AN FPGA IMPLEMENTATION OF INCREMENTAL CLUSTERING FOR RADAR PULSE DEINTERLEAVING

By

Scott Bailie

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT
NORTHEASTERN UNIVERSITY
BOSTON, MASSACHUSETTS
APRIL 2010

This work is sponsored by the Department of the Air Force under Air Force Contract FA8721-05-C-0002. The Opinions, interpretations conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

NORTHEASTERN UNIVERSITY
DEPARTMENT OF
GRADUATE SCHOOL OF ENGINEERING

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**An FPGA Implementation of Incremental Clustering for Radar Pulse Deinterleaving**" by **Scott Bailie** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: April 2010

Supervisor: ______________________
Prof. Miriam Leeser

Readers: ______________________
Prof. Jennifer Dy

____
Dr. Brie Howley

NORTHEASTERN UNIVERSITY

Date: **April 2010**

Author:	**Scott Bailie**
Title:	**An FPGA Implementation of Incremental Clustering for Radar Pulse Deinterleaving**
Department:	**Graduate School of Engineering**

Degree: **M.Sc.** Convocation: **May** Year: **2010**

Acknowledgements

This thesis would not have been possible without the help of countless people. First, I'd like to express my sincere thanks to my advisor, Prof. Miriam Leeser. I am grateful for her technical guidance, tremendous enthusiasm, and constant support in both my undergraduate and graduate careers. Second, I'd like to thank Prof. Jennifer Dy whose expertise in clustering served as an invaluable resource.

I'd also like to thank my colleagues at MIT Lincoln Laboratory for their support. In particular Dr. Brie Howley, who was always available to share his vast knowledge of radar systems and signal processing.

Finally I would like to thank my family, especially my wife, Katherine, and my daughter Elizabeth. I could not have completed this degree without their love, patience, never ending encouragement.

Abstract

Incremental clustering is the unsupervised classification of dynamic streaming data samples into related groups called clusters. The process considers each data point only once so it is applicable to real-time problems requiring low latency solutions. One such application is the deinterleaving of radar pulse streams in an electronic warfare (EW) systems. Given a single stream of combined radar signals deinterleaving attempts to identify individual radar emitters based on characteristics of the received signal.

This thesis focuses on implementing an incremental clustering algorithm on a field-programmable gate array (FPGA) for the purposes of radar pulse deinterleaving. We introduce *ICED*, an algorithm for the *I*ncremental *C*lustering of *E*volving *D*ata, and discuss the details of implementing it in an FPGA. Experimental results show the applicability of the algorithm to the real-time requirements of EW pulse deinterleaving. The resulting design provides a 16 cluster implementation that consumes 70% of a Xilinx Virtex-5 SX95T FPGA and requires a processing latency of 420ns, resulting in a 39x speedup over software.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Electronic warfare (EW) is defined as military action involving the use of electromagnetic and directed energy to control the electromagnetic spectrum or to attack the enemy [6]. One possible task of an EW system is to sort and classify received pulses from a dense environment of hostile radars so that the pulses can be processed; this process is known as pulse deinterleaving. The deinterleaving process needs to be completed with very low latency to support timely decision making required in modern EW environments. One method for deinterleaving streams of incoming radar pulses is incremental clustering.

Clustering is the unsupervised partitioning of similar data samples into groups called clusters. The goal is to create clusters in which members of a particular cluster are as similar as possible to one another, and as different as possible from members of other clusters. Many clustering algorithms require a complete dataset to be present *a priori* and require multiple passes through the data in order to produce a result. Such algorithms are referred to as non-incremental. On the other hand, an incremental clustering algorithm considers each input data point only once, at which point it assigns it to a cluster. Such a technique allows streaming data, such as radar pulse descriptor words (PDWs), to be clustered in real-time.

This thesis discusses the implementation of an incremental clustering algorithm on a field-programmable gate array (FPGA) for the purposes of radar pulse deinterleaving and emitter identification. Building on previous clustering research we present *ICED*, an algorithm for the *I*ncremental *C*lustering of *E*volving *D*ata. This algorithm is a hybrid of existing clustering approaches and was developed to suit the particular needs of our deinterleaving application. In addition to the benefits of reconfigurability and cost, the high performance of current FPGA technology lends itself well to the low-latency real-time requirements of deinterleaving in modern EW systems. We'll discuss the steps of implementing the algorithm in hardware and present results for simulated radar data.

The rest of this thesis is arranged as follows. Chapter 2 motivates the need for low-latency pulse deinterleaving, presents some possible solutions, and provides an overview of incremental clustering. Additionally, it presents related work on both deinterleaving and FPGA implementations of classification algorithms. Chapter 3 introduces *ICED* and discusses its application to the deinterleaving problem. Chapter 4 discusses the details of implementing the *ICED* algorithm in an FPGA. Chapter 5 presents experimental results including performance metrics. Chapter 6 concludes with a summary and a discussion of future work. A complete list of acronyms is provided in Appendix A.

Chapter 2

Background

This chapter motivates our research by presenting relevant background information. We start by covering the basics of EW and discuss the importance of pulse deinterleaving. Next we look at a several different deinterleaving approaches and provide an overview of incremental clustering. Finally we present related work in both pulse deinterleaving and FPGA implementations of classification algorithms.

2.1 EW Systems

An EW system can be broken down into three major components, electronic support (ES), electronic attack (EA), and electronic protection (EP) [7]. ES is responsible for receiving incoming signals and collecting information about the EW environment, EA is responsible for disrupting or jamming adversary's EW systems, and EP is responsible for protecting against adversary's EA [1]. These three components, however, are not independent. Instead, a modern EW system is composed of ES, EA, and EP components which must work together in order to perform their individual tasks. For example, ES data can be used to queue EA and EP systems, and EP may use forms of EA in order to protect itself [8].

A simplified block diagram of an ES system is shown in Figure 2.1. A typical system is composed of a passive antenna and a receiver which down converts the high frequency RF signal received by the antenna to a lower intermediate frequency (IF) which can be processed more easily. The IF signal is then brought into the digital domain using an A/D converter. Operating on the digital samples the pulse measurement block produces PDWs which are a collection of measurements that describe the characteristics of the signal. The PDWs are fed to the ES processing block whose job is to sort received signals and identify the number and types of radars in its signal environment. Doing so provides situational awareness to ES applications and aids in response selection, such as the appropriate jamming technique, in EA and EP applications [1]. A typical environment consists of both uncooperative hostile and cooperative friendly radar emitters. Unlike a communication receiver which is aware of the signals it receives, an EW receiver has the added complication of not necessarily knowing the characteristics of incoming uncooperative radar signals [1]. Early EW receivers used a man-in-the-loop to help identify and track radar emitters but modern environments are far too complex and therefore require some form of automation for these tasks. The next section will discuss details of the first step in ES processing, pulse deinterleaving.

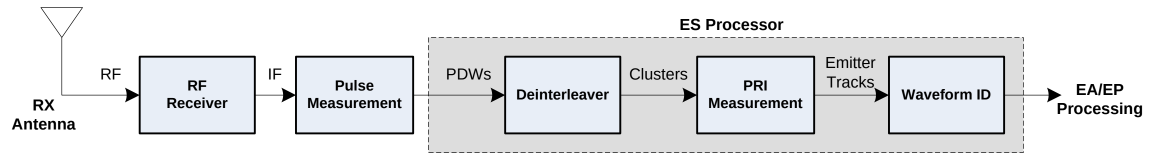


Figure 2.1: ES Block Diagram

2.1.1 Pulse Deinterleaving

An ES receiver is typically composed of either a single channel or a collection of channels covering a particular frequency band of interest. Since the signal environment can contain many radars the EW receiver will receive a single stream of input corresponding to the combination of all radars emitting at a particular time. In order to make intelligent use of the received signals one of the first tasks of the ES system is to separate the combined, or interleaved, signals into individual streams. This process is known as pulse deinterleaving and is demonstrated in Figure 2.2. The top two plots represent periodic pulse trains emitted from two individual radars. The center plot shows how the interleaved signals will appear at the EW receiver. The bottom plots represent the successful deinterleaving of the received signal, which in a perfect scenario, should identically match the the top plots.

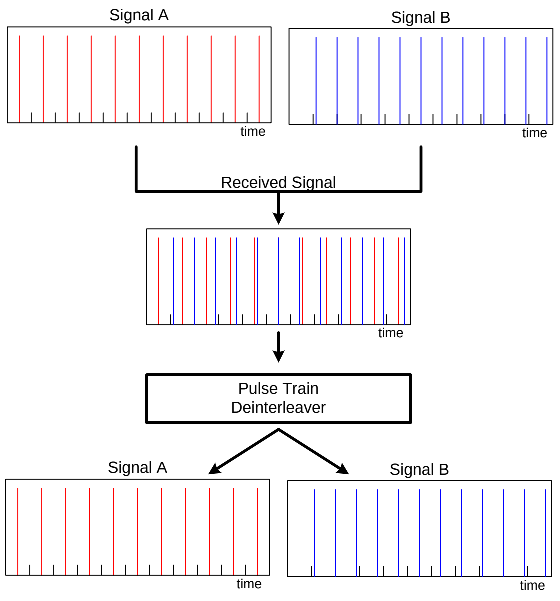


Figure 2.2: Deinterleaved Example [3]

Radars transmit in one of two modes, continuous wave (CW) or pulsed. In CW mode the radar constantly transmits an output signal while in pulsed mode the radar alternates between transmit and receive operation. The radars we are concerned with typically operate in pulsed mode. Generally speaking, an entire stream of pulses from a particular radar will have similar characteristics. We can sort the received pulses based on their characteristics, in turn deinterleaving a single receive stream into individual streams, one for each emitter. The following section will discuss the specific characteristics of radar pulses.

2.1.2 Pulse Descriptor Words

ES receivers measure pulse characteristics as each pulse is received. For every pulse the measured parameters are packed into a structure called a pulse descriptor word (PDW) and passed along for emitter identification processing. PDWs often contain information on pulse amplitude (PA), carrier frequency (RF), pulse width (PW), time of arrival (TOA), and angle of arrival (AOA). Although it can not be measured directly on an interleaved pulse train, another important characteristic useful in correlating a pulse to a particular emitter is pulse repetition interval (PRI) or its reciprocal, pulse repetition frequency (PRF). Figure 2.3 illustrates some of these parameters.

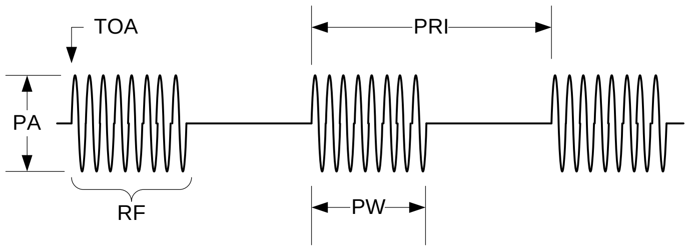


Figure 2.3: RF Pulse Parameters

Pulse Amplitude

Pulse amplitude is simply the strength of the received signal as seen by the receiver. The PA can be heavily influenced by the signal environment and the transmitter / receiver geometry.

Carrier Frequency

Carrier frequency is the sinusoidal frequency of a particular pulse. To complicate matters some radars use frequency modulation in which they vary the frequency over the duration of the pulse (intrapulse modulation) in an effort to improve range resolution through pulse compression [2]. Additionally, frequency agile radars periodically hop between numerous frequencies in an effort to allude or confuse an enemy receiver [2, 9]. For simplicity we focus most of our attention on fixed frequency emitters.

Pulse Width

Pulse width is the duration of time for which a pulse is being transmitted. Most radars have fairly low duty-cycles in order keep the average output power significantly low when compared with its peak output power [9].

Time of Arrival and Pulse Repetition Interval

Time of arrival is the system timestamp corresponding to the start of of a particular pulse. TOA is an important measurement required for determining the PRI, or the time period between successive transmitted pulses from a particular radar. PRI determines the maximum unambiguous range of the radar [9]. As with the RF parameter, PRI need not remain constant for all time. Instead radars are capable of transmitting staggered or jittered PRIs to improve range ambiguity or as an EP technique [7].

Angle of Arrival

Angle of arrival is a measurement which determines the direction of the transmitter with respect to the receiver. Though a highly reliable characteristic, it is one of the most difficult to measure.

The range and resolution requirement of PDW parameters can vary greatly from system to system based on expected application, performance requirements, and measurement accuracy. Table 2.1 provides an example of maximum range, bit-width for fixed point representation, and corresponding resolution for each parameter. The ranges tend to represent worst case scenarios and typical systems would unlikely require the maximum range for each parameter.

Table 2.1: A Possible PDW Format [1]

Parameter	Range	Bit Width
Frequency	DC - 32 GHz	15 (1-MHz resolution)
Pulse Amplitude	0 - 128 dB	7 (1-dB resolution)
Pulse Width	0.05 204 μs	12 (0.05 μs resolution)
TOA	0 - 50 sec	30 (0.05 μs resolution)
AOA	360 degrees	9 (1 degree resolution)

2.2 Deinterleaving Methods

Though there are numerous methods for deinterleaving pulse trains, we focus on techniques that operate on the measured PDW parameters of received signals as this is the typical information available at the output of an EW receiver front-end. Generally speaking, deinterleaving algorithms fall into one of two categories, time difference of arrival techniques and multi-parameter methods [3]. These approaches are compared and contrasted in the following two sections.

2.2.1 Time Difference of Arrival Methods

Time difference of arrival, or TOA methods for short, use only the measured TOA parameter in the deinterleaving process. Popular methods include histogramming approaches such as Cumulative difference (CDIF) histogramming [3] and Sequential difference (SDIF) histogramming [10].

The CDIF algorithm is as follows. Assume a collection of N pulses is collected over a particular time period. The difference between the TOA for adjacent pulses is calculated resulting in $N-1$ values. For example, the i^{th} difference d_i is calculated as

$$d_i = |\text{TOA}_i - \text{TOA}_{i+1}| \tag{2.1}$$

A histogram of the difference values is constructed with a bin corresponding to each unique calculated difference. Bins with counts exceeding a particular threshold are determined to be due to a constant PRI emitter. Pulses contributing to this detected emitter are removed from the interleaved sequence and the histogram is re-generated for the remaining pulses. This process continues until no histogram bins exceed the threshold. Next, the second differences are calculated. These are the distances between a particular pulse and its neighbor two pulses away. These difference values are cumulatively added to the previous histogram and constant PRI emitters are again removed. Figure 2.4 illustrates the the CDIF algorithm up through two difference measurements.

