Rochester Institute of Technology

RIT Scholar Works

Theses

8-1-2007

Hardware and software optimization of fourier transform infrared spectrometry on hybrid-FPGAs

Dmitriy Bekker

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

Bekker, Dmitriy, "Hardware and software optimization of fourier transform infrared spectrometry on hybrid-FPGAs" (2007). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Hardware and Software Optimization of Fourier Transform Infrared Spectrometry on Hybrid-FPGAs

by

Dmitriy L. Bekker

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Supervised by

Dr. Marcin Lukowiak
Assistant Professor
Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August 2007

Approved By:

Dr. Marcin Lukowiak
Assistant Professor, Department of Computer Engineering
Rochester Institute of Technology

Dr. Muhammad Shaaban
Associate Professor, Department of Computer Engineering
Rochester Institute of Technology

Dr. Jean-Francois Blavier
Science Division
NASA Jet Propulsion Laboratory (Caltech)

Thesis Release Permission Form

Rochester Institute of Technology

Kate Gleason College of Engineering

Title: Hardware and Software Optimization of Fourier Transform Infrared Spectrometry on Hybrid-FPGAs

I, Dmitriy L. Bekker, hereby grant permission to the Wallace Memorial Library to reproduce my thesis in whole or in part.

Dmitriy L. Bekker

Date

Dedication

To my family and friends, for their love and support.

Acknowledgments

I would like to thank my RIT advisers, Dr. Lukowiak and Dr. Shabaan, for teaching me most of what I know in Computer Engineering and for their help and support on this thesis. They made sure I had everything necessary to carry out this work, including top of the line development hardware and software rarely seen in universities yet. I would also like to thank Dr. Blavier from JPL for working with me on this project from the very start and for his dedicated help on the FTIR spectrometry algorithm and the system requirements. I also thank Paula Pingree, Gary Block, Charles Norton, and Abdullah Aljabri from Instrument and Science Data Systems Division at JPL who got me involved with hybrid-FPGAs and their applications to space flight, and Ben Jones from Xilinx for his support on the APU-FPU and co-processor integration.

Abstract

With the increasing complexity of today's spacecrafts, there exists a concern that the on-board flight computer may be overburdened with various processing tasks. Currently available processors used by NASA are struggling to meet the requirements of scientific experiments [1, 2]. A new computational platform will soon be needed to contend with the increasing demands of future space missions.

Recently developed hybrid field-programmable gate arrays (FPGA) offer the versatility of running diverse software applications on embedded processors while at the same time taking advantage of reconfigurable hardware resources, all on the same chip package. These tightly coupled HW/SW systems consume less power than general-purpose single-board computers (SBC) and promise breakthrough performance previously impossible with traditional processors and reconfigurable devices.

This thesis takes an existing floating-point intensive data processing algorithm, used for on-board spacecraft Fourier transform infrared (FTIR) spectrometry, ports it into the embedded PowerPC 405 (PPC405) processor, and evaluates system performance after applying different hardware and software optimizations and architectural configurations of the hybrid-FPGA. The hardware optimizations include Xilinx's floating-point unit (FPU) for efficient single-precision floating-point calculations and a dedicated single-precision dot-product co-processor assembled from basic floating-point operator cores. The software optimizations include utilizing a non-ANSI single-precision math library as well as IBM's PowerPC performance libraries recompiled for double-precision arithmetic only.

The outcome of this thesis is a fully functional, optimized FTIR spectrometry algorithm implemented on a hybrid-FPGA. The computational and power performance of this system is evaluated and compared to a general-purpose SBC currently used for spacecraft data processing. Suggestions for future work, including a dual-processor concept, are given.

Contents

Dedication . . . . **iii**

Acknowledgments . . . . **iv**

Abstract . . . . **v**

Assumptions . . . . **xiii**

Acronym Glossary . . . . **xiv**

1 Introduction **1**

1.1 Background and Motivation 1

1.2 Thesis Description 3

1.3 Overview 5

2 Fourier Transform Infrared Spectrometry **6**

2.1 Data Collection 6

2.2 Data Processing Steps 8

2.3 First Evaluation on FPGAs 10

3 Xilinx Virtex-4 FX Hybrid-FPGA **13**

3.1 Xilinx Virtex-4 FPGA Overview 13

3.2 The Hybrid-FPGA Concept 14

3.2.1 Motivation 14

3.2.2 Previous Work 15

3.3 Virtex-4 FX Hybrid-FPGA 16

3.3.1 PowerPC 405 Embedded Hard Processor 17

3.3.2 MicroBlaze Soft Processor 19

3.3.3 System Interfaces 20

3.3.4 Auxiliary Processor Unit Controller 22

3.4 ML410 Development Board 25

4 FTIR Base System **28**
4.1 Generating a Hardware Platform 28
4.2 Configuring Software 33
4.2.1 Software Structure 33
4.2.2 Porting FORTRAN to C for PPC405 Embedded Processor 35
4.2.3 Modifying Converted Code 36
4.3 Checking Processing Results 37
4.4 Initial Performance Evaluation 38

5 Software Optimizations **41**
5.1 Removing Double-precision Math Library Calls 41
5.2 Linking IBM PowerPC Performance Libraries 42
5.3 V2P SW Optimization Results 45

6 Hardware Optimizations **46**
6.1 Xilinx APU Floating-point Unit 46
6.1.1 *F2C* Compatibility 49
6.1.2 Recompiling *Perflib* for Double-precision Only 49
6.1.3 Accuracy and Performance Evaluation 52
6.2 Dot-product Hardware Co-processor 53
6.2.1 Concept 53
6.2.2 Load/Store Unit Design 55
6.2.3 Dot-product Core Design 57
6.2.4 Behavioral Simulation 60
6.2.5 System Deployment 61
6.2.6 Software Considerations 62
6.2.7 Accuracy and Performance Evaluation 66
6.3 FPU / Dot-product Compatibility and Integration 67
6.3.1 Hardware Issues 67
6.3.2 Accuracy and Performance Evaluation 71
6.4 Increased FPU System Frequencies 73
6.4.1 Motivation 74
6.4.2 Meeting Timing 75
6.4.3 Accuracy and Performance Evaluation 77

7 Result Analysis **78**
7.1 Performance Evaluation 78
7.2 Power Estimation 80

8 Conclusions and Future Work **82**

Bibliography **86**

A Building *f2c* for EDK **89**

B Recompiling IBM PowerPC *Perflib* for Double-precision Optimization Only . **92**

C Select Code Listings **94**
C.1 Makefile for *libf2c* 94
C.2 Makefile (main) for Double-precision Only *Perflib* 98
C.3 Makefile (fpopt) for Double-precision Only *Perflib* 100
C.4 Original Top-level FTIR Spectrometry Source (*matmos-ipp.f*) 103
C.5 No I/O Top-level FTIR Spectrometry Source (*matmos-ipp-chk-noio.f*) 106
C.6 Partial FTIR Spectrometry C-source (*xilinx-matmos-ipp-chk_orig.c*) 109
C.7 The *dotprod* Function (from *xilinx-matmos-ipp-chk_orig.c*) 117
C.8 The *dotprod* Function with HW Support (*dotprod.c*) 119
C.9 FCM Load/Store Module (*apu_fcm_ldst.v*) 121
C.10 Dot-product Module (*fp_dot_prod.vhd*) 131

List of Figures

1.1 A conceptual drawing of the MARVEL spacecraft [1] with the MATMOS instrument [3] gathering data through the Martian atmosphere as it points towards the Sun . . . 2
1.2 Solar occultation observations [1, 2] . . . 2

2.1 Simulated Mars occultation spectra [2] . . . 7
2.2 An ideal Fourier Transform Spectrometer with only the axial rays shown [1] . . . 7
2.3 The time-domain-sampled interferogram is re-sampled to the path-difference domain [1] . . . 9
2.4 The BAE RAD750 SBC and the Xilinx V2P board [2] . . . 11

3.1 Spatial vs. temporal computing [15] . . . 15
3.2 GARP block diagram [17] . . . 16
3.3 PPC405 core inside a V4FX hybrid-FPGA [19] . . . 17
3.4 PPC405 pipeline utilization by instruction type [19] . . . 18
3.5 MicroBlaze core block diagram [21] . . . 19
3.6 The APU integrates directly into the processor pipeline and decodes soft coprocessor supported instructions [30] . . . 24
3.7 The ML410 development board [32] . . . 26
3.8 ML410 interfaces block diagram [32] . . . 27

4.1 Simulated time-domain interferograms used as input data [2] . . . 29
4.2 PPC405 base system configuration . . . 29
4.3 FTIR base system diagram (main components only) . . . 31
4.4 The spectrum produced from simulated interferogram data [2] . . . 34
4.5 Software flow of the FTIR spectrometry algorithm . . . 34
4.6 The spectrum produced by the FTIR base system on the ML410 board . . . 37
4.7 Profiling results for FTIR base system . . . 39

5.1 Profiling results after single-precision math functions optimization . . . 42

5.2 Profiling results for system with SP math functions and *Perflib* 44

6.1 System diagram of the APU floating-point unit co-processor [34] 47
6.2 FTIR system with FPU co-processor 48
6.3 Profiling results for system with APU-FPU, SP math functions, and DP *Perflib* 52
6.4 Cycles needed to complete one iteration of dot-product loop 55
6.5 FCM load/store unit interface to FCB and dot-product co-processor 55
6.6 FCM load/store unit state machine 56
6.7 Resource usage vs. latency for the multiply and add floating-point operators 58
6.8 Dot-product core block diagram 59
6.9 Dot-product core state machine 60
6.10 Dot-product core behavioral simulation 61
6.11 FTIR system with dot-product co-processor 63
6.12 FTIR system with dot-product and FPU co-processors 68
6.13 Profiling results for system with dot-product and FPU co-processors, SP math functions, and DP *Perflib* 73
6.14 DCM configuration for a 266.67 MHz system 74
6.15 FPU core placement before (a,c) and after (b,d) manual constraining 76
6.16 The spectrum produced by the high-frequency FTIR system on the ML410 board 77

8.1 Dual-core concept targeting ML410 development board 84

List of Tables

2.1 Reduction in data volume due to on-board data processing 10

2.2 Results of NASA JPL research task comparing FTIR spectrometry execution times between the BAE RAD750 SBC and the Xilinx V2P board [2] 11

3.1 MicroBlaze Performance for Xilinx FPGAs (with multiplier and barrel shifter) [20] 19

3.2 Most common buses used in Xilinx embedded processor systems [28] 23

4.1 Device utilization summary for FTIR base system 32

4.2 Execution times for FTIR base system on ML410 board and comparison to NASA JPL V2P research 38

5.1 Execution times for system with single-precision math functions 41

5.2 Execution times for system with SP math functions and *Perflib* 43

5.3 Comparison of V4FX and V2P results for system with SP math functions and *Perflib* 45

6.1 Xilinx's APU floating-point unit v3.0 [34]. 47

6.2 Device utilization summary for FTIR system with FPU co-processor 50

6.3 Optimized floating-point routines provided by *Perflib* 51

6.4 Execution times for system with APU-FPU, SP math functions, and DP *Perflib* 51

6.5 Dot-product Core Summary 62

6.6 Device utilization summary for FTIR system with dot-product co-processor 64

6.7 Execution times for system with dot-product co-processor, SP math functions, and *Perflib* 67

6.8 Instruction op-codes decoded by the APU controller [25] 69

6.9 Device utilization summary for FTIR system with dot-product and FPU co-processors 70

6.10 Execution times for system with dot-product and FPU co-processor, SP math functions, and DP *Perflib* 72

6.11 Dot-product core testing with smaller data set 72
6.12 Execution times for a high-frequency system with APU-FPU, SP math functions, and DP *Perflib* 77

7.1 Execution times for all V4FX FTIR system builds 78
7.2 Execution times for high-frequency FTIR system on ML410 board and comparison to NASA JPL V2P board 79
7.3 Execution times for high-frequency FTIR system on ML410 board and comparison to BAE RAD750 SBC 80
7.4 Estimated power consumption of V4FX60, V2P, and BAE RAD750 81

Assumptions

The target environment is Windows XP Service Pack 2. The development software used is:

- Xilinx ISE 9.1.03i with IP Update 3
- Xilinx Platform Studio (XPS) 9.1.02i (includes EDK and SDK) with GNU-GCC 4.1.1 compiler (Xilinx edition)
- Xilinx ChipScope Pro 9.1.03i
- FORTRAN to C Translator (f2c) version 20060506
- IBM PowerPC Perflib version 1.1

Windows™is trademark of Microsoft Corp.
ISE™, ChipScope™, and Virtex™are trademarks of Xilinx Inc.
PowerPC™is trademark of IBM.

Acronym Glossary

APU Auxiliary Processor Unit. A controller embedded inside the PPC405 core (V4FX only) that manages FCB-connected co-processors. See subsection 3.3.4 on page 22 for a complete description.

ASIC Application Specific Integrated Circuit. An integrated circuit that is customized for a particular use.

BRAM Block RAM. On-chip memory inside Xilinx FPGAs.

DCR Device Control Register. See subsection 3.3.3 on page 20.

DMIPS Dhrystone MIPS. A common representation of the Dhrystone benchmark.

EDK Embedded Development Kit. Xilinx design tool for developing processor-based FPGA systems.

F2C FORTRAN-to-C Converter. A tool for automatic conversion of FORTRAN code to C code. Its use is detailed in subsection 4.2.2 on page 35.

FCB Fabric Co-processor Bus. See subsection 3.3.3 on page 20.

FTIR Fourier Transform Infrared. Refers to IR spectroscopy, dealing with the infrared region of the electromagnetic spectrum. See Chapter 2 on page 6 for a complete description.

FPGA Field Programmable Gate Array. A programmable device with reconfigurable hardware resources.

FPU Floating Point Unit. A dedicated hardware co-processor that handles floating-point arithmetic.

FSL Fast Simplex Link. See subsection 3.3.3 on page 20.

GCC GNU Compiler Collection. The C compiler used in this thesis.

GMACS Giga multiply-accumulate operations per second.

ISA Instruction set architecture.

LMB Local Memory Bus. See subsection 3.3.3 on page 20.

MARVEL Mars Volcanic Emissions and Life. A proposed Mars Scout mission. See section 1.1 on page 1 for a complete description.

MATMOS Mars Atmospheric Trace Molecule Spectroscopy. A Fourier transform spectrometer; the primary science instrument on the MARVEL spacecraft. See section 1.1 on page 1 for a complete description.

OCM On-chip Memory. See subsection 3.3.3 on page 20.

OPB On-chip Peripheral Bus. See subsection 3.3.3 on page 20.

Perflib IBM Performance Libraries. A set of floating-point and string manipulation routines that significantly outperform those supplied in GCC.

PLB Processor Local Bus. See subsection 3.3.3 on page 20.

PPC405 The PowerPC 405 embedded processor inside the V2P and V4FX hybrid-FPGAs. See subsection 3.3.1 on page 17 for a complete overview.

PPC750 The PowerPC 750 processor inside the BAE RAD750 SBC.

RAD750 A radiation-hardened SBC developed by BAE. It utilized the PPC750 processor.

SBC Single-board computer.

SDK Software Development Kit. An Eclipse-based Xilinx design tool for developing embedded software.

XCL Xilinx Cache Link. See subsection 3.3.3 on page 20.

XPS Xilinx Platform Studio. Contains EDK, SDK, and a Xilinx version of GCC.

Chapter 1

Introduction

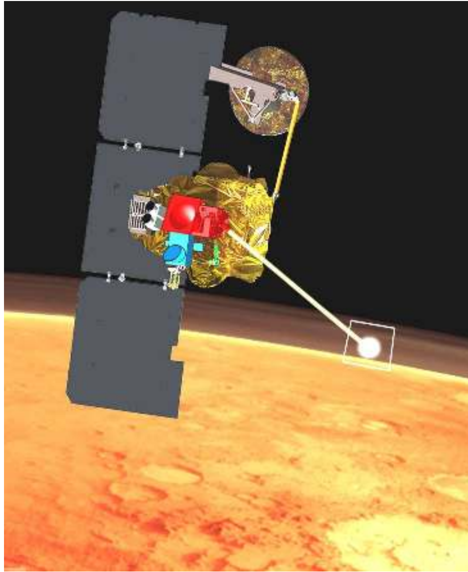
1.1 Background and Motivation

From the dawn of the Space Age, on-board flight computers have played an ever increasing role in the exploration of the universe. Although constantly improving, the performance of computers used for space flight typically falls a decade or more behind that of modern PCs. This is due to the stringent requirements imposed on space-bound processors and computer peripherals for tolerating the higher dose of radiation that is present outside the Earth's atmosphere. Radiation hardening requires special fabrication and packaging techniques that adversely affects the performance of these components. Although necessary, such measures can limit the scope of scientific experiments to be carried out by the spacecraft.

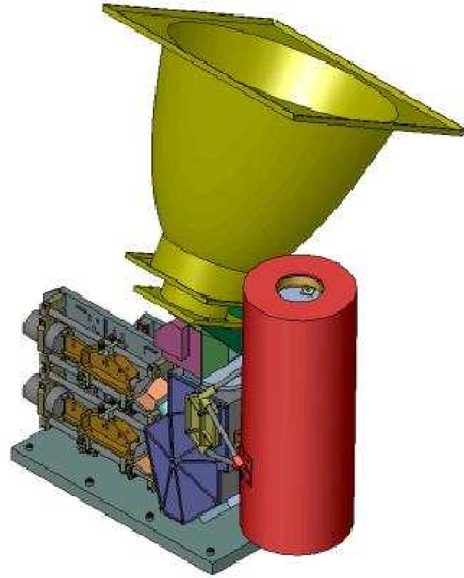
The proposed Mars Scout Mission known as MARVEL[1] is a prime example where data processing demands really push the limits of currently available radiation hardened processors. The goal of this mission is to find evidence of active Martian volcanism and life [3]. Although not funded past the proposal stage, the instruments and scientific experiments from MARVEL are applicable to many similar missions that intend to analyze the chemical composition of an atmosphere. The primary science instrument on the MARVEL spacecraft is a solar occultation Fourier Transform Spectrometer (FTS) called MATMOS[2] used for very sensitive detection of trace gases such as CH_4 and N_2O that might be produced by life or volcanism (see Figure 1.1 on the following page) [2].

The MATMOS instrument will measure the infra-red spectrum of direct sunlight and produce large volumes of data in two short, 3-minute bursts during its on-orbit observations of sunrise and sunset (see Figure 1.2 on the next page). The remaining orbit time of 112 minutes is available for on-board data processing to reduce data volume prior to down-link. The steps involved in the data processing are computationally intensive and

[1] Mars Volcanic Emissions and Life (MARVEL)
[2] Mars Atmospheric Trace Molecule Spectroscopy (MATMOS)



(a) MARVEL spacecraft

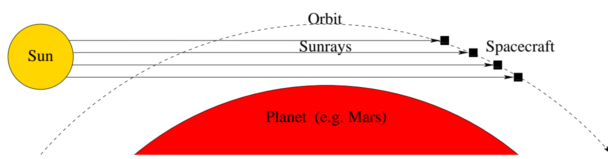


(b) MATMOS instrument

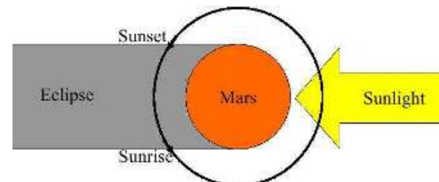
Figure 1.1: A conceptual drawing of the MARVEL spacecraft [1] with the MATMOS instrument [3] gathering data through the Martian atmosphere as it points towards the Sun

carry a heavy emphasis on floating-point calculations. Currently, two BAE RAD750 (radiation hardened) processors are required to perform such processing. Although these processors have flown successfully on numerous NASA missions, they consume significant power (20 W in a SBC package), and require extensive interface logic [1, 2].

As missions become more complex with more demanding requirements, the traditional approach of using radiation hardened SBCs will no longer be the optimal. MATMOS requires two processors, but other missions may require six or eight, all working simultaneously, all consuming power and adding to the net weight of the spacecraft. A new,



(a) One observation (sunset)



(b) Two opportunities per orbit

Figure 1.2: Solar occultation observations [1, 2]

more efficient computational platform is urgently needed; one that can execute complex software and at the same time efficiently implement algorithms in hardware like an application-specific integrated circuit (ASIC). Hybrid-FPGAs fit this description as they typically have one or more embedded processor cores immersed in a sea of reconfigurable logic. Although not yet radiation hardened, strong efforts are currently being made to qualify these devices for space flight [4, 5].

In 2005, NASA Jet Propulsion Laboratory looked into the possibility of using the Xilinx Virtex-II Pro (V2P) hybrid-FPGA as the computational platform for MATMOS. That study concluded that the V2P could not keep up with the data processing when performed in software on the embedded PPC405 processor core. The study further suggested that the lack of a hardware FPU on the V2P is responsible for its slow processing times as all floating-point calculations are emulated in the software [2].

In recent years, hybrid-FPGAs have evolved significantly. Currently, the Xilinx Virtex-4FX (V4FX) hybrid-FPGA is the most advanced of its kind that is available commercially. The V4FX brings with it new capabilities for custom co-processor integration, including a soft core single precision FPU with full compiler support. This, along with other improvements, warrants that MATMOS data processing be evaluated again on this new platform with optimizations not tried in the past.

1.2 Thesis Description

This thesis takes the MATMOS data processing software for FTIR spectrometry and, after porting it to the PPC405 processor, implements various hardware and software optimizations that reduce the overall execution time. Although the main focus is on the V4FX FPGA, the older V2P is also targeted for comparison. The results presented are actual run times on fully functional hybrid-FPGA systems built with Xilinx's Embedded Development Kit (EDK)[3].

The FTIR spectrometry software is written entirely in FORTRAN and is ported to the PPC405 processor with the help of the FORTRAN-to-C Converter (*f2c*) and its supporting libraries [6]. Configuring *f2c* and its libraries to generate valid PPC405 code requires a specific set of compilation and linking options which are discussed in this thesis. Once ported, the FTIR spectrometry software is carefully studied in order to identify areas of

[3]This thesis developed out of an internship in the summer of 2006 with the Instrument and Science Data Systems Division at NASA Jet Propulsion Laboratory and continues the work presented in [1].

improvement. Profiling tools are used to locate bottlenecks and computationally intensive portions of the algorithm.

Two software optimizations are evaluated as part of this work:

- Use of non-ANSI single-precision math library functions
- Use of IBM Performance Libraries (*Perflib*)

The techniques above are compatible with both the V2P and the V4FX. The first technique requires modification of the FTIR spectrometry code to use single-precision math function calls where acceptable. Single-precision arithmetic is performed much faster than double-precision thus reducing the overall execution time. *Perflib* is a set of libraries that replaces string manipulation functions and standard floating-point emulation with hand-optimized routines written specifically for the PPC405 processor [7]. Xilinx EDK provides a version of *Perflib* compiled for string, single, and double-precision optimization. This thesis additionally provides a build of *Perflib* that only optimizes double-precision floating-point arithmetic and discusses where it is applicable.

Most of the work, however, deals exclusively with the V4FX and is focused on hardware optimizations, their integration with the system, and compatibility with the software. Different system architectures, memory configurations, and bus frequencies are evaluated to find the optimal solution. The new soft-core single-precision Xilinx FPU and its integration with the auxiliary processor unit (APU) controller is studied extensively. Additionally, a custom HW accelerator that optimizes single-precision dot-product calculations is presented and implemented alongside the FPU thus demonstrating multiple co-processors sharing the same physical hardware interface - a capability not previously tested by Xilinx.

Overall, this thesis achieves a 10x reduction in execution time of the FTIR spectrometry algorithm when compared to a software-only implementation on the V4FX60 FPGA. Only one of two available PPC405 cores is utilized and with minimal changes to the FTIR spectrometry software. This is the fastest implementation of the algorithm on an FPGA platform to date. Although, in its current form, incapable to meet the data processing requirements for MATMOS, future improvements to the software as a well as a dual-core design (presented in this thesis) will come very close.

1.3 Overview

This thesis starts with an overview of Fourier transform infrared spectrometry and its past implementation on an FPGA, presented in Chapter 2. Chapter 3 introduces the Virtex-4 FX hybrid-FPGA with a detailed explanation of its main architectural features. Chapter 4 presents an all-software implementation of the FTIR spectrometry algorithm on the V4FX60 FPGA and analyzes the initial performance results. Chapters 5 and 6 describe software and hardware optimizations that reduce the execution time of the FTIR base system, presenting and analyzing performance results along the way. Chapter 7 takes a look at all of the implementations done in this thesis and compares their computational and power performance to that of a general-purpose SBC used for spacecraft data processing. Finally, Chapter 8 concludes the thesis with a brief overview of the work accomplished and presents suggestions for future research in this area.

Chapter 2

Fourier Transform Infrared Spectrometry

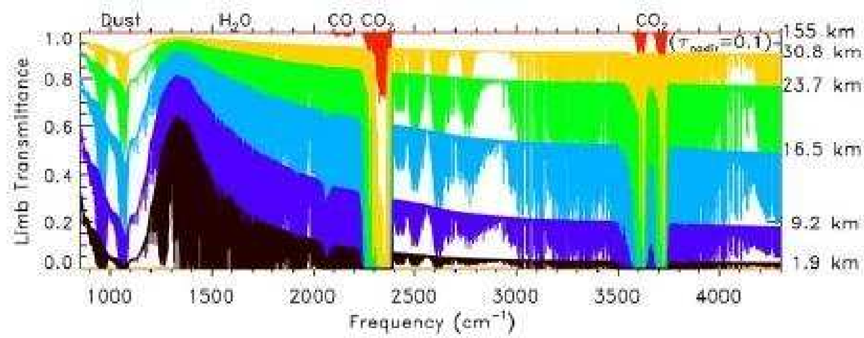
This chapter introduces Fourier transform infrared (FTIR) spectrometry as it is applied to the MATMOS instrument and the MARVEL mission. A description is given of how the solar occultation data is collected, the steps involved in the data processing after collection, and the necessary memory requirements. The results of past work done with the FTIR spectrometry algorithm and the V2P FPGA are also presented.

2.1 Data Collection

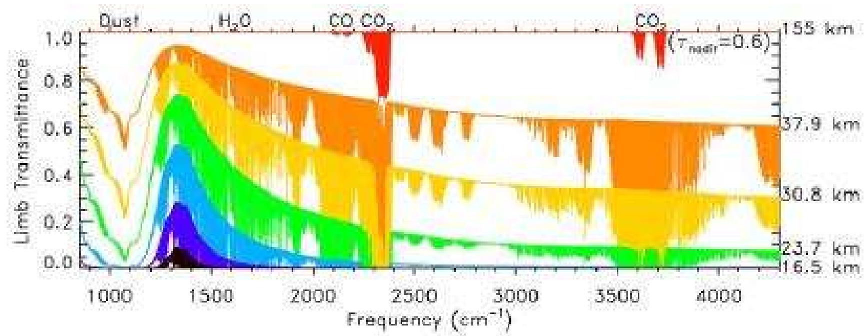
The MATMOS instrument measures the 850-4300 cm^{-1} region of the infra-red spectrum of sunlight as it shines through the Martian atmosphere. This measurement is done at a high 0.02 cm^{-1} spectral resolution necessary to identify certain trace gases. MATMOS records roughly 26 spectra per occultation, with each containing 172,500 spectral elements (see Figure 2.1 on the following page). The duration of an occultation is between 78 and 169 seconds, thus requiring that each spectrum be collected in 3.0 to 6.5 seconds [1].

The spectrum is recorded with a Fourier Transform Spectrometer (FTS), a Michelson interferometer in which the optical path difference of light rays is continuously varied with moving mirrors (see Figure 2.2 on the next page). Using photovoltaic detectors, this modulated light is converted to an electric signal known as an interferogram. To attain the high 0.02 cm^{-1} spectral resolution, the FTS needs a maximum optical path difference (MOPD) of 25 cm. However, for the best quality, velocity of the scanner moving the mirrors should be constant at the point of zero path difference (ZPD). Thus, for MATMOS, the optical path is increased to 50 cm and a dual-sided interferogram is recorded, with the ZPD in the middle of the scan [1].

Given that the shortest spectrum collection time is 3 seconds, and estimating that the



(a) Low dust conditions



(b) High dust conditions

Figure 2.1: Simulated Mars occultation spectra [2]

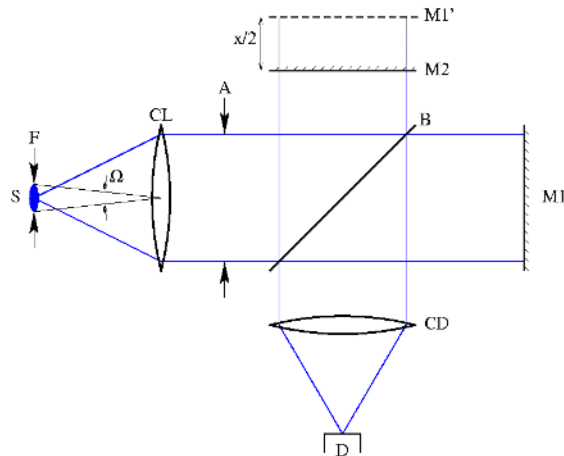


Figure 2.2: An ideal Fourier Transform Spectrometer with only the axial rays shown [1]

scanner turn-around time is 0.5 seconds (to reverse direction), the velocity with which the mirror must travel the 50 cm distance is 20 cm/s. Given this velocity, *v*, and an optical wavenumber, *s*, the corresponding frequency, *f*, recorded at the detector is calculated as:

$$f = s \times v \tag{2.1}$$

With the equation above, the highest detectable wavenumber (4300 cm^{-1}) corresponds to the frequency of 86 kHz. The frequency of the reference laser (internal to the FTS) used to measure the path difference is 129 kHz, given that its wavenumber is 6450 cm^{-1}. Thus, in accordance with the Nyquist Theorem, the minimum sampling frequencies of Analog-to-Digital Converters (ADC) used to record each interferogram are 172 kHz for the solar data and 258 kHz for the laser signal. Oversampling factors of 1.1 and 1.5 are used for the MATMOS FTS in order to improve the quality of the sampled signal. This sets the sampling frequencies to 192 kHz and 384 kHz, respectively [1].

The MATMOS FTS utilizes three separate detectors in the process of collecting occultation spectra. An *HgCdTe* detector is used to collect longer wavelengths (12 μm - 5 μm) and an *InSb* detector collects shorter wavelengths (5 μm - 2 μm). An *Ge* detector is used to collect the reference laser interferogram. The *HgCdTe* and *InSb* detectors use 24-bit ADCs while the *Ge* detector uses a 16-bit ADC. Data from the 24-bit ADCs is stored in 32-bit format to match common computer architectures [1].

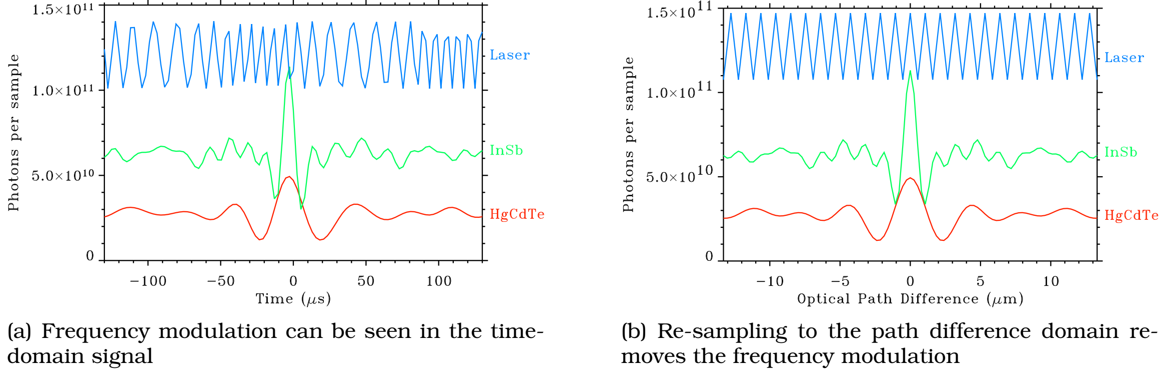
Thus, given that there are two detectors which output 32-bit data at 192 kHz, a detector that outputs 16-bit data at 384 kHz, a scanner duty cycle of 5/6 (mirror moves for 2.5 out of 3.0 seconds with 0.5 seconds turn-around time), and two 3-minute occultations to observe, the amount of data collected on every orbit is:

$$((2 \times 32bit \times 192000\frac{samples}{sec}) + (16bit \times 384000\frac{samples}{sec})) \times (5/6) \times 2 \times (3min \times 60\frac{sec}{min}) = 5.53Gbit$$

That is equivalent to 659 Mbytes. This will fit in the MATMOS memory bank which has the capacity of 2 Gbytes [1].

2.2 Data Processing Steps

The amount of data collected by the MATMOS FTS cannot be transmitted to Earth in its entirety. The data must first be processed by the on-board instrument computer and



(a) Frequency modulation can be seen in the time-domain signal

(b) Re-sampling to the path difference domain removes the frequency modulation

Figure 2.3: The time-domain-sampled interferogram is re-sampled to the path-difference domain [1]

compressed to a reasonable amount for transmission. The steps involved in this process are summarized in this section.

The ADCs used to convert solar data from the *HgCdTe* and *InSb* detectors cannot be triggered externally. The conversion process runs continuously and produces a time-domain data stream with each value corresponding to a point in time. Through re-sampling, this data stream must be converted to the path-difference domain in order to remove frequency modulation in the time-domain caused by variations of the mirror velocity (see Figure 2.3) [1, 8].

Re-sampling reduces the number of points from $192kHz \times 2.5sec = 480,000$ for each solar detector to $2^{18} = 262,144$ points for the *HgCdTe* detector and $2^{19} = 524,288$ for the *InSb* detector. Additionally, laser interferogram data is no longer needed after re-sampling and can be freed from memory. This data accounts for 1/3 of all raw data, as shown in the calculation below [1].

$$\frac{16bit \times 384000\frac{sample}{sec}}{(2 \times 32bit \times 192000\frac{samples}{sec}) + (16bit \times 384000\frac{sample}{sec})} = \frac{1}{3}$$

Thus, removing the laser interferogram data reduces data volume by 3/2. From the initial raw interferogram data, the net reduction due to re-sampling is:

$$\frac{2 \times 480000}{262144 + 524288} \times \frac{3}{2} = 1.83$$

Next, phase correction (using convolution) is performed in order to make the interferogram symmetrical about the ZPD. Being symmetrical, the two halves of the interferogram

DATA PROCESSING STEP	REDUCTION FACTOR	DATA SIZE (MBYTES)
Raw Interferogram	- - -	659.18
Interferogram Re-sampling	1.83	360.21
Phase Correction	2.00	180.10
Fast Fourier Transformation	6.10	29.53
Spectra Averaging	2.00	14.76
Lossless Compression	1.80	8.20
NET DATA REDUCTION	**80.37**	**8.20**

Table 2.1: Reduction in data volume due to on-board data processing

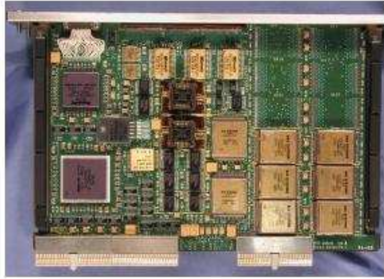
can be averaged together further reducing the amount of data by a factor of 2. Following this step, the spectrum is computed with a fast Fourier transform (FFT) which produces an output with a smaller dynamic range than the interferogram. This resulting data can be represented with 16 bits instead of the 32 bits originally used for the interferogram (2x data reduction). Additional data reduction is attained from reducing the spectral range. The computed spectrum has a range of 5243 $cm^{-1}$for each solar detector, yet the data desired is in the 850-4300 cm^{-1} range and combined into one channel. Altogether, for the two solar detectors, the FFT reduces data volume by the factor computed below [1].

$$2 \times 2 \times \frac{5243cm^{-1}}{(4300cm^{-1} - 850cm^{-1})} = 6.1$$

The final two steps in reducing the data volume are spectra averaging and compression. Averaging scans taken above the atmosphere reduces data volume by a factor of 2. Lossless compression achieves a 1.8 reduction in the data volume. The combined reduction in the volume of data to be transmitted to Earth is summarized in Table 2.1[1].

2.3 First Evaluation on FPGAs

In 2005, NASA Jet Propulsion Laboratory evaluated the performance of the FTIR spectrometry algorithm on the V2P FPGA and compared it to the radiation hardened BAE RAD750 SBC. The FPGA hosted a PPC405 CPU implementation of the algorithm without putting any portions in the reconfigurable hardware. Figure 2.4 on the following page gives a brief overview of the two processing platforms. Table 2.2 on the next page presents the results from that research task. The results clearly indicate that the V2P falls far behind the RAD750. Furthermore, not even the RAD750 can process the data fast enough to meet the time requirement of 112 minutes. Thus, not one but two RAD750 SBCs are required for the MATMOS instrument.



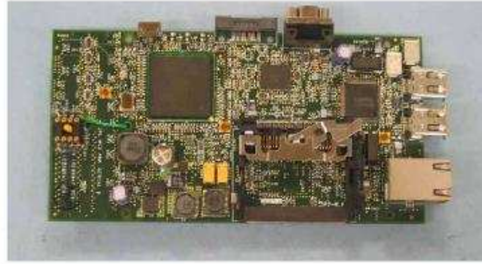


Figure 2.4: The BAE RAD750 SBC and the Xilinx V2P board [2]

Processor:	RAD750	Xilinx	Xilinx
Operating System:	VxWorks	Linux	Linux
Clock Speed:	133 MHz	300 MHz	300 MHz
Memory:	128 MB	128 MB	128 MB
Software Component			w/Perflib
Reject Dark Interferograms	<1	<1	<1
Interferogram Re-sampling	69	3404	780
Non-Linearity Correction	1	14	4
Phase Correction	42	488	142
Fast Fourier Transformation	15	272	90
Spectra Averaging	2	(10)	(3)
Lossless Compression	1	1	1
Total (112 min available)	**130 min**	**4200 min**	**1020 min**

Table 2.2: Results of NASA JPL research task comparing FTIR spectrometry execution times between the BAE RAD750 SBC and the Xilinx V2P board [2]

There are a couple of reasons why the V2P takes so much longer to process the data, namely cache size, instruction issue rate, and hardware floating-point support. The RAD750 SBC contains a variant of the PowerPC750 (PPC750) processor with 32 Kbytes data and 32 Kbytes instruction L1 cache [9]. The V2P has a PPC405 processor with 16 Kbytes data and 16 Kbytes instruction L1 cache [10]. In an application such as this, the tremendous data processing requires frequent accesses to external memory. A larger L1 cache means that more data can fit in this high speed memory resulting in fewer cache misses and fewer accesses to main memory, which carries with it a high latency [1].

Another reason for the RAD750 performing so much better is the instruction issue rate of the PPC750 processor. As this processor is a superscalar, multiple instructions can be fetched, dispatched, and executed in one cycle. In particular, the PPC750 can fetch up to four instructions, dispatch up to two instructions, and execute up to six instructions per clock cycle. The PPC405 processor is a scalar processor with a single issue execution pipeline [11, 12].

The lack of a hardware floating-point unit in the V2P, however, is an even larger performance hit than the size of the cache or the instruction issue rate. The RAD750 has a hardware floating-point unit which performs all floating-point operations and reports the results back to the processor. The V2P FPGA does not have a hardware floating-point unit requiring that all floating-point operations be emulated in the software. Software emulation is inherently slower than computation done with dedicated hardware. The steps in the FTIR spectrometry algorithm require significant floating point calculations. Even with its higher processor frequency (300 MHz vs. 133 MHz) and special optimization library (IBM *Perflib*), the V2P still lags far behind the RAD750. Although its small size/weight and low power consumption is very favorable for future space flight, in its basic configuration the V2P does not meet MATMOS's data processing requirements [1].

