



Norwegian University of
Science and Technology

Co-design implementation of FPGA hardware acceleration of DNA motif identification

Elisabeth Linvåg

Master of Science in Computer Science

Submission date: June 2008

Supervisor: Morten Hartmann, IDI

Co-supervisor: Dag Kristian Rognlien, Sintef

Problem Description

Pattern matching in bioinformatics is a discipline in sturdy growth, and has a great need for searching through large amounts of data. At NTNU, an FPGA prototype specified in VHDL has been developed, which identifies short motifs or patterns in genetic data using Position-Weight Matrices (PWM). The prototype is designed for running on a Cray XD1 supercomputer.

Use of VHDL and similar hardware specification languages is complicated, and success requires thorough knowledge and experience. The design process is also quite time-consuming. As such, it would benefit the bioinformatics community to be able to use co-design languages that simplifies the design process and make FPGA technology more accessible to scientists without special knowledge of electronic hardware.

The project consists of the following tasks:

Specification and implementation of a Impulse-C based alternative to the existing VHDL-based solution.

Evaluation of easy-of-use of the CoDeveloper environment, and productivity vs. final performance when comparing the Impulse-C solution and the existing VHDL-based solution.

Assignment given: 15. January 2008
Supervisor: Morten Hartmann, IDI

Abstract

Pattern matching in bio-informatics is a discipline in sturdy growth, and has a great need for searching through large amounts of data. At NTNU, a prototype specified in VHDL has been developed for an FPGA-solution identifying short motifs or patterns in genetic data using a Position-Weight Matrix (PWM). But programming FPGAs using VHDL is a complicated and time consuming process that requires intimate knowledge of how hardware works, and the prototype is not yet complete in terms of required functionality. Consequently, a desirable alternative is to make use of co-design languages to facilitate the use of hardware for a software developer, as well as to integrate the environment for development of soft- and hardware.

This thesis deal with specification and implementation of a co-design based alternative to the existing VHDL based solution, as well as an evaluation of productivity vs final performance of the newly developed solution compared to the VHDL based solution. The chosen co-design language is Impulse-C, created by Impulse Accelerated Technologies Inc., which is a co-design language designed for data-flow oriented applications, but with the flexibility to support other programming models as well. The programming model simplifies the expression of highly parallel algorithms through the use of well-defined data communication, message passing and synchronization mechanisms. The affiliated development environment, CoDeveloper, contains tools that allow the FPGA system to be developed and debugged using Impulse-C. The software-to-hardware compiler and optimizer translates C-language processes to (RTL) VHDL code, while optimizing the generated logic and identifying opportunities for parallelism. Ease-of-use for the CoDeveloper environment is evaluated in this thesis, based on the authors experiences with the tools.

In total, four variations of the Impulse-C solution has been implemented; a basic solution and a multicore solution, both implemented in a floating-point and a 'fixed-point' version. The implemented solutions are analyzed through various experiments described in this thesis, done during simulation using CoDeveloper. Attempts were made to get the solutions to run on the target platform, the Cray XD1 supercomputer Musculus, but these were unsuccessful. A wrong choice of properties and constraints in Xilinx ISE are believed to have caused the FPGA programming file to be generated faulty. There was no time to confirm and correct this. Some information about device utilization and performance could still be extracted from the Xilinx ISE 'Static timing' and 'Place and route' reports.

Preface

This report is written as a part of my Master's Thesis carried out at NTNU, Trondheim. The main thesis supervisor on this project has been Morten Hartmann at the department of computer and information science (IDI), NTNU. The report is written in English, with a list of utilized abbreviations included in appendix A.

Acknowledgments

I would like to thank my main thesis supervisor Morten Hartmann at IDI for help on writing this report, and Dag Kristian Rognlien at Sintef for all help on technological issues. I would also like to thank Finn Drabløs for help with medical questions.

Elisabeth Linvåg
June 6, 2008

Contents

Abstract	i
Preface	iii
List of Figures	ix
List of Tables	xi
List of algorithms	xiii
I Setting	1
1 Introduction	3
1.1 Introduction	3
1.2 Motivation	4
1.3 Objectives	4
1.4 Challenges	4
1.4.1 Writing the report in English	4
1.4.2 Acquiring knowledge and experience	4
1.4.3 Technical challenges	5
1.5 Thesis Outline	6
II Background	7
2 Background	9
2.1 Introduction	9
2.2 FPGA	9
2.2.1 Introduction	9
2.2.2 Architecture	9
2.3 Cray XD1 Supercomputer	10
2.3.1 Introduction	10
2.3.2 Architecture	10
2.4 PWM	12
2.4.1 Introduction	12
2.4.2 The PWM algorithm	12

2.5	Co-Design	14
2.5.1	Introduction	14
2.5.2	Co-Design languages	15
2.5.3	Co-Simulation	16
2.6	Impulse-C	16
2.6.1	Introduction	16
2.6.2	Programming model	16
2.6.3	CoDeveloper	16
3	Previous Work	19
3.1	Introduction	19
3.2	FPWM	19
3.2.1	Introduction	19
3.2.2	Architecture	20
3.2.3	Challenges	21
3.2.4	Status	21
3.3	Evaluating Co-Design	21
3.3.1	Introduction	21
3.3.2	Comparative analysis of high level programming	22
3.3.3	Exploring Impulse-C	22
III	Solution	25
4	Possible Solutions	27
4.1	Introduction	27
4.2	Applicable processing schemes	27
4.2.1	Introduction	27
4.2.2	Shared memory	28
4.2.3	Pipelined processing	28
4.2.4	Parallel processing	29
4.3	Basic system architecture	30
4.3.1	System partitioning	30
4.3.2	Implicit parallelism	31
4.3.3	Input/Output scheme	31
4.3.4	Application 'pipeline'	32
5	Implemented Solutions	37
5.1	Introduction	37
5.2	Basic application architecture	38
5.2.1	Introduction	38
5.2.2	Framework and modules	38
5.3	Implementation of the software framework - <i>FPWM*_sw.c</i>	39
5.3.1	Choosing filter	39
5.3.2	Reading and streaming input	39
5.3.3	Reading and writing output	40
5.4	Implementation of the general PWM-module	41

5.4.1	Reading input from stream	41
5.4.2	Computing scores	41
5.5	Implementation of the general filter-module	42
5.5.1	Applying summation filter	42
5.5.2	Applying threshold filter	43
5.6	Generating VHDL and hardware	43
5.7	Users manual	43
 IV Analysis		47
 6 Results Analysis		49
6.1	Introduction	49
6.2	Software simulation	49
6.2.1	Input data	49
6.2.2	Software partition	50
6.2.3	Hardware partition	53
6.3	Generated HDL code	54
6.3.1	<i>FPWM*_comp.vhd</i>	54
6.3.2	<i>FPWM*_top.vhd</i>	55
6.3.3	<i>rt_impulse_FPWM*.vhd</i>	56
6.3.4	VHDL library	56
6.4	Generated hardware logic	56
6.4.1	PWM module	57
6.4.2	Filter module	58
6.4.3	StageMaster	58
6.5	FPGA programming file	59
 V Synopsis		63
 7 Discussion		65
7.1	Introduction	65
7.2	Implemented solutions	65
7.2.1	Functionality and features	65
7.2.2	Scope	67
7.2.3	Generated HDL/HW	68
7.3	CoDeveloper ease-of-use	70
7.4	Productivity	71
7.5	Performance	72
7.6	Unanswered questions	73
 8 Conclusion and Future Work		75
8.1	Conclusion	75
8.1.1	Project value	76
8.2	Future work	76
8.2.1	Choosing I/O files	76

8.2.2	Securing written results	76
8.2.3	Multiple matrices and sequences	77
8.2.4	Explicit parallel processing	77
8.2.5	True fixed-point solution	77
8.2.6	Ignoring regions of DNA	78
8.2.7	Utilizing a database connection	78
8.2.8	Web interface	78
VI	Appendices	79
A	Nomenclature	81
B	Source Code	83
C	HDL Build Reports	125
D	Architecture Block Diagrams	139
	Bibliography	141

List of Figures

1.1	Pattern matching in bio-informatics at NTNU	3
2.1	Simplified FPGA architecture	10
2.2	Simplified Cray XD1 architecture [4]	11
2.3	Example matrices	13
2.4	Computing score for a given sub-sequence [9]	14
2.5	Design flow of general co-design approach	15
2.6	Impulse-C design flow [3]	17
3.1	Modules of the FPWM-system	20
3.2	Efficiency vs. ease-of-use [6]	22
3.3	Results from software simulation	23
4.1	Simple streaming on Cray XD1	27
4.2	Shared memory	28
4.3	Pipelined processing	29
4.4	Parallel processing	30
4.5	Application pipeline	30
4.6	Implicit parallel processing	31
4.7	Example input data	32
4.8	Example output data	32
4.9	Data streaming	33
5.1	Implemented architecture	38
5.2	Computing score for a given sub-sequence [9]	41
6.1	Pipelined loop in StageMaster vs source code	58
6.2	First faulty FPGA programming file on Musculus	59
6.3	Second faulty FPGA programming file on Musculus	61

List of Tables

6.1	Input alignment matrices	50
6.2	Input DNA sequences	50
6.3	Converted matrices	51
6.4	Tests done on basic solution	51
6.5	Filtered results - basic solution	52
6.6	Tests done on multicore application	52
6.7	Filtered results - multicore application	53
6.8	Design statistics	57
6.9	Device utilization	57
7.1	Device utilization - slices	70
7.2	Timing statistics	72

List of Algorithms

1	Converting a value of the alignment matrix	13
2	PWM	14
3	Finding correct PWM-scores	42

Part I
Setting

Chapter 1

Introduction

1.1 Introduction

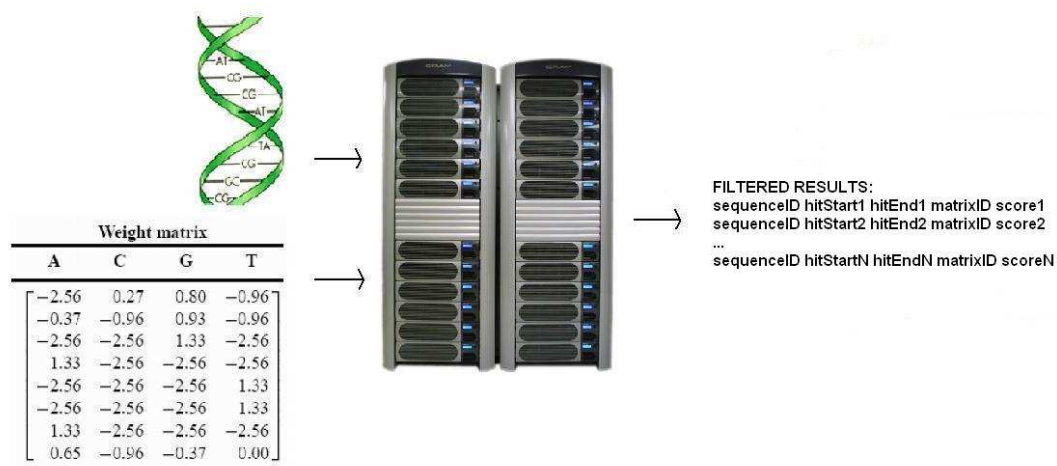


Figure 1.1: Pattern matching in bio-informatics at NTNU

Bio-informatics involve the use of various techniques such as applied mathematics, informatics and computer science to solve biological problems. The Cray XD1 super-computer at NTNU, Musculus, is a computational resource mainly targeted for research within the field of bio-informatics.

Musculus was purchased the summer of 2005, as a part of a collaboration between the department of computer and information science (IDI) and the faculty of medicine (DMF). In this collaboration, IDI will develop bio-informatics modules as part of own research, usable for DMF to solve bio-informatics problems.

1.2 Motivation

FPGA technology is well suited for acceleration of compute-intensive applications that require little or no communication. An implementation of the PWM algorithm is a good example of such an application; a small and compute-intensive part that requires a minimal amount of communication. But the use of hardware descriptive languages to program FPGAs, such as VHDL, is a complicated and time consuming process that requires intimate knowledge of how hardware works. Consequently, it is desirable to make use of co-design languages to facilitate the use of hardware for a software developer, as well as to integrate the environment for development of soft- and hardware.

1.3 Objectives

Pattern matching in bio-informatics is a discipline in sturdy growth, and has a great need for searching through large amounts of data. At NTNU, an FPGA prototype specified in VHDL has been developed, which identifies short motifs or patterns in genetic data using Position-Weight Matrices (PWM). The prototype is designed for running on a Cray XD1 supercomputer.

The project consists of the following tasks:

- Specification and implementation of a Impulse-C based alternative to the existing VHDL-based solution.
- Evaluation of easy-of-use of the CoDeveloper environment, and productivity vs. final performance when comparing the Impulse-C solution and the existing VHDL-based solution.

1.4 Challenges

1.4.1 Writing the report in English

Writing this report in English has to a degree been a challenge, not having English as the mother tongue. Fairly well developed English skills were put to the test when faced with the first experience in writing a more comprehensive report, both explaining and discussing a great deal of technical terms and concepts, by oneself.

1.4.2 Acquiring knowledge and experience

Implementing while learning Implementing the co-design based alternative to the existing VHDL based solution did not go as fast as initially expected. One of the major reasons for this was the great need to acquire further knowledge about developing systems using Impulse-C, while at the same time being in the middle of an actual development process with a steadily approaching thesis deadline. Problems connected with allocating memory in the software framework, such as memory leakage, contributed to slowing down the implementation process also. Limited experience with memory allocation in C resulted in a need to learn more.

Implementing for hardware platform One of the great benefits of developing systems using a co-design approach is not having to possess as much knowledge about the underlying hardware of the target platform as when developing the same system directly in VHDL. Trying to implement the hardware portion of the co-design solution taking on the role of a pure software developer, ignoring a substantial share of possessed hardware knowledge, was somewhat difficult.

Decreased level of control The increase in abstraction level when making the transition from implementing in VHDL to utilizing co-design, having to rely on special compilers to generate the actual VHDL code, decreases the amount of control the developer has over the hardware that is generated. Having to give up a significant amount of the control that one usually have when implementing hardware modules takes some time to get used to.

Memory allocation and leakage As previously mentioned, problems connected with memory allocation in the software framework arose while developing the co-design solution. These problems were seemingly caused by so-called memory leakage. This bug did not manifest itself until the framework read results sent from the filter-module in hardware and stored these in a designated variable. The bug prevented the software simulation of the co-design solution from terminating in the correct manner. Fixing this bug, and even finding the actual cause, took a substantial amount of time. The software was debugged using both Valgrind and the CoDeveloper Application Monitor. Commenting out code snippets that could potentially cause the software simulation to crash, as well as adding snippets that checked for potential errors closing streams and files, were also used as methods for debugging the application.

1.4.3 Technical challenges

Debugging software Debugging the software framework was not intuitive using the Impulse-C development environment, nor compatible with the OS of the development workstation, and it was consequently found necessary to debug using an alternative debug tool. Help was needed for this, not being knowledgeable about software debuggers.

Generating ISE project After letting the development environment generate the required VHDL code, based on the Impulse-C description of the desired hardware functionality, it was made ready to export to Musculus. However, the affiliated template project file also generated did not work as it was supposed to, due to (so far) unknown causes, and the project was consequently not able to be opened in Xilinx ISE. A new ISE-project had to be made manually, based on the generated VHDL files. The default properties in Xilinx ISE did not work well and sabotaged the generation of a fully functional FPGA programming file. A topic about the failed translation process was posted on the Impulse-C support forum, resulting in some problems being fixed. Taking a look at the properties of other functional projects also helped in getting closer to generating a functional programming file.

1.5 Thesis Outline

Chapter 2, 'Background', presents background information on FPGA technology, the Cray XD1 platform, the PWM-algorithm, the concept of co-design, and Impulse-C.

Chapter 3, 'Previous work', presents the previously implemented FPGA-solutions for the PWM-algorithm on the Cray XD1 supercomputer, Musculus. The chapter also present information on work done to evaluate the use of Impulse-C, as well as co-design in general, on Cray XD1.

Chapter 4, 'Possible Solutions', presents and evaluates various design schemes that might be suitable for implementing a PWM FPGA-solution using an Impulse-C co-design approach.

Chapter 5, 'Implemented Solutions', describes the implemented Impulse-C approach to the FPWM-system.

Chapter 6, 'Results Analysis', presents an evaluation of ease-of-use and final performance of the newly developed solutions compared to the previously implemented VHDL-based solution.

Chapter 7, 'Discussion', presents a discussion on the implemented Impulse-C solutions.

Chapter 8, 'Conclusion and Future Work', presents a conclusion to the thesis, as well as suggestions for future work.

In appendix A one can find a list of notation and abbreviations used in the report.

In appendix B one can find the source code for the implemented Impulse-C solutions.

In appendix C one can find reports from the HDL build process (CoBuiler).

In appendix D one can find block diagrams of the implemented architectures as generated by CoDeveloper Application Monitor.

Part II

Background

Chapter 2

Background

2.1 Introduction

This chapter describes the relevant technologies and theory. FPGA technology is presented in 2.2, the Cray XD1 platform in 2.3 and the PWM-algorithm in 2.4. Co-design will then be presented in 2.5, and finally Impulse-C in 2.6.

The background material in this thesis is based on material from the preceding 5'th year project [10].

2.2 FPGA

2.2.1 Introduction

An Field-Programmable Gate Array (FPGA) is a specially made semiconductor used to process digital information, similar to a microprocessor [7]. The FPGA consists of digital gate array technology that can be custom programmed and reprogrammed by the end user after manufacture to define functionality. This can be done both dynamically and statically by changing the electrical connections, turning on switches which make connections between circuit nodes and the metal routing tracks.

Application acceleration

FPGAs in a supercomputer accelerate applications by acting as high speed, specialized co-processors. The FPGAs shift computational work from the main processors, by running subroutines that take over portions of the application.

2.2.2 Architecture

When using FPGAs, it is important that an application circuit is mapped into an FPGA with adequate resources. Although some FPGAs also have more complex units, the average FPGA consists of an array of configurable logic blocks (CLBs) connected by several routing channels. This array is surrounded by a outer boundary of I/O cells that handle the communication in and out of the FPGA, as seen in figure 2.1.

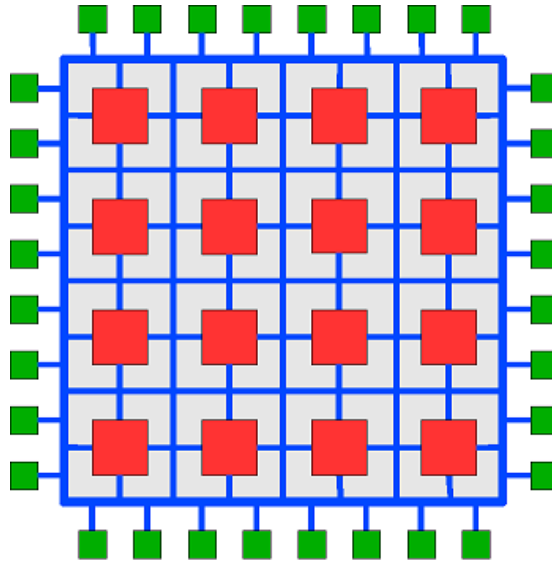


Figure 2.1: Simplified FPGA architecture

A CLB is the array of multi-input and multi-output logic cells that needs to be programmed, and consists of several look-up tables (LUTs), flip flops, multiplexers (MUXs), arithmetic logic (ALU) and dedicated internal routing.

2.3 Cray XD1 Supercomputer

2.3.1 Introduction

The Cray XD1 supercomputer is a computational resource purpose-built for demanding HPC applications. It lets users simulate, analyze and solve complex problems quickly and accurately without having to increase the size of the computer or its power budget [3]. This is done by leveraging reconfigurable computing techniques in the form of special subroutines. Applications that can benefit from running on Cray XD1 include financial computing, bio-informatics and other compute-intensive activities.

2.3.2 Architecture

The Cray XD1 system is based on the Direct Connected Processor (DCP) architecture [4]. The CPU system is built as a standard Symmetrical Multi-Processing (SMP) design, extended with FPGAs for application acceleration. A chassis houses six compute blades, where each compute blades has two AMD Opteron microprocessors and a RapidArray Processor to handle communication. The FPGA expansion module attached to the compute blade contains a Xilinx Virtex-II Pro FPGA that acts as an Application Acceleration Processor (AAP), four QDR-II SRAM, and an additional RapidArray processor. A simplified representation of a Cray XD1 compute blade is shown in figure 2.2.

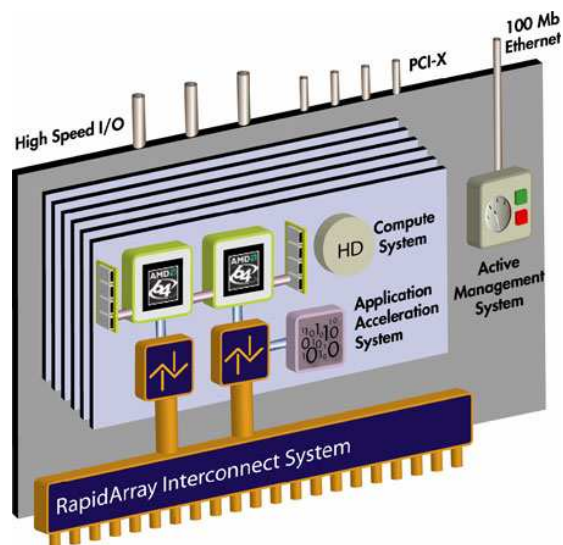


Figure 2.2: Simplified Cray XD1 architecture [4]

The six compute blades, together with their FPGA expansion module, can be viewed as nodes. Each node is an independent system with its own GNU/Linux operating system, and is able to communicate with other nodes via the RapidArray Interconnect System. The six nodes in Musculus at NTNU are configured into one log-in node and five computational nodes [11].

Compute environment

AMD Opteron Microprocessor The microprocessor used in the Cray XD1 supercomputer is a 64-bit single core AMD Opteron 2.4GHz processor with 2GB RAM [2]. The two Opteron microprocessors on each compute blade are connected via AMD's HyperTransport, forming a 2-way SMP [4].

AMD HyperTransport HyperTransport technology is designed to help reduce the number of buses in a system, which in turn can reduce system bottlenecks and enable faster microprocessors to use system memory more efficiently [1]. This will optimize message-passing applications. AMD's HyperTransport appear transparent to the operating system and offer little impact on peripheral drivers. Mellanox on-motherboard InfiniBand switches create a fabric for HyperTransport between compute nodes.

Application acceleration

Xilinx Virtex FPGA On each node there is a Virtex-II Pro FPGA with 3.2GB/s interconnect [14]. Through the use of the interface Cray XD1 QDR II SRAM Core the users can access the QDR SRAM memory from a FPGA-design.

The Xilinx Virtex-II Pro has two built-in PowerPC 405 RISC-processors. It has also several 18Kb Block SelectRAM+ (BRAM) memory modules, in total 232, spread across

the chip. It is preferable to use the BRAMs to store larger amounts of data instead of using the CLBs [9].

RapidArray Interconnect

The RapidArray interconnect is a 96-GB-per-second non-blocking, embedded crossbar-switch fabric that connects the RapidArray processors (RAPs) [4]. The center of the RapidArray Interconnect System is the RapidArray Switch, which has 24 internal links used to connect to the 12 RAPs within the chassis. The nodes within the chassis is therefore connected to the switch using two links each, one for each direction of communication. The RapidArray Switch offers the nodes a bandwidth of 4GB/s for communication; 2GB/s on each link. Communication is, as mentioned earlier, controlled and coordinated by the RAPs on each node. RapidArray Interconnect is also used to connect the SMP with the FPGA. What interface that is utilized depends on which component it is that initiates the communication between the two. The Fabric Request Interface is used in transactions that is initiated by the SMP, while the User Request Interface is used in transactions that is initiated by the FPGA.

2.4 PWM

2.4.1 Introduction

Position Weight Matrices are often credited to Roger Staden, after he introduced the concept in his article 'Computer methods to locate signals in nucleic acid sequences' [9] in 1983.

Position-specific Weight Matrices (PWMs) are the main components of many algorithms used in bio-informatics to identify short patterns or motifs in nucleic acid, such as a DNA sequence. PWMs, also known as log-odds matrices, are simplified representations of known binding sites [13]. The height of such a weight matrix is equivalent to the length of the alphabet in the sequence to search through, while the width is equivalent to the length of the pattern to search for.

DNA

A DNA sequence consists of a double string of symbols that represent the 4 different nucleotide building blocks; A (adenine), G (guanine), C (cytosine) and T (thymine). These strings are usually referred to as the positive and negative strand of the DNA sequence; the second string being the reverse complement of the actual DNA sequence, which is the first string. This second string is a DNA-binding protein, called a transcription factor, that recognize and binds to the main DNA sequence. A DNA sequence will contain some well-defined regions, like genes and the regulatory regions for genes.

2.4.2 The PWM algorithm

An input pattern is usually represented as a count matrix, also known as an alignment matrix. This count matrix is generated by comparing known binding sites and counting symbol frequencies at each position. By normalizing the count matrix it is transformed

into a frequency matrix, which in turn is transformed into a PWM. A pseudo-count is usually added to each position of the matrix when it is converted to a PWM. The formula for converting the individual values of an alignment matrix to produce a PWM is given in algorithm 1.

Pos	Alignment matrix				Frequency matrix				Weight matrix			
	A	C	G	T	A	C	G	T	A	C	G	T
1	0	4	7	1	0.00	0.33	0.58	0.08	-2.56	0.27	0.80	-0.96
2	2	1	8	1	0.17	0.08	0.67	0.08	-0.37	-0.96	0.93	-0.96
3	0	0	12	0	0.00	0.00	1.00	0.00	-2.56	-2.56	1.33	-2.56
4	12	0	0	0	1.00	0.00	0.00	0.00	1.33	-2.56	-2.56	-2.56
5	0	0	0	12	0.00	0.00	0.00	1.00	-2.56	-2.56	-2.56	1.33
6	0	0	0	12	0.00	0.00	0.00	1.00	-2.56	-2.56	-2.56	1.33
7	12	0	0	0	1.00	0.00	0.00	0.00	1.33	-2.56	-2.56	-2.56
8	6	1	2	3	0.50	0.08	0.17	0.25	0.65	-0.96	-0.37	0.00

Figure 2.3: Example matrices

Algorithm 1 Converting a value of the alignment matrix

$c, N : int$; {current count, total count}
 $p : double$;
 $s : double \leftarrow 0.25$; {pseudo-count}
 $P : double \leftarrow 0.25$; {background probability}
 $p = \frac{(c+s)}{(N+4s)}$;
 $\ln(\frac{p}{P})$;

The score of the PWM for a given sub-sequence is computed as the sum of PWM scores, and is illustrated in figure 5.2.

The PWM is moved along the input DNA sequence, and a value is calculated for each position in the sequence from the weights of each symbol in the PWM. The total correspond to the value of the first visible position in the sequence. Calculated totals are consecutively sent to one or more filter processes. Pseudo-code for the PWM algorithm is shown in algorithm 2.

The algorithm allows for both pipelining and parallelizing. Computational loops in hardware can be pipelined, as long as they do not contain nested loops. Parallel instances of the algorithm can also be implemented to search through the sequence using multiple matrices at the same time.

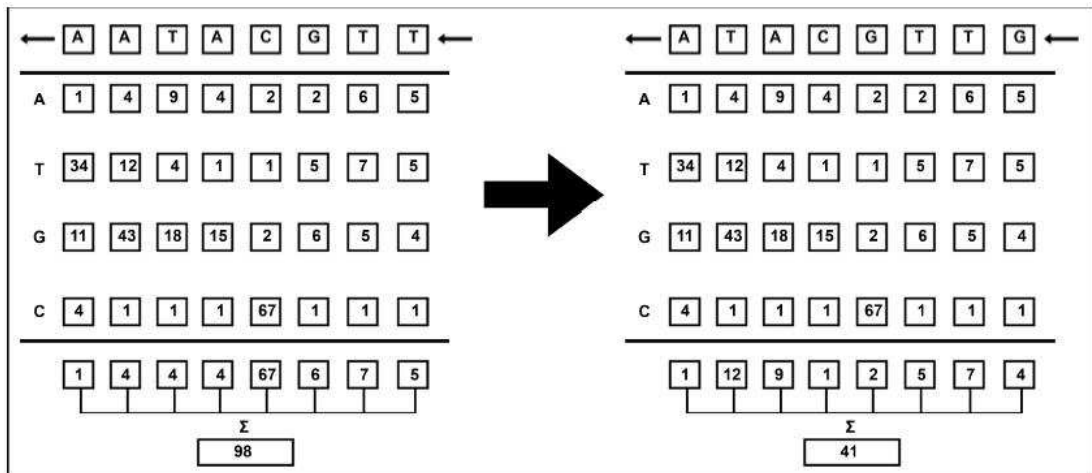


Figure 2.4: Computing score for a given sub-sequence [9]

Algorithm 2 PWM

```

M : int; {length of DNA-sequence}
N : int; {length of sub-sequence}
S : string; {DNA-sequence}
pwm : table[4][N]; {weight matrix}
counter : int ← 0; {current position in DNA-sequence}
i : int; {current position in sub-sequence}
sum : float;
while counter ≤ M - N do
  for i = 0 to N do
    sum ← sum + pwm-value S[counter + i];
  end for
  counter ++
  save sum;
end while

```

Markov process background model A higher-order background model, modeled as a Markov process, is sometimes used in order to improve pattern matching. This model estimates the probability that a given sub-sequence is found in the non-pattern background, which is then used to normalize the final probability. Separate filter modules can adjust the probability according to higher-order background models.

2.5 Co-Design

2.5.1 Introduction

Reconfigurable supercomputing uses FPGAs to improve the performance of microprocessors, but at a high cost as one has to hand-code custom design in a Hardware De-

scription Language (HDL). In areas such as embedded design the collaboration between software and hardware is very intimate. It should therefore be possible to create variants of a product by changing parts of the implementation from software into hardware, or the other way around, without having to redesign unchanged parts of the system. One applicable method to achieve this is using co-design. Co-design is often defined as "The simultaneous design of the software and hardware composing a system". In other words, most approaches to co-design focus on creating systems in which both synthesis and simulation can be done of both the software and the hardware simultaneously.

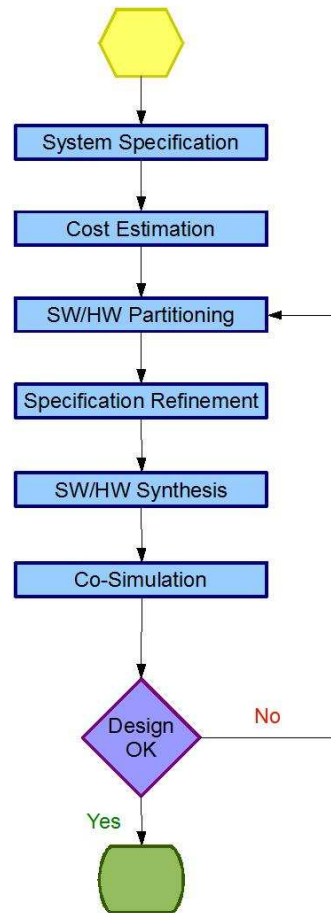


Figure 2.5: Design flow of general co-design approach

2.5.2 Co-Design languages

Hand-coding custom design in an HDL is a very time consuming and error prone task; especially since HDLs are not designed to describe algorithms. Co-design languages provide compilers that translate fixed- and floating-point algorithms implemented in a HLL directly into circuit design in VHDL. Adding hardware platforms is just a question of defining new interface description files and producing the code that ties the design to the description interface.

Implementing applications in a co-design language, the programmer usually partitions the program into software and hardware sections manually, and writes C code to synchronize the data communication between the two parts. An illustration of the design flow in co-design is shown in figure 2.5.

2.5.3 Co-Simulation

Co-simulation is, just like co-design, used to simultaneously validate and experiment with the hardware and the software components of an embedded system. This can be used to collect information about the system before actually building a prototype. A traditional approach is to model the hardware processor and then run the software on it. For tractability reasons, the hardware is usually modeled using bus-functional models (Rowson). Either way, the goal is to verify whether the hardware and software can work together.

2.6 Impulse-C

2.6.1 Introduction

Impulse-C [12], created by Impulse Accelerated Technologies Inc., is a co-design language designed for data-flow oriented applications, but with the flexibility to support other programming models as well. What programming model that is selected depends on the requirements of the application to be implemented, but also on the architectural constraints of the selected programmable platform target.

2.6.2 Programming model

Impulse-C extends standard C to support a modified form of the Communicating Sequential Processes (CSP) programming model. CSP simplifies the expression of highly parallel algorithms through the use of well-defined data communication, message passing and synchronization mechanisms.

At the center of the Impulse-C programming model are processes and streams. The data that is processed by the application will flow from process to process by means of streams, or alternatively by means of messages and/or shared memories.

Processes Processes are independently synchronized, simultaneously operating segments of an application. Processes are written using standard C and perform the work of the application by accepting data, performing computations and generating outputs.

Streams Streams represent one-way channels of communication between simultaneous processes, and are self-synchronizing with respect to the processes by the benefit of buffering. Characteristics are specified at the time a stream is created in the application.

2.6.3 CoDeveloper

CoDeveloper contains tools that allow FPGA systems to be developed and debugged using Impulse-C. The software-to-hardware compiler and optimizer translates C-language

processes to (RTL) VHDL code, while optimizing the generated logic and identifying opportunities for parallelism.

Bottlenecks in the data flow and other areas of acceleration can be identified by an application monitor, allowing designers to iteratively analyze and experiment with alternative strategies for hardware pipelining.

The flow of application development in CoDeveloper (and Impulse-C) is shown in figure 2.6.

