Accelerating the interior point method on a FPGA

Koh, Christopher Seng Lee

2009

Koh, C. S. L. (2009). Accelerating the interior point method on a FPGA. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/18685>

<https://doi.org/10.32657/10356/18685>

Accelerating the Interior Point Method on a FPGA

CHRISTOPHER KOH SENG LEE

School of Electrical & Electronic Engineering

A thesis submitted to the Nanyang Technological University

in partial fulfillment of the requirements for the degree of

Masters of Engineering

2009

Acknowledgements

I would like to express my deep appreciation to my supervisor Dr Ling Keck Voon, for his guidance, patience and encouragement all throughout the entire course of this project leading to this thesis. I have picked many valuable experiences from him on how to formulate my thoughts when tackling difficult problems faced in the course of my thesis.

I would like to thank all the staff and students from the Control Engineering and Automation Lab for providing me the technical equipment and support in completing this project. I would like to especially thank Yue Siew Peng, Ray Tan Hang Kiang and Wu Bingfang especially for being especially patient and helpful, and also for sharing their knowledge and expertise with me.

Last but not least, I would like to thank my wife Wendy for her love, patience and encouragement throughout the course of my study. In addition, I would also like to thank my dad for giving me his support and encouragement for me to further my studies.

Table of Contents

Acknowledgements	ii
Table of Contents	iii
List of Figures	vi
Abstract	xi
Chapter 1 Introduction	1
1.1 Motivation	1
1.2 Major Contributions of the Thesis	3
1.3 Thesis Organization	6
Chapter 2 Accelerating the Microblaze Processor	7
2.1 A Quick Introduction to FPGA	7
2.1.1 Microblaze as a Reconfigurable Processor	8
2.1.2 The Hardware Co-Processor	9
2.1.3 Xilinx Core Generator	9
2.1.4 Performance Characteristics of Xilinx Floating Point Cores	10
2.2 Options for accelerating Microblaze performance with a Co-Processor	10
2.2.1 Accelerating one floating point operation	11
2.2.2 Accelerating N floating point operations	14
2.2.3 Accelerating N floating point operations using loop-unrolling	17
2.2.4 Pipelining to accelerate N floating point operations using Co-Processor	20
2.3 Accelerating Co-Processor performance by reducing data transfer overhead	27
2.3.1 Speed up analysis of using Co-Processor to perform Matrix vector multiplication $y = Ax$ of variable sizes	35
2.4 Conclusion	39
Chapter 3 Design of the Co-Processor Modules for Microblaze	40
3.1 Attaching a Co-Processor to Microblaze	40
3.1.1 Co-Processor Slave Interface	40
3.1.2 Co-Processor Master Interface	41
3.2 Co Processor Design for $y = Ax$ operation	41

3.2.1	Pipeline Depth Selection of Hardware Floating point Adder and Multiplier	49
3.2.2	Verification of functionality of Matrix Vector Multiplier Co-Processor..	51
3.3	Co Processor Design for Floating point Division	53
3.3.1	Verification of functionality of Floating Point Division Accelerator Co-Processor	57
3.4	Actual versus Theoretical Performance of the Matrix Vector Multiplier Co-Processor	58
3.5	Actual versus Theoretical Performance of the Floating Point Division Accelerator Co-Processor	61
Chapter 4	Using Co-Processor to speed up the Interior Point Method	62
4.1	The Interior Point Method for solving QP problems	62
4.2	Determining the Bottleneck of the Interior Point Method	64
4.2.1	The Test-bed	64
4.2.2	Step 2 is the Main Bottleneck	64
4.3	Using the Matrix Vector Multiplier Co-Processor.....	65
4.4	Using the Divide Co-Processor.....	70
4.5	Actual Speed up of the Interior Point Method when using both Co-Processors	71
4.6	FPGA Resource Usage and Code Size Requirements for the IPM	76
Chapter 5	Conclusion	78
5.1	Eliminating Function Call in $y = Ax$ operation on MB	78
5.2	For-loop or Loop-unroll coding style	80
5.3	Recommendations for Further Research.....	82
References		84
Appendix A	Software Required and the Development Cycle.....	87
A.1	Using the Embedded Development Kit (EDK)	87
A.2	The Xilinx Integrated Software Environment (ISE).....	87
A.3	Using Modelsim as a functional verification tool.....	88
A.4	The Design Cycle.....	88

Appendix B	Assembly instructions for $y=Ax$ operation.	90
Appendix C	C code for Step 2.....	97
Appendix D	Co-Processor Source code	99
D.1	Matrix Vector Multiplication Co-Processor Source Code.....	99
D.2	Floating point Divider Co-Processor	108

List of Figures

1.1	Graph of computation time per QP problem vs number of inequalities (m_c) with number of decision variables fixed at $n_v=3$	4
1.2	Graph of computation time per QP problem vs number of decision variables (n_v) with number of inequality constraints fixed at $m_c=32$	5
2.1	The Microblaze core [17].....	8
2.2	The Hardware Co-Processor	9
2.3	C source code to implement a floating point operation	11
2.4	C source code to do N floating point operations by for-loop	14
2.5	C source code to implement N floating point operations by loop-unrolling	18
2.6	Pipelining floating point multiplications/additions when using a Co-Processor	21
2.7	Pipelining floating point divisions when using a Co-Processor	22
2.8	For-loop C coding style to compute floating point operations using Co-Processor with buffer.....	22
2.9	Clock cycles required for Co-Processor to do 4 floating point operations back-to-back (for-loop based firmware)	24
2.10	Using the Co-Processor to perform computation of $y=Ax$	29
2.11	C source codes for implementing matrix vector multiplication.....	30
2.12	C source codes for implementing dot product multiplication.....	30
2.13	For-loop C coding style to compute Matrix Vector Multiplication using Co-Processor	31
2.14	C code implementation of matrix vector multiplication $y=Ax$ (by loop-unrolling)	34
2.15	Speed up factor for various sizes of A	36
2.16	Speed up factor for various sizes of A when code is loop unrolled.....	38
3.1	Simplified Block Diagram of Matrix Vector Multiplier Co-Processor	42
3.2	Section of Matrix Vector Multiplier Co-Processor routing input data from Microblaze into Floating Point Adder and Multiplier	43

3.3 Internal Controller Block State Machine	44
3.4 Output FIFO controller State Machine	47
3.5 Microblaze vs Co-Processor data latency	50
3.6 Functional Simulation of Matrix Vector Multiplier Co-Processor	52
3.7 Full Hardware Divider Co-Processor Block Diagram	53
3.8 Internal Controller Block State Machine for Divider Co-Processor.....	54
3.9 Output FIFO Controller State Machine for Divider Co-Processor.....	55
3.10 Functional Simulation of floating point divider Co-Processor	57
3.11 Using Matlab on PC to communicate with the FPGA board via RS232 [18]	58
3.12 Actual vs Theoretical Clk Count to Perform Matrix Vector Mult using Co-Processor for $m=3$ and varying n	59
3.13 Actual vs Theoretical Clk Count to Perform Matrix Vector Mult using Co-Processor for $n=32$ and varying m	60
4.1 Graph of Percentage of computation time per IP iteration spent in computing step 2 for various combinations of m_c and n_v	65
4.2 Demonstration of usage of Co-Processor to accelerate $y=Ax$ operation	66
4.3 Graphs of Speed up Factor for using Co-Processor to accelerate Matrix Vector Product in Step 2.....	68
4.4 Graphs of Speed up Factor for using Co-Processor to accelerate Matrix Vector Product in Step 3.....	69
4.5 Graph of speed up factor of variant over Microblaze standalone vs number of inequalities (m_c) with fixed $n_v=3$	72
4.6 Graph of speed up factor of variant over Microblaze firmware solution vs number of decision variables (n_v) with fixed $m_c=32$	73
4.7 Plot of average clock count per iteration for step 2 vs number of decision variables n_v (fixed $m_c=32$)	74
4.8 Plot of average clock count per iteration for step 2 vs number of inequalities m_c with ($n_v=3$).....	75
4.9 Graph of code size requirements vs m_c and n_v	77

5.1	Alternative C source codes for implementing matrix vector multiplication.....	78
5.2	Updated Graphs of computation time per QP problem vs mc and nv.....	79
5.3	Size of A when loop-unroll code size exceeds for-loop code size	81
5.4	Size of A when loop-unroll code size exceeds for-loop code size for Co-Processor	82
A.1	The EDK/ISE FPGA design flow[17]	89
B.1	In-lined C and Assembly source code for matrix vector multiplication	91
B.2	In-lined C and Assembly source code for dot-product function	96
C.1	C code for Step 2.....	97
D.1	VHDL source code for Matrix Vector Multiplier Co-Processor.....	99
D.2	VHDL source code for floating point divider Co-Processor	108

List of Tables

2.1	Xilinx Core Generator Core Types	9
2.2	Core Generator Floating Point Core Characteristics	10
2.3	Assembly Instructions corresponding to C code of Figure 2.3 with USE_COPROCESSOR not defined	12
2.4	Assembly Instructions corresponding to C code of Figure 2.3 with USE_COPROCESSOR defined.....	13
2.5	Comparison between Microblaze and Co-Processor for a floating point operation.	13
2.6	Assembly Instructions corresponding to C code in Figure 2.4 with USE_COPROCESSOR not defined	15
2.7	Assembly Instructions corresponding to C code in Figure 2.4 with USE_COPROCESSOR defined.....	16
2.8	Performance and code size comparison on using a Co-Processor with Microblaze to implement N floating point operations	19
2.9	Pipeline stages to calculate floating point operation using Co-Processor	20
2.10	Assembly code to corresponding to C code in Figure 2.8	23
2.11	Clock cycles required to perform $y = Ax$ using Co-Processor	29
2.12	Total clock count required to calculate one Matrix Vector Product using for-loop method.....	31
2.13	Assembly code corresponding to C code in Figure 2.13	32
2.14	Assembly Instructions corresponding to C code in Figure 2.14.....	34
2.15	Clock cycle count and number of instructions required for C code in Figure 2.15.	35
2.16	Speed and code size characteristics table of various implementation styles of matrix vector multiplication	38
3.1	FSL Slave interface from Co-Processor	40
3.2	FSL master interface from Co-Processor.....	41
4.1	Size of Matrix Vector Multiplications calculated by Co-Processor for each iteration of the Interior Point Method	68

4.2 Divisions per iteration for the IPM	70
4.3 Resource Usage.....	76

Abstract

With the dawn of reconfigurable computing, the FPGA has increasingly replaced microcontrollers as a viable alternative in the arena of embedded applications especially in DSP or high performance control applications. FPGA provides engineers the flexibility to employ significant architectural speed-up using techniques such as pipelining and parallel processing, normally not possible in off-the-shelf processors. Such customized hardware solutions however, hold the primary drawback of long development cycles.

The emergence of re-configurable soft-core processors on FPGA during the past decade has changed this reality. Embedded engineers now have an additional option to build a firmware solution running on a reconfigurable microprocessor on FPGA. In addition, they also enjoy the added flexibility of using the remaining resources on the FPGA to build customized hardware-based Co-Processor or accelerators to further enhance the performance the microprocessor. A tightly coupled Co-Processor in co-existence with carefully coded firmware running on a microprocessor could effect in an improvement in performance, whilst maintaining a relatively shorter development life cycle.

In this dissertation, the types of operation that could be accelerated using a Co-Processor were investigated. It was concluded that a key bottleneck lies in time spent in data transfer between the soft-core processor and the Co-Processor. To benefit from using a Co-Processor, the tasks to be migrated from the soft-core processor to the Co-Processor must either allow an opportunity to the reuse of data in the Co-Processor, or have a relatively longer computation time.

The insight obtained was then applied to accelerate the Interior Point Method which was first implemented on the Microblaze, a 32 bit softcore processor from Xilinx. The Interior Point Method is commonly used to solve constrained Model Predictive Control problems.

Two Co-Processors were then built to accelerate the Interior Point Method. A maximum speed up factor of 2.2 times was achieved over Microblaze standalone. This comes very close to the performance of the sequential version of the hardware implementation described in an earlier work which this thesis is founded on.

Chapter 1 Introduction

1.1 Motivation

The number of gates in Field Programmable Gate Array (FPGA) chips has been increasing steadily over the years from tens of thousands of gates to millions of gates. This trend has expanded the usage of FPGAs from mere glue logic to highly specialized re-configurable computing machines. Due to the ample resources available in today's FPGAs, these computing machines can even exist as a combination of microprocessor cores and reconfigurable logic to realize a tightly integrated system-on-a-chip (SOC) [2].

A case in point for embedded applications which has high computational demands is Model Predictive Control (MPC) which has become an established control technology due to its ability in solving problems with physical constraints. Its early use was primarily in the petrochemical industry, but recent years has seen an increase in its usage in various applications ranging from ships [3, 4], aerospace [5, 6, 7], road vehicles [8, 9], semiconductor manufacturing and micro scale devices such as the MEMS (Micro Electro Mechanical Systems) [11]. This trend has led to an increasing need for platforms able to deploy MPC for embedded applications requiring a fast response time.

Realizing specialized hardware systems on FPGA typically requires long development cycles and huge development effort when using Register-Transfer-Level (RTL) languages such as VHDL or Verilog. As a result, other higher level programming languages such as Handel C [10] have emerged in the recent years. Alternatively, one could implement a softcore processor [2, 15] on FPGA. With this approach, development engineers enjoy the added flexibility of choosing between partial or full implementation of the entire system as firmware residing on a microprocessor on FPGA and/or use the remaining resources on the FPGA to realize a customized hardware Co-Processor which can further accelerate the more timing critical portions of the system [10].

The motivation of the thesis is to investigate the realization of MPC algorithm on a FPGA and to determine its performance bottleneck. At each sampling time, the MPC solves a Quadratic Programming (QP) problem online to determine the required control actions. Interior Point and Active Set Methods are two commonly used techniques to solve QP problems. In this thesis, we first implemented the Interior Point Method on a softcore processor (Microblaze from Xilinx) and attempted to accelerate it using Co-Processors (hardware accelerators) which were also implemented on the same FPGA.

This thesis directly builds on the work described in [1], where 2 pure hardware implementations of the Interior Point Method (IPM) were realized using Handel C. In [1], it had identified that step 2¹ of the IPM required the most computation. In addition, [1] profiled and benchmarked the performance of the embedded implementation. The test bed used, the data and results obtained from these experiments were used to benchmark against the hybrid approach adopted in this thesis.

Similar work was also done in [12], where a Co-Processor FPGA platform was adopted to implement real time embedded MPC, extending ideas proposed in [13]. Here, the Co-Processor was used to perform matrix calculations, leaving the Microprocessor to perform the rest of the tasks. However, the work done in [12] involved the use of a different microprocessor (16 bit EISC from ADCUS). In addition, the Newton's iteration method [12] was adopted instead of the Interior Point Method. Also, the Co-Processor was attached to the Microprocessor as a peripheral which results in additional bus arbitration latencies.

Several other related works have also been published concerning the use of Co-Processors to do hardware acceleration on FPGA [10, 15]. In [10], the Co-Processor was used to accelerate iterative multiply and shift operations required in the Discrete Cosine Transform (DCT). Good performance was achieved by reducing the loading of data

¹ More details on Step 2 of the IPM is given in Chapter 4, page 63.

operands in the Co-Processor and parallelizing the operations. In [14], a matrix product accelerator architecture was introduced. The firmware running on the Microprocessor and the architecture of the Co-Processor were carefully designed to exploit an efficient communication protocol for data transfer between it and the host processor to achieve a significant reduction of computation time.

1.2 Major Contributions of the Thesis

In this thesis, a Microblaze firmware implementation of the Interior Point Method was first created and its performance analyzed to determine the bottleneck of the algorithm. Hardware Co-Processors were then built to complement the performance of Microblaze and accelerate the Interior Point Method. Finally, tests were conducted to determine the speed up achieved over the standalone Microblaze implementation.

To summarize, the major contributions of the thesis are as follows:

- 1) Implemented a firmware version of the Interior Point Method on Microblaze (on FPGA) and benchmarked its performance against existing implementations. In addition, further experiments were conducted to verify the bottleneck of the algorithm, previously identified in [1].
- 2) Investigated and identified operations which are suitable to be run on a Co-Processor. It was found that the main bottleneck lies in the data transfer between Microblaze and the Co-Processor. Thus, operations such as $y = Ax$ which gives opportunities to reuse data operands within the Co-Processor are suitable choices. We also discover that the for-loop coding style prefers a tall and thin matrix A (more rows than columns) for faster speed up. The loop-unrolling style however prefers a short and fat matrix A (more columns than rows) for faster speed up.
- 3) Proposed an architectural design and built Co-Processors for the selected operations

- 4) Tested and verified the speed up achieved using Co-Processor to carry out the selected operations
- 5) Applied the proposed Co-Processors to accelerate the Interior Point Method. A maximum speed up of 2.2 times over a firmware implementation running on the Microprocessor standalone was achieved.

We summarize the results obtained in this thesis in Figures 1.1 and 1.2.

Figure 1.1: Graph of computation time per QP problem vs number of inequalities (m_c) with number of decision variables fixed at $n_v = 3$

Figure 1.2: Graph of computation time per QP problem vs number of decision variables (n_v) with number of inequality constraints fixed at $m_c = 32$

It can be seen from Figures 1.1 and 1.2 that Microblaze took the longest computation time to solve QP problems of various sizes parameterized by n_v (the number of decision variables) and m_c (the number of inequality constraints). However, we were able to speed up computation time by up to 25% through the use of Co-Processors. Further speed up of 1.8 to 2.2 times was achievable using loop-unrolling technique. This speed up comes very close to the performance of the Handel C sequential implementation of [1]. This method unfortunately, requires a code size which increases linearly with m_c and n_v . The code size required to solve larger QP problems ($m_c=112, n_v=3$) cannot be accommodated due to the limitation of the 64 KB on-chip RAM available on our experimental platform. Details of the experimental setup and other simulation results will be described in the subsequent chapters.

1.3 Thesis Organization

The rest of the thesis is organized as follows:

In Chapter 2, we investigate the options available for accelerating Microblaze using a Co-Processor and determine the best suited type of operations to be run on the Co-Processor.

In Chapter 3, the design of the Co-Processors are described in detail, including the steps taken to verify the designed Co-processors.

In Chapter 4, we provide details on how the Co-Processors are used to accelerate the bottleneck portion of the Interior Point Method which is one of the commonly used methods for solving QP problems.

In the concluding chapter, conclusions and recommendations for further research are provided.

Chapter 2 Accelerating the Microblaze Processor

The focus of this chapter is to investigate the options available for accelerating Microblaze using a Co-Processor and also to determine the type of operations best suited to be run on a Co-Processor. For completeness and easy reference, the chapter will first provide an introduction to FPGA, Microblaze and the Co-Processor and the type of floating point cores to be used in the project. Further information on the software used can be found in Appendix A.