The only way a hybrid-FPGA solution could come close to the performance of the RAD750 is by integrating a dedicated floating-point unit, implementing certain portions of the algorithm in the FPGA fabric, and optimizing the code to make the best use of the hardware resources on the FPGA. This, along with the advanced capabilities of the Xilinx Virtex-4FX FPGA (described in the next chapter), is the motivation for this thesis.

Chapter 3

Xilinx Virtex-4 FX Hybrid-FPGA

New innovations in the field of reconfigurable computing have recently led to the development of FPGAs with multiple embedded processors. Such a computing platform, known as a hybrid-FPGA, offers the versatility of running diverse software applications on embedded processors while at the same time taking advantage of tightly coupled reconfigurable hardware resources. This allows for the exploitation of coarse-grain data-parallelism via the software as well as fine-grain data-parallelism via the reconfigurable hardware. The recently released Virtex-4 FX (V4FX) FPGA is a true hybrid-FPGA with up to two embedded PPC405 processors. Additionally, the auxiliary processor unit (APU) controller inside each PPC405 core can interface to custom hardware co-processors implemented in the FPGA fabric. This efficient, high speed, low latency interface feeds directly into the processor instruction pipeline, allowing for the extension of the instruction set architecture with user defined instructions (instruction augmentation). With these and other features, the Xilinx Virtex-4 FX FPGA delivers breakthrough performance previously impossible with traditional processors and reconfigurable devices.

This chapter introduces the Virtex-4 FPGA, describes the architecture of the V4FX hybrid-FPGA, and provides an overview of its main features. In particular, processor choices, system buses, and the new auxiliary processor unit controller are described in detail. The chapter concludes with an overview of the target platform for this thesis - the Xilinx ML410 development board.

3.1 Xilinx Virtex-4 FPGA Overview

Devices in Xilinx's Virtex family of FPGAs are known as Platform FPGAs because of the features they deliver for use in system-on-chip (SoC) applications. Built in low power, 90 nm technology, the Virtex-4 comes in three flavors that are tailored to specific applications.

The three Virtex-4 platforms, in the order of release, are:

- LX: Logic optimized
- SX: Signal processing optimized
- FX: Full-featured, with embedded processors

The LX platform has the most general purpose logic resources and is targeted at logic-intensive applications, such as complex interfaces and advanced digital systems. The SX platform has the largest amount of specialized digital signal processing (DSP) blocks, making it ideal for signal processing applications such as filter design or digital image processing. The full-featured FX platform has the highest amount of on-chip memory resources and comes with up to two embedded processors, making it a true hybrid-FPGA. This is a great platform for implementing complete embedded systems that require a robust interface between software and custom hardware [13].

In all three platforms, the Virtex-4 FPGA comes with a rich set of common features, including:

- 500 MHz system clocking
- 1+ Gbps IOBs (input/output blocks)
- 256 giga multiply-accumulate operations per second (GMACS) DSP circuitry (18x18)
- Block RAM with built-in error checking and correction (ECC)

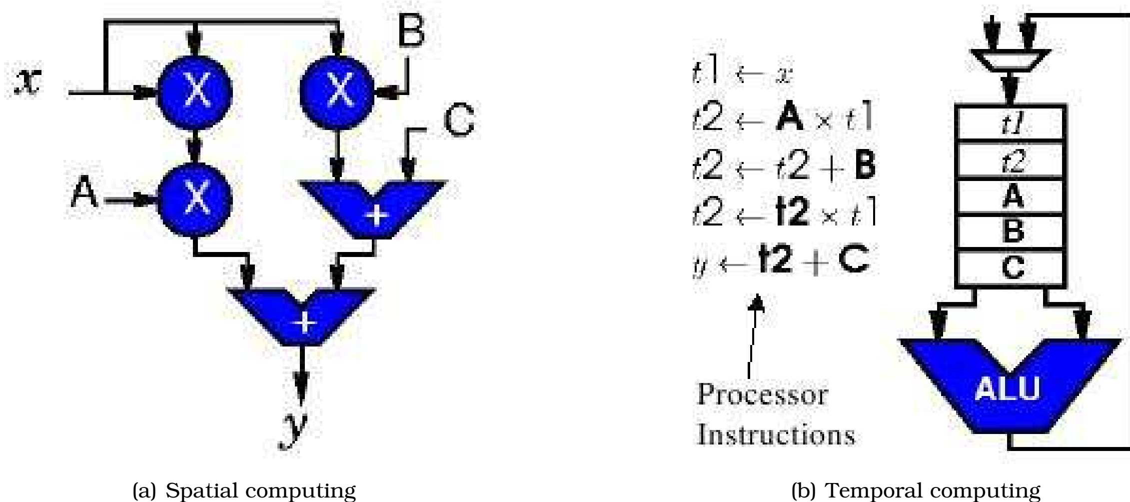
The features above, and others not listed, make the Virtex-4 a very powerful FPGA platform that is suitable for a wide range of applications. According to Xilinx, this FPGA consumes much less power than other competing 90 nm FPGAs, making it ideal for low-power designs [14].

3.2 The Hybrid-FPGA Concept

A hybrid-FPGA is a device that contains one or more processor cores inside a sea of reconfigurable logic resources. The Virtex-4 FX FPGA is a true hybrid-FPGA as it contains up to two PowerPC 405 processor cores embedded inside the FPGA fabric.

3.2.1 Motivation

The hybrid-FPGA concept emerged from the trade-off that developers had to make when selecting a computing platform to meet their processing requirements. The trade-off was



(a) Spatial computing

(b) Temporal computing

Figure 3.1: Spatial vs. temporal computing [15]

between using FPGAs that took advantage of spatial computing and general-purpose processors (GPPs) that took advantage of temporal computing (see Figure 3.1). Under spatial computing, the functionality and connectivity of hardware elements is fixed. Under temporal computing, a processor runs a fixed set of instructions while sharing a functional unit. While spatial computing may offer more efficient implementations of certain algorithms (due to dedicated hardware), temporal computing is more flexible and can accommodate complex, irregular tasks [15].

The hybrid-FPGA offers the benefits of both spatial and temporal computing by including a processor core among the reconfigurable logic and providing an interface between the software and hardware domains. This is an ideal platform for high performance computing applications that are characterized by complex software tasks which interface with algorithms implemented in hardware.

3.2.2 Previous Work

A distant relative of the hybrid-FPGA concept is the RISC4005/R16 FPGA processor implementation by Philip Freidin (Fliptronics) in 1991 [13]. The RISC4005/R16 features a 16 bit RISC processor core implemented on Xilinx's XC4005 FPGA. The instruction set architecture (ISA) is similar to AMD 29000 RISC and can be extended. This design, however, uses 75% of the available resources on the FPGA and leaves little room for other hardware components [16]. The V2P and the V4FX FPGAs feature embedded processors that do not take additional logic resources, thus leaving much room for custom hardware

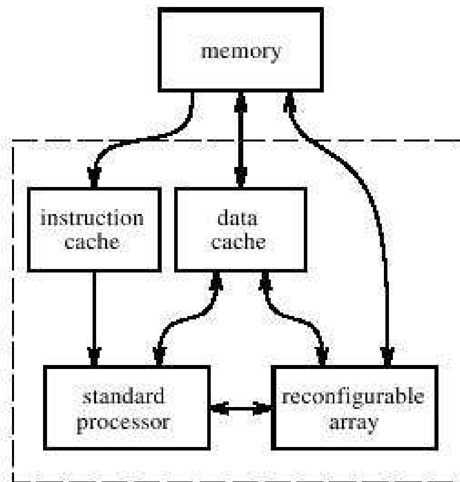


Figure 3.2: GARP block diagram [17]

designs.

The GARP processor (Berkeley, 1997) is architecturally very similar to the V2P and the V4FX. GARP contains a MIPS-II-based processor and a reconfigurable array all on one chip (see Figure 3.2). Both the processor and the reconfigurable array have independent access to the off-chip memory. GARP's reconfigurable array consists of blocks similar to the CLBs of the Xilinx 4000 series. By today's standards, such reconfigurable logic is rather simplistic as it does not offer some of the more sophisticated features such as DSP blocks, dedicated multipliers, and multi-gigabit transceivers (MGTs). Furthermore, GARP was never actual built and exists only on paper (although the full layout has been done). However, GARP does offer some features that the V2P and the V4FX do not have, the most notable being fast, on-the-fly reconfiguration. The main processor in GARP can issue a command to very quickly reconfigure the hardware. The V2P and the V4FX may need seconds to do a full reconfiguration [17].

3.3 Virtex-4 FX Hybrid-FPGA

The V4FX FPGA is a relatively new device and is considered a true hybrid-FPGA as it comes with up to two embedded PowerPC 405 processor cores. In addition to the standard features on all Virtex-4 FPGAs, the V4FX also includes:

- 450 MHz, 680 DMIPS PowerPC 405 processing
- 622 Mbps - 6.5 Gbps multi-gigabit transceivers (RocketIO MGTs)

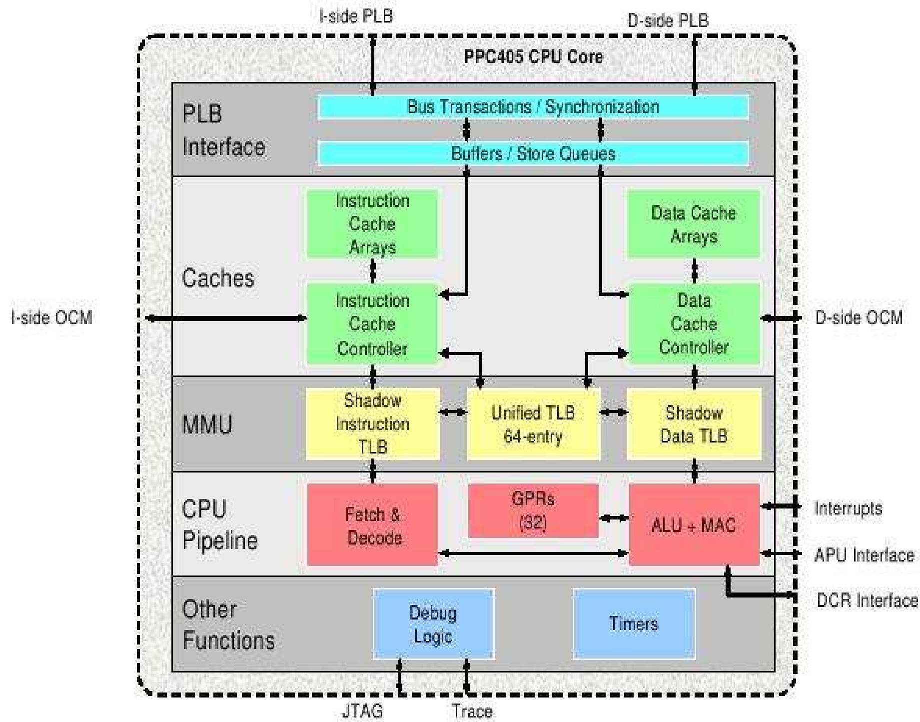


Figure 3.3: PPC405 core inside a V4FX hybrid-FPGA [19]

- Auxiliary processing unit (APU) controller

The following sections will discuss V4FX architecture, including the processing choices (hard PowerPC 405 or soft MicroBlaze), the system interfaces, and the APU controller. The APU controller is a key component in the V4FX hybrid-FPGA as it provides an efficient interface to HW accelerators in the FPGA fabric and supports ISA extension (instruction augmentation) [18].

3.3.1 PowerPC 405 Embedded Hard Processor

Developed by IBM, up to two PPC405 processors are embedded in the V4FX hybrid-FPGA as hard IP cores (do not take additional resources). Natively, the PPC405 is a big-endian processor although it can switch to the little-endian mode. The PPC405 features (as shown in Figure 3.3) [19]:

- Up to 450 MHz operation
- 32-bit Harvard architecture (RISC, separate data and instruction caches / interfaces)

Pipeline Stage -------------------- Instruction Type	Instruction Fetch Buffer	Decode / Dispatch	Execute	Write-Back	Load Write-Back
Integer / Logical	Fetch	Decode / Dispatch	Execute	Write GPR	
Branch	Fetch	Predict / Decode / Dispatch	Execute		
Store	Fetch	Decode / Dispatch	Address Gen	Cache write	
Load	Fetch	Decode / Dispatch	Address Gen	Cache read	Write GPR

Figure 3.4: PPC405 pipeline utilization by instruction type [19]

- Five-stage single issue execution pipeline
- 32 general-purpose registers (GPRs)
- 16 KB 2-way set-associative instruction and data caches
- Write-back (default) or write-through policy for data cache
- 64-entry unified HW TLB memory management unit (MMU)
- Variable page sizes (1KB - 16KB)
- Block RAM (BRAM) interface via on-chip memory (OCM) controllers

The PPC405 five-stage pipeline consists of a fetch, decode, execute, write-back, and load write-back stages (see Figure 3.4). The fetch and decode stages ensure a well fed instruction pipeline with up to two instructions in the fetch queue. The single execute unit contains the GPR register file, the arithmetic logic unit (ALU), and the multiply-accumulate (MAC) unit, but it does not include a floating-point unit (FPU). The PPC405 can natively handle 32-bit PowerPC integer instructions only. However, the APU controller provides an interface to execute instructions that are not part of the PPC405 ISA in custom co-processors, which may include a FPU in the FPGA fabric [19].

FPGA	Size	Clock Freq.	DMIPS	Performance
Virtex-5	1,010 LUTs	210 MHz	240	1.15 DMIPS/MHz
Virtex-4	1,809 LUTs	160 MHz	184	1.15 DMIPS/MHz
Spartan-3 (Performance)	1,843 LUTs	100 MHz	115	1.15 DMIPS/MHz
Spartan-3 (Size)	1,350 LUTs	100 MHz	92	0.92 DMIPS/MHz

Table 3.1: MicroBlaze Performance for Xilinx FPGAs (with multiplier and barrel shifter) [20]

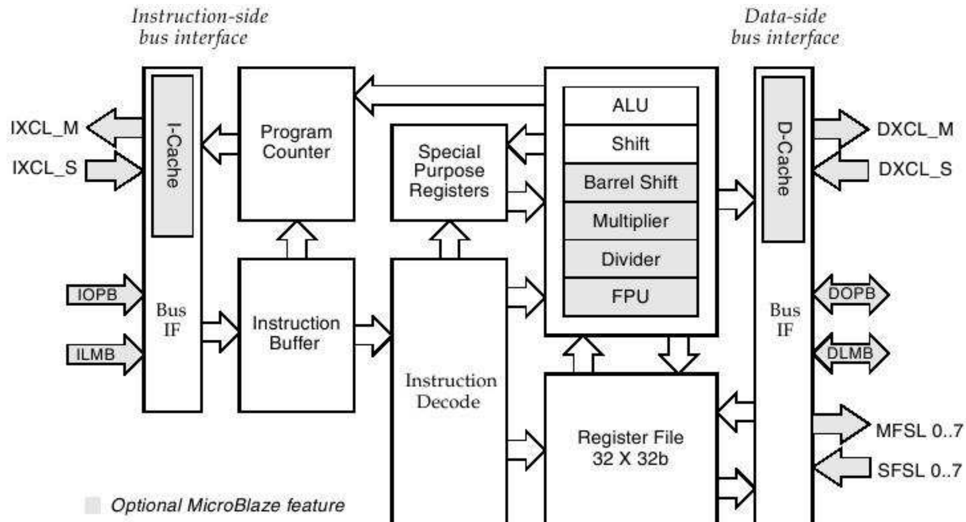


Figure 3.5: MicroBlaze core block diagram [21]

3.3.2 MicroBlaze Soft Processor

An alternative to the hard PPC405 processor is the soft MicroBlaze core. Soft cores are implemented from reconfigurable resources and are thus very portable between different FPGA families. Table 3.1 shows the size and performance of the MicroBlaze processor on different Xilinx FPGAs. As a reference, the V4FX60 FPGA (targeted in this thesis) has 50,560 LUTs.

Like the PPC405, the MicroBlaze is a 32-bit RISC processor. Except for the amount of FPGA resources, there is no limit to how many MicroBlaze processors can be instantiated on a single FPGA. The MicroBlaze is a big-endian processor with the features listed below [21]. The MicroBlaze core block diagram is shown in Figure 3.5.

- 32-bit Harvard architecture (RISC, separate data and instruction caches / interfaces)

- Three-stage (area optimized) or five-stage (performance optimized) single issue execution pipeline
- 32 general-purpose registers (GPRs)
- Up to 64 KB 1-way associative instruction and data caches via Xilinx CacheLink (XCL) interface
- Write-through policy for data cache
- Block RAM (BRAM) interface via local memory bus (LMB) controllers
- Optional features (selectable by user):
  - Hardware barrel shifter
  - Hardware multiplier
  - Hardware divider
  - Single-precision FPU

The depth of the pipeline can be configured as either three-stage (area optimized) or five-stage (performance optimized). The three-stage pipeline has only the fetch, decode, and execute stages. The five-stage pipeline has the fetch, decode, execute, access memory, and write-back stages. The optional FPU is a very popular features as it provides seamless support for single-precision floating-point operations and delivers up to 50 MFLOPs peak performance. The MicroBlaze processor also support hardware co-processor integration via the fast simplex link (FSL) interface (also available on PPC405 systems). The FSL channels provide unidirectional high speed data streams and interface directly into the processor pipeline. In a PPC405 system, the APU controller with the fabric co-processor bus (FCB) is a better choice for communicating with hardware co-processors as this interface supports instruction decoding and larger data transfers. The MicroBlaze processor can host up to eight FSL channels (each with one input and output port) [21].

3.3.3 System Interfaces

This section describes the system interfaces that are typical to a SoC implementation on a Xilinx V4FX FPGA. Both PPC405 and MicroBlaze interfaces are presented. As both processors are in Harvard architecture, the interfaces listed below (except DCR, FCB, and FSL) have an instruction and data side denoted with the letters 'I' and 'D', respectively,

in front of the interface name (for example, IPLB, DPLB, IOPB, DOPB, etc...). Master and slave interfaces are denoted with the letters 'M' and 'S', respectively, in front of the interface name (for example, MOPB, SOPB, etc...).

PLB (Processor Local Bus - PPC405 only). This bus is for high speed and high performance peripherals, such as memory. The processor accesses this bus through the instruction and data cache controllers. Separate 32-bit address and 64-bit data buses are provided. The PLB interface can have up to 16 masters and 16 slaves, with arbitration [19, 22].

OPB (On-chip Peripheral Bus - PPC405 and MicroBlaze). This bus is for less demanding peripherals that do not need to communicate with the processor very frequently and do not need high bandwidth (for example, UART controller). Peripherals on the OPB communicate with the processor either directly (for MicroBlaze) or via a bridge to the PLB (for PPC405). The OPB can have up to 16 masters and 16 slaves, with arbitration. This bus supports 32-bit transactions [19, 23].

DCR (Device Control Register - PPC405 only). This 32-bit data, 10-bit address bus is for accessing on-chip configuration and IP control registers. Through this bus, the processor can control IP cores directly. Multiple IPs are connected to this bus in a daisy-chain fashion (up to 16 IPs in a chain) [19, 24].

OCM (On-chip Memory - PPC405 only). This bus is for connecting local on-chip memory to the processor. Non-cacheable access to this memory usually occurs in a 1:1 or 2:1 time frame ratio compared to access to cached memory through the cache controllers (depending on processor frequency and amount of memory). This bus supports 32-bit bi-directional data-side memory transfers and 64-bit uni-directional instruction-side memory transfers. There is no bus arbitration, so one processor master/slave pair is allowed. On-chip memory connected through the OCM interface often holds interrupt routines that require low-latency access or frequently used data arrays such as filter coefficients in DSP applications. Being non-cacheable, such usage of on-chip memory reduces cache pollution and thrashing [19, 25].

FCB (Fabric Co-processor Bus - PPC405 only). This 32-bit bus is used for connecting fabric co-processor modules (FCM) to the auxiliary processor unit (APU) controller. There is no bus arbitration, so one processor master / multiple slaves are allowed. Each slave must decode a unique set of instructions presented on the FCB. This

interface provides a high bandwidth and low latency connection that integrates directly into the processor pipeline. The FCB is the primary choice for integrating high performance co-processors to the PowerPC system as it provides a mechanism for instruction decoding and allows for larger data transfers (quad word load/store) [26].

FSL (Fast Simplex Link - PPC405 and MicroBlaze). This 32-bit bus provides a uni-directional point-to-point communication interface (FIFO-based) between any two elements on the FPGA. The MicroBlaze processor can communicate with up to eight FSL channels (in each direction). The PPC405 supports up to 32 FSL channels through the APU controller [27].

LMB (Local Memory Bus - MicroBlaze only). This local 32-bit bus is used to connect BRAM memory to the MicroBlaze processor. Separate read and write ports are provided. There is no arbitration, so one processor master and up to 16 slaves are allowed [28].

XCL (Xilinx Cache Link - MicroBlaze only). This is the MicroBlaze cache interface, providing a point-to-point link between main memory and cache implemented out of on-chip memory. Up to two MicroBlaze processors can use the XCL interface as main memory controllers have only 4 XCL ports (each processor requires one port for data and one for instruction) [29].

Table 3.2 on the following page provides an overview of the five most common buses in a Xilinx embedded processor system (PLB, OPB, DCR, OCM, and LMB).

3.3.4 Auxiliary Processor Unit Controller

Arguably the most notable feature of the V4FX is the auxiliary processor unit (APU) controller that tightly couples custom co-processors built in the FPGA fabric to the PPC405 core. It is the APU that sets the V4FX apart from its predecessor, the V2P.

The APU provides accelerated system performance by managing the interface between a fabric co-processor module (FCM) and the processor core. As seen in Figure 3.6 on page 24, the APU connects into the PPC405 instruction pipeline and is able to negotiate the transfer of particular instructions and data to the appropriate FCMs that support such operations. This high bandwidth and low latency direct interface to HW accelerators makes it possible to extend the native PowerPC 405 instruction set with certain special instructions as well as with completely custom, user-defined instructions [30].

Feature	CoreConnect Buses			Other Buses	
	PLB	OPB	DCR	OCM	LMB
Processor family	PPC405	PPC405, MicroBlaze	PPC405	PPC405	MicroBlaze
Data bus width	64	32	32	32	32
Address bus width	32	32	10	32	32
Clock rate, MHz (max)[1]	100	125	125	375	125
Masters (max)	16	16	1	1	1
Masters (typical)	2-8	2-8	1	1	1
Slaves (max)[2]	16	16	16	1	16
Slaves (typical)	2-6	2-8	1-8	1	1
Data rate (peak)[3]	800 MB/s	500 MB/s	500 MB/s	1500 MB/s	500 MB/s
Concurrent read/write	Yes	No	No	No	No
Address pipelining	Yes	No	No	No	No
Bus locking	Yes	Yes	No	No	No
Retry	Yes	Yes	No	No	No
Timeout	Yes	Yes	No	No	No
Fixed burst	Yes	No	No	No	No
Variable burst	Yes	No	No	No	No
Cache fill	Yes	No	No	No	No
Target word first	Yes	No	No	No	No
FPGA resource usage	High	Medium	Low	Low	Low
Compiler support for load/store	Yes	Yes	No	Yes	Yes

Notes:

1. Maximum clock rates are estimates and are presented for comparison only. The actual maximum clock rate for each bus is dependent on device family, device speed grade, design complexity, and other factors.
2. Maximum value set by maximum allowed parameter value specified in the core. Actual bus specification does not limit this value.
3. Peak data rate is the maximum theoretical data transfer rate at the clock rate shown for each bus.

Table 3.2: Most common buses used in Xilinx embedded processor systems [28]

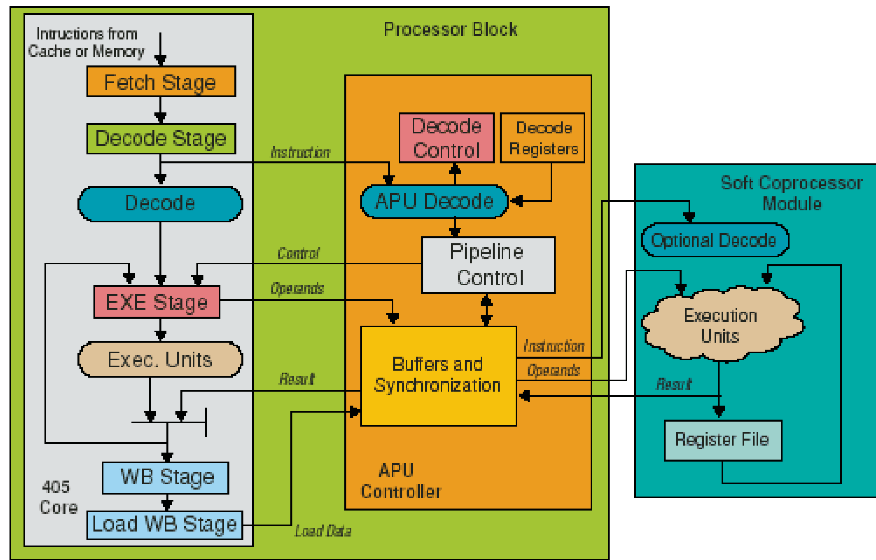


Figure 3.6: The APU integrates directly into the processor pipeline and decodes soft co-processor supported instructions [30]

The APU supports three types of instructions:

- APU load/store (direct to hardware, up to 16 bytes, quadword)
- PPC floating-point instructions
- User-defined instructions (UDIs)

To determine whether an instruction is CPU-bound or APU-bound, both the CPU and the APU simultaneously attempt to decode the instruction. If the instruction is found to be a CPU-bound instruction, execution continues as normal and the APU does not get involved further. If the instruction is found to be an APU-bound instruction, the CPU waits for a response back from the APU, which determines whether this instruction is a supported (valid) or unsupported (invalid). An exception is generated if the CPU does not get a response within one cycle. If the APU responds that the instruction is indeed a valid one, operands are fetched from the CPU to the APU and passed to the hardware co-processor for execution. The result from the hardware co-processing is delivered back to the CPU (via the APU) into the write-back stage [30].

Since the CPU typically runs at a higher frequency than the custom FCMs, a synchronization mechanism is necessary to handle the transfer of data between the two units. The APU manages this synchronization completely independently. The APU knows when

to read operands from the CPU and when to return with the results. The developer never needs to get involved in managing the CPU-APU interface, thus streamlining the design process [30].

The APU supports both autonomous and non-autonomous instructions. Autonomous instructions do not require the CPU to stall and wait for the result while the instruction is being executed. Non-autonomous instructions do require the CPU to stall and wait for the result. These instructions can be further broken down into blocking and non-blocking instructions. Blocking instructions suppress all asynchronous exceptions and interrupts which may be generated while the instruction is being executed. Non-blocking instructions allow for exceptions and interrupt to be serviced, but in effect must flush the HW co-processor [30].

3.4 ML410 Development Board

The target platform for this thesis is the Xilinx ML410 Development Board which features the V4FX60 FPGA. This FPGA is in the -11 speed grade thus allowing the dual embedded PPC405 cores to operate at up to 400 MHz when the APU controller is not in use or up to 275 MHz when the APU controller is in use (as is the case for much of the work in this thesis) [31].

This board comes in a standard ATX form factor with 64 Mbytes of component DDR memory (32-bit) and 256 Mbytes DDR2 DIMM memory (64-bit). The DDR memory is capable of running at up to 266 MHz, however the available memory controller IP core operates at 100 MHz, thus delivering PC-1600 performance (1.6 Gbytes/sec). In the case of DDR2, the controller IP core can integrate with a 266 MHz memory module, however, the module supplied with the board is capable of running at up to 200 MHz, thus delivering PC2-3200 performance (3.2 Gbytes/sec) [32].

The board also features a SystemACE compact flash controller, dual Ethernet PHYs, PCI and PCI express interfaces, VGA interface, USB ports, and much more. The SystemACE controller can be used to access data on a compact flash card (non-volatile storage) and also has the ability to configure the FPGA with hardware bitstreams and software object codes stored in one of eight configuration locations on the card. A compete listing of all the features on the ML410 board is presented in Figure 3.7 on the following page. A block diagram of the interconnection between peripherals is shown in Figure 3.8 on page 27 [32].

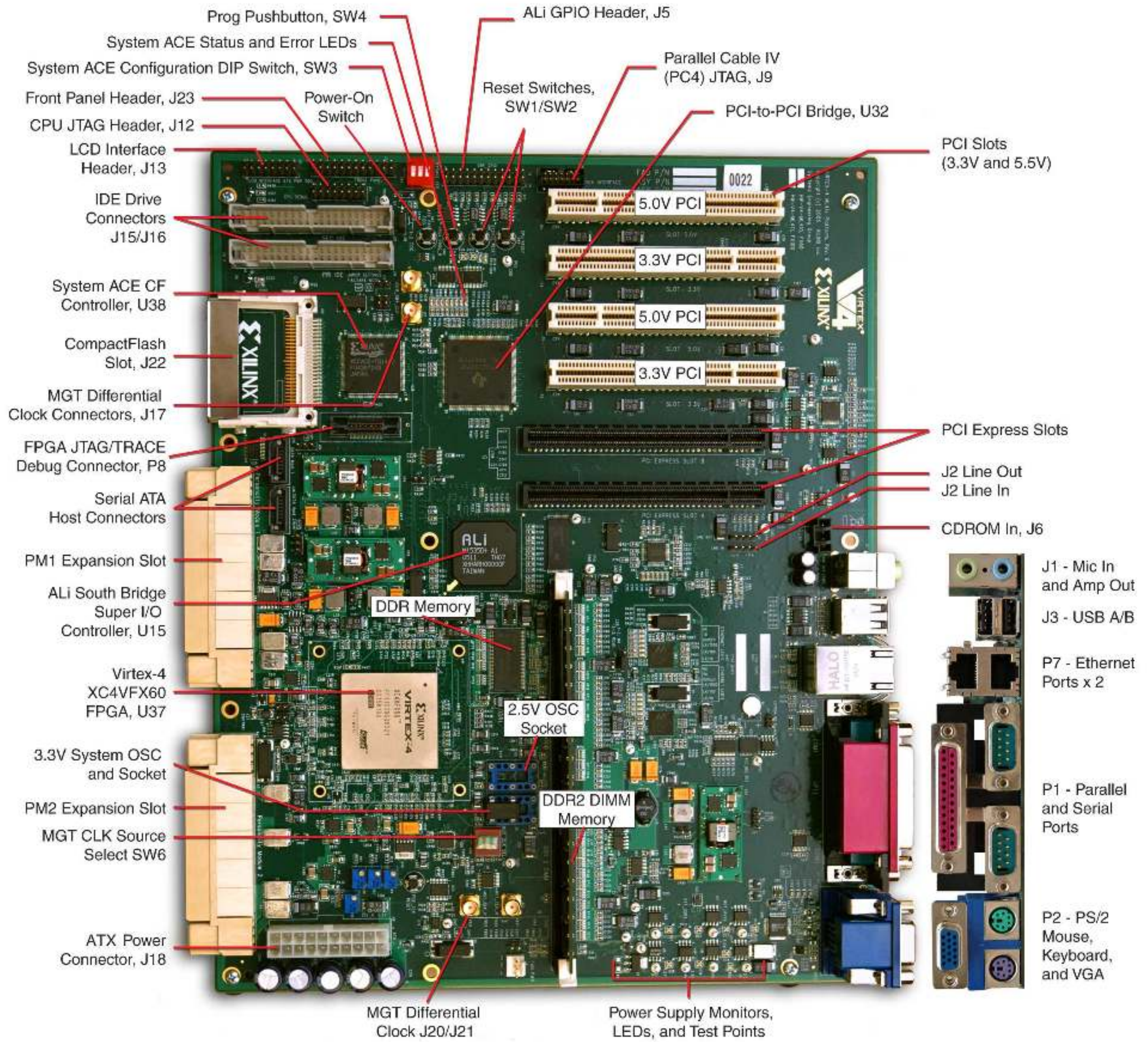


Figure 3.7: The ML410 development board [32]

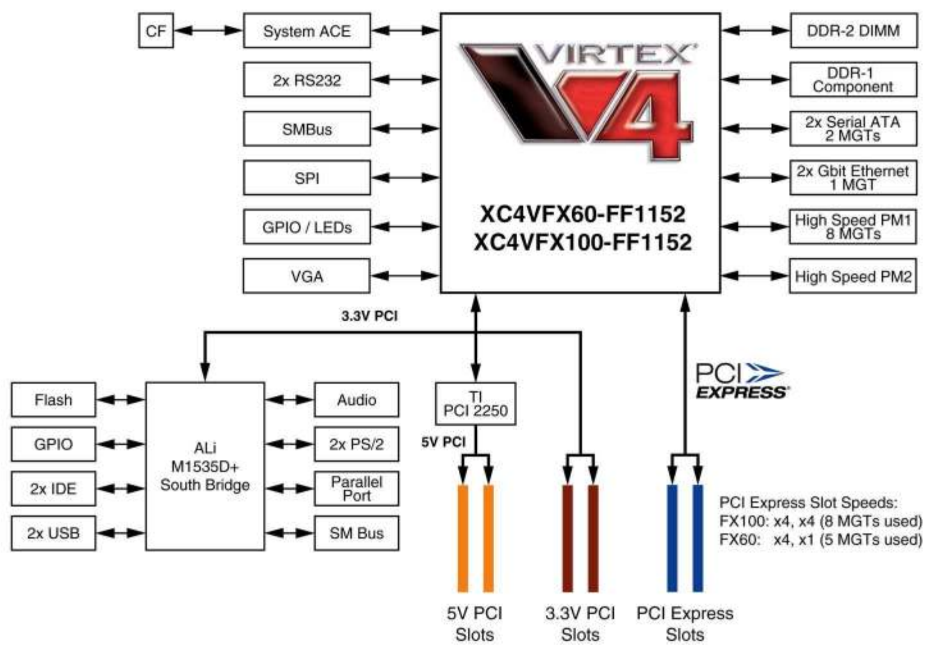


Figure 3.8: ML410 interfaces block diagram [32]

Chapter 4

FTIR Base System

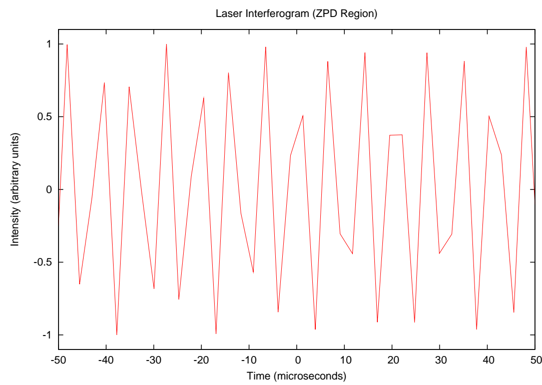
This chapter presents the hardware generation and software configuration of the FTIR base system. The starting point for this work is FORTRAN source code and simulated interferograms provided by NASA Jet Propulsion Laboratory as a result of the V2P research task [2]. The software contains only the three most time consuming data processing steps in Table 2.2 on page 11 - interferogram re-sampling, phase correction and fast Fourier transformation. The base system, implemented on the ML410 development board, must have adequate hardware resources to execute the software, which must first be ported to the PowerPC processor.

The input data represents simulated time-domain interferograms from the reference laser (*Ge* detector) and the *InSb* detector (see Figure 4.1 on the next page). Note that the reference laser interferogram is just a sine wave modulated in frequency by the scanner velocity fluctuations. An *HgCdTe* interferogram is not included as its processing is very similar to that of the *InSb* interferogram. The input data is stored in two columns of single-precision floating-point numbers in ASCII format and represents the number of photons striking the detector in the sample time. The first column is the reference laser interferogram and the second column is the *InSb* interferogram. The file size is roughly 38 Mbytes.

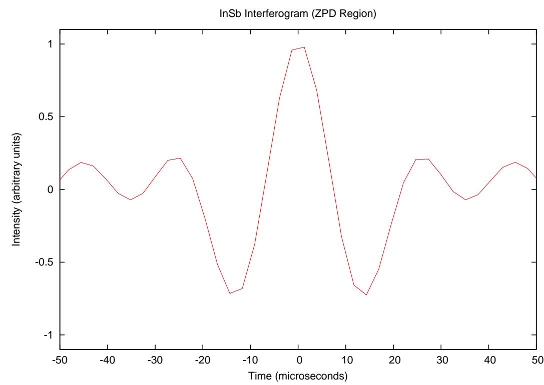
4.1 Generating a Hardware Platform

The FTIR base system hardware platform is built from scratch using the Xilinx Platform Studio (XPS) and the Base System Builder (BSB) in the Embedded Development Kit (EDK). A system diagram, showing the main buses and system components, is presented in Figure 4.3 on page 31 while a detailed description follows below.

First, it is necessary to select and configure the processor and its system interfaces. The hard PPC405 processor is selected for FTIR data processing as it delivers much higher performance than the soft MicroBlaze processor. Next it is configured with the following



(a) Reference laser interferogram



(b) InSb interferogram

Figure 4.1: Simulated time-domain interferograms used as input data [2]

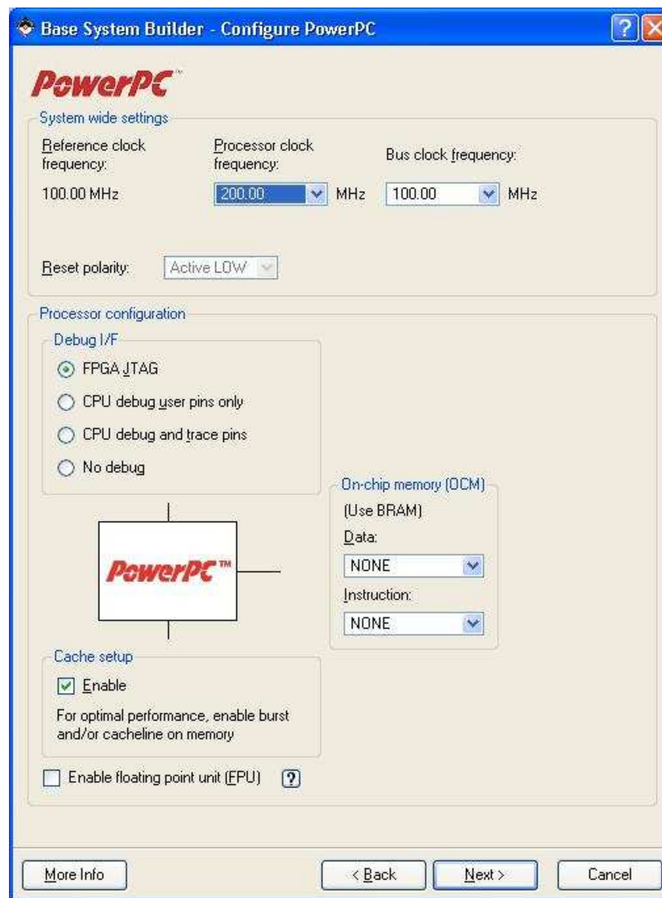


Figure 4.2: PPC405 base system configuration

options (see Figure 4.2 on the previous page):

- 100 MHz (max) PLB and OPB bus clock frequency for highest possible transfer rates on system buses
- 200 MHz processor clock frequency[1]
- Cache enabled for maximum performance (instruction and data, burst and/or cache-line)

The FPU is not selected as part of the base system in order to obtain a benchmark without any hardware co-processors. On-chip memory is also not selected because it is of little use in the FTIR system as all data and text sections are significantly larger than OCM capacity (64 Kbytes max for data and 128 Kbytes max for instruction). Furthermore, OCM may limit system performance as it is harder to meet timing constraints at higher system frequencies with OCM connected to the processor.

