FPGA implementation of Digital Communication Techniques

A. I. Generator

September 25, 2025

Contents

1 Chapter 1: Implementation of Automatic Gain Control (AGC)

In any modern digital communication receiver, a fundamental and critical block is the **Automatic Gain Control (AGC)** loop. Its primary function is to maintain a stable and consistent signal power level at the input of the receiver's demodulation stage, despite large and often rapid fluctuations in the received signal strength. These fluctuations can be caused by a variety of factors in the wireless channel, including path loss, fading, shadowing from obstacles, and changes in the distance between the transmitter and receiver.

Without a robust AGC, a demodulator would be unable to operate effectively. If the signal is too weak, it becomes buried in receiver noise, leading to a significant increase in the bit error rate (BER). Conversely, if the signal is too strong, it can overwhelm the analog-to-digital converter (ADC), causing signal clipping and distortion, which corrupts the data and makes accurate decoding impossible. An AGC loop acts as a dynamic preprocessor, ensuring that the signal presented to the digital processing chain always has a predictable amplitude.

A typical digital AGC loop consists of three main functional components:

1. **Power Detector:** Measures the instantaneous power of the received signal.
2. **Controller:** Compares the measured power to a predefined target power level and computes the necessary gain adjustment.
3. **Gain Element:** Applies the calculated gain to the signal. This is typically a digital multiplier.

1.1 AGC Principles and Operation

1.2 What is AGC?

Automatic Gain Control (AGC) is a closed-loop signal processing system designed to automatically regulate the amplitude of an input signal, maintaining it within a specified range regardless of variations in input level. It plays a critical role in communication receivers by stabilizing signal levels before further processing stages such as analog-to-digital conversion (ADC), carrier tracking, and symbol detection. By adjusting the receiver gain dynamically, AGC prevents signal clipping, reduces quantization noise, and maintains optimal operating conditions across the signal chain. This control mechanism is essential in systems affected by multipath fading, Doppler shifts, or varying transmission powers, such as wireless communication receivers, radar systems, and SDR platforms.

1.3 How AGC Works

AGC operates by continuously measuring the amplitude (envelope or RMS) of the incoming signal and comparing it with a predefined reference level. Based on this comparison, an error signal is generated that indicates whether the gain should be increased or decreased. This error signal is then used to update the gain control variable, which is applied to the input signal. The process forms a feedback loop that iteratively drives the output amplitude toward the desired reference. In practice, this is done either linearly or in the

logarithmic domain, depending on convergence speed and precision requirements. AGC tracks and compensates for time-varying signal conditions, ensuring consistent amplitude levels for optimal demodulation and detection.

1.4 Operational Principles

In principle, an AGC is a feedback control system that drives the amplitude error to zero in an iterative fashion. This establishes, on average, a constant signal amplitude at the start of the DSP chain. To see how this is accomplished, consider the block diagram below.

1.5 Importance and Objectives of Digital AGC

1.6 Project Rationale

The incorporation of a Digital Automatic Gain Control (AGC) system into the Integrated Tracking System (ITS) introduces significant advantages over conventional analog AGC architectures. Traditional analog AGC circuits lack flexibility, require manual tuning for each operational condition, and incur higher costs due to the use of discrete analog components. In contrast, digital AGC implemented via FPGA or SoC platforms enables dynamic reconfiguration of gain control parameters, such as reference amplitude, gain slope, and offset, directly through software or remote interfaces. This programmability is particularly beneficial in ITS, where AGC thresholds and gains must be precisely tuned across SR, XR, and XL bands to ensure robust tracking performance.

1.7 Project Objectives

The following table outlines the key objectives for the design and implementation of the Digital AGC system.

S. No.	**Objective**	**Details**
1	Input Signal Handling	Design the AGC to accept input power levels ranging from -10 dBm to -50 dBm.
2	Output Power Regulation	Maintain a constant output power level of -20 dBm, regardless of input variations.
3	Dynamic Range Achievement	Achieve a minimum dynamic range of 40 dB to handle a wide span of input signal amplitudes.
4	Algorithm Simulation	Simulate and verify the AGC control algorithm in MATLAB/Simulink prior to implementation.
5	Hardware Implementation	Implement the AGC algorithm on an FPGA using Vivado and Vitis tools.
6	Performance Evaluation	Evaluate gain settling time, output stability, and dynamic behaviour under varying inputs.

Table 1: Key objectives for the Digital AGC project.

1.8 AGC Control Law and Loop Dynamics

The core of the AGC loop is its control law, which determines how quickly and how accurately the system responds to power fluctuations. The goal is to adjust the gain, G_k, for the k-th received sample such that the output power, P_{out}, is held constant at a target level, P_{target}. For a complex received baseband signal, $r_k = I_k + jQ_k$, the instantaneous power is:

$$P_k = |r_k|^2 = I_k^2 + Q_k^2$$

A common and effective control law is to use a proportional-integral (PI) controller to adjust the gain. The error signal, e_k, is the difference between the measured power and the target power (often in a logarithmic scale, e.g., dB).

$$e_k = \log(P_{target}) - \log(P_k)$$

The gain is then updated based on this error:

$$G_{k+1} = G_k + K_p e_k + K_i \sum_{i=0}^{k} e_i$$

where K_p and K_i are the proportional and integral gain coefficients. These coefficients are carefully chosen to balance the loop's speed and stability. A higher gain (larger K_p, K_i) results in a faster response but can lead to oscillations and instability, especially in noisy channels. A lower gain makes the loop more stable but slower to settle. The trade-off between speed and stability is known as the **AGC loop bandwidth**. A narrow bandwidth ensures stability and filters out noise but has a longer **settling time**, which is the time it takes for the gain to stabilize after a sudden power change.

1.9 Power Detection with CORDIC Square Root

The most computationally intensive part of the AGC power detection is calculating the signal's magnitude, $|r_k| = \sqrt{I_k^2 + Q_k^2}$, which is then used to determine the necessary gain. While a general-purpose processor can handle this with a floating-point unit, real-time hardware implementations, such as those on an RFSoC, demand a more efficient and low-power approach. The **CO**ordinate **R**otation **DI**gital **C**omputer (CORDIC) algorithm provides an elegant solution that is perfectly suited for a hardware implementation.

1.10 The CORDIC Algorithm

CORDIC is an iterative algorithm that can perform a variety of trigonometric and hyperbolic functions using only simple shifts and additions. It works by progressively rotating a vector in a series of small, pre-defined angles until it reaches a desired orientation. For AGC power detection, we are interested in the **vectoring mode** of CORDIC, which finds the magnitude and phase of a given complex number. The algorithm is initialized with the coordinates of the received complex sample, (I_k, Q_k).

The algorithm proceeds through a series of N micro-rotations. In each iteration i, a rotation of angle $\alpha_i = \arctan(2^{-i})$ is performed. The direction of rotation, σ_i, is chosen to move the vector closer to the real axis (the x-axis).

The iterative equations for the vectoring mode are as follows:

$$x_{i+1} = x_i - \sigma_i y_i 2^{-i}$$
$$y_{i+1} = y_i + \sigma_i x_i 2^{-i}$$
$$z_{i+1} = z_i - \sigma_i \alpha_i$$

where:

- (x_i, y_i) are the coordinates of the vector at iteration i.
- z_i is the accumulated phase angle.
- $\sigma_i = +1$ if $y_i < 0$ and $\sigma_i = -1$ if $y_i \geq 0$ (This choice ensures the vector rotates towards the x-axis).
- 2^{-i} represents a simple hardware bit shift, which is extremely efficient to implement.

After a sufficient number of iterations (N), the final values converge:

$$x_N \approx K\sqrt{I_k^2 + Q_k^2}$$
$$y_N \approx 0$$
$$z_N \approx \text{phase}(I_k, Q_k)$$

where K is a constant gain factor introduced by the rotations, which can be easily compensated for. The final x_N value is a very accurate approximation of the signal's magnitude, which is directly related to the square root of the power.

1.11 Implementation on an ZYNQ7000 board

On a reconfigurable hardware platform like a Zedboard ZYNQ7000, the CORDIC algorithm is ideal for implementation within the programmable logic (FPGA fabric). Its structure of simple shifts and adds can be mapped directly to hardware, creating a highly efficient, pipelined circuit that can perform the square root operation in a single clock cycle after the initial latency. This eliminates the need for a power-hungry digital signal processor (DSP) or complex multiplication circuits, making the AGC loop extremely fast and power-efficient. The output of the CORDIC block, representing the signal magnitude, is then fed into a gain control look-up table or a digital multiplier to apply the correct gain to the incoming data stream, completing the AGC loop.

1.12 Conclusion

The AGC loop is an essential component of any robust digital communication receiver. It ensures that the signal presented for demodulation has a consistent power level, protecting the system from noise and saturation. By leveraging the hardware-friendly **CORDIC algorithm**, the computationally expensive square root operation required for power detection can be performed with minimal hardware resources and power consumption. This makes AGC systems, particularly those implemented on platforms like RFSoCs, incredibly efficient and well-suited for high-performance, real-time communication applications.

