UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

A VERILOG IMPLEMENTATION OF WEIGHTED SUM MULTIOBJECTIVE OPTIMIZATION FOR USE IN COGNITIVE RADAR

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

JERED WESSON
Norman, Oklahoma

2021

A VERILOG IMPLEMENTATION OF WEIGHTED SUM MULTIOBJECTIVE OPTIMIZATION FOR USE IN COGNITIVE RADAR

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY THE COMMITTEE CONSISTING OF

Acknowledgments

I would like to give a special thanks to Dr. Cliff Fitzmorris whose help and guidance have been instrumental in my academic success. Thank you, Dr. Fitzmorris, for all the time and effort you have committed to being my thesis advisor. Additionally, I would like to thank Dr. Justin Metcalf and Dr. Ronald Barnes for taking the time to serve on my committee.

I would also like to thank my family for their extraordinary encouragement and support throughout my academic career.

Lastly, I would like to thank my friend and colleague Rylee Mattingly, along with all the members of our research group at the ARRC, for their advice and support throughout my time as part of the group.

Table of Contents

List of Tables

List of Figures

Abstract

As wireless technologies continue to become more ubiquitous in a variety of different applications, the electromagnetic spectrum will become increasingly saturated. Billions of wireless devices that utilize the spectrum are already in use, and it seems that the production and sale of these devices will not subside in the near future. As the number of these devices increases and the spectrum becomes more crowded, applications that apply the technique of spectrum sharing become even more important. The work in this thesis is specifically focused on cognitive radar applications, but the techniques and concepts used in this hardware implementation could be applied to other devices and applications that employ spectrum sharing.

The contribution of this work is a Verilog implementation of a weighted sum multiobjective optimization (WSMO) algorithm for use in spectrum sharing applications. In a cognitive radar, this algorithm is used to choose the optimal transmit band by balancing the competing objectives of low interference and large bandwidth. In typical cognitive radars, this analysis is performed by the host PC. Data transmission to the host PC and computation times on the CPU make this a non-realistic procedure for algorithms with high computational complexity if real-time operation is desired. Migrating this work to a field programmable gate array (FPGA) located in the radar itself allows the computation of WSMO in real time through parallelization of the algorithm and elimination of the unnecessary transmission of spectrum data to the host PC for processing.

Chapter 1

Introduction

Cognitive radars operate in a cycle of three phases referred to here as “sense,” “learn and decide,” and “adapt” [3]. The sense phase collects data about the surrounding spectrum to undergo analysis in the learn and decide phase. The learn and decide phase performs analysis on this spectrum data through some logical process or algorithm and returns a result that identifies which band the radar should use next based on the current conditions. Finally, the adapt phase uses the results of the learn and decide phase to switch the active transmission band to the newly determined optimal open band, ideally containing no signals from other users. Multiobjective optimization can be used to balance the two fundamentally conflicting objectives of bandwidth and interference in a cognitive radar. Multiobjective optimization has been the focus of a large body of research, and many different multiobjective optimization routines exist [4]. One of the most common methods of multiobjective optimization, weighted sum multiobjective optimization (WSMO), is used in this work [4] [5].

WSMO is used in the “learn and decide” portion of the cognitive radar process to analyze the sensed spectrum data and determine the optimal transmit band based on interference and presence of other users. The implementation of a WSMO algorithm on an FPGA provides multiple benefits for a cognitive radar application by helping overcome some significant technical challenges present in modern cognitive radars. Typically, data must be streamed from the radio hardware to the host PC for processing and analysis. Transfer rates to the host PC, as well as the computational complexity of the algorithm that must be computed sequentially on the CPU, cause a significant bottleneck in the operation of the radar. Development of a Verilog implementation of this algorithm to be run on the field programmable gate array (FPGA) of a cognitive radar would eliminate these bottlenecks and allow for more complex algorithms to be computed in real time by utilizing the parallelization that is available when using an FPGA. Transferring this workload to the FPGA also has the benefit of eliminating slow communication of data to the host PC. When using an FPGA in the radar itself, there will no longer be a need to transfer data out of the radio hardware, to the host PC, then back into the radio hardware post analysis. Data transfer to the host PC and sequential processing becomes more time consuming and impractical for real-time applications as the complexity of the spectrum analysis algorithm increases.

1.1 Thesis Outline

This thesis is divided into several chapters, outlined here. Chapter 2 contains general background information on the functionality of cognitive radars, as well as a brief introduction to the original version of the WSMO algorithm to be implemented on the FPGA. The algorithm description is followed by a section detailing the changes made to the algorithm to support a fast hardware implementation. Lastly, the development platform used for the hardware implementation is discussed.

In Chapter 3, the details of the Verilog Hardware Description Language (HDL) implementation are presented. The implementation description includes several topics corresponding to the calculation steps of the algorithm. In the first section, the methods to construct the objective functions are given. Next, the process of normalization of the objective functions using bit shifting is described. This section also includes a discussion of error introduced using this normalization method, as well as a proposed technique to mitigate the effects of this error. The next two sections of this chapter discuss the creation of the full weighted sum equation and the process to determine the maximum of this function, respectively. Finally, a timing diagram obtained using HDL simulation software is shown to verify the number of clock cycles necessary to calculate the final output of the algorithm.

Chapter 4 presents results of the algorithm for various frames of spectrum data and various weighting parameter changes. Results are presented for normal conditions, reduced signal-to-noise ratio (SNR) scenarios, and for changes in the weighting parameter.

Finally, Chapter 5 presents a conclusion and several topics for future work.

Chapter 2

Background

The algorithm used as the baseline for this work is described in [1]. The goal of that research was to reduce the computational complexity of the spectrum analysis to allow sequential processing methods, such as MATLAB or other programs running on the host CPU, to process the spectrum data in real time. This was accomplished by refining the spectrum data to reduce the input to an optimization routine such as WSMO.

The goal of the work in this thesis is to migrate the spectrum analysis workload entirely into the hardware of the radio using the onboard FPGA. Performing the work in the FPGA will prevent unnecessary, slow communication of spectrum data between the radio hardware and the host CPU without any pre-processing to reduce the input data to the WSMO algorithm. This work utilizes the high level of parallelization available on an FPGA to eliminate the need to reduce inputs to WSMO while also keeping the spectrum analysis workload directly inline with the radio hardware, greatly reducing the communication traffic to the host CPU which often acts as a bottleneck in cognitive radar applications.

2.1 Original Algorithm

Before detailing the final version of the WSMO algorithm that was implemented in hardware, a brief introduction of the source paper for this algorithm is necessary [1]. This section describes the WSMO algorithm exactly as it is presented in the source paper, while future sections in this chapter discuss the changes made to the algorithm to facilitate a fast, efficient FPGA implementation. Original figures and equations from the source material are shown here to provide a clear and accurate representation of the background information presented in that paper.

The radar technique used in the source material is referred to as spectrum sensing multiobjective optimization (SS-MO) [1]. A high level block diagram of the SS-MO technique is shown in Figure 2.1.

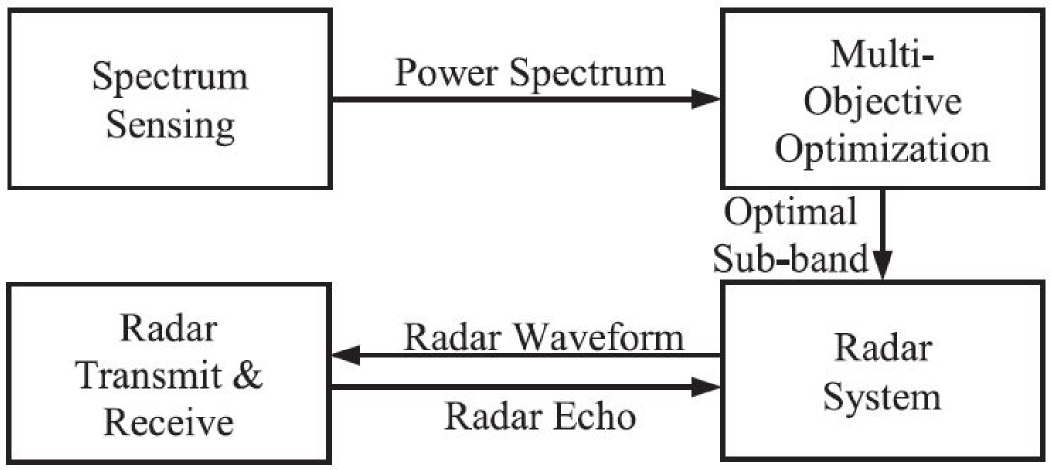


Figure 2.1: SS-MO technique for radar [1].

The SS-MO technique begins by sensing the spectrum to obtain an estimate of the interference surrounding the radar. This power spectrum estimate is the input to the WSMO algorithm. Using this power spectrum data, an interference estimate can be calculated for every possible bandwidth that could be selected by the algorithm.

This results in a triangular structure as shown in Figure 2.2, described by Equation 2.1.

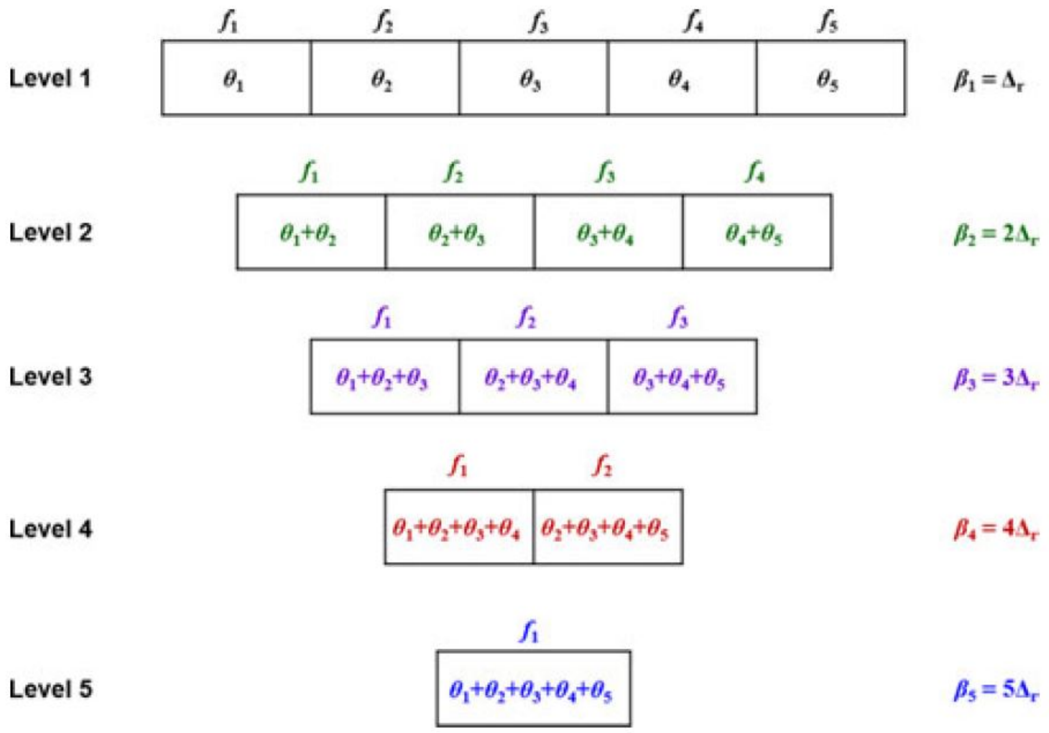

Level 1
f_1 f_2 f_3 f_4 f_5
θ_1 θ_2 θ_3 θ_4 θ_5
$\beta_1 = \Delta_r$
Level 2
f_1 f_2 f_3 f_4
$\theta_1+\theta_2$ $\theta_2+\theta_3$ $\theta_3+\theta_4$ $\theta_4+\theta_5$
$\beta_2 = 2\Delta_r$
Level 3
f_1 f_2 f_3
$\theta_1+\theta_2+\theta_3$ $\theta_2+\theta_3+\theta_4$ $\theta_3+\theta_4+\theta_5$
$\beta_3 = 3\Delta_r$
Level 4
f_1 f_2
$\theta_1+\theta_2+\theta_3+\theta_4$ $\theta_2+\theta_3+\theta_4+\theta_5$
$\beta_4 = 4\Delta_r$
Level 5
f_1
$\theta_1+\theta_2+\theta_3+\theta_4+\theta_5$
$\beta_5 = 5\Delta_r$

Figure 2.2: Structure of the interference estimate $\Gamma(\beta_i, f_j)$ containing 15 subband combinations available for processing. The subband size increases as the samples are merged together. The start frequency is depicted above each cell [1].

$$\Gamma(\beta_i, f_j) = \begin{cases} \theta_j, & i = 1, j = 1, ..., N \\ \Gamma(\beta_1, f_j) + \Gamma(\beta_1, f_{1+j}) & i = 2, j = 1, ..., N-1 \\ \Gamma(\beta_{i-1}, f_j) + \Gamma(\beta_1, f_{i+j-1}) & i = 3, ...N, j = 1, ..., N-i+1 \end{cases} \tag{2.1}$$

The top row of the interference estimate is simply the power spectrum estimate of each subband directly from the FFT. The bandwidth for each sample on that row is Δr, the change in frequency per spectrum sample, which is equivalent to the resolution of the FFT. This quantity is defined as the total bandwidth B divided by the number of FFT samples n, shown in Equation 2.2.

$$\Delta r = B/n \tag{2.2}$$

The elements of each subsequent row of the interference estimate is an addition of spectrum samples, making the bandwidth of each row equal to the number of added samples times the change in frequency per sample Δr, shown in Equation 2.3.

$$\beta_i = i\Delta r \tag{2.3}$$

This interference estimate and the radar receive power P_r are used to form the signal-to-interference ratio (SINR) equation. The radar receive power is defined in Equation 2.4.

$$P_r = P_t G^2 \lambda^2 \sigma N_P / [(4\pi)^3 R^4] \tag{2.4}$$

The parameters in Equation 2.4 are P_t, the radar peak transmit power; G, the transmit and receive antenna gain; λ, the wavelength of the carrier frequency; N_P, the number of pulses within a CPI; R, the arbitrary range to target; and σ, the target radar cross section [1]. The SINR equation, the first objective function for the WSMO algorithm, can now be defined in Equation 2.5 using Equations 2.4 and 2.1.

$$Z_1(\beta_i, f_j) = P_r \tau \beta_i / \Gamma(\beta_i, f_j) \tag{2.5}$$

The second objective function is the subband size, defined in Equation 2.6 [1].

$$Z_2(\beta) = \beta_i \tag{2.6}$$

The optimization goal is to maximize both SINR and bandwidth. As discussed in the source material, these two functions present a fundamental conflict. This conflict is intuitive, because it is expected that larger bandwidths will naturally contain more interference, thus reducing SINR. Therefore, a weighted sum is used to achieve multiobjective optimization to find an optimized solution considering both competing functions. The full WSMO equation is defined in Equation 2.9, where $\acute{Z}_1$ and $\acute{Z}_2$ are the normalized objective functions defined in Equations 2.7 and 2.8.

$$\acute{Z}_1(\beta_i, f_j) = Z_1(\beta_i, f_j)/max[Z_1(\beta_i, f_j)] \tag{2.7}$$

$$\acute{Z}_2(\beta_i) = Z_2(\beta_i)/max[Z_2(\beta_i)] \tag{2.8}$$

$$Z(\beta_i, f_j) = \alpha\acute{Z}_1(\beta_i, f_j) + (1-\alpha)\acute{Z}_2(\beta_i) \tag{2.9}$$

The final step in the algorithm is finding the maximum value of Z and returning the indices of the location of that element. These indices represent the final product of the algorithm, the subband width β_i^* and the starting frequency f_j^*. These two numbers are sufficient to completely and uniquely describe any subband of the full bandwidth, starting at any frequency with a granularity corresponding to the resolution of the FFT. The equation used in the source paper to obtain the optimized solution is given in Equation 2.10 [1].

$$\{\beta_i^*, f_j^*\} = arg\ max[Z(\beta_i, f_j)] \tag{2.10}$$

2.2 Changes made for hardware implementation

While many aspects of the WSMO calculation remain the same, some changes were made to the algorithm to support a parallel implementation of WSMO in an FPGA. The most significant change occurs in the first objective function, previously set as the SINR equation defined in Equation 2.5. In the hardware implementation, the first objective function is defined as solely the interference estimate described by Equation 2.1 and does not include information about the radar signal power. The reason this change was made is twofold. First, this change eliminates the costly division operation from the calculation of the objective function. Performing division for each element of the interference estimate would demand large amounts of logic elements and quickly result in excessive hardware requirements that are not feasible to maintain, especially for larger FFT lengths. Second, including information about the radar transmit and receive power would constrain the result of WSMO using specific radar parameters. In practical cognitive radar applications where this information is necessary, extra logic can be added into a separate stage of the algorithm without requiring division of every element. The inclusion of specific radar parameters could be accomplished by adding an additional stage in the algorithm to scale one or both objective functions in a manner that reflects how the use of pulse compression waveforms affects the performance of the radar. This topic is not deeply explored in this work in favor of achieving a fast implementation of the core functionality of the algorithm to select the widest open band for a general spectrum input without any specific radar constraints. The new objective function is defined in Equation 2.11.

$$Z_1(\beta_i, f_j) = \Gamma(\beta_i, f_j) \tag{2.11}$$

This change in the first objective function clearly also requires a change in the total weighted sum Z to maintain the relationship between the interference and the final sum. Recalling Equation 2.5, it can be seen that increasing the interference estimate in the denominator will cause that element of Z to be smaller. Naturally, the new weighted sum must follow the same behavior. The simplest solution to this is to subtract the interference estimate from the subband width function. This maintains the same relationship between interference and the total sum. The new weighted sum is defined in Equation 2.12 where $\acute{Z}_1$ and $\acute{Z}_2$ are the normalized objective functions.

$$Z(\beta_i, f_j) = (1-\alpha)\acute{Z}_2(\beta) - \alpha\acute{Z}_1(\beta_i, f_j) \tag{2.12}$$

Clearly this function is no longer bounded between 0 and 1 and has the potential for negative results. This does not present any issues, however, since the particular values of Z do not matter so long as the maximum value remains at the same index in Z.

In summary, two major changes were made to the original algorithm. Objective function normalization and the use of radar power in the calculations were removed because both operations require division. Objective function normalization is critical to the performance of the algorithm, so the division operation required to achieve normalization was replaced by bit shifting, an operation that can be easily synthesized in FPGA hardware.

2.3 Development Platform

All non-simulation development was done on an Ettus USRP X310 radio [6] [7] using the on-board Xilinx Kintex®7410T FPGA. The USRP Hardware Driver (UHD) is a software library maintained by Ettus Research that is used to control the radio hardware [8]. GNURadio is another open source software tool used to aid development by allowing a graphical definition of the program through its use of flowgraphs [9]. GNURadio flowgraphs are composed of data processing blocks that are interconnected as defined by the developer. To aid specifically in the FPGA development on the X310, the RF Network-on-Chip (RFNoC) framework is used [2]. FPGA development can be a daunting task, so RFNoC helps make this leap by standardizing the communication between blocks, greatly reducing the amount of work required by the developer to ensure proper data flow through the FPGA or data transfer compatibility with other blocks in the FPGA. For a more thorough description of the hardware, software, and FPGA framework and their capabilities, please reference [10]. A very brief introduction to the RFNoC datapath and the signals relevant to this design are presented in the next section.

