**HDL Implementation of LDPC Decoder**

**Theoretical Background**

The Min-Sum algorithm is a popular decoding algorithm for Low-Density Parity-Check (LDPC) codes, known for its simplicity and effectiveness in hardware implementations. Let's explore the theoretical background and considerations for implementing LDPC decoders using the Min-Sum algorithm in hardware.

**LDPC Codes:**

LDPC codes are linear error-correcting codes characterized by sparse parity-check matrices. These codes offer remarkable error correction capabilities and are widely used in various communication and storage systems. LDPC codes operate based on the principle of paritycheck equations, where the parity bits are computed based on the transmitted message bits. The sparse nature of LDPC codes allows for efficient encoding and decoding algorithms.

**Min-Sum Algorithm:**

The Min-Sum algorithm is an iterative message-passing algorithm for decoding LDPC codes. It operates by exchanging messages between variable nodes (representing the transmitted bits) and check nodes (representing the parity-check equations). The algorithm iteratively updates the messages until a valid codeword is obtained or a maximum number of iterations is reached.

At each iteration, the Min-Sum algorithm computes the messages using the following update rules:

1. **Variable-to-Check Messages**: Variable nodes send messages to connected check nodes, containing the accumulated log-likelihood ratios (LLRs) received from other check nodes.

1. **Check-to-Variable Messages**: Check nodes compute messages based on the received LLRs from connected variable nodes and select the two smallest LLRs. These LLRs are then subtracted from the other LLRs before sending the updated messages back to the variable nodes.

The Min-Sum algorithm continues iterating between variable nodes and check nodes until convergence or until a predefined maximum number of iterations is reached. The LLRs computed at the variable nodes are used to estimate the transmitted bits, thereby decoding the received LDPC codeword.

**Hardware Implementation Challenges**:

Implementing the Min-Sum algorithm for LDPC decoding in hardware presents several challenges:

1. **Computational Complexity**: The Min-Sum algorithm involves complex arithmetic operations, including addition, subtraction, and comparison operations, which can be computationally intensive.

1. **Memory Requirements**: Hardware implementations require large memory buffers to store LLRs, messages, and intermediate computations, increasing the hardware complexity and resource utilization.

3.**Throughput Requirements**: LDPC decoders are often required to process data at high speeds to meet real-time communication demands. Efficient hardware architectures are needed to achieve high throughput rates while maintaining low latency.

4. **Power Consumption**: Power-efficient designs are essential for hardware implementations of LDPC decoders, particularly in battery-operated devices such as mobile phones and IoT devices.

**Hardware Architectures:**

Various hardware architectures have been proposed for LDPC decoders based on the Min-Sum algorithm:

1. **Parallel Architectures**: These architectures exploit parallelism at different levels to accelerate computations and improve throughput, including bit-level parallelism, symbol-level parallelism, and algorithm-level parallelism.

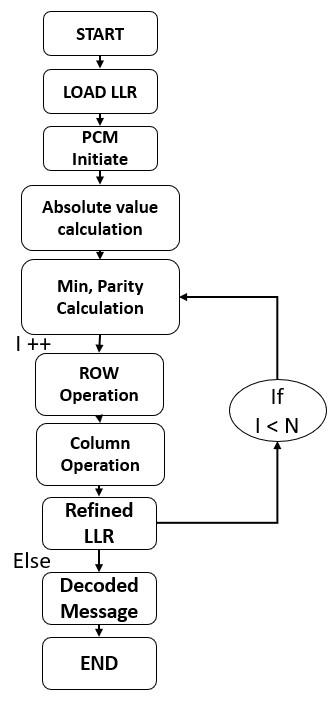
1. **Memory-Efficient Architectures**: Hardware designs utilize memory-efficient data structures and storage schemes to minimize memory requirements and reduce hardware complexity, such as using register files and distributed memory architectures.

1. **Approximate Computing**: Some implementations leverage approximate computing techniques to trade off accuracy for reduced computational complexity and power consumption, such as using truncated arithmetic operations.

1. **Reconfigurable Architectures**: Reconfigurable architectures like FPGAs and ASICs offer flexibility and customization options, allowing for optimized LDPC decoder designs tailored to specific applications and performance requirements.

In summary, the Min-Sum algorithm offers a simple yet effective approach for decoding LDPC codes in hardware. Despite the challenges posed by computational complexity, memory requirements, throughput, and power consumption, innovative hardware architectures and optimization techniques enable efficient implementations of LDPC decoders. These hardware implementations play a crucial role in modern communication and storage systems, offering robust error correction capabilities and high-performance decoding solutions.