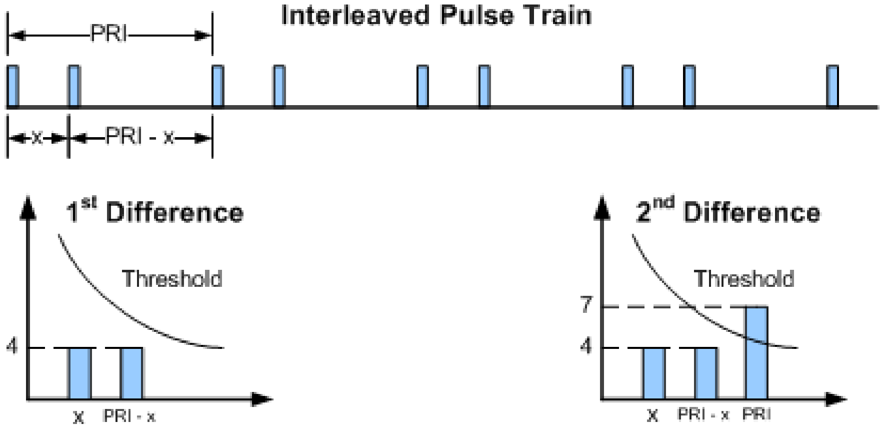


Figure 2.4: CDIF for 1st and 2nd differences [4]

While the CDIF algorithm cumulatively adds subsequent difference calculations to the existing histogram, SDIF starts a new histogram for each difference. Building on the TOA approach, similar algorithms such as the TOA folding algorithm and sequence search algorithm, have been developed in order to provide better results for staggered and jittered PRI emitters [10].

2.2.2 Multi-Parameter Methods

Multi-parameter methods seek to improve deinterleaving accuracy by using more than just the TOA parameter. In addition to TOA they attempt to sort incoming pulses with additional information provided by the PDW such as as RF, PW, and AOA. We can think of each received pulse as a point in an m-dimensional space where m is equal to the number of PDW parameters used to describe the input [3]. By plotting the PDWs we can hope to detect groups or clusters of points, where each cluster corresponds to a unique radar emitter. Figure 2.5 illustrates this point for $m = 2$ with a scatter plot of PW vs. RF for three emitters. The three distinct clusters correspond to the three emitters and the cluster centers of mass are the assumed actual RF and PW for each emitter. The clusters are clearly separated which is not often the case in other clustering applications. Since a radar can receive transmissions for other nearby radars in it's environment, the radar characteristics will likely vary from emitter to emitter in order to decrease interference. Therefore, the resulting clusters should often have a clean separation between them. The clustering process is used to both identify the radars based on their PDW characteristics and filter out clusters formed due to noise or interference from those representing true emitters.

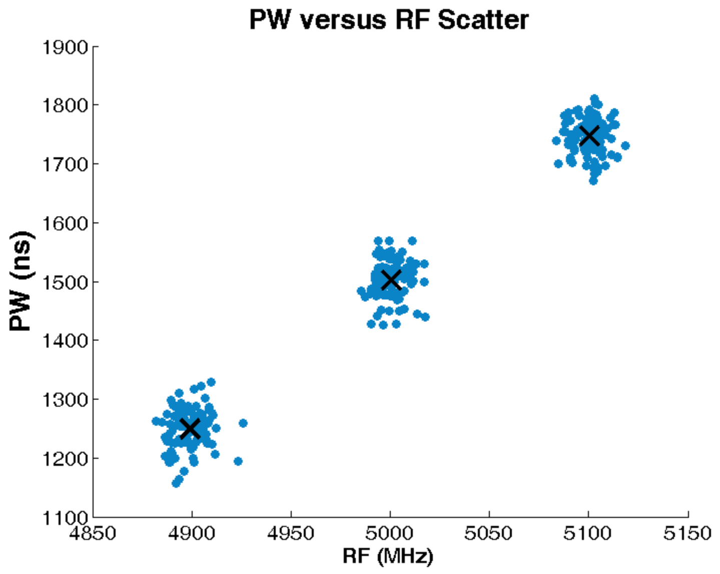


Figure 2.5: PW / RF Plot For Three Emitters

It would seem logical to use as many of the PDW parameters as possible in a multi-parameter deinterleaving approach. The problem, however, is that the accuracy of each parameter can vary greatly with respect to the others, and in actual EW systems not all parameters will always be measured.

Since it is difficult to quickly change the location of a radar, AOA is the most stable and valuable parameter [1]. However, AOA is the most difficult parameter to measure because it requires multiple antennas. RF can be a useful parameter but it can result in the assumption of multiple emitters in the case of a single frequency hopping radar. Additionally, in cases where two or more pulses overlap in time at the EW receiver, the measured RF may be a combination of the overlapping pulses. Though relatively easy to measure, PW is a less valuable parameter for two reasons [3]. First, as in the RF measurement, overlapping pulses will produce an erroneous measurement. For example, when two pulses overlap at the receiver the individual pulses can not be uniquely identified and will result in the detection of a single pulse with a PW greater than either of the contributing pulses. Figure 2.6 shows an example of overlapping pulses producing an incorrect PW measurement. Since the emitters in A and B have different PRIs they will eventually coincide producing a single long pulse. Secondly, the PA is not particularly useful in the case of deinterleaving since the amplitude of a received signal can vary greatly due to the range from the emitter to the EW receiver, fluctuations in the antenna pattern, and multipath effects.

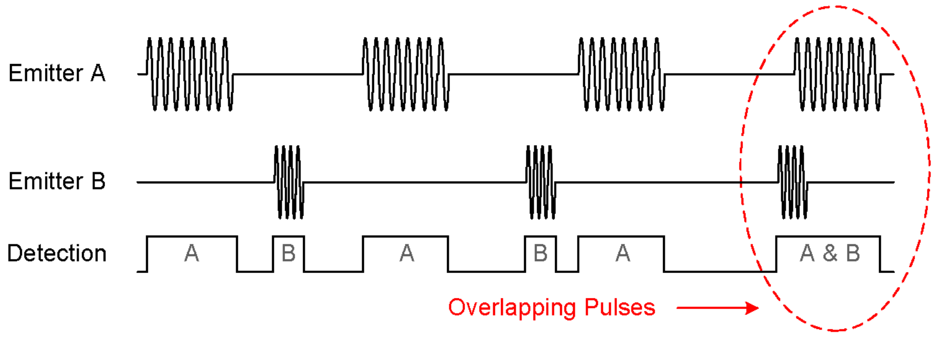


Figure 2.6: Interleaved Overlapped Pulses

2.2.3 Latency Requirements

We have discussed the need for a low latency real-time deinterleaving process but we have neglected to define what we mean by low latency. One way to determine the acceptable processing latency is to consider the frequency of receipt of PDWs in a typical emitter scenario. The overall processing latency is considered because an incremental algorithm requires a serial rather than pipelined implementation. This is true because each input must be fully processed in order to update clusters prior to accepting the next input. Assuming a single serial stream of PDW inputs, the maximum allowable processing latency will be a function of the PRFs of the contributing emitters. Assuming n emitters with corresponding PRFs, $\{P_0, \ldots, P_{n-1}\}$, the required latency r is defined as

$$r = \frac{1}{\sum_{i=0}^{n-1} P_i} \tag{2.2}$$

PRF values are commonly classified into one of three categories: low, medium, and high. The particular ranges can vary based on the operating conditions and characteristics of the radar, Table 2.2 gives typical values for an X-Band radar (8 - 12 GHz).

Table 2.2: Typical X-Band PRFs [2]

PRF Category	PRF Range (KHz)
Low	0.25 - 4
Medium	10 - 20
High	100 - 300

Let's look at the required latency for each of these categories by choosing a particular PRF value in each range and varying the number of emitters. We'll use 2KHz, 15KHz, and 200 KHz corresponding to low, medium, and high PRFs. If we assume all n emitters operate at the same PRF, P, we can simply equation 2.2 to

$$r = \frac{1}{n * P} \tag{2.3}$$

Assume a scenario where a deinterleaver is part of a narrowband EW system covering 250MHz of bandwidth. For such a system we'll define low emitter density to correspond to 1 - 4 emitters, medium density to correspond to 5 - 8 emitters, and high density to correspond to 9 - 12 emitters. Using equation 2.2 the maximum allowable latencies based on emitter density can be calculated. The results are shown in Figure 2.7 which plots latency versus number of emitters for high, medium, and low PRFs.

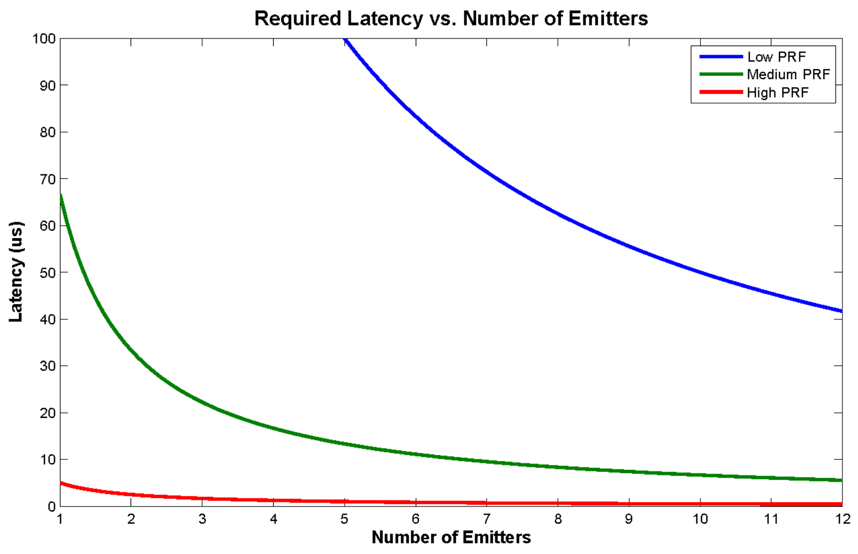


Figure 2.7: Min Processing Latency vs. Number of Emitters

As one would expect, the largest tolerable processing latency corresponds to low PRF emitters (high PRI). This is simply because the time between adjacent pulses of a low PRF emitter is is large. In even the densest low PRF scenario, consisting of 12 emitters, the maximum tolerable latency is approximately $40\mu s$ per pulse. On the other hand the requirements for a high PRF scenario are much more stringent.

Figure 2.8 shows an enlarged view of the high PRF case. A single high PRF emitter requires a maximum processing latency of $5\mu s$, and the most dense scenario, consisting of 12 emitters, requires approximately $500ns$. It's clear that if we are to handle even a medium density of high PRF emitters our maximum processing latency will have to be less than $1\mu s$.

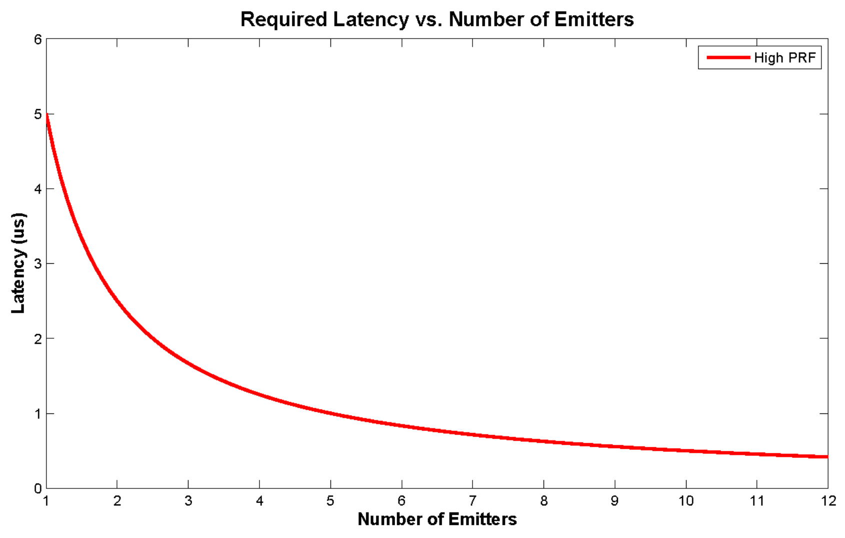


Figure 2.8: Min Processing Latency vs. Number of Emitters - High PRF

2.3 Clustering

Clustering is the unsupervised partitioning of similar data samples into groups called clusters. While a supervised classification process attempts to assign new data to predefined categories, an unsupervised approach determines the categories based on the input data. Though all clustering algorithms have the same goal of categorizing data based on similarities, there exist numerous approaches and countless algorithms.

2.3.1 Types of Clustering

There are many ways to characterize a particular clustering algorithm. Jain et al. [11] present a comprehensive survey of clustering approaches. We discuss the details of some of the most defining categories here. Keep in mind that the categories are not mutually exclusive and quite often approaches will be composed of a combination of techniques.

Hierarchical vs. Partitional

A hierarchical algorithm creates a tree structure of clusters in which each cluster can be composed of a hierarchy of sub-clusters. Using an approach of splitting larger clusters, or merging smaller ones, the root of the tree will contain all the data while the leaf components will contain fewer pieces of data. The alternative, and more common approach is partitional clustering. In partitional clustering, a single partition of the data is created consisting of unique clusters in which each data point belongs to a single cluster.

Agglomerate vs. Divisive

Agglomerate algorithms start with clusters each containing a single data point. Clusters are then successively merged based on their similarity, creating larger clusters. A divisive technique works in reverse. Here the goal is to successively split large clusters into smaller clusters until a defined stopping criteria has been reached.

Hard vs. Soft

A hard clustering is one in which each data point is a member of a single cluster. In soft clustering there is not a one-to-one mapping of objects to clusters. Instead objects are allowed to belong to multiple clusters with varying strengths.

Non-incremental vs. Incremental

Clustering algorithms can be classified as non-incremental, meaning they require all data to be present before learning takes place, or incremental, in which case they process input data data one at a time never requiring old data to be revisited. In this thesis, the focus is on incremental clustering.

2.3.2 Incremental Clustering

Because an incremental algorithm can operate without having to revisit old data it is more applicable to situations with very large data sets that can not fit in memory, and streaming data sets which are being produced for long periods of time. Additionally, based on the incremental nature, these algorithms are better suited to data whose characteristics might be evolving over time. For these reasons, incremental algorithms are of particular interest to pulse deinterleaving and we explore them further in this section.

One of the first incremental clustering algorithms was the leader algorithm described by Hartigan[12]. This algorithm attempts to partition a set of data samples into a number of disjoint clusters. The desired number of clusters is not specified *a priori* but instead a distance metric and maximum threshold must be defined.

The algorithm gets its name from the fact that cluster coordinates are set equal to the input used to create the cluster. This input is known as the leader. Once the cluster is established new members can be assigned to it, but its coordinates are never adjusted from those of the leader. The decision to assign an input to an existing cluster instead of creating a new cluster is made based on distance. If the current input's distance is within a threshold from an existing cluster then it is assigned to that cluster, otherwise a new cluster is created. It should be noted that the algorithm assigns inputs to the first cluster that satisfies the threshold requirement rather than searching for the closest cluster.