2.4 RFNoC Data Communication

In RFNoC, data is transmitted using a stripped-down version of the standard Xilinx AXI bus interface called AXI-Stream (AXIS) [11]. As can be seen in Figure 2.3, there are several different communication datapaths within the RFNoC framework that are based on the AXIS communication protocol.

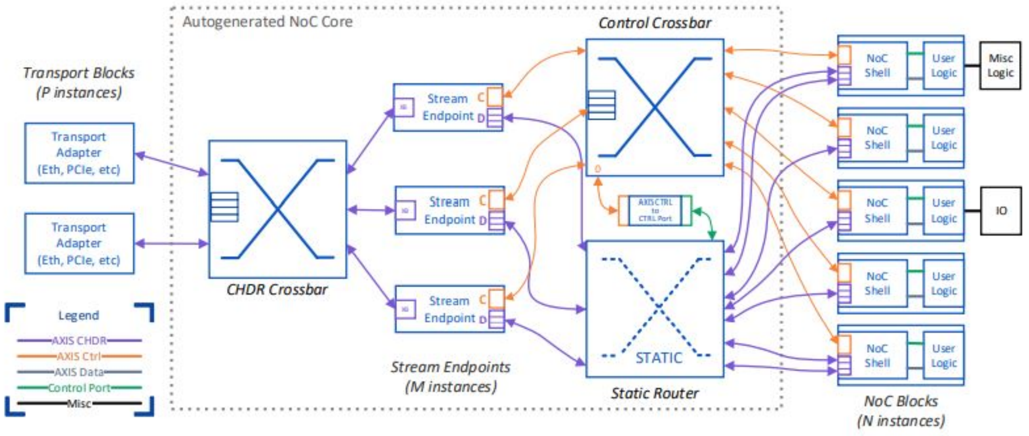


Figure 2.3: A simple RFNoC flowgraph showing internal signals and how they are routed in the RFNoC framework [2].

The RFNoC "NoC Core" is a predefined structure that handles communication between NoC Blocks using the AXIS Compressed Hierarchical Datagram for RFNoC (AXIS CHDR) bus and the control (AXIS Ctrl) bus. Data communicated over these busses is packetized, each containing some header information and a payload. While this format is appropriate for streaming data to and from NoC Blocks and the host PC, it would be a burdensome format for FPGA developers if the user logic in each NoC Block was required to parse these packets into more organized and easily accessible signals. Fortunately, the RFNoC framework provides the NoC Shell module to split the AXIS CHDR and AXIS Ctrl busses into their component parts as individual signals that are readily accessible to the FPGA developer. The relevant signals used in this work are shown in Table 2.1.

Name	Function
axis_data_clk	System clock for data signals
m_in_payload_tdata	Spectrum sample data (32-bits)
m_in_payload_tlast	Asserted when the data on the bus is the last sample in a packet
m_in_payload_tvalid	Asserted when the data on the bus is valid

Table 2.1: Signals used in the RFNoC AXIS interface.

This information about RFNoC is important for this work because the data communication protocol is packet based, so input data into the WSMO module will arrive at a rate of one sample per clock cycle. So, for an FFT length of 256 samples, it will take 256 clock cycles to obtain the entire frame of spectrum data. While partial calculations can be computed as the samples come in, this imposes a limitation on the speed of the algorithm given the number of cycles spent waiting for the spectrum data. This would seem to imply that the algorithm would have a calculation time of 256 + 9 = 265 cycles instead of the claimed 9 cycle computation time in Chapter 3, however, this is a misleading interpretation since the calculations cannot fully begin without first obtaining all the data. If it were possible to obtain all 256 samples at once, the algorithm completion time of 9 cycles would not be significantly increased, so the performance of the algorithm is maintained at an approximately 9 cycle delay from the time all of the data is received. Any calculations done while the data comes in serially over the data bus could be easily parallelized if the data were available in a single cycle. For more information about the RFNoC architecture, please refer to the RFNoC Specification and other RFNoC resources available online [2] [10].

Chapter 3

FPGA Implementation

This chapter describes a proposed implementation of the WSMO algorithm in Verilog. The input data to the algorithm consists of "frames" of frequency domain samples from an FFT block. The FFT data represents the most recent state of the frequency spectrum as seen by the spectrum sensing and signal detection components of the cognitive radar.

3.1 FFT Input Data Collection

The primary input to the WSMO block is a serialized stream of frequency domain samples from a standard FFT block. Since the rate of data arriving from the FFT block is limited to one sample per clock cycle, it is necessary to collect and store the samples until the entire FFT frame is collected for analysis. Sample storage is accomplished using two register files and a demultiplexer. This implementation serves two purposes. First, new data samples are constantly arriving over the input data bus, so data collection must never cease, even as calculations are being done on the previous frame of data. Second, using a demultiplexer to direct incoming samples to two separate register files prevents the need to copy previous frames of data from a single register file to a temporary storage location where analysis can be performed. Using two separate register files allows one file to hold the previous frame's collected data for analysis while the other register file is free to accept the incoming samples. Once the final sample in a frame of data has been collected, signified by the assertion of the m_in_payload_tlast signal from the RFNoC payload stream, the demultiplexer is switched to begin delivering the new samples of the next frame to the second register file. This implementation ensures that one of the register files is always available to collect new samples and eliminates the need for transmission of data to temporary storage for analysis.

3.2 Calculation of the objective functions

Once a full frame of data has been collected, the next step in the WSMO algorithm is to calculate both objective functions.

3.2.1 Interference Estimate Calculation

The first objective function to be optimized is the interference estimate, defined by Equation 2.11. The number of elements in the interference estimate objective function is defined as the n^{th} triangular number T_n [12], where n is the length of a single frame of FFT data in samples.

$$T_n = \binom{n+1}{2} = \frac{n(n+1)}{2} = O(n^2) \tag{3.1}$$

It follows from Equation 3.1 that increasing the FFT length will have a significant effect on the number of elements required for the interference estimate. It is also apparent from Equation 3.1 that the increase in elements is not linear with the increase in FFT length, but rather will increase on the order of n^2, requiring a proportionately large increase in FPGA logic elements. In addition to this, two parallel interference estimate register files must be maintained to facilitate constant data consumption by the WSMO block, doubling the required logic. Table 3.1 provides register requirements for various common FFT lengths.

FFT length	T_n	Registers for Interference Estimate
32	528	1,056
64	2,080	4,160
128	8,256	16,512
256	32,896	65,792
512	131,328	262,656
1024	524,800	1,049,600

Table 3.1: Number of registers required to maintain two parallel interference estimate register files.

Since the input data to the WSMO module is serialized, the FFT length will also affect the number of clock cycles required to calculate the interference estimate. As stated in [1], the computational complexity to form the interference estimate is $(n^2 - n)/2$ summations. Using sequential processing methods on the host CPU would require very large loops that increase in size on the order of n^2. On the FPGA, this calculation can be parallelized to reduce the execution time to O(n) as opposed to the sequential execution time of O(n^2). This is especially significant because the input data arrives to the WSMO block at a rate of one sample per cycle, making the time needed to collect a full frame of data also O(n). The faster computation time provided by an FPGA allows the interference estimate to be completely calculated at only a two-cycle delay from the collection of the last sample for a given frame of input data.

3.2.2 Subband Size Calculation

The second objective function to be optimized is the subband size, defined by Equation 3.2.

$$Z_2(\beta) = \beta_i \tag{3.2}$$

In Verilog, the subband function translates to a one dimensional n-length array containing every possible subband bandwidth that could be selected by the algorithm. Each element is a product of its own index i and the change in frequency per spectrum sample, Δr, as shown in Equation 2.3. Since Equation 3.2 does not rely on the spectrum data, the calculation can begin in the instant the total bandwidth and FFT length are known and initialized. The entire array of values can be calculated in as little as one cycle after initializing the system, depending on the latency of the hardware multipliers inferred by the synthesis tool and the frequency of the clock.

3.3 Interference Estimate and Subband Size Normalization

In the original WSMO algorithm [1], the objective functions Z_1 and Z_2 are normalized before being weighted and added together to form the full weighted sum function. This is accomplished by dividing each element of Z_1 and Z_2 by their maximum values, respectively. Hardware division is a costly operation in both computation time and FPGA resources, so other normalization options were explored.

To determine a faster, less resource intensive alternative to division, the goal of normalization was considered. Dividing each objective function by its maximum serves to constrain the elements of both functions to a common scale, namely the scale of real numbers from 0 to 1. Though 0 to 1 is perhaps the most commonly used normalization scale, it is not strictly necessary to use this scale to accomplish the goal of normalization for the purpose of this algorithm. The purpose of normalization in this instance is simply to scale the objective functions such that their weights are not inherently skewed by their relative difference in magnitude [13].

It is proposed that this task can be accomplished with a simple bit shift, a very fast and inexpensive operation to perform in hardware. The end goal of the shift operation is to produce two objective functions whose maximum magnitudes are as close to equal as can be achieved through multiplication by powers of two. This is accomplished by calculating the number of leading zeros in both function maximum values, then left shifting the function with the smaller maximum until the leading ones in both maximums occupy the same bit position. Clearly, this technique cannot always achieve accurate normalization results, and significant error may be introduced. The causes and magnitude of the error introduced by the shifting normalization technique is discussed in Section 3.3.3, and a solution is presented in Section 3.3.4.

3.3.1 Find the number of leading zeros

To determine the shift amount, the number of leading zeros for each function maximum must be calculated. Counting the number of leading zeros, also known as leading one detection, is a critical component of floating point operations. As such, there has been much research into designs that minimize latency and size of the circuitry required to perform the operation [14] [15] [16]. For the purposes of this work, a simpler approach will comfortably meet the requirements of the design. However, the modular structure of Verilog allows for easy replacement of the module functionality if a faster, smaller, more advanced design is required for other applications [17] [18].

Leading one detection can be accomplished in Verilog using a series of comparators and multiplexers with the results stored in registers in a single cycle. This functionality is implemented in the calculate leading zeros (CLZ) module. The module works by splitting the input into two halves, the most significant bits and the least significant bits. The most significant bits are then compared with zero. If the decimal value of the most significant bits is equal to zero, this implies that each individual bit is also equal to zero, so that number of bits must be added to the final output result of the module. It is expected that inputs to this module have a standard bit length corresponding to a power of two. If this is the case, the number of bits of the most and least significant halves will also be a power of two. Given these facts, it becomes apparent that incrementing the output by the number of bits of the most significant half reduces to simply storing the result of the comparator into the $log_2(k)$ bit position of the output, where k is the number of bits being compared. For example, consider the eight bit input 00000101 with most significant bits 0000. The comparison 0000 = 0 is true, therefore the comparator result is 1. This value is then stored in the $log_2(4) = 2$ bit position of the final output. Without computing the remaining tiers of comparison, the partial result would now be 0100 = 4, indicating that four bits have been recorded in the result, as expected. If the most significant bits are equal to zero, the least significant bits are then processed as the new input and the process begins again. In the case where the most significant bits are not equal to zero, this indicates that the input's leading one is in the most significant half of the bits, making it unnecessary to process the least significant half any further. As such, the most significant half is passed as the new input instead. This process is repeated until the input is only two bits wide and single bits are compared. The flowgraph in Figure 3.1 illustrates this process, generalized to calculate the number of leading zeros for any n-bit input given n is a power of two.

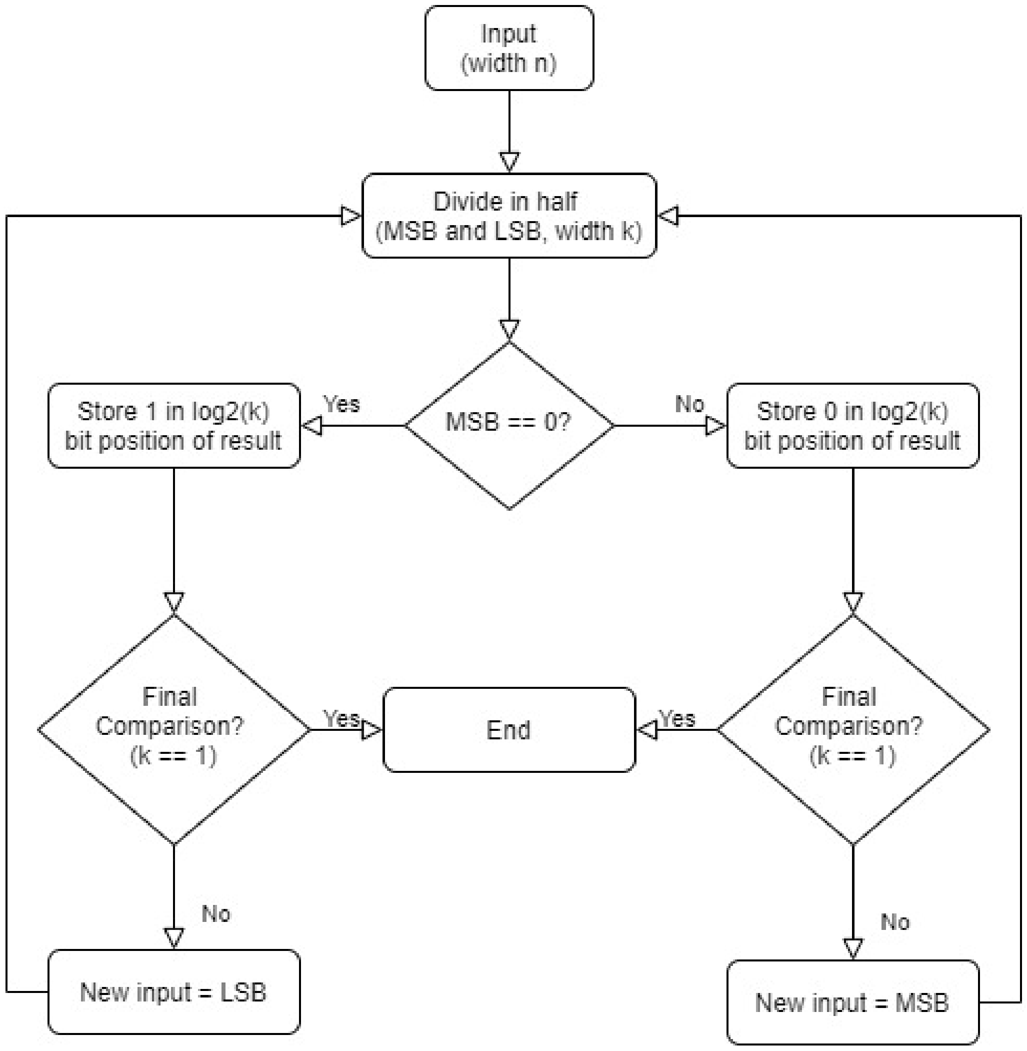

Input
(width n)
Divide in half
(MSB and LSB, width k)
MSB == 0?
Yes
No
Store 1 in log2(k) bit position of result
Store 0 in log2(k) bit position of result
Final Comparison? (k == 1)
Yes
End
Yes
Final Comparison? (k == 1)
No
No
New input = LSB
New input = MSB

Figure 3.1: Flowgraph of CLZ module behavior

The corresponding hardware for the first two comparison tiers of a 32-bit input are illustrated in Figure 3.2. This figure demonstrates why it is possible to complete this calculation in a single cycle. As seen in Figure 3.2, the bit output bit 3 does not depend on the registered bit output bit 4, and by extension does not depend on the clock for its value to be calculated. This pattern will continue as the inputs to be compared are split in two, concluding with the final comparison of a single bit. The results of this system are then simultaneously loaded into the result registers in the same clock cycle.

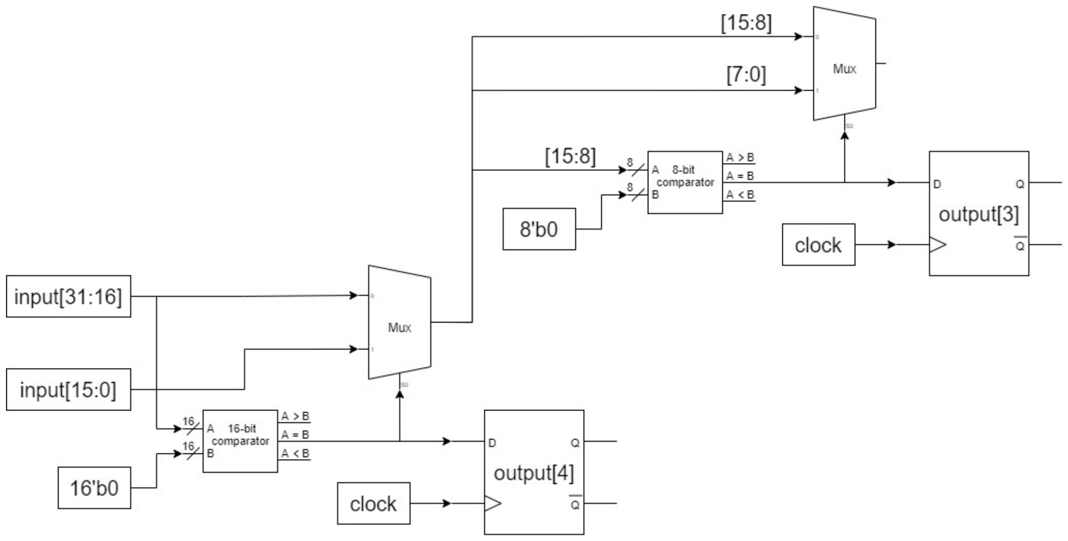
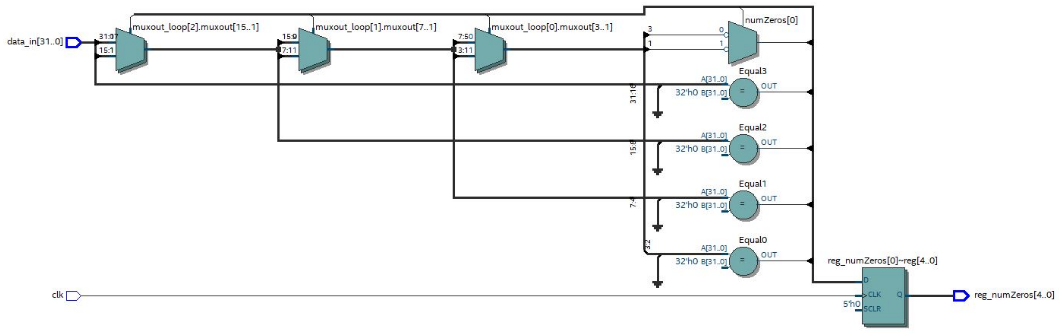


Figure 3.2: CLZ module component diagram

The hardware diagram created by the Quartus Prime synthesis tool shown in Figure 3.3 verifies that the Verilog code synthesizes to the hardware described in Figure 3.2.



data_in[31..0]
muxout_loop[2].muxout[15..1]
muxout_loop[1].muxout[7..1]
muxout_loop[0].muxout[3..1]
numZeros[0]
Equal3
Equal2
Equal1
Equal0
32'h0
reg_numZeros[0]~reg[4..0]
clk
5'h0
reg_numZeros[4..0]

Figure 3.3: CLZ module synthesized hardware diagram

3.3.2 Normalization by shifting

Once the number of leading zeros has been found for both function maximums, the shift amount to normalize the functions can be determined. To reiterate, the goal of this shift is to shift the smaller objective function such that the leading ones in the function maximums are aligned. This is equivalent to multiplying the smaller function by the power of two that will result in both function maximums being as close in magnitude as possible. This ensures that both functions are of similar weights before being added to the full weighted sum function.