**Flowchart of Min-Sum Algorithm in HDL Implementation**



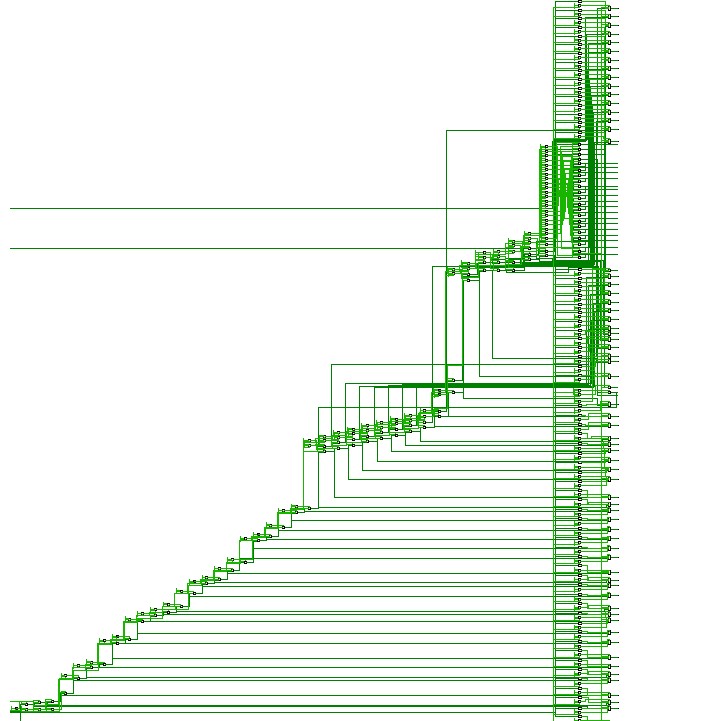
*Figure 21: Flowchart of Min-Sum Algorithm*

1. **Module Declaration**:
   * module named **ldpc\_decoder** with parameters **N**, **r**, **c**, and **size**, and inputs and outputs specified within the parentheses.
   * C =1200
   * R =600 • N(iterations) = 5
2. **Variables and Wires**:
   * Registers (**reg**) and wires (**wire**) are declared to store data and intermediate values during the decoding process.
3. **PCM Initialization**:
   * A matrix **PCM** is initialized with data; i.e., Sparse Matrix
4. **Absolute Value Calculation**:
   * Absolute values for each element of **Rr** are calculated and stored in **Rr\_abs**.
5. **Mins and Parity Calculation**:
   * For each row of the matrix **PCM**, minimums (**min1** and **min2**) and parity are calculated and stored. This is done using the **parity** and **mins** modules instantiated within a generate block.
6. **LLR Refining and Updating**:
   * LLR values are refined and updated based on the calculated minimums and parity. This process involves iterating through the rows and columns of **PCM** and updating the LLR values accordingly.
7. **Finding Mins in Updated Columns**:
   * Minimums are found in the updated columns based on the refined LLR values. This is done within a generate block and involves iterating through the columns of **PCM** to find the minimums.
8. **Control Unit**:
   * A finite state machine controls the overall operation of the module. It manages state transitions and timing of various operations.
   * The state machine transitions through different states (**0** to **8**) based on the **counter** value and timing signals.
9. **Decision Making for Output Decoded Message**:
   * The output decoded message is determined based on the sign of **Rr**. If **Rr** is negative, the corresponding bit in the output message (**message**) is set to **1**, otherwise, it's set to **0**.
10. **Timing Unit**:

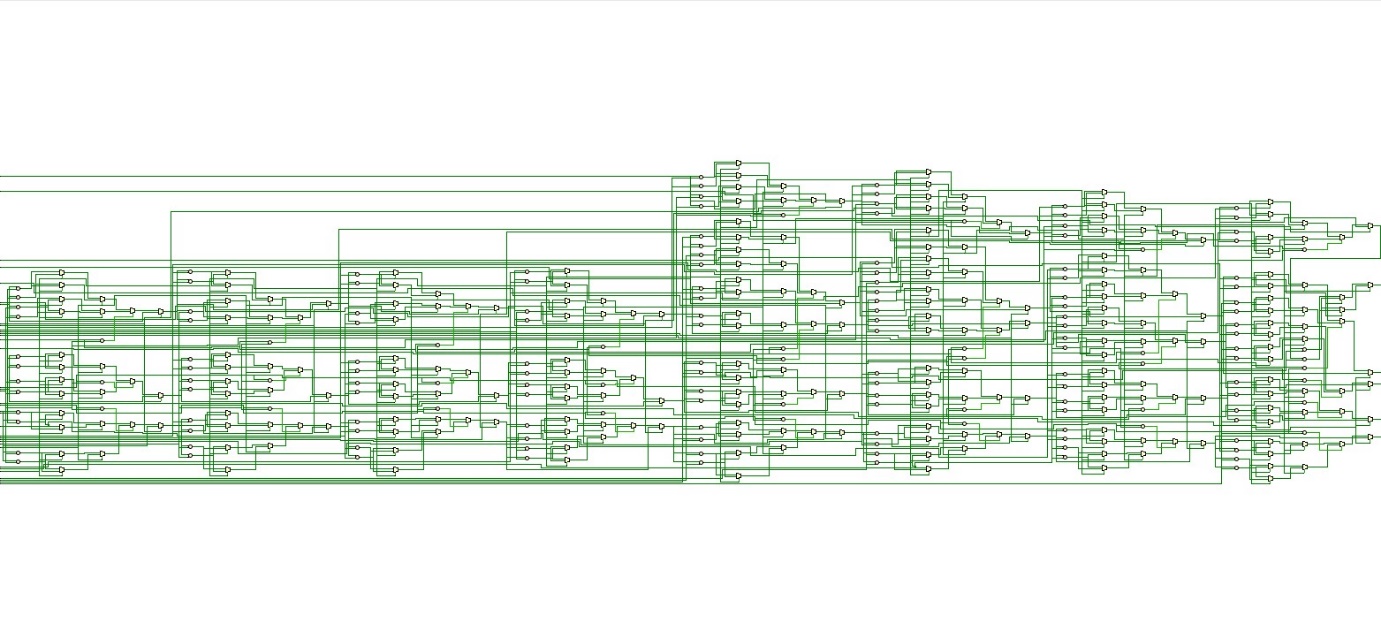
* A timing unit generates timing signals (**counter**) for controlling the operation of the module.
* The **counter** counts clock cycles and triggers state transitions based on predefined delays

(**t0** to **tf**).

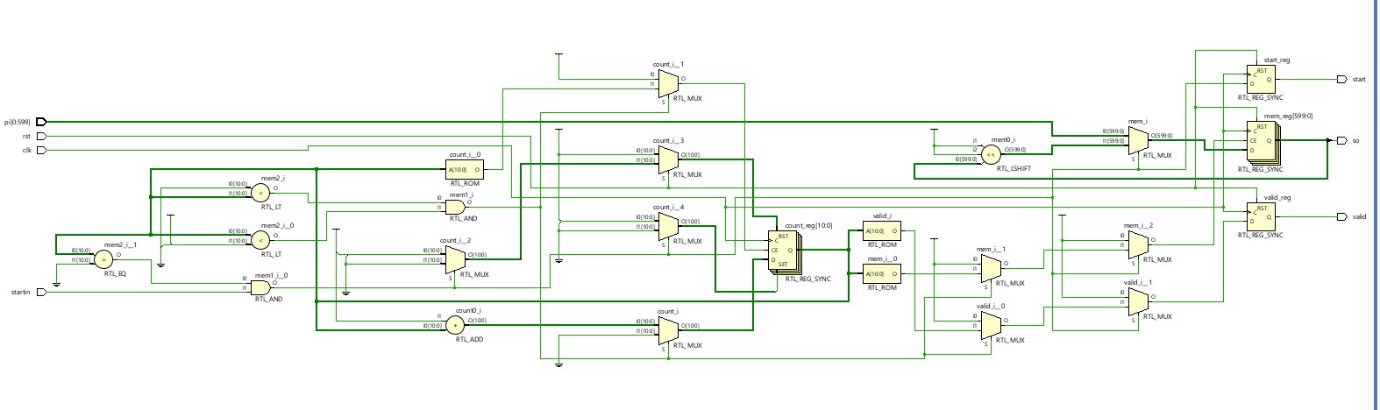
**Elaborated Design of LDPC Decoder**



*Elaborated design of De-serializer*



*Figure 23: Elaborated design of LDPC Decoder for Min-Sum Algorithm*



*Figure 24: Elaborated design of Serializer*

**Verification of LDPC Decoder and Result**

The provided testbench code is designed to simulate the functionality of an LDPC (LowDensity Parity-Check) decoder module. It begins by defining parameters such as the size of the LDPC decoder size, the number of check nodes **c**, the number of variable nodes **r**, and the number of iterations **N**. These parameters configure the LDPC decoder module for simulation.

Within the testbench, signals are declared to facilitate communication between the testbench and the LDPC decoder module. Notably, the **R** register array stores input data, while the **message** wire holds the output **message** produced by the LDPC decoder. Additionally, the **out\_start** wire signals when the LDPC decoder has generated an output.