The leader algorithm requires only one pass through the sample data so it has a speed advantage over non-incremental techniques. A second advantage is that it does not require prior knowledge of the number of expected clusters. A drawback, however, is that the clustering is highly dependent on the order of the input samples. This dependence exists because samples are assigned to the first cluster that meets the threshold requirement rather than finding the nearest cluster.

Since the introduction of the leader algorithm, the emergence of the field of data mining and in particular its application to streaming data has led to the development of many different flavors of incremental clustering. Each algorithm attempts to improve clustering accuracy by adjusting the algorithm to the specific characteristics of the desired application. Though the overall flow of most incremental clustering algorithms is similar, they differ in their initialization or their conditions for creation or deletion of clusters. Let's look at some possible choices for each of these decisions.

Initialization

In non-incremental approaches such as the K-Means algorithm, initialization of cluster centers is often based either on a random selection of data points or a selection of data points based on the statistics of the data set. For example, k centers may be chosen uniformly from the distribution of given data points. These methods are not possible for incremental algorithms used for clustering streaming data because the entire data set is not available at initialization time. Incremental algorithms whose number of clusters evolve over time, such as the leader algorithm [12] or ECM [13], often start with just a single cluster defined by the first data point. Other algorithms, such as GenIc [14], start by initializing m clusters using the first m data points.

Cluster Management

Many different techniques exist for creation of new clusters or adjusting or deleting existing clusters. Methods like [12, 13] use a distance threshold for assigning new data. They assign new data to an existing cluster when the distance between the new data and the existing cluster is less than the threshold and they create a new cluster when the distance is greater than the threshold. This technique can lead to large number of clusters when the thresholds are too low, or when the maximum number of centers is not capped.

Several options exist for updating cluster centers upon the addition of new data. The leader algorithm does not update the center at all. A cluster center is defined by the data sample used to create it, the leader, and although subsequent data can be assigned to the cluster, the center is never moved. In ECM, the cluster centers are allowed to grow and move based on a maximum allowable cluster radius. This radius is defined as the distance from the cluster center to the furthest member point. A cluster center is not adjusted when a new member falls within the current radius, but only when the new member is between the current and maximum radius. Since not all additions to a cluster result in an update, this method requires less processing at the expense of cluster center accuracy. A final method, used in GenIc and discussed in [15], is to adjust the cluster center for every new member. This approach results in cluster centers that that represent an average of their members. Of course, the penalty for this increased accuracy is processing time.

One may want to delete a cluster for several reasons. First, in the case of streaming data, it is quite possible that the data characteristics will change over time [15, 16] necessitating the need to remove old, or stale, clusters. Second, in cases where the data contains outliers, it may be desirable to remove the clusters corresponding to this spurious data; of course, one must first identify the clusters as such.

To handle outliers, GenIc periodically evaluates the strength of each cluster and removes the weakest. The cluster strength corresponds to the weight, or the number of members, of a particular cluster. After processing a fixed number of inputs, called a generation, a probability of survival is calculated for each cluster based on the ratio of its weight to the total weight of all clusters. A random threshold is generated and all clusters with a probability of survival less than the threshold are killed off. Though this procedure is effective at weeding out spurious outlying clusters, the GenIc algorithm does not provide a method for deleting old clusters in the case of evolving data.

Reyzin [15] proposes a method that solves both the outlier and evolving cases simultaneously by fading clusters over time. Cluster weights are increased by one each time a new member is added, and are decreased by one periodically at specified time intervals. When a cluster weight reaches zero it is deleted. When the input stream characteristics change, clusters that are no longer being assigned new members will slowly fade to zero and be removed. The same is true for clusters that were created due to spurius data since they'll infrequently receive additions and will therefore fade quickly.

2.4 Related Work

In this section we present a survey of prior work related to our research. We split the related work into two categories, pulse deinterleaving and FPGA implementations of classification algorithms, and discuss them independently.

2.4.1 Pulse Deinterleaving

It's been established that pulse deinterleaving is an essential part of an EW system and for that reason it has received a significant amount of attention. The earliest deinterleaving methods involved histogramming of the TOA parameter and were presented by Mardia [17] and Milojevic and Popovic [4]. Popular algorithms include CDIF, SDIF, TOA folding, and sequence search.

With the popularity of neural networks came new multi-parametric approaches. In [18] Granger et al. compared four neural networks: Fuzzy Adaptive Resonance Theory (FA), Fuzzy Min-Max Clustering (FMMC), Integrated Adaptive Fuzzy Clustering (IAFC), and Self-Organizing Feature Mapping (SOFM), for the purpose of deinterleaving radar pulse trains. They compared algorithms based on clustering quality, convergence time, and computational complexity, and found SOFM and FA to be the best candidates. SOFM provided the best quality but was considered complex and required long convergence times. Though less accurate, the faster run times of FA made it a good choice for lower latency cases like threat alert systems.

More recently, Ataa and Abdullah [10] proposed a deinterleaving system which combines an FA clustering of RF and AOA followed by a PRF identification block based on TOA folding and CDIF.

In [19] Liu et al. propose two clustering algorithms for deinterleaving. Both cluster high-dimensional vectors representing the discretely sampled pulses. The first algorithm assigns vectors to clusters based on a distance metric and periodically chooses to split or merge clusters based on the Minimum Description Length (MDL) criterion. The second algorithm is an online competitive learning approach which attempts to determine adaptively inter- and intra- cluster threshold values using training sequences of known truth data. Though these algorithms differ from the research presented here in the fact that they do not use PDW data, they are mentioned here for completeness.

2.4.2 FPGA implementations of Classification Algorithms

Though a large body of work exists in pulse deinterleaving, discussions of hardware implementations are scarce. The majority of work proposes a method, describes the algorithm, and presents results derived from a simulated software environment. Therefore, we extend our scope and discuss relevant FPGA implementations of classification algorithms in general. The increasing popularity of data mining coupled with the emergence of higher performance FPGA devices has led to a trend of FPGA implementations of classification algorithms for real-time applications.

Though not an incremental clustering approach, there have been numerous hardware implementations of the K-Means algorithm. Popular applications include image processing [20, 21, 22] and document classification [23].

Additionally, there has been research in hardware implementations of neural network algorithms. Sanchez-Solano et al. discuss a general architecture for rapid and flexible development of embedded fuzzy controllers [24]. The design consists of their generic processing blocks, called fuzzy inference modules (FIM), as well as a soft-core Microblaze processor for control. Using a Xilinx Spartan-3 Development board, they implemented a fuzzy controller for parking an autonomous robotic vehicle.

Arifin and Cheung [25] implemented a time adaptive clustering (TAC) algorithm for logical story unit segmentation of digital video in a Xilinx Virtex-II Xc2V3000. In order to exploit parallelism within the FPGA the design implements 10 clustering blocks in parallel, each able to calculate the distance between itself and the current input simulataneously. Running at 40 Mhz the FPGA implementation demonstrates a 27x speedup over a 3.4 GHz Pentium 4 with 1 GB of RAM.

Baldanza et al. [26] implemented a 2-D cellular neural network for an on-line clustering algorithm pertaining to high-energy physics. Juang and Tsao [27]describe the implementation of a Type-2 Self-Organizing Neural Fuzzy System (T2SONFS) in an FPGA where the rule generation portion of the fuzzy system is performed by an on-line clustering algorithm. The proposed T2SONFS algorithm is implemented in a Xilinx Virtex-4 running at 54 MHz.

In more recent work, Kyrkou and Theocharides [28] present SCoPE, a systolic chain of processing elements for support vector machine (SVM) classification. They provide a thorough discussion of the details of mapping the algorithm to a Xilinx Virtex-5 FPGA. The final design ran at 100 MHz and chip area was the limiting factor in the design as the implementation of 64 multipliers consumed 100% of the FPGA's DSP resources.

There has been some research concerning the difficulties in mapping clustering algorithms to reconfigurable hardware platforms. Changbin and Wahab [29] proposed a hardware implementation of a fuzzy clustering technique based on Discrete Incremental Clustering. They claim that the difficulties in mapping neural network algorithms to FPGAs is mainly due to complex arithmetic such as multiplication and division and note that for powers of 2 these operations can be replaced by simpler bit shifts. In an attempt to reduce complexity and ease the hardware implementation they propose a multiplier free architecture, however, they neglect to discuss the details. Estlick et al. [30] discuss algorithmic tradeoffs in mapping the K-means algorithm to an FPFA. They provide two alternatives to the Euclidean distance metric in an effort to avoid costly multiplication operations and discuss the effects of fixed-width integer arithmetic over a floating point implementation.

2.5 Conclusions

In this chapter we presented a background of EW systems and looked at various methods for the deinterleaving of pulse trains. We provided an overview of clustering and focused on incremental clustering algorithms in particular. Finally, we presented related work in the areas of pulse deinterleaving as well as FPGA implementations of classification algorithms. In the following chapter we present the details of our proposed incremental clustering algorithm which we call *ICED*.

Chapter 3

ICED Algorithm

In the previous chapter we surveyed various methods for radar pulse deinterleaving as well as several FPGA implementations of classification algorithms. Due to the requirements of a real-time pulse deinterleaver we decided to implement a hybrid incremental clustering approach based on a collection of existing algorithms. This chapter will present our algorithm, *ICED*, and discuss the motivation for the algorithmic choices we have made. To aid in the evaluation of tradeoffs along the way and to alleviate the long process of testing each change in hardware, we created a software model for design exploration. We start by looking at the requirements for an effective incremental clustering deinterleaver.

3.1 System I/O

A pulse deinterleaver is a single block in a more complicated EW system. The deinterleaver inputs will be PDW parameters as measured by the EW receiver's parameter estimation block. In Section 2.1.2 we looked at the possible PDW parameters and discussed their individual properties. We have decided to cluster based on two of these parameters, RF and PW. Although we indicated that PW is not the most reliable parameter, it is easy to obtain and is generally available in all EW systems. For the same reason we have decided against using AOA as it is more difficult to measure and

Table 3.1: Resolutions based on 16-bit PDW Parameters

Parameter	System A	System B
Frequency Bandwidth	32 GHz	250 MHz
Frequency Resolution	500 KHz	4 KHz
Pulse Width Resolution	1 ms	1ms
Time Resolution	15 ns	15ns

less likely to be available. Keep in mind, however, that the choice of these particular parameters will not affect the details of our 2-D clustering algorithm and in fact it would be trivial to swap the parameters on which we decide to cluster. In addition to RF and PW, the deinterleaver will also be provided with TOA for each pulse. This measurement will aid in the cluster fade operation.

For each input, the pulse deinterleaver will output the coordinates of the assigned cluster as well as the cluster ID number. The cluster coordinates represent the running average of its members and are useful in tasks such as keeping a down converter appropriately tuned to a signal of interest. In an EA or EP scenario, the cluster ID can serve as an index into a threat table, useful in decision making further down the processing chain.

Typically, PDW parameters are represented in integer format and correspond to a bin based on the quantized resolution of the measurement. We'll assume 16-bit RF and PW parameters and a 32-bit TOA parameter. Table 3.1 shows how these selections correspond to two different implementations, a wide band system covering a larger frequency spectrum and a narrow band system which could represent a single module in a channelized architecture.

It is possible that in certain scenarios one may know that the RF or PW parameters will occupy a range smaller than the full 16-bits. To account for this and improve cluster separation we keep a 16-bits representation within the algorithm but provide the ability to specify minimum and maximum expected values for each parameter. A 16-bit representation results in 65536 bins ranging from 0 - 65535. If we define the minimum and maximum expected values as *min* and *max*, respectively, the equation for finding the normalized value *norm* of a particular *input* is the following:

$$norm = \left[\frac{(input - min)}{(max - min)}\right] * 65535 \tag{3.1}$$

The reverse process needs to be performed on the output side to the coordinates corresponding to the assigned cluster. The equation for returning to the native value, *native*, from the normalized value, *norm*, is the following:

$$native = \frac{(max - min) * norm}{65535} \tag{3.2}$$

3.2 Algorithm Parameters

Ataa and Abdullah [10] discuss the properties of an effective clustering algorithm in an EW system. First they note that no prior knowledge of the number of characteristics or categories should be required. Secondly, they state that because of the high data rate of input streams, the algorithm should be able to cluster non-stationary inputs sequentially, without requiring long-term storage. Finally, they propose that the solution should lend itself well to a high-speed hardware realization. Robust low-latency deinterleaving algorithms are necessary for ESM systems to reliably identify and engage the correct targets [3]. If the ESM system does not work efficiently, radars can be misidentified or false radars can be generated resulting in limited and valuable EA/EP processing resources being wasted [31].

Taking these recommendations into account we have decided to implement an incremental clustering algorithm in an FPGA. Our solution does not require prior knowledge of the number of radar emitters, but it will use a fixed number of characteristics and will assume general ranges of expected data for those characteristics. Being an incremental algorithm, it contains no recursion over input data and requires no long-term storage. Finally, the parallel nature coupled with the simple yet effective approach of the algorithm make it an ideal candidate for a hardware implementation in an FPGA. In the next few sections we address some of the major design decisions with respect to the algorithmic options presented in Section 2.3.2.

3.2.1 Hard vs. Fuzzy

Our first decision was whether we wanted to pursue a hard- or fuzzy clustering method. Many proposed techniques [3, 10, 18, 19] discussed in Section 2.2 were fuzzy-based self-organizing neural networks (SONN). Though often proven to be effective in a simulated environment, not one of these papers discussed the ramifications of a hardware implementation. With the mindset of developing a simpler, and therefore more easily implementable, solution we decided to target a hard clustering approach.

3.2.2 Initialization

In a hardware implementation there will be a fixed amount of processing and memory resources. Therefore, in order to produce an implementable design there will exist some maximum number of clusters, k, that can be supported. We do not define that maximum at this point, as it will be a function of the FPGA device and the complexity of the algorithm; however, we keep it in mind when making decisions such as how to initialize cluster centers. In the previous chapter we presented several options for algorithm initialization. The basic options were:

1. Initialize all clusters based on a sample of expected values
2. Initialize all clusters to correspond the the first k inputs
3. Initialize a single cluster corresponding to the first input, and introduce new clusters based on distance thresholding.

The first option can lead to poor results if the assumption of expected values is incorrect. Additionally, because we do not want to assume any prior information regarding the characteristics or the number of clusters to be generated, we disregard this option. The second option is appropriate because it seeds all clusters with actual data. However, thinking in terms of our application this approach has two drawbacks. First, typical measurements containing some measurement noise from a single emitter will result in multiple clusters close to one another. Second, in an environment with a mixture of high- and low PRF emitters, it is possible that at initialization all clusters can be assigned to the high PRF emitter due to the much higher frequency of its pulses. Such a situation would cause an initial bias towards the high PRF source.