2.1 A Quick Introduction to FPGA

FPGA is an integrated circuit chip which contains the following type of cells [16]:

- a) Configurable logic blocks (CLBs) that are distributed evenly throughout the chip in a matrix form. Each of these CLBs can independently assume a wide range of functionalities.
- b) Input/Output logic blocks (IOBs) that are distributed at the boundaries of the FPGA. Each I/O block has several types of I/Os which can be configured and connected to the pins of the FPGA during configuration.
- c) Clock buffer cells
- d) Multipliers
- e) Dual Port Block RAMs

The CLBs is able to assume a wide set of functionalities. Each CLB contains 4 slices and each slice has two lookup tables (LUT). They are connected together via a matrix of wires and programmable switches. The functionality of each CLB is dependent on the values assumed in the LUTs. After the FPGA is being programmed with a bit file, this essentially closes the switches in the interconnect matrix. This array of logic and interconnect form the basis of how complex FPGA designs are created.

2.1.1 Microblaze as a Reconfigurable Processor

The Microblaze is a configurable soft core provided by Xilinx [17].

Figure 2.1: The Microblaze core [17]

The RISC core is a Harvard architecture and optimized especially for use on Xilinx FPGA. The Microblaze is highly configurable and allows user the flexibility of deciding the features required, based on target application and the size of the FPGA. Through the use of generics and parameters, one has the flexibility to select features needed and configure a customized microprocessor for the targeted application. The user thus has the flexibility to enable additional hardware in the core. Some examples include:

- a) A hardware integer multiplier
- b) A hardware integer divider
- c) Instruction cache
- d) Data cache
- e) Floating Point Unit (FPU)
- f) Hardware barrel shifter

The Microblaze v4.0 was used for our study.

2.1.2 The Hardware Co-Processor

A Hardware Co-Processor is an external application dependent hardware block, tailored to complement the performance of the Microblaze. Figure 2.2 illustrates how a Co-Processor is typically connected to Microblaze.

Figure 2.2: The Hardware Co-Processor

Microblaze simplifies the attachment of a tightly coupled Co-Processor in the following ways:

- 1) Availability of dedicated unidirectional links called the FSL (Fast Simplex Link) on Microblaze for uninterrupted communication with the Co-Processor. Eight such links are available on the Microblaze.
- 2) Availability of specific instructions in its instruction set for direct communication with Microblaze through the FSL bus. The instructions provided are both blocking and non-blocking types, to be used at the discretion of the embedded engineer.

2.1.3 Xilinx Core Generator

The Xilinx Core Generator is an Intellectual Property (IP) management tool which allows users to customize the available cores provided by Xilinx and tailor it for their specific application requirements. Some examples of the cores available are listed in Table 2.1:

Table 2.1: Xilinx Core Generator Core Types

Core types	Description
Memories	Block RAMs which are implemented by either using the LUTs on the FPGA, or the dedicated dual port block RAMs existing on the FPGA chip
Communication	Common low speed peripherals such as UART, SPI, IIC Common high speed peripherals such as Ethernet, Rocket I/O, PCI Express etc.
DSP cores	FFTs , DCTs, FIR filters
Math	Floating point cores, Cordic, Square Root etc.

2.1.4 Performance Characteristics of Xilinx Floating Point Cores

In this project, the Xilinx core generator software version 9.1i was used to generate various IP cores which will be used in the design of the Co-Processor. They include the floating point cores and the on-chip Block RAM peripherals. Xilinx core generator is able to generate synthesizable floating point cores which accept a new input every clock cycle and a latency of between 2 to 28 clock cycles, when running at a clock frequency of at least 100 MHz. Some of these floating point cores are listed below Table 2.2:

Table 2.2: Core Generator Floating Point Core Characteristics

Type	Latency (clk cycles)	Max synthesizable speed (MHz)	Flip flops utilized	LUTs utilized	Multipliers utilized
Multiplier	2	167	144	174	4
	3	181	146	177	4
	4	170	185	188	4
	5	232	228	187	4
	6	200	244	188	4
Adder	2	103	45	525	-
	3	126	115	519	-
	4	136	142	532	-
	6	151	252	531	-
Divider	14	100	621	794	-
	18	122	648	821	-
	20	120	656	832	-
	22	116	656	838	-
	28	164	1313	823	-

2.2 Options for accelerating Microblaze performance with a Co-Processor

In this section, we will investigate the following options possible for hardware acceleration.

1. Accelerate one floating point operation
2. Accelerate N floating point operations by means of a for-loop coding style
3. Accelerate N floating point operations by loop-unrolling
4. Using pipelining to accelerate N floating point operations using a Co-Processor.

The analysis was conducted in the following way:

1. Code the basic floating point operation in C
2. Compile and assemble the C code using version 3.4.1 of the Microblaze GNU compiler and assembler to generate an executable image. Since the focus of the study was to determine the overall speed up achieved when using a Co-Processor, the compilation optimization switches were disabled. This removes any possibility of speed up arising from the optimizing effects of the compiler.
3. Use the “mb-objdump” utility from Xilinx to obtain the generated assembly instructions
4. Determine the number of clock cycles and code size required for the particular floating point operation.

2.2.1 Accelerating one floating point operation

The following C code implements a floating point operation in Microblaze:

```
#include "float.h"
#define FLOP * //could also be + (for addition) or / (for division) operation

#ifdef USE_COPROCESSOR
#include "xparameters.h"
#include "mb_interface.h"
#include "fsl.h"
#define write_into_fsl(val,id) putfsl(val,id)
#define read_from_fsl(val,id) getfsl(val,id)
#endif

int main(){
    float a,b,c;
    #ifdef USE_COPROCESSOR
        write_into_fsl(a,0);
        write_into_fsl(b,0);
        read_from_fsl(c,0);
    #else
        c=a FLOP b;
    #endif
    return 1;
}
```

Figure 2.3: C source code to implement a floating point operation

When the Co-Processor is not used (ie. the flag USE_COPROCESSOR is not defined), the assembly instructions listed in Table 2.3 are generated.

Table 2.3: Assembly Instructions corresponding to C code of Figure 2.3 with USE_COPROCESSOR not defined

Row	Instruction	Description	Clock cycles
1	lwi r4,r19,4	Load operand 1 into register R4. ($R4 \leftarrow M[R19+4]^*$) ¹	2
2	lwi r3,r19,8	Load operand 2 into register R3. ($R3 \leftarrow M[R19+8]$)	2
3	fmul r3, r4, r3/ fadd r3, r4, r3/ fdiv r3, r4, r3/	Perform Floating point operation. ²	$M_{MB} = M_{add} \text{ or } M_{mul} \text{ or } M_{div}$
4	swi r3, r19, 20	Store word back to memory $M[R19+20] \leftarrow R3$	2
		Total clocks required	12

¹ Note that $Rn \leftarrow M[Rn+x]$ means load value obtained from memory address $Rn+x$ into Microblaze register Rn

² From the Microblaze Processor Reference Guide [19], $M_{add}=6$, $M_{mul}=6$, $M_{div}=30$

From Table 2.3, we can see that it takes 4 clock cycles to load the operands into the registers of Microblaze via immediate addressing, M_{MB} clock cycles to carry out the necessary floating point operation, and 2 clock cycles for storing back the results. A floating point addition or multiplication operation in C thus takes 12 clock cycles ($M_{add} = M_{mul} = 6$) to complete and requires 16 bytes of code space. For the case of floating point division ($M_{div} = 30$), it takes 36 clock cycles to compute.

Since basic floating point operations involve 2 input operands, we introduce a buffer in the Co-Processor for operand 1. Both the buffer and the FSL bus are directly connected to the inputs of the floating point core. During the transfer of the 2 operands from Microblaze into the Co-Processor, the first write of operand 1 is stored in the buffer; only after operand 2 is available to the Co-Processor will the floating point operation commence. Once the floating point core has completed computation of the operand, the result is immediately written to the return path of the FSL bus back to Microblaze.

The corresponding assembly instructions for the C code shown in Figure 2.3 (with the flag USE_COPROCESSOR defined) are tabulated in Table 2.4. 24 bytes of code is required to perform this operation:

Table 2.4: Assembly Instructions corresponding to C code in Figure 2.3 with USE_COPROCESSOR defined

Row	Instruction	Description	Clock cycles
1	lwi r3,r19,4	Load operand 1 into register R3. ($R3 \leftarrow M[R19+4]$)	2
2	put r3, rfs10	Load operand 1 onto FSL bus for device 0	2
3	lwi r3,r19,8	Load operand 2 into register R3. ($R3 \leftarrow M[R19+8]$)	2
4	put r3,rfs10	Load operand 2 onto FSL bus for device 0	2
5	-	Wait for coprocessor to calc result	M^1
6	get r3, rfs10	Reading back result into register R3	2
7	swi r3, r19, 12	Store word back to memory $M[R19+12] \leftarrow R3$	2
		Total	$12 + M$

¹M is the number of clock cycles required for the Co-Processor to compute result.

Table 2.5: Comparison between Microblaze and Co-Processor for a floating point operation

Operation type	Clock cycles required by floating point operation	
	Microblaze with hardware FPU	Microblaze and Co-Processor
Multiply	$6 + M_{mul} = 12$	$12 + M$
Add	$6 + M_{add} = 12$	$12 + M$
Divide	$6 + M_{div} = 36$	$12 + M$

From Table 2.5, we can see that for the case of floating point multiplication and addition, the use of a Co-Processor to perform a single floating point operation takes more clock cycles since $M > 0$. There is thus no performance speed up advantage in using a Co-Processor to speed up these operations.

For the case of floating point division however, using a Co-Processor will result in a speed up. A floating point division takes 36 clock cycles on Microblaze and $12+M$ clock cycles on a Co-Processor. A speed up can thus be achieved when a floating point divider of less than 24 clock cycles is used.

2.2.2 Accelerating N floating point operations

Next we investigate the feasibility of using a Co-Processor to accelerate N floating point operations. We first consider the case of using a pure firmware solution running on Microblaze to compute N floating point operations. The corresponding C code is as shown in Figure 2.4 which translates into the assembly code as shown in Table 2.6.

```
#include "float.h"
#define N 100
#define FLOP * //could also be + or /

#ifdef USE_COPROCESSOR
#include "xparameters.h"
#include "mb_interface.h"
#include "fsl.h"
#define write_into_fsl(val,id) putfsl(val,id)
#define read_from_fsl(val,id) getfsl(val,id)
#endif

int main(){
    float op1[N], op2[N], result[N];
    int i;
    /* some initialization code for the array op1 and op2*/
    ...
    ...
    #ifdef USE_COPROCESSOR
        for (i=0;i<N;i++){
            write_into_fsl(op1[i],0);
            write_into_fsl(op2[i],0);
            read_from_fsl(result[i],0);
        }
    #else
        for (i=0;i<N;i++){
            result[i] = op1[i] FLOP op2[i];
        }
    #endif
    return 1;
}
```

Figure 2.4: C source code to do N floating point operations by for-loop

Table 2.6: Assembly Instructions corresponding to C code in Figure 2.4 with USE_COPROCESSOR not defined

Row	Instruction	Operation	Description	Clock Cycles
1	swi r0, r19, 40	$M[R19+40] \leftarrow r0$	Initialize i to 0	2
2	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Determine if value of index i has already exceeded the for-loop limit; jump out of the loop if yes.	2
3	addik r18, r0, 1	$R18 \leftarrow 1$		1
4	Cmp r18, r3, r18			1
5	blti r18, 92	Branch if less than		1 (last time is 3)
6	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Re-read i into R3.	2
7	muli r4, r3, 4	$R4 \leftarrow R3 \times 4$	Calculate address of result[i]	3
8	addik r3, r19, 4	$R3 \leftarrow R19+4$		1
9	addk r3, r4, r3	$R3 \leftarrow R3+R4$		1
10	addk r6, r3, 24	$R6 \leftarrow R3+24$		1
11	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Read index i and store calculated address of op[i] into register R5	2
12	muli r4, r3, 4	$R4 \leftarrow R3 \times 4$		3
13	addik r3, r19, 4	$R3 \leftarrow R19+4$		1
14	addk r5, r4, r3	$R5 \leftarrow R3+R4$		1
15	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Re-read index i and store calculated address of op2[i] into register R3	2
16	muli r4, r3, 4	$R4 \leftarrow R3 \times 4$		3
17	addik r3, r19, 4	$R3 \leftarrow R19+4$		1
18	addk r3, r4, r3	$R3 \leftarrow R3+R4$		1
19	addik r3, r3, 12	$R3 \leftarrow R3+12$		1
20	lwi r4, r5, 0	$R4 \leftarrow M[R5]$	Load op1[i] into R4	2
21	lwi r3, r3, 0	$R3 \leftarrow M[R3]$	Load op2[i] into R3	2
22	fmul r3, r4, r3	$R3 \leftarrow R3 \times R4$ (float)	Float multiply, store result in R3. This could be floating point addition or divide as well, depending on the C code.	M_{MB}
23	swi r3, r6, 0	$M[R6+0] \leftarrow R3$	Store multiply result	2
24	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Increment and update index i used in the for-loop	2
25	addik r3, r3, 1	$R3 \leftarrow R3+1$		1
26	swi r3, r19, 40	$M[R19+40] \leftarrow R3$		2
27	bri -100	branch	Branch back to start of loop (2 nd row in this table)	3
	Total			$9 + (42 + M_{MB})N$

From Table 2.6, we can see that it takes $9 + (42 + M_{MB})N$ clock cycles to compute N additions, multiplications or divisions. Firstly, two clock cycles are needed to initialize the loop counter (row 2). Each for-loop iteration requires 48 clock cycles (from row 2 to 27). Finally it takes 7 cycles (rows 2 to 5) to determine that the terminal count of the for-loop has been reached, and so to branch out of the for-loop. Since there are 27 lines of instructions generated by this for-loop, the code size for this portion of code is $27 \times 4 = 108$ bytes. (Each instruction on Microblaze has a length of 4 bytes).

When USE_COPROCESSOR is set to 1, this translates into the following assembly code as shown in Table 2.7.

Table 2.7: Assembly Instructions corresponding to C code in Figure 3.4 with USE_COPROCESSOR defined

Row	Instruction	Operation	Description	Clock Cycles
1	swi r0, r19, 40	$M[R19+40] \leftarrow r0$	Initialize i to 0	2
2	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Determine if value of index i	2
3	addik r18, r0, 1	$R18 \leftarrow 1$	has already exceeded the for-	1
4	cmp r18, r3, r18		loop limit; jump out of the	1
5	blti r18, 92	Branch if less than	loop if yes.	1 (last time is 3)
6	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Re-read index i and calculate	2
7	muli r4, r3, 4	$R4 \leftarrow R3 \times 4$	the address of op1[i] which is	3
8	addik r3, r19, 4	$R3 \leftarrow R19+4$	stored into register R3	1
9	addk r3, r4, r3	$R3 \leftarrow R3+R4$		1
10	lwi r3, r3, 0	$R3 \leftarrow M[R3]$	Obtain the value of op1[i] and	2
11	put r3, rfs10		load it into Co-Processor	2
12	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Re-read index i and calculate	2
13	muli r4, r3, 4	$R4 \leftarrow R3 \times 4$	the address of op2[i] which is	3
14	addik r3, r19, 4	$R3 \leftarrow R19+4$	stored into register R3	1
15	addk r3, r4, r3	$R3 \leftarrow R3+R4$		1
16	addik r3, r3, 12	$R3 \leftarrow R3+12$		1
17	lwi r3, r3, 0	$R3 \leftarrow M[R3]$	Load value of op2[i] into R3	2
18	put r3, rfs10		Load op2[i] into Co-Processor	2
			Wait for Co-Processor to	M
			calculate result	
19	get r5, rfs10		Get result from Co-Processor	2
20	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Re-read index i and calculate	2
21	muli r4, r3, 4	$R4 \leftarrow R3 \times 4$	the address of result[i]. Finally	3
22	addik r3, r19, 4	$R3 \leftarrow R19+4$	store result read back from	1
23	addk r3, r4, r3	$R3 \leftarrow R3+R4$	Co-Processor into result[i].	1
24	addik r4, r3, 24	$R4 \leftarrow R3+24$		1
25	addk r3, r5, r0	$R3 \leftarrow R5$		1
26	Swi r3, r4, 0	$M[R4] \leftarrow R3$		2
27	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Increment index i	2
28	addik r3, r3, 1	$R3 \leftarrow R3+1$		1
29	swi r3, r19, 40	$M[R19+40] \leftarrow R3$		2
30	bri -112		Branch back to top of for loop	3
	Total			$9 + (49+M)N$

From Table 2.7, we can see that it takes $9 + (49+M)N$ clock cycles to compute N additions, multiplications or divisions. Firstly, two clock cycles are needed to initialize the loop counter (row 2). Each for-loop iteration requires an overhead of $49+M$ clock cycles (from row 2 to 31). Finally it takes 7 cycles (rows 2 to 5) to determine that the terminal count of the for-loop has been reached, and so to branch out of the for-loop.

Since it takes 30 instructions to implement the for-loop for use on a Co-Processor, this section of code requires 120 bytes of code space.

For speed up to occur

$$9 + (49+M)N < 9+(42+M_{MB})N \Rightarrow M < M_{MB} - 7$$

For the case of floating point multiplication and addition, $M_{MB} = 6$, hence speed up is not possible. For the case of floating point division, $M_{MB} = 30$, thus a speed up can be achieved if $M < 23$.

2.2.3 Accelerating N floating point operations using loop-unrolling

A commonly-used technique to speed up code execution time is loop-unrolling. This, however, comes at the expense of code size increasing linearly as the number of operations. We refer to the following C codes in Figure 2.5:

```

#include "float.h"
#define N 100
#define FLOP * //could also be + or /
#ifdef USE_COPROCESSOR
#include "xparameters.h"
#include "mb_interface.h"
#include "fsl.h"
#define write_into_fsl(val,id) putfsl(val,id)
#define read_from_fsl(val,id) getfsl(val,id)
#endif

int main(){
    float op1[N], op2[N], result[N];
    /* some section of C code for initialization */
#ifdef USE_COPROCESSOR
    result[0] = op1[0] FLOP op2[0];
    result[1] = op1[1] FLOP op2[1];
    ...
    result[N-1] = op1[N-1] FLOP op2[N-1];
#else
    write_into_fsl(op1[0],0);
    write_into_fsl(op2[0],0);
    read_from_fsl(result[0],0);
    write_into_fsl(op1[1],0);
    write_into_fsl(op2[1],0);
    read_from_fsl(result[1],0);
    ...
    write_into_fsl(op1[N-1],0); //for the nth floating point calculation on a Co-Processor
    write_into_fsl(op2[N-1],0);
    read_from_fsl(result[N-1],0);
#endif
    return 1;
}

```

Figure 2.5: C source code to implement N floating point operations by loop-unrolling

When the Co-Processor is not used (ie. USE_COPROCESSOR is not defined), the assembly instructions generated are N times of row 1 to 4 previously shown in Table 2.3. Thus, it takes $(6+M_{MB})N$ clock cycles to calculate N floating point additions/multiplications/divisions.

Although loop unrolled code has a much shorter execution time, its size grows linearly with the number of floating point operations required. N floating point operations require 4N instructions (from Table 2.3); this translates to 16N bytes of code size.