Next, various bused peripherals are selected. An OPB RS232 UARTLITE peripheral is selected for standard input/output (I/O) functionality over a serial null modem link to the host PC (and HyperTerminal). The baud rate is set to 115200 for maximum I/O performance. An OPB SystemACE controller is also included for access to the compact flash (CF) card. The simulated time-domain interferogram will be stored on the non-volatile CF card and read into system RAM for processing. This will eliminate the need to download the input data from the PC directly to the FPGA system RAM every time it boots. Furthermore, storing the interferogram data on the CF card better resembles the MATMOS instrument computer which reads in raw interferograms into RAM from the instrument memory bank. As for memory, the FTIR base system utilizes the PLB DDR2 memory controller which interfaces to the external 256 Mbytes DDR2 DIMM. This is the largest capacity highest performance volatile memory available on the ML410 development board. PLB BRAM memory is not necessary in this or other system builds as it is too small to hold any critical text or data sections and would only create an extra load on the PLB if utilized.

One final component that is included in the base system (as well as all other system builds) is the MGT protector core. This core initializes the MGTs to a known state in a

[1]Although the PPC405 core on the V4FX60 -11 speed grade FPGA can be clocked at up to 400 MHz, the use of the APU controller (in later builds) limits this frequency to 275 MHz. A 200 MHz processor clock is selected as it is an integer multiple of the bus frequency and is easier to work with in terms of the on board digital clock managers (DCM).

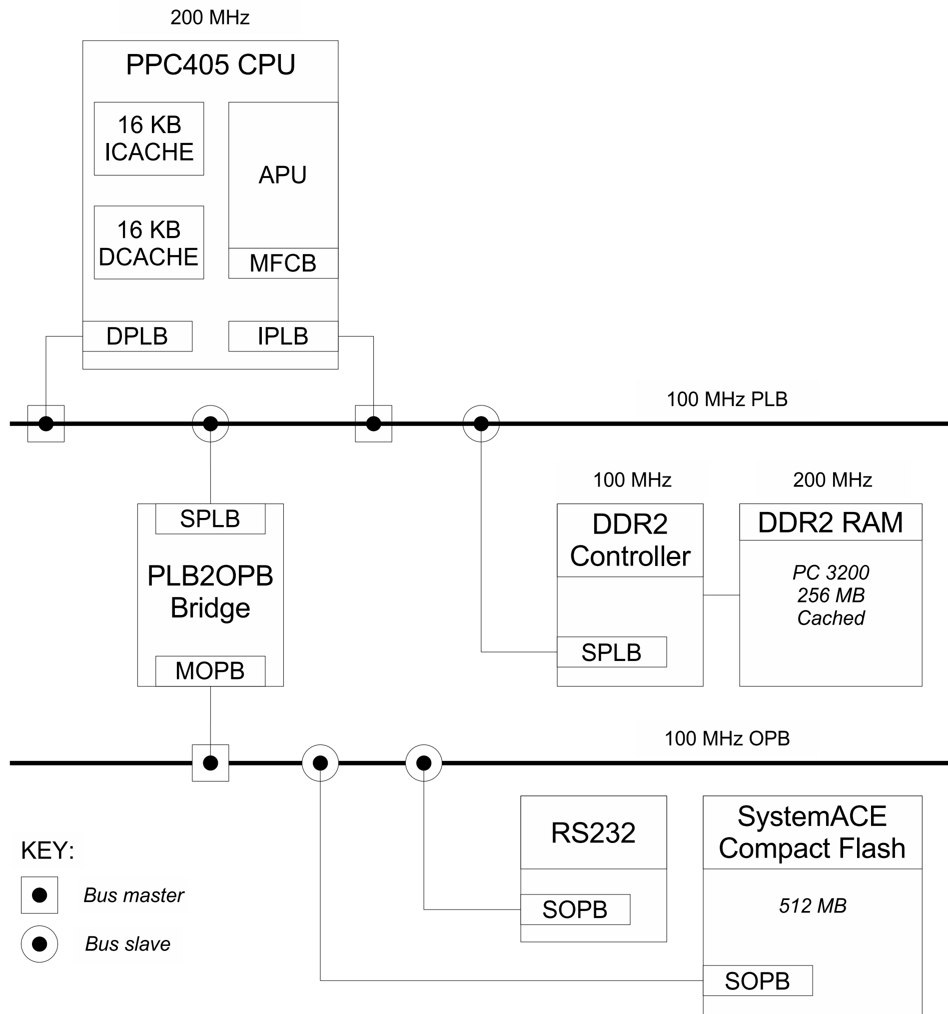


Figure 4.3: FTIR base system diagram (main components only)

system that does not utilize them for data transfer. This is absolutely necessary due to an issue with the Virtex-4 FX that may lead to a breakdown of the MGTs if they are left in an uninitialized state for too long[2].

The FTIR base system builds meeting all timing constraints that are automatically generated based on bus/memory/CPU frequencies. The default synthesis and implementation options are used. The device utilization, shown in Table 4.1 on the following page, is rather modest with only 13% of slices occupied.

[2]For more information on this, please see Xilinx answer record #22471.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	3,511	50,560	6%	
DCM autocalibration logic	14	3,511	1%	
Number of 4 input LUTs	2,722	50,560	5%	
DCM autocalibration logic	8	2,722	1%	
Logic Distribution				
Number of occupied Slices	3,330	25,280	13%	
Number of Slices containing only related logic	3,330	3,330	100%	
Number of Slices containing unrelated logic	0	3,330	0%	
Total Number of 4 input LUTs	3,642	50,560	7%	
Number used as logic	2,722			
Number used as a route-thru	136			
Number used for Dual Port RAMs	648			
Number used as Shift registers	136			
Number of bonded IPADs	32	72	44%	
Number of bonded OPADs	32	32	100%	
Number of bonded IOBs	146	576	25%	
Number of BUFG/BUFGCTRLs	6	32	18%	
Number used as BUFGs	6			
Number used as BUFGCTRLs	0			
Number of DCM_ADVs	2	12	16%	
Number of PPC405_ADVs	2	2	100%	
Number of JTAGPPCs	1	1	100%	
Number of IDELAYCTRLs	4	20	20%	
Number of GT11s	16	16	100%	
Total equivalent gate count for design	102,726			
Additional JTAG gate count for IOBs	10,080			

Table 4.1: Device utilization summary for FTIR base system

4.2 Configuring Software

With the hardware platform ready, the software needs to be prepared for execution on the embedded PPC405 processor. It must be ported from FORTRAN to C and then cross-compiled for the PPC405 processor from XPS using the GCC compiler. Prior to porting the software, it is important to first understand the structure of the FORTRAN source and its functions.

4.2.1 Software Structure

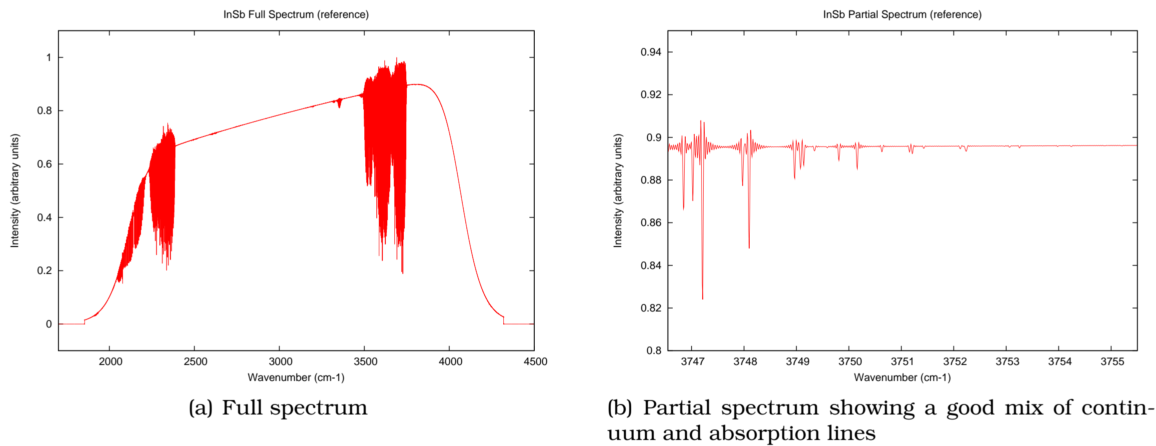
The diagram in Figure 4.5 on the next page shows the software flow of the FTIR spectrometry algorithm used in this thesis (top level FORTRAN code is shown in Appendix C.4 on page 103). The program begins by computing a matrix of windowed interpolation operators as part of the *pcoper* routine. Then it enters a loop for all scans in one occultation. The software assumes 52 interferograms per occultation, or 104 interferograms per orbit (two occultations). However, for testing purposes, only one interferogram (middle one, #26) is processed with only one iteration in the loop and for only one of two solar detectors.

The processing of the raw interferogram begins with reading the data into system RAM. On a PC, this data comes from a text file stored on the hard disk. After the data is read, the *t2f* routine re-samples the time-domain interferogram to the path-difference domain. Then the *ipplite* routine computes the spectrum (phase correction and fast Fourier transformation). At the end, the computed spectrum, in full or in part, is dumped to the screen. This data can be plotted to produce graphs shown in Figure 4.4 on the next page.

The original software was slightly modified in its last step. It no longer prints the partial spectrum (1000 points from 3/5 of the full spectrum) to the screen as in:

```
113  c
114  c  Display a section of the spectrum
115  c
116               do indexa=(3*specount)/5,((3*specount)/5)+1000
117  c             do indexa=1,specount
118                 write(*,'(SP1PE16.8E2)') ryir(indexa,jdet)
119               enddo
```

Instead, this data was stored to a text file and used as a reference spectrum. The program now compares a section of its calculated spectrum to the reference spectrum from file and prints out the maximum deviation to the screen. This was done in preparation to porting the software to the FPGA platform and determining whether the FPGA system



(a) Full spectrum

(b) Partial spectrum showing a good mix of continuum and absorption lines

Figure 4.4: The spectrum produced from simulated interferogram data [2]

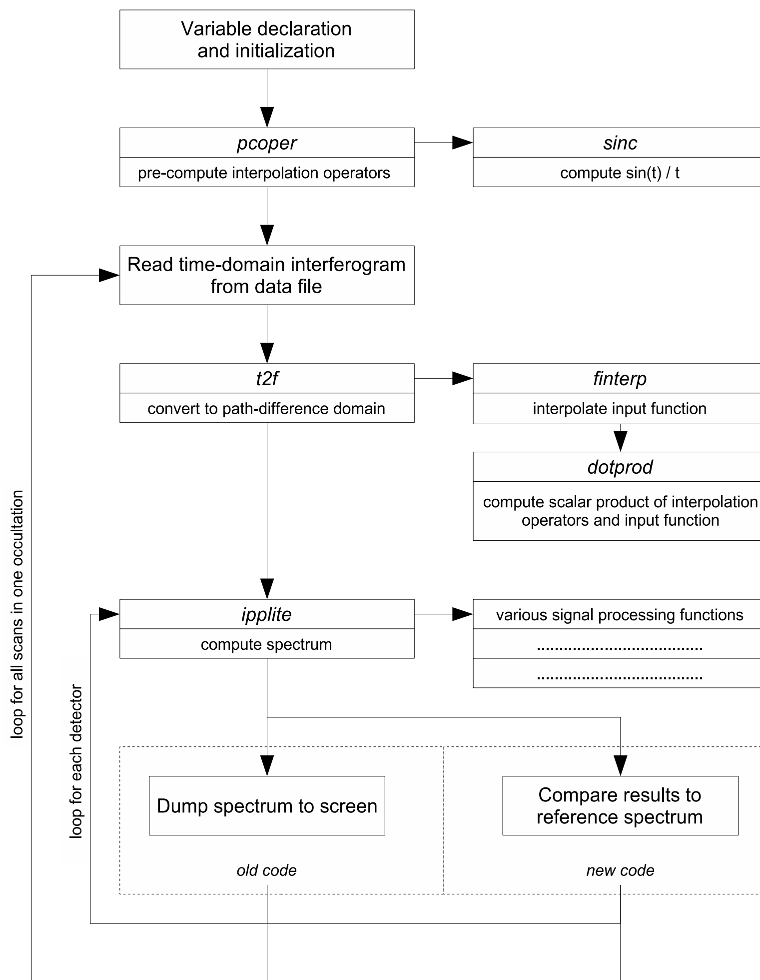


Figure 4.5: Software flow of the FTIR spectrometry algorithm

produces accurate results. The changes to the code are displayed below.

```
122   c
123   c  Search for max deviation between reference and calculated
124   c
125               maxdev=0.0
126               chkdev=1
127               chkoff=((3*specount)/5)-1
128               do indexa=1,chkcnt
129                 curdev=abs(chkspe(indexa)-ryir(chkoff+indexa,jdet))
130                 if (curdev.gt.maxdev) then
131                   maxdev=curdev
132                   chkdev=indexa
133                 endif
134               enddo
135               write (*,*)'Maximum deviation of ',maxdev/chkspe(chkcnt),
136        &       ' at ',chkdev
```

One final modification that is necessary prior to porting is the removal of all I/O operations, except for simple string printing. This was determined through trials of porting the FORTRAN source to C and compiling it with the GCC compiler supplied in XPS. The GCC compiler would not successfully compile code that was ported without first removing file I/O and printing of local variables. The FTIR spectrometry software with the necessary I/O commented out and ready to be ported to C is shown in Appendix C.5 on page 106.

4.2.2 Porting FORTRAN to C for PPC405 Embedded Processor

The FTIR spectrometry algorithm is written in FORTRAN. In order to execute it on the PowerPC 405 embedded processor and evaluate performance, the algorithm has to be converted to C for use with the GCC compiler in XPS. The FORTRAN-to-C Converter (*f2c*) from AT&T Bell Laboratories does this conversion automatically. Once converted, the code is linked with the FORTRAN-to-C Library (*libf2c*) [6]. For instructions on how to set up *f2c*, *libf2c*, and create a sample EDK project, please refer to Appendix A on page 89.

Once *f2c* is properly set up, the software can be converted to C with the following command (main FORTRAN source in *matmos-ipp-chk-noio.f*):

```
f2c -R -Nc40 matmos-ipp-chk-noio.f
```

The *-R* option is necessary so that *f2c* does not promote real (single-precision) functions and operations to double-precision. Such a promotion would adversely affect the execution time as double-precision arithmetic takes more CPU cycles than single-precision arithmetic. The *-Nc40* option is necessary in order to allow for more continuation lines (starting with &) in a given FORTRAN statement than permitted by default. In this case, *f2c* is instructed to allow up to 40 continuation lines.

4.2.3 Modifying Converted Code

Once converted, the FTIR spectrometry C-source needs to be augmented with proper initialization code, file I/O functions, and timing routines[3]. All of the modifications made to the code are right before as well as inside the *MAIN__* function. Compact flash file I/O function are also added. A snapshot of the modified converted code can be seen in Appendix C.6 on page 109. A detailed description (referring to the code in Appendix C.6) follows below.

Initialization

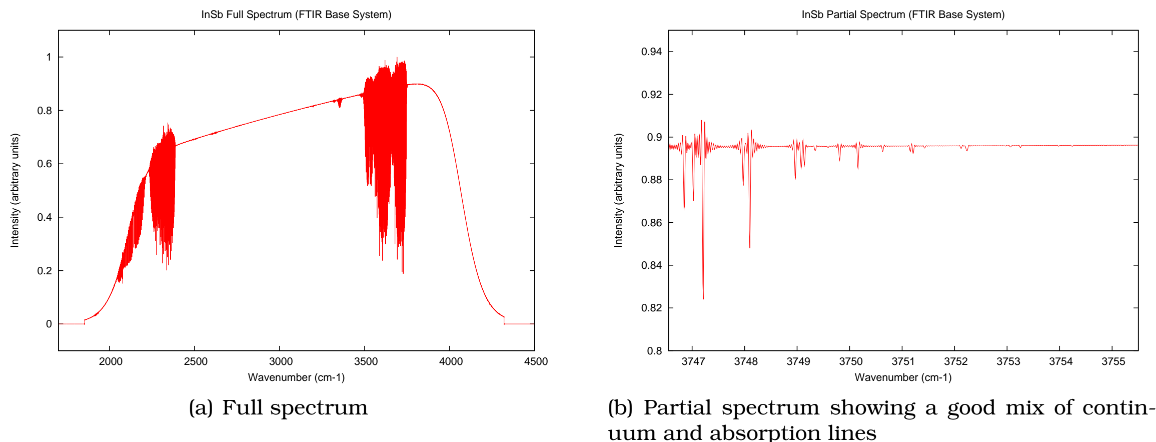
Lines 14-59 contain the necessary *#include* and *#define* statements as well as file I/O function declarations. The timing functions provided by *xtime_l.h* (line 29) are only defined when profiling is turned off. This is to ensure that calls to timing functions do not interfere with code profiling which also uses the built in timers[4]. Cacheable memory regions are also defined in this code segment. Depending on the hardware build, the memory map may change. The *printfloat* function is defined and provides a mechanism for printing floating-point numbers to the screen. The standard *printf* function is too large and does not work well for printing floats on the PPC405 processor. The *read_data* and *write_data* compact flash I/O functions are also declared in this segment of code.

Further modification to the code in the initialization category can be seen on line 371 (defining *spechk* array to hold partial reference spectrum) and in lines 285-399. In the latter, the instruction and data caches are initialized and the SystemACE controller is cleared of any bad bits.

File I/O

Lines 638-717 define two functions that are responsible for reading from and writing to the CF card - *read_data* and *write_data*. Both read and write single-precision floating-point numbers in ASCII format. Although reading/writing in binary format is much more efficient, the ASCII format was chosen for easy portability between big-endian (PowerPC) and little-endian (x86) systems and for ease of plotting. The *read_data* function expects the input file (simulated interferogram) to have floating-point numbers in two columns and of particular width. The data is read one line at a time. The *write_data* function writes floating-point data (the spectra) to a file in a single column format with fixed width.

[3]Modification to the code are blocked off with /********************************/.
[4]All calls to timing functions are blocked off with *#ifndef* PROFILING



(a) Full spectrum

(b) Partial spectrum showing a good mix of continuum and absorption lines

Figure 4.6: The spectrum produced by the FTIR base system on the ML410 board

Timing and Others

Most of the other modifications deal with accessing the PowerPC timer for measuring the performance of data processing. Prior to calling a function, such as to convert to path difference domain (line 528), the free-running timer value is recorded and then accessed again after the function returns. The difference in CPU cycles is converted to an actual number of seconds (as in line 535).

Other changes include a slight modification of the *for* loop on line 574 to store 1000 points from the spectrum to a local array prior to comparing them to previously calculated reference data. Lines 599-609 allow the user to enable dumping calculated spectrum (in part or in full) to a file on the CF card. This data can later be used to compare the spectrum produced by the FPGA system to the known good result in Figure 4.4 on page 34.

4.3 Checking Processing Results

With both the hardware and the software ready for deployment, the ML410 board is configured for FTIR spectrometry data processing. The hardware bitstream is first downloaded from ISE, and then the software executable is downloaded from EDK. Prior to evaluating the performance of the system (in terms of execution time), it is first necessary to verify that the calculated spectrum is correct. The software reports that the maximum deviation in the partial spectrum is a negligible 0.0009795%. Further proof in the accuracy of data processing on the FPGA system can be seen in the plots produced from

	ML410 (XILINX)	V2P (NASA JPL)		ML410 (XILINX)		
PPC405 FREQ.	200 MHz	300 MHz		200 MHz		
INTERFEROGRAMS	1	104		scaled to 104		
DETECTORS	1	2		scaled to 2		
SOFTWARE COMPONENT	TIME (SEC)	TIME (MIN)	CYCLES ($\times 10^{12}$)	TIME (MIN)	CYCLES ($\times 10^{12}$)	SPEEDUP
RE-SAMPLING	1197.7938	3404	61.272	4152	49.824	0.82x
SPECTRUM (PHASE CORRECTION, FFT)	117.4963	488+272	13.680	407	4.884	1.87x
TOTAL	1315.2901	4164	74.952	4559	54.708	0.91x
EFFICIENCY SCORE[5]	-	1.00		1.37		-

Table 4.2: Execution times for FTIR base system on ML410 board and comparison to NASA JPL V2P research

the full and partial spectrum, displayed in Figure 4.6 on the preceding page. These are indistinguishable from the reference plots in Figure 4.4 on page 34.

4.4 Initial Performance Evaluation

With the accuracy of the results verified, the performance can now be evaluated. The FTIR base system running on the embedded PPC405 processor in the V4FX60 FPGA reports the execution times shown in the first column of Table 4.2. These times are presented next to the results obtained from the NASA JPL V2P research task (middle column) [2]. The V2P research task processed 104 interferograms with two detectors for two complete occultations. The FTIR base system processes one interferogram for one detector. The last column in Table 4.2 shows the scaled FTIR base system results for 104 interferograms and two detectors.

Although the V4FX system did not achieve a lower execution time than the V2P system (speedup = 0.91x), the overall efficiency[5] of the V4FX system is higher. The V4FX took fewer CPU cycles to do the same amount of work as the V2P system. Comparing CPU cycles is valid in this case as both the V4FX and the V2P host a PPC405 processor with the same ISA. Thus the processor performs the same amount of work per cycle regardless of its clock rate. The slower clock rate does adversely affect the overall execution time, as is the case in the re-sampling step of the data processing. However, certain V4 enhancements in the reconfigurable logic (i.e. different slice architecture) are most likely

[5]The efficiency score is based on a CPU cycle count of 74.952×10^{12}. Any cycle count less than this value has a higher efficiency score.

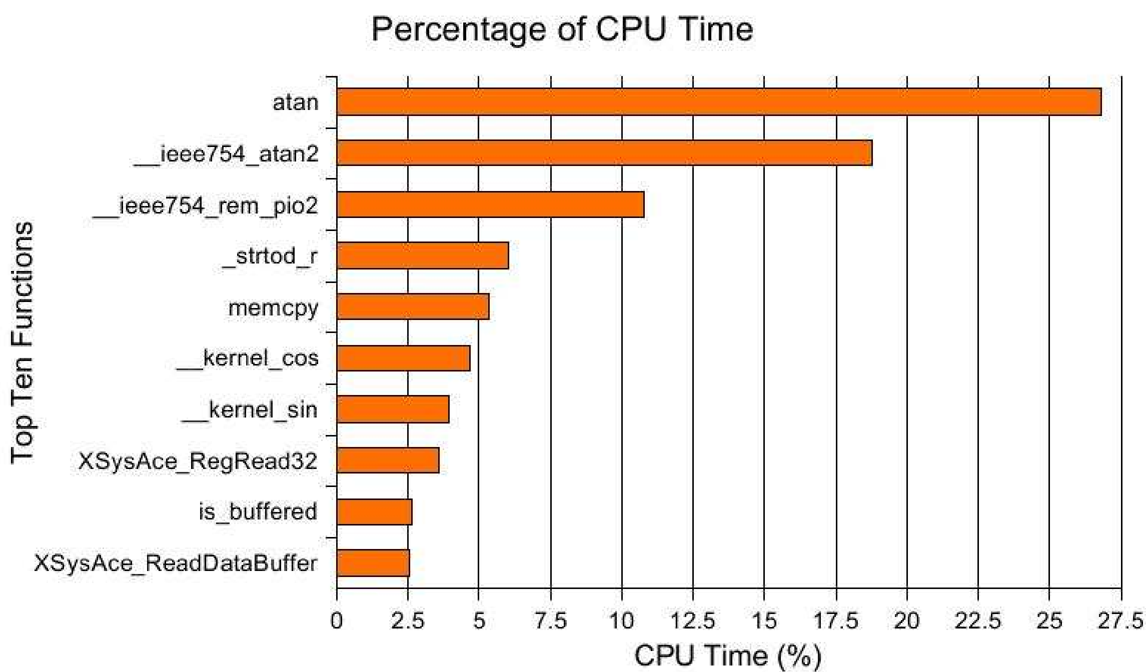


Figure 4.7: Profiling results for FTIR base system

responsible for the improved execution time in the compute spectrum step, even at the slower clock rate.

Even though the V4FX system has a higher efficiency score (and the embedded processor will consume less power at 200 MHz), the more important goal is to demonstrate a reduction in execution time over previous implementations. An excellent tool to identify where in the software the CPU spends most of its time is the Xilinx profiler in the Software Development Kit (SDK)[6]. When software profiling is enabled, a timer is configured on the PPC405 processor to keep track of the amount of time spent in each function called. This data is stored in a memory region specified by the user and can later be downloaded to the host system (i.e. the development PC) for analysis. Profiling was performed on the FTIR base system with the results presented in Figure 4.7.

The graph in Figure 4.7 lists top ten time consuming functions called in the FTIR spectrometry algorithm. Not surprising, most functions called take up very minimal CPU time to execute. However, two functions stand out far above the rest, together taking up over 45% of the CPU time. These two functions are *atan* and *__ieee754_atan2*. From their names, it is clear that both come from the math library. After a close inspection

[6]Xilinx SDK can provide very valuable information during profiling, however the software is still buggy and often does not profile correctly (if at all).

of the original FTIR C-source (Appendix C.6 on page 109), it was found that all math library functions were being called with double-precision arguments even though the input interferogram is in single-precision. Double-precision arithmetic is more complex than single-precision arithmetic and should be avoided when double-precision is not absolutely necessary. For the original FTIR C-source, this was the source of the bottleneck and the reason why the *atan* and *__ieee754_atan2* functions were taking up so much CPU time.

Chapter 5

Software Optimizations

This chapter introduces the software optimizations made to the FTIR spectrometry algorithm. This includes the removal of all double-precision math library calls and the utilization of the IBM performance libraries. These optimization require minimal changes to the source and result in over 4.5x speedup compared to the FTIR base system.

5.1 Removing Double-precision Math Library Calls

After analyzing the profiling data in Figure 4.7 on page 39 and the corresponding C-source in Appendix C.6 on page 109, it was determined that the software unnecessarily utilizes double-precision math library calls. Thus, after carefully searching through the source for all call to the math library, the following double-precision functions were changed to their single-precision (SP) counterparts[1]:

- double acos(doublereal) => float acosf(real)
- double cos(doublereal) => float cosf(real)

[1]The single-precision math library functions are non-ANSI.

SW Optimizations	**None (base system)**	**SP Math Functions**	
PPC405 Freq.	200 MHz	200 MHz	
Interferograms	1	1	
Detectors	1	1	
Software Component	**Time (sec)**	**Time (sec)**	**Speedup**
Re-sampling	1197.7938	780.9376	1.53x
Spectrum (Phase Correction, FFT)	117.4963	109.9881	1.07x
TOTAL	1315.2901	890.9257	1.48x

Table 5.1: Execution times for system with single-precision math functions

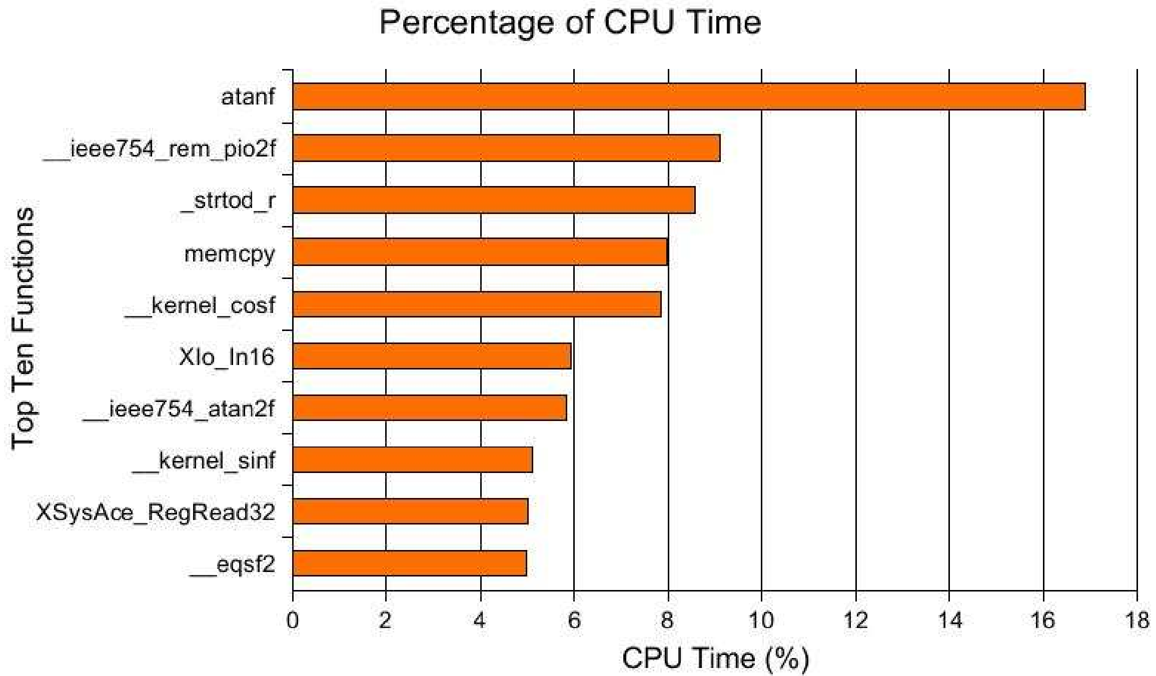


Figure 5.1: Profiling results after single-precision math functions optimization

- double sin(doublereal) => float sinf(real)
- double atan2(doublereal, doublereal) => float atan2f(real, real)
- double sqrt(doublereal) => float sqrtf(real)
- double atan(doublereal) => float atanf(real)

The spectrum produced by this system still demonstrates the same maximum deviation as the FTIR base system - a very small 0.0009795%. Furthermore, a significant speedup is attained, especially in the re-sampling phase, as presented in Table 5.1 on the previous page.

The profiling data (Figure 5.1) shows that the two functions previously taking up over 45% of the CPU time, *atanf* (the SP version of *atan*) and *__ieee754_atan2f* (the SP version of *__ieee754_atan2*), now only take a little under 33%. New functions have risen to the top of the profile suggesting that the CPU is now executing a more balanced program.

5.2 Linking IBM PowerPC Performance Libraries

The IBM PowerPC performance libraries (*Perflib*) is another great optimization that can easily be incorporated into an existing system [7]. Absolutely no changes to the source

SW Optimizations	None (base system)	SP Math Functions IBM Perflib	
PPC405 Freq.	200 MHz	200 MHz	
Interferograms	1	1	
Detectors	1	1	
Software Component	Time (sec)	Time (sec)	Speedup
Re-sampling	1197.7938	244.1836	4.91x
Spectrum (Phase Correction, FFT)	117.4963	44.4243	2.64x
TOTAL	1315.2901	288.6079	4.56x

Table 5.2: Execution times for system with SP math functions and *Perflib*

are required; it is only necessary to link to these libraries for them to take effect. *Perflib* works by replacing string and floating-point routines provided by the compiler with more efficient, hand coded implementations specifically targeting the PPC405 and PPC440 processors. *Perflib* achieves higher performance than standard GCC routines by following four main optimization rules in the following categories [33]:

- Instruction pairing
- Load/Use dependencies
- Operand dependencies
- Compare and branch

Instruction pairing is a technique where the types of instructions issued consecutively are mixed. This achieves higher performance only on the PowerPC440 processor because it can issue two instructions per cycle provided they are of different type (i.e. load and multiply). Instruction pairing may not be very helpful on the PPC405 processor, but it is not harmful [33].

Removing load/use dependencies will help reduce execution time on either processor. A load/use dependency refers to a stall in the pipeline due to the load data not being available for use yet (it takes time to fetch it from cache/memory). Thus, it is wise to put an independent instruction (or a few of them) immediately after a data load and before that data gets used by another instruction. Similarly, an operand dependency refers to a situation where one instruction updates a register that is used by the following instruction. The CPU must wait for the data to settle before executing an instruction that

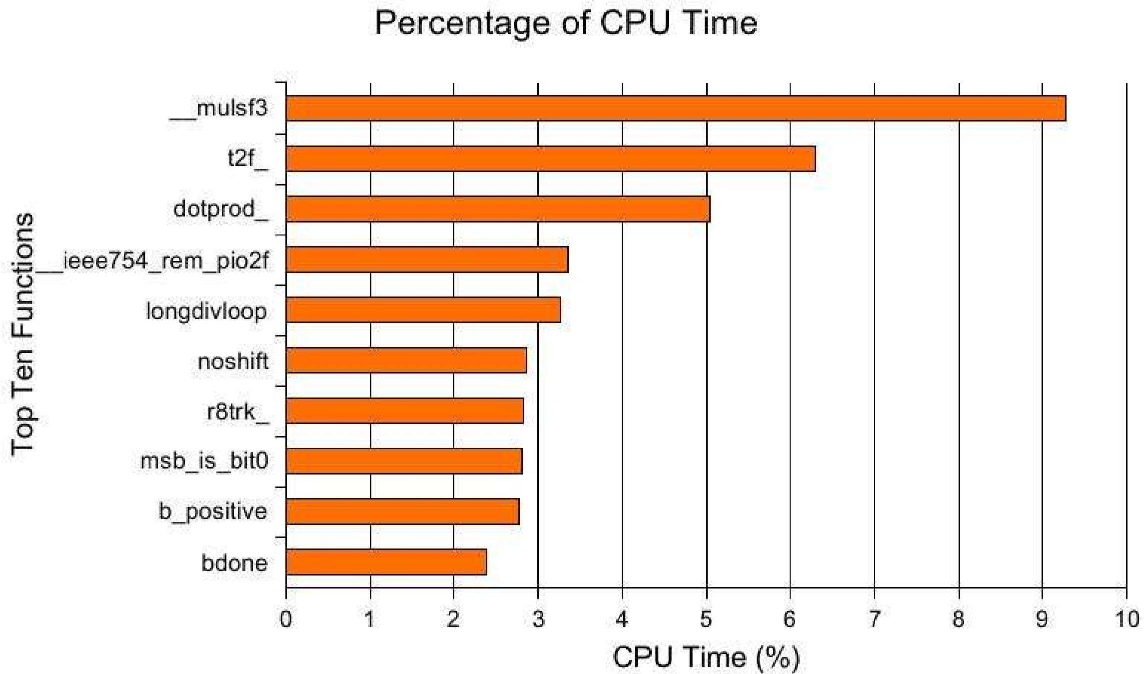


Figure 5.2: Profiling results for system with SP math functions and *Perflib*

depends on that data. Both load/use and operand dependencies are more commonly known as read-after-write (RAW) hazards [33].

The compare and branch category refers to the issue of branch instructions close to CR-updating instruction (i.e. those modifying the compare register, CR). Branches and CR-updating instructions should be separated from each other by at least one other instruction (from a different type). This will prevent stalls as the CPU waits for the CR bits to settle [33].

In EDK, *Perflib* has been pre-compiled and can be linked to the software project by specifying the following option[2]:

```
-mppcperflib
```

This instructs the GCC linker to substitute IBM performance libraries for standard string and floating-point routines. *Perflib* significantly reduces the overall execution time, as shown in Table 5.2 on the previous page. The reported maximum deviation is still 0.0009795%.

[2]In SDK 9.1i, the *-mppcperflib* switch does not work. *Perflib* must be explicitly linked with the *-lppcfp* and *-lppcstr405* options.

System	V4FX (ML410)	V2P (XUPV2P)
SW Optimizations	SP Math Functions IBM Perflib	SP Math Functions IBM Perflib
PPC405 Freq.	200 MHz	200 MHz
Interferograms	1	1
Detectors	1	1
Software Component	Time (sec)	Time (sec)
Re-sampling	244.1836	239.4655
Spectrum (Phase Correction, FFT)	44.4243	47.2607
TOTAL	288.6079	286.7262

Table 5.3: Comparison of V4FX and V2P results for system with SP math functions and *Perflib*

The profile now has an even greater diversity of functions that take up the most CPU time (see Figure 5.2 on the preceding page). The *atanf* function now takes up only 2.23% of CPU time (not shown in graph) while a new function, *__mulsf3*, takes up the most CPU time, 9.27%. This function is responsible for floating-point multiplication and is one replaced by *Perflib*.

5.3 V2P SW Optimization Results

For comparison purposes only, an FTIR spectrometry system (with the previously described SW optimizations) was built on a Digilent XUPV2P board. This board features the XC2VP30 hybrid-FPGA with dual embedded PPC405 processors, 30,816 logic cells, 2,448 Kbits of BRAM, a SystemACE controller for access to a CF card, and a 128 Mbyte DDR DIMM[3]. The hardware design is nearly identical to that of the V4FX FTIR base system, with the only difference being in memory (DDR instead of DDR2) and IP configurations (targeted for V2P instead of the V4FX)[4]. The recorded execution times (see Table 5.3) come very close to those of the V4FX system, with the V2P system slightly outperforming the V4FX. However, any gains that the V2P demonstrates here are dwarfed by the performance of the V4FX after various hardware optimizations described in the next chapter.

[3]This V2P board is different from the custom board used at NASA JPL. Furthermore, the implemented FTIR spectrometry software uses the SP math functions optimization that was not utilized in the NASA JPL research task. Thus, the execution times on this V2P board will be much different from those obtained in the past.

[4]The V2P system was implemented with an earlier version of the Xilinx tools - ISE 8.2i.

Chapter 6

Hardware Optimizations

This chapter describes the various hardware optimizations made to the FTIR spectrometry algorithm. Except in one case (as later described in the FPU section), the software optimizations are kept intact to achieve top performance. The hardware is first augmented with an FPU co-processor and then with a dot-product co-processor. Later, both co-processors are simultaneously integrated in the system. At the conclusion of this chapter, a system with higher CPU / FPU frequencies is presented.

6.1 Xilinx APU Floating-point Unit

The Xilinx APU-FPU is an FCB bound co-processor that extends the native PPC405 ISA to include support for single-precision floating-point operations. It can achieve a sustained performance of up to 100 MFLOPS, with only modest resource utilization - about 5% on the V4FX60 FPGA (see Table 6.1 on the following page) [34].

The FPU consists of an FCB interface component, execution control and decode logic, a 32-bit register file (containing 32 registers), and the various individual execution units for floating-point operations (see Figure 6.1 on the next page). As a Xilinx supplied LogiCORE, only the top level inputs/outputs of the FPU are visible to the developer for interfacing. The user has the option of selecting between the "lite" (no div/sqrt) and the "full" (with div/sqrt) configurations [34].

The FPU is an FCB peripheral that internally runs at half the bus frequency. For best performance, the FCB should be clocked at the same frequency as the PPC405 processor. The clock speeds shown in Table 6.1 on the following page are maximum PPC405 frequencies when the APU controller and FPU are in use. For the V4FX60 FPGA that is on the ML410 board, the maximum CPU frequency when using the APU controller is 275 MHz. If the FCB clock is set to this value, the FPU would internally operate at 137.5 MHz. However, a 275 MHz CPU clock is not compatible with a 100 MHz (max) PLB clock. As described later in the chapter (section 6.4 on page 73), when a 100 MHz

Core Specifics			
Supported Device Family	Virtex-4 FX		
Resources Used	Slices	Xtreme-DSP Blocks	Block RAMs
Lite (no div/sqrt)	1300	4	2
Full (with div/sqrt)	1600	4	2
Clock Speeds	-10 part	233 MHz	
	-11 part	275 MHz	
	-12 part	333 MHz	

Table 6.1: Xilinx's APU floating-point unit v3.0 [34].