The final option seems to make the most sense with respect to our application. Since we have a general idea of the expected measurement jitter and typical separation of emitters in an actual environment, we can set a threshold corresponding to the maximum allowable cluster radius. The first input will create the first cluster. The second input will be assigned to the first cluster if it falls within the radius threshold, or will force the creation of a new cluster if it falls outside the threshold. This approach is then continued for all subsequent pulses. Later, we discuss the situation that occurs if all resources have been allocated and a new input requires the creation of a new cluster.

3.2.3 Distance Metric

We must also decide on a distance metric for measuring similarity between inputs and clusters. This is an important decision with respect to a hardware implementation because substantial complexity can be added based on the particular metric chosen. For example, the use of multipliers, and even more so, the use of dividers and square root operators in an FPGA will result in both area and performance degradations. The distance d between two points, x and y, according to the L_p norm is the following:

$$d = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}} \tag{3.3}$$

The most common distance metric is the L_2 norm, also known as the Euclidean distance:

$$d = \sqrt{\sum_{i=1}^{n} |x_i - y_i|^2} \tag{3.4}$$

A popular alternative to the Euclidean distance is the squared Euclidean distance because it contains no square root operation and can therefore be more easily implemented in hardware.

$$d = \sum_{i=1}^{n} |x_i - y_i|^2 \tag{3.5}$$

Estlick et. al [30] looked at two alternative distance metrics for an FPGA implementation of the K-means algorithm. The first was the L_1 norm, commonly referred to as the Manhattan distance:

$$d = \sum_{i=1}^{n} |x_i - y_i| \tag{3.6}$$

And the second was the L_∞ norm, also known as the Max distance:

$$d = max|x_n - y_n| \tag{3.7}$$

Looking at the Manhattan distance we see that this calculation is simpler than the squared Euclidean as it requires no multiplication. Such a metric would both provide higher performance and result in less area consumption. The benefit of the Max distance is that, for a fixed point representation, the number of bits remains constant as n increases. Additionally Estlick investigated a linear combination of their two alternatives. They found this method produced the best results but for higher dimensions ($n > 10$), however, the Manhattan distance performed well enough and was chosen based on it simpler hardware implementation.

The distance between members of the same cluster, or intra-cluster distance, is primarily a function of source emitter jitter and receiver measurement error. The distances between unique clusters, or inter-cluster distance, is a function of the operating characteristics of the emitter radars, and are typically sufficiently separated so as to not interfere with one another. Therefore, we expect inter-cluster distances to be large compared to intra-cluster distances and the use of the Manhattan distance should be more than adequate even for our low 2-dimensional case. However, to provide the most flexibility, the distance measurement will be self-contained in a single module to allow an area/performance tradeoff to be made more easily at the hardware level.

3.2.4 Updating Cluster Coordinates

We discussed three options for updating cluster centers:

1. Statically define the center based on the first assigned object (Leader algorithm)
2. Update only when the cluster radius increases.
3. Update for each assignment to a cluster.

Static cluster centers are defined by the first object assigned to it, and therefore can exhibit a bias towards the initiating object. Take, for example, a typical case in which the PDW parameters have some associated measurement noise. Using the leader approach, a cluster for a particular emitter might be initiated with a noisy sample, in turn creating a cluster center which contains some bias with respect to the true emitter PDW characteristics. Updating only when the cluster radius increases provides more accuracy at the expense of processing time. Because we are operating in incremental mode and concerned with overall clustering latency for each new input, an occasional processing savings when the center does not have to be adjusted is less valuable. The most accurate approach, and the one we have chosen, is to update the center upon each should converge to the true emitter values. Using a moving average filter we can accomplish this without storing prior assigned members therefore satisfying our requirements.

3.2.5 Cluster Weights

Our algorithm needs to handle evolving data as well as outliers caused by interference or overlapping pulses. We decided to use the method proposed in [15] and fade clusters over time by periodically decreasing their weights, and deleting clusters when their weight reaches zero. We expect pulses from a particular emitter to arrive at a fixed frequency based on the PRF of the radar so this approach is very applicable to our problem. Based on typical PRF values we can set a fade period such that outliers are removed and true emitters persist. Keeping a hardware implementation in mind, we also have to set a maximum cluster weight and decide on a course of action when the maximum is reached. More complicated scenarios can be explored, but for now we reset the weight to half of the maximum. This will provide a compromise between the accuracy of the cluster center based on prior statistics and ease of hardware implementation which requires only a simple bit shift operation.

3.3 Algorithm Details

In this section we discuss the details of *ICED* based on the algorithmic parameters just chosen.

3.3.1 Pseudocode

The algorithm is initialized by setting all cluster coordinates and weights to zero. The initial cluster is created using the coordinates of the first input and a weight of one. As subsequent inputs are received the following steps are performed. The whole number and fractional fade cycle components are calculated based on the elapsed time since the previous input. All cluster weights are decremented by the whole number of fade cycles. The distance from the input to each cluster center is calculated and the lightest (lowest weight) cluster is determined. If the distance to the nearest cluster is less than the specified threshold, the input is assigned to that cluster and the cluster's coordinates and weight are updated. If the distance to the nearest cluster exceeds the threshold, a new cluster is created by overwriting the lightest cluster. The algorithm details are as follows:

```
Configurable Parameters
  L: Fade cycle length
  D: Distance threshold
Inputs
  $x_i$: $i^{th}$ input PDW
  $t_i$: $i^{th}$ input timestamp
Algorithm Variables (Initialized to zero)
  $i$: Input index
  $j$: Cluster index
  $f_i$: Fade cycles to implement prior to clustering the $i^{th}$ input
  $r_i$: Fade cycle remainder for the $i^{th}$ input
  $c_j$: Coordinates of the $j^{th}$ cluster
  $w_j$: Weight of the $j^{th}$ cluster

For each input, $x_i$ do the following:

1. Cluster Fade
    a. Calculate fade cycles and remainder since the previous input
       $f_i = \frac{t_i - t_{i-1} + r_{i-1}}{L}$
       $r_i = t_i - (f_i * L)$
    b. Decrement all cluster weights
         $w_j = w_j - f_i$

2. Distance and Weight Measurements
   a. Let $n$ be the index of the nearest cluster
   b. Let $l$ be the index of the lightest cluster
   c. If $(w_n < D)$ goto Step 3, else goto Step 4
```

```
3. Assign input $x_i$ to cluster $n$
   a. Update cluster coordinates
      $c_n = \frac{c_n * w_n + x_i}{w_n + 1}$
   b. Increment cluster weight
      $w_n = w_n + 1$

4. Create a new cluster from input $x_i$
   a. Overwrite the lightest cluster
      $c_l = x_i$
   b. Reset the weight
      $w_l = 1$
```

3.3.2 New Cluster Creation

We'd like to elaborate on Step 4, the creation of a new cluster, which is reached when the distance from the current input to the nearest cluster exceeds the specified threshold and a new cluster needs to be created. We must consider this situation in two cases, the first when hardware resources exist to support a new cluster and the second when all resources have been allocated. Such a consideration is relevant because unlike a software solution where we can support a seemingly infinite number of clusters, a hardware implementation will have a fixed amount of processing resources. Ideally, there would exist enough resources to always have unallocated clusters in reserve, but the particular application will drive this hardware requirement.

At any point one can consider the number of active clusters equal to the number of clusters with a non-zero weight. Initially all cluster weights are set to zero and there are no active clusters. Cluster weights increase as new members are added and the weights of old clusters or outlying clusters will decrease as they are faded out, eventually reaching zero, at which point they can be considered deleted or inactive. By always selecting the lightest cluster for replacement with the newly formed cluster we can solve both situations, the existence and absence of hardware resources. When hardware resources are available, not all clusters are active, and we are guaranteed to select an inactive cluster with zero weight. When all hardware resources are allocated, all clusters have a non-zero weight, and we simply choose the lightest cluster.

Such a decision has advantages and disadvantages and can result in a correct or incorrect decision based on the situation. Lets first look at cases where this will result in the correct decision. In one situation we may be selecting an outlier whose weight is very low with respect to the other established clusters representing true emitters. Therefore, choosing the lightest cluster results in the correct decision by overwriting an irrelevant cluster. A second case to consider is when the lightest cluster represents an old emitter whose weight is being faded out but has not yet reached zero. Again, choosing the lightest cluster results in the correct decision because we have overwritten a cluster representing an emitter from which we are no longer receiving input.

It is possible, however, when choosing the lightest cluster to accidentally overwrite a relevant cluster. Consider the following situation with two clusters, A and B. Cluster A's weight is decreasing due to fading but the weight is still quite large. Cluster B's weight is increasing but is comparatively small because it represents a new emitter. In this case we'd overwrite the relevant active emitter, cluster B, instead of the older emitter, cluster A. Eventually, as the weight of cluster A is reduced enough it would be overwritten, but the timing of this is a function of its weight and the PRIs of the active emitters.

There are other ways to choose which cluster to overwrite, and more complicated methods can be implemented in the future. For now, we continue with overwriting the lightest cluster and concentrate our effort on achieving a hardware realization.

3.4 Conclusions

In this chapter we discussed an incremental clustering algorithm we developed called *ICED*. The algorithm uses a combination of approaches from existing algorithms and was constructed with our specific pulse deinterleaving application in mind. We presented the major algorithm parameters, discussed their relevance to deinterleaving, and provided a step-by-step description of *ICED*. In the next chapter we discuss implementing this algorithm in an FPGA.

Chapter 4

FPGA Implementation

In Chapter 2 we discussed several alternatives for implementing a radar pulse deinterleaver and decided on using an incremental clustering approach. Chapter 3 introduced *ICED*, an algorithm we developed in order to satisfy the particular needs of our application. In this chapter we discuss the process of mapping ICED to hardware, and in particular on to a Xilinx Virtex-5 FPGA. We start by presenting our goals for a hardware implementation. Next, we provide details on the target board and FPGA. Finally, we discuss the implementation details of the major design components.

4.1 Implementation Goals

Common goals in hardware design, and in particular FPGA implementations, are high levels of parallelism, modularity and configurability, and this design is no different. This section will discuss each of these goals independently.

4.1.1 Parallelism

One of the biggest motivators for a hardware implementation is performance. Quite often the way to obtain speedup over a purely software solution is to exploit the parallelism of an algorithm, and the highly regular architectures of FPGAs lend themselves well to such an implementation. Two common types of parallelism are data parallelism and task parallelism. Data parallelism occurs when the same operation needs to be performed on different pieces of data and is often found in loop constructs. Alternatively, task parallelism is the operation of different tasks on the same or different data.

In our particular case there are several opportunities for parallelization. The first and most obvious is the distance calculation which is an example of data parallelism. For each new input we must measure the distance to each existing center in order to find the nearest cluster. In software this is typically implemented by looping over all clusters and calculating the distance from the input one by one. However, with enough resources, a hardware implementation can perform each of the distance measurements simultaneously. Following the parallel distance calculation we still need to find the smallest distance. In software, this is an $O(n)$ operation where each of the n cluster distances would have to be visited once. By using a binary tree of comparators we can parallelize the search for the smallest distance and decrease the problem to $O(\log n)$.

In addition to data parallelism, *ICED* has several opportunities for exploiting task parallelism. As an example, two steps which must occur prior to the distance measurement are input normalization and cluster fading. In software these are implemented as order independent sequential tasks. In hardware they can be computed simultaneously improving overall algorithm latency. Task parallelism can also be achieved by allowing each cluster to calculate its updated position (the updated cluster coordinates based on the assumption that it will be assigned the current input) while determining the nearest cluster. Doing so can allow for a near instantaneous cluster update once the nearest cluster is determined.

All examples of parallelism opportunities just described assume an adequate amount of processing resources. If resources need to be time shared then the benefit of parallelization will be decreased. We examine some of these cases in greater detail in Chapter 5.2 when we discuss performance / area tradeoffs.

4.1.2 Modularity

Modularity plays an important role in most designs and especially in FPGA designs which are easily reconfigurable. A modular design is one that is composed of smaller blocks, or modules, where each module is as independent and self-contained as possible. Such an approach allows easy integration of components developed by multiple designers, but more importantly it also provides the ability to more easily compare design tradeoffs. Take, for example, the decision of which distance metric to implement, squared Euclidean vs. Manhattan. Given a modular design, such a comparison can be made by simply dropping the appropriate distance module into the design and testing. On the other hand, a non-modular design might require a significant amount of re-design in order to change the distance metric.

We aimed at making the design as modular as possible while at the same time keeping overhead low and performance high. In a truly embedded design where the same function will be performed for the life of the system, a less modular approach can be afforded in order to obtain the best performance, power, and cost. In an EW system it's quite possible that system requirements may change over time requiring retrofitting of components. Examples of modularity in our design include the careful partitioning of tasks such as normalization, distance measurement, cluster coordinate calculation, etc. into separate code modules. Additionally, we took advantage of "generate statements" which auto-generate HDL code at compile time depending on the design needs. Such statements were used to generate the appropriate structures based on the number of desired clusters, type of distance metric, and bit-width requirements.

4.1.3 Configurability

While evolving design requirements tend to occur on a large time scale and necessitate significant design changes, there are also cases where a situation might require a minor change in a short period of time. Here a well thought out design with multiple modes of operation or configurable parameters which can be changed in or as close to real-time as possible is invaluable. There are several cases in our design were we focused on configurability. For example, host accessible registers were provided for real-time configurability of distance thresholds, parameter ranges for normalization, fade cycle lengths, and maximum cluster weights.

We provide specific examples of our modularity and configurability efforts in Section 4.3 when we present details on the major design components.

4.2 Board Overview

We have chosen the X5-400M XMC module [5] from Innovative Integration as our implementation platform. A photo of the board is shown in Figure 4.1. The XMC module plugs into a carrier card which can provide an array of host interfaces such as PCI, PCI-X, or PCI-Express.

