CAPACITIVE FLOW METER REALTIME PROCESSING USING CROSS CORRELATION HDL DESIGN METHODOLOGY AND PERFORMANCE

A Thesis

Submitted to the Faculty of Graduate Studies and Research

In Partial Fulfilment of the Requirements

for the Degree of

Master of Applied Science

in Electronic Systems Engineering

University of Regina

by

Peter Condie Nell

Regina, Saskatchewan

March 2015

UNIVERSITY OF REGINA

FACULTY OF GRADUATE STUDIES AND RESEARCH

SUPERVISORY AND EXAMINING COMMITTEE

Peter Condie Nell, candidate for the degree of Master of Applied Science in Electronic Systems Engineering**,** has presented a thesis titled, ***Capacitive Flow Meter Realtime Processing Using Cross Correlation HDL Design Methodology and Performance,*** in an oral examination held on November 14, 2014. The following committee members have found the thesis acceptable in form and content, and that the candidate demonstrated satisfactory knowledge of the subject material.

External Examiner:	Dr. Craig Gelowitz, Software Systems Engineering
Supervisor:	Dr. Thomas Conroy, Electronic Systems Engineering
Committee Member:	Dr. Paul Laforge, Electronic Systems Engineering
Committee Member:	Dr. Lei Zhang, Industrial Systems Engineering
Committee Member:	Dr. Joseph Toth, Adjunct
Chair of Defense:	Dr. Andrei Volodin, Department of Mathematics & Statistics

Abstract

Accurate low cost flow rate measurements of multiphase varying rate flows would be beneficial to the oil and gas industries in Saskatchewan.

This thesis provides a design methodology, implementation and analysis of the performance characteristics for data processing in a low cost real-time flow-rate meter using embedded hardware and cross-correlation. While pure serial and pure parallel implementations are possible, this is not the most efficient solution. A combination of these two methods combined with pipelining was implemented as the most efficient HDL to be compared to a software solution. This implementation balanced the hardware between minimum silicon and maximum throughput in an efficient manner.

The results of the comparison indicate that the hardware solution performs at a significantly greater level than the software equivalent. Additional optimizations of both hardware and software could change this ratio to some extent; however the performance ratio should remain strongly in favor of the hardware solution in terms of performance.

Acknowledgement

I would like to acknowledge the Faculty of Graduate Studies and Research of the University of Regina and the Faculty of Engineering of the University of Regina for their time and support as well as funding that allowed me to continue my studies.

I am grateful to my supervisor, Dr. Thomas Conroy, for his support and guidance throughout my studies.

I would like to thank Dave Duguid for his support and use of his lab space, as well as Brian Fitzgerald, Kevin Knutson, and Sophit Pongpun for our numerous discussions and brain storming sessions which have helped me during my studies.

I am thankful to CMC Microsystems and NSERC, for their resources and funding used during this work.

Post Defence Acknowledgement

I would like to thank everyone on the committee for its participation in my defence. I would like to thank Dr. Andrei Volodin for taking the time to Chair the defence. The external examiner Dr. Craig M Gelowitz for his time, attention and suggestions which helped to improve this document; as well as Dr. Paul Laforge, Dr. Lei Zhang and Mr. Joe Toth for their similar contributions.

Dedication

This thesis is dedicated to my parents, Bob and Colleen, to whom I am forever grateful for their understanding and support.

I would also like to dedicate this thesis to the memory of Ken Runtz, whose ideas along with those of Dr. Conroy lead to this research investigation.

List of Tables

List of Figures

List of Equations

List of Abbreviations

ASIC	Application Specific Integrated Circuit
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
GPM	Gallons per Minute
MHz	Mega Hertz
PLB	Programmable Logic Block
RAM	Random Access Memory
SRAM	Static Random Access Memory
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits

1. Introduction

The oil and gas sector forms a large portion of the Saskatchewan economy, approximately 21.3% [1]. Thus improvements that could impact the efficiency of these industries can have a relatively large impact on the overall economy.

A particular area where improved flow meter technology would be of great use is in oil field storage batteries. While it is trivial to wait for a known time and then measure the levels found in the storage tank to calculate flow rate. This only gives you the flow rate of the overall field, and not the contributions made by each well head. This will allow managers to monitor the state of their fields and production rates remotely in real time. This will provide information for them to make decisions more quickly to optimize production and to make better estimations on profits.

A low cost flow meter, in addition to giving the flow rates of each well head for monitoring purposes, could be used to optimize which wells should be operated at what rate at any given time. This is of importance due to the high amount of water, or the water cut of each well flow; and is of particular importance in Saskatchewan due to the high percentages of water found here, sometimes on the order of 95% water [2]. Separating the water from the oil and controlling its disposal is an expensive and time consuming process. If this ratio of water could be reduced by even a few percentage points, there would be a large cost savings to industry. A solution to measure these flow rates is therefore beneficial. This improvement in efficiency must provide an overall cost savings including the additional cost of the flow meters and monitoring, and thus a low

cost solution is of the upmost importance. As a result, the goal of this research was to develop a low-cost flow meter that could be made available to industry to improve the efficiency of the oil extraction process in high water-cut areas.

This research is particularly of value to the oil and gas industry of Saskatchewan as it will allow the industry here to operate its fields in a more efficient manner, resulting in improved performance of fields. This in turn leads to a more competitive and profitable industry in the province. Additionally, efficient cross correlation could be applied to other fields that make use of correlation such as image processing or communications for example.

There are a variety of techniques that can be used to measure the flow rate of a multiphase system. However many of these are very expensive and some involve the use of radioactive sources [3]. Impedance measurement techniques are a relatively cheap solution, involving no radioactive sources and no requirement to interrupt the flow inside the pipe. In addition to the cost advantages, the impedance measurement technique can be combined with water-cut measurements in the same instrument. This can be done by using two separate meters along the same section of pipe with a known fixed distance between the water-cut meters. This allows the use of cross-correlation of the signals generated by the two meters to determine flow rates. However this measurement method involves a very fast sensor to avoid measurement uncertainty [3].

Speed is also required simply for the sheer number of calculations that are required to

accurately cross-correlate the data. With windows sizes in the range of 100-150; this window is then offset or slid across the data sets to find the point of maximum correlation.

The flow from an oil well, with a pump jack, is dynamic and can vary from 0-55 gpm or a fluid velocity of 0-2.33 ft/s in a 3-inch pipe; the flow will spurt and stop as the jack moves up and down. The fluid flow rate also varies with the weight and density of the fluid, the length of the stroke, stroke rate, and the depth of the well, so these cannot be predetermined. The instantaneous flow rate needs to be monitored continuously at very small time steps and integrated to obtain accurate volume measurements rather than estimations. A sample rate of 1000 Hz was therefore chosen for the sensor, this will result in approximately one million correlations per second, and thus a high performance solution is required.

In addition to performance concerns, cost is also a driving factor in industry. Existing flow meters cost on the orders of thousands, or tens of thousands of dollars. Some of these meters that use radioactive methods can cost hundreds of thousands. Partly due to these costs, most wells in Saskatchewan are not equipped with flow meters. Additionally the operating environment produces a lot of wear on the devices. Some of these sources of wear include corrosive chemicals, abrasion from fluid and sand, as well as thermal cycling over time. These sources of wear shorten the lifespan of devices, and thus devices need replacing at fairly regular intervals. Therefore in addition to the high performance requirements, a low cost solution is also needed.

1.1 Hardware Processing

The Compact Oxford English Dictionary [4] defines hardware as "the machines, wiring, and other physical components of a computer." Now while everything involved in electronics technically uses hardware, for the purposes of execution of algorithms or tasks, hardware generally has a different meaning. In this thesis hardware refers to base hardware performing the required tasks specifically and directly without a general purpose processor or central processing unit (CPU) involved. This means that hardware is designed to perform a specific task and is optimized for this one specific task alone.

This thesis will focus on using available commercial parts to accomplish this task. While hardware can be designed and implemented inside an Application Specific Integrated Circuit (ASIC), this would add a significant cost increase compared with using off the shelf components. The hardware equivalent to an off the shelf processor is the Field Programmable Gate Array (FPGA). These devices allow hardware operations to be defined inside their programmable logic after the chip has been manufactured. This allows for the hardware developer to pre-purchase parts for their design in the same way that a software developer does for a software design.

Through the flexibility of the FPGA architecture, the designer is then free to focus on the design of their specific solution without having to design all of the support architecture as well. Design of the hardware solution can then be done through use of a Hardware Design Language (HDL) solution. The two most common HDL languages are the Virtual Hardware Design Language (VHDL) and Verilog. While each of these languages has

their own advantages and disadvantages, in general they are interchangeable and each can be used to produce identical hardware. The choice for this thesis was to use VHDL.

1.2 Available hardware technologies

Hardware solutions can be implemented directly by assembling prebuilt parts as needed to perform the required function. However this can be both costly and time consuming. The most common ways to implement complex hardware are FPGAs, Complex Programmable Logic Controllers (CPLDs), and ASICs.

The choice for this thesis is the FPGA. Inside this family of products there is a large variety of options available [5]. The FPGA implementation architectures can be broken down into three categories that are currently in use in today.

1.2.1 Anti-fuse architectures

These architectures use physical connections or lack of connections to program the FPGA. Devices with this architecture are therefore one-time programmable. This results in a stable architecture regardless of power states or in general, time. These devices are invaluable in high criticality applications such as aerospace and automotive. They have the benefit of being live at power-up, as well as being resistant to harsh environmental conditions such as temperature and radiation, as well as using very little power. These architectures are generally less space efficient, and in general have a smaller number of available gates for the same footprint. Additionally they operate at lower performance levels than other architectures due to reduced logic density and longer signal propagation times. The general configuration gate control method can be seen in Figure 1.1.

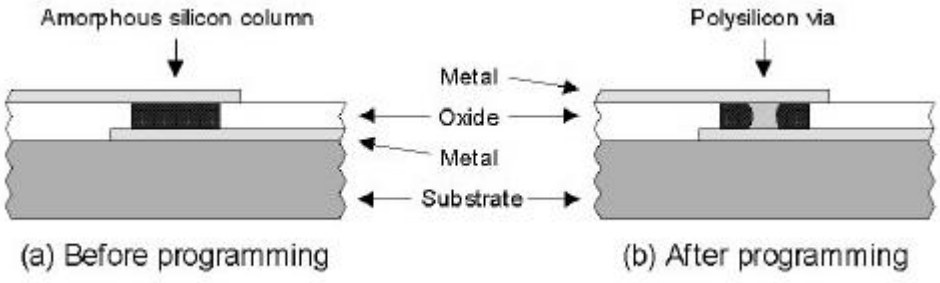

Figure 1.1 Antifuse Gate

1.2.2 Flash based configuration architectures

These architectures store configuration information with flash memory. These architectures are a mix of the benefits and disadvantages of anti-fuse and SRAM architectures. They are capable of being reprogrammed, albeit at a much slower speed than SRAM devices and yet they maintain their configuration through power cycles. This provides the capability of being live at power-up in a similar manner as anti-fuse devices. These architectures are in general in between anti-fuse and SRAM architectures in terms of performance and power use. In terms of power use however, they are much closer to the anti-fuse devices. The general configuration gate control method can be seen in Figure 1.2.