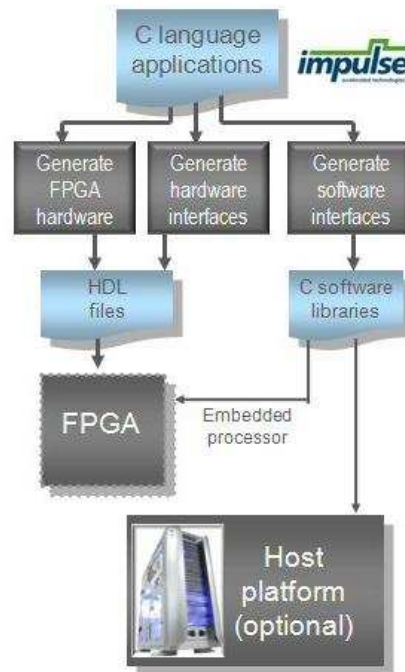


Figure 2.6: Impulse-C design flow [3]

An application is first developed in Impulse-C, debugged in CoDeveloper using the application monitor, then translated to HDL by the CoBuilder tools integrated in CoDeveloper and exported. The rest of the process takes place in a design environment more appropriate to the target platform, such as Xilinx ISE for Xilinx FPGAs, where the design is synthesized and implemented, and the FPGA programming file is generated. This environment also allows for further simulation of the design. Finally, the software is exported to the target platform together with the generated FPGA programming file.

Impulse-C API

Impulse-C's API includes C-compatible functions that let designers create system-level parallelism using the Impulse-C programming model. Impulse C also includes platform support packages that simplify C-to-hardware compilation for specific software-hardware targets, including Cray XD1, with the help of an infrastructure for on-chip bus communication. In addition, the API offer support for single and dual precision

floating point, as well as pipelined floating point operators, for Xilinx FPGAs.

CoBuilder

CoBuilder is a set of hardware generation tools that produces a set of files ready for use with the target platform development software and hardware.

Only those source files that have been specified as hardware get analyzed by the hardware compiler, while all other files are overlooked. Impulse-C processes that are found within the specified hardware source files, but that are not specified as hardware processes, will also be overlooked.

- **RTL Generator** : Reads the application source files, compiles those processes identified as hardware and creates equivalent hardware descriptions in the form of (RTL) HDL files.
- **Architecture Generator** : Reads the application source files and generates hardware wrappers implementing the required stream, signal and memory interfaces allowing hardware processes to communicate with other parts of the application.
- **Library Generator** : Creates software library elements required for compilation of Impulse-C software processes on the target embedded processor.

HDL library The hardware generation tools make references to additional HDL components that are provided in a special library called "impulse". For the purpose of simulation and/or synthesis, this library must be combined with the generated component- and system-level HDL files.

In the form of C-libraries and HDL wrapper components, CoBuilder is capable of generating the required hardware/software interfaces. This simplifies the process of moving a complete hardware/software application to selected programmable platforms.

Chapter 3

Previous Work

3.1 Introduction

This chapter describe previous work relevant to this thesis. The previously implemented FPGA-solution for the PWM-algorithm in VHDL, on the Cray XD1 supercomputer Musculus, is presented in section 3.2, while previous work on comparing co-design languages to use with Cray XD1 is presented in 3.3.

The previous work presented in this thesis is based on material from the preceding 5'th year project [10].

3.2 FPWM

3.2.1 Introduction

FPWM (FPGA PWM) is a prototype on a custom-made FPGA-solution for identifying short motifs or patterns in genetic data using a position-weight matrix (PWM). Lars Krutådal at IDI had in his master's thesis done an early implementation of the prototype, and was later hired by the department to work on completing the system. Krutådal pictured two possible extensions to his implementation in order to achieve parallel execution; a multicore solution and an multiple node solution - the optimal solution being a combination of the two.

Per Andreas Gulbrandsen developed the FPWM prototype further in his 2007 master thesis, implementing a more functional FPGA-solution in VHDL code [9]. Gulbrandsen was hired by IDI the fall semester of 2007 to continue Krutådals work on completing the FPWM system.

Related work

"Accelerating Motif Discovery: Motif Matching on Parallel Hardware" [8] was published in 2006 in the book "Algorithms in Bioinformatics", as a part of the book series "Lecture Notes in Computer Science". The authors of the article include among others several people at IDI, NTNU. In this article, the authors propose and define an abstract module PAMM (Parallel Acceleration of Motif Matching) with motif matching on parallel hardware in mind. There is provided a concrete implementation of the authors

approach called MAMA, based on the MEME algorithm.

3.2.2 Architecture

The FPWM system consists of an overlaying framework for control, and four underlying modules. These modules include a reader-, pwm-, filter- and writer-module, as seen in figure 3.1.

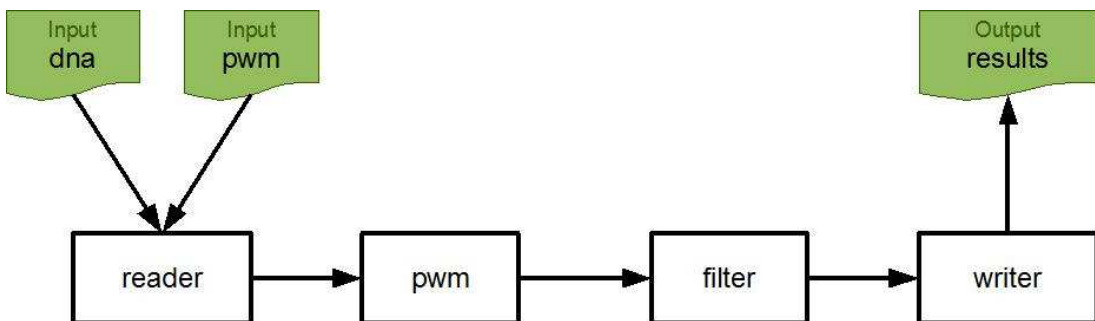


Figure 3.1: Modules of the FPWM-system

Overlaying framework

The overlaying framework initialize the modules, both at start-up and reset. It also manages communication and data from the SMP-node. The SMP-node writes results to a file.

reader-module The reader-module reads the DNA-sequence and PWM from an input-file, and passes a new sub-string to the pwm-module every cycle.

pwm-module The pwm-module perform the necessary computation, by applying the pwm-algorithm to the input sub-strings. The result is passed to the filter-module, together with a result index.

filter-module The filter-module processes the results from the pwm-module, both in order to reduce the amount of results, by applying a threshold filter, and to generate more useful results, by applying the summation filter. The filtered results are then passed to the writer-module.

writer-module The writer-module perform all writing to the SMP-node. Two buffers are initiated, with eight elements each, so that the module don't have to constantly stall. Instead, the module can read data at the same time as it writes to the SMP-node. Thanks to the filter-module, the frequency of write-calls to the writer-module is limited.

3.2.3 Challenges

As previously mentioned in chapter 1, developing systems in VHDL is a time consuming and error prone task. This was also a significant challenge while developing the FPWM system. VHDL is not well suited for expressing algorithms in general, and does not support the use of floating-point operands. The latter requires the designer to always convert scores between floating-point and fixed-point when sending scores to, or receiving scores from, hardware.

In the previous implementation of the FPWM-system, some results were duplicated, while others were lost when the execution finished. This could have had some effect, small or large, on the implementation of modules filtering the results.

Filter functionality offered an improvement to the FPWM system as it limited the burden on the bandwidth between the CPU and FPGA, one of the most critical parts of the system [9]. Filtering the results of the computation, the queue of data to write back to the CPU is greatly reduced, giving the system a performance boost.

3.2.4 Status

The VHDL solution still does not work completely accurately after being worked on for the duration of more than one 5'th year project and two master's thesis's, which is mostly due to the challenging task of implementing at HDL level. The biggest bottleneck in the system today is transferring data between the CPU and the FPGA, as well as to the embedded memory. There is also still a problem with validating the output (results) of the FPWM system. Results may be false as the different modules of the system could translate the fixed point values, represented in two's complement form, incorrectly. This is mostly an issue when working with negative values.

Some of the possible expansions to consider for the FPWM system include:

- Implementing a function that converts DNA for later use.
- Reading data from, and writing data, to a database instead of a file.
- Implementing a multiple node, or multicore, solution utilizing MPI.
- Implementing compensation for missing/skipped regions of the DNA sequence.
- Implementing more complex configurations.

3.3 Evaluating Co-Design

3.3.1 Introduction

Sometimes the explicit and precise descriptions offered by VHDL are necessary to realize designs goals, but not always. Hardware design can benefit from development tools that makes it possible to mix high-level and low-level descriptions as needed to meet the design goals as fast as feasible. A co-design program is typically just a few hundred lines of code, implemented in a few days - compared to months or years in the case of RTL VHDL. The skill pre-requisites to writing good co-design programs are not

extensive. Prior experience with C an advantage when using a C-based language, as are skills in parallel programming methods such as MPI or OpenMP.

3.3.2 Comparative analysis of high level programming

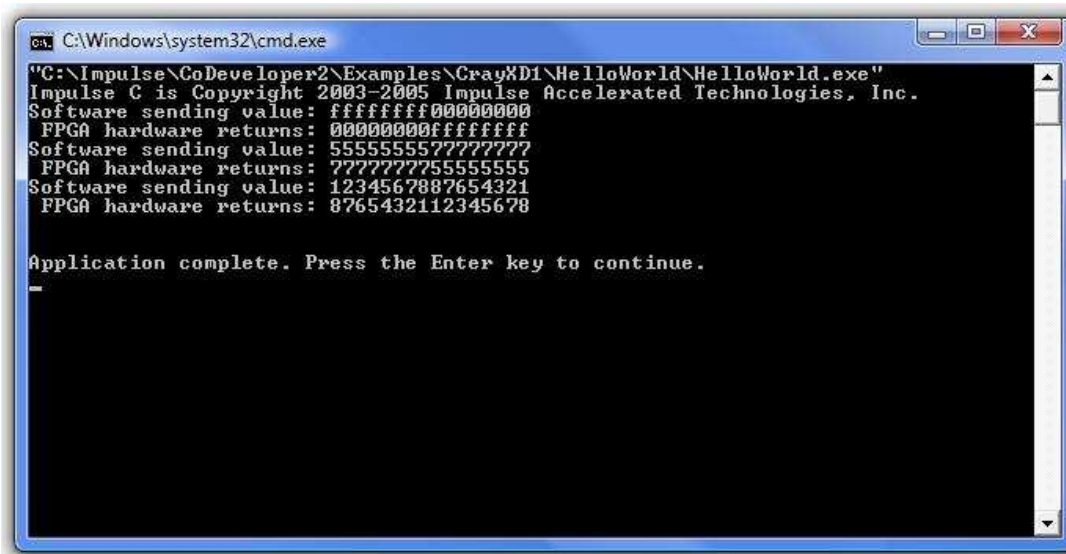
Co-design languages normally trade performance with ease-of-use [6], and it is therefore of interest to know in a general sense how much performance and utilization is given up and how much ease-of-use is gained. "Comparative analysis of high level programming for reconfigurable computers: Methodology and empirical study" [6] is an article written by a team of computer scientists at the George Washington University, Arctic Region Supercomputing Center. The article describes a comparative analysis of three co-design languages, each representing different high-level programming paradigms. In the experiment, four workloads were selected for implementation on Cray XD1, using each of the three selected co-design languages. The experiment also involved three independent users with different degrees of experience in the field. The resulting evaluation of the results are illustrated in figure 3.2.



Figure 3.2: Efficiency vs. ease-of-use [6]

3.3.3 Exploring Impulse-C

The 5'th year project report "High level description for FPGA hardware acceleration of DNA motif identification" [10], written at IDI NTNU, takes a special look at Impulse-C as an applicable tool methodology to re-implement the FPWM system. Observations made on the usability of Impulse-C and CoDeveloper, by doing simple testing, are both described and discussed. Completing the tutorials available with CoDeveloper was found to be an important part of learning about the CoDeveloper tools, as well as gaining experience in using them. These tutorials give step-by-step instructions on how to run example applications on various target platforms, with all the necessary illustrations and explanations needed to understand the process. In addition to example applications, CoDeveloper also offer several application templates to help software designers get started on developing their own applications.



```
ca: C:\Windows\system32\cmd.exe
"C:\Impulse\CoDeveloper2\Examples\CrayXD1\HelloWorld\HelloWorld.exe"
Impulse C is Copyright 2003-2005 Impulse Accelerated Technologies, Inc.
Software sending value: ffffffff00000000
FPGA hardware returns: 00000000ffffffff
Software sending value: 5555555577777777
FPGA hardware returns: 7777777755555555
Software sending value: 1234567887654321
FPGA hardware returns: 8765432112345678

Application complete. Press the Enter key to continue.
-
```

Figure 3.3: Results from software simulation

Update

During the duration of the project, attempts at running the example application selected for testing on Musculus, the Cray XD1 cluster at NTNU, were unsuccessful. Unfortunately, there were no time to solve this problem before the project deadline. After completion however, further exploration led to several theories as to why the example application would not run on Musculus. For some time, it was believed to possibly be a compatibility issue, either between Musculus and CoDeveloper, or between Musculus and the Xilinx ISE version used to generate the FPGA programming file. The author didn't find any proof confirming this theory.

After opening up to the possibility that it was only this specific application that did not work properly, and that the other Impulse-C example applications designed for Cray XD1 could still run correctly 'out of the box', a different application was selected for testing. The new test application ran successfully on the first try, confirming the new theory that there are no compatibility issues. The software simulation of the first test application ran successfully as well, as shown in figure 3.3, suggesting that the application did not run correctly due to an undetected error during VHDL generation.

Part III
Solution

Chapter 4

Possible Solutions

4.1 Introduction

This chapter presents and evaluates the various design schemes possibly suitable for implementing a PWM FPGA-solution using the Impulse-C co-design approach. Applicable processing schemes are first presented in section 4.2, after which the planned system architecture is presented in 4.3.

4.2 Applicable processing schemes

4.2.1 Introduction

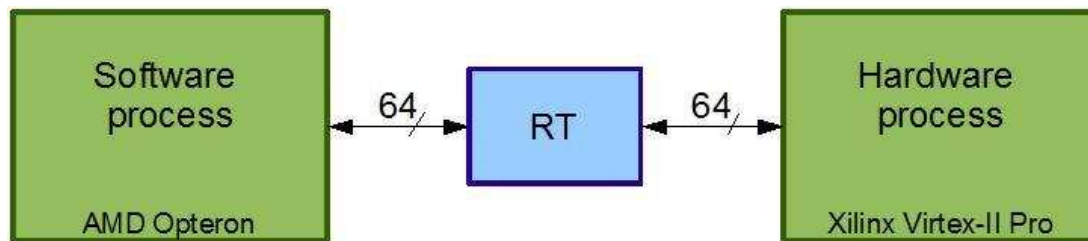


Figure 4.1: Simple streaming on Cray XD1

As illustrated by figure 4.1, applications generally include software processes running on the Opteron processor and hardware processes running in the FPGA that communicate via streams over the Cray XD1 RapidArray Transport (RT) fabric interface. A reimplementing of the FPWM-system in Impulse-C should have various potential advantages over the VHDL-solution. Some significant ones being that it is easier to express algorithms in Impulse-C, due to the higher abstraction level, and that Impulse-C more easily support the use of floating-point values in hardware. In addition to this, Impulse-C streams send sequential data between processes by implicitly incorporating the use of a FIFO-queue. Implementing these things using VHDL requires considerably more time and competence.

This section presents applicable processing schemes for implementing the PWM-system. Although they are presented separately, the ideal solution will incorporate a combination of all these schemes.

4.2.2 Shared memory

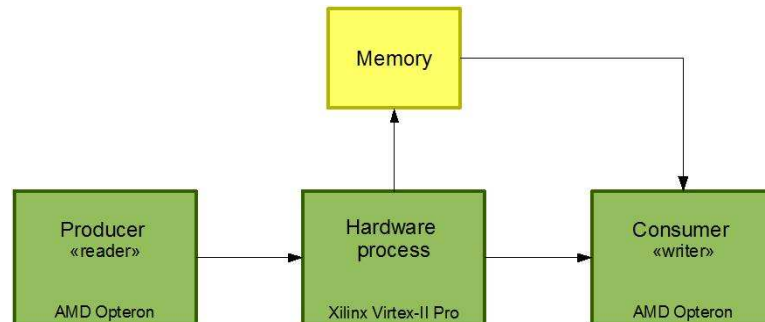


Figure 4.2: Shared memory

This scheme deals with incorporating the use of shared memory in simple data streaming on Cray XD1. The application includes two software processes, *Producer* and *Consumer*, running on the Opteron processor, and a number of hardware processes running on the FPGA.

The producer can request data to be written to the shared memory, while a hardware process will read and process the data. When computation is done, the consumer can then either try to read result data stored in the memory embedded on the FPGA, or receive other data directly from the hardware process. The hardware process will have to signal the consumer when data is ready to be read.

For communication between two hardware processes, however, embedded memory is only needed if the processes require random or irregular access to the data. In the case of sequential data, like in the FPWM system, a stream would be the most efficient method of data transfer. Deciding whether or not to use shared memory as described above, for communication between a software process and a hardware process, is a more complex task and strongly dependent on the application and the memory available. Whether or not it is necessary, or even favorable, to use shared memory in the FPWM system on Musculus will become more clear during implementation of, and experimentation with, an actual Impulse-C solution.

4.2.3 Pipelined processing

This scheme deals with incorporating the use of pipelined processes. The application includes two software processes, *Producer* and *Consumer*, running on the Opteron processor, and a number of pipelined hardware processes running on the FPGA.

The producer will pass data to the first hardware process, which will process the data and pass the result to the next hardware process. The data keeps getting passed from one hardware process to another in this matter until it reaches the last hardware

process, at which time all processing of the data should be finished and the results are sent to the consumer. The hardware processes in the pipeline can be either distinctly different, various instances of the same initial process, or both.

Looking at the general structure of an Impulse-C application, as well as that of the FPWM system itself, there will be a degree of pipelining implicitly applied regardless of the number of hardware processes. In other words, only the degree of pipelining is optional, not whether or not to utilize pipelining in the first place. The application modules will implicitly act jointly as a pipeline, but smaller snippets of code can also be pipelined by the hardware compilers.

Pipelining of instructions is not automatic, but requires an explicit declaration using the Impulse-C pragma `CO PIPELINE`. This declaration must be included within the body of a loop and prior to any statements that are to be pipelined.

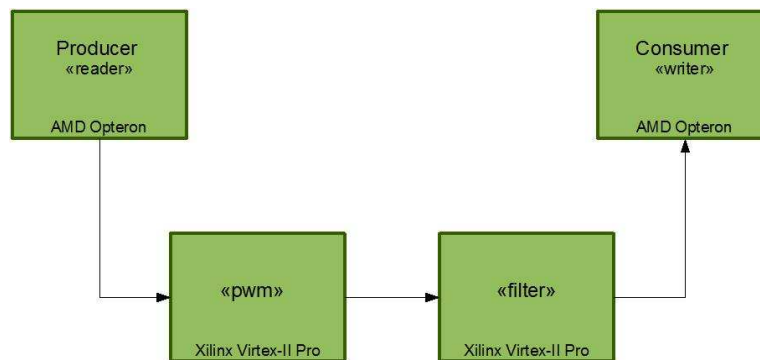


Figure 4.3: Pipelined processing

4.2.4 Parallel processing

This scheme deals with incorporating the use of parallel processes to handle computation. The application includes two software processes, *Producer* and *Consumer*, running on the Opteron processor, and a number of identical parallel hardware processes running on the FPGA.

The producer will divide the workload evenly between the hardware processes, which in turn process the data in parallel. When the parallel processes have finished processing their part of the data, the results are sent back to the software consumer.

In the case of the FPWM system, explicitly computing results in parallel should be easier done with multiple input matrices and/or sequences, as entire matrices and sequences can be distributed to each parallel process. With only one sequence and only one matrix, the sequence would have to be broken up into multiple sub-sequences to distribute to each parallel process. Breaking up the sequence would, based on the PWM algorithm, reduce the number of results. The results not being computed could be results that would have been filtered out later anyway, but could also be some of the more relevant results. Because of this, it would be no point in explicitly implement parallel instances of the hardware modules if there is only one matrix and sequence to process.

More on the use of implicit parallel processing is explained later, in the next section of this chapter. Explicit parallel processing will only be implemented if it is found to be time for it. First and foremost, a basic solution (successfully) processing a single matrix-sequence pair needs to be implemented.

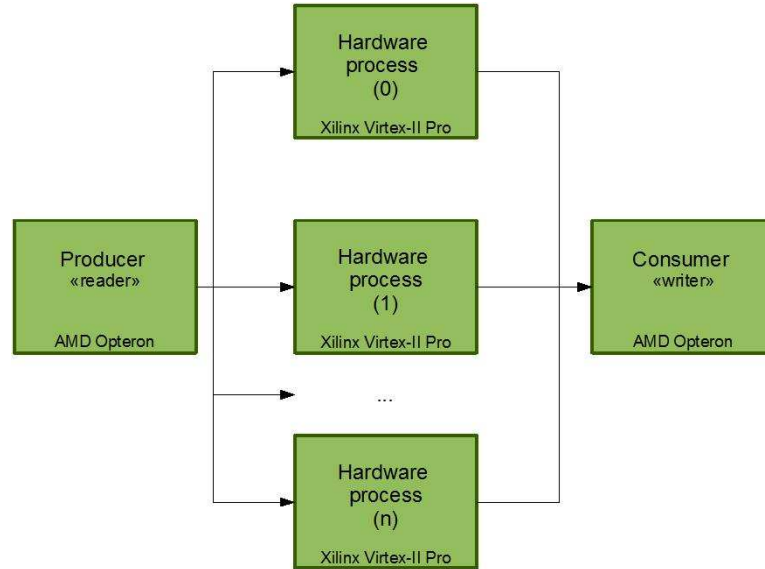


Figure 4.4: Parallel processing

4.3 Basic system architecture

4.3.1 System partitioning

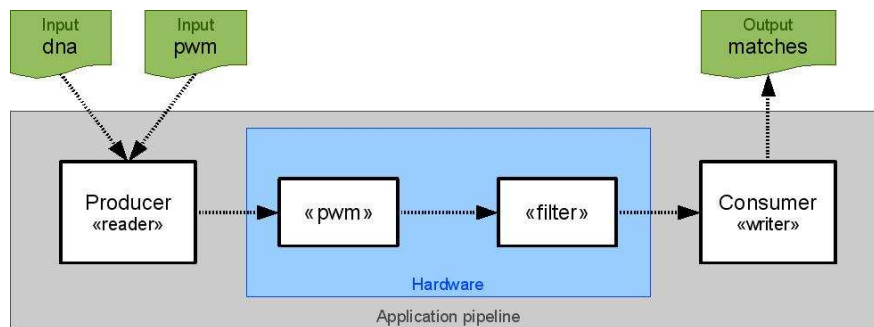


Figure 4.5: Application pipeline

Figure 4.5 show a suggested general architecture for the application, similar to that of the previously implemented VHDL solution presented in chapter 3.2. Following the nature of co-design, the application will be partitioned into a software section and a hardware section. All compute intensive work, such as executing the PWM-algorithm

itself, will be implemented in hardware for acceleration, while trivial tasks, such as reading input and writing output, will be implemented in software. All hardware processes must be specified as such, and located in a file separate from where the software processes are defined. This is to make sure that all parts of the application we want to run as hardware is translated correctly to VHDL, while the parts we want to run as software is compiled accordingly.

The planned partitioning of the FPWM system is illustrated in figure 4.5.

4.3.2 Implicit parallelism

Figure 4.6 indicate how computation can be accelerated with the help of parallel processing of the application pipeline. Implicitly achieving this means that the parallelism is extracted automatically from the implemented system. Tasks that can be executed simultaneously, will likely be executed simultaneously. This means being able to execute a new process iteration, and produce a new output, while the next process in the application pipeline manage the output computed in the previous one. When pipelining is enabled for inner code loops of a application through the use of the pipeline pragma, CoBuilder will attempt to parallelize the statements appearing within that loop with the goal of reducing the number of instruction cycles required to process the entire pipeline. Pipelined loops may not contain any nested loops.

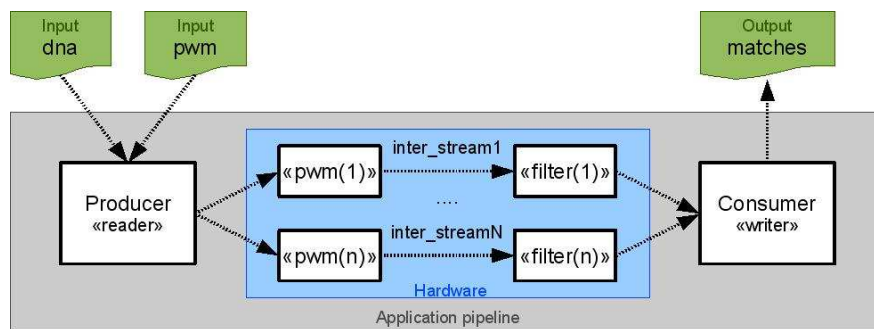


Figure 4.6: Implicit parallel processing

4.3.3 Input/Output scheme

Input The main input to the system will be the DNA sequence and the alignment matrix representing symbol frequencies in each sub-sequence (motif) position. The DNA sequence and the alignment matrix should be read from separate files; either in .dat-format or .txt-format. According to the PWM-specification supplied by Finn Drabløs at DMF [5], the input data itself will be expressed in Fasta-format. What using the Fasta-format implies is that both input sequences and input matrices must be preceded with a header line. This header line will contain a '>' symbol, indicating the start of a new sequence or matrix, and the identifier of the sequence or matrix to come. Example input data is illustrated, with the matrix already in PWM format, in figure 4.7.

```

>ENSG00000205139
ACCATATATACTGTAAAATCACAAATATATGTAAtatatacatatatacatataGTGAT
ATATGTGTGTATATATGCATACTTATACATGTATACATGTGTATATATACATATTTATGT
GTGTGCATATATATGTATACATCCCCAAACTATCTTAATTTAACTTTAAATCCAGTAATA
CTTTACAATAGAACATTCTT
>MA0801
-2.56  0.27  0.80 -0.96
-0.37 -0.96  0.93 -0.96
-2.56 -2.56  1.33 -2.56
 1.33 -2.56 -2.56 -2.56
-2.56 -2.56 -2.56  1.33
-2.56 -2.56 -2.56  1.33
 1.33 -2.56 -2.56 -2.56
 0.65 -0.96 -0.37  0.00

```

Figure 4.7: Example input data

```

ENSG00000205139  20  27 MA0801  7.85
ENSG00000205139 113 120 MA0801  2.14

```

Figure 4.8: Example output data

Output The output of the system will be a list containing relevant information about each of the filtered results, stored in a file. This final output data will be formatted with one hit per line, consisting of sequence ID, hit start, hit end, matrix ID and score. Example output data is illustrated in figure 4.8.

4.3.4 Application 'pipeline'

Application 'pipeline' stages

- Step 1 : Read input data from file - software
- Step 2 : Convert input alignment matrix - software
- Step 3 : Send input data to hardware - software
- Step 4 : Receive input data from software - hardware
- Step 5 : Compute result scores - hardware
- Step 6 : Filter result scores - hardware
- Step 7 : Send filtered results to software - hardware
- Step 8 : Receive filtered results from hardware - software
- Step 9 : Write filtered results to file - software

Read input data from file - software (step1)

One of the first things the application needs to do is to read the application input. As previously mentioned, all input will be read from files; the input matrix from one file, and the input sequence from another. The header line needs to be read first, trimming of the starter symbol ('>'), and stored for later use; after which the matrix or sequence can be read.

Reading the input matrix The input matrix will be read and stored into a two-dimensional integer array, using a designated function called from the 'reader' software process *Producer*.

Reading the input sequence The DNA sequence will be read and stored into a one-dimensional character array, using a designated function called from *Producer*.

Convert the input alignment matrix - software (step 2)

As the name of the 'reader' software process, *Producer*, indicates, this process produces the data needed by hardware to compute results. A part of this job is to make sure that the input matrix is converted to a matrix the PWM-algorithm can operate with. The input matrix is an alignment matrix, not a position-weight matrix (PWM), and have to be converted into a PWM before it is sent to hardware for execution of the PWM-algorithm.

The matrix can be converted using two functions; one for converting the individual values of a matrix, and another one calling the first function in order to convert the entire matrix. Exactly how these conversions are done is explained in chapter 2.4.

Floating-point values The weights in the PWM will be computed as floating-point values, as will the result score values being computed and filtered in hardware.

Send input data to hardware - software (step 3)

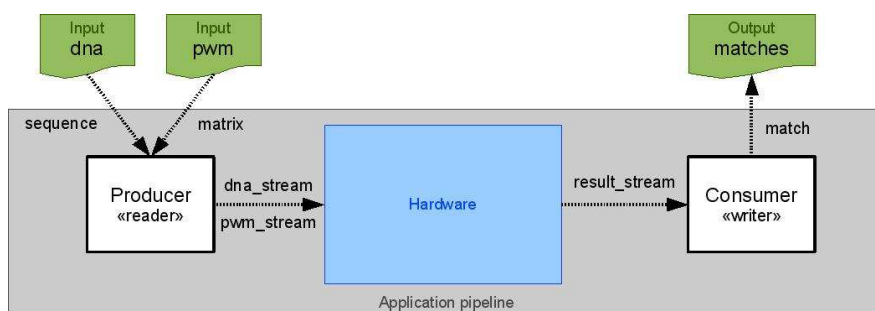


Figure 4.9: Data streaming

Figure 4.9 show a rough suggestion on how to perform streaming of data both to and from the software.

Before the application can start actual computation, the converted PWM and the DNA sequence has to be sent from software and received by the PWM-module in hardware. The length of the DNA sequence should also be sent to hardware, as it is used by the PWM-algorithm to keep track of when no more results can be produced.

Sending the weight matrix The weight matrix will be sent to hardware through streaming; one weight at a time, and one row at a time.

Sending the DNA sequence The DNA sequence will also be sent to hardware through streaming; one symbol at a time. All symbols should be sent in the same order as they appear in the sequence.

Receive input data from software - hardware (step 4)

The PWM-module implemented in hardware needs to receive matrix and sequence data from the software process *Producer* before it can start producing results. A stream implemented in Impulse-C is a form of FIFO-queue. This means that all data written to a stream will be read from the stream in the exact same order that it was written; first element written is the first element read.

Receiving the weight matrix The PWM will be read from stream and stored in a two-dimensional array straight away, as the hardware need the complete PWM to start computation.

Receiving the DNA sequence The DNA sequence will be read from stream in small portions at a time and stored into a one-dimensional array. As the hardware do not need the entire sequence to start computation, it is sufficient to have a portion of the sequence equivalent to a motif-length to start of with. The rest of the sequence can be read consecutively during computation; one new symbol for each new score to be computed. When a new symbol is read from software it is added to the array storing the DNA sequence, and the portion of the sequence for which the algorithm is computing a score for is updated.

Compute result scores - hardware (step 5)

The PWM-algorithm, described in chapter 2.4, will be implemented as a part of a hardware process called *PWM*. The process should only need to compute two values; a score computed for the current section of the DNA sequence, equivalent to a motif, and the position in the sequence in which this sub-sequence starts. The rest of the information relevant to each result will be added in software, before writing the results to file, for reasons to be explained soon.

After a result is computed it will be streamed to a second hardware process for filtering.

Filter result scores - hardware (step 6)

The hardware process *Filter* handles all filtering of results. This process will help the system to reduce the amount of strain on the bandwidth between software and hardware, when returning results, as well as to eliminate less relevant results.

There will be implemented two different means of filtering the results; one filtering results based on a given threshold, and another one adding all the scores together to a total score. Only one of these filters will be used when executing the PWM-application; there is no need to use both at the same time. Taking ease-of-use into consideration, the person using the application should have the possibility to choose a specific filter with the help of a command line argument when starting the application.

Send filtered results to software - hardware (step 7)

All filtered results will be streamed back to software, so that the 'writer' process *Consumer* can write relevant data about the results to file. Results can be written to stream and sent to software as soon as they pass through the filter.

Receive filtered results from hardware - software (step 8)

Filtered results sent from hardware are read from stream in software by the 'writer' process *Consumer*. As the name indicates, this process consumes the results produced by hardware.

Write filtered results to file - software (step 9)

When information about a result is received by *Consumer*, it is not complete and ready to write to the output file. First of all, we need to compute the position in the DNA sequence in which the result motif ends. This is a trivial enough task when the starting position and the motif length is given, to be computed in software. Saving this task for the consumer process will also help ease the strain on the bandwidth between hardware and software. The sequence ID and matrix ID should be accessible and ready to be written directly to file along with the other result data.

Chapter 5

Implemented Solutions

5.1 Introduction

This chapter presents the implemented Impulse-C co-design solutions, as well as the reasoning behind some of the design choices made (More design choices are discussed and explained in the synopsis). First, a rough summary of the system architecture is presented in section 5.2. Implementation of the software framework is then presented in section 5.3, while implementation of the PWM-module and the filter-module is presented in 5.4 and 5.5 respectively. Finally, a short users manual for the applications is presented in section 5.6.

In total there has been implemented four variations of the application; a basic solution and a multicore solution, both implemented in a floating-point and a 'fixed-point' version. The four implemented solutions are all based on the suggested solutions described in chapter 4. Impulse-C source code for all of the implemented solutions is included in appendix B:

- FPWM (*FPWM.h*, *FPWM_sw.c*, *FPWM_hw.c*): Basic solution
- FPWMi (*FPWMi.h*, *FPWMi_sw.c*, *FPWMi_hw.c*): 'Fixed-point' basic solution
- FPWM2008 (*FPWM2008.h*, *FPWM2008_sw.c*, *FPWM2008_hw.c*): Multicore solution
- FPWM2008i (*FPWM2008i.h*, *FPWM2008i_sw.c*, *FPWM2008i_hw.c*): 'Fixed-point' multicore solution

Brute force fixed-point There is no commonly-accepted standard for representing fixed-point numbers, as yet. However, Impulse-C provides support for fixed-point arithmetic in the form of macros and data types. Designing in Impulse-C, fixed-point applications are usually created from a well-tested floating-point implementation, rather than written from scratch; the process of converting a floating-point application to fixed-point is a non-trivial task, with several issues to consider. There was no time for creating a true Impulse-C fixed-point solution when working on this thesis. The implemented 'fixed-point' solutions presented in this thesis are therefore implemented using a more brute force tactic, due to time considerations, converting floating-point values to and

from 32-bit integer values. Fixed-point solutions utilizing the Impulse-C fixed-point macros and data types is left as future work.