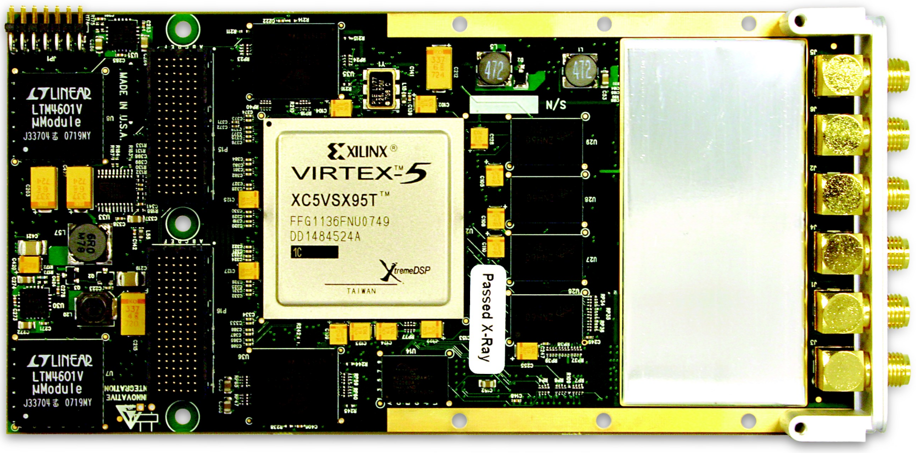


Figure 4.1: X5-400M Module [5]

The X5-400M, whose block diagram is shown in Figure 4.2, targets real-time signal processing applications and features two 14-bit 400 MSPS A/D converters; two 16-bit 500 MSPS D/A converters; 512MB of DDR2 DRAM; 4MB of QDR-II SRAM; and both high-speed serial and general purpose I/O. At the heart of the board is a Xilinx Virtex-5 SX-95T FPGA [32] providing significant processing resources as well as access to the A/D and D/A channels, on-board memory, I/O, and the host interface. Clocks sources available to the FPGA include onboard oscillators at 200MHz and 250MHz as well as a connector for providing an external clock.

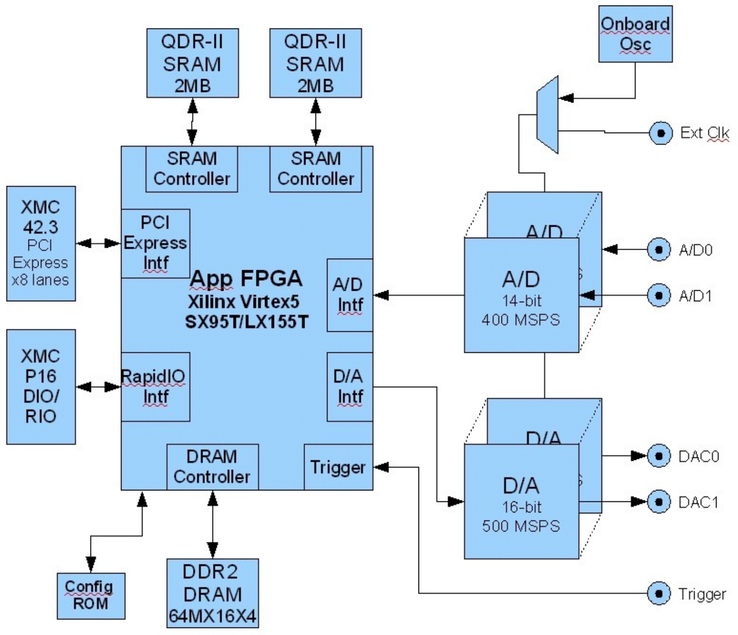


Figure 4.2: X5-400M Block Diagram [5]

Provided with the board is a development kit containing template code for both the FPGA interfaces as well as a board-level testbench. This code decreases the often lengthy time required to get a design working on a new board for the first time.

Though our pulse deinterleaver is clustering pre-processed PDWs and does not need access to the A/D or D/A converters, a fully integrated system could very well perform the pulse measurements, deinterleaving, and response technique on the same board requiring for such components. We focus entirely on the construction of the deinterleaver with the understanding that our modular design can be easily integrated into a complete EW system.

4.3 Major Components

As discussed in Section 4.1 one of our main design goals was modularity. Figure 4.3 shows a block diagram of the major top-level modules within *ICED*.

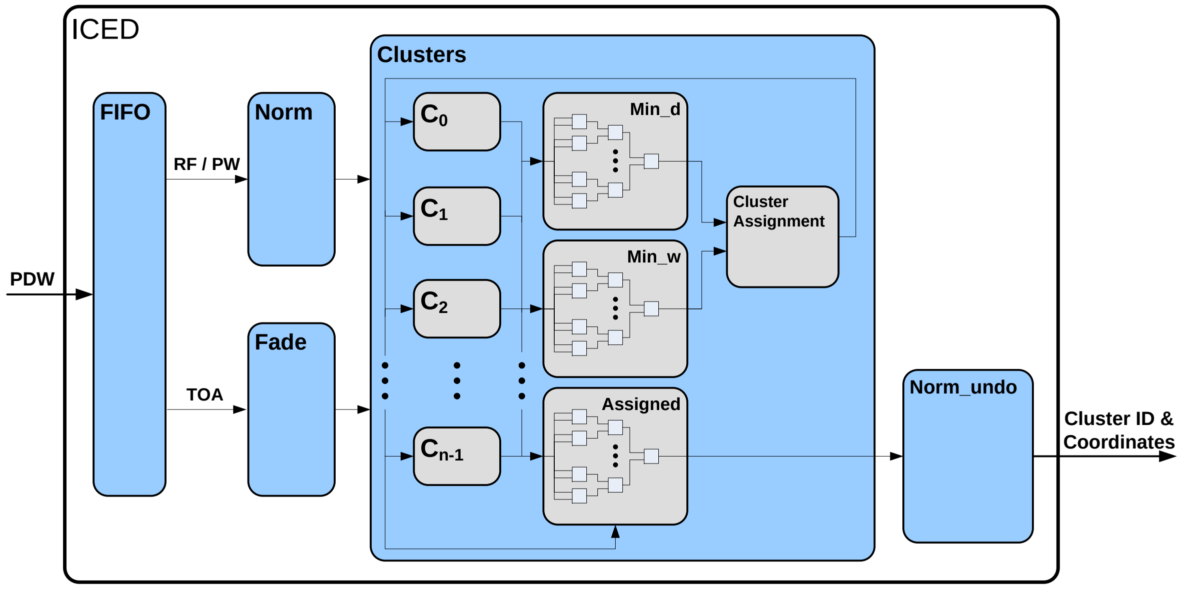


Figure 4.3: ICED - Top-level block diagram

The following is a brief description of the general processing flow. As PDWs are received they are placed in a FIFO to decouple the receipt and processing of inputs. After reading a PDW from the FIFO, parameter normalization and fade calculations are performed. These results are passed to the *Clusters* module which measures the distance to all active clusters and decides to which cluster the input should be assigned. Finally, the coordinates of the assigned cluster are converted from normalized to native units and output along with a unique cluster ID. We now take a closer look at each of these modules.

4.3.1 Normalization and Fade Modules

The PDW parameters are extracted from the output of the FIFO and the RF and PW are passed to the normalization module (*Norm*) and the TOA is passed to the fade cycle calculation module (*Fade*). Both *Norm* and *Fade* modules require division, which is an expensive operation in an FPGA. In situations when the divisor is a power of two the division operation can be replaced by a simple bit-shift and can often be performed in a single clock cycle. However, for this case, the divisors are variable and not powers of two. Because both the normalization and fade calculation are independent of one another we can perform them in parallel in turn removing some of the latency and decreasing the total computation time.

Normalization

To decrease latency even further, the normalization of RF and PW are parallelized using two 1-D normalization modules, as shown in Figure 4.4. Both RF and PW are normalized using the same *Norm_1D* module by supplying each with unique normalization parameters (not shown in Figure 4.4).

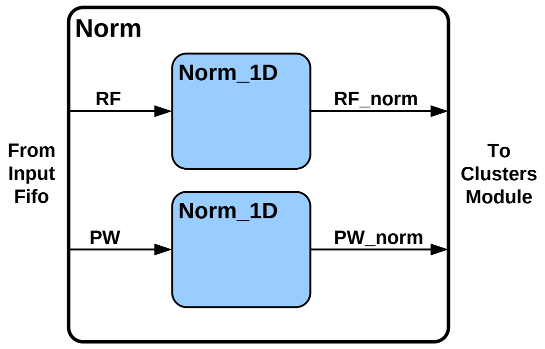


Figure 4.4: 2-D normalization module

Recall from Section 3.1 the formula for normalization.

$$norm = \left[\frac{(input - min)}{(max - min)}\right] * 65535 \tag{4.1}$$

Since we are normalizing to 16-bits we require a multiplication by 65535. With an insignificant loss of precision we replace the multiplication by 65535 with a 16-bit right shift, equivalent to a multiplication by 65536. This eliminates the need for a multiplier freeing up FPGA resources and decreasing latency.

Fade

The *Fade* module is responsible for calculating the whole number of fade cycles since the previous input. The calculated number is passed to the *Clusters* module and subtracted from the weight of the cluster centers, also known as the fade operation. If a whole number of cycles has not passed, the module outputs zero and retains the residual for the next calculation. We considered using a free-running counter but decided against it. Instead, by having the counter updated by each TOA value, we can provide more accuracy in cases when several PDWs back up in the FIFO preventing their immediate processing.

4.3.2 Clusters Module

Once RF and PW have been normalized and the number of fade cycles calculated, the results are passed to the *Clusters* module. As shown in Figure 4.3 the *Clusters* module can be broken down into three major components. First we have an array of n *Cluster_center* modules, $C_0 \dots C_{n-1}$. Next, we have three binary trees, *Min_d*, *Min_w*, and *Assigned*. Finally, we have the *Cluster Assignment* module.

Cluster Center Module

For an n-cluster implementation we require the instantiation of n independent *Cluster_center* modules. Using Verilog generate statements our code supports automatic compile-time instantiation and interconnect of cluster centers. This allows the implemented number of clusters to be varied by specifying a single parameter and without re-writing any source code. The *Cluster_center* modules are responsible for measuring the distance to inputs and calculating and updating their center coordinates. Figure 4.5 shows the major modules within each *cluster_center*.

The *Cluster_center* module receives normalized RF and PW values from the *Norm* module and the number of fade cycles from the *Fade* module. The RF ad PW are sent to the *Dist_meas* module which calculates the distance from the input to the

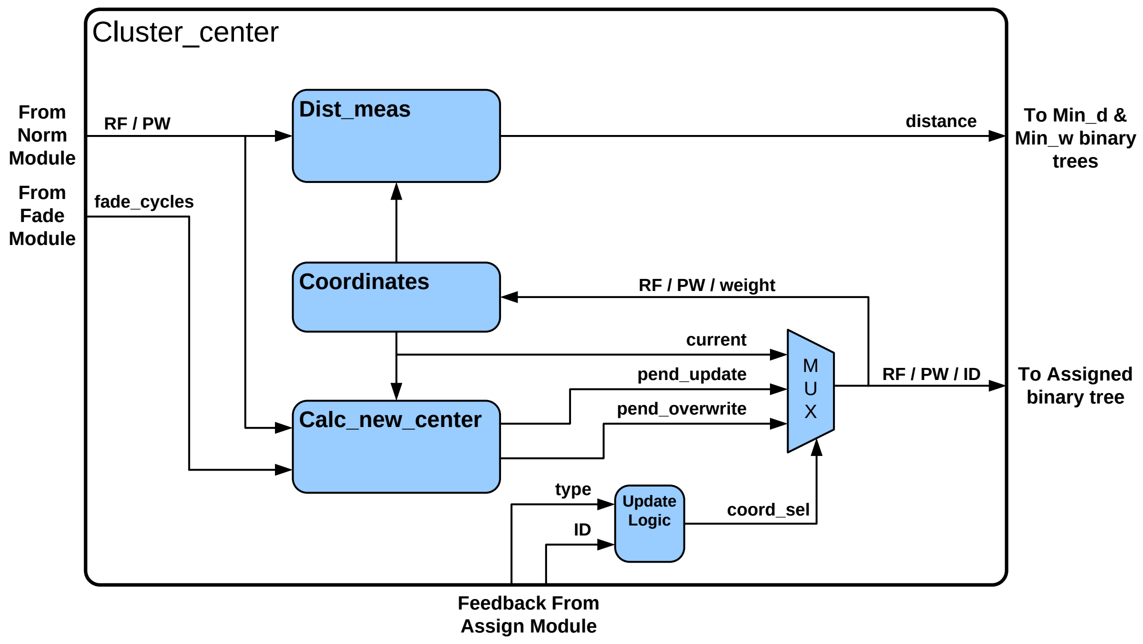


Figure 4.5: Cluster center block diagram

current cluster center. Simultaneously the *Calc_new_center* module receives the RF, PW, and number of fade cycles and performs two calculations. First, it decreases the current cluster weight based on the number of fade cycles. Next, it determines the pending cluster coordinates and weight for two cases. The first is if the current input is added to the cluster (*pend_update*), and the second is if the cluster needs to be overwritten (*pend_overwrite*) based on the requirement to start a new cluster.

The output of the *Dist_meas* module from each cluster center is sent to the *Min_d* and *Min_w* binary trees which determine the nearest and lightest clusters. Using the nearest and lightest information the *Cluster_assignment* module decides to which cluster the current input should be assigned, and whether the cluster should be updated or overwritten. This decision is fed back to each *Cluster_center* module and used as multiplexer control to select the *current*, *pend_update*, or *pend_overwrite* coordinates and weight.

Finally, each *Cluster_center* updates (if required) and outputs its coordinates to the *Assigned* binary tree. This tree is responsible for determining the coordinates and ID of the "winning cluster". In addition to updating coordinates, every time an input is assigned to a particular cluster the time difference of arrival between it and the previous cluster is calculated and stored. This value provides a first order estimate of the emitter PRI and can aid in further TOA processing.

Minimum distance, Minimum Weight, and Assigned Binary Trees

Min_d and *Min_w* are responsible for finding the nearest and lightest clusters, respectively. The *Assigned* tree selects the appropriate cluster coordinates and ID after an assignment has been made by the *Cluster Assignment* module. These three blocks are constructed as binary trees of two-input comparators (*Min_d* and *Min_w*) and two-input multiplexers(*Assigned*), rather than a single design. By doing so. the trees can be automatically generated at compile time based on the number of clusters. This auto-generation not only instantiates the correct number of sub-blocks but also performs the required interconnect among the sub-blocks as well as to the adjacent modules. Another example of modularity and design-reuse is the parameterization of the the bit widths of the two-input multiplexer module. Doing so allows use of the same module in both the (*Min_d* and *Min_w*) trees.

Cluster Assignment

As discussed in Section 3.3, an input can either be assigned to an active cluster or used to start a new cluster. If the distance from the nearest cluster is within a specified threshold the *Cluster_assignment* module outputs the ID of the corresponding cluster which is fed back to all *Cluster_center* modules. The "winning" cluster center then updates its coordinates with *pend_update*. If the distance from the nearest cluster exceeds the threshold, a new cluster is started by overwriting the lightest cluster. The ID of the lightest cluster is then fed back to all *Cluster_center* modules and the "winning" *Cluster_center* then updates its coordinates with *pend_overwrite*. All other cluster centers retain their *current* coordinates.