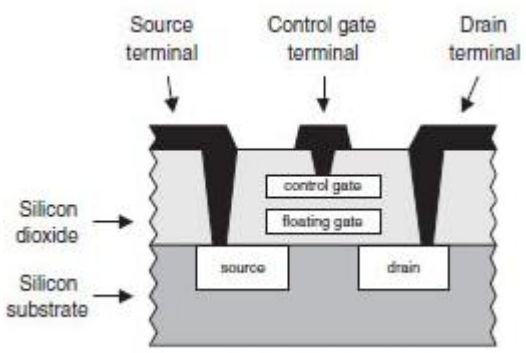

Figure 1.2 Flash Gate

1.2.3 SRAM based configuration architectures

These architectures are the performance winners of the three architectures. Devices based on SRAM can be reprogrammed very quickly, offer the highest performance of the available devices, as well as the highest density of logic. Some devices also have an additional reprogrammable feature that flash architectures do not, partial reprogramability. These types of devices can also be reprogrammed while the device is operating without interfering with the operation of the sections not being reprogrammed. All this performance comes at a cost however, the most telling of which is the lack of configuration retention after power loss and high power consumption. Any time the device is power cycled, it must be reprogrammed. This prevents the device from being live at power-up, as well as requiring additional support hardware to program the device in system. Its large power consumption must also be taken into account, as well as the additional thermal loading for heat dissipation. The general configuration gate control method can be seen in Figure 1.3.

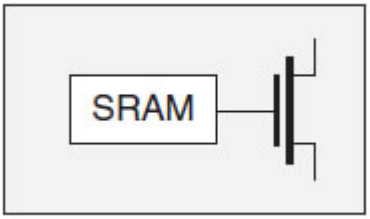

Figure 1.3 SRAM gate

1.3 Software Processing

Software in general, is used to describe a series of basic instructions or operations on a fixed set of hardware that perform a more complex task. These instructions are executed by the processor or CPU that can perform these basic instructions at a very high rate. Because these basic instructions can be combined to perform complex tasks, processors are very flexible and adaptable. The drawback of this versatility is generally performance, as well as the ability to do multiple things concurrently. Concurrent task execution can be simulated by having the processor jump from one serial task to another, only performing parts of each task, before returning to the beginning of the cycle. And while this does allow a process to perform multiple concurrent tasks from a high level prospective, the total performance of the processor has not increased, and additionally, the overall performance on each task has been decreased. An example of the internal architecture of a CPU; in this case the Z80; used here for an example due to its relatively simple design; can be seen in Figure 1.4.

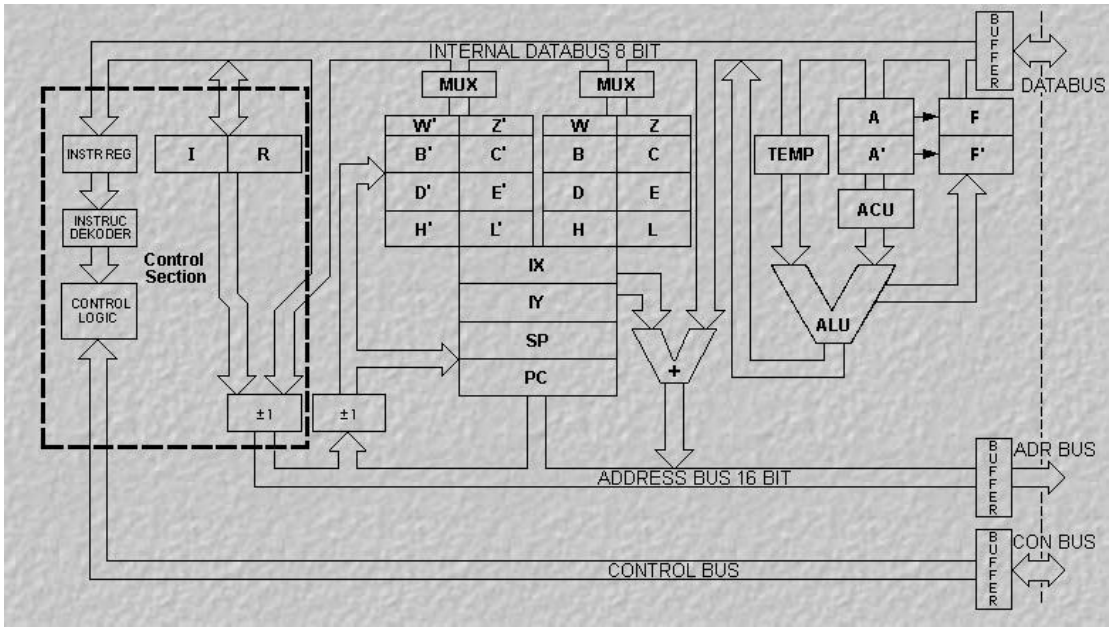

Figure 1.4 Basic Processor Architecture [6]

This basic hardware configuration can then execute a series of basic instructions that are defined using operation codes stored in memory; referred to as software. These instructions are fetched from address space and executed by the CPU controller. Each of these instructions can take a different number of cycles to execute, and this will vary from architecture to architecture. An example of how the Z80 architecture executes instructions can be seen in Figure 1.5.

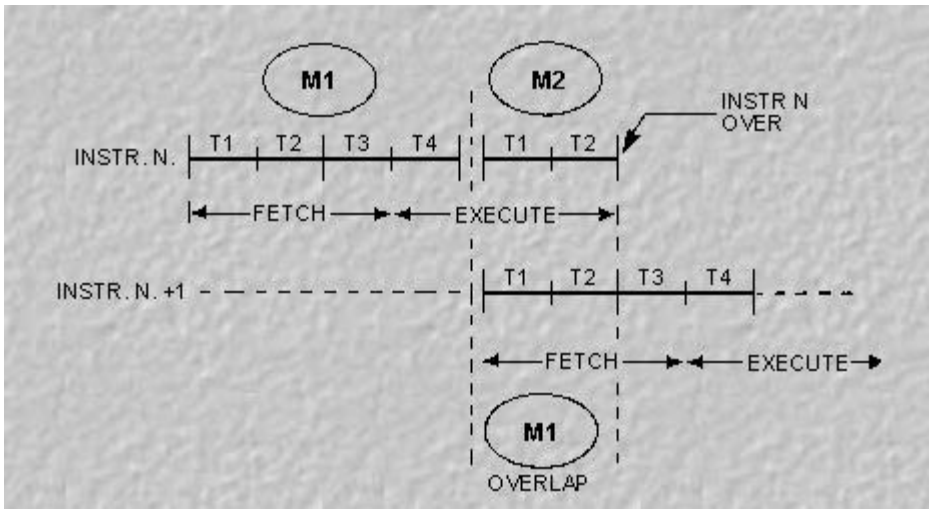

Figure 1.5 Processor Operation Sequence [6]

1.4 Implementation overview

Intel 8086 devices were considered based on their performance and cost. As devices move further away from cutting edge performance, their price tends to drop dramatically. As can be seen in [7], the more cost efficient CPUs tend to be of older generations. Even then, reasonably performing x86 processors cost $50 to $200 US dollars. In contrast to this, reasonably performing FPGAs such as the Xilinx Spartan 3 can be purchased for as little as $7-8 dollars. Or, as used for this research a more powerful Virtex II Pro FPGA, can be purchased for ~66 dollars from Digi-key as of 2014.

In addition to this, the CPUs require other additional support hardware such as memory and IO drivers, which are included internally to the FPGA, further improving the cost advantage of the FPGA implementation.

For this thesis the hardware was implemented using an FPGA and synthesised using Xilinx ISE, and software was written in the 'C' programming language using an x86 compiler. An image of the FPGA development board used in this research can be seen in Figure 1.6. The specific architecture implementation of FPGA chosen was SRAM based. Specifically the FPGA used was a Xilinx Virtex-II Pro. Xilinx FPGAs were used as a result of their availability for educational purposes at the UofR. This high performance FPGA can then be compared technically equivalent high performance CPU solutions. The computer running the ‘C’ code was an Intel Pentium 4 at 1Ghz**.** These two devices were current at approximately the same point in time around 2000-2001, and are suitable for a general comparison of hardware/software performance of the same level of

technology.

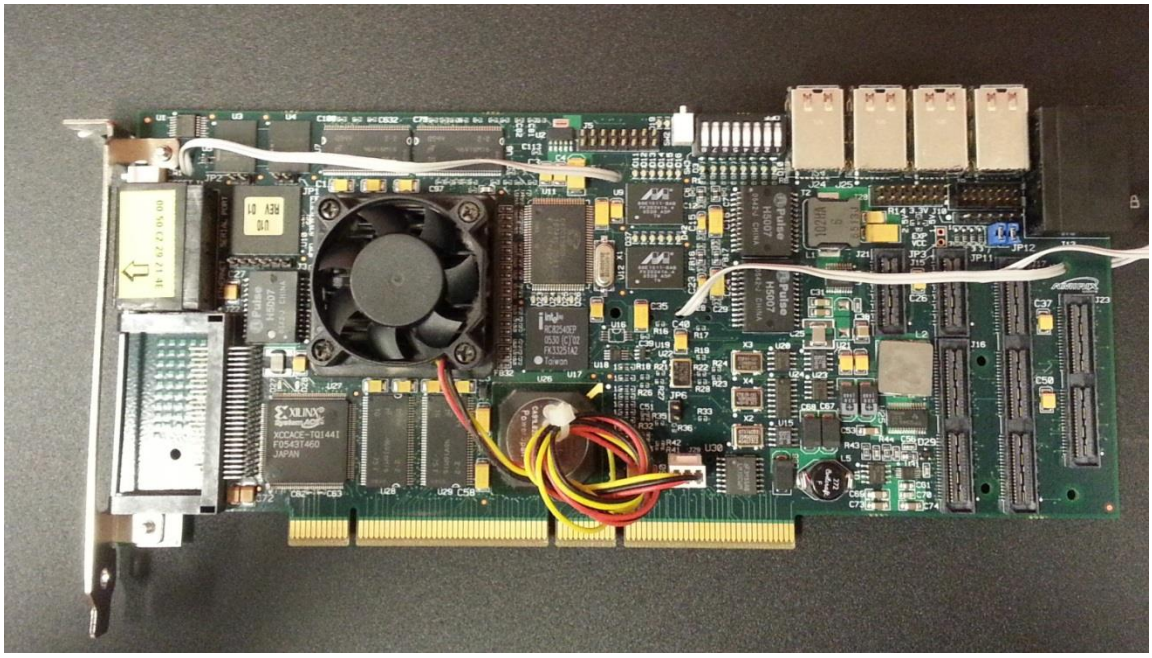

Figure 1.6 VirtexII Pro Development Board (original in colour)

Data from real devices was used for the testing of the implemented solution. With the source of the data was acquired from a pair of NuFlo^{TM} (see Figure 1.7) water cut meters used in a test environment as can be seen in Figure 1.8. The two meters were spaced at a known distance apart along the pipe and their signals measured using an analog to digital converter circuit to generate two data sets of 12 bit values representing water-cut (fluid dielectric). These data sets could then be cross-correlated to determine flow rate through the pipe.