When a Co-Processor is used (USE_COPROCESSOR is defined), the corresponding assembly instructions generated are N times the instructions of row 1 to 7 in Table 2.4. Thus it takes $(12+M)N$ clock cycles to compute N floating point operations on Microblaze with Co-Processor using the loop-unrolled coding style. The code size grows linearly at a rate of 6 instructions or 24 bytes of code for every additional floating point operation required. We now summarize the results obtained so far in Table 2.8:

Table 2.8: Performance and code size comparison on using a Co-Processor with Microblaze to implement N floating point operations

		Software on Microblaze (a)		Software on Microblaze + Co-Processor (b)	
		Clock Cycle	Code Size	Clock Cycle	Code Size
Add or multiply or divide	For-loop	$9 + (42 + M_{MB})N$	108	$9 + (49 + M)N$	120
	Loop-unroll	$(6 + M_{MB})N$	16N	$(12 + M)N$	24N

From Table 2.8, we can conclude the following:

- 1) It is not possible to achieve any speed up when using the above Co-Processor design with Microblaze for computing N floating multiply and add operations since $M_{MB}=6$ for both multiplication and addition operations.
- 2) Some speed up is achievable in the case of performing N floating point divisions (where $M_{MB}=30$). Maximum speed up when using a Co-Processor is achieved when the pipeline depth of the floating point division core is the smallest ($M=14$). Thus the speed up factor can be calculated with $M_{MB}=30$ and $M=14$ as follows:

For the case of for-loop implementation:

$$\text{Speed up} = \frac{9 + (42 + M_{MB})N}{9 + (49 + M)N} = \frac{9 + 72N}{9 + 63N} \approx \frac{72N}{63N} = 1.14$$

For the case of loop-unroll implementation:

$$\text{Speed up} = \frac{(6 + M_{MB})N}{(12 + M)N} = \frac{36N}{26N} = \frac{18}{13} = 1.38$$

- 3) For both the for-loop and loop-unrolled cases, the code size increases by 11% and 50% respectively for the case when Co-Processor is used.

2.2.4 Pipelining to accelerate N floating point operations using Co-Processor

Hardware techniques such as pipelining could be employed to enhance speed-up performance when using a Co-Processor. Pipelining generally breaks a given task into smaller sub-tasks or stages, executing them like an assembly line [19]. The first stage accepts the inputs, and passes the intermediate results to the next stage which stores the result in its internal register. This happens till the last stage which outputs the result. Each pipeline stage ideally should have equal cycle time (or granularity) for optimum performance.

Since there are two independent execution units (ie. The Microblaze and the Co-Processor), a two stage pipeline can be designed as shown below in Table 2.9. For an optimum pipeline, each stage should be occupied throughout the cycle time and this is achievable only when all the pipeline stages have equal cycle time [19]. Since pipeline stage 1 takes 8 clock cycles to execute at 100 MHz, the Co-Processor in pipeline stage 2 should optimally take 8 clock cycles at 100 MHz as well to compute and return the result.

Table 2.9: Pipeline stages to calculate floating point operation using Co-Processor

Pipeline stage no.	Pipeline stage description	Executed on	Latency (Clock Cycles)
1	Loading of operands into Co-Processor	Microblaze	8
2	Floating point operation on Co-Processor	Co-Processor	M

Consider the case when $M \leq 8$. In order to achieve maximum throughput, we leave the reading of results from the Co-Processor right at the end, after all the calculations have been completed on the Co-Processor (See Figure 2.6). This of course imposes an

additional requirement on the Co-Processor that an internal buffer will be needed to store all the floating point results.

Figure 2.6: Pipelining floating point multiplications/additions when using a Co-Processor ($M \leq 8$)

Referring to Table 2.2, we see that Xilinx Core Generator is able to generate floating point multiplier and adder cores of latency ranging from 2 to 6 clock cycles. However, we can also see that more resources are used up when the latency of these cores are increased. In view of this, we select the latency of the multiplier and adder cores to be 4 clock cycles ($M=4$). When using this latency setting, we could in principle clock the Co-Processor at half the frequency (50MHz) of the Microprocessor (100MHz) and still achieve equal cycle time across the pipeline stages without compromising throughput. This however, increases the complexity of the design as additional circuitry is required to handle data transfer between the 2 clock domains. For simplicity of the design, we thus clock the Co-Processor at the same speed as the Microprocessor (at 100 MHz), and leave the latency of the core at 4 clock cycles.

For the case when $M > 8$, Figure 2.7 shows the timing diagram for N operations. The condition for maximum throughput of the pipeline is

$$\begin{aligned} X_2 &\geq X_1 \\ 8N + 4(N-1) &\geq 8 + M + 8(N-1) \\ \Rightarrow 4(N-1) &\geq M \end{aligned}$$

The case when $M > 8$ is relevant when floating point division is implemented in the Co-Processor. Given a minimum of latency of $M=14$ for a floating point division, $N \geq 5$.

Figure 2.7: Pipelining floating point divisions when using a Co-Processor

We now analyze the performance of using the pipeline design. Figure 2.8 shows the for-loop coding style for the pipeline design. The corresponding assembly code is shown in Table 2.10, which shows that there are 39 assembly instructions generated, hence requires 156 bytes of code space.

```
#include "xparameters.h"
#include "mb_interface.h"
#include "fsl.h"
#define write_into_fsl(val, id) putfsl(val, id)
#define read_from_fsl(val, id) getfsl(val, id)
#define N 100
int main(){
    float op1[N], op2[N], result[N];
    int i;
    /* some section of C code for initialization */
    ...
    ...
    for (i=0;i<N;i++){ //start loading then unloading floating point operands to/from Co-Processor
        write_into_fsl(op1[i],0);
        write_into_fsl(op2[i],0);
    }
    for (i=0;i<N;i++){
        read_from_fsl(result[i],0);
    }
    return 1;
}
```

Figure 2.8: For-loop C coding style to compute floating point operations using Co-Processor with buffer

Table 2.10: Assembly code to corresponding to C code in Figure 2.8

Row	Instruction	Operation	Description	Clock Cycles
1	swi r0, r19, 40	$M[R19+40] \leftarrow r0$	Initialize i to 0	2
2	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Read out i into R3	2
3	addik r18, r0, 1	$R18 \leftarrow 1$	Determine if value of index i has already exceeded the for-loop limit; jump out of the loop if yes. (next instruction @ after last instruction in this table).	1
4	cmp r18, r3, r18			1
5	blti r18, 72	Branch if less than		1 (last time is 3)
6	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Re-read index i and calculate address of op1[i]	2
7	muli r4, r3, 4	$R4 \leftarrow R3 \times 4$		3
8	addik r3, r19, 4	$R3 \leftarrow R19+4$		1
9	addk r3, r4, r3	$R3 \leftarrow R3+R4$		1
10	lwi r3, r3, 0	$R3 \leftarrow M[R3]$	Load op1[i] into Co-Processor	2
11	put r3, rfs10			2
12	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Re-read index i and calculate address of op2[i]	2
13	muli r4, r3, 4	$R4 \leftarrow R3 \times 4$		3
14	addik r3, r19, 4	$R3 \leftarrow R19+4$		1
15	addk r3, r4, r3	$R3 \leftarrow R3+R4$		1
16	addik r3, r3, 12	$R3 \leftarrow R3+12$		1
17	lwi r3, r3, 0	$R3 \leftarrow M[R3]$	Load op2[i] into Co-Processor	2
18	put r3, rfs10			2
19	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Increment i by 1	2
20	addik r3, r3, 1	$R3 \leftarrow R3+1$		1
21	swi r3, r19, 40	$M[R19+40] \leftarrow R3$		2
22	bri -80		Branch back to top of for first loop	3
23	swi r0, r19, 40	$M[R19+40] \leftarrow r0$	Initialize i to 0. start of 2 nd loop to read back result	2
24	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Determine if value of index i has already exceeded the for-loop limit; jump out of the loop if yes. (next instruction @ after last instruction in this table).	2
25	addik r18, r0, 1	$R18 \leftarrow 1$		1
26	cmp r18, r3, r18			1
27	blti r18, 52	Branch if less than		1 (last time is 3)
28	get r5, rfs10		Read back result from Co-Processor	2
29	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Re-read index i and calculate address of result[i]. Subsequently store the result obtained from Co-Processor into result[i].	2
30	muli r4, r3, 4	$R4 \leftarrow R3 \times 4$		3
31	addik r3, r19, 4	$R3 \leftarrow R19+4$		1
32	addk r3, r4, r3	$R3 \leftarrow R3+R4$		1
33	addik r4, r3, 24	$R4 \leftarrow R3+24$		1
34	addk r3, r5, r0	$R3 \leftarrow R5+R0$		1
35	swi r3, r4, 0	$M[R4] \leftarrow R3$		2
36	lwi r3, r19, 40	$R3 \leftarrow M[R19+40]$	Increment i by 1.	2
37	addik r3, r3, 1	$R3 \leftarrow R3+1$		1
38	swi r3, r19, 40	$M[R19+40] \leftarrow R3$		2
39	bri -60		Branch back to top of for second loop	3
	Total clock count			18+62N clock cycles

A detailed analysis of the assembly code shown in Table 2.10 produces the timing diagram as shown in Figure 2.9. It describes the time taken to compute 4 floating point multiplications/additions using the for-loop approach based firmware.

Figure 2.9: Clock cycles required for Co-Processor to do 4 floating point operations back-to-back (for-loop based firmware)

In Figure 2.9, block A refers to row 1 of Table 2.10 which sets the index i of the first for loop to 0. This is followed by block B, which loads the operands into the Co-Processor as described in row 2 to 18 of Table 2.10. At this point, as described in block C, the Co-Processor is able to start computing the floating point operation. This occurs in parallel with block D, whereby Microblaze executes the rest of the iteration in the first for-loop, as seen from rows 19 to 22 of Table 2.10. Following this, a branch back to the start of the first for-loop occurs and execution of code referred to by block B occurs. The whole sequence is then repeated N times, where N is the number of floating point operations to be calculated.

The first for-loop terminates at block E whereby the last condition check of index i shows that terminal count has reached. This starts the second for-loop (rows 24 to 39 of Table 2.10) which iterates repeatedly until all the N results are completed. The second for-loop

terminates when block H (rows 24 to 27 of row 13) is completed, which describes the last condition check of the index i in the second for-loop when terminal count is reached.

Both blocks B and D of the code are grouped together as stage 1 of the pipeline, whilst block C is stage 2 of the pipeline described earlier on in Table 2.10. We can see that the granularity of the pipeline stages is not balanced in this case. It takes considerably much longer to load in the operands, than to compute the floating point result in the Co-Processor (floating point multiplication/addition core computes results in 4 clock cycles; floating point division core computes results in 14 clock cycles). Further improvements can be achieved by shortening the cycle time for pipeline stage 1 using techniques such as loop-unrolling in firmware running on Microblaze.

Next, we determine the speed up of using a Co-Processor over Microblaze standalone to compute N floating point operations. A Co-Processor takes a shorter time than Microblaze standalone to compute a floating point division. From Tables 2.8 (with $M_{MB}=30$) and 2.10, we see that the speed up achieved over Microblaze standalone is:

$$\text{Speed up for floating point division} = \frac{72N + 9}{62N + 18} \approx \frac{72N}{62N} = 1.16$$

However, it takes only $9+48N$ clocks for Microblaze standalone to compute N floating point additions/multiplications using the for-loop approach (since $M_{MB}=6$). Thus no speed up is possible in using the Co-Processor to compute these operations.

Finally we also revisit the method of software loop-unrolling with this pipelined styled architecture. Figure 2.6 depicts the computation time required to compute 4 floating point multiplications/additions back-to-back. We can see that a reduction in cycle time for stage 1 of the pipeline due to loop-unrolling results in a better performance. However, as mentioned earlier, the code size grows linearly with the number of floating point operations required to be computed by the Co-Processor.

We can thus conclude that in general, it takes $12N$ clock cycles for the Co-Processor to calculate N multiplications or additions back-to-back. This is the same amount of time required by Microblaze standalone with loop-unrolled based firmware to compute the same operations. This can largely attributed to the amount of loading time required to load the operands into the Co-Processor. There is thus no speed up advantage of using a Co-Processor to compute this type of operations.

Similarly, we revisit Figure 2.7, which shows 5 floating point divisions are performed back-to-back using a Co-Processor. We achieve a 3x speed up factor for this configuration since the loop-unrolled type of firmware running on Microblaze standalone takes $36 \times 5 = 180$ clocks to calculate 5 floating divisions.

In general, N divisions performed back-to-back using a Co-Processor with loop-unrolled firmware requires $12N$ clock cycles to complete. Since it takes $36N$ cycles for Microblaze standalone to perform the same calculations, the speed up achieved is 3 times. This is the optimum speed-up as all latencies arise from purely data transfer and none from the Co-Processor. Any further improvements are possible only with an architectural improvement in Microblaze.

In summary, the floating point division is the only basic floating point operation that could benefit from having a Co-Processor. The maximum speed up achieved is when using the loop unrolled approach; it yields a speed up of 3 times over the pure firmware, loop-unrolled approach on Microblaze standalone. This comes with a price of a linear increase of code size of 24 bytes for every additional floating point division. This translates to 1.5 times the code size required for the loop-unrolled approach on Microblaze standalone (it requires 36 bytes of code for every additional floating point division required).

For applications with stringent on-chip Microprocessor memory requirements, the for-loop approach should also be used together with the Co-Processor to compute N floating

point divisions. This yields only a 16% speed up in comparison to a standalone Microblaze.

2.3 Accelerating Co-Processor performance by reducing data transfer overhead

As observed in the last section, there is substantial overhead in data transfer between Microblaze and the Co-Processor. In other words, to benefit from the use of Co-Processor, the data transfer overhead has to be reduced.

One such operation is the matrix vector multiplication operation² $y=Ax$ where A is a m -by- n matrix. Its general formula can be defined as follows:

$$y_i = \sum_{j=1}^n a_{ij} x_j \quad \text{where } i=1,2,\dots,m$$

From the above formula, it can be seen that the 2^{nd} operand x_j is repeatedly needed for computation of each y_i . Thus, $y=Ax$ operations can be re-arranged as follows:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_m \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n} \\ a_{31} & a_{33} & a_{33} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & a_{m3} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ a_{31} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ a_{32} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ a_{3n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

² More details on the operations involving Step 2 and that it is the main bottleneck of the IPM is mentioned in Section 4.1 on page 62 and in Section 4.2.2 on page 64 respectively. In addition, rows 1 to 3 of Table 4.1 on page 68 shows the number of times in which the $y=Ax$ operation is used in computing Step 2 of the IPM. The $y=Ax$ operation is thus crucial in the solution of linear equations central to Step 2.

This suggests that we load in the operand x one element at a time followed by one column of A at a time. For example, if A is a 3-by-2 matrix and x is a 2-by-1 vector, we load the operands in the following way:

- a. load x_1
- b. load a_{11}
- c. load a_{21}
- d. load a_{31}
- e. load x_2
- f. load a_{12}
- g. load a_{22}
- h. load a_{32}
- i. read back result y_{11}
- j. read back result y_{21}
- k. read back result y_{31}

This saves $(m-1) \times (n)$ times of loading x_j into the Co-Processor. This however comes at the expense of an additional n -by-1 input buffer in the Co-Processor for storing operand x .

If the operands were loaded into the Co-Processor using the loop-unrolled approach, this can be translated to the following timing diagram as shown in Figure 2.10. We obtain two advantages by loading the operands in this manner:

- 1) reduced overhead in loading the operands
- 2) Calculation of $y = Ax$ operation occurs in parallel with operand loads, and results are immediately available once the last operand has been loaded into the Co-Processor. Hence there is no additional clock cycles spent in waiting for results from Co-Processor.

Any further improvements in performance will require an architectural improvement in Microblaze to improve operand loading time into/from FSL bus.

Figure 2.10: Using the Co-Processor to perform computation of $y=Ax$. (Note that floating point multipliers and adders with latency of 4 clock cycles were used. This is to ensure equal cycle time across the various pipeline stages).

From Figure 2.10 and rows 1,2,6,7 in Table 2.4, we are able to calculate, for the general case, the clock cycles and the number of instructions required to compute the $y=Ax$ operation as shown in Table 2.11. Since each instruction is 4 bytes, the total code size required is thus $2(n+mn+m)*4 = 8(n + mn + m)$ bytes.

Table 2.11: Clock cycles required to perform $y=Ax$ using Co-Processor

Operation	Clock cycles required	Number of assembly instructions required:
Loading of operand x (size n -by-1)	$4n$	$2n$
Loading of operand A (size m -by- n)	$4mn$	$2mn$
Reading back results (size m -by-1)	$4m$	$2m$
Computation time required for $y=Ax$	0	0
Total	$4(n + mn + m)$	$2(n + mn + m)$

In contrast, when using the Microblaze, we will not be able to exploit the benefits derived from reduction of operand loads. Thus, for the pure firmware implementation on Microblaze, we coded the $y=Ax$ operation in C as shown in Figure 2.11³

³ A comment on the clock cycle overhead incurred in calling the “dotprod” function is given in Chapter 5 page 78.

```

/* snippet of C code in main function */
do {
    result[i] = dotprod(col_size,&op1[i*col_size],&op2[0],1,1);
    i++;
} while (i!=row_size);

```

Figure 2.11: C source codes for implementing matrix vector multiplication

As we can see from the above snippet of C code, the while loop calls function dotprod iteratively to implement matrix vector multiplication. The dotprod C code is as follows:

```

float dotprod(Xuint32 n,float *a,float *b, Xuint32 inc1, Xuint32 inc2) {
    float adotb,sum, outrow;
    Xuint32 i,j,q;
    sum = 0;
    q = n*inc1;
    j = inc2;
    adotb = a[0]*b[0];
    i = inc1;

    while (i<q){
        sum = sum + adotb;
        adotb = a[i]*b[j];
        j+=inc2;
        i+=inc1;
    }
    outrow = sum + adotb;
    return outrow;
}

```

Figure 2.12: C source codes for implementing dot product multiplication

The assembly codes corresponding to Figures 2.11 and 2.12 can be found in Appendix B. Using the same method of analyzing the assembly code, we then calculate the actual number of clock cycles required by the function to perform each stage of the matrix-vector product.

The total clock count for computing a matrix vector multiplication of a matrix of size m-by-n and vector of n-by-1 is :

$$\text{total clock count} = m \times [46 + (56 + 64 \times (n - 1) + 20)] = m[122 + 64(n - 1)]$$

In addition, the code size of corresponding firmware is 91 instructions or 364 bytes (see Table 2.12).