2 Chapter 2: Phase Ambiguity Resolution: Differential and Unique Word Methods

2.1 Introduction to Phase Ambiguity

In digital communication systems, a fundamental challenge is **phase ambiguity**. This occurs when a receiver cannot determine the true absolute phase of a received signal, leading to potential decoding errors. This problem arises from various factors inherent to the communication channel and hardware. A primary cause is the arbitrary initial phase of the transmitter's oscillator, which is not synchronized to the receiver's local oscillator. Additionally, fixed but unknown phase shifts can be introduced by the physical channel itself (e.g., multipath propagation) and various analog hardware components within the RF front-end. This static but unpredictable phase offset must be resolved for a receiver to correctly demodulate data and prevent symbol-to-bit mapping errors.

2.2 QPSK Symbol Representation

Quadrature Phase-Shift Keying (QPSK) is a widely used modulation scheme where data is encoded by modulating the phase of a carrier signal. Each symbol represents two bits of data, with the four possible phases typically defined as $0, \pi/2, \pi$, and $3\pi/2$ radians. A transmitted QPSK symbol can be represented in the complex plane as a phasor:

$$s_k = Ae^{j\phi_k} = A(\cos\phi_k + j\sin\phi_k)$$

where s_k is the k-th symbol, A is the constant amplitude, and ϕ_k is the phase representing the encoded data. This can also be expressed in terms of its in-phase (I) and quadrature (Q) components:

$$s_k = I_k + jQ_k$$

The received signal, r_k, is the transmitted symbol, s_k, corrupted by channel noise and a fixed, unknown phase offset, θ:

$$r_k = s_k e^{j\theta} = Ae^{j(\phi_k+\theta)}$$

The receiver's primary task is to recover the original data phase ϕ_k from the received signal r_k, despite the presence of the unknown offset θ.

2.3 The Differential Method: DQPSK

The differential method, formally known as Differential Quadrature Phase-Shift Keying (DQPSK), is an elegant solution that sidesteps the problem of absolute phase altogether. Instead of relying on a stable phase reference, it encodes information in the **change in phase** between consecutive symbols.

2.4 Principle of Operation

At the transmitter, a stream of input bits (b_k, b_{k+1}) is mapped to a phase shift, $\Delta\phi_k$, from the previous symbol's phase, ϕ_{k-1}. A common mapping is given in Table 2.

Input Bits (b_k, b_{k+1})	Phase Shift ($\Delta\phi_k$)
00	0°
01	90°
11	180°
10	270°

Table 2: Common DQPSK data-to-phase mapping.

The phase of the current symbol, ϕ_k, is calculated relative to the previous symbol's phase:

$$\phi_k = \phi_{k-1} + \Delta\phi_k$$

The first symbol is often initialized with a known starting phase, e.g., $\phi_0 = 0°$. The constellation points for DQPSK are the same as for QPSK, but the demodulation logic is entirely different.

2.5 Demodulation and Mathematical Proof

At the receiver, the demodulator does not attempt to find the absolute phase of the signal. Instead, it measures the phase difference between consecutive received symbols. The received symbols are: $r_k = Ae^{j(\phi_k+\theta)}$ and $r_{k-1} = Ae^{j(\phi_{k-1}+\theta)}$.

The crucial step in demodulation is to calculate the complex ratio of the current symbol to the previous symbol. This operation naturally cancels out the unknown, fixed phase offset, θ:

$$\frac{r_k}{r_{k-1}} = \frac{Ae^{j(\phi_k+\theta)}}{Ae^{j(\phi_{k-1}+\theta)}} = \frac{e^{j\phi_k}e^{j\theta}}{e^{j\phi_{k-1}}e^{j\theta}} = e^{j(\phi_k-\phi_{k-1})} = e^{j\Delta\phi_k}$$

The resulting complex value has an angle equal to the original phase shift $\Delta\phi_k$. The receiver can then map this angle back to the original data bits using the inverse of Table 2.

2.6 Advantages and Disadvantages

Advantages

- **Simplicity:** DQPSK avoids the need for a complex carrier recovery phase-locked loop (PLL) circuit to track the absolute phase. This simplifies the receiver design and reduces costs.
- **Robustness to Phase Jumps:** It is inherently robust to phase jumps or step changes that can occur in the channel, as long as the change is static between adjacent symbols.

Disadvantages

- **Error Propagation:** This is the most significant drawback. A single decoding error at symbol k means that the demodulator uses an incorrect phase reference for symbol $k+1$, leading to a high probability of decoding error for that symbol as well. A single bit error can thus lead to a burst of errors.

- **Noise Performance:** DQPSK typically requires a higher Signal-to-Noise Ratio (SNR) for the same Bit Error Rate (BER) compared to coherent QPSK, due to the demodulation process involving two noisy symbols.

2.7 The Unique Word Method

The unique word (UW) method is a frame-based synchronization approach that sacrifices a small amount of data bandwidth to embed a known, pre-defined bit sequence for robust phase resolution.

2.8 Principle of Operation

The UW is a specific, non-repeating pattern with good auto-correlation properties. It is inserted at the beginning of each transmission frame. At the receiver, the primary task is to detect the unique word and simultaneously determine the phase offset, θ.

The receiver performs a **cross-correlation** or **matched filtering** operation on the incoming data stream to detect the unique word. For a received data stream $r[n]$ and a known unique word sequence $w[n]$ of length L, the receiver computes a correlation function at each possible phase rotation.

For example, the correlation for a 0° rotation at a given time index n is:

$$R_{0^\circ}[n] = \sum_{k=0}^{L-1} r_{n+k} \cdot (w_k)^*$$

To find the correct phase, the receiver performs this same calculation for all four possible rotations $(0^\circ, 90^\circ, 180^\circ, 270^\circ)$ and compares the correlation scores. The rotation with the highest correlation score indicates not only the presence of the unique word but also the correct absolute phase of the entire data frame.

$$\text{Correct Rotation} = \underset{\theta \in \{0, \pi/2, \pi, 3\pi/2\}}{\operatorname{argmax}} \left| \sum_{k=0}^{L-1} r_{n+k} \cdot (w_k e^{-j\theta})^* \right|$$

Once the unique word is found, the receiver knows the absolute phase of that sequence, which it then uses as a reference for all subsequent data symbols in that frame. The detected unique word effectively acts as a pilot signal to resolve the phase ambiguity.

2.9 Advantages and Disadvantages

Advantages

- **Robustness to Burst Errors:** The UW method is highly reliable and prevents error propagation. Because the phase reference is re-established with every new frame, a decoding error within a frame does not carry over to the next.
- **Higher Spectral Efficiency (for long packets):** For long data packets, the overhead of the UW is a small percentage of the total data. This can make it more spectrally efficient than using a continuously tracked phase reference.

Disadvantages

- **Overhead:** The unique word represents a non-data overhead, which reduces the overall spectral efficiency of the communication link. This is particularly noticeable with short packets.
- **Increased Complexity:** The receiver requires a dedicated correlator or matched filter to continuously search for the unique word, increasing implementation complexity compared to DQPSK.

2.10 Comparative Analysis

The choice between the differential and unique word methods depends heavily on the application's specific requirements. A direct comparison across key metrics highlights their trade-offs.

Feature	**Differential Method (DQPSK)**	**Unique Word Method (UW)**
Implementation	Low complexity (no PLL needed)	High complexity (requires correlator)
Robustness	Susceptible to error propagation	Highly robust to burst errors
Spectral Efficiency	High (no overhead)	Dependent on packet length (overhead)
Application	Simpler, low-cost systems	High-reliability, burst-oriented systems (e.g., WLAN)
Phase Reference	Relative (symbol-to-symbol)	Absolute (frame-based)

Table 3: Comparison of Differential and Unique Word methods.

2.11 Application in RFSoC Loopback Systems

The methods described are particularly relevant in a test environment such as an RFSoC (Radio Frequency System-on-Chip) loopback setup. An RFSoC integrates high-speed data converters and programmable logic, allowing for a complete digital communication system to be implemented on a single chip.

In a loopback test, the data is transmitted from the RF-DAC and immediately received by the RF-ADC on the same chip. While the environment is controlled, phase ambiguity still arises due to the uncalibrated phase relationship between the transmit and receive direct-digital synthesizers (DDS) and mixers.

A practical implementation on an RFSoC would involve the following:

1. **DQPSK Implementation:** The digital baseband processing within the programmable logic would implement the differential encoder and a simple differential demodulator. This approach would be suitable for initial testing to confirm basic connectivity and functionality, as it requires minimal logic resources.
2. **UW Implementation:** For a more rigorous test, a UW sequence would be prepended to the data. The receiver logic would include a correlator block to continu-

ously search for the UW. The correlator output would then drive a phase-correction engine to align the constellation and enable correct symbol-to-bit decoding. This method would provide a precise measure of the loopback channel's phase response and overall BER performance.