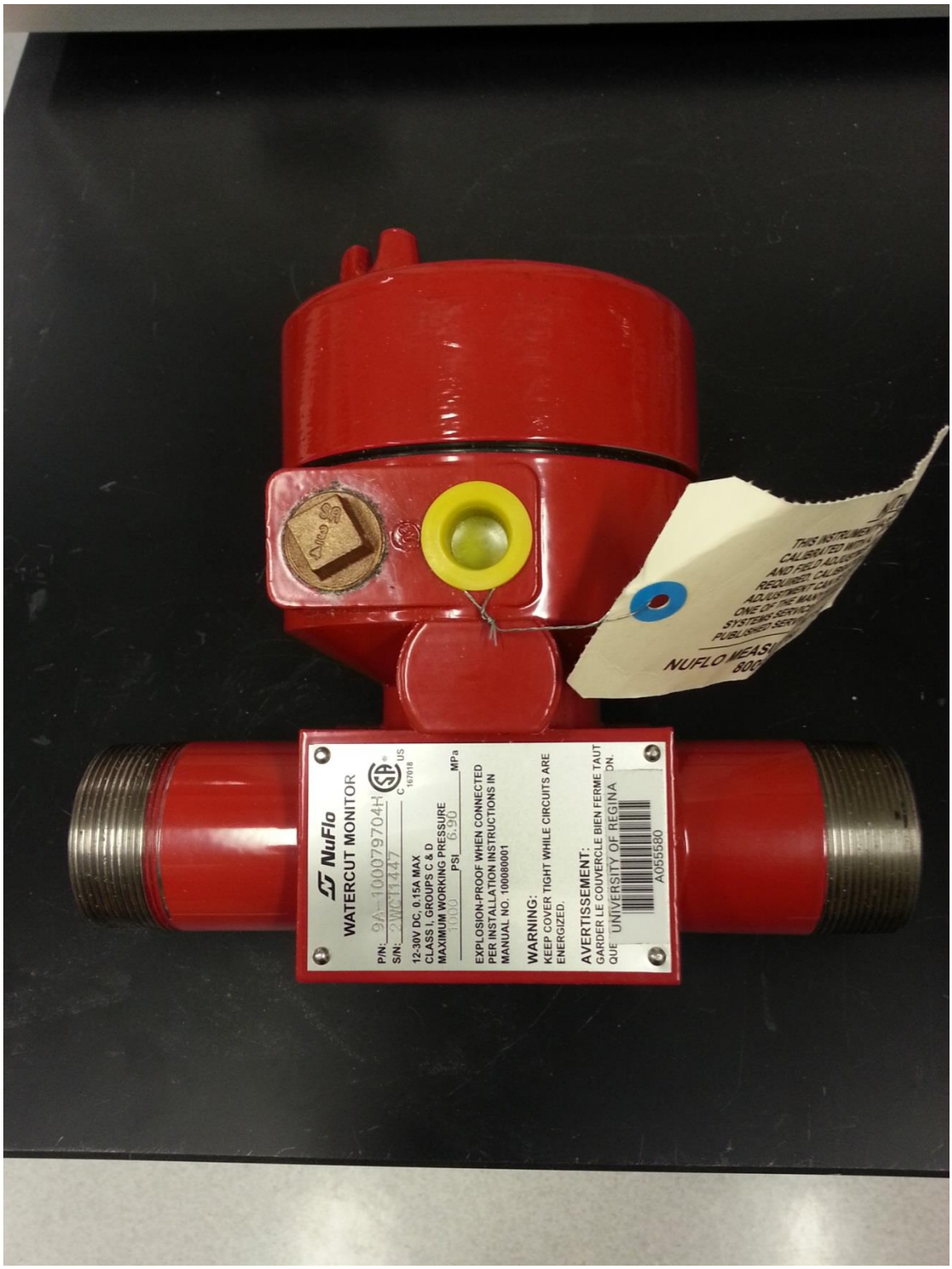

Figure 1.7 NuFlo Water Cut Meter (original in colour)

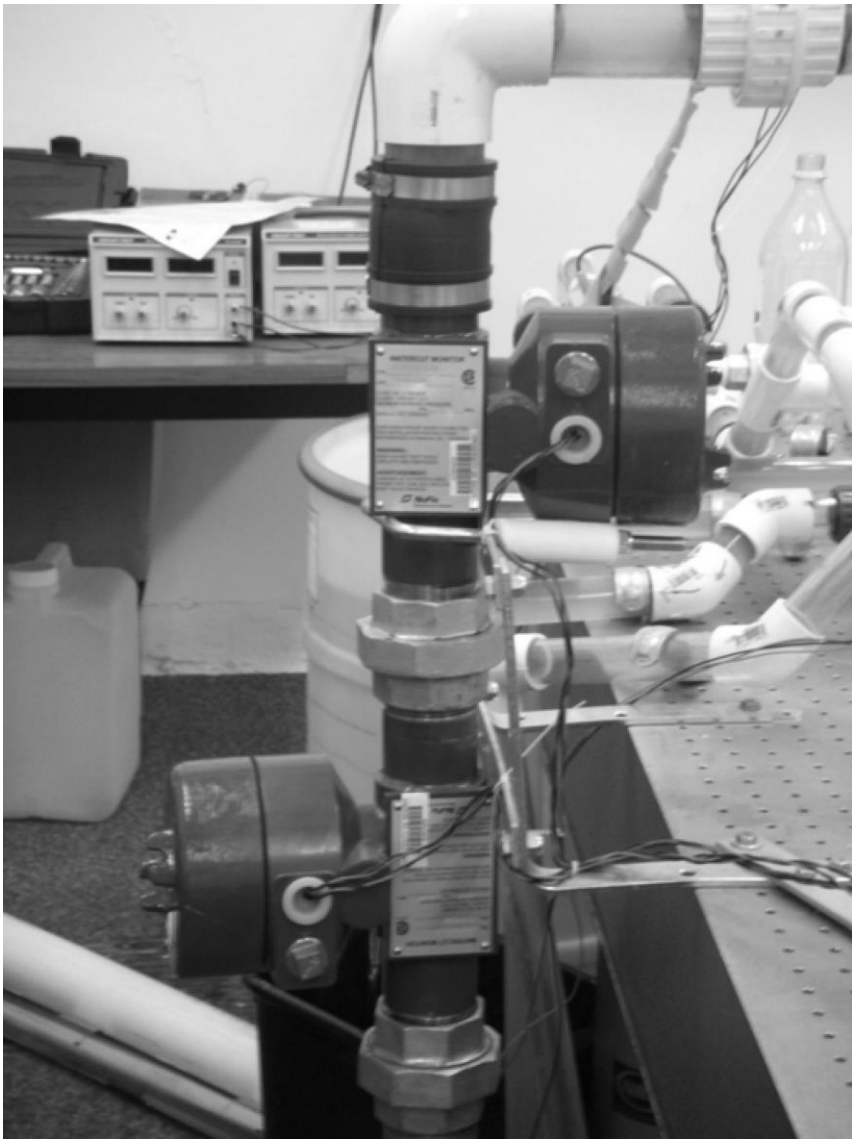

Figure 1.8 Test Environment

1.5 Hardware implementation methodology

The hardware design methodology used for the design of the FPGA implementation involves breaking the problem down into the individual operations required. These operations are then split into two parts; a data path and a control path. Once these two paths have been determined, they are implemented separately, the first as specific hardware components such as registers, multiplexors, etc.; and the second being implemented generally, unless extremely simple, as a state-machine. This allows for the data path or the control path to be adjusted or changed without impacting the other unless specifically required.

In addition to this macro concept, the specific method of VHDL coding used to implement these paths is done in such a way as to create specific hardware in a known predictable way. Not written using complex types and unpredictable conversions leaving it up to the synthesis tools to produce the required hardware. This method will be covered in more detail in section 4.3.

1.6 Software implementation methodology

The software's inputs and outputs should match those of the hardware. Both implementations will receive the same data and be trying to come to the same output. However, as 'C' is a sequential language, this is the only method available to it. The primary goal of the 'C' code is therefore to use as few CPU instructions as possible to accomplish the same task as the hardware so as to form a basis for comparison of

performance.

1.7 Contribution of work to ongoing project

The implementation of this low cost hardware correlation circuit was intended be used as part of a flow meter measurement system. This flow meter was an ongoing research project being worked on by a variety of graduate students working with Dr. Conroy [8]–[10].

2. Background and FPGA's

2.1 FPGA operation

The basic operating block of FPGA's is the programmable logic block (PLB). This in its most basic terms consists of a Look-up table, a flip-flop and a logic gate as seen in Figure 2.1.

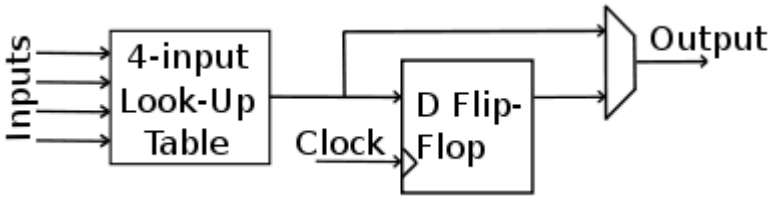

Figure 2.1 Basic Programmable Logic Block

There are variations to this basic circuit, however all these variations will not be discussed further, as they are generally device specific. Suffice it to say that most of the variations involve structural changes and bit width variations, both of which are not important to the general concept of hardware correlation design and comparison with software performance being presented in this thesis. What is of more importance is the structure of the programmable logic blocks that are actually used in industry today, and specifically used in this research. Xilinx and Altera are the current FPGA market leaders and long-time industry rivals. Together, they control over 80 percent of the market, with Xilinx alone representing around 50 percent [11]. For this thesis, a Xilinx Virtex-II Pro FPGA was used. The PLB's used in this device can be seen in seen Figure 2.2.