Consider the following example with only 3 spectrum samples and a total bandwidth of 500 to illustrate this process. Assume the elements of two objective functions A and B are defined as in Table Figure 3.4.

Objective Function Elements				
Function A (Subband Size)		Function B (Interference Estimate)		
β_1	1	3	4	4
β_2	250	7	8	
β_3	500	15		

Figure 3.4: Normalization example elements

In Figure 3.4, it can be seen that the objective functions are not the same size. This arises from the fact that each row of the interference estimate corresponds to a single bandwidth β_i, and thus to a single element of the subband size. For illustration purposes, Figure 3.5 has been provided to clarify how each element from A will interact with the elements of B. When the weighted sum is performed, each element from A is summed with the element from B that occupies the same row and column in Figure 3.5.

Objective Function Elements					
Function A (Subband Size)			Function B (Interference Estimate)		
1	1	1	3	4	4
250	250		7	8	
500			15		

Figure 3.5: Normalization example elements

First, it will be useful to calculate the results using division as a reference. The normalized objective functions and remaining calculations are shown in Figure 3.6.

Normalized by Division					
(Function A)/500			(Function B)/15		
0.002	0.002	0.002	0.2	0.2667	0.2667
0.5	0.5		0.4667	0.5333	
1			1		

$$\Big\downarrow$$

$$Z = (1-\alpha)A - \alpha B$$

$$\Big\downarrow$$

Weighted Sum Results (Z) with α = 0.5		
-0.099	-0.13233	-0.13233
0.016667	-0.01667	
0		

Optimal Objective Function Elements					
Function A (Subband Size)			Function B (Interference Estimate)		
1	1	1	3	4	4
250	250		7	8	
500			15		

Figure 3.6: Calculation using division

The results of the WSMO algorithm using division to normalize the functions show that the optimal transmit bandwidth for these inputs has a subband width of 250 and a total interference of 7. Now the shifting approach can be compared. Figures 3.7 and 3.8 show the same calculation process as Figure 3.6, changing only the method of normalization.

Binary Representation					
Function A (Subband Size)			Function B (Interference Estimate)		
000000001	000000001	000000001	000000011	000000100	000000100
011111010	011111010		000000111	000001000	
111110100			000001111		

Figure 3.7: Normalization example elements

Normalized by Shifting					
Function A			Function B << 5 bits		
000000001	000000001	000000001	001100000	010000000	010000000
011111010	011111010		011100000	100000000	
111110100			111100000		

$$\Downarrow$$

Conversion to Decimal					
Function A (Subband Size)			Function B (Interference Estimate)		
1	1	1	96	128	128
250	250		224	256	
500			480		

$$\Downarrow$$

$$Z = (1 - \alpha)A - \alpha B$$

$$\Downarrow$$

Weighted Sum Results (Z) with α = 0.5		
-47.5	-63.5	-63.5
13	-3	
10		

Optimal Objective Function Elements					
Function A (Subband Size)			Function B (Interference Estimate)		
1	1	1	3	4	4
250	250		7	8	
500			15		

Figure 3.8: Calculation using shift

As shown in Figure 3.8, the same results have been achieved using normalization by shifting. Since the scale of the numbers in Z is irrelevant and only the maximum value must be considered, different methods of normalization can be used to produce the same final result.

In Verilog, the $<<$ operator is used to perform the shift. This operation can occur in one clock cycle because the synthesis tool is able to infer a barrel shifter as a series of multiplexers for each element in the objective function. A very fast clock or an excessively large number of bits to shift could cause the barrel shifter latency to exceed the clock period and cause multi-cycle shifts. However, assuming the clock rate of 200MHz used by the X310, shifts of the 32-bit objective function elements should fit comfortably within the 5 nanosecond clock period. According to [19], even operands of 128 bits are not estimated to exceed 2 nanoseconds of delay using any of the four barrel shifter implementations discussed in the paper. From this it can be concluded that each element of the entire objective function can be easily shifted in a single clock cycle.

3.3.3 Normalization error introduced by Shifting

To understand the cause and possible magnitude of the error introduced by using shifting rather than division, it will be useful to begin by revisiting the example from Section 3.3.2 with different data input for the interference estimate function, provided in Figure 3.9.

Objective Function Elements					
Function A (Subband Size)			Function B (Interference Estimate)		
1	1	1	1	2	3
250	250		3	5	
500			8		

Figure 3.9: Error causing example elements

Again, the WSMO result is first calculated using division as a reference, shown in Figure 3.10.

Normalized by Division					
(Function A)/500			(Function B)/8		
0.002	0.002	0.002	0.125	0.25	0.375
0.5	0.5		0.375	0.625	
1			1		

Weighted Sum Results (Z) with α = 0.5		
-0.0615	-0.124	-0.1865
0.0625	-0.0625	
0		

$$Z = (1 - \alpha)A - \alpha B$$

Optimal Objective Function Elements					
Function A (Subband Size)			Function B (Interference Estimate)		
1	1	1	3	4	4
250	250		7	8	
500			15		

Figure 3.10: Calculation using division

From this calculation it can be seen that the optimized bandwidth and interference are the same as the previous example in Section 3.3.2. However, this set of data will create a discrepancy between the division and shifting methods. Binary representation of the elements are provided in Figure 3.11, and calculations for normalization by shifting are shown in Figure 3.12.

Binary Representation					
Function A (Subband Size)			Function B (Interference Estimate)		
000000001	000000001	000000001	000000001	000000010	000000011
011111010	011111010		000000011	000000101	
111110100			000001000		

Figure 3.11: Error causing example elements

Normalized by Shifting					
Function A			Function B << 5 bits		
000000001	000000001	000000001	000100000	001000000	001100000
011111010	011111010		001100000	010100000	
111110100			100000000		

Conversion to Decimal					
Function A (Subband Size)			Function B (Interference Estimate)		
1	1	1	32	64	96
250	250		96	160	
500			256		

$$Z = (1 - \alpha)A - \alpha B$$

$$\Downarrow$$

Weighted Sum Results (Z) with α = 0.5		
-15.5	-31.5	-47.5
77	45	
122		

$$\Downarrow$$

Optimal Objective Function Elements					
Function A (Subband Size)			Function B (Interference Estimate)		
1	1	1	1	2	3
250	250		3	5	
500			8		

Figure 3.12: Calculation using shift

As shown in Figure 3.12, the results for the optimal bandwidth and interference are now in error with the reference results in Figure 3.10. In this example, it is apparent by the large difference in the maximum values of 500 and 256 that significant error has occurred. This error is caused by the difference in significant bits following the leading 1 of each maximum. The numbers 500 and 8 are perfect examples for the possible magnitude of error because they exhibit nearly the maximum error that can be introduced by using this method. Consider the following example of how shifting affects several different maximum values, shown in Figure 3.13.

Original		
	Decimal	Binary
Reference	500	111110100
Max 1	7	111
Max 2	8	1000
Max 3	15	1111
Max 4	16	10000

Shifted		
	Binary	Decimal
Reference	111110100	500
Max 1	111000000	448
Max 2	100000000	256
Max 3	111100000	480
Max 4	100000000	256

Figure 3.13: Example showing how pre-shift differences of just one decimal digit can result in large errors of the shifted maximum

It is interesting to note that a difference of even one decimal digit can cause nearly 50% difference in the amount of error introduced by the shift. Figure Figure 3.13 demonstrates that both best and worst case scenarios for a given maximum weight range can often be just one digit apart, with the transition occurring where a binary string of ones carries over to the next digit and the lower bits roll over to zeros. However, this also largely depends on the values of the significant bits in the unshifted reference maximum. For example, if the reference maximum was 260 instead of 500, then Max 2 and Max 4 from Figure 3.13 would now be very accurate normalizations while Max 1 and Max 3 would introduce significant error. Both 500 and 260 have a leading 1 in the 8th bit position, yet the difference between the two numbers is large enough to heavily influence which shifted maximums will be accurate and which will contain large amounts of error. The maximum possible magnitude of error due to shifting is described by Equation 3.3, where m is the bit position of the leading 1 in the larger, unshifted number. Error of this significance can greatly influence the output of the WSMO algorithm, so a solution to mitigate the impact on the final result is discussed in the following section.

$$max\ \epsilon = 2^{m-1} - 1 \tag{3.3}$$

3.3.4 Adjustment of the function weighting parameter

In general, the maximums of either function are not known until the algorithm is applied using specific hardware and radar parameters. From this it can be concluded that finding a general solution for all cases would likely be both time and resource intensive, given the highly unpredictable source of the error. Consequently, an active correction strategy is implemented on an individual frame-by-frame basis based on the observed error in each case. The solution presented here adds a 3 cycle latency to the algorithm, but greatly increases the accuracy of the final results.

The approach used to correct this error does not directly address the normalization technique, but rather measures the error resulting from normalization and compensates accordingly in the calculation of the final weighted sum. Since the purpose of normalization in this algorithm is to prevent large weight differentials in the objective functions, it follows that errors in normalization can be compensated for with adaptive weighting. It is proposed that based on the magnitude of the observed error, an adjustment factor λ can be added to the weighting parameter α to compensate for weighting errors introduced by shift normalization while also preserving the weighting capability of α as a user defined value.

3.3.4.1 Calculation of the adjustment factor

To begin the calculation of the adjustment factor λ, recall the weighted sum defined in Equation 2.12. To rectify weighting errors caused by shifting, a new weight described by Equation 3.4 must be applied to Equation 2.12 to more accurately balance the two objective functions, forming Equation 3.5.

$$x = \alpha + \lambda \tag{3.4}$$

$$Z(\beta_i, f_j) = (1 - x)Z_2(\beta_i, f_j) - xZ_1(\beta_i, f_j) \tag{3.5}$$

For each frame of data, one objective function maximum will be larger than the other objective function maximum. The larger maximum will remain fixed, while the smaller maximum is shifted. Let these two values be denoted $fMax$ and $sMax$ for "fixed" and "shifted" maximum respectively. After the shift has occurred, the error in normalization ϵ is described by the remaining difference between $fMax$ and $sMax$.

$$\epsilon = fMax - sMax \tag{3.6}$$

In the case where $fMax$ and $sMax$ are exactly equal, then clearly the error would be zero. Although the theoretical minimum error of zero is possible, this occurrence would be exceedingly rare and cannot be assumed when calculating the final results. The maximum amount of error is defined in Equation 3.7 using Equations 3.3 and 3.6.

$$max\ \epsilon = fMax - sMax = 2^{m-1} - 1 \tag{3.7}$$

With error values ranging from 0 to $2^{m-1} - 1$, it would be ineffective to adjust the weight parameter x based on the maximum error, the minimum error, or any other fixed value within the range of possible error values. Clearly, it is necessary to adaptively scale the weight parameter to match the observed error for any particular frame of data.

The range of possible error values can be divided into groups using the CLZ module. Determining the difference between the leading 1 of $fMax$ and the leading 1 of the error ϵ divides the range into more refined groups of possible error, making the correction estimates more accurate. Table 3.2 shows the minimum and maximum error given that the difference between objective function maximums has a leading 1 within four bit positions of $fMax$. As in Equation 3.3, m is defined as the bit position of the leading 1 in $fMax$. The difference in bit position between leading ones is defined as D. Leading 1 differences of more than four bits have a negligible impact on the final result and are ignored.

D	Maximum error	Minimum error
1	$2^{m-1} - 1$	2^{m-2}
2	$2^{m-2} - 1$	2^{m-3}
3	$2^{m-3} - 1$	2^{m-4}
4	$2^{m-4} - 1$	2^{m-5}

Table 3.2: Ranges for different magnitudes of error

Table 3.2 can be generalized into a single inequality, defined in Equation 3.8.

$$2^{m-(D+1)} \leqslant \epsilon \leqslant 2^{m-D} - 1 \tag{3.8}$$

Now that the amount of error is established, the strategy to calculate the adjustment factor λ can be discussed. To begin this process, recall the original weighted sum in Equation 2.12. Now imagine this equation rewritten for only two elements, the two function maximums, as shown in Equation 3.9.

$$Z = (1 - \alpha)maxZ_2 - \alpha * maxZ_1 \tag{3.9}$$

Recall also that, using the division normalization method, both function maximums are always divided by themselves and will thus always be equal to 1. Given an equal weighting parameter $\alpha = 0.5$, Equation 3.9 will always evaluate to 0 for any two function maximums. This is the relationship that will be applied to Equation 3.5 to solve for the minimum and maximum values of the adjustment factor λ. Using this fact, the following can be derived:

$$Z = (1 - x)fMax - x * sMax = 0$$

Therefore,

$$(1 - x)fMax = x * sMax \tag{3.10}$$

Recalling 3.4,

$$max\ \epsilon = fMax - sMax = 2^{m-1} - 1$$

$$sMax = fMax - (2^{m-1} - 1)$$

Substituting into 3.10,

$$(1 - x)fMax = x * (fMax - (2^{m-1} - 1))$$

$$fMax - x * fMax = x * (fMax - (2^{m-1} - 1))$$

$$fMax = x * (fMax - (2^{m-1} - 1)) + x * fMax$$

$$fMax = x * (2fMax - (2^{m-1} - 1))$$

And finally,

$$x = \frac{fMax}{(2fMax - 2^{m-1} + 1)} \tag{3.11}$$

Equation 3.11 represents the the new weight to be used in the case that $fMax$ and $sMax$ create the largest possible error. However, this occurrence will be just as rare as the case where there is zero error, so the new weight must also be calculated for every case, shown in Table 3.2.

D	Maximum weight	Minimum weight
1	$x = \frac{fMax}{2fMax-(2^{m-1}-1)}$	$x = \frac{fMax}{2fMax-(2^{m-2})}$
2	$x = \frac{fMax}{2fMax-(2^{m-2}-1)}$	$x = \frac{fMax}{2fMax-(2^{m-3})}$
3	$x = \frac{fMax}{2fMax-(2^{m-3}-1)}$	$x = \frac{fMax}{2fMax-(2^{m-4})}$
4	$x = \frac{fMax}{2fMax-(2^{m-4}-1)}$	$x = \frac{fMax}{2fMax-(2^{m-5})}$

Table 3.3: New adjusted weights for varying differences in leading ones

As with Table 3.2, Table 3.3 can be generalized to an inequality, defining the upper and lower bounds of the new weight.

$$\frac{fMax}{2fMax - (2^{m-(D+1)})} \leqslant x \leqslant \frac{fMax}{2fMax - (2^{m-D} - 1)} \tag{3.12}$$

Fortunately, $fMax$ is available long before the new weight is needed, so calculation of these bounds is possible in real time. However, the bounds in Equation 3.12 still only describe the maximum and minimum amount of error and say nothing of where the error will fall between these bounds for any given frame of spectrum data. To accurately adjust the weight for each incoming frame, the actual error must be observed for each individual case. This has the unfortunate effect of relying on the calculation of the interference estimate maximum $sMax$, which completely negates any benefit provided by the early knowledge of $fMax$. Therefore, if the new weight is to be calculated precisely, the latency of the division in Equation 3.12 would be added to the critical path. As discussed previously, avoiding division is a top priority to preserve the speed of the calculation and reduce unnecessary resource usage. To meet these goals, an estimate of the error must be made quickly and with minimal resource consumption.

Before applying this information in an example, one more implementation detail must be discussed. In Verilog and in hardware in general, it is often easier and more effective to deal in only integers for any calculations performed. This is also true in the case of the weighting parameter α. In the original implementation, α is constrained to values between 0 and 1, consistent with normalization using division. However, as with normalization, using this scale is not strictly necessary. In the Verilog implementation, alpha is defined as an integer between 0 and 255 with a default value of 128, declared in Verilog as an 8-bit wire. Equation 2.12 then becomes

$$Z(\beta_i, f_j) = (255 - \alpha)Z_2(\beta_i, f_j) - \alpha Z_1(\beta_i, f_j) \tag{3.13}$$

which is functionally equivalent to Equation 2.12. With a default value of 128, the functions are multiplied by 127 and 128 respectively. This simulates an even weight value of $\alpha = 0.5$. The value 255 was chosen as the maximum weight because it is the largest number that can be represented with an 8-bit number. Making 255 the largest weight value and the largest value capable of being represented by the 8-bit wire α in the Verilog code prevents users from entering values of α over the allowed maximum.

As discussed previously, the scale of Z is irrelevant and does not affect the outcome of the algorithm. Scaling Z up by 255 will not change the maximum element in the function, and thus the indices of the maximum element representing the subband width b_i and starting frequency f_j remain unchanged.

As expected, the only effect this has on Table 3.3 and Equation 3.12 is an additional scaling factor of 255, as shown in Table 3.4 and Equation 3.14.

D	Maximum weight	Minimum weight
1	$x = \frac{255fMax}{2fMax-(2^{m-1}-1)}$	$x = \frac{255fMax}{2fMax-(2^{m-2})}$
2	$x = \frac{255fMax}{2fMax-(2^{m-2}-1)}$	$x = \frac{255fMax}{2fMax-(2^{m-3})}$
3	$x = \frac{255fMax}{2fMax-(2^{m-3}-1)}$	$x = \frac{255fMax}{2fMax-(2^{m-4})}$
4	$x = \frac{255fMax}{2fMax-(2^{m-4}-1)}$	$x = \frac{255fMax}{2fMax-(2^{m-5})}$

Table 3.4: New adjusted weights for varying differences in leading ones

$$\frac{255fMax}{2fMax - (2^{m-(D+1)})} \leqslant x \leqslant \frac{255fMax}{2fMax - (2^{m-D} - 1)} \tag{3.14}$$

Now consider an example based on collected sample data with a total bandwidth of 100MHz, making $fMax = 100 * 10^6$. One hundred million has a leading one in the 27th bit position, so $m = 27$. Applying these values to Table 3.4, Table 3.5 shows the resulting weight values corresponding to maximum and minimum error conditions.