To provide real-time configurability the threshold value is programmable through a software accessible register. This allows threshold values to be easily selected based on particular scenarios and expected emitter characteristics. Future work might investigate the implementation of adaptive thresholding techniques which can be set based on the statistics of the input data.

4.3.3 Normalization Undo

The final step in the *ICED* algorithm is to output the assigned cluster ID and coordinates. Before they are output, however, the coordinates are converted from normalized values back to their original native units using the following equation

$$native = \frac{(max - min) * norm}{65535} \tag{4.2}$$

Similar to the optimization used in the *Norm* module, we can avoid a costly division by using a right shift operation. This results in a negligible error but saves FPGA resources and decreases processing latency. As a comparison, we created a 32-bit by 16-bit divider using the Xilinx Divider Generator 3.0 [33]. A 32-bit numerator is required based on the result of the multiplication of $(max - min)$ and $norm$, both 16-bits. The 16-bit denominator is a result of the maximum normalized value of 65535. The latency of the resulting divider was 25 clock cycles, 24 clock cycles greater than a single clock cycle shift operation. For typical clock frequencies of 100 Mhz and 200 Mhz this is a savings of 240 ns and 120 ns respectively.

4.4 Conclusions

This chapter presented three implementation goals: parallelism, modularity, and configurability. After providing a brief overview of the board we covered details of the FPGA design. We looked at the overall architecture as well as specifics of several major modules. In doing so we discussing the relevance of our design decisions to our goals stated at the onset. We briefly covered a few optimizations, but further discussion will be left for the next chapter which will include performance and area results for the FPGA.

Chapter 5

Results

This chapter presents the results of our FPGA implementation. We start by discussing our experimental setup and verification process. We then present our experimental results, focusing in particular on clustering quality, FPGA resource consumption, and FPGA performance.

5.1 Experimental Setup

Our design flow consists of design entry using Verilog HDL, synthesis and implementation using Xilinx ISE v11.2, and simulation using Mentor Graphics ModelSim SE 6.4c. Hardware testing was performed on the Innovative Integration X5-400M module running at 200MHz. Test data was generated using MATLAB which allowed full control over test scenarios and produced a more controlled experimental setup with fewer variables than in a real-world setup with measured data.

5.1.1 Test Data

In a typical EW system implementation, the pulse parameter measurements are performed in hardware and the resulting PDWs are then passed along to the deinterleaver. The measurement module and deinterleaver may be co-located or can be implemented on separate chips or even separate boards. Although we do not have a pulse measurement module available in hardware, we do have access to a high-fidelity MATLAB Simulink model. Since we are concerned with evaluating the deinterleaver independently, an accurate pulse measurement model is more than adequate.

The model represents a channelized architecture consisting of six 32 MHz channels. Each channel provides a 13-bit RF and 16-bit PW measurement corresponding to resolutions of 4KHz and 16ns, respectively. Our algorithm normalized these values to 16-bit representations based on the range of expected inputs, and for simplicity we simulated emitters in a single 32-MHz channel.

The generation of PDWs is composed of several steps and is depicted in Figure 5.1. First, the signal environment is defined by specifying the characteristics of the desired emitters. These characteristics include RF, PW, PRI, and time scheduling. This environment definition is then used to render simulated analog signals for each emitter. The individual signals are then interleaved in order to simulate their receipt at a single receiver. Finally, the TOA, RF, and PW measurements are made for each detected pulse and the PDWs are then stored to a file.

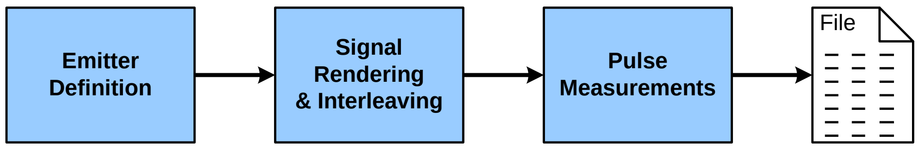


Figure 5.1: PDW Generation Sequence

The PDW file is transferred to the hardware using a C program which accesses the FPGA over the X5-400M's PCI interface. In a real system a hardware module would be responsible for supplying PDW inputs to the FIFO, but doing so via software allows us to independently measure the effectiveness of the *ICED* module. In addition to supplying the PDW inputs, the software allows real-time configuration and status monitoring of the algorithm details. Parameters such as cluster threshold, maximum cluster weight, and fade cycle length can be set from the host and cluster characteristics such as coordinates and weights can be read back to the host.

5.1.2 Verification

Early in the design process we created a software model, written in C, to aid in the exploration of algorithm parameters and evaluation of design tradeoffs. When the design was transitioned to the FPGA this model served as our "golden" reference allowing us to verify the output of the FPGA with a known good result. Therefore initial verification focused solely on clustering accuracy of the software model.

We started by using simple datasets consisting of several interleaved emitters with no overlapping pulses. We would expect such datasets to result in a cluster being created for each emitter and the weight of the clusters to be a function of both the emitter PRF and the fade cycle length. We logged the cluster assignments for each input as well as the coordinates and weights of all clusters after each input was processed. This gave us periodic snapshots of the cluster details allowing us to keep track of position and growth/fade characteristics. Verification of simple, non-overlapping cases was done by inspecting the log files by hand and verifying that the cluster parameters correlated well with the known emitter characteristics.

The next step was to expand verification to include scenarios with overlapping pulses. In this case some PDWs describe combined pulses and may not correlate well to a particular emitter in the environment. In this scenario we expect the number of clusters to exceed the number of known emitters. The extra clusters will correspond to outliers generated by overlapped pulses and should have relatively low weights based on the frequency of overlap. Furthermore, these outlying clusters may come and go in cases when they are faded faster than they are incremented since new inputs are not being added. Verifying such scenarios by hand is both tedious and difficult so we decided to plot the results and inspect them visually.

Since we are logging the cluster assignments, coordinates, and weights after each input we combined sequential plots to create movies depicting the evolution of the clustering process. We color coded PDWs based on cluster assignment and plot only those PDWs currently contributing to the cluster's weight. Finally, we indicated the location of each cluster by overlaying an X on the plot for each center. Using the logged weights we were able to construct histograms showing the evolution of cluster weights over time. For incremental clustering algorithms, the intermediate results are no less important than the results at the end of a data set. These movies were the ideal tool for viewing the clusters during the process and proved invaluable in verification of more complicated scenarios.

By logging identical data in both the software model and actual FPGA implementation, the same verification tools can be shared enabling an easy comparison of results both qualitatively from visual inspection and quantitatively through log files of known "golden" results.

5.2 Results

This section presents the results of our experiments and implementation. First we evaluate the clustering quality for several different input scenarios. Next, we look at the FPGA resources required for our design. Finally, we discuss the performance of the FPGA implementation and compare it to a software approach.

5.2.1 Clustering Quality

We examine the clustering quality using several specific datasets representing typical emitter scenarios. But first, let's look at a generic dataset consisting of a single emitter with a Gaussian distribution of points. Clusters formed from actual emitter PDWs may not result in Gaussian distributions, but studying the Gaussian case allows us to evaluate more general performance characteristics of the *ICED* algorithm by investigating the effects of varying cluster threshold and fade rate.

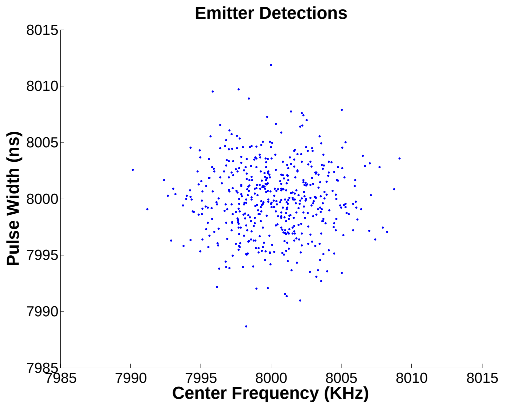
Evaluating clustering quality for an incremental algorithm is not straightforward. With a non-incremental algorithm one can compare the final clustering to the known input set and calculate the percentage of correct assignments. In an incremental case, however, the performance must be continuously evaluated because the clustering at any given point is just as important as the result at the end of a dataset. Because the algorithm does not revisit data and can only assign inputs once, minor errors can be tolerated along the way as long as the algorithm performs well on average.

For the generic case we represented a single emitter with a Gaussian distribution of points with a given mean, μ, and variance, σ^2. With a single emitter we would expect a perfect clustering to produce one cluster whose coordinates are the means of each dimension (μ_{rf}, μ_{pw}) of the dataset. The coordinates may vary at first but should stabilize over time as the cluster weight increases. We consider the cluster with the greatest weight to represent the true emitter and any other clusters to represent incorrect assignments. Using this scheme we can calculate the percentage of correct assignments, and the maximum number of predicted emitters at any given point (number of active clusters). We evaluate performance based on these two measurements as well as a visual inspection of the clustering over time.

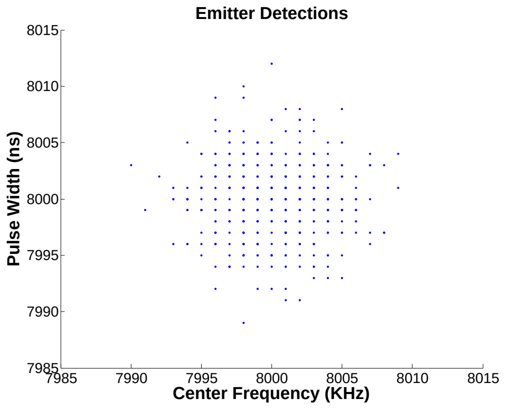
Single Emitter - Gaussian Distribution

This case uses a single emitter whose RF and PW measurements have mean and variance values of 8000 and 10, respectively, and whose PRI is 2000. Although the particular values of these input parameters are not important, we are interested in their relation to the selection of algorithm parameters such as maximum cluster threshold and fade cycle length. Figure 5.2 shows a scatter plot of PW vs. RF for 500 sample points. The plot on the left shows the raw values and the plot on the right shows the values quantized to integers to conform with the algorithm input requirements.

Given this dataset we can now look at the clustering results using the Manhattan distance metric and with varying threshold and fade values. We considered fade values ranging from 2000 (1x PRI) to 10000 (5x PRI), and thresholds ranging from 7 (2x standard deviation) to 15 (1.5x variance). We found a general trend between the performance and the relationship of the threshold and fade values. As the threshold decreased more points fell outside the main cluster and in turn created more clusters. According to our evaluation parameters, members assigned to these additional clusters, which we'll call outliers, are classified as incorrect assignments. For short



(a) Original

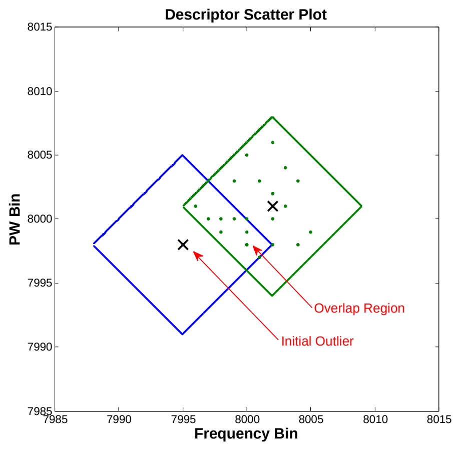


(b) Quantized

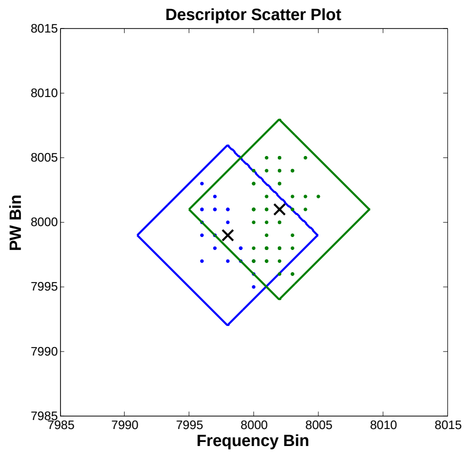
Figure 5.2: Gaussian Distribution ($\mu = 8000, \sigma^2 = 10$)

fade values the outliers weights are kept low and these clusters are therefore quickly deleted. Short fade values close to the PRI, however, resulted in increased jitter in the main cluster as the cluster weights were rapidly faded and the center could not accumulate enough points to stabilize. As the fade value increased, the outliers existed for a longer period and therefore had more opportunity to “steal” inputs from the main cluster. This happens because, although the outlying cluster is created with a center outside of the radius of the main cluster, the resulting radius of the outlier can overlap the main. Therefore, data points that fall in the overlap region, though they are contained within the main cluster boundary, can actually be closer to the outlier’s center.

An example of an outlying cluster stealing points from the main cluster is shown in Figure 5.3. The cluster boundaries are in the shape of diamonds based on the Manhattan distance metric. Notice that in Figure 5.3(a) the introduction of a point outside of the main cluster threshold has forced the creation of a new cluster. Because the new point is within twice the threshold of the main cluster, the two clusters will partially overlap. Future points in the overlap region that are closer to the new cluster can enable the new cluster to migrate towards the main cluster, as shown in Figure 5.3(b). Eventually we might see a bi-modal distribution of points between the two clusters even though there is only a single emitter. This is an unwanted phenomenon and can be remedied by increasing the threshold, decreasing the fade cycle length, or both.



(a) Creation of outlying cluster ($t = 0$)



(b) Migration towards main ($t > 0$)

Figure 5.3: Outlying Cluster

Let's look at the results for three specific sets of algorithm parameters. For each case three different input datasets with the same Gaussian distribution parameters were tested. The results were averaged and are shown in Table 5.1.

Table 5.1: Test Cases for Gaussian Distribution

Trial	Threshold	Fade Length	Correct Assignment	Max Clusters
A	7	10000	46%	5
B	7	5000	62%	3.7
C	10	5000	90%	3

In Trial A, which used a small threshold and a long fade cycle, outliers quickly developed and often resulted in two or more clusters being assigned the majority of the inputs. In trial B we kept the same threshold but decreased the fade length.

This change resulted in outliers being killed off faster and fewer inputs being stolen from the main cluster. Finally, in trial C, we increased the threshold in an effort to limit the number of outliers. This worked well, and even though there were at most 2 outliers at any time, their weights remained low due to the short fade length, and were therefore quickly deleted. In the end, this resulted in 90% of the inputs being assigned to the main cluster. With respect to the properties of the input, this case used a threshold equal to the emitter PDW variance and a fade length equal to 2.5x the emitter PRI. Even better performance can be achieved by further increasing the threshold at the expense of ability to distinguish between neighboring emitters.