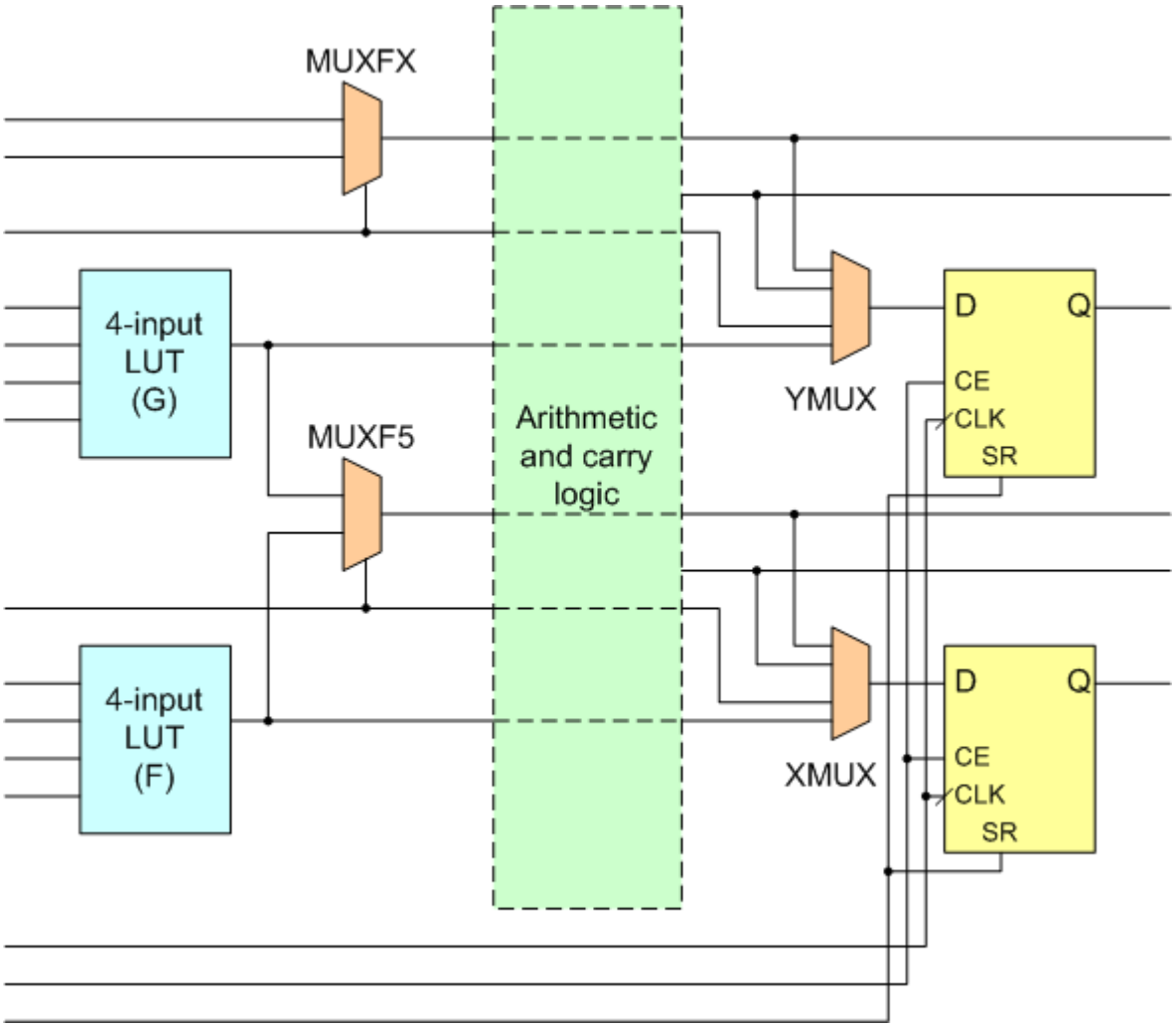

Figure 2.2 Virtex II PLB [12] (original in colour)

In addition to the basic PLB's most FPGA's include dedicated primitives for specific functions. Common ones would include: Multipliers, RAM, PLLs, etc. For this implementation, some of these components were used, namely: 18x18 Multipliers, RAM blocks, and one Digital clock manager (DCM). While these embedded components are not required (with the exception of the DCM for clock control), as the PLB's can generally accomplish all of these functions, by using them the amount of resources required decreases as these parts are there regardless of use, and are generally faster than a PLB equivalent. Therefore overall performance increases, as the dedicated components are faster and more streamlined than PLB combinations. This performance increase comes at the cost of reduced portability of the design however; resulting in a larger number of design changes if there are changes to the underlying hardware.

3. Design methodology

3.1 The correlation equation

The formula for the linear correlation used in this research is as show below.

$$r_{xy} = \frac{\sum x_i y_i - n\bar{x}\bar{y}}{(n-1)s_x s_y} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - (\sum x_i)^2}\sqrt{n\sum y_i^2 - (\sum y_i)^2}}.$$

(1)

The performance problems with this equation result from the large number of individual calculations required every time it is evaluated. Generating the sums aside for the moment, there are 7 multiplications, 3 subtractions, 2 square-roots, and 1 divide. This by itself does not seem like a large amount, but add in the calculations of the sums, and the operation load increases dramatically. The total number of operations for calculating the sums is in direct proportion to N, the number of elements in the data set. For one calculation from scratch, this results in:

N * 5+3 addition/subtractions

N * 3+7 multiplications

2 square-roots and 1 divide.

This can quickly result in a huge number of calculations for large data sets. And for single operation or changing data sets you are stuck with this number of calculations. In most cases however, this type of correlation does not need to be repeated often so the

large number of calculations is less troublesome. More common applications of correlation involve multiple calculations over a large range of data. Most commonly used are:

Cross-Correlation [13] - is a measure of similarity of two waveforms with a time-lag applied to one of them.

This cross-correlation is done by taking one data set and sliding it across the other and calculating the correlation value at each step.

Auto-Correlation [14] - is the cross-correlation of a signal with itself.

Although cross-correlation can be implemented in the transform domain to reduce the number of multiplications required, the normalized form of cross correlation does not have a simple frequency domain expression. Normalized cross-correlation should not generally be implemented in the transform domain for this reason.[15] This normalization problem can be overcome through normalization of the data beforehand, but this is not always possible in an efficient way.

In addition to the normalization problems, the performance increase of the transform method is also dependent on a number of other factors. These primarily include the total dataset size, the difference in size between the data and the information being matched, as well as the length of the correlation sweep.[15] The transform method works most efficiently vs. direct computation when the datasets are large, the items being matched are similar in size, and the sweep is across the entire range. The further you move away

from any of these conditions, the more efficient the direct method becomes in comparison.

For the cross-correlation application in this research, the requirements move away from two of these conditions, additionally, the cost of multiplies in hardware are less of a penalty as compare to software implementations. The data sets in general are small, and the sweep length is much shorter than the total data length. There are also good ways to reduce the number of calculations needed at each individual calculation through storage of calculations done in the previous step.

3.2 Numerical system choice of fixed point

Hardware lends itself very naturally to fixed point mathematical operations. Other numerical systems such as floating point are implemented in hardware using fixed point math and converting and compressing during each step of the calculation to produce floating point number results. This process trades precision for numerical range in the same number of bits. As the results of a correlation calculation are between -1 and +1 and therefore a large range was not needed, as well as the fact that fixed-point math is faster and more precise, the use of a fixed-point implementation was chosen.

3.3 Precision choices and its effects

The precision required for the calculation is one of the primary deciding factors in the number of bits that are going to be used. The second major factor is the number of resources available in the FPGA being used. Of these resources the two main concerns

are the number of PLCs and the size and number of dedicated hardware resources, such as Multipliers and RAM. For the Xilinx Virtex-II Pro that was used, there are many 18x18 bit multipliers and dedicated RAM blocks. So the size of the multipliers and the IO ports of the RAMs are a major consideration. In addition to this, the total size of the window being used and the amount of precision required in the result played a key part. As the total size of the individual sums were a product of the input-size x window-size.

$$Number\ of\ sum\ bits = input\ bits \times window\ bits$$

This results in bit sizes that are:

$$Number\ of\ sum\ bits = 12\ bit\ number \times 10\ bit\ window = 22\ bits$$

For the implementation required for the sensor design this provides a range of 0-511(9 bits) for the window, and 12 bit sample data going into the inputs. From here each of the sums in the worst case must be squared and multiplied by the window size again.

$$Bits\ before\ decimal\ precision = 22\ bits * 22\ bits * 10\ bits = 54\ bits$$

The next step is to take the square root of this number. This is a fractional result in most cases, and for the desired result we want a final output of 32 bits of precision, all of which are significant. 32 bits of output was chosen for a couple of reasons. The two most significant of which, are the size of the IO bus available in the development board the

hardware was developed on, and the size of standard integers being 32 bits making the software case comparable. The 32 bits of binary precision is equivalent to approximately 11 decimal places of precision. Considering that the correlation equation is normalized to plus or minus one, this will result in a single ones place bit, and 31 following the decimal. So we need at least 31 decimal places of precision in our square-roots. Increasing our minimum number of bits required for the output of the square-roots from 54 to: ½ (54) + 31 = 58 bits.

The two square-roots are then multiplied together before the final divide increasing it yet further to 116. To keep the numbers as bases of 2, the square-root was increased to 64 bits, and the divider to 128. This increase in size allows the inputs to be up to 16 bits if required. This size fits nicely into the 16 bit ram ports, as well as the 18×18 multipliers available on the Virtex-II Pro.

4. Hardware implementation methodology

4.1 Methodology principles and preliminary evaluation

The VHDL hardware design methodology used for this design was used for creating exactly the hardware the designer intends, and not leave variable implementations to the synthesis tools. The specific hardware layout to be designed and implemented in VHDL was done using my own methodology based on the above principles.

4.1.1 VHDL coding methodology

One of the methods primary principles is to keep the VHDL code as simple as possible for the synthesis tools to interpret. This involves using only standard logic, ulogic and their vectors for signals. As these signal types are very unlikely to be misinterpreted by either the synthesis tools or the user, and any type changes are done specifically by user. This makes any changes to the synthesis tools unlikely to result in different hardware being synthesised. Other restrictions include only using If and Case statements vs. using complex structures such as wait, while, when to build logic.

For example, rather than assigning a multiplexor using:

Source : IN ARRAY (1 downto 0) OF bus;

value_out <= source (conv_integer(unsigned(sel)));

Use a more generic and simple definition of:

if sel = '1'

then value_out <= desired_value;

else value_out <= other_value;

end if;

While this does result in more lines of code in many situations, each of these lines is very simple to understand. The overall complexity of the code remains similar, but the simple definition results in precise control over what hardware is created. There are papers discussing coding methodologies out there *[16]*,*[17]*; but many of these papers do not agree with each other as to what method is best, or what hardware is most efficient. In general they discuss quality in terms of the number of CLB's used, but this does not measure performance of a design. Performance vs. CLB use is important in the design of a working system.

4.2 Equation evaluation for implementation

The cross-correlation equation requires many mathematical operations to solve. These operations must be done in some semblance of order to optimize the overall process, as many of the operations require results from previous operations. The operations required can be broken down in general as follows:

Table 4.1 Cross-Correlation Sequence

Step.	Description
a.	Get input data to be cross-correlated
b.	Calculate sums required, such as ∑x, ∑y, $\sum x^2$, etc
c.	Multiply these sums together as required
d.	Subtract these multiplied sums from each other as required
e.	Solve the two square roots for the denominator
f.	Divide the numerator and denominator

The two primary options that require consideration when implementing each of these steps are: performance and resource use. These considerations not only affect what implementation should be used, but also how the chosen method is implemented. Not all serial, parallel, or pipelined solutions are equal. Each of the operations required should therefore be looked at separately to find the optimal solution.

