depth of the execution units in the WvFEv3 design has lead to one additional hazard, a write after write (WAW) hazard. Since the FPU writes its

results several stages after the control unit, an opportunity arises where the FPU can overwrite the CU's result even though the CU instruction came programmatically later than the FPU's instruction. An example illustrating the sensitivity to WAW hazards is shown in Figure 3-10. The processor does not include any detection of such an event, and the programmer must be aware of this situation when writing applications.

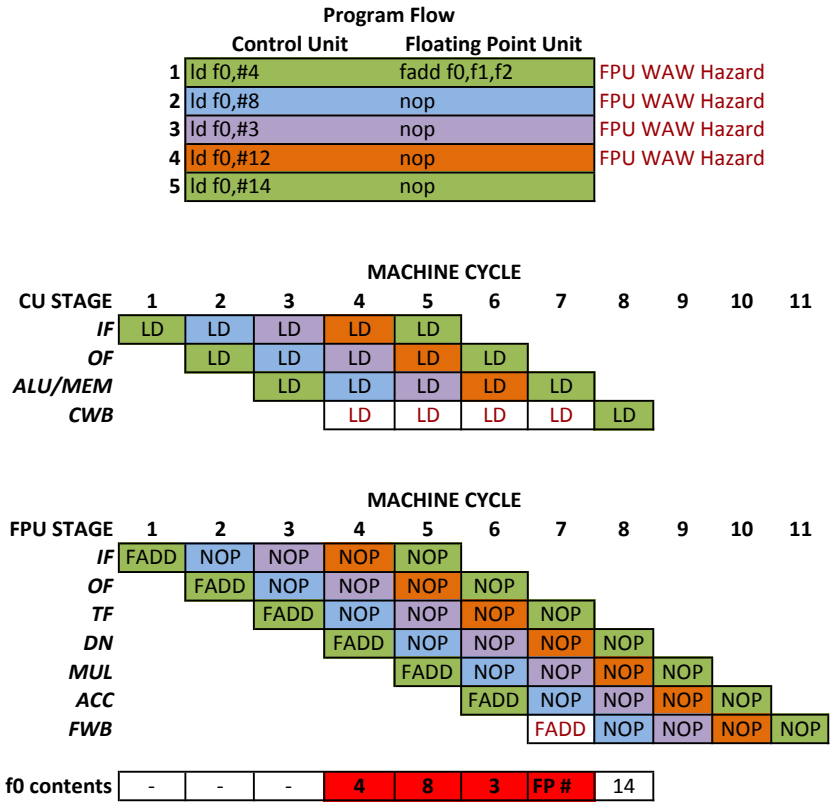


Figure 3-10: Write after write hazards of the mismatched *WvFEv3* pipelines

Writing software for the *WvFEv3* architecture is made possible by a custom defined assembly language and a port of the GNU binutil toolset. The assembly language highlights the parallel nature of the instruction bundles increasing the programmer's awareness of instruction scheduling with examples of code shown in Appendix B. The

GNU binutil toolset, a community supported system software infrastructure, provides the foundation for an excellent set of tools to develop re-locatable software modules; an assembler, linker, and disassembler.

CHAPTER IV

RESULTS

An analysis of the physical FPGA resources needed by the *WvFE SoC co-processor* is presented. This is followed by the design's critical path and a detailed analysis of the execution of the fast Fourier transform algorithm.

Physical Resources

The *WvFE SoC co-processor* was synthesized, placed and routed for an Actel RTAX2000S FPGA with the Actel Libero IDE v9.1 toolset. These tools provide utilization metrics of the various resources needed to implement the design in RTAX FPGA fabric. Several components described in VHDL make up the *WvFEv3 SoC co-processor*, most of these are external cores and interconnects to the *WvFEv3* processor and will not be characterized to a detailed level. The VHDL description of the *WvFEv3* processor is sub-divided into several components, a register file, an instruction cache, a table cache and a *WvFEv3* core representing a combined control unit and floating point unit.

Sequential elements such as flip-flops and latches are categorized as R-cells in the RTAX FPGA fabric. The total R-cell utilization of the *WvFEv3* processor is 42.4% of the RTAX2000S compliment or 4564 cells total. The register file, as seen in Figure 4-1, requires the majority of these at 2048, followed closely by the *WvFEv3* core at 1823. The instruction cache and table cache each use a relatively small portion of R-cells at 386 and 307, respectively.

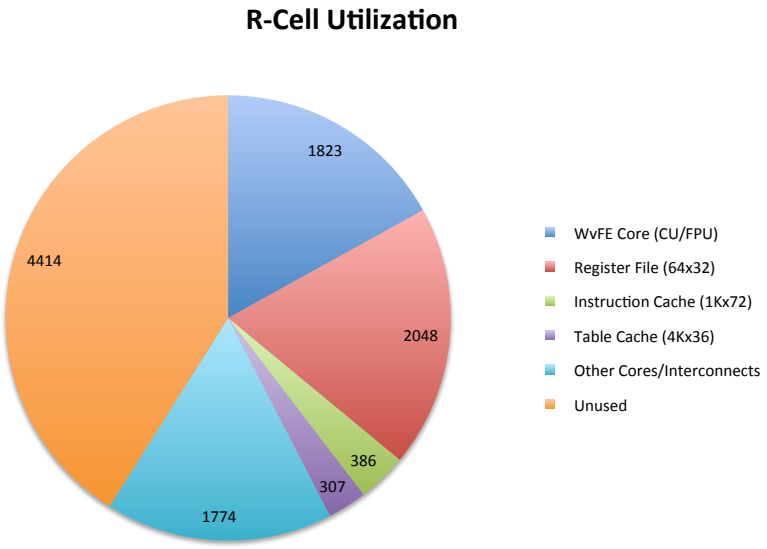


Figure 4-1: RTAX2000S R-cell utilization of the *WvFE SoC co-processor*

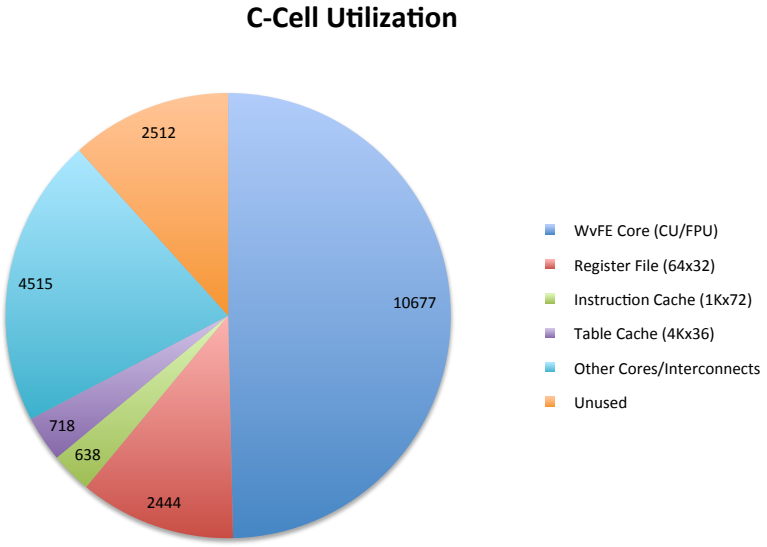


Figure 4-2: RTAX2000S C-cell utilization of the *WvFE SoC co-processor*

Combinatorial logic or C-cell utilization of the *WvFEv3* processor is significantly higher than that of the R-cells usage. The design's total, shown in Figure 4-2**,** is 14477 C-cells or 67.3% of an RTAX2000S. The majority of these cells, nearly 74%, are required to implement the control unit and floating point unit. The register file, instruction cache, and table cache implementations require 2444, 638, and 718 cells respectively.

