Executing Computation Intensive Algorithms on Digital Hardware

Digital Algorithm Optimisation

Gerhard Oosthuizen

Dissertation submitted in partial fulfilment of the requirements of the degree

Master of Engineering

Faculty of Engineering
School of Electric, Electronic and Computer Engineering
North-West University, Potchefstroom Campus

Supervisor: Prof. W.C. Venter

2005

Abstract

Even with the advancement of new technology in the field of digital signal processing, it is some times difficult to implement advanced signal processing algorithms on such technologies. When the implementation of these algorithms fails to be as effective as initially planned, the design of the system becomes an optimisation task. More often then not it is possible to review the implementation of an algorithm to run at the desired effectiveness. This task then saves on total system cost or can reduce the time to market.

This dissertation investigates implementation methods for computation intensive algorithms. These methods include optimising the code for a digital signal processor and optimising the application executing the algorithm on the processor. Another method investigated is implementing the algorithm on programmable logic to provide a hardware accelerated algorithm for the system.

When optimising the code for the signal processor, certain C code optimisations could be done to improve algorithm performance. When the performance gain reached a maximum while optimising the C code, the way the algorithm receives data can be optimised to further the overall application optimisation. Also by implementing the algorithm on programmable logic, such as a Field Programmable Gate Array, greatly improves the effectiveness of the algorithm since the hardware's intrinsic speed is used. However, implementing the algorithm on programmable logic can be a more tedious task than implementing it on a Digital Signal Processor.

Even though significantly optimising the algorithm on the Digital Signal Processor, the desired effectiveness was not achieved. The nature of the algorithm required a constant data stream and this proved difficult to achieve. The Field Programmable Gate Array implementation proved more effective and seems to be the most viable option for this type of algorithm. Even though the programmable logic implementation is the implementation of choice for this algorithm, the research on algorithm implementation on a Digital Signal Processor proves that it is possible to implement an algorithm effectively on cheaper hardware. The hardware accelerated algorithm is always a more effective option, but adds development time to the project.

Uittreksel

Selfs met die vooruitgang van nuwe tegnologie op die gebied van digitale seinverwerking, is dit soms moeilik om gevorderde seinverwerking algoritmes te implementeer. Wanneer die uitvoering van die algoritme nie so effektief is soos wat aanvanklik beplan is nie, word die ontwerp van die stelsel 'n optimaliserings taak. Dit is gewoonlik moontlik om die algoritme implementering te hersien om sodoende die benodigde effektiwiteit te bereik. Dit veroorsaak dat die totale stelsel koste kan daal en/of die tyd na die mark kan verminder.

In hierdie verhandeling word metodes ondersoek om verwerking intensiewe algoritmes volgens 'n effektiewe wyse op digitale hardeware te implementeer. Hierdie metodes omvat die optimalisering van kode vir 'n digitale seinverwerker, sowel as die optimalisering van die beheerstelsel wat die algoritme gaan uitvoer. Ander opsies soos om die algoritme op programmeerbare logika te implementeer, word ook ondersoek.

Tydens die optimalisering van die kode vir die sein verwerker, is dit moontlik om sekere C-kode optimalisering uit te voer om sodoende die algoritme se verrigting te verhoog. Wanneer die maksimum C-kode optimalisering bereik is, word die stelsel geoptimaliseer deur die data op die mees effektiewe manier aan die algoritme te stel. Deur programmeerbare logika soos 'n "Field Programmable Gate Array" se intrinsieke spoed te benut, is dit ook moontlik om die effektiwiteit van die algoritme te verhoog. In vergelyking met die digitale seinverwerker opsie, is die ontwerp tyd vir die programmeerbare logika baie langer en soms moeiliker om te ontfout.

Alhoewel die geoptimaliseerde digitale seinverwerker kode meer effektief as die oorspronklike kode uitgevoer word, was die verlangde effektiwiteit nie bereik nie. 'n Konstante data stroom is nodig vir effektiewe algoritme werking, maar vir die digitale seinverwerker stelsel kon so 'n konstante data stroom nie verkry word nie. Die programmeerbare logika stelsel is om hierdie rede meer effektief en die beter opsie vir die tipe algoritme. Nie te min dui die digitale sein verwerker studie aan dat algoritmes geoptimaliseer kan word om op goedkoper hardeware uitvoerbaar is. Die hardeware versnelde opsie sal altyd die spoed effektiefste opsie wees, maar vermeerder die ontwikkelings tyd.

Preface

The signal processing market is ever demanding more feature rich applications and sophisticated algorithms. These applications usually run multiple computation intensive algorithms and in so doing, require faster, more feature rich processors to run them. As hardware becomes increasingly more complex, so to does the development of optimised applications/algorithms.

To reduce costs and time to market it is important to choose the most appropriate piece of hardware for the application. When the most cost effective hardware is chosen, the application implementation becomes an optimisation task. In this project the aim is to provide methods to effectively optimise and implement computation intensive algorithms on digital hardware.

I would like to thank Prof. W.C. Venter, my mentor, for guiding me through my studies. I have learnt so much from him. This dissertation is dedicated to my parents.

Thank you.

Table of Contents

List of Tables

List of Figures

Code listings

Abbreviations

ADC	Analog-to-Digital Converter
ALU	Arithmetic Logic Unit
BBC	British Broadcasting Corporation
BIOS	Basic Input and Output
CCS	Code Composer Studio
CD	Compact Disk
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
dB	Decibel
DCP	Data Converter Plug-in
DSK	DSP Starter Kit
DSP	Digital Signal Processor
EDMA	Enhanced Direct Memory Access
EEPROM	Electrically-Erasable Programmable Read-Only Memory
EMIF	External Memory Interface
EVM	Evaluation Module
FIFO	First-In-First-Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
HDL	Hardware Description Language
I/O	Input-Output
IC	Integrated Circuit
IDE	Integrated Development Environment
ISR	Interrupt Service Routine
JTAG	Joint Test Action Group
kb/s	Kilo Bits per Second
LE	Logic Element
LOG	Logging Object
MHz	Mega Hertz
MSB	Most Significant Bit
PC	Personal Computer
PLL	Phase Locked Loop
PPD	Pulse Peak Detection
RF-ID	Radio Frequency Identification
SIMD	Single Instruction, Multiple Data
STS	Statistics Object

UART	Universal Asynchronous Receiver-Transmitter
UAT	Universal Asynchronous Transmitter
USB	Universal Serial Bus
VHDL	Very-High-Speed Integrated Circuit, Hardware Description Language
VLIW	Very Long Instruction Word

Chapter 1

1 Introduction

Radio Frequency Identification (RF-ID) detection systems are mostly analogue systems. These systems are very sensitive to noise interference. A more robust digital detection system, intended to replace an existing analogue system, was developed at North-West University, Potchefstroom Campus. The digital system comprised of a detection algorithm running on digital hardware. This algorithm is proven to be more robust in detecting RF-ID tags in noisy signals than the previous analogue system.

1.1 Background and objectives

In the original digital system, the RF-ID tag detection algorithm was implemented on a Texas Instruments TMS320C6713 floating-point digital signal processor. The algorithm, although being more robust for noisy signals, is computationally expensive. The floating-point processor was not adequate for detecting tags in a noisy environment or for detecting a large amount of tags at a time. The reason for this is that the floating point processor was too slow in processing a data buffer. This caused the analogue-to-digital converter's buffer to over-flow and tag data was lost in the process.

The analogue-to-digital converter used in the digital system is fixed-point and the algorithm's structure only uses fixed-point arithmetic for its calculations. These two facts lead to a systems review and the processor was changed to a TMS320C6416 fixed-point digital signal processor. This greatly improved the digital system, but the desired efficiency was still not reached.

By optimising the algorithm and utilising hardware specific features, it was possible to improve the digital system and algorithm to perform as well as or even better than the analogue system. This document investigates methods of algorithm optimisation for digital signal processors (DSPs) and system optimisation when using specific hardware peripherals. Techniques for C code optimisation on the algorithm and hardware specific code optimisation are investigated. The hardware specific code optimisations include the use of hardware features for system optimisations.

These optimisations greatly improved the DSP based digital system, but the system still failed to be as efficient as the analogue system.

After another system review it was decided to hardware accelerate the digital system. Hardware accelerating algorithms is another way to increase an algorithm's performance. When using programmable logic, such as a field programmable gate array (FPGA), the algorithm performance can be significantly improved. The FPGA based digital system is not only a C code porting task but the interfacing peripherals also need to be designed and implemented. One of the advantages of the FPGA system is that it is able to utilise a point processing method, see page 12, for processing the RF-ID data. As long as the sample frequency is lower than the intrinsic speed of the FPGA, no data will be lost and the system will be able to perform as well as the analogue system.

Even though a specific algorithm is used, the dissertation tries to be as universal as possible. That is, the techniques mentioned in the dissertation can be applied to any type of algorithm.

The objective is to optimise a digital signal processing algorithm to run on digital hardware. This includes optimising C code for a DSP based system and to implement the algorithm on programmable logic.

1.2 Methodology

The idea is first to optimise the TMS320C6416 DSP implementation of the detection algorithm with existing C code optimisation techniques. After achieving basic C optimisation, the hardware specific optimisations are implemented. These optimisations, combined with the use of hardware specific features provide a total system optimisation.

The algorithm is also implemented on a FPGA and is tested against the analogue and TMS320C6416 DSP systems. This hardware accelerated version of the algorithm tries to overcome some of the downfalls of the TMS320C6416 DSP system, such as using a point processing method instead of the TMS320C6416 DSP block processing method. In the FPGA implementation the control and interfacing peripherals are also designed and implemented.

1.3 Report organization

The dissertation is organised in three main parts: background information, implementation and testing.

Chapter two handles all the background information on the current algorithm implementation as well as some background information on the hardware used in the project.

Chapter three is a discussion on code optimisation for TMS320C6416 DSPs and also contains some information on the benefits of hardware accelerating an algorithm. The development tools used for the different hardware architectures are also discussed.

Chapter four describes the revised TMS320C6416 DSP algorithm implementation and illustrates the optimisation method proposed in chapter three. Chapter five is a discussion on the implementation of the hardware accelerated algorithm on a FPGA.

Chapter six tests the analogue system (the baseline), the TMS320C6416 DSP system and the FPGA system against each other. Also, the different optimisation techniques are compared to the first TMS320C6416 DSP implementation to show that it actually makes a speed difference. This chapter also tests the functionality of the FPGA based system with simulation software.

The dissertation is concluded in Chapter seven, which summarises the results and the conclusions of the preceding chapters.

Chapter 2

2 Digital System Specification

From a software developer's point of view it is important to know what type of hardware is used in the system and how it is connected. Knowledge of the features of the different hardware blocks and how to effectively use them will result in a hardware specific optimised application. This chapter also focuses on the manner in which the data is presented to the TMS320C6416 DSP, the TMS320C6416 DSP itself and the algorithm.

FPGA's are discussed and how they are used to implement an algorithm to optimise the system. This technology provides system specific logic that can improve the efficiency of digital systems.

Most importantly, this chapter familiarises the reader with the pulse peak detection algorithm and tries to explain its inner workings.

2.1 Converting from analogue to digital

With the current analogue RF-ID tag detection system it is difficult to detect ID tags in noisy environments. Some examples of noise that affects the analogue system are: switch mode power supply noise (typically from laptop or desktop personal computers) and BBC broadcasting signals.

Adding analogue filters to this system would deform the incoming RF-ID signal to such an extent that the system would not be able to detect the tags. This deformation is caused by the phase shift induced by the analogue filter. A solution to this problem is to process the signal digitally. An algorithm was developed at the North-West University to combat this problem [16].

One advantage of detecting the RF-ID tags digitally is that a finite impulse response (FIR) filter could be added to the system to reduce noise components affecting the detection rate. The algorithm developed for the digital detection is of such a nature that it is more robust against noisy signals.

2.2 Pulse peak detection algorithm

The pulse peak detection (PPD) algorithm developed by C. Vorster [16], detects events in a sampled signal. In this case events are defined as RF-ID bit pulses. When these events are ordered like the pulses a tag would transmit, the algorithm detects the tag. The data is then checked for errors and encoded to be transmitted to a host PC.

Before looking at the PPD, it is necessary to understand the manner in which the ID data is encoded and what the packet data looks like. It is vital to understand the structure of the tag data to understand the PPD.

Currently the system should be able to detect tags with data rates of 64 kb/s and 128 kb/s, and in the future 256 kb/s tags.

2.2.1 Tag data glitch encoded format

The tag ID data packet is structured into a start, sync and data bit format. These bits are glitch encoded to reduce the energy needed to transmit the ID bits. The glitch encoding scheme

encodes a logic ONE in only the first quarter of a bit period and a logic ZERO in the third quarter of a bit period, as shown in *figure 2.1* [6]. The figure also shows the data packet format of the ID.
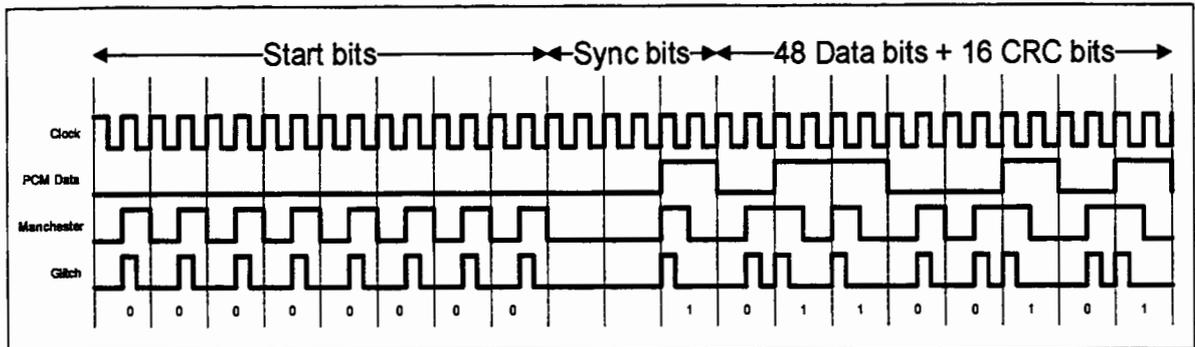
Figure 2.1 – Glitch encoded ID data packet

The total number of broadcasted bits is 75. The first eight logical ZERO bits are the start bits. The start bits are followed by three synchronisation bits that consist of two dead bits followed by a logical ONE bit. The synchronisation bits are followed by the 64 data bits of which the last 16 bits are the cyclic redundancy check (CRC) bits.

Cyclic redundancy checks are used to verify data over a transmission medium. The RF-ID tag's CRC bits are generated using the CRC-16 generating polynomial, $X15 + X2 + X0$ [6]. The CRC algorithm implementation is a standard bitwise CRC-16 algorithm and an example is readily available on the internet.

2.2.2 Peak detection

First of all, the sampled signal is passed through an averaging filter to minimise high frequency noise. Thereafter peaks in the sampled signal are detected using an event threshold method [16]. As the name suggests, the event threshold method is based on a threshold value that is continuously updated from the sampled data bits. To start, this method calculates the maximum of the first hundred data points in the data buffer and adds a fixed value to obtain the start threshold value. The fixed value is added to prevent false peak detections.