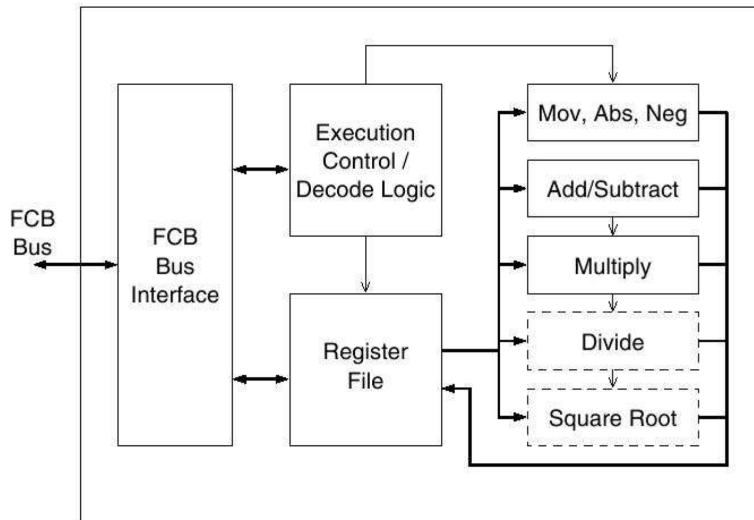


Figure 6.1: System diagram of the APU floating-point unit co-processor [34]

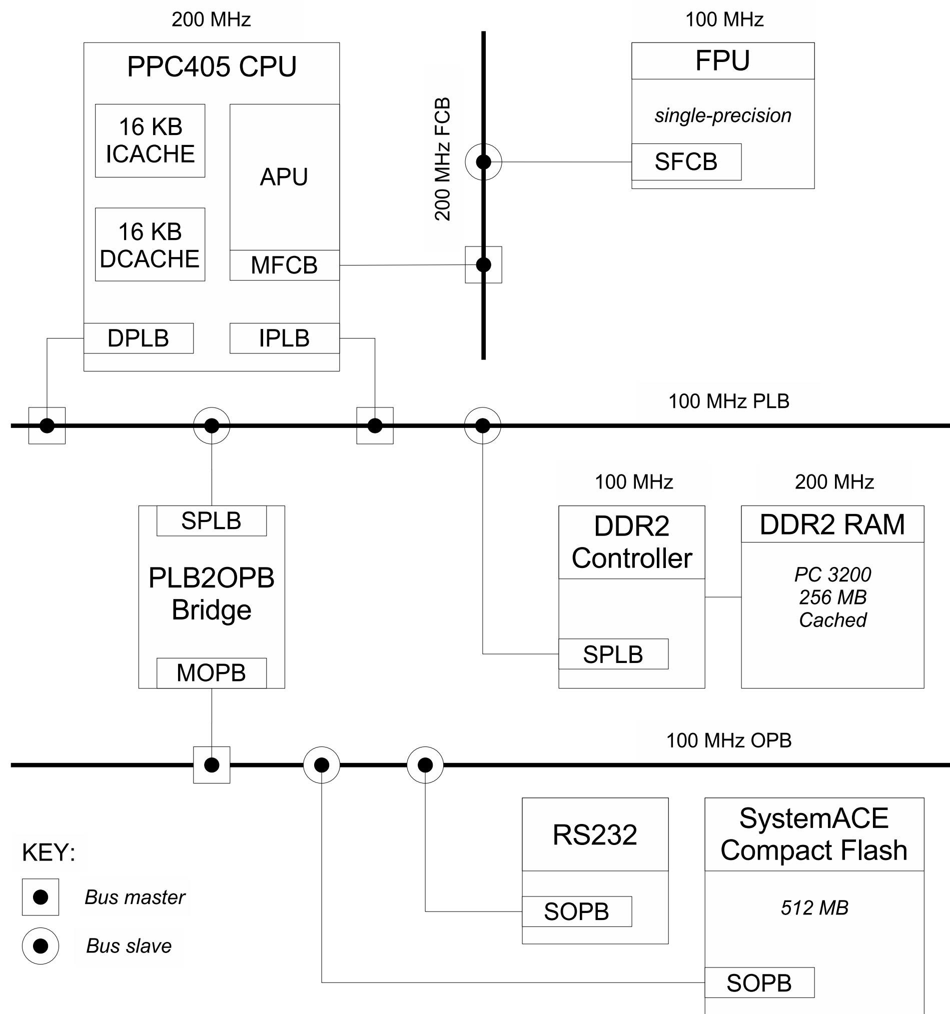


Figure 6.2: FTIR system with FPU co-processor

PLB is in use (for example, for DDR2 memory), the maximum attainable CPU clock rate that allows for proper APU controller operation is 266.67 MHz.

Inserting the FPU core into an existing PowerPC system is a four step process [34]:

1. Configure the PPC405 processor to enable APU controller operation (set APU control configuration register to initial value of *0b1*)

2. Insert the FCB core and configure it to use processor clock and bus reset

3. Insert the FPU core, select the appropriate configuration ("lite" or "full") and configure it to use the half-rate clock

4. Connect the FCB to the processor (master) and the FPU (slave)

The FPU (in "full" configuration) is added to the FTIR base system hardware design following the steps outlined above. The CPU clock rate is kept at 200 MHz and the FPU clock is set to 100 MHz, which feeds off the same digital clock manager as the PLB and OPB. The diagram in Figure 6.2 on the preceding page represents this new system. The device utilization (19% of slices occupied) is shown in Table 6.2 on the next page. In order to meet timing for this build, it was necessary to increase the place-and-route (PAR) effort to "high" with "normal" extra effort (XE) turned on. The necessity to increase the PAR effort suggests that in its current configuration and with the selected clock rates, the system is running close to its timing margin. It may be challenging to meet timing if additional high-speed peripherals or co-processors are added to the system.

6.1.1 *F2C* Compatibility

In the original FTIR base system, the *f2c* library was compiled without FPU support and thus all floating-point operations were done through the compiler's emulation routines (or through *Perflib*). Now that a hardware single-precision FPU is present, it is desirable that all single-precision floating-point operations be performed in the hardware. For this reason, all externally linked libraries must be recompiled with hardware FPU support turned on. For instructions on how to recompile the *f2c* library for FPU support, please see Appendix A on page 89.

6.1.2 Recompiling *Perflib* for Double-precision Only

As is the case with *f2c*, *Perflib* also needs to be recompiled for compatibility with the FPU; however, the procedure is a bit different. As mentioned previously, *Perflib* is a collection of efficient floating-point (SP and DP) and string routines that replace corresponding functions provided by the compiler. In its given form (as supplied with EDK), *Perflib* is not compatible with the FPU because both try to replace single-precision operations normally provided by the compiler. The way around this is to separate out the double-precision optimization routines provided by *Perflib* and rely on the FPU for all single-precision arithmetic. This provides the best of both worlds - good double-precision performance through *Perflib* and excellent single-precision performance though the hardware FPU. String optimization is not necessary in the case of the FTIR spectrometry algorithm because most time is spent on floating-point arithmetic.

Table 6.3 on page 51 lists the optimized floating-point routines provided in the original

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	5,037	50,560	9%	
DCM autocalibration logic	14	5,037	1%	
Number of 4 input LUTs	5,224	50,560	10%	
DCM autocalibration logic	8	5,224	1%	
Logic Distribution				
Number of occupied Slices	5,019	25,280	19%	
Number of Slices containing only related logic	5,019	5,019	100%	
Number of Slices containing unrelated logic	0	5,019	0%	
Total Number of 4 input LUTs	6,377	50,560	12%	
Number used as logic	5,224			
Number used as a route-thru	311			
Number used for Dual Port RAMs	648			
Number used as Shift registers	194			
Number of bonded IPADs	32	72	44%	
Number of bonded OPADs	32	32	100%	
Number of bonded IOBs	146	576	25%	
Number of BUFG/BUFGCTRLs	6	32	18%	
Number used as BUFGs	6			
Number used as BUFGCTRLs	0			
Number of FIFO16/RAMB16s	2	232	1%	
Number used as FIFO16s	0			
Number used as RAMB16s	2			
Number of DSP48s	4	128	3%	
Number of DCM_ADVs	2	12	16%	
Number of PPC405_ADVs	2	2	100%	
Number of JTAGPPCs	1	1	100%	
Number of IDELAYCTRLs	4	20	20%	
Number of GT11s	16	16	100%	
Number of RPM macros	4			
Total equivalent gate count for design	268,714			
Additional JTAG gate count for IOBs	10,080			

Table 6.2: Device utilization summary for FTIR system with FPU co-processor

Routine	Type	Description
fadd	DP	add two DP numbers
fsub	DP	subtract two DP numbers
fmul	DP	multiply two DP numbers
fdiv	DP	divide two DP numbers
fadds	SP	add two SP numbers
fsubs	SP	subtract two SP numbers
fmuls	SP	multiply two SP numbers
fdivs	SP	divide two SP numbers
dtof	conv	convert DP to SP
ftod	conv	convert SP to DP
dtoi	conv	convert DP to INT
ftoi	conv	convert SP to INT
fcmpd	DP	compare two DP numbers
fcmps	SP	compare two SP numbers
fneg	SP/DP	negate a SP or DP number
itod	conv	convert INT to DP
itof	conv	convert INT to SP
ftoui	conv	convert SP to unsigned INT
dtoui	conv	convert DP to unsigned INT

Table 6.3: Optimized floating-point routines provided by *Perflib*

Perflib package. Through a number of trials, it was determined that only the double-precision routines from *Perflib* can exist alongside the hardware FPU. The conversion routines did not work properly when deployed in a system with the FPU present. Thus, the original *Perflib* makefile was modified to only compile the following routines into the library - *fadd*, *fsub*, *fmul*, *fdiv*, *fcmpd*. Complete instructions for recompiling *Perflib* and using it in a system with the FPU are in Appendix B on page 92.

SW Optimizations	**None (base system)**	**SP Math Functions IBM Perflib (DP)**	
HW Optimizations	**None (base system)**	**APU-FPU (100 MHz)**	
PPC405 Freq.	200 MHz	200 MHz	
Interferograms	1	1	
Detectors	1	1	
Software Component	**Time (sec)**	**Time (sec)**	**Speedup**
Re-sampling	1197.7938	151.1817	7.92x
Spectrum (Phase Correction, FFT)	117.4963	14.3672	8.18x
TOTAL	1315.2901	165.5489	7.95x

Table 6.4: Execution times for system with APU-FPU, SP math functions, and DP *Perflib*

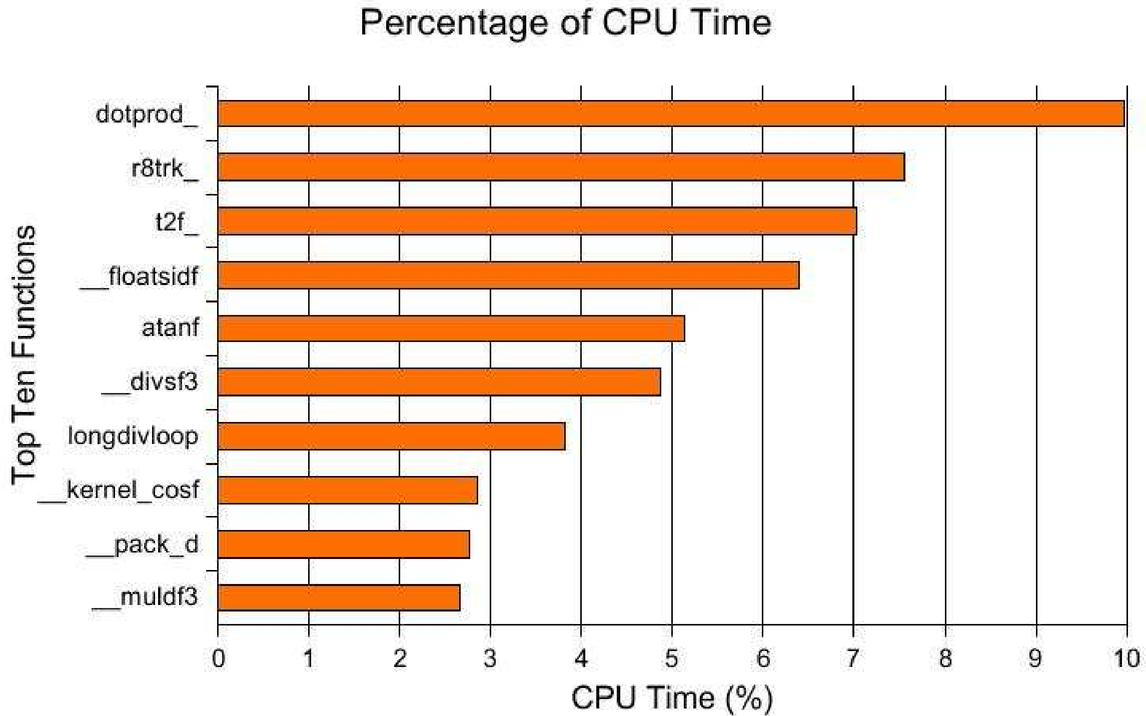


Figure 6.3: Profiling results for system with APU-FPU, SP math functions, and DP *Perflib*

6.1.3 Accuracy and Performance Evaluation

The FTIR system with a hardware FPU, single-precision math functions, and double-precision *Perflib* optimization demonstrates a significant speedup over previous implementations. As seen in Table 6.4 on the previous page, this build achieves an almost eight-fold reduction in the overall execution time. This comes at no price to the accuracy. The maximum deviation is still reported as 0.0009795%.

The hardware FPU, single-precision math functions, and double-precision *Perflib* are general optimizations that can be applied to almost any type of data processing system that requires extensive floating-point arithmetic. Once these optimizations are implemented, the next step is to look at more application specific ways to reduce the execution time. For this matter, the profiling information is extremely important. The profile of the current build is shown in Figure 6.3. The most time consuming routine here is *dotprod_*, accounting for almost 10% of CPU time. This function is part of the re-sampling phase and is targeted for a hardware implementation, as described in the next section.

6.2 Dot-product Hardware Co-processor

As seen in the profile data in Figure 6.3 on the previous page, the *dotprod* function is a good candidate for evaluating the feasibility of a hardware implementation. The complete function is included in Appendix C.7 on page 117. The *dotprod* function computes the scalar product of interpolation operators and the input function. It is called by *finterp*, which is called by *t2f* as part of the re-sampling phase (refer to Figure 4.5 on page 34). The computationally intensive portion of the function is pasted below.

```
1211     for (i__ = 1; i__ <= i__1; ++i__) {
1212 /* Dot product fin(kin+1) with oper(1,j) */
1213         ret_val = ret_val + fin[i__ + kin] * oper[i__ + j * oper_dim1] + fin[*
1214                 nop + 1 - i__ + kin] * oper[i__ + jr * oper_dim1];
1215     }
```

As shown in the code segment above, the format of computing the dot product is as follows:

$$retval = retval + fin[a+] \times oper[b+] + fin[c-] \times oper[d+]$$

In the notation above, the '+' and '-' signs inside the brackets indicate the direction of change (i.e. incrementing or decrementing index). In this case, the dot-product is calculated from both ends of the operator array simultaneously. Since the largest values are typically located in the middle of the operator array, a higher degree of precision is achieved by working with these values in the final steps of the computation. However, this comes at the price of longer memory latencies as the array access is not consecutive.

One iteration in the loop requires two multiplications, an addition, and an accumulation, all in single-precision floating-point format. The loop has 28 iterations, as set by a *nop/2* (*nop* is the number of interpolation operator points and is initialized to 56 in this version of the code). A dot-product co-processor could be built from individual floating-point operators that are available as part of the Xilinx IP collection. However, it is first necessary to determine whether a hardware implementation of the dot-product calculation can be more efficient than utilizing the FPU.

6.2.1 Concept

A dot-product co-processor needs to acquire data from memory efficiently, perform the computation, and deliver the data back to the memory. One way to facilitate such a data transfer is through a direct memory access (DMA) engine. A DMA engine can be integrated

to one of the system buses and can transfer data from memory to a co-processor without the CPU getting directly involved. The CPU just needs to authorize the transfer (from where and to where), and sometime later it will receive an indication that the transfer is complete. However, such an approach will not work well in this particular dot-product implementation because the data comes from non-contiguous memory locations and it comes in short bursts (28 loop iterations).

A good alternative is to use the APU controller and develop the dot-product co-processor as a load/store peripheral. APU load/store instructions are one of three types that can be decoded by an attached FCM, in this case the dot-product co-processor. Furthermore, APU load/store instructions can transfer up to 16 bytes of data (quadword), which is the exact amount in one dot-product loop iteration. The CPU is involved in the transfer which means that any data that is in cache will be fetched from there. On the one hand this is good because cache access is very fast, however, this does take up CPU time whereas the DMA engine does not. Nevertheless, an APU load/store co-processor is a practical solution given the data access pattern. From the point of view of the software, this is the best solution as only minimal changes to the source are required.

In the best case, a quadword APU load can be issued on every seventh clock cycles (at FCB clock rate). This is given that instruction decode takes one cycle, the four data words are transferred immediately after (one per cycle), and the co-processor responds with a '*done*' signal immediately after the data has been transferred. In the worst case, the data transfer or co-processor response can take many more cycles, and thus is not a valid comparison metric [25]. As far as the FPU, a single word load takes at best three cycles. Two loads can be followed by a fused multiply-add, which has a ten-cycle latency. Two sets of load-load-multiply/add are needed to account for a single iteration of the dot-product loop. As demonstrated in Figure 6.4 on the following page, about 20 cycles are needed to complete one iteration of the loop [34].

Assuming that the clock rate is the same between the FPU and an APU-attached dot-product co-processor (i.e.the system bus clock, half-rate from the CPU), and the hardware receiving the load data can keep up with the 7-cycle issue rate, the best case speedup factor is $20 \div 7 = 2.86x$. If the APU-attached dot-product co-processor is capable of running at the CPU clock rate, then the best case speedup factor is double, or $5.72x$. These numbers are preliminary but provide a rough estimate of what to expect.

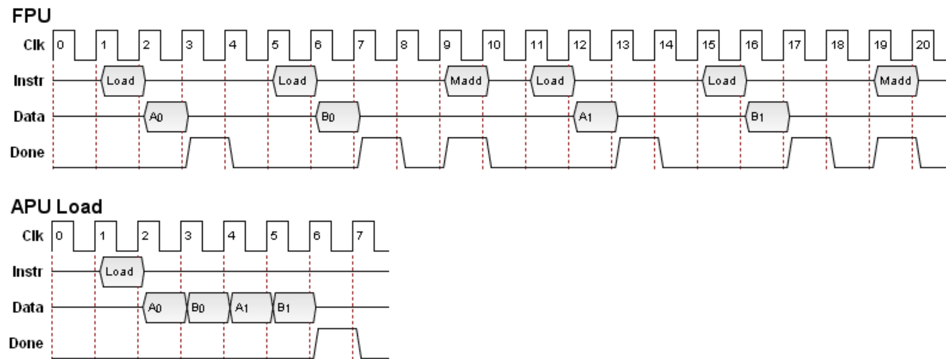


Figure 6.4: Cycles needed to complete one iteration of dot-product loop

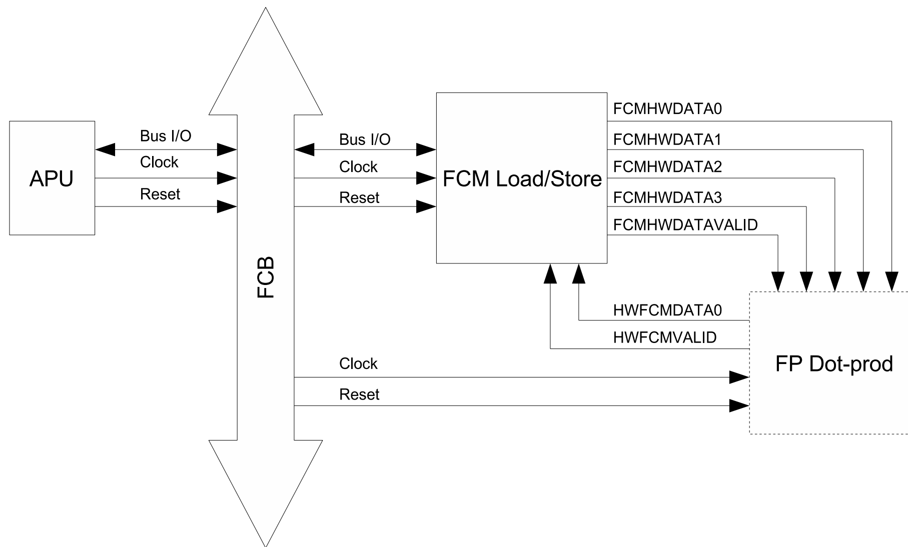


Figure 6.5: FCM load/store unit interface to FCB and dot-product co-processor

6.2.2 Load/Store Unit Design

The load/store unit is the bus front end for the dot-product module. It is responsible for decoding APU load/store instructions and delivering data to/from the dot-product co-processor. The load/store unit is a modified version of the design provided in a Xilinx application note [35]. The modifications include various bug fixes to ensure proper compliance with the FCB interface, input/output to the dot-product co-processor, and control logic to facilitate the data transfer. The diagram in Figure 6.5 shows the load/store unit interface to the FCB and the dot-product module.

The load/store unit contains two 4-entry register banks for 32-bit load/store data. The load register bank is available on the output of the core for direct connection to

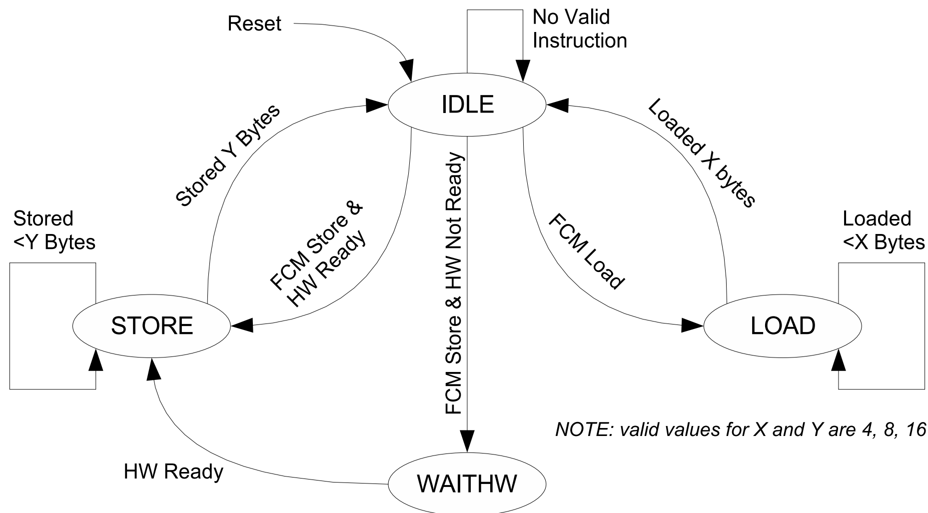


Figure 6.6: FCM load/store unit state machine

custom hardware, in this case the dot-product co-processor. In this implementation, the first entry in the store register bank is tied to the output of the custom hardware, with the other three entries inaccessible. The load/store unit expects that a single store instruction will always follow a certain number of load instructions. In other words, the load instructions transfer data to the hardware co-processor and the store instruction reads back the result.

There are four states in the load/store control logic - *idle*, *load*, *waithw* (wait for hw), and *store* (see Figure 6.6). On reset, the core starts in the *idle* state. In that state, every APU instruction that comes out on the FCB is evaluated to see whether it is a load/store. Since the APU controller and the FCB only support unique co-processors, there could only be one unit which decodes load/store instructions. On a valid load instruction, the unit goes into the *load* state and stays there until all data has been loaded (4 single-precision floats, or 16 bytes in this case). On a valid store instruction, the unit goes into the *store* state but only if the connect HW co-processor (i.e. the dot-product core) has finished the computation. If the co-processor is still working on the data set, the load/store unit goes into the *waithw* state until the co-processor is done, at which point the transition to the *store* state is automatic. The load/store unit stays in the *store* state until all data has been transferred to the APU (1 single-precision float, or 4 bytes in this case). Not shown in the state diagram in Figure 6.6 is the immediate transition to the *idle* state from any one of the other three states when the *APUFCMFLUSH* signal is asserted. This happens when the APU controller needs to flush the outstanding instruction and

must reissue it at a later time. The complete Verilog source for the load/store unit is presented in Appendix C.9 on page 121.

In the software, specific assembly instructions must be used to transfer data to the load/store unit. For example, the quadword load and single word store instructions are defined as the following assembly mnemonics [36]:

```
1  #define lqfcmx(rn, base, adr)  __asm__ __volatile__
2  ("lqfcmx " #rn ",%0,%1\n" : : "b" (base), "r" (adr))
3
4  #define stwfcmx(rn, base, adr) __asm__ __volatile__
5  ("stwfcmx " #rn ",%0,%1\n" : : "b" (base), "r" (adr))
```

In the code segment above, the *lqfcmx* is the quadword load instruction and *stwfcmx* is the single word store instruction. The destination register for a load or source register for a store is specified in place of *rn*. The base memory address is specified in place of *base*, and the byte offset is specified in place of *adr*. Thus, to load a quadword from *src[4]* to FCM register 0, the following must be written:

```
lqfcmx(0, src, 16);
```

For the above to work properly, the *src* array must be aligned on the 32-byte boundary, as follows (assuming 8 elements):

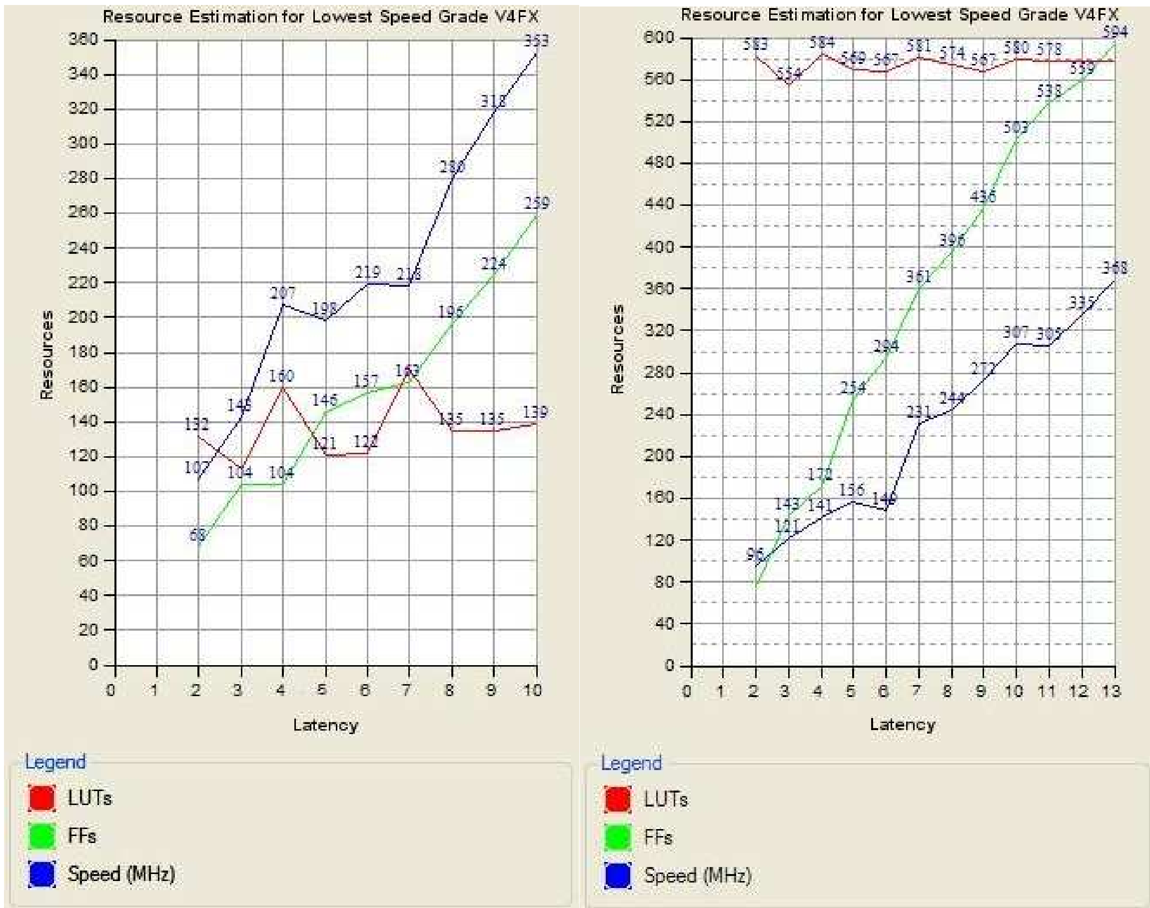
```
real __attribute__ ((aligned (32))) src[8];
```

The alignment is necessary because the load/store instructions are actually PowerPC AltiVec vector instructions which require aligned data[1]. For example, the Xilinx modified compiler actually interprets the *lqfcmx* instruction as an *lvx* (load vector indexed) AltiVec instruction. The *stwfcmx* instruction is interpreted as a *stvx* (store vector indexed) AltiVec instruction. This is hidden from the developer and is only visible when debugging the assembly code [37, 38].

6.2.3 Dot-product Core Design

The dot-product core needs to work with four single-precision floating-point numbers at a time. This is the amount of data that is delivered to the core by the load/store unit every seven cycles. To calculate the dot-product, the core needs two multipliers in parallel, followed by an adder and then an accumulator. Floating-point adders and multipliers

[1] AltiVec instructions require data to be aligned on the 16-byte boundary, however, Xilinx reference designs place the alignment on the 32-byte boundary.



(a) Multiply operator, using 4 DSP48 slices

(b) Add operator, using logic only

Figure 6.7: Resource usage vs. latency for the multiply and add floating-point operators

are available as individual operators from the Xilinx LogiCORE IP collection. However, they must be properly configured to meet the frequency and latency requirements of the dot-product core.

It is desirable that the individual floating-point operators work at 200 MHz and don't exceed seven cycle latency. At 200 MHz, the operators will match the frequency of the processor and the FCB, thus requiring no additional re-synchronization logic. With a latency of seven cycles or less, the operators will not require any buffers as new data will come in once every seven cycles from the load/store module. Figure 6.7 shows the resource trade-offs (look-up tables and flip-flops) for different latencies for the floating-point multiplier and adder operator. It was determined that the optimal resource trade-offs for the target frequency are obtained under a 4 DSP48 slice usage for the multiplier and under a logic only implementation for the adder.

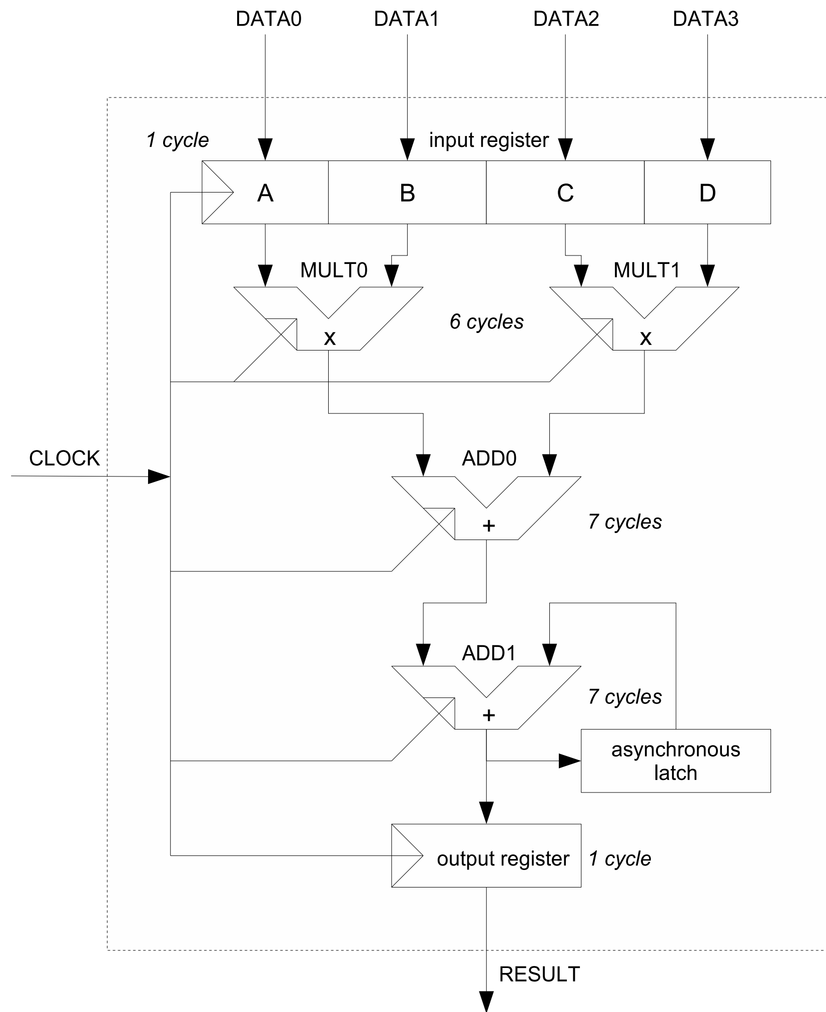


Figure 6.8: Dot-product core block diagram

For the multipliers, a target latency of six cycles was chosen as it can deliver over 200 MHz performance. This is the optimal configuration as on every seventh cycle new data will appear on the inputs and thus the multipliers are busy 6 out of 7 cycles. The adders, however, have a tighter margin. The lowest latency that delivers at least a 200 MHz performance is seven cycles. This isn't a problem for adding the result of the multipliers in parallel, however, the accumulator will need to have its output latched so that the data is immediately available for the start of every seventh cycle, for the purpose of accumulating. A block diagram for the dot-product core is shown in Figure 6.8 (not shown are the various handshaking signals). The total latency of the core is 22 cycles, after which the result is produced every seven cycles.

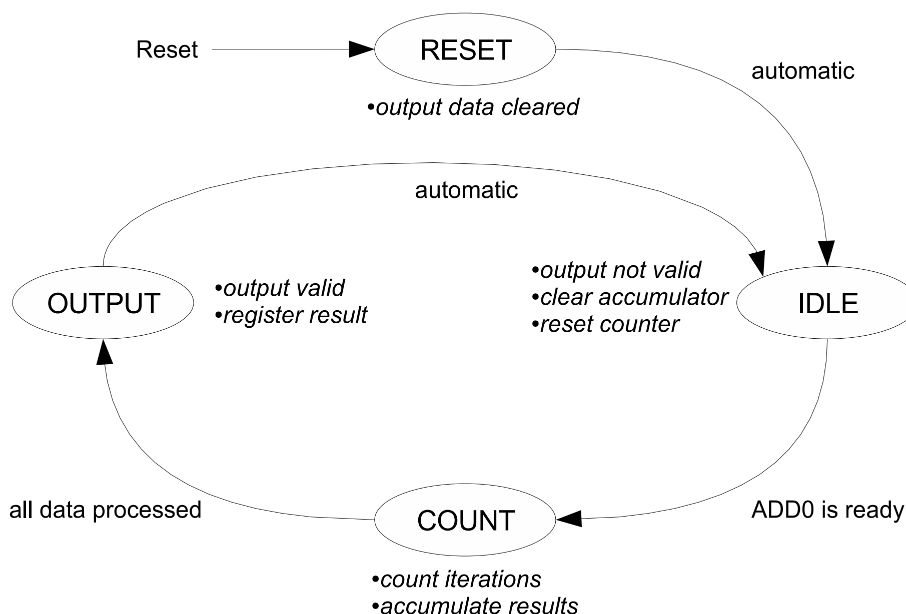


Figure 6.9: Dot-product core state machine

There are four states that control the operation of the dot-product core - *reset*, *idle*, *count*, and *output* (see Figure 6.9). On reset, the core starts in the *reset* state, clears the output register, and automatically transitions to the *idle* state. In the *idle* state, the output valid line is lowered (indicates invalid output), the accumulator latch is asynchronously cleared, and the counter variable is initialized to 0. When ADD0 reports that its output is valid, the core transitions to the *count* state and the accumulator latch clear is de-asserted. In the *count* state, the counter is incremented every time ADD1 reports that its output is valid. During this process, the accumulator latch clear is held low to allow the output data to accumulate. Once the counter reaches the predefined number of loop iterations (set by a generic to match the half width of the convolution operator - in this case 28), the core transitions to the *output* state where the accumulated result is registered and the output is marked as valid. From the *output* state, the core automatically transitions back to the *idle* state. The complete VHDL source for the dot-product core is presented in Appendix C.10 on page 131.

6.2.4 Behavioral Simulation

Prior to implementation, the dot-product core is first synthesized and simulated to verify correct logical functionality. The synthesis tools report a maximum clock frequency of 237 MHz, which is well above the minimum desired frequency of 200 MHz. The behavioral

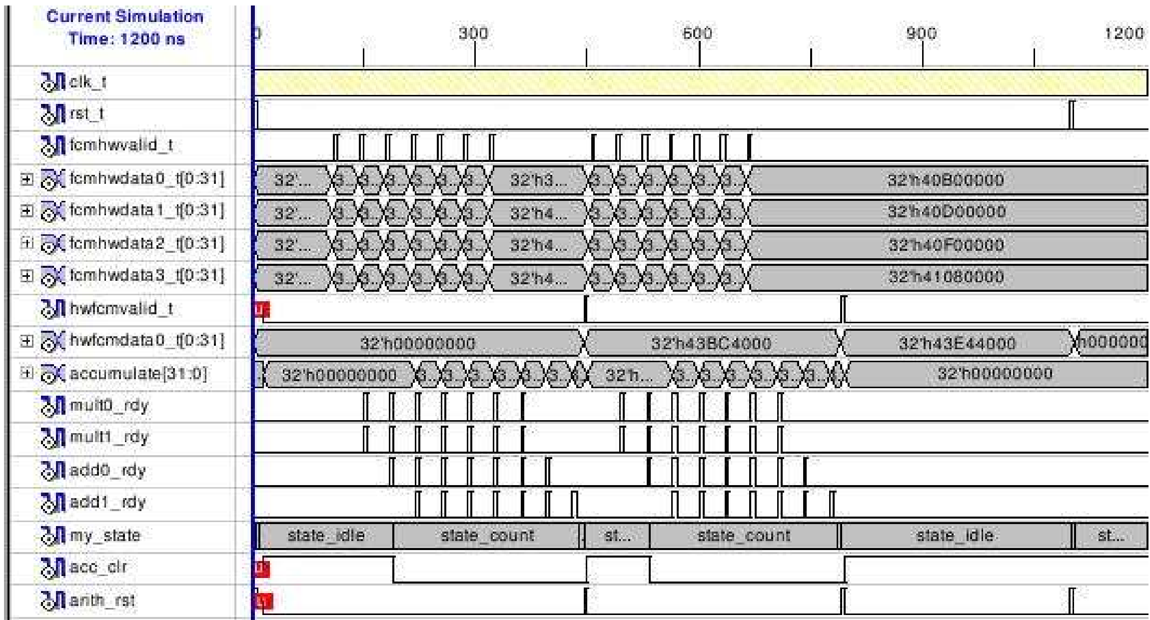


Figure 6.10: Dot-product core behavioral simulation

simulation, shown in Figure 6.10, covers two sets of computations each with seven loop iterations. Each computation set has two types of data sequences - sequence A (1.5, 2.5, 3.5, 4.5) and sequence B (5.5, 6.5, 7.5, 8.5). The first computation set simulated the loading of data in the pattern *ABABABA*. The dot-product result for this pattern is 376.5, which correctly corresponds to the simulation output of *0h43BC4000*. The second computation set simulated the loading of data in the pattern *BABABAB*. The dot-product result for this pattern is 456.5, which correctly corresponds to the simulation output of *0h43E44000*. From these two tests, as well as others not described here, the dot-product core is verified to produce accurate results.

6.2.5 System Deployment

When implementing the dot-product core without first integrating it into the FTIR spectrometry FPGA system, it is necessary to set a few timing constraints. In particular, the clock must be constrained to 200 MHz and the accumulator latch must be constrained to produce output in less than one clock cycle from the rising edge of the clock. This is to ensure that the latched data appears on the input of the accumulator just before the rising edge of the clock starts off the next addition. The constraints file is pasted below. When deploying in the FPGA system, the clock constraint will automatically be specified, but the latch constrain must be manually added.

Dot-product Core Summary				
Target Device	**Speed Grade**	**Slices**	**DSP48 Blocks**	**Max Freq.**
V4FX60	-11	1021	8	237 MHz

Table 6.5: Dot-product Core Summary

```
1  # Set the correct stepping for ML410 V4FX60 FPGA
2  CONFIG STEPPING="SCD1";
3
4  # System level constraints
5  Net CLK TNM_NET = CLK;
6  TIMEGRP "RISING_CLK" = RISING "CLK";
7
8  # Constrain CLK to 200 MHz
9  TIMESPEC TS_CLK = PERIOD CLK 5 ns;
10
11 # Constrain the accumulate latch (one CLK cycle)
12 TIMESPEC TS_ACC_LATCH = FROM "RISING_CLK" TO LATCHES( accumulate<*> ) TS_CLK * 0.99
       DATAPATHONLY;
```

With the timing constraints specified above, the PAR effort must be set to “high” with extra effort set to “normal” in order to meet timing. The resource utilization for the dot-product core is shown in Table 6.5.