2.12 Conclusion

Phase ambiguity is a universal problem in digital communications, but it can be effectively managed through various techniques. The **Differential Method (DQPSK)** provides a simple, low-complexity solution by encoding information in the relative phase shifts between symbols. Its primary advantage is its simplicity and lack of overhead, but this comes at the cost of vulnerability to error propagation.

Conversely, the **Unique Word Method (UW)** offers a highly reliable solution by using a known sequence to re-establish an absolute phase reference for each frame. While it introduces a spectral efficiency penalty due to overhead, its robustness against burst errors and channel impairments makes it the preferred choice for modern communication standards. The selection of the appropriate method is a critical design decision based on a careful trade-off analysis of implementation cost, spectral efficiency, and link reliability.

3 Chapter 3: Symbol Synchronization

3.1 The Importance of Symbol Synchronization

Symbol synchronization, or timing recovery, is arguably the most critical function in a digital communications receiver. Its fundamental purpose is to precisely align the receiver's local clock with the symbol boundaries of the incoming signal. This meticulous alignment is paramount because it ensures that data is sampled at the precise moment of **maximum signal energy**, where the signal-to-noise ratio (SNR) is highest and the signal is least susceptible to corruption. The successful execution of this task is essential for minimizing **Inter-Symbol Interference (ISI)**, a pervasive form of distortion that can critically degrade a system's performance.

ISI arises from the non-ideal characteristics of the communication channel and the filtering applied during pulse shaping. These imperfections cause symbols to "spread" in the time domain, leading to the "tail" of one symbol overlapping with the "main energy" of a subsequent symbol. This overlap distorts the signal and can make it difficult, if not impossible, for the receiver to correctly identify the transmitted data. The impact of ISI is most effectively visualized through an **eye diagram**. In this visual representation, a wide-open eye signifies minimal ISI, allowing for straightforward and accurate sampling. Conversely, a closed or severely distorted eye indicates the presence of significant ISI, making it nearly impossible for the receiver to distinguish between distinct symbols. The symbol synchronizer's role is to ensure the receiver's clock samples the signal exactly in the center of the eye, which corresponds to the optimal sampling point. This action maximizes the system's robustness against noise and ensures reliable data recovery. This entire process is a self-correcting, closed-loop system, wherein a timing error detector measures the deviation from the ideal sampling point, and a control loop continuously adjusts the sampling clock to correct the detected error. This feedback mechanism creates a robust system capable of tracking minor drifts between the transmitter and receiver clocks, which can be caused by ambient temperature changes, component aging, or other physical factors.

The consequences of a failure to achieve and maintain robust symbol synchronization are severe and far-reaching. A small timing error, even a fraction of a symbol period, can lead to a significant increase in the **Bit Error Rate (BER)**. This happens because sampling off-center in the "eye" places the receiver in a more vulnerable position, where the margin for distinguishing a '1' from a '0' is reduced. A large or persistent timing error can lead to a complete loss of communication, as the receiver's demodulator becomes unable to make a single correct decision. The closed-loop nature of the synchronizer is its most powerful attribute. The system continuously seeks to minimize the detected error, creating a dynamic equilibrium where the local clock is in constant, subtle adjustment to match the incoming signal's timing. This makes it resilient not only to static clock offsets but also to dynamic changes like phase jitter and frequency drift, which are common in real-world wireless channels. The synchronizer, therefore, acts as a primary defensive layer, a prerequisite for the successful operation of all subsequent stages in the digital receiver chain.

3.2 ZCTED-Based Timing Error Detection

The **Zero-Crossing Timing Error Detector (ZCTED)** is a non-data-aided synchronization algorithm. Its classification as a feedforward timing error detector signifies that

its operation is independent of any knowledge of the transmitted data. Instead, it relies on the statistical properties and inherent structure of the incoming signal. This "blind" nature is a highly desirable attribute for a practical communications receiver, as it allows for synchronization before data can be reliably decoded.

The core principle of the ZCTED is to identify and utilize the zero-crossing points of the matched-filtered signal. In an ideal digital communication system, the signal is shaped using a raised-cosine filter, and its output has a predictable zero-crossing at every half-symbol interval. The detector measures the timing offset by analyzing the signal at or around these zero-crossing points, where the signal's energy is at its minimum. This is in contrast to the sampling point where the energy is at its maximum.

The mathematical formulation for the timing error signal, denoted as $\epsilon(k)$, is derived from a specific operation on the signal samples. A common and robust implementation involves multiplying a symbol-spaced sample by the signal's derivative at the same point. The derivative of the signal is at its maximum absolute value precisely at the zero-crossings of the ideal raised-cosine pulse. Therefore, the product of the signal and its derivative provides a powerful and robust error signal. The error signal at time k can be abstractly represented as:

$$\epsilon(k) = v_k \cdot \frac{dv_k}{dt}$$

In a practical digital implementation on a Field-Programmable Gate Array (FPGA), the derivative is approximated using a simple difference between adjacent samples. For a signal that is oversampled at a rate $f_s = m \cdot R_s$, where R_s is the symbol rate and m is the oversampling factor, the error can be calculated using the in-phase and quadrature components of the signal, $I(k)$ and $Q(k)$, as:

$$\epsilon(k) = I(k) \cdot (I(k+1) - I(k-1)) + Q(k) \cdot (Q(k+1) - Q(k-1))$$

A positive error value indicates that the sampling instant is too late, while a negative error indicates it is too early. One significant advantage of the ZCTED is its robust performance in the presence of noise and its relatively simple implementation in hardware. However, it can exhibit sensitivity to large clock frequency offsets and may therefore require a prior coarse frequency acquisition stage to ensure reliable initial lock. A key reason for using the zero-crossing method is that the slope of the raised-cosine pulse is steepest at its zero-crossing points. This steep slope provides a highly sensitive indicator of timing offset; even a small timing deviation from the ideal zero-crossing results in a significant change in the error signal, making the detector highly responsive. This contrasts with data-aided algorithms, such as the **Early-Late Gate (ELG)**, which require knowledge of the transmitted data symbols. While ELG can offer excellent performance, it necessitates a prior data-decoding stage, which is not always feasible during the initial synchronization phase. The ZCTED's ability to operate blindly makes it a preferred choice for the initial, coarse acquisition of timing.

3.3 Piecewise Parabolic Interpolation (PPI)

Once the ZCTED provides a timing error signal, the receiver must resample the signal at a new, more accurate time point. This crucial task is performed by the **interpolator**. The interpolator estimates the signal's value at fractional time points based on the available integer-spaced samples. The **Piecewise Parabolic Interpolator (PPI)** is a

highly effective method that offers superior accuracy compared to a simpler linear interpolator. A linear interpolator, which simply draws a straight line between two samples, can introduce significant errors, especially when the signal's curvature is pronounced.

The PPI algorithm uses three adjacent samples to fit a parabola. This parabola serves as a localized, high-fidelity model of the pulse shape. The function for the output of the parabolic interpolator, based on samples x_0, x_1, x_2, can be written as:

$$\hat{x}(t) = c_0 + c_1(t - t_1) + c_2(t - t_1)^2$$

where t is the fractional timing offset, and c_0, c_1, c_2 are coefficients calculated from the input samples. The coefficients are determined as follows:

$$c_0 = x_1$$

$$c_1 = \frac{x_2 - x_0}{2}$$

$$c_2 = \frac{x_2 - 2x_1 + x_0}{2}$$

The output of the interpolator, $\hat{x}(t)$, is the perfectly timed sample, which is then passed to the demodulator for subsequent processing. The PPI's ability to model the curvature of the pulse shape provides a significantly more accurate estimate, thereby reducing the residual ISI and improving the overall system performance. This higher accuracy comes at a slight increase in computational complexity compared to a linear interpolator, but the performance gains typically justify the added hardware cost. A linear interpolator, for instance, only requires two samples and a simple weighted average, but its accuracy is limited. More complex methods, like cubic splines, offer even greater accuracy by using four or more samples, but at the cost of significantly increased computational complexity and hardware resources. The PPI represents a practical and optimal compromise, providing a robust solution that is both highly accurate and hardware-efficient.

3.4 The PI Filter and NCO Control Loop

The symbol synchronizer functions as a classic **feedback control loop**, a concept fundamental to all systems that require self-correction. The loop's components work in concert to constantly adjust the sampling time to track any drift or jitter.

- **PI (Proportional-Integral) Filter:** The **PI filter** is the "brain" of the control loop. It takes the timing error, $\epsilon(k)$, from the ZCTED and calculates the necessary correction. The filter's output is composed of two distinct terms:
  - The **proportional (P)** term applies an instantaneous correction that is directly proportional to the current error. This term provides a rapid response to sudden changes in timing. The proportional gain, K_p, determines the aggressiveness of this response. A high K_p can lead to a faster acquisition time but may also cause overshoot and instability.