It is assumed that in the first hundred data points of the buffer, or after a successful tag read, the buffer has no tag data. A sampled value exceeding the threshold after a hundred data points is considered a peak. The threshold is reset after a tag ID is detected correctly or after the next hundred data points. The value of the threshold in a sampled signal is demonstrated in

After detecting the eighth start bit, the same "look forward" method is used to detect the synchronising bit. The difference here is that instead of looking forward one bit period for an event, the algorithm looks forward three bit periods where the synchronising bit is located. A maximum (greater than the threshold) value in the first quarter of the bit period is assumed to be the synchronising bit. This assumption is based on the uniqueness of the start bits.

2.2.4 Data bit detection

The synchronisation bit indicates the starting position of the data bits and the data bits are detected using the same method as that of the start bits. The exception with the data bits is that the next type of bit is unknown. Thus, the comparison is done by comparing both for a ONE-ZERO bit order and a ZERO-ONE bit order.

From the start bit, the algorithm looks forward in the first quarter of the next bit period for a logic ONE and in the same bit period's third quarter for a logic ZERO. The value found at the third quarter is subtracted from the value found in the first quarter. A logic ONE is detected if the result is positive because the energy lies within the first quarter of the bit period. The opposite accounts for a logic ZERO. This is also the case when looking forward from a logic ZERO. The difference between looking forward from a logic ONE and looking forward from a logic ZERO is the distance the algorithm looks forward. This is illustrated in *figure2.4*:
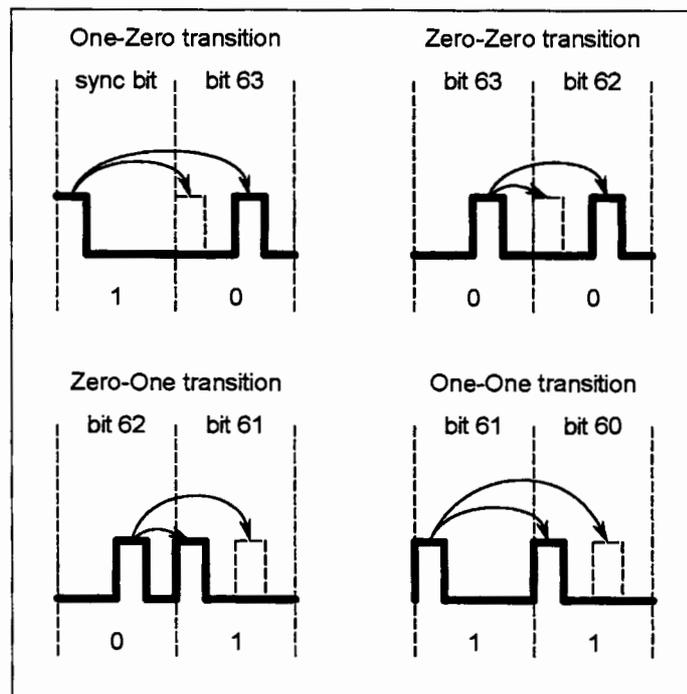
Figure 2.4 – Detecting tag data bits

2.2.5 Look forward distance

This distance is calculated from the sampling frequency and the tag bit rate using $N = F_s/F_t$. With N the number of samples per bit period, F_s the sample frequency and F_t the tag bit rate.

The distance from a ZERO bit to another ZERO bit is equal to N. This is also the case for the distance between a ONE bit and the following ONE bit. The distance from a ONE bit to a ZERO bit is $N + N/2$. The ZERO-ONE distance is $N - N/2$. As shown in *figure 2.1* the synchronising bit is spaced two bit periods away from the last start bit. The look forward distance for the synchronising bit is the distance of a ZERO-ONE bit plus $2N$ or $3N - N/2$.

Each distance has two markers. One marker marks the beginning of a bit pulse and another marks the end. The maximum value between these markers will yield the pulse peak. This ensures that each pulse's peak is compared with where another pulse peak is or should be. The first marker is three data points less than the peak position and the second marker is three points more than the peak.

2.2.6 Error checking and host communication

After a complete tag ID is detected, the PPD algorithm calls the CRC algorithm mentioned above to determine whether or not the ID was detected correctly. If the ID has no errors the ID array is converted to a character array of hexadecimal values. The character array is then sent to the host PC through the host communications block.

2.3 Digital signal processor implementation

In the first stages of the digital RF-ID detection system the algorithm was implemented on a TMS320C6713 floating-point processor. This implementation could detect 64kb/s tag rates relatively well. The system, however, was required to detect tags with tag rates of 128kb/s and 256kb/s. The floating-point system did not leave room for scalability, meaning that the floating point DSP was not able to handle higher tag rates because of the smaller amount of time available between samples. Also, some analogue RF-ID detection systems have two channels from which it detects tags. With two channels to detect tags from and having to filter the data proved too much for the floating-point processor.

A system review provided certain facts about the digital tag detection system. Firstly, all the

algorithm calculations are fixed-point calculations and secondly, the analogue-to-digital converter (ADC) is a fixed-point data converted. This resulted in a move to a fixed-point TMS320C6416 DSP, discussed later in this chapter. A fixed-point processor is generally faster than its floating-point counterpart and this already significantly improved the digital tag detection system. Even though the fixed-point system significantly improved the detection rates, it was still not adequate for future system upgrades like higher tag rates, the addition of a FIR filter and a second detection channel.

The TMS320C6416 DSP based system uses a block processing scheme to process the incoming tag data. What this means is that the ADC fills a buffer and the DSP then processes the data in the buffer. While this buffer is being processed, the ADC fills another buffer and waits for the DSP to start processing it. During this waiting period, data is lost and the tag detection rate falls to well below that of the analogue system.

It is, however necessary to have a good understanding of the hardware used in the TMS320C6416 DSP based digital system when considering optimisation. The hardware that is discussed next, is the THS1206 ADC used to digitise the RF signal and naturally the TMS320C6416.

2.3.1 Analogue to digital conversion

In the digital RF-ID detection system, the signal is digitised using a THS1206 analogue-to-digital converter. This is a 12-bit ADC with the following features:

- high-speed 6 MSPS (mega samples per second) ADC;
- 4 analogue Inputs;
- signal-to-noise and distortion ratio: 68 dB at fl = 2 MHz;
- glueless parallel μC/DSP interface;
- integrated 16 word deep FIFO (first-in-first-out) buffer.

The FIFO buffer has a configurable trigger level, which means it can be set to a level that is most efficient for the system it is used for. The sample speed is controlled by the host hardware [14]. The ADC is mounted on an evaluation board and connected to the development board via a daughter card. The ADC receives its digital supply voltage from the development board and the analogue supply voltage is provided by an external 5 volt power supply or by connecting the 5 volt digital supply to the analogue supply. The latter setup is not recommended as the digital supply could add noise to the analogue system. The daughter board mounted evaluation board

is shown in *figure 2.5.*
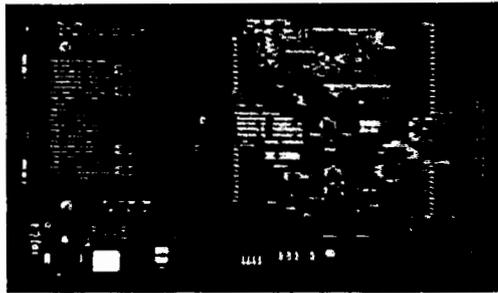
Figure 2.5 – THS1206 evaluation board with daughter card

Detailed information about the THS1206 ADC and the ADC evaluation module (EVM) can be found in [14] and [12] respectively. The previous digital system had to sample the analogue signal at 1.536 MSPS for a tag rate of 64kb/s and 3.072 MSPS for a tag rate of 128kbit/s. This gives the detection algorithm ±24 data points per bit period and 6 points per bit pulse.

2.4 TMS320C6416

The TMS320C6416 used in the digital detection system ineffectively detected the RF-ID tags with the current un-optimised algorithm and two main optimisation techniques are considered for improving its effectiveness; software optimisation and hardware optimisation. Both are discussed later in the text. This section gives a background on the hardware options.

2.4.1 The Texas Instruments digital signal processor

The TMS320C6416 DSP is mounted on a development board and is clocked at 600 MHz. The DSP is a fixed point very-log-instruction-word (VLIW) processor with TIs VelociTI.2™ architecture. The VelociTI.2™ expands on the previous VelociTI™ architecture by including single instruction multiple data (SIMD) processing capabilities [1].

The TMS320C6416 has 64 32-bit general-purpose registers and two fixed point data paths and each path has four functional units [11]. That is eight functional units consisting of:

- six arithmetic logic units (ALU's) supporting single 32-bit, dual 16-bit or quad 8-bit arithmetic instructions per clock cycle; and
- two multipliers supporting four 16 × 16-bit multiplies with 32-bit results or eight 8 × 8-bit multiplies with 16-bit results, per clock cycle.

The TMS320C6416 DSP supports packed data processing and has a 16-kB L1 program cache, a 16-kB L1 data cache and a 1024-kB L2 unified mapped RAM/cache. The C6416 also has an eleven-stage pipeline and three 32-bit timers. The pipeline architecture of the C6416 processor enables it to have greater data throughput and faster processing times. These features make the DSP ideal for processing huge amounts of data as effectively as possible.

The TMS320C6416 DSP includes advanced on-chip peripherals that, if used correctly, could improve overall system performance. These peripherals include the Enhanced Direct Memory Access (EDMA) controller and External Memory Interfaces (EMIF) (for a complete list visit www.ti.com).

2.5 Field programmable gate array implementation

As stated before, the TMS320C6416 DSP implementation did not reach the efficiency required for the digital tag detection. Other hardware platforms were considered and a decision was made to replace the TMS320C6416 DSP with a FPGA. The FPGA is an acceptable hardware alternative because the algorithm is implemented in such a way that different sections run in parallel. This then replaces the block processing scheme with a point processing scheme. In the point processing scheme each sampled value is processed before the next sample is received. The FPGA is capable of utilising the point processing scheme because each processing block runs at the speed of the FPGA and independently from the other blocks.

Another reason for the transition from TMS320C6416 DSP to FPGA is that the FPGA is scalable. The peripheral control blocks used in the system use a small amount of logical units on the FPGA. Thus, adding another ADC to the system is only a matter of adding it to the board and reprogramming the FPGA. Adding a FIR filter to the FPGA implementation also only requires the FPGA to be reprogrammed.

All these factors will eventually enable the system to process higher tag rates, more effectively. This is discussed in the FPGA implementation chapter. The FPGA of choice is the Altera Cyclone range of FPGAs.

2.5.1 Altera Cyclone FPGA

The Cyclone range of FPGA's from Altera is a low cost, high logic element (LE) density

alternative to Application Specific Integrated Circuit (ASIC) designs [2]. The Cyclone, specifically the EP1C6, has the following features:

- 5,980 LE's;
- 20 MK4 RAM blocks, translating to 92 160 RAM bits;
- two phase lock loops (PLL);
- 185 user I/O pins which includes global clock pins; and
- is serial configuration device (EEPROM) configurable [3].

Programmes for FPGA's are written in a hardware description language (HDL). The HDL describes the circuit that is synthesised on the FPGA. Altera offers an IDE that compiles and synthesises the HDL for the FPGA. It also includes the programming software needed to program the chip and run the code.

Chapter 3

3 Algorithm optimisation considerations

This chapter firstly looks at how the developer can write/rewrite the algorithm in C code to produce a more effective compiler generated assembler for the TMS320C6416 DSP. After the algorithm is optimised to satisfaction, the developer can then focus on optimising the rest of the application. This is done by using hardware specific peripherals to optimise data throughput to the central processing unit (CPU) for processing.

Hardware accelerating the algorithm and the considerations that are made when implementing the algorithm on a FPGA is also discussed. Furthermore, the development tools are introduced in this chapter.

3.1 Software optimization

Developing optimised algorithms on digital hardware is a three phase procedure [10]. The first phase is implementing the algorithm in C. If this is sufficient then the design is complete. This is seldom the case and in the next phase the C code is refined (optimised). When phase two fails to be sufficient; the final phase is rewriting the code in linear assembly. This phase is, however, a last resort and is not within the scope of this dissertation.

C programmes are made up of data structures, control structures and functions. Data structures represent the data needing processing or variables that control the control structures. Control structures are used to direct program flow such as loops. Loops are more often than not the part of a function or program that takes up most of the CPU time.

3.1.1 Loop optimisation

As stated, loops are the most common code structures that reduce efficiency. This means that if a loop executes faster the overall performance of the code is increased. Loop optimisation is done by implementing the following techniques [8]:

- Combine loops: When two functions should be executed a certain number of times it is best to avoid the extra loop overhead by executing both functions in one loop, as shown in *code listing 1*. Keep in mind that if the executed instruction do not fit into the platforms instruction cache it would be better to keep the loop apart.

```
High cycle count                    Lower cycle count

for(i=0; i<x; i++)                  for(i=0; i<x; i++)
{                                   {
      foo1();                             foo1();
}                                         foo2();
                                    }
for(i=0; i<x; i++)
{
      foo2();
}
```

Code listing 1 – Loop combining

- Loop unrolling: As stated before loops can be slow. This is because the loop needs to

check and increment/decrement the loop iterating value on each pass. By unrolling the loop, the loop overhead is minimised or removed completely. If the loop contains control structures it is hard to unroll and in some cases not applicable. In *code listing 2* a normal coded loop is shown and compared to an unrolled loop. This is however only valid for loop with small iterations.

```
         Normal loop          Unrolled loop

for(i=0; i<3; i++)              foo(0);
{                               foo(1);
     foo(i);                    foo(2);
}
```

Code listing 2 – Loop unrolling

- Reverse iterated loops: Consider a loop where the direction of the iteration is not important or where the data being accessed is ordered in reverse before entering the loop. An optimisation can be made by decrementing the loop iteration value instead of incrementing it as shown in *code listing 3*:

```
Normal loop      for(i=0; i<x; i++){...}

Faster loop      for(i=x; i>0; --i){...}
```

Code listing 3 – Faster loops

The loop over head for the normal loop is a subtraction, a compare and a decrement. The loop has to subtract 'i' from 'x' and compare the result to zero. If the result is not zero; the next loop iteration is executed. As for the reversed iterated loop, 'i' is only compared to zero and if the result is greater than zero, the next loop iteration is processed. No subtraction instruction is used in the loop. This greatly improves the performance of tight loops.

3.1.2 Decision based statement optimisation

Decision control structures divert the program flow to another branch in the program. Most programmers fail to realise that the careful coding of these structures could boost the

applications performance. This performance boost may seem insignificant on its own, but it becomes apparent when a decision is made several times in a loop. The 'switch' and 'if' control structures are discussed here.

3.1.2.1 Switch statement

When the switch statement is used, be sure to keep the case labels in as small a range as possible. This causes the compiler to generate a jump table with the case labels. This is considerably faster than that of an if-else-if cascade code generated if the case labels are far apart [7]. For example a switch statement with the conditions 1, 50 and 10 causes the compiler to generate the if-else-if cascade code. But if the case labels are 1, 2 and 3 the compiler generates a jump table of the case labels.