Table 2.12: Total clock count required to calculate one Matrix Vector Product using for-loop method

Portion of code	Number of times per matrix vector multiply operation	Clock cycle Per iteration	Total clock cycles required.
Matrix vector multiplication while loop	m	46 + time to calc one dotprod	m (46 + time to calc one dotprod)
Start of dotprod function before while loop	m	56	56m
While loop in dotprod function	$M(n - 1)$	64	$64m(n - 1)$
Portion of code after while loop till dotprod function branches back to caller function.	m	20	20m
		Total	$m(122+64(n-1))$

Where m= row size of matrix A, n=column size of matrix A

Next, we also investigate the computation time and code size required to compute a matrix vector multiplication on Co-Processor. Since we are using a Co-Processor, the C code implementation is designed to reduce overall computation time by reducing operand loads into the Co-Processor as shown earlier in Figure 2.9.

```

#include "float.h"
#include "xparameters.h"
#include "mb_interface.h"
#include "fsl.h"

#define write_into_fsl(val, id) putfsl(val, id)
#define read_from_fsl(val, id) getfsl(val, id)
#define M 3 /* number of rows */
#define N 2 /* number of columns */

int main(){
    float a[M][N], x[N], result[M];
    int i,k,colsize,rowsize;
    rowsize = 3;
    colsize = 2;

    /* C code for loading the Co-Processor */
    for (k=0;k<N;k++){
        write_into_fsl(x[k],0);
        for (i=0;i<M;i++){
            write_into_fsl(a[i][k],0);
        }
    }
    for (k=0;k<M;k++){
        read_from_fsl(result[k],0);
    }
    return 1;
}

```

Figure 2.13: For-loop C coding style to compute Matrix Vector Multiplication using Co-Processor

The corresponding assembly instructions generated are shown in Table 2.13

Table 2.13: Assembly code corresponding to C code in Figure 2.13

Row	Instruction	Operation	Description	Clock Cycles
1	swi r0, r19, 48	$M[R19+48] \leftarrow 0$	Initialize k to 0	2
2	lwi r4, r19, 48	$R4 \leftarrow M[R19+48]$	Determine if value of index i has already exceeded the for-loop limit; jump out of the loop if yes.	2
3	lwi r3, r19, 52	$R3 \leftarrow M[R19+52]$		2
4	cmp r18, r3, r4	Compare		1
5	bgei r18, 120	Branch operation		1 (last time is 3)
6	lwi r3, r19, 48	$R3 \leftarrow M[R19+48]$	Read variable k and calculate address of x[k]. The address is stored in register R3	2
7	muli r4, r3, 4	$R4 \leftarrow R3 * 4$		3
8	addik r3, r19, 4	$R3 \leftarrow R19 + 4$		1
9	addk r3, r4, r3	$R3 \leftarrow R4 + R3$		1
10	addik r3, r3, 24	$R3 \leftarrow R3 + 24$	Load value of x[k] into R3.	2
11	lwi r3, r3, 0	$R3 \leftarrow M[R3+0]$		2
12	put r3, rfs10		load x[k] into the Co-Processor.	2
13	swi r0, r19, 44	$M[R19+44] \leftarrow 0$	Initialize i to 0	2
14	lwi r4, r19, 44	$R4 \leftarrow M[R19+44]$	Determine if value of index i has already exceeded the for-loop limit; jump out of the loop if yes.	2
15	lwi r3, r19, 56	$R3 \leftarrow M[R19+56]$		1
16	cmp r18, r3, r4	Compare		1
17	bgei r18, 56	Branch		1 (last time is 3)
18	lwi r3, r19, 44	$R3 \leftarrow M[R19+44]$	Read index i and k. It is used to calculate the address of a[i][k].	2
19	addk r4, r3, r3	$R4 \leftarrow R3 + R3$		3
20	lwi r3, r19, 48	$R3 \leftarrow M[R19+48]$		1
21	addk r3, r4, r3	$R3 \leftarrow R4 + R3$		1
22	muli r4, r3, 4	$R4 \leftarrow R3 * 4$		2
23	addik r3, r19, 4	$R3 \leftarrow R19 + 4$		2
24	addk r3, r4, r3	$R3 \leftarrow R3 + R4$		1
25	lwi r3, r3, 0	$R3 \leftarrow M[R3]$	Load value a[i][k] into R3	2
26	put r3, rfs10		load a[i][k] into Co-Processor	2
27	lwi r3, r19, 44	$R3 \leftarrow M[R19+44]$	Increment index i in the inner for-loop	2
28	addik r3, r3, 1	$R3 \leftarrow R3 + 1$		2
29	swi r3, r19, 44	$M[R19+44] \leftarrow R3$		2
30	bri -64		Jump back to start of inner loop	3
31	lwi r3, r19, 48	$R3 \leftarrow M[R19+48]$	Re-read out index k	2
32	addik r3, r3, 1	$R3 \leftarrow R3 + 1$	Increment index k in the outer for-loop	2
33	swi r3, r19, 48	$M[R19+48] \leftarrow R3$		2
34	bri -128		Jump back to start of the outer for-loop (2 nd row of this table).	3
35	swi r0, r19, 48	$M[R19+48] \leftarrow 0$	Initialize index k back to 0 (start of third loop).	2
36	lwi r4, r19, 48	$R4 \leftarrow M[R19+48]$	Determine if value of	2

37	lwi	r3, r19, 52	$R3 \leftarrow M[R19+52]$	index k has already exceeded the for-loop limit; jump out of the loop if yes.	2
38	cmp	r18, r3, r4	Compare		1
39	bgei	r18, 52	Branch		1 (last time = 3)
40	get	r5, rfs10	$R5 \leftarrow \text{Co-Processor}$	Read back from Co-Processor and store result in result[i].	2
41	lwi	r3, r19, 48	$R3 \leftarrow M[R19+48]$		2
42	muli	r4, r3, 4	$R4 \leftarrow R3*4$		2
43	addik	r3, r19, 4	$R3 \leftarrow R19+4$		2
44	addk	r3, r4, r3	$R3 \leftarrow R4+R3$		1
45	addik	r4, r3, 32	$R4 \leftarrow R3+32$		2
46	addk	r3, r5, r0	$R3 \leftarrow R5$		1
47	swi	r3, r4, 0	$M[R4] \leftarrow R3$		2
48	lwi	r3, r19, 48	$R3 \leftarrow M[R19+48]$	Increment for-loop index k.	2
49	addik	r3, r3, 1	$R3 \leftarrow R3+1$		2
50	swi	r3, r19, 48	$M[R19+48] \leftarrow R3$		2
51	bri	-60	Branch	Jump to start of the third loop.	3
Total					20+68n+30mn

From Figure 2.13, we can see that the code (in bold) takes $10 + n(39+30m)$ clock cycles. The last for-loop takes $10+29n$ clock cycles. It thus requires a total of $20+(39+30m)n+29n = 20+68n+30mn$ clock cycles to compute (where A is an m-by-n matrix and x is a n-by-1 vector). The code size is 51 instructions or 204 bytes. The latency of the floating point adder and multiplier on the core do not contribute to the clock cycle count as the result is ready by the time the read back occurs.

Finally, we investigate the computation time and code size required to compute a matrix vector multiplication on Co-Processor using loop-unrolled technique. The C code and corresponding assembly code are shown in Figure 2.14 and Table 2.14 respectively:

Figure 2.14: C code implementation of matrix vector multiplication $y=Ax$ (by loop-unrolling)

Table 2.14: Assembly Instructions corresponding to C code in Figure 2.14

No	Instruction	Action Performed	Clock Cycles	Comments
1	lwi r4, r19, 4	result[0] = a[0][0]*x[0]; (mul)	2	Calculation of $x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}$, repeat for m times
2	lwi r3, r19, 28		2	
3	fmul r3, r4, r3		6	
4	swi r3, r19, 36		2	
5	lwi r4, r19, 8	result[0] += a[0][1]*b[1]; (mac)	2	Calculation of $x_{(n-1)} \begin{bmatrix} a_{1(n-1)} \\ a_{2(n-1)} \\ \vdots \\ a_{m(n-1)} \end{bmatrix} + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$ repeat for m-by-(n-1) times
6	lwi r3, r19, 32		2	
7	fmul r4, r4, r3		6	
8	lwi r3, r19, 36		2	
9	fadd r3, r3, r4		6	
10	swi r3, r19, 36		2	

We can thus calculate the clock cycle count and code size required as follows:

Table 2.15: Clock cycle count and number of instructions required for C code in Figure 2.14

Operation, $y=Ax$ A is m-by-n-matrix x is n-by-1 matrix	Total clock cycle count:	Total number of assembly instructions required:
$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix}$	12m	4m
$x_{(n-1)} \begin{bmatrix} a_{1(n-1)} \\ a_{2(n-1)} \\ \vdots \\ a_{m(n-1)} \end{bmatrix} + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$	20m(n-1)	6m(n-1)
Total	4m(5n-2)	2m(3n-1)

Thus for the generic case, it takes 4m(5n-2) clock cycles to compute a matrix vector multiply operation on Microblaze using the loop-unrolling method. Since 2m(3n-1) instructions are required to realize this, the code size is 8m(3n-1) bytes.

2.3.1 Speed up analysis of using Co-Processor to perform Matrix vector multiplication $y=Ax$ of variable sizes.

Using for-loop coding style

Referring to Tables 2.12 and 2.13 respectively, we compare the speed up of using Microblaze with Co-Processor over Microblaze standalone.

The speed up factor X is given by:

$$X = \frac{\text{clk cycle needed by Microblaze using for loop coding style}}{\text{clk cycle needed by Co - Processor using for loop coding style}} = \frac{m(122 + 64(n - 1))}{20 + 68n + 30mn}$$

where for $y=Ax$, A is a m-by-n matrix and x is a n-by-1 vector, $n=1,2,..$

For speed up to occur, $X > 1$

$$\Rightarrow 58m + 64mn > 20 + 68n + 30mn$$

$$\Rightarrow m > \frac{20 + 68n}{58 + 34n}, n = 1, 2, 3, \dots \quad \text{----- (1)}$$

For $n=1$, Equation (1) gives

$$m > \frac{20 + 68}{58 + 34} = \frac{88}{92}$$

$$\Rightarrow m = 1$$

This implies that there is speed up even for a scalar multiplication. However, this should not be the case because Equation (1) compares the speed up for the for-loop implementation, and there is no need to use the for-loop in the scalar case.

For $n=2, 3, 4, \dots$

$$\frac{20 + 68n}{58 + 34n} > 1$$

We can thus conclude that there will be speed up if the number of rows m is greater than or equals to 2. Next, we plot the speed up factor X for various sizes of A in Figure 2.15:

Figure 2.15: Speed up factor for various sizes of A

From the graphs in Figure 2.15, we can see that the speed up factor X increases as the number of rows m of matrix A increases. In addition, we can also see that as the number of columns n is increased for each row, the speed up factor X decreases. The $y=Ax$ operation thus achieves more speed up for the case where $m \geq n$ when using the for-loop technique. In other words, a tall and thin matrix A is preferred.

Using loop-unrolled coding style

Referring to Tables 2.11 and 2.15 respectively, we compare the speed up of using Microblaze with Co-Processor over Microblaze standalone

The speed up factor Y is given by:

$$Y = \frac{\text{clk cycle needed by Microblaze using loop unroll coding style}}{\text{clk cycle needed by Co - Processor using loop unroll coding style}} = \frac{4m(5n - 2)}{4(n + nm + m)}$$

For speed up to occur, $Y > 1$

$$\Rightarrow 16mn - 12m > 4n$$

$$\Rightarrow m > \frac{n}{(4n - 3)}, n = 1, 2, 3, \dots \text{-----} (2)$$

Again, we do not consider the case $n=1$.

For $n \geq 2$,

$$\frac{n}{(4n - 3)} < 1$$

$$\Rightarrow m \geq 1$$

In other words, there will always be speed up for all sizes of A for the case of loop-unroll.

Next we plot the speed up factor for various sizes of A in Figure 2.16 below:

Figure 2.16: Speed up factor for various sizes of A when code is loop unrolled

From Figure 2.16, we can see that the speed up factor X increases as the number of rows m and columns n of matrix A increases. However, the $y=Ax$ operation achieves more speed up for the case where $n \geq m$ i.e. a short and fat A matrix.

From Tables 2.11, 2.12, 2.13, 2.15, we summarize the speed performance and code size requirements for the various implementations of the matrix vector multiplication operation below in Table 2.16 and graphically presented in Figures 2.17 and 2.18

Table 2.16: Speed and code size characteristics table of various implementation styles of matrix vector multiplication

Software coding style	Hardware	Clock cycle count	Code size (bytes)	Speed up A/B
For-loop	Microblaze (A)	$m(122+64(n-1))$	364	$\frac{m(122 + 64(n - 1))}{20 + 68n + 30mn}$
	Microblaze + Co-Processor (B)	$20+68n+30mn$	204	
Loop-unroll	Microblaze (A)	$4m(5n-2)$	$8m(3n-1)$	$\frac{4m(5n - 2)}{4(n + nm + m)}$
	Microblaze + Co-Processor (B)	$4(n+mn+n)$	$8(n+mn+m)$	
here m= row size of matrix A, n is column size of matrix A				

2.4 Conclusion

It has been identified that the main bottleneck in using Co-Processors to speed up Microblaze is in the transfer of data operands to the Co-Processor. Thus, only operations which minimize data operand transfer or have very long computation time on Microblaze are suitable candidates for hardware acceleration. Two such examples are:

- 1) Floating point division
- 2) Floating point matrix vector multiplication, ie. $y = Ax$ operation.

In addition, we have also established the appropriate firmware style to achieve maximum speed up. In both cases, the loop-unrolled method of coding style would result in the maximum speed up when used with the hardware Co-Processor. We know that this method is optimum as the bottleneck becomes the data transfer latency incurred by Microblaze not being able to load/read back from the FSL bus fast enough. However, the code size for the loop-unrolled method increases linearly as the problem size increases. Thus, for the case of memory limited applications, one can utilize the for-loop method to achieve a speed up over the firmware method running on Microblaze.

Chapter 3 Design of the Co-Processor Modules for Microblaze

Two independent Co-Processors will be designed to accelerate

- 1) The floating point division
- 2) Matrix vector multiplication.

This chapter discusses the design details and the validation of these two Co-Processors.

3.1 Attaching a Co-Processor to Microblaze

As mentioned in Section 2.1.2, a Co-Processor communicates with Microblaze via the Fast Simplex Link (FSL) bus which is unidirectional (See Figure 2.2). As such, in order to transfer data to and from the Co-Processor, two FSL buses are needed:

- a) A bus slave in one direction. This link allows Microblaze to transfer data via the FSL bus to the Co-Processor.
- b) A bus master in the other direction. This link allows the Co-Processor to transfer data via the FSL bus back to Microblaze.

3.1.1 Co-Processor Slave Interface

The following signals describe the Co-Processor Slave Interface.

Table 3.1: FSL Slave interface from Co-Processor

Name	Direction	Function
FSL_S_Exists	Input	FSL slave data available for the Co-Processor
FSL_S_Read	Output	Signal to Microblaze that Co-Processor has read the existing input data and to put the next data if existing
FSL_S_Clk	Input	Clk into the Co-Processor
FSL_S_Data	Input (width 32 bits)	Input data into the Co-Processor

A typical slave transaction is as follows. When Microblaze has written into the FSL bus, the FSL bus asserts the FSL_S_Exists signal. When the Co-Processor has latched in the data, it immediately asserts the FSL_S_Read signal. If the Microblaze has previously written any more data for the Co-Processor via the FSL bus, the FSL_S_Data will be updated on the next clock cycle. In addition, the new FSL_S_Data will also be accompanied by a new assert on the FSL_S_Exists signal [20]

3.1.2 Co-Processor Master Interface

The following signals describe the Co-Processor Master Interface.

Table 3.2: FSL master interface from Co-Processor

Name	Direction	Function
FSL_M_Write	Output	Co-Processor strobe to Microblaze that it has data to transfer back to Microblaze
FSL_M_Full	Input	Microblaze/FSL bus strobe to Co-Processor that it is not ready to accept any more new data from the Co-Processor
FSL_M_Clk	Output	Clk from the Co-Processor
FSL_M_Data	Output (width 32 bits)	Output data from the Co-Processor

A master transaction is as follows. When the Co-Processor has new data available for Microblaze, it asserts the FSL_M_Write signal along with the FSL_M_Data. This will happen so long as FSL_M_Full is deasserted. If FSL_M_Full is asserted, the Co-Processor postpones asserting the new data on the bus until the FSL bus is ready to accept new data.[20]

3.2 Co Processor Design for $y=Ax$ operation

This section covers the design details of the Co-Processor performing Matrix Vector Multiplication. There are 3 main portions to this Co-Processor block:

- 1) Internal state machine controller to route the input data into the multiplier; store the result into the FPGA FIFO, read out the data from the FIFO back into the adder.

- 2) Output FIFO state machine controller to route the data coming out from the adder back to Microblaze via the FSL Master bus. If the FSL_M_Full is asserted when the result is available, store the result first in a FIFO. Once FSL_M_Full goes low, re-channel data out from the FIFO to the FSL_M_Data and assert the FSL_M_Write signal.
- 3) OPB interface to control registers; these control registers determine the size of the matrix and vector to be multiplied together.

Figure 3.1 below is a simplified block diagram of the matrix vector multiplication Co-Processor.

Figure 3.1: Simplified Block Diagram of Matrix Vector Multiplier Co-Processor

Figure 3.2 below describes in greater detail a portion of the internal state machine controller (internal controller). It controls the data coming in from the FSL slave link interface and routes it into the multiplier and adder. It is to be noted that the entire Co-Processor is clocked by the same master FPGA clock (not shown in this diagram).

Figure 3.2: Section of Matrix Vector Multiplier Co-Processor routing input data from Microblaze into Floating Point Adder and Multiplier

To use the Co-Processor, the user has first to initialize an OPB bus controlled register to configure this Co-Processor the size of the row and column of the matrix. Only 16 bits of this register are used; the lower byte of this 32 bit register will be connected to the column size input of the internal controller block; the next lowest byte of this register will be connected to the row size input of the internal controller block.

The State Machine of the Internal Controller block controls only how data is piped into the multiplier.

Figure 3.3: Internal Controller Block State Machine

When in IDLE state, the state machine continually latches in the value of the OPB control registers every cycle. In order to simplify our discussion, we will denote a particular node in the Matrix as op1 (operand 1) and a particular node in the Vector as op2 (operand 2). In addition, the row_proc counter (row processed counter) and col_proc (column processed counter) are reset to zero at this state. These counters will track the number of rows and columns of data being fed into the Co-Processor.

When the FSL_S_Exists signal is first asserted, it latches in the FSL_S_Data value into the op2 register. This is the 32 bit register shown in Figure 3.2 above. The State machine also asynchronously asserts FSL_S_Read. In addition, it transits to the RD_MUL_OP1 state in the next clock cycle.

In the RD_MUL_OP1 state, the controller waits for more data to come in. Once FSL_S_Exists signal is again asserted, it will start a multiply operation immediately by asserting mul_nd (multiply new data). Op2 data which is now stored in the 32 bit register is then multiplied with the incoming FSL_S_Data which holds op1 data, and the row_proc counter is incremented. The controller remains in this state until all the rows for one column of the matrix is fed already into the multiplier. When this happens, it clears the row_proc counter and increments the col_proc counter by 1 in the next clock cycle. A state transition to RD_MUL_OP2 state also happens during this time.