Using these weight ranges as a basis, many approaches to choose an appropriate value within the ranges were attempted. Using the maximum or minimum values directly did not yield results with a desired level of success. Averaging the minimum ad maximum values produced slightly better results, but ultimately

D	Maximum weight x	Minimum weight x	Max λ	Min λ
1	$\frac{255*100*10^6}{200*10^6-(2^{27-1}-1)} = 191.9$	$\frac{255*100*10^6}{200*10^6-(2^{27-2})} = 153.2$	63.9	25.2
2	$\frac{255*100*10^6}{200*10^6-(2^{27-2}-1)} = 153.2$	$\frac{255*100*10^6}{200*10^6-(2^{27-3})} = 139.2$	25.2	11.2
3	$\frac{255*100*10^6}{200*10^6-(2^{27-3}-1)} = 139.2$	$\frac{255*100*10^6}{200*10^6-(2^{27-4})} = 133.1$	11.2	5.1
4	$\frac{255*100*10^6}{200*10^6-(2^{27-4}-1)} = 133.1$	$\frac{255*100*10^6}{200*10^6-(2^{27-5})} = 130.2$	5.1	2.2

Table 3.5: Ranges of new weight at fMax = 100MHz

these were still unsatisfactory. After experimentation with different combinations of adjustment factors, it was discovered that adjusting the weight by multiples of 8 produced highly effective results for the 100MHz data. This is a somewhat specific solution, however when comparing the multiples of 8 to the ranges for λ it becomes more obvious that this is a logical conclusion based on the computations done up to this point. The Verilog statement "adjustment <= 8*(4 - diffLeading1s);" is used to calculate λ and store the result in the register "adjustment." This results in λ values of 8, 16, and 24 for bit differences of 3, 2, and 1 respectively. When comparing these with Table 3.5, these values match remarkably well with the calculated ranges, making it clear why the results improve significantly using this adjustment factor. In the following section, results before and after weight adjustment are compared.

3.3.4.2 Result comparison

To test the effectiveness of the weight adjustment strategy, comparisons were made on 100 frames of spectrum data with a bandwidth of 100MHz collected using the X310 radio. Figures 3.14 and 3.15 show the results before weight adjustment is applied, and Figures 3.16 and 3.17 show the results of the same spectrum frames after adjustment has been applied.

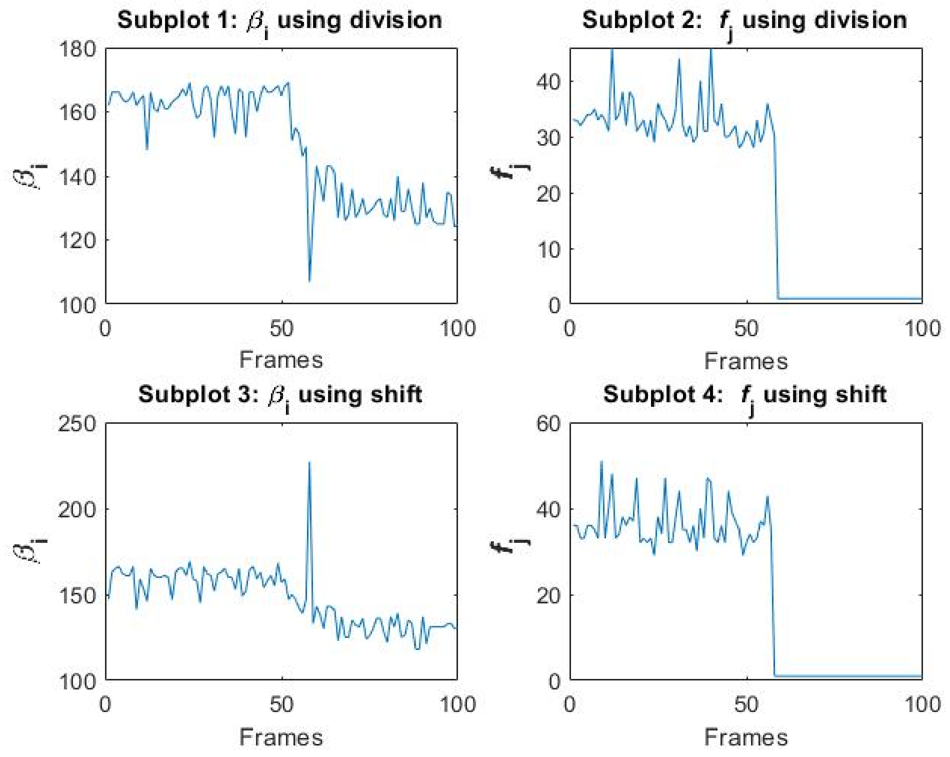


Figure 3.14: Comparison between division and shifted β_i and f_j values before applying error correction adjustment factor

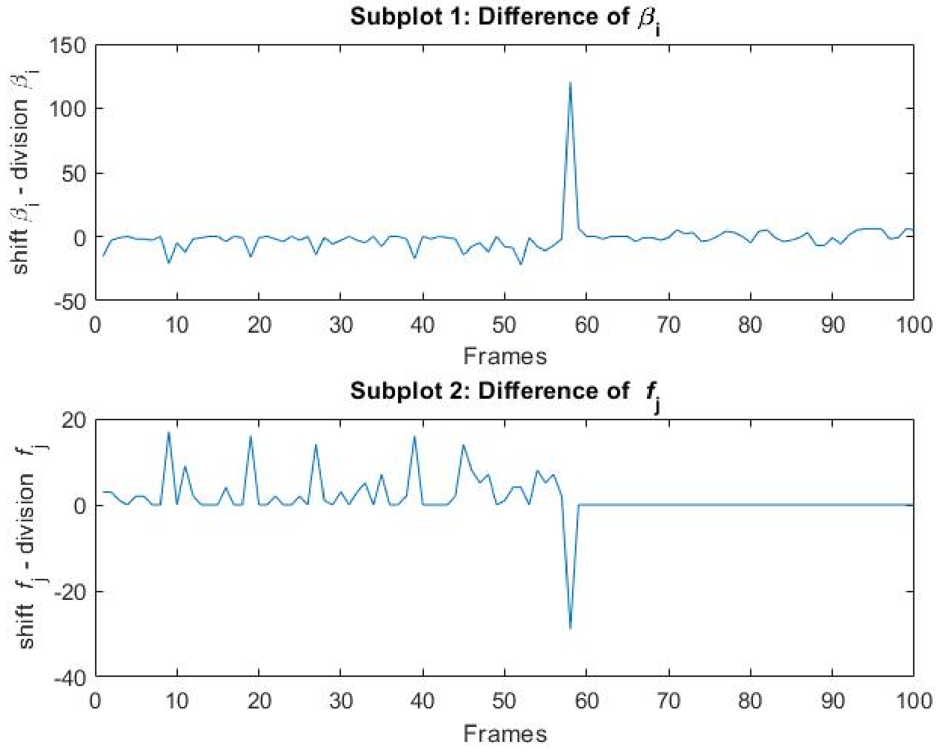


Figure 3.15: Difference plots showing how much shifting deviates from division β_i and f_j values before applying error correction adjustment factor

Figure 3.14 compares the output values β_i and f_j of the shift normalization method with the reference results using division. The most easily observed occurrence of error occurs at frame 58 where a sharp decrease occurs when using division, but shifting causes a sharp increase in bandwidth. This is the largest example of the error due to shifting. Figure 3.15 provides plots of the difference between the outputs to better visualize how different the plots in Figure 3.14 are from each other. Notice the difference in vertical scale between Figure 3.15 and the weight adjusted plot shown in the following figures. Before adjustment, β_i exhibits a maximum error of nearly 125 and f_j shows a maximum error of approximately -30 with several other peaks showing errors near 20.

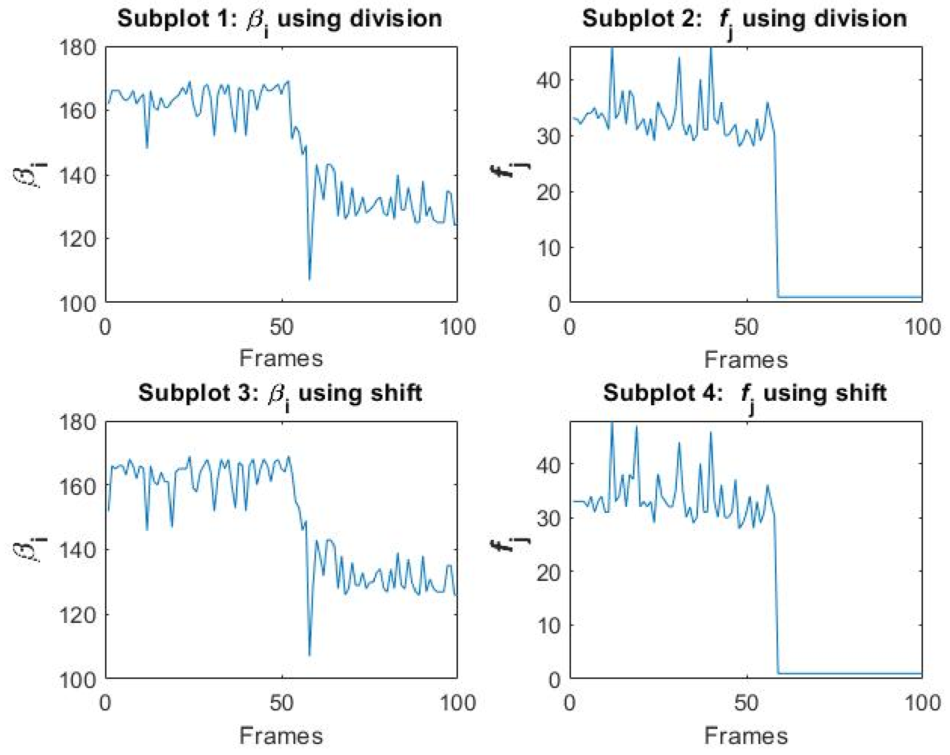


Figure 3.16: Comparison between division and shifted β_i and f_j values after applying error correction adjustment factor

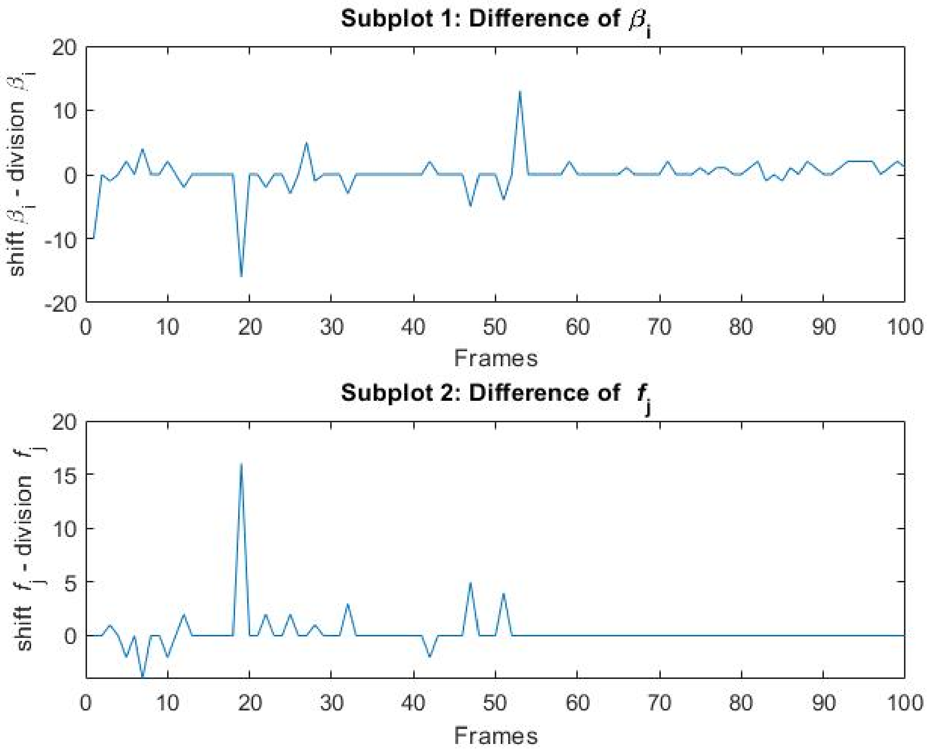


Figure 3.17: Difference plots showing how much shifting deviates from division β_i and f_j values after applying error correction adjustment factor

As expected, Figures 3.16 and 3.17 show great improvement over the pre-adjustment figures. The outputs β_i and f_j in Figure 3.16 follow the trends of the division reference plots much more closely than the pre-adjustment results. This is verified in Figure 3.17 which reveals that the difference compared to division has been greatly decreased. The maximum error for β_i is just over -15 while the smaller peaks have been reduced even further to errors less than 5. This is nearly a 90% reduction in error from the non-adjusted results. Likewise, the maximum error for f_j has been reduced to just over 15 with the smaller peaks being reduced to less than 5, marking an error reduction of approximately 75%.

3.4 Full weighted sum function Z

Calculation of the full weighted sum is straightforward. In Verilog, it is possible to multiply each element of the objective functions by its corresponding weighting parameter in parallel and assign the results to registers in a single cycle. Using generate statements that perform loop unrolling on for loops ensures that each element can be multiplied and assigned independently of the other elements. Once each separate objective function is calculated, the interference estimate is subtracted from the subband width to form the full weighted sum Z. Again, generate statements and for loops provide the means to perform the subtraction of all elements in parallel, taking only one cycle to complete. The intermediate results of the multiplication and subtraction steps are not registered, so given that the latency of a multiplier and adder/subtracter is not longer than the clock period, both operations can occur in a single clock cycle.

3.5 Finding the maximum value of Z

Finding the maximum value of Z is also conceptually straightforward, but requires a large amount of comparison logic. In Verilog, Z is essentially a 2D array. In the hardware, this translates to a large matrix of registers where the values of Z are stored. Verilog does not support passing memory (an array of registers) as an input to a module. The comparisons could have been done directly in the top module, but this is bad practice and would have made the code bloated and difficult to understand [20]. So, the best solution to get the values from memory into the comparison block was to concatenate all elements of a single row and assign them to a wire which can be passed as an input to the module. The concatenated rows are then separated back into individual elements within the module. This approach does, however, rely heavily on efficient routing by the synthesis software. For the example of 256, 64-bit elements, the resulting concatenated wire would be 256x64 = 16,384 bits wide. In addition, these 16,384 bit wires would only carry the elements of a single row of Z, resulting in only the maximum value for that row. So, for the 256x256 matrix of Z, a 16,384 bit wire would be necessary for each row, resulting in 256 16,384-bit wires to route from memory to the comparison module. This is quite a large task that is caused by Verilog constraints, however, if the memory were able to be directly routed to the module, the total number of bits would be the same, so the concatenation likely causes little extra work than would ordinarily need to be done. A brief discussion of the Verilog modules used to perform the comparisons is considered in the following paragraph.

Two modules are used to complete this task, named greatest_of_N and compare_two. The compare_two module is the lowest level of comparison, comparing only two 64-bit signed elements. The behavior of this module is simple and straightforward. The module accepts 6 inputs: two element values of Z and their associated indices in the Z matrix, representing β_i and f_j for that element. The two elements are compared using a signed comparison, and the larger value and its associated β_i and f_j are passed as the three module outputs. This synthesizes to hardware as simple as one comparator and three multiplexers, pictured in Figure 3.18.

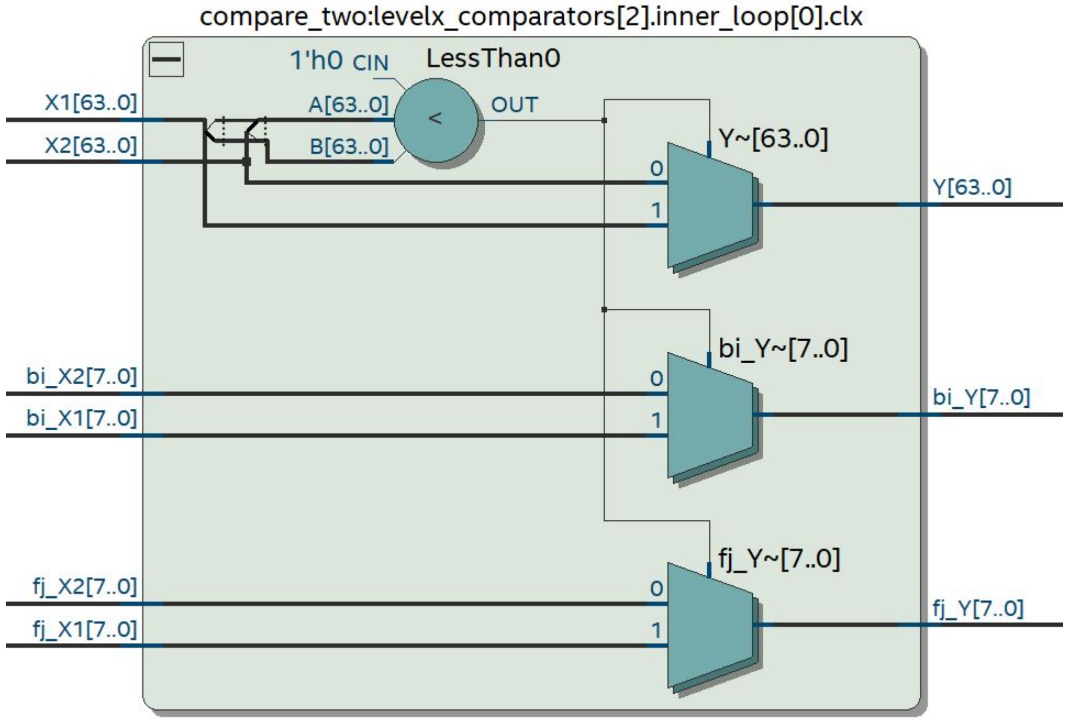


Figure 3.18: compare_two module synthesized hardware diagram

The second module, greatest_of_N, uses compare_two as the basic building block to compare every element of Z and ultimately find the maximum value. As with the calculation of Z in Section 3.4, this module relies heavily on generate statements and for loops to accomplish the thousands of module instantiations. Each greatest_of_N module must create $N-1$ instantiations of compare_two to compare every element of the row and determine the maximum. The module is designed to be parameterized in N to allow for different FFT lengths to be used while still comparing an entire row of Z using one module instantiation. Upon receiving the concatenated array of elements, the elements are separated and passed in pairs into the compare_two module to determine which is larger. For rows of 256 elements, a generate statement and for loop are used to instantiate 128 compare_two modules for the first layer of comparisons. After the first layer which requires the concatenated array as an input to the compare_two modules, the remaining layers of compare_two modules can be instantiated using nested for loops because the inputs come from a previous layer and the indices used to index the previous comparison results are uniform. After each comparison layer, half of the input elements are discarded and the larger half of the elements are passed to the next layer of compare_two modules. The number of comparison layers will be equal to $log_2(N)$ where N is the number of elements in a row of Z. The layers will continue until only one element remains. The last remaining element is the largest element of that row, so it is passed along with its corresponding β_i and f_j as the output to the module. This synthesizes to a typical comparator tree structure, a section of which is pictured in Figure 3.19.