Another simple optimisation technique for switch statements is to place frequently accessed case labels at the top of the case statements to insure that an early break will occur most of the time. At run time the number of comparisons is on average less, thus providing, on average, an increase in performance. When the switch statement is big, create a nested switch with the more frequent labels in the outer switch and the less frequent in the inner switch. This is demonstrated in *code listing 4*:

```
Standard Switch statement          Early Exit Switch
switch (x)
{                                  switch (x)
 case FrequentX1:                  {
    DoSomething();                 case FrequentX1:
    break;                            DoSomething();
                                      break;
 case FrequentX2:
    DoSomething();                 case FrequentX2:
    break;                            DoSomething();
                                      break;
 case InFrequentX1:
    DoSomething ();                default:
    break;                            switch (x) //Nested for infrequent
                                      {
 case InFrequentX2:                   case InFrequentX1:
    DoSomething ();                      DoSomething();
    break;                               break;
}
                                      case InFrequentX2:
                                         DoSomething();
                                         break;
                                      }
                                   }
```

Code listing 4 – Improved switch statements

3.1.2.2 IF-ELSE statements

Another performance increase could be gained when an if-statement has more then one condition [5]. Assume for the illustration that condition 1 is the most likely to be true whereas condition 2 is the most likely to be false. When using the || (or) operator, checking the most frequent true condition first could improve executions time (*code listing 5*):

```
Normal IF                          More efficient IF

If(condition2 || condition1)       if(condition1 || condition2)
{                                  {
   dosomething()                      dosomething()
}                                  }
```

Code listing 5 – Multiple conditions using || operator

If the conditions are independent of one another when using the && (and) operator, checking the least frequent true condition first could improve executions time (*code listing 6*):

```
Normal IF                          More efficient IF

if(condition1 && condition2)       if(condition2 && condition1)
{                                  {
   dosomething()                      dosomething()
}                                  }
```

Code listing 6 – Multiple conditions using && operator

The if-else-if statement can be structured as to promote early exiting. By placing the most common true condition as the topmost if-statement's parameter, early exiting is achieved and the average cycle count is lowered [5]. This is shown in *code listing 7*.

```
Normal if-else                     Efficient if-else

if(infrequent_condition1)          if(frequent_condition)
{                                  {
   dosomething2();                    dosomething1();
}                                  }
                                   else if(frequent_condition)
if(frequent_condition)             {
{                                     dosomething2();
   dosomething1();                 }
}                                  else
                                   {
if(infrequent_condition2)             dosomething3();
{                                  }
   dosomething3();
}
```

Code listing 7 – Nested if-else statement

3.1.3 Registers

It is preferred that the number of local variables are less than or equal to the number of processor registers. When variables are on the heap it is accessed faster than external memory accesses and the compiler does not need to incur the overhead of setting and restoring the frame pointer [7].

This is done by declaring the variables as global or as static within a function. Current compilers can invoke register optimisation when a variable is declared with the register keyword. These variables can also be reused for variables that are mutually exclusive. This leads to faster code because the variables are always held on the heap and thus accessed faster.

3.1.4 Miscellaneous optimisations

Always try to use the processors default word length for arithmetic [7]. C uses integers for arithmetic operations and parameter passing, and has to convert other data types to integers before doing the requested operation. It should be noted that for DSP's the processor's vendor usually supplies an optimising C compiler that will optimise the code for the word length of the hardware.

The trade off between memory and speed should always fall in favour of speed. Unless the system is low on memory, storing often used data is an easy way of removing redundant operations.

3.2 Hardware Optimization

Texas Instruments has developed an optimising ANSI C compiler for its range of digital signal processors. This compiler does all the strenuous work for the programmer such as instruction selection, parallelizing, pipelining, and register allocation [10]. But the compiler cannot do everything for the programmer on its own. The programmer still needs to direct the compiler in the best possible direction. This includes the following considerations:

- In *table 1* the data type sizes, as defined by the C6x compiler, are show. Avoid code that assumes that the integer and long data types are the same. All 40-bit operations are done with the long data type.

Data type	Size
(unsigned) char	8bit
(unsigned) short	16bit
(unsigned) int	32bit
long	40bit
float	32bit
double	64bit

Table 1 – C6x compiler data type sizes

Use the short data type for the 16-bit multiplier. The multiplication is done in one clock cycle compared to five cycles for an integer multiplication. Use the int or unsigned int data type for loop counters to avoid unnecessary sign extension instructions.

- Compiler settings: Always use the highest optimisation level the compiler can perform. After profiling the code the programmer will be able to know the exact cycle count for his code. This will make it easier to optimise the critical code sections.

- Use intrinsic functions: Texas Instruments has included intrinsic functions in the compiler to allow for easy access for inline instruction. These intrinsic functions are sets of instructions not easily expressed in C code and give the programmer a quick way to optimise the code.

- Wide memory access on data: Some of the intrinsics mentioned above are used to do a 32-bit access on two consecutive 16-bit memory locations. This doubles the speed of operation in code that needs for two calculations to be performed on consecutive memory spaces. This is called packed data processing.

- Software Pipelining: Loops take up most of the processors time. Software pipelining attempts to schedule instructions in such a way that some iterations of the loop execute in parallel. This is a form of loop unrolling performed by the compiler.

Remember that these optimisation and those in 3.2 can only be implemented if the code allows it.

3.3 Using the hardware peripherals

Most digital signal processors have on-chip peripherals that increase data processing and accessing speeds. Using these peripherals can greatly increase the efficiency of an application and the manner in which the algorithm gets the data. As stated before the TMS320C6416 has an EDMA, EMIF and the timer. These three peripherals are used in the current digital system to provide reliable communications with the ADC. The peripherals are set up using a data converter plug-in. This plug-in provides the accurate setup of the EMIF, the timer and a single transfer EDMA setup. It is, however, important to mention that the EDMA transfer can be optimised.

3.3.1 Enhanced direct memory access

The C6416 EDMA controller handles data transfers between the L2 cache memory controller and peripherals. It has a number of features such as 64 channels, programmable priority and link or chainable transfers. The EDMA can also move data to/from any addressable memory space [13].

The previous TMS320C6416 DSP system used the EDMA only to move data from the ADC to a memory buffer. This meant that the EDMA had to be initialised for every transfer and terminated after each successful transfer. Consequently, this is ineffective because of the overhead caused by each initialisation of the EDMA. A more effective way to use the EDMA is to use its link feature. *Figure 3.1* shows the parameter table of the EDMA used to initialise an EDMA transfer:

31	16	15	0	
Options (OPT)				Word 0
SRC Address (SRC)				Word 1
Array/frame count (FRMCNT)		Element count (ELECNT)		Word 2
DST address (DST)				Word 3
Array/frame index (FRMIDX)		Element index (ELEIDX)		Word 4
Element count reload (ELERLD)		Link address (LINK)		Word 5

Figure 3.1 – Parameter table for an EDMA transfer [13]

In this figure the structure of the parameter RAM table is shown. The options section holds information about the type of transfer. This includes the priority, the data type (8-, 16- or 32-bit),

the interrupt number and the link state, the transfer complete code, and other options. The source address is that of the ADC and the destination is that of the data buffer. The element count is the number of elements per frame and the frame count is equal to the ADC FIFO trigger level. The element index and frame index is set to one and the element count reload value is set to zero.

The link address allows the EDMA to preload the next parameter RAM table. This allows the next transfer to start without the initial overhead of the table setup. Furthermore, this also allows the CPU to process data while the EDMA handles the next data transfer.

3.4 Hardware accelerated algorithm

If the C code optimisation fails to produce the desirable efficiency, other options need to be investigated. Hardware acceleration is becoming a more favoured option for imbedded systems design. The term hardware acceleration, in this case, can be defined as replacing the software algorithm with an external hardware peripheral [9]. This will utilise the hardware's intrinsic speed to benefit the overall system speed.

The software version of the PPD algorithm uses a block processing method. That is, the algorithm works on a fixed buffer size at each pass. This method works well and is effective if the data rate that fills the buffer is much lower than the rate of processing each block. If this is not the case some data may be lost and the algorithm's effectiveness is compromised. Point processing on the other hand processes each incoming data point before receiving and processing the next data point. This can be effective but requires a lot more processing power.

After deciding on the use of a hardware accelerated setup, point processing is the desired method and a type of pipelined processing scheme needs to be implemented. This pipelined scheme is illustrated in *figure 3.2*:

			simultaneous processing	block 3 starts processing	block 2 starts processing	block 1 starts processing
point 4	point 3	point 2	point 1			
	point 4	point 3	point 2	point 1		
		point 4	point 3	point 2	point 1	
			point 4	point 3	point 2	point 1
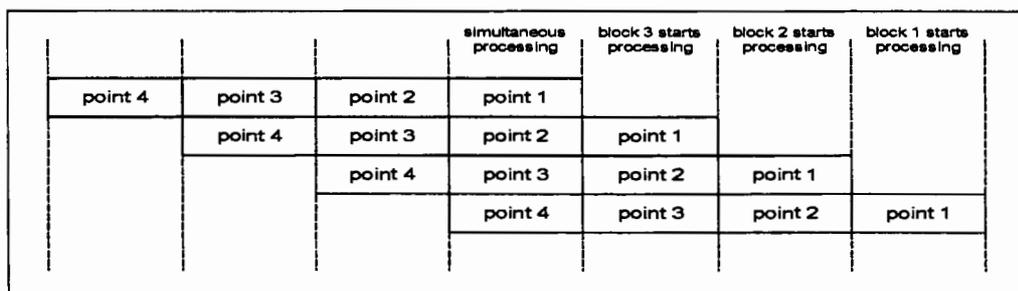
Figure 3.2 – Pipelined point processing

Programmable logic, such as the FPGA, is becoming more affordable and easy to use. This project investigates the use of FPGA's and in chapter 5 the details of the implementation are discussed.

3.5 Development tools

The development tools, used to implement the PPD algorithm on both the DSP and the FPGA, help to optimise the algorithm effectively and efficiently on both platforms. These tools, especially the tools for the TMS320C6416 DSP, help to profile and debug the code. All this reduces the time to market as well as the overall project costs.

Most integrated circuit (IC) vendors supply the development tools and the supplied tools are optimised to produce the best code for the hardware.

3.5.1 TMS320C6416 DSP development tools

Texas Instruments has developed an integrated development environment (IDE) called Code Composer Studio (CCS). CCS combines all the tools needed to develop applications with their range of DSP's. CCS also has the ability to run plug-ins that furthers the development of digital signal processing applications.

CCS provides the developer with tools that simplify the debugging tasks associated with DSP development. These include a real-time operating system, the data converter plug-in and the development board.

3.5.1.1 DSP/BIOS

DSP/BIOS is a Texas Instruments developed real-time operating system (OS) that provides an easy to use multithreaded operating system for use with TI DSP's. It provides real-time scheduling and synchronization and host-to-target communication. It also provides hardware abstraction and pre-emptive multithreading.

The multithreaded tasks and software interrupts can be fully synchronised when using task hooks (for tasks) or mailboxes (software interrupts). All three thread types, hardware interrupts, software interrupts and tasks have a priority hierarchy that easily sets the thread execution

priorities. With the multitasking capabilities of the DSP/BIOS OS it is now possible to enhance the PPD algorithm with multiple threads. Because the algorithm is re-entrant, each thread of the algorithm is allowed to finish processing the current data buffer while the next buffer is being processed.

DSP/BIOS also provide an easy means of getting statistics from the CPU and an API (application programming interface) for standard host I/O. This is done with LOG and STS objects. These two objects manipulate the standard output and statistics gathering functions respectively. This feature simplifies the profiling of the code and enables the programmer to tweak the code to his needs.

3.5.1.2 Data converter plug-in

The TI data converter plug-in (DCP) is an easy to use wizard that generates the ADC source code. The tool is used to setup the user defined control registers of the ADC and the TMS320C6416 DSP peripherals used by the ADC. A screenshot of the DCP settings is shown in *figure 3.3*:
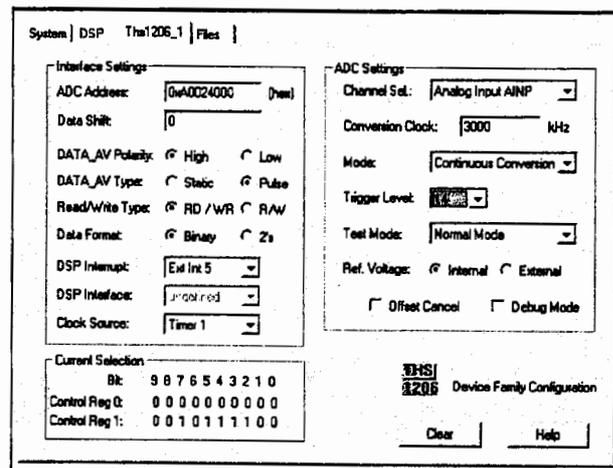
Figure 3.3 – ADC control registers settings

The interrupt service routine (ISR) uses a dispatcher, which is selected in the DCP DSP settings tab. The dispatcher allows the ISR to start other threads without waiting for them to return. This, in turn, allows the ISR to exit early and wait for the next interrupt as apposed to missing one.

The DSP timer and EMIF is set up with for the most effective settings and allows the programmer to focus on other parts of the application.

3.5.1.3 DSP development board

The DSP digital system is implemented on a development board that handles all host communications through USB. This board, illustrated in *figure 3.4*, also enables on-chip debugging through JTAG (Joint Test Action Group) boundary-scan technology and other non application related host I/O. The debugging capabilities allow the programmer to quickly and easily address problems in the algorithm being optimised.
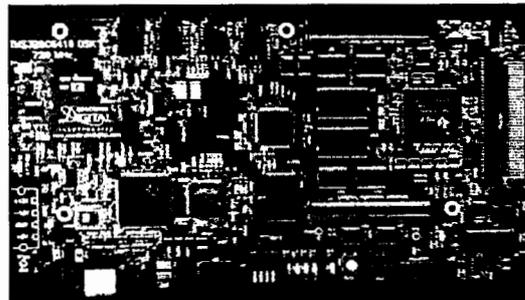
Figure 3.4 – TMS320C6416 DSK

Combined with the CCS IDE and the THS1206 ADC; the development board acts as a complete digital RF-ID tag detection system.

3.5.2 FPGA development tools

The Altera Quartus II v4.1 IDE allows a programmable logic designer to visualise the design. This makes it easy to view the system connections and interconnections. For each module block the designer can generate the hardware description language (HDL) code. Quartus supports four different HDL's, but VHDL (very high speed integrated circuit hardware description language) is the language of choice.

Quartus also allows the assignment of I/O pins for a specific device to conform to the HDL design. A system clock specification can also be assigned to the project and allows the designer to determine whether or not the design will reach the timing requirements with that specific clock setting.

3.5.2.1 FPGA development board

Quartus also has built-in programming software and uses the USB-Blaster USB programmer to program Altera devices. A FPGA development board was developed to test the HDL code. This board has the THS1206 ADC and pre-amp stages onboard and uses the MAX232 chip for RS-232 host communications. This board is shown in *figure 3.5*.
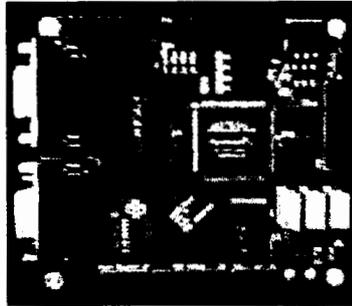
Figure 3.5 – FPGA development board

The Mentor Graphics Corporation created a simulation package for simulating HDL programs, ModelSim Altera 5.8c, which can be regarded as an important tool when designing programmable logic devices. This simulation package enables the designer to visualize the way in which HDL signals work together. Consequently, this reduces the development time by giving the developer the ability to spot errors in the design more quickly.

3.5.3 Support software

To test the different systems, software is needed to display the detected tag data. The TMS320C6416 DSP system uses the CCS IDE to display the tag data as mentioned before. The analogue and FPGA systems, however, require other software to display the tag data.

The analogue system uses a program, ShowTags, developed by iPico. This program is able to display the tags currently being detected, the tag rate in tags per second and to control the analogue reader. This also gives the test baseline for the digital systems.