4.2.1 Serial implementations

Serial operation, is executing in a sequential one-after-the-other fashion. Each step while walking is an example of serial operation.

Figure 4.1 Serial Example

Serial operation has primarily two advantages over other methods, and only one

significant disadvantage. Its main benefits are a reduced amount of silicon, and reduced design complexity, while the primary disadvantage is speed.

In the case of the operations to be performed for the correlation equation a number of these operations lend themselves well to serial implementation. These include primarily the calculation of individual sums, as well as the individual square-roots and the divider processes.

4.2.2 Parallel implementations

Put simply, operations in parallel are operations done at the same time. Walking and breathing is an example of a parallel operation.

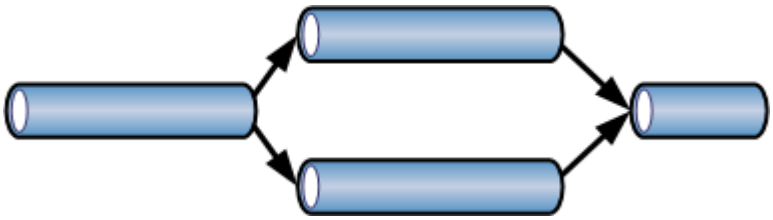

Figure 4.2 Parallel Example

Parallel operation has one major advantage over serial and one primary disadvantage. It can greatly increase the speed of an operation at a direct cost of increased resource usage. In general the speed increase is proportional to the resource increase, plus the overhead of the separating and recombining processes. Because of this cost, parallel implementations should only be used if the speed gains are beneficial to the overall process. For example, a bad parallel implementation would be accelerating a process that

was already faster than the slowest step. This costs additional resources for no gain in overall performance.

In the case of the correlation operations, some steps can benefit from a parallel implementation. These include: calculating all the sums at once, doing all the subtractions simultaneously and solving the two square-roots in tandem. A more questionable case would be the required multiplications; this will be discussed in more detail in section 5.

4.2.3 Pipelined implementations

Pipelining is a method of speeding up a process by using multiple stages to reduce the reaction time of the input to the output at each stage. A group of people passing an object around the table is an example of pipelining; each person is only involved for a short amount of time.

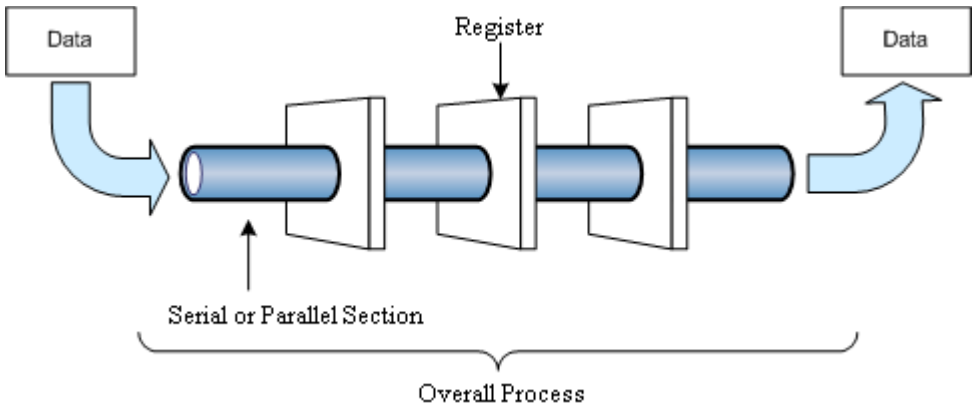

Figure 4.3 Pipelining Example

The primary function of pipelining in hardware design is to allow for higher clock rates by reducing the propagation delay though a circuit. Take the case in Figure 4.3. If this overall process without the registers had a propagation delay of 40ns, this would result in an ideal maximum clock frequency of 25MHz. By pipelining this process with three pipeline registers (assuming uniform sections) the propagation delay is reduced to 10ns for each section, resulting in an ideal maximum clock frequency of 100MHz. This provides a four times improvement for the cost of three registers. Each step in a process should be analyzed as to whether pipelining would be beneficial. However, the overall system must be considered as well, as pipelining a section that is not the slowest section will have a negative overall performance effect.

The question becomes, clock speed vs. register use. Once a single section can no longer be subdivided, further pipelining of other sections will have no effect on the overall maximum clock rate. This is the same principle as the weakest link in a chain being the limiting factor. Pipelining has the additional disadvantage of creating a multi clock cycle delay from the input to the output. This means that for the above three stage pipeline, the output is not valid until four clock cycles after the first input was loaded, as compared to one cycle for a non-pipelined implementation. And three additional clock cycles will be required to get the final output after all the input has been completed. In effect the three stages of pipelining ads a time lag of three clock cycles between input and output. The specific operations that were pipelined will be discussed in section 5.

4.3 Methodology of process breakdown

Once the required steps of the process are determined, each of these steps should be broken into self-contained process blocks where possible. This will allow these process blocks to be implemented as a sub process with its own data and control path. From Table 4.1 two of the steps jump out immediately as suitable for this, both {F} and {E}. Also steps {C} and {D} would lend themselves well to being done together in the same process, as the numbers from step {C} are used in step {D}. However, there are many of these operations, so a process to control their order will be required. This leaves steps {A} and {B}, which can also be conveniently done together, as the calculation of the sums is done from the inputs. There is the additional complication of the XY sum that must be recalculated in its entirety each time the equation is evaluated, as there is no way to use a simple add and subtract method as in the case of the pure x and y sums. This can still be done in the same process however, as the required hardware remains the same regardless of the number of times the operation is required. This results in five basic operational sub-processes. Once the process blocks are determined, encapsulation of these processes can be planned; meaning, which processes will be contained inside others. This should be done in such a way as to optimize resources/performance and avoid confusing structures.

5. Implemented hardware

The overall process for the calculation of the cross-correlation in hardware was implemented as five separate task blocks. These tasks will be explained in a bottom up sequence one building upon the others, and are as follows:

Table 5.1 Implemented Hardware Blocks

Divider	(section 5.1)
Square-root	(section 5.2)
Selectable size multiplier	(section 5.3)
Equation calculation control	(section 5.4)
Main controller	(section 5.5)
Overall System	(section 5.6)

5.1 Divider

The divider used in this process block is a 128 bit non-restoring divider that takes one clock cycle per bit to execute. An additional four clock cycles are used as a result of overhead for setup and control, for a total of 132 clock cycles. While there are other divider methodologies that would improve the number of bits processed per clock cycle [18][19], these methods involve large amounts of additional logic, as well as longer logic chains per cycle, resulting in a reduced clock frequency. Additionally, the variable step dividers have unpredictable cycle times [18]. Unpredictable cycle times result in either beneficial or unbeneficial improvements. Unbeneficial effects can include fewer cycles required than the next slowest section; or taking longer than average and becoming the

slowest section if all others are finished prior to the divider; resulting in wasted resources and a possible unbeneficial overall clock speed reduction. Only in cases where the number of cycles falls in an optimum range will the result be an improvement in performance without wasting resources. The divider process was therefore designed to use a fixed number of clock cycles per bit, with a low propagation delay to allow for faster clock rates. The primary components of the design include:

1	130-bit adder
1	7-bit up counter
258	1-bit registers

This configuration has a maximum clock frequency of approximately 146MHz. This frequency can vary slightly depending on how the routing software maps the circuit but is very unlikely to be more than ±1MHz. Once the circuit is routed, this maximum clock frequency is known and fixed. The divider is then pipelined and is therefore operated simultaneously with both the xy sum calculation and the equation calculation control process (which includes the multiplier and the square-roots). This allows the divider to do the division of the previous step simultaneously with the calculation of the next numerator and denominator. This pipelining has the disadvantage however of adding an additional cycle of delay before the final output can be read.

For future diagrams, this process will be represented as follows in Figure 5.1:

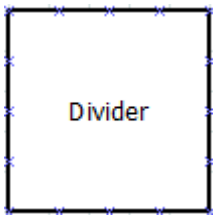

Figure 5.1 Divider Diagram Representation

5.2 Square-root

The square-root machine was implemented using a methodology that works very well with binary hardware calculations. The method follows the same long hand technique that used to be taught in primary school in the early part of the 20th century and works with any number base. The basics are as follows:

Table 5.2 Square Root Process

A Starting at the decimal point pair off the digits.

B Find the largest square that subtracts from the left-most pair and still yields a positive result. This is the first digit of the square root of the whole number.

C Concatenate the next pair of digits with the remainder.

D Multiply the square root developed so-far by 20 (for decimal). Note that the least significant digit is a zero.

E The next digit in the square root is the one that satisfies the inequality:

(20 × current + digit) × digit <<= remainder

"current" is the current square root and "digit" is the next digit being produced.

F Form the new positive remainder by subtracting the left side of the equation in the previous step from the right side.

G GOTO step C until desired number of digits is reached then you are finished.

The procedure for binary numbers consists of taking the square root developed so far, appending 01 to it and subtracting it from the current remainder. The 0 in 01 equates to multiplying by two; the 1 is a new trial bit. If the resulting remainder is positive, the new

root bit is truly 1; if the remainder is negative, the new root bit 0. If the result is a 1, repeat this process. If the remainder is a 0 however an alternate step needs to be taken. Instead '11' is appended to the root developed so far and on the next iteration is added rather than subtracted. If the addition causes an overflow, then on the next iteration you go back to the previous subtraction mode. For an example of the method described above see Figure 5.2:

```
    1  0  0  1 .
  -----------------
  | 01 01 11 11 . 00
    -01 <-- Subtract 01
    ---
    00 01  <-- positive: first bit is a 1
    -1 01  <-- Root is "1" in previous step ; append 01 ; subtract
    -----
    11 00 11  <-- negative: 2nd bit is a 0
      +10 11  <-- Root is "10"; append 11 and add.
   ---------
    11 11 10 11  <-- positive (no overflow); 3rd bit is a 0
        1 00 11  <-- Root is "100"; append 11 and add
   ---------
  1 00 00 11 10  <-- Overflow: 4th bit is a one
... etc
```

Figure 5.2 Square Root Example

This method results in the process taking half the number of clock cycles as input bits to calculate the integer root. However it will take an additional number of clock cycles equal to the number of decimal places of accuracy required. The number that is being square-rooted is input as 64 bits, however as we want the final calculation to have 31 accurate decimals, we need at least that here. To keep the number of IO bits at a power of 2, 32 decimal bits were calculated. This resulted in a total calculation time of 64 clock cycles, plus 3 clock cycles of setup and control. This brings the total number of clock

cycles to 67.

The primary components of a single of these square-root processes include:

1	65-bit adder
1	6-bit counter
3	64-bit registers

The propagation delay for this circuit is lower than the divider due to the smaller number of bits being operated on at one time, allowing for a higher maximum clock rate of 194MHz. This again is ±≈1MHz depending on the exact routing positions. All future MHz maximum clock rates should also be regarded with the ±≈1MHz routing variation. The square root process is run in parallel with itself by using two of these process blocks.