In RD_MUL_OP2 state, no state transition happens until FSL_S_Exists is asserted. A new value is loaded into op2 (loading of a new node from the vector) before transiting back to RD_MUL_OP1 to await matrix data for the next new column.

The internal controller transits back in state between RD_MUL_OP1 and RD_MUL_OP2 in this manner until the number of columns to be processed is completed. The internal controller will then transit back to IDLE state.

The other signals shown in Figure 3.2 are derived from the signals in the state machine.

The storing of data into the FPGA FIFO is controlled by the counters from this state machine. When the col_proc counter is zero, fifo_src_sel (refer to Figure 4.2) is de-asserted, channeling the data from the multiplier into the FIFO. The controller is designed in this way because no additions is needed when all the rows in the first column of the matrix is multiplied with the first row in the vector. (ie. During the calculation of

$$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix})$$

Once col_proc is no longer zero, the Co-Processor will be calculating the portion:

$$x_{(n-1)} \begin{bmatrix} a_{1(n-1)} \\ a_{2(n-1)} \\ \vdots \\ a_{m(n-1)} \end{bmatrix} + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$$

where $x_{(n-1)} \begin{bmatrix} a_{1(n-1)} \\ a_{2(n-1)} \\ \vdots \\ a_{m(n-1)} \end{bmatrix}$ is retrieved from the FIFO; $x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix}$ is the result from the

multiplier.

The adder result will be stored in the FIFO at this point; the `fifo_src_sel` is then asserted to channel the result from the hardware adder into the FIFO instead.

The hardware adder is also controlled in a similar way. The `add_nd` will only be asserted (to start an add operation) when the `col_proc` counter is no longer zero and `mul_rdy` (multiply result ready) signal is asserted by the hardware multiplier. The `rd` signal, which reads data from the FPGA FIFO will be asserted $M-1$ clock cycles after `mul_nd` is asserted. (M is the pipeline depth of the hardware multiplier and is selected to be 3 in this case). This is to ensure that when the result from the multiplier is available, it is added to the previous result stored in the FIFO.

The data output back to Microblaze via the FSL master bus is controlled by another state machine. This can be described by Figure 3.4 below:

Figure 3.4: Output FIFO controller State Machine

The output FIFO controller state machine is used, in the case when Microblaze is not ready to receive the result when available from the adder. In this case, the controller will become active, and pipe the data into the output FIFO instead. When that happens, the controller will start reading out data from the output FIFO every time once FSL_M_FIFO full is de-asserted until the point when output FIFO becomes empty.

The output FIFO controller first begins from the reset state of OUTPUT_ADDER_DATA state. It never leaves this state until when output data is available from the adder and when FSL_M_Full is asserted. When that happens, this implies that Microblaze/FSL bus is not ready to receive the result thus the result is written into the output FIFO before transiting to the OUTPUT_FIFO_NOT_EMPTY stage.

When in OUTPUT_FIFO_NOT_EMPTY stage and signal FSL_M_Full is de-asserted, this means that Microblaze is ready once again to receive data; thus data is read out from the output fifo and put onto the master FSL bus back to the Microblaze. No state machine occurs at this point; until one of the following 2 scenarios occurs:

- 1) the output FIFO becomes empty. At this point, a state transition back to the default state OUTPUT_ADDER_DATA occurs, whereby adder data will be channeled to the FSL master bus once more, whenever it becomes available
- 2) the FSL_M_Full signal is asserted again, one clock cycle after the rd_output_fifo signal is asserted. This means that the data just read out from the RAM cannot be fed back to Microblaze yet. A state transition to the WAIT_TILL_BUS_AVAIL signal thus occurs.

In the WAIT_TILL_BUS_AVAIL stage, once FSL_M_Full signal is deasserted, the FSL_M_Write signal is immediately asserted to push the data previously read out from the FIFO in the previous state onto the bus. A state transition now occurs in the following scenario:

- 1) the output FIFO becomes empty when FSL_M_Full is de-asserted. In this case, a state machine back to default OUTPUT_ADDER_DATA state occurs.
- 2) The output FIFO is not empty. The rd_fifo is thus asserted to read the next FIFO data onto the bus. A state transition thus occurs back to the previous state OUTPUT_FIFO_NOT_EMPTY.

In both OUTPUT_FIFO_NOT_EMPTY and WAIT_TILL_BUS_AVAIL states, any available data from the adder will be channeled into the FIFO.

3.2.1 Pipeline Depth Selection of Hardware Floating point Adder and Multiplier

In this section, factors which influenced the selection of the pipeline depth of the floating point multiplier and adder of the Co-Processor will be discussed.

In the case of the FSL bus, there is an additional one clock cycle latency required to transfer data through FSL bus, from the Microblaze, to the Co-Processor and vice versa. This arises from the FSL bus style selected, being that of a synchronous 16-word-depth FIFO which allows buffering up of words from the Microblaze in the event that the Co-Processor is busy.

The effect of this selection on the Co-Processor Design can be described in Figure 3.5 below:

Figure 3.5: Microblaze vs Co-Processor data latency

The diagram above extends the original scenario mentioned in Chapter 2 when the Co-Processor is performing a matrix vector multiply operation on an input matrix of size [3x2] and a vector of size [2x1]. In this case, data from Microblaze reaches the Co-Processor, one cycle late.

This factor must be taken into account during the selection of pipeline depth of an appropriate multiplier/adder. As mentioned in the previous chapter, the pipeline depth of the multiplier and adder should ideally be 4. However, as shown from the above diagram, the pipeline depth of the multiplier has to be reduced to 3, to ensure that the first result is immediately available after the last operand from the matrix has been loaded into the Co-Processor.

The pipeline depth of the adder in this case still remains at 4. The effect of the latency of the bus in this case does not impact the reading back of the result; the first Matrix Vector multiplication result is ready on the FSL bus by the time the Microblaze acknowledges a read from the bus. This effect can be easily observed by simulation as mentioned in the next section.

3.2.2 Verification of functionality of Matrix Vector Multiplier Co-Processor

The functionality of the Matrix Vector Multiplier Co-Processor is verified by simulation with Modelsim. A simulation is performed for the same scenario in which Co-Processor multiplies a matrix of size 3-by-2 and a vector of size 2-by-1. The firmware running on Microblaze is of the loop-unrolled type performing the loading of operands as described in Figure 2.11.

Figure 3.6: Functional Simulation of Matrix Vector Multiplier Co-Processor

We can conclude the following from the above diagram:

- 1) the Microblaze does a load every 4 clock cycles
- 2) the first add result is ready (add_rdy signal) when the last load into the Co-Processor occurs (fsl0_m_write signal) from Microblaze.
- 3) By the time Microblaze reads back the first data from the FSL bus, by asserting the fsl0_s_read signal, the data is already available. This confirms that the Microblaze does not require any additional time to wait for data arrival from the Co-Processor after its last operand load.

The Matrix Vector Multiplier Co-Processor's behavioral functionality thus meets the design specifications mentioned earlier in the last chapter.

3.3 Co Processor Design for Floating point Division

This section covers the design details of the Co-Processor performing floating point division. This Co-Processor has the following main blocks:

- 1) Internal state machine controller to route the input data into the hardware floating point divider.
- 2) A similar output FIFO state machine controller to route the data coming out from the divider back to Microblaze via the FSL Master bus. If the FSL_M_Full is asserted when the result is available, store the result first in a FIFO. Once FSL_M_Full goes low, re-channel the stored divide result out from the FIFO to the FSL_M_Data and assert the FSL_M_Write signal.

Figure 3.7: Full Hardware Divider Co-Processor Block Diagram

The entire circuitry of the Divider Co-Processor is clocked by the same FPGA clock as well.

The Co-Processor is a simplified version of the Matrix Vector Multiplier Co-Processor. The internal controller in this case is much simpler as it handles only the floating point divider operation.

Figure 3.8: Internal Controller Block State Machine for Divider Co-Processor

Referring to Figure 3.8, when the first set of data is available from Microblaze (`FSL_S_Exist` goes active), the Internal Controller Block asserts the signal `latch_in_data` which causes the `FSL_S_Data` (operand 1 data) to be registered in the 32 bit register. When `FSL_S_Exist` next becomes active, operand 2 is now available. The `div_nd` (divide new data available) signal is thus asserted and a new divide operation commences.

Figure 3.9: Output FIFO Controller State Machine for Divider Co-Processor

Figure 3.9 above describes how the output FIFO controller outputs the divide result onto the FSL data bus. The hardware divider asserts the rdy signal once the divide result is completed. During this time, the output controller will monitor simultaneously the FSL_M_Full signal from the default IDLE state; if unasserted, the output controller will assert the wr_o signal which in turn asserts the FSL_M_Write signal and drives the divide result to the FSL_M_Data output at the same time. The src_sel signal remains unasserted for this case.

If the FSL_M_Full signal is asserted when the div_rdy signal is asserted by the hardware divider, this implies that Microblaze/FSL bus is not ready to receive the result. The output controller will then instead assert the wr_fifo signal which will buffer the divide

result into the FIFO before transiting to the OUTPUT_FIFO_NOT_EMPTY stage. During this time, any new data available from the divider will still be buffered.

The output controller remains in this state until FSL_M_FULL is de-asserted by the FSL bus. When that happens, this implies that the FSL bus is ready to receive data thus the rd_fifo and src_sel signals are asserted, subsequently channeling the divide result from the FIFO back to Microblaze. Once the output FIFO becomes empty, the output FIFO controller de-asserts src_sel and returns to the default IDLE state

When in the OUTPUT_FIFO_NOT_EMPTY state, if the FSL_M_FULL is reasserted by the FSL bus one clock cycle later after rd_fifo is asserted, this implies that the data read from the FIFO cannot be transferred yet to the FSL bus so the output controller transits to the WAIT_TILL_DATA_AVAIL stage.

When FSL_M_FULL is next de-asserted, it will output first the data already read from the FIFO onto to the FSL bus. If the FIFO is empty, the output controller transits to IDLE state. Otherwise, it transits back to the OUTPUT_FIFO_NOT_EMPTY state.

The pipeline depth of the floating point divider in this Co-Processor is selected to be 14. This selection is based on the minimum pipeline depth which can be synthesized to run at 100 MHz.

The FSL buses attached to the Hardware Divider Co-Processor has the same configuration as the Matrix Vector product Co-Processor. This implies that the bus style configured contains a FIFO to buffer up data in the event that the destination node is unable to receive the data sent by the source node. More importantly, the presence of a FIFO would introduce an extra clock cycle latency to transfer data from Microblaze to Divide Co-Processor and vice versa.

The divide Co-Processor latency is thus increased from 14 clocks to 16 clocks as it takes 2 clock cycles extra to transfer data to and from the Co-Processor. The total divide operation (inclusive of loading operands and reading back result) is thus $8+2+16 = 26$ clock cycles.

3.3.1 Verification of functionality of Floating Point Division Accelerator Co-Processor

The verification of the floating point division accelerator Co-Processor is done also using Modelsim. It is verified similarly that the Microblaze obtains the result after 26 clock cycles.

Figure 3.10: Functional Simulation of floating point divider Co-Processor

3.4 Actual versus Theoretical Performance of the Matrix Vector Multiplier Co-Processor

Having covered the design details of the Co-Processor, the performance of the designed Co-Processor will now be verified against the theoretical calculations. The test is being conducted in the following way using the test-bed shown below:

Figure 3.11: Using Matlab on PC to communicate with the FPGA board via RS232

It is a XUP Virtex II Pro FPGA prototyping board [18] connected to the desktop PC via an RS232 serial communication cable, with the main clock derived from an on board crystal oscillator running at 100 MHz.

- 1) Matlab generates the operands and sends them to the FPGA board via the UART interface
- 2) firmware running on Microblaze will accept these parameters and turn on the timer
- 3) Microblaze will load the operands into the Co-Processor and read back the results

- 4) The results calculated by the Co-Processor and the number of clock cycles taken to calculate the operation will relayed back to Matlab on PC via the UART interface again.

Figure 3.12 shows the actual performance versus theoretical number of clock cycles required to perform matrix-vector multiplication using a Co-Processor using the loop-unrolled technique. The theoretical number of clock cycles required to calculate $y=Ax$ where A is of size m-by-n, and x is of size n-by-1 is $4(n+mn+m)$.

Figure 3.12: Actual vs Theoretical Clk Count to Perform Matrix Vector Mult using Co-Processor for $m=3$ and varying n

Figure 3.13: Actual vs Theoretical Clk Count to Perform Matrix Vector Mult using Co-Processor for $n=32$ and varying m

As seen from Figures 3.12 and 3.13, we can see that the theoretical performance matches that of the actual performance for the Matrix Vector Multiplication Co-Processor. The actual clock cycle count is consistently more than the theoretical count by a few clock cycles due to some inaccuracies arising from when the on-chip timer is switched off. Switching off the timer requires a few additional clock cycles which has not been discounted in the actual clock cycle count as shown in Figure 3.13 above. As such, the theoretical speed up achievable (which was mentioned in the previous chapter) will also match the actual speed up attainable.

Due the constraints of Block RAM size on the FPGA, the theoretical versus actual performance cannot be done for larger matrices. However, it can be deduced that if a larger size FPGA is used, the same performance metric will be achieved.

3.5 Actual versus Theoretical Performance of the Floating Point Division Accelerator Co-Processor

The verification of the theoretical performance of the floating point division Co-Processor is done in the same way as how the Matrix Vector multiplier Co-Processor is being verified. Operands were generated in MATLAB and passed to the FPGA board through the serial port. On completion of the division, the result as well as the number of clock cycles it needed to compute the results were sent back to matlab. The loop-unrolled coding style is adopted in the firmware.

The actual performance of the floating point division accelerator Co-Processor is also same as the theoretical performance. It takes 26 clock cycles to return the result of one floating point division. In addition, it is also verified in the same way that a series of more than 5 divisions in parallel will also effect in a speed up of 3 times.

Chapter 4 Using Co-Processor to speed up the Interior Point Method

In Chapters 2 and 3, we identified the type of operations suitable to be run on a Co-Processor and built the corresponding Co-Processors accordingly. In this chapter, we discuss how the Co-Processors are used to speed up the IPM and the corresponding speed up results obtained.

4.1 The Interior Point Method for solving QP problems

A QP problem can be represented as follows:

$$\min_{z \in \mathbb{R}^n} \frac{1}{2} z^T Q z + c^T z$$

such that $Jz \leq g$

where Q is a $n_v \times n_v$ positive definite matrix and c is a $n_v \times 1$ vector. Also, J and g are of sizes $m_c \times n_v$ and $m_c \times 1$ respectively, where m_c is the total number of constraints and n_v is the number of decision variables.

The Interior Point Method (IPM) is one of the most widely adopted methods for solving QP problems. This method performs a series of successive Newton-like descent steps to approach the solution of the Karush-Kuhn-Tucker (KKT) equations. Each descent step is obtained by solving a system of linear equations.

From the infeasible interior point framework introduced discussed in [15], the QP can be solved iteratively as follows:

Step 1: Choose an initial condition (z_0, λ_0, t_0) with $(\lambda_0, t_0) > 0$.

Step 2: At the k -th iteration step, solve for the increments $(\Delta z_k, \Delta \lambda_k, \Delta t_k)$ with

$$\begin{bmatrix} Q & J^T \\ J & \Gamma_k \end{bmatrix} \begin{bmatrix} \Delta z_k \\ \Delta \lambda_k \end{bmatrix} = \begin{bmatrix} r_k \\ s_k \end{bmatrix}, \quad (4)$$

and

$$\Delta t_k = -t_k + (\Lambda_k)^{-1}(\sigma \mu_k e - T_k \Delta \lambda_k). \quad (5)$$

Here Q is a symmetric matrix, $\Lambda_k = -(\Lambda_k)^{-1} T_k$

$$\Lambda_k = \begin{bmatrix} \lambda_k(1) & & \\ & \ddots & \\ & & \lambda_k(m_c) \end{bmatrix}, \quad T_k = \begin{bmatrix} t_k(1) & & \\ & \ddots & \\ & & t_k(m_c) \end{bmatrix}, \quad e = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \quad (6)$$

In addition,

$$\begin{aligned} r_k &= -Qz_k - J^T \Lambda_k e - c, \\ s_k &= -Jz_k + g - \sigma \mu_k (\Lambda_k)^{-1} e \end{aligned} \quad (7)$$

Step 3: Increment the variables by

$$(z_{k+1}, \lambda_{k+1}, t_{k+1}) = (z_k, \lambda_k, t_k) + \alpha_k (\Delta z_k, \Delta \lambda_k, \Delta t_k) \quad (8)$$

for some $\alpha_k \in (0, 1]$ subject to $(\lambda_{k+1}, t_{k+1}) > 0$.

Step 4: Judge the convergence. If the iterations converge, stop the process and the decision control z_{k+1} is obtained; otherwise, go back to Step 2 with $(z_{k+1}, \lambda_{k+1}, t_{k+1})$ and continue the iteration process.

Since, the number of constraints m_c is usually much larger than the number of variables N_u in most embedded systems, it could possibly be computationally advantageous to solve from equation (4) and (7) as the following decoupled system

$$\begin{cases} (Q - J^T \Gamma_k^{-1} J) \Delta z_k &= (r_k - J^T \Gamma_k^{-1} s_k) \\ \Gamma_k \Delta \lambda_k &= s_k - J \Delta z_k \end{cases} \quad (9)$$

Note that Γ_k is a diagonal matrix.

Studies on the Interior Point Method have shown that step2 is the most computationally

intensive [1].

4.2 Determining the Bottleneck of the Interior Point Method

In this section, we first discuss the test bed used to obtain the results. Next, we highlight the experimental results which confirm that step 2 is the bottleneck of the Interior Point Method.

4.2.1 The Test-bed

The test-bed used is the same one as shown earlier in Figure 3.11. In this case, the matrices of the QP problems are passed to the FPGA board using the serial port. Next, the Interior Point Method algorithm on FPGA compute the corresponding result and return it back to PC also via the same serial port.

4.2.2 Step 2 is the Main Bottleneck

Having realized a firmware implementation of the Interior Point Method running on Microblaze, we then use the test-bed shown in Figure 3.11 to empirically determine its bottleneck. An on-chip timer attached to Microblaze was used to track the computation time required by the major steps of the IPM. Subsequently, these timer values were sent back to the PC via the serial port. We plot the percentage of total computation time per iteration spent in step 2 of the IPM. From Figure 2.2, we can see that step 2 is indeed the bottleneck as it consumes 80% to 97% of the total computation time per iteration, depending on the size of the QP problem, parameterized by m_c , the number of inequality constraints and n_v , the number of decision variables.

Figure 4.1: Graph of Percentage of computation time per IP iteration spent in computing step 2 for various combinations of m_c and n_v

The results also show that an increase in the number of decision variables (n_v) in a QP problem has a larger impact than an increase in the number of inequalities (m_c) on the overall increase of computation time of step 2⁴. The percentage of total computation time for step 2 per iteration is rather constant when m_c varies and n_v is held constant.