The FPGA system uses the RS-232 communications protocol to send data to the host (explained in chapter 5). A program, running on the host, capable of serial communication is needed to display tag data. The host application is a modified version of a program developed by Microsoft for demonstration purposes [4]. This program with the modifications is available on

the Appendix CD.

Chapter 4

4 DSP system design

Using some of the techniques mentioned in the previous chapter, the algorithm is again implemented on the TMS320C6416 DSP. This chapter focuses on the C code optimisation and makes a comparison on the changes made in the code. The use of hardware peripherals and the real-time operating system is also discussed.

4.1 Tag detection application

While implementing the optimised algorithm on the TMS320C6416 DSP some considerations had to be revised. These included how the system handled the data acquisition loop, how the data is moved from the ADC to memory and how the host communication is formatted.

4.1.1 Data conversion setup

One of the development tools mentioned is the DCP used with CCS. This plug-in handles the setup of the ADC and the DSP peripherals. These peripherals are the EMIF, the EDMA and the timer.

The timer acts as the external clock for the ADC sampling. When working with the 128kbits/s tags and sampling the signal at a frequency of 3 MHz, the timer period is set to 0x00C or 12. This makes the actual sampling frequency 3.125 MHz. Knowledge of the latter is important for the pulse distance and is discussed later in this chapter.

The main difference in this setup is the removal of the DCP generated ISR and the EDMA setup. With the new EDMA setup scheme, the DCP generated setup became redundant and was removed. The DCP generated ISR had a data shift loop used when the ADC hardware was connected on the Most significant bit (MSB) side of the DSP memory addresses. With the ADC evaluation board this loop is not necessary and it is removed.

4.1.2 EDMA setup

The EDMA is setup to fill a buffer in a "ping-pong" fashion. The link parameter is set up and two identical parameter tables are filled with the settings provided by the data converter plug-in. One slight difference is that the link state is activated and for the "ping" parameter table the link address is set to point to the "pong" table and vice versa. This starts the ADC data transmission and first fills the "ping" data buffer destination. When this buffer is full, the EDMA interrupt service routine is called and the "pong" buffer starts being filled.

These changes along with the changes made in the ADC setup reduced the overall CPU usage. This in turn gave the algorithm more CPU time to perform its calculations.

4.1.3 DSP/BIOS setup

4.1.3.1 Hardware interrupts

The hardware interrupt for the EDMA controller is setup to point to the EDMAIsr() interrupt service routine. The interrupt dispatcher is activated and no mask is applied as shown in *figure 4-1*:
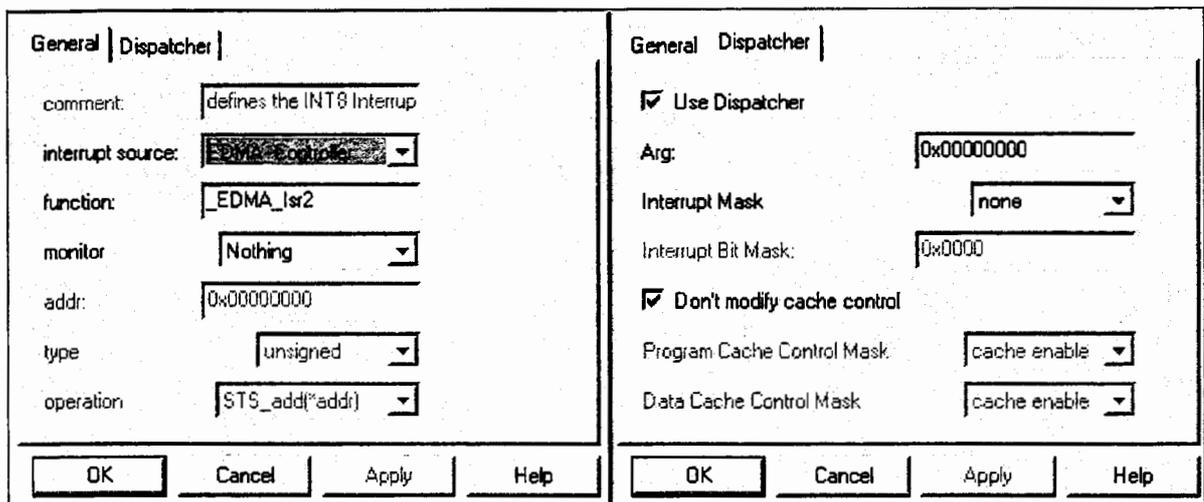
Figure 4.1 – Hardware interrupt setup

The EDMAIsr() clears the transfer complete code to start the next EDMA transfer, switches the "ping-pong" state and posts the software interrupt that handles the detection.

4.1.3.2 Software interrupts

The optimised system incorporates only one software interrupt as apposed to two for the original system. While the original system used an extra thread to switch the buffers, this is now done by the EDMA.

The software interrupt calls the DataSWI() function. This function depending on the "ping-pong" state calls the detection algorithm with either the "ping" or the "pong" buffer as data source. After the buffer is processed the cache memory, which was occupied by it, is invalidated. This is done to ensure that the EDMA will function properly.

4.1.3.3 Statistics and host I/O

The DSP sends successfully detected tag data to the host via USB by calling the `LOG_printf()` function. For this, a LOG object is created in DSP/BIOS. The statistics on instructions and execution times are gathered with a STS DSP/BIOS object. This STS object gathers high resolution time data and the number of instructions executed.

4.2 The optimised PPD algorithm

The techniques discussed in chapter three are used in the optimisation of the PPD algorithm. Loop unrolling is not possible with the type of branching code used in the algorithm, since each branch cannot run independently from another. The branching code, however, could be greatly improved, providing the TI optimising compiler with enough room to produce optimising assembly code.

The initial algorithm, being a research project, was implemented and designed with a stable and robust system in mind, which still remains a high priority specification.

4.2.1 Main loop

The main algorithm loop contains a series of control statements that decide whether pulses are part of tag data or not. These control statements are divided into four processing blocks: the averaging filter, the threshold section, the start and synchronising bits section and the data bits section.

4.2.1.1 Averaging filter section

In the first development stages of the detection algorithm, the algorithm negated the signal during the averaging filter stage. This resulted in extra redundant negation instructions. These instructions are executed in each loop cycle causing unnecessary overhead.

An extra if-statement is removed from the original algorithm that chooses which buffer to use for the filter data. This buffer choice is done on each loop pass. These extra instructions are removed and the buffer is now passed as a referenced parameter to the detection function. The choice is now handled by the EDMA interrupt service routine.

4.2.1.2 Threshold section

The threshold section of the first implementation of the PPD algorithm used a number of if-statements that directed the algorithm from setting up the threshold to processing a possible tag pulse. The if-statements have redundant code not necessary for detecting tags, as shown in *code listing 8*:

```
if(counter1 < 100 && signal < max1 && NaID == 1)
{
  //determine max initial value for threshold
  max1 = signal;
}

if(counter1 == 100)
  NaID = 0;

uplim = max1 - 200; //adding a buffer to the threshold

//looking for possible sync pulses
if(counter1 >= 100 && signal < uplim && Possible == 0)
{
  Possible = 1;
  counter1 = -2;
}
```

Code listing 8 – Threshold control statements

This code listing also shows that an extra redundancy is introduced by adding the buffer to the threshold in the main loop path. The optimised algorithm uses the early exit and nested if-else-if techniques to reduce the code executed in each loop pass. *Code listing 9* illustrates how this was implemented and shows that the buffer added to the threshold is now only added when the threshold is being calculated:

```
if( Possible == 0)
{
  if(counter1 < 100)
  {
    if(signal > max1)
      max1 = signal + 200;
  }
  else if(counter1 > 99)
  {
    if(signal > max1)
    {
      Possible = 1;
      counter1 = -2;
    }
    if(signal > max2)
    {
      max2 = signal+200;
    }
  }
  else if(counter1 > 199 )
  {
    max1 = max2+200;
    max2 = 0;
    counter1 = 100;
  }
}
else
{
  start/sync_bit_detection();
  data_bit_detection();
}
```

Code listing 9 – Optimised threshold section

The most common state of the algorithm is when it is looking for a pulse. This means that for early exiting the topmost condition for the nested if-else-if statement is that no pulse is detected. Removing the `uplim` variable and adding the threshold buffer when looking for the threshold causes a performance gain because the buffer is only added at iterations where the threshold is being calculated and not at each iteration of the mail loop.

4.2.1.3 Start bits and synchronising bit section

The detection of the start bits starts at the detection of a pulse. If these pulses are detected in sequence together with the synchronising pulse, the algorithm activates the data bit section. A further optimisation could be done here by again splitting the if-statement into a nested if-else-if statement for the start/non-start state of the algorithm. When looking for the start bits and the synchronising bit the algorithm is in the non-start state and this is set as the topmost condition. The latter is illustrated in *code listing 10.*

```
if(Possible == 0)
{
  ..
}
else
{
  if(Sart == 0)
  {
    look_for_start_bits();
  }
  else
  {
    look_for_data_bits();
  }
}
```

Code listing 10 – Optimised start and sync bits section

With the 3.125 MHz sampling frequency the TMS320C6416 DSP can provide the ADC, the bit distance is calculated as 24.4140625 rounded to 24. That is a ZERO to ZERO bit, which is 24 data points apart and the synchronising ONE bit is found 54 points after the last start bit.

4.2.1.4 Data bit section

In the original algorithm the data bits were stored in a 65 element array of short data type values. By changing the CRC function to work with two unsigned integer values, it was possible to save memory space and unnecessary memory fetch and store instructions.

The tag data is now split into two 32-bit values and cycle saving bit operations are performed on

these values. The difference between the first method and the new method is shown in *code listing 11*:

```
                New version                  |              Original version

   Diff = compareB - compareA;               |   Diff = compareB - compareA;
   if(Diff > 200)                            |   if(Diff < -200)
   {                                         |   {
     if(IDcounter < 32)                      |     IDcounter++;
     {                                       |     ID[IDcounter] = 0;
       msb <<= 1;                            |     counter1 = i - stoor_iB;
     }                                       |     compareA = 0;
     else                                    |     compareB = 0;
     {                                       |     Bit = 0;
       lsb <<= 1;                            |   }
     }
     IDcounter++;
     ID = 0;
     counter1 = i - stoor_iB;
     compareA = 0;
     compareB = 0;
   }
```

Code listing 11 – Data bits detection section

This section also incorporates the nested if-else-if statements to improve code speed. The bit distances are as follows: a ONE-ZERO bit sequence is 32 points apart and a ZERO-ONE bit sequence is 12 points apart.

4.2.1.5 CRC and host IO

In chapter 2 the tag data bit array had to be formatted to a character array of hexadecimal values. The hexadecimal array is then sent to the host via the DSP/BIOS LOG object. This transformation function is redundant and is removed from the optimised code. The redundancy is a direct result of the new format of the tag data mentioned in the data bits section. The LOG_printf() function is able to send the unsigned integer values to the standard output preformatted to its hexadecimal equivalent.

The new CRC function uses the bit shift and XOR (exclusive or) bit operators to compute the CRC. This approach removed a lot of the redundant "for loops" used in the first implementation, which significantly improved the algorithms performance.

The result of the optimisation techniques on the performance of the digital tag detection application is shown in chapter 6.

Chapter 5

5 FPGA system design

Implementing the algorithm on the FPGA system does not involve the simple task of porting the C code to the VHDL equivalent, but it also requires the implementations of peripheral communications and host port communications.

Each process block on the FPGA is discussed here. It is important to remember that timing is very important in FPGA designs. Getting the timing right when working with waveforms, such as those received by the FPGA from other hardware, will ensure that the system is stable. This becomes apparent in this chapter.

5.1 Overview of FPGA design

Certain things need to be decided on before implementing the PPD algorithm on the FPGA. First, the base clock speed should be able to easily drive the fastest block in the system. This will ensure that the slowest block (the bottleneck block) is able to run effectively. The system speed is then regarded as the speed of the slowest block.

In this case, the slowest block is the host communications block, which runs at 115 200 Hz. The fastest block is the ADC communications block and should be able to drive the ADC at its maximum sample rate of 6 MSPS. The clock speed is chosen at 18.432 MHz. This clock frequency is easily divided to drive the host communications at 115.2 kHz and is able to drive the ADC at full speed.

The FPGA system consists of four primary processing blocks, each with its own sub-blocks. These blocks are the main control block, the ADC control block, the PPD algorithm block and the host communications block, all shown in *figure 5.1*:
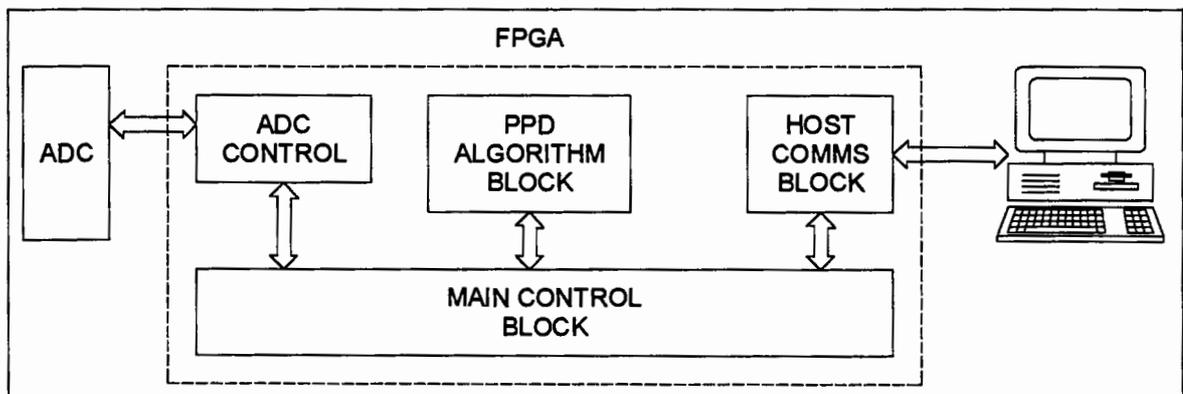
Figure 5.1 – FPGA based digital detection system

The main control block controls the data movement between each of the other blocks and to the output pins. This block also keeps everything synchronised. The ADC control block configures the ADC and starts the sampling process. The PPD algorithm block receives data from the ADC block and does point processing to detect tags. If a tag is detected, the tag bits are processed for transmission and sent to the host PC.

5.2 ADC control block

As the name states, the ADC control block is used to control the ADC and to get the sampled data from the ADC into the system. The ADC firstly needs to be set up to the user's specifications and reset to start the sampling process. The ADC setup flowchart [14] is shown in *figure 5.2*:
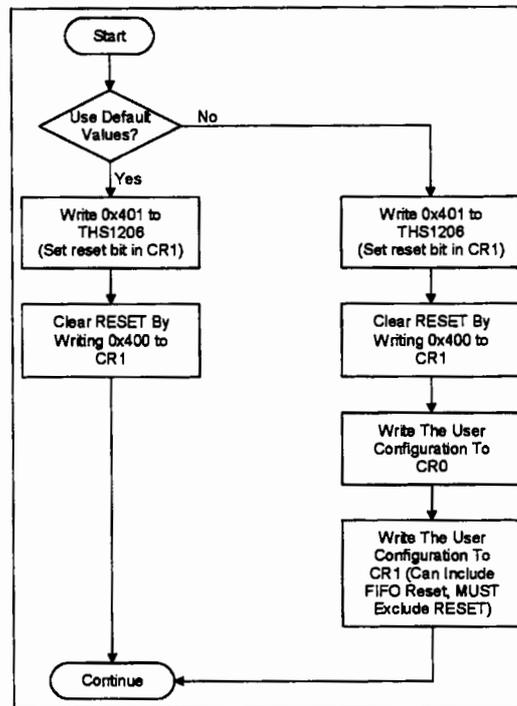
Figure 5.2 – THS1206 ADC setup flowchart

The FPGA implementation for the digital RF-ID detection system requires the ADC to be setup for a 3 MSPS sample rate with an unsigned 12-bit value and the FIFO trigger level set to 1. For these settings the control register CR0 gets the value 0x000 and control register CR1 gets the value 0x4A0. For more information on the ADC control registers consult the THS1206 data sheet [14]. The data sheet also gives information on the timing waveforms for the ADC.