For future diagrams, this process will be represented as follows in Figure 5.3:

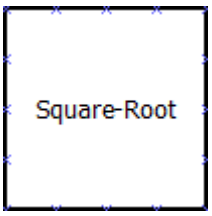

Figure 5.3 Square Root Diagram Representation

5.3 Selectable size multiplier

The selectable size multiplier is one of the overworked sections of the process. This single block will be called upon many times over the course of the calculation and is one of the main processes that could slow down the overall operation. The multiplier block consists of control logic and embedded 'primitive' component multipliers. The 'primitive' multiplier included in the Virtex-II Pro chip is an 18×18 bit multiplier that takes a single clock to execute.

The size of the multiplications that are required for the correlation calculation are often larger than this. The width of the multiplications range from a small <= 16×16 (in the required application 12×12 of non-zero data) for the x^2, y^2 and x × y multiplies; to 32x32 and 64x64 for the correlation equation. The 12×12 multiplies can be executed directly with a single multiplier primitive, and this is done in the main control process not with the variable sized multiplier component, and will be discussed in section 5.5.

The 32x32 and 64x64 cases require either more than one 18×18 bit multiplier primitive or multiple passes through a single 18×18 bit multiplier, or a combination of both. Another consideration for these large multiplies is the size of the accumulator used to store the results of each multiply. As a 64×64 multiply requires in theory a 128 bit accumulator, this can produce a very limiting restriction to clock speed when combined with the delay though the multiplier primitive as well. The accumulator has a longer and longer propagation delay depending on its length, so using smaller individual steps and adders has advantages over a single large accumulator.

The variable size multiplier block was implemented using a single 18x18 multiplier primitive to do 16×16 multiplies. This was done for two primary reasons. The first being that the number of available multiplier primitives is limited so the fewer used in a single implementation, the more copies of that implementation can be used on the same chip. Secondly, the number of clock cycles required to do the 64×64 multiply using only the one multiplier primitive is still a fairly small 18 even counting overhead. As the divider requires 132 clock cycles to complete, this leaves us with the ability to accomplish 7 multiplies per divide with 6 clock cycles to spare, and this is assuming 64×64 multiplies in each case. Conveniently the correlation equation requires 7 multiplications and 1 divide (after the sum calculations, see Section 3.1) to solve in each iteration.

A second major design point of the multiplier block was to reduce the overall propagation delay of each clock cycle. This was done in one way by reducing the size of the accumulators required at each step, and building the final result 16 bits at a time. This was accomplished through use of the fact that multiplies can be broken down into groups of shift and add operations. For example, take the multiplication of the following theoretical numbers: DCBA×dcba. This will result in a final total equal to the sum of the following multiplies:

Table 5.3 64x64 Multiplication Steps

A×a	A×b	A×c	A×d
B×a	B×b	B×c	B×d
C×a	C×b	C×c	C×d
D×a	D×b	D×c	D×d

If we then order the small multiplies in such a way as to do all operations in a rightmost to leftmost fashion with respect to how far left-shifted each pair is we get:

Table 5.4 Multiplication Sequence

Set -	**# of 16 bit shifts**	**16x16 multiplications**
1	0	A×a
2	1	A×b, B×a
3	2	C×a, B×b(done first of these three), A×c
4	3	D×a, C×b, B×c, A×d
5	4	D×b, C×c, B×d
6	5	D×c, C×d
7	6	D×d

By doing these multiplies in the proper order we can build our result one half multiply at a time. Additionally by doing the B×b multiplication first in set 3 we can use the same

operational logic for both the 32x32 and the 64x64 cases. Due to the Bxb multiply operation being the final multiply of the 32×32 case when the 32x32 multiply is loaded into the BAxba section of the 64x64 inputs. By doing it first, the same multiplier can be used to do the 32 bit and 64 bit cases, simply by stopping earlier. 18 clock cycles for a 64×64 multiply, and 6 for a 32×32 multiply including two overhead cycles due to pipelining and buffering.

In the 18×18 multiplier implemented doing 16×16 sets; each shift translates to 16 bits. Each 16×16 multiplication resulting in a 32 bit result, the lower 16 of which are now known to be correct in the final result and are saved to the output register. The upper 18 bits are added into the next set, the top section is 18 bits not 16 bits because it includes the extra two bits required from adding four 16×16 multiplies together in Set 4 (see Table 5.4) plus the maximum carry forward of the previous sets upper bits, which has a maximum value equal to:

$$(0xFFFF \times 0xFFFF) \times 4 + 0x2\_FFFA =$$

$$0x3\_FFFA\_FFFE \text{ (totalling 34 bits)}$$

Through this process the size of the accumulator required at each step is reduced from 128 bits to 34 bits. This significant reduction in the size of the accumulator required reduces the propagation delay. However, the delay though the multiplier must be added as well, reducing the clock speed as compared to a pure adder structure.

This results in a maximum clock frequency of 114MHz. This frequency can be increased by pipelining the adder section with the multiplier primitive, increasing the maximum clock frequency to 156MHz at the expense of an extra 34 bit register and a single clock propagation delay. This is worth doing, as it raises the maximum clock frequency from the 114MHz limit here, to the 146MHz of the divider block without becoming the limiting component in the number of clock cycles taken.

For future diagrams, this process will be represented as follows in Figure 5.4:

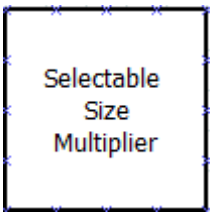

Figure 5.4 Multiplier Diagram Representation

5.4 Equation calculation control

The equation calculation control process's primary task to coordinate the order of operation of the multiplications and subtractions required to solve the correlation equation. This process is started and stopped by the main control process. Whereas this process controls the size selectable multiplier process as well as the square root processes.

When this process is done in a primarily serial manner, the total number of clock cycles required is 213 doing 64×64 multiplies in each case. This is therefore the longest overall process so far when executing in this manner. However, many of these clock cycles are a result of waiting for the multiplier and square-roots calculations to complete. The multiplier is run 7 times, and the square-roots once. These total; 126 clock cycles for the multipliers and 67 for the square-roots, for a grand total of 193 clock cycles of internal wait time and 20 clock cycles for controlling the process. This is not the whole story however; as there is an external wait time spent waiting for the calculation of the X×Y set of sums, but this can be done in parallel with waiting for the multipliers and varies in time depending on the size of the window being used. This will be discussed further in section 5.5.

The total cycle time can be reduced by using the variable size of the multiplier discussed in section 5.3 to only do 32×32 multiplies when this is all that is required. As 6 of the 7 required multiplies can be done in this manner, the total multiplier iterations are six 32×32 and one 64×64 sequences. This results in a new total of: 6 clock cycles × 6

repetitions + 18 clock cycles × 1 repetition = 54 clock cycles, less than half the number required using a fixed 64×64 multiply.

When this new total is combined with the square-root and control totals, the total number of clocks is now 54+67+20 = 141. This can be further reduced by partially paralleling the square-root and some of the multiplier cycles. Specifically; calculating the denominator multiplies first, then calculating the square-roots and the numerator simultaneously. This means doing the square-roots in parallel with two of the 32×32 bit multiplies and their subtraction, which conveniently does not involve additional hardware but a change in the order of execution of the state machine. This saves us 16 clock cycles, bringing the total down to 125 clock cycles.

This new total of 125 clock cycles is below the 133 required by the divider, so further reduction of this process will not result in savings on overall process time. Even though by pipelining and parallelizing additional times the number of clock cycles in this process can be reduced further; this step is not necessary however, without reducing the number of clock cycles the divider takes as well.

The maximum clock frequency of this process is limited by one primary source; the multiplexed data selection for the multiplier input. This propagation delay results in a maximum of 148MHz. This clock frequency is faster than the 146MHz of the divider process however; therefore there is no need to improve the clock rate limit of this process any further as it will not result in a better overall system performance.

For future discussions, this process will be represented as follows in Figure 5.5:

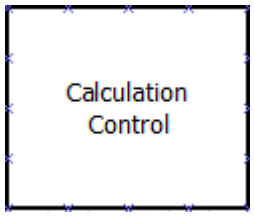

Figure 5.5 Calculation Controller Diagram Representation

5.5 Main controller

The main control block controls what order each of the sub-process operates as well as dealing with the data input and control of the XY sum calculation. In addition it starts and stops the sub-processes, controls the storage registers for the pipelining between those processes and keeps track of the pass and sweep counts for the cross-correlation slide.

There are two primary functions to optimize in this process; the XY sum calculation and the data input. In addition there are control structures to coordinate, but these involve basic counters and a state machine, and are fairly straight forward to optimize. The primary optimization involved is keeping the propagation delays between the RAMS and the 18×18 multiplier used for calculating the input squares as low as possible.

The logic chain starts in the RAM blocks, from there the data path goes through multiplexors controlled by the state machine, then the 18×18 multiplier, more multiplexors, and finally the adders and subtractors needed to update the XY sums. This long chain results in a high propagation delay and as a result, a maximum clock rate of 106MHz.

Therefore to improve the clock rate a pipelining stage was added after the multipliers. When this pipelining stage was added it resulted in a new maximum clock rate of 156MHz. This is higher than the limit of the divider, so further pipelining is unnecessary.

A further consideration for this process was the total amount of resources used. As this process deals with many things simultaneously, the potential to have large amounts of extra resources used was high. In some cases reducing the amount of resources was possible at the cost of performance. The most notable of this was the multiplexors surrounding the 18×18 multipliers.

These could have been tri-state buses or multiplexors. And while the multiplexors are faster, they require far more resources than a tri-state bus in the FPGA [16]. As long as these tri-state buffer resources are available, they are more efficient to use as far as overall resource usage is concerned. However, in general tri-state buses have slower propagation times that multiplexors, and not all FPGA architectures support internal tri-state buses, therefore this should be considered when designing the data path.

In this design, the choice was made to use a multiplexor implementation. This was done primarily to allow for the design to be duplicated inside the Virtex-II Pro a larger number of times. As the total number of available tri-state buses would limit the total number of copies that could be simultaneously instantiated.

For future discussions, this process will be represented as follows in Figure 5.6.