The dot-product core is added to the FTIR base system initially without the FPU co-processor. This is done to first validate proper dot-product core functionality in hardware before putting it on the FCB with another unit (i.e. the FPU). PAR simulations are not done for the core since integrating it into the actual hardware and running it there takes less time than building and simulating a complete FCB - load/store - dot-product subsystem. The system diagram for this build is presented in Figure 6.11 on the following page. The device utilization is shown in Table 6.6 on page 64.

6.2.6 Software Considerations

The GCC compiler is not well suited to work with custom hardware that integrates with the processor down at the instruction level. This is quite a nuisance especially when compiler optimization is turned on. Such optimizations often cause incorrect rearrangement of instructions and over-optimization of data accesses. For this matter certain modification are required in the software, some of which are very obscure.

One of the very first actions to be taken by the software is the enabling of the APU by writing to the machine state register (MSR). This is only necessary when the dot-product core is present without the FPU. When the FPU is present, the MSR is properly set in the boot sequence.

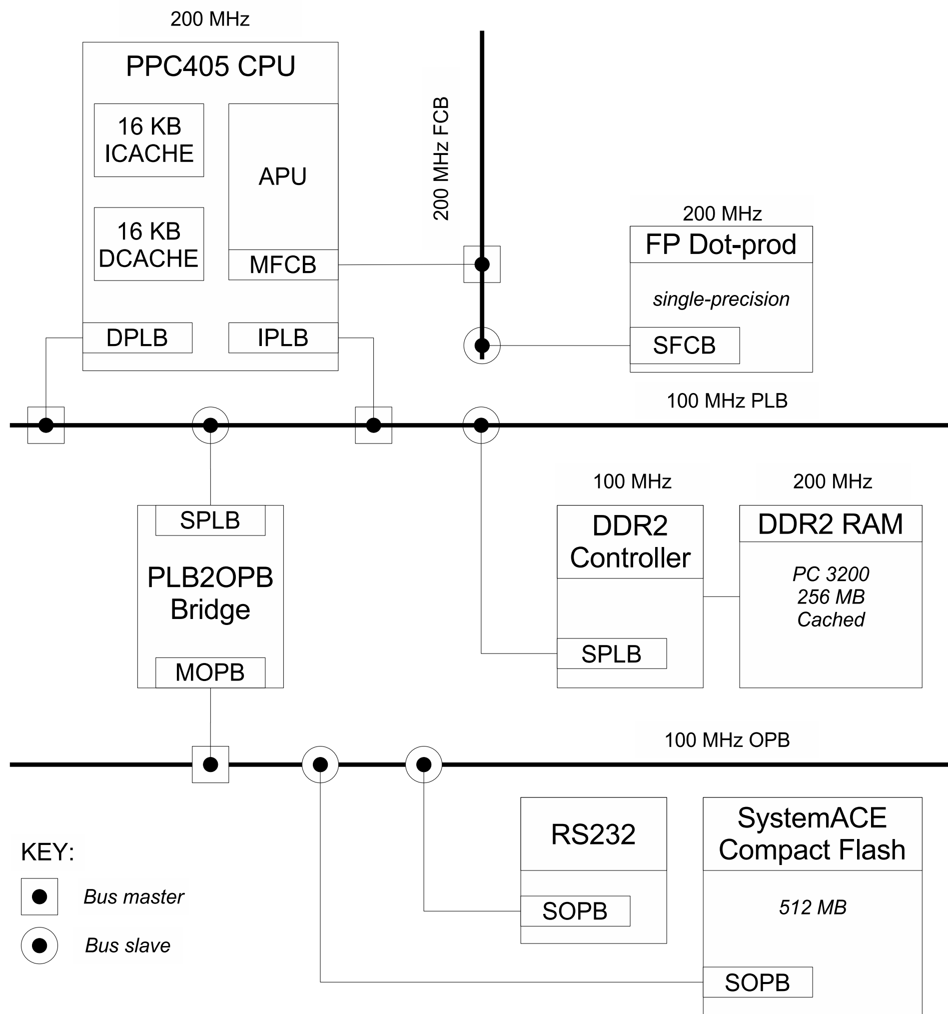


Figure 6.11: FTIR system with dot-product co-processor

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	4,985	50,560	9%	
Number used as Flip Flops	4,953			
Number used as Latches	32			
DCM autocalibration logic	14	4,953	1%	
Number of 4 input LUTs	4,186	50,560	8%	
DCM autocalibration logic	8	4,186	1%	
Logic Distribution				
Number of occupied Slices	4,558	25,280	18%	
Number of Slices containing only related logic	4,558	4,558	100%	
Number of Slices containing unrelated logic	0	4,558	0%	
Total Number of 4 input LUTs	5,199	50,560	10%	
Number used as logic	4,186			
Number used as a route-thru	170			
Number used for Dual Port RAMs	648			
Number used as Shift registers	195			
Number of bonded IPADs	32	72	44%	
Number of bonded OPADs	32	32	100%	
Number of bonded IOBs	146	576	25%	
Number of BUFG/BUFGCTRLs	7	32	21%	
Number used as BUFGs	7			
Number used as BUFGCTRLs	0			
Number of DSP48s	8	128	6%	
Number of DCM_ADVs	2	12	16%	
Number of PPC405_ADVs	2	2	100%	
Number of JTAGPPCs	1	1	100%	
Number of IDELAYCTRLs	4	20	20%	
Number of GT11s	16	16	100%	
Number of RPM macros	24			
Total equivalent gate count for design	129,346			
Additional JTAG gate count for IOBs	10,080			

Table 6.6: Device utilization summary for FTIR system with dot-product co-processor

```
1  /* local includes */
2  #include "xreg405.h"       /* for other system #defines (APU controller) */
3
4  /* Define instruction to set MSR */
5  #define mtmsr(v) __asm__ __volatile__ ("mtmsr %0\n" : : "r" (v))
6
7  /* ... somewhere in main ... */
8  /* Initialize APU (not necessary when FPU is present) */
9  mtmsr(XREG_MSR_APU_AVAILABLE);
```

The *dotprod* function is completely removed from the main source and placed into a separate file to be compiled into a static library (see Appendix C.8 on page 119). This is done to prepare for later system builds which also include the FPU. The *dotprod* function, however, must be compiled separately without FPU support.

A few changes are made to the computation structure in the *dotprod* function. In the loop that computes the dot-product, all the operands are first collected into an array in the right order. Then a second loop iterates over the newly built array and calls the dot-product co-processor. Finally, the result is stored into an externally defined, volatile variable (*dphw_result*). This variable needs to be defined volatile since it is modified outside the scope of the compiler. The relevant code segment from the *dotprod* function is pasted below.

```
94      indexer = 0;
95      for (i__ = 1; i__ <= i__1; ++i__) { /* !!! ASSUMING nop/2 = 28 !!! */
96        src[indexer] = fin[i__ + kin];
97        src[indexer+1] = oper[i__ + j * oper_dim1];
98        src[indexer+2] = fin[*nop + 1 - i__ + kin];
99        src[indexer+3] = oper[i__ + jr * oper_dim1];
100       indexer+=4;
101     }
102     for (i__ = 1; i__ <= i__1; ++i__) {
103       /* Dot product fin(kin+1) with oper(1,j) */
104       lqfcmx(0, src, (i__-1)*16);
105     }
106     /* compile with this to force proper assembly code */
107     /* then remove by hand (in assembly) and rebuild */
108     usleep(1);
109     stwfcmx(0, &dphw_result, 0);
```

One particular line in the code segment above warrants a detailed explanation. On line 108, the function *usleep* is called right before the result is read back from the co-processor. What the function does is irrelevant (it is a microsecond sleep function), but its presence is important. When developing the sequence of instructions to access the dot-product co-processor, it was noticed that data was not being sent to the co-processor

properly when compiler optimizations were turned on. Compiler optimizations are necessary because without them the performance suffers tremendously. After closely inspecting the generated assembly, it was observed that all stores into the *src* array were ignored. That is because the compiler was not aware that data from the *src* array was being used by the following load instruction (*lqfcmx*) as it is an extension to the standard PowerPC ISA of the PPC405 embedded processor. Therefore, by placing the *usleep* function call in the source, the compiler is forced to guarantee the correct state of all data in the *dotprod* function before entering *usleep*. Of course, the *usleep* function is not really desired in the *dotprod* source as it would adversely affect the performance. Thus, the workaround is to first compile the C-source to assembly with the *usleep* function present, then remove the call to *usleep* in the generated assembly, and finally recompile the assembly into a static library. Although this is somewhat of a hack, it is the only solution that works consistently and produces correct results between different system builds, with compiler optimization turned on and with or without the FPU co-processor present. This procedure is explained in the header of the *dotprod* function in Appendix C.8 on page 119.

In the top level source that calls the *dotprod* function (in *finterp* function), one subtlety that must be pointed out is the necessity to refresh the *dphw_result* variable prior to using it as an operand. This is demonstrated in the code segment below.

```
1 /* call dotprod_ function in static library */
2 dotprod_(&fin[1], nin, &oper[oper_offset],nop, nso, &xx);
3 dphw_result; /* necessary to refresh volatile */
4 ret_val += fr * dphw_result;
```

6.2.7 Accuracy and Performance Evaluation

The FTIR system with the dot-product co-processor builds successfully, meeting all timing constraints and producing results with a maximum deviation of 0.0009795%. As shown in Table 6.7 on the following page this system has a lower total execution time than the SW-only optimized FTIR build (with SP math functions and *Perflib*). However, the execution time is higher than on the FTIR system with the FPU co-processor (FPU system speedup = 7.95x). This is because the FPU optimizes all single-precision floating-point arithmetic whereas the dot-product co-processor only optimizes the dot-product calculation. Nevertheless, the core produces accurate results and achieves a good speedup over

SW Optimizations	None (base system)	SP Math Functions IBM Perflib	
HW Optimizations	None (base system)	Dot-prod (200 MHz)	
PPC405 Freq.	200 MHz	200 MHz	
Interferograms	1	1	
Detectors	1	1	
Software Component	Time (sec)	Time (sec)	Speedup
Re-sampling	1197.7938	211.4479	5.66x
Spectrum (Phase Correction, FFT)	117.4963	44.4456	2.64x
TOTAL	1315.2901	255.8935	5.14x

Table 6.7: Execution times for system with dot-product co-processor, SP math functions, and *Perflib*

the software.[2].

6.3 FPU / Dot-product Compatibility and Integration

With the dot-product core functionality verified, the FPU is put back into the system (see Figure 6.12 on the next page). This configuration, where the FPU is sharing the FCB with another co-processor, has never been tested by Xilinx. After a very productive collaboration with the Xilinx FPU designer, various hardware issues were identified and addressed.

6.3.1 Hardware Issues

There are two issues with the FPU in Xilinx Platform Studio 9.1.02i that prevent it from working properly with another co-processor on the FCB. The first issue deals with instruction decoding. Table 6.8 on page 69 shows the op-codes of various instructions decoded by the APU controller. Both the FCM load/store instruction and the FPU load-/store instruction share the same primary op-code 31. The only difference between the two instructions is in the first bit of the extended op-code. While debugging with the ChipScope Logic Analyzer core, it was found that the FPU attempted to decode ordinary FCM load/store instructions. This, of course, caused a conflict on the FCB as the load-/store core also attempted to decode this instruction. Xilinx quickly fixed this issue and delivered an updated FPU core that properly ignored FCM load/store instructions[3].

[2]For this particular build, profiling information could not be obtained most likely due to bugs in the SDK profiler. This is not uncommon as the SDK tools are constantly under development.

[3]The updated FPU should be available in future releases of XPS.

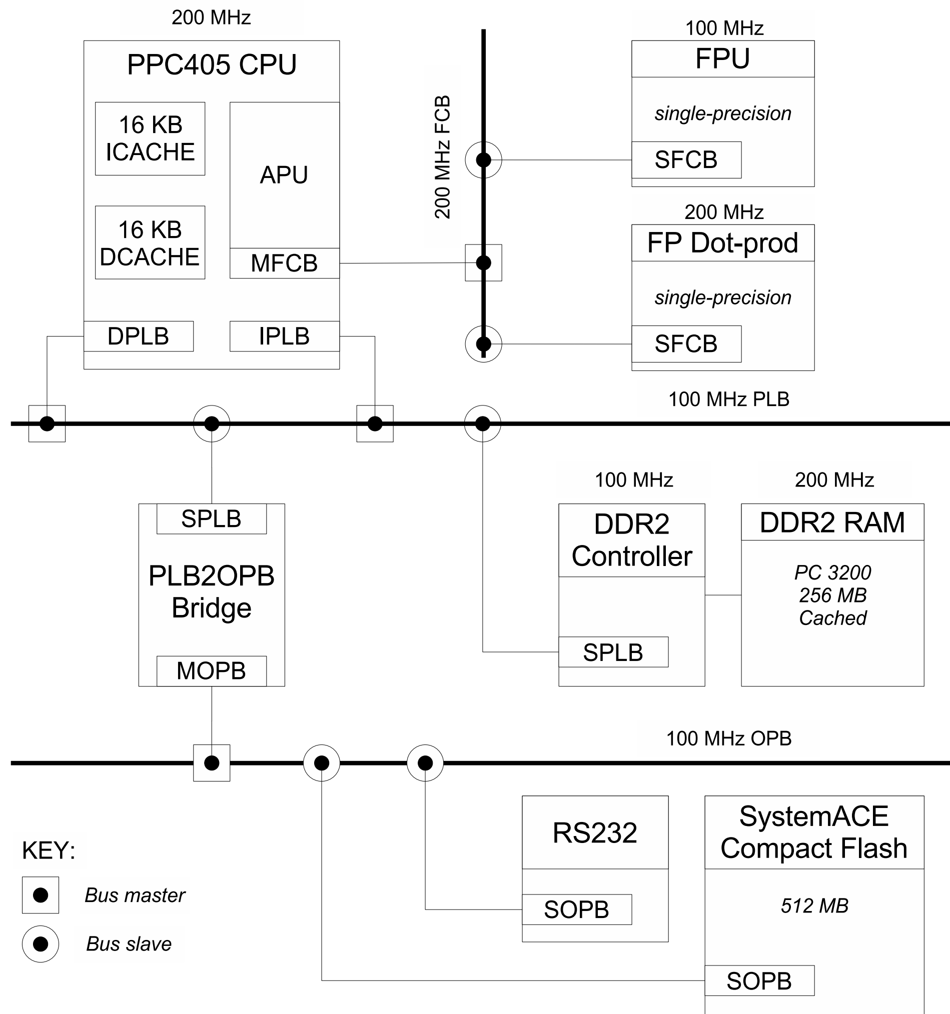


Figure 6.12: FTIR system with dot-product and FPU co-processors

The second issue deals with the *FCMAPULOADWAIT* signal that is part of the FCB interface. This signal is used to indicate to the APU controller that the FCM is not yet ready to receive the next load data. The FPU constantly toggles this line as part of its internal synchronization logic that lets it operate at half the clock rate of the FCB. When other cores are present on the FCB, toggling the *FCMAPULOADWAIT* signal interferes with their operation since the FCB is basically a wired-or bus. This was the case with the dot-product co-processor, which would not load data correctly with the FPU toggling the *FCMAPULOADWAIT* signal. According to Xilinx, the FPU only needs to use the *FCMAPU-LOADWAIT* signal under two conditions:

Primary Op-code	Extended Op-code	Description
0 (= 0b000000)	0b00000000000	Illegal
	all except above	Available for UDIs[(1)] that do not set PPC405(CR) bits
4 (= 0b000100)	0b------1--0-	MAcc and Xilinx reserved
	0b1----000110	Available for UDIs that need to set PPC405(CR) bits
	all except above	Available for UDIs that do not set PPC405(CR) bits
31 (= 0b011111)	0b-----001110	Pre-defined FCM Load/Store
	0b-111-010-1-	FCM integer divide
(= 0b110---)[(2)]	0b-----------	Pre-defined FPU Load/Store
31 (= 0b011111)	0b1----1-111-	Pre-defined FPU Load/Store
59 (= 0b111011)	0b-----------	Pre-defined PowerPC FPU instructions
62 (= 0b111110)	0b--------1--	Pre-defined FPU Load/Store
63 (= 0b111111)	0b----------	Pre-defined PowerPC FPU instructions

Notes:

1. User-defined Instruction.

2. In this case, the first three bits are defined and the last three will change depending on the FPU instruction.

Table 6.8: Instruction op-codes decoded by the APU controller [25]

1. The APU controller sends the two halves of a double-precision load transfer back-to-back, and the FPU can't keep up.

2. The APU controller flushes an outstanding operation and then immediately provides load data to the FPU before it has had time to process the flush.

The first scenario does not apply to this design as the FPU only supports single-precision floating-point operation. The second scenario is, according to Xilinx, "unlikely" although the probability of it happening depends on the code being executed. Since the FPU implementation is hidden from the developer, the only possible (but not ideal) solution to this issue is to disconnect the *FCMAPULOADWAIT* signal from the FPU. Future releases of Xilinx XPS will most likely fix this issue at its source.

Once the hardware issues described above are addressed, the system builds correctly, meeting all timing constraints. The device utilization is shown in Table 6.9 on the next page. This is the maximum utilization of FPGA resources seen in this entire research project with only 24% of the slices occupied.

Device Utilization Summary				
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	6,508	50,560	12%	
Number used as Flip Flops	6,476			
Number used as Latches	32			
DCM autocalibration logic	14	6,476	1%	
Number of 4 input LUTs	6,722	50,560	13%	
DCM autocalibration logic	8	6,722	1%	
Logic Distribution				
Number of occupied Slices	6,276	25,280	24%	
Number of Slices containing only related logic	6,276	6,276	100%	
Number of Slices containing unrelated logic	0	6,276	0%	
Total Number of 4 input LUTs	7,968	50,560	15%	
Number used as logic	6,722			
Number used as a route-thru	345			
Number used for Dual Port RAMs	648			
Number used as Shift registers	253			
Number of bonded IPADs	32	72	44%	
Number of bonded OPADs	32	32	100%	
Number of bonded IOBs	146	576	25%	
Number of BUFG/BUFGCTRLs	7	32	21%	
Number used as BUFGs	7			
Number used as BUFGCTRLs	0			
Number of FIFO16/RAMB16s	2	232	1%	
Number used as FIFO16s	0			
Number used as RAMB16s	2			
Number of DSP48s	12	128	9%	
Number of DCM_ADVs	2	12	16%	
Number of PPC405_ADVs	2	2	100%	
Number of JTAGPPCs	1	1	100%	
Number of IDELAYCTRLs	4	20	20%	
Number of GT11s	16	16	100%	
Number of RPM macros	28			
Total equivalent gate count for design	295,514			
Additional JTAG gate count for IOBs	10,080			

Table 6.9: Device utilization summary for FTIR system with dot-product and FPU co-processors

6.3.2 Accuracy and Performance Evaluation

The FTIR system with dot-product and FPU co-processors produces valid results with the maximum deviation still 0.0009795%. However, the execution time of this system is nearly the same as that of the FPU-only system (see Table 6.10 on the following page). This is somewhat surprising as the expected best case speedup of a system with both the FPU and the dot-product co-processor should have been close to 5.72x for the dot-product function. Since this function accounts for about 10% of the CPU time, or roughly 16.55 sec (as seen in the profile data in Figure 6.3 on page 52 and the execution times in Table 6.4 on page 51), the net reduction in time should have been at best 13.66 sec in the re-sampling phase. However, less than a 2 sec reduction was observed.

One explanation for the lack of significant reduction in the execution time can be attributed to a higher than expected overhead in getting the data from memory to the dot-product co-processor. To validate this theory, a number of similar systems were built and the performance of the dot-product core was compared to the FPU when working with a smaller data set. The code segment below is an excerpt from the *dotprod* function. The *fin* and *oper* arrays are each 3,472 elements long, built from *sinf* and *cosf* functions, respectively. The loop iterates 28 times (*max_iter* = 28), and the whole dotprod function is called 62 times, covering all values in the *fin* and *oper* arrays. These values were chosen so the performance could also be evaluated when the data is in the on-chip BRAM (64 Kbytes). To optimize performance, *Perflib* is utilized in all tests. The results of these tests comparing the dot-product core to software-only implementation and to the FPU co-processor are shown in Table 6.11 on the next page.

```
1  j = 0;
2
3  /* align data */
4  for (i = 0; i < max_iter; i++ ) {
5     src[j] = fin[i];
6     src[j+1] = oper[i];
7     src[j+2] = fin[i+max_iter];
8     src[j+3] = oper[i+max_iter];
9     j += 4;
10 }
11
12 /* send to dot-prod core */
13 for (i = 0; i < max_iter; i++ ) {
14    lqfcmx(0, src, i*16);
15 }
16
17 /* compile with this to force proper assembly code */
18 /* then remove by hand (in assembly) and rebuild */
19 usleep(1);
20 stwfcmx(0, &hw_result, 0);
```

SW Optimizations	None (base system)	SP Math Functions IBM Perflib		SP Math Functions IBM Perflib	
HW Optimizations	None (base system)	APU-FPU (100 MHz) Dot-prod (200 MHz)		APU-FPU (100 MHz)	
PPC405 Freq.	200 MHz	200 MHz		200 MHz	
Interferograms	1	1		1	
Detectors	1	1		1	
Software Component	Time (sec)	Time (sec)	Speedup	Time (sec)	Speedup
Re-sampling	1197.7938	149.3297	8.02x	151.1817	7.92x
Spectrum (Phase Correction, FFT)	117.4963	14.3675	8.18x	14.3672	8.18x
TOTAL	1315.2901	163.6972	8.03x	165.5489	7.95x

Table 6.10: Execution times for system with dot-product and FPU co-processor, SP math functions, and DP *Perflib*

CPU (MHz)	FPU (MHz)	Dot-prod (MHz)	Memory	SW-only (sec)	FPU (sec)	Dot-prod (sec)	Speedup
200	-	200	BRAM	0.006500670	-	0.000320910	20.26x
200	-	200	DDR2	0.002594890	-	0.000423610	6.13x
200	100	200	BRAM	-	0.000699365	0.000321390	2.18x
200	100	200	DDR2	-	0.000824490	0.000424330	1.94x

Table 6.11: Dot-product core testing with smaller data set

A few conclusions can be made from the data in Table 6.11. First, it is clearly evident that working with a smaller, more contiguous data set results in better performance than working with larger, less organized chunks of data, as seen in the FTIR spectrometry algorithm. Additionally, both the FPU and the dot-product co-processor achieve better execution times when the data set is stored entirely in BRAM. The results also indicate that the performance of the dot-product co-processor is not significantly affected by the addition of the FPU to the FCB; in other words, the two co-processors sharing the same bus are not slowing each other down. Even though the dot-product core is a valid co-processor, it only improves system performance under favorable memory access conditions, which is not the case in the FTIR spectrometry algorithm.

In the execution time profile (see Figure 6.13 on the next page), the *dotprod_* function is shown to consume less CPU time than before. With both the FPU and dot-product co-processor, the *dotprod_* function consumes 2.95% of the CPU time, compared to the previously observed value of 9.97% (see Figure 6.3 on page 52). However, this does not mean the function executed faster. In fact, as seen in the results in Table 6.10 the speedup is negligible. The value for *dotprod_* function is lower because the profiler does

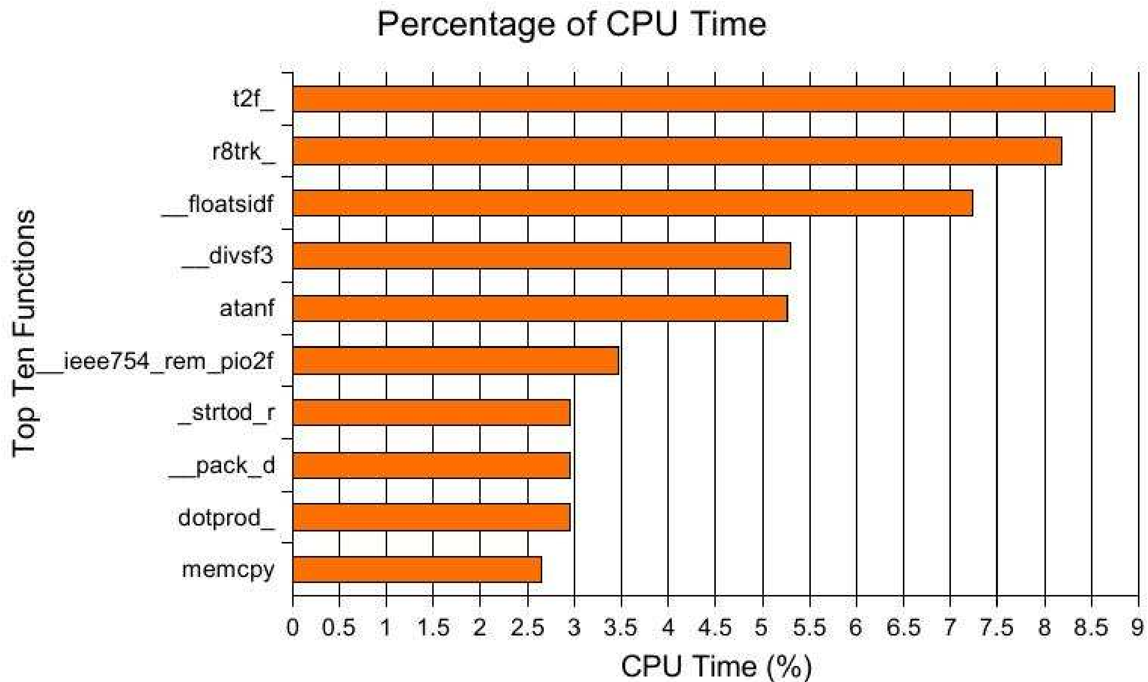


Figure 6.13: Profiling results for system with dot-product and FPU co-processors, SP math functions, and DP *Perflib*

not include helper functions that are called within the *dotprod_* function (otherwise the function *main* would consume 100% of CPU time). These helper functions could include various low level data copy and manipulation operations. In fact, the profile now shows new functions that take up significant CPU time that were not visible before (for example, *memcpy*). The cumulative contribution of the helper functions to the overall execution time is the reason why no speedup is seen with the dot-product co-processor, even with the *dotprod_* function itself taking up less CPU time.

6.4 Increased FPU System Frequencies

Short of implementing another hardware co-processor, a further reduction in the overall execution time can be attained by increasing system frequencies. This is not a trivial task as at higher clock rates it becomes harder to meet the timing constraints. Furthermore, care must be taken to maintain proper CPU/bus frequency ratios.

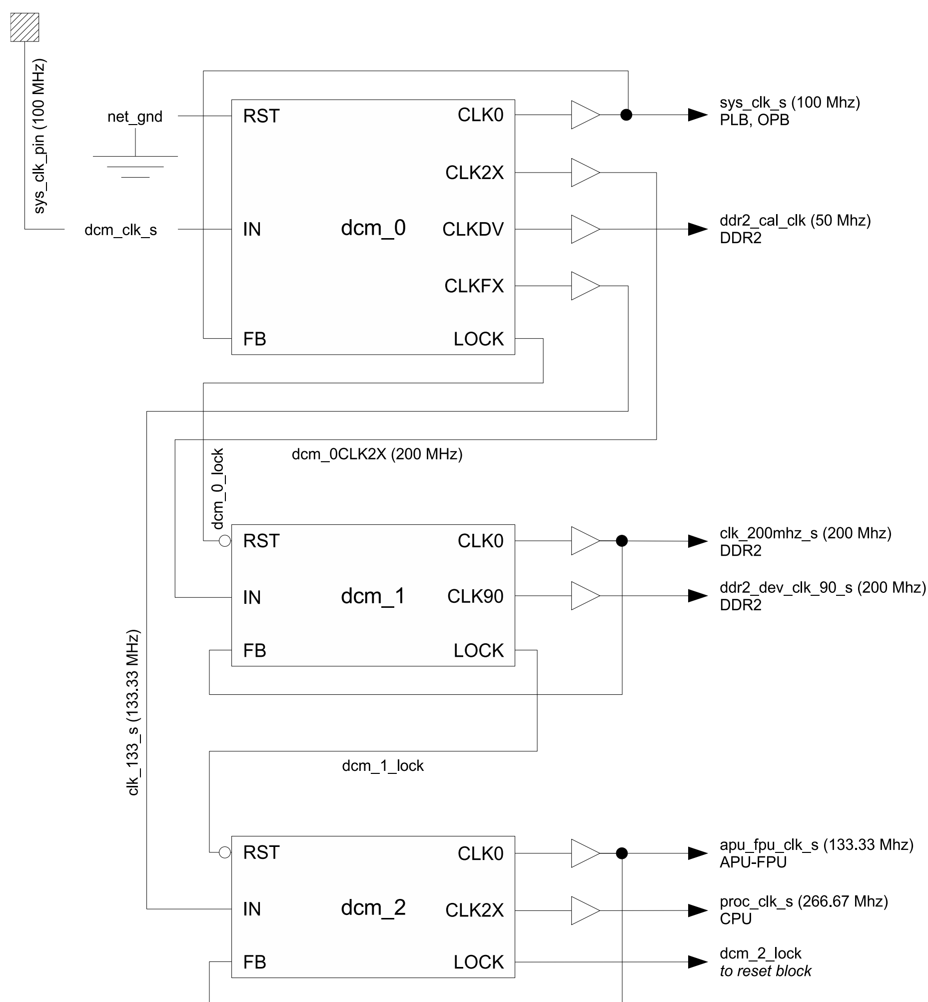


Figure 6.14: DCM configuration for a 266.67 MHz system

6.4.1 Motivation

When the APU-FPU is part of the system, the CPU can be clocked at up to 275 MHz in the -11 speed grade V4FX FPGA. The corresponding FPU frequency is thus 137.5 MHz. This ratio, however, is not a valid one for CPU and PLB clock synchronization. There are two groups of ratios that are valid for synchronizing between the CPU and the PLB and they are determined by the *CPMC405SYNCBYPASS* option in the PPC405 core configuration (Virtex-4 FX only):

1. *CPMC405SYNCBYPASS* enabled (default): integer ratios between 1:1 and 1:16 are possible

2. *CPMC405SYNCBYPASS* disabled: $N/2$ and $N/3$ ratios are possible for any integer N

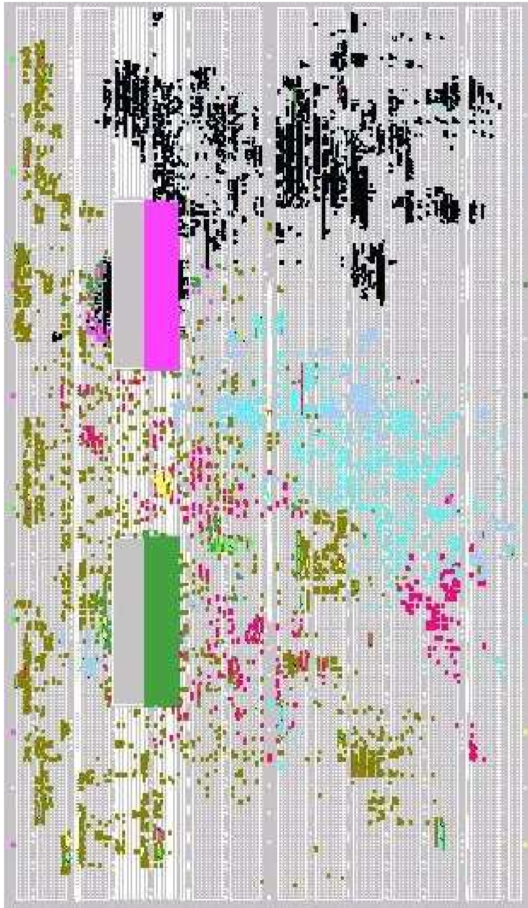
By default, the *CPMC405SYNCBYPASS* option is enabled for backwards compatibility with the Virtex-II Pro PPC405 processor. Disabling this option allows for fractional ratios provided that the CPU and PLB clocks are rising-edge aligned and the user is ready to accept additional latency for the synchronization. With this option disabled, the maximum CPU frequency under 275 MHz that is synchronized with the maximum PLB clock rate of 100 MHz is 266.67 MHz, achieving a ratio of *8/3*. The following sections describe a system build with such a ratio, allowing the FPU to run at a higher clock frequency of 133.33 MHz. A block diagram showing the cascading DCM configuration used to achieve the necessary frequencies is shown in Figure 6.14 on the preceding page. The dot-product core is removed for this build.

6.4.2 Meeting Timing

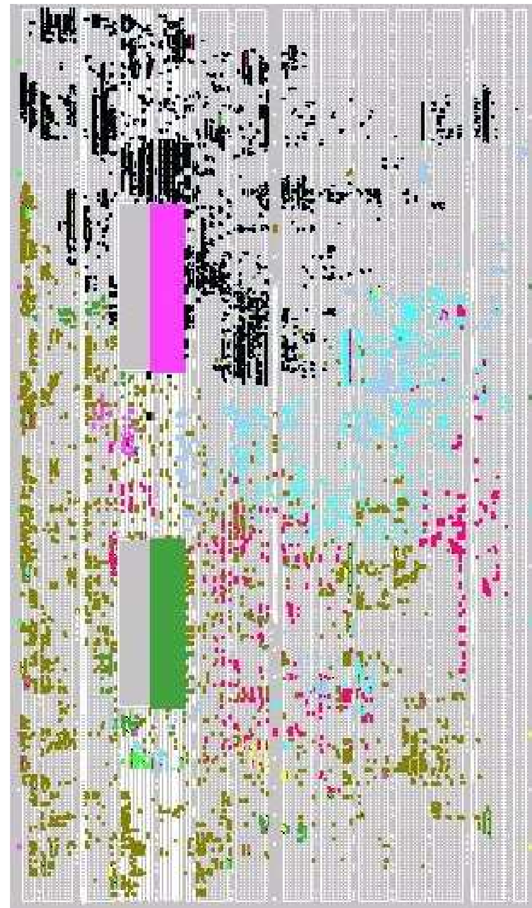
This particular system build, with higher CPU and bus frequencies, had trouble meeting timing. The reason for this can be partially contributed to the non-standard choice of frequencies: 266.67 MHz for the CPU and 133.33 MHz for the FPU. Although the DCMs can generate these frequencies from the 100 MHz reference clock, the *CLKFX* output must be used as it has options for multiplication and division of the delay-locked loop. This output has a higher jitter characteristic than the others and can thus lead to poor timing performance.

With the dedicated support of a Xilinx design engineer, and after closely examining the floorplan of the system not meeting timing, it was found that the FPU was being placed too far away from the CPU. Since the PAR tools would not place the FPU closer to the CPU even when running on "high" effort with extra effort "normal," the only alternative was to manually constrain the placement of the FPU in the UCF file. This was done by constraining the two BRAMs used by the FPU core to be located as close as possible to the CPU. With this fix, the system met all timing constraints. The floorplans before and after manual constraining are shown in Figure 6.15 on the next page. The addition to the UCF file is shown below.

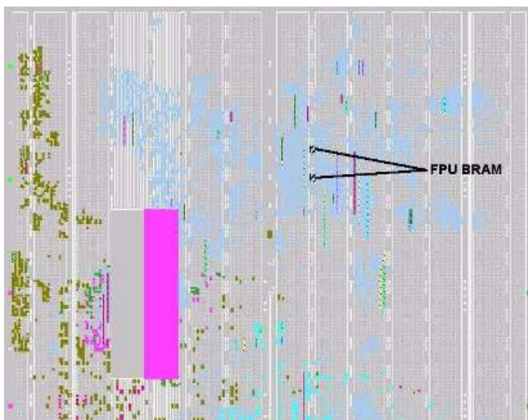
```
1 # Placement for processor and FPU (Ben Jones, Xilinx, Inc.) #
2 INST "*/ppc405_0/ppc405_0/PPC405_ADV_i" LOC = "PPC405_ADV_X0Y1";
3 INST "*/apu_fpu_0/apu_fpu_0/gen_apu_fpu_sp_full.netlist/fpu_rf_twobanks[0].msw_lsw[0].
      regmem" LOC = "RAMB16_X2Y25";
4 INST "*/apu_fpu_0/apu_fpu_0/gen_apu_fpu_sp_full.netlist/fpu_rf_twobanks[1].msw_lsw[0].
      regmem" LOC = "RAMB16_X2Y26";
```



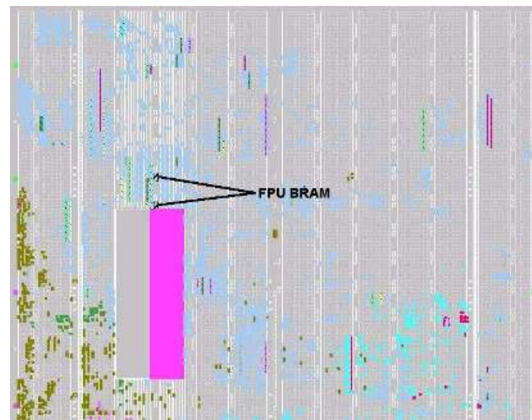
(a) Original placement of FPU (in black)



(b) FPU placement after manual constraining (in black)



(c) Original location of FPU BRAMs

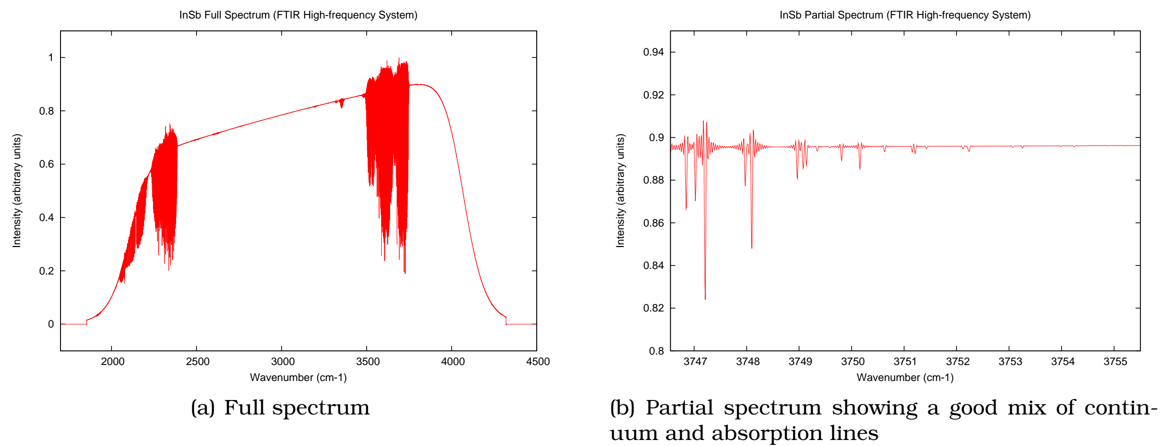


(d) Location of FPU BRAMs after manual constraining

Figure 6.15: FPU core placement before (*a*,*c*) and after (*b*,*d*) manual constraining

SW Optimizations	None (base system)	SP Math Functions IBM Perflib (DP)	
HW Optimizations	None (base system)	APU-FPU (133.33 MHz)	
PPC405 Freq.	200 MHz	266.67 MHz	
Interferograms	1	1	
Detectors	1	1	
Software Component	Time (sec)	Time (sec)	Speedup
Re-sampling	1197.7938	114.2044	10.49x
Spectrum (Phase Correction, FFT)	117.4963	12.5779	9.34x
TOTAL	1315.2901	126.7823	10.37x

Table 6.12: Execution times for a high-frequency system with APU-FPU, SP math functions, and DP *Perflib*



(a) Full spectrum

(b) Partial spectrum showing a good mix of continuum and absorption lines

Figure 6.16: The spectrum produced by the high-frequency FTIR system on the ML410 board

6.4.3 Accuracy and Performance Evaluation

The FTIR spectrometry system running on a 266.67 MHz CPU and with a 133.33 MHz FPU achieves the lowest execution time compared to all the previous system builds. A speedup of more that 10x is seen compared to the FTIR base system (see Table 6.12). Since this system build is just a higher frequency version of a previous build, device utilization, result accuracy, and the profile data remained unchanged. As a verification of the data processing accuracy, the full and partial spectrum produced by this system is shown in Figure 6.16. These plots show identical resemblance to the reference plots in Figure 4.4 on page 34.