The FIFO is not used because the FPGA implementation of the PPD algorithm is designed in such a way that each value from the ADC is processed, while other values are processed further down the processing line.

5.2.1 Counter sub-block

The counter is used to time the conversion clock needed by the ADC for the sample rate. The FPGA is driven by an 18.432 MHz external clock source and is divided by six to get a sample rate of 3.072 MHz. This is not the ideal sample rate of 3MHz but works well with the PPD algorithm implementation.

```
Conv_CLK:
  PROCESS (clk, reset)
  VARIABLE count    : INTEGER RANGE 0 TO 5;
  BEGIN
        IF reset = '1' THEN
           count := 0;
           CONVCLK <= '0';
        ELSIF rising_edge(clk) THEN
           IF START_CONV = '1' THEN
              IF count = 5 THEN
                 count := 0;
                 CONVCLK <= not CONVCLK;
              ELSE
                 count := count + 1;
              END IF;
           END IF;
        END IF;
  END PROCESS Conv_CLK;
```

Code listing 12 – Counter sub-block for ADC conversion clock

Code listing 5 shows the VHDL code for the counter. One of the FPGA user I/O pins is connected to a switch and acts as the reset pin. If this pin is high the counter block sets the CONVCLK pin low and resets the counter to 0. If the reset pin goes low the counter starts counting to six and then drives the CONVCLK pin high. The ADC sampling is then driven by CONVCLK. The START_CONV condition allows the ADC control state machine to drive the counter only when it is in the conversion state.

5.2.2 ADC control state machine

The ADC control block uses state machines to determine when to reset/set the ADC and read data from the ADC. The main state machine, AD_STATE, sets the ADC state (Reset, Set, Write, and Read) and controls the CTRL_STATE state machine. CTRL_STATE does the actual work of setting and resetting the ADC, setting up the ADC control registers and reading from the ADC FIFO.

On reset, both AD_STATE and CTRL_STATE is set to IDLE. AD_STATE sets the state to RESET_AD, and CTR_STATE writes 0x401 to the AD_DATA bus and sets the WR pin high. This causes an ADC reset and all other settings are written the same way. This is shown in

figure 5.3:
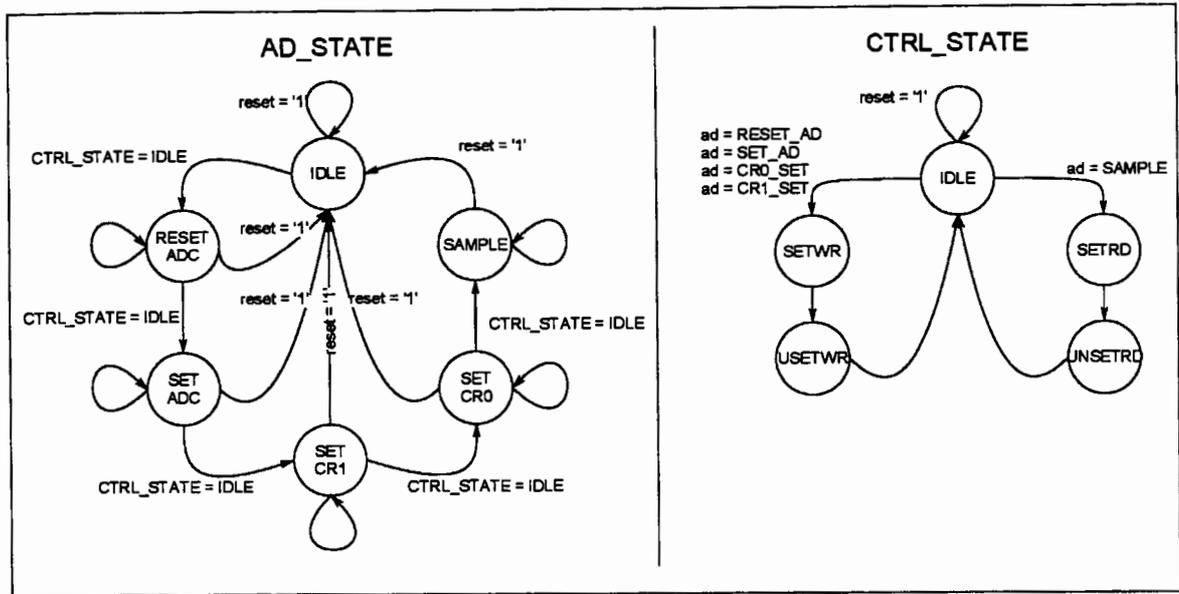
Figure 5.3 – ADC control state machines

The data written to the ADC by CTRL_STATE is determined by the state of AD_STATE. A register is set to a value depending on AD_STATE, and SETWR in the CTRL_STATE state machine writes the required value to the ADC.

After the ADC is set up, the ADC control block goes into an endless sampling loop. This is exited by resetting the system.

The ADC control block is found in the *adc.vhd* file located on the Appendix CD.

5.3 PPD algorithm block

The C code implementation of the PPD algorithm can be divided into different detection stages. These are the averaging filter stage, the threshold stage, the start bit detection stage and the ID detection stage. These four stages can run independently from one another on the FPGA. This creates a parallel pipelined processing architecture.

5.3.1 Averaging filter stage

Upon receiving a data point from the ADC, the averaging filter stage adds it to the four previous data values and sends the answer to the threshold stage. This stage receives a control signal from the ADC which indicates that a new data value is available. After this value is added to the other four values, other processing blocks are notified that the filtered data can be read. This block is found in *filter.vhd*.

5.3.2 Threshold stage

As with the C code, this stage counts to a hundred and adds a threshold to the maximum value of these hundred values. The maximum of the next hundred values is also stored for changes in the noise level. The threshold value is sent to the start bit detection section of the algorithm.

This stage, found in *limiter.vhd*, starts processing the filtered data when it receives a signal from the averaging filter stage. When the signal is received a counter is started and a maximum is calculated. After these hundred values, the stage notifies the pulse detection stage that it can begin to look for pulses. If a pulse is detected in the pulse detection stage the threshold stage receives a signal and consequently stops setting the threshold. It resumes the threshold calculation after an error is detected or a successful ID detection.

5.3.3 Pulse detection

The pulse detection section, found in *syncpos.vhd* simply looks for a value exceeding that of the threshold value received from the previous stage. If a possible pulse is found it activates the start bit detection section and deactivates the threshold section.

5.3.4 Start bits detection

The start bits and the synchronising bit are detected in the start bit detection block (*synchron.vhd*) and it is done in much the same way as in the C implementation. With the ADC sampling rate at 3.072MHz, number of values per tag bit is set at 3.072 MHz/128 kHz = 24. When this stage is activated by the pulse detection stage, a counter is enabled and if a maximum is detected at 24 point intervals the start bits are detected.

When the start bits are detected the counter counts a further 52 data points and sets that value as the maximum for the data bit detection stage. This in turn sends a signal to the data bit detection stage that starts the data bit detection.

5.3.5 ID bits detection

This section is started by the start bits section when a successful synchronising bit is detected. As mentioned before the tag bit period on the FPGA system is 24 data points per bit period. The bit detection looks for a ONE-ONE bit order at 24 points, and a ONE-ZERO bit order at 32 points. Looking from a ZERO bit, the next ONE is 8 points away or a ZERO is 24 points away. Each bit detected is placed in a 64-bit register.

When all 64 bits are detected the bit detection section signals all other sections, dependant on it, that the data bits are ready. This block is found in the *extracted.vhd* file.

5.3.6 CRC block

The FPGA implementation of the CRC is inherently a serial processing implementation. The 64 tag data bits are shifted through a register and XORed with the CRC-16 polynomial. This is shown in *code listing 13*:

```
IF start = '1' THEN
  IF count = 64 THEN
    start <= '0';
    IF data_sig = "0000000000000000" THEN
      correct <= '1';
    ELSE
      fail <= '1';
    END IF;
  ELSE
    count <= count + 1;
    data_sig(0)  <= ID_buf(63-count) xor data_sig(15);
    data_sig(15) <= ID_buf(63-count) xor data_sig(15) xor data_sig(14);
    data_sig(2)  <= ID_buf(63-count) xor data_sig(15) xor data_sig(1);
  END IF;

  data_sig(1)  <= data_sig(0);
  data_sig(3)  <= data_sig(2);
  data_sig(4)  <= data_sig(3);
  data_sig(5)  <= data_sig(4);
  data_sig(6)  <= data_sig(5);
  data_sig(7)  <= data_sig(6);
  data_sig(8)  <= data_sig(7);
  data_sig(9)  <= data_sig(8);
  data_sig(10) <= data_sig(9);
  data_sig(11) <= data_sig(10);
  data_sig(12) <= data_sig(11);
  data_sig(13) <= data_sig(12);
  data_sig(14) <= data_sig(13);

END IF;
```

Code listing 13 – CRC section

When the ID is correctly detected, the CRC section signals to the communications block that an ID is available for transmission. This section also signals that a failed detection occurred. The *crc.vhd* file contains the CRC block implementation.

5.4 Communications block

The communications block is broken up into four sub-blocks: a data packing block, a FIFO memory block, the RS-232 control block and the RS-232 transmission block.

5.4.1 Data packing block

The data packing block, *pack.vhd*, simply divides the 64-bit tag ID into eight bytes and adds a header (0xAA) to the packet (*figure 5.4*):

HEADER	Byte 8	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1
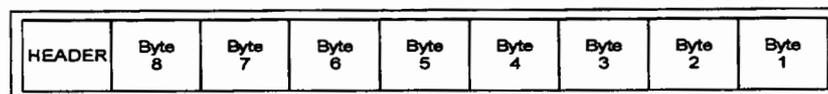
Figure 5.4 – Data packet

This is also implemented using a state machine. The state machine waits for a count and then sets a data available signal high and waits for the FIFO block to read a byte. After each consecutive byte that is read, the data pack block provides the next byte for reading.

5.4.2 FIFO block

After receiving the packet available signal from the data packing block, the FIFO block, *fifo.vhd*, reads the nine bytes of the data packet into the FIFO buffer and indicates that it has data packets available to send to the host computer.

Altera provides Megafunctions that are common blocks, efficiently implemented on their FPGA's. For the FIFO block the Alter FIFO Megafunction is used and is set up to be 8-bits wide and 8092 words deep. This enables the FPGA to store 899 ID's for transmission. The FIFO also has an asynchronous clear and a FULL and EMPTY signal.

5.4.3 Universal asynchronous transmitter control block

The universal asynchronous transmitter (UAT) control block, *uart.vhd* controls the flow of data into/out of the FIFO block and out of the FPGA to the host computer through the transmission block. This block is also state machine based as illustrated in *figure 5.5*:
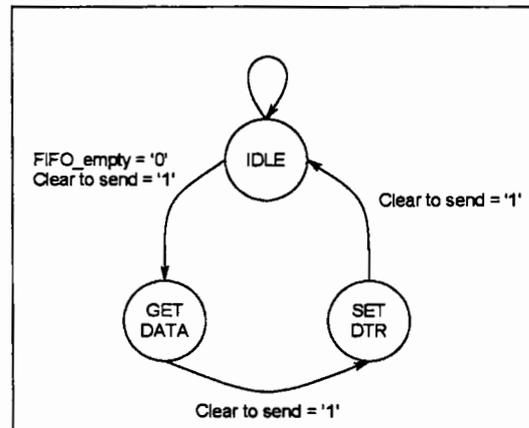
Figure 5.5 – UAT state machine

When a byte is sent to the host PC the control block is informed and it requests that the next byte is sent form the FIFO block. It stops sending when the FIFO is empty.

5.4.4 UAT transmission block

In order to send data over the UAT, the transmit block, *transmit.vhd*, needs to setup the bit speed and the RS-232 data format. The bit speed is simply calculated by dividing the system clock to the required bit speed. The data setup for RS-232 standard is a start bit, four to eight data bits, a parity bit option and one, one and a half or two stop bits.

The bit speed is calculated with a counter similar to that used in the ADC block. With a clock divisor of 160 for a system clock of 18.432 MHz, the bit speed is hard coded to 115 200 bits/s. The data setup is set to one start bit, eight data bits, no parity and one stop bit.

Chapter 6

6 Testing and simulation

The FPGA based system and the optimised TMS320C6416 DSP system are tested against the original analogue RF-ID detection system and the original digital system. Since it is difficult to compare the FPGA system to the TMS320C6416 DSP systems based solely on speed, the tag range test and the total detected tags test are used to compare the two systems.

For the optimised TMS320C6416 DSP versus the original TMS320C6416 DSP systems, the cycle count and the number of instructions executed are also included to the test criteria. This shows how certain modifications positively affected the speed of the old system.

6.1 FPGA simulation

The ModelSim HDL verification tool from Mentor Graphics allows the HDL designer to test his design without having to program a physical device. The simulation offers graphical display of the HDL signals interacting with each other.

To get accurate results from the simulation, the ADC control block needs to receive signals from the THS1206 ADC device. Since it is not possible to connect the ADC to the simulation software, it also has to be simulated. Fortunately VHDL supports test benches. These test benches can be designed to accurately simulate external signals sent to the design being simulated. The test bench used in the detection simulation is given in *code listing 14*:

```
ARCHITECTURE test_bench OF test IS
  COMPONENT detector
    PORT
    (
      clk,reset        :  IN      STD_LOGIC;
      TxD              :  OUT     STD_LOGIC;
      ADC_WR           :  OUT     STD_LOGIC;
      ADC_RD           :  OUT     STD_LOGIC;
      ADC_DATA_AV      :  IN      STD_LOGIC;
      ADC_DATA         :  INOUT   UNSIGNED(11 DOWNTO 0);
      ADC_CONVCLK      :  OUT     STD_LOGIC
    );
  END COMPONENT;

  SIGNAL clk           :  STD_LOGIC := '0';
  SIGNAL reset         :  STD_LOGIC;
  SIGNAL TxD           :  STD_LOGIC;
  SIGNAL ADC_WR        :  STD_LOGIC;
  SIGNAL ADC_RD        :  STD_LOGIC;
  SIGNAL ADC_CONVCLK:  STD_LOGIC;
  SIGNAL ADC_DATA_AV   :  STD_LOGIC := '0';
  SIGNAL ADC_DATA      :  UNSIGNED(11 DOWNTO 0) := "ZZZZZZZZZZZZ";
  SIGNAL period        :  time := 54.253 ns;
  SIGNAL dataperiod    :  time := 325.521 ns;

BEGIN

detect : detector
  PORT MAP
  (
    clk => clk,
    reset => reset,
    TxD => TxD,
    ADC_WR => ADC_WR,
    ADC_RD => ADC_RD,
    ADC_DATA_AV => ADC_DATA_AV,
    ADC_DATA => ADC_DATA,
    ADC_CONVCLK => ADC_CONVCLK
  );

reset <= '1','0' after period;                              --Reset system
clk <= not clk after (period/2);                            --Start clocking
PROCESS
  FILE data : TEXT OPEN READ_MODE IS "./output.txt";--Open data file
  VARIABLE sample : line;                                   --Text line variable
  VARIABLE in0_var : INTEGER RANGE 0 TO 4095 ;   --Formated data variable
BEGIN
  WAIT FOR 10*period;                                       --Wait until ADC setup compleats
  WHILE NOT endfile(data) LOOP                              --Start reading sampled data
    WAIT FOR dataperiod;                                    --Wait for 3 MHz clock
    ADC_DATA_AV <= '1';                                     --Set ADC data availeble signal
    READLINE(data,sample);                                  --Read next line
    read(sample, in0_var);                                  --Convert to integer
    ADC_DATA <= CONV_UNSIGNED(in0_var,12);    --Convert integer to ADC unsigned
    WAIT FOR period;
    ADC_DATA_AV <= '0';                                     --Disable ADC data available
  END LOOP;
  WAIT;
```

Code listing 14 – FPGA detection system simulation test bench

6.1.1 Detection block

The FPGA's main block is the detection block and is located in detect.vhd. This encapsulates all the other blocks and ports the pins seen by the outside world. After running the simulation for awhile, it is apparent that an ID was detected and sent to the host computer.