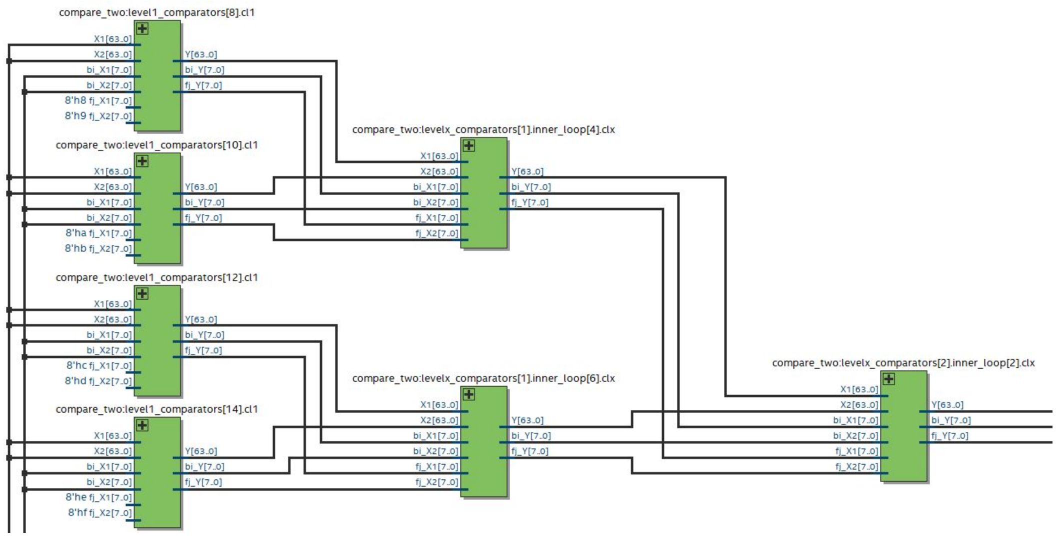
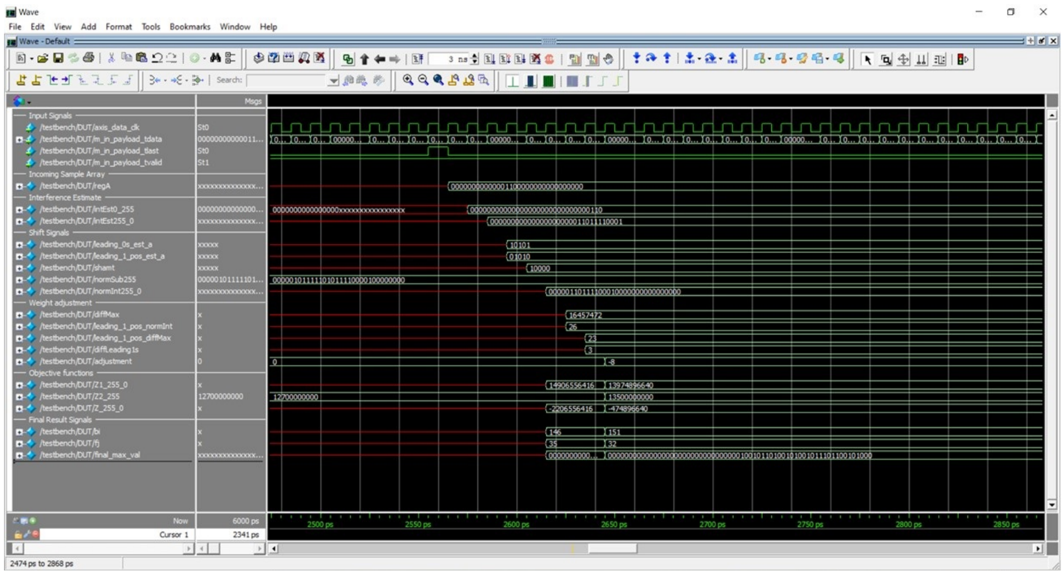

compare_two:level1_comparators[8].cl1
compare_two:level1_comparators[10].cl1
compare_two:level1_comparators[12].cl1
compare_two:level1_comparators[14].cl1
compare_two:levelx_comparators[1].inner_loop[4].clx
compare_two:levelx_comparators[1].inner_loop[6].clx
compare_two:levelx_comparators[2].inner_loop[2].clx
X1[63..0]
X2[63..0]
bi_X1[7..0]
bi_X2[7..0]
fj_X1[7..0]
fj_X2[7..0]
Y[63..0]
bi_Y[7..0]
fj_Y[7..0]
8'h8
8'h9
8'ha
8'hb
8'hc
8'hd
8'he
8'hf

Figure 3.19: Partial greatest_of_N module synthesized hardware diagram

In the main module, the maximum of each row is stored, concatenated, and passed into one final greatest_of_N module to produce the final maximum of Z. The indices of this element are the final outputs of the WSMO algorithm and represent the optimized subband width β_i and starting frequency f_j.

3.6 Total timing diagram

The entirety of this algorithm can be completed in only 9 cycles after the last spectrum sample becomes available. Figure 3.20 provides a screenshot of the Modelsim FPGA simulation software showing many critical signals used in calculating the final results, β_i and f_j. As can be seen in Figure 3.20, the last sample of spectrum data is available during the cycle that the signal m_in_payload_tlast is asserted high, and the final adjusted results of $\beta_i = 151$ and $f_j = 32$ are available 9 clock cycles after this assertion.


```
Input Signals
  /testbench/DUT/axis_data_clk                St0
  /testbench/DUT/m_in_payload_tdata           000000000000011...
  /testbench/DUT/m_in_payload_tlast           St0
  /testbench/DUT/m_in_payload_tvalid          St1
Incoming Sample Array
  /testbench/DUT/regA                         xxxxxxxxxxxxxxx...
Interference Estimate
  /testbench/DUT/intEst0_255                  00000000000000...
  /testbench/DUT/intEst255_0                  xxxxxxxxxxxxxxx...
Shift Signals
  /testbench/DUT/leading_0s_est_a             xxxxx
  /testbench/DUT/leading_1_pos_est_a          xxxxx
  /testbench/DUT/shamt                        xxxxx
  /testbench/DUT/normSub255                   0000010111101...
  /testbench/DUT/normInt255_0                 xxxxxxxxxxxxxxx...
Weight adjustment
  /testbench/DUT/diffMax                      x
  /testbench/DUT/leading_1_pos_normInt        x
  /testbench/DUT/leading_1_pos_diffMax        x
  /testbench/DUT/diffLeading1s                x
  /testbench/DUT/adjustment                   0
Objective functions
  /testbench/DUT/Z1_255_0                     x
  /testbench/DUT/Z2_255                       12700000000
  /testbench/DUT/Z_255_0                      x
Final Result Signals
  /testbench/DUT/bi                           x
  /testbench/DUT/fj                           x
  /testbench/DUT/final_max_val                xxxxxxxxxxxxxxx...

Now        6000 ps
Cursor 1   2341 ps
2474 ps to 2868 ps
```

Figure 3.20: WSMO timing diagram

Chapter 4

Results

Results of the algorithm given varying spectrum data inputs are provided in this chapter. Section 4.1 showcases the performance of the algorithm under good conditions, Section 4.2 shows the limits of the algorithm when faced with low SNR inputs, and Section 4.3 shows how the user defined weight parameter α can be changed to cause the algorithm to select different bands.

4.1 Initial Results

The following figures present various frames of spectrum data to show the performance of the WSMO algorithm for signals with acceptable SNR ($\approx$13.1dB to 16.5dB) [21] and evenly weighted objective functions where $\alpha = 128$, the integer equivalent of $\alpha = 0.5$. Each plot features the result obtained when normalizing using division for comparison with the shifted results. As can be seen in the figures, the shifted results maintain a high level of accuracy to the division results, while also providing the benefit of extremely fast computation.

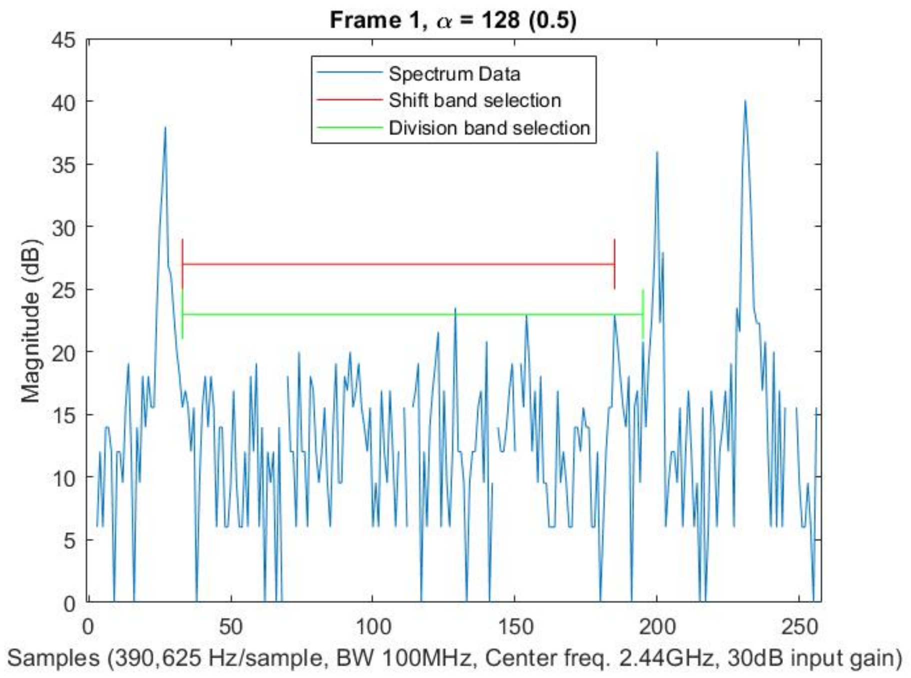


Figure 4.1: Frame 1, well-defined peaks, evenly weighted α

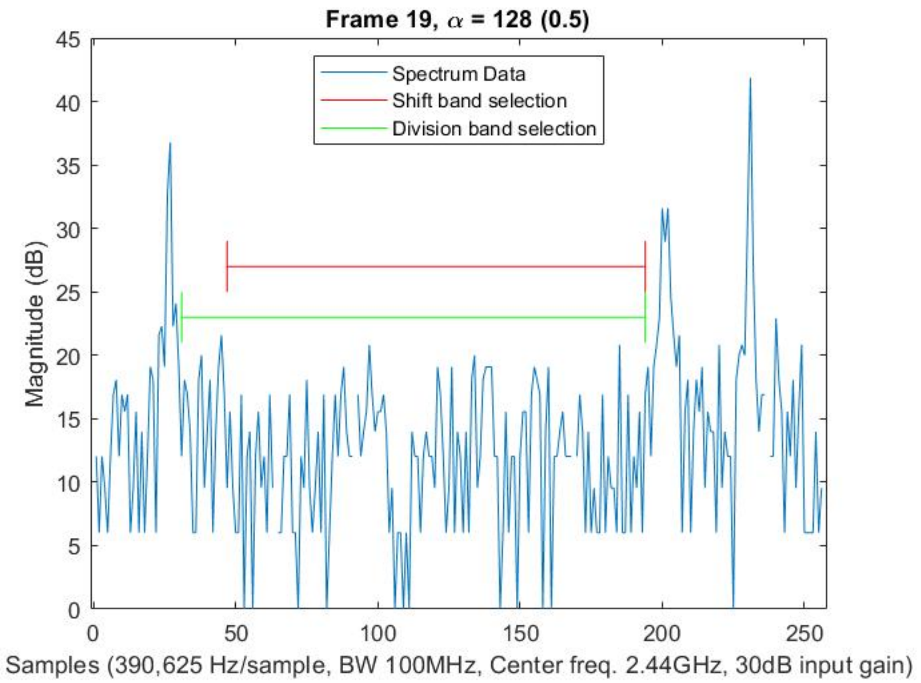


Figure 4.2: Frame 19, well-defined peaks, evenly weighted α

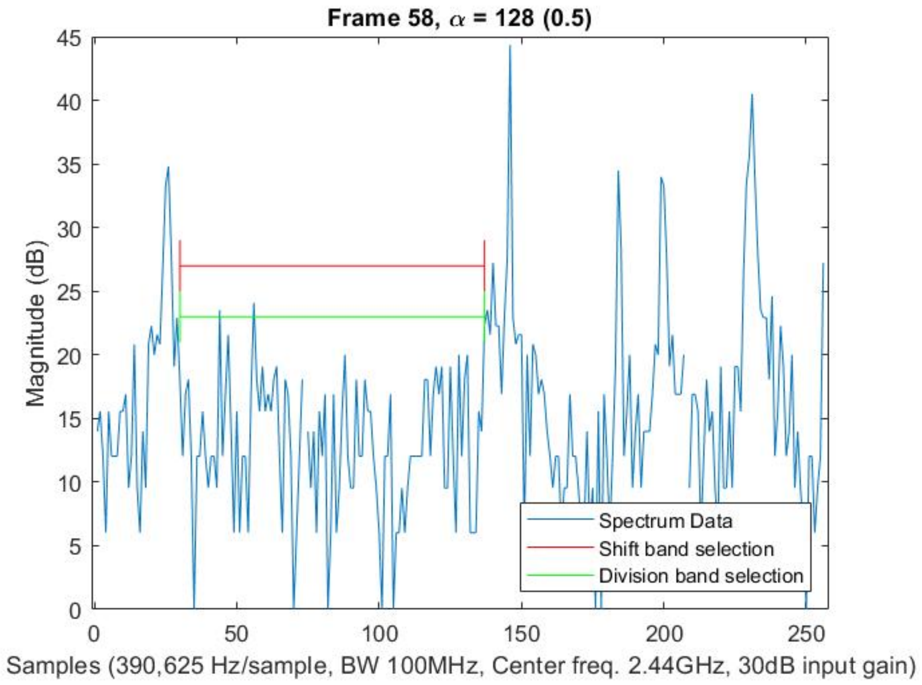


Figure 4.3: Frame 58, well-defined peaks, evenly weighted α

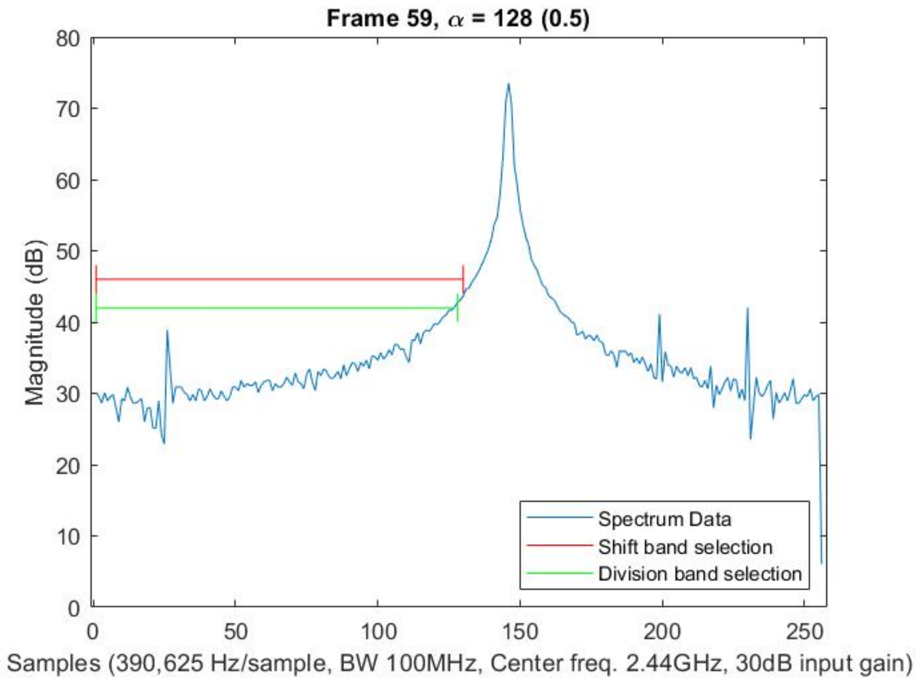


Figure 4.4: Frame 59, well-defined peaks, evenly weighted α

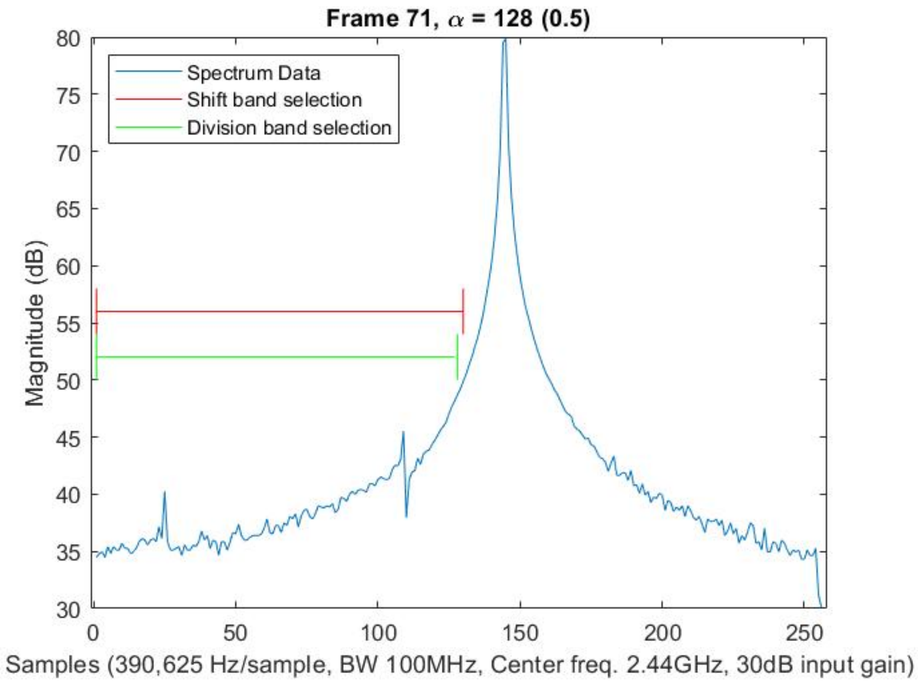


Figure 4.5: Frame 71, well-defined peaks, evenly weighted α

Figures 4.1-4.5 show that the results obtained by shifting are not identical to the division results. As discussed at length in Chapter 3, this is due to the unpredictability of the error caused by the shift. The weight adjustment correction strategy presented in Section 3.3.4 significantly improves the results to the level seen in Figures 4.1-4.5, however, a more robust solution would further improve the accuracy of the shifted results.

4.2 Decreased SNR Results

The figures in the previous section showcase the performance of the algorithm under normal conditions, so this section will examine the performance as the signal peaks are reduced closer to the noise floor. The first frame of collected data, shown in Figure 4.1, is used for this example and is shown again below for reference.