– The **integral (I)** term accumulates the error over time. The integral gain, K_i, determines how quickly this accumulated error is used to correct for any steady-state offset. This term is crucial for eliminating small, persistent timing errors that the proportional term might not fully correct, ensuring that the loop eventually settles with a zero-error state. The PI filter's transfer function is:

$$H(z) = K_p + \frac{K_i}{1 - z^{-1}} \tag{1}$$

- **Numerically Controlled Oscillator (NCO):** The **NCO** is the digital equivalent of a voltage-controlled oscillator. It is a digital circuit that generates a clock signal with a precisely controlled frequency and phase. It takes the output of the PI filter as its control input. A positive control signal from the PI filter increases the NCO's output frequency, effectively advancing the sampling time, while a negative signal decreases it. The NCO operates by incrementing an internal phase accumulator. The output of the PI filter determines the size of the increment for each clock cycle. This mechanism closes the loop, continuously adjusting the sampling phase to nullify the timing error. The resolution and accuracy of the timing adjustment are directly determined by the bit-width of the NCO's phase accumulator.

The combination of the PI filter and NCO provides a robust and stable control mechanism that locks onto the correct symbol timing. The loop's performance, specifically its acquisition speed and stability, is governed by its bandwidth and damping factor, which are meticulously controlled by the chosen values of K_p and K_i. The values of K_p and K_i are selected to achieve a **critically damped** response, which provides the fastest possible lock time without excessive overshoot or oscillation. An **underdamped** system (high K_p) will acquire lock faster but may suffer from "hunting" or oscillation around the correct timing. Conversely, an **overdamped** system (low K_p) will be stable but will take an excessively long time to acquire lock, which is undesirable in a communications link.

The NCO's operation is central to the entire process. It can be conceptualized as a fractional-bit counter. The integer part of the NCO's phase accumulator determines the output sample index, while the fractional part determines the fractional timing offset for the interpolator. For example, a 32-bit phase accumulator provides a timing resolution of 2^{-32} of a symbol period, allowing for extremely precise timing adjustments. The NCO's rate is controlled by a value added to its phase accumulator each clock cycle; this value is the output of the PI filter. This allows the NCO to generate a clock signal that is not only a fraction of the system's master clock but also has a dynamically adjustable phase, perfectly suited for tracking the incoming signal.

3.5 Zero-Stuffing and Rate Adaptation

One of the practical challenges in symbol synchronization is managing the variable output rate from the interpolator. If the receiver's master clock is slightly different from the transmitter's, the NCO must continuously adjust its phase, which can result in an inconsistent output data stream.

Zero-stuffing is a technique used to maintain a constant output rate from the synchronizer. It is a form of digital-to-digital rate conversion. When the NCO needs to "slow down" the sampling to compensate for a frequency offset, a "dummy" zero sample

is inserted or "stuffed" into the data stream. Conversely, when the NCO needs to "speed up," a sample is "unstuffed" or removed. This ensures that the output data stream maintains a uniform rate, which is a requirement for downstream processing blocks. This is particularly important for decoders and other blocks that require a continuous, symbol-synchronous input stream. The zero-stuffing logic is often a simple multiplexer controlled by the NCO's overflow signal. A zero-stuffing scheme is computationally less demanding than a full-fledged sample-rate converter and is well-suited for hardware implementation. The zero-stuffing mechanism operates as follows: the NCO's phase accumulator accumulates the timing correction. When the accumulator's value overflows a certain threshold (typically its full scale), it indicates that a new sample is ready. The overflow signal triggers the interpolator to produce a new sample and also controls a simple rate-matching finite state machine. If the NCO's rate is faster than the symbol rate, the state machine may occasionally "skip" an output cycle, effectively unstuffing a sample. If the NCO is slower, it might "add" a cycle with a zero or null value. This simple mechanism is a powerful way to bridge the gap between the receiver's high-speed clock domain and the variable-rate timing of the incoming symbols, all while avoiding the complexity and high resource cost of a full polyphase filter bank or other complex resampling structures.

3.6 Full Block Diagram and Implementation

A complete symbol synchronization module integrates all the components discussed above. A typical hardware implementation on an FPGA would consist of the following carefully designed blocks:

- **Input Buffer:** A small First-In, First-Out (FIFO) buffer or register bank to hold the incoming matched-filtered I and Q samples. This buffer decouples the high-speed input sample rate from the potentially variable processing rate of the synchronizer logic. The buffer's size is a key design parameter, chosen to accommodate the worst-case timing jitter and frequency offset without overflowing or underflowing.
- **ZCTED Module:** A combinational logic block that performs the multiplication and subtraction operations to calculate the timing error. This block must be designed for maximum throughput, often using pipelining to break down the calculation into smaller, faster stages.
- **PI Filter Module:** A pipelined block that performs the addition and multiplication for the proportional and integral terms. Pipelining is essential for achieving high-speed clock rates. The filter uses fixed-point arithmetic, and the number of bits for each signal and coefficient is meticulously selected to maintain accuracy while minimizing hardware resources.
- **NCO Module:** A counter with a variable step size, implemented with a high-resolution phase accumulator. The bit-width of this accumulator is a key design parameter that trades off timing resolution against hardware resource utilization.
- **Interpolator Module:** A pipelined block that takes the input samples and the NCO output to perform the piecewise parabolic interpolation. The interpolator's internal logic must be optimized for fast operation, and it often employs look-up tables or pre-computed coefficients to accelerate the calculations.

- **Zero-Stuffing Logic:** A simple state machine and multiplexer that controls the output data stream based on the NCO's phase accumulator overflow.

Each of these blocks must be carefully designed in a hardware description language (e.g., VHDL or Verilog) with proper fixed-point arithmetic to ensure high-speed, real-time operation. The choice of bit-widths for each signal is critical for balancing precision and dynamic range with hardware resource utilization. For instance, the timing error signal might be represented with a large number of fractional bits to ensure high resolution, while the output data samples might only require enough bits to represent the constellation points accurately. The entire design must be rigorously timed, ensuring that all signal paths meet the tight timing constraints of the high-speed system clock.

3.7 Simulation and Verification

Verification is a crucial part of the design process. The symbol synchronizer was rigorously tested using a multi-pronged approach involving MATLAB/Simulink and a VHDL/Verilog test bench.

MATLAB/Simulink Simulation: A comprehensive test environment was created to simulate the entire system, including a noisy channel, carrier frequency offset, and a clock offset. This allowed for the verification of the algorithm's behavior under various conditions. The most important plots and metrics generated were:

- **Timing Error vs. Time:** This plot shows the timing error converging to zero as the loop locks, a direct measure of the synchronizer's effectiveness. Ideally, the error should decay exponentially to zero, indicating a well-damped, stable system.
- **NCO Output vs. Time:** This plot shows the NCO's output stabilizing as the loop finds the correct clock frequency, demonstrating the stability of the control loop. A perfect plot would show the NCO output settling to a constant value, corresponding to the required frequency offset.
- **Constellation Diagrams:** This is the ultimate proof of synchronization. Constellation plots of the received signal before and after synchronization clearly showed the symbols rotating and scattering widely before locking, and then collapsing into tight clusters after synchronization, indicating successful timing and carrier recovery.
- **Eye Diagram:** The eye diagram of the signal before and after synchronization was a key visual metric. A "closed" or "blurry" eye before synchronization would open up wide and clear after the synchronizer locked on, visually confirming the reduction of ISI. This provides a compelling visual confirmation of the synchronizer's effectiveness in maximizing signal integrity.
- **Bit Error Rate (BER) vs. SNR:** This quantitative metric was used to prove that the synchronizer's performance matched theoretical predictions, ensuring the design's efficacy in a noisy environment. The simulation results for BER should closely match the theoretical curve for the given modulation scheme, confirming that the implementation is not introducing any significant performance penalties.

3.8 Hardware Validation

The final stage of the project was to validate the symbol synchronizer on the **ZCU208 RFSoC board**. This involved:

- **Synthesizing and Implementing the Design:** The VHDL/Verilog code was synthesized into a netlist and then implemented on the Zynq UltraScale+ FPGA fabric. The place-and-route stage ensured that all timing constraints were met for high-speed operation. This is a complex, iterative process where the physical layout of the design is optimized to ensure that signals propagate between logic elements within the allotted clock cycle time.
- **Debugging with an Integrated Logic Analyzer (ILA):** The ILA was used to probe internal signals in real time. This was essential for monitoring the timing error, the PI filter output, and the NCO state, verifying that the control loop was functioning as expected on the physical hardware. The ILA is an indispensable tool for debugging fixed-point overflows, unexpected state machine behavior, and other non-ideal behaviors that only manifest on physical hardware.
- **System-Level Test:** A loopback test was conducted where a known signal was generated by the DAC, passed through the digital chain, and sampled by the ADC. The output of the symbol synchronizer was then checked for correct timing and constellation quality. This test proved that the design was not only theoretically sound but also robust and reliable in a real-world, high-speed hardware environment. This hardware-in-the-loop testing confirmed the design's robustness against real-world clock jitter, power supply noise, and temperature variations.