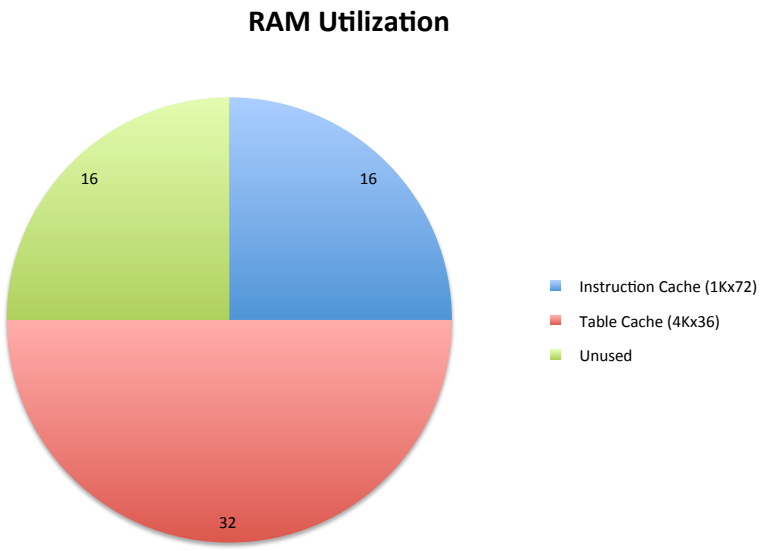


Figure 4-3: RTAX2000S internal RAM block utilization of the *WvFE SoC co-processor*

Only two of the *WvFEv3* processor sub-components require the use of the on-board RAM modules, the instruction cache and table cache. The instruction cache uses sixteen 512x9 RAM blocks to implement a 1Kx72 memory structure. The table cache uses significantly more at thirty-two 512x9 RAM blocks to create a longer but narrower 4Kx36 memory structure. Total utilization of the design requires 48 RAM block or 75% of the RTAX2000S complement.

RTAX cell utilization provides an adequate measure to make comparisons only to other designs that have been targeted for this FPGA family. Often this is not the case and an attempt has been made to convert the number of cells to an ASIC gate count. Due to many factors, it is only possible to calculate an estimate of the number of gates required. The *WvFEv3* processor would require approximately 150,000 to 175,000 gates for an ASIC implementation.

Performance

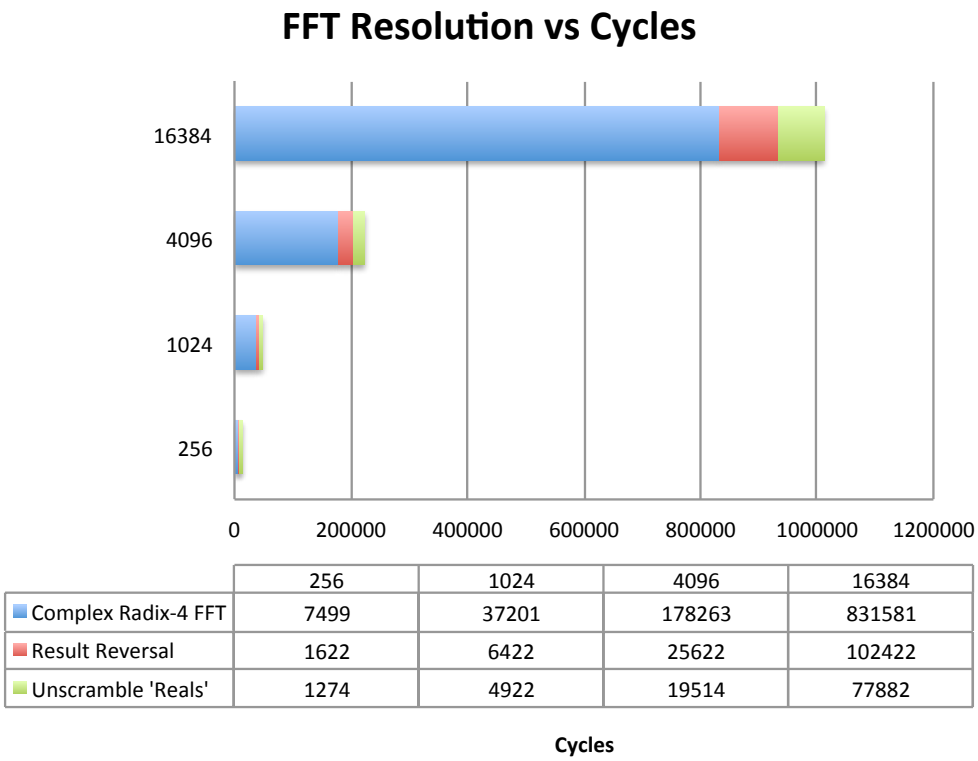
The performance characterization of the *WvFEv3* processor is provided as a detailed analysis of several metrics; including the critical path of the design and an analysis of the execution of the FFT algorithm including cycle counts, branch prediction accuracy, cache hit accuracy, and power utilization.

Critical Path

The critical path is the longest delay path between two sequential elements in the same clock domain; this dictates the maximum frequency for which the clock can operate for reliable operation. The *WvFEv3* critical path is the combinatorial path found between register outputs in control unit's memory pre-fetch stage, through the external local memory, to the inputs of registers in the store result stage. The total delay associated with this path is 73.544 ns, which relates to a maximum clock rate of 13.6 MHz. However, integrated circuit design rules set forth by Jet Propulsion Laboratory [8] require a de-rating on the clock frequency, resulting in an conservative operational clock of 10.5 MHz; this metric will be used for all subsequent analyses.

FFT Performance

The Fast Fourier transform (FFT) algorithm is used as a performance benchmark due to its computational complexity and its direct relevance to performance requirements needed by RBSP. The algorithm is sub-divided into several steps as discussed in the background section, the complex radix-4 FFT, the result reversal, and the unscrambling of real results. This algorithm was executed in simulation to determine the performance of each step in terms of execution cycles, branch prediction accuracy, and cache hit accuracy. For a complete *WvFEv3* assembly listing of the algorithms implemented for this benchmark please refer to Appendix B.



	256	1024	4096	16384
Complex Radix-4 FFT	7499	37201	178263	831581
Result Reversal	1622	6422	25622	102422
Unscramble 'Reals'	1274	4922	19514	77882

Figure 4-4: Number of execution cycles utilized by the *WvFEv3* processor to execute a complex FFT of various resolutions

Figure 4-4 relates the number of cycles needed per step of real FFT execution to four radix-4 resolutions, the lowest 256 and highest 16384 points. The largest component of execution is, unsurprisingly, the execution of the complex FFT algorithm with approximately 80% of the execution cycles spent performing this operation. The number of cycles to perform each of these steps is deterministic and therefore can be described with the following equations where N is the resolution of the complex FFT being calculated.

$$Cycles(Complex\ FFT) = \log_4 N * \left(\frac{N}{4} * 29\ cycles + 6\ cycles\right) + 51\ cycles$$

$$Cycles(Result\ Reversal\ ) = \left(\frac{N}{4} * 25\ cycles\right) + 22\ cycles$$

$$Cycles(Unscramble\ Reals) = \left(\frac{N}{4} * 19\ cycles\right) + 58\ cycles$$

Equation 4-1: Equations to calculate the number of cycles needed to perform various steps in the computation of real FFTs

Execution time of the real FFT calculation is calculated by multiplying the number of cycles needed for computation against the clock period of processor and dividing the result by two for each real FFT result. The physical implementation of the *WvFEv3* processor uses a clock frequency of 10.5 MHz or 95.24 ns per cycle. Using this metric as the clock period and dividing by two yields an execution time of a single real 1024-point FFT in 2.312 milliseconds. The execution time the other radix-4 resolutions are seen in Figure 4-5.