5.2 Basic application architecture

5.2.1 Introduction

The Impulse-C FPWM system consists of an overlaying framework for control, just like the VHDL solution, and four underlying modules implemented as separate processes. These processes include a producer, a PWM- and a filter-module, as well as a consumer. A rough block diagram of the system architecture is shown in figure 5.1. The block diagram was generated during software simulation of the implemented solution.

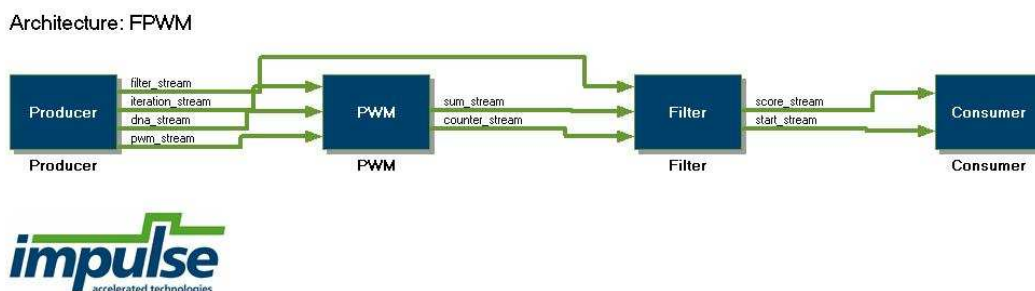


Figure 5.1: Implemented architecture

Following the nature of co-design, the application is partitioned into a software section and a hardware section. All compute intensive work, such as applying the PWM-algorithm and filtering the results, has been implemented in hardware for acceleration, while trivial tasks, such as reading input and writing output, has been implemented in software.

5.2.2 Framework and modules

Overlaying framework The overlaying framework initialize the application architecture and binds the software processes, running on the SMP-node, together with the hardware processes, running on the FPGA extension module.

Producer The producer functions as a reader-module, reading the DNA-sequence and initial alignment matrix from their respective input-files. After converting the alignment matrix to a PWM, it passes the PWM to the PWM-module in hardware. The DNA-sequence is streamed to the PWM-process exactly as it was read from the input-file.

PWM The PWM-module performs all the necessary computation, by applying the PWM-algorithm to the input PWM and DNA-sequence streamed to hardware from the

SMP-node. Result scores are streamed to the filter-process at the same rate as they are computed, together with a result index.

Filter The filter-module processes results computed by and streamed from the PWM-module. The amount of results can either be reduced, by applying threshold filtering, or made more useful, by applying summation filtering. Results are streamed to the SMP-node at the same rate as they pass through the selected filter.

Consumer The consumer functions as a writer-module, reading filtered results streamed to the SMP-node from hardware before writing them to an output-file, formatted with one hit per line. Consumer writes all results to file at the same rate as they are read from stream.

5.3 Implementation of the software framework - *FPWM*_sw.c*

The software framework consists of the the main-function, as well as the reader- and writer-processes *Producer* and *Consumer*.

5.3.1 Choosing filter

The user of the application is informed, by the framework, of what type of filter is being applied during execution. The applied filter is reported to the user as being a threshold filter if an additional argument is given together with the name of the application; the summation filter otherwise. As it is only the main-function that has access to the command line arguments, the main-function reports this directly. It is also the main-functions job to pass the threshold value, if set, as a parameter to *Producer*.

Choosing what filter to actually apply is done by passing the command line argument given at run time as a parameter to *Producer*. If the parameter is found to be NULL the summation filter will be applied; if not the threshold filter is applied. The producer must convert the parameter to a floating-point value in the latter case, as it is initially read as a string, before streaming the converted value to the hardware filter-process.

Variations

Threshold values are converted from floating point to 'fixed-point', directly after being read, in the 'fixed-point' solutions. In the multicore solutions a file containing threshold values for each matrix is given as input, instead of a single threshold value. These thresholds are all read from the given file and sent to their respective instance of the filter-module.

5.3.2 Reading and streaming input

Reading the input DNA-sequence and alignment matrix is done by *Producer*. Two designated functions are called by *Producer* to read and store the input, one for the DNA-sequence and another one for the matrix. These functions take the name and handler of their respective affiliated input file as parameters. It is a requirement that

the input is to be expressed in Fasta-format; giving the sequence and matrix their own header line. The header line consists of a '>' symbol, indicating the start of a new sequence or matrix, and the identifier of the sequence or matrix to come. The correct input format is illustrated by figure 4.7.

When reading the DNA-sequence, three different pieces of information are collected and stored for later use; the identifier, the sequence itself, and the length of the sequence. Except for the sequence identifier, this information is streamed to the hardware PWM-process using two separate streams, one designated stream for each piece of information.

The same type of information is also collected and stored about the alignment matrix, but with two major differences. First of all, the alignment matrix is converted to a PWM before being stored for later use, using the conversion algorithm (algorithm 1) from chapter 2.4. Secondly, we already know the number of rows and columns of the matrix before reading it. We know the length of the sub-sequences we want to compute scores for, giving us the number of rows, and the number of different nucleotide building blocks a DNA-sequence consists of, giving us the number of columns. Of the information collected about the matrix, only the converted PWM is streamed to the hardware PWM-process, using a designated stream.

All progress in reading and streaming of output, as well as conversion of the input alignment matrix to a weight matrix, is reported to the user.

Variations

PWM weights are converted and stored in the PWM as 'fixed-point' values in the 'fixed-point' solutions. In both multicore solutions the read and stream operations are repeated, once for each matrix in the input file. The matrices are sent to each their instance of the PWM-module along with a copy of the DNA-sequence. An array keeps track of which instance each matrix is sent to. At the moment, the parallel solution will read two matrices from the matrix input file. If more matrices are provided in the input file, only the two first matrices are read.

5.3.3 Reading and writing output

Reading the filtered results streamed from the hardware filter-process is done by *Consumer*, as well as expanding the results with more information before writing them to the output file. This final output data is formatted with one hit per line, consisting of sequence ID, hit start, hit end, matrix ID and score. Of this information, only hit start and score is received from hardware; sequence ID and matrix ID is already accessible for the software, while hit end is computed by appropriately expanding hit start. Summation filtering results in only one result being read from stream, processed, and written to the output file. Hit start and hit end is in this case '0' and '199' respectively.

Output data for each result is continuously reported to the user while it is being written to file, at the same rate as the results are read from stream.

Variations

In both multicore solutions the read and write operations are repeated twice, once for each instance of the filter-module to receive results from. The correct matrix ID to add

to each result is fetched from the previously mentioned array of matrix IDs based on the index of the filter that returned the score. Result scores are converted from 'fixed-point' to floating-point in both 'fixed-point' solutions before they are written to screen and to the output file.

5.4 Implementation of the general PWM-module

An instance of the PWM-module is created for each PWM to process. Source code for the module is located in the hardware file (*FPWM*_hw.c*).

5.4.1 Reading input from stream

The DNA-sequence is gradually read from stream during computation, at the minimum rate needed to compute results, while the entire PWM needs to be read and stored before computation can start.

5.4.2 Computing scores

As already mentioned, the PWM-module performs computation by applying the PWM-algorithm to the PWM and DNA-sequence streamed from software. This algorithm (algorithm 2) was previously presented in chapter 2.4 as part of the background information collected about the concept and use of PWMs.

New scores are computed as long as the counter keeping track of the current result index does not exceed the index of the last sub-sequence of the DNA-sequence. The index of the last sub-sequence is computed by subtracting the motif length from the DNA-sequence length.

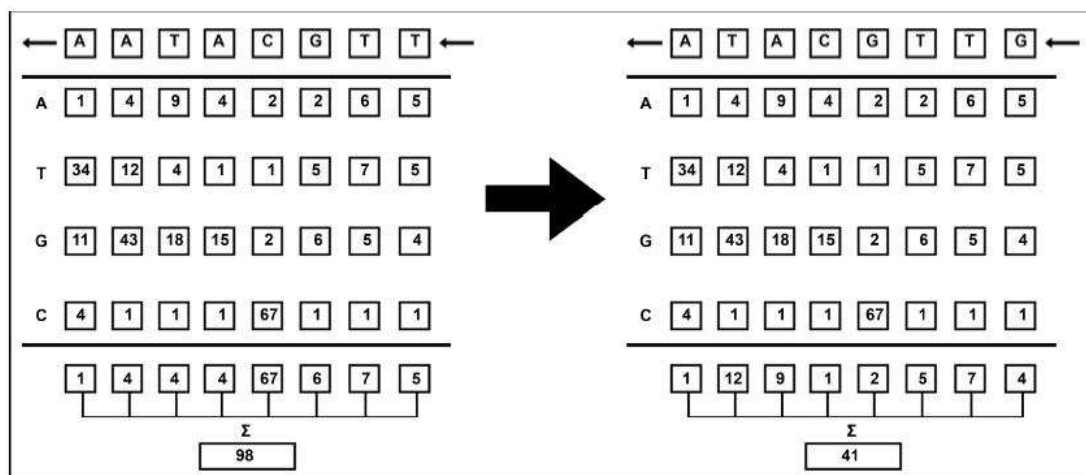


Figure 5.2: Computing score for a given sub-sequence [9]

Figure 5.2 illustrate how the PWM is, metaphorically speaking, moved along the input DNA-sequence one position at a time, and a score is calculated for each position from the weights of each symbol in the PWM. The score for the current sub-sequence is

computed as the sum of the weights found for each of its symbols. The index associated with each result score is the position in the DNA-sequence where the first visible position of the sub-sequence is located.

The following pseudo code describes how the correct weights to add to each score are found. Variables declared only once for the entire PWM-algorithm are mentioned here to make sense of the pseudo code.

Algorithm 3 Finding correct PWM-scores

```

N : int; {length of sub-sequence}
l : int; {current position in sub-sequence}
s : string; {DNA-sequence}
counter : int; {current position in DNA-sequence}
pwm : table[4][N]; {weight matrix}
sum : float ← 0;
if s[l+counter] == 'A' then
    sum+ = pwm[0][l];
else if s[l+counter] == 'C' then
    sum+ = pwm[1][l];
else if s[l+counter] == 'G' then
    sum+ = pwm[2][l];
else if s[l+counter] == 'T' then
    sum+ = pwm[3][l];
else
    {do nothing}
end if
  
```

The CO PIPELINE pragma is used on the inner loop of the algorithm in an attempt to speed up computation, this by increasing the level of parallelism extracted from the source code by the hardware compilers.

To conclude, two values are computed: the score for the current sub-sequence, and the index of the main DNA-sequence in which this sub-sequence starts. The rest of the information relevant to each result will be added later in the application pipeline. After a result is computed it is immediately streamed to the filter-module for further processing. The result score and index are streamed to the filter using each their designated stream.

5.5 Implementation of the general filter-module

An instance of the filter-module is created for each PWM-module. Source code for the module is located in the hardware file (*FPWM*_hw.c*).

5.5.1 Applying summation filter

The summation filter is applied if there is no threshold value written to the designated stream by *Producer* in software for the filter-module to read. All incoming result scores are added to a total score, while the affiliated result indexes are all ignored. When all computed scores have been read and added to the total score, the total is streamed to

Consumer in software together with a result index of 0. The result index is set to '0' as the total score represents the entire DNA-sequence.

5.5.2 Applying threshold filter

A threshold filter is applied if there is a threshold value written to the designated stream by *Producer* in software for the filter-module to read. All incoming result scores are compared with the given threshold value to see if they are higher than or equivalent to the threshold value. If they are, the scores are allowed to pass through the filter. Results are streamed to *Consumer* in software as soon as they pass through the threshold filter; the score and affiliated index in each their designated stream.

5.6 Generating VHDL and hardware

Although all the desired hardware functionality for the PWM application is expressed using Impulse-C, it has to be translated to VHDL by CoBuilder, the set of hardware generation tools, before an FPGA programming file can be generated. How CoBuilder works is roughly described in chapter 2.6. Summed up, the software-to-hardware compiler and optimizer translates C-language processes to (RTL) VHDL code, while optimizing the generated logic and identifying opportunities for parallelism.

From the generated VHDL code an FPGA programming is generated, using Xilinx ISE. Complete reports from the HDL build process are included in appendix C, while block diagrams of the implemented architectures are included in appendix D.

5.7 Users manual

This section presents a users manual for the implemented Impulse-C solutions, describing how to run them on the Musculus Cray XD1 platform. The manual presuppose that a fully functional FPGA programming file has been generated for each solution/application.

To start executing the application you have to have to first make sure you have access to all the required files, as well as a user account on a compatible Cray XD1 cluster such as Musculus at NTNU. The host name for Musculus is musculus.hpc.ntnu.no.

Requirements for the Cray XD1 cluster

- Xilinx Virtex-II Pro model xc2vp50, release 1.3 or newer
- Xilinx ISE development tools, version 7.1 with service pack 4, or newer (ISE 8.1 SP1 not supported)
- Optional: Third-party FPGA synthesis software with support for Xilinx FPGAs
- Optional: Third-party HDL simulation software

Required application files

- Generated folder *sw* - Necessary software source files
- *top.bin.ufp* - FPGA programming file
- *pwm.txt* - Input file containing the alignment matrix
- *dna.txt* - Input file containing the DNA-sequence
- *out.txt* - Output file

When executing the application, it could be recommended to log on to one of the compute nodes of Musculus, and not the log-in node (musculus403-6). Although the application should execute just fine on the log-in node, using one of the compute nodes is a lot safer. Following this recommendation will ensure that you will still have access to the remaining nodes if the one you are currently working on crashes.

If you decide to log on to the designated log-in node it is necessary to use the batch system to distribute the work onto the various compute nodes. As previously mentioned, it is possible to run applications on the log-in node, but this usage should be limited to simple tasks which do not represent a prolonged computational load. In other words, work which can be performed on the compute nodes should be performed there. There are several reasons behind this assertion. First of all, the log-in node is considerably slower than the compute nodes. Also, the speed of execution will be affected for everyone if one user runs a great load on the log-in node.

When logged on to a chosen compute node, the first thing to do is to make sure all required application files are located on an appropriate location, as well as in the same directory. After selecting the application directory as the working directory, the application can be started.

Running the basic application The basic application is started by writing `./FPWM [optional threshold value]`. Giving the application a threshold value will ensure that a threshold filter is applied to computed results; no given threshold value indicates a wish to apply the summation filter.

Running the 'fixed-point' basic application The basic 'fixed-point' application is started by writing `./FPWMi [optional threshold value]`. Giving the application a threshold value will ensure that a threshold filter is applied to computed results; no given threshold value indicates a wish to apply the summation filter.

Running the parallel computation application The multicore application is started by writing `./FPWM2008 [optional threshold input file]`. Giving the application a file containing threshold values for all matrices will ensure that a threshold filter is applied to computed results; no given threshold value indicates a wish to apply the summation filter.

Running the 'fixed-point' parallel computation application The 'fixed-point' multicore application is started by writing `./FPWM2008i [optional threshold input file]`. Giving the application a file containing threshold values for all matrices will ensure that a threshold filter is applied to computed results; no given threshold value indicates a wish to apply the summation filter.

Computed results is continuously written to screen during execution, as well as written to the output-file *out.txt*. Between each execution, there must be made a backup of the results stored in *out.txt* if there is a need for later review of these results.

Part IV

Analysis

Chapter 6

Results Analysis

6.1 Introduction

This chapter presents experiments done during simulation and execution of the implemented Impulse-C solutions, focusing on both the software framework and the hardware modules separately, as well as the solutions as a whole.

Section 6.2 presents tests done through software simulation, while section 6.3 covers observations done during efforts made to understanding the generated HDL code. The generated hardware logic is then analyzed in section 6.4. Finally, efforts made in attempts to successfully generate a functional FPGA programming file is described in section 6.5.

6.2 Software simulation

Through software simulation, the two floating-point solutions are tested with the intention to verify the implemented Impulse-C code; proving that the code describe correct functionality for both the software and the hardware partition. As the 'fixed-point' solutions are mostly identical to the floating-point solutions, except for the chosen score value representation in hardware, simulating these have not been a priority. Also, it is already known that the summation filter in the 'fixed-point' solutions does not work as intended; this from debugging efforts during the development process.

Testing is done by simulating with different combinations of input data, once for each filter. The actual combinations are presented later in this section, for both solutions tested.

6.2.1 Input data

For the software simulation tests, a selection of alignment matrices and DNA sequences have been chosen as input. All of these matrices and sequences are in Fasta-format, with no additional information provided in the header line after the matrix/sequence ID. The selected alignment matrices are shown below in table 6.1.

As the second matrix is the reverse of the first matrix, it is also representing scores for the reversed motifs. Both matrices will have the same threshold value of 1.15. The DNA sequences selected as simulation input are shown in table 6.2.

>MA0801a				>MA0801b			
0	4	7	1	6	1	2	3
2	1	8	1	12	0	0	0
0	0	12	0	0	0	0	12
12	0	0	0	0	0	0	12
0	0	0	12	12	0	0	0
0	0	0	12	0	0	12	0
12	0	0	0	2	1	8	1
6	1	2	3	0	4	7	1

Table 6.1: Input alignment matrices

```
>ENSG00000205139
ACCATTATATACTGTAAAATCACAAATATATGTAATATATACATATATACATATAGTGAT
ATATGTGTGTATATATGCATACTTATACATGTATACATGTGTATATATACATATTTATGT
GTGTGCATATATATGTATACATCCCCAAACTATCTTAATTTAACTTTAAATCCAGTAATA
CTTACAATAGAACATTCTT
```

```
>ENSG00000208641
AGTTATCCACACCTCTATTTCTTGTATGCATTGCATATTACACTTTTATTCCCAAAGAGG
CACTATTTTGGGCTACCATGTTTAGACACATTTATCAAATAGTCTTTCTAGATTTGTTCA
TTTGTCCATGCTCTTTTTTCAGATCCCCTCCTGGGCCTAGCACAGGTAAGTGTGTGCTGGGC
TAAACTGAAATGAATATGAA
```

```
>ENSG00000105971
GTTAGAATTTTATGTGAAATTAACATTTAATTCTCACGGACACCCCTGAAACAGATGCCA
CAGCCCCCATTTTGGCAACGAGGCAGCTGAGGTTCCCAGAGGCTCAATACCAGCACCATG
AGCCGCAGCACGCAAGGCAAACACAGCCGGAGGTGAGCACATACCTGCTTCGCACCCCAT
GCGCCTAACCACAAGGTTCC
```

Table 6.2: Input DNA sequences

The length of the sequences chosen as test input are only a fraction of that of realistic input sequences to an FPWM system, but will serve well as test input when simulating the Impulse-C solutions as they are implemented today.

6.2.2 Software partition

For analysis purposes the software is extended somewhat, printing additional information to screen during execution. Questions which are answered about the software partition during simulation are:

- ...whether the filter chosen by the user is registered by the software: OK
- ...whether the alignment matrix is read correctly from the input file: OK
- ...whether the alignment matrix is converted correctly to PWM: OK
- ...whether the DNA sequence is read correctly from the input file: OK

- ...whether the PWM and DNA sequence is successfully sent to hardware: OK
- ...whether filtered results are received successfully from hardware: OK
- ...whether the correct matrix and sequence data is connected to each result: OK
- ...whether filtered results are written to screen and file correctly: OK
- ...whether the application terminates correctly: OK

>MA0801a				>MA0801b			
-2.56	0.27	0.80	-0.96	0.65	-0.96	-0.37	0.00
-0.37	-0.96	0.93	-0.96	1.33	-2.56	-2.56	-2.56
-2.56	-2.56	1.33	-2.56	-2.56	-2.56	-2.56	1.33
1.33	-2.56	-2.56	-2.56	-2.56	-2.56	-2.56	1.33
-2.56	-2.56	-2.56	1.33	1.33	-2.56	-2.56	-2.56
-2.56	-2.56	-2.56	1.33	-2.56	-2.56	1.33	-2.56
1.33	-2.56	-2.56	-2.56	-0.37	-0.96	0.93	-0.96
0.65	-0.96	-0.37	0.00	-2.56	0.27	0.80	-0.96

Table 6.3: Converted matrices

The converted alignment matrices (PWMs) are shown in table 6.3. The software gives the floating-point values calculated to six decimal places, so the reported weights in the table are rounded off values.

Basic application

Test	Input		Threshold	# Results
	DNA sequence	Alignment matrix		
1a	ENSG00000205139	MA0801a	-	1
1b	ENSG00000205139	MA0801a	1.15	0
2a	ENSG00000205139	MA0801b	-	1
2b	ENSG00000205139	MA0801b	1.15	0
3a	ENSG00000208641	MA0801a	-	1
3b	ENSG00000208641	MA0801a	1.15	2
4a	ENSG00000208641	MA0801b	-	1
4b	ENSG00000208641	MA0801b	1.15	4
5a	ENSG00000105971	MA0801a	-	1
5b	ENSG00000105971	MA0801a	1.15	1
6a	ENSG00000105971	MA0801b	-	1
6b	ENSG00000105971	MA0801b	1.15	0

Table 6.4: Tests done on basic solution

Table 6.4 above presents the tests done through simulation of the basic solution, processing only one matrix at a time. Here, the input data is presented, as well as the number of filtered results.

Test	Results				
1a	ENSG00000205139	0	199	MA0801a	-1562.766357
2a	ENSG00000205139	0	199	MA0801b	-1569.915894
3a	ENSG00000208641	0	199	MA0801a	-1702.865112
3b	ENSG00000208641	33	40	MA0801a	1.687562
	ENSG00000208641	176	183	MA0801a	1.238545
4a	ENSG00000208641	0	199	MA0801b	-1705.091309
4b	ENSG00000208641	35	42	MA0801b	2.643074
	ENSG00000208641	64	71	MA0801b	1.238545
	ENSG00000208641	79	86	MA0801b	2.275349
	ENSG00000208641	96	103	MA0801b	2.709213
5a	ENSG00000105971	0	199	MA0801a	-1854.153076
5b	ENSG00000105971	15	22	MA0801a	3.831082
6a	ENSG00000105971	0	199	MA0801b	-1848.449097

Table 6.5: Filtered results - basic solution

Filtered results from each test done on the basic solution is presented in table 6.5. These results will later be compared to those of the multicore solution.

Multicore application

Test	Input		Threshold	# Results
	DNA sequence	Alignment matrices		
7a	ENSG00000205139	MA0801a	-	1
		MA0801b	-	1
7b	ENSG00000205139	MA0801a	1.15	0
		MA0801b	1.15	0
8a	ENSG00000208641	MA0801a	-	1
		MA0801b	-	1
8b	ENSG00000208641	MA0801a	1.15	2
		MA0801b	1.15	4
9a	ENSG00000105971	MA0801a	-	1
		MA0801b	-	1
9b	ENSG00000105971	MA0801a	1.15	1
		MA0801b	1.15	0

Table 6.6: Tests done on multicore application

Table 6.6 above presents the tests done through simulation of the multicore solution, processing two matrices at a time in parallel. Here, the input data is presented, as well as the number of filtered results. As expected, the number of filtered results produced for each matrix-sequence pair is the same for this solution as for the basic solution.

Filtered results from each test done on the multicore solution is presented in table 6.7. As expected, the filtered results produced for each matrix-sequence pair is the same for this solution as for the basic solution.

Test	Results				
7a	ENSG00000205139	0	199	MA0801a	-1562.766357
	ENSG00000205139	0	199	MA0801b	-1569.915894
8a	ENSG00000208641	0	199	MA0801a	-1702.865112
	ENSG00000208641	0	199	MA0801b	-1705.091309
8b	ENSG00000208641	33	40	MA0801a	1.687562
	ENSG00000208641	176	183	MA0801a	1.238545
	ENSG00000208641	35	42	MA0801b	2.643074
	ENSG00000208641	64	71	MA0801b	1.238545
	ENSG00000208641	79	86	MA0801b	2.275349
	ENSG00000208641	96	103	MA0801b	2.709213
9a	ENSG00000105971	0	199	MA0801a	-1854.153076
	ENSG00000105971	0	199	MA0801b	-1848.449097
9b	ENSG00000105971	15	22	MA0801a	3.831082

Table 6.7: Filtered results - multicore application

6.2.3 Hardware partition

For analysis purposes, the CoDeveloper Application Monitor is utilized to gather information about what happens in the hardware partition of the application during the simulation.

PWM module

Questions which are answered about the hardware PWM module during simulation are:

- ...whether the correct sequence length is received from software: **OK**
- ...whether the correct amount of result scores are computed: **OK**
- ...whether the computed scores are seemingly correct: **OK**

All sequences used for testing have a length of 200 bases and the PWM module(s) computes 193 scores (with affiliated indexes) for each matrix-sequence pair; one score for each sub-sequence. The correctness of these scores have been tested during implementation by printing the steps of computing each score to the simulation log shown in CoDeveloper Application Monitor. A weight is added to a score for all bases of the current sub-sequence, and the added weights are fetched from the correct row and column in the matrix. There are no overflow issues adding the weights together.

Filter

Questions which are answered about the hardware filter module during simulation are:

- ...whether the correct filter is applied: **OK**
- ...whether all results from the PWM module are received and processed: **OK**
- ...whether the applied filter perform correct filtering of all scores: **OK**

6.3 Generated HDL code

Even though the HDL files generated by CoBuilder contain no comments explaining the generated HDL code, some understanding can be gained through studying the actual code. 9-10 hours were set aside for making an effort to understand the generated HDL code and hardware logic.

The component- and system-level HDL files generated by CoBuilder are found in files named as follows:

- *FPWM*_comp.vhd*
- *FPWM*_top.vhd*
- *rt_impulse_FPWM*.vhd*

These files are too big to include in the thesis appendix.

6.3.1 *FPWM*_comp.vhd*

This file contains implementation of the application specific hardware modules. An rtl architecture is created for each module, together with an entity defining signals for all input and output streams. Each stream is also defined with affiliated interface signals.

For the floating-point solutions the floating-point HDL library is included. But all solutions include a reference to external components, as they need to. Each design only define one clock signal, as only one clock is supported by the hardware platform. States determining what actions which will take place each clock cycle are created, defining an appropriate flow for the application. Use of the Impulse-C pragma CO PIPELINE have also created a pipeline for the loop adding correct PWM scores to each result score. The six stages of the pipeline are implemented as records.

```
component PWM_sequence_RAM is
  port (signal rst : in std_ulogic;
        signal clk : in std_ulogic;
        signal we : in std_ulogic;
        signal addr : in std_ulogic_vector (8 downto 0);
        signal addr2 : in std_ulogic_vector (8 downto 0);
        signal din : in std_ulogic_vector (7 downto 0);
        signal dout : out std_ulogic_vector (7 downto 0);
        signal dout2 : out std_ulogic_vector (7 downto 0));
end component;
```

For arrays storing input data in hardware a RAM entity, dualsync architecture and RAM component is created. An example of a RAM component is shown above. Arrays also need memory signals, signals which are created for each array. Regular variables, on the other hand, are implemented only with the help of a single signal.

The actual functionality of the hardware modules, as it was translated to HDL, is too confusing and partially cryptic to understand in the time set to study the HDL code.

6.3.2 *FPWM*_top.vhd*

This file contains definition of the application specific architecture. An entity defining signals for all input and output and output signals for the architecture is created. As intermediate streams between components of the architecture is left out, the architecture definition can be viewed as a 'black box'. The hardware modules are connected to the architecture defined as components of the architecture. Stream signals defined for the architecture are now connected to the correct component (module). A wrapper for the architecture is also created, and the architecture is connected to an architecture structure defined as a component of the structure.

```

architecture structure of FPWM is
  component PWM is
    port (
      reset : in std_ulogic;
      sclk : in std_ulogic;
      clk : in std_ulogic;
      p_pwm_stream_rdy : in std_ulogic;
      p_pwm_stream_en : inout std_ulogic;
      p_pwm_stream_eos : in std_ulogic;
      p_pwm_stream_data : in std_ulogic_vector (31 downto 0);
      p_iteration_stream_rdy : in std_ulogic;
      p_iteration_stream_en : inout std_ulogic;
      p_iteration_stream_eos : in std_ulogic;
      p_iteration_stream_data : in std_ulogic_vector (15 downto 0);
      p_dna_stream_rdy : in std_ulogic;
      p_dna_stream_en : inout std_ulogic;
      p_dna_stream_eos : in std_ulogic;
      p_dna_stream_data : in std_ulogic_vector (7 downto 0);
      p_counter_stream_rdy : in std_ulogic;
      p_counter_stream_en : inout std_ulogic;
      p_counter_stream_eos : out std_ulogic;
      p_counter_stream_data : out std_ulogic_vector (15 downto 0);
      p_sum_stream_rdy : in std_ulogic;
      p_sum_stream_en : inout std_ulogic;
      p_sum_stream_eos : out std_ulogic;
      p_sum_stream_data : out std_ulogic_vector (31 downto 0)
    );
  end component;

  component Filter is
    port (
      reset : in std_ulogic;
      sclk : in std_ulogic;
      clk : in std_ulogic;
      p_threshold_stream_rdy : in std_ulogic;
      p_threshold_stream_en : inout std_ulogic;
      p_threshold_stream_eos : in std_ulogic;
      p_threshold_stream_data : in std_ulogic_vector (31 downto 0);
      p_counter_stream_rdy : in std_ulogic;
      p_counter_stream_en : inout std_ulogic;
      p_counter_stream_eos : in std_ulogic;
      p_counter_stream_data : in std_ulogic_vector (15 downto 0);
      p_sum_stream_rdy : in std_ulogic;
      p_sum_stream_en : inout std_ulogic;
      p_sum_stream_eos : in std_ulogic;
      p_sum_stream_data : in std_ulogic_vector (31 downto 0);
      p_start_stream_rdy : in std_ulogic;
      p_start_stream_en : inout std_ulogic;
      p_start_stream_eos : out std_ulogic;
      p_start_stream_data : out std_ulogic_vector (15 downto 0);
      p_score_stream_rdy : in std_ulogic;
    );
  end component;
end structure;

```

```

    p_score_stream_en : inout std_ulogic;
    p_score_stream_eos : out std_ulogic;
    p_score_stream_data : out std_ulogic_vector (31 downto 0)
  );
end component;

```

The architecture structure defining the hardware modules as components is shown above. Signals are created for each stream and the streams are initialized. All stream signals are then mapped to the correct module instance when these are created.

6.3.3 *rt_impulse_FPWM*.vhd*

This file contains system-level code providing necessary communication between the software and the application specific hardware design over the RapidArray transport interface (RT). Streams running between software and hardware over RT are defined and mapped, and signals assisting in communication through streams are created. All signals are clocked using the same clock. A MUX, implemented as a state machine, controls when to write to or read from the different streams. This MUX is shown below.

```

rt_idata_reg_mux : process (p_Producer_threshold_stream_rt_idata_reg ,
    p_Producer_pwm_stream_rt_idata_reg ,
    p_Producer_iteration_stream_rt_idata_reg ,
    p_Producer_dna_stream_rt_idata_reg , p_Consumer_start_stream_rt_idata_reg ,
    p_Consumer_score_stream_rt_idata_reg , rt_conn_sel) is
begin
    case rt_conn_sel is
        when "000" => rt_resp_rdata <= p_Producer_threshold_stream_rt_idata_reg
            ;
        when "001" => rt_resp_rdata <= p_Producer_pwm_stream_rt_idata_reg;
        when "010" => rt_resp_rdata <= p_Producer_iteration_stream_rt_idata_reg
            ;
        when "011" => rt_resp_rdata <= p_Producer_dna_stream_rt_idata_reg;
        when "100" => rt_resp_rdata <= p_Consumer_start_stream_rt_idata_reg;
        when "101" => rt_resp_rdata <= p_Consumer_score_stream_rt_idata_reg;
        when others => rt_resp_rdata <= (others => 'X');
    end case;
end process rt_idata_reg_mux ;

```

6.3.4 VHDL library

For the purpose of simulation and synthesis, the generated application specific component- and system-level HDL files are combined with additional HDL wrapper components. These components are provided in a special library called "impulse". Wrapper components makes CoBuilder capable of generating the required software/hardware interfaces for the target platform Musculus. The software/hardware interfaces specially generated for Musculus are provided as HDL code in special HDL files as well, and must be included in the same project VHDL library as the application specific HDL code.

6.4 Generated hardware logic

Table 6.8 shows design statistics reported by Xilinx ISE after synthesizing and implementing the designs from the HDL files generated by CoBuilder. The Xilinx Virtex-II

Pro FPGA has a speed grade of -7.

	Device utilization	Minimum period	Maximum frequency
FPWM	16%	12.436ns	80.412MHz
FPWM2008	21%	13.954ns	71.664MHz
FPWMi	14%	10.084ns	99.167MHz
FPWM2008i	18%	11.490ns	87.032MHz

Table 6.8: Design statistics

More detailed information on device utilization is presented in table 6.9.

	Occupied Slices	Slice Flip Flops	4 input LUTs
FPWM	16%	9%	7%
FPWM2008	21%	11%	10%
FPWMi	14%	9%	6%
FPWM2008i	18%	10%	8%

Table 6.9: Device utilization

Even though there were no time to generate functional FPGA programming files for the implemented solutions, some limited understanding of the generated hardware logic can be gained. Methods for doing this include studying the generated HDL code, as well as the reports from the actual HDL build process. 9-10 hours were set aside for making an effort to understand the generated HDL code and hardware logic. The complete reports generated from the HDL build process of the implemented solutions are included in appendix C.