Building on what was learned from the Gaussian test, we move on to cases with more realistic emitter data. The algorithm parameters for these cases is based on experimentation and is selected with prior knowledge of the expected signal characteristics. In a realistic scenario one may have some prior information regarding the general characteristics of expected signals, ranges of probable RF, PW, and PRI, for example, but it is unlikely that this information will be as accurate as in a test setup. Although the *ICED* algorithm performs well in these cases, we motivate the need for adaptive configuration of algorithm parameters in order to ensure high performance without prior knowledge of signal characteristics or real-time user intervention.

Case I: Two Emitters, Overlapping Pulses

Here we use two emitters whose characteristics are shown in Table 5.2. The RF values are offsets from the center of the single 32MHz channel. The offset is the time at which the emitter starts producing pulses. Based on the PW and PRI of the emitters the interleaved pulses will occasionally overlap producing a PDW whose measurements do not correlate with either of the two emitters.

Table 5.2: Case I: Two Emitters With Overlapping Pulses

Case I	RF (MHz)	PW (μs)	PRI (μs)	Offset (μs)
Emitter 1	-5.0	1.0	20	1
Emitter 2	+5.0	0.8	6	500

The threshold value for this case was selected based on the accuracy of the measurement module. The accuracy was determined experimentally by providing PDWs with known RF and PW values and recording the measurement module outputs in terms of normalized values. The results showed maximum measurement errors of approximately 25 for both RF and PW. Using Manhattan distance these errors would result in a distance of 50. In order to provide a cushion we decided to use a threshold of 100, equal to twice the largest expected error. Considering the 16 ns time resolution of the measurement module the largest PRI in this case, 20 μs, corresponds to a value of 1250. To more aggressively combat outliers we chose a fade cycle length of 3000, slightly under the 2.5x used in the Gaussian case. With these parameters we can successfully identify the two unique emitters and notice an occasional emergence of a third emitter representing the overlapping pulses. The cluster corresponding to the overlapping PDW is quickly faded based on the short fade cycle compared to it's infrequent occurrence. Figure 5.4 shows a snapshot of the clustering. The top plot shows a total of three clusters, two representing the actual emitters and one representing the overlapped pulses. The lower left plot shows the current weight of each cluster. We can see that the weights of the actual emitters is much higher than the overlap weight. The right plot shows the estimated PRI based on the time difference of arrival of successive assignments to a particular cluster. Notice that there is no PRI estimate for the overlap cluster because it had recently been faded out and has not had the opportunity to measure the time difference of arrival between two assignments.

The quality of this case is quite good because we can confidently identify two emitters and one interferer. Basing the fade cycle parameter on the known PRI values improved our results over a completely blind case. Although we do have real-time control over parameters, we have not implemented any adaptive techniques. An excellent future addition to this algorithm would be the implementation of adaptive parameter thresholding based on the statistics of the current inputs. We leave further discussion of this topic to the next chapter.

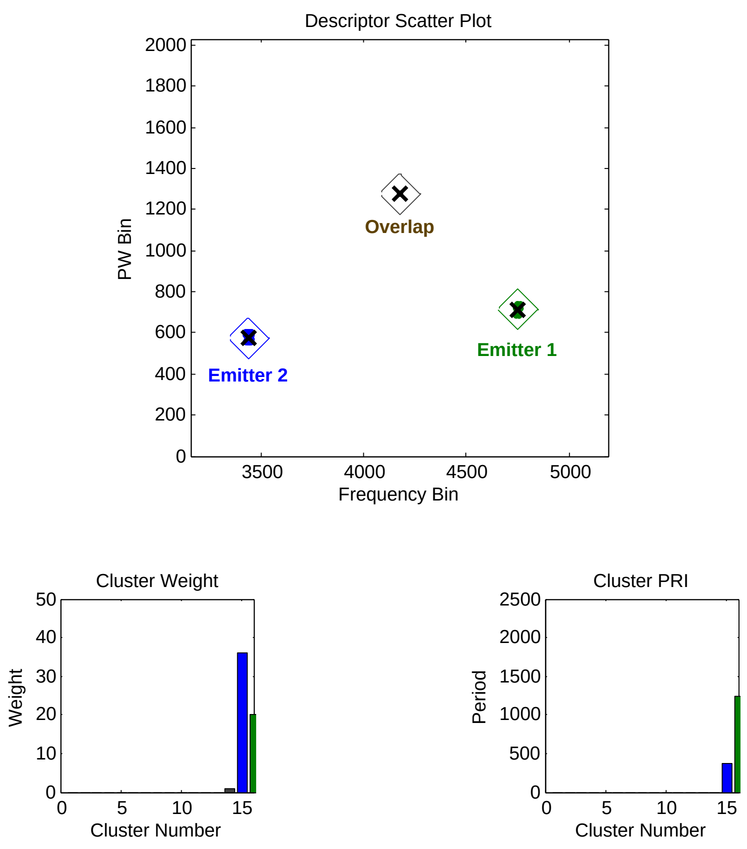


Figure 5.4: Case I Results

Case II: Single Frequency Hopper

This case will simulate a single emitter hopping over four frequencies each with the same PW and PRI. The specific characteristics of the hopper are shown in Table 5.3. The RF values are offsets from the center of the single 32 MHz channel. The duration is the amount of time at which the emitter will remain on a particular frequency. Since there is only a single emitter we have no overlap measurement interference but should expect to see up to four clusters at a time depending on the fade cycle length.

Using the findings of Case I we use a threshold of 100. Knowing that we will have no outliers to filter we extend the fade cycle length to 10000 so that we can maintain a cluster for each of the four hops. Figure 5.5 shows a snapshot of the clustering for

Table 5.3: Case II: Frequency Hopper

Case II	RF (MHz)	PW (μs)	PRI (μs)	Duration (ms)
Hop 1	+11.0	1.2	40	1
Hop 2	+6.0	1.2	40	1
Hop 3	+14.0	1.2	40	1
Hop 4	+1.0	1.2	40	1

this case. Due to the selected fade cycle length and the duration of each hop, after cycling through all hops we have maintained all four clusters. After moving to the next frequency the prior cluster starts to fade, but is not faded out completely before the hopper returns, enabling us to build on the prior statistics. We see that the PRI estimate for each cluster is nearly identical, as would be expected given that each hop has the same PRI value.

Keeping the clusters active until they are revisited is not necessarily a requirement and may or may not be possible for other scenarios based on the combination of hop duration and fade cycle length. Our algorithm identifies each hop as a separate emitter even though all pulses are being generated from a single source. A later stage in the EW system might group these hops and correlate them to an emitter with specific hop characteristics. The integration of such functionality is beyond the scope of our interleaver and is left for future work.

Case III: Fixed plus Hopper

We now investigate a more complicated scenario which is composed of a combination of the previous two cases. We start with the two emitters from Case I which we know will result in an additional cluster due to overlap. Additionally, as a third emitter, we introduce the single hopper from Case II. We see overlapping pulses between the hopper and the first two emitters individually, as well as cases with overlapped pulses from all three emitters. Due to the PW and PRI relationships of the emitters this scenario will stress our algorithm with a large number of false clusters representing the overlapping pulses.

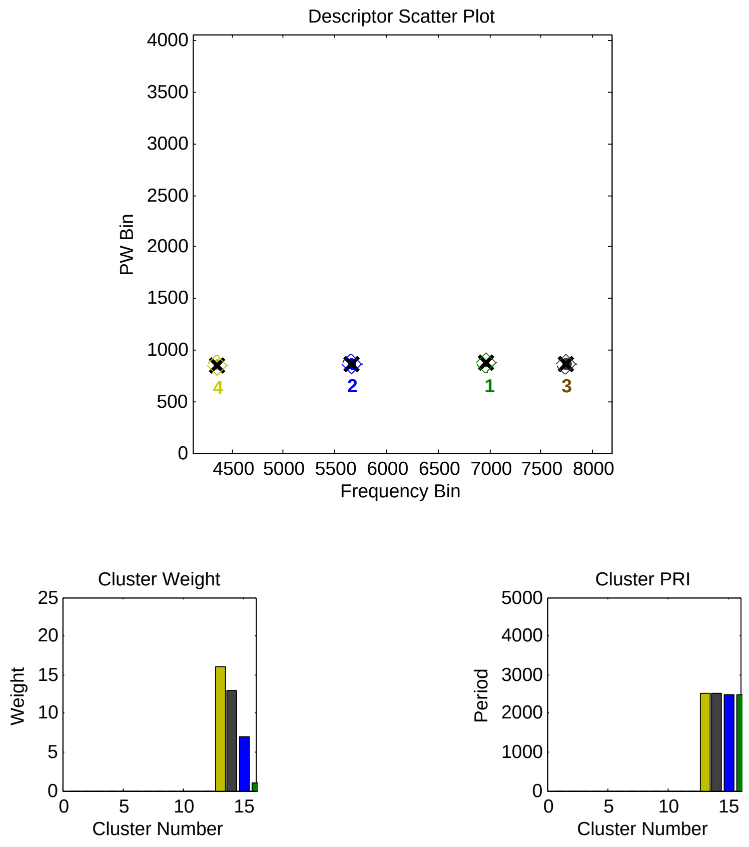
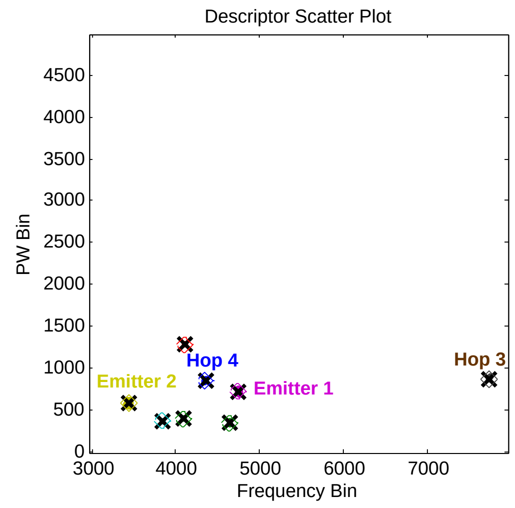
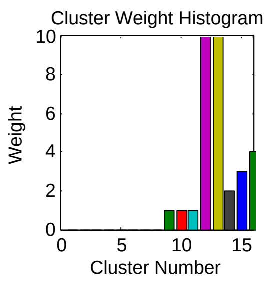


Figure 5.5: Case II Results

Based on the prior experiments we retain a threshold of 100. Through experimentation a fade cycle length of 7500 was chosen in order to stress the algorithm with a large number of outlying clusters. Figure 5.6 shows a snapshot of the clustering soon after the hopper has moved to its fourth frequency. The plot shows the two fixed emitters, Emitter 1 and Emitter 2, as well as the third and fourth locations of the hopper. The other four clusters represent false emitter locations. The cluster weights can help filter out the false locations from the true emitters. For example, in the plot a weight threshold of two would remove three of the four false emitters. Using a threshold as a filter, however, will increase the number of pulses required to be received before identifying a true emitter. Low thresholds can be used if the false clusters are faded frequently enough to keep their weights relatively low.





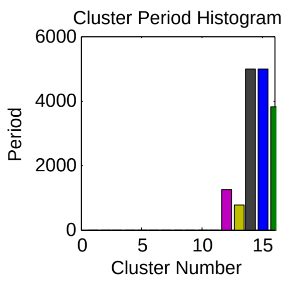


Figure 5.6: Case III Results

With this case we have shown moderate performance for an input data stream containing a significant number of measurements corresponding to overlapping pulses. We expect the ability to filter out some of the false emitters, however, not much can be done when the input is quite noisy. For example, if just as many PDWs are received for a false emitter as a true emitter it is nearly impossible to distinguish them. A possible upgrade would be to improve the measurement module enabling it to identify possible overlap cases and mark these PDWs with a flag. The deinterleaver can then filter out clusters formed with a large number of flagged PDWs.

5.2.2 FPGA Resource Consumption

This section will discuss the FPGA resources consumed by our algorithm. In addition to the *ICED* module some logic is required for general operation such as clock generation and the PCI Express interface controller. When evaluating resource consumption we lump these components into a single category which we consider required overhead. This allows us to concentrate specifically on *ICED* and its sub-modules.

Table 5.4 shows the consumption of resources for implementations of 2, 4, 8, and 16 cluster centers. It should be noted that the number of occupied slices is not always a good indication of resource availability because the implementation tools often spread the logic in order to improve timing. The utilization percentages for these implementations is plotted in Figure 5.7. Additionally, we have fit a line to each resource category and as expected see a linear relationship between consumption and the number of implemented clusters. By extending the lines back to zero clusters we get a good estimate of the utilization of the overhead logic (clock generation, PCI Express controller) and the clustering logic shared by all clusters (normalization, normalization undo, etc).

Table 5.4: FPGA Utilization

	Slices	Slice Registers	LUTs	BRAM	DSP48E
Available	14720	58880	58880	244	640
2 Clusters	5989 (41%)	15121 (26%)	11719 (20%)	17 (7%)	80 (13%)
4 Clusters	6950 (47%)	18834 (32%)	15166 (26%)	21 (9%)	120 (19%)
8 Clusters	9221 (63%)	26267 (45%)	22087 (38%)	29 (12%)	200 (31%)
16 Clusters	12972 (88%)	41146 (70%)	37500 (64%)	44 (18%)	360 (56%)

The limiting factor in increasing the number of clusters is the number of slice registers, but the number of LUTs and DSP48Es are not far behind. Projecting past 16 clusters we note that the theoretical limit would be 25 clusters based on near 100% utilization of LUTs. Increasing the number of clusters requires the addition of *cluster_center* modules (utilization for a single center is broken down in Table 5.5), as well as the expansion of the binary trees used to find the nearest, lightest, and

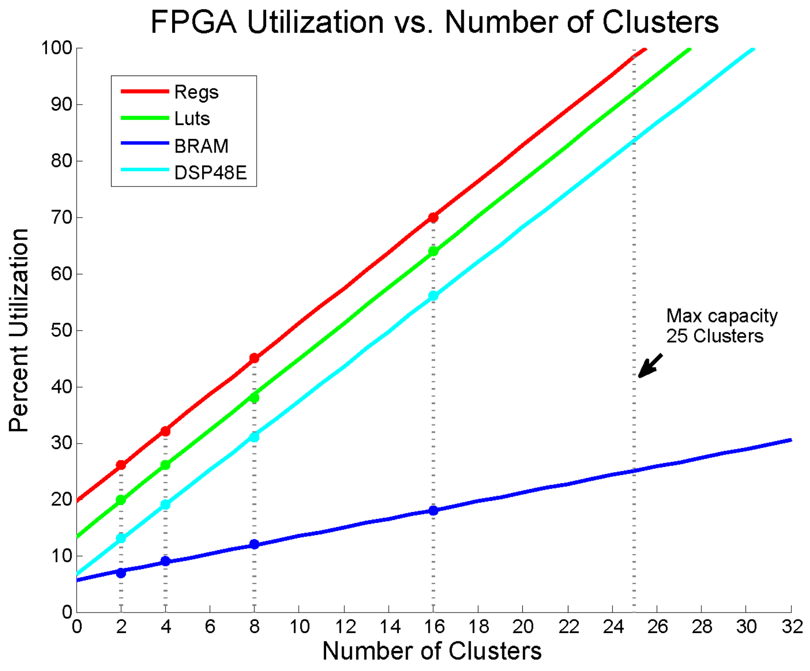


Figure 5.7: FPGA Utilization

assigned clusters. Based on the current design there are not enough resources to double the number of clusters from 16 to 32. Although there is not a requirement to increase clusters in powers of two this method is convenient based on the construction of the binary trees.