Chapter 7

Result Analysis

This chapter brings together all the results presented throughout chapters 4-6. In particular, results obtained in this thesis are compared to the data collected from the V2P research at NASA Jet Propulsion Laboratory [2]. An estimate of the power consumption, using Xilinx's XPower tool, is given at the end of this chapter.

7.1 Performance Evaluation

Table 7.1 presents the execution times of all systems built as part of this thesis. Without a doubt, the two software optimizations (using single-precision math library and *Perflib*) and the inclusion of the APU-FPU core together provide impressive speedup across the board. That is because the aforementioned optimizations affect nearly all portions of the FTIR software, improving both single-precision (SP math functions and APU-FPU) as well as double-precision arithmetic (*Perflib*).

The dot-product co-processor core, however, has almost no effect on the system performance. As described in section 6.3.2 on page 71, this is most likely due to the non-contiguous pattern of memory access in the dot-product computation of the FTIR spectrometry algorithm. As was shown earlier, smaller more uniformly accessed data sets can achieve nearly 2x speedup over the FPU when using the dot-product co-processor.

CPU (MHz)	**FPU** (MHz)	**DOT-PROD** (MHz)	**SP MATH**	**PERFLIB**	**RE-SAMPLING** (SEC)	**SPECTRUM** (SEC)	**TOTAL** (SEC)	**SPEEDUP**
200	-	-	n	-	1197.7938	117.4963	1315.2901	1.00x
200	-	-	y	-	780.9376	109.9881	890.9257	1.48x
200	-	-	y	standard	244.1836	44.4243	288.6079	4.56x
200	100	-	y	DP-only	151.1817	14.3672	165.5489	7.95x
200	-	200	y	standard	211.4479	44.4456	255.8935	5.14x
200	100	200	y	DP-only	149.3297	14.3675	163.6972	8.03x
266	133	-	y	DP-only	114.2044	12.5779	126.7823	10.37x

Table 7.1: Execution times for all V4FX FTIR system builds

	V2P (NASA JPL)		ML410 (Xilinx)		
PPC405 Freq.	300 MHz		266.67 MHz		
Interferograms	104		scaled to 104		
Detectors	2		scaled to 2		
SW Optimizations	**IBM Perflib (standard)**		**SP Math Functions IBM Perflib (DP)**		
HW Optimizations	**None**		**APU-FPU (133.33 MHz)**		
Software Component	**Time (min)**	**Cycles** ($\times 10^{12}$)	**Time (min)**	**Cycles** ($\times 10^{12}$)	**Speedup**
Re-sampling	780	14.040	396	6.336	1.97x
Spectrum (Phase Correction, FFT)	142+90	4.176	44	0.704	5.27x
TOTAL	1012	18.216	440	7.040	2.30x
Efficiency Score	4.11		10.65		-

Table 7.2: Execution times for high-frequency FTIR system on ML410 board and comparison to NASA JPL V2P board

The lowest execution time is obtained with a high-frequency system running at 266.67 MHz CPU and 133.33 MHz FPU. This 33.33% increase in CPU and FPU frequencies over an identical system also increases the speedup by nearly the same amount (from 7.95x to 10.37x). The memory infrastructure was not modified between builds (100 MHz PLB DDR2 controller with 200 MHz memory module), thus suggesting that the peak memory bandwidth has not yet been reached as otherwise the speedup would have been much less than what was observed.

Comparing the high-frequency FTIR system build on the ML410 development board to the V2P NASA JPL implementation shows an overall speedup of 2.30x (see Table 7.2). The results of the high-frequency FTIR build are first scaled to the appropriate number of interferograms and detectors to match what was used in the NASA JPL V2P research task. The reported speedup is seen even though the CPU on the ML410 system is clocked at a lower frequency than on the V2P. This also leads to a higher efficiency score on the ML410 system (the efficiency score is described in section 4.4 on page 38).

In Table 7.3 on the next page the performance of the high-frequency FTIR system is compared to the performance of the BAE RAD750 SBC evaluated at NASA JPL. Even with hardware and software optimizations, the FTIR system on the V4FX hybrid-FPGA still lags behind a software-only implementation on the RAD750; however, the margin is a lot smaller than anything seen previously. Overall, the FPGA system processes the data about 3.5x slower than the RAD750. This is mostly due to the time spent in the re-sampling phase of the FTIR spectrometry algorithm. The spectrum computation is

	RAD750 (BAE)	ML410 (Xilinx)	
PPC405 Freq.	133.33 MHz	266.67 MHz	
Interferograms	104	scaled to 104	
Detectors	2	scaled to 2	
SW Optimizations	**None**	**SP Math Functions IBM Perflib (DP)**	
HW Optimizations	**Native FPU**	**APU-FPU (133.33 MHz)**	
Software Component	**Time (min)**	**Time (min)**	**Speedup**
Re-sampling	69	396	0.17x
Spectrum (Phase Correction, FFT)	42+15	44	1.30x
TOTAL	126	440	0.29x

Table 7.3: Execution times for high-frequency FTIR system on ML410 board and comparison to BAE RAD750 SBC

actually faster on the FPGA than on the RAD750 (1.30x speedup). Additionally, the FPGA is only utilizing one of its two PPC405 cores. A dual-core implementation (investigate in Chapter 8 on page 82) will narrow the margin even further.

7.2 Power Estimation

Xilinx ISE software contains a power estimation tool called XPower. This tool, which is run after PAR, provides a rough estimate of the power consumption of the entire reconfigurable device. The data shown in Table 7.4 on the next page was collected with XPower and is presented next to estimate values for the NASA V2P and BAE RAD750 processing platforms.

The 200 MHz V4FX60 designs consume about 7.5 W of power, from which a constant 4.859 W is attributed to output power, mostly for DDR2[1]. This value is a very rough estimate as the type of DDR2 module connected to the board is not taken into consideration. The NASA JPL V2P board consumes 5 W of power, but no data is available on how much of that is attributed to its DDR memory. Thus, the power consumption value for the V4FX60 and the V2P can shift in either direction depending on the particular type of memory used. However, both the V4FX60 and the V2P consume far less power than the BAE RAD750 (20 W).

[1] An accurate power consumption figure for the 266 MHz V4FX60 design could not be obtained with XPower, which reported an enormous value of 15.972 W (clearly wrong). The value listed in the table is an educated guess based on previous system builds and the current CPU and FPU frequencies. The actual power consumption is most likely significantly less than this value.

System	**CPU (MHz)**	**FPU (MHz)**	**Dot-prod (MHz)**	**Power (W)**
Xilinx ML410 (V4FX60)	200	-	-	7.436
Xilinx ML410 (V4FX60)	200	100	-	7.521
Xilinx ML410 (V4FX60)	200	-	200	7.556
Xilinx ML410 (V4FX60)	200	100	200	7.613
Xilinx ML410 (V4FX60)	266	133	-	<10
NASA JPL V2P	300	-	-	5
BAE RAD750	133	present	-	20

Table 7.4: Estimated power consumption of V4FX60, V2P, and BAE RAD750

Chapter 8

Conclusions and Future Work

This thesis started with an all-FORTRAN implementation of the FTIR spectrometry algorithm, converted it to C code, and developed a number of HW/SW systems on the V4FX60 hybrid-FPGA. As part of the conversion task, a detailed process was developed for porting FORTRAN code to C using *f2c* and targeting the V4FX platform with or without hardware FPU support (see Appendix A on page 89). The execution time of the all software C implementation of the FTIR spectrometry algorithm was recorded and used for comparison as a "base case." Two software-based optimizations were applied that reduced the execution time by more than 4.5x. These included modifying the code to use all single-precision math library functions (non-ANSI) when dealing with single-precision data and utilizing the IBM Performance Libraries (*Perflib*) to improve the speed of all single and double-precision arithmetic. Detailed instructions for isolating double-precision optimization from *Perflib* were presented in Appendix B on page 92. A DP-only *Perflib* was later used in conjunction with the single-precision APU-FPU to further improve system performance.

The bulk of this thesis dealt with looking into hardware-based improvements to the FTIR spectrometry system. These included the Xilinx APU-FPU, and a single-precision dot-product co-processor. The APU-FPU delivered significant speedup for all single-precision floating-point operations. Its effectiveness was maximized when the system frequencies were increased. The dot-product co-processor, although ineffective in the FTIR spectrometry system due to poor spatial locality of the data, showed nearly a 2x improvement over the APU-FPU when working with smaller, sequentially accessed data sets. Furthermore, it was implemented as a load/store-based APU-connected FCM thus establishing a reference for creating similar APU co-processors. The design of a non-system-bused CPU-coupled co-processor is frequently overlooked in design guides, yet it is a very effective way to offload software routines to hardware implementation.

The ML410 development board, on which all of this work was conducted, hosts the

V4FX60 hybrid-FPGA containing two PPC405 processors. This thesis focused on optimizing the performance of the FTIR spectrometry algorithm on a single PPC405 core, however, the design can be extended to utilize both available cores. The block diagram in Figure 8.1 on the following page shows a dual-core design that can be implemented on the ML410 board. The two PPC405 processors each have dedicated PLB interfaces but share a common OPB. On the common OPB, the processors need to negotiate access to the RS232 UART and SystemACE CF controller. This negotiation can be done through dual-ported shared BRAM, accessible by each processor from their respective PLB. The ML410 board has two external memory interfaces that are both utilized in this concept. PPC405 CPU0 uses the DDR2 DIMM (256 Mbytes) while PPC405 CPU1 utilizes the DDR on-board component memory (64 Mbytes). Each of the processors has some dedicated on-chip memory connected through the OCM interface. The instruction side OCM is particularly necessary so each processor can store its own boot code in its own on-chip memory as booting both processors from external memory may not be possible. Both processors have their own FPUs connected on dedicated FCB interfaces. The selected CPU/FPU frequencies for the concept design are 200/100 MHz. The 266/133 MHz ratio was not chosen as with it there may be difficulties in meeting the timing constraints. Once the 200/100 MHz CPU/FPU dual-core system works, higher system frequencies should be investigated. Since the processing of individual interferograms is a completely independent task, an up to 2x reduction in execution time may be possible with a dual-core system. However, one bottleneck that may limit the speedup is negotiating access to the shared CF card controller.

Additional improvement to the overall performance of the FTIR spectrometry system may be possible by rewriting the software in C. The automatic conversion from FORTRAN to C using *f2c* most likely does not produce optimal code, and it is certainly not appealing to read. Some functions may also need to be rewritten with an optimized pattern of data access. This can help in cases such as the dot-product co-processor.

Further performance improvement may be achieved by trying a different compiler, one that is specifically targeted for the embedded PPC405 processor. A V2P performance study done at the NASA Goddard Space Flight Center concluded that using the WindRiver Diab DCC 5.2 compiler provides a 38% performance increase over the GNU-GCC 3.4 compiler. The comparison was based on running a Dhrystone benchmark application on a 400 MHz PPC405 design. The GNU-GCC compiler achieved 458 DMIPS while the WindRiver Diab DCC achieved 628 DMIPS (as reported by Xilinx) [39].

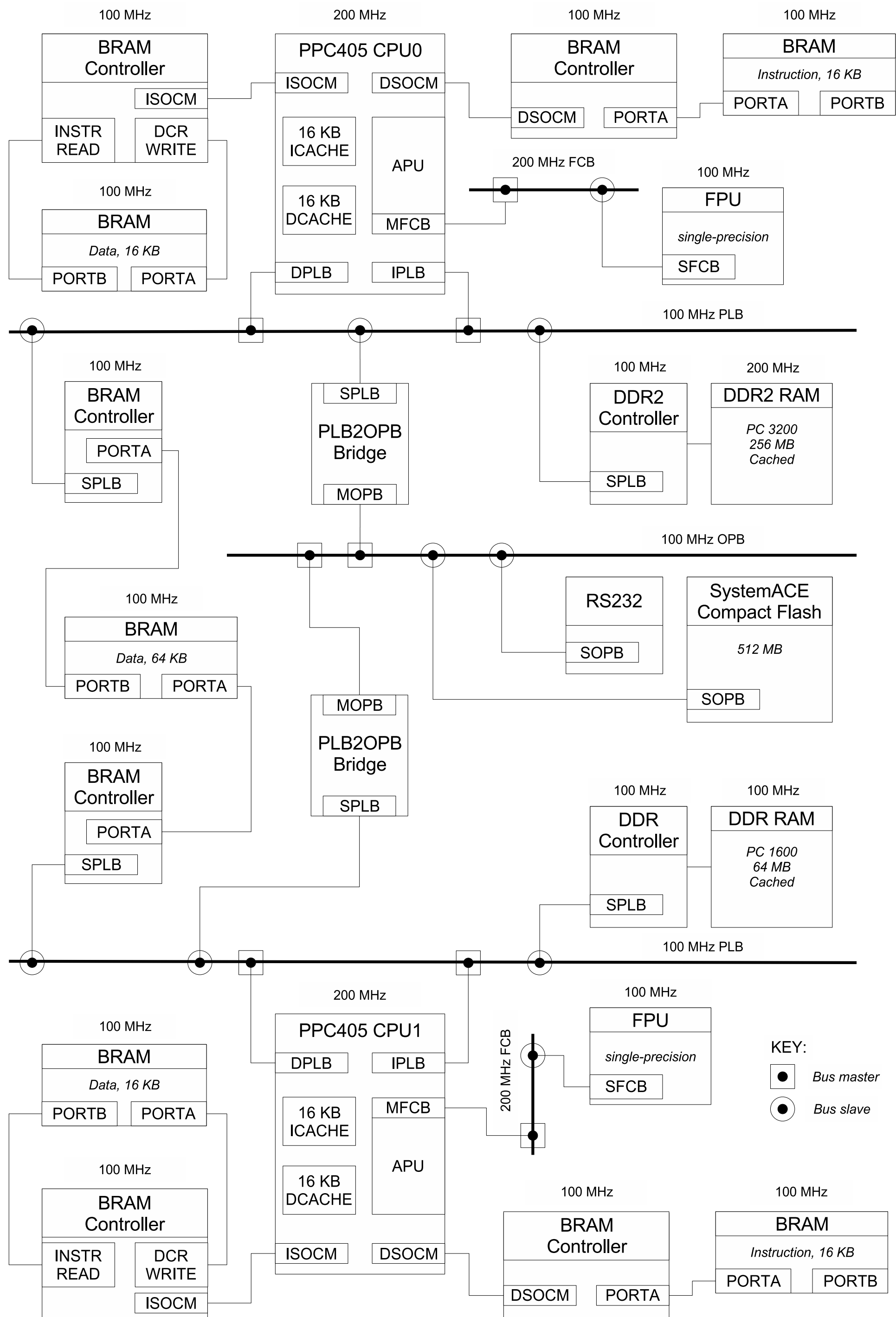


Figure 8.1: Dual-core concept targeting ML410 development board

Implementing additional hardware co-processors may result in the further reduction of execution time. Using the dot-product design as a reference, the FFT function, for example, can be implemented in the hardware. This will help in the spectrum computation component of the software processing. It may be necessary to re-arrange the data access pattern for optimal co-processor performance, to avoid the pitfall seen when deploying the dot-product core.

Finally, no embedded processing system is complete without an OS. Linux is a good choice and is supported by Xilinx in EDK. It is important to first finalize the hardware design prior to deploying the OS. Support for the APU may be lacking in Linux and getting the OS to recognize the hardware FPU may be a project in itself.

For the FTIR spectrometry algorithm, this thesis started the process of moving from an all software system to a mixed HW/SW implementation on the V4FX60 hybrid-FPGA. In the best case, a more than 10x speedup was achieved compared to the FTIR base system. This implementation, although over 2x faster the V2P system at NASA JPL, still lags behind the current state-of-the-art space processor - the BAE RAD750. However, the margin between the two was narrowed down significantly and with further research, as suggested above, will most likely be eliminated altogether.

Bibliography

[1] P. J. Pingree, J.-F. L. Blavier, G. C. Toon, and D. L. Bekker, "An FPGA/SoC Approach to On-Board Data Processing - Enabling New Mars Science with Smart Payloads," in *IEEE Aerospace Conference, 2007*, (Big Sky, MT), March 2007.

[2] G. Toon, J.-F. Blavier, M. McAuley, and A. Kiely, "Advanced On-Board Science Data Processing System for a Mars-orbiting FTIR Spectrometer," *R&TD Task 01STCR R.05.023.048*, NASA Jet Propulsion Laboratory, Pasadena, CA, 2005.

[3] G. Toon, P. Wennberg, M. Allen, M. Richardson, and J. Eiler, "Solar Occultation FTIR at Mars: A Space Odyssey for 2011," tech. rep., 2004.

[4] J. George, R. Koga, G. Swift, G. Allen, C. Carmichael, and C. W. Tseng, "Single Event Upsets in Xilinx Virtex-4 FPGA Devices," in *IEEE Radiation Effects Data Workshop, 2006*, (Ponte Vedra, FL), July 2006.

[5] G. Swift and G. Allen, "An Upset-Mitigated FPGA-based High Performance Compute Platform for Space Applications," in *MAPLD International Conference, 2006*, (Washington, D.C.), September 2006.

[6] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, "A Fortran-to-C Converter," Computing Science Technical Report 149, AT&T Bell Laboratories, Murray Hill, NJ, March 22 1995.

[7] "IBM PowerPC Embedded Processor Performance Libraries," tech. rep., IBM Microelectronics Division, Hopewell Junction, NY, December 12 2003.

[8] J. W. Brault, "New Approach to High-precision Fourier Transform Spectrometer Design," *Applied Optics*, vol. 35, pp. 2891–2896, June 1996.

[9] A. Burcin, "RAD750TM MRQW 2002," tech. rep., BAE Systems, Manassas, VA, 2002.

[10] "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet," Datasheet DS083, Xilinx, Inc., San Jose, CA, 2005.

[11] "Power Architecture Offerings," datasheet, IBM Systems and Technology Group, Hopewell Junction, NY, 2006.

[12] "PowerPC 750 RISC Microprocessor Technical Summary," datasheet, IBM Microelectronics Division, Hopewell Junction, NY, 1997.

[13] "Xilinx Virtex-4 Revolutionizes Platform FPGAs," white paper, Xilinx, Inc., San Jose, CA.

[14] "Low Power Consumption," tech. rep., Xilinx Inc., San Jose, CA, 2006.

[15] M. Shaaban, "Instroduction to Reconfigurable Computing," *Advanced Computer Architecture (EECC 722), lecture notes*, p. 2, Rochester Institute of Technology, Rochester, NY, Fall 2006.

[16] S. Guccione, "List of FPGA-based Computing Machines," tech. rep., 2000.

[17] J. R. Hauser and J. Wawrzynek, "GARP: A MIPS Processor with a Reconfigurable Coprocessor," in *The 5th Annual IEEE Symposium on FPGAs for Custom Computing Machines, 1997*, (Napa Valley, CA), pp. 12–21, April 1997.

[18] "Virtex-4 Family Overview," Datasheet DS112, Xilinx, Inc., San Jose, CA, January 23 2007.

[19] "Product Overview - PowerPC 405 CPU Core," datasheet, IBM Systems and Technology Group, Hopewell Junction, NY, September 2 2006.

[20] "MicroBlaze Performance," tech. rep., Xilinx Inc., San Jose, CA, 2006.

[21] *MicroBlaze Processor Reference Guide.* Xilinx, Inc., San Jose, CA, September 15 2006.

[22] "Processor Local Bus (PLB) v3.4," Datasheet DS400, Xilinx, Inc., San Jose, CA, July 7 2003.

[23] "On-chip Peripheral Bus (OPB) v2.0," Datasheet DS401, Xilinx, Inc., San Jose, CA, December 2 2005.

[24] "Device Control Register Bus (DCR) v2.9," Datasheet DS402, Xilinx, Inc., San Jose, CA, July 18 2005.

[25] *PowerPC 405 Processor Block Reference Guide.* Xilinx, Inc., San Jose, CA, July 20 2005.

[26] “Fabric Co-processor Bus (FCB) v1.00a,” Datasheet DS308, Xilinx, Inc., San Jose, CA, August 2 2005.

[27] “Fast Simplex Link (FSL) Bus v2.00a,” Datasheet DS449, Xilinx, Inc., San Jose, CA, December 1 2005.

[28] “Local Memory Bus (LMB) v1.00a,” Datasheet DS445, Xilinx, Inc., San Jose, CA, April 4 2005.

[29] V. Asokan, “Designing Multiprocessor Systems in Platform Studio,” White Paper WP262, Xilinx, Inc., San Jose, CA, April 27 2007.

[30] A. Ansari, P. Ryser, and D. Isaacs, “Accelerated System Performance with APU-Enhanced Processing,” *Xcell Journal*, pp. 36–39, 1st Quarter 2005.

[31] “Virtex-4 Data Sheet: DC and Switching Characteristics,” Datasheet DS302, Xilinx, Inc., San Jose, CA, March 27 2007.

[32] *ML410 Embedded Development Platform*. Xilinx, Inc., San Jose, CA, March 6 2007.

[33] “Software Optimization Techniques for PowerPC 405 and 440,” application note, IBM Microelectronics Division, Hopewell Junction, NY, 2003.

[34] “APU Floating-Point Unit v3.0,” product specification, Xilinx, Inc., San Jose, CA, January 26 2007.

[35] H. H. Ng and L. Pillai, “Accelerated System Performance with the APU Controller and XtremeDSP Slices,” Application Note XAPP717, Xilinx, Inc., San Jose, CA, September 29 2005.

[36] *Assembler Instructions with C Expression Operands*, ch. 5.35. GNU GCC 3.4.3 Manual, Free Software Foundation, Inc., Boston, MA, 2004.

[37] *Power ISA Version 2.04*. IBM Corp., Austin, TX, April 3 2007.

[38] *PowerPC Instruction Set Extension Guide*. Xilinx, Inc., San Jose, CA, April 28 2005.

[39] D. Petrick, “Analyzing the Xilinx Virtex-II Pro PowerPC with the Dhrystone Benchmark Applications,” tech. rep., NASA Goddard Space Flight Center, Greenbelt, Maryland.

Appendix A

Building *f2c* for EDK

The following is required to use *f2c* for EDK:

1. A program that converts FORTRAN source into C source:

   `http://www.netlib.org/f2c/mswin/f2c.exe.gz`

2. A library that must be linked with the generated C source:

   `http://www.netlib.org/f2c/libf2c.zip`

To set up *f2c*, start by launching the EDK shell from:

```
Start>Programs>Xilinx Platform Studio 9.1i>Accessories>Launch EDK Shell
```

This shells opens a Cygwin environment with many common Linux commands. Extract the *f2c* executable and place it in the Cygwin *usr/bin* directory (the actual location of this directory is *C:\EDK\cygwin\bin*):

```
$ ls
f2c.exe.gz libf2c.zip
$ gzip -d  f2c.exe.gz
$ ls
f2c.exe libf2c.zip
$ cp f2c.exe /usr/bin/f2c.exe
```

As a quick test to make sure everything is set up correctly, check the version of *f2c* (it is now in Xilinx's path and can be called from any directory in the EDK shell):

```
$ f2c --version
f2c (Fortran to C Translator) version 20060506.
```

Next extract the *f2c* library source:

```
$ unzip libf2c.zip
$ ls
f2c.exe  libf2c  libf2c.zip
```

In the *libf2c* directory, edit *makefile.u* with a common text editor like WordPad. Paste over the modified makefile provide in Appendix C.1 on page 94. Do not compile yet[1].

Start Xilinx Platform Studio and use the base system builder wizard to create a new PowerPC project for a Virtex-4FX FPGA. Specify a 200 MHz processor clock frequency and a 100 MHz bus clock frequency. Check the "enable floating point unit (FPU)" checkbox if a hardware FPU is desired. In the following screens, include a *RS232_Uart* and a 64 KB *PLB BRAM IF CNTLR*. Do not include the memory or peripheral test. Generate the system.

Next create a new software application project and mark it to initialize BRAMs (and do not initialize BRAMs for the ppc405_0_bootloop). From the *libf2c* directory add *arithchk.c* to the project. Specify the following compile options (notice the linking against the math library)[2]:

```
-mfpu=sp_full -DNO_FPINIT -lm
```

Build the whole hardware and software systems. Configure HyperTerminal with the proper baud rate to listen to the board output (the default baud rate is 9600). Download the bitstream to the board and watch for the output. It should be:

```
#define IEEE_MC68k
#define Arith_Kind_ASL 2
#define Double_Align
```

The define statements above are produced by the *arithchk.c* program which runs on the embedded PowerPC 405 processor in the FPGA. This program determines the specific arithmetic characteristics that properly represent the embedded processor.

Next copy the *ppc405_0\lib* directory inside the EDK project directory into the *libf2c* directory. The *ppc405_0\lib* directory includes processor-specific library files that are necessary for proper compilation of *libf2c*. These files are different from build to build depending on the system configuration. It is absolutely crucial that these files come from a system that matches the floating-point support options specified in the makefile. Now compile *libf2c* from the location of its makefile:

```
$ make -f makefile.u
[... compile messages ...]
```

[1]This document assumes the user wants full single precision floating-point support through the APU-FPU. If divide and square root operations support through the co-processor is not necessary, replace all instances of *-mfpu=sp_full* with *-mfpu=sp_lite*. Note that this requires updating the *CFLAGS* field in the makefile provided in Appendix C.1 on page 94. In this case, the divide and square root operations will be done through software emulation. If floating-point support through APU-FPU is not desired at all (for example, when compiling with *Perflib* option *-mppcperflib*), remove the *-mfpu=* option altogether.

[2]The *-mfpu=sp_full* and *-lm* options can be omitted as they are inserted automatically by EDK.

```
./a.out >arith.h
./a.out: 1: Syntax error: "(" unexpected
make: *** [arith.h] Error 2
```

On the first compilation, it is expected to receive the error seen above. The first compilation is necessary in order to generate certain header files which will be needed for the second compilation. Modify the newly generated *arith.h* file in the *libf2c* directory. Into it, paste the define statements produced by the development board in the previous step. Once again compile *libf2c*. This time, no errors should be generated.

At this point, *f2c* is now ready to be used with an EDK project provided that *f2c.h* and the compiled *libf2c.a* are in the project's include (*-I*) and library (*-L*) search paths, respectively. To convert FORTRAN source into C source, issue the following command in the EDK shell:

```
$ ls
hello.f
$ f2c hello.f
hello.f:
   MAIN:
$ ls
hello.c  hello.f
```

The generated C source can now be used in the EDK project. The following compile flags must be specified to properly link the f2c library, the math library, and utilize the APU-FPU hardware (if enabled):

```
-mfpu=sp_full -lf2c -lm
```

Appendix B

Recompiling IBM PowerPC *Perflib* for Double-precision Optimization Only

To recompile the IBM PowerPC *Perflib*, first obtain the source[1] (*ibmeppcperflib-1.1.tar.gz*) from the project homepage:

`http://sourceforge.net/projects/ppcperflib/`

Next launch the EDK shell from:

```
Start>Programs>Xilinx Platform Studio 9.1i>Accessories>Launch EDK Shell
```

This shell opens a Cygwin environment with many common Linux commands. Extract the downloaded source:

```
$ ls
ibmeppcperflib-1.1.tar
$ tar -xvf ibmeppcperflib-1.1.tar
$ ls
ibmeppcperflib-1.1.tar perflibs
$ cd perflibs
$ ls
COPYING.LIB  Makefile  doc  fpopt  include  stropt
```

Edit *Makefile* with a common text editor like WordPad. Paste over the modified makefile provided in Appendix C.2 on page 98. Also edit *Makefile* in the *fpopt* directory and paste over the modified makefile provided in Appendix C.3 on page 100.

Next browse to the *ppc405_0\lib* directory as described in Appendix A on page 89. Copy the contents of this directory into the *fpopt* directory under *perflibs*. Issue the *make* command from the *perflibs* directory to build *Perflib*.

Once the compilation is done, the *Perflib* double-precision only library can by used with an EDK project provided that *lib\libppcfp.a* is in the project's library (*-L*) search

[1] This project uses *Perflib* v1.1.

path. Link to the *Perflib* routines by specifying the *-lppcfp* option[2].

[2]This option may be used together with *-mfpu=sp_full* and *-mfpu=sp_lite* options, which are automatically inserted by EDK if an APU-FPU unit is present.

Appendix C

Select Code Listings

C.1 Makefile for *libf2c*[1]

```
1  # f2c makefile
2  # FOR USE WITH Xilinx EDK (targeting embedded PowerPC processor)
3  # >> with full APU-FPU support <<
4  #
5  # NOTE: for lite APU-FPU support, change -mfpu=sp_full to -mfpu=sp_lite
6  #       in CFLAGS; remove -mfpu= if no APU-FPU support is desired
7  #
8  # Modified by: Dmitriy Bekker
9  # Rochester Institute of Technology
10 # May 2007
11 # ==============================================================
12 #
13 # Unix makefile: see README.
14 # For C++, first "make hadd".
15 # If your compiler does not recognize ANSI C, add
16 #       -DKR_headers
17 # to the CFLAGS = line below.
18 # On Sun and other BSD systems that do not provide an ANSI sprintf, add
19 #       -DUSE_STRLEN
20 # to the CFLAGS = line below.
21 # On Linux systems, add
22 #       -DNON_UNIX_STDIO
23 # to the CFLAGS = line below.  For libf2c.so under Linux, also add
24 #       -fPIC
25 # to the CFLAGS = line below.
26
27 .SUFFIXES: .c .o
28 CC = powerpc-eabi-gcc
29 LD = powerpc-eabi-ld
30 AR = powerpc-eabi-ar
31 RNLIB = powerpc-eabi-ranlib
32 SHELL = /bin/sh
33 CFLAGS = -O3 -DNON_UNIX_STDIO -DNO_TRUNCATE -DNON_POSIX_STDIO -mfpu=sp_full
34 LPATH = -L./lib/
35 # compile, then strip unnecessary symbols
36 .c.o:
37         $(CC) -c -DSkip_f2c_Undefs $(CFLAGS) $(LPATH) $*.c
38         #$(LD) $(LPATH) -r -x -o $*.xxx $*.o
39         #mv $*.xxx $*.o
40
41 ## Under Solaris (and other systems that do not understand ld -x),
42 ## omit -x in the ld line above.
```

[1] **This is a modified version of the makefile provided in** `http://www.netlib.org/f2c/libf2c.zip`

```
43  ## If your system does not have the ld command, comment out
44  ## or remove both the ld and mv lines above.
45
46  MISC =  f77vers.o i77vers.o main.o s_rnge.o abort_.o exit_.o getarg_.o iargc_.o\
47          getenv_.o signal_.o s_stop.o s_paus.o system_.o cabs.o\
48          derf_.o derfc_.o erf_.o erfc_.o sig_die.o uninit.o
49  POW =   pow_ci.o pow_dd.o pow_di.o pow_hh.o pow_ii.o pow_ri.o pow_zi.o pow_zz.o
50  CX =    c_abs.o c_cos.o c_div.o c_exp.o c_log.o c_sin.o c_sqrt.o
51  DCX =   z_abs.o z_cos.o z_div.o z_exp.o z_log.o z_sin.o z_sqrt.o
52  REAL =  r_abs.o r_acos.o r_asin.o r_atan.o r_atn2.o r_cnjg.o r_cos.o\
53          r_cosh.o r_dim.o r_exp.o r_imag.o r_int.o\
54          r_lg10.o r_log.o r_mod.o r_nint.o r_sign.o\
55          r_sin.o r_sinh.o r_sqrt.o r_tan.o r_tanh.o
56  DBL =   d_abs.o d_acos.o d_asin.o d_atan.o d_atn2.o\
57          d_cnjg.o d_cos.o d_cosh.o d_dim.o d_exp.o\
58          d_imag.o d_int.o d_lg10.o d_log.o d_mod.o\
59          d_nint.o d_prod.o d_sign.o d_sin.o d_sinh.o\
60          d_sqrt.o d_tan.o d_tanh.o
61  INT =   i_abs.o i_dim.o i_dnnt.o i_indx.o i_len.o i_mod.o i_nint.o i_sign.o\
62          lbitbits.o lbitshft.o
63  HALF =  h_abs.o h_dim.o h_dnnt.o h_indx.o h_len.o h_mod.o h_nint.o h_sign.o
64  CMP =   l_ge.o l_gt.o l_le.o l_lt.o hl_ge.o hl_gt.o hl_le.o hl_lt.o
65  EFL =   ef1asc_.o ef1cmc_.o
66  CHAR =  f77_aloc.o s_cat.o s_cmp.o s_copy.o
67  I77 =   backspac.o close.o dfe.o dolio.o due.o endfile.o err.o\
68          fmt.o fmtlib.o ftell_.o iio.o ilnw.o inquire.o lread.o lwrite.o\
69          open.o rdfmt.o rewind.o rsfe.o rsli.o rsne.o sfe.o sue.o\
70          typesize.o uio.o util.o wref.o wrtfmt.o wsfe.o wsle.o wsne.o xwsne.o
71  QINT =  pow_qq.o qbitbits.o qbitshft.o ftell64_.o
72  TIME =  dtime_.o etime_.o
73
74  # If you get an error compiling dtime_.c or etime_.c, try adding
75  # -DUSE_CLOCK to the CFLAGS assignment above; if that does not work,
76  # omit $(TIME) from OFILES = assignment below.
77
78  # To get signed zeros in write statements on IEEE-arithmetic systems,
79  # add -DSIGNED_ZEROS to the CFLAGS assignment below and add signbit.o
80  # to the end of the OFILES = assignment below.
81
82  # For INTEGER*8 support (which requires system-dependent adjustments to
83  # f2c.h), add $(QINT) to the OFILES = assignment below...
84
85  OFILES = $(MISC) $(POW) $(CX) $(DCX) $(REAL) $(DBL) $(INT) \
86          $(HALF) $(CMP) $(EFL) $(CHAR) $(I77) $(TIME)
87
88  all: f2c.h signal1.h sysdep1.h libf2c.a
89
90  libf2c.a: $(OFILES)
91          $(AR) r libf2c.a $?
92          -$(RNLIB) libf2c.a
93
94  ## Shared-library variant: the following rule works on Linux
95  ## systems.  Details are system-dependent.  Under Linux, -fPIC
96  ## must appear in the CFLAGS assignment when making libf2c.so.
97  ## Under Solaris, use -Kpic in CFLAGS and use "ld -G" instead
98  ## of "cc -shared".
99
100 libf2c.so: $(OFILES)
101         $(CC) -shared -o libf2c.so $(OFILES)
102
103 ### If your system lacks ranlib, you don't need it; see README.
104
105 f77vers.o: f77vers.c
106         $(CC) -c f77vers.c
```

```
107
108 i77vers.o: i77vers.c
109         $(CC) -c i77vers.c
110
111 # To get an "f2c.h" for use with "f2c -C++", first "make hadd"
112 hadd: f2c.h0 f2ch.add
113         cat f2c.h0 f2ch.add >f2c.h
114
115 # For use with "f2c" and "f2c -A":
116 f2c.h: f2c.h0
117         cp f2c.h0 f2c.h
118
119 # You may need to adjust signal1.h and sysdep1.h suitably for your system...
120 signal1.h: signal1.h0
121         cp signal1.h0 signal1.h
122
123 sysdep1.h: sysdep1.h0
124         cp sysdep1.h0 sysdep1.h
125
126 # If your system lacks onexit() and you are not using an
127 # ANSI C compiler, then you should uncomment the following
128 # two lines (for compiling main.o):
129 #main.o: main.c
130 #       $(CC) -c -DNO_ONEXIT -DSkip_f2c_Undefs main.c
131 # On at least some Sun systems, it is more appropriate to
132 # uncomment the following two lines:
133 #main.o: main.c
134 #       $(CC) -c -Donexit=on_exit -DSkip_f2c_Undefs main.c
135
136 install: libf2c.a
137         cp libf2c.a $(LIBDIR)
138         -$(RNLIB) $(LIBDIR)/libf2c.a
139
140 clean:
141         rm -f libf2c.a *.o arith.h signal1.h sysdep1.h
142
143 backspac.o:     fio.h
144 close.o:        fio.h
145 dfe.o:          fio.h
146 dfe.o:          fmt.h
147 due.o:          fio.h
148 endfile.o:      fio.h rawio.h
149 err.o:          fio.h rawio.h
150 fmt.o:          fio.h
151 fmt.o:          fmt.h
152 iio.o:          fio.h
153 iio.o:          fmt.h
154 ilnw.o:         fio.h
155 ilnw.o:         lio.h
156 inquire.o:      fio.h
157 lread.o:        fio.h
158 lread.o:        fmt.h
159 lread.o:        lio.h
160 lread.o:        fp.h
161 lwrite.o:       fio.h
162 lwrite.o:       fmt.h
163 lwrite.o:       lio.h
164 open.o:         fio.h rawio.h
165 rdfmt.o:        fio.h
166 rdfmt.o:        fmt.h
167 rdfmt.o:        fp.h
168 rewind.o:       fio.h
169 rsfe.o:         fio.h
170 rsfe.o:         fmt.h
```

```
171 rsli.o:         fio.h
172 rsli.o:         lio.h
173 rsne.o:         fio.h
174 rsne.o:         lio.h
175 sfe.o:          fio.h
176 signbit.o:      arith.h
177 sue.o:          fio.h
178 uio.o:          fio.h
179 uninit.o:       arith.h
180 util.o:         fio.h
181 wref.o:         fio.h
182 wref.o:         fmt.h
183 wref.o:         fp.h
184 wrtfmt.o:       fio.h
185 wrtfmt.o:       fmt.h
186 wsfe.o:         fio.h
187 wsfe.o:         fmt.h
188 wsle.o:         fio.h
189 wsle.o:         fmt.h
190 wsle.o:         lio.h
191 wsne.o:         fio.h
192 wsne.o:         lio.h
193 xwsne.o:        fio.h
194 xwsne.o:        lio.h
195 xwsne.o:        fmt.h
196
197 arith.h: arithchk.c
198         $(CC) $(CFLAGS) $(LPATH) -DNO_FPINIT arithchk.c -lm ||\
199          $(CC) -DNO_LONG_LONG $(CFLAGS) $(LPATH) -DNO_FPINIT arithchk.c -lm
200         ./a.out >arith.h
201         rm -f a.out arithchk.o
202
203 check:
204         xsum Notice README abort_.c arithchk.c backspac.c c_abs.c c_cos.c \
205         c_div.c c_exp.c c_log.c c_sin.c c_sqrt.c cabs.c close.c comptry.bat \
206         d_abs.c d_acos.c d_asin.c d_atan.c d_atn2.c d_cnjg.c d_cos.c d_cosh.c \
207         d_dim.c d_exp.c d_imag.c d_int.c d_lg10.c d_log.c d_mod.c \
208         d_nint.c d_prod.c d_sign.c d_sin.c d_sinh.c d_sqrt.c d_tan.c \
209         d_tanh.c derf_.c derfc_.c dfe.c dolio.c dtime_.c due.c ef1asc_.c \
210         ef1cmc_.c endfile.c erf_.c erfc_.c err.c etime_.c exit_.c f2c.h0 \
211         f2ch.add f77_aloc.c f77vers.c fio.h fmt.c fmt.h fmtlib.c \
212         fp.h ftell_.c ftell64_.c \
213         getarg_.c getenv_.c h_abs.c h_dim.c h_dnnt.c h_indx.c h_len.c \
214         h_mod.c h_nint.c h_sign.c hl_ge.c hl_gt.c hl_le.c hl_lt.c \
215         i77vers.c i_abs.c i_dim.c i_dnnt.c i_indx.c i_len.c i_mod.c \
216         i_nint.c i_sign.c iargc_.c iio.c ilnw.c inquire.c l_ge.c l_gt.c \
217         l_le.c l_lt.c lbitbits.c lbitshft.c libf2c.lbc libf2c.sy lio.h \
218         lread.c lwrite.c main.c makefile.sy makefile.u makefile.vc \
219         makefile.wat math.hvc mkfile.plan9 open.c pow_ci.c pow_dd.c \
220         pow_di.c pow_hh.c pow_ii.c pow_qq.c pow_ri.c pow_zi.c pow_zz.c \
221         qbitbits.c qbitshft.c r_abs.c r_acos.c r_asin.c r_atan.c r_atn2.c \
222         r_cnjg.c r_cos.c r_cosh.c r_dim.c r_exp.c r_imag.c r_int.c r_lg10.c \
223         r_log.c r_mod.c r_nint.c r_sign.c r_sin.c r_sinh.c r_sqrt.c \
224         r_tan.c r_tanh.c rawio.h rdfmt.c rewind.c rsfe.c rsli.c rsne.c \
225         s_cat.c s_cmp.c s_copy.c s_paus.c s_rnge.c s_stop.c scomptry.bat sfe.c \
226         sig_die.c signal1.h0 signal_.c signbit.c sue.c sysdep1.h0 system_.c \
227         typesize.c \
228         uio.c uninit.c util.c wref.c wrtfmt.c wsfe.c wsle.c wsne.c xwsne.c \
229         z_abs.c z_cos.c z_div.c z_exp.c z_log.c z_sin.c z_sqrt.c >xsum1.out
230         cmp xsum0.out xsum1.out && mv xsum1.out xsum.out || diff xsum[01].out
```