6.4.1 PWM module

17 blocks are generated for the PWM module. The total number of stages differ between the floating-point and the 'fixed-point' implementation; the floating-point implementation requires 30 stages, while the 'fixed-point' implementation requires only 27 stages. Maximum unit delay is however 33 for both implementations.

Analyzing the PWM module further, it is reported that multiple access to the PWM array reduces the minimum rate to two in the pipelined loop adding PWM scores to the result scores. A warning is also reported, saying that recursively used variables may reduce the pipeline rate.

```

|-----
| Operators:
| 4 Adder(s)/Subtractor(s) (5 bit)
| 2 Adder(s)/Subtractor(s) (9 bit)
| 1 Adder(s)/Subtractor(s) (16 bit)
| 7 Adder(s)/Subtractor(s) (32 bit)
| 2 Comparator(s) (2 bit)
| 4 Comparator(s) (8 bit)
| 7 Comparator(s) (32 bit)
| 4 Floating-point Adder(s)/Subtractor(s) (32 bit)
|-----

```

From the general view of generated operators one can see that the only difference in operators between the floating-point and the 'fixed-point' solution is the type of four

of the 32-bit adder(s)/subtractor(s) utilized. In the floating-point solution these need to be floating-point operators. Operators for the floating-point PWM module is shown above.

6.4.2 Filter module

The number of blocks generated for the filter module differ between the floating-point and the 'fixed-point' implementation; the floating-point implementation has 13 blocks, while the 'fixed-point' implementation has 14 stages. The total number of stages differ between the floating-point and the 'fixed-point' implementation also for the filter module; the floating-point implementation requires 22 stages, while the 'fixed-point' implementation requires only 17 stages. Maximum unit delay is however 32 for both implementations, less than for the PWM module.

```

-----
| Operators:
| 1 Adder(s)/Subtractor(s) (32 bit)
| 3 Comparator(s) (2 bit)
| 2 Comparator(s) (3 bit)
| 2 Comparator(s) (32 bit)
|-----

```

From the general view of generated operators one can see that the difference in operators between the floating-point and the 'fixed-point' solution is that the 'fixed-point' solution requires two extra comparators. Both the extra block and the extra comparators could be explained by the extra if-block in the fixed-point Impulse-C implementation. Operators for the 'fixed-point' filter module is shown below. In the floating-point solution, the adder/subtractor is of course a floating-point operator.

6.4.3 StageMaster

```

do {
  suif_tmp_0 = 1 { sequence [1+0 suif_tmp90] } 1 ;
  suif_tmp5 = 1 suif_tmp_0 == 1 65 ;
  if (suif_tmp5) {
    sum = 5 sum + 2 pwm [ ] 2 ;
  } else {
    suif_tmp6 = 1 suif_tmp_0 == 1 67 ;
    if (suif_tmp6) {
      sum = 5 sum + 2 pwm [ ] 2 ;
    } else {
      suif_tmp7 = 1 suif_tmp_0 == 1 71 ;
      if (suif_tmp7) {
        sum = 6 sum + 3 pwm [ ] 3 ;
      } else {
        suif_tmp8 = 1 suif_tmp_0 == 1 84 ;
        if (suif_tmp8) {
          sum = 6 sum + 3 pwm [ ] 3 ;
        } else {
        }
      }
    }
  }
}

#pragma CO PIPELINE
#pragma CO SET stageDelay 32
switch(sequence[1+counter]) {
  case((char)'A'):
    sum += pwm[0][1];
    break;
  case((char)'C'):
    sum += pwm[1][1];
    break;
  case((char)'G'):
    sum += pwm[2][1];
    break;
  case((char)'T'):
    sum += pwm[3][1];
    break;
  default:
    // Do nothing here
    break;
}

```

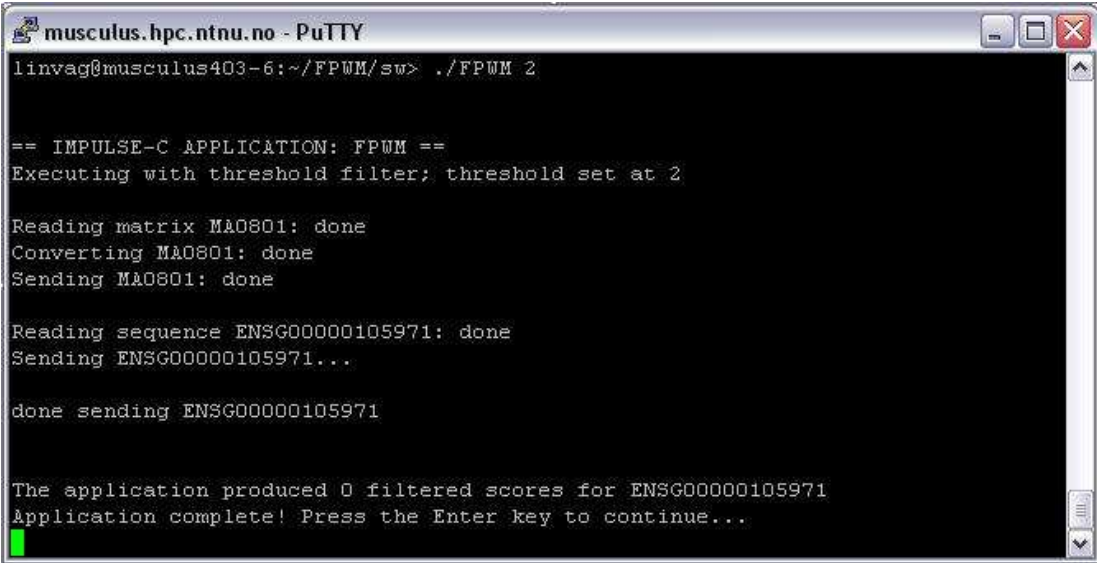
Figure 6.1: Pipelined loop in StageMaster vs source code

Simulating generated hardware logic with the help of the StageMaster debugger gives an interesting view of the instruction flow within the two hardware processes (modules). StageMaster shows which instructions that are executed at the same time, and how some lines of code take multiple stages to complete. Figure 6.1 shows the pipelined loop of the PWM algorithm.

6.5 FPGA programming file

Unfortunately, there were no time to successfully generate a fully functional FPGA programming file for the solutions. The main theory this far is that because the ISE project had to be created from scratch, and not from the template generated by CoBuilder, the properties applied during the different stages of generation the FPGA programming file were incompatible with the design. Several properties had to be changed in order to generate a programming file at all, functional or not. An example of such a property is the Xilinx specific property adding I/O buffers to signals. This property had to be disabled as all necessary I/O buffers were already included in the design as a part of implementing Impulse-C streams.

Non functional means, in this case, that the FPGA programming file is (apparently) successfully loaded on the target FPGA, but none of the expected results are returned by the FPGA, as illustrated in figure 6.2.



```
musculus.hpc.ntnu.no - PuTTY
linvag@musculus403-6:~/FPWM/sw> ./FPWM 2

== IMPULSE-C APPLICATION: FPWM ==
Executing with threshold filter; threshold set at 2

Reading matrix MA0801: done
Converting MA0801: done
Sending MA0801: done

Reading sequence ENSG00000105971: done
Sending ENSG00000105971...

done sending ENSG00000105971

The application produced 0 filtered scores for ENSG00000105971
Application complete! Press the Enter key to continue...
```

Figure 6.2: First faulty FPGA programming file on Musculus

Questions which are answered about the FWPM system during execution on Musculus are:

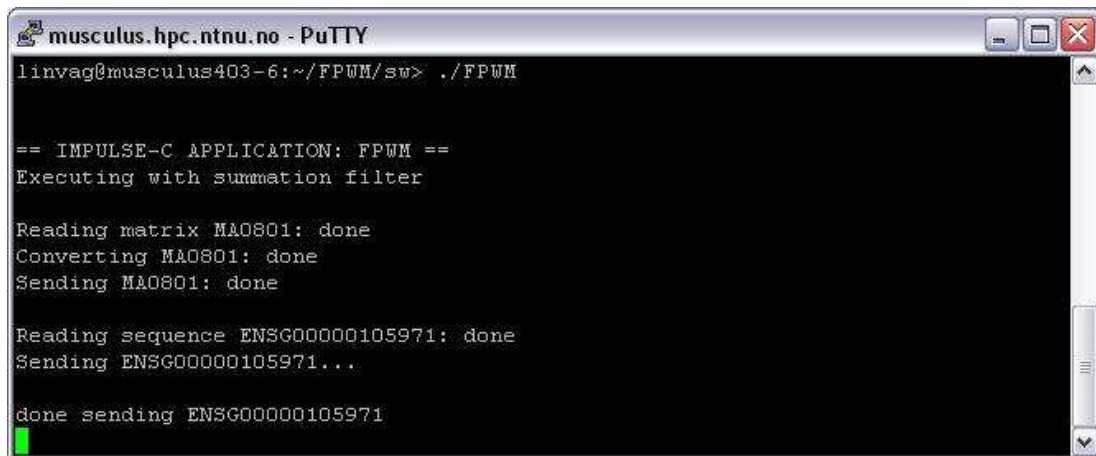
- ...whether the filter chosen by the user is registered by the software: OK
- ...whether the alignment matrix is read successfully from the input file: OK

- ...whether the alignment matrix is converted successfully to PWM: OK
- ...whether the DNA sequence is read successfully from the input file: OK
- ...whether the PWM and DNA sequence is sent successfully to hardware: OK
- ...whether the data sent from software is received by the hardware: FAILED
- ...whether the correct amount of result scores are computed: NOT TESTABLE
- ...whether the computed scores are correct: NOT TESTABLE
- ...whether the correct filter is applied: NOT TESTABLE
- ...whether (all) results from the PWM module are processed by the filter: NOT TESTABLE
- ...whether the applied filter perform correct filtering of all scores: NOT TESTABLE
- ...whether filtered results are received successfully from hardware: FAILED
- ...whether the correct matrix and sequence data is connected to each result: NOT TESTABLE
- ...whether filtered results are written to screen and file correctly: NOT TESTABLE
- ...whether the application terminates correctly: FAILED/NOT TESTABLE

Various changes to the properties in Xilinx ISE were done in attempts at getting closer to a fully functional programming file to run on the FPGA. One attempt involved fully duplicating the properties applied during the generation of a known functional FPGA programming file. This functional programming file belonged to a Cray XD1 example project supplied with CoDeveloper. These attempts had some kind of impact on the generated programming file, as illustrated in figure 6.3, but did not result in a functional programming file.

A large amount of warnings reported during the ISE synthesis process talk about unconnected signals in the RT (RapidArray Transport) part of the design. However, some of the unconnected signals reported are connected in the VHDL code and should be so in the generated FPGA programming file also.

For a while, incompatible/incorrect timing constraints was also believed to play a part in the generated programming file not functioning. The frequency of the converted FPGA programming file was set both to the minimum and the maximum frequency. Of course, as vital signals were not connected in the FPGA programming file, no apparent impact on the programming file was observed during these experiments.



```
musculus.hpc.ntnu.no - PuTTY
linvag@musculus403-6:~/FPWM/sw> ./FPWM

== IMPULSE-C APPLICATION: FPWM ==
Executing with summation filter

Reading matrix MA0801: done
Converting MA0801: done
Sending MA0801: done

Reading sequence ENSG00000105971: done
Sending ENSG00000105971...

done sending ENSG00000105971
```

Figure 6.3: Second faulty FPGA programming file on Musculus

Part V

Synopsis

Chapter 7

Discussion

7.1 Introduction

This chapter presents a discussion on the implemented Impulse-C solutions; comparing them to both the planned solution and the previously implemented VHDL solution. The newly implemented Impulse-C solutions are discussed in section 7.2, then compared to the VHDL solution in following sections. Ease-of-use of the CoDeveloper environment is discussed in section 7.3. Productivity and performance when comparing the Impulse-C solutions and the existing VHDL-based solution is discussed in section 7.4 and section 7.5 respectively. The chapter is finally rounded off with some thoughts and reflections on unanswered questions in section 7.6.

7.2 Implemented solutions

7.2.1 Functionality and features

Implemented variations In this discussing there will be references to Impulse-C solutions in plural, but in reality they are all variations of the same basic solution. Each of the four implemented variations use a different combination of PWM weight value representation and number of matrices processed. The reason behind implementing these multiple variations is that it was considered to be a good way of testing and comparing the effects of floating-point arithmetic's and fixed-point arithmetic's in hardware, for a multicore implementation as well as a basic implementation.

Floating-point vs fixed-point There is, as yet, no commonly-accepted standard for representing fixed-point numbers. Floating-point, on the other hand, is codified in IEEE standard. Impulse-C does however provide support for a chosen standard of fixed-point arithmetic in the form of macros and data-types. Fixed-point is an alternative to floating-point, but has a smaller range of values and/or less precision. Converting from floating-point to fixed-point is a non trivial task and is easier done on a well tested floating-point solution, as the decision to scale fixed-point variables depends on the actual values each variable is expected to take on. Floating-point variables must be converted to fixed-point individually, and needs to be scaled throughout computation to align decimal points, prevent overflow and manage precision. The FPWM system

should be tested further before attempting to convert the floating-point alternatives to Impulse-C fixed-point.

Command line arguments A missing feature of the implemented solutions is the possibility of processing multiple input command line arguments. This is influencing the possibility to choose which files to read input data from. Processing multiple matrices in parallel reduces this drawback somewhat, as the user does not have to replace the content of the default input files as often. The software should be able to process multiple command line arguments in the future, but proved to be more work than anticipated when implementing the Impulse-C solutions. The list of arguments can not be processed directly in the main function, but need to be passed as a parameter first to the architecture definition and then to the appropriate software function. This turned out to be a less straight forward task than it sounds, thus the possibility to choose which matrix and sequence input files to read from was given a low priority. Being able to choose the name and location of the matrix and sequence input files were deemed to be more a question of increased user friendliness than of being a vital functionality.

Ignoring regions of DNA The Fasta-format allows for additional information about a matrix or sequence in its header line. The implemented solutions however, does not support this. It has been contemplated whether or not to support the possibility of ignoring regions of the DNA sequence, a feature which requires index considerations to be made, but was valued as a low priority. In addition to being mentioned in the header line, regions to ignore are also usually indicated by being typed in small letters. The implemented solutions have no support for such sequences either, as of yet.

DNA sequence lengths At the moment, only sequences with a length of 300 or less symbols are supported by the implemented Impulse-C solutions. Supporting the use of longer DNA-sequences is easy up to a certain point. In the source code it is just a matter of changing a single constant, increasing the depth of the FIFO queue for the sequence stream. In hardware, larger sequences at up to 300 Mbases could require a large amount of RAM components to be implemented, which is not favorable as it would limit the amount of space left on the FPGA for the rest of the design. In any case, it does not seem like CoDeveloper/CoBuilder would allow stream depths of 300 Mbases to be created in the first place. Either way, long sequences could be divided into several smaller sections and sent to hardware one such section at a time.

Parallel computation Implementing the use of multiple instances of the hardware modules in parallel one need to take the required number of parameters (memory, streams, signals etc.) into consideration. Impulse-C support the use of maximum 32 parameters for each process. The amount of streams, signals and memory supported by the hardware itself is also limited due to general resource limitations such as routing capabilities and available logic blocks.

Pipelining Pipelining of instructions is not automatic, but requires an explicit declaration. This declaration must be included within the body of a loop and prior to

any statements that are to be pipelined. Implicit flattening of logic is applied when using the CO PIPELINE pragma. Because of this, one need to consider the size of the required logic before choosing to pipeline a loop. Loops with many iterations are not well suited for pipelining, while nested loops are not suited at all. One also have to consider if there are any potential benefits in pipelining a specific loop at all. Only one loop was found somewhat suitable for pipelining in the implemented solutions, and that was the loop adding matrix weights to the score of the current sub-sequence.

Threshold values Choosing correct threshold filter values for matrices in the multi-core solutions is done by listing them in the same order in the threshold input file as the matrices are listed in the matrix input file. This was deemed to be intuitive, or that it at least should be.

Filters Implemented filtering is summation and threshold filtering, which is the same filters as in the existing VHDL solution. More filter types could be implemented in the future, though there is no obvious need for it at the moment. There is also a question about what other filters that could be applicable, if there are any. In any case, more filters to choose from would introduce a need to implement a different method of determining what filter to apply also. Command line parameter flags, instead of an optional threshold value, could be utilized.

Shared memory It has been contemplated whether or not to include the use of shared memory or not, as the Impulse-C FPWM system is implemented today. There is a higher transfer rate with shared memory, compared to that of Impulse-C streams, when using one or both of the Xilinx Virtex-II Pro PowerPCs. There is however no use of the PowerPC technology in the implemented solutions. As there is no need for the PWM module to have read and locally stored the entire DNA sequence at the start of computation, the sequence is rather read consecutively during computation at the minimum rate required by the PWM algorithm. Transferring the PWM to hardware using shared memory would potentially yield greater benefits, depending on the transfer rate, but did not work as intended when it was tried out during implementation. Making it work was therefore left as potential future work.

FIFO depths FIFO queues are implicitly created to pass computed results to the filter, when creating Impulse-C streams. FIFO depths in the implemented solutions are chosen based on the maximum DNA sequence length. This length is the same as the FIFO depth anticipated to be needed in order to avoid a communication bottleneck. Too large depths can result in a value overflow when computing the depth of one stream based on the depth of another, as the implemented solutions does in some cases at the moment. This limits the maximum sequence length possible to support, and should therefore be changed in the future.

7.2.2 Scope

- **DNA sequence length:** Maximum sequence length supported is currently 300, but only temporary during development, and should be increased in the future.

This length is only a fraction of the realistic sequence lengths mentioned in the FPWM specification from Drabløs [5], which are up to 300Mbases.

- **DNA sequence bases:** The solutions are case sensitive, so that all bases in the input DNA sequence must be in capital letters. Small letters will be ignored, and the index of the following valid bases are skewed.
- **Motif length:** Motif length is set to eight; this length is hard coded in all four solutions. All input matrices must therefore have a row count of eight, representing the selected motif length.
- **Matrix weights:** There are no observed limitations on matrix weights for the floating-point solutions. The floating-point solutions should be both more functional and safer than the 'fixed-point' solutions, due to the absence of overflow issues.
- **Multiple matrices:** The maximum number of matrices which can be processed in parallel in hardware is set to be two at the moment. More can easily be added later, but if more than two matrices are given in the matrix input file today, only the first two will be read and processed.
- **Threshold values:** No negative threshold values are allowed in software simulation, at least when given as command line arguments. This should not be an issue, as negative scores are not too interesting and matrices will most likely not have a negative threshold value anyway. In software simulation, the summation filter is executed instead of the threshold filter if a negative threshold value is given as argument.
- **'Fixed-point' summation filter:** A brute force method for avoiding potential sign issues connected to negative values prunes negative scores in the 'fixed-point' summation filter. The incorrect summation of the 'fixed-point' values are most likely due to overflow though, but as the majority of the observed scores during analysis were negative, pruning negative scores will prevent overflow in these cases. The possibility of overflow in a fixed-point solution is troublesome.
- **Simulation vs execution:** Simulation of the solutions is possible today, but not execution on the actual target platform Musculus.

7.2.3 Generated HDL/HW

Understanding the logic During analysis of the implemented solutions, it was of interest to gain an understanding of the generated HDL and resulting hardware logic. There were both factors making this task easy and factors making it difficult, but the latter were in majority. A lack of comments and documentation of the generated application specific HDL files was definitely one of the factors making it difficult. Structured, tidy, organized code helped somewhat however. To a degree, it was intuitive what Impulse-C code the individual HDL code blocks was a translation of. Details surrounding the translation of actual computation were more cryptic and far less intuitive, on the other hand. This, of course, was not helped by the limited time to study and

gain knowledge about the code. An analysis of the generated VHDL code and resulting hardware logic was presented in chapter 6.3.

Parallel behavior Which instructions of a hardware process that is actually performed in parallel is indicated by the hardware simulator/debugger StageMaster, as illustrated in figure 6.1 in chapter 6. There is a partially sequential execution of instructions in the hardware modules, though some instructions in the same block can be executed in parallel. These are usually instructions of same nature. Recursively used variables limits parallel computation in the pipelined loop as these are references to the same array.

Pipelining Pipelining is an optimization that reduces the number of cycles required to execute a loop by allowing the operations of one iteration to execute in parallel with operations of one or more subsequent iterations. In some cases it is not possible to perform all stages of a pipeline in parallel, such as when two stages read from the same memory (local variable). This was initially not believed to be an issue for the implemented FPWM solutions, though it turned out to be. Sometimes, the rate of a pipeline that contains multiple reads of the same array can be reduced by dividing the array into several smaller arrays. This could be done for the Impulse-C FPWM solutions.

Device utilization Device utilization for the implemented designs is reported in the Xilinx ISE 'Place and route' report, and summarized and repeated in table 7.1. In comparison, the existing VHDL solution utilize 8% of the FPGA. Floating-point is generally significantly more expensive, in terms of computation time and hardware logic required, than integer or fixed-point math. This is proved by comparing the device slice utilization for the floating-point and 'fixed-point' solutions presented in table 7.1. More detailed information on device utilization is presented in chapter 6.4. There are also other factors than value representations that can have an impact on device utilization. As the compiler will implicitly flatten control logic when the CO PIPELINE pragma is used, the depth of the generated logic may have been dramatically increased by pipelining the computational loop of the PWM module. Whether or not this is the case for the implemented FPWM solutions can not be proved at this time. The increase in logic when adding instances of the hardware modules PWM and Filter on the other hand, is indicated by comparing the device utilizations presented in table 7.1. The logic is not doubled, as only parts of the system needs to be duplicated. It can be worth pointing out how there is a larger increase in logic for the floating-point solution, as was expected beforehand. Looking at the reported utilization, some calculation can be made towards the number of parallel processing element (PE) pairs there could possibly be room for on the same FPGA. These numbers will of course be very approximate. For example, there is no guarantee that the increase in hardware logic is linear. Being realistic, it could also be the case that the entire FPGA can not be utilized, due to resource limitations such as routing capabilities.

Utilization without PEs + utilization PEs * X = 100%. Following this formula, 17 possible PWM-Filter pairs is calculated for the floating-point alternative, and 22 for

the 'fixed-point' alternative. This is under the incorrect assumption that the increase in logic is linear and the entire FPGA can be utilized. The real numbers are probably only half of the computed estimates.

	Floating-point	'Fixed-point'
Basic	16%	14%
Multicore	21%	18%

Table 7.1: Device utilization - slices

FPGA programming file Some theories as to why it has not been possible to generate a functional FPGA programming file was presented in chapter 6. A warning is reported by Xilinx ISE that the entity 'top' is duplicated. What is known is that signals passing over the RT are unsuccessfully connected to the top module, so this could be the cause. This is likely due to incorrect handling of the design specifications during synthesis and/or implementation of the design in Xilinx ISE; though it can not be proved or disproved at this point. If it can be proved in the future, finding a way to successfully open the template ISE project generated by CoDeveloper could solve the problem. The constraints in the .ucf-file is correct, and is also the standard constraints file for Cray XD1 projects developed in Impulse-C. The alternative solution is to study all warnings in Xilinx ISE and find the correct properties to change (in order to eliminate the critical warnings). If it on the other hand is not in Xilinx ISE it all goes wrong, it should be in CoBuilder. This is however harder to prove, as well as to straighten out, due to the limited insight into and control over the HDL build and export processes.

7.3 CoDeveloper ease-of-use

The level of abstraction for the development language and tools is different from the VHDL solution, but the target platform is the same. Considerations do need to be made when implementing in Impulse-C also, in order to avoid implementing functionality that requires hardware logic not supported by the target platform.

Pure software developers should be able to implement a FPWM system in Impulse-C, according to the co-design "principles". Impulse-C must therefore support a 'brute force' implementation of hardware processes, as has been done in the solutions presented in this thesis. Hardware consideration can not be a must, except for the need to stand clear of obvious violations of what the hardware platform support. Example of such considerations when implementing for Musculus is floating-point formats, dual-clocks, etc. Musculus, and the Cray XD1 platform in general, does not support the use of dual-clocks or single precision floating-point (double). The report from the HDL build process will give feedback on the success for the HDL build process. Notice will be given if the CoBuilder tools have observed functionality that clearly violates what is possible to implement on the selected hardware platform. The Impulse-C code can be easily optimized for performance later after a solution is initially developed. Optimizing Impulse-C code for performance can be done in a few easy steps, as described in the Impulse-C user guide.

The abstraction level of Impulse-C allows for the designer to create and use variables of different data-types to a large degree without having to worry about the hardware logic required and how it should be set up. FIFO queues are also implicitly created to pass computed results to the filter, when streams are created. This saves a designer a lot of work when implementing.

Problems with successfully generating functional programming files from the generated HDL has its effect on the experienced ease-of-use. CoDeveloper should integrate the functionality of Xilinx ISE, and similar tools for other platforms, allowing the designer to get more constructive and helpful feedback/confirmation on what exactly could be problematic at any time and in what part of the design process something fails. Having to go from the FPWM design from CoDeveloper to Musculus by way of a completely separate and independent synthesis/implementation tool gives reduced control over the process. If something fails along the way, it is a non trivial task to figure out the exact problem(s) and how to fix it/them. Also, it reduces the ease-of-use considerably.

7.4 Productivity

The VHDL solution was developed over a period of more than one 5'th year project, two masters theses, and a semester of continued work. The solution still does not work completely accurately after all this time, which is mostly due to the challenging task of implementing at HDL level.

When deciding to utilize the co-design approach to implementing a system, designers willingly sacrifice performance in order to gain more productivity. During the period of time available for working on this thesis, both a basic and a multicore FPWM solution were implemented using Impulse-C. In order to explore the effects floating point operations in hardware would have on performance, a fixed point version of both the basic solution and the multicore solution was also implemented and tested. Both variations of the multicore solution process multiple matrices in parallel in hardware. The multicore solutions was originally left as future work, as it was believed that there would be no time to finish implementing them during the time available to work on the thesis.

The process of implementing the basic Impulse-C solution started off slower than what was expected in advance of the thesis period, creating a fear that the rest of the implementation process would go just as slow, or even slower if implementing a more complex multicore solution. Needless to say, the lack of previous programming experience in Impulse-C, and co-design in general, have lead to lower productivity than if previous experience was present. There was a considerable need to acquire further knowledge about developing systems using Impulse-C, while at the same time being in the middle of an actual development process with a steadily approaching thesis deadline.

'Brute force' high level programming of the hardware partition in Impulse-C allows for high productivity, at the expense of potential performance. As previously mentioned, the Impulse-C code can be easily optimized for performance later, after a solution is initially developed. Optimizing Impulse-C code for performance can be done in a few easy steps, as described in the Impulse-C user guide.

As previously mentioned, floating-point is generally significantly more expensive than integer or fixed-point math. Converting floating-point applications to fixed-point

is an inherently non trivial and time-consuming process involving managing trade-offs in precision, range and performance. In hardware processes the size of the generated hardware also needs to be considered. Designers must convert each individual variable initially to a fixed-point format and keep track of that format as the variable is operated on. Fixed-point programs will therefore also be full of scaling operations. This is easier done with well tested floating-point solutions, which the implemented floating-point solutions are not yet. The process of converting the implemented floating-point solutions did therefore not fit into the time schedule of this thesis.

Lack of control over the HDL build process, and insight into the generated HDL, could easily affect productivity if something fails along the way. This has been the case when working on this thesis. For the implemented Impulse-C FPWM solutions, the template ISE project fails to open on the development environment machine. This has led to problems successfully generating a functional programming file. As there is limited amount of insight into what happens to the Impulse-C source after the 'Build HDL' button has been pressed, it is a rather time-consuming task to investigate what has gone wrong, and in what part of the design/implementation process. Was it implementation of the Impulse-C source code, generation of HDL from the Impulse-C source code, export of the HDL code, synthesis of the design, or implementation of the design that was faulty? Is it the designer, the CoBuilder tools, Xilinx ISE, or the OS that is the culprit? It could be any combination of these. When finally sorting it out, the problem also has to be fixed in an appropriate manner; which is not necessarily a trivial task either. The level of abstraction for the development language and design tools is different from the VHDL solution, but the target platform is the same, as well as the synthesis/implementation tools. Some productivity issues can not be avoided.

7.5 Performance

Despite limited possibility to compare performance of the implemented Impulse-C solutions and the existing VHDL solution, it has been assumed that the basic Impulse-C solution have lower performance than the VHDL solution. As a rule, designers of co-design solutions have to sacrifice performance in order to increase the productivity, this due to the abstraction level of the Impulse-C and other co-design languages.

Even though performance data from actual execution of the solutions on Musculus is missing, as the programming files are not functional, various design statistics are reported by the static timing summary in Xilinx ISE. Table 7.2 present timing information reported by Xilinx ISE for the implemented solutions. Due to the generated programming files being faulty, this data should be viewed as approximate values only, giving at least a general impression as to what the difference in computation time is between the different solutions.

	Minimum period	Maximum frequency
FPWM	12.436ns	80.412MHz
FPWM2008	13.954ns	71.664MHz
FPWMi	10.084ns	99.167MHz
FPWM2008i	11.490ns	87.032MHz

Table 7.2: Timing statistics

Floating-point offers a greater range of values and more precision, but is also significantly more expensive in terms of computation time than integer or fixed-point math, as illustrated in table 7.2. Embedded systems and digital signal processing (DSP) designers often choose fixed-point in order to achieve greater speed and reduce hardware costs in their designs. The existing VHDL solution operate with fixed-point in hardware.

Parallel computation is extracted automatically from the implemented code by the hardware compiler where it sees the possibility for it, while the compiler will attempt to extract pipelined computation from loops the designer have marked with the Impulse-C pragma `CO PIPELINE`. Both extracted the pipelined and parallel computation should have a positive effect on performance for the Impulse-C solutions. Processing multiple matrices in parallel should also have a significantly better performance than processing them serial (with the same PE), as illustrated in 7.2.

The thesis report for the existing VHDL solution refer to the maximum clock frequency for the Xilinx Virtex-II Pro as the realistic best case scenario. This makes it somewhat harder to compare performance of the Impulse-C solutions with that of the VHDL solution, as it is doubtful that the VHDL solution really can have a maximum frequency equal to the FPGA clock frequency of 200MHz. It can seem a bit too optimistic even as a best case? As the VHDL solution sacrifice productivity for performance, while the opposite is the case for the Impulse-C solutions, it is still a fair assumption to make that the VHDL solution have a better performance than the Impulse-C solutions.

7.6 Unanswered questions

There is a limited possibility to compare actual performance of the implemented Impulse-C solutions and the existing VHDL solution. The VHDL solution does not work entirely as intended, first of all, and there is not much performance data available. There is not much performance data available for the Impulse-C solutions either, as they have not yet been successfully tested on the target platform.

Chapter 8

Conclusion and Future Work

This chapter will present a conclusion for the thesis, as well as some thoughts on future work.

8.1 Conclusion

As it says in the thesis objectives, an FPGA prototype specified in VHDL has been developed at NTNU, which identifies short motifs or patterns in genetic data using Position-Weight Matrices.

This thesis present work done on the following major tasks:

- Specification and implementation of a Impulse-C based alternative to the existing VHDL-based solution.
- Evaluation of easy-of-use of the CoDeveloper environment, and productivity vs. final performance when comparing the Impulse-C solution and the existing VHDL-based solution.

In total, four variations of a Impulse-C alternative have been implemented; a basic solution and a multicore solution, both implemented in a floating-point and a 'fixed-point' version. These solutions have all been successfully software simulated. The floating-point solutions have also been tested and analyzed. Tests and analysis done during software simulation show that the implemented floating-point solutions function correctly for the tested sequence length. The VHDL code generated by CoBuilder has also been slightly analyzed.

Unfortunately, attempts made to get the solutions to run on the target platform Musculus were all unsuccessful. Some information about device utilization and performance can still be extracted from the Xilinx ISE 'Static timing' and 'Place and route' reports. Creating a functional FPGA programming file for the most functional of the implemented solutions should never the less be priority number one as far as future work goes.

Designers of co-design solutions must sacrifice performance in order to increase the productivity, this due to the abstraction level of co-design languages such as Impulse-C. This has been demonstrated by this thesis. Co-design would clearly ease further development of the FPWM in the future. The thesis has also demonstrated how device

utilization is also affected by the use of co-design and giving preference to productivity. The effects of floating-point arithmetic's in hardware on both performance and device utilization have also been observed and documented in the thesis. Floating-point is generally significantly more expensive, in terms of computation time and hardware logic required, than integer or fixed-point math.

Finally, it has been shown how some productivity issues can not be avoided even with the use of co-design. The level of abstraction for the development language and design tools is different from implementing a VHDL solution, but the target platform is the same, as well as the required synthesis/implementation tools.

The implemented Impulse-C solutions have lower performance than the previously implemented VHDL solution, as well as higher FPGA utilization. However, productivity when implementing in Impulse-C is significantly higher than when implementing in VHDL.

8.1.1 Project value

The use of hardware descriptive languages to program FPGAs, such as VHDL, is a complicated and time consuming process that requires intimate knowledge of how hardware works. Consequently, it is truly beneficial for productivity to make use of co-design languages that facilitate the use of hardware, as well as to integrate the environment for development of soft- and hardware.

There is a lack of Impulse-C competence at NTNU, as there is on a general basis no considerably widespread use of Impulse-C as of yet. Although co-simulation languages such as SystemC is already in use, the use of Impulse-C at NTNU could have both great educational and significant research value also.

8.2 Future work

This section include specific recommendations for future work. These recommendations are presented in a suggested order of priority.

8.2.1 Choosing I/O files

Giving the user more control over the I/O process would improve both ease-of-use and efficiency. One of the important changes to make to the system in the future is therefore to implement the possibility for the user to choose both the input files to read data from and the output file to write data to. As the system is implemented today, the name of all the I/O files are specified in the application source code. Changing this, to letting the user specify which I/O files to use as command line arguments, would reduce the amount of time needed between each execution to prepare the next input and take backup of the previous output.