3.9 Real-World Impairments and Robustness

The design's robustness against various real-world impairments was a key focus of the validation process. The synchronizer's performance was measured across a wide range of conditions to determine its operational limits.

- **Additive White Gaussian Noise (AWGN):** The synchronizer's ability to maintain lock and low BER was verified across a range of SNR values, proving its resilience in noisy channels. A well-designed synchronizer can maintain lock even at very low SNRs, provided the link remains viable.
- **Carrier Phase Noise:** The algorithm was found to be largely immune to small amounts of carrier phase noise, as it is a timing recovery loop and is independent of the carrier phase. However, a significant carrier frequency offset can degrade the ZCTED's performance, highlighting the need for a separate carrier frequency acquisition stage before the timing loop.
- **Fading Channels:** The synchronizer's ability to track rapid changes in signal amplitude (which is typically handled by an Automatic Gain Control loop) and timing was a key focus of the system-level tests. In a fading channel, the signal's amplitude can drop significantly, which can cause the timing detector to lose its lock. A robust design accounts for this by integrating tightly with the AGC and potentially having a wider acquisition range.

The integration of symbol synchronization with other receiver functions is critical. The symbol synchronizer's output often feeds directly into a carrier recovery loop, which uses the well-timed symbols to achieve accurate carrier phase alignment. The interaction between these two loops is a complex trade-off, with the symbol synchronizer typically having a wider bandwidth to acquire lock quickly and provide a stable input for the narrower-bandwidth carrier loop. The synchronizer and carrier recovery loops are often designed to work in tandem, with the timing loop acquiring first and then providing a high-quality input to enable the more precise carrier loop to lock.

3.10 Conclusion of Symbol Synchronization

The implementation of the ZCTED-based symbol synchronizer was a significant technical achievement. The integration of a piecewise parabolic interpolator, a PI filter, and a numerically controlled oscillator created a robust, self-correcting feedback loop capable of tracking clock drift and minimizing Inter-Symbol Interference (ISI). The combination of comprehensive MATLAB/Simulink simulations and real-time hardware validation on the ZCU208 ensured that the design was not only theoretically sound but also highly reliable and performed optimally in a real-world, high-speed hardware environment. This project provided invaluable experience in designing, implementing, and verifying complex digital signal processing algorithms on state-of-the-art hardware, demonstrating the critical link between theoretical concepts and practical engineering application.

4 Chapter 4: Viterbi Decoding

4.1 Introduction to Viterbi Decoding

Viterbi decoding is a powerful algorithm for decoding convolutional codes, which are a class of error-correcting codes widely used in digital communications to improve data reliability. Unlike block codes that encode data in fixed-size blocks, convolutional codes process a continuous stream of data by using a sliding-window approach. The encoder's output at any given time depends not only on the current input bit but also on the previous input bits stored in a shift register. This dependency creates a finite number of possible "states" for the encoder. Viterbi decoding functions by finding the most likely sequence of transmitted symbols from a noisy, corrupted signal by efficiently searching through a trellis diagram, which represents all possible state transitions of the convolutional encoder. The core of the Viterbi algorithm is a dynamic programming approach that systematically eliminates unlikely paths through the trellis, thereby reducing the computational complexity to a manageable level. This makes it a maximum likelihood sequence estimator (MLSE), ensuring the best possible decoding performance in the presence of channel noise.

4.2 The Trellis Diagram and Viterbi's Logic

The **trellis diagram** is the graphical representation of all possible state transitions of a convolutional encoder over time. It's a key concept for understanding how Viterbi decoding works. The trellis consists of:

- **Nodes (States):** Each node in the trellis represents a unique state of the convolutional encoder's memory at a given time step.
- **Branches (Transitions):** A branch connects a state at time $t-1$ to a state at time t. Each branch corresponds to a single input bit (0 or 1) and has a specific output symbol generated by the encoder.
- **Paths:** A path through the trellis is a sequence of connected branches that represents a possible sequence of transmitted symbols.

The Viterbi algorithm works by processing the received data on a symbol-by-symbol basis, traversing the trellis. For each received symbol, it calculates the "cost" of taking each possible branch, then updates the total "cost" to reach each state. This cost is known as the **path metric**. The algorithm's key insight is that for any given state, the most likely path to have reached that state must be a part of the overall most likely path. Therefore, if two or more paths converge on the same state, the algorithm only needs to keep the path with the minimum path metric (the "survivor path") and discard all the others. This process of elimination is what makes the Viterbi algorithm computationally efficient. The Viterbi algorithm operates in three main stages: the Branch Metric Unit, the Add-Compare-Select Unit, and the Traceback Unit.

4.3 The Branch Metric Unit (BMU)

The **Branch Metric Unit (BMU)** is the first stage of a Viterbi decoder. Its primary function is to calculate the **branch metrics**, which quantify the "distance" or likelihood between the received symbols and all possible transmitted symbols for each state

transition. The metric can be a measure of distance, such as the Euclidean distance for continuous-valued received signals (soft-decision decoding), or a Hamming distance for binary symbols (hard-decision decoding). For a convolutional code, each state transition (or "branch") corresponds to a specific encoder output. The BMU takes the received symbol as input and compares it against the expected symbol for each branch emanating from every state in the trellis. The output of the BMU is a set of branch metrics for each possible transition at the current time step.

For a hard-decision, binary input signal, the branch metric is the Hamming distance, which is the number of bit positions in which the two symbols differ. If the received symbol is $\mathbf{y} = [y_1, y_2, ..., y_n]$ and the expected symbol for a given branch is $\mathbf{x} = [x_1, x_2, ..., x_n]$, the branch metric λ is:

$$\lambda = \sum_{i=1}^{n} (y_i \oplus x_i)$$

where $\oplus$ denotes the XOR operation.

4.4 The Add-Compare-Select Unit (ACSU) and Path Metric Unit (PMU)

The **Add-Compare-Select Unit (ACSU)** and **Path Metric Unit (PMU)** are implemented together as the central, iterative stage of the Viterbi algorithm. The ACSU's job is to update the path metrics for each state in the trellis at every time step. For a given state, the ACSU identifies all incoming branches. It then performs the following operations:

1. **Add:** It adds the branch metric of each incoming branch to the path metric of the previous state from which the branch originated.

2. **Compare:** It compares the results of all the additions to find the one with the smallest path metric. This minimum value represents the "best" or most likely path to the current state.

3. **Select:** It selects the path with the minimum path metric and stores its information, including a pointer to the previous state from which it originated. This chosen path is called the "survivor path."

The **Path Metric Unit (PMU)** is the memory element that stores these updated path metrics for all states. The PMU's contents are essential for the next time step's ACSU operations. The path metric, denoted as $\Gamma_t(S_i)$, for a state S_i at time t is calculated recursively as:

$$\Gamma_t(S_i) = \min_{\text{incoming branches}} [\Gamma_{t-1}(S_j) + \lambda_{j,i}]$$

where $\Gamma_{t-1}(S_j)$ is the path metric of the previous state S_j, and $\lambda_{j,i}$ is the branch metric for the transition from S_j to S_i. The ACSU and PMU's combined operation ensures that only the most likely paths are kept in memory, which is the key to the algorithm's efficiency.

4.5 The Traceback Unit (TBU)

The **Traceback Unit (TBU)** is the final stage of the Viterbi decoder. Its purpose is to reconstruct the decoded output sequence once a sufficient number of symbols have been processed. After a predetermined delay (known as the traceback depth), the path metrics for all states in the trellis will have converged, meaning a single, most likely path is shared by all surviving paths.

The TBU works backward from the final state of the trellis. It uses the stored "survivor path" pointers generated by the ACSU to trace a path back through the trellis to its origin. Each step of the traceback corresponds to a decoded bit. The traceback operation can be described as follows:

1. It starts at the state with the minimum path metric at the current time step.
2. It follows the stored pointer back to the previous state's survivor path.
3. It continues this process, step by step, for a length equal to the traceback depth.
4. Once the traceback is complete, the sequence of bits associated with the traced path is the decoded output.

The TBU's memory requirements and the traceback delay are directly related to the traceback depth. A longer depth increases the decoding delay but also improves the accuracy, as it allows the algorithm to correct for temporary path errors. The traceback output is typically a single, reliable bit that is an estimate of the originally transmitted data bit.

4.6 A Practical Example of Viterbi Decoding

Let's walk through a concrete example using a simple rate 1/2 convolutional encoder with a constraint length $K = 3$. This means for every 1 input bit, we get 2 output bits, and the output depends on the current input bit and the 2 previous bits stored in the shift register. The encoder state is defined by the contents of the shift register. In this case, there are $2^{K-1} = 2^2 = 4$ possible states.

Our example encoder has two generator polynomials:

- $g_1 = (1, 0, 1)$, which means the first output bit is the XOR sum of the current input and the second bit in the shift register.
- $g_2 = (1, 1, 1)$, which means the second output bit is the XOR sum of the current input and all two bits in the shift register.