C.2 Makefile (main) for Double-precision Only *Perflib*

```
# Perflib makefile (main)
# FOR USE WITH Xilinx EDK (targeting embedded PowerPC processor)
# >> double-precision optimization only <<
#
# Modified by: Dmitriy Bekker
# Rochester Institute of Technology
# April 2007
# ==============================================================
#
# makefile, pl_common, pl_linux 12/12/03 16:07:34
#-------------------------------------------------------------------------------
#
# Copyright (C) 2003 IBM Corporation
#   All rights reserved.
#
#   Redistribution and use in source and binary forms, with or
#   without modification, are permitted provided that the following
#   conditions are met:
#
#     * Redistributions of source code must retain the above
#       copyright notice, this list of conditions and the following
#       disclaimer.
#     * Redistributions in binary form must reproduce the above
#       copyright notice, this list of conditions and the following
#       disclaimer in the documentation and/or other materials
#       provided with the distribution.
#     * Neither the name of IBM nor the names of its contributors
#       may be used to endorse or promote products derived from this
#       software without specific prior written permission.
#
#   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
#   CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
#   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
#   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
#   DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
#   BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
#   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
#   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
#   PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
#   OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
#   (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
#   USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
#
#-------------------------------------------------------------------------------
#-------------------------------------------------------------------------------
# Make FP and string libraries
#-------------------------------------------------------------------------------
PREFIX        = powerpc-eabi-
MAKE          = make
FPDIR         = ./fpopt
LIBDIR        = ./lib
FPLIBA        = libppcfp.a
FPLIBSO       = libppcfp.so

libs:
        cd $(FPDIR);$(MAKE) PREFIX="$(PREFIX)"
        mkdir -p $(LIBDIR)
        cp -p $(FPDIR)/$(FPLIBA) $(LIBDIR)
        cp -p $(FPDIR)/$(FPLIBSO) $(LIBDIR)

```

```
61  clean:
62                  @cd $(FPDIR);$(MAKE) clean
63                  @rm $(LIBDIR)/*
64
65  clobber: clean
66
67  ###
```

C.3 Makefile (fpopt) for Double-precision Only *Perflib*

```
1  # Perflib makefile (fpopt)
2  # FOR USE WITH Xilinx EDK (targeting embedded PowerPC processor)
3  # >> double-precision optimization only <<
4  #
5  # Modified by: Dmitriy Bekker
6  # Rochester Institute of Technology
7  # April 2007
8  # ============================================================
9  #
10 #-----------------------------------------------------------------------------
11 #
12 #   Copyright (C) 2003 IBM Corporation
13 #   All rights reserved.
14 #
15 #   Redistribution and use in source and binary forms, with or
16 #   without modification, are permitted provided that the following
17 #   conditions are met:
18 #
19 #     * Redistributions of source code must retain the above
20 #       copyright notice, this list of conditions and the following
21 #       disclaimer.
22 #     * Redistributions in binary form must reproduce the above
23 #       copyright notice, this list of conditions and the following
24 #       disclaimer in the documentation and/or other materials
25 #       provided with the distribution.
26 #     * Neither the name of IBM nor the names of its contributors
27 #       may be used to endorse or promote products derived from this
28 #       software without specific prior written permission.
29 #
30 #   THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
31 #   CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES,
32 #   INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF
33 #   MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
34 #   DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS
35 #   BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY,
36 #   OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
37 #   PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
38 #   PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
39 #   OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
40 #   (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE
41 #   USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
42 #
43 #-----------------------------------------------------------------------------
44 # Options for GCC compiler
45 #-----------------------------------------------------------------------------
46 PREFIX ?=/opt/hardhat/devkit/ppc/405/bin/ppc_405-
47 CC =           $(PREFIX)gcc
48 CPP =          $(PREFIX)cpp
49 AS =           $(PREFIX)as
50 AR =           $(PREFIX)ar
51 LINKRW =       $(PREFIX)ld -Ttext=$(TEXT_ORG) -Tdata=$(DATA_ORG)
52 LD =           $(PREFIX)ld
53 INCLINK =$(PREFIX)ld -r
54 RANLIB =       $(PREFIX)ranlib
55
56 ASFLAGS =      -g
57 CPPFLAGS =     -I../include
58 CFLAGS  =      -g -msoft-float -static
59
60 # The following may be required with some cross-compilers to resolve startup symbols
```

```
61  # in the test programs.
62  #CFLAGS =       -g -msoft-float -static -mads
63
64  # The following sends a link map to standard out
65  #LDFLAGS                = -Wl,-M
66
67  # following flag settings are required to make a RiscWatch executable (testfloatr)
68  #ASFLAGS =      -gdwarf
69  #CFLAGS =       -gdwarf -msoft-float -static
70  #TEXT_ORG=      0x26000
71  #DATA_ORG=      0x36000
72
73  TFLOATG= testfloatg
74  TFLOATO= testfloato
75  TFLOATR= testfloatr
76  TARGET_LIB= libppcfp.a
77  TARGET_SO= libppcfp.so
78  LIBOBJS= ppc_fadd.o ppc_fsub.o ppc_fmul.o ppc_fdiv.o ppc_fcmpd.o
79
80  OBJTF = genfloat.o
81
82  all:            $(TARGET_LIB) $(TARGET_SO)
83  testprogs:              $(TFLOATG) $(TFLOATO)
84
85  # rule to make optimized floating point library for linking with static executables
86  $(TARGET_LIB):   $(LIBOBJS)
87                  $(AR) cr $(TARGET_LIB) $(LIBOBJS)
88                  $(RANLIB) $(TARGET_LIB)
89
90  # rule to make shared floating point library for linking with dynamic executables
91  $(TARGET_SO): $(LIBOBJS)
92                  $(CC) -shared -Wl,-soname,$(TARGET_SO).1 -o $(TARGET_SO).1.0  $(LIBOBJS
                        )
93                  ln -s $(TARGET_SO).1.0 $(TARGET_SO).1
94                  ln -s $(TARGET_SO).1 $(TARGET_SO)
95
96  # This executable links with the default GCC floating point operations
97  $(TFLOATG):     $(OBJTF) $(MAKEFILE)
98                  @echo "Loading $(TFLOATG) ..."
99                  $(CC) $(CFLAGS) $(LDFLAGS)  $(OBJTF) -o $(TFLOATG)
100                 @echo "Done ..."
101
102 # This executable links with the optimized floating point libraries
103 # Note:  libppcfp is included a second time in the link options below so that
104 #        any references to floating point operations that occur in libc are
105 #        resolved in libppcfp, rather than the default libgcc.
106 $(TFLOATO):     $(OBJTF) $(MAKEFILE) $(TARGET_LIB)
107                 @echo "Loading $(TFLOATO) ..."
108                 $(CC) $(CFLAGS) $(LDFLAGS) $(OBJTF) -o $(TFLOATO) \
109                         -L. -lppcfp -lc -lppcfp
110                 @echo "Done ..."
111
112 # This executable is for standalone testing under RiscWatch
113 # Note:  Prior to making this executable, it is necessary to uncomment the
114 #        RiscWatch definitions above and re-make the floating point library.
115 $(TFLOATR):     $(OBJTF) $(MAKEFILE) $(TARGET_LIB)
116                 @echo "Loading $(TFLOATR) ..."
117                 $(LINKRW) $(OBJTF)  -e main -o $(TFLOATR) \
118                         -L. -lppcfp -lc -lppcfp -lgcc
119                 @echo "Done ..."
120
121
122 clean:
123                 @rm -f *.o
```

```
124                     @rm -f core  *.a *.so*
125                     @rm -f $(TFLOATG) $(TFLOATO) $(TFLOATR) $(TARGET_LIB)
126
127     clobber: clean
128
129     ###
```

C.4 Original Top-level FTIR Spectrometry Source (*matmos-ipp.f*[2])

```
1        program matmos_ipp
2        implicit none
3   c
4   c  Data declarations
5   c
6        integer*4
7       & errnum,     ! Error code (0=ok, <0=fatal, >0=recoverable)
8       & inpstat,    ! Value of IOSTAT from input file read
9       & lun,        ! Logical Unit Number for file I/O
10      & mip,        ! Maximum number of input points or FFT size
11      & ndet,       ! Number of detectors
12      & jdet,       ! Loop variable for the current detector
13      & winfun,     ! Exponent of the COS windowing function
14      & nop,        ! Number of interpolation Operator Points
15      & nso,        ! Number of pre-computed Sub-Operators
16      & irun,       ! Scan number in the occultation
17      & counter,    ! Point count, data destination into buffers
18      & specount,   ! Spectrum Point count
19      & izpd,       ! Point index (location) of ZPD
20      & nphr,       ! Number of points on one side of ZPD used for phase
21      & indexa      ! Generic loop index
22       parameter (mip=2**21)
23       parameter (ndet=1)
24       parameter (lun=20)
25       parameter (winfun=8)
26       parameter (nop=56)
27       parameter (nso=8192)
28
29       character
30      & infile*(*)  ! Name of program input file
31       parameter (infile='ascii-tint07000.026')
32
33       logical*4
34      & filexist    ! Keeps track of file existence
35
36       real*4
37      & ylm(mip),      ! Input time-domain laser interferogram
38      & yir(mip,ndet), ! Input time-domain IR interferograms
39      & oper(nop*nso), ! Array holding the pre-computed operators
40      & ryir(mip,ndet) ! Path-difference domain IR interferograms
41
42       real*8
43      & dphase(mip)    ! Used internally by subroutine 't2x'
44  c
45  c  Initialize variables.
46  c
47       errnum=0
48       inpstat=0
49  c
50  c  Pre-compute interpolation operator
51  c
52       write(*,'(a)')'Pre-computing interpolation operator'
53       call pcoper(1.0,winfun,nop,nso,oper)
54  c
55  c  (Future) Loop over the scans that make up one occultation
56  c
```

[2] **This source was provided by the NASA Jet Propulsion Laboratory**

```
57        do irun=26,26
58  c
59  c  Check that input file exists, if so open it.
60  c
61          inquire(file=infile,exist=filexist,iostat=inpstat)
62          if(inpstat.ne.0) then
63            errnum=-1
64            write(*,'(2a)')'Error: inquire failed on input file ',infile
65          elseif(filexist) then
66            open(unit=lun,file=infile,status='old',iostat=inpstat)
67            if(inpstat.ne.0) then
68              errnum=-1
69              write(*,'(2a)')'Error: open failed on input file ',infile
70            else
71              write(*,'(2a)')'Reading time-domain interferogram: ',infile
72            endif
73          else
74            errnum=-1
75            write(*,'(2a)')'Error: please provide input file ',infile
76          endif
77  c
78  c  Read time-domain interferogram
79  c
80          counter=0
81          do while((inpstat.eq.0).and.              ! Not at end-of-file
82       &          (errnum.eq.0))                    ! And no errors
83            counter=counter+1
84            read(unit=lun,fmt=*,iostat=inpstat)
85       &      (ylm(counter),yir(counter,jdet),jdet=1,ndet)
86            if (inpstat.ne.0) then
87              counter=counter-1
88            endif
89          enddo      ! while((inpstat.eq.0).and.(errnum.eq.0))
90  c
91  c  Close the input file.
92  c
93          close(unit=lun,iostat=inpstat)
94          if(inpstat.ne.0) then
95            errnum=-1
96            write(*,'(2a)')'Error: close failed on input file ',infile
97          endif
98  c
99  c  Convert from time domain to path difference domain
100 c
101         if(errnum.eq.0) then
102           write(*,'(a)')'Converting to path-difference domain'
103           call t2f (nop,nso,oper,mip,ndet,ylm,yir,ryir,counter,dphase)
104         endif
105 c
106 c  Compute spectrum
107 c
108         if(errnum.eq.0) then
109           write(*,'(a)')'Computing spectrum'
110           do jdet=1,ndet
111             specount=counter
112             call ipplite(mip,22,0,1,ryir(1,jdet),specount,izpd,nphr)
113 c
114 c  Display a section of the spectrum
115 c
116             do indexa=(3*specount)/5,((3*specount)/5)+1000
117 c            do indexa=1,specount
118               write(*,'(SP1PE16.8E2)') ryir(indexa,jdet)
119             enddo
120           enddo
```

```
121           endif
122
123         end do  !  irun=26,26
124         stop
125         end
126
127         include 'pcoper.f'
128         include 't2x.f'
129         include 'ipp-lite.f'
```

C.5 No I/O Top-level FTIR Spectrometry Source (*matmos-ipp-chk-noio.f*[3])

```
1        program matmos_ipp
2        implicit none
3  c
4  c  Data declarations
5  c
6        integer*4
7       & errnum,     ! Error code (0=ok, <0=fatal, >0=recoverable)
8  !    & inpstat,    ! Value of IOSTAT from input file read
9  !    & lun,        ! Logical Unit Number for file I/O
10      & mip,        ! Maximum number of input points or FFT size
11      & ndet,       ! Number of detectors
12      & jdet,       ! Loop variable for the current detector
13      & winfun,     ! Exponent of the COS windowing function
14      & nop,        ! Number of interpolation Operator Points
15      & nso,        ! Number of pre-computed Sub-Operators
16      & irun,       ! Scan number in the occultation
17      & counter,    ! Point count, data destination into buffers
18      & specount,   ! Spectrum Point count
19      & izpd,       ! Point index (location) of ZPD
20      & nphr,       ! Number of points on one side of ZPD used for phase
21      & chkcnt,     ! Number of spectral points compared with reference result
22      & chkoff,     ! Point offset in spectrum of the reference result
23      & chkdev,     ! Index of max deviation between reference and calculated
24      & indexa      ! Generic loop index
25       parameter (mip=2**21)
26       parameter (ndet=1)
27 !     parameter (lun=20)
28       parameter (winfun=8)
29       parameter (nop=56)
30       parameter (nso=8192)
31       parameter (chkcnt=1000)
32
33 !     character
34 !    & infile*(*)  ! Name of program input file
35 !     parameter (infile='ascii-tint07000.026')
36
37 !     logical*4
38 !    & filexist    ! Keeps track of file existence
39
40       real*4
41      & ylm(mip),      ! Input time-domain laser interferogram
42      & yir(mip,ndet), ! Input time-domain IR interferograms
43      & oper(nop*nso), ! Array holding the pre-computed operators
44      & ryir(mip,ndet),! Path-difference domain IR interferograms
45      & chkspe(chkcnt),! Reference spectrum for comparison with calculated
46      & curdev,        ! Current deviation between reference and calculated
47      & maxdev         ! Maximum deviation between reference and calculated
48
49       real*8
50      & dphase(mip)    ! Used internally by subroutine 't2x'
51
52       include 'chkspe-data.inc'
53 c
54 c  Initialize variables.
55 c
56       errnum=0
```

[3] This is a modified version of the source provided by the NASA Jet Propulsion Laboratory

```
57  !     inpstat=0
58  c
59  c  Pre-compute interpolation operator
60  c
61        write(*,'(a)')'Pre-computing interpolation operator'
62        call pcoper(1.0,winfun,nop,nso,oper)
63  c
64  c  (Future) Loop over the scans that make up one occultation
65  c
66        do irun=26,26
67  c
68  c  Check that input file exists, if so open it.
69  c
70  !       inquire(file=infile,exist=filexist,iostat=inpstat)
71  !       if(inpstat.ne.0) then
72  !         errnum=-1
73  !         write(*,'(2a)')'Error: inquire failed on input file ',infile
74  !       elseif(filexist) then
75  !         open(unit=lun,file=infile,status='old',iostat=inpstat)
76  !         if(inpstat.ne.0) then
77  !           errnum=-1
78  !           write(*,'(2a)')'Error: open failed on input file ',infile
79  !         else
80  !           write(*,'(2a)')'Reading time-domain interferogram: ',infile
81  !         endif
82  !       else
83  !         errnum=-1
84  !         write(*,'(2a)')'Error: please provide input file ',infile
85  !       endif
86  c
87  c  Read time-domain interferogram
88  c
89          counter=0
90  !       do while((inpstat.eq.0).and.              ! Not at end-of-file
91  !    &           (errnum.eq.0))                   ! And no errors
92  !         counter=counter+1
93  !         read(unit=lun,fmt=*,iostat=inpstat)
94  !    &      (ylm(counter),yir(counter,jdet),jdet=1,ndet)
95  !         if (inpstat.ne.0) then
96  !           counter=counter-1
97  !         endif
98  !       enddo      ! while((inpstat.eq.0).and.(errnum.eq.0))
99  c
100 c  Close the input file.
101 c
102 !       close(unit=lun,iostat=inpstat)
103 !       if(inpstat.ne.0) then
104 !         errnum=-1
105 !         write(*,'(2a)')'Error: close failed on input file ',infile
106 !       endif
107 c
108 c  Convert from time domain to path difference domain
109 c
110         if(errnum.eq.0) then
111           write(*,'(a)')'Converting to path-difference domain'
112           call t2f (nop,nso,oper,mip,ndet,ylm,yir,ryir,counter,dphase)
113         endif
114 c
115 c  Compute spectrum
116 c
117         if(errnum.eq.0) then
118           write(*,'(a)')'Computing spectrum'
119           do jdet=1,ndet
120             specount=counter
```

```
121              call ipplite(mip,22,0,1,ryir(1,jdet),specount,izpd,nphr)
122   c
123   c  Search for max deviation between reference and calculated
124   c
125              maxdev=0.0
126              chkdev=1
127              chkoff=((3*specount)/5)-1
128              do indexa=1,chkcnt
129                curdev=abs(chkspe(indexa)-ryir(chkoff+indexa,jdet))
130                if (curdev.gt.maxdev) then
131                  maxdev=curdev
132                  chkdev=indexa
133                endif
134              enddo
135   !           write (*,*)'Maximum deviation of ',maxdev/chkspe(chkcnt),
136   !     &       ' at ',chkdev
137            enddo
138          endif
139
140        end do  !  irun=26,26
141        stop
142        end
143
144        include 'pcoper.f'
145        include 't2x.f'
146        include 'ipp-lite.f'
```

C.6 Partial FTIR Spectrometry C-source (*xilinx-matmos-ipp-chk_orig.c*[4])

```
/* xilinx-matmos-ipp-chk.f -- translated by f2c (version 20060506).
   You must link the resulting object file with libf2c:
        on Microsoft Windows system, link with libf2c.lib;
        on Linux or Unix systems, link with .../path/to/libf2c.a -lm
        or, if you install libf2c.a in a standard place, with -lf2c -lm
        -- in that order, at the end of the command line, as in
                cc *.o -lf2c -lm
        Source for libf2c is in /netlib/f2c/libf2c.zip, e.g.,

                http://www.netlib.org/f2c/libf2c.zip
*/

/****************************************************************************/
/* standard includes */
#include <stdio.h>         /* for standard i/o, xil_printf */
#include <stdlib.h>        /* common functions, atof */
#include <math.h>          /* for math functions */

/* local includes */
#include "xparameters.h"   /* system specific parameters and addresses */
#include "xcache_l.h"      /* cacheable memory enable/disable */
#include "sysace_stdio.h"  /* sysACE i/o */
#include "xsysace_l.h"     /* sysACE parameters and addresses */
#include "qxFpu_utils.h"   /* for qxFpu_printFloat */
#include "f2c.h"           /* for f2c functions */

/* include timing routines only if profiling is off */
#ifndef PROFILING
#include "xtime_l.h"       /* for timing */
#endif

/* define cacheable memory regions
 * each bit in the regions variable stands for 128MB of memory:
 *      regions    --> cached address range
 *      ------------|--------------------------------------------------
 *      0x80000000  | [0, 0x7FFFFFF]
 *      0x00000001  | [0xF8000000, 0xFFFFFFFF]
 *      0x80000001  | [0, 0x7FFFFFF],[0xF8000000, 0xFFFFFFFF]
*/
#define INSTR_CACHE 0x00000003
#define DATA_CACHE 0x00000003

/* for timing calculations */
#define CYCLES_PER_SEC XPAR_CPU_PPC405_CORE_CLOCK_FREQ_HZ

/* for sysACE setup */
#define SYSACE_BASEADDR XPAR_SYSACE_COMPACTFLASH_BASEADDR

/* for ASCII read/write functions */
#define READLENGTH 29      /* number of characters in a line + 1 terminator */
#define FLOATLENGTH 14     /* number of characters for one float, exp, signs */
#define WRITELENGTH 18     /* number of characters in a line + 1 terminator */

/* redefine the qxFpu_printFloat function */
#define printfloat(a,b,c) qxFpu_printFloat(a,b,c)

```

[4] **Only the initialization, global variables, MAIN__ and file I/O function are shown**

```
57  /* ASCII read/write functions */
58  int read_data( char[], real *, real * );
59  int write_data( char[], real *, int );
60  /*************************************************************************/
61
62  /* Common Block Declarations */
63
64  struct {
65      real pii, p7, p7two, c22, s22, pi2;
66  } con_;
67
68  #define con_1 con_
69
70  struct {
71      real pii, p7, p7two, c22, s22, pi2;
72  } con1_;
73
74  #define con1_1 con1_
75
76  /* Table of constant values */
77
78  static integer c__1 = 1;
79  static real c_b4 = 1.f;
80  static integer c__8 = 8;
81  static integer c__56 = 56;
82  static integer c__8192 = 8192;
83  static integer c_b13 = 2097152;
84  static integer c__22 = 22;
85  static integer c__0 = 0;
86  static integer c__5 = 5;
87  static integer c__3 = 3;
88  static integer c_n1 = -1;
89  static integer c__1024 = 1024;
90  static integer c__15 = 15;
91  static integer c__512 = 512;
92  static integer c__2 = 2;
93  static real c_b71 = .125f;
94  static real c_b96 = 0.f;
95
96  /* Main program */ int MAIN__(void)
97  {
98      /* Initialized data */
99
100     static real chkspe[1000] = { 40913887200.f,40869351400.f,40946221100.f,
101             41011777500.f,40966717400.f,40894693400.f,40911945700.f,
102             4.0980865e10f,40983662600.f,40918589400.f,40896036900.f,
103             4.0952533e10f,40992866300.f,4.0951124e10f,40900083700.f,
104             40928673800.f,40990851100.f,40979410900.f,40907661300.f,


            /* MORE DATA POINTS HERE (TOTAL = 1000) */


346             40986394600.f,40983822300.f,40980279300.f,40981770200.f,
347             4.0985858e10f,40985747500.f,4.0981631e10f,40980332500.f,
348             40984035300.f,40986681300.f,40984100900.f,40980774900.f,
349             40982343700.f,40986218500.f,40985968600.f,40982106100.f,
350             40981151700.f };
351
352     /* System generated locals */
353     real r__1;
354
```

```
355     /* Builtin functions */
356     integer s_wsfe(cilist *), do_fio(integer *, char *, ftnlen), e_wsfe(void);
357     /* Subroutine */ int s_stop(char *, ftnlen);
358 
359     /* Local variables */
360     static integer specount;
361     extern /* Subroutine */ int t2f_(integer *, integer *, real *, integer *,
362             integer *, real *, real *, real *, integer *, doublereal *);
363     static real ylm[2097152], yir[2097152]      /* was [2097152][1] */;
364     static integer jdet;
365     static real oper[458752];
366     static integer izpd, nphr, irun;
367     static real ryir[2097152]   /* was [2097152][1] */;
368     static integer chkoff, chkdev;
369     static doublereal dphase[2097152];
370     static integer indexa;
371     static real spechk[1000], maxdev, curdev;
372     extern /* Subroutine */ int pcoper_(real *, integer *, integer *, integer
373             *, real *);
374     static integer errnum;
375     extern /* Subroutine */ int ipplite_(integer *, integer *, integer *,
376             integer *, real *, integer *, integer *, integer *);
377     static integer counter;
378 
379     /* Fortran I/O blocks */
380     static cilist io___3 = { 0, 6, 0, "(a)", 0 };
381     static cilist io___7 = { 0, 6, 0, "(a)", 0 };
382     static cilist io___12 = { 0, 6, 0, "(a)", 0 };
383 
384 /***************************************************************************/
385    #ifndef PROFILING
386    XTime timer, last;           /* timing variables */
387    #endif
388 
389    /* Enable Caches */
390    XCache_EnableICache(INSTR_CACHE);
391    XCache_EnableDCache(DATA_CACHE);
392 
393    xil_printf("\n\r*** STARTING MATMOS-IPP ***\n\n\r");
394 
395    /* Reset the sysace controller to clean any bad state, leave it in MPU mode */
396    XSysAce_RegWrite16(SYSACE_BASEADDR + XSA_BMR_OFFSET, XSA_BMR_16BIT_MASK);
397    XSysAce_mSetControlReg(SYSACE_BASEADDR, XSA_CR_CFGSEL_MASK |
        XSA_CR_FORCECFGMODE_MASK);
398    XSysAce_mSetControlReg(SYSACE_BASEADDR, XSA_CR_CFGSEL_MASK |
        XSA_CR_FORCECFGMODE_MASK | XSA_CR_CFGRESET_MASK);
399    XSysAce_mSetControlReg(SYSACE_BASEADDR, XSA_CR_CFGSEL_MASK |
        XSA_CR_FORCECFGMODE_MASK);
400 /***************************************************************************/
401 
402 /*  Data declarations */
403 
404 /*    & inpstat,    ! Value of IOSTAT from input file read */
405 /*    & lun,        ! Logical Unit Number for file I/O */
406 /* Error code (0=ok, <0=fatal, >0=recoverable) */
407 /* Maximum number of input points or FFT size */
408 /* Number of detectors */
409 /* Loop variable for the current detector */
410 /* Exponent of the COS windowing function */
411 /* Number of interpolation Operator Points */
412 /* Number of pre-computed Sub-Operators */
413 /* Scan number in the occultation */
414 /* Point count, data destination into buffers */
415 /* Spectrum Point count */
```

```
416  /* Point index (location) of ZPD */
417  /* Number of points on one side of ZPD used for phase */
418  /* Number of spectral points compared with reference r */
419  /* Point offset in spectrum of the reference result */
420  /* Index of max deviation between reference and calcul */
421  /* Generic loop index */
422  /*     parameter (lun=20) */
423  /*     character */
424  /*    & infile*(*)  ! Name of program input file */
425  /*     parameter (infile='ascii-tint07000.026') */
426  /*     logical*4 */
427  /*    & filexist    ! Keeps track of file existence */
428  /* Input time-domain laser interferogram */
429  /* Input time-domain IR interferograms */
430  /* Array holding the pre-computed operators */
431  /* Path-difference domain IR interferograms */
432  /* Reference spectrum for comparison with calculate */
433  /* Calculated spectrum */
434  /* Current deviation between reference and calculat */
435  /* Maximum deviation between reference and calculat */
436  /* Used internally by subroutine 't2x' */
437
438  /*  Initialize variables. */
439
440      errnum = 0;
441  /*     inpstat=0 */
442
443  /*  Pre-compute interpolation operator */
444
445      s_wsfe(&io___3);
446      do_fio(&c__1, "Pre-computing interpolation operator", (ftnlen)36);
447      e_wsfe();
448      pcoper_(&c_b4, &c__8, &c__56, &c__8192, oper);
449
450  /*  (Future) Loop over the scans that make up one occultation */
451
452      for (irun = 26; irun <= 26; ++irun) {
453
454  /*  Check that input file exists, if so open it. */
455
456  /*       inquire(file=infile,exist=filexist,iostat=inpstat) */
457  /*       if(inpstat.ne.0) then */
458  /*         errnum=-1 */
459  /*         write(*,'(2a)')'Error: inquire failed on input file ',infile */
460  /*       elseif(filexist) then */
461  /*         open(unit=lun,file=infile,status='old',iostat=inpstat) */
462  /*         if(inpstat.ne.0) then */
463  /*           errnum=-1 */
464  /*           write(*,'(2a)')'Error: open failed on input file ',infile */
465  /*         else */
466  /*           write(*,'(2a)')'Reading time-domain interferogram: ',infile */
467  /*         endif */
468  /*       else */
469  /*         errnum=-1 */
470  /*         write(*,'(2a)')'Error: please provide input file ',infile */
471  /*       endif */
472
473  /*  Read time-domain interferogram */
474
475          counter = 0;
476  /*       do while((inpstat.eq.0).and.               ! Not at end-of-file */
477  /*    &           (errnum.eq.0))                    ! And no errors */
478  /*         counter=counter+1 */
479  /*         read(unit=lun,fmt=*,iostat=inpstat) */
```

```
480 /*    &       (ylm(counter),yir(counter,jdet),jdet=1,ndet) */
481 /*         if (inpstat.ne.0) then */
482 /*           counter=counter-1 */
483 /*         endif */
484 /*       enddo      ! while((inpstat.eq.0).and.(errnum.eq.0)) */
485
486 /*  Close the input file. */
487
488 /*       close(unit=lun,iostat=inpstat) */
489 /*       if(inpstat.ne.0) then */
490 /*         errnum=-1 */
491 /*         write(*,'(2a)')'Error: close failed on input file ',infile */
492 /*       endif */
493
494 /*****************************************************************************/
495    /* Initialize timer */
496    #ifndef PROFILING
497    XTime_SetTime(0);
498
499    /* Read data from file */
500    XTime_GetTime(&timer);
501    #endif
502
503    counter = read_data( "a:\\data.txt", ylm, yir );
504
505    #ifndef PROFILING
506    XTime_GetTime(&last);
507    #endif
508    xil_printf( "### %d points read: ", counter+1 );
509    #ifndef PROFILING
510    printfloat( ((float)(last-timer)) / CYCLES_PER_SEC, 4, " sec\n\r" );
511    #endif
512 /*****************************************************************************/
513
514
515 /*  Convert from time domain to path difference domain */
516
517         if (errnum == 0) {
518             s_wsfe(&io___7);
519             do_fio(&c__1, "Converting to path-difference domain", (ftnlen)36);
520             e_wsfe();
521
522 /*****************************************************************************/
523       #ifndef PROFILING
524       XTime_GetTime(&timer);
525       #endif
526 /*****************************************************************************/
527
528             t2f_(&c__56, &c__8192, oper, &c_b13, &c__1, ylm, yir, ryir, &
529                     counter, dphase);
530
531 /*****************************************************************************/
532       #ifndef PROFILING
533       XTime_GetTime(&last);
534       xil_printf( "### TIME: " );
535       printfloat( ((float)(last-timer)) / CYCLES_PER_SEC, 4, " sec\n\r" );
536       #endif
537 /*****************************************************************************/
538
539         }
540
541 /*  Compute spectrum */
542
543         if (errnum == 0) {
```

```
544             s_wsfe(&io___12);
545             do_fio(&c__1, "Computing spectrum", (ftnlen)18);
546             e_wsfe();
547             for (jdet = 1; jdet <= 1; ++jdet) {
548                 specount = counter;
549
550 /*****************************************************************************/
551       #ifndef PROFILING
552       XTime_GetTime(&timer);
553       #endif
554 /*****************************************************************************/
555
556                 ipplite_(&c_b13, &c__22, &c__0, &c__1, &ryir[(jdet << 21) -
557                         2097152], &specount, &izpd, &nphr);
558
559 /*****************************************************************************/
560       #ifndef PROFILING
561       XTime_GetTime(&last);
562       xil_printf( "### TIME: " );
563       printfloat( ((float)(last-timer)) / CYCLES_PER_SEC, 4, " sec\n\r" );
564       #endif
565
566       xil_printf( "### COMPUTATION DONE!\n\r" );
567 /*****************************************************************************/
568
569 /*  Search for max deviation between reference and calculated */
570
571                 maxdev = 0.f;
572                 chkdev = 1;
573                 chkoff = specount * 3 / 5 - 1;
574                 for (indexa = 1; indexa <= 1000; ++indexa) {
575                     spechk[indexa - 1] = ryir[chkoff + indexa + (jdet << 21)
576                             - 2097153];
577                     curdev = (r__1 = chkspe[indexa - 1] - spechk[indexa - 1],
578                             abs(r__1));
579                     if (curdev > maxdev) {
580                         maxdev = curdev;
581                         chkdev = indexa;
582                     }
583                 }
584 /*            write (*,*)'Maximum deviation of ',maxdev/chkspe(chkcnt), */
585 /*     &       ' at ',chkdev */
586
587 /*****************************************************************************/
588                 r__1 = maxdev / chkspe[999];
589                 xil_printf( "Maximum deviation of " );
590                 printfloat( r__1, 9, " " );
591                 xil_printf( "at %d\n\r", chkdev );
592 /*****************************************************************************/
593
594             }
595         }
596     }
597
598 /*****************************************************************************/
599    #ifdef SPEDUMP
600    /* Write spectrum to file */
601    #ifndef PROFILING
602    XTime_GetTime(&timer);
603    #endif
604
605    #ifdef DUMPDATA
606    counter = write_data( "a:\\datdump.txt", ryir, counter );
607    #else
```

```
608    counter = write_data( "a:\\spedump.txt", spechk, 999 );
609    #endif
610
611    #ifndef PROFILING
612    XTime_GetTime(&last);
613    #endif
614    xil_printf( "### %d points written: ", counter+1 );
615    #ifndef PROFILING
616    printfloat( ((float)(last-timer)) / CYCLES_PER_SEC, 4, " sec\n\r" );
617    #endif
618    #endif
619
620    #ifndef PROFILING
621    XTime_GetTime(&timer);
622    xil_printf( "\n\r### Total time: " );
623    printfloat( ((float)timer) / CYCLES_PER_SEC, 4, " sec\n\r" );
624    #endif
625
626    /* Disable cache */
627    XCache_DisableDCache();
628    XCache_DisableICache();
629 /*****************************************************************************/
630
631 /*  irun=26,26 */
632     s_stop("", (ftnlen)0);
633     return 0;
634 } /* MAIN__ */
635
636
637 /*****************************************************************************/
638 int read_data( char FileName[], real *data1, real *data2 ) {
639
640         SYSACE_FILE *ptest;
641         int count = 0;
642         int total_bytes_read = 0;
643         int numread;
644         char val[READLENGTH];
645         ptest = sysace_fopen(FileName , "r" );
646
647         if(ptest) {
648                 xil_printf("Reading file: %s\n\r", FileName);
649
650                 /* Read a line of characters */
651                 numread = sysace_fread(val, 1, sizeof( char ) * READLENGTH, ptest);
652                 total_bytes_read = total_bytes_read + numread;
653                 while( numread ) {
654
655                         if( (total_bytes_read % 1024) == 0 ) {
656                                 xil_printf( "%d KB read\r", total_bytes_read/1024 );
657                         }
658
659                         /* Extract the two floats */
660                         data1[count] = atof(val);
661                         data2[count] = atof(val+FLOATLENGTH);
662                         count++;
663
664                         /* Read a line of characters */
665                         numread = sysace_fread(val, 1, sizeof( char ) * READLENGTH,
                            ptest);
666                         total_bytes_read = total_bytes_read + numread;
667
668                 }
669
670                 count--;
```

```
671                 sysace_fclose (ptest);
672         }
673         else {
674                 xil_printf("Failed to open: %s\n\r", FileName);
675         }
676
677         xil_printf("==> %d bytes read\n\r", total_bytes_read);
678         return count;
679 } /* read_data */
680
681 int write_data( char FileName[], real *data, int N ) {
682
683    SYSACE_FILE *ptest;
684         int count = 0;
685    int total_bytes_write = 0;
686         int numwrite;
687         char val[WRITELENGTH];
688    ptest = sysace_fopen(FileName ,"w" );
689
690    if( ptest ) {
691
692                 xil_printf("Writing to file: %s\n\r", FileName);
693
694                 while( count <= N ) {
695
696                         /* Write a line of characters */
697                         sprintf(val, "  %+.8E\n", data[count] );
698                         numwrite = sysace_fwrite(val, 1, sizeof( char ) * WRITELENGTH,
                                ptest);
699                         total_bytes_write= total_bytes_write + numwrite;
700
701                         if( (total_bytes_write % 1024) == 0 ) {
702                                 xil_printf( "%d KB written\r", total_bytes_write/1024 )
                                     ;
703                         }
704
705                         count++;
706                 }
707
708                 count--;
709       sysace_fclose(ptest);
710    }
711         else {
712                 xil_printf( "Failed to open: %s\n\r", FileName );
713    }
714
715         xil_printf("==> %d bytes written\n\r", total_bytes_write);
716         return count;
717 } /* write_data */
```

C.7 The *dotprod* Function (from *xilinx-matmos-ipp-chk_orig.c*)

```
1160 real dotprod_(real *fin, integer *nin, real *oper, integer *nop, integer *nso,
1161          doublereal *xx)
1162 {
1163     /* System generated locals */
1164     integer oper_dim1, oper_offset, i__1;
1165     real ret_val;
1166
1167     /* Local variables */
1168     static integer i__, j, jr, kin;
1169
1170 /*     FIN(NIN)       R*4  Input function vector */
1171 /*     NIN            I*4  Number of points in input vector */
1172 /*     OPER(NOP,ODEC) R*4  Resampling Operators */
1173 /*     NOP            I*4  Number of operator points (Length of each sub-operator) */
1174 /*     NSO            I*4  Number of sub-operators */
1175 /*     XX             R*8  Value at which to sample FIN */
1176
1177 /*  Outputs: */
1178 /*     DOTPROD        R*4 */
1179
1180 /*  Evaluating the scalar product of the appropriate sub-operator */
1181 /*  of OPER with the relevant section of FIN. */
1182
1183 /*  Note that the full array oper(nso,nop) is rotationally symmetric about the */
1184 /*  center such that oper(i,j) = oper(nop+1-i,nso+1-j) */
1185 /*  This means that we only have to precompute half the full aray */
1186 /*  We could use just the left half (i=1,nop/2) or just the top half (j=1,nso/2) */
1187 /*  We choose the former and note that since ir=nop+1-i, and jr=nso+1-j */
1188 /*     oper(i,j) = oper(nop+1-i,nso+1-j) = oper(ir,jr) */
1189
1190     /* Parameter adjustments */
1191     --fin;
1192     oper_dim1 = (*nop + 1) / 2;
1193     oper_offset = 1 + oper_dim1;
1194     oper -= oper_offset;
1195
1196     /* Function Body */
1197     kin = (integer) (*xx);
1198     jr = (integer) ((*nso - 1) * (*xx - kin)) + 2;
1199     j = *nso + 1 - jr;
1200     ret_val = 0.f;
1201 /*  Old dot product code was very simple */
1202 /*      do i=1,nop  ! Dot product fin(kin+1) with oper(1,j) */
1203 /*         dotprod=dotprod+fin(i+kin)*oper(i,j) */
1204 /*      end do */
1205
1206 /*  New dot product code. */
1207 /*  Starting from the two ends of the operator and works toward the middle. */
1208 /*  This should provide higher precision if the largest values reside in */
1209 /*  the middle of the operator, which they typically do. */
1210     i__1 = *nop / 2;
1211     for (i__ = 1; i__ <= i__1; ++i__) {
1212 /* Dot product fin(kin+1) with oper(1,j) */
1213         ret_val = ret_val + fin[i__ + kin] * oper[i__ + j * oper_dim1] + fin[*
1214                 nop + 1 - i__ + kin] * oper[i__ + jr * oper_dim1];
1215     }
1216     if (*nop % 2 == 1) {
1217         ret_val += fin[i__ + kin] * oper[i__ + j * oper_dim1];
1218     }
1219 /* nop is */
```

```
1220     return ret_val;
1221 } /* dotprod_ */
```