8.2.2 Securing written results

An alternative method for securing previous results written to the output file, instead of writing the new results to a different output file, is to be more precise about how the

software should handle the existing content of the output file. The way the software handle existing content of the output file today is to simply ignore it; effectively erasing it from the file. By telling the software to append new content to the existing content, if there is any, all results should be secured for later review.

Whether or not to append new results to the existing content, instead of writing over the existing content, could also be a decision left for the user of the application to make.

8.2.3 Multiple matrices and sequences

An interesting addition to make to the system in the future could be to add the possibility to process multiple DNA-sequences during a single execution of the FPWM application, in addition to processing multiple matrices. All DNA-sequences would be stored in the same input file, just as all matrices are today.

The currently implemented solution, only allowing one sequence to be read and processed at a time, has an input scheme that is prepared for being expanded to handle multiple DNA-sequences. By already requiring all input to be expressed in Fasta-format, the transition should go smoother than it would have otherwise. In the Fasta-format, the header line of a matrix or sequence starts of with a '>' symbol to indicate the start of a new input element.

Even though there is a multicore implementation of the Impulse-C solution today, it should be extended to process more than two matrices in parallel in the future. The use of multiple matrices also introduce the need for a module that can pre-process all the input alignment matrices, making them ready for later use, by converting them to PWMs. This is a more vital change to make to the system, taking performance into consideration, especially if the system should process more than two matrices in parallel.

Shared memory and registers

Allowing multiple matrices and/or sequences to be read and processed by the system, as suggested in this subsection, could increase the need for implementing the use of shared memory to transfer data between software and hardware.

8.2.4 Explicit parallel processing

Explicitly computing results in parallel could be done more effectively if other methods of passing data between software and hardware were utilized. Shared memory is a good example of such a method. Parallel processing using shared memory to transfer data could potentially increase the performance of the system. It could also allow for more matrices to be processed.

A parallel implementation of the software framework, reading input and writing output, could also significantly increase performance.

8.2.5 True fixed-point solution

Fixed-point is an alternative to floating-point, a more well-known method of representing real numbers. Impulse-C provides support for fixed point arithmetic in the form of

macros and data-types that allow you to express fixed point operations in ANSI C and perform computations either as software on an embedded CPU or as hardware modules running in an FPGAs logic. There is however, as yet, no commonly-accepted standard for representing fixed-point numbers.

Fixed-point applications are often created from a well-tested floating-point implementation, rather than written from scratch. The process of converting a floating-point application to fixed-point is a non-trivial effort, with many issues to consider. Precision and range of the variables as it is run with sample data must be tracked in order to determine the variables' fixed-point formats. Designers must convert each variable initially to a fixed-point format and keep track of that format as the variable is operated on. Fixed-point programs will be full of scaling operations to align decimal points, prevent overflow, and manage precision. It is important to be able to characterize the range and precision of input, intermediate, and output variables throughout a fixed point program.

8.2.6 Ignoring regions of DNA

PWM scores at a positions including a non-standard symbol could either be ignored or discarded. In some cases lower case symbols are used to indicate positions in the input string that should be ignored/discarded during PWM scoring ("repeat-masked sequences"). This is however optional, but could be an interesting feature to implement in the future. Implementing the feature should not take too much time or effort.

8.2.7 Utilizing a database connection

An alternative input source to reading input data from file is to implement a database connection so that the DNA-sequence and/or alignment matrix can be fetched from an external database. The database can either replace the input files completely as an input source, or be used in combination with the input files. The database connection could also be used as an alternative to the output file for storing filtered results.

Using a database connection to handle I/O in the software framework could be a more effective method than reading from and writing to files, and result in reduced execution time for the system. It would also make stored results more easily accessible for later review or processing, than when stored in a file.

The importance a database connection would have for the system is somewhat debatable, but should in any case be easy to implement.

8.2.8 Web interface

A method of making the FPWM system more easily accessible to the user is to connect the system to a web interface. A seemingly functional prototype for such an interface has been implemented prior to this thesis. The time and effort needed to connect the Impulse-C solution to this interface, and possibly extending the functionality somewhat if found necessary, is strongly dependent on the final functionality of the interface.

Part VI

Appendices

Appendix A

Nomenclature

Abbreviations

- AAP : Application Acceleration Processor
- API : Application Program Interface
- CLB : Configurable Logic Block
- CPU : Central Processing Unit
- CSP : Communicating Sequential Processes
- DMF : The Medical Faculty
- FPGA: Field Programmable Gate Array
- HLL : High Level Language
- HDL : Hardware Description Language
- HPC : High Performance Computing
- IDI : Department of Computer and Information Science
- MPI : Message Passing Interface
- NTNU : Norwegian University of Science and Technology
- PWM : Position Weight Matrix
- RAP : RapidArray Processor
- RTL : Register Transfer Level
- SMP : Symmetrical Multi-Processing
- TF : Transcription Factor
- VHDL : VLSIC Hardware Description Language
- VLSIC : Very Large-Scale Integrated Circuit

Appendix B

Source Code

FPWM

FPWM.h

```
//////////////////////////////////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
#define MAX_STREAMWIDTH 64 /* buffer width for FIFO in hardware */
#define MIN_STREAMDEPTH 1 /* minimum buffer size for FIFO in hardware */

#define DNA_INPUT_FILE "dna.txt"
#define PWM_INPUT_FILE "pwm.txt"
#define OUTPUT_FILE "out.txt"

#define COLUMNS 4 /* A,C,G,T */
#define ROWS 8 /* length of pattern/motif */

#define MAX_SEQUENCE 300

#define FILTER_QUEUE MAX_SEQUENCE /* FIFO length between PWM and Filter */
#define WRITE_QUEUE ((MAX_SEQUENCE*10)/100) /* FIFO length between Filter and Consumer */

#define VALID_CHAR(a) (((a) > 64) && ((a) < 91) && ((a) != 74) && ((a) != 79) && ((a) != 85)
? (1) : (0))
```

FPWM_sw.c

```
//////////////////////////////////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
// FPWM_sw.c: includes the software test bench processes and
// main() function for the basic floating-point version of
// the FPWM system.
//
// See additional comments in FPWM.h.
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "co.h"
#include "cosim_log.h"
#include "FPWM.h"

extern co_architecture co_initialize(void *);

// Globals
static char *pwmHeader, *dnaHeader;
static float *pwm;
static char *dnaSequence;
static int pwmSize, dnaLength;
static int filterMode;
```

```

//
// This is the function for calculating pwm-values from count-values
//
float convert_value(float c, float N) {
double s = 0.25;
double P = 0.25;
double p, valueTemp;
float pwmValueTemp;

p = (double) ((c+s)/(N+(4*s)));
valueTemp = (double) (p/P);
pwmValueTemp = (float) log(valueTemp);

return pwmValueTemp;
}

//
// This is the function for converting an alignment matrix to pwm
//
float *convert_matrix(float *matrix) {
int i, j;
float *pwmTemp;
double N;

pwmTemp = (float *) malloc(ROWS*COLUMNS*sizeof(float));
for (j = 0; j < COLUMNS; j++) {
N = (matrix[j*COLUMNS+0]+matrix[j*COLUMNS+1]+matrix[j*COLUMNS+2]+matrix[j*COLUMNS+3]);
for (i = 0; i < ROWS; i++) {
pwmTemp[j*COLUMNS+i] = convert_value(matrix[j*COLUMNS+i], N);
}
}
return pwmTemp;
}

//
// This is the function for reading an input matrix from a file and converting it to a pwm,
// based on a function from the Impulse CoDeveloper example project 'SmithWatermanSerial'
//
void read_matrix(const char *PwmFileName, FILE *pwmInFile) {
char *buffer = (char *) malloc(32*sizeof(char));
char *header;
int status = 0;
int size = 32; /* size of header */
int matrixSize = COLUMNS*ROWS;
float *matrix;
int i, j;
char character;

// Opening matrix input file
pwmInFile = fopen(PwmFileName, "r");
if (pwmInFile == NULL) {
fprintf(stderr, "Error opening matrix input file %s\n", PwmFileName);
character = getc(stdin);
exit(-1);
}

// Finding the size of the header line
status = fread(buffer, sizeof(char), 1, pwmInFile);

// Making sure the matrix is in the correct input format
if(buffer[0] != '>') {
printf("Matrix in %s must be in FASTA format, starting with '>'\n", PwmFileName);
}

// Closing matrix input file
if(fclose(pwmInFile) != 0) {
printf("Error closing matrix input file\n");
}

// Allocating memory to store header line
header = (char *) malloc(size*sizeof(char));
matrix = (float *) malloc(matrixSize*sizeof(float));

// Reopening matrix input file
pwmInFile = fopen(PwmFileName, "r");
if (pwmInFile == NULL) {
printf("Error opening matrix input file %s file\n", PwmFileName);
exit(-1);
}

// Pruning '>'
fread(buffer, sizeof(char), 1, pwmInFile);

// Reading matrix name
status = fscanf(pwmInFile, "%s", header);
if(status <= 0 || status > size){

```

```

    printf("Error reading %s file header\n", PwmFileName);
    exit(-1);
}

// Reading and storing the matrix
printf("\nReading matrix %s: ", header);
for(j = 0; j < ROWS; j++) {
    for(i = 0; i < COLUMNS; i++) {
        fscanf(pwmInFile, "%f", &matrix[j*COLUMNS+i]);
    }
}
printf("done\n");

// Closing matrix input file
if(fclose(pwmInFile) != 0) {
    printf("Error closing matrix input file\n");
}

// Saving pwm data
pwmHeader = header;
printf("Converting %s: ", header);
pwm = convert_matrix(matrix);
printf("done\n");
pwmSize = matrixSize;

free(matrix);
free(buffer);
}

//
// This is the function for reading an input sequence from a file, based on a function
// from the Impulse CoDeveloper example project 'SmithWatermanSerial'
//
void read_sequence(const char *DnaFileName, FILE *dnaInFile) {
    char *buffer = (char*)malloc(32*sizeof(char));
    char *sequence, *header;
    int sequenceLength;
    int size = 32;
    int status = 0;
    int nonvalid;
    char character;

    // Opening sequence input file
    dnaInFile = fopen(DnaFileName, "r");
    if ( dnaInFile == NULL ) {
        fprintf(stderr, "Error opening sequence input file %s\n", DnaFileName);
        character = getc(stdin);
        exit(-1);
    }

    // Finding the size of the header line
    status = fread(buffer, sizeof(char), 1, dnaInFile);

    // Making sure the sequence is in the correct input format
    if(buffer[0] != '>') {
        printf("Sequence in %s must be in FASTA format, starting with '>'\n", DnaFileName);
    }

    // Trimming off non-valid characters before sequence
    buffer[0] = '\n';
    while(!(VALID_CHAR((int)buffer[0]))) {
        fread(buffer, sizeof(char), 1, dnaInFile);
    }

    // Finding the size of the sequence, after pruning the header line
    fscanf(dnaInFile, "%s", buffer);
    sequenceLength = 0;
    nonvalid = 0;
    while(fread(buffer, sizeof(char), 1, dnaInFile)) {
        if(VALID_CHAR((int)buffer[0]))
            sequenceLength++;
        if (!(VALID_CHAR((int)buffer[0])))
            nonvalid++;
    }

    // Closing sequence input file
    if(fclose(dnaInFile) != 0) {
        printf("Error closing sequence input file\n");
    }

    // Allocating memory to store the sequence and header line
    header = (char*)malloc(size*sizeof(char));
    sequence = (char*)malloc((sequenceLength + 1)*sizeof(char));

    // Reopening sequence input file
    dnaInFile = fopen(DnaFileName, "r");
    if( dnaInFile == NULL ){

```

```

    printf("Error opening sequence input file %s file\n", DnaFileName);
    exit(-1);
}

// Pruning '>'
fread(buffer, sizeof(char), 1, dnaInFile);

// Reading sequence name
status = fscanf(dnaInFile, "%s", header);
if(status <= 0 || status > size){
    printf("Error reading %s file header\n", DnaFileName);
    exit(-1);
}

size = 1; /* Why shouldn't this value be 0? Would cause an error */
// Reading the sequence
printf("\nReading sequence %s: ", header);
while (fread(buffer, sizeof(char), 1, dnaInFile)) {
    if (VALID_CHAR((int)buffer[0])) {
        sequence[size++] = buffer[0];
    }
}
printf("done\n");
if(size-1 != sequenceLength){
    printf("Error reading %s file sequence\n", DnaFileName);
    exit(-1);
}

// Closing sequence input file
if (fclose(dnaInFile) != 0) {
    printf("Error closing sequence input file\n");
}

// Saving sequence data
dnaHeader = header;
dnaSequence = sequence;
dnaLength = sequenceLength;

free(buffer);
}

//
// This is the software 'reader' process
//
void Producer(co_stream threshold_stream, co_stream pwm_stream, co_stream iteration_stream,
             co_stream dna_stream, co_parameter filter)
{
    float threshold;
    IF_SIM(cosim_logwindow_log = cosim_logwindow_create("Producer"));

    // Opening streams
    co_stream_open(threshold_stream, O_WRONLY, FLOAT_TYPE);
    co_stream_open(pwm_stream, O_WRONLY, FLOAT_TYPE);
    co_stream_open(dna_stream, O_WRONLY, CHAR_TYPE);
    co_stream_open(iteration_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));

    // Determining type of filter based on command line argument
    if (filter == NULL) {
        // Filter is set to summation filter
        filterMode = 1;
    } else {
        // Filter is set to threshold filter
        filterMode = 2;
        threshold = atof(filter);
        co_stream_write(threshold_stream, &threshold, sizeof(float));
    }

    // Reading matrix from input file and converting it to pwm
    const char * PwmFileName = PWM_INPUT_FILE;
    FILE * pwmInFile;
    read_matrix(PwmFileName, pwmInFile);

    // Sending matrix
    int i;
    float nSamplePwm;
    printf("Sending %s: ", pwmHeader);
    for(i = 0; i < pwmSize; i++) {
        nSamplePwm = (float)pwm[i];
        co_stream_write(pwm_stream, &nSamplePwm, sizeof(float));
    }
    printf("done\n");

    free(pwm);

    // Reading sequence from input file
    const char * DnaFileName = DNA_INPUT_FILE;
    FILE * dnaInFile;

```



```

read_sequence(DnaFileName, dnaInFile);

// Sending sequence length and actual sequence
int k;
char nSampleDna;
co_stream_write(iteration_stream, &dnaLength, sizeof(int16));
printf("Sending %s...\n\n", dnaHeader);
for(k = 1; k <= dnaLength; k++) {
    nSampleDna = (char)dnaSequence[k];
    co_stream_write(dna_stream, &nSampleDna, sizeof(char));
}
printf("done sending %s\n", dnaHeader);

free(dnaSequence);

// Closing streams
co_stream_close(threshold_stream);
co_stream_close(pwm_stream);
co_stream_close(iteration_stream);
co_stream_close(dna_stream);
}

//
// This is the software 'writer' process
//
void Consumer(co_stream start_stream, co_stream score_stream)
{
    int16 nResultStart;
    int resultEnd;
    float nResultScore;
    unsigned int count = 0;
    const char * FileName = OUTPUT_FILE;
    FILE * outFile;

    IF_SIM(cosim_logwindow_log = cosim_logwindow_create("Consumer");)

    // Opening output file
    outFile = fopen(FileName, "w");
    if ( outFile == NULL ) {
        fprintf(stderr, "Error opening file %s for writing\n", FileName);
        exit(-1);
    }

    // Opening streams
    co_stream_open(start_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(score_stream, O_RDONLY, FLOAT_TYPE);

    IF_SIM(cosim_logwindow_write(log, "Consumer reading results...\n");)

    // Reading filtered results from stream; then writing them to screen and file
    while (co_stream_read(start_stream, &nResultStart, sizeof(int16)) == co_err_none) {
        if (co_stream_read(score_stream, &nResultScore, sizeof(float)) == co_err_none) {
            if (filterMode == 1) { // Summation filter
                resultEnd = dnaLength - 1;
            } else { // Threshold filter
                resultEnd = nResultStart + (ROWS-1);
            }
            fprintf(outFile, "%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmHeader,
                nResultScore);
            IF_SIM(cosim_logwindow_fwrite(log, "Result: %s %d %d %s %f\n", dnaHeader, nResultStart,
                resultEnd, pwmHeader, nResultScore);)
            printf("%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmHeader, nResultScore)
                ;
            count++;
        }
    }
    IF_SIM(cosim_logwindow_fwrite(log, "Consumer read %d filtered results\n", count);)
    printf("\n\nThe application produced %d filtered scores for %s\n", count, dnaHeader);

    free(dnaHeader);
    free(pwmHeader);

    // Closing output file
    if (fclose(outFile) != 0) {
        printf("Error closing result output file\n");
    }

    // Closing streams
    co_stream_close(start_stream);
    co_stream_close(score_stream);
}

//
// Impulse C Main Function
//
int main(int argc, char **argv)

```

```

{
  co_architecture my_arch;
  void *param = NULL;
  char *filter;
  char **arg;
  int c;

  printf("\n\n== IMPULSE-C APPLICATION: FPWM ==\n");
  switch(argc) {
    case 1:
      printf("Executing with summation filter\n");
      my_arch = co_initialize(param);
      co_execute(my_arch);
      break;
    case 2:
      printf("Executing with threshold filter; ");
      arg = (char**)argv;
      filter = (char*)arg[1];
      printf("threshold set at %s\n", filter);
      my_arch = co_initialize(filter);
      co_execute(my_arch);
      break;
    default:
      printf("\nWrong use of parameters!\n");
      break;
  }

  printf("Application complete! Press the Enter key to continue...\n");
  c = getc(stdin);

  return(0);
}

```

FPWM_hw.c

```

/////////////////////////////////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
// FPWM_hw.c: includes the hardware processes and configuration
// function for the basic floating-point version of the FPWM
// system.
//
// See additional comments in FPWM.h.
//
#include "co.h"
#include "cosim_log.h"
#include "FPWM.h"
#include "co_math.h"

// Software process declarations (see FPWM_sw.c)
extern void Producer(co_stream threshold_stream, co_stream pwm_stream, co_stream
  iteration_stream, co_stream dna_stream,
  co_parameter filter);
extern void Consumer(co_stream start_stream, co_stream score_stream);

//
// This is the hardware 'pwm' process
//
void PWM(co_stream pwm_stream, co_stream iteration_stream, co_stream dna_stream, co_stream
  counter_stream, co_stream sum_stream)
{
  int i, j, k, l;
  float nSamplePwm;
  float pwm[COLUMNS][ROWS];
  int16 dnaLength;
  int motifLength;
  char nSampleDna;
  char sequence[MAX_SEQUENCE];
  int16 counter;
  float sum;
  int16 nHitStart;
  float nHitScore;

  IF_SIM(int samplesread; int resultswritten;)

  IF_SIM(cosim_logwindow log;)
  IF_SIM(log = cosim_logwindow_create("PWM");)

  do { // Hardware processes run forever
    IF_SIM(samplesread=0; resultswritten=0;)

    // Stating motif length
    motifLength = ROWS;

```

```

// Opening streams
co_stream_open(pwm_stream, O_RDONLY, FLOAT_TYPE);
co_stream_open(iteration_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
co_stream_open(dna_stream, O_RDONLY, CHAR_TYPE);
co_stream_open(counter_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
co_stream_open(sum_stream, O_WRONLY, FLOAT_TYPE);

// Reading pwm from stream
for(j=0; j<ROWS; j++) {
    for(i=0; i<COLUMNS; i++) {
        co_stream_read(pwm_stream, &nSamplePwm, sizeof(float));
        pwm[i][j] = (float)nSamplePwm;
    }
}

// Reading sequence length from stream
if(co_stream_read(iteration_stream, &dnaLength, sizeof(int16)) == co_err_none) {
    IF_SIM(cosim_logwindow_fwrite(log, "Sequence length is %d\n", dnaLength);)
}

// Reading portion of sequence from stream necessary to start computation
for(k = 0; k < motifLength; k++) {
    co_stream_read(dna_stream, &nSampleDna, sizeof(char));
    IF_SIM(samplesread++);
    sequence[k] = (char)nSampleDna;
}

// Initiating counter and stream
counter = 0;
sum = 0;

// Computing result scores for all subsequences
while(counter <= (dnaLength - motifLength)) {

    // Calculating score for current position
    for(l = 0; l < motifLength; l++) {
        #pragma CO PIPELINE
        #pragma CO SET stageDelay 32
        switch(sequence[l+counter]) {
            case((char)'A'):
                sum += pwm[0][l];
                break;
            case((char)'C'):
                sum += pwm[1][l];
                break;
            case((char)'G'):
                sum += pwm[2][l];
                break;
            case((char)'T'):
                sum += pwm[3][l];
                break;
            default:
                // Do nothing here
                break;
        }
    }

    // Stating that counter and sum will be result data
    nHitStart = counter;
    nHitScore = sum;

    // Sending result to filter
    co_stream_write(counter_stream, &nHitStart, sizeof(int16));
    co_stream_write(sum_stream, &nHitScore, sizeof(float));

    IF_SIM(resultswritten++);
    IF_SIM(cosim_logwindow_fwrite(log, "Wrote score %f to filter, for pattern starting at
        position %d.\n", nHitScore, nHitStart);)

    // Reading new sequence base from stream
    if(co_stream_read(dna_stream, &nSampleDna, sizeof(char)) == co_err_none) {
        IF_SIM(samplesread++);
        sequence[k] = (char)nSampleDna;
        k++;
    } else {
        break;
    }

    // Updating counter and resetting sum
    counter++;
    sum = 0;
}

// Closing streams
co_stream_close(pwm_stream);
co_stream_close(iteration_stream);
co_stream_close(dna_stream);
co_stream_close(counter_stream);

```

```

co_stream_close(sum_stream);

IF_SIM(cosim_logwindow_fwrite(log, "Closing PWM process; Symbols in sequence read: %d,
    results written: %d\n", samplesread, resultswritten);)
IF_SIM(break;) // Only run once for desktop simulation
} while(1);
}

//
// This is the hardware 'filter' process
//
void Filter(co_stream threshold_stream, co_stream counter_stream, co_stream sum_stream,
    co_stream start_stream, co_stream score_stream)
{
    float threshold;
    int filterMode;
    int16 nResultStart;
    float nResultScore;
    int16 nHitStart;
    float nHitScore;
    IF_SIM(int resultsread; int resultswritten;)

    IF_SIM(cosim_logwindow_log;)
    IF_SIM(log = cosim_logwindow_create("Filter");)

    do { // Hardware processes run forever
        IF_SIM(resultsread=0; resultswritten=0;)

        // Opening streams
        co_stream_open(threshold_stream, O_RDONLY, FLOAT_TYPE);
        co_stream_open(counter_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
        co_stream_open(sum_stream, O_RDONLY, FLOAT_TYPE);
        co_stream_open(start_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
        co_stream_open(score_stream, O_WRONLY, FLOAT_TYPE);

        // Reading threshold value from stream; determining filter mode
        if(co_stream_read(threshold_stream, &threshold, sizeof(float)) != co_err_none) {
            // Filter is set to summation filter
            filterMode = 1;
            IF_SIM(cosim_logwindow_fwrite(log, "Filter Mode: %d\n", filterMode);)
        } else {
            // Filter is set to threshold filter
            filterMode = 2;
            IF_SIM(cosim_logwindow_fwrite(log, "Filter Mode: %d\n", filterMode);)
            IF_SIM(cosim_logwindow_fwrite(log, "Threshold value: %f\n", threshold);)
        }

        // Initiating filtered result data
        nResultStart = 0;
        nResultScore = 0;

        // Reading computed results from stream
        while(co_stream_read(counter_stream, &nHitStart, sizeof(int16)) == co_err_none) {
            if(co_stream_read(sum_stream, &nHitScore, sizeof(float)) == co_err_none) {
                IF_SIM(resultsread++;)

                if(filterMode == 1) { // Summation filter
                    nResultScore += nHitScore;
                } else { // Threshold filter
                    if(nHitScore >= threshold) {
                        // Stating that the result score will pass through the filter
                        nResultStart = nHitStart;
                        nResultScore = nHitScore;

                        // Sending filtered result to consumer
                        co_stream_write(start_stream, &nResultStart, sizeof(int16));
                        co_stream_write(score_stream, &nResultScore, sizeof(float));

                        IF_SIM(resultswritten++;)
                        IF_SIM(cosim_logwindow_fwrite(log, "Filtered score %f for pattern starting at
                            position %d.\n", nResultScore, nResultStart);)
                    }
                }
            }
        }

        if(filterMode == 1) { // Summation filter
            // Sending result score from summation filter to consumer
            co_stream_write(start_stream, &nResultStart, sizeof(int16));
            co_stream_write(score_stream, &nResultScore, sizeof(float));

            IF_SIM(resultswritten++;)
            IF_SIM(cosim_logwindow_fwrite(log, "Filtered combined score %f for the entire sequence
                \n", nResultScore);)
        }

        // Closing streams
        co_stream_close(threshold_stream);
    }
}

```

```

    co_stream_close(counter_stream);
    co_stream_close(sum_stream);
    co_stream_close(start_stream);
    co_stream_close(score_stream);

    IF_SIM(cosim_logwindow_fwrite(log,
        "Closing Filter process; Results read: %d, results filtered: %d\n", resultsread,
        resultswritten);)

    IF_SIM(break;) // Only run once for desktop simulation
} while(1);
}

//
// Impulse C configuration function
//
void config_FPWM(void *arg)
{
    co_stream threshold_stream;
    co_stream pwm_stream;
    co_stream iteration_stream;
    co_stream dna_stream;
    co_stream counter_stream;
    co_stream sum_stream;
    co_stream start_stream;
    co_stream score_stream;

    co_process producer_process;
    co_process pwm_process;
    co_process filter_process;
    co_process consumer_process;
    IF_SIM(cosim_logwindow_init());

    char *parameter;

    if (arg != NULL) {
        parameter = (char*) arg;
    } else {
        parameter = NULL;
    }

    threshold_stream = co_stream_create("threshold_stream", FLOAT_TYPE, MIN_STREAMDEPTH);
    pwm_stream = co_stream_create("pwm_stream", FLOAT_TYPE, COLUMNS*ROWS);
    iteration_stream = co_stream_create("iteration_stream", INT_TYPE(MAX_STREAMWIDTH/4),
        MIN_STREAMDEPTH);
    dna_stream = co_stream_create("dna_stream", CHAR_TYPE, MAX_SEQUENCE);
    counter_stream = co_stream_create("counter_stream", INT_TYPE(MAX_STREAMWIDTH/4),
        FILTER_QUEUE);
    sum_stream = co_stream_create("sum_stream", FLOAT_TYPE, FILTER_QUEUE);
    start_stream = co_stream_create("start_stream", INT_TYPE(MAX_STREAMWIDTH/4), WRITE_QUEUE);
    score_stream = co_stream_create("score_stream", FLOAT_TYPE, WRITE_QUEUE);

    producer_process = co_process_create("Producer", (co_function)Producer,
        5,
        threshold_stream,
        pwm_stream,
        iteration_stream,
        dna_stream,
        parameter);

    pwm_process = co_process_create("PWM", (co_function)PWM,
        5,
        pwm_stream,
        iteration_stream,
        dna_stream,
        counter_stream,
        sum_stream);

    filter_process = co_process_create("Filter", (co_function)Filter,
        5,
        threshold_stream,
        counter_stream,
        sum_stream,
        start_stream,
        score_stream);

    consumer_process = co_process_create("Consumer", (co_function)Consumer,
        2,
        start_stream,
        score_stream);

    co_process_config(pwm_process, co_loc, "PE0");
    co_process_config(filter_process, co_loc, "PE0");
}

co_architecture co_initialize(int param)
{

```

```
    return(co_architecture_create("FPWM", "generic_vhdl", config_FPWM, (void *)param));  
}
```

FPWM2008

FPWM2008.h

```

////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
#define MAX_STREAMWIDTH 64 /* buffer width for FIFO in hardware */
#define MIN_STREAMDEPTH 1 /* minimum buffer size for FIFO in hardware */

#define DNA_INPUT_FILE "dna.txt"
#define PWM_INPUT_FILE "pwm.txt"
#define OUTPUT_FILE "out.txt"

#define COLUMNS 4 /* A,C,G,T */
#define ROWS 8 /* length of pattern/motif */

#define MAX_SEQUENCE 300

#define FILTER_QUEUE MAX_SEQUENCE /* FIFO length between PWM and Filter */
#define WRITE_QUEUE ((MAX_SEQUENCE*10)/100) /* FIFO length between Filter and Consumer */

#define VALID_CHAR(a) (((a) > 64) && ((a) < 91) && ((a) != 74) && ((a) != 79) && ((a) != 85)
? (1) : (0)

```

FPWM2008_sw.c

```

////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
// FPWM2008_sw.c: includes the software test bench processes and
// main() function for the parallel floating-point version of
// the FPWM system.
//
// See additional comments in FPWM2008.h.
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "co.h"
#include "cosim_log.h"
#include "FPWM2008.h"

extern co_architecture co_initialize(void *);

// Globals
static char *pwmHeader, *dnaHeader;
static float *pwm;
static char *dnaSequence;
static int pwmSize, dnaLength;
static int filterMode;
static char* pwmList[2];

//
// This is the function for reading an input sequence from a file, based on a function
// from the Impulse CoDeveloper example project 'SmithWatermanSerial'
//
void read_sequence(const char *DnaFileName, FILE *dnaInFile) {
    char *buffer = (char*) malloc(32*sizeof(char));
    char *sequence, *header;
    int sequenceLength;
    int size = 32;
    int status = 0;
    int nonvalid;
    char character;

    // Opening sequence input file
    dnaInFile = fopen(DnaFileName, "r");
    if ( dnaInFile == NULL ) {
        fprintf(stderr, "Error opening sequence input file %s\n", DnaFileName);
        character = getc(stdin);
        exit(-1);
    }

    // Finding the size of the header line
    status = fread(buffer, sizeof(char), 1, dnaInFile);

    // Making sure the sequence is in the correct input format

```

```

if (buffer[0] != '>') {
    printf("Sequence in %s must be in FASTA format, starting with '>'\n", DnaFileName);
}

// Trimming off non-valid characters before sequence
buffer[0] = '\n';
while (!(VALID_CHAR((int)buffer[0]))) {
    fread(buffer, sizeof(char), 1, dnaInFile);
}

// Finding the size of the sequence, after pruning the header line
fscanf(dnaInFile, "%s", buffer);
sequenceLength = 0;
nonvalid = 0;
while (fread(buffer, sizeof(char), 1, dnaInFile)) {
    if (VALID_CHAR((int)buffer[0]))
        sequenceLength++;
    if (!(VALID_CHAR((int)buffer[0])))
        nonvalid++;
}

// Closing sequence input file
if (fclose(dnaInFile) != 0) {
    printf("Error closing sequence input file\n");
}

// Allocating memory to store the sequence and header line
header = (char*)malloc(size*sizeof(char));
sequence = (char*)malloc((sequenceLength + 1)*sizeof(char));

// Reopening sequence input file
dnaInFile = fopen(DnaFileName, "r");
if (dnaInFile == NULL) {
    printf("Error opening sequence input file %s file\n", DnaFileName);
    exit(-1);
}

// Pruning '>'
fread(buffer, sizeof(char), 1, dnaInFile);

// Reading sequence name
status = fscanf(dnaInFile, "%s", header);
if (status <= 0 || status > size) {
    printf("Error reading %s file header\n", DnaFileName);
    exit(-1);
}

size = 1; /* Why shouldn't this value be 0? Would cause an error */
// Reading the sequence
printf("\nReading sequence %s: ", header);
while (fread(buffer, sizeof(char), 1, dnaInFile)) {
    if (VALID_CHAR((int)buffer[0])) {
        sequence[size++] = buffer[0];
    }
}
printf("done\n");
if (size-1 != sequenceLength) {
    printf("Error reading %s file sequence\n", DnaFileName);
    exit(-1);
}

// Closing sequence input file
if (fclose(dnaInFile) != 0) {
    printf("Error closing sequence input file\n");
}

// Saving sequence data
dnaHeader = header;
dnaSequence = sequence;
dnaLength = sequenceLength;

free(buffer);
}

//
// This is the function for reading a filter threshold from file
//
float read_threshold(const char * FilterFileName, FILE *filterInFile) {
    char *header;
    int status = 0;
    int size = 32; /* size of header */
    float threshold;

    header = (char*)malloc(size*sizeof(char));

    // Reading matrix name
    status = fscanf(filterInFile, "%s", header);
}

```



```

if(status <= 0 || status > size){
    printf("Error reading %s file header\n", FilterFileName);
    exit(-1);
}

// Reading threshold value
fscanf(filterInFile, "%f", &threshold);

free(header);

// Returning threshold value
return threshold;
}

//
// This is the function for calculating pwm-values from count-values
//
float convert_value(float c, float N) {
    double s = 0.25;
    double P = 0.25;
    double p, valueTemp;
    float pwmValueTemp;

    p = (double) ((c+s)/(N+(4*s)));
    valueTemp = (double) (p/P);
    pwmValueTemp = (float) log(valueTemp);

    return pwmValueTemp;
}

//
// This is the function for converting an alignment matrix to pwm
//
float *convert_matrix(float *matrix) {
    int i, j;
    float *pwmTemp;
    double N;

    pwmTemp = (float *)malloc(ROWS*COLUMNS*sizeof(float));
    for (j = 0; j < ROWS; j++) {
        N = (matrix[j*COLUMNS+0]+matrix[j*COLUMNS+1]+matrix[j*COLUMNS+2]+matrix[j*COLUMNS+3]);
        for (i = 0; i < COLUMNS; i++) {
            pwmTemp[j*COLUMNS+i] = convert_value(matrix[j*COLUMNS+i], N);
        }
    }
    return pwmTemp;
}

//
// This is the function for reading an input matrix from a file and converting it to a pwm,
// based on a function from the Impulse CoDeveloper example project 'SmithWatermanSerial'
//
void read_matrix(const char *PwmFileName, FILE *pwmInFile) {
    char *buffer = (char *)malloc(32*sizeof(char));
    char *header;
    int status = 0;
    int size = 32; /* size of header */
    int matrixSize = COLUMNS*ROWS;
    float *matrix;
    int i, j;

    // Finding the size of the header line and pruning '>'
    status = fread(buffer, sizeof(char), 1, pwmInFile);

    // Making sure the next matrix have been found
    while(buffer[0] != '>') {
        status = fread(buffer, sizeof(char), 1, pwmInFile);
    }

    // Allocating memory to store header line and matrix
    header = (char *)malloc(size*sizeof(char));
    matrix = (float *)malloc(matrixSize*sizeof(float));

    // Reading matrix name
    status = fscanf(pwmInFile, "%s", header);
    if(status <= 0 || status > size){
        printf("Error reading %s file header\n", PwmFileName);
        exit(-1);
    }

    // Reading and storing the matrix
    printf("\nReading matrix %s: ", header);
    for(j = 0; j < ROWS; j++) {
        for(i = 0; i < COLUMNS; i++) {
            fscanf(pwmInFile, "%f", &matrix[j*COLUMNS+i]);