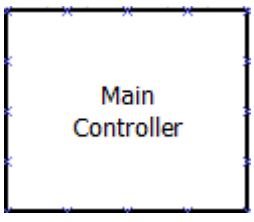

Figure 5.6 Main Controller Diagram Representation

5.6 Overall system and performance

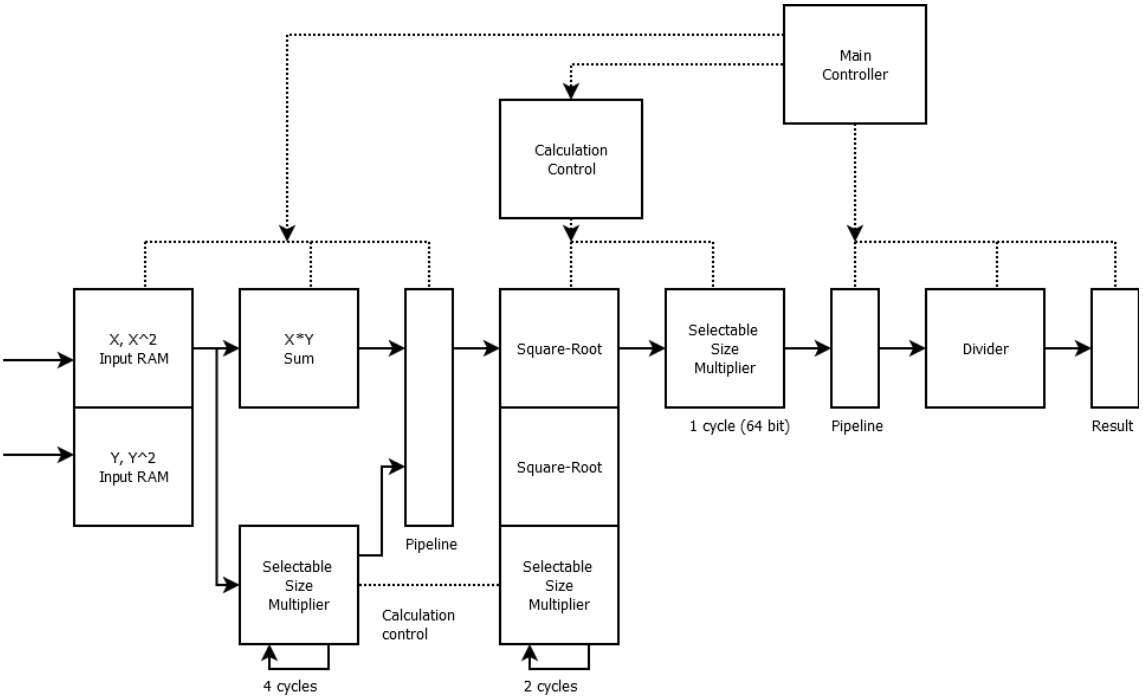

Figure 5.7 Hardware System Organization

The basic overall operation of the system can be seen in Figure 5.7.

The overall performance of the correlation system is therefore a 146MHz clock rate and takes 133 clock cycles to complete, limited by the divider in both cases. This results in a throughput of ~1.1 million complete correlation calculations per second per instantiation. Each full calculation therefore takes 911ns.

The total number of resources used for one instantiation can be seen in Table 5.5 and Table 5.6. Using the largest available chip, the total number of correlation calculations increases to 244.2 million per second.

Table 5.5 Resources per Hardware Implementation

Resource Type	**Number**	**Notes**
Logic Cells	~400	Depends on place and route settings.
18x18 Multipliers	2	1 for 32/64 variable, 1 for xy sums.
Block RAM	4*+1	4* RAMs for the input data, only needed once regardless of the # of copies.
DCM	1*	Only needed once regardless of the # of copies

Table 5.6 Maximum Resource Utilization

Virtex-II Pro Device	**Logic Cells**	**Distr RAM (Kb)**	**18x18 Multiplier Blocks**	**18kb RAM Blocks**	**DCMs**	**Approximate max # of Correlation Copies**	**Approximate % of logic cells used**
XC2VP2	3,168	44	12	12	4	6	75.76%
XC2VP4	6,768	94	28	28	4	14	82.74%
XC2VP7	11,088	154	44	44	4	22	79.37%
XC2VP20	20,880	290	88	88	8	44	84.29%
XC2VPX20	22,032	306	88	88	8	44	79.88%
XC2VP30	30,816	428	136	136	8	68	88.27%
XC2VP40	43,632	606	192	192	8	96	88.01%
XC2VP50	53,136	738	232	232	8	116	87.32%
XC2VP70	74,448	1,034	328	328	8	164	88.12%
XC2VPX70	74,448	1,034	308	308	8	154	82.74%
XC2VP100	99,216	1,378	444	444	12	222	89.50%

6. Software comparison

6.1 Software implementation

The implementation of software was done using a basic recreation of the operations performed by hardware. A method for improving the speed of the cross correlation equation was investigated [15]; however this was not implemented as part of the software comparison, this technique involves pre normalizing the data and is also very sensitive to noise. Pre-normalization of the data involves a large number of additional calculations and would greatly slow the overall processing time. Additionally, using normalization for real time data can result in a loss of precision or involve large lags to allow time to calculate the real working average or maximum.

As the focus of this thesis was the implementation of the hardware and the process of designing hardware though a methodized approach. The 'C' code was therefore designed to mimic the function of hardware by working with the same data, so as to compare the performance of doing identical tasks with both methods.

The software process was executed in a serial manner from one calculation to another as can be seen in Table 6.1 and Figure 6.1.

Table 6.1 Software Calculation Sequence

Step	**Action performed**
1-4	Updated X, Y, X^2, and Y^2 sums by subtracting the oldest sample and adding the new sample. In the case of the squared sums, the squared value is used for the subtraction and addition.
5	Calculate the X*Y sum
6-11	Multiply all sums together as required by equation (1)
12-14	Do the subtractions required
15-16	Calculate the two square roots
17	Do the final division

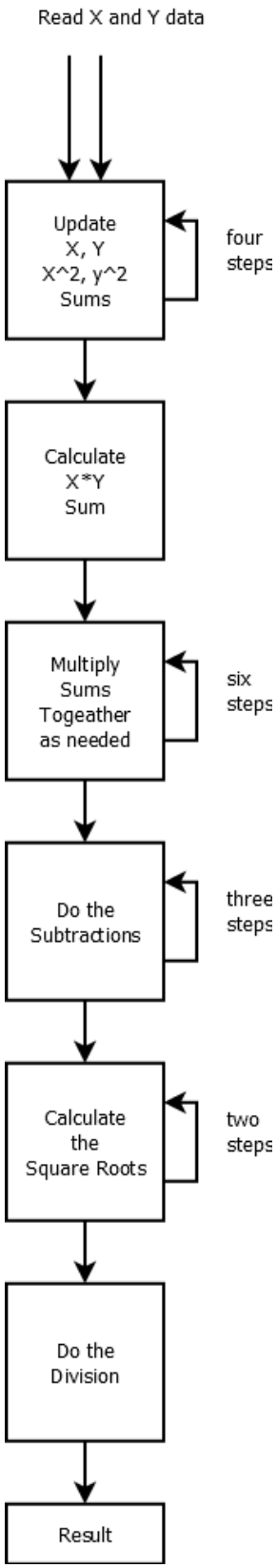

Figure 6.1 Software Sequence Diagram

Due to the imprecision of measuring time within the windows environment, measurement of a single correlation equation was unfeasible due to the imprecision of the timer function. This imprecision was due to the counter only returned values to 15ms precision. To make measurement possible, the correlation equation was performed 150,000 times and the total time needed to complete all of the calculations was measured. This total time was divided by the number of runs to determine the average time for each run.

These 150,000 calculations took a combined average total time of ~700ms, which results in each individual calculation taking approximately 4.7us. Therefore the total throughput of the software solution is approximately 213,000 correlation calculations per second.

This speed could be improved by using multi-core processors using multiple threads. For the purposes of this thesis, multi-threading was outside the scope of what would be covered, however it can be roughly assumed that with modern multicore processors, an optimal solution would be approximately linearly faster than the single threaded results.

7. Conclusions and future work

7.1 Conclusions

This research was conducted to provide an implementation of a cross correlation algorithm using a structured hardware design methodology for use in low cost real time flow meters; as well as to provide a comparison of this hardware solution to a software solution in terms of performance on equivalent technology platforms.

The design methodology of breaking the overall process into small sub units that could be optimized separately allowed each sub process to be designed to use space efficiently in terms of hardware resources in addition to optimizing performance. By defining the data flow in a structured manner in terms of specific operations controlled by state machines, the determination of bottlenecks in the design was simplified. This in turn allowed for efficient insertion of pipeline registers and application of parallelism in order to maximize performance without wasting available resources.

Additionally, the HDL coding methodology involved creating known HDL code structures that would be interpreted consistently by synthesis tools allowed for a predictable implementation and so allowed any changes for performance optimizations to be done in a predictable way. In this manner, propagation delays as well as routing flow could be predictively controlled and improved.

This allowed for nearly all of the available resources in the target FPGA to be utilized. This high level of utilization presented some problems during place and route of the

Virtex-II Pro. With resource utilization approaching 90 percent meeting timing requirements in the design proved challenging as well as time consuming. However, as each separate instantiation did not need to pass information between themselves, the timing only had to be maintained across a small area of the device at one time. This allowed the tool to adjust the timing of one instantiation without interfering with another. In a design with more inter-dependency of the copies, utilization at this high a level may not be achievable at peak clock rates.

A single instantiation of the correlation flow has a throughput of one complete calculation every 911ns when operating in full pipelined mode, as compared to 4.7us for the software solution. With only a single instantiation, the hardware solution outperformed the software solution by approximately 5:1 during continuous operation.

In the case of a single standalone calculation, the performance of a single hardware instantiation was very similar to that of software, closer to 2:1 in favor of the hardware.

The true performance of the hardware relative to the software solution was substantially better with full device utilization. This comparison is the most valuable for relative performance levels, as the software solution makes full use of all the silicon resources available to it. Therefore to make a true comparison of performance vs. real-estate, the hardware solution should also use all available resources of the die. The largest of the Virtex-II Pro FPGAs are capable of holding 222 copies of the correlation hardware. This results in a performance ratio of approximately 1145:1 in favor of the hardware for the

largest device.

Implementation of the software design on multi-core processors could close this gap somewhat, but only by at most, the number of cores available. This in general will not exceed 16 cores in a single CPU, and will be in the 2-8 range for typical devices.

It should also be noted that in addition to the FPGA fabric resources there are two 400MHz embedded processors inside the Virtex II Pro device that were not utilized. Used of these embedded processors could be used for system level control logic and external communications to save having to implement this inside the FPGA fabric itself.

A comparison with the design of a multiplier developed in [20] will be discussed as an example of how the method described in this thesis helps to arrive at an optimal solution for a given problem. The level of optimization of their process using the potential speedups with regards to optimization of parallelism and pipelining in individual logic chains as discussed in this thesis led to performance increases, however their implementation of parallelism was not as efficient as the one presented in this thesis in terms of throughput. The multiplier solution discussed in [20] used eighty-one 18x18 multipliers to do a single 256x256 multiplication in 3 clock cycles at ~ 30MHz, using the technique for the multiplier developed for this thesis, a single 256x256 multiply could be done using only a single 18x18 multiplier primitive taking 256 clock cycles at ~150MHz. Using the same number of multipliers involves using 81 copies of the variable sized multiplier; this results in a performance of 47 vs. 10 million mult/s in

favor of the multiplier design discussed in this thesis. It should be noted that the FPGA used for their development was not the same as the one used in this thesis, however they compared their solution to one implemented on the same Virtex II Pro family; which had even poorer performance.