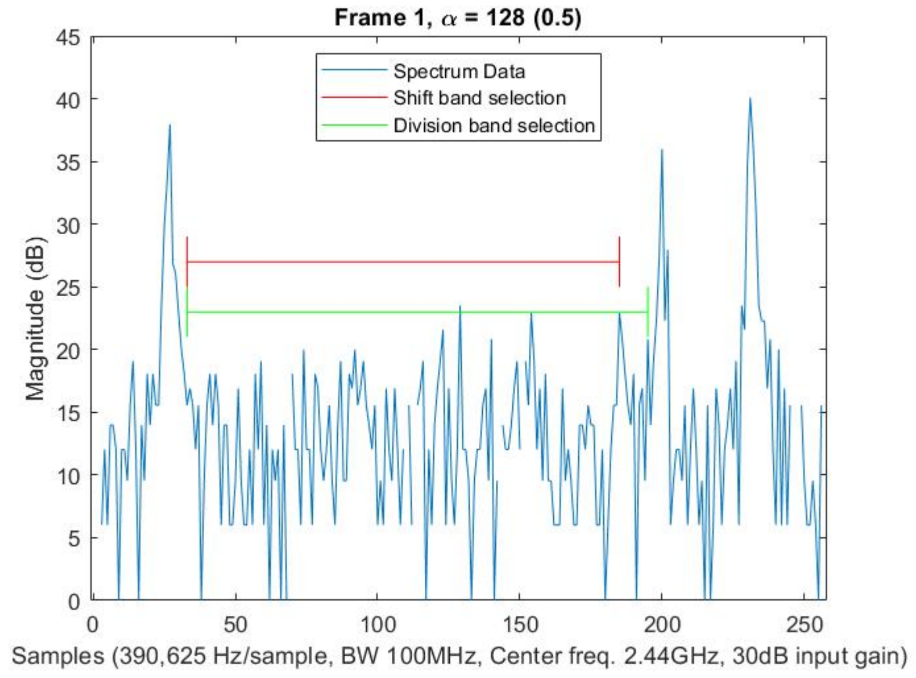


Figure 4.6: Natural signal power

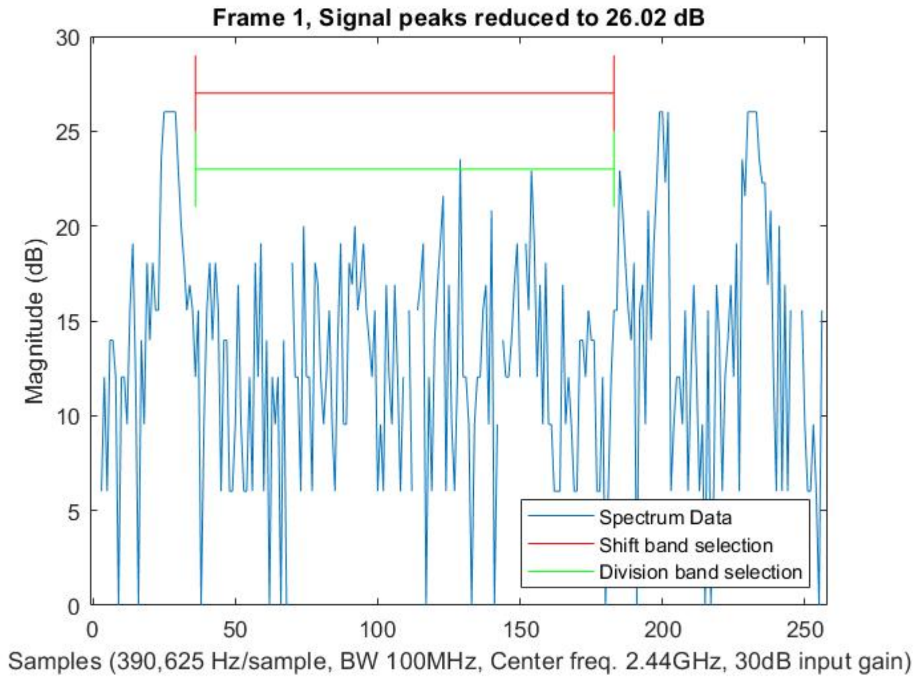


Figure 4.7: Peaks reduced to $\approx$ 26dB

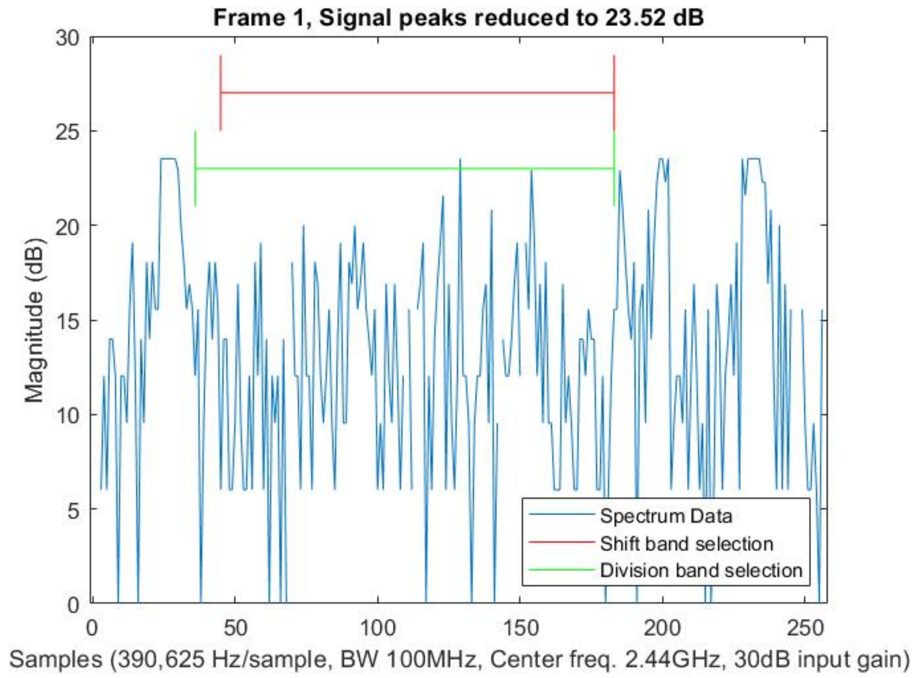


Figure 4.8: Peaks reduced to ≈ 23dB

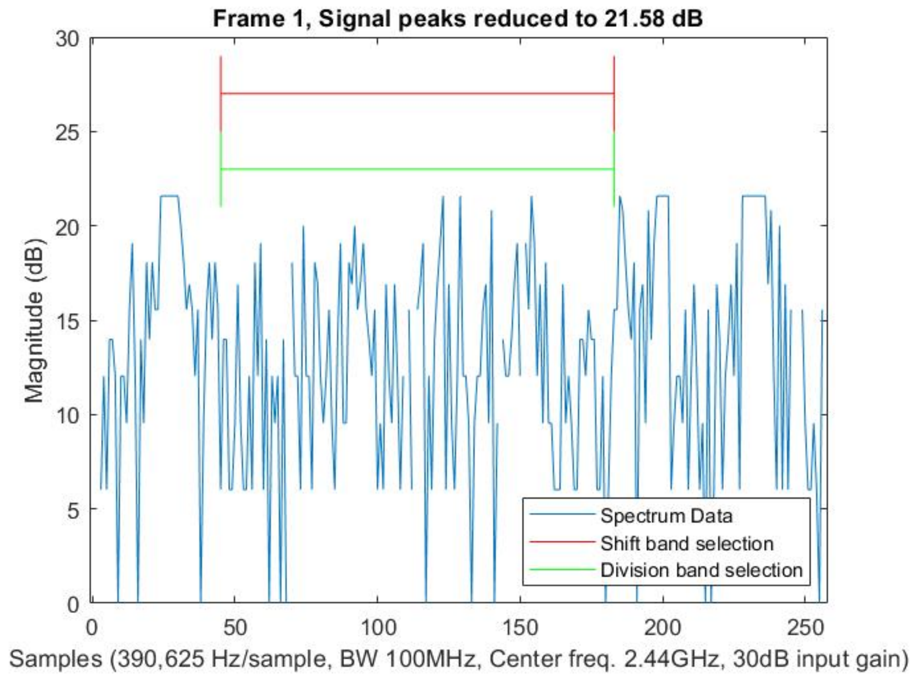


Figure 4.9: Peaks reduced to ≈ 21dB

The algorithm continues to pick the appropriate band, even when the signal peaks have been reduced so much that they appear nearly indistinguishable from the noise.

By examining Figures 4.6-4.9 more closely, a pattern can be observed between the division and shift selections. As the peaks are reduced, both results begin to move to a more conservative bandwidth, just at slightly different rates. This behavior is expected because reducing the peaks is equivalent to amplifying the noise. Naturally, the weighted sum will begin to shift to smaller bandwidths that reduce the amplified interference. Both methods exhibit the same bandwidth reducing behavior, but the shift method begins at a more conservative estimate to begin with, causing it to lag behind the division results as the peaks are reduced. But what does this mean? In this particular case, the difference in results shows simply that the weights are not exactly equivalent. For example, in frame 1, α is set to 128 for the shift method and 0.5 for the division method. This would appear to weight the functions evenly, as has often been done in this paper. However, the resulting difference in selected bandwidths shows that this is not quite the case. Again, this stems from the fact that shifting does not produce exactly equivalent objective function maximums, and that the method to correct this skewed weight is also an approximation and does not provide a precisely accurate correction. To demonstrate this, Figures 4.10-4.13 are provided to show the results for the same frame of data when α is set to 126 for the shift method and 0.5 for the division method.

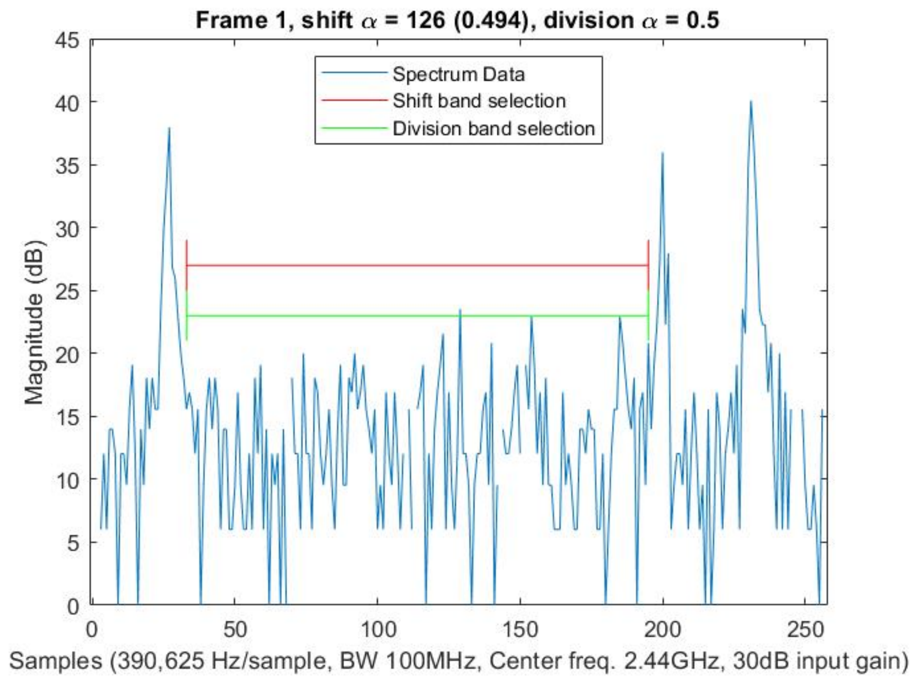


Figure 4.10: Non-equivalent alphas, natural signal strength

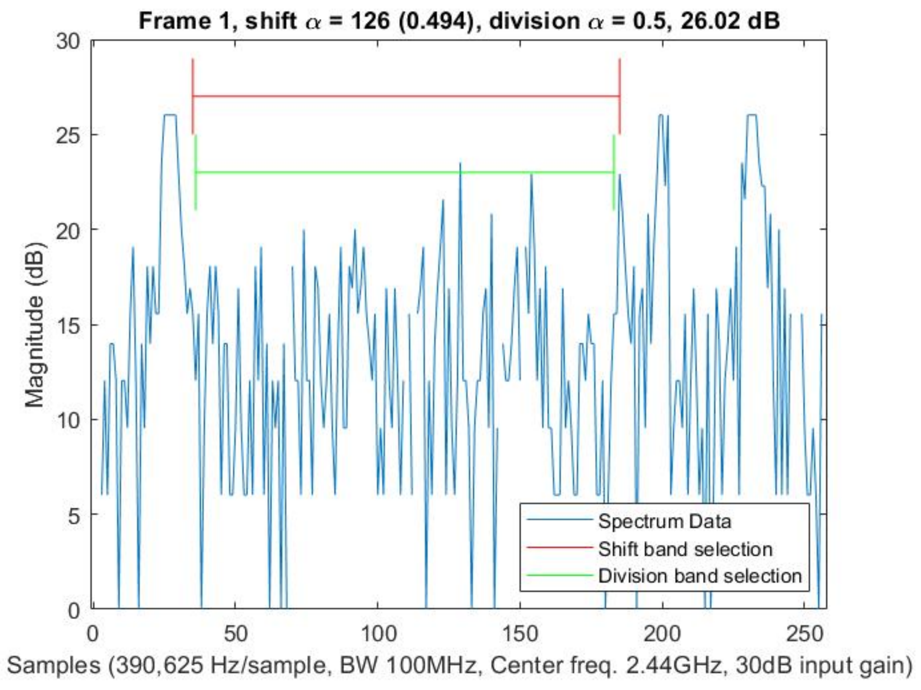


Figure 4.11: Non-equivalent alphas, signal peaks $\approx$ 26dB

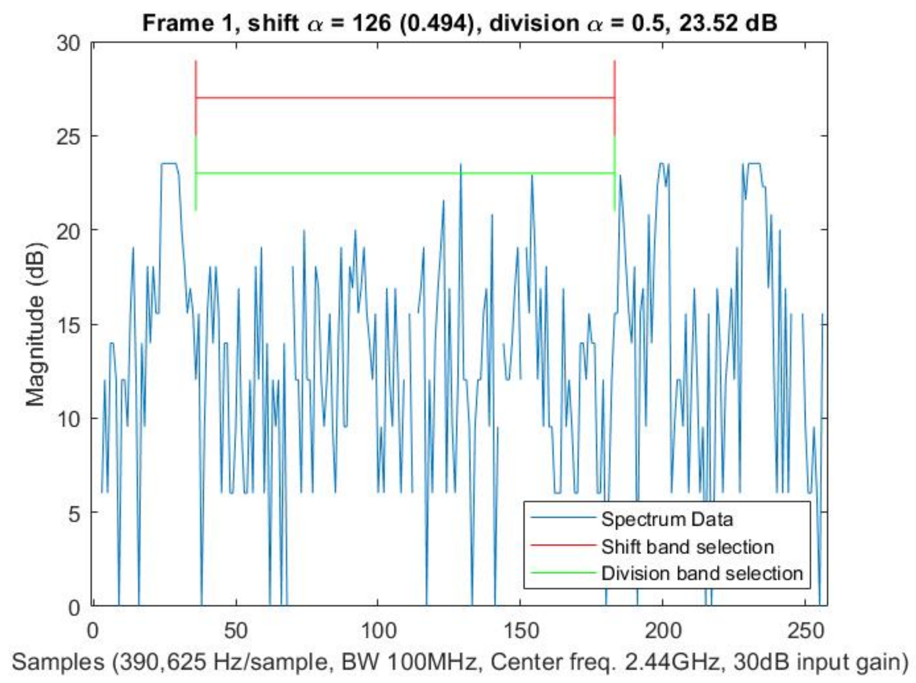


Figure 4.12: Non-equivalent alphas, signal peaks $\approx$ 23dB

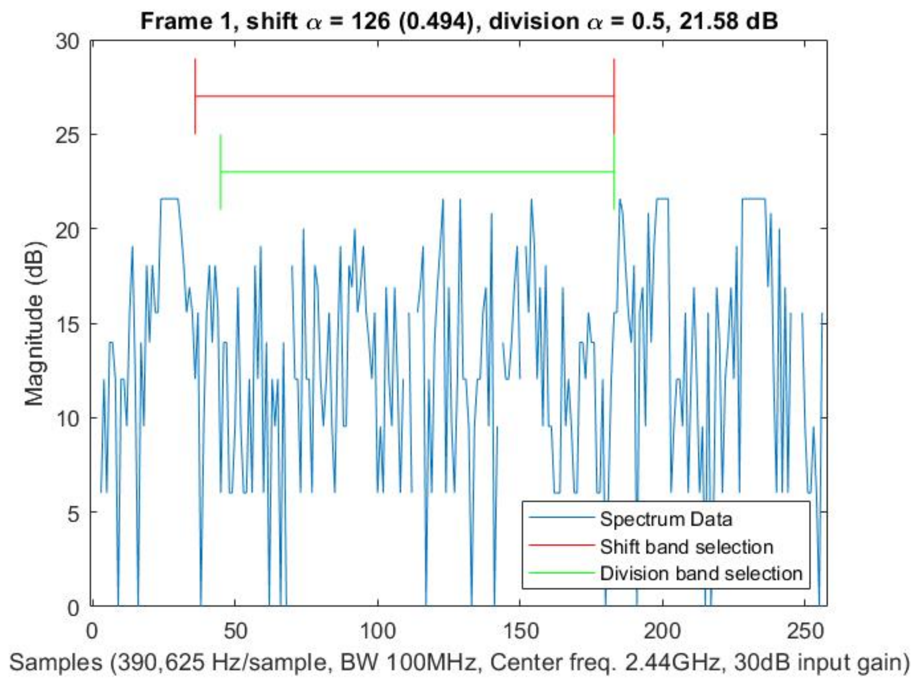


Figure 4.13: Non-equivalent alphas, signal peaks $\approx$ 21dB

The shift band selections now align much more closely with the division selections as a result of only a slight weight adjustment. This discovery is what prompted the investigation into the weight adjustment correction strategy presented in Section 3.3.4.

It is also interesting to note that this weight adjustment to slightly favor bandwidth over interference also causes the shift method in Figure 4.13 to choose a slightly larger band than the division method, without overtaking the valid signal. Depending on the radar application, this could be seen as a positive change over the division method. This demonstrates that not every difference between the results of each method has to be considered in terms of a “right” or “wrong” selection. Different radar applications will have different preferred band selections, prompting the increase or reduction of α to meet the specific application requirements. When this algorithm is running, a second “correct” reference method will not be available to compare to, so the only reference that matters will be the performance of the radar. If performance is poor because the selected band is too small, then α can be decreased to favor bandwidth. It will not matter that the change in α would have been larger or smaller using division to achieve this result, only that the result is achievable with some change in α. However, it should be noted that this variance in α is not constant across all frames of input data. Rather, the variation in α could change from frame to frame, resulting in a closely matching band selection as the division method for some frames, but causing increased difference in band selection for other frames. A more advanced method to determine the adaptive weighting parameter would effectively reduce the variance in α between the shift and division methods by ensuring that both objective functions are perfectly weighted to begin with. A few examples of using α to select different bands are shown in the next section.

4.3 Changing α

The user defined parameter α is responsible for changing the priority of the weighted sum to favor increasing bandwidth or minimizing interference. An increase in α will weight the interference estimate function more heavily causing a tendency for the algorithm to select smaller bandwidths with less interference. Inversely, decreasing α will more heavily weight the bandwidth function causing a tendency to select larger bandwidths that contain more interference. Figures 4.14 and 4.15 show the bandwidth selection for two similar frames of data using an evenly weighed α, while Figures 4.16 and 4.17 show the same frames when alpha is changed to select a smaller bandwidth. The ability to change the preferred selected bandwidth using α has obvious benefits when considering that different radar applications often perform better under different conditions. However, the figures below also showcase another benefit changing α can have for a cognitive radar. In Figures 4.14 and 4.15, it can be seen that one signal clearly dominates the spectrum, causing the algorithm to select a bandwidth containing some possibly valid signals at approximately 40dB. If it is desired to avoid transmission over these signals, changing α can help the algorithm recognize smaller peaks as valid signals and select a smaller bandwidth around those points.