```

```

    }
    printf("done\n");

    // Saving pwm data
    pwmHeader = header;
    printf("Converting %s: ", header);
    pwm = convert_matrix(matrix);
    printf("done\n");
    pwmSize = matrixSize;

    free(matrix);
    free(buffer);
}

//
// This is the software 'reader' process
//
void Producer(co_stream threshold_0, co_stream threshold_1, co_stream pwm_0, co_stream pwm_1,
             co_stream iteration_0,
             co_stream iteration_1, co_stream dna_0, co_stream dna_1, co_parameter filters)
{
    int c;
    IF_SIM(cosim_logwindow log = cosim_logwindow_create("Producer");)

    // Opening streams
    co_stream_open(threshold_0, O_WRONLY, FLOAT_TYPE);
    co_stream_open(threshold_1, O_WRONLY, FLOAT_TYPE);
    co_stream_open(pwm_0, O_WRONLY, FLOAT_TYPE);
    co_stream_open(pwm_1, O_WRONLY, FLOAT_TYPE);
    co_stream_open(iteration_0, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(iteration_1, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(dna_0, O_WRONLY, CHAR_TYPE);
    co_stream_open(dna_1, O_WRONLY, CHAR_TYPE);

    // Preparing to send thresholds
    int n;
    float nSampleThreshold;

    // Preparing to send matrices
    int i, j;
    float nSamplePwm;

    // Preparing to send sequence
    int k;
    char nSampleDna;

    // Reading sequence from input file
    const char * DnaFileName = DNA_INPUT_FILE;
    FILE * dnaInFile;
    read_sequence(DnaFileName, dnaInFile);

    // Determining type of filter based on command line argument
    if (filters == NULL) {
        // Filter is set to summation filter
        filterMode = 1;
    } else {
        // Filter is set to threshold filter
        filterMode = 2;

        // Opening threshold input file
        const char * FilterFileName = filters;
        FILE * filterInFile;
        filterInFile = fopen(FilterFileName, "r");
        if ( filterInFile == NULL ) {
            fprintf(stderr, "Error opening filter input file %s\n", FilterFileName);
            c = getc(stdin);
            exit(-1);
        }

        // Reading threshold value for first matrix from input file and sending it
        nSampleThreshold = read_threshold(FilterFileName, filterInFile);
        co_stream_write(threshold_0, &nSampleThreshold, sizeof(float));
        // Reading threshold value for second matrix from input file and sending it
        nSampleThreshold = read_threshold(FilterFileName, filterInFile);
        co_stream_write(threshold_1, &nSampleThreshold, sizeof(float));

        // Closing threshold input file
        if (fclose(filterInFile) != 0) {
            printf("Error closing filter input file\n");
        }
    }

    // Opening matrix input file
    const char * PwmFileName = PWM_INPUT_FILE;
    FILE * pwmInFile;
    pwmInFile = fopen(PwmFileName, "r");

```

```

if ( pwmInFile == NULL ) {
    fprintf(stderr, "Error opening matrix input file %s\n", PwmFileName);
    c = getc(stdin);
    exit(-1);
}

// Reading first input matrix from input file and converting it to pwm
read_matrix(PwmFileName, pwmInFile);
pwmList[0] = (char*)pwmHeader;

// Sending first matrix
printf("Sending %s: ", pwmList[0]);
for(i = 0; i < pwmSize; i++) {
    nSamplePwm = (float)pwm[i];
    co_stream_write(pwm_0, &nSamplePwm, sizeof(float));
}
printf("done\n");
free(pwm);

// Reading second input matrix from input file and converting it to pwm
read_matrix(PwmFileName, pwmInFile);
pwmList[1] = (char*)pwmHeader;

// Sending second matrix
printf("Sending %s: ", pwmList[1]);
for(i = 0; i < pwmSize; i++) {
    nSamplePwm = (float)pwm[i];
    co_stream_write(pwm_1, &nSamplePwm, sizeof(float));
}
printf("done\n");
free(pwm);

// Closing matrix input file
if(fclose(pwmInFile) != 0) {
    printf("Error closing matrix input file\n");
}

// Sending sequence length and actual sequence
co_stream_write(iteration_0, &dnaLength, sizeof(int16));
co_stream_write(iteration_1, &dnaLength, sizeof(int16));
printf("\nSending %s...\n\n", dnaHeader);
for(k = 1; k <= dnaLength; k++) {
    nSampleDna = (char)dnaSequence[k];
    co_stream_write(dna_0, &nSampleDna, sizeof(char));
    co_stream_write(dna_1, &nSampleDna, sizeof(char));
}
printf("done sending %s\n", dnaHeader);
free(dnaSequence);

// Closing streams
co_stream_close(threshold_0);
co_stream_close(threshold_1);
co_stream_close(pwm_0);
co_stream_close(pwm_1);
co_stream_close(iteration_0);
co_stream_close(iteration_1);
co_stream_close(dna_0);
co_stream_close(dna_1);
}

//
// This is the software 'writer' process
//
void Consumer(co_stream start_0, co_stream score_0, co_stream start_1, co_stream score_1)
{
    int16 nResultStart;
    int resultEnd;
    float nResultScore;
    unsigned int count = 0;
    const char * FileName = OUTPUT_FILE;
    FILE * outFile;

    IF_SIM(cosim_logwindow log = cosim_logwindow_create("Consumer");)

    // Opening output file
    outFile = fopen(FileName, "w");
    if ( outFile == NULL ) {
        fprintf(stderr, "Error opening file %s for writing\n", FileName);
        exit(-1);
    }

    // Opening streams
    co_stream_open(start_0, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(score_0, O_RDONLY, FLOAT_TYPE);
    co_stream_open(start_1, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(score_1, O_RDONLY, FLOAT_TYPE);
}

```

```

IF_SIM(cosim_logwindow_write(log, "Consumer reading results...\n");)

// Reading filtered results streamed from first filter; then writing them to screen and
file
while (co_stream_read(start_0, &nResultStart, sizeof(int16)) == co_err_none) {
    if (co_stream_read(score_0, &nResultScore, sizeof(float)) == co_err_none) {
        if (filterMode == 1) { // Summation filter
            resultEnd = dnaLength - 1;
        } else { // Threshold filter
            resultEnd = nResultStart + (ROWS-1);
        }
        fprintf(outFile, "%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmList[0],
                nResultScore);
        IF_SIM(cosim_logwindow_fwrite(log, "Result: %s %d %d %s %f\n", dnaHeader, nResultStart,
                resultEnd, pwmList[0], nResultScore);)
        printf("%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmList[0], nResultScore
                );
        count++;
    }
}

// Reading filtered results streamed from second filter; then writing them to screen and
file
while (co_stream_read(start_1, &nResultStart, sizeof(int16)) == co_err_none) {
    if (co_stream_read(score_1, &nResultScore, sizeof(float)) == co_err_none) {
        if (filterMode == 1) { // Summation filter
            resultEnd = dnaLength - 1;
        } else { // Threshold filter
            resultEnd = nResultStart + (ROWS-1);
        }
        fprintf(outFile, "%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmList[1],
                nResultScore);
        IF_SIM(cosim_logwindow_fwrite(log, "Result: %s %d %d %s %f\n", dnaHeader, nResultStart,
                resultEnd, pwmList[1], nResultScore);)
        printf("%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmList[1], nResultScore
                );
        count++;
    }
}

IF_SIM(cosim_logwindow_fwrite(log, "Consumer read %d filtered results\n", count);)
printf("\n\nThe application produced %d filtered scores for %s\n", count, dnaHeader);

free(dnaHeader);
free(pwmHeader);

// Closing output file
if (fclose(outFile) != 0) {
    printf("Error closing result output file\n");
}

// Closing streams
co_stream_close(start_0);
co_stream_close(score_0);
co_stream_close(start_1);
co_stream_close(score_1);
}

//
// Impulse C Main Function
//
int main(int argc, char **argv)
{
    co_architecture my_arch;
    void *param = NULL;
    char *filter;
    char **arg;
    int c;

    printf("\n\n== IMPULSE-C APPLICATION: FPWM 2008 ==\n");
    switch (argc) {
        case 1:
            printf("Executing with summation filter\n");
            my_arch = co_initialize(param);
            co_execute(my_arch);
            break;
        case 2:
            printf("Executing with threshold filter\n");
            arg = (char**) argv;
            filter = (char*) arg[1];
            my_arch = co_initialize(filter);
            co_execute(my_arch);
            break;
        default:
            printf("\nWrong use of parameters!\n");
            break;
    }
}

```

```

printf("Application complete! Press the Enter key to continue...\n");
c = getc(stdin);

return(0);
}

```

FPWM2008_hw.c

```

////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
// FPWM2008_hw.c: includes the hardware processes and configuration
// function for the parallel floating-point version of the FPWM
// system.
//
// See additional comments in FPWM2008.h.
//
#include "co.h"
#include "cosim_log.h"
#include "FPWM2008.h"
#include "co_math.h"

// Software process declarations (see FPWM2008_sw.c)
extern void Producer(co_stream threshold_0, co_stream threshold_1, co_stream pwm_0, co_stream
pwm_1, co_stream iteration_0,
co_stream iteration_1, co_stream dna_0, co_stream dna_1, co_parameter filters);
extern void Consumer(co_stream start_0, co_stream score_0, co_stream start_1, co_stream
score_1);

//
// This is the hardware 'pwm' process
//
void PWM(co_stream pwm, co_stream iteration, co_stream dna, co_stream counter, co_stream sum,
co_parameter nInstance)
{
int i, j, k, l;
float nSamplePwm;
float matrix[COLUMNS][ROWS];
int16 dnaLength;
int motifLength;
char nSampleDna;
char sequence[MAX_SEQUENCE];
int16 nCounter;
float nSum;
int16 nHitStart;
float nHitScore;

IF_SIM(int samplesread; int resultswritten;)

IF_SIM(cosim_logwindow log;)
IF_SIM(log = cosim_logwindow_create("PWM");)

do { // Hardware processes run forever
IF_SIM(samplesread=0; resultswritten=0;)

// Stating motif length
motifLength = ROWS;

// Opening streams
co_stream_open(pwm, O_RDONLY, FLOAT_TYPE);
co_stream_open(iteration, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
co_stream_open(dna, O_RDONLY, CHAR_TYPE);
co_stream_open(counter, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
co_stream_open(sum, O_WRONLY, FLOAT_TYPE);

// Reading pwm from stream
for(j=0; j<ROWS; j++) {
for(i=0; i<COLUMNS; i++) {
co_stream_read(pwm, &nSamplePwm, sizeof(float));
matrix[i][j] = (float)nSamplePwm;
}
}

// Reading sequence length from stream
if(co_stream_read(iteration, &dnaLength, sizeof(int16)) == co_err_none) {
IF_SIM(cosim_logwindow_fwrite(log, "Sequence length is %d\n", dnaLength);)
}

// Reading portion of sequence from stream necessary to start computation
for(k = 0; k < motifLength; k++) {
co_stream_read(dna, &nSampleDna, sizeof(char));
IF_SIM(samplesread++);
sequence[k] = (char)nSampleDna;
}
}

```

```

// Initiating counter and sum
nCounter = 0;
nSum = 0;

// Computing result scores for all subsequences
while(nCounter <= (dnaLength - motifLength)) {

    // Calculating score for current position
    for (l = 0; l < motifLength; l++) {
        #pragma CO PIPELINE
        #pragma CO SET stageDelay 32
        switch(sequence[l+nCounter]) {
            case((char)'A'):
                nSum += matrix[0][l];
                break;
            case((char)'C'):
                nSum += matrix[1][l];
                break;
            case((char)'G'):
                nSum += matrix[2][l];
                break;
            case((char)'T'):
                nSum += matrix[3][l];
                break;
            default:
                // Do nothing here.
                break;
        }
    }

    // Stating that counter and sum will be result data
    nHitStart = nCounter;
    nHitScore = nSum;

    // Sending result to filter
    co_stream_write(counter, &nHitStart, sizeof(int16));
    co_stream_write(sum, &nHitScore, sizeof(float));

    IF_SIM(resultswritten++);
    IF_SIM(cosim_logwindow_fwrite(log, "Wrote score %f to filter, for pattern starting at
        position %d.\n", nHitScore, nHitStart);)

    // Reading new sequence base from stream
    if(co_stream_read(dna, &nSampleDna, sizeof(char)) == co_err_none) {
        IF_SIM(samplesread++);
        sequence[k] = (char)nSampleDna;
        k++;
    } else {
        break;
    }

    // Updating counter and resetting sum
    nCounter++;
    nSum = 0;
}

// Closing streams
co_stream_close(pwm);
co_stream_close(iteration);
co_stream_close(dna);
co_stream_close(counter);
co_stream_close(sum);

IF_SIM(cosim_logwindow_fwrite(log, "Closing PWM process %d; Symbols in sequence read: %d,
    results written: %d\n", nInstance, samplesread, resultswritten);)
IF_SIM(break;) // Only run once for desktop simulation
} while(1);
}

//
// This is the hardware 'filter' process
//
void Filter(co_stream threshold, co_stream counter, co_stream sum, co_stream start, co_stream
    score, co_parameter nInstance)
{
    float nThreshold;
    int filterMode;
    int16 nResultStart;
    float nResultScore;
    int16 nHitStart;
    float nHitScore;
    IF_SIM(int resultsread; int resultswritten;)

    IF_SIM(cosim_logwindow log;)
    IF_SIM(log = cosim_logwindow_create("Filter");)

    do { // Hardware processes run forever

```

```

IF_SIM(resultsread=0; resultswritten=0;)

// Opening streams
co_stream_open(threshold, O_RDONLY, FLOAT_TYPE);
co_stream_open(counter, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
co_stream_open(sum, O_RDONLY, FLOAT_TYPE);
co_stream_open(start, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
co_stream_open(score, O_WRONLY, FLOAT_TYPE);

// Reading threshold value from stream; determining filter mode
if(co_stream_read(threshold, &nThreshold, sizeof(float)) != co_err_none) {
// Filter is set to summation filter
filterMode = 1;
IF_SIM(cosim_logwindow_fwrite(log, "Filter Mode: %d\n", filterMode));
} else {
// Filter is set to threshold filter
filterMode = 2;
IF_SIM(cosim_logwindow_fwrite(log, "Filter Mode: %d\n", filterMode));
IF_SIM(cosim_logwindow_fwrite(log, "Threshold value: %f\n", nThreshold));
}

// Initiating filtered result data
nResultStart = 0;
nResultScore = 0;

// Read computed results from stream
while(co_stream_read(counter, &nHitStart, sizeof(int16)) == co_err_none) {
if(co_stream_read(sum, &nHitScore, sizeof(float)) == co_err_none) {
IF_SIM(resultsread++);

if(filterMode == 1) { // Summation filter
nResultScore += nHitScore;
} else { // Threshold filter
if(nHitScore >= nThreshold) {
// Stating that the result score will pass through the filter
nResultStart = nHitStart;
nResultScore = nHitScore;

// Sending filtered result to consumer
co_stream_write(start, &nResultStart, sizeof(int16));
co_stream_write(score, &nResultScore, sizeof(float));

IF_SIM(resultswritten++);
IF_SIM(cosim_logwindow_fwrite(log, "Filtered score %f for pattern starting at
position %d.\n", nResultScore, nResultStart));
}
}
}
}

if(filterMode == 1) { // Summation filter
// Sending result score from summation filter to consumer
co_stream_write(start, &nResultStart, sizeof(int16));
co_stream_write(score, &nResultScore, sizeof(float));

IF_SIM(resultswritten++);
IF_SIM(cosim_logwindow_fwrite(log, "Filtered combined score %f for the entire sequence
.\n", nResultScore));
}

// Closing streams
co_stream_close(threshold);
co_stream_close(counter);
co_stream_close(sum);
co_stream_close(start);
co_stream_close(score);

IF_SIM(cosim_logwindow_fwrite(log,
"Closing filter process %d; Results read: %d, results filtered: %d\n", nInstance,
resultsread, resultswritten));

IF_SIM(break;) // Only run once for desktop simulation
} while(1);
}

//
// Impulse C configuration function
//
void config_FPWM2008(void *arg)
{
co_stream threshold_0;
co_stream threshold_1;
co_stream pwm_0;
co_stream pwm_1;
co_stream iteration_0;
co_stream iteration_1;
co_stream dna_0;

```

```

co_stream dna_1;
co_stream counter_0;
co_stream sum_0;
co_stream counter_1;
co_stream sum_1;
co_stream start_0;
co_stream score_0;
co_stream start_1;
co_stream score_1;

co_process producer_process;
co_process PWM0;
co_process PWM1;
co_process Filter0;
co_process Filter1;
co_process consumer_process;
IF_SIM(cosim_logwindow_init());

char *parameters;

if (arg != NULL) {
    parameters = (char*) arg;
} else {
    parameters = NULL;
}

threshold_0 = co_stream_create("threshold_0", FLOAT_TYPE, MIN_STREAMDEPTH);
threshold_1 = co_stream_create("threshold_1", FLOAT_TYPE, MIN_STREAMDEPTH);
pwm_0 = co_stream_create("pwm_0", FLOAT_TYPE, COLUMNS*ROWS);
pwm_1 = co_stream_create("pwm_1", FLOAT_TYPE, COLUMNS*ROWS);
iteration_0 = co_stream_create("iteration_0", INT_TYPE(MAX_STREAMWIDTH/4), MIN_STREAMDEPTH);
;
iteration_1 = co_stream_create("iteration_1", INT_TYPE(MAX_STREAMWIDTH/4), MIN_STREAMDEPTH);
;
dna_0 = co_stream_create("dna_0", CHAR_TYPE, MAX_SEQUENCE);
dna_1 = co_stream_create("dna_1", CHAR_TYPE, MAX_SEQUENCE);
counter_0 = co_stream_create("counter_0", INT_TYPE(MAX_STREAMWIDTH/4), FILTER_QUEUE);
sum_0 = co_stream_create("sum_0", FLOAT_TYPE, FILTER_QUEUE);
counter_1 = co_stream_create("counter_1", INT_TYPE(MAX_STREAMWIDTH/4), FILTER_QUEUE);
sum_1 = co_stream_create("sum_1", FLOAT_TYPE, FILTER_QUEUE);
start_0 = co_stream_create("start_0", INT_TYPE(MAX_STREAMWIDTH/4), WRITE_QUEUE);
score_0 = co_stream_create("score_0", FLOAT_TYPE, WRITE_QUEUE);
start_1 = co_stream_create("start_1", INT_TYPE(MAX_STREAMWIDTH/4), WRITE_QUEUE);
score_1 = co_stream_create("score_1", FLOAT_TYPE, WRITE_QUEUE);

producer_process = co_process_create("Producer", (co_function)Producer,
9,
    threshold_0,
    threshold_1,
    pwm_0,
    pwm_1,
    iteration_0,
    iteration_1,
    dna_0,
    dna_1,
    parameters);

PWM0 = co_process_create("PWM0", (co_function)PWM,
6,
    pwm_0,
    iteration_0,
    dna_0,
    counter_0,
    sum_0,
    0);

PWM1 = co_process_create("PWM1", (co_function)PWM,
6,
    pwm_1,
    iteration_1,
    dna_1,
    counter_1,
    sum_1,
    1);

Filter0 = co_process_create("Filter0", (co_function)Filter,
6,
    threshold_0,
    counter_0,
    sum_0,
    start_0,
    score_0,
    0);

Filter1 = co_process_create("Filter1", (co_function)Filter,
6,
    threshold_1,
    counter_1,

```

```
        sum_1,
        start_1,
        score_1,
        1);

consumer_process = co_process_create("Consumer", (co_function)Consumer,
        4,
        start_0,
        score_0,
        start_1,
        score_1);

co_process_config(PWM0, co_loc, "PE0");
co_process_config(PWM1, co_loc, "PE0");
co_process_config(Filter0, co_loc, "PE0");
co_process_config(Filter1, co_loc, "PE0");
}

co_architecture co_initialize(int param)
{
    return(co_architecture_create("FPWM2008", "cray_rt", config_FPWM2008, (void *)param));
}
```

FPWMI

FPWMI.h

```

////////////////////////////////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
#define MAX_STREAMWIDTH 64 /* buffer width for FIFO in hardware */
#define MIN_STREAMDEPTH 1 /* minimum buffer size for FIFO in hardware */

#define DNA_INPUT_FILE "dna.txt"
#define PWM_INPUT_FILE "pwm.txt"
#define OUTPUT_FILE "out.txt"

#define COLUMNS 4 /* A,C,G,T */
#define ROWS 8 /* length of pattern/motif */

#define MAX_SEQUENCE 300

#define FILTER_QUEUE MAX_SEQUENCE /* FIFO length between PWM and Filter */
#define WRITE_QUEUE ((MAX_SEQUENCE*10)/100) /* FIFO length between Filter and Consumer */

#define VALID_CHAR(a) (((a) > 64) && ((a) < 91) && ((a) != 74) && ((a) != 79) && ((a) != 85)
? (1) : (0))

```

FPWMI_sw.c

```

////////////////////////////////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
// FPWMI_sw.c: includes the software test bench processes and
// main() function for the basic "fixed-point" version of the
// FPWM system.
//
// See additional comments in FPWMI.h.
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "co.h"
#include "cosim_log.h"
#include "FPWMI.h"

extern co_architecture co_initialize(void *);

// Globals
static char *pwmHeader, *dnaHeader;
static int *pwm;
static char *dnaSequence;
static int pwmSize, dnaLength;
static int filterMode;

//
// This is the function for encoding a floating-point value to fixed-point,
// taken from the software framework of the existing VHDL solution
//
int fpEncode (float x) {
    return x * 16777216;
}

//
// This is the function for decoding a fixed-point value back to floating-point,
// taken from the software framework of the existing VHDL solution
//
float fpDecode (int x) {
    return (float) ((x)/16777216.0);
}

//
// This is the function for calculating pwm-values from count-values
//
int convert_value(float c, float N) {
    double s = 0.25;
    double P = 0.25;
    double p, valueTemp;
    float pwmValueTemp;

```

```

    p = (double) ((c+s)/(N+(4*s)));
    valueTemp = (double) (p/P);
    pwmValueTemp = (float) log(valueTemp);

    return fpEncode(pwmValueTemp);
}

//
// This is the function for converting an alignment matrix to a pwm
//
int *convert_matrix (float * matrix) {
    int i, j;
    int *pwmTemp;
    double N;

    pwmTemp = (int*) malloc(ROWS*COLUMNS*sizeof(int));
    for (j = 0; j < ROWS; j++) {
        N = (matrix[j*COLUMNS+0]+matrix[j*COLUMNS+1]+matrix[j*COLUMNS+2]+matrix[j*COLUMNS+3]);
        for (i = 0; i < COLUMNS; i++) {
            pwmTemp[j*COLUMNS+i] = convert_value(matrix[j*COLUMNS+i], N);
        }
    }
    return pwmTemp;
}

//
// This is the function for reading an input matrix from a file and converting it to a pwm,
// based on a function from the Impulse CoDeveloper example project 'SmithWatermanSerial'
//
void read_matrix(const char * PwmFileName, FILE *pwmInFile) {
    char *buffer = (char*) malloc(32*sizeof(char));
    char *header;
    int status = 0;
    int size = 32; /* size of header */
    int matrixSize = COLUMNS*ROWS;
    float *matrix;
    int i, j;
    char character;

    // Opening matrix input file
    pwmInFile = fopen(PwmFileName, "r");
    if (pwmInFile == NULL) {
        fprintf(stderr, "Error opening matrix input file %s\n", PwmFileName);
        character = getc(stdin);
        exit(-1);
    }

    // Finding the size of the header line
    status = fread(buffer, sizeof(char), 1, pwmInFile);

    // Making sure the matrix is in the correct input format
    if(buffer[0] != '>') {
        printf("Matrix in %s must be in FASTA format, starting with '>'\n", PwmFileName);
    }

    // Closing matrix input file
    if(fclose(pwmInFile) != 0) {
        printf("Error closing matrix input file\n");
    }

    // Allocating memory to store header line
    header = (char*) malloc(size*sizeof(char));
    matrix = (float*) malloc(matrixSize*sizeof(float));

    // Reopening matrix input file
    pwmInFile = fopen(PwmFileName, "r");
    if (pwmInFile == NULL) {
        printf("Error opening matrix input file %s file\n", PwmFileName);
        exit(-1);
    }

    // Pruning '>'
    fread(buffer, sizeof(char), 1, pwmInFile);

    // Reading matrix name
    status = fscanf(pwmInFile, "%s", header);
    if(status <= 0 || status > size){
        printf("Error reading %s file header\n", PwmFileName);
        exit(-1);
    }

    // Reading and storing the matrix
    printf("\nReading matrix %s: ", header);
    for(j = 0; j < ROWS; j++) {
        for(i = 0; i < COLUMNS; i++) {

```

```

        fscanf(pwmInFile, "%f", &matrix[j*COLUMNS+i]);
    }
    printf("done\n");

    // Closing matrix input file
    if (fclose(pwmInFile) != 0) {
        printf("Error closing matrix input file\n");
    }

    // Saving pwm data
    pwmHeader = header;
    printf("Converting %s: ", header);
    pwm = convert_matrix(matrix);
    printf("done\n");
    pwmSize = matrixSize;

    free(matrix);
    free(buffer);
}

//
// This is the function for reading an input sequence from a file, based on a function
// from the Impulse CoDeveloper example project 'SmithWatermanSerial'
//
void read_sequence(const char *DnaFileName, FILE *dnaInFile) {
    char *buffer = (char*)malloc(32*sizeof(char));
    char *sequence, *header;
    int sequenceLength;
    int size = 32;
    int status = 0;
    int nonvalid;
    char character;

    // Opening sequence input file
    dnaInFile = fopen(DnaFileName, "r");
    if ( dnaInFile == NULL ) {
        fprintf(stderr, "Error opening sequence input file %s\n", DnaFileName);
        character = getc(stdin);
        exit(-1);
    }
    // Finding the size of header line
    status = fread(buffer, sizeof(char), 1, dnaInFile);

    // Making sure the sequence is in the correct input format
    if (buffer[0] != '>') {
        printf("Sequence in %s must be in FASTA format, starting with '>'\n", DnaFileName);
    }

    // Trimming off non-valid characters before sequence
    buffer[0] = '\n';
    while (!(VALID_CHAR((int)buffer[0]))) {
        fread(buffer, sizeof(char), 1, dnaInFile);
    }

    // Finding the size of the sequence, after pruning the header line
    fscanf(dnaInFile, "%s", buffer);
    sequenceLength = 0;
    nonvalid = 0;
    while (fread(buffer, sizeof(char), 1, dnaInFile)) {
        if (VALID_CHAR((int)buffer[0]))
            sequenceLength++;
        if (!(VALID_CHAR((int)buffer[0])))
            nonvalid++;
    }

    // Closing sequence input file
    if (fclose(dnaInFile) != 0) {
        printf("Error closing sequence input file\n");
    }

    // Allocating memory to store the sequence and header line
    header = (char*)malloc(size*sizeof(char));
    sequence = (char*)malloc((sequenceLength + 1)*sizeof(char));

    // Reopening sequence input file
    dnaInFile = fopen(DnaFileName, "r");
    if ( dnaInFile == NULL ){
        printf("Error opening sequence input file %s file\n", DnaFileName);
        exit(-1);
    }

    // Pruning '>'
    fread(buffer, sizeof(char), 1, dnaInFile);

    // Reading sequence name
    status = fscanf(dnaInFile, "%s", header);
}

```

```

if(status <= 0 || status > size){
    printf("Error reading %s file header\n", DnaFileName);
    exit(-1);
}

size = 1; /* Why shouldn't this value be 0? Would cause an error */
//Reading the sequence
printf("\nReading sequence %s: ", header);
while (fread(buffer, sizeof(char), 1, dnaInFile)) {
    if (VALID_CHAR((int)buffer[0])) {
        sequence[size++] = buffer[0];
    }
}
printf("done\n");
if (size-1 != sequenceLength){
    printf("Error reading %s file sequence\n", DnaFileName);
    exit(-1);
}

// Closing sequence input file
if (fclose(dnaInFile) != 0) {
    printf("Error closing sequence input file\n");
}

// Save sequence data
dnaHeader = header;
dnaSequence = sequence;
dnaLength = sequenceLength;

free (buffer);
}

//
// This is the software 'reader' process
//
void Producer(co_stream threshold_stream, co_stream pwm_stream, co_stream iteration_stream,
             co_stream dna_stream, co_parameter filter)
{
    float threshold;
    IF_SIM(cosim_logwindow log = cosim_logwindow_create("Producer");)

    // Opening streams
    co_stream_open(threshold_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));
    co_stream_open(pwm_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));
    co_stream_open(dna_stream, O_WRONLY, CHAR_TYPE);
    co_stream_open(iteration_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));

    // Determining type of filter based on command line argument
    if (filter == NULL) {
        // Filter is set to summation filter
        filterMode = 1;
    } else {
        // Filter is set to threshold filter
        filterMode = 2;
        threshold = fpEncode(atof(filter));
        co_stream_write(threshold_stream, &threshold, sizeof(int32));
    }

    // Reading matrix from input file and converting it to pwm
    const char * PwmFileName = PWM_INPUT_FILE;
    FILE * pwmInFile;
    read_matrix(PwmFileName, pwmInFile);

    // Sending matrix
    int i, j;
    int nSamplePwm;
    printf("Sending %s: ", pwmHeader);
    for(i = 0; i < pwmSize; i++) {
        nSamplePwm = (int)pwm[i];
        co_stream_write(pwm_stream, &nSamplePwm, sizeof(int32));
    }
    printf("done\n");

    free (pwm);

    // Reading sequence from input file
    const char * DnaFileName = DNA_INPUT_FILE;
    FILE * dnaInFile;
    read_sequence(DnaFileName, dnaInFile);

    // Sending sequence length and actual sequence
    int k;
    char nSampleDna;
    co_stream_write(iteration_stream, &dnaLength, sizeof(int16));
    printf("Sending %s...\n\n", dnaHeader);
    for(k = 1; k <= dnaLength; k++) {
        nSampleDna = (char)dnaSequence[k];
    }
}

```

```

    co_stream_write(dna_stream, &nSampleDna, sizeof(char));
}
printf("done sending %s\n", dnaHeader);

free(dnaSequence);

// Closing streams
co_stream_close(threshold_stream);
co_stream_close(pwm_stream);
co_stream_close(iteration_stream);
co_stream_close(dna_stream);
}

//
// This is the software 'writer' process
//
void Consumer(co_stream start_stream, co_stream score_stream)
{
    int16 nResultStart;
    int resultEnd;
    int32 nResultScore;
    float resultScore;
    unsigned int count = 0;
    const char * FileName = OUTPUT_FILE;
    FILE * outFile;

    IF_SIM(cosim_logwindow_log = cosim_logwindow_create("Consumer");)

    // Opening output file
    outFile = fopen(FileName, "w");
    if ( outFile == NULL ) {
        fprintf(stderr, "Error opening file %s for writing\n", FileName);
        exit(-1);
    }

    // Opening streams
    co_stream_open(start_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(score_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/2));

    IF_SIM(cosim_logwindow_write(log, "Consumer reading results...\n");)

    // Reading filtered results from stream; then writing them to screen and file
    while (co_stream_read(start_stream, &nResultStart, sizeof(int16)) == co_err_none) {
        if(co_stream_read(score_stream, &nResultScore, sizeof(int32)) == co_err_none) {
            if(filterMode == 1) { // Summation filter
                resultEnd = dnaLength - 1;
            } else { // Threshold filter
                resultEnd = nResultStart + (ROWS-1);
            }
            resultScore = fpDecode(nResultScore);
            fprintf(outFile, "%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmHeader,
                resultScore);
            IF_SIM(cosim_logwindow_fwrite(log, "Result: %s %d %d %s %f\n", dnaHeader, nResultStart,
                resultEnd, pwmHeader, resultScore);)
            printf("%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmHeader, resultScore);
            count++;
        }
    }
    IF_SIM(cosim_logwindow_fwrite(log, "Consumer read %d filtered results\n", count);)
    printf("\n\nThe application produced %d filtered scores for %s\n", count, dnaHeader);

    free(dnaHeader);
    free(pwmHeader);

    // Closing output file
    if (fclose(outFile) != 0) {
        printf("Error closing result output file\n");
    }

    // Closing streams
    co_stream_close(start_stream);
    co_stream_close(score_stream);
}

//
// Impulse C Main Function
//
int main(int argc, char **argv)
{
    co_architecture my_arch;
    void *param = NULL;
    char *filter;
    char **arg;
    int c;

    printf("\n\n== IMPULSE-C APPLICATION: FPWMI ==\n");
}