The four possible states are:

- State $A = (0, 0)$
- State $B = (0, 1)$
- State $C = (1, 0)$
- State $D = (1, 1)$

A transition from a state is made by a new input bit. The first bit of the state represents the oldest bit in memory. For a new input of 0:

- From state A (00), the new state is also A (00). The output is (0,0).
- From state B (01), the new state is A (00). The output is (1,1).
- From state C (10), the new state is B (01). The output is (1,0).
- From state D (11), the new state is B (01). The output is (0,1).

For a new input of 1:

- From state A (00), the new state is C (10). The output is (1,1).
- From state B (01), the new state is C (10). The output is (0,0).
- From state C (10), the new state is D (11). The output is (0,1).
- From state D (11), the new state is D (11). The output is (1,0).

These transitions form the basis for our trellis. A solid line represents an input of '0', and a dashed line represents an input of '1'.

Example Problem: Let's assume the transmitted bit sequence was '1 0 1', and the received, corrupted sequence is '(1,1), (0,0), (1,0)'. We will use **hard-decision decoding** (Hamming distance). The initial path metrics for all states are infinite, except for the initial state, which is A(0,0), with a path metric of 0.

Time $t = 1$: Received '(1,1)'

- **From State A (00):**
  - Input '0': Transition to A(00). Expected output is '(0,0)'. Branch metric (Hamming distance) = '(1,1)' vs '(0,0)' = 2. Path metric = $0 + 2 = 2$.
  - Input '1': Transition to C(10). Expected output is '(1,1)'. Branch metric = '(1,1)' vs '(1,1)' = 0. Path metric = $0 + 0 = 0$.
- The other states (B, C, D) are unreachable from the initial state at this time, so their path metrics remain infinite.
- **Update Path Metrics:**
  - PM(A) = 2.
  - PM(B) = ∞.
  - PM(C) = 0.
  - PM(D) = ∞.

Time $t = 2$: Received '(0,0)'

- **To State A (00):**
  - From A (input 0): PM = PM(A) + Branch Metric = 2+ ('(0,0)' vs '(0,0)') = $2 + 0 = 2$.
  - From B (input 0): PM = PM(B) + Branch Metric = ∞+ ('(0,0)' vs '(1,1)') = ∞.

  - **Select Survivor:** The survivor path to A is from state A, with a path metric of 2.

- **To State B (01):**
  - From C (input 0): PM = PM(C) + Branch Metric = 0+ (‘(0,0)‘ vs ‘(1,0)‘) = $0 + 1 = 1$.
  - From D (input 0): PM = PM(D) + Branch Metric = ∞+ (‘(0,0)‘ vs ‘(0,1)‘) = ∞.
  - **Select Survivor:** The survivor path to B is from state C, with a path metric of 1.

- **To State C (10):**
  - From A (input 1): PM = PM(A) + Branch Metric = 2+ (‘(0,0)‘ vs ‘(1,1)‘) = $2 + 2 = 4$.
  - From B (input 1): PM = PM(B) + Branch Metric = ∞+ (‘(0,0)‘ vs ‘(0,0)‘) = ∞.
  - **Select Survivor:** The survivor path to C is from state A, with a path metric of 4.

- **To State D (11):**
  - From C (input 1): PM = PM(C) + Branch Metric = 0+ (‘(0,0)‘ vs ‘(0,1)‘) = $0 + 1 = 1$.
  - From D (input 1): PM = PM(D) + Branch Metric = ∞+ (‘(0,0)‘ vs ‘(1,0)‘) = ∞.
  - **Select Survivor:** The survivor path to D is from state C, with a path metric of 1.

- **Update Path Metrics:**
  - PM(A) = 2.
  - PM(B) = 1.
  - PM(C) = 4.
  - PM(D) = 1.

Time $t = 3$: Received ‘(1,0)‘

- **To State A (00):**
  - From A (input 0): PM = PM(A) + BM = 2+ (‘(1,0)‘ vs ‘(0,0)‘) = $2 + 1 = 3$.
  - From B (input 0): PM = PM(B) + BM = 1+ (‘(1,0)‘ vs ‘(1,1)‘) = $1 + 1 = 2$.
  - **Select Survivor:** Survivor to A is from B with PM of 2.

- **To State B (01):**
  - From C (input 0): PM = PM(C) + BM = 4+ (‘(1,0)‘ vs ‘(1,0)‘) = $4 + 0 = 4$.
  - From D (input 0): PM = PM(D) + BM = 1+ (‘(1,0)‘ vs ‘(0,1)‘) = $1 + 2 = 3$.

  - **Select Survivor:** Survivor to B is from D with PM of 3.

- **To State C (10):**
  - From A (input 1): PM = PM(A) + BM = 2+ (‘(1,0)‘ vs ‘(1,1)‘) = $2+1=3$.
  - From B (input 1): PM = PM(B) + BM = 1+ (‘(1,0)‘ vs ‘(0,0)‘) = $1+1=2$.
  - **Select Survivor:** Survivor to C is from B with PM of 2.

- **To State D (11):**
  - From C (input 1): PM = PM(C) + BM = 4+ (‘(1,0)‘ vs ‘(0,1)‘) = $4+2=6$.
  - From D (input 1): PM = PM(D) + BM = 1+ (‘(1,0)‘ vs ‘(1,0)‘) = $1+0=1$.
  - **Select Survivor:** Survivor to D is from D with PM of 1.

Traceback (from the final state with the lowest path metric): The path metrics at time $t = 3$ are: PM(A)=2, PM(B)=3, PM(C)=2, PM(D)=1. The lowest path metric is 1, which belongs to state D.

- **Step 1:** Start at D at $t = 3$. The survivor path came from D at $t = 2$ (Input 1). We decode ‘1‘.
- **Step 2:** From D at $t = 2$, the survivor path came from C at $t = 1$ (Input 1). We decode ‘1‘.
- **Step 3:** From C at $t = 1$, the survivor path came from A at $t = 0$ (Input 1). We decode ‘1‘.

The decoded sequence is ‘1 1 1‘.

Analysis of the Result: Our decoded sequence, ‘1 1 1‘, is different from the original sequence, ‘1 0 1‘. Why? The Viterbi algorithm found the most likely path given the received, corrupted sequence. The received sequence ‘(1,1), (0,0), (1,0)‘ has a low overall Hamming distance from the sequence corresponding to ‘1 1 1‘. This demonstrates that even with errors, the Viterbi algorithm finds the most probable transmitted sequence, correcting for the second bit error.

4.7 Conclusion of Viterbi Decoding

The Viterbi decoder’s four main components—the Branch Metric Unit (BMU), the Add-Compare-Select Unit (ACSU), the Path Metric Unit (PMU), and the Traceback Unit (TBU)—form a complete and highly effective error-correcting system. The BMU’s role is to quantify the likelihood of each received symbol. The ACSU and PMU work together to dynamically find and store the most likely paths through the trellis. Finally, the TBU reconstructs the most probable data sequence by tracing back the survivor paths. This elegant and robust algorithm is a cornerstone of modern digital communications, widely used in everything from mobile phones to deep-space communication, where reliable data transmission is paramount.

5 Chapter 5: Reed-Solomon Decoding

5.1 Introduction to Reed-Solomon Codes

Reed-Solomon (RS) codes are a class of non-binary, cyclic block codes widely used in digital communications and storage systems, such as compact discs, DVDs, and QR codes. Unlike convolutional codes, which operate on a continuous stream of bits, RS codes process data in fixed-size blocks of symbols. A symbol is a collection of m bits, and all code operations are performed within a finite field known as a **Galois Field** or $GF(2^m)$. This field-based arithmetic is the key to their power.

An RS code is defined as $RS(n, k)$, where:

- n is the total number of symbols in the codeword.
- k is the number of data symbols.

This means that $n - k$ symbols are added for error correction. The number of symbols in a codeword is $n = 2^m - 1$, and the code can correct up to t symbol errors, where $2t = n - k$. This is a powerful property; the code can correct any pattern of up to t errors, which makes it ideal for channels prone to burst errors.

The decoding process is a sophisticated chain of algorithms that systematically detect errors, locate their positions, and calculate their magnitudes. The core steps of this process, which will be detailed in the following sections, are:

1. **Syndrome Calculation**: Detect the presence of errors.
2. **Berlekamp-Massey Algorithm**: Find the error locator polynomial.
3. **Chien Search**: Locate the positions of the errors.
4. **Forney's Algorithm**: Determine the magnitude of the errors.
5. **Error Correction**: Apply the corrections to the received codeword.

5.2 The Foundation: A Deep Dive into Galois Fields $GF(2^m)$

To understand Reed-Solomon codes, you must first understand the "math world" they live in: the **Galois Field**, $GF(2^m)$.

Imagine a special universe where there is only a finite number of "numbers." This isn't like our familiar number line that goes on forever. In a Galois Field, if you do math with any two numbers, the answer is always another number that's still inside this finite set. It's a closed world for arithmetic.