Figure 6.1 – Detect.vhd simulation

In *figure 6.1* when the possync_sig is driven to a logical ONE, the start bits and synchronising bit detection starts. After successfully detecting the start and synchronising bits startbit_sig is driven to a logical ONE and this forces detectid_sig to go high. When an ID is successfully detected idyes_sig goes high and the algorithm block is reset for the next ID. The detected ID, in this case 0x058000000B631F97, is passed to the CRC block. This ID was detected correctly and crc_correct_sig is driven high. With crc_correct_sig high, the ID is formatted for transmission and the pda_sig signal indicated that the data can be sent to the host. In *figure 6.1* the txd signal show the RS-232 signal being sent.

6.1.1.1 ADC block

Figure 6.2 – adc.vhd state machine simulation

After a system reset, the ADC control block resets and configures the ADC (*figure 6.2*). The ad

and `ctrl` signals show how the control block propagates through the states of the ADC. When the ADC control block is in the sampling state it starts the conversion clock by toggling the `ad_convclk` signal and waits for the `data_av` signal to go high.

Figure 6.3 – ADC sampling simulation

When the ADC sends its data available signal, the ADC control block sets the `rd` signal low and reads the sampled data into the `din_reg` register. The data in this register is then made available through the `dout` signal to other processing blocks. Other blocks are signalled by the `data_av` signal that the data is ready to be read.

6.1.1.2 RF-ID block

Figure 6.4 – rfid.vhd simulation

In *figure 6.4* the signal wave forms of the rfid.vhd simulation are shown. Here, three possible start bit pulses were detected prior to the actual ID start bits detection, as indicated by the `possync` and `psync` signals. After each false detection or ID detection the `resetcombine` signal is pulsed to restart the detection process. The `subreset` signal drives the `resetcombine` signal after a false start bit sequence has been detected. This file only serves

as the connection between the subsequent algorithm stages. Its main function is to port the necessary signals to the other blocks in the algorithm implementation.

6.1.1.2.1 Filter stage

With the ADC now sending samples to the FPGA the averaging filter stage reads the sampled data on each `adcreadyf` signal pulse, which is connected to the `data_av` signal. *Figure 6.5* illustrates this and also shows that after five sampled values are summed the `adcreadyf` signal goes high for a clock period. This signals the rest of the algorithm block that this filtered signal is available.

Figure 6.5 – filter.vhd simulation

6.1.1.2.2 Limiter stage

Figure 6.6 – limiter.vhd simulation

The limiter stage receives the `data_av` signal from the filter stage and determines the maximum noise floor and adds 100 as shown in *figure 6.6*. As expected, the threshold calculation is halted once a possible pulse is detected, as shown in *figure 6.7*:

Figure 6.7 – limiter.vhd simulation while a tag is detected

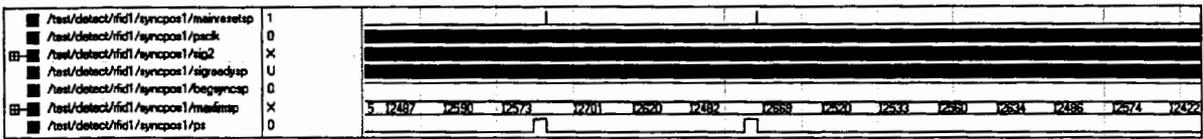### 6.1.1.2.3 Pulse detection stage

Figure 6.8 – syncpos.vhd simulation

Figure 6.8 illustrates how the pulse detection block detects a pulse and notifies the start bit detection block. In the figure two false start pulses were detected. At the detection of a pulse the `ps` signal goes high. The block then waits for a reset which occurs when a false pulse is detected or when a tag was detected. In both instances the reset is driven by the `mainiresetsp` signal.

6.1.1.2.4 Start and synchronising bits detection stage

The start bit detection is started with the detection of a pulse. As stated this is signalled by the `ps` signal. After detecting 8 start bits and the synchronising bit the `startid` signal goes high and goes low after a ID was detected. If an error occurs with the detection of the start bits the system is reset by the `resetsync` signal after it was notified by the `syncerr` signal. This is illustrated in *figure 6.9*:
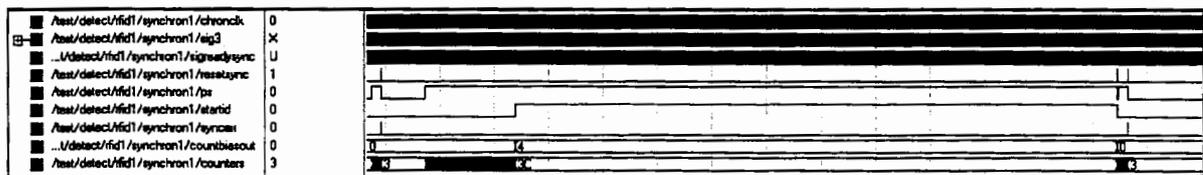

Figure 6.9 – synchron.vhd simulation

6.1.1.2.5 ID data detection stage

The ID data detection block receives a signal from the start bit detection block, `startid` signal, that indicats that the ID data detection can start. When the ID detection completes the `idready` signal goes high and indicates the ID is ready to be read. The `resetid` signal also goes high as the detection completes and this results in an algorithm reset. *Figure 6.10* illustrates this:

Figure 6.10 – extractid.vhd simulation

6.1.1.3 Cyclic redundancy check

Figure 6.11 illustrates how the CRC block receives an ID, calculates the CRC and the signals the data packing block that it can start building a Host I/O packet.

Figure 6.11 – crc.vhd simulation

The CRC block receives the `id_av` signal and reads the ID data int the `id_buf` register. When the CRC is reset by the `rfreset` signal, the `data_sig` signal is initialised to 0xFFFF. This then starts the CRC algorithm. When the CRC clears the ID the `correct` signal is pulsed and the data packing block continues its work.

6.1.1.4 Data packing

After the successful detection of a tag, the tag data is read by the data packing block and stored in the `id_buff` register. After each byte is calculated the `pda` signal is pulsed. This signals the UAT block to send the data to the host. This is illustrated in *figure 6.12*:
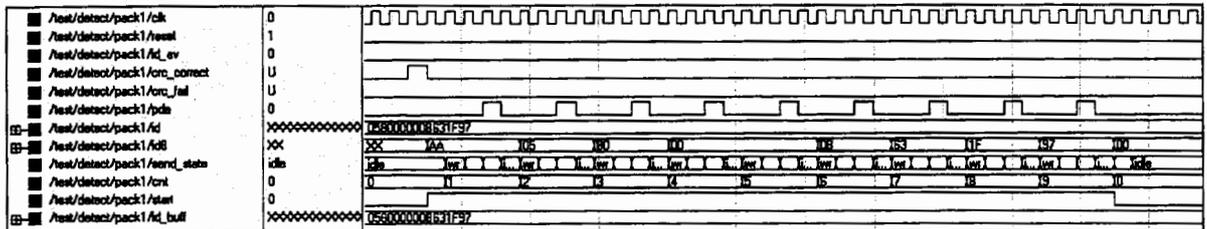

Figure 6.12 – pack.vhd simulation

6.1.1.5 Universal asynchronous transmitter

The universal asynchronous transmitter block passes signals from the data packing block to the FIFO and transmitter block. The UAT receives a `wrreq` signal (a write request) from the data packing block via the `pda` signal. The 9 bytes of the data packet is written to the FIFO. This causes the `empty_sig` signal to go low, indicating that there is data available to be sent.
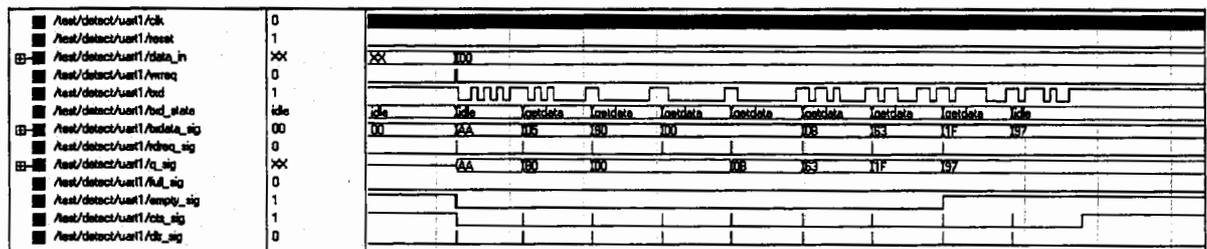
Figure 6.13 – uart.vhd simulation

In *figure 6.13* the `dtr_sig` signal indicated that the FPGA is ready to send data and this forces the `rdreq_sig` signal high. The data is then READ out of the FIFO into the transmit block and sent to the host. While the data is sent, the `cts_sig` signal goes low which signals that no other reads from the FIFO should occur. If the `cts_sig` signal goes high, another transmission is started.

6.1.1.5.1 FIFO

In *figure 6.14* a screenshot of the FIFO simulation results is shown. Here the `wrreq` signal gets pulsed nine times and the `data` register is written into the FIFO. On each `rdreq` signal the data is read out of the FIFO and into the `q` register. The transmit block then reads the `q` register and transmits the data.
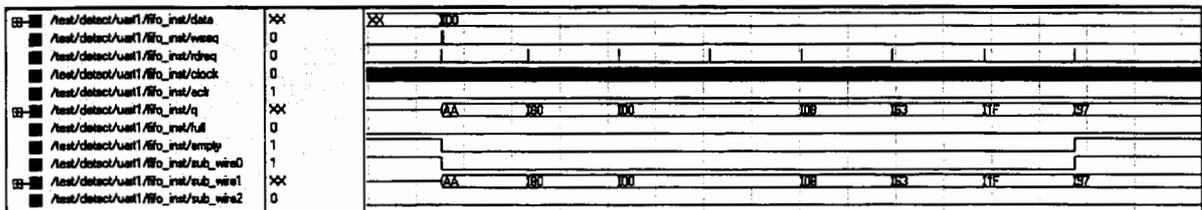
Figure 6.14 – fifo.vhd simulation

6.1.1.5.2 Transmitter

The transmitter block converts the packet data into a serial data stream. For a complete tag ID, this is done nine times (one header byte and eight data bites). *Figure 6.15* illustrates a complete ID transmission.
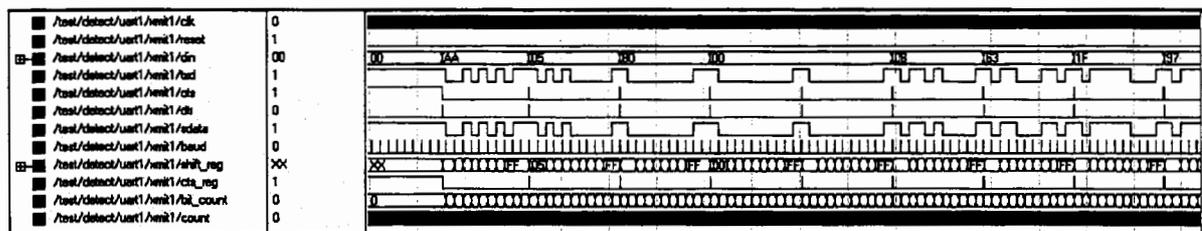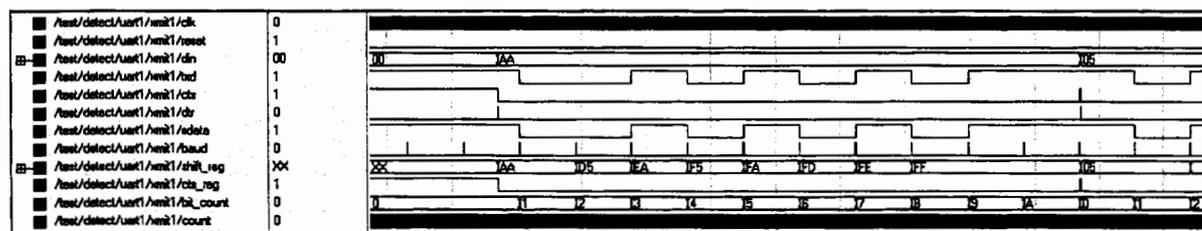
Figure 6.15 – xmit.vhd simulation during tag transmission

Figure 6.16 – transmit.vhd simulation during data packet header transmission

Figure 6.16 illustrates how the transmitter sends a byte of data to the host computer. With each pulse of the `baud` signal, the least significant bit is shifted onto the `txd` pin. After each byte is

sent the `shift_reg` register is filled with the next byte to send. The transmitter drives the `dtr` signal high to indicate that it is ready to send another byte. While a byte is being transmitted, the `cts_reg` signal is set low and ported out to the `cts` signal.

6.2 Test setup

All tests are done using 128 kbit/s tags in a lab environment with the least possible amount of noise. This ensures that the analogue system performs at its best and thus becomes a suitable baseline for the other two systems. All tests are performed ten times for an accurate statistical result.