```

```

switch(argc) {
  case 1:
    printf("Executing with summation filter\n");
    my_arch = co_initialize(param);
    co_execute(my_arch);
    break;
  case 2:
    printf("Executing with threshold filter; ");
    arg = (char**)argv;
    filter = (char*)arg[1];
    printf("threshold set at %s\n", filter);
    my_arch = co_initialize(filter);
    co_execute(my_arch);
    break;
  default:
    printf("\nWrong use of parameters!\n");
    break;
}

printf("Application complete! Press the Enter key to continue...\n");
c = getc(stdin);

return(0);
}

```

FPWMI_hw.c

```

/////////////////////////////////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
// FPWMI_hw.c: includes the hardware processes and configuration
// function for the basic "fixed-point" version of the FPWM
// system.
//
// See additional comments in FPWMI.h.
//
#include "co.h"
#include "cosim_log.h"
#include "FPWMI.h"
#include "co_math.h"

// Software process declarations (see FPWMI_sw.c)
extern void Producer(co_stream threshold_stream, co_stream pwm_stream, co_stream
  iteration_stream, co_stream dna_stream,
  co_parameter filter);
extern void Consumer(co_stream start_stream, co_stream score_stream);

//
// This is the hardware 'pwm' process
//
void PWM(co_stream pwm_stream, co_stream iteration_stream, co_stream dna_stream, co_stream
  counter_stream, co_stream sum_stream)
{
  int i, j, k, l;
  int32 nSamplePwm;
  int32 pwm[COLUMNS][ROWS];
  int16 dnaLength;
  int motifLength;
  char nSampleDna;
  char sequence[MAX_SEQUENCE];
  int16 counter;
  int32 sum;
  int16 nHitStart;
  int32 nHitScore;

  IF_SIM(int samplesread; int resultswritten;)

  IF_SIM(cosim_logwindow log;)
  IF_SIM(log = cosim_logwindow_create("PWM");)

  do { // Hardware processes run forever
    IF_SIM(samplesread=0; resultswritten=0;)

    // Stating motif length
    motifLength = ROWS;

    // Opening streams
    co_stream_open(pwm_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/2));
    co_stream_open(iteration_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(dna_stream, O_RDONLY, CHAR_TYPE);
    co_stream_open(counter_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(sum_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));

    // Reading pwm from stream

```

```

for(j=0; j<ROWS; j++) {
    for(i=0; i<COLUMNS; i++) {
        co_stream_read(pwm_stream, &nSamplePwm, sizeof(int32));
        pwm[i][j] = (int32)nSamplePwm;
    }
}

// Reading sequence length from stream
if(co_stream_read(iteration_stream, &dnaLength, sizeof(int16)) == co_err_none) {
    IF_SIM(cosim_logwindow_fwrite(log, "Sequence length is %d\n", dnaLength);)
}

// Reading portion of sequence from stream necessary to start computation
for(k = 0; k < motifLength; k++) {
    co_stream_read(dna_stream, &nSampleDna, sizeof(char));
    IF_SIM(samplesread++;)
    sequence[k] = (char)nSampleDna;
}

// Initiating counter and stream
counter = 0;
sum = 0;

// Computing result scores for all subsequences
while(counter <= (dnaLength - motifLength)) {

    // Calculating score for current position
    for(l = 0; l < motifLength; l++) {
        #pragma CO PIPELINE
        #pragma CO SET stageDelay 32
        switch(sequence[l+counter]) {
            case((char)'A'):
                sum += (int32)pwm[0][l];
                break;
            case((char)'C'):
                sum += (int32)pwm[1][l];
                break;
            case((char)'G'):
                sum += (int32)pwm[2][l];
                break;
            case((char)'T'):
                sum += (int32)pwm[3][l];
                break;
            default:
                // Do nothing here.
                break;
        }
    }

    // Stating that counter and sum will be result data
    nHitStart = counter;
    nHitScore = sum;

    // Sending result to filter
    co_stream_write(counter_stream, &nHitStart, sizeof(int16));
    co_stream_write(sum_stream, &nHitScore, sizeof(int32));

    IF_SIM(resultswritten++;)
    IF_SIM(cosim_logwindow_fwrite(log, "Wrote score %d to filter, for pattern starting at
        position %d.\n", nHitScore, nHitStart);)

    // Reading new sequence base from stream
    if(co_stream_read(dna_stream, &nSampleDna, sizeof(char)) == co_err_none) {
        IF_SIM(samplesread++;)
        sequence[k] = (char)nSampleDna;
        k++;
    } else {
        break;
    }
}

// Updating counter and resetting sum
counter++;
sum = 0;
}

// Closing streams
co_stream_close(pwm_stream);
co_stream_close(iteration_stream);
co_stream_close(dna_stream);
co_stream_close(counter_stream);
co_stream_close(sum_stream);

IF_SIM(cosim_logwindow_fwrite(log, "Closing PWM process; Symbols in sequence read: %d,
    results written: %d\n", samplesread, resultswritten);)
IF_SIM(break;) // Only run once for desktop simulation
} while(1);
}

```



```

//
// This is the hardware 'filter' process.
//
void Filter(co_stream threshold_stream, co_stream counter_stream, co_stream sum_stream,
           co_stream start_stream, co_stream score_stream)
{
    int32 threshold;
    int filterMode;
    int16 nResultStart;
    int32 nResultScore;
    int16 nHitStart;
    int32 nHitScore;
    IF_SIM(int resultsread; int resultswritten;)

    IF_SIM(cosim_logwindow log;)
    IF_SIM(log = cosim_logwindow_create("Filter");)

    do { // Hardware processes run forever
        IF_SIM(resultsread=0; resultswritten=0;)

        // Opening streams
        co_stream_open(threshold_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/2));
        co_stream_open(counter_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
        co_stream_open(sum_stream, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/2));
        co_stream_open(start_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
        co_stream_open(score_stream, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));

        // Reading threshold value from stream; determining filter mode
        if(co_stream_read(threshold_stream, &threshold, sizeof(int32)) != co_err_none) {
            // Filter is set to summation filter
            filterMode = 1;
            IF_SIM(cosim_logwindow_fwrite(log, "Filter Mode: %d\n", filterMode);)
        } else {
            // Filter is set to threshold filter
            filterMode = 2;
            IF_SIM(cosim_logwindow_fwrite(log, "Filter Mode: %d\n", filterMode);)
            IF_SIM(cosim_logwindow_fwrite(log, "Threshold value: %d\n", threshold);)
        }

        // Initiating filtered result data
        nResultStart = 0;
        nResultScore = 0;

        // Reading computed results from stream
        while(co_stream_read(counter_stream, &nHitStart, sizeof(int16)) == co_err_none) {
            if(co_stream_read(sum_stream, &nHitScore, sizeof(int32)) == co_err_none) {
                IF_SIM(resultsread++;)

                if(filterMode == 1) { // Summation filter
                    if(nHitScore > 0) {
                        nResultScore += (int32)nHitScore;
                    }
                } else { // Threshold filter
                    if(nHitScore >= threshold) {
                        // Stating that the result score will pass through the filter
                        nResultStart = nHitStart;
                        nResultScore = nHitScore;

                        // Sending filtered result to consumer
                        co_stream_write(start_stream, &nResultStart, sizeof(int16));
                        co_stream_write(score_stream, &nResultScore, sizeof(int32));

                        IF_SIM(resultswritten++;)
                        IF_SIM(cosim_logwindow_fwrite(log, "Filtered score %d for pattern starting at
                            position %d.\n", nResultScore, nResultStart);)
                    }
                }
            }
        }

        if(filterMode == 1) { // Summation filter
            // Sending result score from summation filter to consumer
            co_stream_write(start_stream, &nResultStart, sizeof(int16));
            co_stream_write(score_stream, &nResultScore, sizeof(int32));

            IF_SIM(resultswritten++;)
            IF_SIM(cosim_logwindow_fwrite(log, "Filtered combined score %d for the entire sequence
                \n", nResultScore);)
        }

        // Closing streams
        co_stream_close(threshold_stream);
        co_stream_close(counter_stream);
        co_stream_close(sum_stream);
        co_stream_close(start_stream);
        co_stream_close(score_stream);

        IF_SIM(cosim_logwindow_fwrite(log,

```

```

        "Closing Filter process; Results read: %d, results filtered: %d\n", resultsread,
        resultswritten);)

    IF_SIM(break;) // Only run once for desktop simulation
} while(1);
}

//
// Impulse C configuration function
//
void config_FPWMI(void *arg)
{
    co_stream threshold_stream;
    co_stream pwm_stream;
    co_stream iteration_stream;
    co_stream dna_stream;
    co_stream counter_stream;
    co_stream sum_stream;
    co_stream start_stream;
    co_stream score_stream;

    co_process producer_process;
    co_process pwm_process;
    co_process filter_process;
    co_process consumer_process;
    IF_SIM(cosim_logwindow_init());

    char *parameter;

    if (arg != NULL) {
        parameter = (char*) arg;
    } else {
        parameter = NULL;
    }

    threshold_stream = co_stream_create("threshold_stream", INT_TYPE(MAX_STREAMWIDTH/2),
        MIN_STREAMDEPTH);
    pwm_stream = co_stream_create("pwm_stream", INT_TYPE(MAX_STREAMWIDTH/2), COLUMNS*ROWS);
    iteration_stream = co_stream_create("iteration_stream", INT_TYPE(MAX_STREAMWIDTH/4),
        MIN_STREAMDEPTH);
    dna_stream = co_stream_create("dna_stream", CHAR_TYPE, MAX_SEQUENCE);
    counter_stream = co_stream_create("counter_stream", INT_TYPE(MAX_STREAMWIDTH/4),
        FILTER_QUEUE);
    sum_stream = co_stream_create("sum_stream", INT_TYPE(MAX_STREAMWIDTH/2), FILTER_QUEUE);
    start_stream = co_stream_create("start_stream", INT_TYPE(MAX_STREAMWIDTH/4), WRITE_QUEUE);
    score_stream = co_stream_create("score_stream", INT_TYPE(MAX_STREAMWIDTH/2), WRITE_QUEUE);

    producer_process = co_process_create("Producer", (co_function)Producer,
        5,
        threshold_stream,
        pwm_stream,
        iteration_stream,
        dna_stream,
        parameter);

    pwm_process = co_process_create("PWM", (co_function)PWM,
        5,
        pwm_stream,
        iteration_stream,
        dna_stream,
        counter_stream,
        sum_stream);

    filter_process = co_process_create("Filter", (co_function)Filter,
        5,
        threshold_stream,
        counter_stream,
        sum_stream,
        start_stream,
        score_stream);

    consumer_process = co_process_create("Consumer", (co_function)Consumer,
        2,
        start_stream,
        score_stream);

    co_process_config(pwm_process, co_loc, "PE0");
    co_process_config(filter_process, co_loc, "PE0");
}

co_architecture co_initialize(int param)
{
    return(co_architecture_create("FPWMI", "cray_rt", config_FPWMI, (void *)param));
}

```

FPWM2008i

FPWM2008i.h

```

////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
#define MAX_STREAMWIDTH 64 /* buffer width for FIFO in hardware */
#define MIN_STREAMDEPTH 1 /* minimum buffer size for FIFO in hardware */

#define DNA_INPUT_FILE "dna.txt"
#define PWM_INPUT_FILE "pwm.txt"
#define OUTPUT_FILE "out.txt"

#define COLUMNS 4 /* A,C,G,T */
#define ROWS 8 /* length of pattern/motif */

#define MAX_SEQUENCE 300

#define FILTER_QUEUE MAX_SEQUENCE /* FIFO length between PWM and Filter */
#define WRITE_QUEUE ((MAX_SEQUENCE*10)/100) /* FIFO length between Filter and Consumer */

#define VALID_CHAR(a) (((a) > 64) && ((a) < 91) && ((a) != 74) && ((a) != 79) && ((a) != 85)
? (1) : (0)

```

FPWM2008i_sw.c

```

////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
// FPWM2008i_sw.c: includes the software test bench processes and
// main() function for the parallel "fixed-point" version of the
// FPWM system.
//
// See additional comments in FPWM2008i.h.
//
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "co.h"
#include "cosim_log.h"
#include "FPWM2008i.h"

extern co_architecture co_initialize(void *);

// Globals
static char *pwmHeader, *dnaHeader;
static int *pwm;
static char *dnaSequence;
static int pwmSize, dnaLength;
static int filterMode;
static char* pwmList[2];

//
// This is the function for encoding a floating-point value to fixed-point,
// taken from the software framework of the existing VHDL solution
//
int fpEncode(float x) {
    return x * 16777216;
}

//
// This is the function for decoding a fixed-point value back to floating-point,
// taken from the software framework of the existing VHDL solution
//
float fpDecode(int x) {
    return (float) ((x)/16777216.0);
}

//
// This is the function for reading an input sequence from a file, based on a function
// from the Impulse CoDeveloper example project 'SmithWatermanSerial'
//
void read_sequence(const char *DnaFileName, FILE *dnaInFile) {
    char *buffer = (char*)malloc(32*sizeof(char));
    char *sequence, *header;

```

```

int sequenceLength;
int size = 32;
int status = 0;
int nonvalid;
char character;

// Opening sequence input file
dnaInFile = fopen(DnaFileName, "r");
if ( dnaInFile == NULL ) {
    fprintf(stderr, "Error opening sequence input file %s\n", DnaFileName);
    character = getc(stdin);
    exit(-1);
}

// Finding the size of header line
status = fread(buffer, sizeof(char), 1, dnaInFile);

// Making sure the sequence is in the correct input format
if(buffer[0] != '>') {
    printf("Sequence in %s must be in FASTA format, starting with '>'\n", DnaFileName);
}

// Trimming off non-valid characters before sequence
buffer[0] = '\n';
while (!(VALID_CHAR((int)buffer[0]))) {
    fread(buffer, sizeof(char), 1, dnaInFile);
}

// Finding the size of the sequence, after pruning the header line
fscanf(dnaInFile, "%s", buffer);
sequenceLength = 0;
nonvalid = 0;
while(fread(buffer, sizeof(char), 1, dnaInFile)) {
    if(VALID_CHAR((int)buffer[0]))
        sequenceLength++;
    if (!(VALID_CHAR((int)buffer[0])))
        nonvalid++;
}

// Closing sequence input file
if(fclose(dnaInFile) != 0) {
    printf("Error closing sequence input file\n");
}

// Allocating memory to store the sequence and header line
header = (char*)malloc(size*sizeof(char));
sequence = (char*)malloc((sequenceLength + 1)*sizeof(char));

// Reopening sequence input file
dnaInFile = fopen(DnaFileName, "r");
if ( dnaInFile == NULL ){
    printf("Error opening sequence input file %s file\n", DnaFileName);
    exit(-1);
}

// Pruning '>'
fread(buffer, sizeof(char), 1, dnaInFile);

// Reading sequence name
status = fscanf(dnaInFile, "%s", header);
if(status <= 0 || status > size){
    printf("Error reading %s file header\n", DnaFileName);
    exit(-1);
}

size = 1; /* Why shouldn't this value be 0? Would cause an error */
//Reading the sequence
printf("\nReading sequence %s: ", header);
while (fread(buffer, sizeof(char), 1, dnaInFile)) {
    if (VALID_CHAR((int)buffer[0])) {
        sequence[size++] = buffer[0];
    }
}
printf("done\n");
if(size-1 != sequenceLength){
    printf("Error reading %s file sequence\n", DnaFileName);
    exit(-1);
}

// Closing sequence input file
if(fclose(dnaInFile) != 0) {
    printf("Error closing sequence input file\n");
}

// Saving sequence data
dnaHeader = header;
dnaSequence = sequence;
dnaLength = sequenceLength;

```

```

    free(buffer);
}

//
// This is the function for reading a filter threshold from file
//
int read_threshold(const char * FilterFileName, FILE *filterInFile) {
    char *header;
    int status = 0;
    int size = 32; /* size of header */
    float threshold;

    header = (char*)malloc(size*sizeof(char));

    // Reading matrix name
    status = fscanf(filterInFile, "%s", header);
    if(status <= 0 || status > size){
        printf("Error reading %s file header\n", FilterFileName);
        exit(-1);
    }

    // Reading threshold value
    fscanf(filterInFile, "%f", &threshold);

    free(header);

    // Returning threshold value
    return fpEncode(threshold);
}

//
// This is the function for calculating pwm-values from count-values
//
int convert_value(float c, float N) {
    double s = 0.25;
    double P = 0.25;
    double p, valueTemp;
    float pwmValueTemp;

    p = (double) ((c+s)/(N+(4*s)));
    valueTemp = (double) (p/P);
    pwmValueTemp = (float) log(valueTemp);

    return fpEncode(pwmValueTemp);
}

//
// This is the function for converting an alignment matrix to a pwm
//
int *convert_matrix(float * matrix) {
    int i, j;
    int *pwmTemp;
    double N;

    pwmTemp = (int*)malloc(ROWS*COLUMNS*sizeof(int));
    for (j = 0; j < COLUMNS; j++) {
        N = (matrix[j*COLUMNS+0]+matrix[j*COLUMNS+1]+matrix[j*COLUMNS+2]+matrix[j*COLUMNS+3]);
        for (i = 0; i < COLUMNS; i++) {
            pwmTemp[j*COLUMNS+i] = convert_value(matrix[j*COLUMNS+i], N);
        }
    }
    return pwmTemp;
}

//
// This is the function for reading an input matrix from a file and converting it to a pwm,
// based on a function from the Impulse CoDeveloper example project 'SmithWatermanSerial'
//
void read_matrix(const char * PwmFileName, FILE *pwmInFile) {
    char *buffer = (char*)malloc(32*sizeof(char));
    char *header;
    int status = 0;
    int size = 32; /* size of header */
    int matrixSize = COLUMNS*ROWS;
    float *matrix;
    int i, j;

    // Finding the size of the header line and pruning '>'
    status = fread(buffer, sizeof(char), 1, pwmInFile);

    // Making sure the next matrix have been found
    while(buffer[0] != '>') {
        status = fread(buffer, sizeof(char), 1, pwmInFile);
    }
}

```

```

}

// Allocating memory to store header line and matrix
header = (char*)malloc(size*sizeof(char));
matrix = (float*)malloc(matrixSize*sizeof(float));

// Reading matrix name
status = fscanf(pwmInFile, "%s", header);
if(status <= 0 || status > size){
    printf("Error reading %s file header\n", PwmFileName);
    exit(-1);
}

// Reading and storing the matrix
printf("\nReading matrix %s: ", header);
for(j = 0; j < ROWS; j++) {
    for(i = 0; i < COLUMNS; i++) {
        fscanf(pwmInFile, "%f", &matrix[j*COLUMNS+i]);
    }
}
printf("done\n");

// Saving pwm data
pwmHeader = header;
printf("Converting %s: ", header);
pwm = convert_matrix(matrix);
printf("done\n");
pwmSize = matrixSize;

free(matrix);
free(buffer);
}

//
// This is the software 'reader' process
//
void Producer(co_stream threshold_0, co_stream threshold_1, co_stream pwm_0, co_stream pwm_1,
             co_stream iteration_0,
             co_stream iteration_1, co_stream dna_0, co_stream dna_1, co_parameter filters)
{
    int c;
    IF_SIM(cosim_logwindow log = cosim_logwindow_create("Producer");)

    // Opening streams
    co_stream_open(threshold_0, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));
    co_stream_open(threshold_1, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));
    co_stream_open(pwm_0, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));
    co_stream_open(pwm_1, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));
    co_stream_open(iteration_0, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(iteration_1, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(dna_0, O_WRONLY, CHAR_TYPE);
    co_stream_open(dna_1, O_WRONLY, CHAR_TYPE);

    // Preparing to send thresholds
    int n;
    int nSampleThreshold;

    // Preparing to send matrices
    int i, j;
    int nSamplePwm;

    // Preparing to send sequence
    int k;
    char nSampleDna;

    // Reading sequence from input file
    const char * DnaFileName = DNA_INPUT_FILE;
    FILE * dnaInFile;
    read_sequence(DnaFileName, dnaInFile);

    // Determining type of filter based on command line argument
    if(filters == NULL) {
        // Filter is set to summation filter
        filterMode = 1;
    } else {
        // Filter is set to threshold filter
        filterMode = 2;

        // Opening threshold input file
        const char * FilterFileName = filters;
        FILE * filterInFile;
        filterInFile = fopen(FilterFileName, "r");
        if ( filterInFile == NULL ) {
            fprintf(stderr, "Error opening filter input file %s\n", FilterFileName);
            c = getc(stdin);
            exit(-1);
        }
    }
}

```

```

// Reading threshold value for first matrix from input file and sending it
nSampleThreshold = read_threshold(FilterFileName, filterInFile);
co_stream_write(threshold_0, &nSampleThreshold, sizeof(int32));
// Reading threshold value for second matrix from input file and sending it
nSampleThreshold = read_threshold(FilterFileName, filterInFile);
co_stream_write(threshold_1, &nSampleThreshold, sizeof(int32));

// Closing threshold input file
if (fclose(filterInFile) != 0) {
    printf("Error closing filter input file\n");
}
}

// Opening matrix input file
const char * PwmFileName = PWM_INPUT_FILE;
FILE * pwmInFile;
pwmInFile = fopen(PwmFileName, "r");
if (pwmInFile == NULL) {
    fprintf(stderr, "Error opening matrix input file %s\n", PwmFileName);
    c = getc(stdin);
    exit(-1);
}

// Reading first input matrix and converting it to pwm
read_matrix(PwmFileName, pwmInFile);
pwmList[0] = (char*)pwmHeader;

// Sending first matrix
printf("Sending %s: ", pwmList[0]);
for(i = 0; i < pwmSize; i++) {
    nSamplePwm = (int)pwm[i];
    co_stream_write(pwm_0, &nSamplePwm, sizeof(int32));
}
printf("done\n");
free(pwm);

// Reading second input matrix and converting it to pwm
read_matrix(PwmFileName, pwmInFile);
pwmList[1] = (char*)pwmHeader;

// Sending second matrix
printf("Sending %s: ", pwmList[1]);
for(i = 0; i < pwmSize; i++) {
    nSamplePwm = (int)pwm[i];
    co_stream_write(pwm_1, &nSamplePwm, sizeof(int32));
}
printf("done\n");
free(pwm);

// Closing matrix input file
if (fclose(pwmInFile) != 0) {
    printf("Error closing matrix input file\n");
}

// Sending sequence length and actual sequence
co_stream_write(iteration_0, &dnaLength, sizeof(int16));
co_stream_write(iteration_1, &dnaLength, sizeof(int16));
printf("\nSending %s...\n", dnaHeader);
for(k = 1; k <= dnaLength; k++) {
    nSampleDna = (char)dnaSequence[k];
    co_stream_write(dna_0, &nSampleDna, sizeof(char));
    co_stream_write(dna_1, &nSampleDna, sizeof(char));
}
printf("done sending %s\n", dnaHeader);
free(dnaSequence);

// Closing streams
co_stream_close(threshold_0);
co_stream_close(threshold_1);
co_stream_close(pwm_0);
co_stream_close(pwm_1);
co_stream_close(iteration_0);
co_stream_close(iteration_1);
co_stream_close(dna_0);
co_stream_close(dna_1);
}

//
// This is the software 'writer' process
//
void Consumer(co_stream start_0, co_stream score_0, co_stream start_1, co_stream score_1)
{
    int16 nResultStart;
    int resultEnd;
    int32 nResultScore;
    float resultScore;

```

```

unsigned int count = 0;
const char * FileName = OUTPUT_FILE;
FILE * outFile;

IF_SIM(cosim_logwindow_log = cosim_logwindow_create("Consumer");)

// Opening output file
outFile = fopen(FileName, "w");
if ( outFile == NULL ) {
    fprintf(stderr, "Error opening file %s for writing\n", FileName);
    exit(-1);
}

// Opening streams
co_stream_open(start_0, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
co_stream_open(score_0, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/2));
co_stream_open(start_1, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
co_stream_open(score_1, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/2));

IF_SIM(cosim_logwindow_write(log, "Consumer reading results...\n");)

// Reading filtered results streamed from first filter; then writing them to screen and
// file
while (co_stream_read(start_0, &nResultStart, sizeof(int16)) == co_err_none) {
    if(co_stream_read(score_0, &nResultScore, sizeof(int32)) == co_err_none) {
        if(filterMode == 1) { // Summation filter
            resultEnd = dnaLength - 1;
        } else { // Threshold filter
            resultEnd = nResultStart + (ROWS-1);
        }
        resultScore = fpDecode(nResultScore);
        fprintf(outFile, "%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmList[0],
            resultScore);
        IF_SIM(cosim_logwindow_fwrite(log, "Result: %s %d %d %s %f\n", dnaHeader, nResultStart,
            resultEnd, pwmList[0], resultScore);)
        printf("%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmList[0], resultScore)
            ;
        count++;
    }
}

// Reading filtered results streamed from second filter; then writing them to screen and
// file
while (co_stream_read(start_1, &nResultStart, sizeof(int16)) == co_err_none) {
    if(co_stream_read(score_1, &nResultScore, sizeof(int32)) == co_err_none) {
        if(filterMode == 1) { // Summation filter
            resultEnd = dnaLength - 1;
        } else { // Threshold filter
            resultEnd = nResultStart + (ROWS-1);
        }
        resultScore = fpDecode(nResultScore);
        fprintf(outFile, "%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmList[1],
            resultScore);
        IF_SIM(cosim_logwindow_fwrite(log, "Result: %s %d %d %s %f\n", dnaHeader, nResultStart,
            resultEnd, pwmList[1], resultScore);)
        printf("%s %d %d %s %f\n", dnaHeader, nResultStart, resultEnd, pwmList[1], resultScore)
            ;
        count++;
    }
}

IF_SIM(cosim_logwindow_fwrite(log, "Consumer read %d filtered results\n", count);)
printf("\n\nThe application produced %d filtered scores for %s\n", count, dnaHeader);

free(dnaHeader);
free(pwmHeader);

// Closing output file
if(fclose(outFile) != 0) {
    printf("Error closing result output file\n");
}

// Closing streams
co_stream_close(start_0);
co_stream_close(score_0);
co_stream_close(start_1);
co_stream_close(score_1);
}

//
// Impulse C Main Function
//
int main(int argc, char **argv)
{
    co_architecture my_arch;
    void *param = NULL;
    char *filter;
}

```



```

char **arg;
int c;

printf("\n\n== IMPULSE-C APPLICATION: FPWM 2008i ==\n");
switch(argc) {
  case 1:
    printf("Executing with summation filter\n");
    my_arch = co_initialize(param);
    co_execute(my_arch);
    break;
  case 2:
    printf("Executing with threshold filter\n");
    arg = (char**)argv;
    filter = (char*)arg[1];
    my_arch = co_initialize(filter);
    co_execute(my_arch);
    break;
  default:
    printf("\nWrong use of parameters!\n");
    break;
}

printf("Application complete! Press the Enter key to continue...\n");
c = getc(stdin);

return(0);
}

```

FPWM2008i_hw.c

```

/////////////////////////////////////////////////////////////////
//
// Impulse-C(c) 2003-2008 Impulse Accelerated Technologies, Inc.
//
// FPWM2008i_hw.c: includes the hardware processes and configuration
// function for the parallel "fixed-point" version of the FPWM
// system.
//
// See additional comments in FPWM2008i.h.
//
#include "co.h"
#include "cosim_log.h"
#include "FPWM2008i.h"
#include "co_math.h"

// Software process declarations (see FPWM2008i_sw.c)
extern void Producer(co_stream threshold_0, co_stream threshold_1, co_stream pwm_0, co_stream
  pwm_1, co_stream iteration_0,
  co_stream iteration_1, co_stream dna_0, co_stream dna_1, co_parameter filters);
extern void Consumer(co_stream start_0, co_stream score_0, co_stream start_1, co_stream
  score_1);

//
// This is the hardware 'pwm' process
//
void PWM(co_stream pwm, co_stream iteration, co_stream dna, co_stream counter, co_stream sum,
  co_parameter nInstance)
{
  int i, j, k, l;
  int32 nSamplePwm;
  int32 matrix[COLUMNS][ROWS];
  int16 dnaLength;
  int motifLength;
  char nSampleDna;
  char sequence[MAX_SEQUENCE];
  int16 nCounter;
  int32 nSum;
  int16 nHitStart;
  int32 nHitScore;

  IF_SIM(int samplesread; int resultswritten;)

  IF_SIM(cosim_logwindow log;)
  IF_SIM(log = cosim_logwindow_create("PWM");)

  do { // Hardware processes run forever
    IF_SIM(samplesread=0; resultswritten=0;)

    // Stating motif length
    motifLength = ROWS;

    // Opening streams
    co_stream_open(pwm, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/2));
    co_stream_open(iteration, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
    co_stream_open(dna, O_RDONLY, CHAR_TYPE);

```

```

co_stream_open(counter, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
co_stream_open(sum, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));

// Reading pwm from stream
for(j=0; j<ROWS; j++) {
    for(i=0; i<COLUMNS; i++) {
        co_stream_read(pwm, &nSamplePwm, sizeof(int32));
        matrix[i][j] = (int32)nSamplePwm;
    }
}

// Reading sequence length from stream
if(co_stream_read(iteration, &dnaLength, sizeof(int16)) == co_err_none) {
    IF_SIM(cosim_logwindow_fwrite(log, "Sequence length is %d\n", dnaLength);)
}

// Reading portion of sequence from stream necessary to start computation
for(k = 0; k < motifLength; k++) {
    co_stream_read(dna, &nSampleDna, sizeof(char));
    IF_SIM(samplesread++);
    sequence[k] = (char)nSampleDna;
}

// Initiating counter and sum
nCounter = 0;
nSum = 0;

// Computing result scores for all subsequences
while(nCounter <= (dnaLength - motifLength)) {

    // Calculating score for current position
    for (l = 0; l < motifLength; l++) {
        #pragma CO PIPELINE
        #pragma CO SET stageDelay 32
        switch(sequence[l+nCounter]) {
            case((char)'A'):
                nSum += (int32)matrix[0][l];
                break;
            case((char)'C'):
                nSum += (int32)matrix[1][l];
                break;
            case((char)'G'):
                nSum += (int32)matrix[2][l];
                break;
            case((char)'T'):
                nSum += (int32)matrix[3][l];
                break;
            default:
                // Do nothing here.
                break;
        }
    }

    // stating that counter and sum will be result data
    nHitStart = nCounter;
    nHitScore = nSum;

    // Sending result to filter
    co_stream_write(counter, &nHitStart, sizeof(int16));
    co_stream_write(sum, &nHitScore, sizeof(int32));

    IF_SIM(resultswritten++);
    IF_SIM(cosim_logwindow_fwrite(log, "Wrote score %d to filter, for pattern starting at
        position %d.\n", nHitScore, nHitStart);)

    // Reading new sequence base from stream
    if(co_stream_read(dna, &nSampleDna, sizeof(char)) == co_err_none) {
        IF_SIM(samplesread++);
        sequence[k] = (char)nSampleDna;
        k++;
    } else {
        break;
    }

    // Updating counter and resetting sum
    nCounter ++;
    nSum = 0;
}

// Closing streams
co_stream_close(pwm);
co_stream_close(iteration);
co_stream_close(dna);
co_stream_close(counter);
co_stream_close(sum);

IF_SIM(cosim_logwindow_fwrite(log, "Closing PWM process %d; Symbols in sequence read: %d,
    results written: %d\n", nInstance, samplesread, resultswritten);)

```

```

    IF_SIM(break;) // Only run once for desktop simulation
} while(1);
}

//
// This is the hardware 'filter' process
//
void Filter(co_stream threshold, co_stream counter, co_stream sum, co_stream start, co_stream
score, co_parameter nInstance)
{
    int32 nThreshold;
    int filterMode;
    int16 nResultStart;
    int32 nResultScore;
    int16 nHitStart;
    int32 nHitScore;
    IF_SIM(int resultsread; int resultswritten;)

    IF_SIM(cosim_logwindow log;)
    IF_SIM(log = cosim_logwindow_create("Filter");)

    do { // Hardware processes run forever
        IF_SIM(resultsread=0; resultswritten=0;)

        // Opening streams
        co_stream_open(threshold, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/2));
        co_stream_open(counter, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/4));
        co_stream_open(sum, O_RDONLY, INT_TYPE(MAX_STREAMWIDTH/2));
        co_stream_open(start, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/4));
        co_stream_open(score, O_WRONLY, INT_TYPE(MAX_STREAMWIDTH/2));

        // Reading threshold value from stream; determining filter mode
        if(co_stream_read(threshold, &nThreshold, sizeof(int32)) != co_err_none) {
            // Filter is set to summation filter
            filterMode = 1;
            IF_SIM(cosim_logwindow_fwrite(log, "Filter Mode: %d\n", filterMode);)
        } else {
            // Filter is set to threshold filter
            filterMode = 2;
            IF_SIM(cosim_logwindow_fwrite(log, "Filter Mode: %d\n", filterMode);)
            IF_SIM(cosim_logwindow_fwrite(log, "Threshold value: %d\n", nThreshold);)
        }

        // Initiating filtered result data
        nResultStart = 0;
        nResultScore = 0;

        // Reading computed results from stream
        while(co_stream_read(counter, &nHitStart, sizeof(int16)) == co_err_none) {
            if(co_stream_read(sum, &nHitScore, sizeof(int32)) == co_err_none) {
                IF_SIM(resultsread++;)

                if(filterMode == 1) { // Summation filter
                    if(nHitScore > 0) {
                        nResultScore += (int32)nHitScore;
                    }
                } else { // Threshold filter
                    if(nHitScore >= nThreshold) {
                        // Stating that the result score will pass through the filter
                        nResultStart = nHitStart;
                        nResultScore = nHitScore;

                        // Sending filtered result to consumer
                        co_stream_write(start, &nResultStart, sizeof(int16));
                        co_stream_write(score, &nResultScore, sizeof(int32));

                        IF_SIM(resultswritten++;)
                        IF_SIM(cosim_logwindow_fwrite(log, "Filtered score %d for pattern starting at
position %d.\n", nResultScore, nResultStart);)
                    }
                }
            }
        }

        if(filterMode == 1) { // Summation filter
            // Sending result score from summation filter to consumer
            co_stream_write(start, &nResultStart, sizeof(int16));
            co_stream_write(score, &nResultScore, sizeof(int32));

            IF_SIM(resultswritten++;)
            IF_SIM(cosim_logwindow_fwrite(log, "Filtered combined score %d for the entire sequence
.\n", nResultScore);)
        }

        // Closing streams
        co_stream_close(threshold);
        co_stream_close(counter);
        co_stream_close(sum);
    }
}