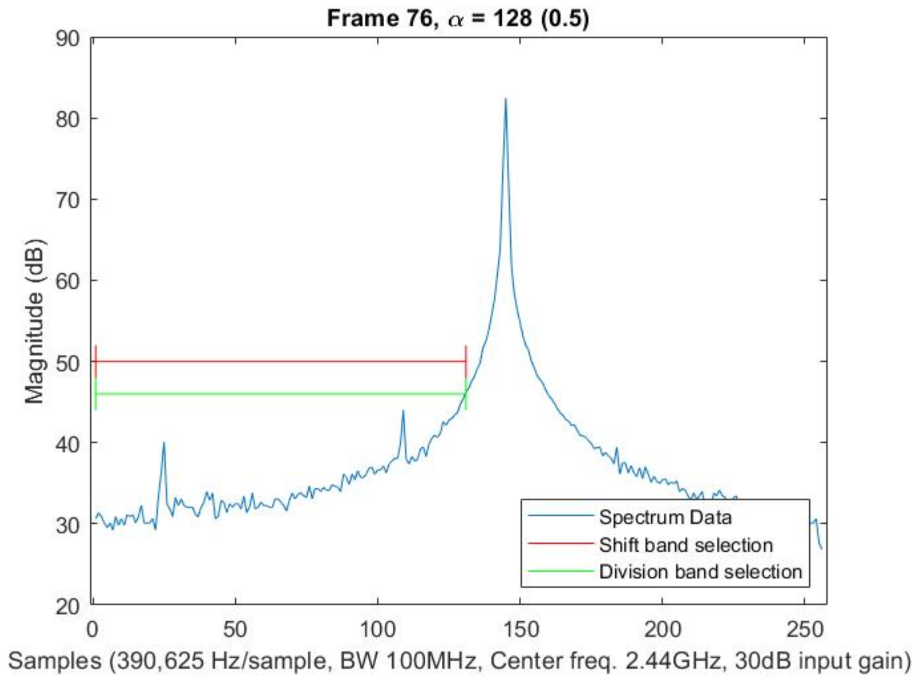


Figure 4.14: Frame 76, evenly weighted α

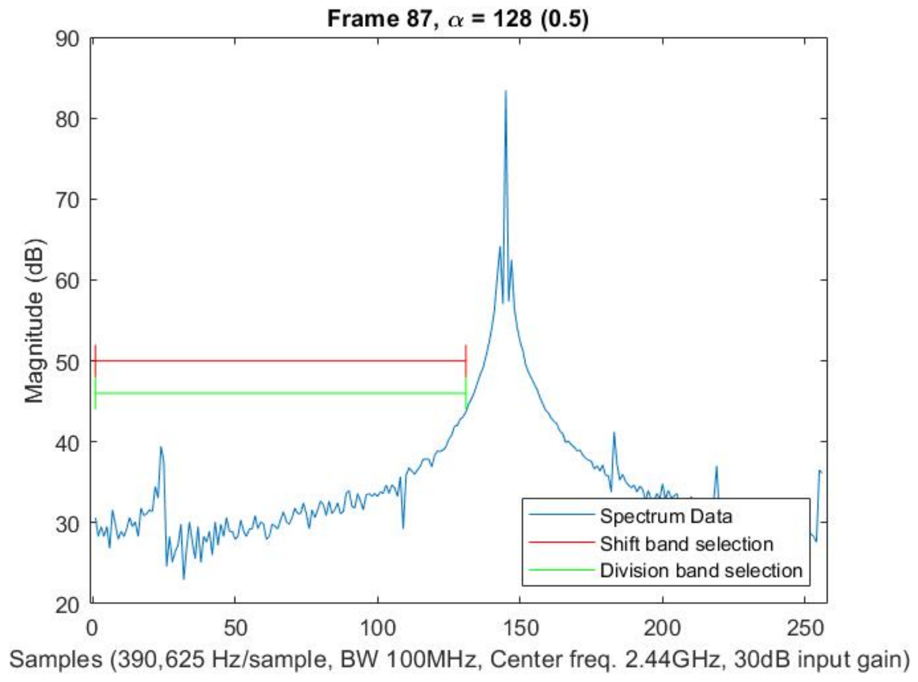


Figure 4.15: Frame 87, evenly weighted α

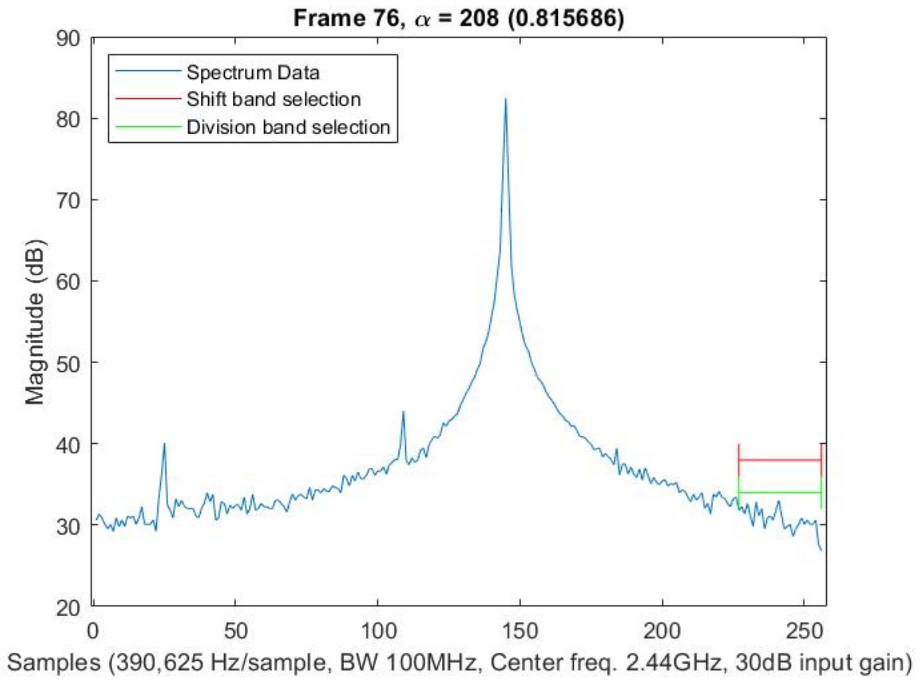


Figure 4.16: Frame 76, increased α selects smaller band without interfering with 40dB signals

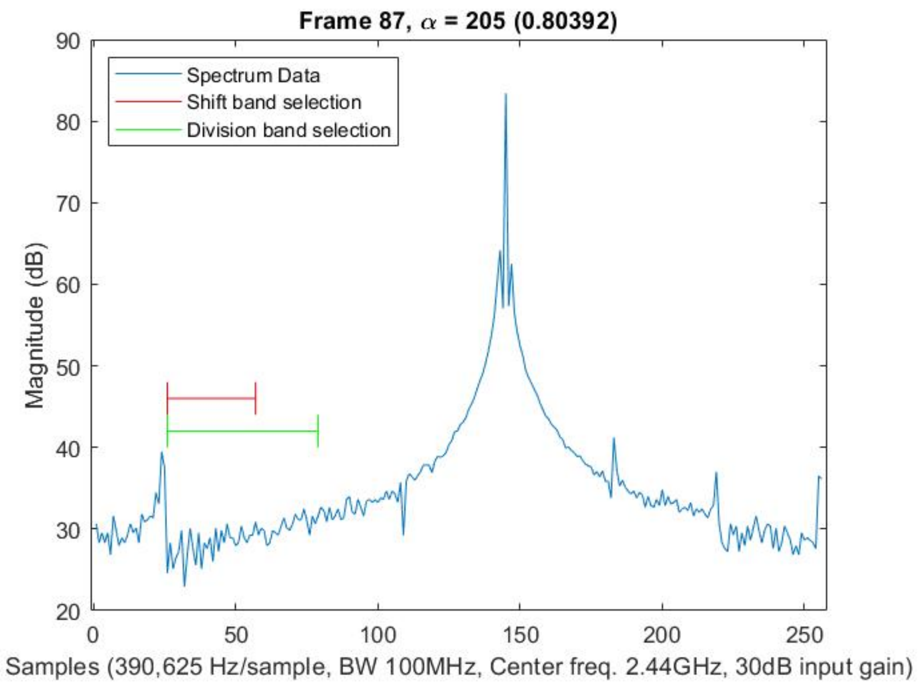


Figure 4.17: Frame 87, increased α selects smaller band without interfering with 40dB signal

Again, Figure 4.17 shows that the α for each method is not perfectly equivalent, so Figure 4.18 shows that the exact division result can be achieved by the shift method with an α of 198.

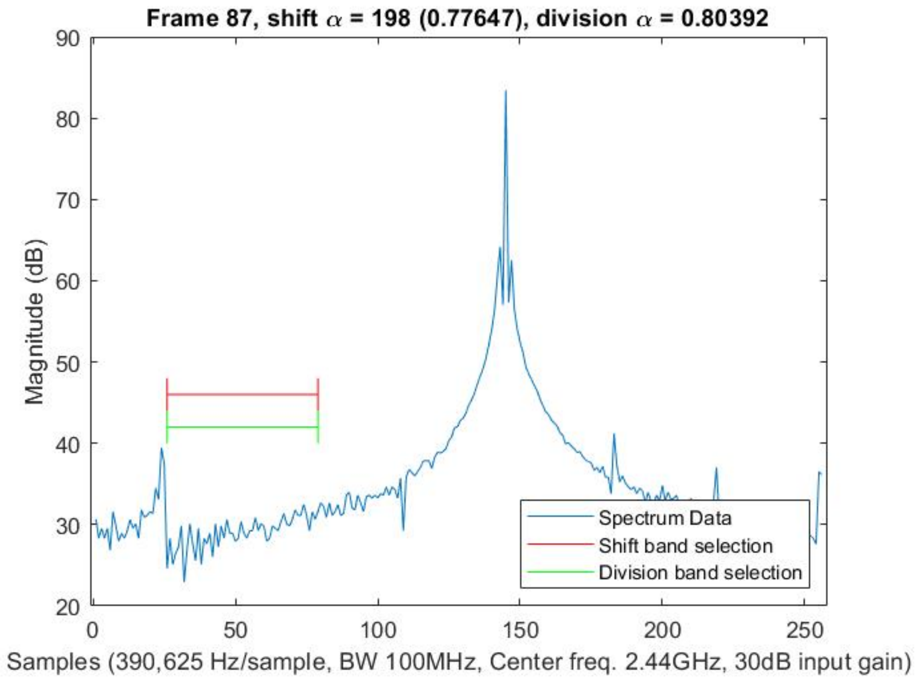


Figure 4.18: The shift method is able to achieve the same results as the division method with a slightly different α

Chapter 5

Conclusion and Future Work

The purpose of this work is to develop a fast implementation of the WSMO algorithm for deployment on an FPGA. Such an implementation can provide benefits to devices that employ spectrum sharing and contain an FPGA with free space available for custom user logic. Several implementation strategies and changes to the original WSMO algorithm contributed to the speed-up achieved by this FPGA implementation.

To help facilitate the high-speed design, the original weighted sum equation was rearranged to use subtraction to replace division in the first objective function. As one of the most expensive basic mathematical operations, eliminating the need for division is critical to save computation time and reduce FPGA resource usage.

Eliminating division is also the impetus for using bit-shift operations for normalization of the objective functions. Since the particular values of the weighted sum are immaterial in producing a valid result, using division to achieve a traditional normalization scale is not necessary. Replacing division with a shift operation provides a significant reduction in the number of clock cycles required to compute the final result.

While the use of shift operations to replace division provides a sizable speed increase, the introduction of error due to the fundamental difference between the operations is inevitable. A method to reduce the impact of this error was developed. The use of adaptive weighting of the objective functions allows error caused by shift normalization to be actively reduced on a frame-by-frame basis.

The result of the listed implementation changes is a new algorithm that closely matches the performance of the original algorithm while significantly increasing the speed. While the computation time and complexity of the original sequential implementation increases at an exponential rate as the number of FFT samples increases, a total computation time of only nine clock cycles is achieved with the parallelized FPGA implementation.

Many improvements could be made to this implementation in future work. To enable practical use in cognitive radar applications, reintroduction of the radar specific parameters used in the original algorithm is necessary. Techniques such as inverse scaling of the opposite objective function should be explored, as this could produce the same end result without reintroducing division operations into the calculations.

To increase the reliability and consistency of the algorithm, a formal analysis of the shift error should be performed. If the error can be modeled by a statistical distribution or approximated as additive noise, similar to quantization error in analog-to-digital converters, then a more rigorous, random signals-based approach to determine the adaptive weighting parameter could be developed. A more robust method to set the value of the adaptive weight would help reduce the difference between the shift and division normalization methods, improving the accuracy of shift normalization when using division as the standard.

An investigation into increasing the scalability of the algorithm should be conducted. As the FFT length increases, the amount of required logic increases at an alarming rate that would render most modern FPGAs incompatible with this implementation. Further examination of more efficient logic structures that enable reduction of FPGA resource utilization would result in greater practicality in the application of this algorithm and allow for larger FFT frames to be used.

References

[1] A. F. Martone, K. I. Ranney, K. Sherbondy, K. A. Gallagher, and S. D. Blunt, "Spectrum allocation for noncooperative radar coexistence." *IEEE Transactions on Aerospace and Electronic Systems*, vol. vol. 54, no. no. 1, p. pp. 90–105, Feb 2018.

[2] *RF Network-On-Chip (RFNoC^{TM}) Specification*, 1st ed., Ettus Research, Oct. 2020.

[3] B. H. Kirk, R. M. Narayanan, K. A. Gallagher, A. F. Martone, and K. D. Sherbondy, "Avoidance of time-varying radio frequency interference with software-defined cognitive radar," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 55, pp. 1090–1107, Jun. 2019.

[4] R. T. Marler and J. S. Arora, "Survey of multi-objective optimization methods for engineering," *Structural and Multidisciplinary Optimization*, vol. 26, pp. 369–395, Apr. 2004.

[5] I. Kim and O. de Weck, "Adaptive weighted-sum method for bi-objective optimization: Pareto front generation," *Structural and Multidisciplinary Optimization*, vol. 29, pp. 149–158, Feb. 2005.

[6] *USRP™ X300 and X310 X Series*. [Online]. Available: https://www.ettus.com/wp-content/uploads/2019/01/X300_X310_Spec_Sheet.pdf

[7] *X300/X310 - Ettus Knowledge Base*. [Online]. Available: https://kb.ettus.com/X300/X310

[8] *USRP Hardware Driver and USRP Manual*. [Online]. Available: https://files.ettus.com/manual/

[9] *GNU Radio Manual and C++ API Reference*. [Online]. Available: https://www.gnuradio.org/doc/doxygen/

[10] R. Mattingly, "Implementation and analysis of adaptive spectrum sensing," Aug. 2021. [Online]. Available: https://hdl.handle.net/11244/330198

[11] *AXI Reference Guide*, Mar. 2011. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf

[12] R. C. Castillo, "A survey on triangular number, factorial and some associated numbers," *Indian Journal of Science and Technology*, vol. 9, Nov. 2016.

[13] R. T. Marler and J. S. Arora, "The weighted sum method for multi-objective optimization: new insights," *Structural and Multidisciplinary Optimization*, vol. 41, pp. 853–862, Jun. 2010.

[14] S. Paidi, R. Sreerama, and K. Neelima, "A novel high speed leading zero counter for floating point units," *International Journal of Engineering Research and Applications*, vol. 2, pp. 1103–1105, Mar-Apr 2012.

[15] K.-N. Han, S.-W. Han, and E. Yoon, "A new floating-point normalization scheme by bit parallel operation of leading one position value." IEEE, 2002.

[16] I. A. Patil, V. V. Balpande, V. P. Meshram, and I. S. Chintwar, "Implementation of floating point fused basic arithmetic module using verilog." IEEE, Apr. 2015.

[17] A. Zahir, A. Ullah, P. Reviriego, and S. R. U. Hassnain, "Efficient leading zero count (lzc) implementations for xilinx fpgas," *IEEE Embedded Systems Letters*, 2021.

[18] J. Miao and S. Li, "A design for high speed leading-zero counter." IEEE, Nov. 2017.

[19] M. R. Pillmeier, M. J. Schulte, and E. G. W. III, "Design alternatives for barrel shifters," Seattle, WA, Dec 2002.

[20] B. C. Readler, *Verilog by example: A concise introduction for FPGA design*, 1st ed. Full ARC Press, 2011.

[21] M. I. Skolnik, *Introduction to radar systems*, 2nd ed. McGraw-Hill, 1980.