Other papers have discussed and compared performance of hardware vs. software for specific problems as well [21]–[26]. The general trend of these papers on various FPGA and processor technology levels is the same as that concluded in this paper, namely that FPGA solutions are in general orders of magnitude faster than software solutions. Performance improvements discussed in the mentioned papers were generally in the range of 25x-200x for their calculations. However in most cases they did not discuss the exact resources used in the target FPGA devices, so they may have been able to get additional performance through use of multiple instantiations of their solution; in the case [21] multiple instantiations were used.

While this research was performed using older versions of hardware due to their availability at the time this work was performed, the performance ratio of hardware to software should remain relatively constant across technology levels and will be discussed below.

As an example as to the expected performance ratio at different technology levels, a comparison will be made between the Virtex II Pro and Pentium 4 (1.70 GHz, 256K L2 Cache) used in this work and the more modern Virtex 7 and Intel i7-975 processor.

Based on the exacting comparison of processors for specific tasks performed in [27], the i7-975 is between 5-15 times faster than a Pentium 4 depending on the specific task being performed.

Table 7.1 gives an overview of the performance characteristics between Virtex II Pro and the more modern Virtex 7. And as can be seen, the performance increase varies between different components

Table 7.1 Virtex II vs Virtex 7

Available Hardware	**# logic cells**	**# mult/dsp**	**ram (dist) kb**	**ram (block) kb**
virtex 2	99216	444	1378	7992
virtex 7	1139200	3360	17700	67680
RATIO	11.5	7.6	12.8	8.5
performance clock rates (MHz)	**DCM**	**mult/dsp**	**ram (dist)**	**ram (block)**
virtex 2	450.0	200.0	400.0	355.0
virtex 7	1600.0	741.0	740.7	601.3
RATIO	3.6	5.0	1.9	1.7
Total increase ratio(# * clock)	40.8	28.0	23.8	14.3

Table 7.1 does not tell the entire story however; as there are other improvements to the operating architecture that should further improve performance. Using the example of the Multiplier block, in the Virtex II is a simple 18x18 multiplier. In the Virtex 7

however, these are now called DPS blocks and quoting from the Virtex 7 Family Datasheet "*Each DSP slice fundamentally consists of a dedicated 25 × 18 bit two's complement multiplier and a 48-bit accumulator, both capable of operating up to 741MHz. The multiplier can be dynamically bypassed, and two 48-bit inputs can feed a single-instruction-multiple-data (SIMD) arithmetic unit (dual 24-bit add/subtract/accumulate or quad 12-bitadd/subtract/accumulate), or a logic unit that can generate any one of ten different logic functions of the two operands*....."[28] so further optimizations could be possible.

The slice improvements will be ignored for the purposes of this comparison however as it would be hard to predict the exact performance results of these improvements.

Therefore comparing the performance increase in the processors (5x-15x) and the increase in performance on the Virtex parts (15x-40x); the relative performance levels have increased at a relatively even rate, with a slight edge going to the FPGA devices. The exact ratio of the performance changes will depend on the task being done, with the FPGA gaining even more performance if the block rams are not the limiting factor. If the RAM accesses are the limiting factor this can be overcome through the use of parallelism and use of additional RAMS to allow for more values to be read simultaneously, improving data rate. An example using the cross-correlation performed in this thesis would be to store multiple duplicate copies of the X and Y data and then read more values at once when calculating the X*Y sum.

Since this research work was done, other papers have also done analysis of hardware

cross-correlation. For example in [29] a cross-correlator was developed on a Virtex II Pro as was done in this thesis. While their application and specific implementation was different it allows for a general comparison of results. Their correlation engine achieved a clock rate of 144MHz as well as using ~40% of the XC2VP40 chip, with 32 copies of their unit. Based on these numbers their implementation is slightly slower 144MHz vs 146MHz, and uses slightly more resources 1.25% vs 0.92% of the XC2VP40 per instantiation, than the solution presented in this thesis. Additionally, they are processing data that has 8 bits of data and a window length of 8, in an 8x8 matrix as compared to the 16 bits x 100 length window presented in this thesis. Each of their implemented calculations takes 256 ns, which equates to 37 clock cycles or 4.625 clocks per window length as compared to 1.44 clocks per window length in the solution presented in this thesis.

7.2 Future work

The following list suggests directions for future development and study.

- Over the course of this research improved methods of software cross correlation as well as hardware architectures were published. Investigation of these suggested methods could further improve the performance of the implemented solutions.
- Experimentation with different FPGA vendors could result in improved performance vs. cost as compared to the Xilinx devices used in this research.
- Investigation of improved binary divider methods could significantly improve performance of the solution. As the divider was the limiting step both in terms of

both clock cycles, and clock rate, this should be the primary focus for design level performance improvements.

- Greater investigation into the optimizations of the place and route tools. This played a larger role than was anticipated in creating an optimized design. The place and route constraints that are defined for the tool to use play a substantial role in overall performance.
- Design of a custom PCB for use in a specific application should be investigated. The commercial development board used for this research placed additional limitations on what could be done, as the physical IO of the FPGA was predefined.
- Research into optimizing power and minimizing cost would also be beneficial for future work. Many applications would benefit greatly from a low power, high performance hardware solution for cross correlation, GPS receivers would be an example.
- Hybrid or mixed method solutions such as 'system on a chip' designs should be investigated.

References

[1] "Saskatchewan's Economy Economic Overview." [Online]. Available: http://economy.gov.sk.ca/economicoverview. [Accessed: 26-Aug-2014].

[2] "Royalty/Tax Program For High Water-Cut Oil Wells." [Online]. Available: http://economy.gov.sk.ca/PR-IC12. [Accessed: 27-Aug-2014].

[3] G. Falcone, G. Hewitt, and C. Alimonti, *Multiphase Flow Metering: Principles and Applications*. Elsevier, 2009.

[4] C. Soanes, *Compact Oxford English Dictionary of Current English*. Oxford University Press, 2008.

[5] C. "Max" Maxfield, *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Elsevier, 2004.

[6] "Thomas Scherrer Z80-Family Official Support Page." [Online]. Available: http://www.z80.info/index.htm#BASICS_ARCH. [Accessed: 24-Feb-2013].

[7] "State of the CPU: graph showing price vs performance for AMD and Intel," *PC & Tech Authority*. [Online]. Available: http://www.pcauthority.com.au/News/169071,state-of-the-cpu-graph-showing-price-vs-performance-for-amd-and-intel.aspx. [Accessed: 28-Aug-2014].

[8] K. Knutson, "Low-Cost Grain Bin Moisture Sensor Using Multiple Capacitive Elements," University of Regina, Regina, Saskatchewan, 2014.

[9] B. Fitzgerald, "Flow Measurement Using Capacitance Based Watercut Meters For Oil Well Applications," University of Regina, Regina, Saskatchewan, 2012.

[10] S. Pongpun, "Low-Cost Multi-Element Capacitive Monitor For Measuring Levels Of Substances In Storage Tanks At Oil Fields," University of Regina, Regina, Saskatchewan, 2011.

[11] "Altera and Xilinx Report: The Battle Continues -- Seeking Alpha." [Online]. Available:

http://seekingalpha.com/article/85478-altera-and-xilinx-report-the-battle-continues. [Accessed: 09-Apr-2010].

[12] *Xilinx ISE Design Suite*. Xilinx.

[13] E. W. Weisstein, "Cross-Correlation -- from Wolfram MathWorld." [Online]. Available: http://mathworld.wolfram.com/Cross-Correlation.html. [Accessed: 14-Apr-2010].

[14] E. W. Weisstein, "Autocorrelation -- from Wolfram MathWorld." [Online]. Available: http://mathworld.wolfram.com/Autocorrelation.html.

[15] J. P. Lewis, "Fast normalized cross-correlation," in *Vision Interface*, 1995, vol. 10, pp. 120–123.

[16] M. Gschwind and V. Salapura, "A VHDL design methodology for FPGAs," in *Field-Programmable Logic and Applications*, 1995, pp. 208–217.

[17] S. L. INAMDAR, "VHDL Coding Style Guidelines and Synthesis: A Comparative Approach," University of South Florida, 2004.

[18] M. R. Patel and K. H. Bennett, "Analysis of speed of a binary divider using a variable number of shifts per cycle," *The Computer Journal*, vol. 21, no. 3, pp. 246–252, Jan. 1978.

[19] D. G. Bailey, "Space efficient division on FPGAs," in *Electronics New Zealand Conference (EnzCon 2006), Christchurch, New Zealand*, 2006, pp. 206–211.

[20] Y. Gong and S. Li, "High-Throughput FPGA Implementation of 256-bit Montgomery Modular Multiplier," in *2010 Second International Workshop on Education Technology and Computer Science (ETCS)*, 2010, vol. 3, pp. 173–176.

[21] X. Tian and K. Benkrid, "Mersenne Twister Random Number Generation on FPGA, CPU and GPU," in *NASA/ESA Conference on Adaptive Hardware and Systems, 2009. AHS 2009*, 2009, pp. 460–464.

[22] J. Gonzalez and R. C. Núñez, "LAPACKrc: Fast linear algebra kernels/solvers for FPGA accelerators," *J. Phys.: Conf. Ser.*, vol. 180, no. 1, p. 012042, Jul. 2009.

[23] N. Kapre and A. DeHon, "Performance comparison of single-precision SPICE Model-

Evaluation on FPGA, GPU, Cell, and multi-core processors," in *International Conference on Field Programmable Logic and Applications, 2009. FPL 2009*, 2009, pp. 65–72.

[24] J. Teubner, R. Mueller, and G. Alonso, "FPGA acceleration for the frequent item problem," in *2010 IEEE 26th International Conference on Data Engineering (ICDE)*, 2010, pp. 669–680.

[25] A. Mitra, M. Vieira, P. Bakalov, W. Najjar, and V. Tsotras, "Boosting XML Filtering with a Scalable FPGA-based Architecture," *arXiv:0909.1781*, Sep. 2009.

[26] C. He, A. Papakonstantinou, and D. Chen, "A novel SoC architecture on FPGA for ultra fast face detection," in *IEEE International Conference on Computer Design, 2009. ICCD 2009*, 2009, pp. 412–418.

[27] S. Wasson, "Core i3 takes on Athlon II ...and everything else, including a Pentium 4," 16-Feb-2010. [Online]. Available: http://techreport.com/review/18448/core-i3-takes-on-athlon-ii. [Accessed: 22-May-2013].

[28] "Xilinx 7 Series FPGAs Overview." Xilinx, 30-Nov-2012.

[29] J. Buchholz, J. W. Krieger, G. Mocsár, B. Kreith, E. Charbon, G. Vámosi, U. Kebschull, and J. Langowski, "FPGA implementation of a 32x32 autocorrelator array for analysis of fast image series," *Optics Express*, vol. 20, no. 16, p. 17767, Jul. 2012.