C.8 The *dotprod* Function with HW Support (*dotprod.c*)

```
1  /***************************************************************************
2  * Modified version of dotprod_ function
3  * Compiled separetly into a static library. Compile to assembly first and then
4  * hand tune.
5  *
6  * Modified by Dmitriy Bekker
7  * Rochester Institute of Technology, June 2007
8  *
9  * Compile commands (example, from EDK project dir):
10 * $ powerpc-eabi-gcc -O2 -S -c matmos-ipp-sp-dotprod-perflib_lg/src/dotprod.c \
11 *    -I./ppc405_0/include/ -Imatmos-ipp-sp-dotprod-perflib_lg/src/ \
12 *    -I./shared_libs/ -L./ppc405_0/lib/ -L./shared_libs/ -Wall
13 *
14 * >> then modify assembly to remove call to sleep
15 *
16 * $ powerpc-eabi-gcc -O2 -c matmos-ipp-sp-dotprod-perflib_lg/src/dotprod.s \
17 *    -I./ppc405_0/include/ -Imatmos-ipp-sp-dotprod-perflib_lg/src/ \
18 *    -I./shared_libs/ -L./ppc405_0/lib/ -L./shared_libs/ -Wall
19 *
20 * $ powerpc-eabi-ar rcs libdotprod_lg.a dotprod.o
21 ***************************************************************************/
22
23 #include "f2c.h"           /* for f2c functions */
24
25 /* define assembly instructions for dot-product hardware */
26 /* alti-vec load/store */
27 #define lqfcmx(rn, base, adr)  __asm__ __volatile__(\
28                                "lqfcmx " #rn ",%0,%1\n"\
29                                : : "b" (base), "r" (adr)\
30                                )
31
32 #define stwfcmx(rn, base, adr) __asm__ __volatile__(\
33                                "stwfcmx " #rn ",%0,%1\n"\
34                                : : "b" (base), "r" (adr)\
35                                )
36
37 extern real dphw_result;
38
39 void dotprod_(real *fin, integer *nin, real *oper, integer *nop, integer *nso,
40          doublereal *xx)
41 {
42     /* System generated locals */
43     integer oper_dim1, oper_offset, i__1, indexer;
44
45 /***************************************************************************/
46     real __attribute__ ((aligned (32))) src[112];
47 /***************************************************************************/
48
49     /* Local variables */
50     static integer i__, j, jr, kin;
51
52 /*     FIN(NIN)        R*4  Input function vector */
53 /*     NIN             I*4  Number of points in input vector */
54 /*     OPER(NOP,ODEC)  R*4  Resampling Operators */
55 /*     NOP             I*4  Number of operator points (Length of each sub-operator) */
56 /*     NSO             I*4  Number of sub-operators */
57 /*     XX              R*8  Value at which to sample FIN */
58
59 /*  Outputs: */
60 /*     DOTPROD         R*4 (stored to global variable) */
```

```
61
62  /*  Evaluating the scalar product of the appropriate sub-operator */
63  /*  of OPER with the relevant section of FIN. */
64
65  /*  Note that the full array oper(nso,nop) is rotationally symmetric about the */
66  /*  center such that oper(i,j) = oper(nop+1-i,nso+1-j) */
67  /*  This means that we only have to precompute half the full aray */
68  /*  We could use just the left half (i=1,nop/2) or just the top half (j=1,nso/2) */
69  /*  We choose the former and note that since ir=nop+1-i, and jr=nso+1-j */
70  /*     oper(i,j) = oper(nop+1-i,nso+1-j) = oper(ir,jr) */
71
72      /* Parameter adjustments */
73      --fin;
74      oper_dim1 = (*nop + 1) / 2;
75      oper_offset = 1 + oper_dim1;
76      oper -= oper_offset;
77
78      /* Function Body */
79      kin = (integer) (*xx);
80      jr = (integer) ((*nso - 1) * (*xx - kin)) + 2;
81      j = *nso + 1 - jr;
82  /*  Old dot product code was very simple */
83  /*      do i=1,nop  ! Dot product fin(kin+1) with oper(1,j) */
84  /*         dotprod=dotprod+fin(i+kin)*oper(i,j) */
85  /*      end do */
86
87  /*  New dot product code. */
88  /*  Starting from the two ends of the operator and works toward the middle. */
89  /*  This should provide higher precision if the largest values reside in */
90  /*  the middle of the operator, which they typically do. */
91      i__1 = *nop / 2;
92
93  /*************************************************************************/
94      indexer = 0;
95      for (i__ = 1; i__ <= i__1; ++i__) { /* !!! ASSUMING nop/2 = 28 !!! */
96        src[indexer] = fin[i__ + kin];
97        src[indexer+1] = oper[i__ + j * oper_dim1];
98        src[indexer+2] = fin[*nop + 1 - i__ + kin];
99        src[indexer+3] = oper[i__ + jr * oper_dim1];
100       indexer+=4;
101     }
102     for (i__ = 1; i__ <= i__1; ++i__) {
103       /* Dot product fin(kin+1) with oper(1,j) */
104       lqfcmx(0, src, (i__-1)*16);
105     }
106     /* compile with this to force proper assembly code */
107     /* then remove by hand (in assembly) and rebuild */
108     usleep(1);
109     stwfcmx(0, &dphw_result, 0);
110 /*************************************************************************/
111
112     if (*nop % 2 == 1) {
113         dphw_result += fin[i__ + kin] * oper[i__ + j * oper_dim1];
114     }
115 /* nop is */
116 } /* dotprod_ */
```

C.9 FCM Load/Store Module (*apu_fcm_ldst.v*)

```
/******************************************************************************
 *
 * File:             apu_fcm_ldst.v
 * Version:          1.03.b
 * Description:      APU connected FCM module to handle execution of FCM load
 *                   and store instructions. This module contains two 4-entry
 *                   register files for 32-bit load and store data. The 4-entry
 *                   load register, along with handshaking signals, is avaible
 *                   on the output for direct connection to custom hardware.
 *                   The first entry in the 4-entry store register is tied to a
 *                   32-bit input and can be connected to the output of custom
 *                   hardware. In this core version, entries 1 to 3 in the
 *                   store register can not be modified.
 *                   This core expects a single hw unit to be connected to it
 *                   and work with the load register data. This core also
 *                   expects that a single store instruction will always follow
 *                   a certain number of load instructions (and then this cycle
 *                   can repeat). The number of load instructions prior to the
 *                   store instruction is determined by the custom hw.
 *                   This core is a modified version of the core presented in
 *                   XAPP717, http://www.xilinx.com/bvdocs/appnotes/xapp717.pdf
 *                   with a number of bug fixes to guarantee proper operation
 *                   with other hardware co-processors on the FCB.
 * Original Author:  SEG, XAPP717, Xilinx, Inc.
 * Date:             Aug 17, 2004 (ver 1.00.b)
 * Modified by:      Dmitriy Bekker, Rochester Institute of Technology
 * Date:             June 14, 2007 (ver 1.03.a)
 *
 * Target:           Virtex-4FX
 * Maximum Freq:     316 MHz (-11 grade, reported by XST)
 *
 ******************************************************************************/

/******************************************************************************
 *
 *     XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
 *     SOLELY FOR USE IN DEVELOPING PROGRAMS AND SOLUTIONS FOR
 *     XILINX DEVICES.  BY PROVIDING THIS DESIGN, CODE, OR INFORMATION
 *     AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION
 *     OR STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS
 *     IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
 *     AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
 *     FOR YOUR IMPLEMENTATION.  XILINX EXPRESSLY DISCLAIMS ANY
 *     WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
 *     IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
 *     REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
 *     INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
 *     FOR A PARTICULAR PURPOSE.
 *
 *     (c) Copyright 2005 Xilinx, Inc.
 *     All rights reserved.
 *
 ******************************************************************************/


`timescale 1 ns / 1 ps

module apu_fcm_ldst (

   // outputs to APU
```

```
61    FCMAPUINSTRACK,
62    FCMAPURESULT,
63    FCMAPUDONE,
64    FCMAPUSLEEPNOTREADY,
65    FCMAPUDECODEBUSY,
66    FCMAPUDCDGPRWRITE,
67    FCMAPUDCDRAEN,
68    FCMAPUDCDRBEN,
69    FCMAPUDCDPRIVOP,
70    FCMAPUDCDFORCEALIGN,
71    FCMAPUDCDXEROVEN,
72    FCMAPUDCDXERCAEN,
73    FCMAPUDCDCREN,
74    FCMAPUEXECRFIELD,
75    FCMAPUDCDLOAD,
76    FCMAPUDCDSTORE,
77    FCMAPUDCDUPDATE,
78    FCMAPUDCDLDSTBYTE,
79    FCMAPUDCDLDSTHW,
80    FCMAPUDCDLDSTWD,
81    FCMAPUDCDLDSTDW,
82    FCMAPUDCDLDSTQW,
83    FCMAPUDCDTRAPLE,
84    FCMAPUDCDTRAPBE,
85    FCMAPUDCDFORCEBESTEERING,
86    FCMAPUDCDFPUOP,
87    FCMAPUEXEBLOCKINGMCO,
88    FCMAPUEXENONBLOCKINGMCO,
89    FCMAPULOADWAIT,
90    FCMAPURESULTVALID,
91    FCMAPUXEROV,
92    FCMAPUXERCA,
93    FCMAPUCR,
94    FCMAPUEXCEPTION,
95
96    // outputs to custom hardware
97    FCMHWDATA0,
98    FCMHWDATA1,
99    FCMHWDATA2,
100   FCMHWDATA3,
101   FCMHWVALID,
102
103   // inputs from custom hardware
104   HWFCMDATA0,
105   HWFCMVALID,
106
107   // inputs from APU
108   APUFCMINSTRUCTION,
109   APUFCMINSTRVALID,
110   APUFCMRADATA,
111   APUFCMRBDATA,
112   APUFCMOPERANDVALID,
113   APUFCMFLUSH,
114   APUFCMWRITEBACKOK,
115   APUFCMLOADDATA,
116   APUFCMLOADDVALID,
117   APUFCMLOADBYTEEN,
118   APUFCMENDIAN,
119   APUFCMXERCA,
120   APUFCMDECODED,
121   APUFCMDECUDI,
122   APUFCMDECUDIVALID,
123
124   // clock and reset
```

```
125     clock,
126     reset
127
128     );
129
130     // state mnemonics
131     parameter
132        STATE_IDLE   = 2'b00, // idle state
133        STATE_LOAD   = 2'b01, // valid load instruction detected
134        STATE_STORE  = 2'b10, // valid store instruction detected
135        STATE_WAITHW = 2'b11; // wait for hw to finish processing
136
137     // port declarations
138     // outputs to APU
139     output        FCMAPUINSTRACK;
140     output [0:31] FCMAPURESULT;
141     output        FCMAPUDONE;
142     output        FCMAPUSLEEPNOTREADY;
143     output        FCMAPUDECODEBUSY;
144     output        FCMAPUDCDGPRWRITE;
145     output        FCMAPUDCDRAEN;
146     output        FCMAPUDCDRBEN;
147     output        FCMAPUDCDPRIVOP;
148     output        FCMAPUDCDFORCEALIGN;
149     output        FCMAPUDCDXEROVEN;
150     output        FCMAPUDCDXERCAEN;
151     output        FCMAPUDCDCREN;
152     output [0:2]  FCMAPUEXECRFIELD;
153     output        FCMAPUDCDLOAD;
154     output        FCMAPUDCDSTORE;
155     output        FCMAPUDCDUPDATE;
156     output        FCMAPUDCDLDSTBYTE;
157     output        FCMAPUDCDLDSTHW;
158     output        FCMAPUDCDLDSTWD;
159     output        FCMAPUDCDLDSTDW;
160     output        FCMAPUDCDLDSTQW;
161     output        FCMAPUDCDTRAPLE;
162     output        FCMAPUDCDTRAPBE;
163     output        FCMAPUDCDFORCEBESTEERING;
164     output        FCMAPUDCDFPUOP;
165     output        FCMAPUEXEBLOCKINGMCO;
166     output        FCMAPUEXENONBLOCKINGMCO;
167     output        FCMAPULOADWAIT;
168     output        FCMAPURESULTVALID;
169     output        FCMAPUXEROV;
170     output        FCMAPUXERCA;
171     output [0:3]  FCMAPUCR;
172     output        FCMAPUEXCEPTION;
173
174     // outputs to custom hardware
175     output [0:31] FCMHWDATA0;
176     output [0:31] FCMHWDATA1;
177     output [0:31] FCMHWDATA2;
178     output [0:31] FCMHWDATA3;
179     output        FCMHWVALID;
180
181     // inputs from custom hardware
182     input  [0:31] HWFCMDATA0;
183     input         HWFCMVALID;
184
185     // inputs from APU
186     input  [0:31] APUFCMINSTRUCTION;
187     input         APUFCMINSTRVALID;
188     input  [0:31] APUFCMRADATA;
```

```
189     input  [0:31] APUFCMRBDATA;
190     input         APUFCMOPERANDVALID;
191     input         APUFCMFLUSH;
192     input         APUFCMWRITEBACKOK;
193     input  [0:31] APUFCMLOADDATA;
194     input         APUFCMLOADDVALID;
195     input  [0:3]  APUFCMLOADBYTEEN;
196     input         APUFCMENDIAN;
197     input         APUFCMXERCA;
198     input         APUFCMDECODED;
199     input  [0:2]  APUFCMDECUDI;
200     input         APUFCMDECUDIVALID;
201
202     // clock and reset
203     input         clock;
204     input         reset;
205
206     // internal signals
207     // PPC instruction-related
208     reg  [0:5] reg_RT;              // target register of PPC instruction
209     reg  [0:5] reg_RA;              // base register of PPC instruction
210     reg  [0:5] reg_RB;              // offset register of PPC instruction
211     wire instrreg_we;               // write enable
212     wire data_read;                 // hw data has been read
213     wire loaddata_inidle;           // load data came in idle state
214     reg  apufcminstrvalid_reg;      // registered version of APUFCMINSTRVALID
215     reg  apufcmdecoded_reg;         // registered version of APUFCMDECODED
216     reg  [0:31] apufcmloaddata_reg; // registered version of APUFCMLOADDATA
217     reg  apufcmloaddvalid_reg;      // registered version of APUFCMLOADDVALID
218     reg  apufcmflush_reg;           // registered version of APUFCMFLUSH
219
220     // loads and stores
221     wire store_or_loadn;            // 1=store, 0=load
222     reg  store_or_loadn_reg;        // stores the store_or_loadn signal
223     wire ldst_update;               // 1=update RA
224     wire [0:1] ldst_size;           // number of words to transfer
225     reg  [0:1] ldst_size_reg;       // stores number of words to transfer
226     reg  [0:1] ldst_size_counter;   // counter for number of words
227     wire ldst_size_counter_we;      // write enable
228     wire ldst_valid;                // 1=valid load/store instruction
229     reg  ldst_valid_reg;            // registered version of ldst_valid
230
231     // register file for FCM loads
232     reg  [0:31] regfile [0:3];      // 32-bit entry register file with 4 regs
233     wire regfile_we;                // write enable
234     wire [0:1]  regfile_waddr;      // write address
235     wire [0:31] regfile_wdata;      // write data
236
237     // register file for FCM stores
238     reg  [0:31] regfile_store[0:3]; // 32-bit entry register file with 4 regs
239     wire regfile_raddr_we;          // write enable
240     reg  [0:1]  regfile_raddr;      // register to store read address
241     wire [0:31] regfile_rdata;      // read data
242
243     // state registers
244     reg  [0:1]  curr_state;         // current state
245     reg  [0:1]  next_state;         // next state
246
247     // custom hw interconnect
248     wire [0:31] hwdata0;            // data from hw
249     reg  hw_datardy;                // hw data is ready
250     wire hw_valid;                  // valid signal from HW to FCM
251
252     /****************************** sequential blocks ************************/
```

```
253     // Delayed (registered) versions of certain signals
254     always @(posedge clock or posedge reset)
255     begin
256        if (reset)
257        begin
258           apufcminstrvalid_reg <= 1'b0;
259           apufcmdecoded_reg    <= 1'b0;
260           apufcmloaddvalid_reg <= 1'b0;
261           apufcmflush_reg      <= 1'b0;
262           ldst_valid_reg       <= 1'b0;
263        end
264        else
265        begin
266           apufcminstrvalid_reg <= APUFCMINSTRVALID;
267           apufcmdecoded_reg    <= APUFCMDECODED;
268           apufcmloaddvalid_reg <= APUFCMLOADDVALID;
269           apufcmflush_reg      <= APUFCMFLUSH;
270           ldst_valid_reg       <= ldst_valid;
271        end
272     end
273
274     // Synchronize load data in register
275     always @(posedge clock or posedge reset)
276     begin
277        if (reset)
278           apufcmloaddata_reg <= 32'b0;
279        else if (~loaddata_inidle)
280           apufcmloaddata_reg <= APUFCMLOADDATA;
281     end
282
283     // PPC instruction-related registers
284     always @(posedge clock or posedge reset)
285     begin
286        if (reset)
287        begin
288           reg_RT <= 5'b0;
289           reg_RA <= 5'b0;
290           reg_RB <= 5'b0;
291           store_or_loadn_reg <= 1'b0;
292           ldst_size_reg <= 2'b0;
293        end
294        else if (instrreg_we) // capture instruction information
295        begin
296           reg_RT <= APUFCMINSTRUCTION[6:10];
297           reg_RA <= APUFCMINSTRUCTION[11:15];
298           reg_RB <= APUFCMINSTRUCTION[16:20];
299           store_or_loadn_reg <= store_or_loadn;
300           ldst_size_reg <= ldst_size;
301        end
302     end
303
304     // read address of register file
305     always @(posedge clock or posedge reset)
306     begin
307        if (reset)
308           regfile_raddr <= 5'h0;
309        else if (instrreg_we)
310           regfile_raddr <= APUFCMINSTRUCTION[6:10];
311        else if (regfile_raddr_we)
312           regfile_raddr <= regfile_raddr + 1;
313     end
314
315     // load / store counter for number of words
316     always @(posedge clock or posedge reset)
```

```
317     begin
318        if (reset)
319           ldst_size_counter <= 2'b0;
320        else if (instrreg_we)          // keep counter reset
321           ldst_size_counter <= 2'b0;
322        else if (ldst_size_counter_we) // increment counter
323           ldst_size_counter <= ldst_size_counter + 1;
324     end
325
326     // register file (load)
327     always @(posedge clock or posedge reset)
328     begin
329        if (reset) // set all register values to zero
330        begin
331           regfile[0]  <= 32'b0; regfile[1]  <= 32'b0;
332           regfile[2]  <= 32'b0; regfile[3]  <= 32'b0;
333        end // if (reset)
334        else if (regfile_we) // write data to specified register
335           regfile[regfile_waddr] <= regfile_wdata;
336     end
337
338     // register file (store)
339     always @(posedge clock or posedge reset)
340     begin
341        if (reset) // set all register values to zero
342        begin
343           regfile_store[0]  <= 32'b0; regfile_store[1]  <= 32'b0;
344           regfile_store[2]  <= 32'b0; regfile_store[3]  <= 32'b0;
345           hw_datardy <= 1'b0;
346        end // if (reset)
347        else if (hw_valid) // load data from hw
348        begin
349           regfile_store[0] <= hwdata0;
350           hw_datardy <= 1'b1;
351        end // else if (hw_valid)
352        else if (data_read)    // will be high every time on valid load
353           hw_datardy <= 1'b0; // expects n loads to follow a single store
354     end
355
356     // state machine
357     always @(posedge clock or posedge reset)
358     begin
359        if (reset)
360           curr_state <= STATE_IDLE;
361        else
362           curr_state <= next_state;
363     end
364
365     /*************************** combinational blocks ***********************/
366     // decoder
367     decode_ldst decode_ldst_0 (
368        // outputs
369        .update(ldst_update),
370        .size(ldst_size),
371        .store_or_loadn(store_or_loadn),
372        .valid_ldst(ldst_valid),
373        // inputs
374        .APUFCMINSTRUCTION(APUFCMINSTRUCTION) );
375
376     // state machine logic
377     always @(curr_state or store_or_loadn_reg or ldst_size_counter or
378              ldst_size_reg or hw_datardy or ldst_valid_reg or APUFCMFLUSH or
379              apufcmflush_reg or apufcminstrvalid_reg or
380              apufcmloaddvalid_reg or apufcmdecoded_reg)
```

```
381      begin
382         case (curr_state)
383
384         // wait for valid instruction
385         STATE_IDLE:
386            // valid instruction from APU (and not flushed)
387            // check the dalyed version (in order to meet timing)
388            if (apufcminstrvalid_reg & apufcmdecoded_reg & ldst_valid_reg &
389                ~APUFCMFLUSH & ~apufcmflush_reg)
390               if (store_or_loadn_reg) // store instruction
391                  if (hw_datardy) // is hw ready?
392                     next_state = STATE_STORE;
393                  else
394                     next_state = STATE_WAITHW;
395               else // load instruction
396                  if (apufcmloaddvalid_reg) // load data arrived at the same time
397                     if (ldst_size_counter < ldst_size_reg)
398                        next_state = STATE_LOAD;
399                     else
400                        next_state = STATE_IDLE;
401                  else
402                     next_state = STATE_LOAD;
403            else
404               next_state = STATE_IDLE;
405
406         // seen a valid load instruction, wait for valid data
407         STATE_LOAD:
408            if( APUFCMFLUSH )
409               next_state = STATE_IDLE;
410            else
411               // keep track of how many words to access
412               if (ldst_size_counter < ldst_size_reg)
413                  next_state = STATE_LOAD;
414               else
415                  if (apufcmloaddvalid_reg)
416                     next_state = STATE_IDLE;
417                  else
418                     next_state = STATE_LOAD;
419
420         // wait for hw to finish
421         STATE_WAITHW:
422            if( APUFCMFLUSH )
423               next_state = STATE_IDLE;
424            else
425               if( hw_datardy)
426                  next_state = STATE_STORE;
427               else
428                  next_state = STATE_WAITHW;
429
430         // seen a valid store instruction, output data
431         STATE_STORE:
432            // keep track of how many words to access
433            if ( (ldst_size_counter < ldst_size_reg) & ~APUFCMFLUSH )
434               next_state = STATE_STORE;
435            else
436               next_state = STATE_IDLE;
437
438         default:
439            next_state = STATE_IDLE;
440
441         endcase // case(curr_state)
442      end // always @ (APUFCMINSTRUCTION or APUFCMINSTRVALID or ...
443
444      // internal signal assignments
```

```
445     assign instrreg_we = (curr_state == STATE_IDLE);
446
447     // flag when data comes in idle state
448     assign loaddata_inidle = (curr_state == STATE_IDLE) & apufcmloaddvalid_reg
449                               & ldst_valid_reg;
450
451     // mark data read only when a new load instruction has been issued
452     // this implies that a store must have completed successfully since
453     // an n number of loads always follow a single store
454     assign data_read = ( apufcminstrvalid_reg & apufcmdecoded_reg &
455                          ldst_valid_reg & ~store_or_loadn_reg );
456
457     // data might arrive at the same time as the instruction
458     assign regfile_we = ( ( (curr_state == STATE_LOAD) & apufcmloaddvalid_reg)
459                          | loaddata_inidle );
460
461     // reg_RT is the target register value when data comes after the instruction
462     assign regfile_waddr = (reg_RT + ldst_size_counter);
463
464     // write data from memory to register file
465     assign regfile_wdata = apufcmloaddata_reg;
466
467     // address of register file to read data from for a store operation
468     assign regfile_raddr_we = (((curr_state == STATE_LOAD) &
469                                 apufcmloaddvalid_reg) |
470                                (curr_state == STATE_STORE) | loaddata_inidle);
471
472     // data read for a store operation
473     assign regfile_rdata = regfile_store[regfile_raddr];
474
475     // update counter for number of data words transferred
476     assign ldst_size_counter_we =(((curr_state == STATE_LOAD) &
477                                     apufcmloaddvalid_reg) |
478                                   (curr_state == STATE_STORE) | loaddata_inidle);
479
480     // hw to fcm valid
481     assign hw_valid = HWFCMVALID;
482
483     // data inputs (from hw)
484     assign hwdata0 = HWFCMDATA0;
485
486     // output assignments
487     // fcm-hw valid when done loading data
488     assign FCMHWVALID = ( ( (curr_state == STATE_LOAD) & apufcmloaddvalid_reg &
489                             (ldst_size_counter == ldst_size_reg) ) |
490                           ( loaddata_inidle &
491                           ( ldst_size_counter == ldst_size_reg ) ) );
492
493     // instruction completed
494     assign FCMAPUDONE = ( ( (curr_state == STATE_LOAD) & apufcmloaddvalid_reg &
495                             (ldst_size_counter == ldst_size_reg) ) |
496                           ( (curr_state == STATE_STORE) &
497                             (ldst_size_counter == ldst_size_reg) ) |
498                           ( loaddata_inidle &
499                           ( ldst_size_counter == ldst_size_reg ) ) );
500
501     // the result (for stores)
502     assign FCMAPURESULT = (curr_state == STATE_STORE) ? regfile_rdata : 32'b0;
503
504     // valid in the store state
505     assign FCMAPURESULTVALID = (curr_state == STATE_STORE);
506
507     // ask the APU to wait only in one case
508     assign FCMAPULOADWAIT = loaddata_inidle;
```

```
509
510     // don't allow the CPU to go to sleep while executing an instruction
511     assign FCMAPUSLEEPNOTREADY = ( (apufcminstrvalid_reg & apufcmdecoded_reg &
512                                    ldst_valid_reg) | (curr_state == STATE_LOAD)
513                                    | (curr_state == STATE_WAITHW) |
514                                      (curr_state == STATE_STORE) );
515
516     // data outputs (to hw)
517     assign FCMHWDATA0 = regfile[0];
518     assign FCMHWDATA1 = regfile[1];
519     assign FCMHWDATA2 = regfile[2];
520     assign FCMHWDATA3 = regfile[3];
521
522     // unused output signals
523     assign FCMAPUINSTRACK = 1'b0;
524     assign FCMAPUDECODEBUSY = 1'b0;
525     assign FCMAPUDCDGPRWRITE = 1'b0;
526     assign FCMAPUDCDRAEN = 1'b0;
527     assign FCMAPUDCDRBEN = 1'b0;
528     assign FCMAPUDCDPRIVOP = 1'b0;
529     assign FCMAPUDCDFORCEALIGN = 1'b0;
530     assign FCMAPUDCDXEROVEN = 1'b0;
531     assign FCMAPUDCDXERCAEN = 1'b0;
532     assign FCMAPUDCDCREN = 1'b0;
533     assign FCMAPUEXECRFIELD = 3'b0;
534     assign FCMAPUDCDLOAD = 1'b0;
535     assign FCMAPUDCDSTORE = 1'b0;
536     assign FCMAPUDCDUPDATE = 1'b0;
537     assign FCMAPUDCDLDSTBYTE = 1'b0;
538     assign FCMAPUDCDLDSTHW = 1'b0;
539     assign FCMAPUDCDLDSTWD = 1'b0;
540     assign FCMAPUDCDLDSTDW = 1'b0;
541     assign FCMAPUDCDLDSTQW = 1'b0;
542     assign FCMAPUDCDTRAPLE = 1'b0;
543     assign FCMAPUDCDTRAPBE = 1'b0;
544     assign FCMAPUDCDFORCEBESTEERING = 1'b0;
545     assign FCMAPUDCDFPUOP = 1'b0;
546     assign FCMAPUEXEBLOCKINGMCO = 1'b0;
547     assign FCMAPUEXENONBLOCKINGMCO = 1'b0;
548     assign FCMAPUXEROV = 1'b0;
549     assign FCMAPUXERCA = 1'b0;
550     assign FCMAPUCR = 4'b0;
551     assign FCMAPUEXCEPTION = 1'b0;
552
553  endmodule // apu_fcm_ldst
554
555  // decoder for load/store instructions
556  module decode_ldst (
557     // outputs
558     update,
559     size,
560     store_or_loadn,
561     valid_ldst,
562     // inputs
563     APUFCMINSTRUCTION );
564
565     // output signals
566     output       update;        // 1=RA is loaded with effective address
567     output [0:1] size;          // transaction size
568     reg    [0:1] size;
569     output       store_or_loadn;// 1=store, 0=load
570     output       valid_ldst;    // if this instruction is a valid FCM load/store
571
572     // input signals
```

```
573     input [0:31] APUFCMINSTRUCTION;
574
575     assign update = APUFCMINSTRUCTION[21];
576     assign store_or_loadn = APUFCMINSTRUCTION[23];
577     assign valid_ldst = ( (APUFCMINSTRUCTION[0:5]   == 6'b011111) &
578                           (APUFCMINSTRUCTION[26:31] == 6'b001110) );
579
580     always @(APUFCMINSTRUCTION[22] or APUFCMINSTRUCTION[24:25])
581     begin
582        case({APUFCMINSTRUCTION[22], APUFCMINSTRUCTION[24:25]})
583
584        3'b100: size = 2'b01; // double-word
585        3'b011: size = 2'b11; // quad-word
586        3'b111: size = 2'b11; // quad-word
587        default: size = 2'b0;
588
589        endcase // case({APUFCMINSTRUCTION[22], APUFCMINSTRUCTION[24:25]})
590     end
591
592  endmodule // decode_ldst
```

C.10 Dot-product Module (*fp_dot_prod.vhd*)

```
1  --------------------------------------------------------------------------------
2  --                                                                            --
3  -- Name:      fp_dot_prod.vhd                                                 --
4  -- Version:   1.00.f                                                          --
5  -- Author:    Dmitriy Bekker                                                  --
6  --            Rochester Institute of Technology                               --
7  -- Date:      May 14, 2007                                                    --
8  --                                                                            --
9  -- Target:    Virtex-4                                                        --
10 -- Max Freq:  237 MHz (-11 grade, reported by XST)                            --
11 -- Latency:   22 cycles                                                       --
12 -- Max Rate:  4, 32-bit floats once every 7 cycles                            --
13 -- NOTE:      In this version of the core, the accumulator is latched         --
14 --                                                                            --
15 -- Description:                                                               --
16 -- This is a single precision floating-point dot-product core. This core can --
17 -- be integrated with the apu_fcm_ldst core for interfacing with the APU on  --
18 -- the Virtex-4FX FPGA. The 'iterations' generic is used to set the amount   --
19 -- of input data to expect. For example, with 'iterations' = 7, this core    --
20 -- calculates the dot-product of 7 sets of 4, 32-bit floats. This core has   --
21 -- been tested in actual hardware with 'iterations' = 7 and clock frequency  --
22 -- of 200 MHz.                                                                --
23 --------------------------------------------------------------------------------
24
25 -- Use necessary packages
26 library IEEE;
27 use IEEE.STD_LOGIC_1164.ALL;
28
29 entity fp_dot_prod is
30 generic (
31    iterations   : natural := 7 );
32 port (
33
34    -- inputs from LD/ST module
35    FCMHWVALID   : in std_logic;
36    FCMHWDATA0   : in std_logic_vector(0 to 31);
37    FCMHWDATA1   : in std_logic_vector(0 to 31);
38    FCMHWDATA2   : in std_logic_vector(0 to 31);
39    FCMHWDATA3   : in std_logic_vector(0 to 31);
40
41    -- outputs to LD/ST module
42    HWFCMVALID   : out std_logic;
43    HWFCMDATA0   : out std_logic_vector(0 to 31);
44
45    -- clock and reset
46    CLK          : in std_logic;
47    RST          : in std_logic );
48
49 end fp_dot_prod;
50
51 architecture behavioral of fp_dot_prod is
52
53    -- Coregen single-precision multiplier
54    component fp_mult is
55    port (
56       a             : in std_logic_vector(31 downto 0);
57       b             : in std_logic_vector(31 downto 0);
58       operation_nd  : in std_logic;
59       clk           : in std_logic;
60       sclr          : in std_logic;
```

```
61          result        : out std_logic_vector(31 downto 0);
62          rdy           : out std_logic);
63       end component;
64
65       -- Coregen single-precision adder
66       component fp_add is
67       port (
68          a             : in std_logic_vector(31 downto 0);
69          b             : in std_logic_vector(31 downto 0);
70          operation_nd  : in std_logic;
71          clk           : in std_logic;
72          sclr          : in std_logic;
73          result        : out std_logic_vector(31 downto 0);
74          rdy           : out std_logic);
75       end component;
76
77       -- input data
78       signal DATA0           : std_logic_vector(31 downto 0);
79       signal DATA1           : std_logic_vector(31 downto 0);
80       signal DATA2           : std_logic_vector(31 downto 0);
81       signal DATA3           : std_logic_vector(31 downto 0);
82       signal data0_reg       : std_logic_vector(31 downto 0);
83       signal data1_reg       : std_logic_vector(31 downto 0);
84       signal data2_reg       : std_logic_vector(31 downto 0);
85       signal data3_reg       : std_logic_vector(31 downto 0);
86
87       -- intermediate results
88       signal mult0_result   : std_logic_vector(31 downto 0);
89       signal mult1_result   : std_logic_vector(31 downto 0);
90       signal add0_result    : std_logic_vector(31 downto 0);
91       signal add1_result    : std_logic_vector(31 downto 0);
92       signal accumulate     : std_logic_vector(31 downto 0);
93
94       -- intermediate control
95       signal mult0_rdy      : std_logic;
96       signal mult1_rdy      : std_logic;
97       signal add0_rdy       : std_logic;
98       signal add1_rdy       : std_logic;
99       signal arith_rst      : std_logic;
100      signal acc_clr        : std_logic;
101
102      -- internal input signals
103      signal FCMHWVALID_i    : std_logic;
104      signal fcmhwvalid_reg1 : std_logic;
105      signal fcmhwvalid_reg2 : std_logic;
106
107      -- internal output signals
108      signal HWFCMVALID_i   : std_logic;
109      signal HWFCMDATA0_i   : std_logic_vector(31 downto 0);
110
111      -- state machine
112      type state_type is ( STATE_RESET, STATE_IDLE, STATE_COUNT, STATE_OUTPUT );
113      signal my_state : state_type;
114
115   begin
116
117      -- first multiplier (parallel)
118      fp_mult_0 : fp_mult
119      port map (
120         a             => data0_reg,
121         b             => data1_reg,
122         operation_nd  => fcmhwvalid_reg2,
123         clk           => CLK,
124         sclr          => arith_rst,
```

```
125         result         => mult0_result,
126         rdy            => mult0_rdy );
127
128      -- second multiplier (parallel)
129      fp_mult_1 : fp_mult
130      port map (
131         a              => data2_reg,
132         b              => data3_reg,
133         operation_nd  => fcmhwvalid_reg2,
134         clk            => CLK,
135         sclr           => arith_rst,
136         result         => mult1_result,
137         rdy            => mult1_rdy );
138
139      -- first adder (sum of multipliers)
140      fp_add_0 : fp_add
141      port map (
142         a              => mult0_result,
143         b              => mult1_result,
144         operation_nd  => mult0_rdy,
145         clk            => CLK,
146         sclr           => arith_rst,
147         result         => add0_result,
148         rdy            => add0_rdy );
149
150      -- second adder (accumulate result)
151      fp_add_1 : fp_add
152      port map (
153         a              => add0_result,
154         b              => accumulate,
155         operation_nd  => add0_rdy,
156         clk            => CLK,
157         sclr           => arith_rst,
158         result         => add1_result,
159         rdy            => add1_rdy );
160
161      -- input assignments
162      DATA0 <= FCMHWDATA0;
163      DATA1 <= FCMHWDATA1;
164      DATA2 <= FCMHWDATA2;
165      DATA3 <= FCMHWDATA3;
166      FCMHWVALID_i <= FCMHWVALID;
167
168      -- reset control
169      arith_rst <= RST or HWFCMVALID_i;
170
171      -- output assignments
172      HWFCMVALID <= HWFCMVALID_i;
173      HWFCMDATA0 <= HWFCMDATA0_i;
174
175      -- synchronize the arithmetic
176      control: process( CLK )
177      variable counter : integer range 0 to iterations;
178      begin
179
180         if( rising_edge( CLK ) ) then
181
182            -- synchronous reset
183            if( RST = '1' ) then
184               my_state <= STATE_RESET;
185            else
186
187               case my_state is
188
```

```
189            when STATE_RESET =>
190               HWFCMDATA0_i <= (others => '0'); -- just clear output data, idle
191               my_state <= STATE_IDLE;          -- state will take care of rest
192
193            when STATE_IDLE =>
194               HWFCMVALID_i <= '0';             -- lower the output valid line
195               acc_clr <= '1';                  -- let latch clear accumulator
196               counter := 0;                    -- reset counter
197               if( add0_rdy = '1' ) then        -- go to next state when add0
198                  my_state <= STATE_COUNT;      -- is ready
199                  acc_clr <= '0';               -- release latch from reset
200               end if;
201
202            when STATE_COUNT =>
203               if( add1_rdy = '1' ) then        -- count ready pulses
204                  counter := counter + 1;
205                  acc_clr <= '0';               -- make sure latch is not reset
206               elsif( counter = iterations ) then
207                  my_state <= STATE_OUTPUT;     -- go to output state when done
208               end if;
209
210            when STATE_OUTPUT =>
211               HWFCMVALID_i <= '1';             -- mark valid line
212               HWFCMDATA0_i <= add1_result;     -- register output data
213               my_state <= STATE_IDLE;          -- go back to idle state
214
215            end case;
216
217         end if;
218
219      end if;
220
221   end process control;
222
223   -- latch process for accumulator data
224         latch: process( acc_clr, add1_rdy, add1_result )
225         begin
226      if( acc_clr = '1' ) then                     -- reset accumulator
227         accumulate <= (others => '0');
228      elsif( add1_rdy = '1' ) then                 -- latch add1 result
229         accumulate <= add1_result;                -- when add1 is ready
230      end if;
231         end process latch;
232
233   -- register input data
234   in_reg: process( CLK )
235   begin
236
237      if( rising_edge( CLK ) ) then
238         fcmhwvalid_reg1 <= FCMHWVALID_i;
239         fcmhwvalid_reg2 <= fcmhwvalid_reg1;    -- necessary delay for valid
240
241         -- synchronous reset
242         if( RST = '1' ) then
243            fcmhwvalid_reg1 <= '0';
244            fcmhwvalid_reg2 <= '0';
245            data0_reg <= (others => '0');
246            data1_reg <= (others => '0');
247            data2_reg <= (others => '0');
248            data3_reg <= (others => '0');
249
250         -- update data registers
251         elsif( fcmhwvalid_reg1 = '1' ) then
252            data0_reg <= DATA0;
```

```
253              data1_reg <= DATA1;
254              data2_reg <= DATA2;
255              data3_reg <= DATA3;
256           end if;
257        end if;
258
259     end process in_reg;
260
261  end behavioral;
```