6.2.1 Criteria 1: Tag range test

The range test is performed by moving a tag through the antenna beam at different distances starting at the absolute maximum range a tag is visible. The setup is illustrated in *figure 6.17* and the systems are compared by the amount of times the tag is detected at each distance in 15 seconds.
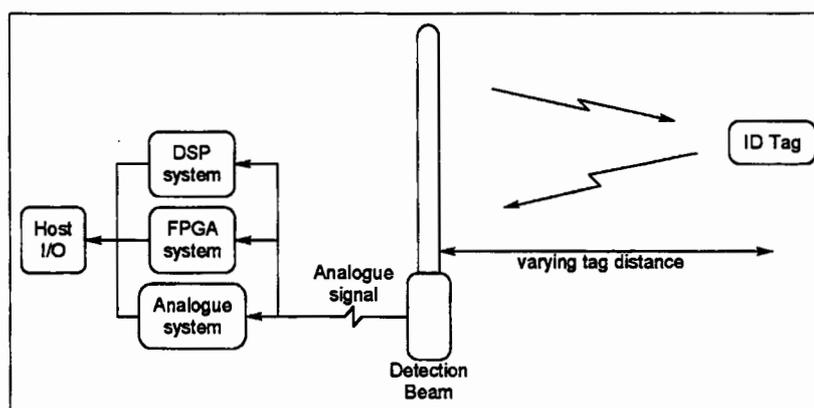

Figure 6.17 – Tag range test

With the analogue system set as the baseline, this test tries to illustrate that the digital systems can keep up with the analogue system. The expected results are that the digital systems perform as well as or better than the analogue system. As illustrated in paragraph 6.3.1

6.2.2 Criteria 2: Total tags detected

In the total tags detected test, the tags are moved from an out of range position to the maximum detection range (*figure 6.18*). The tags are held in the beam for a period of 15 seconds and the tag count is recorded. Six differant tags are used for this test.
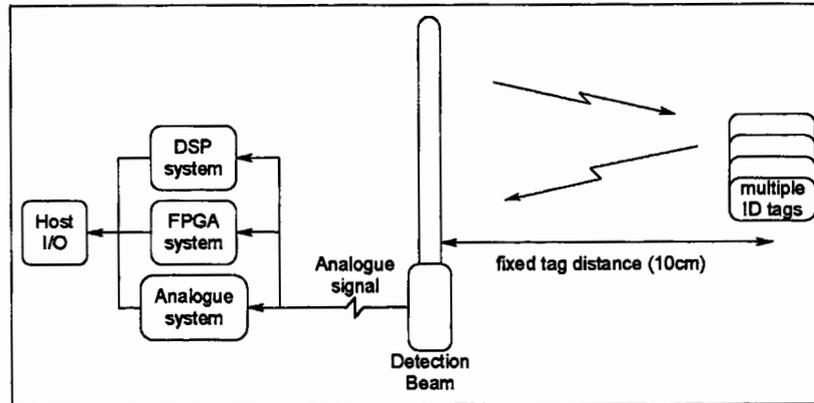

Figure 6.18 – Total tags detected

With the high tag rates being detected in this test, the short comings of the digital systems become apparent. This test more clearly defines the difference between the FPGA system and the TMS320C6416 DSP system. As illustrated in paragraph 6.3.2.

6.2.3 Criteria 3: Cycle count test and number of instructions

To compare the optimised TMS320C6416 DSP system to that of the original TMS320C6416 DSP system, the CPU cycle count of the two systems is compared. The cycle count is measured while 15 different tags, at the maximum detection range, are being detected. The total number of instructions used for the pulse detection algorithm in both systems is also compared.

This test is done at different optimisation stages to illustrate that the methods used actually have an effect on the cycle count and execution time. Each set of tests are done at a 30 second interval to give the CCS software a chance to retrieve a representative amount of data for the comparison between the unoptimised and optimised systems.

Version one (V1) of the TMS320C6416 DSP based system is the unoptimised version and is set as the test baseline. Version two (V2) includes the sign optimisation for the averaging filter and the tag data storage method optimisation, which includes the CRC function optimisation.

Version three (V3) introduces the if-else-if statement optimisations and includes all of V2's optimisation. Version four (V4), the final version, includes all of V3's optimisation and has a more refined if-else-if statement optimisation. This version also includes the EDMA hardware peripheral optimisation. The results are shown in paragraph 6.3.3.

6.3 Test results

6.3.1 Criteria 1: Tag range test

Test 1 - Averages				
Range	Analogue (Base line)	FPGA	Optimised DSP	Unoptimised DSP
10cm	215.2	212.6	175	164
25cm	215.2	211.6	173.8	163
50cm	217.2	214	163.8	154
75cm	210	204	96	93.2
100cm	154.6	192.8	83.2	80.6
125cm	87.2	94.2	5.2	4

Table 2 – Tag range test result averages

Table 2 shows the results of the tag range test. With the analogue system as the baseline for the test, these results now show the performance of the other digital systems. From the data it is seen that for the FPGA system does fairly well to keep up this the analogue system. On an average over all distances the FPGA system shows an increase in performance by 4%. This is due to an increase in detection at distances further away. Both the optimised and unoptimised DSP systems had a 43% and 46% decrease in performance. *Figure 6.19* graphs the table data to give an overall view of how each system performed at various stages.
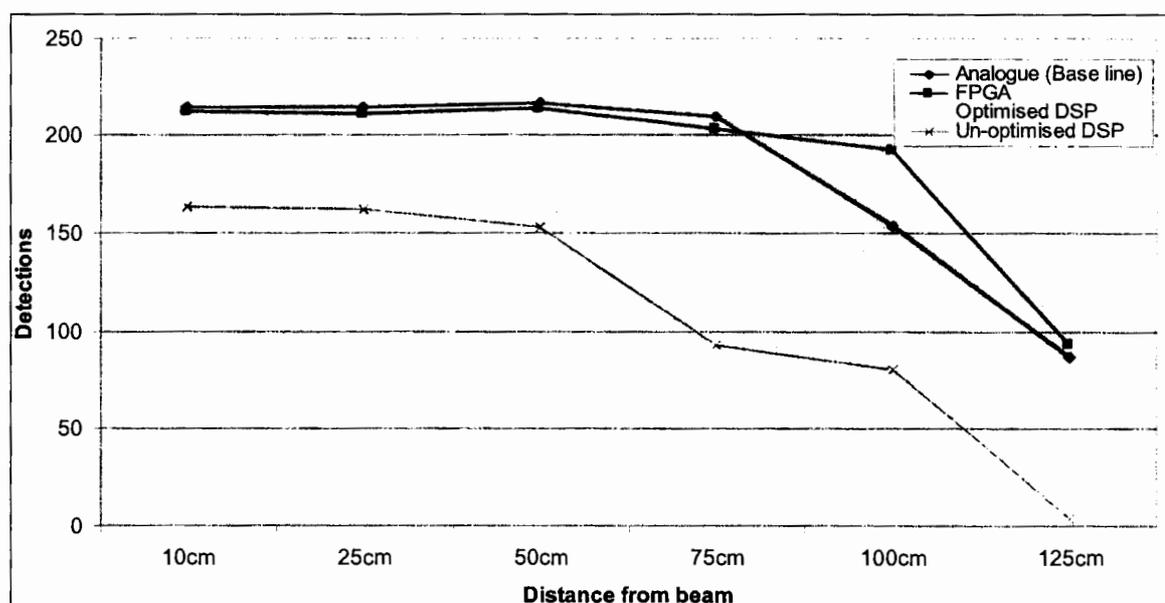

Figure 6.19 – Tag range test result graph

The FPGA performs better at larger distances because it can extract the ID with a smaller signal to noise ratio.

6.3.2 Criteria 2: Total tag Detected

Total detected tags test					
Test run	1	2	3	4	5
Analogue	1201	1184	1182	1206	1219
FPGA	1180	1169	1166	1178	1209
Optimised DSP	876	872	873	875	863
Un-optimised DSP	835	836	830	837	835

Table 3 – Total detected tags test results

Table 3 shows the tag count after 15 seconds, while six tags are held in the antenna beam at the maximum detection range. All six tag where detected. Here it is evident that the DSP systems can not keep up with the analogue and FPGA systems.
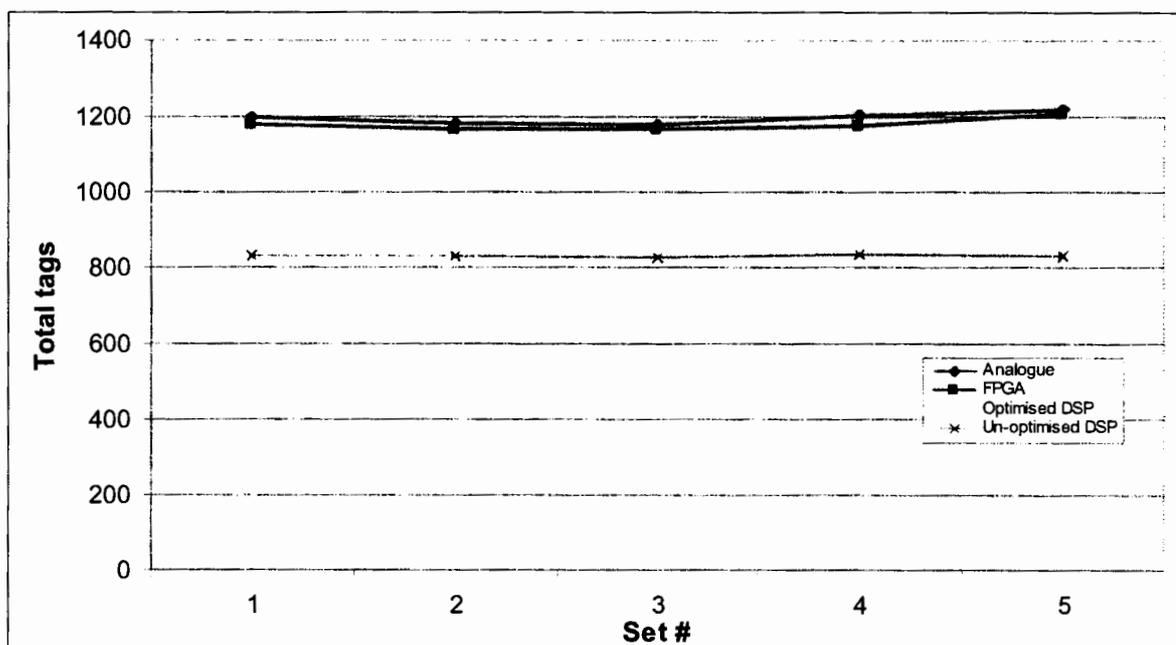

Figure 6.20 – Total detected tags test results graph

Figure 6.20 is a graphical representation of the table data. Here the FPGA has a 1.5% loss over the analogue system. This is due to unoptimised algorithm thresholds. The TMS320C6416 DSP systems have a loss of 27% and 30% for the optimised and unoptimised versions, respectively, over the analogue system. This clearly illustrates that tag data is being lost during the buffer swapping on the TMS320C6416 DSP systems.

6.3.3 Criteria 3: Cycle count test and number of instructions

6.3.3.1 V1 test results

The baseline, V1, has a maximum of 4 862 432 instructions for a complete pass through the PPD algorithm. The CRC calculation has an instruction count of 6 032 instructions. During this, the CPU usage started at 51.21% during the initialisation phase and settled at about 39.63%. This is shown in *figure 2.21*:
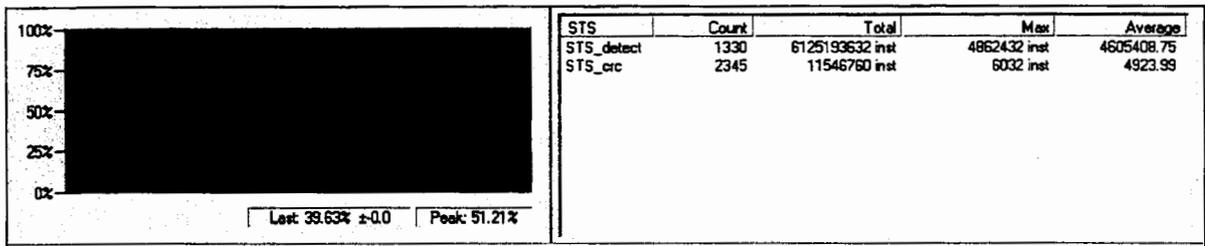

STS	Count	Total	Max	Average
STS_detect	1330	6125193632 inst	4862432 inst	4605408.75
STS_crc	2345	11546760 inst	6032 inst	4923.99

Last: 39.63% ±0.0   Peak: 51.21%

Figure 6.21 – Version 1 instructions count

The cycle time, shown in *figure 6.22*, is 8194.25 µs (8.2 ms) and 10.56 µs for the algorithm and the CRC calculation, respectively.
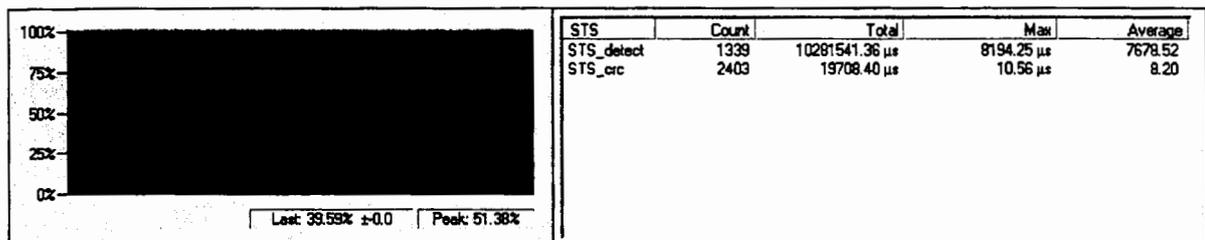

STS	Count	Total	Max	Average
STS_detect	1339	10281541.36 µs	8194.25 µs	7678.52
STS_crc	2403	19708.40 µs	10.56 µs	8.20

Last: 39.59% ±0.0   Peak: 51.38%

Figure 6.22 – Version 1 cycle time

6.3.3.2 V2 test results

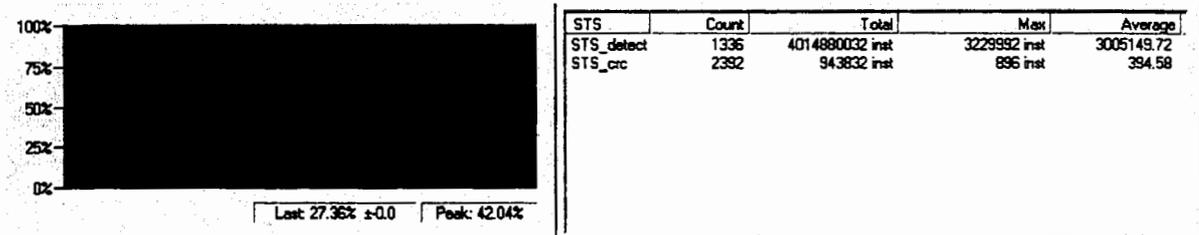In the second version of the PPD algorithm, the results show a significant decrease in code size and increase speed.

STS	Count	Total	Max	Average
STS_detect	1336	4014880032 inst	3229992 inst	3005149.72
STS_crc	2392	943832 inst	896 inst	394.58

Last 27.36% ±0.0   Peak: 42.04%

Figure 6.23 – Version 2 instructions count

Figure 6.23 and *6.24* shows the instruction count and the cycle time for V2 respectively. Here it is seen that V2 has a maximum of 3 229 992 instructions and a cycle time of 5386.48 µs. The figures also show that CRC calculation now uses 896 instructions and executes in 1.57 µs.

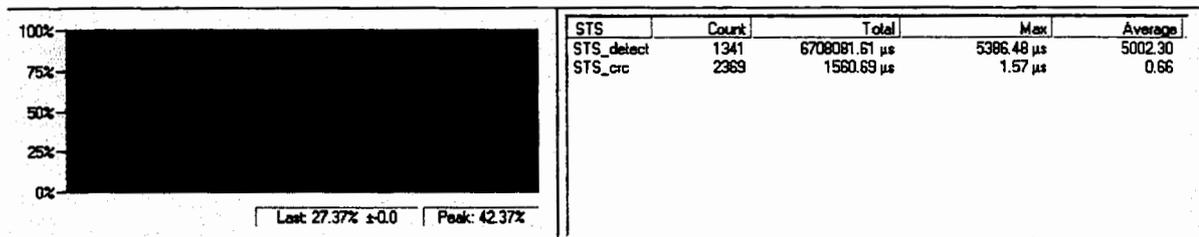| STS	Count	Total	Max	Average
STS_detect	1341	6708081.61 µs	5386.48 µs	5002.30
STS_crc	2369	1560.69 µs	1.57 µs	0.66

Last 27.37% ±0.0   Peak: 42.37%

Figure 6.24 – Version 2 cycle time

Here the CRC calculation, data storage and filter stage optimisation gave a 12% speed increase and had a total CPU usage of 27.37%.