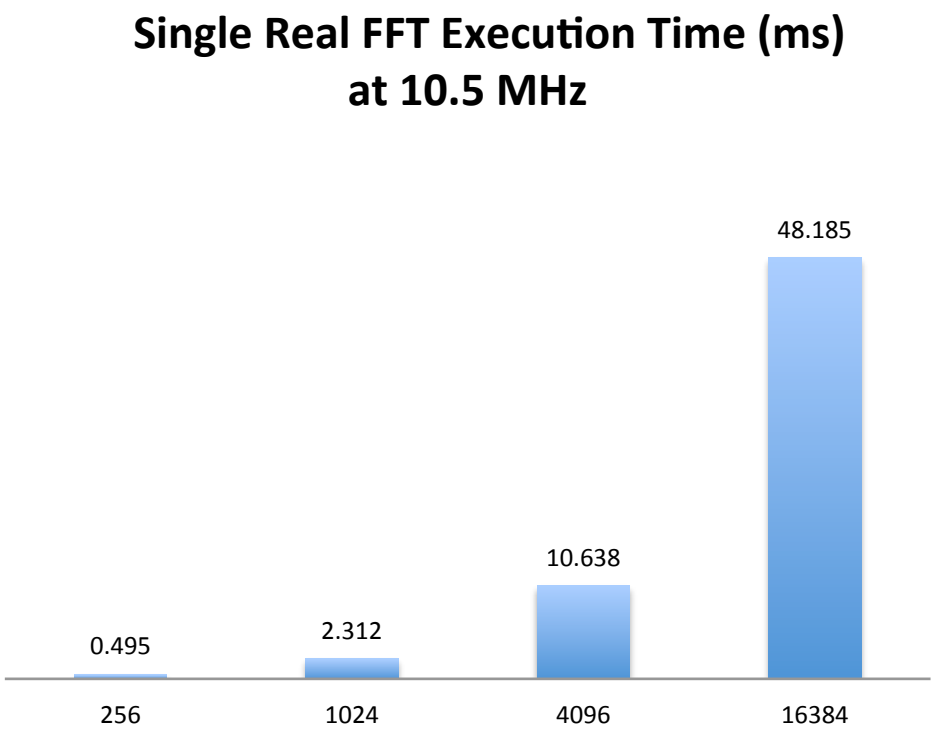


Figure 4-5: Execution time for the calculation of a real radix-4 FFT for various resolutions.

A useful metric related to performance of a pipelined processor is that of cache hit and branch prediction accuracies. The higher the accuracy of these functions the fewer stalls the processor incurs when executing instructions. Table 4-1 depicts accuracies for the computation of the dual real FFT algorithm after a processor reset. In this scenario, the caches do not contain any valid data for FFT execution and represent the highest number of cache misses incurred. This decreases the optimal accuracy from 100% to an average of 99.9% for both caches. Once the caches have been loaded with valid FFT computation data any subsequent FFT execution will achieve the optimal 100% accuracy. The accuracy of the branch prediction method is not dependent on the state of the caches and as a result will stay static upon repeated FFT executions. The prediction accuracy for the all steps of the FFT computation averaged is 99.122%.

Table 4-1: Profile of the branch prediction and caching scheme performance during FFT execution.

Attribute	Complex FFT	Result Reversal	Unscramble Reals	Average
Branch Prediction	99.367%	99.481%	98.517%	99.122%
I-Cache Hit Rate	99.996%	99.979%	99.974%	99.983%
T-Cache Hit Rate	99.919%	N/A	N/A	99.919%

Power Utilization

The power utilization of the *WvFE SoC co-processor* was calculated through the execution of the complex FFT algorithm in physical simulation. During simulation various scenarios of operation were observed, the loading of the instruction cache, the loading of the table cache and the execution of the FFT algorithm. A waveform for each scenario was captured and analyzed at the gate level using Actel's SmartPower tool, providing cycle accurate power measurements. Figure 4-6 depicts the analysis of each of these scenarios where the static power utilized by the design is unchanging at 131.14 mW. The static power represents the amount of power the design consumes at idle.

The dynamic power is the additional power needed to perform the operation associated with each scenario. For instance, the dynamic power associated with loading of the instruction cache is 192 mW for a total power consumption of 323.6 mW. A slight increase in power is seen when loading the table cache at 346.4 mW and the power peaks during FFT execution at 521.2 mW.

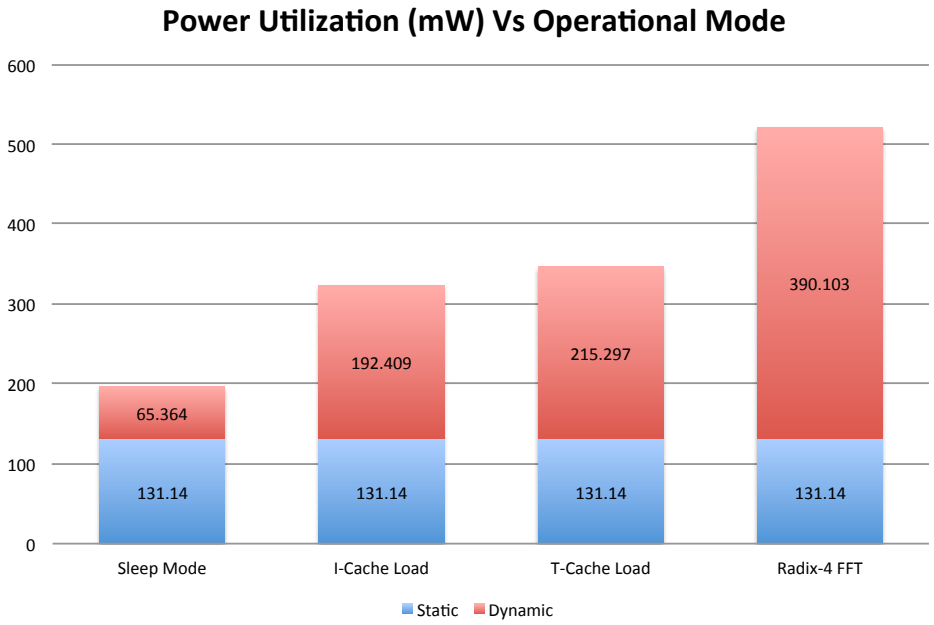


Figure 4-6: Power profiles for various modes of processor operation during program execution.

Finally, one additional scenario is portrayed that characterizes the power profile of the *WvFEv3* during the low power *sleep* mode. As expected, the dynamic power utilization for this mode is significantly lower than the other scenarios at 65.364 mW. The low dynamic power is attributed to the system clock gating to sequential elements not needed to support the functions of *sleep* mode in the processor design.

CHAPTER V

DISCUSSION

Space Qualification

The foremost issue in developing any digital design for space flight is tolerating radiation effects and temperature extremes of the space environment. To address these issues an Actel RTAX2000S FPGA was chosen as the implementation technology due to several inherent features that make it well suited for space flight. One feature in particular, triple module redundancy provides a hardening solution that reduces single event upsets in sequential elements to less than 10^{-10} errors/bit-day at geosynchronous orbit [18]. However, since the number of SEUs is non-zero, several attempts have been made to further the design's resilience by detecting and recovering from SEUs. These features include the hardening of state machines, instruction pipeline parity checking, and cache line parity checking.

The highest fidelity approach to testing the fault tolerant features of the *WvFEv3* processor would be in a temperature-controlled vacuum under radiation exposure, however the equipment costs make this impractical. A secondary, lower cost method was employed that utilizes computer simulations to emulation SEUs. During simulation, the instruction and table cache lines received 'bit upsets' by overwriting a single bit in cache memory that resulted in the invalidation of the cache's parity bits. Once the processor tried to fetch one of these corrupted lines, the cache stalled the processor and re-fetched the appropriate data from local memory. The cache state machine then re-loaded the line with the new data, re-calculated the parity bits and allowed the processor to continue operation.