Clock and reset signals **clk** and **rstb** are established to synchronize the simulation. The clock signal toggles at each time unit **#1**, providing a timing mechanism for the LDPC decoder's operation. The reset signal **rstb** initializes the LDPC decoder's state at the beginning of the simulation.

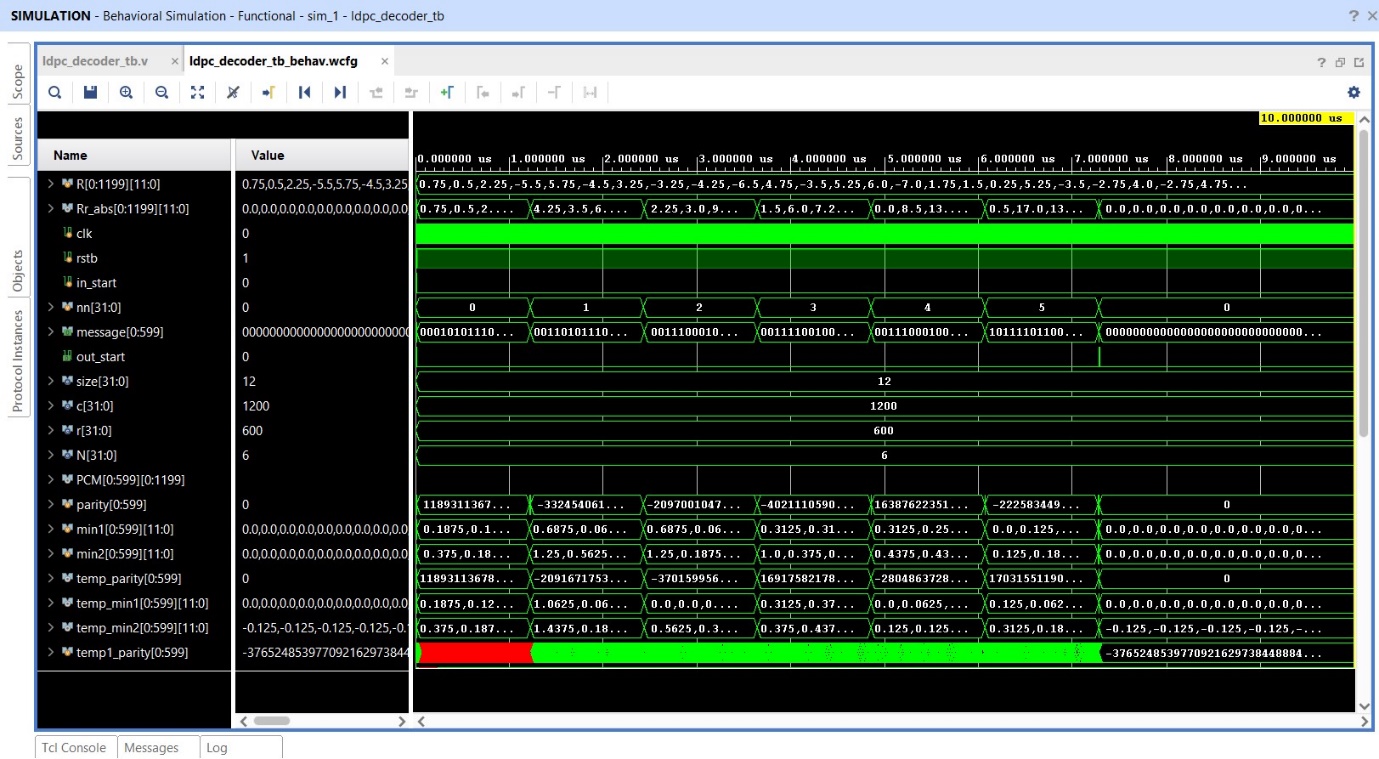
The LDPC decoder module itself is instantiated within the testbench, with connections made to the declared signals. This instantiation enables the LDPC decoder's functionality to be tested within the simulation environment.

To drive the clock signal and control the simulation flow, **always** blocks are used. One **always** block toggles the clock signal, ensuring that it transitions at each time unit. Another initial block initializes the simulation, opening a **file** to write the output message produced by the LDPC decoder.

Furthermore, the testbench initializes input data for the LDPC decoder module. This includes asserting and de-asserting the reset signal to ensure proper initialization, as well as populating the **R** register array with input data.

Overall, the testbench orchestrates the simulation of the LDPC decoder module, providing input data, controlling timing through clock signals, and capturing the output message for analysis.

**Behavioural Simulation of LDPC Decoder**



*Figure 25: Simulation of LDPC Decoder code up to 6 iterations*