4.3 Using the Matrix Vector Multiplier Co-Processor

Figure 4.2 shows how the Matrix Vector Multiplier is used to accelerate $y=Ax$ operation in the IPM.

⁴ Analysis of the operations on page 63, equation (9) show that we are required to invert the matrix $(Q - J^T k^{-1} J)$ which is of size n_v -by- n_v . The number of computations required to invert a matrix of this size is proportional to n_v^3 . This explains why the number of decision variables n_v has a larger impact in the number of inequalities m_c on the overall computation of Step 2 of the IPM.

<p>Calculation of equation: $\text{Artp2} = \mathbf{g} - \mathbf{J} * \mathbf{z}$</p> <p>J is matrix of size mc-by-num where mc=32, num=3</p>	<p>C source code with MB standalone (original)</p>	<pre>do { dell = dotprod(num,&matJ[i*knum],&z[0],1,1); Artp2[i] = g[i] - dell; i++; }while (i!=mc);</pre>
	<p>C source code with Co-Processor (for-loop coding style)</p>	<pre>/* dell_array = J*z using Co-Processor */ for (ii=0;ii<num;ii++){ write_into_fsl(z[ii],0); for (jj=0;jj<mc;jj++){ kk = jj*num + ii; write_into_fsl(matJ[kk],0); } } for (ii=0;ii<mc;ii++){ read_from_fsl(dell_array[ii],0); } do{ Artp2[i] = g[i] - dell_array[i]; i++; }while (i!=mc);</pre>
	<p>C source code with Co-Processor (loop-unroll coding style)</p>	<pre>/* dell_array = J*z using Co-Processor */ //load col 1 of matJ with z[0] write_into_fsl(z[0],0); write_into_fsl(matJ[0],0); write_into_fsl(matJ[7],0); write_into_fsl(matJ[14],0); . . write_into_fsl(matJ[217],0); //load col 2 of matJ with z[1] write_into_fsl(z[1],0); write_into_fsl(matJ[1],0); write_into_fsl(matJ[8],0); write_into_fsl(matJ[15],0); . . write_into_fsl(matJ[218],0); //load col 7 of matJ with z[6] write_into_fsl(z[6],0); write_into_fsl(matJ[6],0); write_into_fsl(matJ[13],0); write_into_fsl(matJ[20],0); . . write_into_fsl(matJ[223],0); /* read back results*/ read_from_fsl(dell_array[0],0); read_from_fsl(dell_array[1],0);</pre>

		<pre> . . read_from_fsl(dell_array[31],0); do{ Artp2[i] = g[i] - dell_array[i]; i++; }while (i!=mc); </pre>
--	--	--

Figure 4.2: Demonstration of usage of Co-Processor to accelerate $y = Ax$ operation

In Figure 4.2, three different implementations of the $y=Ax$ operation is shown. The first shows the original C code implementation running on MB standalone. The second and third demonstrate the use of the Co-Processor with the for-loop and loop-unrolling coding styles respectively.

In the Interior Point Method, there are several additional considerations with regards to usage of the Matrix Vector Multiplier Co-Processor:

- 1) Matrix vector multiplications of various sizes are performed throughout the Interior Point Method. As such, the OPB registers which control the Co-Processor has to be updated each time the size of the matrix and vector changes. This introduces some additional steps which are previously not required in the pure firmware solution of the Interior Point Method
- 2) The Co-Processor is used to accelerate $y=Ax$ operations for $(5+n_v)$ times in Step 2 and once in step 3 per iteration. We refer to Table 4.1 as a reference to their corresponding sizes.

Table 4.1: Size of Matrix Vector Multiplications calculated by Co-Processor for each iteration of the Interior Point Method

Row	Size of matrix A	Number of times Co-Processor is used per iteration	Used in Step
1	m_c -by- n_v	2	2
2	n_v -by- m_c	n_v+1	2
3	n_v -by- n_v	2	2
4	1-by- m_c	1	3

We plot the speed up factor X,Y (refer to Section 2.3.1 for the formula) achievable for selected rows (m) and column sizes (n) for matrix A in Figure 4.1 below:

Figure 4.3: Graphs of Speed up Factor for using Co-Processor to accelerate Matrix Vector Product in Step 2

As seen from Figure 4.3, using a Co-Processor accelerates the matrix vector product (both X and $Y > 1$) for all the selected sizes of m_c and n_v . The loop-unroll implementation yields a much better speed up (Y is a minimum of 3.2 times) than the for-loop implementation (X is a minimum of 1.3 times). The matrix vector product Co-Processor is thus used to accelerate all the matrix vector product operations in Step 2 of the Interior Point Method.

From Table 4.1, we see that there is only one occurrence of the $y = Ax$ operation in Step 3. The speed up factors X and Y achieved when using a Co-Processor is as follows:

Figure 4.4: Graphs of Speed up Factor for using Co-Processor to accelerate Matrix Vector Product in Step 3

From Figure 4.4, we see that using the for-loop method with Co-Processor in Step 3 does not result in any speed up as $X < 1$. It is thus not used to accelerate the matrix vector product in step 3. The loop-unroll method yields a minimum speed up factor Y of at least 2.43 times and is thus used to accelerate step 3.

4.4 Using the Divide Co-Processor

The number divisions per iteration required in both step 2 and 3 of the IPM is $m_c + n_v^2 + 1$. We refer to Table 4.2 for further details.

Table 4.2: Divisions per iteration for the IPM

Row	Number of divisions	Comments
1	m_c	Can be performed back-to-back
2	$n_v \times n_v$	Performed standalone
3	1	Performed standalone

From Table 4.2, we see that m_c divisions are performed back-to-back per iteration. Since, $m_c > 5$, the pipelining method mentioned earlier in Figure 2.8 is used. All the operands for all the m_c divisions are loaded into the divide Co-Processor at one go, reading back the results only at the end.

In the cases where one division is calculated at a time (rows 2 and 3 of Table 4.2), the normal usage of the Co-Processor is adopted (refer to Figure 2.3). Since the use of a Co-Processor accelerates floating point divisions for all cases, (as shown earlier in Chapter 2), it is used to accelerate all the divisions.

4.5 Actual Speed up of the Interior Point Method when using both Co-Processors

Since, it is rather difficult to calculate the theoretical overall speed up achievable using the Co-Processor to solve QP problems of various sizes, we resort to experimentation to obtain the speed up performance achieved when using the Co-Processor with Microblaze.

The speed up results was presented earlier on in Section 1.2 (refer to Figures 1.1 and 1.2). Here, we analyze these results in greater details. Since Microblaze standalone requires the longest computation time, we use it as a basis of comparison when calculating the speed up factor as shown below:

$$\text{Speed up} = \frac{\text{average time taken by Microblaze standalone to compute each QP problem}}{\text{average time by variant to compute each QP problem}}$$

Figures 4.5 and 4.6 show the speed up factor of the other variants over Microblaze standalone.

Figure 4.5: Graph of speed up factor of variant over Microblaze standalone vs number of inequalities (mc) with fixed $nv=3$

Figure 4.6: Graph of speed up factor of variant over Microblaze firmware solution vs number of decision variables (nv) with fixed mc=32

As we can see from the above Figures 4.5 and 4.6, the performance improvement of using a Co-Processor with the for-loop based approach firmware is roughly about 1.12 to 1.15 times. Loop unrolling the for-loops in the firmware yield better results, achieving roughly 1.8 to 2.2 times improvement in computation time. This approaches the performance of the sequential Handel C sequential variant⁵. However, the loop-unroll approach quickly runs out of memory as the size of the QP problem increases in size.

As shown earlier in Section 4.3.2, step 2 consumes 80% to 97% of the overall computation time for each iteration of the Interior Point Method, thus the Co-Processors were used primarily to accelerate this portion of the Interior Point Method. Thus, in

⁵ It is to be noted that the Handel C implementation is a pure hardware implementation on FPGA without the use of any soft cores. It is thus expected to out perform that of the pure firmware solution. The results shown are thus rather promising as the hardware accelerated MB solution nearly reaches the performance of a pure hardware implementation.

Figures 4.7 and 4.8 below, we show the improvement in average computation time per iteration required to compute step 2 when using the Co-Processors.

Figure 4.7: Plot of average clock count per iteration for step 2 vs number of decision variables n_v (fixed $mc=32$)

From Figure 4.7, for varying n_v , we see that using the Co-Processors with for-loop based firmware results in a slight improvement of about 10 to 15% in calculation of step 2. The best speed up factor of over 2 times is achieved when using the Co-Processor with loop-unrolled firmware.

Figure 4.8: Plot of average clock count per iteration for step 2 vs number of inequalities mc with (nv=3)

*Note: no performance data is available for Microblaze + Co-Processor (loop unrolled firmware) for mc=112 as firmware code size exceed on-chip memory size.

From Figure 4.8, for varying m_c , we see that using the Co-Processors with for-loop based firmware results in a slightly better improvement of about 16 to 18% in calculation of step 2. The best speed up factor of between 2 to 2.2 times is achieved when loop unrolled firmware is employed as well for use with the Co-Processors.

4.6 FPGA Resource Usage and Code Size Requirements for the IPM

The FPGA resources utilized by MB standalone and with the 2 Co-Processors are summarized in Table 4.3.

Table 4.3: Resource Usage

Resources	MB	MB with Co-Processors
LUT	2284 (8%)	4984 (18%)
FFs	1545 (5%)	3060 (11%)
Block RAM	32 (23%)	35 (25%)
MULT18x18	7 (5%)	11 (8%)

From Table 4.3, we see that the MB with Co-Processors requires about nearly 2.2 times as many LUTs as MB standalone as they are used to realize the floating point multiply, add and divide cores in the 2 Co-Processors. There is also an increase in the number of multipliers used to realize the floating point multiply core. Block RAM usage is also increased by 2% to realize the internal buffers in the Co-Processor and within the FSL busses. Next, we summarize in Figure 4.9 the overall code size requirements of the firmware required to solve the various sized QP problems presented earlier in Figure 1.1 and 1.2

Figure 4.9: Graph of code size requirements vs mc and nv .

From Figure 4.9, we see that the loop-unroll firmware has the largest code size for all QP problems and is larger than the code size of MB by a maximum of 1.7 times. This is followed by the for-loop firmware for Co-Processor and for MB standalone respectively.

Chapter 5 Conclusion

In this thesis, we have studied the general considerations when partitioning an embedded algorithm to run on a Microprocessor (Microblaze) or on a Co-Processor. It was found that suitable candidates should either allow reuse of data operands on the Co-Processor (e.g. $y = Ax$ operations) or have very long computation time (e.g. floating point division). In addition, the for-loop and loop-unroll coding styles were studied and it was found that the latter provided better speed up when used with the Co-Processor. In terms of code size, the loop-unroll coding style increases linearly as the problem size increases. The Co-Processors were then built based on these considerations and used to accelerate the IPM to achieve a maximum speed up of 2.2 times.

5.1 Eliminating Function Call in $y = Ax$ operation on MB

In our implementation of $y = Ax$ operation presented in Section 2.3, we use a function to implement the dot product to make the code modular. It can be argued such coding style incurred additional clock cycle overhead. We shall now assess the impact of this overhead on the overall performance of the IPM method.

Instead of adopting the style shown in Figure 2.12 and Figure 2.13 which uses the function “dotprod”, we could implement the required calculations directly using a inner for-loop as shown in Figure 5.1.

```
/* alternative C code for y=Ax operation*/
for (i=0;i<m;i++){
    y[i] = 0;
    for (j=0;j<n;j++){
        y[i] += A[i][j] *x[j];
    }
}
```

Figure 5.1: Alternative C source codes for implementing matrix vector multiplication

This coding style has the advantage of no additional branching overheads incurred as a result of function calls. However, the elimination of branching overheads was found to have negligible speed up effects (differences of up 3%) on the average computation time required by MB to solve a QP problem. The graphical plots presented earlier in Figure 1.1 and 1.2 are updated in Figure 5.2 below to compare the computation time required by this alternative coding style.

Figure 5.2: Updated Graphs of computation time per QP problem vs mc and nv

As seen from Figure 5.2, the firmware version of the IPM without the use of the dotprod function (the line 'MB with no dotprod') has about the same computation time as the one using the dotprod function (differences by at most 3%). It can thus be concluded that using the dotprod function in Section 2.3 to calculate $y = Ax$ is a reasonable software implementation approach on MB.

5.2 For-loop or Loop-unroll coding style

While it is in general true that using the For-loop coding style give more space efficient code, in our work presented so far, we did not exploit the fact, for some small matrix, it will be advantages to use loop-unroll coding style instead of for-loop coding style. We shall now address this by finding out, for the $y=Ax$ operation, where A is a matrix of size m -by- n , for what values of m and n will the loop-unroll coding style give a smaller or equal code size than the for-loop coding style?

For the case of using only the Microblaze softcore processor, the values of m and n when the loop-unroll coding style gives a smaller or equal code size than the for-loop coding style can be calculated, by referring to Table 2.16, as follows

$$8m(3n - 1) \leq 364$$

$$\Rightarrow m \leq \frac{45.5}{3n - 1}$$

This inequality is represented graphically presented in Figure 5.3. When the size of matrix A falls within the blue area in the graph, it would be more advantages to use the loop-unroll coding style than the For-loop coding style, since this will give better speed performance without sacrificing code size.

Figure 5.3: Size of matrix A when loop-unroll code size is smaller than or equal to for-loop code size for MB standalone

Similarly, for the case when Co-Processor is used, referring to Tables 2.16, the values of m and n when the loop-unroll coding style gives a smaller or equal code size than the for-loop coding style is given by

$$8(n + mn + m) \leq 204$$

$$\Rightarrow n + mn + m \leq 25.5$$

$$\Rightarrow m \leq \frac{25.5 - n}{n + 1}$$

This inequality is represented graphically in Figure 5.4. When the size of matrix A falls within the red area in the graph, it would be more advantages to use the loop-unroll coding style than the For-loop coding style, since this will give better speed performance without sacrificing code size.

Figure 5.4: Size of matrix A when loop-unroll code size is smaller than or equal to for-loop code size using Co-Processor

5.3 Recommendations for Further Research

In this section, we mention a few directions worthy of further research:

- 1) In this thesis, we have worked on acceleration of $y = Ax$ operation. It was found that the for-loop implementation preferred the tall and thin A matrix while the loop-unroll implementation prefers a short and fat one. Since, there are both types of A exist in the IPM. Further study should be carried out to exploit this fact.
- 2) Studies have also been performed to categorize the type of optimum operations which are suitable to be run on the Co-Processor. Reduction of data transfer could be achieved further by grouping together operations involving the same operands. For instance, in the IPM, there are two subsequent $y = Ax$ operations which require the same A matrix. Additional buffers could be added to store matrix A to provide further opportunities for reuse. This however comes with the penalty of increased

buffer size within the Co-Processor to store A and more configuration registers to indicate which matrix vector multiplications to calculate.

- 3) In the current implementation, we have used the Co-Processor to accelerate the floating point divisions in the Gaussian elimination method in order to solve the linear systems in the IPM. Since the Gaussian elimination method requires a large portion of computation cost, it is worthwhile to consider more efficient methods like LU decomposition and investigate the potential for acceleration using a Co-Processor. In addition, this could alleviate the numerical errors which often arise during the solving of linear systems.
- 4) In MPC applications, the constraints applied to the input or outputs or internal states of the system results in a sparse J which maybe exploited to reduce the computational complexity of solving the linear systems.

References

- [1] K.V.Ling, B.F.Wu and J.M.Maciejowski, “Embedded Model Predictive Control (MPC) using a FPGA” in Proceedings of the 17th IFAC World Congress, 2008, pp. 15250-15255
- [2] I. Gonazles, S. Lopez-Buedo, F.J. Gomez-Arribas, “Implementation of secure applications in self-reconfigurable systems”, Microprocessors and Microsystems, Vol 32, Issue 1, February 2008, pp. 23-32
- [3] T. Perez, G. Goodwin, and C. Tzeng, “Model predictive rudder roll stabilization control for ships,” in Proc. 5th IFAC Conf. on Manoeuvring and Control of Marine Craft, Aalborg, Denmark, Jan 2000.
- [4] P. Kaczynski and P. Domanski, “Evolutionary predictive controller for ship control,” in ICSC Congress on Computational Intelligence Methods and Applications, Istanbul, Turkey, 2005.
- [5] R. Murray, J. Hauser, A. Jadabie, M. Milam, N. Petit, W. Dunbar, and R. Franz, “Online control customization via optimization-based control,” in Software-Enabled Control, T. Samad and G. Balas, Eds. IEEE Press and Wiley, 2003.
- [6] I. Skrjanc, “A decomposed-model predictive functional control approach to air-vehicle pitch-angle control,” Journal of Intelligent and Robotic Systems: Theory and Applications, vol. 48, no. 1, pp. 115–127, 2007.
- [7] A. Richards and J. How., “Model predictive control of vehicle maneuvers with guaranteed completion time and robust feasibility,” in Proc. American Control Conference, Denver, June 2003.

-
- [8] M. Morari, M. Baotic, and F. Borrelli, "Hybrid systems modeling and control," *European Journal of Control*, vol. 9, no. 2-3, pp. 177–189, 2003.
- [9] B. Kim, D. Neculescu, and J. Sasiadek, "Model predictive control of an autonomous vehicle," in *IEEE/ASME International Conference on Advanced Intelligent Mechatronics Proceedings*, Piscataway, NJ, USA, 2001.
- [10] J.M. Saul, "Hardware Software Codesign for FPGA-based systems" In *Proceedings of the 32nd Hawaii International Conference on System Sciences*, Jan 1999
- [11] L. G. Bleris and M. V. Kothare, "Real-time implementation of model predictive control," in *American Control Conference*, Portland, OR, USA, June 2005.
- [12] L. G. Bleris, P.D. Vouzis, M.G.Arnold, M. V. Kothare, "A Co-Processor FPGA platform for the implementation of the Real-Time Model Predictive Control", *American Control Conference 2006 (ACC' 2006)*, Minneapolis, Minnesota, 14-16 June 2006.
- [13] P. Vouzis, L.G. Bleris, M.V Kothare, and M. G. Arnold "Towards a Co-design Implementation of a System for Model Predictive Control", 2005 *AICHE Annayak Meeting*, Cincinnati, OH, November 2005.
- [14] P. Zicari, P. Corsonello, S. Perri, G. Cocorullo, "A Matrix product accelerator for field programmable systems on chip", *Microprocessors and Microsystems*, Volume 32, Issue 2, March 2008.
- [15] Wright, S.J, "Applying new optimization algorithms to model predictive control", *Chemical Process Control-V*, CACHE, *AICHE Symposium Series No. 316*, Vol. 93, pp 147-155, (1997)

[16] 'Virtex-II Pro and Virtex-II Pro X FPGA : Complete data sheet' can found on-line at www.xilinx.com

[17] Microblaze Processor Reference Guide, Embedded Development Kit EDK 7.1i, Jan 20, 2005

[18] Xilinx University Program Virtex II Pro Development System Hardware Reference Manual pp15.