This method successfully verified the cache's fault tolerance to SEUs but not all features could be tested in this manner. The effects of radiation are realized at the circuit

level, beyond the resolution of gate level simulations. This disconnect has created a void where it is not possible to use the same technique to verify flip-flop based upsets, such is the case in fault tolerance features in state machine and parity checking on executing instructions. Although every practical attempt has been made to verify the correctness of the designs' fault tolerant features, the final verification will only be possible during space flight.

The combination of the RTAX radiation tolerant features, the *WvFEv3* fault tolerant features, and the verification efforts put forth yield an acceptable design for space flight.

Performance

The most demanding application intended for the *WvFEv3* processor, as described in the problem statement, is the calculation of a 1024-point real FFT every 10.4 ms. To achieve this goal, many architectural considerations and trade-offs have been studied to allow for the efficient computation of various DSP algorithms, including the FFT.

The *WvFEv3* computer architecture was designed with the cyclic nature of digital signal processing algorithms in mind. The very long instruction word allows for up to two instructions to be executed every clock by the control unit and floating point unit. To increase the rate of instruction execution, pipelining of each execution unit was employed. This technique decreased the worst-case delay in the design and subsequently increased the allowable clock frequency to 10.5 megahertz. The increase in the execution rate of the processor led to a bandwidth bottleneck to local memory. To alleviate this issue, a caching architecture based a direct mapping scheme was included into the design achieving a 98% or better hit accuracy, drastically reducing the number of stalls the processor encountered during execution. Additional processor stalls caused by instruction

flow branches were removed by utilizing a simple 'always taken' branch prediction scheme. The cyclic nature of DSP algorithms has made this scheme quite successful having achieved better than 99% accuracy. The incorporation of these design choices has yielded a processor architecture that is capable of performing a 1024-point real FFT computation in 24,273 cycles or 2.312 milliseconds at 10.5 MHz achieving a 77.8% margin on the requirement.

The excessive performance of the design has resulted in various advantages beyond performance metrics alone. The first is related to the power profile of *WvFEv3* processor during execution. Higher clock rates cause the gates in the design to switch more frequently over a shorter time span, increasing the impulse power utilization. The performance margin seen above would allow the designs' clock to be lowered to as low as 2.33 MHz while still meeting the performance goals. At this clock frequency, the number of gate transitions is the same but spread out over time, decreasing the impulse power utilization. Although the decrease in impulse power utilization may be advantageous in power supply design, the most significant advantage is the availability of additional execution cycles. These extra cycles can be utilized to perform additional or more complex DSP operations, increasing the processors' usefulness and flexibility.

Flexibility

The programmatic nature of the *WvFEv3* has led to an implementation that is quite flexible when dealing with algorithms in digital signal processing and basic data manipulations. Table 5-1**,** Table 5-2 and

Table 5-3 reflect a subset of algorithms and functions that have been implemented in the *WvFEv3* instruction set. Although the majority in this list are DSP specific algorithms, the flexibility of the design allows for additional operations such as the

control of external cores and even higher-level mathematical functions such as a floating point reciprocal and square root functions.

Table 5-1: *WvFEv3* supported digital signal processing algorithms

Algorithm	Functional Description
De-spin	Removes artifacts from captured waveforms caused by the rotation of the spacecraft
De-trend	Removes a DC to very low frequency linear components in the sample set.
Windowing	Reduce waveform edge effects that result in spectral leakage, increases in spectral resolution (Hanning Window).
Complex FFT	Performs a fast Fourier transform on a complex time domain data set using the radix-4 butterfly method.
Unscramble real FFT	Separates and normalizes the results of two simultaneous real FFTs from one complex FFT operation.
Calibration	Flattens the response of the antenna and receiver by applying frequency dependent magnitude and phase metrics.
Spectral Matrix	Calculates auto and cross correlations between several signals.
Binning	Reduces the spectral resolution by averaging consecutive frequency bins either linearly or logarithmically
Adaptive Noise Cancellation	Adaptively computes the noise transfer function to remove the noise component from the captured waveforms.

Table 5-2: *WvFEv3* supported external core software drivers

Algorithm	Functional Description
A/D Capture	Software driver to perform waveform captures using the A/D controller cores, included in the *WvFE SoC co-processor*.
Rice Compression	Software driver to perform waveform compression using the Rice compression core, included in the *WvFE SoC co-processor*.

Table 5-3: *WvFEv3* supported software mathematical functions

Algorithm	Functional Description
Reciprocal	Generates an approximate reciprocal of a floating-point number then converges upon the precise solution using the Newton-Raphson method.
Square Root	Generates an approximate square root of a floating-point number then utilizes the reciprocal function to converge upon a precise answer using the Newton-Raphson method.

The Waves instrument aboard the *Juno* spacecraft provides further testament of the *WvFEv3* processor's flexibility by allowing for an unconventional solution to an electromagnetic interference issue discovered at the spacecraft level. The *Juno* spacecraft utilizes solar panels where one or more strings of cells are switch on or off depending on the power generated by the strings and the power needed by the spacecraft. Late in the design phase, it was discovered that the switching frequency of these panels would

produce electromagnetic interference that would induce noise in signals the *Waves'* instrument was designed to measure [23]. In an attempt to mitigate the impact of the noise on the scientific data, spare processing cycles on the *WvFEv3* were tasked to perform an adaptive noise cancellation algorithm, similar to that found in Bose noise-cancelling headphone technology. This functionality has been verified to remove noise from a signal through unit level testing.

The *WvFEv3* processor architecture provides a level of flexibility that falls between that of a general-purpose central processing unit and an application specific logic implementation. The level of flexibility achieved is appropriate for the DSP applications intended aboard the *Juno* and *RBSP* spacecraft.

Future Work

The development of the *WvFEv3* architecture and associated system software tools has presented several occasions where efforts could be spent to further the designs' usefulness. First, the most significant effort along these lines would be the development of a high-level language compiler targeted specifically for the *WvFEv3* platform. Secondly, the adoption of an industry standard bus architecture would allow the *WvFEv3* to interface to third party soft cores increasing its ability to be integrated into future designs.

Several tools exist for developing software for the *WvFEv3* architecture but they rely on the programmer to understand detailed architectural and assembler specifics to write software. This simple fact increases the amount of time it takes a software developer to implement code when compared to writing in a high-level language. The relative ease of learning a high-level language with respect to assembly also increases the number of software developers capable of implementing software, dramatically

increasing the architecture's exposure. For these reasons, the development of a high-level compiler targeted for the *WvFEv3* platform would considerably increase its usefulness and flexibility.

One open source project in particular, the low-level virtual machine (LLVM) compiler infrastructure, is well suited to providing high-level language compiler support to new architectures [24]. The premise behind LLVM is the compilation of an assortment of high-level languages to an intermediary assembly representation targeted for a virtual RISC-like machine. The intermediary representation is translated by a custom 'backend' to the processor specific assembly and machine code. The implementation of a *WvFEv3* LLVM 'backend' would provide platform support for several high-level languages leading to a simplification in software development and wider adoption among software engineers.

Additionally, the *WvFEv3* processor architecture can be improved by the replacement of the Wishbone SoC bus interconnect. The Wishbone bus architecture is an open source specification provided by OpenCores [25] that allows for on-chip communications between various cores. This interconnect was selected for its low cost and satisfactory documentation. Although the Wishbone bus has served its purpose well, it would be advantageous to adopt a bus specification that is widely supported by the industry to allow the *WvFEv3* processor to be integrated with third party cores. The greatest benefit would come from the adoption of the de facto standard for on-chip interconnects, ARM's advanced microcontroller bus architecture specification [26, 27]. This solution is also well documented and low cost with the additional benefit of wide support in the processor design industry leading to the opportunity to integrate third party IP cores with the *WvFEv3* processor architecture.