6.3.3.3 V3 test results

With the if-else-if statement optimisation implemented, V3 has a considerable performance gain over the unoptimised version. The total CPU usage over the detection period is about 18.24% and the CPU executes a total of 2 475 240 instructions while executing the algorithm. This is shown in *figure 6.25*:
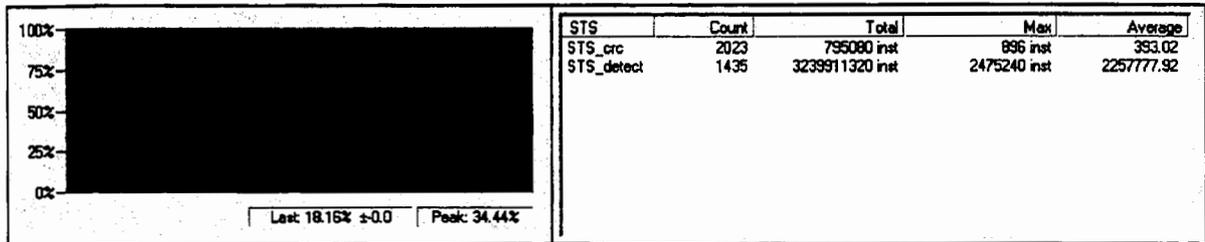
STS	Count	Total	Max	Average
STS_crc	2023	795080 inst	896 inst	393.02
STS_detect	1435	3239911320 inst	2475240 inst	2257777.92

Last 18.16% ±0.0   Peak: 34.44%

Figure 6.25 – Version 3 instructions count

The figure also shows that the CRC calculation instruction has not changed as would be expected. In *figure 6.26* the cycle time for V3 is 4180.59 µs for the algorithm and 1.47 µs for the CRC calculation.

STS	Count	Total	Max	Average
STS_crc	2083	1362.61 µs	1.47 µs	0.65
STS_detect	1437	5413725.97 µs	4180.59 µs	3767.38

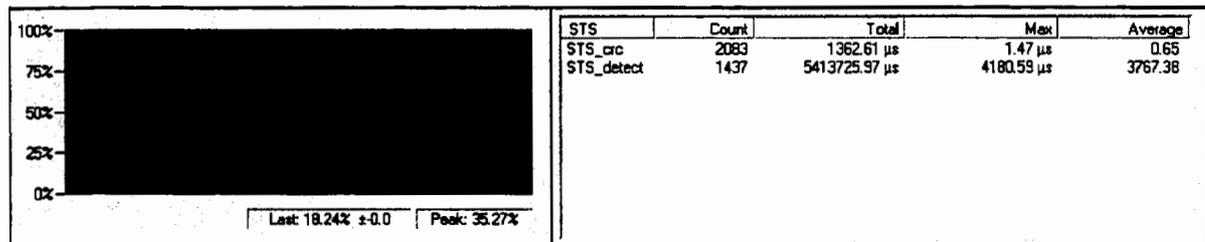Last 18.24% ±0.0   Peak: 35.27%

Figure 6.26 – Version 3 cycle time

In this version the CRC algorithm is unchanged from that of V2 and the cycle time and instruction count has not changed.
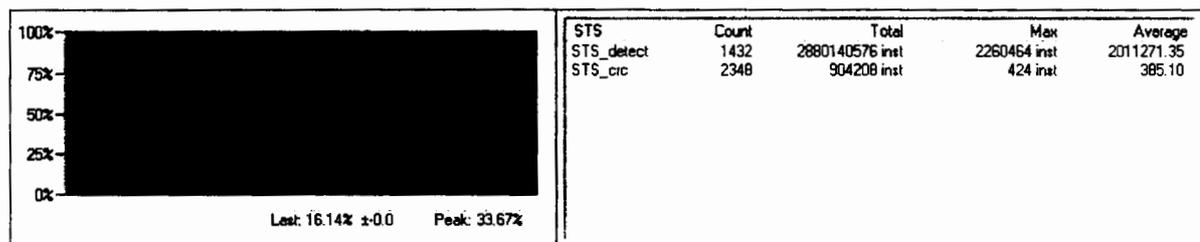
6.3.3.4 V4 test results

STS	Count	Total	Max	Average
STS_detect	1432	2880140576 inst	2260464 inst	2011271.35
STS_crc	2348	904208 inst	424 inst	385.10

Last: 16.14% ±0.0   Peak: 33.67%

Figure 6.27 – Version 4 instructions count

Figure 6.27 and *6.28* show the V4 instruction counts and cycle times, respectively. More importantly, the CPU usage is now 16.24% because the data transfer and buffer switching is not handled by the CPU. The instruction count for the algorithm is now 2 260 464 and has a cycle time of 3766.52 µs. The CRC calculation time and instruction count, again, is the same as V3 since it has not changed.
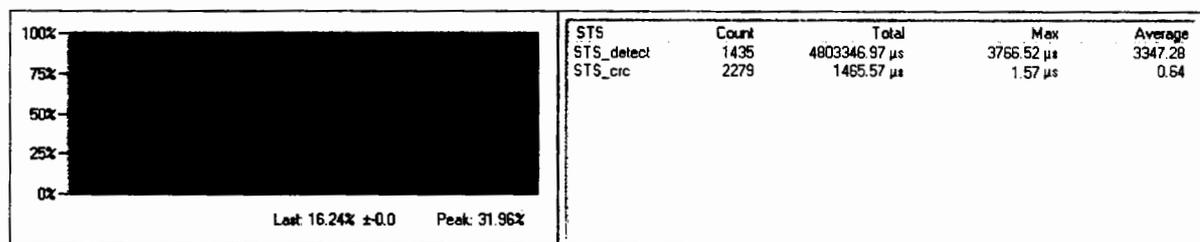

STS	Count	Total	Max	Average
STS_detect	1435	4803346.97 µs	3766.52 µs	3347.28
STS_crc	2279	1465.57 µs	1.57 µs	0.64

Last: 16.24% ±0.0   Peak: 31.96%

Figure 6.28 – Version 4 cycle time

These results constitute an increase of 53.5% for the PPD algorithm and 85.1% for the CRC algorithm. The CPU usage improved 59% in this version to that of V1.

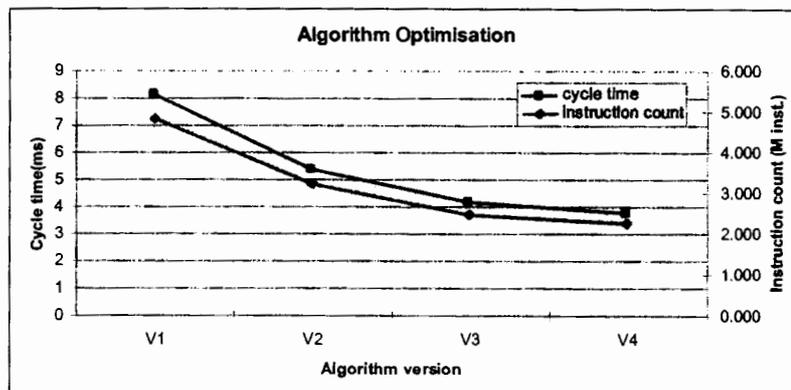The following graphs summarises the results for the DSP algorithm optimisations:
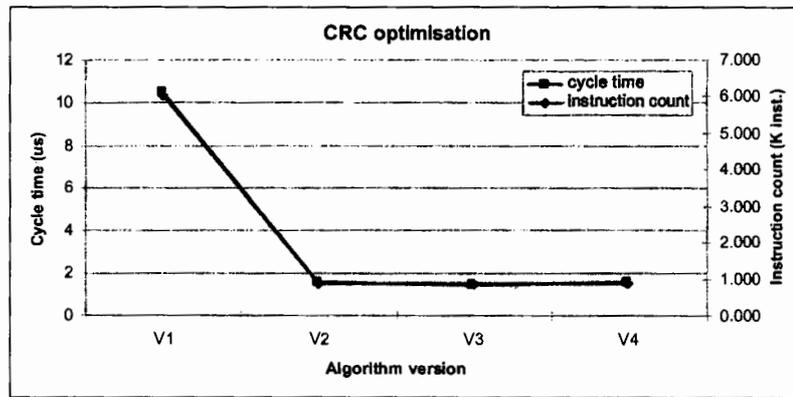

Figure 6.29 – Algorithm Optimisation summery
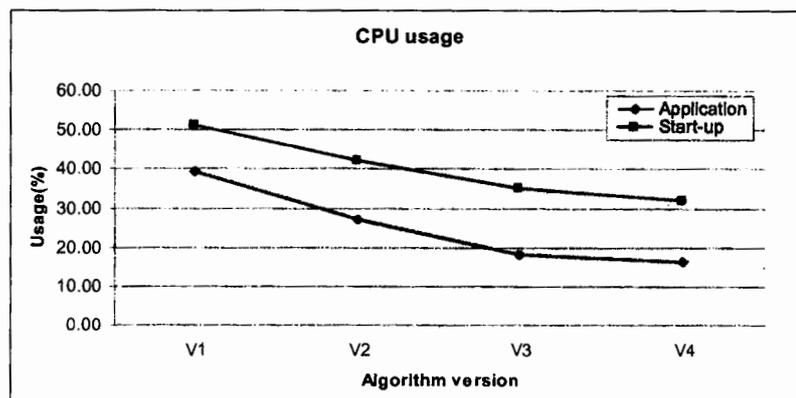
Figure 6.30 – CRC Optimisation summery

Figure 6.31 – CPU usage summery

Chapter 7

7 Conclusion

7.1 Overall conclusions

With the development of the pulse peak detection algorithm a more robust RF-ID tag detection system was realised. Even though the original TMS320C6416 DSP based system could theoretically detect tags better in all ambient noise situations it lacked the processing power to be as/ more effective than the analogue system. This problem was rectified by implementing the hardware on programmable logic that proved to be the more viable solution to this specific problem.

When looking at the test results it is evident that the FPGA based digital system out performed the TMS320C6416 DSP based system. On average the FPGA system even out performed the analogue system in the tag range test by four percent. The optimised and unoptimised TMS320C6416 DSP system lagged the analogue system by about 43 % and 46% respectively. Even though the FPGA and TMS320C6416 DSP based digital systems had a lower detection rate closer to the beam, it is most likely a result of the algorithm's threshold setup. This problem is not in the scope of this project and falls under another projects objectives.

In the total tags detected test, the FPGA, optimised and the unoptimised TMS320C6416 DSP systems performed 1.5 percent, 27 percent and 30 percent worse than the analogue system, respectively. This proves that the FPGA system can keep up with the analogue system. As for the DSP systems, the lower tag rate is a result of the block processing architecture. The DSP system only has 20.971 ms to process 65 534 data points. Currently it is processing the buffer at 3.347 ms. This seems fast enough, but with the handover from buffer to buffer data is lost. The lost data could easily be tag IDs and this causes the lower tag rates.

What the results do show is that optimising the TMS320C6416 DSP version of the digital detection system, resulted in an increase in the detection rates for the DSP system. This is significant because systems using TMS320C6416 DSPs, that are not reaching their real-time goals, can use the optimising techniques to reach these goals. The results for the cycle count tests show that these techniques free op CPU time and improve algorithm speeds. In the end an improvement of 53% in instruction count and cycle time was made.

7.2 Future recommendations

This study showed that the FPGA based digital RF-ID tag detection system is a viable alternative to the current analogue system. Even though the FPGA system performs on par with the analogue system, reasons why the FPGA based digital system is developed are:

- the FPGA digital system is cheaper,
- the FPGA digital system is more scalable,
- the FPGA digital system can still be optimised for better performance,
- the FPGA digital system performs better under noisy conditions than the analogue system,
- the digital realm provides a lot of algorithms that are not available in an analogue system

With this in mind some improvements can still be made to the FPGA digital system. The addition of a finite impulse response filter is the logical next step and implementing the updated optimised PPD algorithm.

More improvement in the HDL design could be made for the FPGA system. With the steep learning curve involved in HDL design it was not possible to design the system as to use the least amount of logic gates possible. A more knowledgeable HDL programmer would probably be more successful in doing so. With the FPGA system using the least amount of gates, it is then possible to implement better filters or other algorithms that extend the PPD algorithms power.

With tags with higher bit rates becoming more readily available the digital systems must be able to process data which is sampled at higher frequencies. For the FPGA systems this will not be a problem as the FPGA processing speed is only a factor of its clock speed. This is not the case for the TMS320C6416 DSP systems. The TMS320C6416 DSP system will be able to handle higher sample rates but at a certain point the TMS320C6416 DSP will be saturated. Data loss is also a concern for the DSP systems sampling at higher frequencies.

References

1 Berkeley Design Technologie, Inc 2004. *Texas Instruments TMS320C64x*. Berkeley Design Technology, inc. Available from: www.BDTi.com [Accessed 10 October 2004].

2 Altera Corporation *Cyclone FPGA*. Altera Corporation. Available from: http://www.altera.com/products/devices/cyclone/cyc-incex.jsp [Accessed 14 March 2005].

3 Altera Corporation 2003. [*Cyclone FPGA family data sheet, section 1*]. Portable Document Format (PDF) ed. Revision 1.5: Altera Corporation. Available from: www.altera.com [Accessed 20 March 2005].

4 Denver, A. *Multi Threaded TTY*. Microsoft. Available from: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnfiles/html/msdn_serial.asp [Accessed 12 May 2005].

5 Deitle, H.M., Deitle, P.J. 2002. *How to program C++, 3rd edition*. Upper Saddle River, New Jersey: Prentice-Hall inc.

6 EM Microelectronic - Marin SA 2002. *Multi Fequency contacless Identification device*. EM Microelectronic - Marin SA. [Available from: www.emmicroelectronic.com [Accessed 2 June 2005].

7 EventHelix.com Inc. 2000. *Optimizing C and C++ Code*. EventHelix.com Inc. Available from: http://www.eventhelix.com/RealtimeMantra/Basics/Code_optimizing.htm [Accessed 27 October 2004].

8 Barnett, A *C optimisation tutorial*. Available from: http://www.abarnett.demon.co.uk/tutorial.html [Accessed 30 October 2004].

9 Simsic, L. 2004. *Accelerating algorithms in hardware*. Embedded.com. Available from: www.embedded.com [Accessed 1 April 2005].

10 Texas Instruments 2002. [*TMS320C6000 Programmers Guide*]. Portable Document Format (PDF) ed. Revision G: Texas Instruments. Available from: www.ti.com/spru198g.pdf [Accessed 2 October 2004].

11 Texas   Instruments   *TMS320C6416.*   Available   from: .
http://focus.ti.com/docs/prod/folders/print/tms320c6416.html [Accessed 23 January 2005].

12 Texas Instruments 1999, *THS1206 modular EVM (SLVU098.pdf)*. Texas Instruments.

13 Texas Instruments 2005, *TMS320C6000 DSP Peripherals Overview Reference Guide. Texas Instruments.*

14 Texas Instruments [*THS1206 data sheet*]. Portable Document Format (PDF) ed. Revision H: Texas Instruments March 2005

16 Vorster, C. *Detection of RF-ID tag signals using digital signal processing and neural networks.* Thesis, North-West University, Potchefstroom Campus.

Appendix

Attached to this document is an appendix compact disk (CD). The CD contains this document in PDF format, the unoptimised DSP source code and project, the optimised DSP source code and project and the FPGA source code and project. All the test results and the simulation data are also available.