Appendix A

compare_two Module

```
module compare_two ( //Returns the maximum value, row
   index, and column index of two signed 64-bit
   numbers
input wire [63:0] sample_1,
input wire [7:0] bi_1,
input wire [7:0] fj_1,
input wire [63:0] sample_2,
input wire [7:0] bi_2,
input wire [7:0] fj_2,

output wire [63:0] result,
output wire [7:0] bi_result,
output wire [7:0] fj_result
);

assign  result = ($signed(sample_1) > $signed(sample_2)
   ) ? sample_1 : sample_2;
assign bi_result = ($signed(sample_1) > $signed(
   sample_2)) ? bi_1 : bi_2;
assign fj_result = ($signed(sample_1) > $signed(
   sample_2)) ? fj_1 : fj_2;

endmodule
```

Listing A.1: Verilog code for compare_two module

Appendix B

greatest_of_N Module

```
// Compare N 64-bit elements at a time
module greatest_of_N (array, row, max_row, max_col,
   max_val);

parameter N = 256; //Must be a power of 2
parameter NUM_LEVELS = 8; //Should be log2(N) can use
   $clog2(N) to calculate this outside this module

input wire [(N*64)-1:0] array; // Data array.
   Comprised of N 64-bit numbers concatenated into a
   single array of bits

input wire [7:0] row;

output wire [7:0] max_row, max_col;
output wire [63:0] max_val;

genvar i,j;

generate
for (i = 0; i < NUM_LEVELS; i = i + 1) begin :
   wire_loop //Access these wires like so: wire_loop
   [0].values
wire [63:0] values [0:((N>>(i+1))-1)]; //Results from
   the comparison levels. Ex. for 256 valuess there
   will be 128 results
wire [7:0] rows [0:((N>>(i+1))-1)]; //bi indices for
   the comparison results
wire [7:0] cols [0:((N>>(i+1))-1)]; //fj indices for
   the comparison results
end
endgenerate
```

```
generate
for (j=0;j<N;j=j+2) begin :level1_comparators //Level
   1 of comparisons
compare_two cl1 (array[(j*64)+63:j*64],
row, //bi is the row of the matrix
j,  //fj is the column of the matrix, given by j
array[(j+1)*64+63:(j+1)*64],
row,
(j+1),
wire_loop[0].values[j/2],
wire_loop[0].rows[j/2],
wire_loop[0].cols[j/2]
);
end
endgenerate

generate
for (i = 1; i < NUM_LEVELS; i = i + 1) begin :
   levelx_comparators //The rest of the comparison
   levels
for (j = 0; j < (N >> i); j = j + 2) begin :
   inner_loop //N shifted right by i divides by 2 each
    iteration. This is because each comparison level
   yields half as many outputs as inputs
compare_two clx (wire_loop[i-1].values[j],
wire_loop[i-1].rows[j], //bi is the row of the matrix
wire_loop[i-1].cols[j], //fj is the column of the
   matrix, given by i
wire_loop[i-1].values[j+1],
wire_loop[i-1].rows[j+1],
wire_loop[i-1].cols[j+1],
wire_loop[i].values[j/2],
wire_loop[i].rows[j/2],
wire_loop[i].cols[j/2]
);
end
end
endgenerate

assign max_row = wire_loop[NUM_LEVELS-1].rows[0];
assign max_col = wire_loop[NUM_LEVELS-1].cols[0];
assign max_val = wire_loop[NUM_LEVELS-1].values[0];

endmodule
```

Listing B.1: Verilog code for greatest_of_N module

Appendix C

CLZ Module

```
//Calculate leading 0s module
module clz(clk, data_in, reg_numZeros, valid);

input clk;
input [31:0] data_in;

wire [4:0] numZeros;

output wire valid;
output reg [4:0] reg_numZeros;

assign valid = 1;

genvar i;

generate
for (i = 0; i < 4; i = i + 1) begin : muxout_loop
wire [(2 << (i + 1)) - 1:0] muxout;
end
endgenerate

assign muxout_loop[3].muxout = data_in;

generate
for (i = 0; i < 3; i = i + 1) begin :
   assign_muxout_loop
assign muxout_loop[i].muxout = numZeros[i + 2] ?
   muxout_loop[i + 1].muxout[(2 << (i + 1)) - 1:0] :
   muxout_loop[i + 1].muxout[(2 << (i + 2)) - 1:(2 <<
   (i + 1))];
end
endgenerate
```

```
assign numZeros[0] = numZeros[1] ? ~muxout_loop[0].
   muxout[1] : ~muxout_loop[0].muxout[3];

generate
for (i = 1; i < 5; i = i + 1) begin : clz_loop
assign numZeros[i] = (muxout_loop[i - 1].muxout[((2 <<
    i) - 1):(2 << (i - 1))] == 0);
end
endgenerate

always @(posedge clk) begin

reg_numZeros = numZeros; //Synchronize output of this
   module with clock edges

end //End always

endmodule
```

Listing C.1: Verilog code for calculate leading zeros (CLZ) module

Appendix D

WSMO Module

```
module WSMO(
input axis_data_clk,
input axis_data_rst,
input [31:0] m_in_payload_tdata,
input m_in_payload_tlast,
input m_in_payload_tvalid
);

////////////////////////////////////////////////////////////////////
//Begin user logic in NoC Block

//This code is currently only capable of recieveing a
   fixed packet size of 256 samples per packet (SPP)
localparam BANDWIDTH = 100000000; //Full bandwidth in
   Hz, currently 100MHz
localparam DELTA_R = 390625; //Change in frequency per
    sample. DELTA_R = BANDWIDTH/FFT_LENGTH If either
   of these parameters are variable, DELTA_R needs to
   also be variable
localparam FFT_LENGTH = 256; //Incoming FFT length in
   samples
localparam COUNTER_SIZE = 8; //The sample counter
   width in bits. This should be at least log2(
   FFT_LENGTH) or overflow will occurr. Ex: A 256
   point FFT would be log2(256) = 8 bits
localparam SAMP_COUNT_DEFAULT = 0;
localparam REG_FILE_SEL_DEFAULT = 0;


reg reg_file_select = REG_FILE_SEL_DEFAULT; //Selects
   between filling register files A and B. Select 0
   for A and 1 for B
```

```
reg [COUNTER_SIZE-1:0] samp_count = 8'b0; //The number
    of samples (for a given frame of data) that have
   been collected and stored in one of the register
   files

reg [31:0] samples_in_a [0:FFT_LENGTH-1]; //reg file A
   . Currently a 256 element array of 32 bit values
reg [31:0] samples_in_b [0:FFT_LENGTH-1]; //reg file B
   . Currently a 256 element array of 32 bit values

reg [31:0] interference_estimate_a [0:FFT_LENGTH-1][0:
   FFT_LENGTH-1]; //2D array rows=256 cols=256, 32
   bits each
reg [31:0] interference_estimate_b [0:FFT_LENGTH-1][0:
   FFT_LENGTH-1]; //2D array rows=256 cols=256, 32
   bits each

always @(posedge axis_data_clk) begin //Data
   collection always block. Fills the register files
   with incoming samples
if (axis_data_rst) begin
reg_file_select <= REG_FILE_SEL_DEFAULT;
samp_count <= SAMP_COUNT_DEFAULT;
end
else begin //Else #0
if (m_in_payload_tlast) begin //If we reach the end of
    a packet (256 samples), begin filling the other
   register file with the next frame of data
reg_file_select <= ~reg_file_select;
samp_count <= SAMP_COUNT_DEFAULT; //Reached the end of
    a packet (and end of FFT frame with the current
   implementation) so reset the sample counter
end //End if

if (m_in_payload_tvalid) begin
case (reg_file_select)
1'b0 : samples_in_a[samp_count] <= m_in_payload_tdata;
1'b1 : samples_in_b[samp_count] <= m_in_payload_tdata;
endcase
samp_count <= samp_count + 1;
end //End if
end //End else #0
end //End data collection always

/////// Split into real and imaginary data
```

```
// SC16 -> i is real part, upper 16 bits [31:16]
// SC16 -> q is imag part, lower 16 bits [15: 0]

wire [15:0] samples_in_a_real [0:FFT_LENGTH-1];
wire [15:0] samples_in_b_real [0:FFT_LENGTH-1];

wire [15:0] samples_in_a_imag [0:FFT_LENGTH-1];
wire [15:0] samples_in_b_imag [0:FFT_LENGTH-1];

genvar x,y;

generate
for (x = 0; x < FFT_LENGTH; x = x + 1) begin :
   real_loop
assign samples_in_a_real[x] = samples_in_a[x][31:16];
assign samples_in_b_real[x] = samples_in_b[x][31:16];
end
for (y = 0; y < FFT_LENGTH; y = y + 1) begin :
   imag_loop
assign samples_in_a_imag[y] = samples_in_a[y][15:0];
assign samples_in_b_imag[y] = samples_in_b[y][15:0];
end
endgenerate

/////// Interference estimate calculation section

integer o,p,q;

always @(posedge axis_data_clk) begin //Interference
   calculation always block. Stores the result in the
   interference_estimate 2D array
for (o = 0; o < FFT_LENGTH; o = o + 1) begin
interference_estimate_a[0][o] <= samples_in_a_real[o];
    //Load in spectrum samples to the "top" row of the
    adder hierarchy
interference_estimate_b[0][o] <= samples_in_b_real[o];
end

for (p = 1; p < FFT_LENGTH; p = p + 1) begin
for (q = 0; q < FFT_LENGTH - p; q = q + 1) begin
interference_estimate_a[p][q] <=
   interference_estimate_a[p-1][q] +
   interference_estimate_a[0][p+q]; //Interference
   estimate is valid after 2 cycles into the next
   frame.
```

```
interference_estimate_b[p][q] <=
   interference_estimate_b[p-1][q] +
   interference_estimate_b[0][p+q];
end
end
end //End interference estimate calculation always


/////// Subband size calculation section
genvar g;
wire [31:0] subband_size [0:FFT_LENGTH-1]; //The
   second equation in the WSMO algorithm

generate
for (g = 0; g < FFT_LENGTH; g = g + 1) begin :
   subband_size_loop
assign subband_size[g] = (g+1) * DELTA_R;
end
endgenerate

/////// Interference estimate max and leading 1
   section
wire [31:0] max_int_est_a;
assign max_int_est_a = interference_estimate_a[
   FFT_LENGTH-1][0];

wire est_valid_a;
wire [4:0] leading_0s_est_a; //Contains the number of
   leading 0s for the interference estimate maximum
wire [4:0] leading_1_pos_est_a; //Bit position of the
   leading 1 in the interference estimate maximum

assign leading_1_pos_est_a = 31 - leading_0s_est_a;

clz est_lz_counter_a (
.clk (axis_data_clk),
.data_in (max_int_est_a),
.reg_numZeros (leading_0s_est_a),
.valid (est_valid_a)
);


wire [31:0] max_int_est_b;
assign max_int_est_b = interference_estimate_b[
   FFT_LENGTH-1][0];
```

```
wire est_valid_b;
wire [4:0] leading_0s_est_b; //Contains the number of
   leading 0s for the interference estimate maximum
wire [4:0] leading_1_pos_est_b; //Bit position of the
   leading 1 in the interference estimate maximum

assign leading_1_pos_est_b = 31 - leading_0s_est_b;

clz est_lz_counter_b (
.clk (axis_data_clk),
.data_in (max_int_est_b),
.reg_numZeros (leading_0s_est_b),
.valid (est_valid_b)
);


/////// Subband max and leading 1 section

//This is equal to the full bandwidth, defined by the
   parameter BANDWIDTH declared above. Max BW for 32
   bit number is 2^32 ~= 4.295GHz
wire [31:0] max_sub_size;
assign max_sub_size = FFT_LENGTH * DELTA_R;

wire sub_valid;
wire [4:0] leading_0s_sub; //Contains the number of
   leading 0s for the subband maximum
wire [4:0] leading_1_pos_sub; //Bit position of the
   leading 1 in the subband maximum

assign leading_1_pos_sub = 31 - leading_0s_sub;

clz sub_lz_counter (
.clk (axis_data_clk),
.data_in (max_sub_size),
.reg_numZeros (leading_0s_sub),
.valid (sub_valid)
);


///////Shifting section
integer i,j,k,l,m,n,r,s;
reg [4:0] shamt = 0; //Shift amount to make the two
   WSMO exquations of the same order
```

```
reg [31:0] norm_subband_size [0:FFT_LENGTH-1]; //
   Shifted to normalize with interference estimate
reg [31:0] norm_interference_estimate [0:FFT_LENGTH
   -1][0:FFT_LENGTH-1]; //Shifted to normalize with
   subband size

always @(posedge axis_data_clk) begin
case (reg_file_select)
1'b0 : begin //Filling A now. Data on B is valid

if (leading_1_pos_sub < leading_1_pos_est_b) begin //
   Subband is smaller, shift suband
shamt <= leading_1_pos_est_b - leading_1_pos_sub;
if (sub_valid && est_valid_b) begin
for (i = 0; i < FFT_LENGTH; i = i + 1) begin
norm_subband_size[i] <= subband_size[i] << shamt; //
   Shift subband data left by shamt and store in a
   normalized array
for (r = 0; r < FFT_LENGTH; r = r + 1) begin
norm_interference_estimate[i][r] <=
   interference_estimate_b[i][r]; //Store unshifted
   estimate in the normalized array
end
end
end
end
else begin //Estimate is smaller, shift estimate
shamt <= leading_1_pos_sub - leading_1_pos_est_b;
if (sub_valid && est_valid_b) begin
for (j = 0; j < FFT_LENGTH; j = j + 1) begin
norm_subband_size[j] <= subband_size[j]; //Store
   unshifted subband in the normalized array
for (k = 0; k < FFT_LENGTH; k = k + 1) begin
norm_interference_estimate[j][k] <=
   interference_estimate_b[j][k] << shamt; //Shift
   estimate data left and store in a normalized array
end
end
end
end

end //End case 0
1'b1 : begin //Filling B now. Data on A is valid

if (leading_1_pos_sub < leading_1_pos_est_a) begin //
```

```
   Subband is smaller, shift suband
shamt <= leading_1_pos_est_a - leading_1_pos_sub;
if (sub_valid && est_valid_a) begin
for (l = 0; l < FFT_LENGTH; l = l + 1) begin
norm_subband_size[l] <= subband_size[l] << shamt; //
   Shift subband data left by shamt and store in a
   normalized array
for (s = 0; s < FFT_LENGTH; s = s + 1) begin
norm_interference_estimate[l][s] <=
   interference_estimate_a[l][s]; //Store unshifted
   estimate in the normalized array
end
end
end
end
else begin //Estimate is smaller, shift estimate
shamt <= leading_1_pos_sub - leading_1_pos_est_a;
if (sub_valid && est_valid_a) begin
for (m = 0; m < FFT_LENGTH; m = m + 1) begin
norm_subband_size[m] <= subband_size[m]; //Store
   unshifted subband in the normalized array
for (n = 0; n < FFT_LENGTH; n = n + 1) begin
norm_interference_estimate[m][n] <=
   interference_estimate_a[m][n] << shamt; //Shift
   estimate data left and store in a normalized array
end
end
end
end

end //End case 1
endcase
end

///////Adjustment section
reg signed [7:0] adjustment = 8'b0;
reg signed [63:0] diffMax = 64'b0;

wire [63:0] normIntMax;
assign normIntMax = norm_interference_estimate[
   FFT_LENGTH-1][0];
wire [63:0] normSubMax;
assign normSubMax = norm_subband_size[FFT_LENGTH-1];

always @(posedge axis_data_clk) begin //set diffMax,
   the difference between the shifted maximums
if(normIntMax > normSubMax) begin //unusual case,
   interference > bandwidth
diffMax <= normIntMax - normSubMax;
end
else begin //usual case, bandwidth > interference
diffMax <= normSubMax - normIntMax;
end
end

wire normInt_valid;
wire [4:0] leading_0s_normInt; //Contains the number
   of leading 0s for the normalized interference
   estimate
wire [4:0] leading_1_pos_normInt; //Bit position of
   the leading 1 in the normalized interference
   estimate

assign leading_1_pos_normInt = 31 - leading_0s_normInt
   ;

clz normInt_lz_counter (
.clk (axis_data_clk),
.data_in (normIntMax),
.reg_numZeros (leading_0s_normInt),
.valid (normInt_valid)
);

wire diffMax_valid;
wire [4:0] leading_0s_diffMax; //Contains the number
   of leading 0s for the difference in the shifted
   maximums
wire [4:0] leading_1_pos_diffMax; //Bit position of
   the leading 1 in the difference in the shifted
   maximums

assign leading_1_pos_diffMax = 31 - leading_0s_diffMax
   ;

clz diffMax_lz_counter (
.clk (axis_data_clk),
.data_in (diffMax),
.reg_numZeros (leading_0s_diffMax),
.valid (diffMax_valid)
);
```

```
wire [4:0] diffLeading1s = leading_1_pos_normInt -
   leading_1_pos_diffMax;

always @(posedge axis_data_clk) begin
if(diffLeading1s < 4) begin //difference in maximums
   is large
if(normIntMax > normSubMax) begin //unusual case Z1 (
   interference) > Z2 (bandwidth)
adjustment <= -8*(4 - diffLeading1s);
end
else begin //usual case, Z2 (bandwidth) > Z1 (
   interference)
adjustment <= 8*(4 - diffLeading1s);
end
end
else begin
adjustment <= 0;
end
end

wire [7:0] adjustedAlpha;
wire [7:0] adjustedAlpha_c;

assign adjustedAlpha = alpha + adjustment;
assign adjustedAlpha_c = 255 - adjustedAlpha;

///////Multiplication by alpha and addition to create
   the full weighted sum function
wire [63:0] Z1 [0:FFT_LENGTH-1][0:FFT_LENGTH-1];
wire [63:0] Z2 [0:FFT_LENGTH-1];

wire [7:0] alpha;
wire [7:0] alpha_c;

//In a practical application using RFNoC, alpha would
   be defined by a user register
assign alpha = 8'd128; //This is equivalent to alpha =
    0.5 The scale for alpha is 0 to 255 instead of 0
   to 1
assign alpha_c = 255 - alpha; //This is equivalent to
   1 - alpha

genvar a, b, c;
```

```
generate
for (b = 0; b < FFT_LENGTH; b = b + 1) begin :
   Z1_outer_loop
for (c = 0; c < FFT_LENGTH; c = c + 1) begin :
   Z1_inner_loop
assign Z1[b][c] = adjustedAlpha*
   norm_interference_estimate[b][c];
end
end
for (a = 0; a < FFT_LENGTH; a = a + 1) begin : Z2_loop
assign Z2[a] = adjustedAlpha_c*norm_subband_size[a];
end
endgenerate

localparam MIN_64_BIT = 64'h8000000000000000; //Most
   negative possible number for a 64-bit value

wire [63:0] Z [0:FFT_LENGTH-1][0:FFT_LENGTH-1]; //Full
    weighted sum function

genvar d, e;

generate
for (d = 0; d < FFT_LENGTH; d = d + 1) begin :
   Z_outer_loop
for (e = 0; e < FFT_LENGTH - d; e = e + 1) begin :
   Z_inner_loop
assign Z[d][e] = Z2[d] - Z1[d][e];
end
end
endgenerate

generate
for (d = 1; d < FFT_LENGTH; d = d + 1) begin :
   Z_pad_outer_loop
for (e = 255; e > FFT_LENGTH - d - 1; e = e - 1) begin
    : Z_pad_inner_loop
assign Z[d][e] = MIN_64_BIT; //Set the unused entries
   to the most negative possible number for a 64-bit
   value
end
end
endgenerate

///////Find the max of Z
```

```
wire [7:0] max_rows [0:FFT_LENGTH-1];
wire [7:0] max_cols [0:FFT_LENGTH-1];
wire [63:0] max_vals [0:FFT_LENGTH-1];
wire [(FFT_LENGTH*64)-1:0] final_array;
wire [(FFT_LENGTH*64)-1:0] array [0:FFT_LENGTH-1];

generate //Build concatenated arrays
for (d = 0; d < FFT_LENGTH; d = d + 1) begin :
   array_row_loop
for (e = 0; e < FFT_LENGTH; e = e + 1) begin :
   array_column_loop
assign array[d][(e*64)+63:e*64] = Z[d][e];
end
end
endgenerate

generate
for(d = 0; d < FFT_LENGTH; d = d + 1) begin :
   find_max_loop
greatest_of_N #(.N(FFT_LENGTH), .NUM_LEVELS($clog2(
   FFT_LENGTH))) row(
.array(array[d]),
.row(d),
.max_row(max_rows[d]),
.max_col(max_cols[d]),
.max_val(max_vals[d])
);
end
endgenerate

generate
for (e = 0; e < FFT_LENGTH; e = e + 1) begin :
   array_column_loop
assign final_array[(e*64)+63:e*64] = max_vals[e]; //
   Array containing the max value from each row
end
endgenerate

wire [7:0] index, bi, fj;
wire [63:0] final_max_val;

greatest_of_N #(.N(FFT_LENGTH), .NUM_LEVELS($clog2(
   FFT_LENGTH))) final_comparison(
.array(final_array),
.row(8'b0),
.max_row(), //This is meaningless for the final
   comparison
.max_col(index),
.max_val(final_max_val)
);

//Final results
assign bi = max_rows[index];
assign fj = max_cols[index];

endmodule // rfnoc_block_WSMO
```
Listing D.1: Verilog code for the main WSMO module