For Reed-Solomon codes, this "world" is $GF(2^m)$, and its numbers are actually **polynomials** with coefficients that are either 0 or 1. A symbol with m bits (like a byte for $m = 8$) corresponds to a polynomial of degree up to $m-1$. For example, the 8-bit symbol '10110001' corresponds to the polynomial $x^7 + x^5 + x^4 + 1$.

As a concrete example, let's consider the popular **$RS(15, 9)$ code**.

- The total number of symbols is $n = 15$.
- The number of data symbols is $k = 9$.

The number of symbols is defined by the Galois Field size, $n = 2^m - 1$. Since $15 = 2^4 - 1$, this code operates in $^{**}GF(2^4)^{**}$, where each symbol is $m = 4$ bits long.

In this field, the arithmetic is based on polynomial operations modulo a specific irreducible polynomial. For $GF(2^4)$, a common and widely used irreducible polynomial is $^{**}x^4 + x + 1^{**}$. This is the "prime number" of our field that ensures the multiplication operation always results in a unique, valid symbol within the set of 16 elements.

All RS decoding algorithms use these specific Galois Field rules. Without this unique algebraic structure, Reed-Solomon codes wouldn't work.

5.3 Step 1: Syndrome Calculation - The Initial Clue

The first step in decoding is like a diagnostic check. You need to find out if there are any errors in the received codeword. This is what the **syndromes** are for.

Think of the original, valid Reed-Solomon codeword as a secret message that follows a very specific mathematical rule. The rule is: if you plug certain special "keys" (the roots of the generator polynomial) into the codeword's polynomial, the result is always zero.

When the received codeword arrives, possibly corrupted by noise, the decoder performs the exact same check. It takes the received codeword polynomial, $r(x)$, and plugs in each of these special "keys," $\alpha^1, \alpha^2, \ldots, \alpha^{n-k}$. The result of each check is a syndrome, S_j.

$$S_j = r(\alpha^j)$$

The summation is done using Galois Field arithmetic.

* **If a check results in zero** ($S_j = 0$), it means that for that specific key, the received message still follows the rule. * **If a check results in a non-zero value** ($S_j \neq 0$), it means the rule has been broken! The non-zero value, the syndrome, is a direct result of the corruption. It's not just a flag; this non-zero value actually contains all the information needed to pinpoint and fix the errors later.

The decoder calculates $n - k$ syndromes. If they are all zero, it assumes no errors occurred and stops. If any are non-zero, it proceeds to the next step, using these non-zero syndromes as the raw data to begin the correction process.

5.4 Step 2: The Berlekamp-Massey Algorithm

The **Berlekamp-Massey (BM) algorithm** is a powerful, iterative procedure used to find the **error locator polynomial**, $\Lambda(x)$. This polynomial's roots are the inverse of the error locations. The algorithm's purpose is to find the shortest linear feedback shift register (LFSR) that can generate the calculated syndrome sequence.

The algorithm takes the syndrome values, $S_1, S_2, ..., S_{n-k}$, as input and iteratively builds the error locator polynomial $\Lambda(x)$. It also finds the **error evaluator polynomial**, $\Omega(x)$, which will be used later.

The BM algorithm can be seen as a recursive process. For each syndrome value S_j, it does the following:

1. It computes a **discrepancy** Δ_j, which measures how well the current estimate of $\Lambda(x)$ predicts the next syndrome.

2. If the discrepancy Δ_j is zero, the algorithm's current estimate is correct, and it continues to the next syndrome without changing $\Lambda(x)$.

3. If Δ_j is non-zero, the algorithm updates $\Lambda(x)$ and its other internal state variables. The update procedure is designed to correct the discrepancy, ensuring that the new $\Lambda(x)$ is a better fit for the syndrome sequence.

The final $\Lambda(x)$ polynomial, after processing all syndromes, will have a degree equal to the number of errors, t.

5.5 Step 3: The Chien Search

The **Chien Search** is a straightforward algorithm used to find the roots of the error locator polynomial, $\Lambda(x)$. Each root corresponds to a specific error location in the received codeword.

The algorithm is based on the fundamental property that if $\Lambda(x_i) = 0$, then x_i is a root of the polynomial. For Reed-Solomon codes, the error locations are a subset of the Galois Field elements. The error locator polynomial is defined such that its roots are the reciprocals of the error locations. That is, if the errors are at locations $i_1, i_2, ..., i_t$, then the roots of $\Lambda(x)$ are $\alpha^{-i_1}, \alpha^{-i_2}, ..., \alpha^{-i_t}$.

The Chien Search performs a simple, exhaustive check: it evaluates $\Lambda(x)$ for every possible field element, $\alpha^0, \alpha^{-1}, \alpha^{-2}, ..., \alpha^{-(n-1)}$.

$$\Lambda(\alpha^{-j}) = 0 \quad \text{for } j = 0, 1, ..., n-1$$

If the evaluation results in zero, it means an error occurred at location j. This algorithm is computationally efficient because it involves only multiplication and addition in the Galois Field, which can be done quickly with lookup tables.

5.6 Step 4: Forney's Algorithm

Once the error locations have been found by the Chien Search, the next step is to determine the magnitude of the error at each of those locations. This is the purpose of **Forney's Algorithm**.

Forney's Algorithm calculates the error value, e_j, at each error location j. The formula uses the error evaluator polynomial $\Omega(x)$ and the derivative of the error locator polynomial, $\Lambda'(x)$.

The error value e_j at error location j is given by:

$$e_j = \frac{\Omega(\alpha^{-j})}{\Lambda'(\alpha^{-j})}$$

where α^{-j} is the root of $\Lambda(x)$ corresponding to the error location j.

The derivative of the error locator polynomial $\Lambda'(x)$ is calculated as follows:

$$\text{If } \Lambda(x) = \sum_{i=0}^{t} \lambda_i x^i \quad \text{then} \quad \Lambda'(x) = \sum_{i=1}^{t} i\lambda_i x^{i-1}$$

where the coefficients i are integers, and the arithmetic is performed in the Galois Field. This division in the Galois Field is equivalent to multiplication by the inverse of the denominator.

5.7 Step 5: Error Correction

This is the final and most straightforward step of the decoding process. All the previous algorithms have provided all the necessary information to correct the received codeword: the locations of the errors and their corresponding magnitudes.

The corrected codeword polynomial, $c(x)$, is found by subtracting the error polynomial, $e(x)$, from the received polynomial, $r(x)$. Since subtraction is the same as addition in $GF(2^m)$, the formula is:

$$c(x) = r(x) + e(x)$$

The error polynomial $e(x)$ is constructed such that it has non-zero values only at the error locations, with the magnitudes calculated by Forney's algorithm.

For example, if an error of magnitude e_j was found at location j, the corrected symbol c_j is:

$$c_j = r_j + e_j$$

All other symbols remain unchanged. The corrected codeword is now a valid codeword in the RS code and can be passed on to the next stage of the communication system.

5.8 Summary of Reed-Solomon Decoding

The Reed-Solomon decoding process is a sophisticated and highly effective chain of algorithms. It begins by calculating **syndromes** to detect the presence of errors. The **Berlekamp-Massey algorithm** then uses these syndromes to compute the **error locator polynomial**. The **Chien Search** finds the roots of this polynomial to determine the locations of the errors. Finally, **Forney's algorithm** calculates the error magnitudes at those locations, and the original codeword is restored by simply adding the calculated errors to the received symbols. This methodical approach ensures that RS codes can reliably correct a high number of burst errors, making them an indispensable tool in modern data communication and storage.

6 Chapter 6: ADC-DAC Loopback on ZYNQ Ultrascale+ RFSoC board (ZCU208)

The DAC-to-ADC loopback is a foundational test for any RFSoC-based system. Its purpose is to validate the integrity of the high-speed data path from the digital domain, through the Digital-to-Analog Converters (DACs), back into the digital domain via the Analog-to-Digital Converters (ADCs). This process serves as a crucial sanity check for system functionality, signal integrity, and timing alignment without the complexity of external RF components. It ensures that the digital processing and data converters are operating correctly before introducing external factors like antennas and RF front-ends.

The loopback test on the ZCU208 evaluation board involves a specific set of hardware and software components, working together to create a closed-loop signal chain. This report will detail these components and their roles in a typical loopback workflow.

6.1 Hardware and Physical Setup

The loopback process on the ZCU208 is simplified by two key hardware components that manage the physical signal path and clocking.

6.2 XM655 FMC+ Loopback Card

The XM655 is a specialized FPGA Mezzanine Card (FMC+) module designed specifically for the ZCU208. Its sole purpose is to provide a clean, direct connection between the output of the DAC tiles and the input of the ADC tiles without the need for external RF cables. This eliminates signal degradation, impedance mismatches, and noise that would be introduced by external cabling, providing a pristine test environment. The card routes the analog signals from the DAC outputs directly to the corresponding ADC inputs, which is essential for accurate performance characterization.