[19] Deso Sima, Terence Fountain, Peter Kacsuk, 'Advanced Computer Architecture, a design state space approach'. pp120-131, 1997, Addison Wesley Publishing Co.

[20] Fast Simplex Bus Product Specification(v2.10a) DS449 Nov 2006. Can found on-line at www.xilinx.com

Appendix A Software Required and the Development Cycle

In the project, peripherals and Co-Processor are being built around the selected reconfigurable processor microcontroller (Microblaze). The design of the peripherals and Co-Processor is implemented in VHDL which is later translated into a bit file and downloaded into the FPGA.

In this section, the tools required and the FPGA design cycle will be briefly described.

A.1 Using the Embedded Development Kit (EDK)

The Embedded Development Kit is software from Xilinx which contains

- a) existing embedded IP which can be instantiated for use with the Microblaze core
- b) board support packages, and drivers for the existing IP cores
- c) the Xilinx Platform Studio, a GUI based software to assist in hardware (building of soft cores) and software development around Microblaze
- d) Software Development Kit, a dedicated GUI based software for software development

For the project, EDK9.1i will be used.

A.2 The Xilinx Integrated Software Environment (ISE)

The Integrated Software Environment (ISE) is a software provided by Xilinx for FPGA development. It does the following:

- a) translates the HDL into a generic hardware representation of the design (in logic gates)

- b) maps the design to specific cells in the CLBs and IOBs based on the constraints given by the user. The constraints refer to timing, and position of where the IOs should be pin-locked to in the design
- c) does a layout of the generic hardware design onto the FPGA

in the end stage, a bit file is generated by the software which can be conveniently downloaded into the FPGA.

The ISE allows one to customize any portions of the FPGA flow as required. This is not supported in Xilinx Platform Studio. For the project, ISE9.1i will be used.

A.3 Using Modelsim as a functional verification tool

The Modelsim simulator from Mentor Graphics is used in functional verification of the peripherals designed in VHDL and also to verify the firmware written in C.

This simulator is used as Xilinx platform studio has precompiled libraries for Modelsim. These libraries include the behavioral models of both Microblaze and the provided embedded IP. In addition, Xilinx Platform Studio's automatic generation of compile and run scripts facilitates easier usage of the tool. For the project, Modelsim 6.0d will be used.

A.4 The Design Cycle

Having provided some background to the software tools used in the implementation stage of the project, the design cycle to be adopted in this project is described in the diagram below.

Figure A.1: The EDK/ISE FPGA design flow[17]

In a nutshell, co-hardware and software development will be developed in EDK's Xilinx Platform Studio. The design will be largely exported to ISE for FPGA bit file generation and implementation. XPS will also be used in auto-generation of compile and run scripts for use in the modelsim simulator for functional verification of the customized blocks used in the project.

Appendix B Assembly instructions for $y=Ax$ operation.

In this section, we study the assembly codes corresponding to the C code from Figures 2.12 and 2.13. For ease of understanding, the C code is presented along with the generated assembly code.

<pre>do { result[i] = dotprod(col_size,&op1[i*col_size],&op2[0],1,1);</pre>				
row	instruction	description	Operation details	Clock cycle count
1	lwi r3, r19, 36	R3 = m[r19+36]	Read from memory and store index i in register R3. Index i is originally stored at memory location address R19+36	2
2	muli r4, r3, 4	R4 = r3 * 4	Generate and store into register R4, the address corresponding to index i	3
3	addik r3, r19, 28	R3=r19+28	calculate start pointer address for result. store result in r3	1
4	addk r3, r4, r3	R3 = r3+r4	Generate address for result[i]	1
5	addik r22, r3, 36	R22=r3+36	Store result address of current result index in register r22	1
6	lwi r4, r19, 36	R4 = m[r19+36]	store index i in register R4	2
7	lwi r3, r0, 7796	R3 = m[7796]	Get col_size value, store in register R3	2
8	mul r3, r4, r3	R3= r4*r3	R3= i*col_size	3
9	muli r4, r3, 4	R4=r3*4	Generate corresponding address offset value for i*col_size	3
10	addik r3, r0, 3696	R3=0+3696	determine op1 Start address	1
11	addk r3, r4, r3	R3 = r4+r3	Calculate address of op1[i*col_size]	1
12	lwi r5, r0, 7796		Read memory for variable col_size and store result in register	2

			R5	
13	addk r6, r3, r0	R6 = r3	Save op1 start address pointed by index	1
14	addik r7, r0, 5744	R7 = 5744	Get Op2 Start addr	1
15	addik r8, r0, 1	R8 = 1	2 nd last variable into operand dotprod stored in R8	1
16	addik r9, r0, 1	R9 = 1	Last variable required in operand dotprod stored in R9	1
17	brlid r15, 1244		Unconditional branch to dotprod	3
18	or r0, r0, r0	NOP		1
19	swi r3, r22, 0	M[r22]=r3	Store returned result into result array	2
	total			32
i++;				
20	lwi r3, r19, 36	R3= m[r19+36]	Get i variable from memory	2
21	addik r3, r3, 1	Increment I count		1
22	swi r3, r19, 36	M[r19+36]=r3	Store i value back to memory	2
	total			5
} while (i!=row_size);				
23	lwi r4, r19, 36	R4 =M[r19+36]	Get i variable from memory	2
24	lwi r3, r0, 7792	R3=m[r0+7992]	Get row size variable	2
25	rsubk r18, r3, r4	R18=r3-r4	Subtract i from row_size variable	1
26	beqi r18, 8		If 0, this means that row_size equals to I, and a jump out of the loop is required. The next instruction below is thus bypassed.	1 if branch not taken. 3 if branch is taken
27	bri -104		Jump unconditionally to start of loop. This happens only if the previous condition is not met.	3
				9 (8 last time)

Figure B.1: In-lined C and Assembly source code for matrix vector multiplication

In addition, the assembly code generated for the dotprod function C source code is as follows:

float dotprod(Xuint32 n,float *a,float *b, Xuint32 inc1, Xuint32 inc2) {				
// start of dotprod function before while loop				
row		description	Operation details	Clock cycle count
1	addik r1, r1, -32	R1=R1-32	Decrement stack pointer	1
2	swi r19, r1, 28	M[R1+28] = R19	Save previous variable base pointer onto stack	2
3	addik r1, r1, -32	R1=R1-32	Decrement stack pointer	1
4	swi r19, r1, 28	M[R1+28] = R19	Save previous variable pointer onto another portion of the stack	2
5	addk r19, r1, r0	R19=R1+R0	Load instead R19 with R1 value	1
6	swi r5, r19, 36	M[r19+36]=r5	Save col_size variable address into memory location M[R19+36]	2
7	swi r6, r19, 40	M[r19+40]=r6	Save start address of variable a into M[r19+40]. Variable a is start address of op1[i*col_size]	2
8	swi r7, r19, 44	M[r19+44]=r7	Save start address of variable b into M[r19+44]. Variable b is start address of op2[0]	2
9	swi r8, r19, 48	M[r19+48]=r8	Save inc1 value into M[r19+48]	2
10	swi r9, r19, 52	M[r19+52]=r9	Save inc2 value into M[r19+52]	2
	Total			17
float adotb,sum, outrow;				

Xuint32 i,j,q;

sum = 0;

11	addk r3, r0, r0	R3=r0+r0	Clear R3 register to 0	1
12	swi r3, r19, 8	M[r19+8]=r3	Initialize sum variable to 0	2
	Total			3

q = n*inc1;

13	lwi r4, r19, 36	R4= M[r19+36]	Get n value (this was previously col_size value)	2
14	lwi r3, r19, 48	R3= M[r19+48]	Get inc1 value from memory	2
15	mul r3, r4, r3	R3 = r4 * r3	Multiply n*inc1, store result in R3	6
16	swi r3, r19, 24	M[r19+24] = r3	Save the multiply result in M[R19+24]. This memory location now contains q variable.	2
	Total			12

j = inc2;

17	lwi r3, r19, 52	R3= M[r19+52]	Get inc2 value from memory	2
18	swi r3, r19, 20	M[r19+20]=r3	Store in in M[R19+20]. This memory location now contains j variable	2
	total			4

//adotb = (double)a[0]*(double)b[0];

adotb = a[0]*b[0];

19	lwi r3, r19, 40	R3=m[r19+40]	Read out start address of a.	2
20	lwi r4, r19, 44	R4=m[r19+44]	Read out start address of b	2
21	lwi r5, r3, 0	R5 = m[r3]	Get a[0] value from memory	2
22	lwi r3, r4, 0	R3 = m[r4]	Get b[0] value from memory	2
23	fmul r3, r5, r3	R3=r3*r5	R3 = a[0]*b[0]	6
24	swi r3, r19, 4	M[r19+4]=r3	Store result in	2

			M[r19+4]. This memory location now holds adotb variable	
	total			16
i = inc1;				
25	lwi r3, r19, 48	R3=m[r19+48]	Read out inc1 variable	2
26	swi r3, r19, 16	M[r19+16]=r3	Transfer the value to M[r19+16]. This memory location now holds local i variable	2
	total			4
// while loop portion of dotprod function				
while (i<q){				
27	lwi r4, r19, 16	R4 = M[R19+16]	Read out i	2
28	lwi r3, r19, 24	R3 = M[R19+24]	Read out q	2
29	cmpu r18, r3, r4		Compare q and i	2
30	bgei r18, 104		If r18 > 0, jump to end of loop	1 if branch not taken; 2 if branch taken
	total			7 or 8
sum = sum + adotb; //myFloatAdd(sum,adotb, &sum);				
31	lwi r4, r19, 8	R4 = M[R19+8]	Read out sum into r4	2
32	lwi r3, r19, 4	R3 = M[R19+4]	Read out adotb into r3	2
33	fadd r3, r4, r3	R3= R3+R4	Add the two floating point numbers	6
34	swi r3, r19, 8	M[R19+8]=R3	Store add result back to sum variable	2
	total			12
adotb = a[i]*b[j]; //myFloatMult(a[i], b[j], &adotb);				
35	lwi r3, r19, 16	R3 = m[r19+16]	Read and store i variable into r3	2
36	muli r4, r3, 4	R4 = R3 *4	Generate the corresponding address offset for index i	3

37	lwi	r3, r19, 40	$R3 = m[r19+40]$	Read out start address of a	2
38	addk	r5, r4, r3	$R5 = r4 + r3$	Generate the true address of a[j]	1
39	lwi	r3, r19, 20	$R3 = m[r19+20]$	Read out and store j variable into r3	2
40	muli	r4, r3, 4	$R4 = j * 4$	Generate the corresponding address offset for index j	3
41	lwi	r3, r19, 44	$R3 = m[r19+44]$	Read out start address of b	2
42	addk	r3, r4, r3	$R3 = R3+R4$	Generate the true address of a[j]	1
43	lwi	r4, r5, 0	$R4 = m[r5]$	read out current a[i] value into R4	2
44	lwi	r3, r3, 0	$R3 = m[r3]$	read out current b[i] value into R3	2
45	fmul	r3, r4, r3	$R3 = R3 * R4$	$R3 = a[i] * b[i]$	6
46	swi	r3, r19, 4	$M[R19+4] = R3$	store mul result into adotb variable	2
	total				28

```
//adotb = (double)a[i]*(double)b[j]; //myFloatMult(a[i], b[j], &adotb);
```

```
j+=inc2;
```

47	lwi	r4, r19, 20	$R4 = m[r19+20]$	read out j variable into R4	2
48	lwi	r3, r19, 52	$R3 = m[r19+52]$	read out inc2 variable	2
49	addk	r3, r4, r3	$R3 = r4 + r3$	$R3 = j + inc2$	1
50	swi	r3, r19, 20	$M[r19+20] = r3$	store back j value into memory location	2
	total				7

```
i+=inc1;
```

51	lwi	r4, r19, 16	$R4 = m[r19+16]$	read out i variable into R4	2
52	lwi	r3, r19, 48	$R3 = m[r19+48]$	read out inc1 variable	2
53	addk	r3, r4, r3	$R3 = r3 + r4$	$R3 = i + inc1$	1
54	swi	r3, r19, 16	$M[r19+16] = r3$	store back i value into memory location	2
55	bri	-112// 8ac		Branch unconditionally back to start of the while loop	3
					10

}				
// Portion of code after while loop till dotprod function branches back to caller function.				
outrow = sum + adotb;				
56	lwi r4, r19, 8	R4=m[r19+8]	read out sum variable	2
57	lwi r3, r19, 4	R3=m[r19+4]	read out adotb variable	2
58	fadd r3, r4, r3	R3 = r3+r4	R3 = adotb + sum	6
59	swi r3, r19, 12	M[r19+12]=r3	Save sum into M[R19+12]. This memory location holds outrow variable	2
				12
return outrow;				
60	lwi r3, r19, 12	R3= M[r19+12]	First load back variable into the register to be returned back to the caller function.	2
				2
} //void dotprod(Xuint32 n,float *a,float *b,float *outrow, Xuint32 inc1, Xuint32 inc2)				
61	addk r1, r19, r0	R1 = R19+R0		1
62	lwi r19, r1, 28	M[r1+28]=r19	Update memory location	2
63	addik r1, r1, 32	R1 = r1 +32	Increment stack pointer to get back caller function variables later	1
64	rtsd r15, 8		Return back to caller function	2
total				6

Figure B.2: In-lined C and Assembly source code for dot-product function

Appendix C C code for Step 2

This section presents the section of C code implementation for step 2. The lines with green comments show the operations that can be accelerated by the Co-Processors

```

i=0;
do{
    dell = dotprod(num,&matJ[i*knum],&z[0],1,1); //y=Ax operation
    tp = g[i] - dell;
    Artp2[i] = tp;
    tp1 = lam[i]*tp;
    tp2 = tp1 - vtmp;
    rgt[i]=tp2;
    i++;
}while (i!=mc);

i=0;

do{
    tp = invT[i]* lam[i];
    Artp4[i] = tp;
    i++;
}while(i!=mc);
j=0;

do {
    jj=0;i=0;
    do{
        tp1 = Artp4[i] * matJ[i*knum+j];
        Artp5[i]= tp1;
        i++;
    }while(i!=mc);

    do{
        dell = dotprod(mc, &matJ[jj], &Artp5[0], num, 1); // y=Ax operation
        mcmc = jj*kmc + j;
        tp1 = matQ[mcmc]+dell;
        matCat[mcmc] = tp1;
        jj++;
    }while(jj!=num);
    j++;
}while(j!=num);

i=0;

do{
    tp = rgt[i] * invT[i];
    Artp7[i] = tp;
    tp1 = lam[i] - tp;
    Artp5[i]=tp1;

```

```

    i++;
}while(i!=mc);

j=0;
do{
    dell = dotprod(mc, &matJ[j], &Artp5[0], num, 1); // y=Ax operation
    tp = dotprod(num, &matQ[j*kmc], &z[0], 1,1 ); // y=Ax operation
    temp = -1 * c[j];
    tp1 = temp - dell;
    temp = tp1 - tp;
    Artp11[j] = temp;
    j++;
}while(j!=num);
j=0;
Gaussian_Inverse(); //floating point division operation.
do{
    // y=Ax operation.
    dz[j] = dotprod(num, &matCat[j*kmc], &Artp11[0], 1, 1);
    j++;
}while(j!=num);

i=0;

do{
    // y=Ax operation.
    dell = dotprod(num, &matJ[(i)*knum], &dz[0], 1,1);
    Artp12[i]=dell;
    limitcheck(&dell);
    tp_z1 = Artp4[i];
    limitcheck(&tp_z1);
    tp = dell * tp_z1;
    tp1 = tp - Artp7[i];
    dlam[i] = tp1;
    i++;
}while(i!=mc);

i = 0;

do{
    tp = Artp2[i]-t[i];
    temp = tp-Artp12[i];
    dt[i]=temp;
    i++;
}while ( i != mc );

```

Figure C.1 C code for Step 2

Appendix D Co-Processor Source code

D.1 Matrix Vector Multiplication Co-Processor Source Code

```

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

library unisim;
use unisim.all;

entity fsl_matvecprod is
  port (
    clk : in std_logic;
    res : in std_logic;
    --from OPB bus configuration peripheral
    op1_row_size_i : in std_logic_vector(7 downto 0);
    op1_col_size_i : in std_logic_vector(7 downto 0);
    -- Slave FSL Signals
    FSL0_S_Clk    : out std_logic;
    FSL0_S_Read   : out std_logic;
    FSL0_S_Data   : in  std_logic_vector(0 to 31);
    FSL0_S_Control : in  std_logic;
    FSL0_S_Exists : in  std_logic;

    -- Master FSL Signals
    FSL0_M_Clk    : out std_logic;
    FSL0_M_Write  : out std_logic;
    FSL0_M_Data   : out std_logic_vector(0 to 31);
    FSL0_M_Control : out std_logic;
    FSL0_M_Full   : in  std_logic
  );
end fsl_matvecprod;

architecture rtl of fsl_matvecprod is
  constant MUL_LATENCY : integer := 3;
  constant ADD_LATENCY : integer := 4;
  component bram_512_32
    port (
      clka : in  std_logic;
      dina : in  std_logic_vector(31 downto 0);
      addra : in  std_logic_vector(8 downto 0);

```



```

ena : in std_logic;
wea : in std_logic_vector(0 downto 0);
douta : out std_logic_vector(31 downto 0);
clkb : in std_logic;
dinb : in std_logic_vector(31 downto 0);
addrb : in std_logic_vector(8 downto 0);
enb : in std_logic;
web : in std_logic_vector(0 downto 0);
doutb : out std_logic_VECTOR(31 downto 0));
end component;
component bram_16_32
port (
  clka : in std_logic;
  dina : in std_logic_vector(31 downto 0);
  addra : in std_logic_vector(3 downto 0);
  ena : in std_logic;
  wea : in std_logic_vector(0 downto 0);
  douta : out std_logic_vector(31 downto 0);
  clkb : in std_logic;
  dinb : in std_logic_vector(31 downto 0);
  addrb : in std_logic_vector(3 downto 0);
  enb : in std_logic;
  web : in std_logic_vector(0 downto 0);
  doutb : out std_logic_VECTOR(31 downto 0));
end component;
component fifo_512_32
port (
  clk: in std_logic;
  din: in std_logic_VECTOR(31 downto 0);
  rd_en: in std_logic;
  srst: in std_logic;
  wr_en: in std_logic;
  dout: out std_logic_VECTOR(31 downto 0);
  empty: out std_logic;
  full: out std_logic);
end component;
component fp_mul32
port (
  a      : in std_logic_vector(31 downto 0);
  b      : in std_logic_vector(31 downto 0);
  operation_nd : in std_logic;
  clk    : in std_logic;
  sclr   : in std_logic;
  result : out std_logic_VECTOR(31 downto 0);
  overflow : out std_logic;