Table 5.5: Cluster_center Module Utilization

	Slices	Slice Registers	LUTs	BRAM	DSP48E
Available	14720	58880	58880	244	640
Cluster_center	936 (6%)	1791 (3%)	1545 (3%)	1 (<1%)	20 (3%)

5.2.3 Performance

Now that we have evaluated the accuracy of the algorithm and the resource consumption, let's look at the performance of the FPGA implementation. We do so by measuring the overall processing latency of the 16 cluster implementation.

In Section 2.2.3 we investigated various emitter scenarios in order to define the required maximum allowable processing latency. We found that to handle moderately stressing cases we could tolerate a 1μs latency but to keep up with the most stressing case the latency must be no greater than 500 ns. The clock frequency used by the *ICED* module is 200 MHz, which has a period of 5 ns and would require a latency no greater than 100 clock cycles.

In order to gain better insight into the individual latencies we have created a time line, shown in Figure 5.8, consisting of the major processing steps. Stacked boxes represent operations which are performed in parallel and therefore hide some latency over a serial implementation of such tasks. The 16 cluster implementation requires 84 clocks to complete, which corresponds to 420 ns, modestly exceeding even the most stressing scenario.

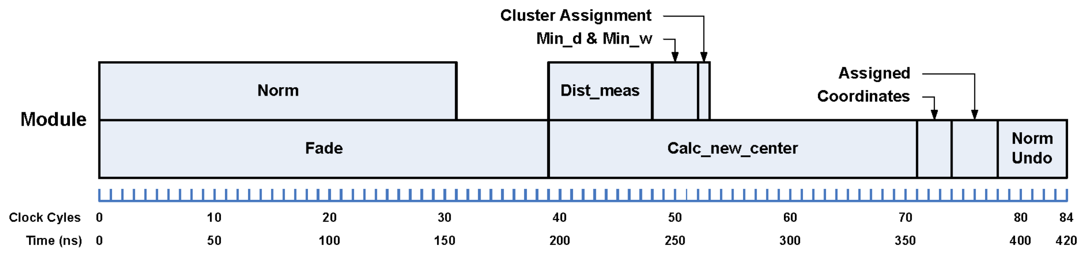


Figure 5.8: Module Processing Latencies

Seeing that we have met our worst case latency requirement let's now calculate the performance gain over a software implementation. As a software benchmark we used the C model developed for investigating algorithm tradeoffs. The model was configured with the same parameters as the implemented hardware. To provide a fair comparison between hardware and software implementations we did not include the time required by the software to load inputs or store results to and from disk.

The software was run on a single core of a 3.00 GHz Intel Xeon 5160 with 4GB of RAM. The required run time for 80,000 inputs was 13.0s, corresponding to 16.25μs per input. The resulting FPGA speedup over software is 39x. A typical implementation should show even greater speedup because an actual system would require that the inputs, be it discrete samples or further processed PDWs, be transferred from hardware to the local machine for software clustering. We have decided to ignore such added latencies in order to provide a more clear cut comparison of the clustering implementation itself.

5.3 Conclusions

This chapter presented the results of our hardware implementation including clustering quality, FPGA resource consumption, and speedup over a software implementation. We demonstrated effective clustering of a Gaussian distribution enabling us to compare results for varying algorithmic parameters. Next, we moved on to simulated emitter environments including cases with frequency hopping and overlapped pulses. We showed that for even heavy overlap cases, outlying clusters can be filtered out using an appropriate cluster weight threshold. We provided detailed resource consumption for 2, 4, 8, and 16 clusters, and estimated that the maximum number of implementable clusters would be 25 for this particular chip. Finally, in evaluating the hardware processing latency we showed that our implementation, which provides a 39x speedup over software, can keep up with even the most dense emitter environments.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

A major component of an EW system is the pulse deinterleaver. A deinterleaver separates combined streams of radar pulses into individual streams, each corresponding to a particular radar emitter. This process is crucial for effective EA and EP processing. One demanding requirement of deinterleaving is that it must be performed with low latency in order to handle dense emitter environments and provide timely identification to subsequent processing blocks.

This thesis presents *ICED*, an incremental clustering algorithm suited to deinterleave streams of PDWs. A major benefit of incremental clustering is that it is an "on-line" method. This means that it can operate on streaming data without having to revisit past inputs. This characteristic makes incremental clustering algorithms attractive to real-time problems such as pulse deinterleaving.

The algorithmic decisions for *ICED* are based on the particular characteristics of radar deinterleaving as well as the target FPGA implementation. Choosing a Manhattan distance metric decreases arithmetic complexity in the FPGA without sacrificing clustering performance. A fade mechanism places less importance on older data and provides greater agility and therefore more accurate results in cases with evolving data streams.

The final design is mapped to a Xilinx Virtex-5 SX95T FPGA. This implementation provides a maximum of 16 simultaneous clusters while occupying approximately 70% of the chip's resources. Running at 200 MHz, the processing latency for a single input is 420 ns resulting in a 39x speedup over a software implementation.

6.2 Future Work

Our implementation provides real-time configuration of algorithm parameters such as cluster boundaries and fade cycle length but does not implement any particular logic to adapt them on the fly. Adaptive thresholding based on statistics of current clusters can provide improved results especially for dynamic situations. Additionally, adaptive thresholding can lead to a better embedded solution requiring fewer assumptions about the working environment.

Using cluster fading, *ICED* attempts to filter clusters created by overlapping pulses that do not correlate to actual radar emitters. In cases where the number of PDWs corresponding to these false emitters is high it is very difficult to eliminate these clusters as their statistics closely resemble true emitters. A possible solution would be to more closely integrate the module performing the pulse measurements with the deinterleaver. Taking more frequency and amplitude measurements over the course of each pulse can lead to identification of overlapping cases. In simple cases, such as an overlap of only two pulses, it is possible to resolve the individual contributions [1]. If this is not possible it might be beneficial to at least label the corresponding PDW with a flag indicating a confidence level for the measurements. Special treatment of the PDWs with low confidence ratings may allow the deinterleaver to improve its performance.

Another opportunity for future work is to re-evaluate the chosen implementation architecture. Since processing latency easily exceeded our target goal, we might benefit from decreasing the amount of parallelism in the design. Sharing computation resources amongst several clusters can lead to decreased FPGA utilization and can allow for growth beyond 16 clusters or the integration of additional system blocks into the same FPGA. Such integration can provide savings in both size and cost.

6.3 Summary

We have shown promising results for an FPGA pulse deinterleaver based on an incremental clustering algorithm. Future implementations may improve performance with adaptive configuration of algorithm parameters.

Appendix A

List of Acronyms

AOA	Angle of Arrival
CDIF	Cumulative Difference
CW	Continuous Wave
EA	Electronic Attack
EP	Electronic Protection
ES	Electronic Support
EW	Electronic Warfare
FA	Fuzzy Adaptive Resonance Theory
FIM	Fuzzy Inference Modules
FMMC	Fuzzy Min-Max Clustering
FPGA	Field Programmable Gate Array
IAFC	Integrated Adaptive Fuzzy Clustering
ICED	Incremental Clustering of Evolving Data

MDL Minimum Description Length

PA Pulse Amplitude

PDW Pulse Descriptor Word

PRF Pulse Repetition Frequency

PRI Pulse Repetition Interval

PW Pulse Width

RF Radio Frequency or Frequency

SDIF Sequential Difference

SNR Signal to Noise Ratio

SOFM Self-Organizing Feature Mapping

SONN Self-Organizing Neural Networks

SVM Support Vector Machine

TAC Time Adaptive Clustering

TOA Time of Arrival

Bibliography

[1] J. Tsui, *Digital Techniques for Wideband Receivers*, 2nd ed. Raleigh, NC: SciTech Publishing, Inc, 2004.

[2] G. W. Stimson, *Introduction to Airborne Radar*, 2nd ed. Raleigh, NC: SciTech Publishing, 1988.

[3] S. Lin, M. Thompson, S. Davezac, and J. C. S. Jr., "Comparison of time of arrival vs. multiple parameter based radar pulse train deinterleavers," in *Proceedings of SPIE Vol. 6235. Signal Processing, Sensor Fusion, and Target Recognition XV*, 2006.

[4] D. Milojevic and B. Popovic, "Improved algorithm for the deinterleaving of radar pulses," in *Radar and Signal Processing, IEE Proceedings F*, vol. 139, Feb. 1992, pp. 98–104.

[5] Innovative Integration. (2010, Feb.) X5-400m user's manual. [Online]. Available: http://www.innovative-dsp.com/support/manuals/X5-400M.pdf

[6] USAF Air University. (2010, Mar.). [Online]. Available: http://www.au.af.mil/info-ops/ew.htm

[7] D. Adamy, *EW 101 A First Course in Electronic Warfare.* Norwood, MA: Artech House, 2001.

[8] A. E. Spezio, "Electronic warfare systems," *IEEE Transactions on Microwave Theory and Techniques*, vol. 50, pp. 633–644, Mar. 2002.

[9] D. Adamy, *EW 102 A Second Course in Electronic Warfare.* Norwood, MA: Artech House, 2004.

[10] A. Ataa and S. Abdullah, "Deinterleaving of radar signals and prf identification algorithms," vol. 1, Oct. 2007, pp. 340–347.

[11] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: A Review," in *ACM Computing Surveys (CSUR), v.31 n.3*, Sep. 1999, pp. 264–323.

[12] J. A. Hartigan, *Clustering Algorithms.* New York: John Wiley and Sons, 1975.

[13] Q. Song and N. Kasabov, "ECM, A Novel On-line, Evolving Clustering Method and its Applications," in *Proceedings of the Fifth Biannual Conference on Artificial Neural Networks and Expert Systems (ANNES2001)*, 2001, pp. 87–92.

[14] C. Gupta and R. Grossman, "GenIc: A Single Pass Generalized Incremental Algorithm for Clustering," in *Proceedings of the Fourth SIAM International Conference on Data Mining*, 2004, pp. 147–153.

[15] L. Reyzin, "Online Clustering of Linguistic Data," BSE Junior Independent Work, Princeton University, 2005.

[16] D. Barbara, "Requirements for Clustering Data Streams," in *SIGKDD Explorations*, vol. 3, 2002, pp. 23–27.

[17] H. Mardia, "New techniques for the deinterleaving of repetitive sequences," in *Radar and Signal Processing, IEE Proceedings F*, vol. 136, Aug. 1989, pp. 149–154.

[18] E. Granger, Y. Savaria, P. Lavoie, and M.-A. Cantin, "A comparison of self-organizing neural networks for fast clustering of radar pulses," in *Signal Processing*, vol. 64, 1998, pp. 249–269.

[19] J. Liu, J. P. Lee, L. Li, Z.-Q. Luo, and K. M. Wong, "Online clustering algorithms for radar emitter classification," in *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*, vol. 27, Aug. 2005, pp. 1185–1196.

[20] M. Leeser, P. Belanov, M. Estlick, M. Gokhale, J. Szymanski, and J. Theiler, "Applying reconfigurable hardware to the analysis of multispectral and hyperspectral imagery," in *Proceedings of SPIE*, vol. 4480, 2001, pp. 100–107.

[21] T. Saegusa and T. Maruyama, "An fpga implementation of k-means clustering for color images based on kd-tree," in *16th Annual Conference on Field Programmable Logic and Applications (FPL 2006)*, 2006, pp. 411–417.

[22] A. G. da S. Filho, A. C. Frery, C. C. de Araujo, H. Alice, J. Cerqueira, J. A. Loureiro, M. E. de Lima, M. das Gracas S. Oliveira, and M. M. Horta, "Hyperspectral images clustering on reconfigurable hardware using the k-means algorithm," in *Proceedings of the 16th Symposium on Integrated Circuits and Systems Design*, 2003.

[23] G. A. Covington, "Architecture for document clustering in reconfigurable hardware," Master's Thesis, Department of Computer Science & Engineering, Washington University in St. Louis, 2006.

[24] S. Sanchez-Solano, A. J. Cabrera, I. Baturone, F. J. Moreno-Velo, , and M. Brox, "Fpga implementation of embedded fuzzy controllers for robotic applications," in *IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS*, vol. 54, 2007.

[25] S. Arifin and P. Y. K. Cheung, "A novel fpga-based implementation of time adaptive clustering for logical story unit segmentation," in *Proceedings of the conference on Design, automation and test in Europe*, 2006, pp. 227–232.

[26] C.Baldanza, F.Bisi, M.Bruschi, I.D'Antone, S.Meneghini, M.Rizzi, and M. Zufa, "A cellular neural network for peak finding in high-energy physics," in *Proceedings of the 6th IEEE International Workshop on Cellular Neural Networks and Their Applications*, 2000, pp. 443–448.

[27] C.-F. J. nd Yu-Wei Tsao, "A type-2 self-organizing neural fuzzy system and its fpga implementation," in *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICSPART B: CYBERNETICS*, vol. 36, 2008, pp. 1537–1548.

[28] C. Kyrkou and T. Theocharides, "Scope: Towards a systolic array for svm object detection," in *IEEE EMBEDDED SYSTEMS LETTERS*, vol. 1, 2009, pp. 46–49.

[29] Y. Changbin and A. Wahab, "On the impelmentation of a fuzzy cmac," in *Proceedings of the Eighth Australian and New Zealand Intelligent Information Systems Conference (ANZIIS 2003)*, 2003, pp. 265–270.

[30] M. Estlick, M. Leeser, J. Theiler, and J. J. Szymanski, "Algorithmic transformations in the implementation of k-means clustering on reconfigurable hardware," in *Proceedings of the 2001 ACM/SIGDA Ninth International Symposium on Field Programmable Gate Arrays*, 2001, pp. 103–110.

[31] H. Hassan, "Joint deinterleaving/recognition of radar pulses," in *Proceedings of the International Conference on Radar*, Sep. 2003, pp. 177–181.

[32] Xilinx. (2010, Feb.) Virtex-5 family overview. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

[33] ——. (2010, Feb.) Divider generator v3.0. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/div_gends530.pdf