6.3 Clock 104

Clock synchronization is critical for any high-speed data converter system. The CLK104 is an external clock generation module that provides highly stable, low-jitter clock signals to both the RF-ADC and RF-DAC tiles on the ZCU208. It supports differential outputs, ensuring a robust clock signal. Proper configuration of the CLK104, typically through a dedicated utility like TICS Pro, is the first step in the loopback process. A single, synchronized clock source for both the DAC and ADC ensures that the data is sampled and converted coherently, preventing timing errors that would invalidate the loopback test.

6.4 Zynq UltraScale+ RFSoC ZU48DR Architectural Components

The ZCU208 evaluation board features the Zynq UltraScale+ RFSoC ZU48DR chip. The following table provides a detailed breakdown of its key architectural components, which are essential for understanding the board's capabilities.

Category	Details

RF Transceiver Subsystem	• Consists of multiple ADC and DAC tiles • Each tile has 2 or 4 channels • Channels within a tile are time-aligned and operate at the same sampling rate • Inter-tile alignment is not guaranteed and may require correction in systems such as MIMO
ADC Tile Features	• Each tile includes 2 or 4 ADC channels • Each channel contains: – Digital Down Converters (DDC) – Programmable decimation filters – Numerically Controlled Oscillators (NCOs) for frequency shifting
DAC Tile Features	• Each tile includes 2 or 4 DAC channels • Each channel contains: – Digital Up Converters (DUC) – Programmable interpolation filters – NCO mixers for spectral positioning
Programmable Logic (PL)	• Based on UltraScale+ FPGA fabric • Supports: – Custom DSP logic – Soft-decision Forward Error Correction (FEC) – High-speed connectivity (e.g., AXI4-Stream, GT interfaces)
Processing System (PS)	• Quad-core ARM Cortex-A53 for general-purpose processing • Dual-core ARM Cortex-R5 for real-time control tasks • Handles system configuration, management, and protocol processing

Digital-Only RF Front-End	• Does not include analog RF components such as: – Low-Noise Amplifiers (LNAs) – SAW filters – Matching networks • Requires connection to external RF front-end for RF transmission and reception

6.5 RF Data Converter Configuration

The RFSoC architecture allows for extensive programmability of the RF data converter tiles. For a successful loopback, the DAC and ADC must be configured to work seamlessly together.

6.6 Tool: Zynq RFSoC Board UI

The RFSoC Board User Interface (UI) is a graphical tool that provides direct access to the RF data converters. It is the go-to tool for configuring the behavior of the DAC and ADC tiles without writing low-level code. For a loopback test, the UI is used to:

- **Set Sampling Rates**: Define the exact sampling frequencies for the DACs and ADCs.
- **Mixer Configuration**: The RFSoC features integrated Numerically Controlled Oscillators (NCOs) and mixers. For a baseband loopback, these are configured to a center frequency of zero, ensuring the signal is not up-converted or down-converted in the analog domain.
- **Tile and Channel Selection**: The UI allows you to enable and select the specific DAC and ADC channels you wish to use for the loopback. This is crucial for isolating the signal path for testing.
- **Decimation and Interpolation**: The UI is used to set the decimation factors on the ADC and the interpolation factors on the DAC to match the data rates with the digital logic in the FPGA fabric.

The UI simplifies the setup process and provides live status feedback, which is invaluable for debugging any synchronization or configuration issues.

6.7 JESD204B Interface

The data transfer between the RF data converter tiles and the FPGA's programmable logic is managed by a high-speed serial interface known as JESD204B. This standard is designed to reduce the number of physical data lines (pins) needed for high-bandwidth data, which is critical for complex, high-channel-count systems. In the context of the RFSoC, JESD204B acts as a gigabit-level serial link that transports the digitized RF

samples from the ADC to the PL and the digital baseband samples from the PL to the DAC.

A key feature of JESD204B is its ability to ensure **lane synchronization** and **frame alignment**. The RFSoC leverages Subclass 1, which uses a dedicated SYSREF signal from an external clock source like the CLK104 to provide a precise, deterministic timing reference. This ensures that data from multiple lanes and multiple tiles arrive at the PL in perfect alignment, a crucial requirement for coherent processing in applications like beamforming and MIMO.

6.8 Signal Processing in the Data Converters

The RF data converter tiles are not just simple ADCs and DACs; they contain a sophisticated digital front-end that performs essential signal processing tasks, offloading these computationally intensive operations from the FPGA fabric.

6.9 Numerically Controlled Oscillators (NCOs)

NCOs are digital sine wave generators. In the RFSoC, they are used as part of the integrated mixers to perform **frequency shifting** or **digital up/down-conversion**. When configured for a baseband loopback, the NCOs are typically bypassed or set to a frequency of zero to preserve the signal's original frequency. For more advanced tests, an NCO can be configured to digitally mix a baseband signal to an intermediate frequency (IF) before it is passed to the DAC. Conversely, on the receive side, an NCO can be used to down-convert an IF signal back to baseband. This functionality is configured through the RFSoC Board UI or a dedicated software application.

6.10 Digital Down-Converters (DDCs) and Digital Up-Converters (DUCs)

The ADC and DAC tiles feature integrated DDCs and DUCs, respectively. These blocks combine the NCO mixing with configurable digital filters to perform decimation (on the ADC path) and interpolation (on the DAC path).

- **Decimation (ADC Path)**: The ADC may sample at a very high rate (e.g., 4 Gsps). A DDC with a decimation factor of 8 would take 8 samples from the ADC and produce a single output sample, thereby reducing the data rate to 500 Msps. This is essential for matching the high ADC sampling rate to the lower clock rate of the digital logic in the FPGA fabric.
- **Interpolation (DAC Path)**: The DUC performs the reverse operation. It takes a lower-rate data stream from the FPGA (e.g., 500 Msps) and produces a higher-rate stream of samples for the DAC (e.g., 4 Gsps) by inserting zeros and filtering. This increases the sampling rate and helps to shape the spectrum of the output signal.

Properly setting these interpolation and decimation factors is a critical step in the loopback setup to ensure a seamless data path.

6.11 Software and Hardware Design Flow

The complete loopback workflow is a hardware-software co-design process that leverages the Vivado, Vitis, and RF Analyzer tool suites.

6.12 Vivado Design Suite

Vivado is used to design the digital hardware on the Zynq RFSoC's Programmable Logic (PL). For a loopback, the key task is to create a digital data path from the ADC to the DAC. This is typically done using the **IP Integrator** tool, where you:

- Instantiate the **RF Data Converter IP** block. This is the core intellectual property that controls the DAC and ADC tiles.
- Configure the IP to enable the desired DAC and ADC channels and set up the AXI4-Stream interfaces for data transfer.
- Create a "loopback" path in the PL. This can be as simple as connecting the AXI4-Stream output of the DAC to the AXI4-Stream input of the ADC. This ensures that the digital data sent to the DAC is the same data received from the ADC.

The Vivado project is then synthesized and implemented, creating a bitstream file that configures the FPGA fabric with the defined hardware logic.

6.13 Vitis Unified Software Platform

Vitis is the software development environment that controls the hardware designed in Vivado. The ARM processor (Processing System or PS) on the RFSoC is used to:

- Configure the RF data converters by writing to their registers via the AXI4-Lite interface.
- Program the DAC to generate a test signal, such as a sine wave or a simple tone.
- Read the captured data from the ADC into a buffer for analysis.

The Vitis application orchestrates the entire loopback test, from initializing the hardware to capturing and logging the final data.

6.14 RF Analyzer

The RF Analyzer is the primary tool for verifying and debugging the loopback test. After Vitis has captured the data from the ADC, this data can be imported into the RF Analyzer for visual inspection and analysis. The tool provides a suite of features including:

- **Spectrum View**: Displays the frequency spectrum of the captured signal, allowing you to see the generated tone from the DAC and verify that it has been received cleanly by the ADC.
- **Time Domain View**: Shows the raw I/Q samples over time, useful for checking for signal clipping or distortion.

- **Constellation Diagrams**: For modulated signals, this view shows the constellation points, which is a key indicator of signal quality.

The RF Analyzer is essential for confirming that the loopback is successful and that the signal fidelity has been maintained through the entire analog-to-digital-to-analog conversion chain.

6.15 Summary of the Loopback Workflow

The entire process is a systematic validation of the RFSoC's core functionality.

1. The **XM655** physically connects the DAC outputs to the ADC inputs on the ZCU208.
2. The **CLK104** provides a synchronized, low-jitter clock to both sets of data converters.
3. The **RFSoC Board UI** is used to graphically configure the DAC and ADC tiles (sampling rates, NCOs, etc.).
4. **Vivado** is used to design the digital loopback path in the FPGA fabric, which includes the **JESD204B** interface.
5. **Vitis** provides the software control to generate a test signal and capture the returned data.
6. The **RF Analyzer** is used to inspect the captured data and verify the loopback's success, analyzing the spectrum, time domain, and constellation.

This methodical workflow provides a robust method for testing the most critical signal path on the Zynq RFSoC board.