```

```

    rdy: out std_logic);
end component;
component fp_add32
port (
    a      : in  std_logic_vector(31 downto 0);
    b      : in  std_logic_vector(31 downto 0);
    operation_nd : in  std_logic;
    clk     : in  std_logic;
    sclr    : in  std_logic;
    result   : out std_logic_VECTOR(31 downto 0);
    overflow  : out std_logic;
    rdy: out std_logic);
end component;
attribute box_type : string;
attribute box_type of bram_512_32 : component is "black_box";
attribute box_type of bram_16_32 : component is "black_box";
attribute box_type of fifo_512_32 : component is "black_box";
attribute box_type of fp_mul32 : component is "black_box";
attribute box_type of fp_add32 : component is "black_box";
signal FSL_S_Data : std_logic_vector(31 downto 0);
type mul_state_t is (IDLE, RD_MUL_OP2, RD_MUL_OP1);
signal curr_mul_state, next_mul_state : mul_state_t;
signal load_mat_size : std_logic;
signal inc_row_proc, inc_col_proc : std_logic;
signal clr_row_proc, clr_col_proc : std_logic;
signal op1_row_size, row_proc : std_logic_vector(7 downto 0);
signal op1_col_size, col_proc : std_logic_vector(7 downto 0);
signal mul_op1, mul_op2_reg, mul_result : std_logic_vector(31 downto 0);
signal mul_op_nd, mul_rdy : std_logic;
signal mul_op_nd_buffer : std_logic_vector(MUL_LATENCY-2 downto 0);
signal ld_mul_op2 : std_logic;
signal add_op1, add_op2, add_result, add_result_reg : std_logic_vector(31 downto 0);
signal add_op_nd, add_rdy : std_logic;
signal add_op_enable : std_logic;
--fifo signals
signal wr_fifo, rd_fifo : std_logic;
signal fifo_din : std_logic_vector(31 downto 0);
signal wr_output_fifo : std_logic;
signal rd_output_fifo : std_logic;
signal output_fifo_empty : std_logic;
signal output_fifo_dout, stored_result : std_logic_vector(31 downto 0);
type output_state_t is (OUTPUT_ADDER_DATA,
OUTPUT_FIFO_NOT_EMPTY, WAIT_TILL_BUS_AVAIL);
signal curr_output_state, next_output_state : output_state_t;
signal rd_output_fifo_reg, FSL0_M_Write_s : std_logic;

```

```

signal output_data_fr_adder, output_data_fr_fifo : std_logic;
signal res_s : std_logic;
constant BUFF_SIZE : integer := ADD_LATENCY+MUL_LATENCY-1;
type proc_buffer_t is array (BUFF_SIZE downto 0) of std_logic_vector(7 downto 0);
signal row_proc_buffer,col_proc_buffer : proc_buffer_t;
begin -- rtl

    res_s <= not res;          --for required active high reset

    endian_conversion_loop: for i in 0 to 31 generate
        FSL_S_Data(31-i) <= FSL0_S_Data(i);
    end generate endian_conversion_loop;

    process (clk, res)
    begin -- process
        if clk'event and clk = '1' then -- rising clock edge
            if res = '0' then
                op1_row_size <= (others => '0');
                op1_col_size <= (others => '0');
                row_proc <= (others => '0');
                col_proc <= (others => '0');
                mul_op2_reg <= (others => '0');
                curr_mul_state <= IDLE;
                mul_op_nd_buffer <= (others => '0');
                add_op_enable <= '0';
                for i in 0 to BUFF_SIZE loop
                    row_proc_buffer(i) <= (others => '0');
                    col_proc_buffer(i) <= (others => '0');
                end loop; -- i
            else
                curr_mul_state <= next_mul_state;
                row_proc_buffer(BUFF_SIZE downto 1) <= row_proc_buffer(BUFF_SIZE-1
downto 0);
                row_proc_buffer(0) <= row_proc;
                col_proc_buffer(BUFF_SIZE downto 1) <= col_proc_buffer(BUFF_SIZE-1 downto
0);
                col_proc_buffer(0) <= col_proc;

                if col_proc = "00000000" then
                    if row_proc_buffer(BUFF_SIZE) = op1_row_size and add_rdy = '1' then
                        add_op_enable <= '0';
                    end if;
                else
                    if ld_mul_op2 = '1' then
                        add_op_enable <= '1';
                    end if;
                end if;
            end if;
        end if;
    end process;
end rtl;

```

```

        end if;
    end if;
    for i in 0 to MUL_LATENCY-2 loop
        mul_op_nd_buffer(0) <= mul_op_nd;
        mul_op_nd_buffer(MUL_LATENCY-2 downto 1) <=
mul_op_nd_buffer(MUL_LATENCY-3 downto 0);
    end loop; -- i
    if load_mat_size = '1' then
        op1_row_size <= op1_row_size_i+"11111111";
        op1_col_size <= op1_col_size_i+"11111111";
    end if;
    if ld_mul_op2='1' then
        mul_op2_reg <= FSL_S_Data;
    end if;
    if clr_row_proc='1' then
        row_proc <= (others => '0');
    elsif inc_row_proc='1' then
        row_proc <= row_proc+1;
    end if;
    if clr_col_proc='1' then
        col_proc <= (others => '0');
    elsif inc_col_proc='1' then
        col_proc <= col_proc+1;
    end if;
    end if;
end if;
end process;

process (curr_mul_state, FSL0_S_Exists, row_proc, col_proc, op1_row_size,
op1_col_size)
begin -- process
    next_mul_state <= curr_mul_state;
    load_mat_size <= '0';
    FSL0_S_Read <= '0';
    mul_op_nd <= '0';
    ld_mul_op2 <= '0';
    clr_row_proc <= '0';
    clr_col_proc <= '0';
    inc_row_proc <= '0';
    inc_col_proc <= '0';
    case curr_mul_state is
        when IDLE =>
            load_mat_size <= '1';

```

```

    if FSL0_S_Exists = '1' then
        FSL0_S_Read <= '1';
        next_mul_state <= RD_MUL_OP1;
        ld_mul_op2 <= '1';
    end if;
    when RD_MUL_OP2 =>
        if FSL0_S_Exists = '1' then
            FSL0_S_Read <= '1';
            next_mul_state <= RD_MUL_OP1;
            ld_mul_op2 <= '1';
        end if;
    when RD_MUL_OP1 =>
        if FSL0_S_Exists = '1' then
            FSL0_S_Read <= '1';
            mul_op_nd <= '1';
            inc_row_proc <= '1';
            if row_proc = op1_row_size then
                clr_row_proc <= '1';
                if col_proc = op1_col_size then
                    clr_col_proc <= '1';
                    next_mul_state <= IDLE;
                else
                    inc_col_proc <= '1';
                    next_mul_state <= RD_MUL_OP2;
                end if;
            end if;
        end if;
    when others => null;
end case;
end process;

mul_op1 <= FSL_S_Data;

i_fp_mul32 : fp_mul32
port map (
    a      => mul_op1,
    b      => mul_op2_reg,
    operation_nd => mul_op_nd,
    clk     => clk,
    sclr    => res_s,
    result  => mul_result,
    overflow => open,
    rdy     => mul_rdy);

add_op1 <= mul_result;

```

```

add_op2 <= stored_result;
add_op_nd <= mul_rdy when add_op_enable = '1'
    else '0';

i_fp_add32 : fp_add32
port map (
    a      => add_op1,
    b      => add_op2,
    operation_nd => add_op_nd,
    clk     => clk,
    sclr    => res_s,
    result  => add_result,
    overflow => open,
    rdy     => add_rdy);
rd_fifo <= mul_op_nd_buffer(MUL_LATENCY-2) when add_op_enable = '1'
    else '0';
wr_fifo <= '0' when col_proc_buffer(BUFF_SIZE) = op1_col_size
    else add_rdy when add_op_enable = '1'
    else mul_rdy;
fifo_din <= add_result when add_op_enable = '1'
    else mul_result;
i_data_buffer : fifo_512_32
port map (
    clk  => clk,
    din  => fifo_din,
    rd_en => rd_fifo,
    srst => res_s,
    wr_en => wr_fifo,
    dout => stored_result,
    empty => open,
    full  => open
);

--output fifo handling

i_output_buffer : fifo_512_32
port map (
    clk  => clk,
    din  => add_result,
    rd_en => rd_output_fifo,
    srst => res_s,
    wr_en => wr_output_fifo,
    dout => output_fifo_dout,
    empty => output_fifo_empty,
    full  => open

```

```

);
process (clk, res)
begin -- process
  if clk'event and clk = '1' then -- rising clock edge
    if res = '0' then -- asynchronous reset (active low)
      curr_output_state <= OUTPUT_ADDER_DATA;
      rd_output_fifo_reg <= '0';
      output_data_fr_fifo <= '0';
      add_result_reg <= (others => '0');
    else
      add_result_reg <= add_result;
      curr_output_state <= next_output_state;
      rd_output_fifo_reg <= rd_output_fifo;
      output_data_fr_fifo <= rd_output_fifo;
    end if;
  end if;
end process;

FSL0_M_Write <= (FSL0_M_Write_s or rd_output_fifo_reg) when FSL0_M_Full = '0'
  else '0';
FSL0_M_Data <= output_fifo_dout when output_data_fr_adder = '0'
  else add_result;

process (curr_output_state, add_rdy, op1_col_size, col_proc_buffer(BUFF_SIZE),
FSL0_M_Full,
  output_fifo_empty, rd_output_fifo_reg)
begin -- process
  next_output_state <= curr_output_state;
  output_data_fr_adder <= '0';
  wr_output_fifo <= '0';
  rd_output_fifo <= '0';
  FSL0_M_Write_s <= '0'; --to be registered
  case curr_output_state is
    when OUTPUT_ADDER_DATA =>
      if add_rdy = '1' and col_proc_buffer(BUFF_SIZE) = op1_col_size then
        if FSL0_M_Full='1' then
          wr_output_fifo <= '1';
          next_output_state <= OUTPUT_FIFO_NOT_EMPTY;
        else
          FSL0_M_Write_s <= '1';
          output_data_fr_adder <= '1';
        end if;
      end if;
    when OUTPUT_FIFO_NOT_EMPTY =>
      if FSL0_M_Full = '0' then

```

```
        rd_output_fifo <= '1';
    end if;
    if add_rdy = '1' and col_proc_buffer(BUFF_SIZE) = op1_col_size then
        wr_output_fifo <= '1';
    end if;
    if FSL0_M_Full = '1' and rd_output_fifo_reg = '1' then
        next_output_state <= WAIT_TILL_BUS_AVAIL;
    elsif output_fifo_empty = '1' then
        next_output_state <= OUTPUT_ADDER_DATA;
    end if;
    when WAIT_TILL_BUS_AVAIL =>
        if FSL0_M_Full = '0' then
            FSL0_M_Write_s <= '1';
            if output_fifo_empty = '1' then
                next_output_state <= OUTPUT_ADDER_DATA;
            else
                rd_output_fifo <= '1';
                next_output_state <= OUTPUT_FIFO_NOT_EMPTY;
            end if;
        end if;
        if add_rdy = '1' and col_proc_buffer(BUFF_SIZE) = op1_col_size then
            wr_output_fifo <= '1';
        end if;
        when others => null;
    end case;
end process;
end rtl;
```

Figure D.1: VHDL source code for Matrix Vector Multiplier Co-Processor

D.2 Floating point Divider Co-Processor

```

-----
-- fsl_div - entity/architecture pair
-----

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-----
--
--
-- Definition of Ports
-- FSL_Clk      : Synchronous clock
-- FSL_Rst      : System reset, should always come from FSL bus
-- FSL_S_Clk    : Slave asynchronous clock
-- FSL_S_Read   : Read signal, requiring next available input to be read
-- FSL_S_Data   : Input data
-- FSL_S_CONTROL : Control Bit, indicating the input data are control word
-- FSL_S_Exists : Data Exist Bit, indicating data exist in the input FSL bus
-- FSL_M_Clk    : Master asynchronous clock
-- FSL_M_Write  : Write signal, enabling writing to output FSL bus
-- FSL_M_Data   : Output data
-- FSL_M_Control : Control Bit, indicating the output data are control word
-- FSL_M_Full   : Full Bit, indicating output FSL bus is full
--
-----

-- Entity Section
-----

entity fsl_div is
  port
  (
    clk   : in  std_logic;
    res_n : in  std_logic;
    -- DO NOT EDIT BELOW THIS LINE -----
    -- Bus protocol ports, do not add or delete.
    FSL_Clk   : in  std_logic;
    FSL_Rst   : in  std_logic;

```

```

        FSL_S_Clk   : out   std_logic;
        FSL_S_Read  : out   std_logic;
        FSL_S_Data  : in    std_logic_vector(0 to 31);
        FSL_S_Control : in    std_logic;
        FSL_S_Exists : in    std_logic;
        FSL_M_Clk   : out   std_logic;
        FSL_M_Write : out   std_logic;
        FSL_M_Data  : out   std_logic_vector(0 to 31);
        FSL_M_Control : out   std_logic;
        FSL_M_Full  : in    std_logic
        -- DO NOT EDIT ABOVE THIS LINE -----
    );

attribute SIGIS : string;
attribute SIGIS of FSL_Clk : signal is "Clk";
attribute SIGIS of FSL_S_Clk : signal is "Clk";
attribute SIGIS of FSL_M_Clk : signal is "Clk";

end fsl_div;

-----
-- Architecture Section
-----

architecture EXAMPLE of fsl_div is
    component fp_div32
        port (
            rdy          : out std_logic;
            overflow      : out std_logic;
            invalid_op    : out std_logic;
            operation_nd  : in  std_logic;
            clk           : in  std_logic;
            underflow     : out std_logic;
            divide_by_zero : out std_logic;
            a             : in  std_logic_vector ( 31 downto 0 );
            b             : in  std_logic_vector ( 31 downto 0 );
            result        : out STD_LOGIC_VECTOR ( 31 downto 0 )
        );
    end component;
    component fifo_128_32
        port (
            clk: in std_logic;
            din: in std_logic_VECTOR(31 downto 0);
            rd_en: in std_logic;
            srst: in std_logic;

```

```

    wr_en: in std_logic;
    dout: out std_logic_VECTOR(31 downto 0);
    empty: out std_logic;
    full: out std_logic;
end component;

attribute box_type : string;
attribute box_type of fp_div32 : component is "black_box";
attribute box_type of fifo_128_32 : component is "black_box";
type state_t is (IDLE,RD_OP2);
signal curr_state, next_state : state_t;

type output_state_t is (IDLE, OUTPUT_FIFO_NOT_EMPTY,
WAIT_TILL_BUS_AVAIL);
signal curr_output_state, next_output_state : output_state_t;

--div signals
signal div_rdy : std_logic;
signal start_div_operation : std_logic;
signal div_ov : std_logic;
signal invalid_op : std_logic;
signal underflow : std_logic;
signal divide_by_zero : std_logic;
signal ld_operand1 : std_logic;
signal operand1_reg : std_logic_vector(31 downto 0);
signal operand2 : std_logic_vector(31 downto 0);
signal div_result : std_logic_vector(31 downto 0);
signal res : std_logic;
signal FSL_M_Write_s : std_logic;
--fifo signals
signal rd_fifo, wr_fifo, rd_fifo_reg : std_logic;
signal fifo_empty : std_logic;
signal stored_result : std_logic_vector(31 downto 0);
begin

    res <= not res_n;
    FSL_M_Control <= '0';
    FSL_M_Write <= FSL_M_Write_s when FSL_M_Full = '0'
        else '0';
    FSL_M_Data <= stored_result when curr_output_state /= IDLE --rd_fifo_reg = '1'
        else div_result;
    process (clk, res_n)
    begin -- process
        if clk'event and clk = '1' then -- rising clock edge
            if res_n = '0' then -- asynchronous reset (active low)

```

```

curr_state <= IDLE;
curr_output_state <= IDLE;
operand1_reg <= (others => '0');
rd_fifo_reg <= '0';
else
  --input state controls
  curr_state <= next_state;
  if ld_operand1 = '1' then
    operand1_reg <= FSL_S_Data;
  end if;
  curr_output_state <= next_output_state;
  rd_fifo_reg <= rd_fifo;
end if;
end if;
end process;

operand2 <= FSL_S_Data;

process (curr_state, FSL_S_Exists)
begin -- process
  next_state <= curr_state;
  FSL_S_Read <= '0';
  ld_operand1 <= '0';
  start_div_operation <= '0';
  case curr_state is
    when IDLE =>
      if FSL_S_Exists = '1' then
        FSL_S_Read <= '1';
        next_state <= RD_OP2;
        ld_operand1 <= '1';
      end if;
    when RD_OP2 =>
      if FSL_S_Exists = '1' then
        FSL_S_Read <= '1';
        start_div_operation <= '1';
        next_state <= IDLE;
      end if;
    when others => null;
  end case;
end process;

process (curr_output_state, FSL_M_Full, div_rdy, rd_fifo_reg)
begin -- process
  next_output_state <= curr_output_state;

```

```

FSL_M_Write_s <= '0';
wr_fifo <= '0';
rd_fifo <= '0';
case curr_output_state is
when IDLE =>
  if div_rdy = '1' then
    if FSL_M_Full = '0' then
      FSL_M_Write_s <= '1';
    else
      wr_fifo <= '1';
      next_output_state <= OUTPUT_FIFO_NOT_EMPTY;
    end if;
  end if;
when OUTPUT_FIFO_NOT_EMPTY =>
  if div_rdy = '1' then
    wr_fifo <= '1';
  end if;
  if FSL_M_Full = '0' then
    rd_fifo <= '1';
  end if;
  if rd_fifo_reg = '1' then
    FSL_M_Write_s <= '1';
  end if;
  if FSL_M_Full = '1' and rd_fifo_reg = '1' then
    next_output_state <= WAIT_TILL_BUS_AVAIL;
  elsif fifo_empty = '1' then
    next_output_state <= IDLE;
  end if;
when WAIT_TILL_BUS_AVAIL =>
  --this state to cover scenario when FSL_M_Full is asserted
  --when one clock cycle b4, rd_fifo has been read
  --needed cos, the next time FSL_M_Full=0 again; we will need
  --just to assert the FSL_M_Write signal again, can already.
  if FSL_M_Full = '0' then
    FSL_M_Write_s <= '1';
    if fifo_empty = '1' then
      next_output_state <= IDLE;
    else
      rd_fifo <= '1';
      next_output_state <= OUTPUT_FIFO_NOT_EMPTY;
    end if;
  end if;
  if div_rdy = '1' then
    wr_fifo <= '1';
  end if;

```

```
        when others => null;
    end case;
end process;

i_fp_div32 : fp_div32
port map (
    rdy          => div_rdy,
    overflow      => div_ov,
    invalid_op    => invalid_op,
    operation_nd  => start_div_operation,
    clk          => clk,
    underflow     => underflow,
    divide_by_zero => divide_by_zero,
    a            => operand1_reg,
    b            => operand2,
    result       => div_result);

i_output_buffer : fifo_128_32
port map (
    clk => clk,
    din => div_result,
    rd_en => rd_fifo,
    srst => res,
    wr_en => wr_fifo,
    dout => stored_result,
    empty => fifo_empty,
    full => open
);

end architecture EXAMPLE;
```

Figure D.2: VHDL source code for floating point divider Co-Processor