Conclusion

The launch of the *Juno* spacecraft in August 2011 will mark the beginning of the *Waves* instrument's five-year journey to Jupiter. Although it may be the beginning of its journey, the launch marks the end of a demanding development effort to achieve a goal that had not been attempted before. This effort, entitled *WvFEv3*, was to design and implement a general-purpose digital signal processor targeted for a radiation tolerant FPGA. The processor is unique for several reasons, the greatest of which is the design's emphasis on a small silicon footprint allowing it to be implemented in current generation FPGAs. While the gate utilization is relatively small, the *WvFEv3* achieves substantial performance for a variety of DSP algorithms while also being flexible enough to implement additional algorithms programmatically. The implementation of the *WvFEv3* processor has surpassed the needs of the *Waves* instruments aboard *Juno* and *RBSP*; both in performance and flexibility but the final test of the design's space worthiness will only be proven in flight. Godspeed and safe travels!

APPENDIX A

WVFEV3 INSTRUCTION SET ARCHITECTURE

General assembler semantics can be found in the GNU binutils assembly language manual [28] while *WvFEv3* specifics are described in this appendix. Please refer to, Table A-6, Table A-7, Table A-8, Table A-9, and Table A-10 for a complete listing of control unit supported instructions and assembly formats. Floating point unit instruction and assembly formats can be found in Table A-11. Finally, valid assembly values for each instruction field can be found in Table A-1 and Table A-2. For an example of the assembly format please refer to Appendix B.

Table A-1: *WvFEv3* assembly constructs for control unit instructions

Field	Valid assembly field values
C_{op}	`halt, fault, call, brch, sleep, nop, clr, test, ld, add, sub, sra, sla, srl, sll, and, or, xor, log2, fp, int, fsqrta, frcpa, bfly2, bfly4`
R_d	`r0 - r31 & f0 - f31`
R_a	`r0 - r31 & f0 - f31`
R_b	`r0 - r31 & f0 - f31`
$signed\ imm_{24}$	#24-bit signed value, integer or hexadecimal
W_n	`w0, w1, w2, w3`
$flag$	`unc, zero, neg, par, ovf, bso, bli, pca, fzer, fneg, fovf, fdze, fing, fexp`

Table A-2: *WvFEv3* assembly constructs for floating point unit instructions

Field	Valid assembly field values
F_{op}	`fadd, fsub, fmul, fmac, tbl, nop`
R_z	`f0 - f31`
R_x	`f0 - f31`
R_y	`f0 - f31`
V_t	`t0, t1, t2 t3, t4, t5, t6, t7, t8`

The following table describes each condition flag individually along with a reference to the bit location in the flag register.

Table A-3: *WvFEv3* conditional flags

Condition Flag	Bit	Description
unc	0	Unconditional, true during processor execution
zero	1	Integer arithmetic result is zero
neg	2	Integer arithmetic result is negative
par	3	Integer arithmetic parity is even
ovf	4	Integer arithmetic result has overflowed 32-bit representation
	5	Unused
bso	6	Bufferfly seed out of bounds
bli	7	Butterfly increment to next segment
pca	8	Task complete on external core/device[1]
fzer	9	Floating point result is zero
fneg	10	Floating point result is negative
	11	Unused
fovf	12	Floating point result has overflowed single precision format[1]
fdze	13	Floating point divided by zero[1]
finv	14	Invalid floating point operation[1]
fexp	15	Floating point exception[1]

[1] Denotes a "sticky" flag that once set will continue to be so. These flags must be reset using the *clear* instruction or through a processor reset.

Fault flags are not readable by the *WvFEv3* processor and cannot be used during program execution. These flags provide a status to the host processor managing the *WvFEv3*.

Table A-4: *WvFEv3* fault flags

Fault Flag	Bit	Description
flt_ins	16	Invalid instruction fault
flt_st	17	Processor state machine fault
flt_wb	18	Wishbone bus interface error
flt_reg	19	Register file interface error

Table A-5: *WvFEv3* program flow control instructions

Type	Assembly Format	Operation(s)	Flag(s)
Halt execution conditionally	`halt flag`	$if(flag = true)\ state \leftarrow idle$	*unc*
	`halt !flag`	$if(flag = false)\ state \leftarrow idle$	
Halt execution conditionally with fault	`fault flag`	$if(flag = true)\ state \leftarrow error$	None
	`fault !flag`	$if(flag = false)\ state \leftarrow error$	
Function call [2]	`call Rd,LABEL`	$Memory[Rd] \leftarrow PC$ $PC \leftarrow PC + LABEL$ $Rd \leftarrow Rd - 1$	None
	`call Rd,(LABEL)`	$Memory[Rd] \leftarrow PC$ $PC \leftarrow LABEL$ $Rd \leftarrow Rd - 1$	

[2] Instruction induces a pipeline bubble

Table A-5: Continued

Type	Assembly Format	Operation(s)	Flag(s)
Function return [3]	`return Rd`	$PC \leftarrow Memory[Rd + 1]$	None
Conditional branch [24]	`brch flag, LABEL`	$if(flag = true)\ PC \leftarrow PC + LABEL$	None
	`brch !flag, LABEL`	$if(flag = false)\ PC \leftarrow PC + LABEL$	
	`brch flag, (LABEL)`	$if(flag = true)\ PC \leftarrow LABEL$	
	`brch !flag, (LABEL)`	$if(flag = false)\ PC \leftarrow LABEL$	
Sleep Mode [5]	`sleep`	$processor\ state \leftarrow sleep$	None
No Operation	`nop`	$No\ operation$	None

[3] Instruction induces a pipeline flush

[4] Instruction may induce a pipeline flush

[5] The processor will stay in the sleep state until an external event sets the *pca* condition flag

Table A-6: *WvFEv3* condition flag manipulation instructions

Type	Assembly Format	Operation(s)	Flag(s)
Clear flags	`clr`	$Condition(all) \leftarrow false$	All conditionals
	`clr flag`	$Condition(flag) \leftarrow false$	
Bit test operator [6]	`test Ra,Rb`	$R_a \wedge R_b$	*zer neg par*
	`test Ra,#simm24`	$R_a \wedge signed\ imm_{24}$	

[6] Instruction calculates a result but does not store the result, only sets flags

Table A-7: *WvFEv3* load and store instructions

Type	Assembly Format	Operation(s)	Flag(s)
Load constants	`ld Rd,#simm24`	$R_d \leftarrow signed\ imm_{24}$	None
	`ld Rd,Ra`	$R_d \leftarrow R_a$	
Read from memory	`ld Rd,@(Ra+Rb)`	$R_d \leftarrow Memory[R_a + R_b]$	None
	`ld Rd,@(Ra+#simm24)`	$R_d \leftarrow Memory[R_a + signed\ imm_{24}]$	
Write to memory	`ld @(Rd+Rb),Ra`	$Memory[R_d + R_b] \leftarrow R_a$	None
	`ld @(Rd+#simm24),Ra`	$Memory[R_d + signed\ imm_{24}] \leftarrow R_a$	
Read from I/O	`ld Rd,%(Ra+Rb)`	$R_d \leftarrow I/O[R_a + R_b]$	None
	`ld Rd,%(Ra+#simm24)`	$R_d \leftarrow I/O[R_a + signed\ imm_{24}]$	
Write to I/O	`ld %(Rd+Rb),Ra`	$I/O[R_d + R_b] \leftarrow R_a$	None
	`ld %(Rd+#simm24),Ra`	$I/O[R_d + signed\ imm_{24}] \leftarrow R_a$	