```

```

co_stream_close(start);
co_stream_close(score);

IF_SIM(cosim_logwindow_fwrite(log,
    "Closing filter process %d; Results read: %d, results filtered: %d\n", nInstance,
    resultsread, resultswritten));

IF_SIM(break;) // Only run once for desktop simulation
} while(1);
}

//
// Impulse C configuration function
//
void config_FPWM2008i(void *arg)
{
    co_stream threshold_0;
    co_stream threshold_1;
    co_stream pwm_0;
    co_stream pwm_1;
    co_stream iteration_0;
    co_stream iteration_1;
    co_stream dna_0;
    co_stream dna_1;
    co_stream counter_0;
    co_stream sum_0;
    co_stream counter_1;
    co_stream sum_1;
    co_stream start_0;
    co_stream score_0;
    co_stream start_1;
    co_stream score_1;

    co_process producer_process;
    co_process PWM0;
    co_process PWM1;
    co_process Filter0;
    co_process Filter1;
    co_process consumer_process;
    IF_SIM(cosim_logwindow_init());

    char *parameters;

    if (arg != NULL) {
        parameters = (char*) arg;
    } else {
        parameters = NULL;
    }

    threshold_0 = co_stream_create("threshold_0", INT_TYPE(MAX_STREAMWIDTH/2), MIN_STREAMDEPTH);
    ;
    threshold_1 = co_stream_create("threshold_1", INT_TYPE(MAX_STREAMWIDTH/2), MIN_STREAMDEPTH);
    ;
    pwm_0 = co_stream_create("pwm_0", INT_TYPE(MAX_STREAMWIDTH/2), COLUMNS*ROWS);
    pwm_1 = co_stream_create("pwm_1", INT_TYPE(MAX_STREAMWIDTH/2), COLUMNS*ROWS);
    iteration_0 = co_stream_create("iteration_0", INT_TYPE(MAX_STREAMWIDTH/4), MIN_STREAMDEPTH);
    ;
    iteration_1 = co_stream_create("iteration_1", INT_TYPE(MAX_STREAMWIDTH/4), MIN_STREAMDEPTH);
    ;
    dna_0 = co_stream_create("dna_0", CHAR_TYPE, MAX_SEQUENCE);
    dna_1 = co_stream_create("dna_1", CHAR_TYPE, MAX_SEQUENCE);
    counter_0 = co_stream_create("counter_0", INT_TYPE(MAX_STREAMWIDTH/4), FILTER_QUEUE);
    sum_0 = co_stream_create("sum_0", INT_TYPE(MAX_STREAMWIDTH/2), FILTER_QUEUE);
    counter_1 = co_stream_create("counter_1", INT_TYPE(MAX_STREAMWIDTH/4), FILTER_QUEUE);
    sum_1 = co_stream_create("sum_1", INT_TYPE(MAX_STREAMWIDTH/2), FILTER_QUEUE);
    start_0 = co_stream_create("start_0", INT_TYPE(MAX_STREAMWIDTH/4), WRITE_QUEUE);
    score_0 = co_stream_create("score_0", INT_TYPE(MAX_STREAMWIDTH/2), WRITE_QUEUE);
    start_1 = co_stream_create("start_1", INT_TYPE(MAX_STREAMWIDTH/4), WRITE_QUEUE);
    score_1 = co_stream_create("score_1", INT_TYPE(MAX_STREAMWIDTH/2), WRITE_QUEUE);

    producer_process = co_process_create("Producer", (co_function)Producer,
        9,
        threshold_0,
        threshold_1,
        pwm_0,
        pwm_1,
        iteration_0,
        iteration_1,
        dna_0,
        dna_1,
        parameters);

    PWM0 = co_process_create("PWM0", (co_function)PWM,
        6,
        pwm_0,
        iteration_0,
        dna_0,

```

```
        counter_0,
        sum_0,
        0);

PWM1 = co_process_create("PWM1", (co_function)PWM,
        6,
        pwm_1,
        iteration_1,
        dna_1,
        counter_1,
        sum_1,
        1);

Filter0 = co_process_create("Filter0", (co_function)Filter,
        6,
        threshold_0,
        counter_0,
        sum_0,
        start_0,
        score_0,
        0);

Filter1 = co_process_create("Filter1", (co_function)Filter,
        6,
        threshold_1,
        counter_1,
        sum_1,
        start_1,
        score_1,
        1);

consumer_process = co_process_create("Consumer", (co_function)Consumer,
        4,
        start_0,
        score_0,
        start_1,
        score_1);

co_process_config(PWM0, co_loc, "PE0");
co_process_config(PWM1, co_loc, "PE0");
co_process_config(Filter0, co_loc, "PE0");
co_process_config(Filter1, co_loc, "PE0");
}

co_architecture co_initialize(int param)
{
    return(co_architecture_create("FPWM2008i", "cray_rt", config_FPWM2008i, (void *)param));
}
```


Appendix C

HDL Build Reports

FPWM

```
==== Building target 'build' in file _Makefile =====
HW_SRC="FPWM_hw.c"; \
for i in $HW_SRC; do CPP_INCLUDES="$CPP_INCLUDES -include $i"; done; \
echo | "C:\Impulse\CoDeveloper2\MinGW\bin\gcc" -E -DWIN32 -DIMPULSE_C_SYNTHESIS -DRC_INVOKED -DBYTE="unsigned char" -DWOR
"C:\Impulse\CoDeveloper2\bin\impulse_snoot" -Timpulse-c FPWM.i FPWM.snt
"C:\Impulse\CoDeveloper2\bin\impulse_prep" FPWM.snt FPWM.pk0
Impulse C Transformations
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
processing FPWM.snt...
"C:\Impulse\CoDeveloper2\bin\impulse_porky" -iterate -fold -unused-syms -unused-types -Dmemcpys -const-prop -scalarize
"C:\Impulse\CoDeveloper2\bin\impulse_porky" -build-arefs FPWM.pk1 FPWM.pky
"C:\Impulse\CoDeveloper2\bin\impulse_imp" FPWM.pky FPWM.sic
Impulse C Preprocessor
Copyright 2003-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
---Software activated---
processing FPWM.pky...
analyzing co_initialize...
found architecture definition: FPWM/generic_vhdl
analyzing config_FPWM...creating stream threshold_stream ...
creating stream pwm_stream ...
creating stream iteration_stream ...
creating stream dna_stream ...
creating stream counter_stream ...
creating stream sum_stream ...
creating stream start_stream ...
creating stream score_stream ...
creating process producer_process ...
creating process pwm_process ...
creating process filter_process ...
creating process consumer_process ...
"C:\Impulse\CoDeveloper2\bin\impulse_s2xml" FPWM.sic > FPWM.xic
Impulse C to XML
Copyright 2003-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
"C:\Impulse\CoDeveloper2\bin\impulse_arch.exe" -aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -std_logic -no
Impulse C HDL Design Generator
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Loading C:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml ...
```

```

Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Cray/RT/bus.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/target.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Cray/technology.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/system.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Xilinx/float.xml ...
Loading FPWM.xic ...
"C:\Impulse\CoDeveloper2\bin\impulse_sm.exe" -g "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -lfl
Stage Master Version 2.7
Copyright by Green Mountain Computing Systems, 2003-2006.
All rights reserved.
Build Jan 19 2007.
Analyzing PWM:
.....Multiple access to pwm reduces minimum rate to 2
..... done 17 blocks.
Analyzing Filter:
..... done 13 blocks.
Generating
Warning: Recursively used variables may reduce pipeline rate in PWM b10 (1 sum)
Generating .. done.Results:
-----
| PWM
|-----
| Block #0 loop:
|   Stages: 28
|   Max. Unit Delay: 0
| Block #1 loop:
|   Stages: 6
|   Max. Unit Delay: 1
| Block #2 loop:
|   Stages: 3
|   Max. Unit Delay: 32
| Block #3:
|   Stages: 1
|   Max. Unit Delay: 32
| Block #4:
|   Stages: 2
|   Max. Unit Delay: 1
| Block #5:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #6 loop:
|   Stages: 3
|   Max. Unit Delay: 32
| Block #7:
|   Stages: 1
|   Max. Unit Delay: 33
| Block #8 loop:
|   Stages: 13
|   Max. Unit Delay: 1
| Block #9:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #10 pipeline:
|   Latency: 7
|   Rate: 6
|   Max. Unit Delay: 32
|   Effective Rate: 192
| Block #11:
|   Stages: 2
|   Max. Unit Delay: 1
| Block #12:
|   Stages: 1
|   Max. Unit Delay: 32
| Block #13:
|   Stages: 1

```



```
|   Max. Unit Delay: 0
| Block #14:
|   Stages: 1
|   Max. Unit Delay: 33
| Block #15:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #16:
|   Stages: 1
|   Max. Unit Delay: 0
|-----
| Operators:
| 4 Adder(s)/Subtractor(s) (5 bit)
| 2 Adder(s)/Subtractor(s) (9 bit)
| 1 Adder(s)/Subtractor(s) (16 bit)
| 7 Adder(s)/Subtractor(s) (32 bit)
| 2 Comparator(s) (2 bit)
| 4 Comparator(s) (8 bit)
| 7 Comparator(s) (32 bit)
| 4 Floating-point Adder(s)/Subtractor(s) (32 bit)
|-----
| Total Stages: 30
| Max. Unit Delay: 33
|-----
| Filter
|-----
| Block #0 loop:
|   Stages: 20
|   Max. Unit Delay: 1
| Block #1:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #2:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #3:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #4 loop:
|   Stages: 13
|   Max. Unit Delay: 1
| Block #5:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #6:
|   Stages: 4
|   Max. Unit Delay: 32
| Block #7:
|   Stages: 4
|   Max. Unit Delay: 1
| Block #8:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #9:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #10:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #11:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #12:
|   Stages: 1
```

```

|   Max. Unit Delay: 0
|-----
| Operators:
|   3 Comparator(s) (2 bit)
|   2 Comparator(s) (3 bit)
|   1 Floating-point Adder(s)/Subtractor(s) (32 bit)
|-----
| Total Stages: 22
| Max. Unit Delay: 32
|-----
Writing output done.
"C:\Impulse\CoDeveloper2\bin\impulse_genvhdl.exe" FPWM.xhw hw/FPWM_comp.vhd
Impulse C RTL Component Generator
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Generating PWM ...
Generating Filter ...
---Software activated---
chmod -R +rw hw
"C:\Impulse\CoDeveloper2\bin\impulse_lib.exe" "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -hwdir
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Loading C:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/Opteron/RT/cpu.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/system.xml ...
Loading FPWM.xic ...
for i in FPWM_sw.c; do cp $i sw; done
for i in FPWM.h; do cp $i sw; done
chmod -R +rw sw

===== Build of target 'build' complete =====

```

FPWM2008

```

===== Building target 'build' in file _Makefile =====
HW_SRC="FPWM2008_hw.c"; \
for i in $HW_SRC; do CPP_INCLUDES="$CPP_INCLUDES -include $i"; done; \
echo | "C:\Impulse\CoDeveloper2\MinGW\bin\gcc" -E -DWIN32 -DIMPULSE_C_SYNTHESIS -DRC_INVOKED -DBYTE="unsigned char" \
"C:\Impulse\CoDeveloper2\bin\impulse_snoot" -Timpulse-c FPWM2008.i FPWM2008.snt
"C:\Impulse\CoDeveloper2\bin\impulse_prep" FPWM2008.snt FPWM2008.pk0
Impulse C Transformations
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
processing FPWM2008.snt...
"C:\Impulse\CoDeveloper2\bin\impulse_porky" -iterate -fold -unused-syms -unused-types -Dmemcpys -const-prop -sc
"C:\Impulse\CoDeveloper2\bin\impulse_porky" -build-arefs FPWM2008.pk1 FPWM2008.pky
"C:\Impulse\CoDeveloper2\bin\impulse_imp" FPWM2008.pky FPWM2008.sic
Impulse C Preprocessor
Copyright 2003-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
---Software activated---
processing FPWM2008.pky...
analyzing co_initialize...
found architecture definition: FPWM2008/cray_rt
analyzing config_FPWM2008...
creating stream threshold_0 ...
creating stream threshold_1 ...
creating stream pwm_0 ...
creating stream pwm_1 ...
creating stream iteration_0 ...
creating stream iteration_1 ...
creating stream dna_0 ...

```

```

creating stream dna_1 ...
creating stream counter_0 ...
creating stream sum_0 ...
creating stream counter_1 ...
creating stream sum_1 ...
creating stream start_0 ...
creating stream score_0 ...
creating stream start_1 ...
creating stream score_1 ...
creating process producer_process ...
creating process PWM0 ...
creating process PWM1 ...
creating process Filter0 ...
creating process Filter1 ...
creating process consumer_process ...
"C:\Impulse\CoDeveloper2\bin\impulse_s2xml" FPWM2008.sic > FPWM2008.xic
Impulse C to XML
Copyright 2003-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
"C:\Impulse\CoDeveloper2\bin\impulse_arch.exe" "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -std_logic -no
Impulse C HDL Design Generator
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Loading C:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Cray/RT/bus.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/target.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Cray/technology.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/system.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Xilinx/float.xml ...
Loading FPWM2008.xic ...
"C:\Impulse\CoDeveloper2\bin\impulse_sm.exe" -g "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -lfloat FPWM2
Stage Master Version 2.7
Copyright by Green Mountain Computing Systems, 2003-2006.
All rights reserved.
Build Jan 19 2007.
Analyzing PWM:
.....Multiple access to matrix reduces minimum rate to 2
..... done 17 blocks.
Analyzing Filter:
..... done 13 blocks.
Generating
Warning: Recursively used variables may reduce pipeline rate in PWM b10 (1 nSum)
Generating .. done.
Results:
|-----
| PWM
|-----
| Block #0 loop:
|   Stages: 28
|   Max. Unit Delay: 0
| Block #1 loop:
|   Stages: 6
|   Max. Unit Delay: 1
| Block #2 loop:
|   Stages: 3
|   Max. Unit Delay: 32
| Block #3:
|   Stages: 1
|   Max. Unit Delay: 32
| Block #4:
|   Stages: 2
|   Max. Unit Delay: 1
| Block #5:
|   Stages: 1

```

```

|   Max. Unit Delay: 0
| Block #6 loop:
|   Stages: 3
|   Max. Unit Delay: 32
| Block #7:
|   Stages: 1
|   Max. Unit Delay: 33
| Block #8 loop:
|   Stages: 13
|   Max. Unit Delay: 1
| Block #9:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #10 pipeline:
|   Latency: 7
|   Rate:    6
|   Max. Unit Delay: 32
|   Effective Rate: 192
| Block #11:
|   Stages: 2
|   Max. Unit Delay: 1
| Block #12:
|   Stages: 1
|   Max. Unit Delay: 32
| Block #13:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #14:
|   Stages: 1
|   Max. Unit Delay: 33
| Block #15:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #16:
|   Stages: 1
|   Max. Unit Delay: 0
|-----
| Operators:
| 4 Adder(s)/Subtractor(s) (5 bit)
| 2 Adder(s)/Subtractor(s) (9 bit)
| 1 Adder(s)/Subtractor(s) (16 bit)
| 7 Adder(s)/Subtractor(s) (32 bit)
| 2 Comparator(s) (2 bit)
| 4 Comparator(s) (8 bit)
| 7 Comparator(s) (32 bit)
| 4 Floating-point Adder(s)/Subtractor(s) (32 bit)
|-----
| Total Stages: 30
| Max. Unit Delay: 33
|-----
| Filter
|-----
| Block #0 loop:
|   Stages: 20
|   Max. Unit Delay: 1
| Block #1:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #2:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #3:
|   Stages: 1
|   Max. Unit Delay: 0

```

```

| Block #4 loop:
|   Stages: 13
|   Max. Unit Delay: 1
| Block #5:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #6:
|   Stages: 4
|   Max. Unit Delay: 32
| Block #7:
|   Stages: 4
|   Max. Unit Delay: 1
| Block #8:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #9:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #10:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #11:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #12:
|   Stages: 1
|   Max. Unit Delay: 0
|-----
| Operators:
| 3 Comparator(s) (2 bit)
| 2 Comparator(s) (3 bit)
| 1 Floating-point Adder(s)/Subtractor(s) (32 bit)
|-----
| Total Stages: 22
| Max. Unit Delay: 32
|-----
Writing output done.
"C:\Impulse\CoDeveloper2\bin\impulse_genvhdl.exe" FPWM2008.xhw hw/FPWM2008_comp.vhd
Impulse C RTL Component Generator
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Generating PWM ...
Generating Filter ...
---Software activated---
chmod -R +rw hw
"C:\Impulse\CoDeveloper2\bin\impulse_lib.exe" "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -hwdirhw -files
Impulse C Software Interface Generator
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Loading C:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/Opteron/RT/cpu.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/system.xml ...
Loading FPWM2008.xic ...
for i in FPWM2008_sw.c; do cp $i sw; done
for i in FPWM2008.h; do cp $i sw; done
chmod -R +rw sw

===== Build of target 'build' complete =====

```

FPWMI

```

===== Building target 'build' in file _Makefile =====
HW_SRC="FPWMI_hw.c"; \
for i in $HW_SRC; do CPP_INCLUDES="$CPP_INCLUDES -include $i"; done; \

```

```

echo | "C:\Impulse\CoDeveloper2\MinGW\bin\gcc" -E -DWIN32 -DIMPULSE_C_SYNTHESIS -DRC_INVOKED -DBYTE="unsigned ch
"C:\Impulse\CoDeveloper2\bin\impulse_snoot" -Timpulse-c FPWMI.i FPWMI.snt
"C:\Impulse\CoDeveloper2\bin\impulse_prep" FPWMI.snt FPWMI.pk0Impulse C Transformations
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
processing FPWMI.snt...
"C:\Impulse\CoDeveloper2\bin\impulse_porky" -iterate -fold -unused-syms -unused-types -Dmempys -const-prop -sc
"C:\Impulse\CoDeveloper2\bin\impulse_porky" -build-arefs FPWMI.pk1 FPWMI.pky
"C:\Impulse\CoDeveloper2\bin\impulse_imp" FPWMI.pky FPWMI.sicImpulse C Preprocessor
Copyright 2003-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
---Software activated---
processing FPWMI.pky...
analyzing co_initialize...
found architecture definition: FPWMI/cray_rt
analyzing config_FPWMI...
creating stream threshold_stream ...
creating stream pwm_stream ...
creating stream iteration_stream ...
creating stream dna_stream ...
creating stream counter_stream ...
creating stream sum_stream ...
creating stream start_stream ...
creating stream score_stream ...
creating process producer_process ...
creating process pwm_process ...
creating process filter_process ...
creating process consumer_process ...
"C:\Impulse\CoDeveloper2\bin\impulse_s2xml" FPWMI.sic > FPWMI.xic
Impulse C to XML
Copyright 2003-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
"C:\Impulse\CoDeveloper2\bin\impulse_arch.exe" "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -std_
Impulse C HDL Design Generator
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Loading C:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Cray/RT/bus.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/target.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Cray/technology.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/system.xml ...
Loading FPWMI.xic ...
"C:\Impulse\CoDeveloper2\bin\impulse_sm.exe" -g "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" FPW
Stage Master Version 2.7
Copyright by Green Mountain Computing Systems, 2003-2006.
All rights reserved.
Build Jan 19 2007.
Analyzing PWM:
.....Multiple access to pwm reduces minimum rate to 2
..... done 17 blocks.
Analyzing Filter:
..... done 14 blocks.
Generating
Warning: Recursively used variables may reduce pipeline rate in PWM b10 (1 sum)
Generating .. done.
Results:
|-----
| PWM
|-----
| Block #0 loop:
|   Stages: 25
|   Max. Unit Delay: 0

```

```
| Block #1 loop:
|   Stages: 6
|   Max. Unit Delay: 1
| Block #2 loop:
|   Stages: 3
|   Max. Unit Delay: 32
| Block #3:
|   Stages: 1
|   Max. Unit Delay: 32
| Block #4:
|   Stages: 2
|   Max. Unit Delay: 1
| Block #5:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #6 loop:
|   Stages: 3
|   Max. Unit Delay: 32
| Block #7:
|   Stages: 1
|   Max. Unit Delay: 33
| Block #8 loop:
|   Stages: 10
|   Max. Unit Delay: 1
| Block #9:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #10 pipeline:
|   Latency: 4
|   Rate: 2
|   Max. Unit Delay: 32
|   Effective Rate: 64
| Block #11:
|   Stages: 2
|   Max. Unit Delay: 1
| Block #12:
|   Stages: 1
|   Max. Unit Delay: 32
| Block #13:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #14:
|   Stages: 1
|   Max. Unit Delay: 33
| Block #15:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #16:
|   Stages: 1
|   Max. Unit Delay: 0
|-----
| Operators:
| 4 Adder(s)/Subtractor(s) (5 bit)
| 2 Adder(s)/Subtractor(s) (9 bit)
| 1 Adder(s)/Subtractor(s) (16 bit)
| 11 Adder(s)/Subtractor(s) (32 bit)
| 2 Comparator(s) (2 bit)
| 4 Comparator(s) (8 bit)
| 7 Comparator(s) (32 bit)
|-----
| Total Stages: 27
| Max. Unit Delay: 33
|-----
| Filter
```

```

-----
| Block #0 loop:
|   Stages: 15
|   Max. Unit Delay: 1
| Block #1:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #2:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #3:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #4 loop:
|   Stages: 8
|   Max. Unit Delay: 1
| Block #5:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #6:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #7:
|   Stages: 1
|   Max. Unit Delay: 32
| Block #8:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #9:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #10:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #11:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #12:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #13:
|   Stages: 1
|   Max. Unit Delay: 0
-----
| Operators:
| 1 Adder(s)/Subtractor(s) (32 bit)
| 3 Comparator(s) (2 bit)
| 2 Comparator(s) (3 bit)
| 2 Comparator(s) (32 bit)
-----
| Total Stages: 17
| Max. Unit Delay: 32
-----

```

Writing output done.

"C:\Impulse\CoDeveloper2\bin\impulse_genvhdl.exe" FPWMI.xhw hw/FPWMI_comp.vhd

Impulse C RTL Component Generator

Copyright 2002-2007, Impulse Accelerated Technologies, Inc.

All rights reserved.

Generating PWM ...

Generating Filter ...

---Software activated---

chmod -R +rw hw

"C:\Impulse\CoDeveloper2\bin\impulse_lib.exe" "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -hwdir

Impulse C Software Interface Generator

Copyright 2002-2007, Impulse Accelerated Technologies, Inc.


```

All rights reserved.
Loading C:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/Opteron/RT/cpu.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/system.xml ...
Loading FPWMI.xic ...
for i in FPWMI_sw.c; do cp $i sw; done
for i in FPWMI.h; do cp $i sw; done
chmod -R +rw sw

===== Build of target 'build' complete =====

```

FPWM2008i

```

===== Building target 'build' in file _Makefile =====
HW_SRC="FPWM2008i_hw.c"; \
for i in $HW_SRC; do CPP_INCLUDES="$CPP_INCLUDES -include $i"; done; \
echo | "C:\Impulse\CoDeveloper2\MinGW\bin\gcc" -E -DWIN32 -DIMPULSE_C_SYNTHESIS -DRC_INVOKED -DBYTE="unsigned char" -DWOR
"C:\Impulse\CoDeveloper2\bin\impulse_snoot" -Timpulse-c FPWM2008i.i FPWM2008i.snt
"C:\Impulse\CoDeveloper2\bin\impulse_prep" FPWM2008i.snt FPWM2008i.pk0
Impulse C Transformations
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
processing FPWM2008i.snt...
"C:\Impulse\CoDeveloper2\bin\impulse_porky" -iterate -fold -unused-syms -unused-types -Dmemcpy -const-prop -scalarize
"C:\Impulse\CoDeveloper2\bin\impulse_porky" -build-arefs FPWM2008i.pk1 FPWM2008i.pky
"C:\Impulse\CoDeveloper2\bin\impulse_imp" FPWM2008i.pky FPWM2008i.sic
Impulse C Preprocessor
Copyright 2003-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
---Software activated---
processing FPWM2008i.pky...
analyzing co_initialize...
found architecture definition: FPWM2008i/cray_rt
analyzing config_FPWM2008i...
creating stream threshold_0 ...
creating stream threshold_1 ...
creating stream pwm_0 ...
creating stream pwm_1 ...
creating stream iteration_0 ...
creating stream iteration_1 ...
creating stream dna_0 ...
creating stream dna_1 ...
creating stream counter_0 ...
creating stream sum_0 ...
creating stream counter_1 ...
creating stream sum_1 ...
creating stream start_0 ...
creating stream score_0 ...
creating stream start_1 ...
creating stream score_1 ...
creating process producer_process ...
creating process PWM0 ...
creating process PWM1 ...
creating process Filter0 ...
creating process Filter1 ...
creating process consumer_process ...
"C:\Impulse\CoDeveloper2\bin\impulse_s2xml" FPWM2008i.sic > FPWM2008i.xic
Impulse C to XML
Copyright 2003-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Build May 7 2007.
"C:\Impulse\CoDeveloper2\bin\impulse_arch.exe" "-a:C:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -std_logic -no

```

```

Impulse C HDL Design Generator
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Loading C:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Cray/RT/bus.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/target.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/VHDL/Cray/technology.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/system.xml ...
Loading FPWM2008i.xic ...
"C:\Impulse\CoDeveloper2\bin\impulse_sm.exe" -g "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" FPW
Stage Master Version 2.7
Copyright by Green Mountain Computing Systems, 2003-2006.
All rights reserved.
Build Jan 19 2007.
Analyzing PWM:
.....Multiple access to matrix reduces minimum rate to 2
..... done 17 blocks.
Analyzing Filter:
..... done 14 blocks.
Generating
Warning: Recursively used variables may reduce pipeline rate in PWM b10 (1 nSum)
Generating .. done.
Results:
|-----|
| PWM |
|-----|
| Block #0 loop: |
|   Stages: 25 |
|   Max. Unit Delay: 0 |
| Block #1 loop: |
|   Stages: 6 |
|   Max. Unit Delay: 1 |
| Block #2 loop: |
|   Stages: 3 |
|   Max. Unit Delay: 32 |
| Block #3: |
|   Stages: 1 |
|   Max. Unit Delay: 32 |
| Block #4: |
|   Stages: 2 |
|   Max. Unit Delay: 1 |
| Block #5: |
|   Stages: 1 |
|   Max. Unit Delay: 0 |
| Block #6 loop: |
|   Stages: 3 |
|   Max. Unit Delay: 32 |
| Block #7: |
|   Stages: 1 |
|   Max. Unit Delay: 33 |
| Block #8 loop: |
|   Stages: 10 |
|   Max. Unit Delay: 1 |
| Block #9: |
|   Stages: 1 |
|   Max. Unit Delay: 0 |
| Block #10 pipeline: |
|   Latency: 4 |
|   Rate: 2 |
|   Max. Unit Delay: 32 |
|   Effective Rate: 64 |
| Block #11: |
|   Stages: 2 |
|   Max. Unit Delay: 1 |
| Block #12: |

```

```
|   Stages: 1
|   Max. Unit Delay: 32
| Block #13:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #14:
|   Stages: 1
|   Max. Unit Delay: 33
| Block #15:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #16:
|   Stages: 1
|   Max. Unit Delay: 0
|-----
| Operators:
| 4 Adder(s)/Subtractor(s) (5 bit)
| 2 Adder(s)/Subtractor(s) (9 bit)
| 1 Adder(s)/Subtractor(s) (16 bit)
| 11 Adder(s)/Subtractor(s) (32 bit)
| 2 Comparator(s) (2 bit)
| 4 Comparator(s) (8 bit)
| 7 Comparator(s) (32 bit)
|-----
| Total Stages: 27
| Max. Unit Delay: 33
|-----
| Filter
|-----
| Block #0 loop:
|   Stages: 15
|   Max. Unit Delay: 1
| Block #1:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #2:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #3:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #4 loop:
|   Stages: 8
|   Max. Unit Delay: 1
| Block #5:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #6:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #7:
|   Stages: 1
|   Max. Unit Delay: 32
| Block #8:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #9:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #10:
|   Stages: 1
|   Max. Unit Delay: 1
| Block #11:
|   Stages: 1
```

```

|   Max. Unit Delay: 0
| Block #12:
|   Stages: 1
|   Max. Unit Delay: 0
| Block #13:
|   Stages: 1
|   Max. Unit Delay: 0
|-----
| Operators:
|   1 Adder(s)/Subtractor(s) (32 bit)
|   3 Comparator(s) (2 bit)
|   2 Comparator(s) (3 bit)
|   2 Comparator(s) (32 bit)
|-----
| Total Stages: 17
| Max. Unit Delay: 32
|-----
Writing output done.
"C:\Impulse\CoDeveloper2\bin\impulse_genvhdl.exe" FPWM2008i.xhw hw/FPWM2008i_comp.vhd
Impulse C RTL Component Generator
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Generating PWM ...
Generating Filter ...
---Software activated---
chmod -R +rw hw
"C:\Impulse\CoDeveloper2\bin\impulse_lib.exe" "-aC:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml" -hwdi
Impulse C Software Interface Generator
Copyright 2002-2007, Impulse Accelerated Technologies, Inc.
All rights reserved.
Loading C:\Impulse\CoDeveloper2\Architectures\cray_xd1_vhdl.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/Opteron/RT/cpu.xml ...
Loading C:/Impulse/CoDeveloper2/Architectures/Cray/system.xml ...
Loading FPWM2008i.xic ...
for i in FPWM2008i_sw.c; do cp $i sw; done
for i in FPWM2008i.h; do cp $i sw; done
chmod -R +rw sw

===== Build of target 'build' complete =====

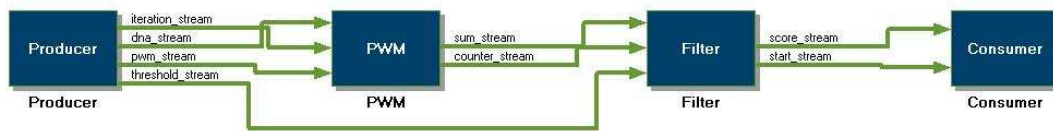
```

Appendix D

Architecture Block Diagrams

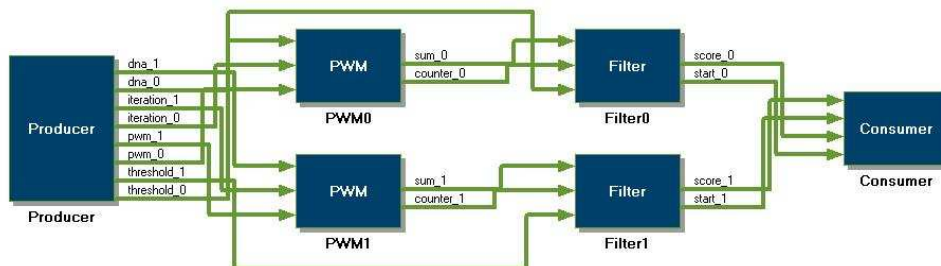
FPWM/FPWMi

Architecture: FPWM



FPWM2008/FPWM2008i

Architecture: FPWM2008



Bibliography

- [1] Amd homepage. [<http://www.amd.com/>]. Accessed before: 11/29/2007.
- [2] AMD. Amd opteron processor datasheet. [http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23932.pdf], 2007.
- [3] Cray homepage. [<http://www.cray.com/>]. Accessed before: 11/29/2007.
- [4] Inc. Cray. Cray xd1 datasheet. [http://www.cray.com/downloads/Cray_XD1_Datasheet.pdf], 2005.
- [5] Finn Drabløs. Specification of the pwm module, 2007.
- [6] Mohamed Abouellail Tarek El-Ghazawi Esam El-Araby, Mohamed Taher and Gregory B. Newby. Comparative analysis of high level programming for re-configurable computers: Methodology and empirical study. In *Programmable Logic, 2007. SPL '07. 2007 3rd Southern Conference on*, pages 99–106, 2007. [<http://ieeexplore.ieee.org/>].
- [7] Fpgas on wikipedia. [http://en.wikipedia.org/wiki/Field-programmable_gate_array]. Accessed before: 11/29/2007.
- [8] Øyvind Bø Syrstad Lars Andreas Eidsheim Osman Abul Geir Kjetil Sandve, Mag-nar Nedland and Finn Drabløs. Accelerating motif discovery: Motif matching on parallel hardware. In *Algorithms in Bioinformatics*, 2006. [<http://springerlink.com/>].
- [9] Per Andreas Gulbrandsen. Rekonfigurerbar maskinvare som applikasjonsakselerator ved søk i dna. MSc thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, June 2007.
- [10] Elisabeth Linvåg. High level description for fpga hardware acceleration of dna motif identification. Master's thesis, Norwegian University of Science and Technology, Department of Computer and Information Science, December 2007.
- [11] Norgrid homepage, musculus introduction. [<http://norggrid.ntnu.no/norggrid/Xd1Introduction>]. Accessed before: 11/29/2007.
- [12] David Pellerin and Scott Thibault. *Practical FPGA Programming in C*. Prentice Hall PTR, 2005.

- [13] Pwm on wikipedia. [http://en.wikipedia.org/wiki/Position-specific_scoring_matrix]. Accessed before: 11/29/2007.
- [14] Xilinx. Xilinx virtex-ii pro datasheet. [http://www.xilinx.com/support/documentation/data_sheets/ds083.pdf], 2007.