Table A-8: *WvFEv3* Integer arithmetic instructions

Type	Assembly Format	Operation(s)	Flag(s)
Addition	`add Rd,Ra,Rb`	$R_d \leftarrow R_a + R_b$	*zer neg ovf*
	`add Rd,Ra,#simm24`	$R_d \leftarrow R_a + signed\ imm_{24}$	
Subtraction	`sub Rd,Ra,Rb`	$R_d \leftarrow R_a - R_b$	*zer neg ovf*
	`add Rd,Ra,#simm24`	$R_d \leftarrow R_a - signed\ imm_{24}$	
Arithmetic Shift	`sra Rd,Ra,Rb`	$R_d \leftarrow R_a \gg R_b$	*zer neg ovf*
	`sra Rd,Ra,#simm24`	$R_d \leftarrow R_a \gg signed\ imm_{24}$	
	`sla Rd,Ra,Rb`	$R_d \leftarrow R_a \ll R_b$	
	`sla Rd,Ra,#simm24`	$R_d \leftarrow R_a \ll signed\ imm_{24}$	
Logical Shift	`srl Rd,Ra,Rb`	$R_d \leftarrow R_a \gg R_b$	*zer neg ovf*
	`srl Rd,Ra,#simm24`	$R_d \leftarrow R_a \gg signed\ imm_{24}$	
	`sll Rd,Ra,Rb`	$R_d \leftarrow R_a \ll R_b$	
	`sll Rd,Ra,#simm24`	$R_d \leftarrow R_a \ll signed\ imm_{24}$	

Table A-8: Continued

Type	Assembly Format	Operation(s)	Flag(s)
Logical AND	`and Rd,Ra,Rb`	$R_d \leftarrow R_a \wedge R_b$	*zer neg ovf*
	`and Rd,Ra,#simm24`	$R_d \leftarrow R_a \wedge signed\ imm_{24}$	
Logical OR	`or Rd,Ra,Rb`	$R_d \leftarrow R_a \vee R_b$	*zer neg ovf*
	`or Rd,Ra,#simm24`	$R_d \leftarrow R_a \vee signed\ imm_{24}$	
Logic XOR	`xor Rd,Ra,Rb`	$R_d \leftarrow R_a \oplus R_b$	*zer neg ovf*
	`xor Rd,Ra,#simm24`	$R_d \leftarrow R_a \oplus signed\ imm_{24}$	
Logarithmic	`log2 Rd,Ra`	$Rd \leftarrow \log_2 R_a$	*zer*

Table A-9: *WvFEv3* numerical conversion and approximation instructions

Type	Assembly Format	Operation(s)	Flag(s)
Integer to floating point conversion	`fp Rd,Ra,Rb`	$R_d \leftarrow (float)[R_a * 2^{R_b}]$	None
	`fp Rd,Ra,#simm24`	$R_d \leftarrow (float)[R_a * 2^{signed\ imm_{24}}]$	
Floating point to integer conversion[7]	`int Rd,Ra,Rb`	$R_d \leftarrow (int)[R_a * 2^{R_b}]$	None
	`int Rd,Ra,#simm24`	$R_d \leftarrow (int)[R_a * 2^{signed\ imm_{24}}]$	
Square root approximation	`fsqrta Rd,Ra`	$\frac{1}{3}\sqrt{R_a} < R_d \leq \frac{3}{2}\sqrt{R_a}$	*finv fexp*
Reciprocal approximation	`frcpa Rd,Ra`	$\frac{3}{4}R_a^{-1} < R_d \leq \frac{3}{2}R_a^{-1}$	*fdze fexp*

[7] Rounds to nearest integer

Table A-10: *WvFEv3* butterfly address calculation instructions

Type	Assembly Format	Operation(s)	Flag(s)
Radix-2 butterfly addresses	`bfly2 Rd,Ra,Rb,Wn`	$R_d \leftarrow \text{butterfly}(R_d, R_a, R_b, W_n, 1)$	*bli bso*
Radix-4 butterfly addresses	`bfly4 Rd,Ra,Rb,Wn`	$R_d \leftarrow \text{butterfly}(R_d, R_a, R_b, W_n, 2)$	*bli bso*

Table A-11: *WvFEv3* floating point unit instructions

Type	Assembly Format	Operation(s)	Flag(s)
Addition	fadd Rz,Rx,Ry	$R_z \leftarrow R_x + R_y$	*fzer fneg fovf*
	fadd Rz,Rx,-Ry	$R_z \leftarrow R_x - R_y$	*fexp*
Subtraction	fsub Rz,Rx,Ry	$R_z \leftarrow R_x + R_y$	*fzer fneg fovf*
	fsub Rz,Rx,-Ry	$R_z \leftarrow R_x + R_y$	*fexp*
Multiply	fmul Rz,Rx,Ry	$R_z \leftarrow R_x * R_y$	*fzer fneg fovf*
	fmul Rz,Rx,-Ry	$R_z \leftarrow R_x * -R_y$	*fexp*
	fmul Rz,Rx,@(Vt+Ry)	$R_z \leftarrow R_x * Table[V_t][R_y]$	
	fmul Rz,Rx,-@(Vt+Ry)	$R_z \leftarrow R_x * -Table[V_t][R_y]$	
Multiply Accumulate	fmac Rz,Rx,Ry	$R_z \leftarrow R_z + R_x * R_y$	*fzer fneg fovf*
	fmac Rz,Rx,-Ry	$R_z \leftarrow R_z + R_x * -R_y$	*fexp*
	fmac Rz,Rx,@(Vt+Ry)	$R_z \leftarrow R_z + R_x * Table[V_t][R_y]$	
	fmac Rz,Rx,-@(Vt+Ry)	$R_z \leftarrow R_z + R_x * -Table[V_t][R_y]$	

Table A-11: Continued

Type	Assembly Format	Operation(s)	Flag(s)
Load virtual table configuration	tbl Vt,Rx,Ry	$Table[V_t].Descriptor \leftarrow R_x$ $Table[V_t].Overlap\ Value \leftarrow R_y$	None
No Operation	nop	$No\ operation$	None

APPENDIX B

WVFEV3 SOFTWARE ALGORITHMS

Figure B-1: Complex radix-4 FFT in *WvFEv3* assembly

```
.Title Complex radix-4 FFT
.text
     .include "macro.asm"
     .global _fft_radix4           ; export entry point

_fft_radix4:
     ; First read arguments off stack and shuffle around the
     ; program counter so that the return is nice and clean
     ; In the unused space between pops, fetch the sine/cosine
     ; configuration and load it.

     pop  r15,f31                  ; Pop the program counter off the stack
     ld   f0,(_sine_table)         ; Load address of sine table
```

Figure B-1: Continued

```
pop  r15,r2                    ; Pop the waveform address off the stack
ld   f1,(_sine_config)         ; Load pointer to sine configuration
pop  r15,r1                    ; Pop the twiddle factor cadence off stack
ld   f2,@(f1+#0)               ; Load sine configuration word
pop  r15,r0                    ; Pop the size of the waveform off the stack
ld   f3,@(f1+#1)               ; Load the sine table overlap value
push r15,f31                   ; Put the program counter back on the stack

; Finished with reading in arguments
; Load the rest of the table configurations

or   f2,f2,f0
ld   f1,(_cosine_config)
ld   r8,#0
ld   f2,@(f1+#0)          || tbl t0,f2,f3
ld   f3,@(f1+#1)
```

Figure B-1: Continued

```
or   f2,f2,f0
add  r3,r2,r0

; Start calculating butterfly addresses
bfly4     r4,r0,r8,w0          || tbl    t1,f2,f3
bfly4     r5,r0,r8,w1
bfly4     r6,r0,r8,w2
bfly4     r7,r0,r8,w3
ld   f3,@(r4+r2)
ld   f4,@(r5+r2)
ld   f5,@(r6+r2)
ld   f6,@(r7+r2)
ld   f7,@(r4+r3)
ld   f8,@(r5+r3)               || fadd   f11,f3,f5
ld   f9,@(r6+r3)               || fsub   f3,f3,f5
ld   f10,@(r7+r3)              || fadd   f5,f4,f6
```

Figure B-1: Continued

```
        add   r8,r8,#4                  || fadd   f12,f7,f9
        log2  r1,r1                     || fadd   f13,f8,f10
        nop                             || fsub   f8,f8,f10

        ; Main butterfly radix-4 loop
    _butterfly4:
        ld    r9,r5                     || fsub   f10,f7,f9
        ld    r10,r6                    || fsub   f7,f4,f6
        ld    r11,r7                    || fadd   f20,f11,f5
        sla   f0,r4,r1                  || fadd   f21,f12,f13
        bfly4      r5,r0,r8,w1          || fadd   f14,f3,f8
        sla   f1,f0,#1                  || fsub   f16,f11,f5
        bfly4      r6,r0,r8,w2          || fsub   f15,f10,f7
        add   f2,f0,f1                  || fsub   f17,f12,f13
        ld    @(r4+r2),f20              || fsub   f18,f3,f8
        ld    @(r4+r3),f21              || fadd   f19,f10,f7
```

Figure B-1: Continued

```
bfly4     r4,r0,r8,w0          || fmul   f22,f14,@(t1+f0)
bfly4     r7,r0,r8,w3          || fmul   f23,f15,@(t1+f0)
add  r8,r8,#4                  || fmul   f24,f16,@(t1+f1)
ld   f3,@(r4+r2)               || fmul   f25,f17,@(t1+f1)
ld   f4,@(r5+r2)               || fmul   f26,f18,@(t1+f2)
ld   f5,@(r6+r2)               || fmul   f27,f19,@(t1+f2)
ld   f6,@(r7+r2)               || fmac   f22,f15,@(t0+f0)
ld   f7,@(r4+r3)               || fmac   f23,f14,-@(t0+f0)
ld   f8,@(r5+r3)               || fmac   f24,f17,@(t0+f1)
ld   f9,@(r6+r3)               || fmac   f25,f16,-@(t0+f1)
ld   f10,@(r7+r3)              || fmac   f26,f19,@(t0+f2)
ld   @(r9+r2),f22              || fmac   f27,f18,-@(t0+f2)
ld   @(r9+r3),f23              || fadd   f11,f3,f5
ld   @(r10+r2),f24             || fsub   f3,f3,f5
ld   @(r10+r3),f25             || fadd   f5,f4,f6
ld   @(r11+r2),f26             || fadd   f12,f7,f9
```

Figure B-1: Continued

```
ld   @(r11+r3),f27            || fadd   f13,f8,f10
brch !bli,_butterfly4         || fsub   f8,f8,f10 ; level of b-flies complete?
add  r1,r1,#2
brch !bso,_butterfly4    ; butterfly seed out of range?
return    r15
.end
```

Figure B-2: Complex FFT result reversal in *WvFEv3* assembly

```
.Title Complex FFT result reversal
.text
     .psize    56
     .include "macro.asm"
     .global _fft_radix4_descramble          ; export entry point

_fft_radix4_descramble:
     pop  r15,f31                  ; Pop the program counter off the stack
     nop
     pop  r15,r12                  ; Pop the descramble buffer address off the stack
     nop
     pop  r15,r2                   ; Pop the scrambled buffer address of the stack
     nop
     pop  r15,r0                   ; Pop the size of the waveform off the stack
     ld   r8,#0
     push r15,f31                  ; Put the program counter back on the stack
```

Figure B-2: Continued

```
        add   r3,r2,r0
        add   r13,r12,r0
        bfly4       r4,r0,r8,w0
        bfly4       r5,r0,r8,w1
        bfly4       r6,r0,r8,w2
        bfly4       r7,r0,r8,w3
        add   r8,r8,#4

    _descramble_butterfly4:
        ld    f3,@(r4+r2)
        ld    f4,@(r5+r2)
        ld    f5,@(r6+r2)
        ld    f6,@(r7+r2)
        ld    f7,@(r4+r3)
        ld    f8,@(r5+r3)
        ld    f9,@(r6+r3)
```

Figure B-2: Continued

```
ld   f10,@(r7+r3)
bfly4     r4,r0,r8,w0
bfly4     r5,r0,r8,w1
bfly4     r6,r0,r8,w2
bfly4     r7,r0,r8,w3
add  r8,r8,#4
ld   @(r12+#0),f3
ld   @(r12+#1),f4
ld   @(r12+#2),f5
ld   @(r12+#3),f6
ld   @(r13+#0),f7
ld   @(r13+#1),f8
ld   @(r13+#2),f9
ld   @(r13+#3),f10
add  r12,r12,#4
add  r13,r13,#4
```

Figure B-2: Continued

```
        brch !bli,_descramble_butterfly4
        return    r15
        .end
```

Figure B-3: Simultaneous real FFT result unscramble in *WvFEv3* assembly

```
.Title FFT real result unscramble
.text
     .psize    56
     .include "macro.asm"
     .global _fft_unscramble       ; export entry point
;
;    This program unscrambles the output of a dual FFT. Two real functions
;    h(t) and g(t) have been FFT'ed simultaneously by placing one data set (h)
;    in the real locations, and the other data set (g) in the imaginary locations.
;    Address of data is on stack, size of input data vactor is on stack.
;    Second channel is assumed to be stored at Address+size.
;    Address of output is on the stack.

.data
_norm_factor:
     .int 0x3A000000                          ; 1./(2.*1024.)
```

Figure B-3: Continued

```
        .int 0x39000000                         ; 1./(2.*4096.)
        .int 0x38000000                         ; 1./(2.*16384.)
_half:
        .float    0.5                           ; 1./2.
        .int 0
        .int 0
        .int 0

.text

_fft_unscramble:
;
        ; First read arguments off stack and shuffle around the
        ; program counter so that the return is nice and clean.
;
        pop  r15,f31                    ; Pop the program counter off the stack
```

Figure B-3: Continued

```
nop
pop  r15,r2                    ; Pop the output buffer address off the stack
nop
pop  r15,r1                    ; Pop the input buffer address off the stack
nop
pop  r15,r0                    ; Pop the size of the waveform off the stack
nop
push r15,f31                   ; Put the program counter back on the stack

; Finished with reading in arguments
sra  r14,r0,#1                 ; r14 = N/2
ld   r6,(_half)                ; get pointer to 1./2.
add  r4,r1,#1                  ; r4 = SRC_A (starts at BUFFER+1)
sub  r13,r0,#1                 ; r13 = N-1 (N_MINUS1)
add  r3,r2,r14                 ; 2nd output buffer address (SAVE_B)
add  r8,r1,r13                 ; 2nd input buffer (SRC_B)
```

Figure B-3: Continued

```
        ld   f16,@(r6+#0)              ; load the constant f16 = 0.5
        sra  r13,r13,#1                ; r13 = (N-1)/2
    ;
    ;   Figure out the normalization factor, i.e., 1/N
    ;
        ld   r7,(_norm_factor)
        nop
        sub  r5,r0,#1024
        brch !zer, next1
        brch unc, found
    next1:
        add  r7,r7,#1
        nop
        sub  r5,r0,#4096
        brch !zer, next2
        brch unc, found
```

Figure B-3: Continued

```
next2:
      add   r7,r7,#1
      nop
      nop
      nop
found:
      ld    f16,@(r7+#0)                    ; get normalization factor (1/(2*N))
      ld    f0,@(r4+#0)                     ; load data from source buffer1
      ld    f1,@(r8+#0)
      ld    f4,@(r4+r0)
      ld    f5,@(r8+r0)
      nop
      nop
      nop
      ld    @(r6+#3),f16
_loop:
```

Figure B-3: Continued

```
add   r2,r2,#1            || fmul f2,f0,f16   ; normalize data by 1/(2*N)
add   r3,r3,#1            || fmul f3,f1,f16
add   r4,r4,#1            || fmul f6,f4,f16
sub   r8,r8,#1            || fmul f7,f5,f16
nop
nop
ld    f0,@(r4+#0)         || fadd f9,f2,f3
ld    f1,@(r8+#0)         || fsub f10,f3,f2
ld    f4,@(r4+r0)         || fadd f11,f6,f7
ld    f5,@(r8+r0)         || fsub f12,f6,f7
nop
nop
ld    @(r2+#0),f9                       ; store results
ld    @(r3+r0),f10
ld    @(r3+#0),f11
ld    @(r2+r0),f12
```

Figure B-3: Continued

```
sub  r13,r13,#1                    ; decrement loop counter
brch !zer, _loop                   ; done looping?
return    r15
.end
```

BIBLIOGRAPHY

[1] W. S. Kurth, D. L. Kirchner, G. B. Hospodarsky, D. A. Gurnett, P. Zarka, R. Ergun and S. Bolton, "A wave investigation for the juno mission to jupiter," in *AGU Fall Meeting Abstracts,* 2008, pp. 1680.

[2] C. A. Kletzing, W. Kurth, M. Acuna, R. Torbert, R. Thorne, V. Jordanova, S. Bounds, C. Smith, O. Santolik and R. Pfaff, "The electric and magnetic field instrument suite with integrated science (EMFISIS) on the radiation belt storm probes," in *AGU Fall Meeting Abstracts,* 2006, pp. 0332.

[3] D. A. Gurnett, W. S. Kurth, G. B. Hospodarsky, A. M. Persoon, T. F. Averkamp, B. Cecconi, A. Lecacheux, P. Zarka, P. Canu, N. Cornilleau-Wehrlin, P. Galopeau, A. Roux, C. Harvey, P. Louarn, R. Bostrom, G. Gustafsson, J. -. Wahlund, M. D. Desch, W. M. Farrell, M. L. Kaiser, K. Goetz, P. J. Kellogg, G. Fischer, H. -. Ladreiter, H. Rucker, H. Alleyne and A. Pedersen, "Radio and Plasma Wave Observations at Saturn from Cassini's Approach and First Orbit," *Science,* vol. 307, pp. 1255-1259, February 25, 2005.

[4] A. Gatherer, T. Stetzler, M. McMahan and E. Auslander, "DSP-based architectures for mobile communications: past, present and future," *Communications Magazine, IEEE,* vol. 38, pp. 84-90, 2000.

[5] B. Ackland and C. Nicol, "High performance DSPs-what's hot and what's not?" in *Proceedings of the 1998 International Symposium on Low Power Electronics and Design,* 1998, pp. 1-6.

[6] R. Schneiderman, "DSPs Evolving in Consumer Electronics Applications," *Signal Processing Magazine, IEEE,* vol. 27, pp. 6-10, 2010.

[7] L. J. Karam, I. AlKamal, A. Gatherer, G. A. Frantz, D. V. Anderson and B. L. Evans, "Trends in multicore DSP platforms," *Signal Processing Magazine, IEEE,* vol. 26, pp. 38-49, 2009.

[8] Project Reliability Group. (1990, *Jet Propulsion Laboratory D-5703 Reliability Analysis Handbook* (1st ed.).

[9] F. McDonald, J. Naugle, S. Uyeda, M. Kamogawa, K. Alverson, R. Hooper, E. Foufoula-Georgiou and F. M. Meyer, "Discovering Earth's Radiation Belts: Remembering Explorer 1 and 3," *EOS,* vol. 89, pp. 361–363, 2008.

[10] T. F. Tascione, "Introduction to the Space Environment, 2nd Edition," pp. 1-172, 1994.

[11] S. R. Elkington, M. K. Hudson and A. A. Chan, "Resonant acceleration and diffusion of outer zone electrons in an asymmetric geomagnetic field," *J. Geophys. Res.,* vol. 108, pp. 1116, 03/14, 2003.

[12] G. D. Reeves, "Radiation Belt Storm Probes: A New Mission for Space Weather Forecasting," *Space Weather,* vol. 5, pp. S11002, 11/02, 2007.

[13] S. Matousek, "The Juno New Frontiers mission," *Acta Astronaut.,* vol. 61, pp. 932-939, 2007.

[14] D. A. Gurnett, W. S. Kurth and F. L. Scarf, "Plasma Waves Near Saturn: Initial Results from Voyager 1," *Science,* vol. 212, pp. 235-239, April 10, 1981.

[15] D. A. Gurnett, W. S. Kurth and F. L. Scarf, "Plasma Wave Observations Near Jupiter: Initial Results from Voyager 2," *Science,* vol. 206, pp. 987-991, November 23, 1979.

[16] J. Seon, L. A. Frank, W. R. Paterson, J. D. Scudder, F. V. Coroniti, S. Kokubun and T. Yamamoto, "Observations of slow-mode shocks in Earth's distant magnetotail with the Geotail spacecraft," *J. Geophys. Res.,* vol. 101, pp. 27383-27398, 1996.

[17] E. Brigham 1940-, E. Oran Brigham. and E. O. Brigham, *The Fast Fourier Transform and its Applications.* Englewood Cliffs, N.J.: Englewood Cliffs, N.J. : Prentice Hall, 1988.

[18] M. Berg, J. -. Wang, R. Ladbury, S. Buchner, H. Kim, J. Howard, K. LaBel, A. Phan, T. Irwin and M. Friendlich, "An Analysis of Single Event Upset Dependencies on High Frequency and Architectural Implementations within Actel RTAX-S Family Field Programmable Gate Arrays," *Nuclear Science, IEEE Transactions on,* vol. 53, pp. 3569-3574, 2006.

[19] F. Koebel and J. Coldefy, "SCOC3: A space computer on a chip: An example of successful development of a highly integrated innovative ASIC," in *Proceedings of the Conference on Design, Automation and Test in Europe,* Dresden, Germany, 2010, pp. 1345-1348.

[20] R. Berger, A. Dennis, D. Eckhardt, S. Miller, J. Robertson, D. Saridakis, D. Stanley, M. Vancampen and Q. Nguyen. (2007, 2007). RAD750 SpaceWire-enable flight computer for lunar reconnaissance orbiter. *1*pp. 1.

[21] G. R. Goslin, "A guide to using field programmable gate arrays (FPGAs) for application-specific digital signal processing performance," *Xilinx Inc,* 1995.

[22] Honeywell International Inc., "HXSR01608 2M x 8 Static RAM Datasheet," vol. 2011, pp. 11, 11/01, 2009.

[23] K. Blackburn, B. Lessard, D. Kirchner and W. Kurth, "Controlling Low Frequency Interference from Direct Energy Transfer Spacecraft Power Systems," 2011.

[24] C. Lattner, "LLVM: A compilation framework for lifelong program analysis & transformation," in 2004, pp. 75-75.

[25] W. D. Peterson, "WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores," *OpenCores.Org,* 2002.

[26] D. Flynn, "AMBA: enabling reusable on-chip designs," *Micro, IEEE,* vol. 17, pp. 20-27, 1997.

[27] E. Salminen, V. Lahtinen, K. Kuusilinna and T. Hamalainen, "Overview of bus-based system-on-chip interconnections," in *Circuits and Systems, 2002. ISCAS 2002. IEEE International Symposium on,* 2002, pp. II-372-II-375 vol.2.

[28] D. Elsner, "Using as, the GNU assembler. Free Software Foundation," *Inc., March,* 1993.